



INSTITUT
POLYTECHNIQUE
DE PARIS

**École Nationale des Ponts et Chaussées**

2023-2024

# Projet de Fin d'Études

BASTIEN LE CHENADEC

ÉLÈVE INGÉNIEUR DU DÉPARTEMENT IMI

MASTER MATHÉMATIQUES VISION APPRENTISSAGE

GÉNÉRATION DE DONNÉES SYNTHÉTIQUES AVEC UN ASSISTANT DE PREUVE POUR L'ENTRAÎNEMENT DE MODÈLES DE LANGAGE

Stage réalisé au sein de JP MORGAN
14 Place Vendôme, 75 001 Paris
8 Avril 2024 - 30 Septembre 2024
MAÎTRE DE STAGE : DR. NELSON VADORI

# Table des matières



1. Note : ce rapport se limite à 40 pages pour se conformer aux consignes de l'École des Ponts.

# Table des figures

# Remerciements

Je tiens à remercier chaleureusement mes collègues au sein de JP Morgan pour leur accueil et leur soutien tout au long de mon stage. Je remercie particulièrement mon maître de stage, Nelson Vadori, pour son accompagnement et ses conseils. Je remercie également Ahmad Rammal, également stagiaire, pour notre collaboration fructueuse. Merci aussi à Antoine Gorceix et Mathieu Sibue pour leurs conseils précieux. Merci enfin à M. Pascal Monasse, responsable de stage, pour son suivi attentif.

Cordialement,

Bastien Le Chenadec

# Résumé

Mots-clés : Raisonnement mathématique, Assistant de preuve, LLM, Données synthétiques

Ce rapport de stage s'inscrit dans le cadre du Master 2 Mathématiques, Vision et Apprentissage à l'ENS Paris-Saclay, en collaboration avec l'entreprise JP Morgan au sein de leur département de recherche en Intelligence Artificielle (IA). L'objectif principal de ce travail était d'utiliser des modèles de langage (LLM) pour des tâches de raisonnement mathématique. On s'est en particulier intéressé à l'intégration d'assistants de preuve avec les LLM, pour améliorer leur capacité à raisonner de manière formelle.

Le rapport explore d'abord les fondements théoriques des assistants de preuve, leur rôle dans la formalisation des preuves mathématiques, ainsi que leur intégration avec l'IA. Le projet a consisté à générer des données synthétiques via l'assistant de preuve Lean, afin d'entraîner des LLM à raisonner de manière plus rigoureuse et formelle. Le processus de génération des données, la mise en œuvre technique et les outils utilisés sont décrits en détail, avec un accent particulier sur la métaprogrammation en Lean.

Les résultats obtenus sont encourageants, avec la mise en place d'un système innovant de génération de données synthétiques. De nombreuses pistes d'amélioration ont été identifiées, pour pleinement exploiter le potentiel de cette méthode. Ce stage a permis de développer des compétences pointues en programmation fonctionnelle et en métaprogrammation, tout en explorant des synergies entre l'IA et les assistants de preuve.

# Abstract

Keywords : Mathematical reasoning, Proof assistant, LLM, Synthetic data

This internship report is part of the Master's program in Mathematics, Vision, and Learning at ENS Paris-Saclay, in collaboration with JP Morgan within their Artificial Intelligence (AI) research department. The main objective of this work was to use large language models (LLMs) for mathematical reasoning tasks. Specifically, we focused on integrating proof assistants with LLMs to enhance their ability to reason in a formal manner.

The report first explores the theoretical foundations of proof assistants, their role in the formalization of mathematical proofs, and their integration with AI. The project involved generating synthetic data using the Lean proof assistant to train LLMs to reason more rigorously and formally. The data generation process, technical implementation, and the tools used are described in detail, with particular emphasis on metaprogramming in Lean.

The results obtained are promising, with the development of an innovative synthetic data generation system. Several areas for improvement have been identified to fully exploit the potential of this approach. This internship provided the opportunity to develop advanced skills in functional programming and metaprogramming while exploring synergies between AI and proof assistants.

# Introduction

*Ce projet de fin d'études s'inscrit dans le cadre du Master 2 Mathématiques, Vision et Apprentissage de l'École Normale Supérieure Paris-Saclay. Il a été réalisé au sein de l'entreprise JP Morgan, dans le département de recherche en intelligence artificielle (IA).*

Le développement de l'IA a permis des avancées considérables dans divers domaines, en particulier dans le traitement du langage naturel. Une application prometteuse de ces technologies est l'utilisation de grands modèles de langage (LLMs) pour la résolution de problèmes mathématiques. Il est en effet naturel de penser que ces modèles, capables de générer du texte de manière cohérente et pertinente, pourraient être utilisés pour raisonner sur des énoncés mathématiques. Cependant, ils se révèlent souvent inadaptés pour le raisonnement mathématique, car ils se contentent de prédire le token le plus probable en fonction du texte qui précède. Ils sont donc sujets à de nombreuses erreurs et hallucinations, et ne garantissent pas un raisonnement correct. Pour résoudre des problèmes mathématiques, il est nécessaire de structurer le raisonnement, de vérifier la validité des étapes intermédiaires, et bien d'autres aspects qui ne sont pas pris en compte par les LLMs.

Une première partie de ce projet de fin d'études a porté sur l'enseignement de règles mathématiques fondamentales à un LLM. L'objectif était de lui apprendre des opérations de base en langage naturel, telles que la résolution d'équations simples, le calcul, les règles de manipulation dans un anneau commutatif, etc. On a mis l'accent sur la quantification de la réutilisation de ces règles par le modèle dans le cadre de la résolution de problèmes plus complexes. Ce projet nous a notamment poussés à implémenter notre propre système de calcul formel, basé sur une représentation des expressions mathématiques en arbres. Ce travail a été réalisé en collaboration avec l'équipe de recherche de JP Morgan, et a mené à la rédaction d'un article soumis à NeurIPS. Bien que ce projet ait occupé la moitié de notre temps, nous ne le détaillerons pas davantage dans ce rapport par manque de place. À la place, nous présentons un second projet plus personnel qui a été mené en parallèle.

Face aux limites des LLMs pour le raisonnement mathématique, une autre piste de recherche est l'utilisation d'assistants de preuve. Ces outils, traditionnellement employés pour vérifier la validité de preuves mathématiques formelles, sont aujourd'hui explorés

comme un levier pour l'IA dans l'apprentissage et la résolution de problèmes complexes. Leur principal avantage est de garantir la validité des preuves produites, en s'appuyant sur des fondements logiques solides. Au-delà de cette simple vérification, ces outils ouvrent la voie à des méthodes de recherche de preuve plus sophistiquées, telles que la recherche arborescente monte-carlo (MCTS). D'autre part, les assistants de preuve ont vocation à assister les mathématiciens dans leur travail, en automatisant certaines tâches fastidieuses ou en donnant des indications sur l'état d'avancement d'une preuve. Ces éléments peuvent également être exploités pour améliorer les performances des modèles de langage en raisonnement mathématique.

Dans ce contexte, le présent rapport s'intéresse à l'interaction entre les assistants de preuve et l'IA, et plus précisément à la manière dont ils peuvent être utilisés pour améliorer les performances des modèles de langage en raisonnement mathématique. Précisément, nous nous intéresserons à la génération de données synthétiques à partir d'un assistant de preuve, et à la manière dont ces données peuvent être utilisées pour entraîner des modèles de langage à raisonner de manière formelle. Nous motivons cette approche par le manque de données existantes pour l'entraînement de tels modèles.

Le rapport commence par une revue de la littérature qui couvre les concepts clés liés aux assistants de preuve et à la théorie des types. Nous y détaillons également les contributions récentes qui intègrent l'IA et les assistants de preuve, notamment dans le domaine de la recherche de preuve automatisée avec des méthodes inspirées de MCTS et *AlphaZero*, initialement développés pour les jeux stratégiques. Ce contexte est essentiel pour comprendre les enjeux de notre projet et les perspectives qu'il ouvre.

Dans la seconde section, nous présentons notre approche de génération de données synthétiques à partir d'un assistant de preuve. Nous détaillons les différentes étapes de ce processus, de l'extraction de données brutes à la génération de données utilisables pour l'entraînement de modèles de langage. Nous discutons également des choix de conception et des outils utilisés pour mettre en œuvre cette méthode. Enfin, les résultats de notre travail sont présentés et discutés.

Bonne lecture !

# Présentation de l'organisme d'accueil - JP Morgan

JP Morgan est une institution financière de renommée mondiale, filiale de JPMorgan Chase & Co. Le groupe JPMorgan Chase & Co est séparé entre les activités de banque d'investissement (JP Morgan) et de banque de détail (Chase). Fondée en 1871, JP Morgan est aujourd'hui l'une des plus grandes banques d'investissement et institutions financières au monde. L'entreprise est présente dans plus de 100 pays et emploie environ 25 000 personnes. À Paris, JP Morgan dispose d'un bureau stratégique qui joue un rôle clé dans les opérations européennes de la banque.

JP Morgan a une longue histoire de croissance et d'innovation. Depuis sa création, la banque s'est adaptée aux évolutions des marchés financiers et a joué un rôle central dans de nombreuses transactions historiques. Aujourd'hui, JP Morgan est un acteur majeur dans les domaines de la banque d'investissement, de la gestion d'actifs, de la gestion de patrimoine et des services financiers pour les entreprises et les particuliers.

Le marché des services financiers est extrêmement compétitif, avec des acteurs de renom tels que Goldman Sachs, Morgan Stanley, et Deutsche Bank parmi les principaux concurrents de JP Morgan. Chaque institution a ses propres spécificités et met en avant ses points forts. Pour JP Morgan, ce sont le service client, l'excellence opérationnelle et l'intégrité.

Au sein de JP Morgan, l'entité *AI Research* est un département de recherche dédié à l'IA et à l'apprentissage automatique. L'équipe est composée de chercheurs, d'ingénieurs et de développeurs spécialisés dans les domaines de l'apprentissage automatique, de l'analyse de données et de l'optimisation. Ses objectifs sont duals : d'une part, développer des modèles de pointe pour des applications au sein de JP Morgan, et d'autre part, contribuer à la recherche académique en publiant des articles et en participant à des conférences.

Le département *AI Research* regroupe une centaine de chercheurs, en majorité à New York, avec de plus petites équipes à Londres, Paris, Singapour et Madrid. Il est dirigé par Manuela Veloso, une chercheuse de renom dans le domaine de l'IA, qui a notamment travaillé en robotique et dirigé le département d'apprentissage automatique de l'univer-

sité Carnegie Mellon à Pittsburgh aux États-Unis. Le département n'a pas pour vocation première de développer des produits commerciaux, mais plutôt de mener des projets de recherche qui sont motivés par des problèmes concrets rencontrés par les équipes opérationnelles de la banque. D'autres départements de JP Morgan sont à même de mener des projets plus appliqués.

Au sein de ce département, l'équipe de Paris est composée de 5 personnes dont le principal sujet de recherche est le raisonnement automatique. L'équipe est dirigée par Nelson Vadori, un chercheur passé par Centrale Supélec et ayant soutenu sa thèse à l'université de Calgary au Canada. Il se spécialise dans les domaines des mathématiques financières, de la théorie des jeux et du raisonnement automatique, et travaille au sein du département *AI Research* de JP Morgan depuis 2019.

Les chercheurs de l'équipe de Paris sont amenés à travailler sur plusieurs projets en parallèle, en collaboration avec les d'autres équipes au sein de JP Morgan. Certains projets sont directement liés aux activités de la banque, comme la détection de fraudes, et d'autres sont plus fondamentaux, comme le raisonnement automatique. Un équilibre doit être trouvé entre ces projets plus appliqués et des projets de recherche plus fondamentale. Pour ma part, j'ai travaillé sur deux projets de recherche fondamentale, en lien avec le raisonnement automatique et les modèles de langage.

# Chapitre 1

# Revue de la littérature

Comme nous le notons dans le propos introductif, les LLM montrent leurs limites dans leur capacité à mener des raisonnements logiques et mathématiques. En particulier, leur performance est entravée par des hallucinations et des approximations qui rendent les tâches de raisonnement complexe difficiles. La capacité à vérifier l'exactitude d'une preuve mathématique est une caractéristique souhaitable qui peut être obtenue par l'utilisation d'assistants de preuve. Nous commençons par introduire quelques éléments de contexte avant de présenter la littérature existante.

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
    rw [mul_comm a b]
    rw [mul_assoc b a c]
```

FIGURE 1.1 – Exemple de preuve en Lean. La première étape de la preuve utilise la commutativité de la multiplication dans $\mathbb{R}$, et la seconde utilise l'associativité de la multiplication.

## 1.1 Contexte

### 1.1.1 Assistants de preuve

Les assistants de preuve fournissent un moyen de vérifier l'exactitude d'une preuve mathématique. En partant d'un énoncé mathématique écrit dans un language formel (première ligne de la figure 1.1), une démonstration formelle (lignes suivantes) peut être vérifiée automatiquement. Le domaine est divisé entre les prouveurs interactifs (ITPs) et les prouveurs automatiques (ATPs). Les ITPs nécessitent une intervention humaine pour écrire la majeure partie de la preuve (lignes 2 et 3 de la figure 1.1), tandis que les ATPs sont entièrement automatisés. Les ITPs sont plus expressifs et peuvent être utilisés pour prouver une gamme plus large de théorèmes en logique d'ordre supérieur (figure 1.2). Notons que

des problèmes comme le problème de satisfiabilité booléenne (SAT) restent NP-complets [1], ce qui nous donne l'intuition que les ATPs ne présentent pas de garantie de terminaison (ils sont généralement basés sur des heuristiques complexes). Les ITPs les plus populaires incluent Coq, Isabelle, Lean et Metamath, tandis que les ATPs incluent E, CVC4, Z3, Vampire et SPASS.

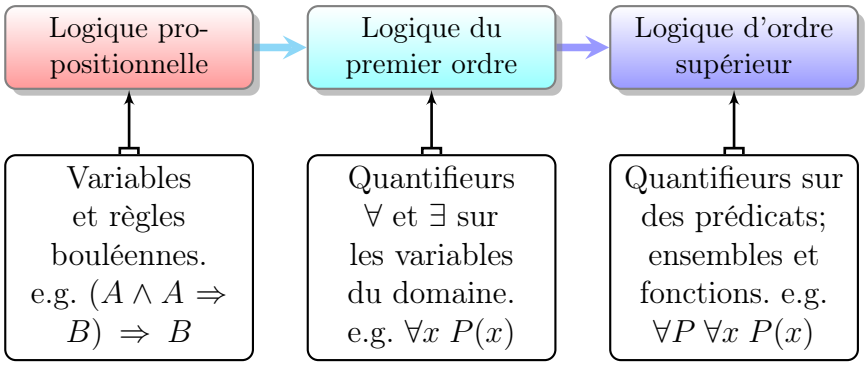


FIGURE 1.2 – Logique propositionnelle, logique du premier ordre et logique d'ordre supérieur. Les logiques d'ordre supérieur sont plus expressives, tandis que celles d'ordre inférieur peuvent offrir de meilleures propriétés comme la décidabilité.

### 1.1.2 Théorie des types

L'assistant de preuve autour duquel se structure ce rapport, Lean, est basé sur la théorie des types. Il s'agit d'une alternative à la théorie des ensembles en tant que fondement des mathématiques. Le principe central de la théorie des ensembles est que chaque objet est un ensemble. En théorie des types, le principe central est que chaque objet a un type. Il existe plusieurs théories des types, celle dont nous dessinerons les contours est le *Calcul des Constructions avec types inductifs*.

Les `types` classifient des `objets`, ce qu'on dénotera avec " : ". Par exemple `f` : $\mathbb{R}$ `→` $\mathbb{R}$ désigne une fonction ($\to$) des objets de type $\mathbb{R}$ vers les objects de type $\mathbb{R}$.

**Paramètre univers** Les types sont eux-même des objets au sein de la théorie des types, ce qui définit plusieurs niveaux de types appelés univers. En Lean, ceux-ci sont définis par un entier naturel, ainsi on a $\forall n \in \mathbb{N}$, `Type` $n$ : `Type` $n+1$. À la base de cette hiérarchie se trouve `Type 0` qui suffira pour la plupart des applications, par exemple $\mathbb{R}$ : `Type 0` et $\mathbb{N}$ :

1. Un problème est NP si on peut vérifier sa solution en temps polynomial. Il est NP-complet s'il est au moins aussi difficile que tous les problèmes de la classe NP.

`Type 0`. Il existe un autre type important, `Prop`, qui est le type des propositions. Dans la hiérarchie des types, on a en fait `Prop : Type -1`. Ce type est celui des propositions, et donc des théorèmes.

**Isomorphisme de Curry-Howard** La théorie des types permet une représentation des mathématiques équivalente à la traditionnelle théorie des ensembles. Informellement, une proposition `p : Prop` est l'énoncé d'un théorème, et une preuve de `p` est un objet `o : p`. Autrement dit les preuves sont des programmes, et la proposition prouvée est le type du programme. Cet isomorphisme est connu sous le nom de *Curry-Howard*. Il permet la vérification des preuves : l'assistant de preuve fournit les outils pour construire les propositions et les preuves, et pour calculer leurs types. Il suffit alors de vérifier que ces objets sont bien formés et que le type de la preuve est bien la proposition à prouver.

**Polymorphisme** Lean permet de définir des objets polymorphes, c'est-à-dire des objets qui dépendent d'un ou plusieurs types. Par exemple, il est naturel de définir des structures de données polymorphiques (figure 1.3). En particulier on peut avoir un polymorphisme de type simple, ou un polymorphisme d'univers (c'est-à-dire que la définition est aussi paramétrée par le niveau d'univers du type).

```
inductive List (α : Type u) where
  | nil : List α
  | cons (head : α) (tail : List α) : List α
```

FIGURE 1.3 – Définition d'une liste polymorphe en Lean.

**Classes de type** Les classes de type sont un mécanisme de polymorphisme plus avancé. Elles permettent de définir des interfaces pour des types généraux et de définir des instances pour ces interfaces. Par exemple, on peut définir une classe de type pour l'addition (figure 1.4) et une instance pour les entiers naturels. En Lean, les structures mathématiques usuelles (groupes, anneaux, corps, etc.) sont définies comme des classes de type.

**Unification** L'unification est un processus qui consiste à la bonne instantiation des variables dans une expression. Dans l'exemple de la figure 1.5, on a un théorème qui affirme que pour tout groupe $G$, l'inverse à droite d'un élément $a$ est $a^{-1}$. Lorsqu'on applique ce théorème à $x : \mathbb{R}$, Lean va unifier $G$ avec $\mathbb{R}$, $u$ avec 0 (paramètre univers), et

trouver une instance de la classe de type `Group` pour $\mathbb{R}$. Ainsi l'utilisateur n'a pas besoin de spécifier ces paramètres qui sont implicites.

```
class Add (α : Type u) where
    add : α → α → α

instance : Add Nat where
    add := Nat.add
```

FIGURE 1.4 – Définition d'une classe de type pour l'addition en Lean.

```
theorem mul_right_inv {G : Type*} [Group G] (a : G) : a * a⁻¹ = 1 := by
    rw [← mul_left_inv a⁻¹, inv_eq_of_mul (mul_left_inv a)]
```

FIGURE 1.5 – Théorème sur l'inverse à droite d'un élément dans un groupe.

Tous ces éléments ont pour but de faciliter la construction de preuves en Lean. Ils prennent une place importante dans le travail que nous avons réalisé.

### 1.1.3 Tactiques et recherche de preuve

La construction de preuves devenant rapidement fastidieuse, les ITPs fournissent des tactiques qui peuvent être utilisées pour faciliter cette tâche. Vulgairement, les tactiques sont des fonctions qui prennent un objet et retournent un nouvel objet. L'entrée en mode "tactique" se fait par le mot-clé `by`. Certaines tactiques peuvent implémenter des algorithmes complexes, par exemple pour résoudre des inégalités (`linarith`) ou pour simplifier des expressions (`simp`). On illustre ces concepts avec un exemple simple en Lean dans la figure 1.6.

Le mode "tactique" est fondamentalement différent du mode "terme" en ce sens qu'il introduit la notion de but. Un but est une méta-variable, c'est-à-dire un trou à remplir dans la preuve, dont on connaît le type désiré mais pas la valeur. Il est généralement dénoté par le symbole "`_`" ou par un point d'interrogation "`?m`". Au début, la preuve est vide et le but est de prouver le théorème. Les tactiques sont utilisées pour transformer le but en buts plus simples jusqu'à ce que le but soit trivial, comme illustré dans la figure 1.7. Une tactique peut créer plusieurs sous-buts, qui peuvent généralement être résolus indépendamment (sauf lorsqu'une métavariable apparaît dans plusieurs sous-buts, par

exemple en prouvant deux buts de la forme $\exists x$ t.q. $A(x) \wedge B(x)$ où les deux buts doivent être réalisés par le même $x$).

```
-- Preuve en mode "terme"
theorem rfl_term : x = x :=
    Eq.refl x

-- Preuve en mode "tactique"
theorem rfl_tactic : x = x := by
    rfl
```

FIGURE 1.6 – Comparaison entre une preuve en mode "terme" et une preuve en mode "tactique". La preuve en mode "terme" est plus explicite, car on écrit directement le terme de la preuve. Dans le cas de la réflexivité, on utilise l'unique constructeur de l'égalité (`a = b` est une syntaxe pour `Eq a b` dont l'unique constructeur est la réflexivité). La preuve en mode "tactique" est plus concise et intuitive, car on n'a pas besoin de connaître les détails d'implémentation de l'égalité. Ces différences sont exacerbées dans les preuves plus complexes.

```
example (A B : Prop) (h₁ : A → B) (h₂ : A) : B := by -- But : B
    apply h₁ -- Nouveau but : A
    exact h₂ -- Plus aucun but
```

FIGURE 1.7 – Évolution du but lors de la preuve d'un théorème en Lean.

Si on exclut le cas de tactiques menant à plusieurs buts, la recherche de preuve peut être vue comme une recherche dans un arbre, où chaque nœud est un but et chaque arête est une tactique (figure 1.8). L'objectif est de trouver une séquence de tactiques qui mène à un but trivial. De façon équivalente, on pourrait voir la recherche de preuve comme la prédiction de termes manquants (les méta-variables), mais l'approche en mode "tactique" est préférée car elle est plus intuitive et plus proche de la manière dont les mathématiciens écrivent des preuves. D'autre part, l'utilisation de buts permet de définir des tactiques complexes capables de donner des résultats différents en fonction du but courant.

## 1.2 Littérature existante

### 1.2.1 Mathématiques et Intelligence Artificielle

**Historique** L'idée d'utiliser des méthodes automatisées pour résoudre des problèmes mathématiques date des débuts de l'informatique, avec Alan Turing [20]. Cependant, les véritables applications de l'IA en mathématiques ont commencé à émerger dans les années 1960 et 1970, dans des domaines tels que la logique mathématique et les démonstrations automatiques de théorèmes.

En 1960 est conçu le premier programme de vérification automatique de formules en logique du premier ordre [1]. Par la suite se développent des systèmes de calcul formel, tels que *Mathematica* et *Maple*, ainsi que des assistants de preuve interactifs, comme *Coq*, *Isabelle* et *Lean*.

Les tentatives d'utiliser ces outils pour résoudre des problèmes mathématiques complexes ont été moins fructueuses. Bien que ces systèmes aient montré leur efficacité dans des domaines spécifiques, ils ont souvent rencontré des limitations lorsqu'il s'agissait de traiter des problèmes plus généraux ou de découvrir de nouveaux théorèmes.

**Avancées récentes** L'engouement autour de l'application de l'IA aux problèmes mathématiques a été retrouvé ces dernières années. D'une part, grâce au succès des algorithmes d'apprentissage par renforcement (RL) comme *AlphaZero* [17] pour résoudre certains types de problèmes complexes. La capacité de tels algorithmes à apprendre des stratégies gagnantes dans des jeux comme le Go et les échecs laisse entrevoir des applications potentielles dans d'autres domaines, y compris les mathématiques. D'autre part, les progrès récents des LLM ont ouvert de nouvelles perspectives. En effet, ces modèles permettent de générer du texte de manière cohérente et pertinente, dans un espace (le langage) très large.

Les techniques actuelles les plus efficaces se concentrent sur l'utilisation combinée de modèles de langage avec des outils de calcul formel (par exemple le package *sympy* en *Python*) [9]. Ces approches restent limitées par la nécessité d'avoir les outils adaptés à disposition, ce qui n'est pas le cas pour la recherche de pointe. En revanche, les assistants de preuve permettent à la fois de définir des objets mathématiques et de prouver des résultats qui y sont liés. Autrement dit, les outils de calcul formel fournissent une interface

limitée avec des algorithmes prédéfinis, tandis que les assistants de preuve permettent une interaction plus riche avec les objets mathématiques.

### 1.2.2 Assistants de preuve et Intelligence Artificielle

La nature textuelle des tactiques et des buts dans les ITPs en font de bons candidats pour être générés par des LLMs. Nous distinguons quatre tâches principales abordées par la recherche dans ce domaine [15] :

- **Autoformalisation** : Étant donné un énoncé mathématique en langage naturel, générer une représentation formelle de l'énoncé. Cela peut également inclure la formalisation de la preuve de l'énoncé. Le problème inverse de l'informalisation est également pertinent.
- **Sélection de prémisses** : Étant donné un énoncé mathématique en language formel, générer un ensemble de prémisses qui sont susceptibles d'être utiles pour prouver l'énoncé.
- **Génération d'étapes de preuve** : Étant donné un but à prouver, générer une étape de preuve. En d'autres termes, générer une tactique à utiliser conditionnée par le but actuel.
- **Recherche de preuve** : Étant donné un but à prouver, générer une suite d'étapes qui résolvent le but. Les travaux initiaux se sont concentrés sur la recherche d'un chemin (séquence d'étapes) dans l'arbre de preuve en observant l'évolution du but à chaque étape, tandis que des travaux plus récents prédisent directement la preuve entière.

Une autre tâche qui a reçu moins d'attention est la **conjecture**, qui consiste à générer de nouveaux énoncés mathématiques susceptibles d'être vrais. C'est pourtant peut-être la tâche la plus intéressante, mais les difficultés rencontrées sur les tâches précédentes ne permettent pas encore de l'aborder. Les principales difficultés sont :

- **Rareté des données** : Il y a peu de données de mathématiques formalisées. L'apprentissage auto-supervisé est donc difficile. Parmi les jeux de données qui existent, la plupart se concentrent sur la *construction de librairies*[2] et non sur la résolution de problèmes.

2. La construction de librairies est le fait de formaliser des théories mathématiques existantes dans un language formel. Généralement, on se donnera un résultat significatif auquel on veut arriver, et on formalisera les théorèmes et définitions nécessaires pour y arriver.

- **Complexité** : Contrairement à la plupart des jeux comme les échecs ou le Go, l'espace d'états des assistants de preuve est infini. Pire encore, l'espace d'actions est également infini, notamment en raison de la possibilité de *générer des termes exogènes* à chaque étape (c'est-à-dire introduire des termes qui ne font pas partie des hypothèses).
- **Pas d'auto-jeu** : L'auto-jeu a été une clé du succès des algorithmes de RL, par exemple dans *AlphaZero*. Cependant une telle approche est difficile à mettre en place dans le contexte des assistants de preuve, car le jeu n'est pas adversarial.
- **Retour d'information rare** : Le retour d'information de l'ITPs est rare, il vous informe que les étapes de preuve sont correctes jusqu'à présent, mais ne vous donne pas d'indication sur la suite de la preuve. Or mener un raisonnement correct ne garantit pas de trouver une preuve, car on peut atteindre des états improuvables. Évaluer la qualité d'une preuve intermédiaire est donc difficile.
- **Dépendance à une librairie** : Le succès d'une preuve peut dépendre de nombreuses définitions et résultats établis (par exemple, il y a plus de 100000 théorèmes dans la librairie *mathlib* en Lean). Exploiter cette information est un défi, qu'il s'agisse de la mémoriser pendant l'entraînement, ou d'y accéder pendant la recherche de preuve.

### 1.2.3 Recherche de preuve

Dans cette partie, nous nous concentrons sur le raisonnement automatique dans un language formel. Par limite de place, nous ne pouvons pas couvrir toutes les contributions, mais nous nous concentrons sur les plus significatives.

Dans *Generative Language Modeling for Automated Theorem* [8] est introduite l'idée de générer des étapes de preuve avec un LLM. En utilisant un jeu de données supervisées de preuves formelles, les auteurs génèrent un jeu de données d'étapes de preuve, c'est-à-dire des paires `(but, tactique)` où le but à prouver est avancé en appliquant la tactique. Ils entraînent ensuite un LLM à prédire la tactique à partir du but. La possibilité de choisir différentes tactiques motive l'utilisation d'algorithmes de recherche plus élaborés.

On modélise généralement la recherche de preuve comme un problème de recherche arborescente. Les nœuds de l'arbre sont les buts à prouver, et les arêtes sont les tactiques qu'on applique pour avancer au prochain but (figure 1.8). En réalité, une tactique peut générer plusieurs sous-buts, qu'on pourra parfois traiter séparément pour réduire la complexité de

l'arbre de recherche. Si on se place dans ce paradigme, on modélisera plutôt le problème comme un hyper-arbre, où chaque arrête peut mener à un ensemble de nœuds.

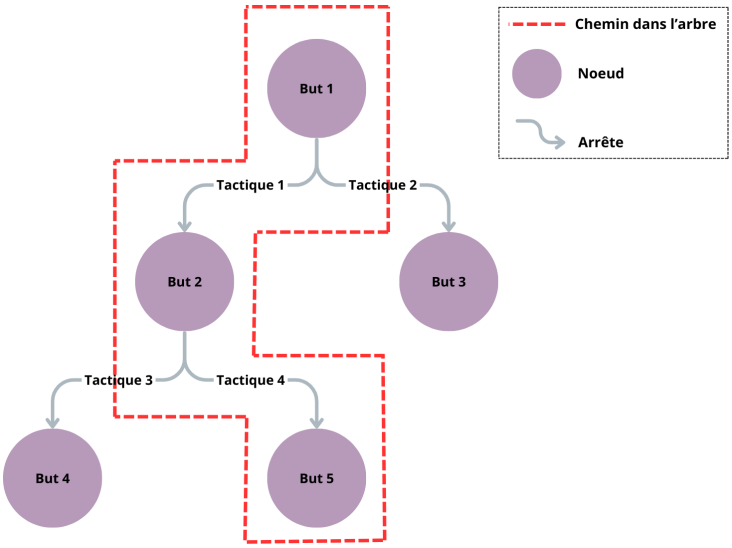


FIGURE 1.8 – Exemple d'arbre de preuve simple. Les nœuds sont les buts à prouver, et les arêtes sont les tactiques appliquées pour avancer au prochain but.

**Algorithm 1** Recherche "best-first".

**Entrée :** But initial $s_0$, Heuristique $h(s)$
**Sortie :** Séquence de preuve ou échec
Initialiser liste *ouvert* $\leftarrow [s_0]$
Initialiser liste $fini \leftarrow []$
**tant que** *ouvert* n'est pas vide **faire**
    $s \leftarrow$ DEPILE_MEILLEUR(*ouvert*)
    **si** EST_BUT(s) **alors**
        $G \leftarrow$ ECHANTILLONE_TACTIQUES($s$) # *Utilise le LLM*
    **fin si**
    Ajoute $s$ à $fini$
    **pour** chaque $s' \in G$ **faire**
        **si** $s' \notin fini$ et $s' \notin$ *ouvert* **alors**
            INSERT($ouvert, s', h$) # *Insérer* $s'$ *avec priorité* $h(s')$
        **fin si**
    **fin pour**
**fin tant que**
**retourner** échec

Les premiers travaux se contentent d'utiliser un algorithme "best-first" (algorithme 1) avec des log-probabilités cumulatives (i.e. prenant en compte les probabilités de chaque action ayant mené à un nœud) pour évaluer les nœuds dans l'arbre de recherche (les probabilités sont celles prédites par le LLM). Quand un nœud est sélectionné, il est étendu en échantillonnant plusieurs actions à partir du LLM et en les vérifiant avec l'ITP avant de les ajouter à la file d'attente. Cependant, cette approche est limitée par la qualité du LLM, qui évalue mal la valeur d'un nœud.

#### 1.2.3.1 Recherche arborescente Monte-Carlo (MCTS)

MCTS est un algorithme de recherche arborescente utilisé pour résoudre des problèmes de décision séquentielle. Il se base sur l'idée de simuler des trajectoires dans l'arbre de recherche pour estimer la valeur des nœuds (d'où le nom de Monte-Carlo). L'algorithme est détaillé dans la figure 1.9. Il est composé de quatre étapes principales :

1. **Sélection** : En partant de la racine, on sélectionne récursivement un nœud jusqu'à atteindre une feuille. La façon dont on sélectionne les nœuds dépend de la stratégie utilisée.
2. **Expansion** : On étend le nœud sélectionné en ajoutant un ou plusieurs nœuds enfants. Cette expansion peut être exhaustive ou basée sur un échantillonnage.
3. **Simulation** : On simule des trajectoires aléatoires à partir des nouveaux nœuds jusqu'à atteindre un état final.
4. **Rétropropagation** : On calcule la valeur d'un nouveau nœud comme la moyenne des valeurs des trajectoires simulées. On met ensuite à jour les valeurs des nœuds parents en remontant l'arbre, la valeur de chaque nœud parent étant la moyenne des valeurs de ses enfants.

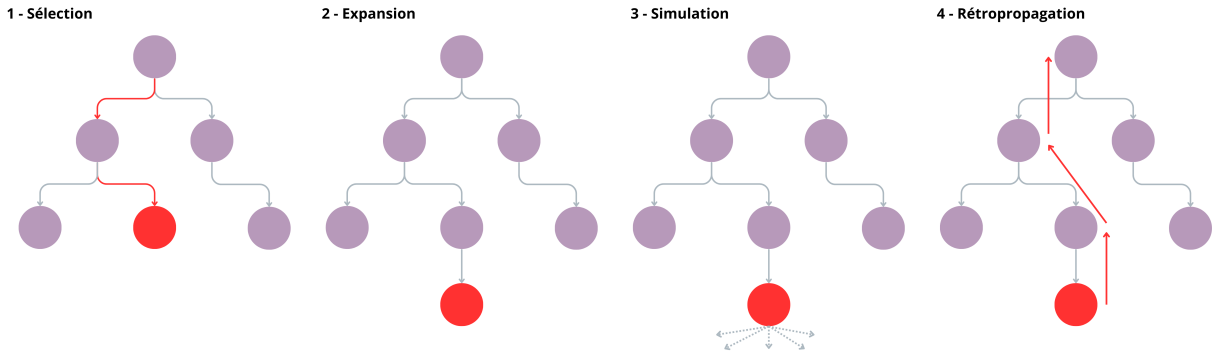


FIGURE 1.9 – Recherche arborescente Monte-Carlo.

L'adaptation de MCTS à la recherche de preuve n'est pas triviale. Il faut construire une heuristique pour évaluer la valeur des nœuds, échantillonner de nouveaux nœuds efficacement, et atteindre des états finaux lors de la simulation.

#### 1.2.3.2 AlphaZero et recherche de preuve

*AlphaZero* est une variante de l'algorithme MCTS où on se donne un réseau de neurones pour estimer la valeur des nœuds (au lieu de faire des simulations aléatoires). Celui-ci est entraîné avec un algorithme de RL, et il est également utilisé pour échantillonner les nœuds à étendre (i.e. il apprend à la fois la valeur d'un état et la politique à suivre). Cet algorithme résout notamment le problème du retour d'information rare, à condition que le réseau de neurones soit suffisamment précis dans sa prédiction de la valeur des nœuds.

Dans *Hyper-Tree Proof Search* [11], les auteurs proposent un tel algorithme de recherche de preuve. Pendant une simulation, ils maintiennent un hypergraphe des nœuds visités jusqu'à présent et des tactiques utilisées. À chaque étape, ils sélectionnent un hyper-arbre (c'est-à-dire une preuve partielle) à étendre, étendent les feuilles non terminales de cet hyper-arbre, et rétropropagent les valeurs estimées des nouvelles feuilles.

#### 1.2.3.3 Autres approches

Dans *Draft, Sketch, and Prove* [4], les auteurs génèrent d'abord une preuve ébauche en langage naturel, puis esquissent [3] la preuve dans un language formel en utilisant l'apprentissage par quelques exemples, et enfin utilisent une méthode en boîte noire pour compléter les trous dans la preuve (ATPs, MCTS, etc.). Non seulement cette méthode suggère d'exploiter des données non formelles, mais les étapes de l'ébauche (assimilable à une conjecture) et de l'esquisse (assimilable à une auto-formalisation) améliorent considérablement la performance de l'algorithme de recherche. Dans *Lego Prover* [14], les auteurs construisent sur le travail de [4] pour créer une bibliothèque de lemmes utiles qui peuvent être récupérés par le modèle.

Dans *DT-Solver* [5], les auteurs se concentrent sur deux problèmes des MCTS traditionnels : a) le budget de calcul devrait dépendre de l'état, car certains états sont plus difficiles à prouver que d'autres, et b) la recherche devrait être conditionnée par la trajectoire. Pour traiter le premier problème, ils utilisent des nœuds virtuels à partir desquels la recherche

3. Dans de nombreux ITPs, vous pouvez insérer des espaces réservés au milieu de votre preuve à compléter plus tard. Dans lean, cet espace réservé est le mot-clé `sorry`.

peut être poursuivie, même lorsque tous les autres nœuds ont été étendus. Pour traiter le second problème, ils introduisent une fonction de valeur au niveau de la preuve (partielle).

Dans *Monte Carlo Tree Self-refine* [21], les auteurs abordent le raisonnement mathématique informel par l'utilisation de l'auto-affinement et de MCTS. De nouveaux nœuds sont ajoutés par auto-affinement, et les valeurs des nœuds sont également estimées par le LLM.

### 1.2.4 Recherche de preuve et augmentation des données

En raison du manque de données formelles, l'augmentation des données, i.e. la génération de données supplémentaires à partir de données existantes, est une approche prometteuse pour améliorer la performance des LLMs dans les assistants de preuve.

#### 1.2.4.1 Itération experte

L'itération experte est une technique d'augmentation des données qui consiste à affiner un modèle pré-entraîné sur des exemples que le modèle réussit à résoudre. Cette technique est particulièrement utile lorsqu'on dispose d'un moyen de vérifier si l'étiquette prédite est correcte, mais que l'on manque de données étiquetées.

**Itération experte**
Soit $M_0$ un modèle pré-entraîné, $\mathcal{D}$ des données d'entraînement non étiquetées et supposons que nous avons un moyen (éventuellement heuristique) de savoir si l'étiquette prédite est correcte.

1. À l'étape $i = 0$, appliquer $M_0$ aux exemples de $\mathcal{D}$. Les exemples pour lesquels l'étiquette prédite est correcte sont ajoutés avec leur étiquette à un ensemble $S_0$.
2. À l'étape $i > 0$, entraîner $M_0$ sur $\cup_{0 \leq j < i} S_i$ (ajustement fin), ce qui donne $M_i$. Appliquer $M_i$ aux exemples de $\mathcal{D}$. Les exemples pour lesquels l'étiquette prédite est correcte sont ajoutés avec leur étiquette à un ensemble $S_i$.

Après $n$ étapes, retournez le modèle $M_n$.

Dans *Formal Mathematics Statement Curriculum Learning* [6], les auteurs pré-entraînent d'abord leur modèle sur la prédiction d'étapes de preuve, puis utilisent l'itération experte pour affiner le modèle (sur un jeu de données de problèmes d'olympiades mathématiques).

#### 1.2.4.2 Tâches auxiliaires à partir de données formelles

Dans *Proof Artifact Co-training* [18], les auteurs définissent un ensemble de 9 tâches auxiliaires en plus de la tâche principale de génération d'étapes de preuve en mode "tactique". Ces tâches sont générées à partir de données formelles et sont conçues pour aider le modèle à apprendre des informations utiles pour la tâche principale. Elles se concentrent principalement sur la compréhension des termes générés par les tactiques, ce qui nécessite une compréhension assez fine du fonctionnement de lean. Les tâches sont les suivantes :

- Étant donné un **état de preuve** (variables, hypothèses, buts), prédire le prochain **lemme** à utiliser.
- Étant donné un **état de preuve**, prédire le **terme** manquant.
- Étant donné un **terme partiel** (c'est-à-dire un état de preuve élaboré avec une métavariable), prédire le **terme** manquant.
- Étant donné un **terme partiel**, prédire le **type** du terme manquant.
- Étant donné un **état de preuve**, prédire l'état **détaillé**, i.e. avec les informations de type usuellement omises.
- Étant donné un **terme**, prédire le terme **détaillé**.
- Étant donné un **état de preuve** et une **prémisse**, prédire si la prémisse est utilisée dans le reste de la preuve.
- Étant donné un **état de preuve** et une **hypothèse** locale (i.e. qui fait partie de l'état), prédire si l'hypothèse est utilisée dans le reste de la preuve.
- Étant donné le **type** du théorème et son **terme**, prédire son nom.

#### 1.2.4.3 Génération de données synthétiques

**Données spécifiques au domaine** Une série d'articles ([2] [3] [13] [16]) utilisent des données spécifiques à un domaine pour entraîner un LLM. En particulier, des données synthétiques générées à partir de règles connues (par exemple intégration, différentiation, etc.) et utilisées pour entraîner le modèle. Ils montrent que le modèle est particulièrement efficace pour apprendre des règles mathématiques complexes. Un intérêt particulier est donné à l'étude de la distribution des données générées et à la capacité de généralisation du modèle sur des distributions différentes.

**AlphaGeometry** *AlphaGeometry* [19] repose exclusivement sur la génération de données synthétiques pour entraîner le modèle. Dans leur travail, les auteurs se concentrent sur le cadre limité de la géométrie euclidienne. Ils en tirent les avantages suivants :

- Ils peuvent générer une quantité illimitée de données par échantillonnage aléatoire d'axiomes constructivistes.
- Ils peuvent prouver des énoncés par recherche exhaustive en raison du petit espace d'action (tant qu'ils n'introduisent pas de nouvelles constructions géométriques).

Le modèle apprend à générer des étapes de preuve dans un langage formel propre à la géométrie euclidienne. En particulier, le modèle apprend à générer les constructions géométriques qui ne font pas partie des hypothèses mais qui sont nécessaires pour prouver le théorème (les **termes exogènes** qui compliquent la tâche). De fait les théorèmes ne nécessitant pas de nouvelles constructions géométriques peuvent être prouvés par recherche exhaustive, mais le modèle gagne quand même à apprendre à les prouver.

## 1.3 Positionnement de notre travail

La littérature existante montre que l'utilisation de LLMs pour la recherche de preuve dans les ITPs est une approche prometteuse. Nous nous inspirons de *AlphaGeometry*, qui a montré que l'utilisation de données synthétiques peut donner d'excellents résultats même sur des tâches aussi complexes que les mathématiques, et nous choisissons de générer des données synthétiques par échantillonnage aléatoire. Nous nous plaçons cependant dans un cadre plus général en utilisant l'assistant de preuve Lean. Ce travail est novateur car il ne repose pas sur l'utilisation de "tactiques", mais sur les outils de métaprogrammation de Lean. Notre objectif est de générer des données synthétiques pouvant être utilisées dans le cadre des méthodes présentées dans la revue de littérature qui précède.

# Chapitre 2

# Raisonnement avant

La performance des modèles de langage sur les tâches d'écriture de démonstrations formelles sont en deçà des attentes. Les difficultés d'apprentissage sont principalement le fait de la rareté des données. Des tâches de difficulté similaire (comme la génération de code) ont atteint des performances satisfaisantes grâce notamment à la grande quantité de données disponibles. Les données formelles sont particulièrement rares, car pour être générées elles nécessitent à la fois une expertise dans le domaine mathématique et dans l'utilisation d'un assistant de preuve.

De plus nous observons que la principale source de données en Lean, *mathlib*, est axée sur les définitions et les théorèmes généraux, et fournit peu d'applications concrètes des résultats qu'elle formalise. En général, un mathématicien souhaite formaliser une théorie précise, et pour cela il définit et prouve les différentes briques de cette théorie. Cependant, il est rare qu'il formalise des applications concrètes de ces résultats. Or, il est souhaitable qu'à la manière d'un élève qui apprendrait les mathématiques, un modèle de langage puisse à la fois apprendre les énoncés des théorèmes, leurs preuves, et leurs applications. C'est d'autant plus souhaitable que mener de la recherche fondamentale automatiquement est un objectif hors de portée pour l'instant, et donc que les utilisations de ces modèles se feront principalement dans des applications concrètes [1].

Nous notons d'autre part que la littérature se concentre presque exclusivement sur les preuves en mode "tactique" qui relèvent du raisonnement arrière : il faut se donner un but et trouver les étapes pour le démontrer [2] (figure 2.1). Pourtant, les mathématiciens utilisent principalement le raisonnement avant, qui consiste à explorer les conséquences d'une hypothèse pour arriver à une conclusion. Le raisonnement avant permet une exploration ouverte, ce qui est plus naturel pour les mathématiciens, et ouvre la voie à la

1. Ce point de vue n'engageant que l'auteur.
2. Si le raisonnement arrière est utilisé pour l'écriture de preuves formelles, c'est d'une part parce qu'il se prête au système de méta-variables utilisé dans les assistants de preuve, et d'autre part parce qu'il permet de concevoir des tactiques plus puissantes pouvant prendre en paramètre le but courant.

génération de données synthétiques. En effet, on peut générer des données en partant d'hypothèses, et en déduire des conséquences nouvelles.

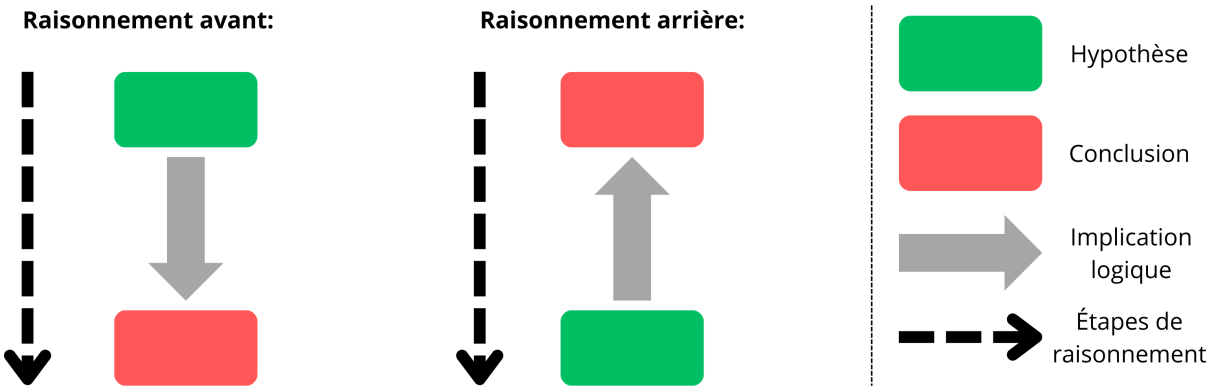


FIGURE 2.1 – Comparaison du raisonnement avant et en arrière. En avant, on part d'hypothèses pour arriver à une (ou plusieurs) conclusion(s). En arrière, on part d'une conclusion pour remonter aux hypothèses (ou plus généralement aux prémisses). Le raisonnement avant permet plus naturellement une exploration ouverte, et est plus adapté à la génération de données synthétiques.

Dans ce chapitre, nous proposons d'explorer le raisonnement avant dans les assistants de preuve. L'idée est de générer un ensemble de données de preuves en utilisant le raisonnement avant, puis de les exploiter pour entraîner un modèle de langage. Nous pensons que cette approche permettra de générer des données plus variées et plus riches que les données actuellement disponibles, et aussi de mieux exploiter les données formelles disponibles.

L'idée d'utiliser le raisonnement avant pour générer des données de preuve en Lean est assez nouvelle. Le seul travail qu'on ait trouvé est un blog post de Siddhartha Gadgil [7], et le code associé, qui a servi de base pour notre travail. En effet, la complexité de Lean nous a poussés à nous appuyer sur des travaux existants pour avancer plus rapidement.

## 2.1 Métaprogrammation en Lean

La métaprogrammation, i.e. la capacité à manipuler des preuves comme des objets de première classe, est une fonctionnalité clé de Lean. En particulier, on manipule des expressions Lean (de type `Expr`) qui représentent à la fois les termes et les types (puisque les types sont des objets de première classe en Lean).

Le figure 2.2 détaille la définition inductive des expressions Lean. Ce sont des objets bas niveau qu'il convient de bien comprendre pour les manipuler correctement. Détaillons quelques constructeurs qui nous intéressent particulièrement :

- `Expr.app` : Application d'une fonction à un argument. Ce constructeur nous permet par exemple d'obtenir le résultat d'un théorème en lui passant ses hypothèses.
- `Expr.forallE` : Quantificateur universel qui décrit les expressions de la forme $\forall x, E(x)$ où $E(x)$ est une expression dépendant de $x$. `binderName`, `binderType` et `body` sont respectivement le nom du paramètre, son type et le corps de l'expression. Ce constructeur nous permet notamment de déduire les hypothèses d'un théorème.
- `Expr.const` : Constante définie dans l'environnement. Ce constructeur nous permet d'une part de récupérer les définitions et théorèmes de *mathlib*, et d'autre part d'introduire de nouvelles définitions dans l'environnement.

```
inductive Expr where
| bvar (deBruijnIndex : Nat) -- bound variable
| fvar (fvarId : FVarId) -- free variable
| mvar (mvarId : MVarId) -- hole in the expression
| sort (u : Level) -- universe level, i.e. Prop, Type, etc.
| const (declName : Name) (us : List Level) -- constant defined in the
    environment
| app (fn : Expr) (arg : Expr) -- function applied to an argument
| lam (binderName : Name) (binderType : Expr) (body : Expr) (binderInfo
    : BinderInfo) -- lambda function
| forallE (binderName : Name) (binderType : Expr) (body : Expr)
    (binderInfo : BinderInfo) -- forall quantifier
| letE (declName : Name) (type : Expr) (value : Expr) (body : Expr)
    (nonDep : Bool) -- let, i.e. let x : α := f y
| lit : Literal → Expr -- string and natural numbers
```

FIGURE 2.2 – Définition inductive des expressions Lean. C'est l'objet central qu'on manipule en métaprogrammation.

On manipule ces expressions de différentes manières. La plus simple est la reconnaissance de motifs, qui permet de décomposer une expression en fonction de son constructeur (figure 2.3). On a aussi des fonctions plus avancées fournies par les monades `CoreM`, `MetaM` et `TermElabM` qui permettent notamment de :

- Calculer le type d'une expression (`inferType`).
- Vérifier si une expression est bien typée (`isTypeCorrect`).
- Inférer automatiquement des méta-variables (`synthesizeSyntheticMVars`). Cette fonctionnalité est particulièrement puissante car elle permet de laisser Lean inférer automatiquement certains arguments d'une fonction.
- Calculer l'égalité définitionnelle [3] de deux expressions (`isDefEq`).

Par exemple, étant donné une expression `e : Expr`, on peut calculer son type `e' : Expr` en utilisant la fonction `inferType`. Si `e'` est égal à `Prop`, alors c'est une proposition. Autre exemple, si le type de `e'`, noté `e''`, est égal à `Prop`, alors `e` est une preuve de la proposition `e'`.

```
match theoremType with
    | Expr.forallE binderName binderType body binderInfo => ...
```

FIGURE 2.3 – Début de la définition d'une fonction permettant de récupérer le domaine d'un théorème. On compare le type du théorème (`theoremType`) à un quantificateur universel.

## 2.2 Extraction de prémisses

Munis de ces outils de métaprogrammation, nous pouvons extraire un ensemble de prémisses d'une librarie Lean (nous choisissons *mathlib*). Notre motivation est la suivante : on souhaite mettre en place un processus de raisonnement avant, qui nécessite donc de se donner un ensemble de prémisses à partir desquelles on pourra déduire des conséquences. On choisit d'abord de travailler avec un `namespace` à la fois. Un `namespace` est un ensemble de définitions et théorèmes regroupés par sujet. Par exemple, `Real` est un `namespace` de *mathlib* regroupant des définitions et théorèmes afférents aux nombres réels. Travailler avec un `namespace` à la fois permet de limiter la complexité du problème en s'assurant que les prémisses sont cohérentes entre elles, mais limite aussi l'espace de recherche.

Grâce aux outils de métaprogrammation de Lean, on extrait tous les `namespace` de *mathlib*, et pour chaque `namespace` on extrait toutes les définitions et théorèmes. Ces définitions

3. L'égalité définitionnelle est l'égalité entre deux expressions qui sont égales par définition. Cette égalité ne nécessite pas de preuve, car Lean peut la vérifier en appliquant les définitions successives des expressions.

et théorèmes sont ajoutés à un ensemble de prémisses, pour pouvoir être utilisés dans le raisonnement avant.

D'autre part, on observe le *domaine* de chaque théorème, c'est-à-dire les hypothèses qu'il nécessite pour être appliqué, et en particulier les types de ces hypothèses. On souhaite introduire des variables de ces types dans notre raisonnement avant, pour pouvoir appliquer les théorèmes qu'on a extrait. En revanche, on ne souhaite pas donner de valeurs à ces variables afin de garder en généralité (si un théorème suppose un réel, on souhaite avoir `x : ℝ` plutôt que `3.14 : ℝ`). Introduire ces variables sans les lier à une définition en particulier correspond en réalité à ce qu'est un axiome en théorie des types. On introduit donc de tels axiomes pour les types des variables.

Afin d'aller à l'essentiel, nous décidons d'abord d'ignorer les dépendances entre hypothèses : si un théorème nécessite $x \in \mathbb{R}^n$, il conviendrait d'abord d'introduire $n \in \mathbb{N}$. Une telle dépendance rend le problème d'introduire les variables beaucoup plus complexe, ce que nous ne réalisons pas dans un premier temps. Un système en entonnoir est envisageable pour introduire les variables dans le bon ordre (chaque variable introduite correspond à l'entrée dans un nouveau contexte local) mais nous ne l'avons pas implémenté.

Pour améliorer le système utilisant un `namespace` à la fois, nous calculons des similitudes entre `namespace` en comptant le nombre de fois qu'une constante d'un `namespace` est utilisée dans un autre (que ce soit dans les définitions ou dans les preuves). On réalise alors une *classification spectrale* en groupes de `namespace` similaires. Cette classification est ajustée par *regroupement hiérarchique* pour obtenir des groupes avec un nombre de `namespace` maximum.

## 2.3 Évolution

Cette partie est inspirée du travail de Siddhartha Gadgil [7]. On se donne deux structures de données principales.

- D'une part, une distribution d'expressions, qui est un ensemble d'expressions Lean qu'on fait évoluer. Cette distribution est séparée en propositions (en réalité des preuves, car on n'a pas besoin de stocker la proposition qui peut être calculée en évaluant le type de la preuve) et en termes (des variables, des fonctions, etc.).
- D'autre part, un ensemble de données externes, qui est un ensemble de prémisses qu'on peut utiliser pour faire évoluer les expressions.

Si vous avez suivi la section précédente, vous aurez compris que les définitions et théorèmes de *mathlib* constituent ces données externes, et que les variables introduites font partie de la distribution d'expressions.

La principale raison pour séparer ces deux structures de données est en réalité technique. Les constantes dans l'environnement dépendent de paramètres univers, qu'il faut préciser pour obtenir une expression Lean. Or ces paramètres dépendent généralement du contexte, donc on ne peut guère les introduire en amont. En revanche on peut les inférer au moment d'utiliser une prémisse, ce qui est plus simple si on a une structure de données séparée.

À partir de ces deux structures de données, on définit un cadre général d'évolution. Cette première étape d'extraction initialise la distribution d'expressions, et les étapes suivantes d'évolution permettent de faire évoluer cette distribution. Le terme d'évolution est ici utilisé pour rappeler que l'on part d'une distribution d'expressions simple, et que l'on souhaite obtenir des expressions plus complexes.

### 2.3.1 Étape d'évolution

L'évolution consiste à appliquer une pseudo-tactique à des expressions. On parle de pseudo-tactique car cette technique est différente des tactiques Lean, qui sont des fonctions qui modifient l'état de la preuve. En revanche, on peut définir des pseudo-tactiques équivalentes à leur homologue Lean. Les pseudo-tactiques utilisent des constructeurs fondamentaux des expressions Lean pour les manipuler, et permettent de faire évoluer les expressions. On est donc dans une approche de métaprogrammation assez bas-niveau comparé à l'utilisation des tactiques Lean. Les pseudo-tactiques que nous avons utilisées sont les suivantes :

- **Application** : étant donné une fonction $f$ et des arguments $x_1, \ldots, x_n$, on crée l'expression $f(x_1, \ldots, x_n)$. Cette pseudo-tactique est particulièrement utile pour appliquer un théorème à ses hypothèses (i.e. $f$ est plus général qu'une simple fonction).
- **Réécriture** : étant donné une expression $e$ et une égalité $a = b$, on remplace $a$ par $b$ dans $e$ ou $b$ par $a$ dans $e$. Cette pseudo-tactique est utile pour simplifier une expression.

On pourrait envisager de nombreuses autres pseudo-tactiques, en s'inspirant des tactiques Lean. En plus de ces pseudo-tactiques, on ajoute une étape de simplification automatique pour simplifier les expressions, utilisant la fonction `simp` de Lean. Cette étape n'est conservée que si elle permet réellement de simplifier l'expression.

L'étape d'évolution est schématisée sur la figure 2.4. Les trois principales étapes sont :

1. Sélection des expressions utilisées pour l'évolution.
2. Application d'une pseudo-tactique avec ces expressions.
3. Vérifications et simplifications des expressions obtenues.

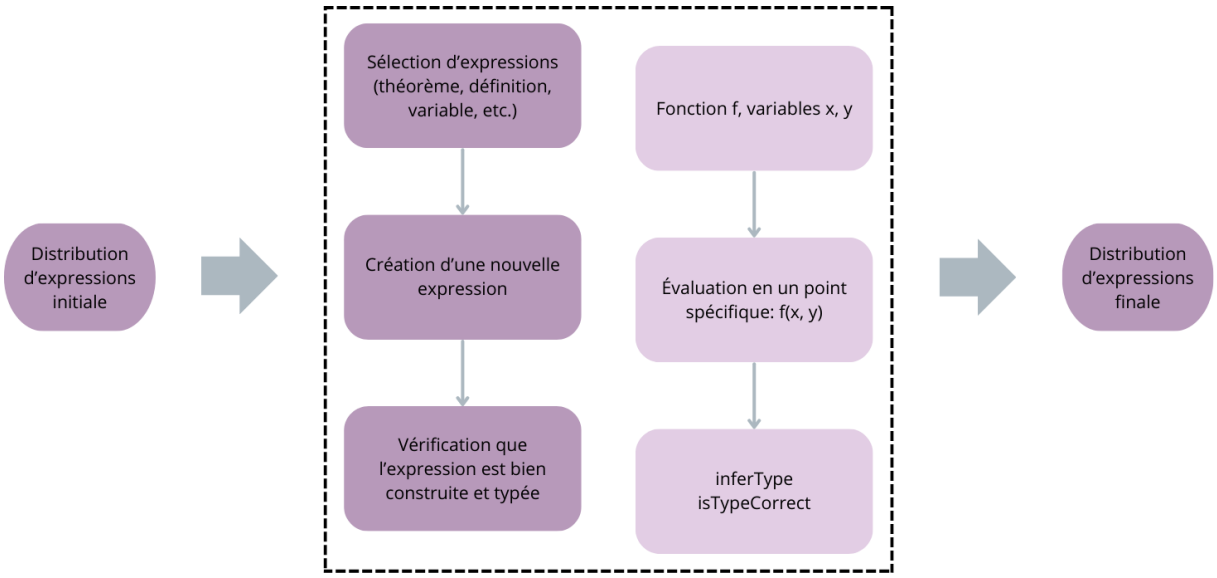


FIGURE 2.4 – Étape d'évolution. Ce cadre général permet d'obtenir de nouvelles expressions à partir d'expressions existantes. Il ne dépend pas des pseudo-tactiques utilisées.

### 2.3.2 Sélection des expressions

Le cas le plus simple consiste, pour chaque pseudo-tactique, à l'appliquer à chaque permutation d'arguments possible. Une telle exploration serait cependant trop large et ne permettrait pas de trouver des résultats intéressants (i.e. avec plus d'étapes de raisonnement). On introduit donc une sélection des expressions à chaque étape d'évolution. On utilise pour cela le même principe que pour l'extraction de prémisses : on calcule le domaine d'un théorème ou d'une définition, et on l'applique aux expressions qui ont le bon type. Cette fois encore on est limité par les dépendances entre hypothèses, qui ne nous permettent pas de facilement calculer le domaine d'un théorème avec de telles dépendances, mais on peut déjà obtenir des résultats intéressants.

Pour les cas où l'utilisation du domaine n'est pas possible, on utilise un échantillonnage aléatoire parmi les combinaisons d'arguments possibles. On a aussi envisagé d'utiliser des

heuristiques pour sélectionner les expressions, mais ces fonctionnalités sont rapidement complexes à mettre en place dans un langage de programmation fonctionnel [4].

Finalement on utilise une notion de degré pour déterminer la profondeur de l'exploration. Ainsi une expression dans la distribution initiale a un degré de 0, et chaque application d'une pseudo-tactique augmente le degré de 1. Pour les pseudo-tactiques prenant plusieurs arguments, on somme les degrés des arguments pour obtenir le degré de l'application. On ne sélectionne pas les expressions dont le degré est supérieur à un certain seuil, qui est un paramètre de l'évolution. Cette notion de degré permet d'équilibrer l'exploration, en évitant de créer des expressions qui nécessitent trop d'étapes de raisonnement pour être obtenues.

En plus de la limite de degré, on introduit d'autres paramètres pour équilibrer l'exploration. Par exemple on limite le nombre d'utilisations d'une prémisse à chaque étape, et on paramétrise l'échantillonnage aléatoire.

### 2.3.3 Évolution globale

L'évolution globale consiste à appliquer en parallèle plusieurs pseudo-tactiques, joindre les résultats, et recommencer. Durant une étape (en parallèle), la même pseudo-tactique est utilisée plusieurs fois (comme illustré par les flèches circulaires sur la figure 2.5). Des pseudo-tactiques différentes peuvent être utilisées en parallèle de cette manière, et les distributions d'expressions obtenues sont jointes pour former la distribution d'expressions de l'étape suivante ("Données après une étape" sur la figure 2.5).

On peut définir plusieurs groupements de pseudo-tactiques (c'est à dire les groupes de pseudo-tactiques qui sont utilisées en parallèle), typiquement nous avons un groupe d'application (car en réalité on a une application qui utilise une expression venant des prémisses, et une application qui utilise une expression venant de la distribution d'expressions) et un groupe de réécriture (car en réalité on a une pseudo-tactique de réécriture gauche vers droite et une pseudo-tactique de réécriture droite vers gauche).

Notons ici que ce cadre d'évolution globale est entièrement arbitraire. On pourrait envisager d'autres cadres, en fonction de l'équilibre entre exploration et exploitation que l'on souhaite obtenir. Des pseudo-tactiques plus complexes pourraient aussi motiver un cadre

4. Notons d'ailleurs qu'un simple échantillonnage aléatoire de permutations doit déjà être implémenté à la main.

d'évolution différent. L'ordre dans lequel interviennent les pseudo-tactiques est déterminant pour l'exploration. Ce cadre assez rigide (car les étapes sont déterminées à l'avance) est un choix de simplicité, qui permet de mettre en place rapidement un système d'évolution.

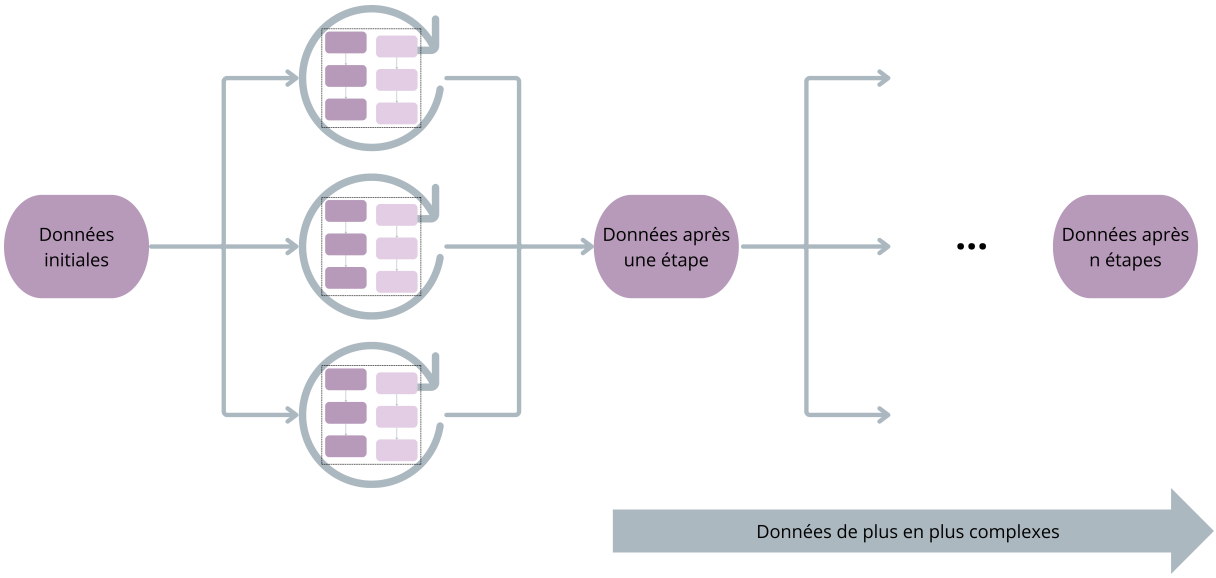


FIGURE 2.5 – Évolution globale. Un groupe de pseudo-tactiques est utilisé en parallèle pour obtenir de nouvelles expressions, qui sont jointes pour former la distribution d'expressions de l'étape suivante.

## 2.4 Traduction en raisonnement arrière

Le système d'évolution permet d'obtenir une distribution d'expressions qui sont les conséquences des prémisses utilisées, et qui sont potentiellement intéressantes pour entraîner un modèle de langage. On souhaite les utiliser pour entraîner un modèle de langage. Or ces preuves générées en raisonnement avant en mode "terme" ne correspondent pas aux preuves Lean, qui utilisent le raisonnement arrière en mode "tactique". On propose donc de traduire nos preuves. Pour cela on utilise une structure de données récursive qui permet de stocker les étapes de construction d'une preuve (figure 2.6). Cette structure d'arbre correspond précisément aux opérations que notre évolution autorise.

Cette structure d'arbre est ensuite traduite en une séquence de tactiques Lean. Notons que la structure récursive nécessite de d'abord traduire les sous-arbres avant de traduire le nœud courant. On commence par parcourir l'arbre pour obtenir une liste d'hypothèses

```
inductive Proof where
| empty
| hypothesis (expr : Expr)
| name (expr : Expr)
| simp (expr : Expr) (at_node : Proof)
| apply (expr : Expr) (function : Proof) (args : List Proof)
| rewrite (expr : Expr) (at_node : Proof) (heq : Proof)
```

FIGURE 2.6 – Arbre de preuve pour la traduction en raisonnement arrière. `.empty` correspond à une preuve vide (utile pour des raisons techniques), `.hypothesis` à une variable qu'on a introduite, `.name` à une prémisse qu'on a utilisée, `.simp` à une simplification, `.apply` à une application, et `.rewrite` à une réécriture.

(les variables que nous avons introduites sous forme d'axiomes), et on calcule le type de la conclusion. On introduit alors les termes de la forme `.apply` avec des hypothèses locales (tactique `have`) et on a des équivalents directs pour les pseudo-tactiques de simplification et de réécriture. Tout cela est réalisé récursivement, et on maintient simplement un compteur pour les noms des hypothèses locales, sachant qu'on a besoin d'y faire référence par la suite et que l'ordre d'introduction n'est pas forcément celui de l'arbre de preuve (par exemple, pour effectuer une réécriture, on a besoin de traduire à la fois l'égalité et l'expression à réécrire avant de pouvoir appliquer la tactique de réécriture).

Au final cette méthode permet d'obtenir des preuves correctes dans la majorité des cas, et on filtre les cas où la traduction n'est pas correcte (en vérifiant la validité du théorème obtenu avec Lean). Le nombre de résultats filtrés n'est pas entièrement négligeable, et on pourrait améliorer cette partie également. Finalement les données obtenues correspondent à des théorèmes Lean classiques que la littérature sait utiliser pour entraîner des modèles de langage.

## 2.5 Résultats

Les nombreuses technicités de Lean et de la métaprogrammation ont rendu difficile la mise en place de ce projet. En particulier, plusieurs choix simplificateurs ont été faits pour pouvoir avancer dans la limite du temps imparti. On a choisi de ne pas introduire de dépendances entre hypothèses, ce qui limite la portée de notre projet. Au final, sur environ 2500 `namespace` extraits de *mathlib*, qu'on a classifiés en 1000 groupes, on a pu générer

des preuves pour environ 20% des groupes. Ces preuves sont en général soit simples (on n'a rien prouvé de spectaculaire), soit trop itératives (certaines prémisses peuvent être utilisées infiniment, par exemple $x > a \implies x + 1 > a$).

On donne d'abord un exemple concret de preuve générée (figure 2.7). Ce résultat utilise deux prémisses de *mathlib*, $\int_0^{\pi/2} \cos^n(x)dx = \frac{1}{2}\int_0^{\pi} \sin^n(x)dx$ et $\int_0^{\pi/2} \cos^n(x)dx > 0$, afin de montrer $\int_0^{\pi} \sin^n(x)dx > 0$ (résultat par ailleurs déjà présent dans *mathlib*). On est assez satisfait d'obtenir ce genre de résultats, qui montrent que notre système est capable de produire des preuves intéressantes.

```
example (n : ℕ) : (0 : ℝ) < ∫ (x : ℝ) in (0 : ℝ)..Real.pi, x.sin ^ n
    := by
    have h1 := EulerSine.integral_cos_pow_eq n
    simp at h1
    have h2 := EulerSine.integral_cos_pow_pos n
    rw [h1] at h2
    simp at h2
    exact h2
```

FIGURE 2.7 – Exemple de preuve générée. Ce code Lean est le résultat de la traduction de l'arbre de preuve en raisonnement arrière.

Par ailleurs, on génère actuellement 20000 preuves pour entraîner un modèle de langage. Ce nombre est arbitraire et les paramètres de l'évolution peuvent être ajustés pour obtenir plus de preuves. Ces données sont ensuite automatiquement transformées pour être utilisées pour l'entraînement d'un modèle de langage. Dans notre cas on a choisi d'utiliser les outils fournis par *lean-dojo* [12] pour transformer les données de preuves en paires *(but, tactique)*. Ces données sont utilisées en addition à celles de *mathlib* (augmentant ainsi de 10% la quantité de données) pour entraîner un modèle de langage.

On choisit d'entraîner DeepSeek Math 7B sur ces données, qui est un modèle de langage pré-entraîné sur des données mathématiques et des données de preuves. On utilise LoRA [10] et on quantise [5] les poids du modèle sur 4 bits pour pouvoir entraîner le modèle sur un GPU équipé de 24 Go de mémoire. On parallélise l'entraînement sur 4 GPUs. On obtient ainsi une augmentation absolue de 0.5% sur le taux de réussite de la prédiction de la prochaine tactique sur un ensemble de validation non vu, de 8.5% à 9.0%.

5. Procédé qui consiste à utiliser des flottants de faible précision (4 ou 8 bits) pour approximer des flottants de plus grande précision (32 bits) afin d'économiser de la mémoire.

# Conclusion et Bilan personnel

Les résultats obtenus lors de ce stage sont encourageants mais pas entièrement satisfaisants. Nous avons réussi à mettre en place un système de génération de données synthétiques en Lean basé exclusivement sur l'assistant de preuve. Cette tâche n'est pas aisée, car elle nécessite une bonne compréhension du fonctionnement de Lean, de ses bibliothèques et de ses outils. Ainsi plus d'un mois a été consacré à la revue de littérature et à la prise en main de Lean. En particulier les notions de programmation fonctionnelle, de monades et de métaprogrammation ont été cruciales pour la réussite de ce projet.

La méthode utilisée est tout à fait pertinente et innovante. Cependant plusieurs améliorations pourraient être apportées pour en tirer le plein potentiel. En particulier, la gestion des dépendances entre types est un point crucial à améliorer. La traduction en raisonnement arrière des preuves générées devrait également être améliorée, certains points de blocages étant aussi liés à la gestion des dépendances entre types. Ces améliorations permettraient de réellement exploiter n'importes quelles données existantes en Lean, et pas juste "les plus simples".

D'autre part, une évaluation rigoureuse des données générées et de leur impact sur les performances des modèles de langage serait nécessaire pour valider l'approche. Le manque de temps a empêché de mener ces expériences, notamment en raison du deuxième projet de recherche mené en parallèle. Le seul résultat quantitatif obtenu sur la prédiction d'étapes de preuve est encourageant, mais il ne suffit pas à valider l'approche. En particulier, on pourrait s'intéresser plus précisément aux données générées à travers plusieurs aspects :

- Performance sur des benchmarks classiques (miniF2F, etc).
- Pertinence des données générées : apprend-on seulement de la syntaxe ou aussi des concepts mathématiques ?
- Comparaison avec des données générées par d'autres méthodes (itération d'expert, méthodes utilisant des LLMs, etc.).
- Impact sur la génération des modèles de langage : les modèles entraînés sur ces données sont-ils biaisés ?

D'une façon générale la littérature manque de benchmarks spécifiques aux assistants de preuve, donc choisir comment évaluer ce jeu de données est un défi en soi.

Finalement, il faut souligner les limites de cette approche. D'une part, de nombreuses tactiques Lean n'ont pas d'équivalent évident dans notre méthode, ce qui limite la variété des données générées, et impose donc d'autres sources de données (contrairement à *AlphaGeometry* qui est entraîné entièrement sur des données synthétiques). D'autre part, les données générées laissent à désirer en terme d'intérêt mathématique. En effet mener un raisonnement correct ne garantit pas qu'il soit intéressant ou utile. Intégrer des heuristiques pour générer des données plus pertinentes serait une piste intéressante pour la suite.

D'un point de vue personnel, ce stage a été une expérience enrichissante. J'ai pu prendre le temps de me plonger dans des sujets complexes et passionnants, et de travailler en collaboration avec des chercheurs de haut niveau. Les compétences acquises en Lean et en programmation fonctionnelle me semblent particulièrement précieuses pour la suite de ma carrière. Du fait de mon parcours hybride entre les mathématiques appliquées et le développement informatique, j'ai été particulièrement séduit par l'outil qu'est Lean et les possibilités qu'il offre. J'espère pouvoir continuer à travailler sur des projets de recherche en lien avec l'IA et les mathématiques, et à explorer les ponts entre ces deux domaines.

# Glossaire

## Définitions

**ajustement fin** En apprentissage automatique, l'ajustement fin (fine-tuning en anglais) est une technique qui consiste à entraîner un modèle pré-entraîné sur un nouveau jeu de données pour l'adapter à une tâche spécifique. 21

**apprentissage par quelques exemples** Technique qui consiste à donner des exemples d'une tâche à un modèle de langage pour qu'il puisse mieux généraliser à de nouveaux exemples (de l'anglais *few-shot learning*). 20

**apprentissage par renforcement** En intelligence artificielle, plus précisément en apprentissage automatique, l'apprentissage par renforcement consiste, pour un agent autonome (ex. : robot, agent conversationnel, personnage dans un jeu vidéo, etc.), à apprendre les actions à prendre, à partir d'expériences, de façon à optimiser une récompense quantitative au cours du temps. (*Wikipédia*). 15

**apprentissage automatique** Champ d'étude de l'intelligence artificielle qui se fonde sur des approches mathématiques et statistiques pour donner aux ordinateurs la capacité d'"apprendre" à partir de données. (*Wikipedia*). 8

**assistant de preuve** Un assistant de preuve est un logiciel qui permet de vérifier la validité d'une preuve mathématique. Il permet de formaliser des théorèmes, de définir des axiomes, et de démontrer des théorèmes en suivant des règles logiques. 6, 7, 10, 11, 12, 17, 21, 23, 24, 25, 35

**auto-affinement** Technique où un LLM génère une critique de sa réponse, puis affine sa réponse en fonction de la critique (de l'anglais *self-refinement*). 21

**auto-jeu** En apprentissage par renforcement, l'auto-jeu est une technique qui consiste à faire jouer un agent contre lui-même, ou contre une version antérieure de lui-même, pour l'entraîner. Les algorithmes de renforcement dépendent de récompenses pour apprendre, et l'auto-jeu est une manière d'adapter la difficulté de la tâche au niveau de l'agent afin de récolter des récompenses plus fréquemment. 17

**but** Dans le contexte des assistants de preuve, un but est une méta-variable, c'est-à-dire un trou à remplir dans la preuve, dont on connaît le type désiré mais pas la valeur. 2, 13, 14, 16, 17, 18, 22, 24, 34

**démonstration formelle** Une démonstration formelle est une séquence finie de propositions (appelées formules bien formées dans le cas d'un langage formel) dont chacun est un axiome, une hypothèse, ou résulte des propositions précédentes dans la séquence par une règle d'inférence (Wikipédia). En ce qui nous concerne, on oppose les démonstrations formelles écrites dans un langage formel, des démonstrations informelles en langage naturel. 10, 24

**grand modèle de langage** Modèle de langage qui a été entraîné sur un très grand corpus de texte, et qui est capable de générer du texte de manière cohérente et pertinente. 6

**intelligence artificielle** Ensemble de théories et de techniques visant à réaliser des machines capables de simuler l'intelligence humaine (Wikipédia). Concept vague et flou, auquel on pourra préférer des termes plus précis en fonction du contexte, comme *apprentissage automatique* ou *apprentissage par renforcement*. 6

**language formel** Un langage formel, en mathématiques, en informatique et en linguistique, est un ensemble de mots1. L'alphabet d'un langage formel est l'ensemble des symboles, lettres ou lexèmes qui servent à construire les mots du langage ; souvent, on suppose que cet alphabet est fini (Wikipédia). En ce qui nous concerne, un langage formel s'approche d'un langage de programmation. 10, 16, 17, 20

**métaprogrammation** La métaprogrammation, nommée par analogie avec les métadonnées et les métaclasses, désigne l'écriture de programmes qui manipulent des données décrivant elles-mêmes des programmes (*Wikipédia*). Dans le contexte de Lean, on s'intéresse à la métaprogrammation pour écrire des programmes qui manipulent des preuves. 23, 25, 26, 27, 29, 33, 35

**monade** Une monade est une structure de données avec un état. En programmation fonctionnelle, les monades sont utilisées pour gérer les effets de bord. Par exemple, une monade peut être utilisée pour gérer les exceptions, un état mutable, etc. 26, 35

**politique** En apprentissage par renforcement, une politique est une fonction qui prend un état en entrée et renvoie une action à effectuer. 20

**prémisse** Une prémisse est une proposition, une affirmation avancée en support à une conclusion (*Wikipédia*). Dans le contexte des assistants de preuve, une prémisse est une proposition dont on a déjà la preuve, et qui est utilisée pour déduire une autre proposition. 16, 22, 25, 27, 28, 30, 31, 33

**prouveur automatique** Outil informatique qui permet de prouver des théorèmes de manière automatique, contrairement aux assistants de preuve qui nécessitent une intervention humaine. 10

**prouveur interactif** Synonyme de "assistant de preuve". Il est dit interactif car la preuve est écrite par le mathématicien. Au contraire, les ATP produisent les preuves en toute autonomie. 10

**raisonnement arrière** Dans un contexte mathématique, le raisonnement arrière consiste à partir de la conclusion pour remonter aux axiomes ou aux hypothèses. 2, 24, 32, 33, 34, 35

**raisonnement avant** Dans un contexte mathématique, le raisonnement avant consiste à partir des axiomes ou des hypothèses pour déduire des conséquences. 2, 24, 25, 27, 28, 32

**recherche arborescente monte-carlo** Algorithme de recherche dans les arbres qui combine la recherche arborescente et l'échantillonnage Monte-Carlo pour trouver une solution à un problème. Il repose sur l'estimation de la valeur des nœuds de l'arbre par des simulations aléatoires. 7

**reconnaissance de motifs** En programmation fonctionnelle, la reconnaissance de motifs (pattern matching en anglais) est une technique qui permet de décomposer une structure de données en ses parties constituantes. Ainsi si on a une liste, on peut la décomposer en sa tête et sa queue. Si on a un arbre, on peut le décomposer en sa racine et ses sous-arbres. 26

**tactique** Dans le contexte des assistants de preuve, une tactique est une fonction qui modifie l'état d'une preuve, en ajoutant des hypothèses, en déduisant des conséquences, en simplifiant des formules, etc. 13, 14, 16, 17, 18, 22, 24, 29, 32, 33, 34, 36

**token** Dans le contexte des modèles de langage, un token est une unité de base de la séquence de texte. Par exemple, un token peut être un mot, un caractère, ou une sous-séquence d'un mot. 6

# Acronymes

**ATP** prouveur automatique.
*Glossaire :* prouveur automatique, 10, 11, 20

**IA** intelligence artificielle.
*Glossaire :* intelligence artificielle, 6, 7, 8, 15, 36

**ITP** prouveur interactif.
*Glossaire :* prouveur interactif, 10, 11, 13, 16, 17, 19, 20, 23

**LLM** grand modèle de langage.
*Glossaire :* grand modèle de langage, 6, 10, 15, 16, 17, 19, 21, 22, 23, 35

**MCTS** recherche arborescente monte-carlo.
*Glossaire :* recherche arborescente monte-carlo, 7, 19, 20, 21

**RL** apprentissage par renforcement.
*Glossaire :* apprentissage par renforcement, 15, 17, 20

# Bibliographie


[1] Martin Davis et Hilary Putnam. « A computing procedure for quantification theory ». In : *Journal of the ACM* 7.3 (juill. 1960), p. 201-215. DOI : 10.1145/321033.321034. URL : https://doi.org/10.1145/321033.321034.

[2] *Deep Learning for Symbolic Mathematics*. Déc. 2019. URL : http://arxiv.org/abs/1912.01412.

[3] *Discovering Lyapunov functions with transformers*. Oct. 2023. URL : https://mathai2023.github.io/papers/13.pdf.

[4] *Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs*. URL : http://arxiv.org/abs/2210.12283.

[5] *DT-Solver: Automated Theorem Proving with Dynamic-Tree Sampling Guided by Proof-level Value Function*. Juin 2023. URL : https://aclanthology.org/2023.acl-long.706.

[6] *Formal Mathematics Statement Curriculum Learning*. Jan. 2022. URL : http://arxiv.org/abs/2202.01344.

[7] Siddhartha Gadgil. *Forward Reasoning in Lean 4*. Mars 2022. URL : https://siddhartha-gadgil.github.io/automating-mathematics/posts/forward-reasoning-in-lean4/.

[8] *Generative Language Modeling for Automated Theorem Proving*. URL : http://arxiv.org/abs/2009.03393.

[9] Zhibin Gou et al. « ToRA: A Tool-Integrated Reasoning Agent for Mathematical Problem Solving ». In : *CoRR* abs/2309.17452 (2023). DOI : 10.48550/ARXIV.2309.17452. arXiv : 2309.17452. URL : https://doi.org/10.48550/arXiv.2309.17452.

[10] Edward J. Hu et al. *LORA: Low-Rank adaptation of Large Language Models*. Juin 2021. URL : https://arxiv.org/abs/2106.09685.

[11] *HyperTree Proof Search for Neural Theorem Proving*. Mai 2022. URL : http://arxiv.org/abs/2205.11491.

[12] *LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*. URL : http://arxiv.org/abs/2306.15626.

[13] *Learning advanced mathematical computations from examples*. Juin 2020. URL : http://arxiv.org/abs/2006.06462.

[14] *LEGO-Prover: Neural Theorem Proving with Growing Libraries*. Juin 2024. URL : http://arxiv.org/abs/2310.00656.

[15] Zhaoyu Li et al. « A Survey on Deep Learning for Theorem Proving ». In : *arXiv preprint arXiv:2404.09939* (2024). URL : https://arxiv.org/pdf/2404.09939.

[16] *Linear algebra with transformers*. URL : http://arxiv.org/abs/2112.01898.

[17] *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Déc. 2017. URL : http://arxiv.org/abs/1712.01815.

[18] *Proof Artifact Co-training for Theorem Proving with Language Models*. URL : http://arxiv.org/abs/2102.06203.

[19] *Solving olympiad geometry without human demonstrations*. URL : https://www.nature.com/articles/s41586-023-06747-5.

[20] Alan Turing. *Intelligent Machinery (1948)*. Sept. 2004. DOI : 10.1093/oso/9780198250791.003.0016. URL : https://doi.org/10.1093/oso/9780198250791.003.0016.

[21] Di Zhang et al. *Accessing GPT-4 level Mathematical Olympiad Solutions via Monte Carlo Tree Self-refine with LLaMa-3 8B*. Juin 2024. URL : https://arxiv.org/abs/2406.07394.