LEÇON 2 : Logique Propositionnelle et Constructivisme

Connecteurs ET, OU et Négation

Pablo Donato

24/03/2026

Après avoir exploré l'implication ($\rightarrow$), notre voyage au cœur de la correspondance de Curry-Howard se poursuit. Aujourd'hui, nous allons découvrir comment Rocq traite la Conjonction (ET), la Disjonction (OU), et l'Absurdité (NON).

Nous adopterons une perspective profondément constructiviste (la sémantique Brouwer-Heyting-Kolmogorov, ou BHK), où chaque connecteur logique n'est pas une simple table de vérité, mais une véritable *structure de données* informatique construite par nos soins !

```
Notation "( x , y )" := (conj x y).
Notation "'ι₁'" := or_introl (at level 0).
Notation "'ι₂'" := or_intror (at level 0).

Section Lesson2.

Variables A B C : Prop.
```

1 La Conjonction (Le ET : $\wedge$)

Posséder une preuve formelle de $A \wedge B$, c'est posséder indépendamment une preuve de A ET une preuve de B. Informatiquement, cela correspond exactement à la notion de *paire* (ou "tuple").

La règle d'introduction de la conjonction en déduction naturelle est la suivante :

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \wedge B} \wedge I$$

Pour construire/introduire une telle paire, nous utilisons la tactique `split`.

```
Lemma ET_intro : A -> B -> A /\ B.
Proof.
  intros x y.
```

Notre but est $A \wedge B$. Nous avons besoin d'en fournir les deux composantes.

```
  split.
```

`split` a divisé notre but en deux. Remarquez l'usage de la puce sous forme de tiret '-' qui permet de se concentrer successivement sur le premier puis second buts générés.

```
  - exact x.
  - exact y.
Qed.
```

Remarque 1

Observons le terme final. Le compilateur nous montre que le terme de preuve est bel et bien structuré comme une paire !

```
Print ET_intro.
```

À l'inverse, si nous *avons* une hypothèse de type $A \wedge B$, nous pouvons la "détruire" pour en extraire les deux composants sous-jacents, grâce à la commande `destruct`. Cela correspond aux "projections" π_1 et π_2.

```
Lemma ET_symetrie : A /\ B -> B /\ A.
Proof.
  intros H_ab.
```

L'hypothèse `H_ab` cache une paire. `destruct` va l'ouvrir et nommer ses deux composants `x` (la preuve de A) et `y` (la preuve de B).

```
  destruct H_ab as [x y].
  Show Proof.
  split.
  - exact y.
  - exact x.
Qed.
```

2 La Disjonction (Le OU : $\vee$)

Voici le grand point de rupture du constructivisme ! Pour prouver $A \vee B$, vous ne pouvez pas joyeusement affirmer "bon, l'un des deux doit être vrai", sans savoir lequel. Vous DEVEZ exhiber consciemment une preuve ou bien de A, ou bien de B.

Informatiquement, c'est ce qu'on appelle un type "somme". L'ordinateur conserve la trace indélébile du choix effectué via des fonctions d'injection : `left` correspond à ι_1 et `right` correspond à ι_2.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \iota_1 M : A \vee B} \vee I_1 \qquad \frac{\Gamma \vdash N : B}{\Gamma \vdash \iota_2 N : A \vee B} \vee I_2$$

```
Lemma OU_intro_gauche : A -> A \/ B.
Proof.
  intros x.
```

Nous faisons le choix fort de prouver le côté gauche en injectant `x`.

```
  left.
  exact x.
Qed.
```

On constate que le terme de preuve final est l'application de l'injection gauche ι_1 à la variable `x` de type A.

On note aussi que le type B est passé en argument juste avant `x` : c'est parce que Rocq a besoin de stocker le type de la branche inutilisée (ici à droite) pour connaître le type complet $A \vee B$ de l'injection, et cette information ne peut être inférée uniquement à partir du type de `x`.

```
Print OU_intro_gauche.
```

À l'inverse, l'injection droite avec la tactique `right` :

```
Lemma OU_intro_droite : B -> A \/ B.
Proof.
  intros y.
  right.
  exact y.
Qed.
```

De même ici, le terme de preuve est l'application de l'injection droite ι_2 à la variable `y`.

```
Print OU_intro_droite.
```

Remarque 2: Distinction Fondamentale : Données vs Contrôle

En informatique, on sépare généralement deux grands piliers :

1. **Structures de Données** : comment la machine représente et stocke l'information (ex : les paires pour le ET, ou les injections pour le OU).
2. **Structures de Contrôle** : comment la machine prend des décisions et oriente le flux du calcul en se basant sur ces données.

Posséder une disjonction $A \vee B$ en hypothèse garantit qu'un choix a été fait (injection gauche ou droite), mais cela forme une structure de donnée "fermée", car on ne sait pas a priori quel choix a été fait. Pour l'exploiter activement dans notre preuve/programme, nous avons besoin d'une structure de contrôle pour "ouvrir" cette boîte et faire bifurquer notre raisonnement.
La structure de contrôle la plus célèbre est le "if... then... else...". Sa généralisation s'appelle le "Filtrage par Motif" (ou `match ... with`). C'est une sorte d'aiguillage, qui correspond en fait au raisonnement par cas : "SI la boîte contient une preuve de A, ALORS applique ce premier raisonnement ; SI elle contient une preuve de B, ALORS applique cet autre raisonnement".
C'est exactement ce que génère la tactique `destruct` pour nous !

```
Lemma OU_symetrie : A \/ B -> B \/ A.
Proof.
  intros H_ou.
```

Appliquer un `destruct` avec la barre séparatrice '|' génère deux univers parallèles à gérer individuellement : l'univers où une valeur de type A avait été injectée, et l'univers où une valeur de type B avait été injectée.

```
  destruct H_ou as [x | y].
```

Comme avec la tactique `split`, Rocq nous génère deux "sous-buts", un pour chaque univers.

```
  - (* Cas 1 : La preuve d'origine etait une injection gauche, pour A *)
    right. exact x.
  - (* Cas 2 : La preuve d'origine etait une injection droite, pour B *)
    left. exact y.
Qed.

Print OU_symetrie.
```

3 L'Absurdité et la Négation

Dans un monde constructiviste, la définition de la négation est d'une élégance presque poétique : $\neg A$ est exactement une *abréviation* pour $A \to \bot$ (où $\bot$ dénote l'absurdité, `False` dans

Rocq). Prouver que A est faux, c'est tout bêtement écrire une fonction qui, si elle recevait une preuve de A, réussirait à cracher l'absurdité `False`.

Et qu'est-ce que `False` ? C'est le type ultimement vide : de par sa conception, il n'a aucun constructeur, donc aucun habitant ! (sinon notre système logique deviendrait trivial à cause du principe d'explosion)

```
Lemma non_contradiction : ~(A /\ ~A).
Proof.
```

Gardez en tête que le but est en fait de prouver `(A /\ (A -> False)) -> False`.

```
  intros H_impossible.
  destruct H_impossible as [a non_a].
```

Nous avons en main `a : A` et une étrange fonction `non_a : A -> False`. Il suffit de faire s'entrechoquer les deux... Modus Ponens !

```
  apply non_a.
  exact a.
Qed.
```

Le Principe d'Explosion (*Ex Falso Quodlibet*)

Si vous obtenez une preuve de `False` dans votre contexte, c'est donc que vous avez au moins deux hypothèses contradictoires A et $\neg A$. Et d'une contradiction, vous pouvez déduire N'IMPORTE QUOI. En Rocq, c'est la tactique `exfalso`.

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \text{abort } M : A} \bot E$$

```
Lemma explosion : False -> A.
Proof.
  intro f.
  exfalso.
  exact f.
Qed.
```

4 La Crise du Constructivisme et le Tiers-Exclu

L'idéalisme intuitionniste a un revers : le Tiers-Exclu $A \vee \neg A$ n'est pas prouvable de manière générique. C'est logique : réussir à le prouver reviendrait à posséder l'Algorithme Universel capable de trancher la vérité de n'importe quelle proposition A sans calcul préalable. Et nous savons tous que ce rêve du mathématicien David Hilbert a été ruiné par le Théorème d'Incomplétude de Gödel et le problème de l'Arrêt de Turing.

```
Lemma tiers_exclu_fail : A \/ ~A.
Proof.
```

On peut esquisser une tentative : si on essaye `left`, l'ordinateur attendra de nous une preuve de A...

```
  left.
```

... que l'on n'a pas. Annulons avec la commande `Undo`.

```
  Undo.
```

Si on essaye `right`, l'ordinateur attendra qu'on réfute A.

```
  right.
  intro H.
```

Enfermés et toujours ignorants du problème précis sous-jacent à la proposition abstraite A, nous ne pouvons que renoncer lâchement avec la commande `Abort`.

```
Abort.
```

Conséquence : si l'on veut retrouver notre zone de confort mathématique (la logique classique et platonicienne de Descartes/Aristote), on doit forcer le système en imposant violemment le principe de Tiers-Exclu comme une vérité infuse via la commande `Axiom`.

```
Axiom LEM : forall X : Prop, X \/ ~X.
```

Muni de cette arme redoutable, un nombre vertigineux de théorèmes inaccessibles sur un plan constructiviste (car incomputables) s'offrent de nouveau à nous. La porte étendard de la pensée classique est sans conteste l'Élimination de la Double Négation : (Non-(Non A) alors A).

```
Lemma double_negation : ~~A -> A.
Proof.
  intros non_non_a.
```

Pour arracher A au néant, on invoque le Tiers-Exclu `LEM` en l'instanciant avec A. Ensuite, `destruct` va de nouveau séparer les univers.

```
  destruct (LEM A) as [x | non_a].
  - (* L'univers 1 correspond au cas ou (par magie cosmique), A etait
     vrai.
       On a A a disposition, la victoire est immédiate. *)
    exact x.
  - (* L'univers 2 correspond au cas ou (par magie cosmique), ~A etait
     vrai.
       Mais attendez! Notre hypothese constructiviste de depart [
          non_non_a] nous
       jure la main sur le coeur que [~A -> False] ! La contradiction
          est totale. *)
    exfalso.
    apply non_non_a.
    exact non_a.
Qed.
```

5 Exercices à faire

Exercice 1: Logique de De Morgan

Prouver formellement un côté des fameuses Lois de De Morgan. Ce sens-ci est tout à fait harmonieux avec la pensée constructiviste (NUL BESOIN d'utiliser le Tiers-Exclu `LEM`, que des règles habituelles !).

```
Lemma de_morgan_1 : (A \/ B) -> ~(~A /\ ~B).
Proof.
  (* ECRIVEZ VOTRE PREUVE ICI *)
Admitted.
```

Exercice 2: Le Sens Classique

Prouvez enfin sa monstrueuse réciproque ! Attention : dans ce sens, le constructivisme est paralysé. Vous n'aurez d'autre choix que de brandir le Tiers Exclu en effectuant des raisonnements par cas sur $A \vee \neg A$ ou $B \vee \neg B$ pour espérer craquer la coquille et conclure.

```
Lemma de_morgan_2 : ~(~A /\ ~B) -> (A \/ B).
Proof.
  (* ECRIVEZ VOTRE PREUVE ICI *)
Admitted.
```

```
End Lesson2.
```