Mémoire du Travail Encadré de Recherche : Lean et localisation de l'anneau

FRANCHI Charles

Octobre 2022 - Mai 2023

Table des matières

1 Liens vers la page Github

Ce mémoire concerne un projet mené sur Lean. Tous les fichiers créés et utilisés durant ce travail sont accessibles sur la page Github qui leur sont dédiés :
https://github.com/FRANCHI-Charles/TER

2 Introduction

Une problématique actuelle du domaine des mathématiques et de la recherche dans ce secteur est la véracité des nouveaux résultats trouvés. Avec plus de 12 000 pré-publications mathématiques chaque mois sur le site Arxiv.org depuis 2020[6], le secteur n'a jamais été aussi productif, et cela entraîne des énoncés nécessitant des compétences poussées et spécifiques pour leur compréhension. Par soucis d'effectif et de temps, les relectures de ces articles ne sont pas toujours irréprochables, et cela peu entraîner des erreurs dans les résultats publiés dans les revues actuelles.
Cela est un handicap considérable dans le monde des mathématiques, car chaque résultat repose toujours sur un autre qui le précède ; baser un théorème sur un lemme faux peut alors effondrer tout une théorie mathématique, ce qui n'est évidemment pas souhaitable.

Depuis une vingtaine d'années, une solution partielle mais efficace à ce problème a été trouvé dans le domaine de l'informatique, et précisément dans les assistants de preuves : ces logiciels sont avant tout des langages informatiques de programmation fonctionnelle, conçus dans l'optique de fournir un vérificateur de preuves mathématiques. Ainsi, en passant par un raisonnement formel poussé, basé sur les axiomes d'une théorie mathématique, nous sommes en mesure – dans la limite du logiciel – d'établir si un théorème est juste, ou non.
Cette solution, bien qu'efficace, demande généralement une approche non conventionnelle de la résolution d'un énoncé, et donc un certain apprentissage de l'utilisation dudit logiciel. Ceci n'est pas nécessairement une tâche aisée et ne semble pas accessible à tout le monde pour le moment. De plus, toute cette théorie s'appuie sur le fait que l'assistant utilisé soit non bugué, ce qui ne peut être prouvé dans l'absolu ; en revanche, des outils puissants de théorie mathématique, comme la théorie des types (voir 4.2), peuvent nous aider à assurer une cohérence des résultats et ainsi limiter les erreurs logicielles.

C'est en tout cas le choix qu'a fait le logiciel qui va nous intéresser : Lean. Lean est un projet d'assistant de preuves de Microsoft Research – un institut orienté principalement sur la recherche mathématique, informatique et technologique[2]. Démarré en 2013, ce projet en est aujourd'hui à sa 4ème version

- cependant, la version utilisé par la communauté est encore principalement la 3ème. C'est par ailleurs dans cette version qu'est codée Mathlib : une bibliothèque collaborative contenant les lemmes et définitions écrites dans ce langage, où chaque proposition contient une preuve rigoureuse. Cette bibliothèque grandit en permanence grâce à sa communauté active, permettant chaque jour de se rapprocher un peu plus d'une carte qui se veut complète des mathématiques actuelles.
Tout le monde peut participer à cette bibliothèque, en codant un résultat (respectant certaines lignes directrices) et le soumettant à la communauté via Zulip et Github.
A ce jour, plus de 45 000 définitions et 110 000 théorèmes ont été codé dans Lean[4].

C'est avec cette approche que j'ai démarré ce travail encadré de recherche. Mon objectif était donc de participer à l'édifice de Mathlib en codant de moi-même un résultat qui ne serait pas encore formalisé.
Mon encadrant, M. Filippo A. E. Nuccio, m'a aiguillé tout le long de ce projet, en m'introduisant Lean et me donnant des lignes d'objectifs claires pour ma progression. Rapidement - dès le mois de Novembre 2022 -, nous avons décidé de nous orienter sur les morphismes de la localisation des modules. Il reste encore de nombreux résultats qui n'ont pas été formalisés, dont celui qui va nous intéresser.

À travers ce mémoire, vous pourrez découvrir la démarche d'apprentissage qui a été effectuée afin de formaliser un résultat portant sur la localisation des modules.

3 Notions mathématiques

3.1 Module

Avant de s'attaquer à la localisation, parlons un peu de module. Considérons A un anneau unitaire - c'est à dire qui possède un élément neutre pour la multiplication. On dit que M est un A-Module à gauche si et seulement si $(M, +)$ est un groupe commutatif muni d'une loi multiplicative externe de $A \times M$ dans A vérifiant les propriétés suivantes :

$\forall (a, b) \in A, \forall (x, y) \in M,$

1. $(a + b).x = a.x + b.x$
2. $a.(x + y) = a.x + a.y$
3. $a.(b.x) = (ab).x$
4. $1.x = x$

Le plus simple est de penser le module comme la généralisation de l'espace vectoriel à un anneau au lieu d'un corps.
De même, on peut définir le morphisme de module ϕ entre deux A-Modules M et N en généralisant l'application linéaire :

$\forall a \in A, \forall (x, y) \in M,$

1. $\phi(x + y) = \phi(x) + \phi(y)$
2. $\phi(a.x) = a.\phi(x)$

On a rapidement $\phi(0) = 0$:

$$\begin{aligned} & \phi(0) = \phi(0 + 0) = \phi(0) + \phi(0) \\ \Leftrightarrow \quad & \phi(0) = 0 \end{aligned}$$

car N est en particulier un groupe.

3.2 Localisation

La localisation est la généralisation du corps des fractions à un anneau unitaire commutatif. Elle permet de rendre inversible les éléments d'une partie multiplicative d'un anneau, c'est à dire un sous-ensemble de l'anneau qui est stable par la multiplication, qui contient l'élément neutre 1 du produit et ne contient pas l'élément neutre 0 de l'addition.
Pour se faire, on utilise une partie multiplicative S de l'anneau unitaire commutatif A. On construit alors la relation d'équivalence suivante :

$$\begin{aligned} &\forall ((a_1, s_1), (a_2, s_2)) \in (A \times S)^2, \\ &\qquad (a_1, s_1) \sim (a_2, s_2) \Leftrightarrow \exists t \in S,\ t(a_1 s_2 - s_2 a_1) = 0 \end{aligned}$$

On écrit par la suite les classes d'équivalences de cette relation $\frac{a}{s}$. Le localisé de A par S, noté $S^{-1}A$ est alors un anneau construit avec l'addition et la multiplication suivante :

$$\begin{gathered} \forall ((a_1, s_1), (a_2, s_2)) \in (A \times S)^2, \\ \frac{a_1}{s_1} + \frac{a_2}{s_2} = \frac{a_1 s_2 + a_2 s_1}{s_1 s_2} \\ \frac{a_1}{s_1} \times \frac{a_2}{s_2} = \frac{a_1 a_2}{s_1 s_2} \end{gathered}$$

On vérifie facilement que ces lois de composition interne sont bien définis et que l'on obtient une structure d'anneau commutatif.

De plus, si A est intègre, on peut simplifier la relation d'équivalence :

$$\forall((a_1,s_1),(a_2,s_2)) \in (A\times S)^2, \quad (a_1,s_1)\sim(a_2,s_2) \Leftrightarrow a_1s_2 - s_2a_1 = 0$$

Cette hypothèse en plus, $S^{-1}A$ est le plus petit anneau commutatif rendant inversible tous les éléments de S, et on peut voir A comme une partie de $S^{-1}A$ avec l'injection canonique :

$$\begin{aligned} \phi : A &\to S^{-1}A \\ a &\mapsto \frac{a}{1} \end{aligned}$$

ϕ est bien injective :
Soit $(a_1,a_2) \in A^2$ tel que $\phi(a_1) = \phi(a_2)$. Alors :

$$\begin{aligned} \phi(a_1) = \phi(a_2) &\Leftrightarrow \frac{a_2}{1} = \frac{a_2}{1} \\ &\Leftrightarrow a_1 \times 1 - a_2 \times 1 = 0 \\ &\Leftrightarrow a_1 = a_2 \end{aligned}$$

Peu importe l'intégrité de A, on peut constater que,

$$\forall s \in S, \quad s \times \frac{1}{s} = \frac{s}{s} = \frac{1}{1} = 1$$

Car :

$$\forall s \in S,\ \forall t \in S, \quad t(1\times s - s \times 1) = 0$$

Remarque :

1. Si l'on prend $S = A^*$, alors $S^{-1}A$ est un corps, que l'on appelle *corps des fractions.*
2. Si A n'est pas intègre, deux éléments de A peuvent être envoyé sur le même élément de S.

En utilisant la même construction, on peut définir le localisé du Module M que l'on note $S^{-1}M$ en tant que $S^{-1}A$-Module à gauche :

$$\forall a \in A,\ \forall (s_1,s_2) \in S^2,\ \forall (m_1,m_2) \in M^2$$
$$\frac{m_1}{s_1} + \frac{m_2}{s_2} = \frac{s_2.m_1 + s_1.m_2}{s_1s_2}$$
$$\frac{a}{s_1}.\frac{m_2}{s_2} = \frac{a.m_2}{s_1s_2}$$

On peut encore une fois voir M comme une partie de $S^{-1}M$, en prenant garde que l'on passe d'un A-Module à un $S^{-1}A$-Module.

3.3 Résultats à formaliser

Avec ces notions en main, nous pouvons désormais nous concentrer sur le cœur mathématique de ce projet.
Soit A un anneau commutatif unitaire. Soient M et N deux A-Modules. Soit $f : M \to N$ un morphisme.
Alors on peut prolonger ce morphisme sur les localisés des modules via le morphisme :

$$\begin{aligned} S^{-1}f : S^{-1}M &\to S^{-1}N \\ \frac{m}{s} &\mapsto \frac{f(m)}{s} \end{aligned} \tag{1}$$

Démonstration : Tout d'abord, cette application est bien défini :
Soit $((m_1, s_1), (m_2, s_2)) \in (M \times S)^2$ tel que $\frac{m_2}{s_2} = \frac{m_1}{s_1}$. Alors :

$$\exists t_0 \in S,\ t_0(s_1 m_2 - s_2 m_1) = 0$$

Et donc :

$$\begin{aligned} & S^{-1}f\left(\frac{m_2}{s_2}\right) = S^{-1}f\left(\frac{m_1}{s_1}\right) \\ \Leftrightarrow\quad & \frac{f(m_1)}{s_1} = \frac{f(m_2)}{s_2} \\ \Leftrightarrow\quad & \exists t \in S,\ t(s_1 f(m_2) - s_2 f(m_1)) = 0 \\ \Leftrightarrow\quad & \exists t \in S,\ f(t(s_1 m_2 - s_2 m_1)) = 0 \end{aligned}$$

Si on utilise t_0, alors, comme f est un morphisme de A-Module :

$$f(t_0(m_2 s_1 - m_1 s_2)) = f(0) = 0$$

Donc $S^{-1}f$ est bien défini.

On vérifie ensuite les propriétés des lois de compositions :

$$\begin{aligned}
&\forall((m_1,s_1),(m_2,s_2)) \in (M\times S)^2,\\
&\quad S^{-1}f\left(\frac{m_1}{s_1}+\frac{m_2}{s_2}\right) = \frac{f(s_1.m_2+s_2.m_1)}{s_1s_2}\\
&\qquad\qquad = \frac{s_1.f(m_2)+s_2.f(m_1)}{s_1s_2}\\
&\qquad\qquad = \frac{f(m_1)}{s_1}+\frac{f(m_2)}{s_2}\\
&\qquad\qquad = S^{-1}f\left(\frac{m_1}{s_1}\right)+S^{-1}f\left(\frac{m_2}{s_2}\right)
\end{aligned}$$

$$\begin{aligned}
&\forall(m_1,s_1)\in M\times S,\ \forall(a_1,s_2)\in A\times S,\\
&\quad S^{-1}f\left(\frac{a_1}{s_2}.\frac{m_1}{s_1}\right) = \frac{f(a_1.m_1)}{s_1s_2}\\
&\qquad\qquad = \frac{a_1.f(m_1)}{s_1s_2}\\
&\qquad\qquad = \frac{a_1}{s_2}.\frac{f(m_1)}{s_1}\\
&\qquad\qquad = \frac{a_1}{s_2}.S^{-1}f\left(\frac{m_1}{s_1}\right)
\end{aligned}$$

On a également la propriété suivante :

$$\begin{aligned}
&\forall f: M\to N, \forall g: P\to M,\ M,N,PA\text{-Module},\\
&\qquad S^{-1}(f\circ g) = S^{-1}f\circ S^{-1}g
\end{aligned} \tag{2}$$

Démonstration :

$\forall f: M\to N, \forall g: P\to M,\ M,N,PA$-Module,
Soit $(p,s)\in P\times S$,

$$\begin{aligned}
S^{-1}f\circ S^{-1}g\left(\frac{p}{s}\right) &= S^{-1}f\left(\frac{g(p)}{s}\right)\\
&= \frac{f(g(p))}{s}\\
&= S^{-1}(f\circ g)\left(\frac{p}{s}\right)
\end{aligned}$$

L'objectif final était donc de coder ces deux résultats dans Mathlib.

3.4 Résultat intermédiaire

Afin de mieux appréhender la programmation d'un tel résultat, il avait été initialement prévu de travailler sur un résultat intermédiaire sur la localisation. Ce dernier n'a finalement pas été formalisé complètement, mais en voici tout de même les notions mathématiques.

Soient A et C deux anneaux commutatifs. Soit S une partie multiplicative de A. Soit B un anneau (commutatif) isomorphe au localisé $S^{-1}A$. On suppose de plus que B est une A-Algèbre, c'est à dire que B est un anneau et un A-Module. Soit f un morphisme de A dans C. On suppose que pour tout élément s de S, $f(s)$ est inversible.
Alors on peut prolonger f sur $S^{-1}A$:

$$\begin{aligned} \tilde{f} : S^{-1}A &\to C \\ \frac{a}{s} &\mapsto f(a) \times f(s)^{-1} \end{aligned}$$

Démonstration : On peut vérifier que $\tilde{f}$ est bien défini.
$\tilde{f}$ est bien un morphisme d'anneau :
Soient $((a_1, s_1), (a_2, s_2)) \in (A \times S)^2$.

$$\begin{aligned} \tilde{f}\left(\frac{a_1}{s_1} + \frac{a_2}{s_2}\right) &= f(a_1 s_2 + a_2 s_1) f(s_1 s_2)^{-1} \\ &= [f(a_1)f(s_2) + f(a_2)f(s_1)] f(s_1)^{-1} f(s_2)^{-1} \\ &= f(a_1)f(s_1)^{-1} + f(a_2)f(s_2)^{-1} \\ &= \tilde{f}\left(\frac{a_1}{s_1}\right) + \tilde{f}\left(\frac{a_2}{s_2}\right) \end{aligned}$$

Car $f(s_1)$ et $f(s_2)$ sont inversibles, donc l'inverse du produit est le produit des inverses.

$$\begin{aligned} \tilde{f}\left(\frac{a_1}{s_1} \times \frac{a_2}{s_2}\right) &= f(a_1)f(a_2)f(s_1)^{-1}f(s_2)^{-1} \\ &= \tilde{f}\left(\frac{a_1}{s_1}\right) \times \tilde{f}\left(\frac{a_2}{s_2}\right) \end{aligned}$$

Par commutativité de C.

$$\begin{aligned} \tilde{f}\left(\frac{1}{1}\right) &= f(1)f(1)^{-1} \\ &= 1 \end{aligned}$$

Par définition d'un inverse.
Enfin, c'est bien un prolongement de f :

$$\forall a \in A$$

$$\begin{aligned}\tilde{f}\left(\frac{a}{1}\right) &= f(a)f(1)^{-1} \\ &= f(a) \times (1^{-1}) \\ &= f(a) \times 1 \\ &= f(a)\end{aligned}$$

Vu que B est isomoprphe à $S^{-1}A$, si on note $\phi : B \to S^{-1}A$ l'isomorphisme associé, alors on peut voir ce prolongement sur B comme étant l'application $\tilde{f} \circ \phi$.

4 Travail préliminaire sur Lean

4.1 Phase de découverte et premiers exercices

J'ai commencé mes premiers pas à travers Lean via les exercices mis à disposition sur le GitHub de mon encadrant. Ceux-ci sont répartis en 3 fichiers :

- Le fichier A.lean porte sur la logique de base de Lean, le fonctionnement qui découle de la théorie des types (Voir 4.2).
- Le fichier B.lean concerne les fonctions, et quelques propriétés simples à formaliser.
- Le dernier fichier C.lean permet de jongler avec quelques théorèmes et définitions de la limite dans le corps réel. Ce dernier m'a également permis de faire mes premiers pas dans Mathlib, grâce à certains lemmes basiques de commutativité ou d'associativité, entre autres.

À travers ces trois exercices, j'ai pu me familiariser avec Visual Studio Code, l'éditeur de code que j'ai par la suite utilisé pour tout le projet, mais aussi avec Lean et son fonctionnement atypique : ce langage de programmation a comme particularité de nous dire directement si notre formulation possède des erreurs à chaque ligne de code, en positionnant notre curseur à l'endroit désiré. Ainsi, cela permet de savoir en direct si l'énoncé de notre théorème possède des erreurs de syntaxes ou si notre démonstration est fausse.

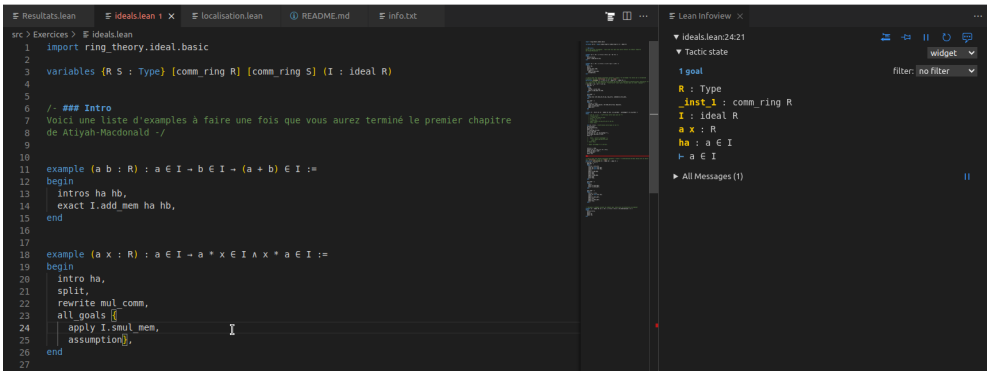

FIGURE 1 – Exemple de retour du langage dans un programme (partie de droite).

De plus, la manière de démontrer un théorème est dans le sens inverse de celui conventionnelle : avec Lean, il faut partir du résultat, puis chercher à rejoindre les hypothèses. Par exemple, si on veut montrer que $P \implies Q$, et que nous savons que $P \implies S$ et $S \implies Q$, au lieu de formuler "Si j'ai P, alors j'ai S, donc j'ai Q", nous allons nous dire "Pour avoir Q, il suffirait d'avoir S, et donc il suffirait d'avoir P, que j'ai par hypothèse".

La partie mathématiques de ces exercices, quant à elle, n'était pas sujette à apprentissage, car cela reposait essentiellement sur des résultats de 1ère année de licence que je connaissais.

4.2 Quelques mots sur la théorie des types

Pour bien maîtriser Lean, il est important de comprendre que la théorie mathématique qu'il utilise n'est pas celle que nous connaissons communément : nous utilisons la théorie des ensembles, Lean utilise la théorie des types.
Il a vite été nécessaire de se renseigner plus précisément sur quelques points importants de cette théorie[5], dont en voici quelques préceptes.

La théorie des types est un autre système d'axiomes fondamentaux pour construire les mathématiques. Il diffère en beaucoup de points de la théorie des ensembles, et notamment sur sa construction. Ici, nous n'avons pas d'ensembles à proprement parler, mais des types. Un type et une catégorie d'éléments, contenant des termes. Chaque terme ne peut avoir qu'un et un seul type, contrairement aux éléments d'un ensemble qui peuvent en appartenir à plusieurs [1]. On note alors un terme a de Type A "a : A"
Chaque Type est lui même un terme d'un type *plus gros*. On hiérarchise alors

1. Par exemple, 1 est un nombre entier et un nombre réel, alors que dans la théorie des types, le 1 de type Entier Naturel est différent du 1 de Type Réel.

les types en les regardant par rapport au type duquel ils en sont les termes. Notamment, on peut considérer les Propositions comme étant les termes du Type Prop. Ainsi, on a "$\{4 \leq 5\}$: Prop". Chaque proposition possède un unique terme : ce terme est comme la valeur de vérité de cette proposition, c'est-à-dire le fait qu'elle soit juste ou fausse. Donc, il existe un unique terme d tel que "d : $\{4 \leq 5\}$". En revanche, le type $\{5 \leq 4\}$ est *vide* et de part ce fait, ce type n'existe pas vraiment.
Ce fonctionnement sur les propositions est une des raisons principales de pourquoi cette théorie est plus forte pour faire des mathématiques formelles - il est par essence même de sa construction impossible de placer dans le code une proposition fausse, ce qui permet plus facilement de voir certaines erreurs.

Tout le but d'une démonstration dans Lean est donc de créer un terme de la proposition que l'on veut démontrer. Autrement dit, lorsque l'on écrit dans Lean un théorème, on insinue qu'avec les hypothèses données, on peut créer un terme du type "le résultat voulu", et on le construit ensuite (voir l'exemple de 5.2).

4.3 Les tactiques dans Lean

Pour fabriquer ce fameux terme qui permet de prouver un lemme ou un théorème, j'utilisais dans lean ce qu'on appelle le *Tactic mode*. Introduit par un `begin` et se clôturant par un `end`, ce mode consiste à construire une preuve en utilisant des fonctions semblables aux étapes d'une démonstration à la main. Voici une explication des principales fonctions que j'ai utilisées durant ce TER.

La fonction `intro [h]` ou `intros [h1, h2, ...]` permet d'introduire une ou plusieurs hypothèse(s). Par exemple, si l'on doit prouver $P \implies Q$, cela permet de supposer P (donc de créer `h : P`).

La fonction `exact [h]` permet de dire que le résultat attendu est exactement l'hypothèse `h` et donc de conclure la démonstration. `assumption` permet de ne pas préciser quel hypothèse on utilise, on indique juste qu'elle est dans les hypothèses établies.

La fonction `apply [h]` permet d'utiliser tout résultat de la forme $P \implies Q$ sur le résultat à démontrer. Par exemple, si le résultat à prouver est Q, faire `apply h : P => Q` transformera le résultat à prouver en P [2].

2. Pour comprendre le raisonnement, voir 4.1

La fonction `rewrite {<-} [h]` permet d'utiliser des résultats de la forme $P = Q$ ou $P \Leftrightarrow Q$, que ce soit dans un sens (P devient Q) ou dans l'autre (en utilisant la flèche : Q devient P). Par exemple, si le résultat à prouver est `a + 1 = b`, alors
`rw`[3] `(h : \forall (x y : X), x + 1 = y <=> x = y - 1)` transformera ce résultat en `a = b - 1`.

La fonction `simp` permet de demander à Lean de trouver tout seul d'éventuels lemmes fondamentaux afin de prouver le résultat espérer. Cela peut être par exemple $a * 0 = 0$ ou $[a + b = a \Leftrightarrow b = 0]$.

La fonction `have [h] : [lemme]` permet de créer un nouveau résultat à démontrer `lemme` qui devient une hypothèse pour le résultat initial. Il faut alors ensuite prouver ce nouveau lemme.

La fonction `use [h]` permet d'utiliser le terme `h` sur le résultat. Généralement, cela consiste à utiliser `h` comme terme satisfaisant un "∃" du résultat à démontrer.

La fonction `cases [h1] with [h2, h3, ...]` permet de séparer une hypothèse en plusieurs hypothèses distinctes. Par exemple, `cases (h : P et Q) with h1 h2` renvoie `h1 : P` et `h2 : Q`. Cela permet aussi de récupérer le terme satisfaisant l'existence d'une hypothèse : `cases (h : \exist (u : U), P(u) with h1 h2` renvoie `h1 : U` et `h2 : P(h1)`.

La fonction `specialize [h1] with [h2]` permet de spécifier une proposition générale pour un terme précis. Généralement, cela consiste à appliquer un "∀" à un élément donné `h2`.

5 Dans le cœur du sujet : méthode, exercices et résultats

5.1 Méthode de résolution

Grâce à ces nouvelles notions, il a été par la suite plus simple de naviguer dans Lean. Il restait tout de même une grande partie du travail de compréhension et avec cela une grande partie des complications : la bibliothèque Mathlib. Celle-ci possède une architecture nécessairement très formelle, ce qui peut rendre sa compréhension difficile.
Pour écrire le code nécessaire afin de résoudre un exercice, j'ai dû adopter une méthode de travail efficace afin de ne pas me perdre dans la librairie. Elle

3. Abréger de *rewrite*

s'est mise en place peu à peu jusqu'à être relativement efficace vers la fin de ce TER. La suite de cette sous-section vous la présente, en étant accompagné d'un exemple (dont le code original se trouve ici).

La première étape est d'analyser le résultat pour le comprendre et de le formaliser mathématiquement. Prenons l'exemple suivant :

```
variables {A : Type*} [comm_ring A]

lemma prod_units (u v : A) :
    is_unit u -> is_unit v -> is_unit (u * v) :=
```

La première chose à faire est de comprendre les fonctions utilisées. Je regarde alors ce que veut dire `is_unit` :

```
def is_unit [monoid M] (a : M) : Prop
    := \exist u : M\^x, (u : M) = a
```

En allant voir la défintion de `M\^x`, on constate que ce sont les éléments inversibles de M, un Monoid, c'est à dire un ensemble contenant une loi de composition interne et un élément neutre. En particulier, cela signifie que `is_unit u` traduit le fait que u est une unité de A, c'est-à-dire qu'il possède un inverse.
Donc le lemme dit la chose suivante : Soit A un anneau commutatif. Alors $\forall (u, v) \in A$, u est une unité $\implies$ [v est une unité $\implies$ $u * v$ est une unité].

La deuxième étape consiste à résoudre mathématiquement le lemme.
Ici : soient u et v deux éléments de A. On suppose que u est une unité. Montrons que [v est une unité $\implies$ $u * v$ est une unité]. Soit v une unité. Soient u^{-1} et v^{-1} les inverses respectifs de u et v. Alors $(u*v)*v^{-1}*u^{-1} = 1$. Donc $u * v$ est une unité.

La troisième étape est souvent la plus dure : il faut chercher les fonctions utiles et les résultats déjà coder dans Mathlib afin de pouvoir résoudre facilement le théorème. Cela permet de savoir à quel point la preuve doit être "détaillée" dans le code.
Par exemple ici, est-ce qu'il faudra que je prouve que si u et v sont inversibles, alors $u*v$ sont inversibles ou est-ce qu'il existe déjà un lemme pour le prouver ? Si non, comment est ce que j'exprime u^{-1} dans Lean ? Quel lemme me dira que $u * u^{-1} = 1$?
Ce cas est assez simple, car le lemme `is_unit.mul_iff` permet exactement l'équivalence souhaitée dans la première question.

Finalement, on peut construire la preuve souhaitée :

```
lemma prod_units (u v : A) :
    is_unit u → is_unit v → is_unit (u * v) :=
begin
  intros hu hv,                  (1)
  simp only [is_unit.mul_iff],   (2)
  split,                         (3)
  exact hu,
  exact hv,                      (4)
end
```

1. On émet les hypothèses d'unités sur u et v que l'on appelle respectivement hu et hv.
2. On applique le lemme présenté plus tôt [4]. On veut désormais prouvé que u et v sont des unités.
3. On sépare le but en deux objectifs distincts (on passe de [u est une unité et v est une unité] à [u est une unité], [v est une unité]).
4. On dit que c'est exactement nos hypothèses de départ.

5.2 Idéaux, localisation et exercices

Afin de mieux s'approprier Mathlib avant de HERE, j'ai effectué des exercices s'approchant peu à peu du résultat à formaliser proposés par mon encadrant. Nous avons donc formalisé des résultats sur les idéaux, qui sont en soit un cas particulier de module [5]. L'objectif étant vraiment concentrer sur Mathlib, les résultats mathématiques en eux-mêmes étaient encore une fois assez accessible. Pour ce faire, j'ai du me renseigner au préalable sur les idéaux grâce à quelques livres d'algèbre[1][3] : définitions de bases, propriétés essentielles comme les idéaux premiers ou maximaux, quelques exercices sur papier, ... C'est également dans ces derniers que j'ai découvert le résultat que je devais formuler (voir 3.3).
Pour illustrer mon propos, voici un résultat détaillé de ces exercices [6] :

```
variables {R : Type} [comm_ring R] (I : ideal R)

example (a x : R) : a \in I -> a * x \in I \and x * a \in I :=
begin
  intro ha,              (1)
```

4. *simp only* agit dans ce cas comme un *rewrite* (voir 4.3)
5. On peut définir un idéal I d'un anneau A comme étant un sous-ensemble de A et un A-module pour la multiplication interne.
6. Vous pouvez retrouver le code original ici.

```
  split,                (2)
  rewrite mul_comm,     (3)
  all_goals {
    apply I.smul_mem,   (4)
    assumption},
end
```

Ce code dit la chose suivante : soient R un anneau commutatif et I un idéal de R. Alors :

$$\forall a \in R,\ \forall x \in R,\ [a \in I \implies a \times x \in I \text{ et } x \times a \in I]$$

On résout alors :

1. On suppose que $a \in I$;
2. On sépare le résultat attendu en 2 : $a \times x \in I$ et $x \times a \in I$;
3. On applique le fait que la multiplication est commutative dans A à la première partie du résultat : on obtient alors que $x \times a = a \times x$, donc les deux parties du résultat sont identiques.
4. Pour les deux parties (`all\_goals`), on utilise le fait que $x \times a$ est dans I si a est dans I (`smul\_mem`), et on précise que cela est dans nos hypothèses (`assumption`).

Ce travail m'a donc obligé à fouiller plus intensément dans des lemmes précis liés aux idéaux, et donc de lire en détails ce qui a déjà été codé dans Mathlib dans le fichier dédié.

Après les idéaux, nous nous sommes concentrés sur la localisation avec la même procédure, et les mêmes livres d'algèbre[1][3].
À la fin de ce travail et de ce fichier, j'ai travaillé sur un résultat intermédiaire (voir 3.4) afin d'être au plus proche du résultat à démontrer et d'être vraiment paré pour le formaliser.

5.3 Résultats finaux

Après tout ce travail, je disposais enfin de tous les outils pour essayer de formaliser les théorèmes voulus [7]. Après plusieurs essais, nous avons réussi à créer une formalisation valide de ces résultats que vous pouvez retrouver sur le repository.

```
import algebra.module.localized_module
import algebra.module.linear_map                    (1)
```

7. Pour la démonstration mathématique, voir 3.3.

```
noncomputable theory

open localized_module


variables {R : Type*} [comm_semiring R] (S : submonoid R)
variables {M : Type*} [add_comm_monoid M] [module R M]
variables {N : Type*} [add_comm_monoid N] [module R N]          (2)
variables {P : Type*} [add_comm_monoid P] [module R P]



lemma mk_wd (f : M →+[R] N) : \forall (p p' : M × S), p \eq p'
    → mk (f p.1) (p.2) = mk (f p'.1) (p'.2):=                   (3)
begin
  intros p p' h,

  have h2: mk p.1 p.2 = mk p'.1 p'.2                            (4)
    → mk (f p.1) (p.2) = mk (f p'.1) (p'.2),
  intro hr,
  rw mk_eq at |- hr,                                            (5)
  obtain ⟨u, hu⟩ := hr,
  use u,
  simp only [submonoid.smul_def,
    ← distrib_mul_action_hom.map_smul] at hu |-,
  rw hu,

  apply h2,
  simp only [localized_module.mk, prod.mk.eta, h, quotient.eq],
end


def extended (f : M →+[R] N) : (localized_module S M)
        →+[localization S] (localized_module S N) :=
{ to_fun := \lambda p1, lift_on p1 (\lambda x,
        mk (f(x.1)) (x.2)) (mk_wd S f),                         (6)
  map_smul' :=                                                  (7)
  begin
    intros m x,
    induction x using localized_module.induction_on with a s,   (8)
    induction m using localization.induction_on with m,
    rw mk_smul_mk,
    repeat {rw lift_on_mk},
    rw mk_smul_mk,
    simp only [map_smul],
  end,

  map_zero' :=                                                  (9)
  begin
```

```
    rw ← zero_mk 1,
    rw lift_on_mk,
    rw map_zero,
    exact zero_mk 1,
  end,

  map_add' :=                                                    (10)
  begin
    intros x y,
    induction x using localized_module.induction_on with a s1,
    induction y using localized_module.induction_on with b s2,
    rw mk_add_mk,
    repeat {rw lift_on_mk},
    rw mk_add_mk,
    simp only [map_add, submonoid.smul_def,
        ← distrib_mul_action_hom.map_smul],
  end }

/--Composition of two extended functions.-/
lemma extended_comp (f : N →+[R] P) (g : M →+[R] N) :
  extended S (f.comp g) = (extended S f).comp (extended S g) :=
begin
  apply distrib_mul_action_hom.ext,
  intro x,
  induction x using localized_module.induction_on with a s,
  refl,                                                          (11)
end
```

Quelques précisions

1. On importe les parties de la bibliothèque dans lesquelles nous allons directement travailler.
2. On crée les espaces que nous allons utilisés par la suite.
3. Ce lemme est un préambule qui sert à montrer que $\tilde{f}$ est bien défini. `mk m s` est l'élément $\frac{m}{s}$.
4. On introduit un résultat plus simple pour pouvoir travailler avec `mk` au lieu de l'équivalent (`\eq`), qui est plus difficile à utiliser de manière formelle.
5. `at \- hr` permet d'appliquer le `rewrite` à l'hypothèse et non au résultat.
6. `to_fun` est la définition de la fonction prolongé. Il est plus facile pour la compréhension, même si légèrement incorrecte, de voir `\lambda` comme étant un "∀".
7. `map_smul'` est le fait que $\tilde{f}$ est bien linéaire pour la multiplication externe.

8. Cette commande permet de voir tout élément du localisé de S comme étant de la forme $\frac{m}{s}$. `induction` a d'autres utilités de manières générales, mais cela n'est pas nécessaire pour comprendre ce code.
9. `map_zero'` est le fait que $\tilde{f}(0) = 0$. Cela est nécessaire de le prouver car nous sommes dans le cas d'un monoid et non d'un groupe.
10. `map_add'` est le fait que $\tilde{f}$ est bien linéaire pour l'addition.
11. Dans ce deuxième résultat, `refl` permet de préciser que l'on utilise ici la partie `to_fun` de `extended`, autrement dit on utilise la définition de la fonction et pas les propriétés qui lui sont associées.

5.4 Difficultés rencontrées

Cette aventure a été pour moi bien plus compliquée que je ne l'aurais imaginé. Tout au long de ce travail, j'ai rencontré de nombreuses difficultés qui ont grandement freinées ma progression, parfois durant de longues périodes.

Ma principale difficulté a été de naviguer dans Mathlib. Comprendre cette librairie n'a pas été du tout instinctif pour moi : je n'arrivais pas toujours à comprendre les chemins empruntés dans les démonstrations, voir même les résultats annoncés.
La raison principale à cette difficulté a été le manque de connaissance de Lean. Chaque théorème m'introduisait toujours une nouvelle fonction, tactic ou même syntaxe que je ne comprenais pas. Cela rendait se travail vraiment épuisant.
Je passais donc souvent l'entièreté de mon temps à chercher la signification des résultats et tactics sur l'API documentation de Lean Community[4], des éventuels méthodes de résolution sur la page Zulip de la communauté qui aurait déjà été discuté, ou même des idées de réponse en demandant à Chat-GPT [8]. En dernier recours, je demandais de l'aide à mon encadrant, qui m'aiguillait sur la voie à suivre.

J'ai par exemple perdu beaucoup de temps a vraiment cerner comment fonctionne les ensembles en théorie des types, et comment les utiliser : comme en théorie des types, chaque terme possède un unique Type, cela entraîne des complication sur des choses qui nous paraisse primordiale. Supposons que j'ai `a : A`, l'hypothèse `h : a in S` et $S \subset A$. Comme j'ai `a : A`, impossible d'avoir `a : S`. Comment faire pour avoir un terme de type `S` ? La solution réside dans le fait qu'il ne faut pas chercher à construire `a : S`, vu que cela est impossible, mais bien un nouveau terme de type `S`. En effet, ici, on peut construire un nouveau terme de la façon suivante : `have b : S := <a, h>`. Ce terme est issu de a, mais n'est pas a. Cependant, il se comportera de la même manière.

8. Vous pouvez trouver un code travaillé en partie grâce à cet outil sur ma page GitHub.

Cette imperméabilité de compréhension m'a amenée quelques fois à résoudre des exercices "à l'aveugle". Dans localisation.lean, j'ai résolu quelques lemmes à l'instinct, sans les comprendre, tant la définition de ce que j'utilisais m'était abstraite. J'arrivais, en voyant le résultat à prouver, à comprendre quel tactic et lemme de Mathlib je devais utiliser sans pour autant avoir compris ce que je disais mathématiquement parlant.
Ce n'est qu'après discussion avec mon encadrant que tout cela s'est éclaircit.

Petit à petit, ma méthode de travail s'améliorant et mes connaissances grandissant, ces difficultés se sont logiquement tassées. Aujourd'hui, je me sens beaucoup plus libre dans Mathlib et je pense avoir cerné les grandes lignes de son fonctionnement.

6 Conclusion

Ce travail m'a beaucoup appris sur la difficulté d'un exercice aussi intense que la recherche. L'autonomie nécessaire pour la résolution de celui-ci m'a fait réaliser l'importance d'avoir des bases solides dans un domaine avant de pouvoir y contribuer, et à quel point acquérir ces bases demandait de la rigueur et surtout de la pratique.
Je suis de plus très satisfait d'avoir réussi à formaliser moi-même ces résultats, avec un peu d'aide de mon encadrant tout de même. J'ai finalement, après toute cette année de travail, réussit à trouver de moi-même l'essentiel des lemmes me permettant de conclure, même si cela aura coûté cher à ma motivation !
Il restera à soumettre ces résultats à la communauté dans les semaines qui suivent afin qu'ils soient définitivement validés, pour avoir le plaisir de savoir avoir contribué à mon échelle à l'édifice des mathématiques fondamentales, dans l'espoir qu'un jour elle puisse définitivement permettre la disparition des publications contenant des erreurs.

Désormais, dans la continuité de mon parcours, j'espère pouvoir accéder à la voie de l'intelligence artificielle afin de mettre toujours plus l'informatique au service des mathématiques, pour faciliter la tâche des chercheurs et nous aider à trouver et démontrer des théorèmes de plus en plus complexes.

Références

[1] Michael Atiyah. *Introduction To Commutative Algebra*. Crc Press, 2019.

[2] Skinny Bones et. *Lean*. Lean, 2023. URL : https://leanprover.github.io/ (visité le 29/03/2023).

[3] N. Bourbaki. *Algèbre commutative*. Springer Science & Business Media, mai 2007.

[4] *Lean community*. leanprover-community.github.io. URL : https://leanprover-community.github.io/ (visité le 29/03/2023).

[5] R P Nederpelt et Herman Geuvers. *Type theory and formal proof : an introduction*. Cambridge University Press, 2014.

[6] Cornell University. *Monthly Submissions*. arxiv.org, mars 2023. URL : https://arxiv.org/stats/monthly_submissions (visité le 29/03/2023).