Programmation en Coq

Sylvain Lippi
Polytech Nice-Sophia
Département informatique
Preuves de programmes

On commence par charger les paquets `List` et `Arith` avec la directive `Require Import`. Ils contiennent la définition des listes, des entiers, de leur opérations usuelles (addition, test d'égalité, test d'inégalité...) et de leurs propriétés élémentaires.

On en profite aussi pour définir les listes d'entiers naturels :

```
Definition listn := (list nat).
```

1 Preuves par inductions

1.1 Définitions des fonctions récursives

On définit la fonction `len` qui rend le nombre d'éléments d'une liste comme suit :

Fixpoint *len* (*l* : *listn*) : *nat* :=
match l with
| *nil* ⇒ *0*
| *u*::*l'* ⇒ 1 + (*len l'*)
end.

De même, on définit la fonction `mirror` qui inverse l'ordre des éléments d'une liste :

Fixpoint *mirror* (*l* : *listn*) (*temp* : *listn*) {*struct l*} : *listn* :=
match l with
| *nil* ⇒ *temp*
| *u*::*l'* ⇒ (*mirror l'* (*u*::*temp*))
end.

On peut tester ces fonctions avec *Eval compute in* (*mirror* (1::2::3::*nil*) *nil*).

Remarque pour les non logiciens. Remarquez que la fonction `fonc` appliquée à l'argument `arg` s'écrit `(fonc arg)` et non `fonc(arg)`.

1.2 Preuves par induction

Utiliser la tactique `induction` pour prouver les théorèmes suivants.

Theorem *len_append* : ∀ *l1 l2* : *listn*, *len* (*l1*++*l2*) = (*len l1*) + (*len l2*).

Theorem *len_mirror* : ∀ *l1 l2* :*listn*, *len* (*mirror l1 l2*) = (*len l1*) + (*len l2*).

Indication 1. Pour le théorème `len_mirror`, faire l'induction le plus tôt possible afin d'avoir l'hypothèse d'induction la plus forte possible.

Indication 2. On pourra utiliser le lemme `plus_n_Sm`. Faire `Check` pour obtenir son énoncé.

2 Application : le tri par insertion

2.1 Les listes triées

2.1.1 Définition

Une liste triée est...
– une liste vide
– ou une liste ayant un seul élément
– ou une liste $p :: q :: l$ où p et q sont deux entiers tels que $p \leq q$ et $q :: l$ est une liste triée.

On pourra définir la notion précédente par un prédicat `sorted: listn → Prop` en utilisant deux instructions `match` imbriquées.

2.1.2 Preuve

Theorem *sorted_inv* : $\forall$ *n* : *nat*, $\forall$ *l* : *listn*, *sorted* (*n*::*l*) $\rightarrow$ *sorted l*.
Indication. L'induction n'est pas nécessaire ici ; une preuve par cas avec `destruct` suffit.

2.2 Une relation d'équivalence entre les listes

2.2.1 Définitions

On dit que deux listes sont équivalentes lorsqu'elles ont les mêmes éléments, un même nombre de fois (mais pas forcément dans le même ordre). On souhaite définir le prédicat `equiv` qui exprime une telle propriété.

On commence par définir une fonction `number_of: nat → listn → nat`.
`(number_of n l)` donne le nombre d'occurrences de l'entier `n` dans la liste `l`.

Indication. Utiliser la fonction `eq_nat_dec` pour tester effectivement l'égalité entre deux entiers (et non simplement l'expression de façon abstraite avec =) et l'opérateur ternaire `if then else`.

On peut maintenant facilement définir une fonction `equiv : listn → listn → Prop` en utilisant la fonction `number_of`.

Remarque. La fonction `number_of` est bien pratique pour exprimer des propriétés sur les listes. En plus, du prédicat `equiv`, on pourrait aussi l'utiliser pour exprimer qu'un entier appartient ou non à une liste, ou que le premier élément d'une liste triée est son plus petit élément ou encore qu'une liste est équivalente à sa liste miroir. Evidemment, les deux dernières propriétés sont des théorèmes que l'on peut prouver.

2.2.2 Preuves

On utilise la commande `Lemma`, synonyme de `Theorem`, uniquement pour des raisons de lisibilité et indiquer qu'il s'agit de propriétés intermédiaires (comme des fonctions auxilaires) permettant de prouver d'autres propriétés plus compliquées.
Lemma *equiv_refl* : $\forall$ *l* : *listn*, *equiv l l*.

Lemma *equiv_sym* : $\forall$ *l l'* : *listn*, *equiv l l'* $\rightarrow$ *equiv l' l*.

Lemma *equiv_trans* : $\forall$ *l0 l1 l2* : *listn*, *equiv l0 l1* $\wedge$ *equiv l1 l2* $\rightarrow$ *equiv l0 l2*.

Lemma *equiv_cons* : $\forall$ *n* : *nat*, $\forall$ *l l'* : *listn*, *equiv l l'* $\rightarrow$ *equiv* (*n*::*l*) (*n*::*l'*).

Lemma *equiv_perm* : $\forall$ *a b* : *nat*, $\forall$ *l l'* : *listn*, *equiv l l'* $\rightarrow$ *equiv* (*a*::*b*::*l*) (*b*::*a*::*l'*).

2.3 La fonction d'insertion

2.3.1 Définition

Définir une fonction `insert: nat → listn → listn` qui insère (au bon endroit !) un élément dans une liste triée.

Remarque : on pourra utiliser la fonction `le_gt_dec` pour comparer deux entiers.

2.3.2 Preuves

Prouver par induction sur ℓ,

Lemma *insert_equiv* : $\forall$ *n* : *nat*, $\forall$ *l* : *listn*, *equiv* (*n*::*l*) (*insert n l*).
Indication. Utiliser les lemmes de la section 2.2.2.

Lemma *insert_sorted* : $\forall$ *n* : *nat*, $\forall$ *l* : *listn*, *sorted l* $\rightarrow$ *sorted* (*insert n l*).
Indication. Utiliser le lemme `lt_le_weak`. Utiliser `Check` pour obtenir son énoncé.

2.4 La fonction de tri

On peut maintenant prouver par induction sur ℓ le théorème suivant :

Theorem *isort* : $\forall$ *l* : *listn*, $\{$*l'* : *listn* | *sorted l'* $\wedge$ *equiv l l'*$\}$.

Et extraire *automatiquement* les fonctions `isort` et `insert` en utilisant les commandes :

Extraction isort.
Extraction insert.

Indication. On prouve le théorème `isort` de la même manière que le théorème suivant : Theorem *isort_bis* : $\forall$ *l* : *listn*, $\exists$ *l'* : *listn*, *sorted l'* $\wedge$ *equiv l l'*.