■ Chapitre 3 ■

L'assistant à la preuve Coq

I - Installer et se documenter

Nous utiliserons l'assistant de preuve Coq. Dans les systèmes Debian et Ubuntu, ce logiciel est distribué sous forme de paquet. Pour les autres systèmes il suffit de télécharger la dernière version disponible sur la page : `http://coq.inria.fr/download`. Nous recommandons d'utiliser `Emacs` avec le module `Proof General` qui facilite l'édition et l'exécution du code Coq, il est disponible à l'adresse :
`http://proofgeneral.inf.ed.ac.uk/`,
ou d'utiliser `coqide` qui est disponible sous forme de paquet debian.

Quelques éléments bibliographiques sur lesquels reposent ce cours :

1. l'ouvrage *The Coq Art* par Yves Bertot et Pierre Castéran, disponible à la page : `http://www.labri.fr/perso/casteran/CoqArt/index.html`.
2. le petit guide de survie en Coq par Alexandre Miquel : `http://www.pps.jussieu.fr/~miquel/ens-0607/lc/guide.html`.

II - Correspondance de Curry-Howard

Introduite dans les années 1960, la correspondance de Curry-Howard permet de faire le lien entre la logique et l'informatique en remarquant que les preuves des théorèmes sont des programmes. Plus précisément, on a la correspondance :

Logique	Informatique
Formule	Type
Preuve	Programme
Élimination des coupures	Calcul

Nous verrons par la suite que le logiciel Coq repose sur cette correspondance.

II.1 - Lambda-calcul simplement typé

On se donne un type de base c et on construit les types par la grammaire :

$$F \quad := \quad c \quad | \quad F \to F$$

où c est un (ou plusieurs) type(s) de base comme par exemple les types `bool` ou `nat`.

La flèche $F_1 \to F_2$ représente l'ensemble des fonctions prenant un argument de type F_1 et renvoyant un résultat de type F_2.

Le lambda-calcul simplement typé est construit à partir des termes donnés par la grammaire suivante :

$$s, t \quad := \quad x \quad | \quad \lambda x.s \quad | \quad (s)\ t$$

où x appartient à un ensemble de variables donnés.

Dans le lambda-calcul simplement typé, on ne considère que les termes *typables*. On dit qu'un terme s est de type F s'il existe un *jugement de typage* $\Gamma \vdash u : F$ dérivable en appliquant les règles de typage suivantes :

$$\frac{}{\Gamma,\ x : F \vdash x : F}\ \text{Var} \qquad \frac{\Gamma, x : F_1 \vdash s : F_2 \quad \text{où } x \notin \Gamma}{\Gamma \vdash \lambda x.s : F_1 \to F_2}\ \text{Abs} \qquad \frac{\Gamma \vdash s : F_1 \to F_2 \quad \Gamma \vdash t : F_1}{\Gamma \vdash (s)t : F_2}\ \text{App}$$

où Γ est un environnement de la forme $x : F_1, \ldots, x_n : F_n$ avec $x_i \neq x_j$ si $i \neq j$.

Pour calculer le résultat d'un terme, on utilise la règle de calcul appelée β-réduction :

$$(\lambda x.s)t \to s[x := t]$$

où $s[x := t]$ est la substitution dans s de toutes les occurrences de la variable x par le terme t.

Exercice 1. Pour chacun des termes suivants, donner dans le langage de programmation Caml un programme et un type correspondants, puis dériver une preuve de typage.

- L'identité : $\lambda x.x$
- L'évaluation : $\lambda af.(f)a$
- La projection : $\lambda ab.a$

II.2 - Logique minimale intuitionniste

La logique minimale est construite à partir de formules définies grâce à la grammaire suivante :

$$F \quad := \quad c \quad | \quad F \Rightarrow F$$

où c est une variable propositionnelle et $\Rightarrow$ est l'implication.

Les formules prouvables de la logique intuitionniste sont des jugements de la forme $\Gamma \vdash F$ (où Γ est une liste éventuellement vide de formule et F est une formule) qui sont obtenus comme conclusion de règles de déduction :

$$\frac{}{\Gamma,\ F \vdash F}\ \text{Ax} \qquad \frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \Rightarrow F_2}\ \Rightarrow\text{-intro} \qquad \frac{\Gamma \vdash F_1 \Rightarrow F_2 \quad \Gamma \vdash F_1}{\Gamma \vdash F_2}\ \Rightarrow\text{-elim}$$

II.3 - Correspondance de Curry-Howard

En effaçant les termes des dérivations de typages, on obtient des preuves de la logique minimale intuitionniste. On a donc une correspondance exacte entre les λ-termes et les preuves.

$$\frac{}{\Gamma,\ \mathtt{x} : F \vdash \mathtt{x} : F}\ \text{Var} \qquad \frac{\Gamma, \mathtt{x} : F_1 \vdash \mathtt{s} : F_2 \quad \text{où } x \notin \Gamma}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{s} : F_1 \Rightarrow F_2}\ \text{Abs} \qquad \frac{\Gamma \vdash \mathtt{s} : F_1 \Rightarrow F_2 \quad \Gamma \vdash \mathtt{t} : F_1}{\Gamma \vdash \mathtt{(s)t} : F_2}\ \text{App}$$

Un terme peut-être vu comme une preuve d'une formule (son type) et un théorème (une formule de la logique) est prouvable s'il existe un arbre de preuve et donc un terme typé par cette formule, « un habitant » de ce type.

Exercice 2. Reprendre les termes que vous avez typés à l'exercice précédent et les traduire dans le langage logique.

La correspondance de Curry-Howard s'étend à de nombreux systèmes logiques. Nous nous intéressons à la logique propositionnelle du premier ordre et à l'extension adéquate du λ-calcul.

Extension au type produit : La conjonction $A \wedge B$ est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim}$$

elle correspond au type produit :

$$\frac{\Gamma \vdash \mathtt{t} : A \quad \Gamma \vdash \mathtt{u} : B}{\Gamma \vdash \langle \mathtt{t},\mathtt{u} \rangle : A \times B}\ \text{Pair} \qquad \frac{\Gamma \vdash \mathtt{t} : A \times B}{\Gamma \vdash \pi_1 \mathtt{t} : A}\ \text{Proj-l} \qquad \frac{\Gamma \vdash \mathtt{t} : A \times B}{\Gamma \vdash \pi_2 \mathtt{t} : B}\ \text{Proj-r}$$

Extension au type somme : La disjonction $A \vee B$ est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-elim}$$

elle correspond au type somme :

$$\frac{\Gamma \vdash \mathtt{t} : A}{\Gamma \vdash \mathtt{inl\ t} : A \vee B}\ \text{Emb-l} \qquad \frac{\Gamma \vdash \mathtt{t} : B}{\Gamma \vdash \mathtt{inr\ t} : A \vee B}\ \text{Emb-r} \qquad \frac{\Gamma \vdash \mathtt{t} : A \vee B \quad \Gamma, \mathtt{x} : A \vdash \mathtt{v} : C \quad \Gamma, \mathtt{y} : B \vdash \mathtt{w} : C}{\Gamma \vdash \mathtt{match\ t\ with\ inl\ x\ ->\ v\ |\ inr\ y\ ->\ w} : C}\ \text{Case}$$

Exercice 3. Pour chacune des formules suivantes, donner un arbre de preuve et le terme correspondant :

$$(P \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow (Q \Rightarrow R))$$
$$P \wedge Q \Rightarrow P$$
$$P \wedge Q \Rightarrow Q \wedge P$$
$$P \vee Q \Rightarrow Q \vee P$$

Extension à la quantification existentielle : La quantification existentielle est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A[x \leftarrow a]}{\Gamma \vdash \exists x.A}\ \exists\text{-intro} \qquad \frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B \quad \text{où } x \notin FV(\Gamma) \text{ et } x \notin FV(B)}{\Gamma \vdash B}\ \exists\text{-elim}$$

elle correspond à :

$$\frac{\Gamma \vdash \texttt{t} : A[x \leftarrow a]}{\Gamma \vdash \texttt{(a,t)} : \exists x.A}\ \text{Wit} \qquad \frac{\Gamma \vdash \texttt{c} : \exists x.A \quad \Gamma, A \vdash \texttt{u} : B \quad \text{où } x \notin FV(\Gamma) \text{ et } x \notin FV(B)}{\Gamma \vdash \texttt{let x=c in u} : B}\ \text{Let}$$

Extension à la quantification universelle : est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A \quad \text{où } x \notin FV(\Gamma)}{\Gamma \vdash \forall x.A}\ \forall\text{-intro} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \leftarrow a]}\ \forall\text{-elim}$$

elle correspond à :

$$\frac{\Gamma \vdash \texttt{t} : A \quad \text{où } x \notin FV(\Gamma)}{\Gamma \vdash \lambda\texttt{x.t} : \forall x.A} \qquad \frac{\Gamma \vdash \texttt{t} : \forall x.A}{\Gamma \vdash \texttt{ta} : A[x \leftarrow a]}$$

Exercice 4. Pour chacune des formules suivantes, donner un arbre de preuve et le terme correspondant :

$$(\forall x, Px \Rightarrow Qx) \Rightarrow (\forall x, Px) \Rightarrow (\forall x, Qx).$$
$$(\exists x, Px \Rightarrow Qx) \Rightarrow (\forall x, Px) \Rightarrow (\exists x, Qx)$$

II.4 - L'assistant à la preuve Coq

Coq est un assistant à la preuve interactif. Il repose sur un langage de programmation fonctionnel (une extension du λ-calcul) et sur le calcul des constructions inductives (une extension de la logique intuitionniste).

En pratique, lorsque l'on prouve une proposition en Coq, on part d'un but (*Goal*) et d'hypothèses (*Assumptions*) et on construit l'arbre de preuve du jugement *Assumptions* $\vdash$ *Goal* à l'aide de tactiques qui transforment un but en une liste de buts à résoudre.

Les tactiques de Coq qui permettent de faire des preuves logiques correspondent aux règles de la logique intuitionniste :

Tactique	**Règle logique**
`Assumption, Apply H`	Axiome
`intro, intros`	$\Rightarrow,\forall,\neg$-intros
`apply`	$\Rightarrow,\forall$-elim
`split`	$\wedge,\top$-intro
`left, right`	$\vee$-intro left, right
`exists`	$\exists$-intro
`destruct`	$\wedge,\vee,\bot,\exists$-elim

Exercice 5. Pour chacune des formules suivantes, donner une preuve en logique intuitionniste et une preuve Coq. Regarder le programme correspondant en utilisant la commande `Print`.

```
Lemma and_to_imp: (P /\ Q -> R) -> (P -> (Q -> R)).
Lemma and_e_left : P /\ Q -> P.
Lemma and_sym : P/\Q -> Q/\P.
Lemma or_sym : P\/Q -> Q\/P.
```

Grâce à la correspondance de Curry-Howard, Coq permet à la fois de prouver des théorèmes (des formules logiques représentant des énoncés mathématiques) et de certifier les programmes.

Plus précisément, un programme bien typé est une preuve de la formule mathématique correspondant à son type. D'autre part, une formule est prouvée lorsqu'elle correspond à un type *habité*, c'est-à-dire qu'il existe un programme ayant ce type.

Ainsi, en Coq, prouver une proposition, un lemme, c'est construire son arbre de preuve.

```
Lemma lem_identity : A -> A.
  intros.
  assumption.
  Qed.
```

```
Lemma lem_identity : A -> A.
  exact (fun (a:A) => a).
  Qed.
```

Une fois la preuve faite, on fournit le λ-terme correspondant au noyau de Coq qui se charge de vérifier que le λ-terme est bien typé, c'est-à-dire que le terme prouve bien le lemme.

D'ailleurs, il est possible de prouver ce lemme en donnant directement le programme correspondant à son arbre de preuve.

Exercice 6. Prouver la formule suivante en fournissant un terme dont le type correspond à la formule.

```
Lemma evaluation : A -> (A -> B) -> B.
```

Réciproquement, il est possible de définir un terme en donnant l'arbre de preuve correspondant à sa dérivation de typage.

Par exemple la première projection peut-être définie par un terme ou un arbre de preuve :

```
Definition fst_proj :
 nat -> nat -> nat.
  intros n m.
  apply n.
  Defined.
```

```
Definition fst_proj :
  nat -> nat -> nat :=
    fun a _ => a.
```

Exercice 7. Définir le programme correspondant à la seconde projection en donnant la preuve de son type.

```
Definition snd_proj : nat -> nat -> nat.
```

Pour finir, quelques mots clefs du langage de Coq à retenir :

`Lemma`, `Proposition`, `Theorem : type.`	Introduisent des formules à prouver
`Qed.`	Termine une preuve
`Definition cst : type := terme.`	Introduit une constante en donnant son terme
`Check t.`	Donne le type d'un terme, la formule prouvée
`Print t.`	Donne le terme et le type correspondant à une preuve

III - Les inductifs

Cette partie du cours est illustrée par le fichier `coq_inductive.v`.

III.1 - Types inductifs

Afin de construire des types de donnés, on utilise l'instruction `Inductive`. Elle permet d'énumérer les termes (programmes) de base habitant ce types.

Exemple 1.

Le type des booléens est construit de cette façon :

```
Inductive bool:Set := true: bool | false: bool
```

Il existe donc deux valeurs de type `bool` : `true` et `false`.

Ce type peut-être utilisé pour définir des programmes à l'aide de définitions par cas (*pattern matching*).

Exemple 2.

```
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
```

Enfin, pour tout type énuméré, la tactique `destruct` permet de raisonner par cas.

Exemple 3.

« Pour prouver un prédicat sur les booléens, il suffit de le vérifier pour `true` et `false` ».

```
Check bool_rect.
 bool_rect: forall P: bool -> Prop,
        P true => P false => forall b:bool, P b
```

En pratique, on peut utiliser ce théorème comme dans la preuve suivante :

```
Lemma no_other_bool :
 forall b, b = true ⋁ b = false.
Proof.
 intro.
 destruct b.
 (* premier cas: b=true *)
 left. reflexivity.
 (* deuxième cas: b=true *)
 right; reflexivity.
Qed.
```

Exercice 8. Définir le type `option_nat` permettant de construire des programmes de type `nat` lorsqu'ils sont définis.

L'instruction `Inductive` permet de construire des types plus complexes que les types énumérés : les types *inductifs* dont les constructeurs de termes peuvent faire appel à d'autres termes comme dans l'exemple des entiers.

Exemple 4.

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat
```

Il y a deux façons de construire un terme de type `nat` : 0 est de type `nat` ou à partir d'un terme `n` de type `nat` et du constructeur successeur `S` on obtient un terme `S n` de type `nat`.

Comme pour les types énumérés, les types inductifs permettent des définitions par cas.

Exemple 5.

Ainsi, le type `nat` peut être utilisé pour construire des programmes récursifs

```
Fixpoint fact (n:nat) : nat :=
   match n with
   | 0 => 1
   | S u => (S u)*(fact u)
   end.
```

ou pas :

```
Definition pred (n:nat) : nat :=
   match n with
   | 0 => 0
   | S u => u
   end.
```

Tout type inductif vient avec un *principe d'induction*. On utilise ce principe dans une preuve via la tactique `induction`.

Exemple 6.

Le principe d'induction se ramène au principe de récurrence sur les entiers : « pour prouver un prédicat sur les entiers, il suffit de le montrer pour le cas de base 0 et de montrer qu'il est stable par le constructeur `S` (s'il est vrai pour `n` alors il est vrai pour `S n`).

```
Check nat_rect.
 nat_rect : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Exercice 9. Définir le type des listes d'entiers, puis le programme permettant de calculer la longueur d'une liste d'entiers. Définir le programme permettant de concaténer deux listes et montrer que la longueur de la liste concaténée est égale à la somme des longueurs des deux listes.

Cet exercices est illustré par le fichier `lists.v`.

III.2 - Prédicats inductifs

Cette partie du cours est illustrée par le fichier `lesson_ind_predicate.v`.

Rappelons qu'un *prédicat* est une proposition portant sur une ou plusieurs variables.

Exemple 7.

Par exemple, le prédicat « être dans l'ordre lexicographique » porte sur deux couples d'entier. En Coq, il prendra la forme :

```
Check lexico.
lexico : nat*nat -> nat*nat -> Prop
```

Le prédicat « être pair » porte sur les entiers. En Coq, il prendra la forme :

```
Check even.
even : nat -> Prop
```

Un prédicat peut-être défini ou bien de manière directe, en utilisant la construction `Definition` ou la construction `Fixpoint` lorsque la définition est récursive.

Exemple 8.

```
Definition lexico (p q :nat*nat)  : Prop :=
   (fst p < fst q) \/ (fst p = fst q /\ snd p < snd q).

Fixpoint Even (n:nat) : Prop :=
  match n with
  | 0 => True
  | S 0 => False
  | S (S n') => Even n'
  end.
```

Mais on peut aussi utiliser une définition *inductive*. Dans ce cas, on donne les *constructeurs* élémentaires de preuve des formules construites à partir de ce prédicat.

Exemple 9.

```
Inductive lex : nat*nat -> nat*nat -> Prop :=
  | lex_fst : forall a a' b b',  a < a' -> lex_lt (a,b) (a',b')
  | lex_snd : forall a b b',  b < b' -> lex_lt (a,b) (a,b').
```

Il y a deux constructeurs de preuves de l'ordre lexicographique. Le premier `lex` construit une preuve de `lex (a,b) (a',b')` à partir d'une preuve de `a < a'`. Le second construit une preuve de `lex (a,b)  (a,b')` à partir d'une preuve de `b < b'` (remarquez que les deux premières composantes sont identiques dans ce cas).

```
Inductive even : nat -> Prop :=
 | even_0 : even 0
 | even_SS : forall n, even n -> even (S (S n)).
```

Il y a deux constructeurs de preuves de la formule `even n`. La première, lorsque $n = 0$ on a, par définition du prédicat `even` la preuve `even_0` de `even 0`. La seconde, lorsque n est de la forme $S(Sn')$, le constructeur `even_SS` permet d'obtenir une preuve de `even S(S n)` à partir d'une preuve de `even n`.

Les deux manières de définir les prédicats sont équivalentes, mais elles servent dans des cas différents selon le contexte où ils sont utilisés. Par exemple, lorsque l'on veut faire un calcule, on utilisera une définition directe, alors que l'on veut faire un preuve on utilisera une définition inductive.

Exemple 10.

Il est plus facile de montrer que 100 est pair en utilisant la première définition :

```
Lemma even_100 : Even 100.
 simpl. auto.
Qed.
```

Il est plus facile de montrer le lemme suivant en utilisant la définition inductive car elle vient avec un principe d'induction qui permet de retrouver les cas permettant de construire une preuve du prédicat :

```
Check even_ind.
 even_ind : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n

Lemma even_double :
 forall n, ev n -> exists m, m+m=n.
Proof.
 intros n H.
 induction H.
 (* La preuve H:ev n provient du constructeur ev_0 et n=0*)
  exists 0; reflexivity.
 (* La preuve H:ev n provient du constructeur plus_2_ev à partir d'une
 preuve IHev:exists m, m+m=n' avec n= S S n'*)
  destruct IHev as [m].
  exists (S m).
  omega.
Qed.
```

En pratique, on donne souvent les deux types de définition, on prouve qu'elle sont équivalentes et on utilise l'une ou l'autre selon les cas.

Exemple 11.

```
Lemma lex_equiv : forall p q,
 lexico p q <-> lex p q.
Proof.
 intros (a,b) (a',b').
 split.
 intro H. destruct H. simpl in H.
 apply lex_fst.  assumption.
 simpl in H. destruct H. rewrite <- H.
 apply lex_snd.  apply H0.
 intro H. inversion H.
 left. auto.
 right. auto.
Qed.
```

Les deux exemples précédents utilisent des tactiques qui permettent de déconstruire les prédicats. La première tactique permet de faire une étude par cas : `inversion H` avec `H:lex (a, b) (a', b')`. La seconde tactique permet de faire une preuve par induction `induction  H` où `H:even  n`, elle utilise le principe d'induction décrit précédemment.

Exercice 10.

1. Donner une définition directe et une définition inductive du prédicat `sorted` « être trié » sur les listes.

2. Montrer que les deux définitions sont équivalentes.

3. Montrer le théorème `sorted le (1::2::3::nil).`

4. Montrer le théorème `~(sorted le (1::3::2::nil)).`, on pourra introduire les lemmes suivant : `forall (A:Set) (R: A -> A-> Prop) (x:A) (l: list A), sorted R (cons x l) -> sorted R l.` et `forall (A:Set) (R: A -> A-> Prop) (x y :A) (l:list A), sorted R (cons x (cons y l)) -> R x y.`

Exercice 11. À l'aide de la commande `Print` comprendre comment sont construits les prédicats `True` et `False`.

IV - Spécification et certification des programmes en Coq

IV.1 - Preuve de programme, un exemple

Cette partie du cours est illustrée par le fichier `binaire.v`

Nous allons détailler l'exemple des *arbres binaires de recherche* afin de montrer comment utiliser Coq pour construire des programmes certifiés.

Les arbres binaires de recherche forment une structure de données permettant de représenter les ensembles avec une fonction de recherche efficace.

On considère les arbres binaires de recherche sur les entiers. Ce sont des arbres binaires vérifiant la structure suivante : s'il n'est pas vide, l'arbre possède une racine dont l'étiquette est plus grande que les racines du sous-arbre gauche et plus petite que celles du sous-arbre droit et les deux sous-arbres sont eux mêmes des arbres binaires de recherche.

Pour représenter les arbres binaires de recherche en Coq, on commence par définir le *type* inductif des arbres binaires :

```
Inductive tree : Set :=
| Empty
| Node : tree -> nat -> tree -> tree.
```

Ensuite, on définit un *prédicat inductif* vérifiant qu'un arbre binaire est un arbre binaire de recherche.

```
Inductive bst : tree -> Prop :=
| bst_empty : bst Empty
| bst_node : forall x l r, bst l -> bst r ->
                  (forall y, In y l -> y < x) ->
                  (forall z, In z r -> x < z) -> bst (Node l x r).
```

On veut définir une fonction `search x t` de recherche qui retourne le booléen `true` si et seulement si `x` appartient à l'arbre binaire de recherche `t` en tirant partie de la structure particulière de l'arbre.

Pour pouvoir énoncer le théorème de correction de l'arbre, il nous faut introduire le prédicat d'appartenance :

```
Inductive In (n:nat) : tree -> Prop :=
| Inleft : forall l x r, (In n l) -> In n (Node l x r)
| Inright : forall l x r, (In n r) -> In n (Node l x r)
| Inroot : forall l r, In n (Node l n r).
```

On peut alors énoncer la spécification de la fonction `search`

```
Theorem search_correct: forall (n:nat) (t:tree), bst t ->  (search n t = true <-> In n t).
```

Définissons maintenant cette fonction :

```
Fixpoint search (n:nat) (t:tree) :=
match t with
 | Empty => false
 | Node l x r => match nat_compare n x with
                    | Lt => search n l
                    | Eq => true
                    | Gt => search n r
                end
end.
```

Exercice 12. Prouver en Coq le théorème de correction.

Enfin, on peut extraire le programme `Caml` associé à cette fonction à l'aide de la ligne de commande Coq

```
Extraction "search.ml" search.
```

On obtient le code Caml suivant :

```
type bool =
| True
| False

type nat =
| O
| S of nat

type comparison =
| Eq
| Lt
| Gt

(** val nat_compare : nat -> nat -> comparison **)

let rec nat_compare n m =
  match n with
  | O ->
    (match m with
     | O -> Eq
     | S n0 -> Lt)
  | S n' ->
    (match m with
     | O -> Gt
     | S m' -> nat_compare n' m')

type tree =
| Empty
| Node of tree * nat * tree

(** val search : nat -> tree -> bool **)

let rec search n = function
| Empty -> False
| Node (l, x, r) ->
  (match nat_compare n x with
   | Eq -> True
   | Lt -> search n l
   | Gt -> search n r)
```

IV.2 - Spécifier les fonctions partielles

Cette partie du cours est illustrée par les fichiers `facto.v` et `pred.v`

Représenter un programme par un terme en Coq est parfois difficile, parfois même impossible. En effet, tous les termes doivent définir des calculs qui terminent.

Exemple 12.

Le programme factoriel en Caml peut être défini par :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact(n-1);;
```

Ce programme ne termine pas toujours. En effet, si $n \leq 0$, alors elle boucle.

Il existe plusieurs solutions pour contourner cette difficulté.

On peut utiliser un prédicat caractérisant la relation entre un argument et le résultat d'un programme.

Exemple 13.

On définit le prédicat `Pfact n m` qui est vrai lorsque le calcule de la factorielle de `n` termine et

vaut `m` :

```
Require Import ZArith.
Open Scope Z_scope.

Inductive Pfact : Z -> Z -> Prop :=
  | Pfact_0 : Pfact 0 1
  | Pfact_h : forall n v : Z, n <> 0%Z -> Pfact (n-1) v -> Pfact n (n*v).
```

Exercice 13. Montrer que le domaine de définition de la factorielle est l'ensemble des entiers positifs, c'est-à-dire la proposition : `forall n v : Z, Pfact n v -> 0 <= n.` Il est possible de prouver le contraire en utilisant les ordres bien fondés.

On peut utiliser le type `option`.

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

Exemple 14.

On peut définir une fonction partielle représentant le prédecesseur :

```
Definition pred (n:nat) : option nat :=
  match n with
      0 -> None
   |S n -> Some n
  end.
```

IV.3 - Spécifier avec les types

Les types permettent de donner des spécifications plus ou moins fortes des programmes.

Exemple 15.

* Un entier premier et plus grand que 5 est de type `nat`
* Une fonction de tri sur les entiers peut être typée par : `nat -> nat`
* La fonction prédecesseur peut être typée par : `nat -> option nat`

Les spécifications de l'exemple précédent n'apportent pas beaucoup d'information sur le résultat si ce n'est qu'il est bien défini.

Afin de définir des spécifications plus précises, on introduit des nouveaux types.

Exemple 16.

En Coq, le type `{p:nat | 5<p /\ is_prime p}` est le sous-ensemble des entiers qui vérifient les deux propositions « être plus grand que 5 » et « être premier ».

Exercice 14.

1. Quel serait le type de la fonction de tri sur les entiers ?

2. Quel serait le type pour la division euclidienne ?

Le constructeur de type `{x:A | P x}` se rapproche de l'existentiel : un programme de ce type — c'est-à-dire cette spécification devra fournir un témoin de l'existence de `x:a` et un certificat (une preuve) de `P x`.

coq_inductive.v

```
Require Import Arith List.

(* 1 Basic usage of inductive types. *)

Print bool.

(* case definition *)
Check andb.

Definition andb' (b1:bool) (b2:bool) : bool :=
  match b1 with
    | true => b2
    | false => false
  end.
Print andb'.

(* inductive principle *)
Check bool_rect.
Print bool_rect.

Definition bool_r (P:bool->Type) (f_0: P true) (f_1: P false) (b:bool): P b :=
  match b with
    |true => f_0
    |false => f_1
  end.
Print bool_r.

(* elim/induction : direct use of inductive principle  *)

Lemma no_other_bool :
 forall b, b = true \/ b = false.
Proof.
  destruct b.
  left. reflexivity.
  right; reflexivity.
Qed.
Print no_other_bool.

(* case : simple pattern matching *)

Lemma no_other_bool' : forall b, b = true \/ b = false.
Proof.
  intro b.
  case b.
  left; reflexivity.
  right; reflexivity.
Qed.
Print no_other_bool'.

Lemma case_danger : forall b, b = true -> b = true.
Proof.
  intros.
  (*case b.*) (*Does not work ... for that use destruct*)
  assumption.
Qed.
```

```
Print option.
Lemma inverse_option :
  forall A:Set, forall a b:A,
  Some a = Some b -> a = b.
Proof.
  intros.
  (* injection H. *)
  (* how it works : *)
  set (phi := fun o =>
    match o with
      | Some x => x
      | None => a
    end).
  change (phi (Some a) = phi (Some b)).
  rewrite H.
  reflexivity.
Qed.

(* NB: This kind of result is only true with inductive objects :
       in general we don't have f x = f y  -> x = y. *)
(* NB: But the reverse fact is general : x = y -> f x = f y.
   See rewrite ... *)

(* la tactique discriminate *)

Inductive day:= lun | mar | mer | jeu | ven | sam | dim.

Lemma dd : lun <> mar.
Proof.
(* discriminate. *)
  intuition.
  change ((fun d:day => match d with |lun => True | _ => False end) mar).
  rewrite <- H. trivial.
Qed.

(* 2 Really recursive Types *)

Print nat.
Check nat_rect.
Print nat_rect.

Fixpoint rec (P : nat-> Type) (p_0 : P 0) (p_1 : forall n, P n -> P (S n)) (n:nat) : P n
  match n with
    |0 => p_0
    | S n_0 => p_1 n_0 (rec P p_0 p_1 n_0)
  end.
Print rec.

(** In fact, any induction is a use of such induction principle *)

Lemma test: forall n, n+0 = n.
induction n.
simpl. reflexivity.
simpl. rewrite IHn. reflexivity.
Qed.
```

```
Print test. (** note: nat_ind = nat_rect *)

Print nat.
Check nat_rect.

Require List.
Print list.
Check list_rect.
Check cons.
Print cons.

(* pattern matching definition *)

Fixpoint length (X:Type) (l:list X) : nat :=
  match l with
  | nil => 0
  | cons h t => S (length X t)
  end.
Print length.
```

lists.v

```
(** Donnees **)

Inductive list (A : Set) : Set :=
| Nil : list A
| Cons : A -> list A -> list A.

(** Programmes **)

Fixpoint concat (A : Set) (l1 l2 : list A) {struct l1} : list A :=
 (* fonction recursive definie par cas *)
  match l1 with
  | Nil       => l2
  | Cons x tl => Cons A x (concat A tl l2)
  end.
Check concat.

Fixpoint length (A : Set) (l : list A) {struct l} : nat :=
 (* fonction recursive definie par cas *)
  match l with
  | Nil       => 0
  | Cons x tl => 1 + length A tl
  end.

(** Des proprietes **)

Lemma Concat_Nil: forall (A : Set) (l : list A), concat A (Nil A) l = l.
 (* Introduit les hypotheses *)
  intros A l.
 (* On calcule *)
  simpl.
 (* Reflexivite de l'egalite *)
  reflexivity.
Qed.
Lemma Concat_Nil': forall (A : Set) (l : list A), concat A l (Nil A) = l.
 (* Introduit les hypotheses *)
  intros A l.
 (* On raisonne par induction sur la structure de la liste *)
  induction l.
 (* Premier cas : l = Nil *)
  simpl. auto.
 (* Deuxieme cas : l = a::l *)
  simpl. rewrite IHl. auto.
Qed.
Lemma Concat_Length : forall (A : Set) (l1 l2 : list A),
  length A (concat A l1 l2 ) = length A l1 + length A l2.
 (* Introduit hypotheses *)
  intros A l1 l2.
 (* Induction sur la structure de l1 *)
  induction l1.
 (* Premier cas : l1 = Nil *)
  simpl. auto.
 (* Deuxieme cas : l1 = a::l1 *)
  simpl. rewrite IHl1. auto.
Qed.
```

lesson_ind_predicate.v

```
Require Import Arith Bool List Omega.

(* 1 non-recursive predicate *)
(* first possibility : just reformulation of a predicate ... *)

Definition lex_lt_orig (p q :nat*nat) :Prop :=
  (fst p < fst q) \/ (fst p = fst q /\ snd p < snd q).

Inductive lex_lt : nat*nat -> nat*nat -> Prop :=
  | lex_fst : forall a a' b b',  a < a' -> lex_lt (a,b) (a',b')
  | lex_snd : forall a b b',  b < b' -> lex_lt (a,b) (a,b').

Lemma lex_equiv : forall p q,
 lex_lt_orig p q <-> lex_lt p q.
Proof.
 intros (a,b) (a',b').
 split.
 intro H. destruct H. simpl in H.
 apply lex_fst.  assumption.
 simpl in H. destruct H. rewrite <- H.
 apply lex_snd.  apply H0.
 intro H. inversion H.
 left. auto.
 right. auto.
Qed.

(* Here, no recursion. Interest:
  - more readable:
     * splits the cases
     * allows to speak of sub-objects (builds instead of breaking)
     * some equalities can be avoided (no a' above)*)

Print True.
Print False.
Print or.
Print or_ind.
Lemma or_sym : forall A B,
 A\/B -> B\/A.
Proof.
intros.
(*
destruct H.
right; assumption.
left; assumption.
*)
exact
(match H with
 | or_introl a => or_intror _ a
 | or_intror b => or_introl _ b
end).
Qed.
Print and.
Print prod.

(* 2 A first inductive predicate with recursion. *)
```

```
(* The main use of inductive predicate is recursivity: *)

Inductive even : nat -> Prop :=
 | even_O : even O
 | even_SS : forall n, even n -> even (S (S n)).

Lemma even_4 : even 4.
Proof.
apply even_SS.
apply even_SS.
apply even_O.
Qed.

Lemma odd_1 : ~ even 1.
Proof.
intro.
inversion H.
Qed.
Check even_ind.

(* Meaning: the smallest set of integers closed under ... *)
(* See for example a Prolog program:
      even(o).
      even(s(s(N))) :- even(N).
*)

(* 3 Could/Should we do otherwise ? *)

(* An equivalent predicate without inductive type: *)
Fixpoint Even (n:nat) : Prop := match n with
 | 0 => True
 | 1 => False
 | S (S n) => Even n
end.

(* Interest here: you can simplify automatically a concrete problem *)

Lemma Even_100 : Even 100.
 simpl.
 exact I.
Qed.

Lemma Odd_1 : ~ Even 1.
Proof.
 simpl.
 intro.
 auto.
Qed.
Print Even_100.
Print even_4.

Hint Constructors even.
Lemma even_100 : even 100.
 auto 51.
Qed.
Print even_100.
```

```
(* But the computability is not necessary implied by a Fixpoint.*)

Fixpoint Even' (n:nat) : Prop := match n with
 | 0 => True
 | S n => ~Even' n
end.
Lemma Even'_6 : Even' 6.
 simpl.
 intuition.
Qed.
Print Even'_6.

(* For ensuring computability, you should rather use bool. *)

Fixpoint even_bool (n:nat) : bool := match n with
 | 0 => true
 | S n => negb (even_bool n)
end.
Lemma even_bool_100 : even_bool 100 = true.
 simpl.
 reflexivity.
Qed.

(* 4 Inductive predicate & induction principle : *)

Print even.
Check even_ind.
Print even_ind.
Lemma even_double :
 forall n, even n -> exists m, m+m=n.
Proof.
intros n H.
induction H.
exists 0; reflexivity.
destruct IHeven as [m HT].
exists (S m). omega.
Qed.

(* one more tactic of interest : inversion. *)
Lemma one_not_even : ~ (even 1).
Proof.
  unfold not.
  intro.
  inversion H.
Qed.
```

facto.v

```
(* Definir une  fonction par un predicat *)

Require Import ZArith.
Open Scope Z_scope.
Print Scope Z_scope.

Inductive Pfact : Z -> Z -> Prop :=
| Pfact_0 : Pfact 0 1
| Pfact_h : forall n v : Z, n > 0%Z -> Pfact (n-1) v -> Pfact n (n*v).

(* Proprietes du predicat *)

Lemma pfact3: Pfact 3 6.
Proof.
  apply (Pfact_h 3 2). auto with zarith.
  apply (Pfact_h 2 1). auto with zarith.
  apply (Pfact_h 1 1). auto with zarith.
  apply Pfact_0.
Qed.

(* Domaine de la fonction *)

Check Pfact_ind.
Theorem f_dom : forall n v : Z, Pfact n v -> 0 <= n.
Proof.
  intros.
  induction H.
  auto with zarith. (* cas de base *)
  auto with zarith. (* heredite *)
Qed.

Print Z.
Print positive.
Require Import Zwf.
Print well_founded_ind.
Check (Zwf 0).

Theorem f_dom' : forall n:Z, 0 <= n -> exists v:Z, Pfact n v.
Proof.
  intros n.
  (* on applique well_founded_ind a la relation Zwf 0 bien fondee*)
  elim n using (well_founded_ind (Zwf_well_founded 0)).
  intros x Hrec Hle.
  (* on separe cas de base et heredite *)
  SearchPattern (_<_ \/ _=_).
  elim (Zle_lt_or_eq _ _ Hle).
  (* cas1: x<0 - heredite *)
  intros. elim (Hrec (x-1)).
  intros. exists (x*x0). apply (Pfact_h x x0). auto with zarith. apply H0.
  unfold Zwf. auto with zarith.
  auto with zarith.
  (* cas2: x=0 - base*)
  intros. rewrite <- H.  exists 1. apply Pfact_0.
Qed.
```