Que signifie “vérifier” une preuve en lean ?
Étude d’un cas : le problème 344 (`PrimeGaps`) du dépôt `openai/math`
Denise Vella-Chemla pilotant l’ia claude, 18 octobre 2026

En logique, on peut voir un théorème comme le fait de démontrer une implication de la forme : “si prémisses alors conclusion”. En lean, on peut se dire, grosso modo, qu’il s’agit de démontrer un théorème (principal) en “remontant” de la conclusion aux prémisses, en démontrant chaque prémisse, selon une sorte de parcours d’arbre qu’on appelle *en profondeur d’abord* en informatique, et si toutes les prémisses (i.e. tous les théorèmes que les prémisses contiennent) ont été démontrées, on peut considérer que la conclusion est démontrée aussi, et donc le théorème principal qui était notre objectif. Or il s’avère qu’en logique binaire, on a que `faux` implique tout. C’est un peu malheureux mais c’est comme ça. Si je pose comme prémisse que la terre est plate, alors je peux en déduire absolument tout ce que je veux. Ceci se représente par l’usage du mot magique “`sorry`” en lean, qui signifie en quelque sorte, considérons qu’à cet endroit-là de l’arborescence des dépendances des théorèmes entre eux, on a, par une chance inouïe, abouti à un théorème dont on sait qu’il est juste, dur comme le marbre, et donc écoutez, plutôt que d’aller le chercher et de le coller là, on va se dire qu’il est garanti, on met une prémisse fausse (le mot `sorry` en prémisse pour ce théorème-là, c’est bon, on peut le considérer comme démontré), et le tour est joué pour ce théorème, continuons à examiner la véracité des autres théorèmes de notre arborescence, les théorèmes (secondaires) dont notre théorème (principal) dépend.

Je ne sais pas si j’explique comme il faut, mais c’est comme ça que j’interprète le “`sorry`” en langage `lean`. Ce qui me gênait, dans le dépôt `github` de la société `openai` récent, et qui prétend fournir la preuve de 719 théorèmes de mathématiques non encore démontrés, c’était le fait qu’en essayant de lire un ou deux répertoires (i.e. un répertoire est un dossier de fichier, en l’occurrence l’endroit où sont déposés tous les fichiers nécessaires à la démonstration des deux théorèmes en question, en fait “presque” tous, et le bât pourrait blesser là aussi), j’ai réalisé que pour les deux répertoires en question, en début du programme `lean` sensé démontrer le théorème, il y avait de très nombreuses lignes de code, qui commençaient par le mot “`import`” et qui signifiaient donc que la preuve du théorème s’appuyait sur de nombreux autres théorèmes [1]. Et je me disais qu’il suffisait qu’un petit `sorry` se soit glissé quelque part bien caché, pour que toute la preuve tombe par terre. C’est donc ce que j’ai étudié ci-dessous avec l’ia claude, qui est produite par une société concurrente d’openai, pour tester les arborescences d’appels de théorèmes et repérer si un `sorry` s’était caché quelque part. Je divulgâche la fin : aucun `sorry` ne s’est caché dans la preuve `primegaps` testée.

L’autre question qui m’a intéressée est la suivante : supposons qu’on considère que la librairie [2], qui modélise de nombreux théorèmes de mathématiques et qui s’appelle Mathlib en lean soit “du marbre” (parce que développée dans le domaine du logiciel libre, donc sans intérêt lucratif en jeu, i.e. “pour l’honneur de l’esprit humain” tout entier dirigé vers les démonstrations mathématiques), quelle quantité de travail approximative représente le reste à charge pour les experts mathématiciens qui auront, s’ils le souhaitent - car après tout, la publication sur la toile par la société openai

1. C’est aussi ce qui se produit dans un article de mathématiques écrit par un humain : une démonstration de théorème fait appel à de nombreux autres théorèmes dont les sources (article, auteur, date, version, paragraphe, etc) doivent être fournis pour que le théorème soit admis comme correct par la communauté.

2. une librairie - *un package* -, c’est un dépôt contenant la démonstration en `lean` de théorèmes existant de mathématiques.

pour l'instant ne respecte pas le standard académique qui consiste à soumettre les preuves à une revue à comité de lecture -, la tâche de vérifier que les preuves sont justes. L'essentiel de ma discussion avec l'ia claude a été transcrit ci-dessous.

L'intégralité de la suite de cette note a été rédigée par l'ia claude.

Résumé

Le dépôt `openai/math` propose, pour une partie de ses résultats, une formalisation en lean 4. Ce texte examine un seul cas, le problème nommé `Problem344` / `PrimeGaps`, pour expliquer ce que la vérification par lean garantit, ce qu'elle ne garantit pas, et ce qu'un lecteur humain doit encore relire. Les vérifications décrites ont été faites sur un ordinateur personnel et peuvent être refaites par n'importe qui (voir la section *Reproduire la vérification*).

Avertissement. L'assistant qui a aidé à rédiger ce texte, l'ia claude, est développé par anthropic, société concurrente d'openai. Le texte s'en tient à des faits vérifiables et reproductibles ; il ne porte aucun jugement sur les intentions d'openai. Ce qui relève d'une appréciation est signalé comme tel.

1. Le cas étudié

Pour ce problème, le dépôt fournit trois choses.

1. Un *fichier de défi*, `ComparatorChallenges/PrimeGaps.lean` : il contient l'énoncé du théorème et les définitions dont il dépend, mais la preuve y est remplacée par le mot-clé `sorry`. En lean, `sorry` est un joker : il dit "admets ce but sans preuve". Un fichier contenant `sorry` est accepté, avec un simple avertissement, mais le théorème n'y est *pas* démontré. Dans un fichier de défi, c'est normal : on y lit l'énoncé à démontrer, la preuve se trouve ailleurs.
2. Un fichier de configuration, `PrimeGaps.json`, qui désigne le module contenant la solution (`OAI.NumberTheory.PrimeGaps.RatioCorollary`), le nom du théorème (`OAI.Problem344.large_gaps_and_ratio_density`) et la liste des seuls axiomes permis : `propext`, `Quot.sound`, `Classical.choice`.
3. La *solution* : 56 fichiers lean, totalisant 22 533 lignes (20 350 lignes une fois retirés les lignes vides et les commentaires) ; le plus long, `TaylorBounds.lean`, compte 591 lignes. Hors de ces fichiers, la preuve ne s'appuie que sur Mathlib, la bibliothèque mathématique communautaire de lean.

2. Ce qui a été vérifié

Vérification	Méthode	Résultat
Aucun `sorry` dans la preuve	Script Python qui calcule la fermeture des imports à partir de `RatioCorollary` (56 fichiers) et cherche le mot `sorry` hors commentaires	Aucune occurrence
Aucun axiome ajouté	Recherche textuelle de `axiom`, `opaque`, `native_decide` dans le dossier	Aucune occurrence
La preuve compile	`lake build` avec lean 4.34.1 et la version de Mathlib exigée par le dépôt	Réussite, sans erreur
Axiomes réellement utilisés	`#print axioms` sur le théorème	`propext`, `Classical.choice`, `Quot.sound`
Même énoncé que le défi	Comparaison textuelle des six définitions et de l'énoncé du théorème, entre le fichier de défi et la solution	Identiques

Ce qui n'a pas été fait. L'outil officiel de comparaison (Comparator) n'a pas été exécuté ; la comparaison des énoncés a été faite à la main, par un programme de comparaison de texte. Un second vérificateur indépendant du noyau de lean n'a pas été utilisé (le fichier de configuration l'a d'ailleurs désactivé : "`enable_nanoda`" : `false`). Enfin, le preprint (le fichier .pdf) correspondant n'a pas été lu : la correspondance entre l'énoncé formalisé et le problème mathématique d'origine n'a donc pas été contrôlée ici.

3. Ce que garantit lean : un vérificateur, pas un relecteur

Une preuve lean n'est pas un texte à lire mais un objet, un "terme", que le *noyau* de lean contrôle pas à pas selon les règles de sa logique. Les tactiques (les commandes qui aident à construire la preuve) et, le cas échéant, l'intelligence artificielle qui a produit le fichier, *ne sont pas dignes de confiance par elles-mêmes* : peu importe comment la preuve a été trouvée, seul compte que le noyau l'accepte. C'est la différence majeure avec un article classique, où un comité doit lire chaque étape.

Il en résulte que, lorsque le noyau accepte la preuve, *personne n'a besoin de la lire* pour savoir qu'elle démontre bien l'énoncé tel qu'il est écrit en lean. Cette vérification est mécanique et se refait à volonté.

4. Faut-il alors relire les 22 533 lignes ?

Non, et c'est le point central. Mais cela ne veut pas dire que plus rien n'est à lire. La confiance repose sur un petit nombre d'éléments, dont certains seulement relèvent d'une relecture humaine.

Élément	Taille	Qui le contrôle
La preuve (56 fichiers)	22 533 lignes	Le noyau de lean ; aucun humain n'a à la lire
L'énoncé et ses définitions	moins de 50 lignes	Un lecteur humain : c'est la partie à relire
Les définitions de Mathlib utilisées dans l'énoncé (`Nat.nth`, `Nat.Prime`, `Real.log`, `sSup`...)	courtes	Communauté Mathlib, code public ; un humain peut les vérifier
Les trois axiomes (`propext`, `Quot.sound`, `Classical.choice`)	3 principes	Standard en mathématiques classiques ; ils s'affichent par `print axioms`
Le noyau de lean lui-même (et l'ordinateur qui l'exécute)	programme public	Communauté lean, code public ; point de confiance irréductible

L'analogie avec l'avion. Le rapprochement avec les logiciels de commande de vol (Esterel et ses successeurs) est juste[3] sur un point : dans les deux cas, on ne contrôle pas tout à la main, on s'appuie sur un petit noyau auquel on a des raisons de se fier. Il y a pourtant une différence : la vérification de lean peut être refaite, à tout moment, par n'importe qui disposant d'un ordinateur, sur le même texte. Et une certitude absolue n'existe dans aucun des deux cas : les articles relus par des comités ont, eux aussi, parfois contenu des erreurs restées longtemps inaperçues.

Les deux scénarios. Faut-il donc que des experts relisent tout, ou bien peuvent-ils s'en remettre à lean ? D'après ce qui précède, c'est le second scénario qui s'applique, avec ces conditions :

(i) faire confiance au noyau de lean et à Mathlib, projets dont le code est public, développés en communauté (lean avec le soutien d'une organisation à but non lucratif, la lean FRO, à ma connaissance) ;

(ii) relire l'énoncé et ses définitions, ce qui est ici l'affaire de quelques dizaines de lignes. Ce qui reste en dehors de lean est donc petit, mais ce n'est pas nul, et cela demande la lecture d'un mathématicien professionnel.

5. Les pièges que la relecture humaine doit chercher

Un énoncé peut être correctement typé, démontré, et pourtant ne pas dire ce qu'on croit.

Voici trois exemples précis de tels cas :

- **Soustraction d'entiers.** Dans $\mathbb{N}$, lean calcule $3 - 5 = 0$. Si l'écart $p_{n+1} - p_n$ était calculé avant conversion en réels, la soustraction serait tronquée. Ici les deux termes sont d'abord convertis en réels : le piège est évité.
- **Division par zéro.** En lean, $a/0 = 0$ pour tout réel a. Ici, les quotients sont $\mathit{compte}(N)/N$ et p_n/n. Le seul cas délicat serait $N = 0$, qui donnerait $\mathit{compte}(0)/0 = 0$. Mais la définition de la densité ne retient que les $N \geq N_0$, et l'on peut toujours prendre $N_0 \geq 1$. Ce cas n'intervient donc pas.

3. l'auteure a fourni cet exemple à l'ia claude

- **Bornes supérieures.** La densité inférieure est définie par un `sSup` sur $\mathbb{R}$, et en lean la borne supérieure d'un ensemble vide ou non majoré vaut 0. Il faut donc s'assurer que l'ensemble en question est non vide et majoré. C'est le cas ici (il contient 0, et il est majoré par 1), et la quantité obtenue est la limite inférieure usuelle de compte$(N)/N$.

Ces vérifications sont du ressort d'un lecteur humain : la machine les ignore, puisque l'énoncé tel qu'écrit est cohérent.

6. Sur la notion de "publié"

Le contrôle par un comité de lecture remplit deux fonctions distinctes :

(1) s'assurer que la preuve est correcte,

(2) juger de l'énoncé, de sa nouveauté et de son exposition.

La vérification par lean prend en charge la fonction (1) pour l'énoncé *formalisé*. Elle ne dit rien de la fonction (2), ni du fait que l'énoncé formalisé coïncide avec le problème d'origine. De même, lean démontre l'énoncé, mais ne garantit pas que la preuve suive le même chemin que l'argument du preprint fourni dans le répertoire des preprints au format texte (pdf) : ce sont deux textes distincts, et leur correspondance n'est pas contrôlée par le noyau.

Dans ce cadre, un comité n'aurait plus à lire plusieurs dizaines de milliers de lignes, mais à relire l'énoncé, ses définitions, et à juger du reste. Que la communauté mathématique accepte ou non ce mode de validation comme équivalent à une publication relue est une question ouverte, qui n'est pas tranchée ici.

7. Limites

- lean est un logiciel : des défauts du noyau ont été signalés puis corrigés dans des assistants de preuve, lean compris, à ma[4] connaissance. Ils sont rares, mais ils font partie de la base de confiance. Cette question mérite un traitement séparé.
- Les vérifications ci-dessus portent sur un seul problème parmi les centaines du dépôt ; rien n'en découle pour les autres sans les refaire.
- Les jugements sur la fidélité des définitions à l'esprit du problème sont des lectures, et non des preuves.

8. L'énoncé en `lean` pour le seul répertoire auquel on s'est intéressé

Voici les définitions et l'énoncé du fichier de défi (la preuve y est remplacée par `sorry`). C'est l'essentiel de ce qu'un lecteur humain doit relire.

4. Ce pronom possessif doit être attribué à l'ia claude, qui parle à la première personne du singulier ici.

```
-
noncomputable def prime (n : ℕ) : ℕ := Nat.nth Nat.Prime (n - 1)

noncomputable def largeGapIndices (C : ℝ) (N : ℕ) : Finset ℕ := by
  classical
  exact (Finset.Icc 1 N).filter fun n =>
    C * Real.log (prime n : ℝ) < (prime (n + 1) : ℝ) - (prime n : ℝ)

def primeAt (n : ℕ) : ℕ := Nat.nth Nat.Prime (n - 1)

def initialCount (A : Set ℕ) (N : ℕ) : ℕ := by
  classical
  exact ((Finset.Icc 1 N).filter (fun n => n ∈ A)).card

def lowerAsymptoticDensity (A : Set ℕ) : ℝ :=
  sSup {d : ℝ | ∃ N_0 : ℕ, ∀ N : ℕ, N_0 ≤ N →
    d ≤ (initialCount A N : ℝ) / (N : ℝ)}

def ratioIncreaseIndices : Set ℕ :=
  {n | 1 ≤ n ∧
    (primeAt n : ℝ) / (n : ℝ) <
      (primeAt (n + 1) : ℝ) / ((n + 1 : ℕ) : ℝ)}

theorem large_gaps_and_ratio_density :
    (∀ C : ℝ, 0 < C →
      ∃ c : ℝ, 0 < c ∧ ∃ N_0 : ℕ, ∀ N : ℕ, N_0 ≤ N →
        c * (N : ℝ) ≤ ((largeGapIndices C N).card : ℝ)) ∧
    0 < lowerAsymptoticDensity ratioIncreaseIndices := by
  sorry
```

En clair. Notons p_n le n-ième nombre premier ($p_1 = 2$). L'énoncé affirme deux choses.

1. Pour toute constante $C > 0$, il existe $c > 0$ tel que, pour N assez grand, au moins cN des indices $n \leq N$ vérifient $p_{n+1} - p_n > C \log p_n$: les grands écarts entre nombres premiers consécutifs ont une densité inférieure strictement positive.
2. L'ensemble des $n \geq 1$ tels que $p_n/n < p_{n+1}/(n+1)$ (le rapport p_n/n croît) a une densité inférieure strictement positive.

(Dans le fichier de défi, l'énoncé fait appel à :

- `LargePrimeGaps.largeGapIndices` et à
- `LargePrimeGaps.prime`,

définis dans un espace de noms voisin ; on les a écrits ici sans préfixe pour alléger.)

9. Reproduire la vérification

Les vérifications ont été faites sous Windows. Voici la marche suivie, dans l'ordre.

1. Récupérer le dépôt : `git clone --depth 1 https://github.com/openai/math`. (Le téléchargement du fichier au format `zip` depuis le navigateur a échoué sur la machine utilisée ; le clonage avec `Git` a fonctionné.)
2. Installer lean via `elan` ; le dépôt exige lean 4.34.1, que `elan` télécharge automatiquement.
3. Lancer le script d'audit des imports, depuis la racine du dépôt :
   `python audit_lean_imports.py OAI.NumberTheory.PrimeGaps.RatioCorollary`.
   Les imports signalés "non résolus" sont tous `Mathlib`, qui n'est pas parcouru.
4. Chercher les axiomes ajoutés, dans le dossier `PrimeGaps`.
5. Compiler. Le projet complet du dépôt dépend d'une dizaine de bibliothèques ; l'une d'elles (AINTLIB) contient un fichier dont le nom comporte le caractère " :", interdit par Windows, ce qui fait échouer l'extraction. Comme la preuve de `PrimeGaps` n'utilise que Mathlib, on a créé un projet minimal ne dépendant que de Mathlib (à la révision `d13f23b723b8a846827a245b89c10fc7d3f11612`, lue dans le manifeste du dépôt), contenant la copie du dossier `PrimeGaps`.
   Dans ce projet : `lake exe cache get`, puis
   `lake build OAI.NumberTheory.PrimeGaps.RatioCorollary`.
6. Contrôler les axiomes : créer un fichier `Test.lean` contenant

```
import OAI.NumberTheory.PrimeGaps.RatioCorollary
#print axioms OAI.Problem344.large_gaps_and_ratio_density
```

puis lancer `lake env lean Test.lean`. Le résultat obtenu est :

```
'OAI.Problem344.large_gaps_and_ratio_density' depends on axioms:
[propext, Classical.choice, Quot.sound]
```

Si un autre nom y figurait, notamment `sorryAx`, ce serait le signe d'un `sorry` caché dans la chaîne de dépendances.

10. Annexe

Les fichiers en python fourni par l'ia claude sont téléchargeables ici :

- https://denisevellachemla.eu/audit-lean-imports.pdf ;
- https://denisevellachemla.eu/preparer-verif-primegaps.pdf ;
- https://denisevellachemla.eu/comptage-lignes-preuves.pdf

Le résultat à l'écran du comptage de lignes des différentes dépôts est fourni ci-dessous ; Le nombre de lignes des différentes preuves est visible à l'adresse
https://denisevellachemla.eu/comptage-lignes.pdf .

```
C:\Users\Denise_Vella\Desktop\math-main>python compter_lignes_preuves.py
Problemes analyses : 416  (ignores : 0)
Fichiers par preuve : mediane 77, max 4774
Lignes de code par preuve : mediane 19610, moyenne 55275, max 2359585
Preuves avec au moins un sorry : 0
Preuves avec axiom/opaque/native_decide : 14

Les 10 plus longues (lignes de code) :
  MUBSix                                3381 fichiers  2359585 lignes
  CycleCliqueRamsey                      443 fichiers  1896491 lignes
  ErdosReciprocal                       4774 fichiers   987564 lignes
  PlanarPacking                          563 fichiers   908568 lignes
  Catalan                                988 fichiers   540774 lignes
  HeckeSevenEighths                     2925 fichiers   441126 lignes
  DirichletSevenEighths                 2924 fichiers   441096 lignes
  QuasiRiemannHypothesis                2924 fichiers   441096 lignes
  OstmannPrimes                         4358 fichiers   315939 lignes
  PattersonFirstMoment                  3594 fichiers   264512 lignes

PrimeGaps : {'fichiers': 56, 'lignes_totales': 22533, 'lignes_de_code': 20350,
                    'plus_long_fichier': 591, 'lignes_enonce_defi': 31, 'sorry': 0}

Tableau complet : C:\Users\Denise_Vella\Desktop\math-main\comptage_lignes.csv
Rappel : Mathlib et les autres biblioth ques externes ne sont pas compt es.
```

Si l'on considère le théorème primegaps par exemple, il y a environ 30 lignes à vérifier, celles de l'énoncé et de ses définitions.

- Le fichier de défi `ComparatorChallenges/PrimeGaps.lean` compte 47 lignes en tout. Sans les lignes vides ni les import, il en reste 31 lignes de code. Elles contiennent les six définitions, l'énoncé du théorème, le sorry de remplacement et quelques lignes de structure (`namespace, end, noncomputable section`).
- La solution répète le même énoncé, mot pour mot. Je l'ai comparé par programme, donc personne n'a besoin de le relire une deuxième fois.

À cela s'ajoute la vérification que les notions de Mathlib employées dans ces définitions (`Nat.nth, Nat.Prime, Real.log, sSup, Finset.filter`) ont bien le sens attendu. Ce sont des définitions courtes et connues, mais elles ne sont pas dans les 31 lignes.

À titre de comparaison, la preuve que Lean contrôle compte 22 533 lignes (20 350 de code) : personne n'a à les lire.