Initiation aux preuves formelles avec Rocq à l'entrée de la licence

Pierre Rousselin

PRofesseur AGrégé au département d'informatique de l'institut Galilée
Université Paris XIII dite « Sorbonne Paris Nord »
Villetaneuse

12 mars 2025

Double licence mathématiques-informatique (DL)

- Université Sorbonne Paris Nord (anciennement Paris 13), campus de Villetaneuse (Seine-Saint-Denis).
- Étudiants de double licence mathématiques et informatique
- Formation très exigeante donc on doit recruter des étudiants très motivés par les deux disciplines
- Entre 40 et 50 inscrits en L1 (2 groupes de TD/TP)

Initiation aux preuves formelles

2 groupes de 20–25 étudiants

- 2021 création avec 6 fois 3h de travaux pratiques (sous `CoqIDE`) avec Micaela Mayero et Marie Kerjean ;
- 2022 idem (avec polissage)
- 2023 passage à l'interface Web `jsCoq` avec maintenant Serguei Lenglet
- 2024 ajout de 7,5h de TD tableau noir, de 3h de cours magistraux et d'un partiel sur papier

Influence forte de *Software Foundations* au moins pour la première partie du cours.

Plan du cours

- Idée de départ : limites de suites numériques
- Plan en bref :
  1. Logique propositionnelle
  2. Entiers naturels et récurrence
  3. Calcul des prédicats et tiers exclu
  4. Nombres réels et suites numériques

Disponibilité du cours

- sujets de TD
- transparents de cours magistral
- sujets de TP (`.v` ou web)

version `waCoq` `https://www.math.univ-paris13.fr/~rousselin/ipf.html`

version `jsCoq-light` `https://www.math.univ-paris13.fr/~rousselin/ipf2.html`

- discussion plus loin sur les deux versions web
- disponibilité des sources complètes sur demande [1]

1. Comment rendre public un cours (par exemple avec `git`) sans donner toutes les solutions ?

Dans cet exposé

- Je dirai parfois `Coq`, par habitude, parfois `Rocq`, par anticipation, c'est bien la même chose.
- Je présenterai le contenu du cours d'initiation aux preuves formelles en mettant l'accent sur
- les difficultés des étudiants
- les points qui, d'après moi, et ça n'engage que moi, posent problème dans `Rocq`, ou autour, dans le cadre de ce cours.

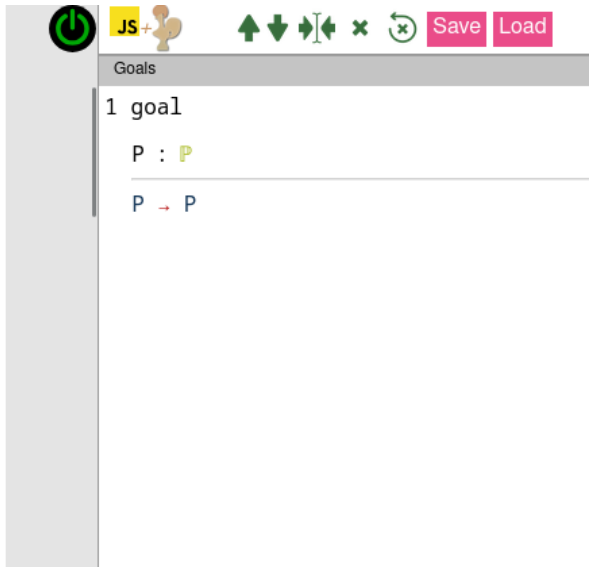
Hello, Rocq !

très détaillée correspondant à la preuve en Coq.

Ce premier théorème énonce que, quelle que soit la proposition `P`, on a `P -> P`. Cela peut paraître évident, mais en Coq il faut tout prouver !

```
19 Theorem imp_refl : ∀ P : ℙ, P → P.
20 Proof.
21   (* Soit [P] une proposition quelconque. *)
22   intros P.
23   (* Pour montrer une implication on suppose que ce qui est à gauche
24      est prouvé.
25      On doit prouver ce qui est à droite avec cette hypothèse supplémentaire.
26   (* On suppose (hypothèse ([H])) que [P] est prouvée. *)
27   intros H.
28   (* On doit prouver [P], or ([H]) prouve exactement [P]. *)
29   exact H.
30 Qed. (* Quod erat demonstrandum. Ce qu'il fallait démontrer. *)
```

Sans entrer dans tous les détails, quelques commentaires s'imposent :



Règles d'écriture des sujets

- Toujours commencer par un exemple commenté.
- Suivi d'au moins un exercice d'application directe.
- Un exercice doit toujours être faisable avec ce qui a été présenté précédemment (contrat).
- Idéalement, ne présenter qu'une notion à la fois et éviter d'ajouter du bruit (maîtrise du flux d'informations).
- Au centre de nos préoccupations : rendre les étudiants les plus actifs possible, le plus vite possible.

Règles d'écriture des sujets

- Toujours commencer par un exemple commenté.
- Suivi d'au moins un exercice d'application directe.
- Un exercice doit toujours être faisable avec ce qui a été présenté précédemment (contrat).
- Idéalement, ne présenter qu'une notion à la fois et éviter d'ajouter du bruit (maîtrise du flux d'informations).
- Au centre de nos préoccupations : rendre les étudiants les plus actifs possible, le plus vite possible.

Ce n'est pas toujours facile :

- On a parfois du mal à se retenir, par exemple l'introduction du premier fichier parlait de logique intuitionniste et de logique classique.
- Coq nous force parfois la main. Même avec `Set Printing Parentheses`, il n'y a pas de parenthèses affichées dans `P -> (Q -> P)` avant la version 8.19, ce qui force à parler d'associativité à droite de `->`.

Tactiques et règles de déduction naturelle

connecteur	introduction (prouver)	élimination (utiliser)
$\rightarrow$	`intros`	`apply`
$\wedge$	`split`	`destruct`
$\vee$	`left` ou `right`	`destruct`
$\forall$	`intros`	`specialize`, instanciation [2]
$\bot$		`destruct`
$\exists$	`exists`	`destruct`

2. explicite ou implicite avec unification

Difficultés pour les étudiants

- Confusions entre les noms d'hypothèses et leurs types.
- Choix trop précoce :
  - de `left` ou `right` dans le cas d'une disjonction
  - du témoin dans le cas d'un quantificateur existentiel
  - ou variations plus subtiles sur ces thèmes
  - et pas assez de recul pour comprendre qu'il faut revenir en arrière.
- Exemple (examen 2023) :

```
Lemma or_induction : forall (A B C : Prop),
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
  intros A B C.
  intros H K L.
  apply H.
  destruct L as [L1 | L2].
  exact L1.
Admitted.
```

Des tactiques trop puissantes à ce stade

Le tableau de correspondance entre règles de déduction naturelle et tactiques montre un idéal pédagogique. En fait, la plupart des tactiques débordent de leurs cadres pédagogiques.
On commence avec `apply`.

```
Context (A B : Prop).
Goal A /\ B -> A.
Proof.
  intros H.
  apply H. (* apply est fort *)
Qed.
```

Des tactiques trop puissantes à ce stade

Le tableau de correspondance entre règles de déduction naturelle et tactiques montre un idéal pédagogique. En fait, la plupart des tactiques débordent de leurs cadres pédagogiques.
On commence avec `apply`.

```
Context (A B : Prop).
Goal A /\ B -> A.
Proof.
  intros H.
  apply H. (* apply est fort *)
Qed.

Goal (forall (X : Prop), X -> A /\ B) -> C -> A.
Proof.
  intros H.
  apply H. (* apply est très fort *)
Qed.
```

destruct destruct

La tactique `destruct` s'applique aussi à une implication lorsque sa conséquence peut se détruire.

```
Goal (C -> A /\ B) -> C -> A.
Proof.
  intros H H'.
  destruct H. (* apparition du sous-but C *)
```

destruct destruct

La tactique `destruct` s'applique aussi à une implication lorsque sa conséquence peut se détruire.

```
Goal (C -> A /\ B) -> C -> A.
Proof.
  intros H H'.
  destruct H. (* apparition du sous-but C *)
```

On peut presque tout `destruct` :

```
Goal forall (n : nat),
  0 = 0 -> 0 <= 1 -> n < 3 -> False -> True -> 0 = 0.
Proof.
  intros n E I J H H'.
  destruct E.
  destruct H'.
  destruct J.
  destruct I.
  destruct n.
  all: destruct H.
Qed.
```

Débordement et *bogo-proof*

- En pratique, les étudiants ont tendance à essayer des tactiques hors du cadre prévu
  - soit parce qu'ils se sont simplement trompé
  - soit parce qu'ils sont bloqués et essaient au hasard

Débordement et *bogo-proof*

- En pratique, les étudiants ont tendance à essayer des tactiques hors du cadre prévu
  - soit parce qu'ils se sont simplement trompé
  - soit parce qu'ils sont bloqués et essaient au hasard
- ce qui pose problème quand
  - l'état de preuve devient incompréhensible parce que 12 tactiques plus haut ils ont utilisé un `destruct` qui déborde
  - une tactique (par exemple `apply`) résout magiquement un exercice, sans que cela ait fait progressé l'étudiant dans le sens prévu par l'enseignant·e.

Tensions

- Coq n'est pas fait au départ pour l'enseignement en L1, on ne peut pas vraiment reprocher à `apply` et `destruct` d'en faire autant.
- Envie d'enseigner tout de même un Coq existant pour, par exemple, que les étudiant·e·s qui le souhaitent puissent suivre « Software Foundations » sans être perdu·e·s.
- Mais pas envie non plus d'expliquer que « $\wedge$ et $\vee$ et $=$ sont des propositions inductives et donc `destruct` se comporte ainsi et aussi il y a des subtilités avec les paramètres et les indices et aussi $\bot$ n'a pas de constructeur et $\top$ a un constructeur qui s'appelle `I` ! ».
- Idéalement, il faudrait une option `Set Chick` (ou `Pebble`) ou des versions affaiblies de ces tactiques.

%Cpu: 100

Les étudiants sont actifs presque 100% du temps pendant ces TP.

- Ils sont même nombreux à ne pas prendre la pause de 15 minutes prévue (alors qu'on leur conseille de la prendre !)
- C'est vraiment frappant comparé à une séance classique d'exercices de maths
- surtout sur des exercices de logique.
- Question ouverte : est-ce que cette activité sert à quelque chose ?

(Presque) rien sous le tapis ici

On a fait des choix clairement orientés « maths-info. »

- Présentation de `nat` comme type inductif
- Utilisation de `discriminate H.` pour prouver `False` à partir de quelque chose comme `H : S n = 0`
- La définition de l'addition est un programme.
- Coq sait calculer : commande `Compute`, tactique `simpl`.

Les égalités : `rewrite`

- `rewrite <-`, `rewrite ->` et les variantes avec `in` ne posent pas de problème particulier
- Ça se complique avec `rewrite at`, c'est difficile d'expliquer

```
Goal n + m = m + n.
Proof.
  Fail rewrite Nat.add_comm at 2.
  (* Invalid occurrence number: 2. *)
```

Les égalités : `rewrite`

- `rewrite <-`, `rewrite ->` et les variantes avec `in` ne posent pas de problème particulier
- Ça se complique avec `rewrite at`, c'est difficile d'expliquer

```
Goal n + m = m + n.
Proof.
  Fail rewrite Nat.add_comm at 2.
  (* Invalid occurrence number: 2. *)
```

- `rewrite` unifie silencieusement
- Difficulté pour les étudiants : quels arguments fournir à `rewrite` ?
- Où sont les parenthèses cachées dans `2 + n - m + k` ?
- *Wishlist :* `Set Printing Parentheses "+".`

Les égalités : `reflexivity`

Pour `reflexivity` on ne veut pas parler de convertibilité, donc on présente cette tactique en disant qu'elle « prouve une égalité lorsque ses deux membres sont identiques (syntaxiquement). » Bien sûr, en réalité :

```
Goal 2 + 2 = 4.
Proof. reflexivity. Qed.
```

Donc on a encore le même phénomène de « débordement du cadre pédagogique » (et ce n'est pas fini...)

Preuves par récurrence

- Les étudiants jouent le *Natural Number Game*[3] jusqu'à la commutativité de la multiplication.
- On s'exerce à la démonstration par récurrence avec `induction`.
- On évite, quand c'est possible, d'utiliser `simpl`, qui est susceptible soit de cacher les arguments utilisés, soit de rendre la preuve beaucoup trop difficile en réduisant beaucoup trop de choses.

```
Eval simpl in (2 ^ (S (S n))).
(* = 2 ^ n + (2 ^ n + 0) + (2 ^ n + (2 ^ n + 0) + 0) : nat *)
```

- Difficulté : la généralité de l'hypothèse de récurrence dépend de la position de la variable sur laquelle ont raisonne par induction dans la formule à prouver.

3. `https://adam.math.hhu.de`

Injectivité, raisonnement vers l'arrière

Les étudiants ont toujours des difficutés au moment d'aborder le calcul des prédicats.

```
Lemma inj_comp (f : X -> Y) (g : Y -> Z) :
  injective f -> injective g -> injective (fun x => g (f x)).
Proof.
  intros Hf Hg x x' H.
  apply Hf.
  apply Hg.
  exact H.
Qed.
```

- Les étudiants (qui avaient pourtant écrit des preuves bien plus complexes) restent souvent bloqués sur cette preuve.
- Avec un raisonnement vers l'arrière, la preuve ci-dessus paraît pourtant totalement évidente.
- Hypothèse : peut-être que la difficulté est liée à l'ordre supérieur (l'énoncé est quantifié sur des fonctions). Il est probable que ces difficultés se retrouvent également dans le cadre d'un cours classique de mathématiques.

Logique classique

- Depuis deux ans, il y a un sujet spécifique sur la logique classique.
- De façon surprenante, les étudiants ont été très lents sur cette partie :
  - le tiers exclu peut être utilisé tout le temps avec n'importe quelle proposition
  - cela contraste avec la logique intuitionniste qui donne un sens assez naturel à la preuve
- Pour la suite, sauf erreur de ma part, le Coq standard propose bien peu de choses (en fait l'axiome `classic` et `NNPP` et... pas tellement plus).
- Pour un cours de mathématiques on voudrait des tactiques pour la logique classique. La Mathlib de Lean a `push_neg`, `contrapose`, `by_contra`, ...

Un devoir sur $\leq$

- En 2023, le devoir à la maison a porté en partie sur $\leq$ (sur `nat`).
- Contenu assez exigeant avec réinvestissement de tout ce qui précède.
- L'ordre $\leq$ n'est pas défini, car on ne veut pas parler de propositions inductives. On part des propositions suivantes comme si elles étaient des axiomes :

```
le_0_l : forall n : nat, 0 <= n
nle_succ_0 : forall n : nat, ~ S n <= 0
succ_le_mono : forall n m : nat, n <= m <-> S n <= S m
```

- `succ_le_mono` est une équivalence (ce qui est mathématiquement très naturel).

Problème d'équivalences et `apply`

```
Arguments Nat.succ_le_mono {n m}.
Lemma foo (n m : nat) : n <= m -> S n <= S m.
Proof.
  intros H.
  apply succ_le_mono.
  (* raté, donne S (S n) <= S (S (S m))
     apply avec équivalence essaie d'abord le sens <-,
     ce qui est un peu contre-intuitif *)
  Restart. intros H.
  Fail apply ->succ_le_mono.
  (* raté, cannot infer implicit parameter n
     apply avec -> ne fonctionne pas comme apply sans flèche
     il cherche d'abord à connaître ses arguments implicites
     avant d'appliquer l'implication voulue, au lieu du
     contraire *)
```

Problème d'équivalences et `apply`

```
Arguments Nat.succ_le_mono {n m}.
Lemma foo (n m : nat) : n <= m -> S n <= S m.
Proof.
  intros H.
  apply succ_le_mono.
  (* raté, donne S (S n) <= S (S (S m))
     apply avec équivalence essaie d'abord le sens <-,
     ce qui est un peu contre-intuitif *)
  Restart. intros H.
  Fail apply ->succ_le_mono.
  (* raté, cannot infer implicit parameter n
     apply avec -> ne fonctionne pas comme apply sans flèche
     il cherche d'abord à connaître ses arguments implicites
     avant d'appliquer l'implication voulue, au lieu du
     contraire *)
```

- Malheureusement, il arrive que la tactique `apply`, avec ou sans direction, ne fonctionne pas comme prévu avec les équivalences (voir *issue* #18177).
- Il n'y a pas d'alternative en `Rocq` *Vanilla* (hormi `setoid_rewrite` qui pose d'autres problèmes).
- C'est un problème important à régler, car les équivalences sont partout en mathématiques.

`reflexivity` et `rewrite`

Nouveaux débordements, cette fois avec `reflexivity` et `rewrite`.

`reflexivity` et `rewrite`

Nouveaux débordements, cette fois avec `reflexivity` et `rewrite`.

```
Theorem le_refl : forall n, n <= n.
Proof.
induction n as [|n']. reflexivity. reflexivity.
Qed.
Theorem le_trans : forall n m p, n <= m -> m <= p -> n <= p.
Proof.
  induction n as [| n' IH].
  intros n m p.
  -intros H. apply le_0_l.
  -intros m p. intros A B. rewrite A. exact B.
Qed.
```

reflexivity

```
(* En fait, on peut directement écrire : *)
Theorem le_refl' : forall n, n <= n.
Proof. reflexivity. Qed.
(* En fait, on peut directement écrire : *)
Theorem le_trans' : forall n m p, n <= m -> m <= p -> n <= p.
Proof. intros n m p H H'. rewrite H. exact H'. Qed.
```

reflexivity

```
(* En fait, on peut directement écrire : *)
Theorem le_refl' : forall n, n <= n.
Proof. reflexivity. Qed.
(* En fait, on peut directement écrire : *)
Theorem le_trans' : forall n m p, n <= m -> m <= p -> n <= p.
Proof. intros n m p H H'. rewrite H. exact H'. Qed.
```

Parce que dans `Coq.Numbers.NatInt.NZOrder` :

```
#[global]
Instance le_preorder : PreOrder le.
Proof. split. - exact le_refl. - exact le_trans. Qed.
```

Typeclasses et setoid

- Dans l'état actuel de `Rocq`,
- `Require` un fichier qui (transitivement) a `Require Setoid`
- c'est-à-dire à peu près n'importe quel fichier de la bibliothèque standard
- transforme silencieusement `rewrite` en `setoid_rewrite`
- et ajoute potentiellement des instances de `Typeclasses` qui modifient le comportement de `reflexivity` et `transitivity`.

```
From Stdlib Require PeanoNat.
(* Je n'ai rien importé *)
Lemma foo (A B C : Prop) : A <-> B -> B <-> A.
Proof.
  intros H1 H2.
  rewrite H1. (* Nanni ?! *)
  reflexivity. (* Hein ?! *)
Admitted.
```

Ça me paraît important d'y remédier, les mathématiciens sont très à cheval sur la différence entre égalité et équivalence.

Introduction de tactiques automatiques

À la fin du cours, on autorise les tactiques automatiques `lia` (*Linear Integer Arithmetic*) et `lra` (*Linear Real Arithmetic*). Cela fait gagner beaucoup de temps, mais n'est pas non plus sans aucune frustration pour les étudiants :

```
Lemma foo (eps : R) : 0 < eps -> 0 < eps / 2.
Proof. lra. Qed.
Lemma bar (eps : R) : 0 < eps -> 0 < 2 / eps.
Proof. intros H. Fail lra. Admitted.
```

Il serait donc, à mon avis, dangereux de ne se reposer que sur les tactiques automatiques pour des preuves « évidentes ».

Objectif du cours

```
Theorem CV_plus (An Bn : nat -> R) (l1 l2 : R) :
  Un_cv An l1 -> Un_cv Bn l2 -> Un_cv (fun n => An n + Bn n) (l1 + l2).
Proof.
  intros HA HB eps Heps.
  destruct (HA (eps / 2)) as [n1 Hn1]. lra.
  destruct (HB (eps / 2)) as [n2 Hn2]. lra.
  remember (max n1 n2) as n3 eqn:def_n3.
  exists n3.
  intros n Hn.
  replace eps with (eps/2 + eps/2) by lra.
  apply (Rle_lt_trans _ ((R_dist (An n) l1) + (R_dist (Bn n) l2))).
    apply R_dist_plus.
  apply Rplus_lt_compat.
  + apply Hn1. lia.
  + apply Hn2. lia.
Qed.
```

Amélioration ?

- Cette preuve est déjà assez jolie, on y retrouve les éléments classiques de la preuve mathématique.
- Les 6 dernières lignes sont tout de même un peu gênantes.

En mathématiques, on écrirait :

$$\begin{aligned}|A_n + B_n - (l_1 - l_2)| &= |A_n - l_1 + l_2 - B_n| \\ &\leq |A_n - l_1| + |B_n - l_2| \quad \text{(par l'inégalité triangulaire)} \\ &< \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.\end{aligned}$$

Ici, on a dû explicitement commencer par la fin :
En raisonnant par transitivité avec l'ordre $\leq$ et l'ordre $<$, il suffit de montrer que $|A_n + B_n - (l_1 - l_2)| \leq |A_n - l_1| + |B_n - l_2|$ et que $|A_n - l_1| + |B_n - l_2| < \varepsilon$. La première inégalité est l'inégalité triangulaire, la seconde s'obtient en sommant les inégalités (Hn1) et (Hn2).

Amélioration ?

- Cette preuve est déjà assez jolie, on y retrouve les éléments classiques de la preuve mathématique.
- Les 6 dernières lignes sont tout de même un peu gênantes.

En mathématiques, on écrirait :

$$\begin{aligned}|A_n + B_n - (l_1 - l_2)| &= |A_n - l_1 + l_2 - B_n| \\ &\leq |A_n - l_1| + |B_n - l_2| \quad \text{(par l'inégalité triangulaire)} \\ &< \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.\end{aligned}$$

Ici, on a dû explicitement commencer par la fin :
En raisonnant par transitivité avec l'ordre $\leq$ et l'ordre $<$, il suffit de montrer que $|A_n + B_n - (l_1 - l_2)| \leq |A_n - l_1| + |B_n - l_2|$ et que $|A_n - l_1| + |B_n - l_2| < \varepsilon$. La première inégalité est l'inégalité triangulaire, la seconde s'obtient en sommant les inégalités (Hn1) et (Hn2).
Est-ce que Coq pourrait avoir un équivalent du `calc` de Lean ?

Choix de `jsCoq`

- Document unique pour le cours et les preuves à remplir
- Avantage non négligeable : tout le monde travaille avec la même version.
- Inconvénient : loin d'être sobre, plusieurs centaines de Mo de bibliothèques `JavaScript` à télécharger contre quelques Ko.

jsCoq-light et waCoq

jsCoq et waCoq développés par Émilio Jesús Gallego Arias et Shachar Itzhaky.

- waCoq, version *web assembly*. Fonctionne bien sur tous les navigateurs, mais :
  - la sauvegarde de son travail (et surtout son chargement) sont compliqués
  - difficile en général de charger sa propre bibliothèque
  - plus mis à jour depuis Coq 8.17 (il y a 1 an et demi)
- jsCoq-light, *fork* de jsCoq par David Hamelin :
  - permet la sauvegarde et le chargement de son travail
  - permet d'utiliser plus facilement sa propre bibliothèque
  - malheureusement, le test grandeur nature montre que de nombreux navigateurs (Safari, Edge) ne parviennent pas à le faire tourner, la pile explose silencieusement, sur une série (même pas récursive) d'appels de fonctions au lancement
  - encore en 8.17
- Aucune version prévue pour accompagner le lancement de Rocq.

Liste de lemmes

- Irréalisable à plus grande échelle de donner la liste de tous les lemmes utiles.
- On revient à `Search`, mais
  - `Search` expose (presque) toutes les constantes, il y en a des milliers
  - Risque de voir des étudiants « prouver `A` avec `Nat.A` »
- Pour l'enseignement, besoin en général de contrôle fin
  - sur la sortie de `Search`,
  - en général sur l'environnement, ou à défaut sur ce que l'étudiant·e peut y voir.
- *Wishlist :* un `Search` dans le fichier courant jusqu'à la ligne courante
- (bonus) avec possibilité d'ajouter des lemmes à cette liste.

Une expérience qui n'est pas isolée

- Gazette de la SMF, octobre 2022 :
  `https://smf.emath.fr/publications/la-gazette-de-la-societe-mathematique-de-france-174-octobre-2022`
- École d'été en 2023 *Proof Assistants for Teaching* :
  `https://pat2023.icube.unistra.fr/`
- *Workshop* ThEdu :
  `https://www.uc.pt/en/congressos/thedu/ThEdu24`
- *Stream* zulip *Teaching [with] Coq* :
  `https://coq.zulipchat.com/`

Tentative de conclusion sur l'existant (1)

- Ce cours existe, est disponible en ligne, et couvre une partie des contenus mathématiques du début de la L1, en commençant par la logique (intuitionniste). Il n'est pas parfait, mais marche assez bien (sans doute beaucoup mieux que ce qu'on pourrait croire pour du `Rocq` en L1 !)
- Par choix, il s'appuie sur Coq *Vanilla* et sa bibliothèque standard (ceci pourrait évoluer), sans surcouche ou abstraction importante.
- Il est clair, cependant, qu'on n'a pas abordé de point particulièrement difficile à formaliser en Coq (espaces vectoriels...).

Tentative de conclusion sur l'existant (2)

- Les apports de ce cours sont difficiles à mesurer, mais :
  - Les anciens étudiants sont en général enthousiastes et disent que ce cours leur a été très utile.
  - Pour les matières formelles d'informatique, l'apport semble évident.
  - Chaque année une demi douzaine d'étudiants mordus font (presque) tout ce qu'on leur propose.
- Les séances tableau noir ajoutées cette année semblent utiles :
  - pour déminer les problèmes un peu fins de Coq (par exemple, ce que fait `discriminate`, comment `Rocq` calcule-t-il $3 + 2$, ...)
  - travailler les passages « preuve papier » $\leftrightarrow$ « preuve formelle ».
- À Villetaneuse, nouveau cours en L2 informatique (plus orienté informatique) en 2025-2026 avec `Rocq`.

Freins à l'amélioration de ce cours, souhaits, perspectives

- Le frein principal est l'interface web. En particulier, les limites actuelles nous empêchent d'écrire notre propre bibliothèque pour ce cours, qui pourrait elle-même résoudre beaucoup des problèmes cités dans cet exposé.
- Il faudrait améliorer `Reals` (notations cohérentes avec le reste de la bibliothèque, définitions plus proches de la pratique courante, ...) et la bibliothèque standard en général.
- Quel est le statut des `Typeclasses` et de `Setoid` dans une bibliothèque vouée à l'enseignement des mathématiques en L1 ?
- Versions restreintes de certaines tactiques, amélioration de `apply` pour le raisonnement par équivalences et ajout (d'une forme) de `calc` pour `Rocq`.
- En général, possibilité de contrôle fin sur ce à quoi a accès l'utilisateur étudiant (recherche personnalisable, etc).