Centre Inria de l'Université Grenoble Alpes,
équipe Spades

Rapport de stage

L3 Informatique Fondamentale

Analyse de fonction en Rocq :
une interface entre papier-crayon et preuves Rocq

Stagiaire :
Amaury Mazoyer

Encadrant :
Martin Bodin

4 Juin - 16 Juillet 2025

Table des matières

Introduction

Les assistants à la preuve sont des logiciels capables de vérifier la cohérence logique d'une preuve mathématique. Il est possible d'y formaliser la plupart des objets mathématiques, d'y énoncer des théorèmes, et de les montrer. Ces logiciels ont majoritairement été pensés pour un usage expert, et leur utilisation requiert donc un apprentissage parfois long.

L'acquisition de la preuve mathématique est un enjeu d'enseignement majeur à l'entrée à l'université. En particulier, le passage du secondaire à l'université augmente sensiblement le niveau d'exigence formel des démonstrations. Dans ce contexte, nous travaillons à la création d'un prototype d'interface graphique d'un assistant de preuve adapté à un public de licence. L'objectif est d'instrumenter la preuve comme on instrumente les autres objets mathématiques : calculatrices, tableurs, traceurs de courbes, logiciels de géométrie ou de calcul formel sont présents en classe de mathématiques et sont intégrés aux usages. Les assistants à la preuve semblent un outil pertinent pour cela, et plusieurs expérimentations [5] ont déjà été faites dans ce sens.

Une difficulté de ces expérimentations réside dans les différences entre la pratique courante de la preuve et le cadre formel imposé par les assistants. Cela se traduit dans la syntaxe (et plusieurs travaux [7, 11] visent à réduire ces différences) mais aussi dans les changements de registre sémiotique [3], notamment graphiques : il est courant en mathématiques de faire appel à des figures, des schémas, des diagrammes, etc. L'association de différents registres enrichit la compréhension en variant les représentations d'un même objet. À l'exception de la géométrie, les assistants de preuve sont généralement focalisés sur la production de démonstrations purement textuelles.

Exemple

En 2013, une des autrices de la librairie d'analyse réelle Coquelicot [4] a passé l'examen du baccalauréat de mathématiques en temps réel, en rédigeant les preuves en Rocq. Cet exercice est une bonne illustration de la différence entre les preuves papier-crayon et les preuves formelles. Une des questions était de déterminer la limite en 0 de la fonction définie par

$$f : x \mapsto \frac{2 + 2\ln x}{x}.$$

Alors que la preuve papier attendue par un professeur [2] est relativement concise,

> On sait que $\lim_{x\to 0^+} \ln x = -\infty$ donc par limite d'un produit et d'une somme, $\lim_{x\to 0^+} 2 + 2\ln x = -\infty$. Comme par ailleurs $\lim_{x\to 0^+} x = 0^+$, alors par limite d'un quotient on a $\lim_{x\to 0^+} f(x) = -\infty$.

prouver ce simple fait en Rocq s'avère tout de suite plus complexe. Le Listing 1 témoigne des différences entre les deux modes de preuve.

Ce stage s'inscrit dans un cadre plus grand. L'objectif final est de construire une interface pour l'assistant de preuve Rocq qui mêle un registre textuel (le mode preuve habituel des assistants de preuve), mais aussi un registre plus graphique : celui d'un tableau de variation. Cette interface permet des interactions sur les deux registres, dans le but de stimuler un passage des tableaux de variations bien connus à des démonstrations textuelles plus formelles. Lors de travaux précédents [3], un prototype d'interface a été construit où l'assistant de preuve a été remplacé par un petit solveur jetable écrit en OCaml, avec une API documentée.

```
Definition f (x : R) : R := (2 + 2 * ln x) / x.

Lemma filterlim_f_0 : filterlim f (at_right 0) (Rbar_locally m_infty).
Proof.
  unfold f.
  eapply (filterlim_comp_2 _ _ Rmult).
  eapply filterlim_comp_2.
  apply filterlim_const.
  eapply filterlim_comp_2.
  apply filterlim_const.
  by apply is_lim_ln_0.
  apply (filterlim_Rbar_mult 2 m_infty m_infty).
  unfold is_Rbar_mult, Rbar_mult'.
  case: Rle_dec (Rlt_le _ _ Rlt_0_2) => // H _ ;
  case: Rle_lt_or_eq_dec (Rlt_not_eq _ _ Rlt_0_2) => //.
  apply (filterlim_Rbar_plus 2 _ m_infty).
  by [].
  by apply filterlim_Rinv_0_right.
  by apply (filterlim_Rbar_mult m_infty p_infty).
Qed.
```

Listing 1 – Une preuve Rocq de l'exemple de la page 2

L'objectif du stage est de développer un module Rocq qui permettra à terme de remplacer le solveur OCaml jetable actuel, via l'interface web JsCoq [1]. La contribution de ce stage a été l'élaboration et le développement d'une structure de données qui soit à l'interface entre les démonstrations papier-crayon et les preuves Rocq afin de pouvoir être utilisée dans l'interface.

Ce rapport peut naturellement être lu en texte brut, mais nous fournissons également des liens hypertexte vers le code source comme support supplémentaire : ils sont indiqués par un symbole '🔫' ou '🏞'.

1 Préliminaires

1.1 Formalisation de $\mathbb{R}$ en Rocq

Alors que l'existence et la manipulation des nombres réels semblent évidentes pour un élève de licence, ce n'est pas le cas pour un assistant de preuve. En effet avant même de pouvoir manipuler le moindre réel, il est nécessaire de les définir formellement et de donner ou en déduire les propriétés les concernant.

En mathématiques, les nombres réels sont utilisés au lycée sans en donner une définition formelle, et sont concrétisés de manière constructive seulement en licence. L'approche classique enseignée est de définir les nombres réels comme les classes d'équivalence de limites de suites de Cauchy. Dans la vie de tous les jours, revenir à la définition est très peu courant et une compréhension intuitive suffit, mais cette approche n'est pas possible pour un assistant de preuve.

Certains assistants de preuve comme Isabelle/HOL ont choisi de formaliser les nombres réels à l'aide de suites de Cauchy dans leur librairie standard [4]. Ce n'est pas le cas de Rocq qui a choisi une approche axiomatique [9]. On admet l'existence d'un ensemble contenant les nombres 0, 1, muni des opérations comme l'addition, la multiplication, l'opposé, l'inverse et la prise de partie entière supérieure et possédant une relation d'ordre $<$. On ajoute ensuite des axiomes concernant l'addition, la multiplication, la distributivité du $+$

sur le $\times$ et la relation d'ordre total ce qui caractérise $\mathbb{R}$ comme un corps, et on ajoute deux axiomes qui caractérisent $\mathbb{R}$ comme archimédien et complet [1]. Cette définition est valide car tout corps ordonné, archimédien et complet est isomorphe à $\mathbb{R}$.

A l'aide de cette définition, il est alors possible de prouver toutes les propriétés usuelles des nombres réels et de définir l'analyse réelle.

1.2 Bibliothèque d'analyse réelle

Rocq possède une librairie d'analyse réelle, cependant elle possède quelques lacunes [4]. Premièrement la librairie n'a pas évolué depuis sa conception en 2001, et ainsi ne supporte pas certaines fonctionnalités plus récentes de Rocq comme les classes de types. De plus, il y a un certain manque d'homogénéité : certaines définitions et théorèmes sont manquants, il n'y a pas de convention de nommage cohérente et beaucoup d'hypothèses redondantes se sont faufilées dans les théorèmes. D'autres incohérences rendent l'usage de cette librairie très difficile, ce qui est particulièrement dérangeant à la vue de l'usage dont nous avons besoin. Il est tout de même possible de formaliser un grand nombre de preuve mathématiques comme le montre la preuve de la formule de d'Alembert pour la résolution de l'équation des ondes [6] faite par les auteurs de la librairie Coquelicot, bien qu'il soit évoqué une certaine difficulté à utiliser certaines définitions de la librairie standard et que la preuve résultante ne fasse pas moins de 2500 lignes.

Pour ces raisons, on utilise pour ce projet la librairie Coquelicot qui se veut être une extension intégralement compatible avec la librairie standard : mêmes définitions ou définitions équivalentes, pas de nouveaux axiomes.

Coquelicot s'affranchit de l'utilisation de types dépendants dans les définitions d'intégrales et de dérivées en considérant toute fonction comme des fonctions totales. Ainsi une fonction comme l'inverse est définie sur $\mathbb{R}$ tout entier et est donnée une valeur arbitraire en 0. La librairie fournit des définitions plus standard et plus exploitables en pratique, tout en fournissant des lemmes permettant la correspondance avec la librairie standard. Elle fournit également d'importants théorèmes manquants tels que l'extensionnalité de la limite ou de la dérivée. Enfin, elle développe certains domaines de l'analyse réelle qui n'étaient que peu, voire pas du tout développés dans la librairie standard, comme les séries entières, les intégrales à paramètres ou les fonctions de plusieurs variables.

1.3 Disparité entre les assistants de preuves et le raisonnement mathématique

Cependant même avec l'aide de cette librairie plus simple d'utilisation, rédiger des preuves d'analyse réelle en Rocq reste difficile. Ceci est le cas pour plusieurs raisons.

Premièrement, le formalisme de Rocq vient entraver le raisonnement mathématique naturel de par les notations ainsi que la nécessité de prouver des sous-buts correspondants à des étapes qui n'auraient jamais été justifiées à l'écrit. Par exemple, alors que déterminer la dérivée d'un polynôme se fait généralement en une unique étape à l'écrit, il est nécessaire d'appliquer les règles de dérivation de somme, de produit, de constante et de l'identité de multiples fois pour prouver le même résultat. L'exemple de l'introduction illustre bien ce fait. Un autre exemple pourtant trivial est de prouver que `2 + 2 = 4`. En effet, à cause de l'approche axiomatique de la construction des réels, il n'est pas possible d'effectuer directement des calculs pour montrer ce but. Il faut alors utiliser des lemmes concernant l'addition comme le montre le Listing 2.

1. Ces axiomes sont disponibles à `https://rocq-prover.org/doc/v8.10/stdlib/Coq.Reals.Raxioms.html`

```
Goal 2 + 2 = 4.
  compute. (* pas suffisant car + se comporte comme une "boîte noire" *)

  (* axiomes *)
  rewrite Rmult_plus_distr_l.
  rewrite Rmult_comm.
  rewrite Rmult_1_l.

  reflexivity.
Qed.
```

Listing 2 – Une preuve que `2 + 2 = 4` utilisant les axiomes de la construction des réels

Une autre différence qui entrave le raisonnement usuel est le choix des définitions. Cela vient probablement du fait que l'on préfère en Rocq la définition la plus générale possible afin de pouvoir traiter le maximum d'énoncés. Ainsi, certaines définitions, bien qu'équivalentes, ne sont pas utilisables par des élèves de licence. C'est le cas notamment de la définition de limite finie en un point finie, où après dépliage de toutes les définitions intermédiaires, le prédicat affirmant que la limite d'une fonction `f` au point `x` est `l` s'écrit

```
forall P : R -> Prop,
  (exists eps : posreal, forall y : R_UniformSpace, ball l eps y -> P y) ->
  exists eps : posreal, forall y : R_UniformSpace, ball x eps y -> y <> x ->
  P (f y)
```

On ne reconnaît pas la définition usuelle enseignée en licence

$$\forall \varepsilon > 0,\ \exists \delta > 0,\ \forall y \in \mathbb{R},\ |x - y| < \delta \Rightarrow |f(y) - l| < \varepsilon$$

et les deux définitions ne sont même pas équivalentes puisque la formalisation choisie en Rocq est celle de limite épointée. Plus brutale est la définition des limites à droite et à gauche qui s'appuie sur la notion de filtres [4], notion complètement inutilisable pour un élève de licence et qui donne du fil à retordre même pour une personne familière avec Rocq et l'analyse réelle, mais qui permet une généralisation des limites en topologie.

2 Résultats

2.1 Arbre de preuve

La syntaxe des preuves Rocq étant trop distante de celle des démonstrations papier-crayon, il est nécessaire de trouver un intermédiaire qui permettrait d'interfacer les deux tout en gardant la rigueur de l'assistant de preuve et la subtilité du papier-crayon. L'objectif est donc de développer une structure de données suffisamment adaptée pour permettre une traduction vers une preuve Rocq ainsi que vers une démonstration lisible par un · e étudiant · e, comme sur la Figure 1.

Afin de pouvoir exploiter cette structure de données dans le futur, il est aussi légitime de vouloir pouvoir traduire des démonstrations d'étudiants vers cette structure de données avec relative aisance.

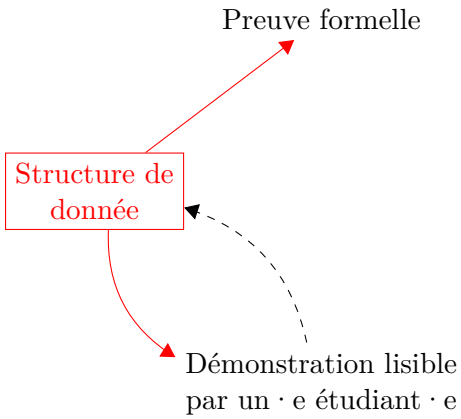


FIGURE 1 – Diagramme expliquant les objectifs

2.1.1 Une première approche

Dans une première approche, on considère une grammaire en λ–calcul typé proche de celle des termes Rocq, à laquelle on ajoute une liste de théorèmes choisis et utilisables dans l'arbre de preuve (grammaire incomplète) :

$$t ::= x \mid t_1\ t_2 \mid \lambda(x : t_1).t_2 \mid \cdots \mid \texttt{thm}$$
$$\texttt{thm} ::= \forall P, Q, P \wedge Q \Rightarrow P \mid \cdots$$

Cette approche répond au premier critère : il est facile de transformer un tel arbre de preuve en terme de preuve Rocq. Cependant beaucoup de nuances des preuves papier-crayon ne sont pas capturées.

Par exemple, il n'est pas possible de distinguer le mode de preuve "forward" (en avant) ou "backward" (en arrière).

Ce premier est un raisonnement constructif et consiste à construire la démonstration à partir des hypothèses afin d'arriver au but. C'est notamment le style d'écriture utilisé dans la preuve papier-crayon de l'exemple de l'introduction. Pour prouver un but `C` en sachant `A -> B -> C`, on prouverait d'abord `A` puis `B` pour en déduire `C`.

Le mode backward lui consiste à partir du but et à fournir des énoncés suffisants et éventuellement d'autres sous-buts afin de le prouver. Ce style de preuve est le plus naturel en Rocq, bien que les deux modes soient possibles. Pour prouver un but `C` en sachant `A -> B -> C`, on dirait qu'il suffit de prouver `B` puis qu'il suffit de prouver `A`.

Il est également possible d'écrire une même preuve dans les deux modes en papier crayon. Voici une preuve backward de l'exemple de la page 2 :

> Par limite d'un quotient (6), déterminons la limite de $x \mapsto 2 + 2\ln x$ et de $x \mapsto x$ en 0. Par limite d'une somme (7), déterminons la limite de $x \mapsto 2$ et de $x \mapsto 2\ln x$ en 0. Par limite d'une constante (8), $\lim\limits_{x\to 0} 2 = 2$. Par limite d'un produit (9), déterminons la limite de $x \mapsto 2$ et de $x \mapsto \ln x$ en 0. Par limite d'une constante (10), $\lim\limits_{x\to 0} 2 = 2$. Par limite du ln en 0 (11), $\lim\limits_{x\to 0^+} \ln x = -\infty$. Puisque $2 \times -\infty = -\infty$ (13,14,15), $\lim\limits_{x\to 0^+} 2\ln x = -\infty$. Puisque $2 + (-\infty) = -\infty$ (17), $\lim\limits_{x\to 0^+} 2 + 2\ln x = -\infty$. Par limite de l'identité (18), $\lim\limits_{x\to 0^+} x = 0^+$. Puisque $\frac{-\infty}{0^+} = -\infty$, $\lim\limits_{x\to 0^+} f(x) = -\infty$.

Avec un oeil averti, on peut déceler la structure de la preuve Rocq initiale à partir de cette preuve papier-crayon (les annotations en violet correspondent [2] aux numéros de ligne dans le Listing 1).

Le mode de raisonnement utilisé lors des preuves papier-crayon n'est pas unanime, il dépend de la preuve en question ainsi que des préférences personnelles. Il semble donc important de garder une trace de ce choix dans l'arbre de preuve.

2.1.2 λ−terme annoté

Une solution simple au problème précédent est d'annoter la grammaire avec des informations relatives à la preuve papier-crayon correspondante :

$$t ::= x \mid t_1 \ \updownarrow t_2 \mid \lambda(x : t_1).t_2 \mid \cdots \mid \texttt{thm}$$
$$\updownarrow ::= \uparrow \mid \downarrow$$

où $\uparrow$ est ajouté pour annoter un terme prouvé avant son utilisation en tant qu'argument (raisonnement forward), et $\downarrow$ est ajouté pour annoter un terme prouvé après son utilisation en tant qu'argument (raisonnement backward).

Cependant cette approche n'est toujours pas satisfaisante : il reste toujours des constructions papier-crayon n'ayant pas d'équivalent. On peut citer par exemple la nuance entre "Soit ..." et "Supposons que ...". De plus, certaines parties des démonstrations découlent de la logique ou de raisonnements communs qu'il est nécessaire de capturer pour obtenir une preuve authentique. Ajouter d'autres annotations dans la grammaire ci-dessus semble maladroit et risque rapidement de devenir illisible. Enfin la structure recherchée doit pouvoir représenter des productions d'élèves, on ne souhaite pas être biaisé par la structure des λ−termes Rocq dans son élaboration.

2.2 Type inductif Rocq pour l'arbre de preuve

Pour les raisons mentionnées dans la partie précédente (2.1.2), nous allons considérer un type inductif possédant des constructions inspirées des preuves papiers tout en possédant la rigueur nécessaire pour reconstruire une preuve Rocq.

Après lecture de plusieurs corrigés d'exercices de mathématiques de niveau licence, nous avons pu identifier plusieurs constructions essentielles d'une démonstration papier-crayon. Ces constructions nous permettent directement de définir un type inductif `Construction` () qui contient toutes les informations nécessaires pour déduire à la fois une preuve Rocq ainsi qu'une démonstration papier-crayon.

2.2.1 Stocker des informations supplémentaires

Cependant quelques considérations sont à prendre en compte pour le développement. Contrairement aux preuves Rocq, rappeler le but est une pratique courante des démonstrations papier-crayon. Il faut donc trouver une manière de stocker cette information dans l'arbre de preuve. Puisque l'on veut créer un objet qui représente la preuve indépendamment du contexte, il est nécessaire que tous les termes stockés soient clos. La solution

2. Il n'y a pas correspondance exacte. Par exemple, la preuve considère $\frac{2+2\ln x}{x}$ comme le produit de $2+2\ln x$ et $\frac{1}{x}$ et non comme un quotient. De plus, certaines lignes de la preuve Rocq n'apparaissent pas ici.

trouvée est de stocker à la fois les hypothèses et le but dans l'arbre, en utilisant une fonction identité

```
Definition goal {A : Type} (x : A) := x.
```

qui sert de wrapper pour le but. De cette manière, on peut discerner les hypothèses du but. Par exemple, `forall (x : R), x > 0 -> goal (x <> 0)` signifie que l'on veut prouver `x <> 0` sachant `x > 0`. Cela revient à annoter l'arbre syntaxique Rocq avec des marqueurs.

2.2.2 Exploiter les informations stockées

Afin d'exploiter ces informations supplémentaires stockées dans l'arbre, il est naturel de vouloir effectuer un pattern matching en Ltac afin de récupérer le terme à l'intérieur du `goal` par exemple. Cette tâche s'avère impossible : dans l'exemple si dessus, le terme `x <> 0` est lié par le quantificateur `forall (x : R)`. Ainsi `x <> 0` n'est pas dans le scope global mais dans le scope local de cette branche de l'arbre de preuve. Ainsi l'exécution du code suivant qui devrait afficher l'intérieur du wrapper `goal` renvoit une erreur "No matching clauses for match."

```
match constr:(forall (x : R), x > 0 -> goal (x <> 0)) with
| context [goal ?inner] => idtac inner
end.
```

Pour pouvoir manipuler ce terme, `x` doit être introduit dans le contexte ce qui est impossible.

Pour ces raisons, nous sommes contraint de dérouler la preuve en même temps que l'exploitation de l'arbre. Cela permet d'utiliser le contexte pour stocker toutes les informations nécessaires à notre place et de les exploiter. En contrepartie, il est nécessaire que les informations stockées dans l'arbre soient suffisantes pour en déduire une preuve Rocq correcte afin de pouvoir exploiter l'arbre.

2.2.3 Présentation du type inductif

Finalement, les cas suivants ont été implémentés :

— L'introduction de variable à l'aide de la construction "Soit ..."

```
| introduction_variable :
^^IConstruction (* reste de la preuve *) -> Construction
```

Remarque : Par analogie avec la tactique `intro` de Rocq, on s'attendrait à devoir stocker le nom de la variable introduite. Il n'est pas nécessaire ici de le faire puisque cette information est portée par le contexte de la preuve. Par nécessité d'avoir des termes clos dans l'arbre de preuve, il ne serait même pas possible de stocker une telle information.

— L'introduction d'hypothèse à l'aide de la construction "Supposons que ..."

```
| introduction_hypothese :
^^IConstruction (* reste de la preuve *) -> Construction
```

— L'utilisation d'une hypothèse pour conclure

```
| hypothese : Construction
```

Remarque : Puisque cette construction sert à conclure, il n'y a pas de sous-arbre.

— L'utilisation d'une ou plusieurs étapes de calcul pour conclure

```
| calcul : Construction
```

— L'application d'un théorème ou d'une hypothèse

```
| application : Thm -> list Thm_arg -> Construction
```

— L'affirmation d'un fait

```
| assertion :
^^Iforall {assertion : Prop}, assertion (* fait affirmé *) ->
^^IConstruction (* preuve du fait affirmé *) ->
^^IConstruction (* reste de la preuve en supposant le fait *) ->
^^IConstruction
```

— L'abandon d'un sous-but, ce qui revient à l'admettre

```
| trou : Construction
```

Remarque : Beaucoup de ces constructions ont des tactiques Rocq correspondantes, mais il s'agit ici d'une simple coïncidence. En effet, cet arbre est le fruit de l'analyse de preuves papier-crayon. Il s'avère que ces constructions sont si communes qu'elles ont leur place en tant que tactique dédiée.

Les types inductifs `Thm` et `Thm_arg` utilisés dans le cas de l'application permettent de capturer des subtilités supplémentaires découlant des démonstrations papier-crayon.

```
Inductive Thm : Type :=
  | nomme : string -> Thm
  | anonyme : forall {enonce : Prop}, enonce -> Thm
  | hyp : forall {enonce : Prop}, enonce -> Thm.
```

— Le cas `nomme` correspond à l'utilisation d'un théorème en l'appelant par son nom. Il s'en suit qu'il est nécessaire de fournir une liste de théorèmes disponibles à l'utilisation.

— Le cas `anonyme` découle d'une observation effectuée lors de la lecture de démonstrations papier-crayon : contrairement à Rocq, beaucoup de théorèmes sont utilisés de manière anonyme en rappelant leur énoncé. Ces théorèmes peuvent être aussi simple que $P \wedge Q \Rightarrow P$ comme dans la phrase "Comme $P \wedge Q$ alors on a P.".

— Le cas `hyp` permet d'utiliser un théorème ou un fait déjà prouvé en hypothèse. Il est strictement plus puissant que la construction `hypothese` du type inductif `Construction` mais cette dernière est utile car elle permet de ne pas rappeler l'hypothèse.

Remarque : On retrouve des constructions similaires à celles proposées dans la section 2.1.2 pour les hypothèses des théorèmes.

```
Inductive Thm_arg : Type :=
  | forward : forall {enonce : Prop}, enonce -> Thm_arg
  | backward : forall {enonce : Prop}, enonce ->
  ^^IConstruction (* preuve de l'argument *) -> Thm_arg.
```

— Le cas `forward` permet de référer un argument qui a déjà été prouvé auparavant dans la preuve. Pour ces raisons, il n'est pas nécessaire de contenir la preuve de l'argument.

— Le cas `backward` permet lui de fournir la preuve de l'argument à la suite de son utilisation dans un théorème.

Enfin, un type inductif supplémentaire vient englober ces constructions de manière à ce que l'intégralité des informations relatives à la preuve soient contenues dans un même objet :

```
Inductive Arbre : Type :=
  | racine : forall {but : Prop}, but -> Construction -> Arbre.
```

En stockant l'énoncé du but dans l'arbre, il contient ainsi l'intégralité des informations relatives à la preuve.

Les constructions présentées ne sont pas les seules et ne sont pas suffisantes à traiter l'intégralité des démonstrations papier-crayon : ce sont celles qui ont été traitées durant ce stage. Les constructions dépendant de la communauté visée, il est de toute manière presque impossible d'être exhaustif.

D'autres constructions nécessaires sont :

— La disjonction de cas

— D'autres types de raisonnement comme le raisonnement par l'absurde ou le raisonnement par récurrence (ou induction de manière plus générale)

2.3 Reconstruction de la preuve

L'exemple de la page 2 ressemblerait [3] alors à quelque chose comme le Listing 3 :

```
racine (filterlim f (at_right 0) (Rbar_locally m_infty)) (
   application (
     (nomme "limite d'un quotient")
     [
       (backward (filterlim (fun x : R => 2 + 2 * ln x) (at_right 0) m_infty)
         (application
           (nomme "limite d'une somme")
           [ (* preuve des limites de 2 et 2 * ln x *) ])
       );
       (backward (filterlim (fun x : R => x) (at_right 0) 0)
         (application
           (anonyme (filterlim (fun x : R => x) (at_right 0) 0))
           [ (* pas d'argument *) ])
       )
     ]
   )
 )
```

Listing 3 – Ébauche d'implémentation de la preuve de la page 2 à l'aide du type inductif.

Il ne reste maintenant plus qu'à pouvoir reconstruire la preuve Rocq ainsi qu'une démonstration papier-crayon à partir de ce type inductif. Cette transformation a été facilitée par certaines considérations lors de l'élaboration de l'arbre de preuve. En particulier les `forall {A : Prop}, A` permettent de prendre n'importe quel terme clos en argument, ce qui nous permet de récupérer des informations supplémentaires que l'on ne peut pas déduire du contexte.

3. En réalité il n'est pas possible actuellement de représenter cette preuve à l'aide du type inductif à cause de la lourdeur de la formalisation des limites en Rocq.

Déduire une preuve Rocq à partir de l'arbre est plutôt aisé. En effet, puisque les constructions sont proches de tactiques existantes il suffit d'utiliser les informations de l'arbre pour appliquer ces tactiques. En résolvant le but à la racine de l'arbre, on a ainsi construit une preuve Rocq avec succès.

2.3.1 Exploiter les assertions

Un cas en particulier est toutefois délicat à traiter : les assertions. En effet, pour les mêmes raisons que dans la section 2.2, il est nécessaire de stocker un terme clos dans la branche `assertion`. Cependant les assertions ne portent pas forcément sur des termes clos, comme dans le cas suivant :

```
Goal forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
intros P Q HPQ.
assert (P /\ Q -> P).
```

On va utiliser encore une fois le wrapper `goal` pour stocker les hypothèses et l'assertion de manière séparée dans un même terme. Ici on aurait donc le terme `forall P Q : Prop, P /\ Q -> goal (P /\ Q -> P)`. Mais le problème ne s'arrête pas là : les `P` et `Q` dans ce terme ne sont pas les mêmes que ceux dans le contexte, ils représentent des `P` et `Q` génériques. On va donc spécialiser la nouvelle hypothèse obtenue une fois trouvée avec les éléments du contexte pour retrouver un terme quantifié sur les bonnes variables. Concrètement, cela revient ici à prouver l'hypothèse `forall P Q : Prop, P /\ Q -> goal (P /\ Q -> P)` avec les données de l'arbre puis de la spécialiser en lui donnant les hypothèses du contexte (à savoir `P : Prop`, `Q : Prop` et `HPQ : P /\ Q`) afin d'obtenir l'hypothèse `P /\ Q -> P` voulue.

2.3.2 Automatisations nécessaires

La construction `calcul` n'a pas d'équivalent direct en Rocq. Il est donc nécessaire de développer une tactique que l'on nomme `solve_equ` permettant de résoudre ces buts ci. Cependant il n'y a pas de consensus sur la spécification de cette tactique. Des calculs considérés comme triviaux pour un élève de master ne le sont pas forcément pour un élève de L1 ou un lycéen. Ainsi, l'implémentation dépend du public visé. Développer cette tactique ne fait pas partie des objectifs du stage, mais on peut quand même déterminer une procédure pour évaluer la puissance attendue. Nous proposons de fabriquer un corpus d'énoncés de niveaux variés qui seront ensuite donnés à un professeur afin qu'il puisse déterminer leur caractère trivial ou non. Des expériences en classe permettront alors d'évaluer s'il manque des énoncés ou s'il y en a trop.

Une autre tactique automatisée est nécessaire pour mener à bien la conversion. En effet, l'appel de théorèmes anonymes en rappelant leur énoncé pose la question de quel énoncé est accepté tel quel, et quel énoncé devrait être prouvé séparément. Cette tactique nommée `solve_trivially` est une extension de celle du paragraphe précédent, et ainsi une même procédure d'évaluation devrait être effectuée pour déterminer la spécification attendue.

2.3.3 Traduction en une démonstration lisible par un · e étudiant · e

Reconstruire une preuve papier-crayon devient tout de suite plus compliqué à cause des spécificités du langage Ltac. La première difficulté vient du fait qu'une tactique ne puisse

pas à la fois renvoyer une valeur et modifier l'état de preuve. En effet, il existe plusieurs types de tactiques en Rocq. Alors que la plupart des tactiques modifient l'état de la preuve afin d'avancer dans sa résolution, d'autres permettent de construire des objets (ce sont les "value_tactic" [10]). Ces deux types de tactiques sont incompatibles : on est soit l'un soit l'autre. Or l'exploitation de l'arbre nécessite de dérouler la preuve ; il est donc impossible de renvoyer la chaîne de caractère. Pour ces raisons, nous utilisons des effets de bord à l'aide de variables existentielles (evar).

Finalement, on peut presque traduire le Listing 3 en une preuve papier-crayon. Cela donnerait une démonstration [4] comme celle de la Figure 2.

> Par limite d'un quotient, il suffit de montrer que `filterlim (fun x : R => 2 + 2 * ln x) (at_right 0) m_infty` et `filterlim (fun x : R => x) (at_right 0) 0`.
> Montrons que `filterlim (fun x : R => 2 + 2 * ln x) (at_right 0) m_infty`.
> Par limite d'une somme, il suffit de montrer que ...
> ...
> Montrons que `filterlim (fun x : R => x) (at_right 0) 0`.
> Comme `filterlim (fun x : R => x) (at_right 0) 0`, on conclut que `filterlim (fun x : R => x) (at_right 0) 0`.

FIGURE 2 – Traduction automatique de l'arbre du Listing 3 en démonstration lisible par un · e étudiant · e.

Il manque cependant une dernière étape. Alors qu'une preuve Rocq fait seulement appel à des tactiques sans jamais rappeler directement le but ou les hypothèses, ce n'est pas le cas d'une démonstration papier-crayon. Dans l'optique de produire des preuves lisibles par un élève, il s'avère donc nécessaire de pouvoir transformer des termes Rocq en string Rocq lisible (par exemple `filterlim (fun x : R => x) (at_right 0) 0` dans la Figure 2). Alors que cette tâche semble relativement simple, il n'est pas possible d'obtenir de résultats satisfaisants à l'aide de simples définitions ou tactiques Rocq. Pour ces raisons, nous avons choisi de développer un plugin en OCaml qui répond à nos besoins.

2.4 Plugin OCaml

Rocq étant écrit en OCaml, il est possible de développer directement des plugins ayant accès à l'API interne afin de rajouter des fonctionnalités qui ne seraient pas possibles autrement. Ici le but est de développer une tactique qui permet de faire appel au pretty printer interne afin de récupérer une représentation convenable des termes pour ensuite construire le terme Rocq correspondant à la string en question. La tâche semble plutôt directe : la tactique `idtac` permet déjà d'afficher un terme sous la forme voulue. Pourtant elle s'est avérée plus compliquée que prévu.

2.4.1 Difficultés d'implémentation

Premièrement, presque aucune documentation n'est donnée sur l'API interne. On peut trouver dans le code source quelques commentaires clairsemés mais la navigation du projet pour un non-expert est rude. Il existe néanmoins un tutoriel présentant les bases de la création d'un plugin, cependant il n'est pas complet et n'était pas suffisant pour répondre à nos besoins.

4. Cette démonstration est une chaîne de caractère du type `string` en Rocq.

De plus, la structure du projet elle-même rend la tâche plus compliquée. Il semble justifié de vouloir que notre tactique soit une "value_tactic" comme expliqué dans la section 2.3 puisqu'on souhaite récupérer une chaîne de caractère, et il existe une syntaxe pour définir de nouvelles tactiques et prévenir le parser,

```
TACTIC EXTEND <groupe_tactique>
| [ "<nom_tactique>" <args> ] -> { <corps de la tactique en OCaml> }
| <éventuelles autres tactiques du même groupe>
END
```

cependant cette syntaxe n'accepte que des `unit tactic` ce qui ne permet pas de renvoyer une valeur. De manière étonnante, il s'avère que les "value_tactic" primitives de Rocq comme `fresh` font partie intégrante du type inductif définissant la syntaxe des tactiques, il n'est donc pas possible de s'en inspirer pour développer notre tactique ni même de les étendre.

Enfin par nécessité d'importer la bibliothèque `String` de la bibliothèque standard, il semble impossible de pouvoir définir une fonction (comme avec `Definition <nom> <args> := <corps>`) : les définitions sont statiques et il est nécessaire d'avoir accès à l'environnement global pour pouvoir manipuler les constructeurs du type `string`, cependant l'environnement n'est pas disponible au moment du chargement du plugin.

Ces problèmes sont peut-être artificiels, mais par manque de documentation il est difficile de savoir pour sûr s'il est effectivement impossible d'accomplir la tâche de la manière envisagée.

2.4.2 Tactiques implémentées

La solution retenue est d'appeler la tactique `pose` depuis le plugin afin de pouvoir définir la string dans le contexte directement et ainsi pouvoir l'utiliser ultérieurement. Deux tactiques ont ainsi été implémentées en OCaml :

— `pose_string_of_term name term` ajoute une hypothèse `name` de type `string` dans le contexte et dont la définition est la représentation de `term` tel qu'affiché par le pretty printer interne. ()

— `pose_string_of_intropattern` se comporte comme `pose_string_of_term` mais est utilisée sur un intropattern. ()

Exemple

A la suite de l'exécution du code suivant

```
Goal forall (f : R -> R) (x : R), ex_derive f x -> continuity_pt f x.
Proof.
match goal with
| |- ?g => pose_string_of_term goal_str g
end.
```

le contexte contient l'hypothèse `goal_str := "(forall (f : R -> R) (x : R), ex_derive f x -> continuity_pt f x)" : string`.

Exécuter ensuite l'instruction

```
match goal with
| |- forall (binder : _), _ => pose_string_of_intropattern binder_str binder
end.
```

> ajoute l'hypothèse `binder_str := "f" : string` dans le contexte puisque la première variable quantifiée dans le but est `f`.

Grâce à ces tactiques, on peut alors obtenir la chaîne de caractère pour le terme `filterlim (fun x : R => x) (at_right 0) 0` dans la Figure 2 par exemple.

3 Related Work

Alors que l'objectif ici était de trouver un intermédiaire entre les preuves Rocq et les démonstrations papier-crayon, d'autres projets visent eux à rendre l'utilisation des assistants de preuve plus abordable pour des non-experts, souvent un public d'université. Deux projets ont été source d'inspiration dans ce projet.

Le premier est Coq Waterproof [11]. Ce plugin est une extension de Rocq définissant des tactiques ressemblant au langage naturel afin de pouvoir écrire des preuves Rocq ressemblant à des preuves papier-crayon. Il introduit également des notations mathématiques améliorant une fois de plus la lisibilité. Alors que la forme d'une preuve Rocq est rarement une préoccupation et que la lecture des tactiques ne suffit pas à comprendre à elle seule la structure de la preuve, l'objectif de Coq Waterproof en est autrement. Les mathématiciens étrangers de la syntaxe du langage sont capables de lire et comprendre les scripts de preuve comme en témoigne le Listing 4.

```
Goal 2 is the infimum of [2, 5).
Proof.
We need to show that (2 is a _lower bound_ for [2, 5)
  /\ (∀ l ∈ ℝ, l is a _lower bound_ for [2, 5) ⇒ l ⩽ 2)).
We show both statements.
  - We need to show that (2 is a lower bound for [2, 5)).
    We need to show that (∀ c ∈ [2, 5), 2 ⩽ c).
    Take c ∈ [2, 5).
    We conclude that (2 ⩽ c).
  - We need to show that (∀ l ∈ ℝ, l is a lower bound for [2, 5) ⇒ l ⩽ 2).
    Take l ∈ ℝ. Assume that (l is a lower bound for [2, 5)).
    We conclude that (l ⩽ 2).
Qed.
```

Listing 4 – Une preuve en Coq Waterproof

De plus Coq Waterproof fait un pas de plus vers les preuves papier-crayon en fournissant des automatisations qui permettent de cacher certains détails qui n'auraient jamais été explicités dans une démonstration et qui sont pourtant nécessaire en Rocq. Des automatisations similaires sont à implémenter dans notre projet si on veut pouvoir traduire des démonstrations d'élèves en preuve Rocq à l'aide de l'inductif, et les tactiques `solve_equ` et `solve_trivially` sont un premier pas vers cette direction.

Le second est Verbose Lean 4 [7], un projet basé sur l'assistant de preuve Lean. Développé par Patrick Massot, un enseignant-chercheur, cette extension de Lean propose des fonctionnalités similaires à celles de Coq Waterproof. Il a été développé dans l'objectif d'aider à la transition entre le lycée et l'enseignement supérieur [8] et est utilisé dans un cours intitulé "Logique et démonstrations assistées par ordinateur" à la faculté des sciences d'Orsay depuis plusieurs années [5]. La syntaxe des tactiques créées est très proche du langage naturel comme le montre le Listing 5 et on peut ainsi apprendre de la correspondance entre les tactiques Lean sous-jacentes et les formulations type utilisées. De plus, le projet entier comporte une traduction en français.

```
Exercice "La continuité implique la continuité séquentielle."
  Données : (f : ℝ → ℝ) (u : ℕ → ℝ) (x₀ : ℝ)
  Hypothèses : (hu : u tend vers x₀) (hf : f est continue en x₀)
  Conclusion : f ∘ u tend vers f x₀
Démonstration :
Montrons que ∀ ε > 0, ∃ N, ∀ n ⩾ N, |f (u n) - f x₀| ⩽ ε
Soit ε > 0
Comme f est continue en x₀ et ε > 0 on obtient δ tel que
  (δ_pos : δ > 0) et (Hf : ∀ x, |x - x₀| ⩽ δ ⇒ |f x - f x₀| ⩽ ε)
Comme u tend vers x₀ et δ > 0 on obtient N tel que Hu : ∀ n ⩾ N, |u n - x₀| ⩽ δ
...
QED
```

Listing 5 – Extrait d'une preuve en Verbose Lean

Conclusion et perspectives

Travail effectué

Ce travail constitue un premier pas vers la réduction de l'écart entre les preuves Rocq et les démonstrations papier-crayon. Il montre les disparités entre les deux et souligne les difficultés liées à la différence fondamentale de ces deux modes de raisonnement.

Grâce à cette nouvelle structure de données, il est possible dans certaines circonstances de générer automatiquement des preuves. C'est le cas par exemple des preuves de dérivation où l'application successive des règles de dérivation en analysant le but suffit. Cela permet alors de générer une preuve lisible par un · e étudiant · e de manière automatique. Cette fonctionnalité peut alors être intégrée dans le prototype d'interface graphique [5] mentionné dans l'introduction : appuyer sur le bouton "voir" (Figure 3a) génère actuellement un énoncé de lemme avec une preuve vide (Figure 3b) ce qui pourra être remplacé par une preuve générée automatiquement. La structure de données aide à la génération automatique de preuve car elle peut être construite par une tactique automatisée résolvant le but.

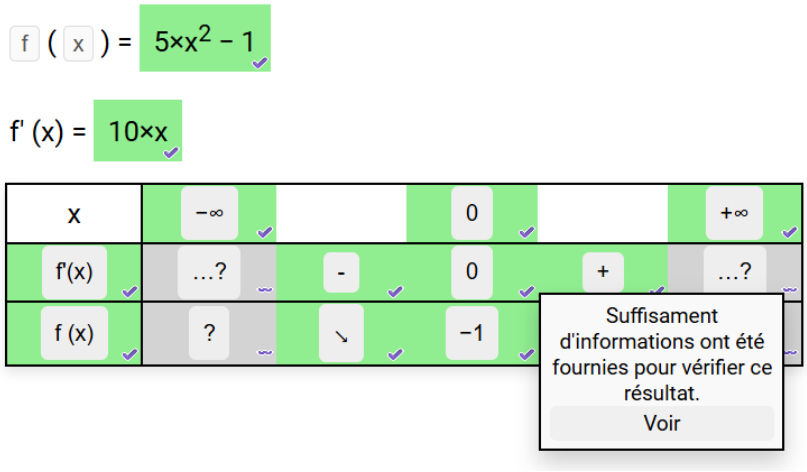
Perspectives

Ce stage a soulevé certains problèmes qui doivent être résolus dans des travaux futurs.

Premièrement, il est nécessaire d'implémenter des constructions supplémentaires dans l'arbre de preuve comme la disjonction de cas, le raisonnement par l'absurde ou le raisonnement par induction par exemple.

Ensuite, il est nécessaire de procéder à l'évaluation des besoins pour les tactiques `solve_equ` et `solve_trivially` comme est décrite dans la section 2.3, ainsi qu'effectuer une évaluation similaire pour déterminer la liste des théorèmes utilisables par les élèves. Ces deux tâches dépendant du public visé et du niveau d'étude, un questionnement sur les objectifs du projet s'impose.

5. Ce prototype est disponible à `https://mbodin1.gitlabpages.inria.fr/td-math-coq/interface.html`



(a) Tableau de variation interactif

```
Require Import Tabvar.

Definition f (x : real) :=
  5x^2-1.
Definition f' (x : real) :=
  10x.
Lemma positive_f__x_ : forall x,
  x > 0 ->
  f'(x) > 0 \/ f'(x) = 0.
Proof.
(* preuve à générer *)
Qed.
```

(b) Section textuelle Rocq interactive générée automatiquement après avoir appuyé sur ”voir”

FIGURE 3 – Prototype d'interface graphique pour les tableaux de variation

Bien que les constructions de l'inductif soient proches de celles des démonstrations papier-crayon, il reste fastidieux de traduire ces démonstrations en arbre de preuve Rocq à cause de la syntaxe lourde et des données supplémentaires nécessaires à stocker. Ainsi il semble utile de développer des tactiques permettant la résolution du but tout en construisant l'arbre de preuve. Ces tactiques correspondront directement aux constructions de l'arbre, ce qui a pour bénéfice d'être également utilisable par des élèves. Les logiciels Coq Waterproof et Verbose Lean pourront également servir de source d'inspiration.

Enfin, dans l'optique d'une utilisation par des élèves et au vu des difficultés rencontrées liées à la formalisation de l'analyse réelle en Rocq, il semble important de poser des notations et de fournir des définitions équivalentes pour certaines notions comme celle de limites afin de se rapprocher des définitions vues en cours par les élèves. Afin de garantir une comptabilité avec les centaines de théorèmes existants, il faut également fournir des lemmes d'équivalence avec les définitions actuelles.

Bibliographie

[1] Emilio Arias, Benoît Pin, and Pierre Jouvelot. jsCoq : Towards Hybrid Theorem Proving Interfaces. 01 2017. doi:10.48550/arXiv.1701.07125.

[2] Association des Professeurs de Mathématiques de l'Enseignement Public. Corrigé du baccalauréat S Métropole. https://www.apmep.fr/IMG/pdf/Correction_MetropoleS_20_juin_2013.pdf, 2013.

[3] Emmanuel Beffara, Martin Bodin, Nadine Mandran, and Rémi Molinier. Instrumentation de l'association de registres sémiotiques dans un assistant de preuve. In *EIAH2023 - 11ème Conférence sur les Environnements Informatiques pour l'Apprentissage Humain*, pages 1–5, Brest, France, Jun 2023. Association des Technologies de l'Information pour l'Éducation et la Formation.

[4] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot : A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science*, pages 41–62, 2015.

[5] M. Kerjean, F. Le Roux, P. Massot, M. Mayero, Z. Mesnil, S. Modeste, J. Narboux, and P. Rousselin. Utilisation des assistants de preuves pour l'enseignement en L1. *Gazette de la Société Mathématique de France*, 2022.

[6] Catherine Lelay and Guillaume Melquiond. Différentiabilité et intégrabilité en Coq. Application à la formule de d'Alembert. In *JFLA - Journées Francophone des Langages Applicatifs - 2012*, Carnac, France, Feb 2012.

[7] Patrick Massot. Verbose Lean 4. `https://github.com/PatrickMassot/verbose-lean4`, 2022.

[8] Patrick Massot. Teaching Mathematics Using Lean and Controlled Natural Language. In *International Conference on Interactive Theorem Proving - 2024*, Sep 2024.

[9] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, décembre 2001. URL : `http://www-lipn.univ-paris13.fr/~mayero/publis/these-mayero.ps.gz`.

[10] The Rocq Team. Ltac - Coq 8.19.2 documentation. `https://rocq-prover.org/doc/v8.19/refman/proof-engine/ltac.html`. Consulté le : 07/07/2025.

[11] David Tuin and Malcolm Vassallo. Coq Waterproof. `https://github.com/impermeable/coq-waterproof`, 2023.

A Contexte du stage

J'ai effectué mon stage au centre Inria de l'Université Grenoble Alpes au sein de l'équipe Spades (Programmation de systèmes embarqués sûrs et adaptatifs), bien que le domaine de recherche abordé ne corresponde pas au domaine de recherche de l'équipe.

J'ai été encadré par Martin Bodin, chercheur dans l'équipe Spades, et co-encadré par Emmanuel Beffara, maître de conférences à l'Université Grenoble Alpes et membre de l'équipe MeTAH au Laboratoire d'Informatique de Grenoble (LIG).

B Code source

Le code source concernant le type inductif Rocq développé et les fonctions de conversion et d'exploitation sont disponibles à `https://gitlab.inria.fr/mbodin1/td-math-coq/-/tree/stage-amaury/logiciels/ltac/ProofTree`.

Les fichiers ayant servi pour se familiariser avec Coquelicot et pour faire les premiers tests sont disponibles à `https://gitlab.inria.fr/mbodin1/td-math-coq/-/tree/stage-amaury/logiciels/ltac/Coquelicot`.

Le plugin OCaml développé pour la conversion de termes Rocq en string Rocq est disponible à `https://github.com/S-c-r-a-t-c-h-y/coq-string-of-term/tree/1.0/`.