LEÇON 1 : Du Lambda-Calcul à Rocq

La Correspondance de Curry-Howard en Action

Pablo Donato

17/03/2026

Bienvenue dans Rocq ! Aujourd'hui, nous allons voir la correspondance de Curry-Howard (Propositions comme Types, Preuves comme Programmes) prendre vie.

Tout ce qui se trouve entre "(*" et "*)" affiché en orange/marron, incluant le texte que vous êtes en train de lire, est ce que l'on appelle couramment en programmation un "commentaire". C'est une partie du code destinée uniquement à la lecture par les êtres humains, qui ne sera donc pas exécutée/interprétée par la machine. Dans les commentaires, nous écrirons entre crochets "⌊" et "⌋" les bouts de texte correspondant à du véritable code Rocq (mais qui restent ignorés par la machine puisqu'en commentaire).

Pour exécuter pas à pas le code situé en-dehors des commentaires, utilisez le bouton Flèche Bas dans la barre d'outils en haut à droite de jsCoq, ou appuyez sur les touches Alt+Bas (Windows/Linux) ou Option+Bas (Mac). Le code déjà exécuté est surligné en gris, ou en rose s'il a déclenché une erreur.

Dernière remarque : il est fortement conseillé de désactiver l'affichage Unicode. Pour cela, accédez au menu via le bouton ":" tout en haut à droite de la page, et décochez l'option "Enable company-coq". Bonne lecture !

1 Langage vernaculaire

Sections

Ci-dessous se trouve notre première commande : la commande `Section`.

Tout comme le titre d'une section dans un article scientifique, elle marque le début d'une partie du document qui exhibe une certaine cohérence thématique, ici capturée par le titre `Lesson1`. Mais contrairement à une section "informelle", la machine a besoin que l'on marque exactement l'endroit où se termine la section à l'aide de la commande `End`.

Un lecteur astucieux pourrait alors croire que le présent usage de `Section` est trivial, car la commande `End Lesson1` se trouve sur la toute dernière ligne, couvrant donc l'entièreté du document.

En réalité, cela s'explique par le fait qu'au-delà de sa fonction thématique/sémantique pour l'humain, la commande `Section` possède aussi une fonction *syntaxique* pour la machine : elle permet en effet de délimiter la *portée* (*scope* en anglais) des variables déclarées avec la commande `Variables`, expliquée ci-après.

```
Section Lesson1.
```

Remarque 1

Toute utilisation d'une commande Rocq doit se terminer par un point '.'.

Remarque 2

On ne peut pas utiliser n'importe quel nom comme titre d'une `Section` : seules certaines chaînes/séquences de caractères sont autorisées, qu'on appelle des **identifiants**.

Identifiants

Le concept d'identifiant est pervasif dans les langages de programmation. Il émerge principalement du besoin d'avoir des algorithmes efficaces pour interpréter le code source d'un programme de manière non-ambigüe. Cela est particulièrement vrai pour la formalisation logique, où l'ambiguïté peut être fatale à la vérification : par exemple, comment faire confiance à une démonstration d'un théorème dont l'énoncé ne serait même pas exactement déterminé ?

Dans Rocq, les identifiants sont des séquences sans espaces et composées uniquement des caractères suivants :

— Lettres (et plus généralement tout symbole Unicode comme les lettres grecques et de nombreux symboles mathématiques)
— Chiffres (0-9), mais l'identifiant ne peut pas commencer par un chiffre
— Le tiret du bas '_'
— L'apostrophe (') mais l'identifiant ne peut pas commencer par une apostrophe

De plus, les identifiants sont sensibles à la casse, et le caractère '_' seul est réservé et n'est pas un identifiant valide.

Déclarations de variables et propositions

Dans un texte mathématique informel, il est courant de déclarer des variables dénotant des objets arbitraires dans un certain domaine de discours. Par exemple en logique, on peut écrire :

"Soient A, B et C des propositions."

Dans ce cas les variables ont pour noms les lettres 'A', 'B' et 'C', et le domaine de discours est un ensemble de formules propositionnelles à préciser, disons celui généré par les connecteurs $\{\neg, \wedge, \vee, \rightarrow\}$ à partir de symboles de propositions atomiques. Une fois la déclaration faite, on peut dénoter de nouveaux objets construits à partir de ceux dénotés par les variables en inscrivant leurs noms dans des expressions plus complexes, par exemple la conjonction $A \wedge B$ des propositions A et B.

Dans Rocq, déclarer des variables se fait via la commande `Variables`, suivie de la liste des noms des variables séparés par des espaces, suivie de deux points `:`, suivis du *type* des variables. Rocq étant basé sur la théorie des types, le domaine de discours va en effet être formalisé par la notion de type.

Rocq dispose d'un type primitif pour les propositions, noté `Prop`. Nous verrons au fil du cours les différents connecteurs logiques supportés par Rocq pour construire des habitants/éléments du type `Prop`. Toutefois nous n'essaierons pas de caractériser exactement l'ensemble des habitants de `Prop`, comme on le ferait habituellement avec une grammaire ou une définition inductive de l'ensemble des formules.

On peut donc formaliser la déclaration de variables précédente comme suit :

```
Variables A B C : Prop.
```

Remarque 3

Les variables déclarées dans une section ne peuvent être utilisées qu'à l'intérieur de cette section. Aussi à l'instar des noms de sections, les noms de variables doivent être des identifiants valides.

2 Mode Terme (langage Gallina)

Vous savez déjà que par la correspondance de Curry-Howard, prouver l'implication $A \to A$ en déduction naturelle revient à écrire la fonction identité ($\lambda x.x$) en λ-calcul.

Écrivons ce λ-terme exact dans Rocq.

— La commande `Definition` indique à Rocq que nous définissons un terme.
— `I_term` est le nom que nous lui donnons, qui doit être un identifiant valide.
— `A -> A` est le type (la proposition que nous prouvons).
— `fun x => x` est le λ-terme (la preuve elle-même).

La notation exacte de Rocq pour les λ-termes et les types peut être consultée dans la Cheat-sheet.

```
Definition I_term : A -> A := fun x => x.
```

Lorsqu'une `Definition` est exécutée sans erreur, cela signifie que le terme à droite de `:=` a bien le type indiqué à droite de `:`. Sous le capot, Rocq va en effet appliquer un algorithme de "type-checking" qui synthétise automatiquement une dérivation de typage, ici pour le jugement $\vdash \lambda x.x : A \to A$.

La définition ci-dessus se base sur un typage "à la Curry", où l'on assigne un type à un λ-terme pur. En réalité, l'objet qui est construit et stocké en mémoire par Rocq est un terme typé "à la Church", avec des annotations de type additionnelles sur chaque variable liée par une λ-abstraction. Cela peut s'observer avec la commande `Print`, qui affiche le terme et le type associés à une définition donnée :

```
Print I_term.
```

Regardez le "panneau de sortie" en bas à droite ! Vous devriez pouvoir y lire :

```
I_term = fun x : A => x
       : A -> A
```

Vous constaterez l'annotation additionnelle dans `fun x : A => x` comparé à `fun x => x`.

Maintenant, définissons le combinateur K : $A \to (B \to A)$. Cela correspond à une fonction prenant deux arguments et retournant le premier.

— En λ-calcul : $\lambda x.\lambda y.x$
— Dans Rocq : `fun x => fun y => x` (ou de façon plus concise : `fun x y => x`)

```
Definition K_term : A -> (B -> A) := fun x y => x.
```

3 Mode Preuve Interactive (langage de tactiques)

Écrire manuellement des λ-termes est pratique et instructif sur de petits exemples, mais devient vite très fastidieux lorsqu'on formalise des démonstrations complexes, en plus de produire un artefact difficilement lisible pour un humain.

Pour pallier ce problème, Rocq dispose d'un langage de **tactiques** permettant de synthétiser automatiquement des "bouts" de λ-termes. L'idée est de travailler étape par étape "à l'envers" en partant du type à habiter (de la proposition à prouver).

Construisons à nouveau la fonction identité, cette fois-ci à l'aide de tactiques. On utilise la commande `Lemma` pour indiquer à Rocq que l'on souhaite définir un nouveau terme `I_tac` de type `A -> A` (prouver un lemme `I_tac` dont l'énoncé est `A -> A`) ; puis la commande `Proof` pour rentrer dans le mode de Preuve Interactive où nous pourrons utiliser les tactiques.

```
Lemma I_tac : A -> A.
Proof.
```

Regardez le panneau de droite. Au-dessus de la ligne se trouve votre contexte d'hypothèses (ce que vous savez). En dessous de la ligne se trouve votre but (ce que vous devez prouver). Cela correspond grosso modo à un séquent, dont la partie gauche est le contexte et la partie droite le but.

La tactique `intros` implémente la règle de typage de l'abstraction en λ-calcul (la règle d'introduction de l'implication en déduction naturelle), lue de bas en haut :

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\,\text{Abs}$$

Elle prend l'antécédant `A` de l'implication et le déplace au-dessus de la ligne/à gauche du séquent, lui donnant le nom `x`.

```
  intros x.
```

Jetons un coup d'œil sous le capot ! `Show Proof` révèle dans le panneau de sortie le λ-terme que Rocq construit pour nous. Remarquez le `?Goal`. Rocq sait que le terme final aura la forme $\lambda x.M$, mais a encore besoin de nous pour déterminer le corps `M` de la fonction.

```
  Show Proof.
```

Pour résoudre le but, nous utilisons la tactique `exact`, qui indique à Rocq : "Le terme dont j'ai besoin est exactement cette hypothèse/variable dans mon contexte."

Elle implémente la règle de typage des variables en λ-calcul (la règle "axiome" en déduction naturelle) :

$$\frac{}{\Gamma, x : A \vdash x : A}\,\text{Var}$$

```
  exact x.
```

Il n'y a maintenant plus de buts à résoudre, puisque la règle n'a pas de prémisses ! On peut donc sortir du mode Preuve Interactive avec la commande `Qed`.

```
Qed.
```

Affichons notre terme de preuve ainsi construit. Remarquez qu'il est parfaitement identique à `I_term` !

```
Print I_tac.
```

On peut aussi facilement construire le combinateur K avec des tactiques :

```
Lemma K_tac : A -> B -> A.
Proof.
```

Nous avons deux prémisses à introduire. Nous pouvons les faire toutes les deux à la fois.

```
  intros x y.
```

Notre but est `A`. Nous avons un `x` de type `A` dans notre contexte.

```
  exact x.
Qed.

Print K_tac.
```

4 Application (Modus Ponens & Transitivité)

Comment utilisons-nous le Modus Ponens ? Si nous avons `f : A -> B` et `x : A` dans notre contexte, nous pouvons appliquer `f` à `x` pour obtenir `B`. La tactique `apply` fonctionne à l'envers : si notre but est `B`, et nous avons `f : A -> B`, `apply f` change notre but en `A`.

Cela correspond à un cas particulier de la règle de typage de l'application en λ-calcul (la règle d'élimination de l'implication en déduction naturelle), là encore lue de bas en haut :

$$\frac{\dfrac{}{\Gamma, f : A \to B \vdash f : A \to B}\,\text{Var} \qquad \Gamma, f : A \to B \vdash M : A}{\Gamma, f : A \to B \vdash f\ M : B}\,\text{App}$$

```
Lemma transitivity : (A -> B) -> (B -> C) -> (A -> C).
Proof.
  intros f g x.
```

But actuel : `C`. Nous avons `g : B -> C`. Si nous pouvons prouver `B`, `g` nous donnera `C`.

```
  apply g.
```

But actuel : `B`. Nous avons `f : A -> B`. Si nous pouvons prouver `A`, `f` nous donnera `B`.

```
  apply f.
```

But actuel : `A`. Nous avons exactement `x : A` !

```
  exact x.
Qed.
```

Remarquez que le λ-terme sous-jacent implémente exactement la composition de fonction $(g \circ f)(x) = g(f(x))$!

```
Print transitivity.
```

5 Exercices

Exercice 1

Construisez le combinateur S en mode Preuve Interactive. (Indice : utilisez les tactiques `intros`, `apply` et `exact`)

```
Lemma S_tac : (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
  (* ECRIVEZ VOTRE PREUVE ICI *)
Admitted. (* Changez 'Admitted' en 'Qed' quand vous avez termine ! *)
```

La commande `Axiom` permet de rajouter une nouvelle constante typée dans l'environnement "global", i.e. qui sera utilisable dans toutes les définitions et démonstrations subséquentes. Elle contraste avec la commande `Variables`, qui se contente de rajouter des variables/hypothèses communes à toutes les définitions se trouvant dans la même section.

Du point de vue de la théorie des types (théorie de la démonstration), cela revient à rajouter une nouvelle règle axiomatique (i.e. sans prémisses) permettant d'habiter n'importe quel type (de démontrer n'importe quelle proposition) :

$$\frac{}{\Gamma \vdash c : A}\,\text{Axiom}$$

C'est donc une règle extrêmement puissante et dangereuse, comme le montre la constante `todo` ci-après capable de prouver n'importe quelle proposition `A` !

```
Axiom todo : forall {A : Prop}, A.
```

Bien que cela rende le système inconsistant, ce n'est pas réellement un problème : Rocq est capable de tracer l'utilisation d'axiomes dans une démonstration, ce qui nous permet de savoir exactement à quel point celle-ci utilise des principes que l'on considère acceptables ou non. Nous verrons ainsi lors de la prochaine séance comment faire de la logique classique avec l'axiome du tiers-exclu $A \vee \neg A$.

Exercice 2

Définissez manuellement un λ-terme pur typable par $(A \to B \to C) \to (B \to A \to C)$, dont le comportement consiste à échanger la position des deux arguments d'une fonction.

```
Definition swap : (A -> B -> C) -> (B -> A -> C) :=
  todo.
  (* REMPLACEZ [todo] PAR VOTRE TERME *)

End Lesson1.
```