Spécification et certification du logiciel

Interrogation écrite – avec son corrigé

5 novembre 2001

On utilisera et étendra la sorte `list` de spécification :

```
Sort: list
Uses: elt, bool
Symbols:
  nil : → list
  cons : elt, list → list
  mem : elt, list → bool
Axioms: ∀ xs:list; x, y:elt
  [A1] (mem x nil) = false
  [A2] (mem x (cons x xs)) = true
  [A3] (x≠y) ⇒ (mem y (cons x xs)) = (mem y xs)
```

Question 1

Spécifiez, en étendant la sorte `list`, un opérateur `rem` qui retire un élément d'une liste.

Corrigé

```
Extends: list
Symbols:
  rem : elt, list ->list
Axioms: ∀ x, y:elt; xs:list.
  [A4] (rem x nil) = nil
  [A5] (rem x (cons x xs)) = xs
  [A6] (x ≠ y) ⇒ (rem x (cons y xs)) = (cons y (rem x xs))
```

Question 2

On peut vérifier qu'une liste `xs` est une permutation d'une liste `ys` en vérifiant que si `xs` commence par l'élément `x` alors `x` apparaît dans `ys` et que `xs` privé de `x` (son premier élément) est une permutation de `ys` privée de `x`.

Donnez les axiomes de l'opérateur `is-perm : list, list → bool` tel que `(is-perm xs ys)` vaut `true` lorsque `xs` est une permutation de `ys`.

Corrigé

```
Extends: list
Symbols:
  is-perm : list, list ->bool
Axioms: ∀ x:elt; xs, ys:list.
  [A7] (is-perm nil nil) = true
  [A8] (is-perm (cons x xs) nil) = false
  [A9] (mem x ys), (is-perm xs (rem x ys)) ⇒ (is-perm (cons x xs) ys) = true
```

Remarques

1. les axiomes [$A8$] et [$A9$] ne sont pas contradictoires, car si *l'implication*

   `(mem x nil), (is-perm xs (rem x nil))` ⇒ `(is-perm (cons x xs) nil) = true`

   est vraie, elle ne permet pas de *conclure*

   `(is-perm (cons x xs) nil) = true`

   puisque l'on a jamais `(mem x nil) = true`.
2. l'axiome [$A9$] a souvent été remplacé par l'axiome plus contraignant

   `(is-perm (cons x xs) ys) = (and (mem x ys) (is-perm xs (rem x ys)))`

   Bien que je n'aime pas cette solution et que je lui préfère, la plus bavarde

   [$A9$] `(mem x ys), (is-perm xs (rem x ys))` ⇒ `(is-perm (cons x xs) ys) = true`
   [$A10$] `(is-perm (cons x xs) ys)` ⇒ `(mem x ys) = true`
   [$A11$] `(is-perm (cons x xs) ys)` ⇒ `(is-perm xs (rem x ys)) = true`

   qui ne déguise pas une conjonction (formule) sous une expression booléenne (terme); dans le mesure où j'ai moi-même utilisé des fonctions booléennes, que ceux et celles qui ont formulé un tel axiome se rajoutent un point !

Question 3

Prouvez, en utilisant vos axiomes et ceux donnés ci-dessus, que ∀`xs:list. (is-perm xs xs) = true`.

Corrigé

On démontre par induction sur `xs` que `(is-perm xs xs) = true`.

- si `xs = nil`, il faut montrer que `(is-perm nil nil) = true`. Ce que nous donne l'axiome [$A7$] .
- si `xs = (cons x xs')`, notre hypothèse de récurrence est `(is-perm xs' xs') = true` et il faut montrer que `(is-perm (cons x xs') (cons x xs')) = true`. Or, on a:
  1. `(mem x (cons x xs')) = true`, par [$A2$] ;
  2. `(rem x (cons x xs')) = xs'`, par [$A5$] . Ce qui nous donne, en remplaçant le second `xs'` de l'hypothèse de récurrence par `(rem x (cons x xs'))`, que `(is-perm xs' (rem x (cons x xs'))) = true`.

  Ces deux égalités, combinées à [$A9$] , donnent alors le résultat attendu.

Question 4

Soit

Extends: `list`
Symbols:
`do-perm : list` → `list`
Axioms: ∀ `x1, x2:elt; xs:list.`
[$A10$] `(do-perm nil) = nil`
[$A11$] `(do-perm (cons x1 nil)) = (cons x1 nil)`
[$A12$] `(do-perm (cons x1 (cons x2 xs))) = (cons x2 (cons x1 (do-perm xs)))`

Énoncez puis démontrez que l'opérateur `do-perm` calcule une permutation de son argument.

Corrigé

L'énoncé à démontrer est:

∀ `xs:list. (is-perm xs (do-perm xs)) = true`.

La démonstration de cet énoncé contient un *méchant* piège ; tellement que moi-même y suis tombé. En effet, de par la définition de `do-perm` dont l'appel récursif se fait sur la liste privée de ses deux premiers éléments, on ne peut utiliser simplement l'induction structurelle qui fait passer d'une liste `xs` à une liste augmentée d'un seul élément `(cons x xs)`.

On a le choix entre utiliser un principe d'induction général sur la longueur (*on suppose la propriété vraie pour toute liste de longueur inférieure strictement à n et on montre qu'elle reste vraie au rang $n + 1$*) ou utiliser un schéma de récurrence *ad hoc* calqué sur la défintion de `do-perm` (on montre la propriété pour `nil`, pour les listes à un élément – `(cons x nil)` –, puis en supposant qu'elle est vraie de `xs`, on montre qu'elle reste vraie de `(cons x1 (cons x2 xs))`. Ce dernier schéma a été subodoré par certains d'entre vous.

Induction générale sur la longueur des listes On a, sur les entiers le principe d'induction généralisé suivant : pour toute propriété `P` sur les entiers,

$$\forall n.[\ (\ \forall m.(m < n) \Rightarrow P(m)\ ) \Rightarrow P(n)\ ] \Rightarrow \forall n.P(n)$$

Concrètement, pour montrer qu'une propriété est vraie pour tous les entiers ($\forall n.P(n)$), ce principe s'utilise de la façon suivante : on montre que $P(n)$ pour un entier n quelconque sous l'hypothèse que $P(m)$ est vraie pour tout $m < n$.

On peut transposer ce principe sur les listes en raisonnant sur leur longueur (fonction `len`) :

$$\forall \texttt{xs:list}.[\ (\ \forall \texttt{ys:list}.(\texttt{(len ys)} < \texttt{(len xs)}) \Rightarrow P(\texttt{ys})\ ) \Rightarrow P(\texttt{xs})\ ] \Rightarrow \forall \texttt{xs:list}.P(\texttt{xs})$$

Appliquons ce principe à notre cas : soit `xs:list`, on suppose, et c'est notre hypothèse d'induction, que

∀ `ys:list. ((len ys) < (len xs))` ⇒ `(is-perm ys (do-perm ys)) = true`

On cherche à montrer `(is-perm xs (do-perm xs) = true` .

Pour pouvoir retrouver les cas de définition de `do-perm`, on raisonne à présent sur la longueur de `xs` :

1. si `(len xs) = 0` alors `xs = nil` et on a notre résultat par $[A10]$ et $[A7]$.
2. si `(len xs) = 1` alors il existe `x:elt` tel que `xs = (cons x nil)`. On obtient

   `(is-perm (cons x nil) (do-perm (cons x nil)) = true`

   par $[A11]$ et le résultat de la question 3.
3. si `(len xs) > 1` alors il existe `x1:elt`, `x2:elt` et `xs':list` tels que

   `xs = (cons x1 (cons x2 xs'))`

   Attention, la contrainte que l'on a sur `xs` vaut aussi pour l'hypothèse d'induction. C'est à dire que celle-ci est devenue

   ∀ `ys:list. ((len ys) < (len (cons x1 (cons x2 xs'))))`
   ⇒ `(is-perm ys (do-perm ys)) = true`

   Le résultat cherché et, dans ce cas

   `(is-perm (cons x1 (cons x2 xs')) (do-perm (cons x1 (cons x2 xs'))) = true`

   En utilisant $[A12]$, on se ramène à chercher à montrer que

   `(is-perm (cons x1 (cons x2 xs')) (cons x2 (cons x1 (do-perm xs')))) = true`

   On va terminer la preuve en utilisant $[A9]$.

   (1) Comme `(len xs') < (len (cons x1 (cons x2 xs')))`, notre hypothèse d'induction nous donne que

   `(is-perm xs' (do-perm xs')) = true`

   (2) Or `(do-perm xs') = (rem x2 (cons x2 (do-perm xs')))`. On a donc également que

   `(is-perm xs' (rem x2 (cons x2 (do-perm xs')))) = true`.

   (3) De plus, `(mem x2 (cons x2 (do-perm xs'))) = true`.

   (4) en appliquant (2) et (3) à $[A9]$, on obtient que

   `(is-perm (cons x2 xs) (cons x2 (do-perm xs'))) = true`

On poursuit selon le même principe :

(5) Or `(cons x2 (do-perm xs'))` = `(rem x1 (cons x2 (cons x1 (do-perm xs'))))`, donc

`(is-perm (cons x2 xs) (rem x1 (cons x2 (cons x1 (do-perm xs'))))) = true`.

(6) On a aussi

`(mem x1 (cons x2 (cons x1 (do-perm xs')))) = true`

(7) En appliquant (6) et (5) à $[A9]$, on obtient enfin que

`(is-perm (cons x1 (cons x2 xs')) (cons x2 (cons x1 (do-perm xs')))) = true`

Induction structurelle *ad hoc* On peut, sur les listes formuler le principe d'induction structurelle suivant : pour toute propriété P

$$\begin{array}{l} P(\texttt{nil}) \wedge \\ \forall \texttt{x:elt}.P(\texttt{cons x nil}) \wedge \\ \forall \texttt{x1,x2:elt}.\forall \texttt{xs':list}.[\ P(\texttt{xs'}) \Rightarrow P((\texttt{cons x1 (cons x2 xs')}))\ ] \\ \Rightarrow \forall \texttt{xs:list}.P(\texttt{xs}) \end{array}$$

La démonstration utilisant ce principe peut être conduite de façon similaire à celle utilisant le principe d'induction généralisée sur la longueur. Elle est même un peu plus simple car l'hypothèse de récurrence donne directement (1).

Cependant, ce principe d'induction structurelle *ad hoc* n'est pas donné directement. Il faut, en toute rigueure en établir la validité, soit en le déduisant du principe d'induction généralisé, soit en le déduisant du principe d'induction structurel de base

$$P(\texttt{nil}) \wedge \forall \texttt{x;elt}.\forall \texttt{xs':list}.[\ P(\texttt{xs'}) \Rightarrow P((\texttt{cons x xs'}))\ ] \Rightarrow \forall \texttt{xs:list}.P(\texttt{xs})$$

Le première façon est directe. Nous donnons la preuve de la seconde façon, car elle utilise un détour astucieux.

Supposons

$$\begin{array}{ll} H1: & P(\texttt{nil}) \\ H2: & \forall \texttt{x:elt}.P(\texttt{cons x nil}) \\ H3: & \forall \texttt{x1,x2:elt}.\forall \texttt{xs':list}.[\ P(\texttt{xs'}) \Rightarrow P((\texttt{cons x1 (cons x2 xs')}))\ ] \end{array}$$

Nous voulons montrer $P(\texttt{xs})$ pour `xs:list` quelconque.

On peut déduira ce résultat de la preuve par indcution struturelle simple sur `xs` de la conjonction

$$P(\texttt{xs}) \wedge \forall \texttt{x:elt}.P(\texttt{cons x xs})$$

1. si `xs = nil`, il faut montrer que

$$P(\texttt{nil}) \wedge \forall \texttt{x:elt}.P(\texttt{cons x nil})$$

   Ce que l'on obtient avec $H1$ et $H2$.

2. si `xs = (cons y ys)`, notre hypothèse de récurrence est

$$HR:\ P(\texttt{ys}) \wedge \forall \texttt{x:elt}.P(\texttt{cons x ys})$$

   et il faut montrer que

$$P((\texttt{cons y ys})) \wedge \forall \texttt{x:elt}.P((\texttt{cons x (cons y ys)})$$

   (a) on déduit $P((\texttt{cons y ys}))$ du second membre de HR.
   (b) on déduit $P((\texttt{cons x (cons y ys)}))$, avec `x` quelconque, en appliquant $H3$ au premier membre de HR.