Chapter 1

theme01fonctionsstudent

Dans ce premier thème, nous abordons la thématique de preuve de programmes par le biais de la programmation fonctionnelle.

En effet, le logiciel Coq intègre une sorte de sous-ensemble de Ocaml dans lequel on peut faire de la programmation fonctionnelle. Pour être précis, il s'agit de programmation fonctionnelle *pure* (sans effet de bord) et *totale*. La contrainte de totalité impose aux fonctions de terminer sur toutes leurs entrées.

1.1 Fonctions simples

On commence par la plus simple des fonction : l'identité

En Ocaml on écrirait par exemple :

`let` *id* (*e*:'*a*) : '*a* = *e*
val id : '*a* → '*a* = <`fun`>

Ce qui se traduit facilement en Coq :

`Definition` id {*A*:`Set`} (*a*:*A*) : *A* := *a*.

Une différence importante est que le type paramètre '*a* en Ocaml devient une variable "normale" *A* dont le type est `Set`. Les accolades indiquent à Coq que le paramètre *A* est implicite et qu'il peut être (en général) inféré depuis les autres paramètres, ici depuis la variable *a* de la fonction identité.

En Ocaml on peut écrire des expressions pour réaliser des calculs :

id 12 ;;
- : *int* = 12
id true ;;
- : *bool* = *true*

En Coq on écrira :

```
Eval compute in id 12.
```

```
= 12
     : nat
```

```
Eval compute in id true.
```

```
= true
     : bool
```

On peut aller un peu plus loin avec en montrant des propriétés sur la fonction *id*.
On peut reprendre un exemple ci-dessus, mais en mode "preuve" :

```
Example id_12: id 12 = 12.
Proof.
compute.
reflexivity.
Qed.
```

On peut démontrer une propriété plus générale (la seule que l'on peut considérer comme intéressante quoiqu'évidente sur l'identité) :

```
Lemma id_id:
  ∀ A : Set, ∀ a : A,
    id a = a.
Proof.
intros A a.
compute.
reflexivity.
Qed.
```

Un `Example` correspond à une sorte de test unitaire. Du point de vue de Coq il s'agit en fait d'un théorème qui doit être logiquement vrai. Pour cela, nous devons donner une démonstration de la validité de l'exemple. Cette preuve est introduite par la commande `Proof` qui place Coq en mode d'édition de preuve. On écrit ensuite un script de preuve rédigé dans le langage de tactiques très puissant et extensible de Coq. Les tactiques sont des outils d'aide à la démonstration, et une bonne partie de notre apprentissage passera par l'acquisition des tactiques de base de Coq (qui sont assez nombreuses).

La tactique `intros` permet d'introduire les variables A et l de notre lemme en tant qu'hypothèses du but courant. La tactique `compute` évalue le but courant sous sa forme normale. Cette tactique est utilisée pour évaluer des termes clos (sans variable libre).

La commande `Qed` permet de conclure la preuve lorsqu'il ne reste plus de *but* à prouver. On sort alors du mode d'édition de preuve.

1.1.1 Exercice : fonction constante

Définir en Ocaml puis traduire en Coq la fonction *constant* qui retourne toujours le premier de ses deux arguments.

Tester avec *constant* 12 *true*

Montrer que *constant* $(id\ a)\ b = a$.

Considérons maintenant une fonction un peu plus complexe :

```
Definition flip {A B C:Set} (f:A → B → C) : B → A → C :=
  fun (b:B) (a:A) ⇒ f a b.
```

En Ocaml on écrirait :

```
let flip (f:'a → 'b → 'c) : 'b → 'a → 'c =
    fun (b:B) (a:A) → f a b.
```

Et une propriété intéressante est la suivante :

```
Lemma flip_flip:
  ∀ A B C : Set, ∀ f : A → B → C,
    flip (flip f) = f.
Proof.
intros A B C f.
unfold flip.
reflexivity.
Qed.
```

Une fois qu'une propriété est démontrée, on peut bien sûr l'exploiter.

```
Lemma flip_flip_flip_flip:
  ∀ A B C : Set, ∀ f : A → B → C,
    f = flip (flip (flip (flip f))).
Proof.
intros A B C f.
rewrite flip_flip.
rewrite flip_flip.
reflexivity.
Qed.
```

On sait qu'il existe souvent plusieurs façons différentes de résoudre un même problème. Certaines façons sont plus simples, d'autres plus complexes. Il se passe un peu la même chose dans le monde des preuves.

```
Lemma flip_flip_flip_flip':
  ∀ A B C : Set, ∀ f : A → B → C,
    f = flip (flip (flip (flip f))).
```

```
Proof.
repeat rewrite flip_flip ; reflexivity.
Qed.
```

1.1.2 Fonctions totales

Considérons maintenant un exemple nous permettant d'illustrer la notion de totalité.

Dans la bibliothèque *Prelude* de Haskell on trouve la fonction suivante (réécrite pour l'occasion en Ocaml).

```
# let rec until (pred: 'a → bool) (f:'a → 'a) (a:'a) : 'a =
    if pred a then until pred f (f a)
    else a ;;
```

Si on traduit directement cette fonction en Coq on tombe sur un os.

```
Fixpoint until {A:Set} (pred: A → bool) (f:A → A) (a:A) : A :=
  if pred a then until pred f (f a)
  else a.
```

Error: *Cannot guess decreasing argument of* `fix`.

Coq ne peut prouver automatiquement que la fonction `until` est totale, donc termine sur toute ses entrées. Et pour cause, avec le prédicat par exemple `fun _ →` *true* la fonction diverge !

D'un point de vue théorique, la fonction `until` ne possède pas de plus petit point fixe unique et elle n'est donc pas récursive. En fait, on peut montrer qu'elle est en fait co-récursive (elle possède un plus grand point fixe unique). Coq permet de raisonner sur les fonctions co-récursive mais cela dépasse le cadre de notre introduction.

En restant dans le monde récursif, on peut fournir une approximation de `until`.

```
Fixpoint untiln {A:Set} (n:nat) (pred: A → bool) (f:A → A) (a:A) : A :=
  if pred a then match n with
                   | 0 ⇒ a
                   | S n' ⇒ untiln n' pred f (f a)
                 end
  else a.
```

Ici, on autorise la fonction à diverger, mais dans les limites fixées par le paramètre n (entier naturel).

Et on peut dès lors démontrer des propriétés approximées de `until`.

```
Lemma untiln_id:
  ∀ A : Set, ∀ n:nat,
  ∀ pred:A → bool, ∀ a : A,
    untiln n pred id a = a.
```

```
Proof.
intros A n pred a.
induction n as [|n'].
  - (*  cas n=0 *)
    simpl.
    case (pred a) ; reflexivity.
  - (*  cas n=S n' *)
    simpl.
    destruct (pred a).
    + (*  cas (pred a) = True *)
      rewrite id_id.
      rewrite IHn'.
      reflexivity.
    + (*  cas (pred a) = False *)
      reflexivity.
Qed.
```

Remarque : la tactique `simpl` permet de réduire un terme avec variables libres. Elle fait partie des plus utiles et des plus fréquemment utilisées car elle permet la preuve par calcul.

Nous reviendrons à maintes reprises sur ces problèmes de totalité, mais il faut bien retenir que Coq - par défaut - n'autorise que des fonctions dont la terminaison peut-être démontrée de façon automatique, selon des critères syntaxiques précis.

1.2 Fonctions sur les listes

Pour commencer à certifier des programmes un peu plus itéressants, nous allons maintenant travailler sur le type *list*. Commençons par charger la bibliothèque correspondante avec la commande suivante :

```
Require Import List.
```

La définition Coq du type *list* est la suivante :

```
Print list.
```

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

La liste vide s'écrit *nil*. Le constructeur *cons* de liste prend en argument un élement de type générique A, une liste de A et retourne une liste de A.

La liste 1, 2, 3 s'écrit de la façon suivante :

```
Check cons 1 (cons 2 (cons 3 nil)).
```

```
1 :: 2 :: 3 :: nil : list nat
```

Coq nous indique ici qu'il s'agit d'une liste d'entiers naturels ou *list nat*. Comme suggéré par l'affichage, on peut utiliser l'opérateur infixe :: pour simplifier les écritures.

```
Example cons_infixe:
 1::2::3::nil = cons 1 (cons 2 (cons 3 nil)).
Proof.
reflexivity.
Qed.
```

1.2.1 Une première fonction

Considérons une fonction *concat* permettant de concaténer deux listes. Cette fonction existe dans la bibliothèque standard et s'appelle *append* mais il est intéressant de la redéfinir.

Commençons par une définition en Ocaml.

```
let rec concat (l1 : 'a list) (l2 : 'a list) : 'a list =
  match l1 with
  | [] -> l2
  | e::l1' -> e::(concat l1' l2)
```

Cette fonction peut être quasiment directement traduite dans le langage fonctionnel de Coq de la façon suivante :

```
Fixpoint concat {A : Set} (l1 l2 : list A) : list A :=
  match l1 with
    | nil ⇒ l2
    | e::l1' ⇒ e::(concat l1' l2)
  end.
```

La transposition est relativement directe. En remplacement de `let` *rec* on utilise la commande `Fixpoint` de Coq. Le paramètre de type α ('*a* en Ocaml) doit être introduite explicitement. On utilise la notation {*A* : `Set`} pour introduire le paramètre de type *A* pouvant être un ensemble quelconque (donc du type des ensembles `Set`). Les accolades indiquent qu'il s'agit d'un paramètre implicite alors que les parenthèses introduisent des paramètres explicites. On retiendra pour l'instant que les paramètres de type (au sens de la programmation en OCaml) sont introduits de façon implicite. Les listes *l1* et *l2* sont introduites ensuite, en exploitant un raccourci d'écriture bien pratique de Coq. Elles sont toutes les deux de type *list A* ("liste de *A*"). Finalement on sait que *concat* retourne également une liste de *A*. Le reste de la définition utilie la construction `match with` ... de Coq qui est proche de celle de Ocaml (mais en fait beaucoup plus puissante).

Tentons maintenant une évaluation par la commande `Eval compute in`.

```
Eval compute in concat (1::2::nil) (3::4::5::nil).
```

$= 1 :: 2 :: 3 :: 4 :: 5 :: nil : list\ nat$

Ecrivons le test correspondant avec `Example`.

```
Example concat_ex1:
  concat (1::2::3::nil) (4::5::nil) = 1::2::3::4::5::nil.
Proof.
compute.
reflexivity.
Qed.
```

Au delà des exemples, l'intérêt de Coq est de permettre la démonstration assistée de propriétés sur les fonctions au travers de lemmes et théorèmes.

Prouvons un lemme simple sur notre fonction *concat*.

```
Lemma nil_concat:
  ∀ A : Set, ∀ l : list A,
    concat nil l = l.
Proof.
intros A l.
simpl.
reflexivity. (*  ou trivial *)
Qed.
```

Le lemme symétrique ne peut être prouvé de façon aussi simple.

```
Lemma concat_nil_fails:
  ∀ A : Set, ∀ l : list A,
  concat l nil = l.
Proof.
intros A l.
simpl.
Abort.
```

Ici, `simpl` ne simplifie rien tout simplement parce que dans la définition de *concat* c'est le premier élément qui est analysé. Or ici on ne connait pas la nature de l.

On peut essayer un raisonnement par cas sur l.

```
Lemma concat_nil_destruct:
  ∀ A : Set, ∀ l : list A,
  concat l nil = l.
Proof.
intros A l.
destruct l as [| e l'].
  - (*  cas l=nil *)
    simpl.
    reflexivity.
  - (*  cas l=e::l' *)
    simpl.
Abort.
```

La tactique `destruct` l `as` [| e l'] décompose la liste l en sous-cas : un pour *nil* et l'autre pour *cons*. Dans [| e l'] on donne explicitement les noms des variables permettant de décomposer l. Pour le cas *nil* aucune variable n'est nécessaire donc on ne spécifie rien et on sépare avec le second cas en utilisant la barre verticale |. Dans le second cas *cons* on indique e l' pour respectivement le nom du premier élément et le nom du reste de la liste. Si on ne mentionne pas de nom explicite, Coq va en choisir selon un algorithme qui peut varier d'une version à l'autre. On préfèrera expliciter les noms pour rendre nos preuves plus robustes aux changements de version.

Dans ce raisonnement par cas, tout se passe bien pour le cas l=*nil* mais pour l=e::l' on se retrouve bloqué dans un état proche de celui de départ, cette fois-ci sur le reste l'. Tout indique que notre raisonnement doit être inductif.

A partir de la définition du type inductif *list*, Coq génère automatiquement des principes de raisonnements inductifs. Celui qui nous intéresse s'appelle *list_ind* et regardons son type.

```
Check list_ind.
```

```
list_ind
 : ∀ (A : Type) (P : list A → Prop),
       P nil →
       (∀ (a : A) (l : list A), P l → P (a :: l)) →
       ∀ l : list A, P l
```

C'est un peu difficile à lire mais essayons de le décrypter. Pour tout type A (de type `Type`, sur-ensemble de `Set`) et toute propriété P sur les listes (en fait une fonction de *list* A vers `Prop` le type des propositions), si on montre :

- que la propriété est vraie sur *nil*, ce qui s'écrit P *nil*,
- que si on suppose pour tout élément a de type A et toute liste l de type *list* A, la propriété est vraie sur l (donc P l), alors la propriété est vraie sur la construction a::l (donc P a::l)

Alors on peut conclure que pour toute liste l de type *list* A, la propriété P est vraie (donc P l).

C'est bien sûr le raisonnement inductif classique sur les listes, que Coq nos présente ici sous la forme d'un type, celui de *list_ind* qui est en fait une fonction récursive ! Mais nous verrons ça un peu plus tard.

La tactique `induction` permet d'appliquer ce principe d'induction dans une preuve. Mettons la tout de suite en pratique.

```
Lemma concat_nil:
  ∀ A : Set, ∀ l : list A,
  concat l nil = l.
Proof.
induction l as [|e l'].
```

```
  - (*  cas de base : l=nil *)
    simpl.
    reflexivity.
  - (*  cas inductif: l=e::l' *)
    simpl.
    rewrite IHl'.
    reflexivity.
Qed.
```

Reprenons le fil de cette preuve pour retrouver un schéma classique de preuve informelle (papier et crayon).

Nous effectuons l'induction sur l.

Dans le cas de base on a l=*nil* donc le but devient

concat nil nil = *nil*

Par simplification on obtient *nil* = *nil* que l'on peut terminer par `reflexivity`.

Dans le cas récursif on a l=e::l' et la tactique `induction` a introduit une hypothèse d'induction. Celle-ci est simplement :

IHl': *concat l' nil* = *nil*

Donc par hypothèse la concaténation du reste de l avec *nil* donne bien *nil*. Notre but est le suivant :

concat (*e* :: *l'*) *nil* = *e* :: *l'*

Après simplification de ce but, on obtient :

e :: *concat l' nil* = *e* :: *l'*

On sait par hypothèse d'induction que *concat l' nil* est égal à *l'*. Pour exploiter cela, on utilise la tactique `rewrite` qui permet de réécrire de gauche à droite le but courant à partir d'un hypothèse de la forme *gauche* = *droite*.

On obtient alors:

e :: *l'* = *e* :: *l'*

qui se finalse avec `reflexivity`. Et voilà nous avons terminé notre première preuve inductive, qui n'est franchement pas plus difficile à conduire que sur papier.

Remarque : on peut réécrire de droite à gauche en utilisant la tactique `rewrite` ←. Nous aurons l'occasion de l'utiliser.

1.2.2 Fonctions partielles

Les fonctions en Coq, on l'a déjà dit, doivent être totales. En plus de terminer sur toutes leurs entrées, il faut aussi accepter toutes les valeurs du type précisé pour les arguments. Pourtant, en pratique, on a souvent besoin de fonctions partielles.

C'est le cas par exemple pour la fonction *head* qui n'est pas définie pour la liste vide. Si en Ocaml on peut utiliser les exceptions, ce n'est pas un principe compatible avec la notion de fonction calculable.

```
# exception Boum ;;
exception Boum
# let boum (e:'a) = raise Boum ;;
val boum : 'a → 'b = <fun>
```

Le type de *boum*, si Coq l'acceptait, correspondrait à en gros : $\forall\ A\ B$: Set, $A \to B$ ce qui serait assez chouette d'un point de vue pratique, mais malheureusement tout à fait illogique !

Une autre option, disponible également en Ocaml (et favorisée en Haskell), est d'exploiter le type *option* (ou *Maybe* en Haskell).

Ce type est disponible en Coq :

```
Print option.
```

```
Inductive option (A : Type) : Type :=
  Some : A → option A
| None : option A
```

On définit alors facilement notre fonction *head*.

```
Definition head {A:Set} (l:list A) : option A :=
  match l with
      | nil ⇒ None
      | e::_ ⇒ Some e
  end.

Example head_ex1:
  head (1::2::3::4::nil) = Some 1.
Proof.
compute.
reflexivity.
Qed.
```

Démontrons un lemme reliant *head* à *concat*.

```
Lemma head_concat:
  ∀ A : Set, ∀ l1 l2 : list A,
    head (concat l1 l2) =
    match l1 with
      | nil ⇒ head l2
      | e1::l1' ⇒ Some e1
    end.
Proof.
```

```
intros A l1 l2.
destruct l1 as [| e1'' l1''].
  - (*  cas l1 = nil *)
    simpl.
    reflexivity.
  - (*  cas l1 = e1'':: l1'' *)
    simpl.
    reflexivity.
Qed.
```

Un petit *one-liner* se profile...

```
Lemma head_concat':
  ∀ A : Set, ∀ l1 l2 : list A,
    head (concat l1 l2) =
    match l1 with
      | nil ⇒ head l2
      | e1 :: l1' ⇒ Some e1
    end.
Proof.
  destruct l1 ; trivial.
Qed.
```

Remarque : pour les preuves simples, de tels *one-liner* sont intéressants mais il ne faut pas en abuser car on perd l'essentiel de la structure de la preuve.

Pour le complément *tail* les versions totales décident généralement de retourner *nil* pour la liste vide, plutôt que de faire une fonction partielle à options.

```
Definition tail {A:Set} (l:list A) : list A :=
  match l with
      | nil ⇒ nil
      | _ :: l' ⇒ l'
  end.
```

Il faut tout de même être conscient d'une petite ambigûité qui ne pose pas de gros problème en pratique dans ce cas précis (mais on peut dire qu'introduire des ambigüité est rarement une bonne idée).

```
Lemma tail_amb:
  ∀ A : Set, ∀ e : A,
    tail nil = tail (e :: nil).
Proof.
intros A e.
simpl.
reflexivity.
Qed.
```

On aimerait énoncer que pour une liste l, $head\ (tail\ l) = l$. Bien sûr, ce n'est pas tout à

fait exacte puisque cela dépend de l. La propriété doit faire l'hypothèse que la liste possède au moins un élément. On pourrait le faire avec un `match` mais on peut ajouter une hypothèse: que la tête existe.

```
Lemma head_tail:
  ∀ A : Set, ∀ e : A, ∀ l : list A,
    head l = Some e → e :: tail l = l.
Proof.
intros A e l Hhead.
destruct l as [| e' l'].
  - (*  cas l = nil *)
    inversion Hhead. (*  contradiction *)
  - (*  cas l = e'::l' *)
    simpl.
    inversion Hhead as [ He ].
    reflexivity.
Qed.
```

1.2.3 Exercice : *untiln* partielle

La fonction *untiln* vue précédemment devrait être réécrite pour être une fonction partielle. En particulier, si la borne d'approximation est atteinte, le résultat devrait être *None*.

Réécrire la fonction *untiln* en conséquence. Que faire du lemme démontré sur cette fonction ?

1.3 Exercices

1.3.1 Exercice : Renversement de liste

Question 1

Définir la fonction *rev* qui renverse les élément d'une liste.

Question 2

Compléter l'exemple suivant :

```
Example rev_ex1 :
  rev (1::2::3::4::5::nil) = 5::4::3::2::1::nil.
Proof.
```

Question 3

On souhaite montrer que *rev* est idempotent, mais la preuve suivante bloque.

```
Theorem rev_rev_fails:
  ∀ A : Set, ∀ l : list A,
    rev (rev l) = l.
Proof.
intros A l.
induction l as [| e l'].
  - (*  cas l=nil *)
    simpl.
    reflexivity.
  - (*  cas l=e::l' *)
    simpl.

>  A : Set
>  e : A
>  l' : list A
>  IHl' : rev (rev l') = l'
>  ============================
>   rev (concat (rev l') (e :: nil)) = e :: l'

Abort.
```

Il nous manque une propriété importante reliant *rev*, *concat* et *nil*.

Démonter le lemme *rev_concat_nil* correspondant. En déduire une preuve complète pour le théorème *rev_rev*.

1.3.2 Exercice : dernier élément

Question 1

Définir la fonction *last* qui retourne le dernier élément d'une liste non-vide. Cette fonction prendra un argument d (pour défaut) que l'on retournera si la liste est vide.

Question 2

Montrer le lemme suivant.

```
Lemma last_single:
  ∀ A : Set, ∀ d e : A,
    last (e::nil) d = e.
```

Question 3

Montrer par induction le lemme suivant.

```
Lemma last_concat:
```

```
  ∀ A : Set, ∀ l1 l2 : list A, ∀ d e : A,
    last (concat l1 (e::l2)) d = last (e::l2) d.
```

1.4 Fonctions d'ordre supérieur

Nous abordons maintenant le raisonnement sur les fonctions d'ordre supérieur, principalement sous la forme d'exercices.

1.4.1 Exercice : la fonction map

La fonction *map* se définit très naturellement en Coq :

```
Fixpoint map {A B :Set} (f:A → B) (l:list A) : list B :=
  match l with
    | nil ⇒ nil
    | e::l' ⇒ (f e) :: map f l'
  end.
```

Question

Démontrer les lemmes suivants :

```
Lemma head_map:
  ∀ A B : Set, ∀ e : A, ∀ l : list A, ∀ f : A → B,
    head (map f (e::l)) = Some (f e).


Lemma map_id:
  ∀ A : Set, ∀ l : list A,
    map id l = l.


Lemma map_concat:
  ∀ A B : Set, ∀ l1 l2 : list A, ∀ f : A → B,
    map f (concat l1 l2) = concat (map f l1) (map f l2).
```

1.4.2 Exercice : la fonction foldr

On considère la fonction suivante de pliage à droite :

```
Fixpoint foldr {A B :Set} (f:A → B → B) (u:B) (l:list A) : B :=
  match l with
    | nil ⇒ u
    | e::l' ⇒ f e (foldr f u l')
  end.
```

Question 1

Définir une fonction *folder_id* telle que l'exemple suivant est prouvable :

```
Example foldr_id_ex1 :
  foldr folder_id nil (1::2::3::nil) = (1::2::3::nil).
Proof.
```

En déduire le théorème général suivant :

```
Theorem foldr_id:
  ∀ A : Set, ∀ l : list A,
    foldr folder_id nil l = l.
Proof.
```

Question 2

En vous inspirant de l'exemple suivant (à compléter) :

```
Example folder_id_ex2:
 foldr folder_id (4::5::6::nil) (1::2::3::nil) = 1::2::3::4::5::6::nil.
Proof.
```

Définir une fonction *fold_concat* telle que l'on peut démontrer :

```
Theorem fold_concat:
  ∀ A : Set, ∀ l1 l2 : list A,
    foldr folder_id l2 l1 = concat l1 l2.
Proof.
```

Question 3

Selon les mêmes principes, redéfinir la fonction *map* en utilisant *foldr*.
Tester avec l'exemple correspondant à *map* (fun n : $nat \Rightarrow S\ n$) (1::2::3::4::5::*nil*).
Démontrer le théorème général correspondant.

Question 4

Mêmes questions pour la fonction *rev*

1.4.3 Exercice : pliage à gauche (difficile)

La fonction de pliage à gauche est la suivante :

```
Fixpoint foldl {A B :Set} (f:A → B → A) (u:A) (l:list B) : A :=
```

```
match l with
  | nil ⇒ u
  | e::l' ⇒ foldl f (f u e) l'
end.
```

Question 1

Définir une fonction *non-récursive foldl'* de pliage à gauche à partir de *foldr*.

Démontrer le théorème suivant :

```
Theorem foldl'_foldl:
  ∀ A B : Set, ∀ f : A → B → A,
  ∀ l : list B, ∀ u : A,
    foldl f u l = foldl' f u l.
Proof.
```

Question 2

Reconstruire les fonctions *id*, *concat*, *rev* et *map* avec *foldl* et *foldl'*.

Démontrer les lemmes et théorèmes qui semblent les plus pertinents. Alterner des preuves pour *foldl* et des preuves pour *foldl'* ainsi que des preuves exploitant le théorème *foldl'_foldl* ci-dessus.