Faire des mathématiques formalisées ?

École thématique XIII (2026)

Antoine Chambert-Loir

22–26 juin 2026

Université Paris Cité, IMJ-PRG

Plan

Qu'est-ce qu'une preuve formelle ?
Mathématiques en théorie des types
Faire des mathématiques dans le logiciel Lean

Qu’est-ce qu’une preuve formelle ?

Démonstration et démonstration formalisée

Depuis les Grecs, qui dit mathématique dit démonstration ; certains doutent même qu'il se trouve, en dehors des mathématiques, des démonstrations au sens précis et rigoureux que ce mot a reçu des Grecs et qu'on entend lui donner ici.

L'analyse du mécanisme des démonstrations dans des textes mathématiques bien choisis a permis d'en dégager la structure, du double point de vue du vocabulaire et de la syntaxe. On arrive ainsi à la conclusion qu'un texte mathématique suffisamment explicite pourrait être exprimé dans une langue conventionnelle ne comportant qu'un petit nombre de « mots » invariables assemblés suivant une syntaxe qui consisterait en un petit nombre de règles inviolables : un tel texte est dit *formalisé.*

— N. Bourbaki (1968), *Théorie des ensembles*, Introduction

Vers une langue conventionnelle

Gottfried Wilhelm Leibniz, pendant toute sa vie (1040-1716) s'est occupé d'« une manière de Spécieuse Générale où toutes les vérités de raison seraient réduites à une façon de calcul. Ce pourrait être en même temps une manière de Langue ou d'Écriture universelle, mais infiniment différente de toutes celles qu'on a projetées jusqu'ici ; car les caractères, et les paroles mêmes, y dirigeraient la Raison ; et les erreurs, excepté celles de fait, n'y seraient que des erreurs de calcul. Il serait très difficile de former ou d'inventer cette langue ou caractéristique ; mais très aisé de l'apprendre sans aucun dictionnaire » (*Opera philosophica*, a. 1840, p. 701).

— G. Peano (1896), *Introduction au tome 2 du « formulaire de mathématiques »*

Vers une langue conventionnelle

Gottfried Wilhelm Leibniz, pendant toute sa vie (1040-1716) s'est occupé d'« une manière de Spécieuse Générale où toutes les vérités de raison seraient réduites à une façon de calcul. Ce pourrait être en même temps une manière de Langue ou d'Écriture universelle, mais infiniment différente de toutes celles qu'on a projetées jusqu'ici ; car les caractères, et les paroles mêmes, y dirigeraient la Raison ; et les erreurs, excepté celles de fait, n'y seraient que des erreurs de calcul. Il serait très difficile de former ou d'inventer cette langue ou caractéristique ; mais très aisé de l'apprendre sans aucun dictionnaire » (*Opera philosophica*, a. 1840, p. 701).

(…) Après deux siècles, ce « songe » de l'inventeur du Calcul infinitésimal est devenu une réalité.

— G. Peano (1896), *Introduction au tome 2 du « formulaire de mathématiques »*

Vers une langue conventionnelle

Axiomata.

1. $1 \in N.$
2. $a \in N. \supset : a = a.$
3. $a, b \in N. \supset : a = b. = .b = a.$
4. $a, b, c \in N. \supset \therefore a = b. = .b = c :\supset .a = c.$
5. $a = b.b \in N :\supset .a \in N.$
6. $a \in N. \supset .a + 1 \in N.$
7. $a, b \in N. \supset : a = b. = .a + 1 = b + 1.$
8. $a \in N. \supset .a + 1 - = 1.$
9. $k \in K \therefore 1 \in k \therefore x \in N.x \in k :\supset_x .x + 1 \in k ::\supset .N \supset k.$

— G. Peano (1889), *Arithmetices principia. Nova methodo exposita*

À l'ère des ordinateurs

In 1966 I had to check a long proof in combinatorics, full of repetitions of a small number of simple ideas, and I felt the need to build a computer program to do the checking.

— N. G. de Bruijn (2003), *Memories of the Automath project*

À l'ère des ordinateurs

In 1966 I had to check a long proof in combinatorics, full of repetitions of a small number of simple ideas, and I felt the need to build a computer program to do the checking.

— N. G. de Bruijn (2003), *Memories of the Automath project*

The Automath forces us to a final and merciless check of all mathematical texts that we wish to submit to it. (…) A long proof with lots of uninteresting details might get its ultimate formulation in Automath code. And then its correctness will not just be a matter of confidence like so often in our present world; it can be established beyond any doubt, and whenever one wants.

— N. G. de Bruijn (1967), « Verification of mathematical proofs by a computer »

À l'ère des ordinateurs

De nombreux systèmes de vérification de preuves mathématiques :

- Automath, N. G. De Bruijn (1967)
- Mizar, A. Trybulec (1973)
- Isabelle, L. Paulson (1986)
- Coq/Rocq, G. Huet (1989)
- Agda, C. Coquand (1999)
- HOL Light, John Harrison (2000)
- Lean, L. de Moura (2013)

et bien d'autres encore…

À l'ère des ordinateurs

De nombreux systèmes de vérification de preuves mathématiques
— avec des résultats parfois fantastiques :

- Automath, N. G. De Bruijn (1967)
- Mizar, A. Trybulec (1973)
- Isabelle, L. Paulson (1986)
- Coq/Rocq, G. Huet (1989)
  - Théorème des 4 couleurs (Gonthier, 2004)
  - Théorème de Feit-Thompson (Gonthier et al., 2012)
- Agda, C. Coquand (1999)
- HOL Light, John Harrison (2000) — conjecture de Kepler (Hales, 2015)
- Lean, L. de Moura (2013)
  - Liquid Tensor Experiment (Commelin, Scholze, Topaz, 2022)
  - Retournement de la sphère (Massot et al., 2024)

et bien d'autres encore...

Bibliothèques mathématiques

Ces systèmes sont souvent accompagnés d'une *bibliothèque mathématique* contenant des définitions et résultats de base, et parfois très avancés.

- Mizar : Mizar Mathematical Library
- Isabelle : Archive of formal proofs
- Coq/Rocq : Mathematical components / MathComp Analysis
- Agda : unimath, TypeTopology…
- Lean : Mathlib

Bibliothèques mathématiques

Ces systèmes sont souvent accompagnés d'une *bibliothèque mathématique* contenant des définitions et résultats de base, et parfois très avancés.

- Mizar : Mizar Mathematical Library (?)
- Isabelle : Archive of formal proofs (5 Mo lignes de code, 600 auteurs)
- Coq/Rocq : Mathematical components / MathComp Analysis (500 000 lignes de code au total)
- Agda : unimath, TypeTopology…
- Lean : Mathlib (2.5 Mo lignes de code, 800 auteurs)

Tactiques

Les *tactiques* sont des algorithmes visant à automatiser une partie du raisonnement. Lorsqu'elles aboutissent, ils fournissent une preuve du résultat voulu, mais elles peuvent mettre longtemps à aboutir, et même échouer.

- simplification de formules
- preuves d'égalité dans des structures calculables (comme les nombres entiers)
- preuves d'égalité dans des structures axiomatisées (anneaux, groupes abéliens, modules…)
- tactiques « logiques » (solveur SAT, programmation linéaire, etc.)
- récriture automatique de définitions/théorèmes dans d'autres contextes (par exemple, version « groupes additifs » d'un énoncé pour les « groupes multiplicatifs »).

En fait, la plupart du travail se fait au moyen de tactiques : on écrit un programme qui écrit une preuve, si bien qu'en un certain sens, on écrit rarement une preuve formalisée.

Mathématiques en théorie des types

Logique

En principe, tout système logique peut donner lieu à un système de formalisation mécanique. Les trois ingrédients nécessaires sont :

- une *syntaxe* permettant de formuler des énoncés mathématiques ;
- une définition de *preuve formelle*, permettant de justifier qu'un énoncé est démontré ;
- un *langage de programmation* permettant de construire automatiquement énoncés et preuves.

Théorie des ensembles

La syntaxe de la *théorie des ensembles* est une possibilité ; les preuves formelles, à la Hilbert, sont des suites d'énoncés tels que chacun se déduisent des précédents par des règles simples.

Tout *terme* de la théorie désigne un ensemble ; deux ensembles sont éventuellement liés par la relation $\in$ (appartenir) : $a \in A$ signifie que l'ensemble a est un élément de l'ensemble A.

On introduit également un symbole d'égalité, avec l'axiome d'extensionalité.

Des *axiomes* affirment l'existence de l'ensemble des nombres entiers, fournissent des règles pour construire de nouveaux ensembles à partir d'anciens (ensemble des parties, remplacement, choix…).

L'efficacité de ce système vient que (presque) toutes les structures mathématiques étudiées sont naturellement et facilement modélisées par des ensembles. (La théorie des catégories supérieures est un contre-exemple.)

Théorie des types

Les systèmes développés récemment (dont Lean, Rocq, Agda...) reposent sur une *théorie des types* qui permettent de mettre *objets* et *preuves* sur le même pied. Le langage de programmation peut être le même (Lean), ou pas (Rocq utilise Ocaml)

La théorie des types est une façon plus structurée de modéliser les mathématiques. Elle a été introduite par Russell et Whitehead (1910–1930), revisitée plusieurs fois : Martin-Löf (1960), Huet et Coquand (1988), Voevodsky (2010)...

Dans un tel système, tout objet est supposé avoir un type et la logique est organisée en *jugements de typage* $a : A$ pour dire que le terme a est un objet de type A.

Les axiomes de théories des types fournissent des règles pour construire des types, des jugements de types, et les relier.

Intérêt des théories des types

- Ce sont des théories structurées qui rendent impossible d'écrire des énoncés sans contenu réel comme 3 est une topologie sur 2 » ou « 42 est un groupe ».
- Elles permettent de représenter les objets mathématiques et les théorèmes sur le même plan : un théorème T est un terme particulier (de type Prop), et une preuve de ce théorème est un terme t de type T. La validation du théorème T et de sa preuve t signifie que l'ordinateur reconnaît la construction du terme T et le jugement de typage $t : T$.
- Elles donnent aux *fonctions* un rôle primordial, et cela leur fournit un contenu calculatoire. En particulier, elles permettent de modéliser les *mathématiques constructives* où les preuves d'existence fournissent un témoin.

Calcul des séquents

Les règles de la théorie des types ont toutes la même forme $\Gamma \vdash t : T$, où

- Γ est un *contexte*, c'est-à-dire une liste de variables x_i et de jugements de typages $t_j : T_j$. (Les seules variables libres des termes t_j et T_j sont parmi les x_i.)
- $t : T$ est un jugement de typage.

Cette règle dit que le jugement de typage $t : T$ est valide dans le contexte Γ.

Par exemple, la règle

$$A, B : \mathrm{Type}, f : A \to B, a : A \quad \vdash \quad fa : B$$

signifie que si A et B sont des types, f une fonction du type A dans le type B, et a un objet de type A, alors fa est un objet de type B.

Règles d'inférence

Les *règles d'inférence* permettent de construire de nouveaux séquents. Ils prennent la forme

$$\frac{S_1 \quad S_2 \quad \cdots \quad S_m}{S'}$$

où $S_1, ..., S_m, S'$ sont des séquents, la règle signifiant que la conjonction des séquents supérieurs (les prémisses) permet d'écrire le séquent inférieur (la conclusion). La règle précédente est plutôt écrite

$$\frac{\Gamma \vdash a : A \quad \Delta \vdash f : A \to B}{\Gamma, \Delta \vdash fa : B}$$

Construction de types : fonctions

Alors qu'en théorie des ensembles, la notion de fonction est secondaire, définie à partir de couples et de graphes, c'est une notion primitive en théorie des types.

Étant donnés deux types A et B, la théorie postule l'existence d'un type $A \to B$ axiomatisé par quatre familles de règles :

- règle de formation : comment construire le type $A \to B$ des fonctions de A dans B ;
- règle d'introduction : comment construire une fonctions $f : A \to B$;
- règle d'élimination : si $f : A \to B$ et $a : A$, alors $f(a) : B$;
- règle de calcul : régit le comportement des règles précédentes.

Construction de types : fonctions

La règle de *formation* affirme l'existence du type des fonctions : si $\Gamma \vdash A : \mathrm{Type}$ et $\Delta \vdash B : \mathrm{Type}$, alors $\Gamma, \Delta \vdash A \to B : \mathrm{Type}$.

La règle d'*introduction* permet de construire des fonctions : si $\Gamma, x : A \vdash t(x) : B$, alors $\Gamma \vdash \lambda x.t(x) : A \to B$ est la fonction définie par le terme t.

La règle d'*élimination* justifie le comportement de la règle d'introduction : si

$$\Gamma \vdash f : A \to B$$

and $\Delta \vdash a : A$, elle affirme $\Gamma, \Delta \vdash fa : B$, c'est-à-dire l'existence d'un terme fa de type B.

Deux règles de *calcul* :

- La règle β affirme que si f est le terme $\lambda x.t(x)$ et $a : A$, alors fa est le terme $t(a)$ obtenu en substituant x par a dans le terme $t(x)$;
- Pour $f : A \to B$, la règle η affirme que f est la fonction $\lambda x.f(x)$.

Produits dépendants

En fait, la théorie de Martin-Löf est une théorie des *types dépendants*, c'est-à-dire que les types eux-mêmes peuvent dépendre de variables.

On dit que l'on a une *famille de types* $B(x)$ dans un contexte Γ lorsque l'on a des séquents $\Gamma \vdash A : \mathrm{Type}$ et $\Gamma, x : A \vdash B(x) : \mathrm{Type}$.

La théorie postule l'existence d'un type $\Pi_{x:A} B(x)$ qui représente les fonctions f sur A telles que pour $x : A$, $f(x) : B(x)$.

Ce type est régi par des règles analogues aux précédentes. En fait, le type des fonctions $A \to B$ est le cas particulier du type $\Pi_{x:A} B$, lorsque la famille de types $B(x)$ est le type « constant » B.

Sommes dépendantes

La théorie postule également l'existence d'un type dual $\Sigma_{x:A} B(x)$ qui représente les couples formés d'un élément $a : A$ et d'un élément $b : B(a)$.

La règle de *formation* déduit de $\Gamma \vdash A : \mathrm{Type}$ et de $\Gamma, x : A \vdash B(x) : \mathrm{Type}$ l'existence du type $\Gamma \vdash \Sigma_{x:A} B(x)$.

La règle d'introduction déduit de $\Gamma \vdash a : A$ et de $\Gamma, a : A \vdash b : B(a)$ le jugement $\Gamma \vdash \langle a, b \rangle : \Sigma_{x:A} B(x)$.

Il y a deux règles d'élimination :

$$\Gamma, c : \Sigma_{x:A} B(x) \vdash p_1(c) : A \qquad \text{et} \qquad \Gamma, c : \Sigma_{x:A} B(x) \vdash p_2(c) : B(p_1(a)).$$

Les règles de calcul spécifient que :

- Pour $a : A$ et $b : B(a)$, on a $p_1(\langle a, b \rangle) = a$ et $p_2(\langle a, b \rangle) = b$;
- Pour $c : \Sigma_{x:A} B(x)$, c est $\langle p_1(c), p_2(c) \rangle$.

Logique et fonctions

Le calcul des types permet de modéliser le raisonnement logique au sens d'une correspondance entre constructions de types et connecteurs logiques:

Logique	Types
Vrai $\top$, faux $\bot$	Unit, Empty
Conjonction $p \wedge q$	Produit $P \times Q$
Disjonction $p \vee q$	Somme $P + Q$
Implication $p \Rightarrow q$	Fonction $P \to Q$
Négation $\neg p$	$P \to \mathrm{Empty}$
Quantification universelle $\forall x, p(x)$	Produit dépendant $\Pi_x P(x)$
Quantification existentielle $\forall x, p(x)$	Somme dépendante $\Sigma_x P(x)$

Dans ce tableau :

- Unit est un type qui possède exactement un constructeur, $u : \mathrm{Unit}$;
- Empty est un type qui ne possède aucun constructeur ;
- Le type Produit $P \times Q$ possède un constructeur, deux éliminateurs (les projections) et les règles de calcul correspondantes ;
- Le type Somme $P + Q$ possède deux constructeurs $\mathrm{inl} : P \to P + Q$ $\mathrm{inr} : Q \to P + Q$ et un éliminateur, pour toute famille de types $R(z)$ indexée par $z : P + Q$, donné par une fonction

$$\mathrm{ind} : \Pi_{x:P} R(\mathrm{inl}\ x) \to \Pi_{y:Q} R(\mathrm{inr}\ y) \to \Pi_{z:P+Q} R(z)$$

  vérifiant les deux règles de calcul correspondantes, pour $f : \Pi_{x:P} R(\mathrm{inl}\ x)$, $g : \Pi_{y:Q} R(\mathrm{inr}\ y)$, $a : P$ et $b : Q$, disant que $\mathrm{ind}(f)(g)(\mathrm{inl}\ a)$ est fa et $\mathrm{ind}(f)(g)(\mathrm{inr}\ b)$ est gb.

Logique et fonctions

La théorie des types ajoute un caractère calculatoire à la logique, distinct de celui de l'algèbre de Boole : il est conforme à la logique *intuitionniste*.

Par exemple, bien que la disjonction $p \vee q$ soit modelée par le type somme $P + Q$, leur comportement diffère. En effet, puisque le type $P + Q$ n'a que deux constructeurs, inl et inr, en construire un élément demande de dire s'il est dans P ou dans Q.

En particulier, prouver $P + \neg P$ demande de prouver que P vaille ou que $\neg P$ vaille : la *loi du tiers exclu* n'est pas forcément vérifiée. On peut l'ajouter comme axiome supplémentaire ; Lean l'ajoute comme conséquence d'un *axiome du choix*.

Axiome du choix

En théorie des ensembles, c'est l'énoncé suivant :

$$(\forall x \in A)(\exists y \in B(x))C(x,y) \Rightarrow (\exists f \in \Pi_{x:A} B(x))(\forall x \in A)C(x, f(x)).$$

D'après Gödel, il est compatible avec la théorie des ensembles de Zermelo–Fraenkel, mais d'après Cohen, il n'y est pas démontrable.

Sa « traduction » $\Pi_{x:A}\Sigma_{y:B(x)}C(x,y) \to \Sigma_{f:\Pi_{x:A}B(x)}\Pi_{x:A}C(x,f(x))$ en théorie des types est un théorème !

La version convenable de l'axiome du choix remplace le type $\Sigma_{y:B(x)}C(x,y)$ par une « troncation propositionnelle » Cet axiome est compatible avec la théorie des types, mais n'y est pas démontrable ; il est incompatible avec l'*axiome d'univalence* proposé par Voevodsky.

Lean suppose cet axiome du choix et, suivant Diaconescu, en déduit le principe du tiers exclus.

Types inductifs : l'exemple des nombres entiers

Le type $\mathbf{N}$ des nombres entiers naturels est axiomatisé.

La règle de formation est simplement $\vdash \mathbf{N} : \mathrm{Type}$.

Il y a deux règles d'introduction :

$$\vdash 0 : \mathbf{N} \qquad \text{et} \qquad x : \mathbf{N} \vdash \mathrm{succ}\ x : \mathbf{N}.$$

Autrement dit, on n'a que deux moyens de définir un entier, soit c'est l'entier 0, soit c'est le successeur $\mathrm{succ}\ x$ d'un entier x.

La règle d'élimination est le principe de récurrence

$$\frac{\Gamma, n : \mathbf{N} \vdash P(n) : \mathrm{Type}}{\Gamma \vdash \mathrm{ind} : P(0) \to \Pi_{n:\mathbf{N}}(P(n) \to P(\mathrm{succ}\ n)) \to \Pi_{n:\mathbf{N}} P(n)}$$

Les règles de calculs spécifient que si f est $\mathrm{ind}(a, h)$, pour $a : P(0)$ et $h : \Pi_{n:\mathbf{N}}(P(n) \to P(\mathrm{succ}\ n))$, alors $f(0)$ est a et $f(\mathrm{succ}\ n) = h(n)(f(n))$, pour tout n.

Égalités

La théorie de Martin–Löf contient deux notions d'égalité entre termes.

La première est l'égalité *définitionelle* (ou jugementale). Elle est purement calculatoire et signifie qu'après récritures itérées des définitions, les deux termes sont égaux. C'est par exemple celle que j'ai utilisé en disant que la fonction f « est » le terme $\lambda x.t(x)$.

Une seconde notion est l'égalité *propositionnelle*. C'est en fait un famille de types $x =_A y$, paramétrée par $x, y : A$, munie d'un unique constructeur $\mathrm{refl}_x : \Pi_{x:A}(x =_A x)$.

La règle d'élimination est donnée, pour tout $a : A$ et pour toute famille de types $P(x, e)$ indexée par $\Sigma_{x:A}(x =_A a)$, par une fonction

$$P(a, \mathrm{refl}_a) \to \Pi_{x,e} P(x, e).$$

Il est remarquable que cette axiomatique munisse la famille $x =_A y$ d'une structure de *groupoïde*.

Types inductifs et structures mathématiques : arbres

Les types inductifs permettent de construire les structures mathématiques usuelles de façon naturelle.

Par exemple, on peut définir les arbres binaires (à sommets dans un type A) par deux constructeurs

- $\mathrm{empty} : \mathrm{Tree}\ A$ construit l'arbre vide
- une fonction $\mathrm{node} : A \to \mathrm{Tree}\ A \to \mathrm{Tree}\ A \to \mathrm{Tree}\ A$ construit un arbre binaire à partir d'une racine et deux sous-arbres.

La règle d'élimination est la fonction, pour toute fonction $f : \mathrm{Tree}\ A \to P$,

$$f(\mathrm{empty}) \to (\Pi_{a:A}\Pi_{t:\ \mathrm{Tree}\ A}\Pi_{u:\ \mathrm{Tree}\ A} f(t) \to f(u) \to f(\mathrm{node}(a)(t)(u))) \to (\Pi_{t:\mathrm{Tree}\ A} f(t)).$$

Elle permet de construire une fonction sur $\mathrm{Tree}\ A$ à partir de sa valeur sur empty et uen formule de récurrence pour les arbres de type node.

Paradoxe de Russell

On pourrait vouloir définir des « gros arbres » qui consisteraient juste en une famille de « gros arbres » indexée par un type quelconque. Le constructeur serait

$$\mathrm{BigTree} : \left(\Sigma_{R:\ \mathrm{Type}}(R \to \mathrm{BigTree})\right) \to \mathrm{BigTree}\ .$$

Le problème est qu'un tel type fournit un paradoxe semblable au paradoxe de Russell. Considérons en effet le sous-type $\mathrm{RussellTree}$ des $t : \mathrm{BigTree}$ qui ne sont pas un sous-arbre immédiat d'eux-mêmes. Ainsi, si t est $\mathrm{BigTree}\ \langle R, f\rangle$, on a $t : \mathrm{RussellTree}$ si $\Pi_{r:R} f(r) \neq t$.

Considérons alors le « gros arbre » BT donné par $\mathrm{BigTree}\ \langle \mathrm{RussellTree}, \lambda t.t\rangle$. Ses sous-arbres immédiats sont de la forme t, pour $t : \mathrm{RussellTree}$.

Supposons que BT soit égal à un de ses sous-arbres immédiats t ; on en déduit que t est égal à BT, donc BT n'est pas un sous-arbre immédiat de lui-même.

Supposons que BT ne soit égal à aucun de ses sous-arbres immédiats. Alors $\mathrm{BT} : \mathrm{RussellTree}$ et donc il est égal à un de ses sous-arbres immédiats.

Univers

Il faut donc interdire la construction de types comme

$$\mathrm{BigTree} : (\Sigma(R : \mathrm{Type}), R \to \mathrm{BigTree}) \to \mathrm{BigTree}.$$

La théorie postule donc une suite de types (« univers ») $\mathrm{Type}_0, \mathrm{Type}_1, \ldots$ et la construction

$$\mathrm{BigTree} : (\Sigma(R : \mathrm{Type}_0), R \to \mathrm{BigTree}) \to \mathrm{BigTree}$$

définit un objet de type Type_1.

Le paradoxe précédent ne fonctionne plus car $\mathrm{RussellTree} : \mathrm{Type}_1$ et on ne peut l'utiliser pour construire BT.

Une théorie des types doit expliquer comment ces différents univers s'organisent, et il y a plusieurs possibilités : celle de Rocq les considère comme « cumulatifs », c'est-à-dire que l'on a une inclusion $\mathrm{Type}_0 \subseteq \mathrm{Type}_1 \subseteq \cdots$, tandis que celle de Lean ne postule pas de telle chaîne.

Faire des mathématiques dans le logiciel Lean

Faire des mathématiques dans le logiciel Lean

Lean est un langage de programmation fonctionnel qui implémente une théorie des types.

Les programmes y définisent des *types* et des *fonctions* qui relient ces types et éventuellement le type Prop des propositions.

La suite de l'exposé se passera directement dans le logiciel, que l'on peut expérimenter via le compilateur en ligne, en y chargeant le fichier lean.

Les nombres premiers de Fermat

Fermat a observé que les nobmres entiers F_n définis par $F_n = 2^{2^n} + 1$ étaient premiers pour $n = 1, 2, 3, 4, ...$

Vérifions cela avec Lean :

```
/-- Nombres de Fermat -/
def F (n : ℕ) : ℕ := 2 ^ (2 ^ n) + 1

-- `#check` est une fonction de Lean qui permet d'afficher un Type
#check F -- F (n : ℕ) : ℕ

-- `#eval` est une fonction de Lean qui permet d'évaluer une expression
#eval F 4 -- 65537
#eval Nat.Prime (F 4) -- true
```

Les nombres premiers de Fermat

Comme l'a observé Euler, la situation est un peu plus compliquée que cela.

```
#eval F 5 -- 4294967297
#eval Nat.Prime(F 5) -- false
#eval Nat.minFac (F 5) -- 641
```

Peut-être un peu paradoxalement pour un logiciel de vérification mathématique, ces évaluations sont faites en utilisant la bibliothèque GMP de calcul multi-précision, c'est-à-dire qu'elles ne sont pas certifiées.

Pour disposer d'une preuve que F_5 n'est pas premier, on peut accepter se contenter de cela.

```
theorem eulerFermat5 : ¬ (Nat.Prime (F 5)) := by
  simp [F]; norm_num
```

Les nombres premiers de Fermat

La démonstration d'Euler ne demande pas de calculs compliqués. Des relations

$$641 = 5 \cdot 2^7 + 1 = 5^4 + 2^4,$$

on déduit les congruences $5 \cdot 2^7 \equiv -1$ et $5^4 \equiv -2^4$ modulo 641.

Alors, $5^4 \cdot 2^{28} \equiv 1$, donc $2^{32} \equiv -1$,
de sorte que $F_5 = 2^{32} + 1$ est multiple de 641.
Puisque $F_5 \neq 641$, cela prouve que F_5 n'est pas premier.

On peut la vérifier en Lean.

La librairie Mathlib

Il faudrait expliquer sur quels principes la librairie Mathlib est construite, structurée, etc.