Chapter 1

theme03curryhowardstudent

```
Require Import Arith.
```

Dans ce thème, nous allons prendre un peu de recul sur une notion essentielle de la spécification et la validation des programmes : la *correspondance de Curry-Howard*.

Le principe est simple: le rapprochement entre d'un part la logique d'un part, et le calcul de l'autre part. Cette correspondance s'est co-développée avec le lambda-calcul typé qui intègre très naturellement les deux mondes :

- les termes pour exprimer les calculs
- les types pour exprimer les propriété des calculs.

1.1 La vérité

La vérité en Coq se nomme *True* et n'est pas primitive, mais définie dans la bibliothèque standard.

```
Print True.
```

```
Inductive True : Prop :=
  I : True
```

Il s'agit d'un type somme avec un unique constructeur *I*. Si on vous pose la question (par exemple au bac philo): "qu'est-ce-que la vérité ?" vous pourrez donc répondre "et bien c'est un type somme avec un seul constructeur sans argument".
On peut construire un type similaire en Ocaml :

```
type vrai = V
```

Mais la traduction de ce dernier en Coq serait plutôt :

```
Inductive Vrai : Set :=
  V : Vrai
```

car `Set` représente les types de données alors que `Prop` est spécialisé pour les propositions logiques. La différence est importante (en théorie mais également en pratique) et nous y reviendrons.
Notre première preuve consistera à montrer que *True* est ... vrai !
Pour construire une preuve de *True*, on doit tout simplement montrer qu'il existe au moins un élément habitant ce type (on dit que *True* est habité). Bien sûr on n'a pas le choix puisque l'unique habitant du type *True* est le terme *I*.

```
Lemma True_est_Vrai: True.
  apply I.
Qed.
```

Ce raisonnement correspond en logique à une règle standard de logique :

$$\frac{}{\Gamma \vdash True}\ (I)$$

Dans la règle ci-dessus le Γ est le contexte des hypothèses séparé le but à droite du symbole $\vdash$. Il s'agit d'une représentation dite de *calcul des séquents* proche du mode preuve interactive de Coq.
Montrer que *True* est vrai de façon aussi triviale explique bien que le type *True* et le terme associé T ne jouent pas de rôle fondamental. Une propriété A est vraie s'il existe un terme a de type A et que l'on peut construire ce dernier. Si on sait faire ça alors il est trivial de déduire *True*

```
Example ex_true_prop: 1 + 1 = 2.
Proof.
  simpl.
  reflexivity.
Qed.

Example ex_true_prop_true: (1 + 1 = 2) → True.
Proof.
  intro H.
  apply I.
Qed.

Example ex_true_true_prop: True → 1 + 1 = 2.
Proof.
  intro H.
  simpl.
```

```
  reflexivity.
Qed.
```

Donc voilà, le *True* ne sert pas vraiment ... Ce qu'il faut retenir, c'est qu'il y a un rapport entre le type de donnée à un seul constructeur constant (comme le type *unit* de Ocaml) et la notion de vérité.

1.2 Le faux

Pour la fausseté, c'est encore plus simple : il s'agit d'un type somme vide ! En effet, on doit pouvoir déclarer un type ne pouvant être habité par aucun terme.
En Coq le type *False* est défini de la façon suivante :

```
Print False.


Inductive False : Prop :=.
```

Ici le type n'a pas de constructeur, on ne peut donc pas construire quoique ce soit avec.
Question : peut-on définir un type non-habité en Ocaml ?
Remarque : en logique classique le faux est tout simplement la négation du vrai (ou vice-versa, mais il faut bien démarrer quelque part). On a par définition : *False* = *not True*. En fait la logique classique repose sur une séparabilité entre une proposition et sa négation, c'est le fameux axiome du tiers exclus :

$$\frac{}{\Gamma \vdash P \vee \neg P} \; (tiers - exclu)$$

En Coq il est possible d'ajouter des axiomes classiques mais il privilégie la logique intuitioniste et les preuves constructives. Pour beaucoup, une preuve non-constructive (donc exploitant le tiers-exclus ou autre axiome classique équivalent) n'est pas une preuve recevable. Sans entrer dans la philosophie, il est clair que nous avons jusqu'à présent construit nos preuves (sans vraiment le savoir) car à chaque fois un terme du lambda-calcul de Coq témoigne du type correspondant à ce qui est démontré.
Dans ce cours, notamment parce que c'est beaucoup plus naturel dans le cadre du rapprochement entre le raisonnement et le calcul, nous resterons constructifs.
Renoncer au tiers-exclu n'empêche pas d'effectuer des raisonnements par contradiction.
Le *False* peut se trouver des deux côtés d'un raisonnement :

- soit en hypothèse
- soit en conclusion

Etudions tour-à-tour ces deux situations.
Si *False* est en hypothèse, alors on peut exploiter directement le principe d'induction généré par Coq à partir de la définition de *False*.

```
Check False_ind.
```

$False\_ind$: `forall` P : `Prop`, $False \to P$
Donc si *False* se trouve en hypothèse (autrement dit, au moins une de nos hypothèses est fausse), alors on a une contradiction et on peut conclure positivement la preuve.
On se retrouve donc assez souvent à chercher à invalider une hypothèse la réduisant à *False*. C'est l'essence du raisonnement par contradiction.
Le principe d'induction ci-dessus correspond à un raisonnement qui consiste grossièrement à "éliminer" l'hypothèse fausse et conclure. On peut résumer ce raisonnement ainsi :

$$\frac{}{\Gamma, False \vdash P}\ (False - elim) \text{ pour n'importe quelle proposition } P$$

Reproduisons ce schéma sur quelques exemples.

```
Example false_elim_ex1:
  False → 2 + 2 = 5.
Proof.
  intro HFalse.
  elim HFalse. (*  élimination *)
Qed.
```

```
Example false_elim_ex1':
  False → 2 + 2 = 4.
Proof.
  intro HFalse.
  elim HFalse. (*  élimination *)
Qed.
```

On voit bien sur ces exemples que la vérité (ou non) de la conclusion n'influence pas la preuve : une hypothèse fausse suffit pour conclure.
Si une hypothèse n'est pas directement *False*, on peut utiliser diverses moyens pour montrer une contradiction. On peut notamment utiliser la tactique *absurd*.

```
Example false_elim_ex2:
  2 + 2 = 5 → 8 × 0 = 42.
Proof.
  intro Hcontra.
  compute in Hcontra.
  absurd (4 = 5).
  SearchPattern ( _ ≠ _ ).
  (*  n_Sn: forall n : nat, n <> S n *)
  apply n_Sn.
  exact Hcontra.
Qed.
```

Dans le cas présent, on a une égalité qui est naturellement fausse, un moyen rapide de résoudre cette contradiction est d'utiliser la tactique `inversion` que l'on utilise également pour décomposer une égalité (cf. section sur l'égalité).

```
Example false_elim_ex2':
  2 + 2 = 5 → 8 × 0 = 42.
Proof.
  intro Hcontra.
  inversion Hcontra. (*  preuve terminée ! *)
Qed.
```

Enfin, on peut construire de toute pièce une contradiction et terminer par la tactique *contradiction*.

```
Example false_elim_ex2'':
  2 + 2 = 5 → 8 × 0 = 42.
Proof.
  intro Hcontra.
  assert (H: 2 + 2 ≠ 5).
  { simpl.
    apply n_Sn.
  }
  contradiction.
Qed.
```

1.2.1 Exercice

Montrer les lemmes suivant :

```
Lemma times_two:
  forall n : nat, 2 × n = n + n.
```

```
Lemma plus_one_S:
  forall n : nat, n + 1 = S n.
```

et en déduire :

```
Lemma false_exo:
  forall n k : nat,
    2 × n = k → n + n = k + 1
    → n × 4 = k.
```

Si *False* est en conclusion, alors on n'a pas d'autre moyen que de trouver une contradiction dans les hypothèses.
Autrement dit, pour prouver $P \rightarrow False$ alors il faut montrer que P est contradictoire.
Ceci nous amène naturellement vers la négation logique.

1.3 La négation logique

Dans la logique intuitioniste de Coq le faux n'est pas la négation de vrai mais alors qu'est-ce-que la négation alors ?
La réponse est assez simple : la négation est simplement l'implication d'une contradiction.
Autrement dit :

$$\neg P \text{ est défini par } P \rightarrow False$$

On peut le vérifier en Coq :

```
Print not.
not = fun A : Prop => A -> False : Prop -> Prop
```

On utilise donc souvent `unfold` *not* pour mettre à jour des contradictions.
Sans tiers-exclu on n'a pas d'équivalence logique entre P et $not\ (not\ P)$.
En fait, dans un sens tout se passe bien :

```
Lemma not_not:
  forall P : Prop, P -> not (not P).
Proof.
  intros P HP.
  unfold not.
  intro Hfalse.
  apply Hfalse.
  exact HP.
Qed.
```

En revanche, pour prouver l'implication converse, il est nécessaire d'introduire un axiome de logique classique équivalence à l'axiome du tiers-exclu.
On ouvre d'abord une `Section` pour ne pas "polluer" l'environnement avec un axiome exterieur.

```
Section not_not_classic.
```

Avec `Hypothesis` on peut introduire notre axiome classique en le limitant à la section courante.

```
  Hypothesis exluded_middle: forall P : Prop, P \/ not P.
```

Pour ensuite prouver notre lemme converse. On essaiera de se convaincre que sans un tel axiome la preuve ne peut être terminée.

```
  Lemma not_not': forall P : Prop, not (not P) -> P.
  Proof.
    intros P HnnP.
    unfold not in HnnP.
    assert (Hem: P \/ not P).
    apply exluded_middle.
    destruct Hem as [HP | HnP]. (*  cf. disjonction *)
```

```
  - (*  cas P *)
    exact HP.
  - (*  cas (not P) *)
    assert (Hfalse: False).
    { apply HnnP.
      exact HnP.
    }
    inversion Hfalse.
Qed.
```

End not_not_classic.

Remarque : ici le lemme *not_not'* est paramétré par l'axiome du tiers-exclu. On ne pourra l'appliquer qu'en faisant l'hypothèse de ce dernier.

```
Example use_not_not':
  not (not True) → True.
Proof.
  apply not_not'.
```

A partir d'ici Coq nous donne le but suivant :

```
  ============================
   forall P : Prop, P ∨ ~ P
```

Donc on ne peut pas conclure.

```
Abort.
```

1.3.1 Exercice

Question 1

Montrer que $\neg (2 * 3 = 5)$.
Remarque : *not P* peut s'écrire ~*P* et *not* $(a = b)$ peut s'écrire $a \neq b$.

Question 2

Montrer par induction (et sans utiliser de lemme auxiliaire) :

```
Lemma S_inject_r:
  forall n : nat, n ≠ S n. (*  ou  not (n = S n) *)
```

Question 3

Démontrer *neq_sym* : $x \neq y \rightarrow y \neq x$

1.4 L'implication

En logique intuitioniste, l'implication est fondamentale puisqu'elle correspond directement au "type flèche" du lambda-calcul et donc des langages de programmation fonctionnelle. Contrairement à *True*, *False* et *not* l'implication est primitive (en fait, on verra, il s'agit d'un cas particulier d'une contruction plus générale).
Nous savons déjà raisonner sur des implications, et en guise d'illustration montrons que l'implication est bien réflexive.

```
Lemma impl_refl: forall P:Prop, P -> P.
Proof.
  intros P HP.
  exact HP.
Qed.
```

La tactique `intro` correspond à la règle logique d'introduction de l'implication.

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ (impl - intro)$$

Il existe bien sûr une règle complémentaire pour éliminer une implication en hypothèse :

$$\frac{\Gamma, P \to Q \vdash Q}{\Gamma \vdash P}\ (impl - elim)$$

En Coq, la tactique `apply` permet ce type de raisonnement, qui correspond en fait à une beta-reduction comme une application de fonction.
De fait, les tactiques `intro` et `apply` sont parmi les rares tactiques primitives de Coq.
Montrons un exemple d'utilisation de `apply`.

```
Lemma impl_lemma:
  forall P Q : Prop, (P -> Q) -> P -> Q.
Proof.
  intros P Q.
  intros Himpl HP.
  apply Himpl.
  exact HP.
Qed.
```

Passons en mode Ocaml pour effectuer une "preuve" similaire mais complètement manuelle. L'idée est de trouver un terme du calcul dont le type est simplement $P \to P$. C'est bien sûr le type de la fonction identité :

```
let id : 'p -> 'p = fun x -> x
```

Il s'agit d'une preuve que quelle que soit la proposition (i.e. le type) $'p$, et bien $'p \to 'p$ est vrai !
Cette preuve directe peut également être écrite en Coq :

```
Lemma impl_refl_direct: forall P : Prop, P → P.
Proof.
  exact (fun (P:Prop) (p:P) ⇒ p).
Qed.
```

La différence par rapport à Ocaml est qu'en Coq il faut préciser que le paramètre de type est une proposition : donc une variable du type `Prop`.
Regardons d'un peu plus près notre preuve indirecte.

```
Print impl_refl.
impl_refl = fun (P : Prop) (HP : P) ⇒ HP : forall P : Prop, P → P
```

Les deux preuves sont en fait représentées par le même terme du lambda-calcul !
Bien sûr, on peut spécialiser notre preuve. On peut par exemple montrer que *True* → *True* est vrai, de la façon suivante (en Ocaml) :

```
# fun (x:vrai) -> id x ;;
- : vrai -> vrai = <fun>
```

En coq cela donne :

```
Example V_impl_V: True → True.
Proof.
  apply impl_refl.
Qed.
```

Ici, on applique simplement la fonction *impl_refl*, et Coq se débrouille pour unifier les arguments d'appel. Encore une fois on peut passer en mode complètement manuel.

```
Example V_impl_V_direct: True → True.
Proof.
  exact (impl_refl True).
Qed.
```

Dans le même genre d'idée:

```
Example F_impl_F: False → False.
Proof.
apply impl_refl.
Qed.
```

Puisque *impl* est juste un alias du type flèche, nous n'utiliserons désormais que ce dernier pour représenter l'implication.
On peut prouver d'autres propriétés basiques sur l'implication.

1.4.1 Exercice

Question 1

Compléter la preuve ci-dessous :

Lemma *impl_trans*: forall $P\ Q\ R$: Prop, $(P \to Q) \to (Q \to R) \to (P \to R)$.
Proof.
  «*COMPLETER ICI*»>

Question 2

Démontrer le même résultat par une preuve manuelle en Ocaml.

Ici le "mode calcul" en Ocaml commence à ressembler à de l'assemblage de spaghettis, alors que le "mode logique" de Coq reste plus facilement compréhensible.
Comparons le résultat obtenu.

Print *impl_trans*.

Encore une fois (si vous avez réussi à répondre à la question), on tombe sur exactement le même terme, mais obtenu de façon beaucoup plus "logique" avec les tactiques de Coq.

1.5 la conjonction

Le *et logique* ou *conjonction* porte bien son puisque prouver un énoncé de la forme $P \wedge Q$ revient à combiner deux preuves : une pour P et l'autre pour Q. On pense naturellement au *paires* des langages de programmation.
En Ocaml, on pourrait définir la fonction suivante :

```
type ('a,'b) conj = Et of ('a * 'b)

let et_intro: 'a -> 'b -> ('a,'b) conj =
  fun p1 p2 -> Et (p1, p2)
```

En Coq la conjonction est introduite de façon un peu différente par le type inductif *and* de la bibliothèque standard :

Print *and*.

Inductive *and* $(A\ B$: Prop) : Prop :=
  conj : $A \to B \to A \wedge B$

Cela correspond en quelque sorte à un *curryfication* de la version OCaml. La raison est que dans la mesure du possible, Coq utilise de façon privilégiée les constructions `Inductive` pour introduire les connecteurs logiques.
La règle usuelle d'introduction des conjonction est la suivante :

$$\frac{\Gamma \vdash P \wedge \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\, conj - intro$$

Autrement dit, à partir d'un but représenté par une conjonction on génère deux sous-buts : l'un pour l'opérande de gauche et l'autre pour l'opérande de droite. En Coq la tactique `split` permet d'effectuer ce raisonnement.

```
Lemma and_split:
  forall P : Prop, P → P ∧ P.
Proof.
  intros P HP.
  split.
  - (*  cas left *)
    exact HP.
  - (*  cas right *)
    exact HP.
Qed.
```

Une petite inconvenance est que `split` ne décompose une conjonction qu'en deux parties.

```
Lemma and_split3:
  forall P : Prop, P → P ∧ P ∧ P.
Proof.
  intros P HP.
  split.
  - (*  cas left *)
    exact HP.
  - (*  cas right *)
    split. (*  on doit resplitter *)
    + (*  cas left *)
      exact HP.
    + (*  cas right *)
      exact HP.
Qed.
```

Heureusement en pratique on peut souvent un peu "accélerer".

```
Lemma and_split3':
  forall P : Prop, P → P ∧ P ∧ P.
Proof.
  intros P HP.
  repeat split ; exact HP.
Qed.
```

On peut aussi créer des tactiques personnalisées (par exemple *split3*) avec le langage de définition de tactiques *LTac* mais cela dépasse le cadre de ce cours.
La règle d'élimination des conjonctions est assez triviale :

$$\frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R}\; conj - elim$$

En Coq on utilise la tactique `elim` pour réaliser ce type de raisonnement.

```
Lemma et_elim_gauche:
  forall P Q:Prop, P ∧ Q → P.
Proof.
  intros P Q H.
  elim H.
  intros HP HQ.
  exact HP.
Qed.
```

Plutôt que de passer par une élimination explicite de la conjonction avec la tactique `elim`, on peut exploiter les introductions structurées.

```
Lemma et_elim_gauche':
  forall P Q:Prop, P ∧ Q → P.
Proof.
  intros P Q [HP HQ].
  exact HP.
Qed.
```

Il faut tout de même faire attention pour introduire de façon structurée des conjonctions au delà de deux opérandes.

```
Lemma et_elim4_gauche:
  forall P Q R S:Prop, P ∧ Q ∧ R ∧ S → P.
Proof.
  intros P Q R S [HP [HQ [HR HS]]].
  exact HP.
Qed.
```

C'est peut-être plus clair avec le parenthèsage explicite.

```
Lemma et_elim4_gauche':
  forall P Q R S:Prop, P ∧ ( Q ∧ ( R ∧ S ) ) → P.
Proof.
  intros P Q R S [HP [HQ [HR HS]]].
  exact HP.
Qed.
```

Si on a déjà une hypothèse sous la forme d'une conjonction, alors plutôt que d'utiliser `elim` on peut faire un `destruct` structuré (sic !).

```
Lemma et_elim_gauche'':
  forall P Q:Prop, P ∧ Q → P.
Proof.
```

```
  intros P Q HPQ.
  destruct HPQ as [HP HQ].
  exact HP.
Qed.
```

On peut reproduire cette preuve de façon manuelle en Ocaml. Pour cela, il faut une fonction prenant une conjonction en argument et retournant la preuve correspondant à l'opérande gauche du et logique. C'est bien sûr la fonction *fst* qui retourne le premier élément d'une paire :

```
let et_elim_gauche : ('p,'q) conj -> 'p = fun (Et x) -> fst x
```

1.5.1 Exercice

Question 1

Définir l'élimination à droite de la conjonction en Ocaml puis en Coq, en utilisant `elim` explicitement, puis avec un `intro` structuré et enfin avec un `destruct` structuré.

Question 2

Démontrer avec un `intro` structuré :
\\`forall` $P\ Q\ R\ S\ T$: `Prop`, $P \wedge Q \wedge R \wedge S \wedge T \rightarrow Q \wedge S$.

1.6 l'équivalence logique

Avec l'implication et la conjonction, nous couvrons naturellement l'équivalence.

`Print` *iff*.
iff = `fun` $A\ B$: `Prop` $\Rightarrow (A \rightarrow B) \wedge (B \rightarrow A)$: `Prop` $\rightarrow$ `Prop` $\rightarrow$ `Prop`

1.6.1 Exercice

Question 1

Définir en ocaml la fonction correspondante *ssi*.

Question 2

Démontrer les lemmes:

- *iff_refl*: `forall` P:`Prop`, $P \leftrightarrow P$ ainsi que la fonction Ocaml correspondante *ssi_refl*.
- *iff_sym*: `forall` $P\ Q$:`Prop`, $(P \leftrightarrow Q) \rightarrow (Q \leftrightarrow P)$ ainsi que la fonction Ocaml *ssi_sym*.
- mêmes questions pour la transitivité de l'équivalence.

1.6.2 Exercice

Question 1

Montrer les propriétés suivantes (si possible en utilisant des `intros` structurés) :

- $(P \wedge Q) \wedge R \leftrightarrow P \wedge (Q \wedge R)$
- $(P \wedge Q) \leftrightarrow (Q \wedge P)$
- $(P \wedge Q) \wedge (Q \wedge R) \rightarrow P \wedge R$

1.7 la disjonction

Pour la disjonction, c'est un peu plus complexe. Pour avoir une preuve de P *ou* Q il nous faut soit une preuve de P soit une preuve de Q. Pour ne pas interférer avec la conjonction, on peut ajouter le fait que les deux solutions ne sont pas liées. Cela conduit naturellement à considérer un type somme binaire.

```
Print or.
```

```
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B
  | or_intror : B -> A \/ B
```

En Ocaml on écrirait :

```
type  ('a, 'b) disj =
  | Ou_gauche of 'a
  | Ou_droite of 'b
```

Chaque constructeur de ce type correspond à une règle d'introduction :

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ (or - intro - l)$$

Ce raisonnement correspond à la tactique `left` de Coq.

```
Lemma ou_intro_l:
  forall P Q : Prop, P → P ∨ Q.
Proof.
  intros P Q HP.
  left.
  exact HP.
Qed.
```

En Ocaml, le raisonnement équivalent est le suivant :

```
# let ou_intro_l : 'p -> ('p, 'q) disj = fun (hp:'p) -> Ou_gauche hp ;;
val ou_intro_l : 'p -> ('p, 'q) disj = <fun>
```

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ (or - intro - r)$$

Ce raisonnement correspond à la tactique `right` de Coq.

```
Lemma ou_intro_r:
  forall P Q : Prop, Q -> P \/ Q.
Proof.
  intros P Q HQ.
  right.
  exact HQ.
Qed.
```

En Ocaml, le raisonnement équivalent est le suivant :

```
# let ou_intro_r : 'q -> ('p, 'q) disj = fun (hq:'q) -> Ou_droite hq ;;
val ou_intro_r : 'q -> ('p, 'q) disj = <fun>
```

1.7.1 Exercice

Montrer *ou_intro_3*: `forall` $P\ Q\ R\ S$: `Prop`, $Q \to P \vee R \vee Q \vee S$.
Lors que la disjonction est en hypohèse on utilise une règle d'élimination qui génère deux sous-buts :

$$\frac{\Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma, P \vee Q\ vdashR}\ (or - elim)$$

En Coq on met cette règle en pratique avec `elim`.

```
Lemma or_idem:
  forall P : Prop, P \/ P -> P.
Proof.
  intros P Hor.
  elim Hor.
  - (*  cas left *)
    intro HP.
    exact HP.
  - (*  cas right *)
    intro HP.
    exact HP.
Qed.
```

On peut utiliser un `intros` structuré en séparant les sous-buts par une barre verticale.

```
Lemma or_idem':
  forall P : Prop, P \/ P -> P.
Proof.
  intros P [ HP1 | HP2 ].
  - (*  cas left *)
    exact HP1.
  - (*  cas right *)
    exact HP2.
Qed.
```

Et encore une fois si on a déjà l'hypothèse sous la forme d'une disjonction, on peut utiliser un `destruct` structuré.

```
Lemma or_idem'':
  forall P : Prop, P \/ P -> P.
Proof.
  intros P Hor.
  destruct Hor as [HP1 | HP2].
  - (*  cas left *)
    exact HP1.
  - (*  cas right *)
    exact HP2.
Qed.
```

Et encore une fois, il faut être un peu vigilant pour les combinaisons de disjonctions.

```
Lemma or_idem3:
  forall P : Prop, P \/ P \/ P -> P.
Proof.
  intros P [ HP1 | [ HP2 | HP3 ] ].
  - (*  cas left/left *)
    exact HP1.
  - (*  cas right/left *)
    exact HP2.
  - (*  cas right/right *)
    exact HP3.
Qed.
```

1.7.2 Exercice

Question 1

Montrer la commutativité *or_sym* de la disjonction.

1.8 Quantificateur universel

Maintenant que nous avons bien étudié le cas propositionnel, intéressons nous à l'étape suivante : la logique du premier ordre.
L'idée est d'introduire des types de données simples, nous nous limiterons aux entiers naturels, ainsi que des quantificateurs universels et existentiels. Nous allons commencer par l'universel.
Disons le tout de suite, si le système de type de Ocaml peut servir de vérifieur de théorème pour la logique propositionnelle (intuitioniste), cela n'est plus vrai lorsque l'on passe au premier ordre. Pour cela il faut un système de type pouvant manipuler des données et effectuer des calculs sur ces dernières, ce sont les fameux types dépendants.
Nous vous avons en quelque sorte "menti" en vous disant que l'implication était primitive dans le système de type de Coq. En effet, le type flèche qui est bien primitif en Ocaml (c'est le type des fonctions donc plutôt primitif dans un langage fonctionnel) n'est en fait qu'un raccourcis d'écriture en Coq.
La construction de type primitive est de la forme :
`forall` A : T, B où A, T et B sont des types.
L'interprétation est bien la quantification universelle :
Pour tout terme A de type T alors B est bien un type.
Si dans B on ne trouve pas d'occurrence de A, alors on a bien ce que l'on nomme le type flèche.
Vérifions ce fait en Coq :

```
Example forall_impl:
  forall A B : Prop,
    (forall pa : A, B) = (A -> B).
Proof.
  intros A B.
  reflexivity.
Qed.
```

La tactique `intro` permet on l'a vu d'introduit une quantification universelle.

$$\frac{\Gamma, a : T \vdash P}{\Gamma \vdash \forall a : T, P}\ (forall - intro)$$

Par exemple :

```
Lemma nat_refl:
  forall n : nat, n = n.
Proof.
  intro n.
  reflexivity.
Qed.
```

Et l'élimination se fait le plus souvent par `apply` ... `with` ... :

$$\frac{\Gamma, \forall a : T, P \vdash Q}{\Gamma \vdash P} \ (forall - elim)$$

Par exemple :

```
Lemma lt_lt_silly:
  (forall n:nat, n > 0)
  → forall m:nat, m > 0.
Proof.
  intros Hforall m.
  apply Hforall with (n:=m).
Qed.
```

Remarque : dans ce dernier exemple, on explicite l'instanciation de la variable quantifiée n par la variable m du contexte. On aurait pu écrire directement une application :

```
Lemma lt_lt_silly':
  (forall n:nat, n > 0)
  → forall m:nat, m > 0.
Proof.
  intros Hforall m.
  apply (Hforall m).
Qed.
```

Ceci illustre bien la nature fonctionnelle de l'hypothèse *Hforall*. Et finalement, dans beaucoup de cas Coq se débrouille pour inférer les arguments de l'application.

```
Lemma lt_lt_silly'':
  (forall n:nat, n > 0)
  → forall m:nat, m > 0.
Proof.
  intros Hforall m.
  apply Hforall.
Qed.
```

On utilise la variante `apply` ... `with` ... lorsque Coq ne peut synthétiser les bons arguments d'appels (car le problème se résume à de l'unification d'ordre supérieur qui est indécidable dans le cas général). La variante avec application fonctionnelle est plus rarement utilisée en pratique.

Nous utilisons très souvent cette synthèse automatique des arguments d'appel lorsque nous appliquons des lemmes auxiliaires, qui sont le plus souvent préfixés par des quantifications universelles.

1.8.1 Exercice

Démontrer le lemme suivant :

```
Lemma forall_impl_point:
  forall E : Type, forall e : E, forall P Q : E → Prop,
   (forall x : E, P x → Q x) → P e → Q e.
```

1.9 la quantification existentielle

Si la quantification universelle est primitive dans le lambda-calcul typé de Coq, ce n'est pas le cas de la quantification existentielle.
Encore une fois, l'existentiel n'est pas la négation de l'universel en logique constructive, contrairement à la logique classique.
Coq défini l'existentielle de la façon suivante :

```
Print ex.


Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : forall x : A, P x → exists x, P x
```

Lorsque l'on a une existentielle en tant que but, on utilise la tactique `exists` pour l'introduction. Il faut trouver dans le contexte un témoin d'existence.

```
Lemma ex_forall: forall E : Set, forall y : E, forall P Q : E → Prop,
  (forall x : E, P x → Q x) → P y → exists z : E, Q z.
Proof.
  intros E y P Q H1 H2.
  exists y.
  apply H1.
  exact H2.
Qed.
```

Dans la preuve ci-dessus, on a utilise le témoin y pour notre preuve d'existence.

1.9.1 Exercice

Montrer :

```
Lemma pred_ex:
  forall n : nat,
    n ≠ 0
    → exists p : nat, S p = n.
```

Lorsque l'existentielle est en hypothèse, on utilise la tactique `elim`.

```
Lemma lt_neq_0:
  forall m : nat,
  (exists n : nat, n < m) → 0 ≠ m.
Proof.
  intros m Hex.
```

```
  elim Hex. clear Hex.
  intros x Hlt.
  destruct m as [|m'].
  inversion Hlt. (*  contradition *)
  SearchPattern (0 ≠ S _ ).
  (*  O_S: forall n : nat, 0 <> S n *)
  apply O_S.
Qed.
```

1.9.2 Exercice

Montrer :

```
Lemma encadrement:
  forall n p : nat,
    (exists m : nat, (n ≤ m) ∧ (m < p)) → n < p.
```

1.10 l'égalité

L'égalité est une notion fondamentale tant en logique qu'en calcul. Il existe en fait plusieurs définitions possibles pour l'égalité mais Coq propose une définition inductive assez simple.

```
Print eq.


Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

Pour montrer une égalité, il suffit d'appliquer le constructeur *eq_refl*, ce que l'on effectue en général avec la tactique `reflexivity` (synonyme de `apply` *eq_refl*).
Par exemple :

```
Lemma eq_refl': forall a:Type, a = a.
Proof.
  intro a.
  apply eq_refl. (*  ou reflexivity *)
Qed.
```

Remarque : on utilise `Type` car l'égalité s'applique tant aux propositions (dans `Prop`) qu'aux données (dans `Set`).
Lorsque une égalité se trouve en hypothèse, on peut par exemple l'exploiter par réécriture avec `rewrite`.

```
Lemma eq_sym': forall a b : Type, a = b → b = a.
Proof.
  intros a b Heq.
  rewrite Heq.
  reflexivity. (*  ou apply eq_refl ou apply eq_refl' *)
```

```
Qed.
```

On peut aussi réécrire de droite à gauche.

```
Lemma eq_sym'': forall a b : Type, a = b → b = a.
Proof.
  intros a b Heq.
  rewrite ← Heq.
  reflexivity.
Qed.
```

1.10.1 Exercice

Démontrer la transititivé de l'égalité : *eq_trans'*.

On se rappelle que l'axiome du tiers-exclu n'est pas primitif en Coq, et qu'on essaye en général de s'en passer pour conserver le caractère constructif des preuves. Cela ne veut pas dire que certaines instances ce sont pas disponible.
Avec l'égalité on a souvent besoin d'une séparation entre le cas d'égalité et le cas d'inégalité. Pour cela, il est nécessaire de disposer d'un axiome ou d'un théorème prouvant la décidabilité de la relation d'égalité pour le type auquel on s'intéresse. En effet, la décidabilité ne peut être imposée puisque dans le lambda-calcul l'égalité sur les fonctions n'est bien sûr pas décidable. Pour les entiers, on se repose sur le lemme *eq_nat_dec* dont le type implique la décidabilité de l'égalité sur les entiers naturels.

```
Check eq_nat_dec.
```

```
eq_nat_dec
     : forall n m : nat, {n = m} + {n ≠ m}
```

Au-delà des notations un peu cryptique, voici comme utiliser ce lemme.

```
Lemma nat_split:
  forall n m : nat, (n = m) ∨ (n ≠ m).
Proof.
  intros n m.
  destruct (eq_nat_dec n m) as [Heq | Hneq].
  - (*  cas n=m *)
    left.
    exact Heq.
  - (*  cas n<>m *)
    right.
    exact Hneq.
Qed.
```

On voit que *nat_split* est une sorte d'instance du tiers-exclu mais limité à l'égalité sur les entiers naturels.
Les fonctions qui nécessitent une notion d'égalité, comme le test d'appartenance à une liste, nécessitent également une propriété de décidabilité pour l'égalité.
Voici un exemple :

```
Require Import List.

Section member.

Variable A : Set.

Hypothesis eq_dec: forall a b : A, {a = b} + {a ≠ b}.

Fixpoint member (e:A) (l: list A) : bool :=
  match l with
    | nil ⇒ false
    | e'::l' ⇒ match eq_dec e e' with
                  | left _ ⇒ true
                  | right _ ⇒ member e l'
                end
  end.

End member.

Example member_ex1:
  member nat eq_nat_dec 3 (1::2::3::4::5::nil) = true.
Proof.
  compute. reflexivity.
Qed.
```

Le premier argument de *member* devrait pour être inféré. On indique ce fait par la commande suivante :

```
Arguments member [A] _ _ _.

Example member_ex2:
  member eq_nat_dec 3 (1::2::3::4::5::nil) = true.
Proof.
  compute. reflexivity.
Qed.
```

En revanche on doit laisser explicite l'hypothèse de décidabilité de l'égalité.

1.10.2 Exercice

Compléter le lemme suivant :

```
Section member_cons.
```

```
Variable A : Set.

Hypothesis eq_dec : forall a b : A, {a = b} + {a ≠ b}.

Lemma member_cons:
  forall e : A, forall l : list A,
    member eq_dec e (e::l) = true.
Proof.
  «COMPLETER ICI»>
End member_cons. (*  fermeture de section *)
```