Rapport du projet de COQ 2016

Guillaume Jicquel, Sylvain Cassier

6 novembre 2016

Ceci est le rapport du projet `coq` 2016 de Guillaume Jicquel et Sylvain Cassier. Il contient l'ensemble des définitions et théorèmes ainsi que leurs explications. Les exercices choisis ont été le premier et le troisième.

1 Exercice 1

Dans cet exercice on travaille sur des listes, on commence donc par définir les listes :

```
Inductive list (A : Set) :
Set :=
| nil : list A
| cons : A → list A → list A.
```

1.1 Question 1

Dans cette question, on s'occupe de définir la fonction de concaténation de deux listes :

```
Fixpoint concat (A : Set) (l1 l2 :list A) :list A :=
match l1 with
|nil _ ⇒ l2
|cons x tl⇒ cons x (concat tl l2)
end.
```

1.2 Question 2

Ici on définit la fonction capable de retourner la longueur d'une liste :

```
Fixpoint length (A : Set) (l1 :list A) : nat :=
match l1 with
|nil _ ⇒ 0
|cons x t1 ⇒ 1 + (length t1)
```

```
end.
```

Le lemme suivant affirme que la longueur de la concaténation de deux listes est égale à la somme des longueurs des deux listes.
Sa preuve est complète.

```
Lemma tailleC (A : Set) :
  ∀ (l1 l2 : list A), length (concat l1 l2) = (length l1) + (length l2).
```

1.3 Question 3

On définit dans cette question le prédicat qui prend en entrée un type A, une variable x et une liste l et qui renvoie True si x appartient à l, False sinon.

```
Fixpoint appartient (A : Set) (x :A) (l : list A) : Prop :=
match l with
| nil _ ⇒ False
| cons x0 t1 ⇒ (x = x0) ∨ (appartient x t1)
end.
```

Ensuite on définit par induction l'appartenance à une liste comme l'appartenance à la tête de la liste, donc le premier élément de la liste ou bien l'appartenance àu reste de la liste.

```
Inductive appartienti (A : Set ) : A → list A → Prop :=
| appart_tete : ∀ (e : A), ∀(q : list A), appartienti e (cons e q)
| appart_rest : ∀ (e : A),∀(f :A), ∀(q : list A), appartienti e q → appartienti e (cons f q).
```

1.4 Question 4

Le but de cette question est de montrer que si un élément appartient à la concaténation de deux listes, alors il appartient à l'une des deux listes. La démonstration du lemme est complète.

```
Lemma appartC (A : Set) : ∀ (l1 l2 : list A) x,
(appartienti x (concat l1 l2)) → (appartienti x l1 ) ∨ (appartienti x l2).
```

2 Exercice 3

Cette exerice va concerner les permutations et les transpositions, on définit donc les types relatives à ces opérations.

```
Definition Tperm := list (nat× nat).
Definition eq_nat_decide : ∀ (A :Set) (x y : A), {x=y}+{~x=y}.
```

2.1 Question 1

Dans cetet question on définit la fonction `act` qui retourne l'entier correspondant à `i` dans la permutation `l`.

```
Fixpoint act (l : Tperm ) (i :nat) : nat :=
match l with
| nil ⇒ i
| cons (k,l) p ⇒
  match eq_nat_decide (act p i) k with
  | left _ ⇒ l
  | right _ ⇒
    match eq_nat_decide (act p i) l with
    |left _ ⇒ k
    |right _ ⇒ act p i
    end
  end
end.
```

2.2 Question 2

Ici, on veut définir la fonction qui retourne la liste d'entier transposés dans une permutation. Pour cela on a besoin de fonctions auxiliaires pour savoir si un entier appartient à une permutation. Ensuite on peut créer la fonction `atoms`.

```
Fixpoint appartient (a : nat) (l : list nat) : bool :=
match l with
| x : :b ⇒
  match eq_nat_decide a x with
  | left _ ⇒ true
  | right _ ⇒ appartient a b
  end
| _ ⇒ false
end.
Fixpoint atoms (l : Tperm) : list nat :=
match l with
| nil ⇒ nil
| cons (x, y) b ⇒
    if appartient x (atoms b) then atoms b else x : :(atoms b)
end.
Lemma atoms_simpl : ∀ l,
atoms l = match l with
| nil ⇒ nil
```

| *cons* $(x,\ y)$ $b \Rightarrow$
    `if` *appartient* x $(atoms\ b)$ `then` *atoms* b `else` $x :: (atoms\ b)$
`end`.

2.3 Question 3

Ici, on veut prouver que pour toute permutation `pi` et pour tout entier `a` , si l'application d'une permutation à cet entier est différente de `a`, alors `a` est dans les atomes de `pi`. Le lemme a été partiellement prouvé

`Lemma` *act_atoms* : $\forall$ *pi a*, $\neg act(pi)(a) = a \rightarrow In\ a\ (atoms(pi))$.

2.4 Question 4

Cette question concerne l'implémentation de la composition de deux permutations. On comme donc par la définir :

`Definition` *compose pi1 pi2* : *Tperm* := *pi1* ++ *pi2*.

Ensuite, on prouve le lemme suivant, assurant que la définition précédente implémente correctement la composition. La preuve du lemme est complète

`Lemma` *act_append* : $\forall$ *pi1 pi2 c*, *act* (*compose pi1 pi2*) *c* = *act pi1* (*act pi2 c*).

2.5 Question 5

Dans cette question, on veut prouver la propriété d'invariance des permutations, c'est à dire que pour deux entiers, si leut résultat par la même permutations sont égaux, alors les deux entiers sont égaux.

`Lemma` *act_invariance* : $\forall$ *pi a b*, *act pi a* = *act pi b* $\rightarrow$ *a* = *b*.

2.6 Question 6

Ici, on s'intéresse à l'inverse d'une permutation. On veut prouver que la composition de l'inverse d'une permutation avec elle-même est l'identité. Pour cela, on a besoin de lemmes intermédiaires :

— Tout d'abord, on montre plusieurs propriétés de l'égalité sur les entiers naturels.
  `Lemma` *rev_eq* : $\forall$ (A :`Set`) ($x\ y : A$), $x = y \leftrightarrow y = x$.
  `Lemma` *rf* : $\forall$ (A :`Set`) ($x\ y : A$), $(x \neq y \rightarrow x = y) \leftrightarrow True$.
  `Lemma` *ra* : $\forall$ (A :`Set`) ($x\ y\ z : A$),$(x{=}y \rightarrow x{\neq}\ z \rightarrow z{=}y) \leftrightarrow True$.

— Finalement, on peut prouver le lemme affirmant pour toute permutation `pi` et pour tout entier `c`, l'application à `c` de la composition de `pi` et l'inverse de `pi` donne c.

`Lemma` *act_reverse* : ∀ *pi c*, *act pi* (*act* (*rev pi*) *c*) = *c*.

La preuve du lemme `rev_eq` est complète.

2.7 Question 7

Dans cette dernière question, on montre que l'application à un entier `c` de la composition de la transposition `(a,b)` et de la permutation `pi` est l'application à `c` de la composition, de la transposition consituée de l'application à `a` de `pi` comme atome et de l'application à `b` de `pi` comme image, et de `pi`. Sa preuve est cependant incomplète.

`Lemma` *act_comm* : ∀ *pi a b c*, *act* (*pi* ++ (*cons* (*a*,*b*) *nil*)) *c* = *act* ((*cons* (*act pi a*, *act pi b*) *nil*) ++ *pi*) *c*.