Rapport

Maryem Hajji, Léa Riant, Ryan Lahfa, Ivan Hasenohr

Table des matières

1 Introduction

Ce travail ainsi que les codes sources sont disponibles à l'URL suivante : https://github.com/RaitoBezarius/projet-maths-lean.

Avant d'expliquer en quoi consiste un assistant de preuve, donnons quelques éléments d'histoire autour de ces derniers.

1.1 Courte histoire des assistants de preuve et du rêve d'Hilbert

En août 1900, David Hilbert présente ses 23 problèmes, dont le second est la cohérence de l'arithmétique, fracassé par le résultat d'incomplétude de Gödel (qui ne résoud pas tout à fait la question et dont on pourra retrouver une démonstration en profondeur dans [6]) en 1931, et dont une réponse positive est obtenue par Gantzen à l'aide de la récurrence transfinie. C'est l'élan qui va lancer la théorie de la démonstration.

En 1966, de Bruijn lance le projet Automath [4] qui a pour visée de pouvoir exprimer des théories mathématiques complètes, c'est-à-dire des théories qui sont des ensembles maximaux cohérents de propositions, i.e. le théorème d'incomplétude de Gödel ne s'y applique pas notamment.

Peu après, les projets Mizar [10], HOL-Isabelle [9] et Coq [5] naissent pour devenir les assistants de preuve mathématiques que l'on connaît.

1.2 Principe d'un assistant de preuves

Ces projets mettent à disposition un ensemble d'outil afin d'aider le mathématicien à formaliser sa preuve dans une théorie mathématiques de son choix : ZFC[1], la théorie des types dépendants [3], la théorie des types homotopiques [11] par exemple.

Certains assistants de preuve ne se contentent pas de vérifier la formalisation d'une preuve mais peuvent aussi effectuer de la décision (dans l'arithmétique de Presburger par exemple).

1.3 Enjeu d'un assistant de preuves et exemples d'usages

L'enjeu des assistants de preuve et des concepts utilisés derrière dépasse le simple outil de mathématicien.

D'une part, ils permettent d'attaquer des problèmes qui ont résisté pendant longtemps, le théorème des quatre couleurs par exemple.

D'autre part, leurs usages se généralisent afin de pouvoir faire de la certification informatique, démontrer qu'un programme vérifie un certain nombre d'invariants, par exemple, dans l'aviation, des outils similaires sont employés pour certifier le comportement de certaines pièces embarquées.

1.4 Éléments de théorie des assistants de preuves

Nous ne nous attacherons pas à faire un état du fonctionnement des assistants de preuves, ceux là dépassent largement le cadre d'une licence, mais on peut donner quelques éléments d'explications.

Distinguons deux opérations, celle de la vérification de preuve et celle de la déduction automatique.

Notons que dans un premier temps, la plupart des opérations idéales d'un assistant de preuve sont indécidables, c'est-à-dire, qu'il n'existe pas d'algorithme permettant de calculer le résultat en temps fini.

Dans ce cas, afin de pouvoir vérifier une preuve, il faut l'écrire dans un langage où toutes les étapes sont des fonctions récursives primitives (ou des programmes), ce qui les rend décidables par un algorithme. L'enjeu ensuite est de le faire efficacement, bien sûr.

Ainsi, rentre en jeu les notions de mots, de langages, de confluences et de systèmes de réécritures et d'avoir des algorithmes de bonne complexité temporelle et mémoire afin de pouvoir manipuler les représentations internes d'une preuve et décider s'ils sont des preuves du résultat désiré.

[1] Théorie de Zermelo-Fraenkel avec l'axiome du choix.

Au dessus de cela, on a besoin de se donner des théories axiomatiques dans lequel on travaille, par exemple ZFC, Peano, la théorie des catégories, la théorie des types dépendants, la théorie des types homotopiques. Dans notre cas, Lean utilise la théorie des types dépendants par défaut mais propose la version homotopique aussi, qui est plus délicate à manipuler. De cela, on peut construire des notions d'ensembles, d'entiers naturels, de catégories aussi.

Ceci est pour la partie vérification et fondations théoriques du modèle.

Pour la partie automatique, selon la logique, le problème passe d'indécidable à décidable, par exemple, pour le calcul des propositions, le problème est décidable mais de classe de complexité co-NP-complete (le complémentaire de la classe NP-complete), indiquant que les algorithmes de décisions prennent un temps exponentiel certainement.

En somme, c'est un problème très difficile, mais sur lequel il a été possible d'avoir des résultats positifs, notamment un qui a résolu un problème de longue date sur lequel aucune bille n'était disponible : la conjecture de Robbins, 1933, résolue en 1996 avec un assistant de preuve à déduction automatique EQP. [12]

Dans une certaine mesure, Lean [2] est capable d'assister à trouver des morceaux de preuve par lui-même à l'aide de tactiques qui peuvent être aussi écrite par les utilisateurs afin d'améliorer l'intelligence de Lean dans certains contextes (chasse aux diagrammes en catégories par exemple).

1.5 Objectifs de ce projet

Nous allons d'abord nous familiariser au langage de Lean [2], l'assistant de preuve de Microsoft Research qui sera utilisé pour ce projet, ses concepts à travers le « Number Games » de Kevin Buzzard [7] qui consiste à redémontrer quelques théorèmes autour des entiers naturels en partant des axiomes de Peano.

Nous fournissons en 2, des solutions détaillées et expliquées des théorèmes qu'on a jugé un peu subtil tout en introduisant le système de tactique, pièce fondamentale des assistants de preuve et de l'automatisation des démonstrations.

Ensuite, nous nous dirigerons vers les espaces métriques et construirons leur formalisme dans un cadre usuel, alors que la bibliothèque mathlib [8] construit les espaces topologiques, uniformes, métriques avec des notions de suites généralisées et de filtres.

Enfin, ambitieux mais si le temps le permet, nous attaquerons une démonstration formalisée du théorème d'Ostrowski[2] en posant la théorie des valuations d'Artin [1].

2 Détail des exercices du « Number Games » de Kevin Buzzard

On se donnera pendant cette section un alphabet Σ qui pourra contenir selon le contexte, les opérateurs usuels en mathématiques $\{+, -, \times, /\}$, les chiffres, l'alphabet grec et latin.

Puis, on munit $(\Sigma^*, \cdot)$ d'une structure de monoïde usuelle où $\cdot$ est la concaténation des mots et Σ^* est la fermeture par l'étoile de Kleene de Σ. [3]

2.1 Tactiques de base en Lean

2.1.1 `intro`

Les tactiques suivantes permettent la manipulation de fonction en Lean, une fonction f : $A \to B$ pour A et B deux types étant simplement un élément de type $A \to B$, qui à une preuve de A renvoie une preuve de B.

Parfois, le but que nous cherchons à atteindre est une implication. Pour prouver que $A \to B$, on va prouver B sachant A vrai. En Lean, cela revient à inclure A dans les hypothèses et à changer le but en B. C'est ce que fait la tactique `intro`. On peut donner un nom à l'hypothèse qu'on introduit : `intro h,` ou laisser Lean choisir un nom par défaut. On peut écrire `intros h1 h2 …hn,` pour introduire plusieurs hypothèses en même temps.

2.1.2 `have`

Pour déclarer une nouvelle hypothèse, on peut utiliser la tactique *have*. `have p : P` divise le but en 2 sous-buts : montrer qu'on peut construire un élément de type P avec les hypothèses actuelles puis montrer le but initial avec l'hypothèse $p : P$ en plus. Lorsque la preuve de l'existence de l'objet qu'on crée est brève, on peut contracter sa définition : `have p := f a` avec $a : A$ et $f : A \to P$ comme hypothèses déjà présentes ajoutera directement $p : P$ dans la liste d'hypothèses.

2.1.3 `refl`

Cette tactique correspond à la réflexivité de l'égalité, d'où le nom `refl`. Elle peut s'appliquer pour prouver toute égalité de la forme $A = A$. C'est à dire, toute égalité dont les deux membres sont égaux terme à terme.
Exemple : soient x, y, z, w des entiers naturels, alors on peut prouver que $x + y \times (z + w) = x + y \times (z + w)$ en exécutant l'instruction `refl,`.

2.1.4 `rw`

Soient F, A et B des mots de Σ^*.
Le nom de cette tactique (rw) correspond au mot anglais `rewrite`. Elle s'applique dans 2 cas distincts :
Soit H une hypothèse, sous la forme $A = B$. Supposons que l'équation à démontrer est le mot F. Si F contient au moins un A, l'instruction `rw` H dérive un mot F' du mot F, en effectuant un seul changement : tous les As (présents dans F) sont réécrits en Bs. De même, si F contient au moins un B et si on utilise `rw ← H`, alors le seul changement sera : tous les Bs (présents dans F) sont réécrits en As.
Soit $T : A = B$, c'est à dire T est une preuve de $A = B$, supposé faite à un niveau qui précède le niveau traité. Dans ce cas, elle figure sur le menu des théorèmes. Alors `rw T` (respectivement `rw ← T`) dérive un mot F' du mot F, en effectuant un seul changement : la première occurrence de A (resp. B) est remplacée par un B (resp. A).

2.1.5 `simp`

C'est une tactique de haut niveau. Elle est disponible à partir du dernier niveau de *Addition World*. Son principe est le suivant : elle utilise la tactique `rw` avec les preuves des théorèmes d'associativité et de commutativité de l'addition pour prouver une certaine égalité (les preuves de l' associativité et la commutativité de la multiplication sont disponibles à partir du dernier niveau de *Multiplication World*). De plus, à l'aide du langage de métaprogrammation de Lean, on peut éventuellement apprendre à `simp` à simplifier une variété de formules plus large en utilisant d'autres preuves outre celles de l'associativité et de la commutativité.
Exemple : Soient x, y, z, w, u des entiers naturels, alors on peut démontrer que $x + y + z + w + u = y + (z + x + u) + w$ en utilisant $\{$`simp,` $\}$

[2] Dont le livre d'Artin fournit une démonstration
[3] i.e. tous les mots sur Σ

2.1.6 exact

La tactique exact permet de dire à Lean que le but recherché correspond exactement à ce que vous lui indiquez. Par exemple, si le but est P, ce qui revient à trouver $p : P$, et que vous disposez de p de type P dans les hypothèses, alors exact p, terminera la preuve. De même, si le but est Q, ce qui revient à trouver $q : Q$ et que vous disposez d'un élément p de type P et d'une fonction $f : P \to Q$, alors exact f(p), terminera la preuve.

2.1.7 apply

Cette technique vous permet de modifier le but sans rajouter de variables : de fait, elle raisonne comme ceci : vous avez pour but un élément de Q. Or vous disposez d'une fonction $f : P \to Q$. De ce fait, pour disposer d'un élément de Q, il vous suffit de disposer d'un élément de P, car $f(p)$ sera de type Q. apply f, fait exactement ça, et donc changera le but de Q en P.

2.1.8 induction

La tactique induction permet de démontrer une proposition quantifié sur un type inductif, à l'aide du principe d'induction.

Sans rentrer dans les détails de théorie des types, dans les axiomes de Peano, cela revient au théorème suivant, pour toute proposition logique P :

$$(P(0) \land \forall n, P(n) \implies P(n+1)) \implies (\forall n, P(n))$$

En Lean, cela se matérialise par la syntaxe induction <variable> with <nom de la variable inductive> <hypothèse d'induction> et transforme le but en deux buts : le cas de base et le cas inductif.

2.2 Addition World

Addition World est le premier monde de **Natural Number Game**. Dans ce monde, on dispose principalement de 3 tactiques : refl, rw (dont l'application était initiée dans *Tutorial*) et induction.
En plus, chaque théorème, une fois démontré, sera utilisé comme un résultat acquis dans les démonstrations de tous les théorèmes qui suivent. Par exemple, en commençant *Addition World*, on peut utiliser les deux théorèmes suivants : add_zero et add_succ, qui sont supposés démontrés dans la partie *Tutorial*. *Addition World* contient 6 niveaux : zero_add, add_assoc, succ_add, add_comm, succ_eq_add_one et add_right_comm. Détaillons la démonstration du théorème suivant :

2.2.1 Niveau 5

succ_eq_add_one

$$\forall n \in \mathbb{N}, succ(n) = n + 1$$

```
theorem succ_eq_add_one (n : mynat) : succ n = n + 1 :=
begin [nat_num_game]
rw one_eq_succ_zero, -- c'est plus facile de manipuler le chiffre 0 que le chiffre 1.
--On réécrit donc 1 en succ(0), puisque 1=succ(0) ( la preuve de cette égalité est
--one_eq_succ_zero). On obtient succ(n)=n+succ(0)
 rw add_succ, -- add_succ fournit l'égalité n+succ(0)=succ(n+0), on l'utilise alors pour réécrire succ(n)=n+succ(0) en succ(n)=succ(n+0). Ainsi, on pourra utiliser un des théorèmes qui manipulent le chiffre 0
rw add_zero, -- utilisation de ce théorème pour réécrire n+0 en n
refl,
end
```

2.3 Multiplication World

Dans ce monde, les théorèmes reposent principalement sur les propriétés basiques de la multiplication, tels que la commutativité, l'associativité, et la distributivité de la multiplication par rapport à l'addition dans les deux sens (à gauche et à droite). *Multiplication World* contient 9 niveaux : zero_mul, mul_one, one_mul, mul_add, mul_assoc, succ_mul, add_mul, mul_comm et mul_left_comm.
Nous explicitons la démonstration du théorème suivant :

2.3.1 Niveau 4

mul_add

La multiplication est distributive à gauche, c'est à dire $\forall a, b, t \in \mathbb{N},\ t \times (a+b) = t \times a + t \times b$

```
lemma mul_add (t a b: mynat): t*(a+b) = t*a+t*b :=
begin [nat_num_game]
induction a with d hd, -- Dans l'induction, a est renommé en d qui varie inductivement
--et hd est l'hypothèse d'induction sur d (cas de base: d=0, cas d'induction: on suppose hd,
-- on démontre h(succ(d)))
--Cas de base: montrons que t×(0+b) = t×0+t×b
rw zero_add, -- on remplace 0+b par b, on obtient t×b = t×0+t×b
rw mul_zero, -- on remplace t×0 par 0, on obtient t×b = 0+t×b
rw zero_add, -- on obtient t×b = t×b
refl,
--Cas d'induction: supposons hd : t×(d+b) = t×d + t × b
--et montrons h(succ(d)): t×(succ(d)+b) = t×succ(d)+t×b
rw succ_add, -- une solution serait de se ramener à une équation où l'un des deux membres
--est égal
--à un membre de hd.
--Pour faire cela, on utilise succ_add qui s'applique uniquement sur une quantité
--de la forme succ(d)+b (d et b étant deux entiers naturels quelconques), nous permettant ainsi
--de la remplacer par succ(d+b)
rw mul_succ, -- on utilise mul_succ (a b: mynat): a×succ(b) = a×b+a
rw hd, -- on remplace t×(d+b)+t par t×d+t×b+t en utilisant hd,
-- on obtient t×d+t*b+t = t×succ(d)+t×b
rw add_right_comm, -- on applique la commutativité de l'addition pour remplacer t×b+t par t+t×b
rw → mul_succ, --on utilise rw ← pour remplacer t×d +t (qui est le membre droit
-- de l'égalité qui correspond au théorème mul_succ) par t×succ(d),
-- on obtient t×succ(d)+t×b = t×succ(d)+t×b
refl
end
```

2.4 Power World

Ce monde contient 8 niveaux : `zero_pow_zero`, `zero_pow_succ`, `pow_one`, `one_pow`, `pow_add`, `mul_pow`, `pow_pow` et `add_squared`. Nous explicitons la démonstration du théorème suivant :

2.4.1 Niveau 7

`add_squared` (Cas particulier de la formule du binôme de Newton : $(a+b)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} a^n b^{n-k}$, pour $n = 2$)

$$\forall a, b \in \mathbb{N},\ (a+b)^2 = a^2 + b^2 + 2 * a * b$$

```
lemma pow_pow (a m n: mynat): (a^m)^n = a^(m*n) :=
begin [nat_num_game]
  -- on simplifie les puissances, en réécrivant les puissances 2 en fonction de 0
  rw two_eq_succ_one, -- on utilise la preuve de succ(1)=2 pour réécrire le chiffre 2 en succ(1)
  rw one_eq_succ_zero, -- on réécrit 1 en succ(0), on obtient donc
  --(a+b)^succ(succ(0)) = a^succ(succ(0)) + b^succ(succ(0)) + succ(succ(0)) × a × b
  repeat {rw pow_succ}, -- on obtient (a+b)^0 × (a+b) × (a+b) = a^0 × a × a + b^0 × b × b + succ(succ(0)) × a × b
  repeat {rw pow_zero}, -- on obtient 1 × (a+b) × (a+b) = 1 × a × a + 1 × b × b + succ(succ(0)) × a × b
  simp, -- on obtient (a+b) × (a+b) = a × a + (b × b + a × (b × succ(succ(0))))
 --donc simp, dans ce cas, applique le théorème one_mul (m: mynat): m × 1 = m
  repeat {rw mul_succ}, -- on obtient (a+b)*(a+b) = a*a+(b*b+a*(b*0+b+b))
  simp, -- on obtient (a+b) × (a+b) = a × a + (b × b + a × (b+b))
  -- donc simp, dans ce cas, applique les théorèmes mul_zero (a: mynat): a × 0 = 0
  -- et zero_add (n: mynat): 0 + n = n
  -- on développe (a+b) × (a+b) :
  rw mul_add,
  -- on développe (a+b) × a
  rw mul_comm,
  rw mul_add,
  -- on développe (a+b) × b
```

```
rw mul_comm (a + b) b,
rw mul_add,
simp, -- on met les termes du membre de gauche dans le bon ordre
rw ← add_assoc (a * b) (a * b) (b * b), -- on obtient a × a + (a × b + a × b + b × b) = a × a + (b × b + a × (b + b))
rw add_right_comm,
rw add_comm (a * b) (b * b),
rw add_assoc (b * b) (a * b) (a * b), -- on obtient
-- a × a + (b × b + (a × b + a × b)) = a × a + (b × b + a × (b + b))
-- on factorise par a :
rw ← mul_add a b b, -- on obtient a × a + (b × b + a × (b + b)) = a × a + (b × b + a × (b + b))
refl
end
```

2.5 Function World

Ce monde nous introduit un outil fondamental de Lean : les fonctions. Un élément important à remarquer est qu'en Lean, toutes les fonctions sont curryfiées.

2.5.1 Niveau 6

Voici un exemple de niveau de ce monde, le niveau 6, qui demande de créer une fonction de fonctions assez fastidieuse, et qui utilise le fait que ces fonctions sont curryfiées. L'énoncé se formule comme ceci :

$$
\begin{aligned}
&(P, Q, R : \text{Type}) : \\
&(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))
\end{aligned}
$$

La preuve est de fait succinte :

```
intros f g p, -- On introduit les différents éléments/fonctions pour créer la fonction demandée
apply f p, -- On modifie le but à l'aide de la fonction curryfiée
exact g p, -- On trouve le résultat demandé
```

Ce qui conclut la preuve.

2.6 Proposition World

Dans ce monde on aborde un aspect fondamental de l'assistant de preuves Lean : une preuve est composée d'implications, et c'est ici que les fonctions prennent toute leur importance : pour montrer que A implique B, il suffit de créer une fonction de A vers B, soit un élément de type $A \to B$.

2.6.1 Niveau 1

Pour illustrer ce point, voici un exemple simple, le tout premier niveau de Proposition World.

```
lemma : (P,Q : Prop) (p : P) (h : P → Q) : Q
```

Donc, en français, on dispose d'une preuve de P, et d'une fonction de P dans Q (i-e d'un élément de type $P \to Q$), trouvons un élément de type Q (montrons que Q est vrai). Ce qui se résout tout aussi succintement :

```
exact h(p),
```

2.6.2 Niveau 8

Un autre niveau intéressant est le niveau 8, qui propose une preuve du lemme suivant, une implication de l'équivalence entre une proposition et sa contraposée (si cela a du sens) :

```
lemma : (P → Q) → (¬Q → ¬P).

-- En Lean, on demande donc de créer une fonction qui prend une preuve que P → Q et renvoie une preuve de ¬Q → ¬P.
intro f, -- On dispose d'une preuve de P → Q
-- Lean nous demande alors de créer un élément de type ¬Q → ¬P, qui serait l'image de la fonction qu'il nous est demandé de créer.
repeat{rw not_iff_imp_false}, -- On retranscrit la définition de ¬P : ¬P ≡ P → false.
-- Le but est alors réécrit en (Q → false) → P → false, ce qui revient à créer une fonction curryfiée des éléments de type (Q → false) × P vers les preuves de false. On réapplique la même technique d'introduire un élément de chacun des ensembles de départ :
intros h p,
```

```
-- On dispose alors d'un élément p de P, d'une fonction f de P dans Q et d'une fonction h de Q dans false, et il nous faut créer une preuve de false, qui est facilement trouvable avec :
exact h(f(p)), -- Ce qui conclut la preuve.
```

2.7 Advanced Proposition World

2.7.1 Niveau 8

Dans ce monde on démontre à l'aide de fonctions et de nouvelles méthodes les règles de base de la manipulation de consjonctions et disjonctions logiques. Un exemple combinant la plupart des nouvelles méthodes est le Lemme suivant :

```
lemma : (P,Q,R : Prop) : P ∧ (Q ∨ R) ↔ (P ∧ Q) ∨ (P ∧ R)
```

Ici on ne démontrera que l'implication directe, l'implication réciproque se faisant de façon similaire. Pour séparer les implications, une technique existe : `split`, qui permet de montrer d'abord l'implication directe puis l'implication réciproque. A noter que cette technique permet aussi de séparer le but en plusieurs buts lorsqu'on a à montrer une conjonction de propositions. Pour gérer les disjonctions de propositions, la technique `cases` existe et permet, par exemple, quand on sait que $P \vee Q$, dans un premier temps supposer P puis supposer Q. Cette technique permet aussi de séparer les conjonctions connues en plusieurs nouvelles données : si l'on a un élément pq de $P \wedge Q$, `cases pq with p q` nous renvoie deux éléments p et q de P et Q respectivement. Finalement, lorsqu'on doit montrer une disjonction de propositions, il suffit d'en montrer une, et les techniques `left` et `right` nous permettent de choisir la proposition à démontrer. La preuve est donc la suivante :

```
intro h,       -- h de type P ∧ (Q ∨ R)
cases h with p qor,       -- p de type P , qor de type Q ∨ R
cases qor with q r,       -- On sépare en deux cas en fonction de la disjonction :

-- Premier cas : q de type Q
left,     -- On choisit de montrer P ∧ Q
split,      -- On sépare en deux buts
exact p,
exact q,

-- Deuxième cas : r de type R
right,        -- On choisit de montrer P ∧ R
split,      -- On sépare en deux buts
exact p,
exact r, -- Ce qui conclut la preuve de l'implication directe.
```

2.8 Advanced Addition World

À ce niveau, nous avons déjà montré que $(\mathbb{N}_{\text{mynat}}, +)$ est un monoïde commutatif. Dans ce monde, nous allons prouvé des propriétés un peu plus complexes, comme l'injectivité de la fonction successeur ou la régularité du monoïde.

2.8.1 Niveau 10

Ce lemme ressemble à la régularité à gauche du monoïde $(\mathbb{N}_{\text{mynat}}, +)$ qu'on a prouvé au niveau 6 de ce monde. On a démontré que :

```
(a b c : mynat) : a + b = a + c → b = c
```

Donc pour $a = 0$ et $c = 0$ et $a + b = a + c$ impliquent $b = 0$. Dans ce lemme, nous allons montrer qu'il suffit d'avoir les hypothèses $a + c = 0$ et $c = 0$ et $a + b = a + c$ pour prouver $b = 0$.

```
lemma add_left_eq_zero {{a b : mynat}} : a + b = 0 → b = 0 :=
begin [nat_num_game]
  intro H,
  --On fait une distinction de cas
  --Soit b = 0 soit il existe d : mynat tel que b = succ(d) :
  cases b with d,

  --Cas b = 0, le but devient 0 = 0, la résolution est triviale :
  refl,

  --Cas b = succ(d), le but devient succ d = 0
  --Ce qui est impossible, cela contredit l'axiome de Peano zero_ne_succ
  --On va donc faire une preuve par l'absurde :
  rw add_succ at H, --on fait rentrer a dans le succ donc H : succ(a + d) = 0
```

```
  exfalso, -- le but est impossible à prouver donc on le change en faux
  --Et on a le théorème succ_ne_zero (n : mynat) : succ n = 0 → faux, donc on sait que l'hypothèse H implique faux
  exact succ_ne_zero H,
  --On a donc prouver que l'hypothèse ∃d ∈ ℕ_mynat, b = succ(d) est est contradictoire avec nos axiomes
end
```

2.9 Advanced Multiplication World

Nous avons prouvé dans les mondes précédents que $(\mathbb{N}_{\text{mynat}}, \times)$ est un monoïde commutatif et que la multiplication est distributive par rapport à l'addition.

2.9.1 Niveau 4

Ce théorème consiste à prouver la régularité à gauche du monoïde $(\mathbb{N}_{\text{mynat}}, \times)$. L'idée est instinctive mais la preuve nécessite en réalité beaucoup de distinctions de cas et l'utilisation d'une nouvelle tactique, `revert`.

```
theorem mul_left_cancel (a b c : mynat) (ha : a  0) : a * b = a * c → b = c :=
begin
  revert b,
  --On ne considère plus b comme une hypothèse,
  --à la place, on rajoute un ∀(b : mynat) au but
  --Ce sera utile plus tard, dans l'hypothèse d'induction

  -- On fait une induction sur c :
  induction c with n hn,

  --Le cas de base ∀(b : mynat), a ∗ b = a ∗ 0 → b = 0 :
  rw mul_zero, --On simplifie
  intros b h, --On introduit un b et l'hypothèse h : a ∗ b = 0
  rw mul_eq_zero_iff a b at h, --h est équivalent à a = 0 ∨ b = 0 donc on la réécrit

  --On casse le a = 0 ∨ b = 0 en deux cas:
  cases h with hha hhb,

  --Si a = 0 (but : b = 0) :
  --On a a ≠ 0 en hypothèse donc on sait que ce cas est impossible
  --On va donc faire une preuve par l'absurde :
  exfalso, --but = false
  apply ha, --but = a = 0
  exact hha, --Il n'y a plus qu'à appliquer l'hypothèse de disjonction de cas

  --Si b = 0 (but : b = 0), c'est trivial :
  exact hhb,

  --Le cas d'induction (but : ∀(b : mynat), a ∗ b = a ∗ succ n → b = succ n) :
  intros b h, --On introduit un b et l'hypothèse h : a ∗ b = a ∗ succ n
  --Le but est juste b = succ n maintenant
  --On fait une distinction de cas sur b :
  cases b with c,

  --Cas b = 0 (but : 0 = succ n) :
  --Cela contredit l'axiome de Peano zero_ne_succ, on va donc passer par l'absurde :
  rw mul_zero at h, --On simplifie h pour obtenir h : 0 = a ∗ succ n
  exfalso,
  apply mul_pos a (succ n), --On a besoin de démontrer les hypothèses de mul_pos :
  --Hypothèse a ≠ 0 :
  exact ha,
  --Hypothèse succ n ≠ 0 :
  exact succ_ne_zero n,
  --Retour à la preuve par l'absurde :
  symmetry,
  exact h,

  --Cas b = succ c (but : succ c = succ n) :
  repeat {rw succ_eq_add_one},
  rw add_right_cancel_iff, --On simplifie le but pour lui appliquer l'hypothèse d'induction
  --C'est là que le revert b prend tout son importance car le but est c = n
  --On n'aurait pas pu appliquer l'hypothèse d'induction si elle prenait un b particulier
  apply hn,
```

```
  --Il ne reste plus qu'a simplifier l'hypothèse h :
  repeat {rw mul_succ at h},
  rw add_right_cancel_iff at h,
  exact h,
end
```

2.10 Inequality World

Dans ce monde, nous allons définir et prouver des propriété sur la relation d'ordre $\leq$ et son ordre strict $<$.

2.10.1 Niveau 15

Dans la suite, nous allons définir $>$ tel que :

```
def a < b := a ≤ b ∧¬ (b ≤ a)
```

Mais la définition :

```
def a < b := succ a ≤ b
```

est plus pratique à utiliser et mathématiquement équivalente dans les entiers naturels. Nous allons donc prouver que

```
(a b : mynat) : a ≤ b ∧¬ (b ≤ a) → succ a ≤ b
```

dans ce lemme (l'autre partie de l'équivalence est le niveau 16).

```
lemma lt_aux_one (a b : mynat) : a ≤ b ∧¬ (b ≤ a) → succ a ≤ b :=
begin
  --On commence par transformer a ≤ b ∧ ¬(b ≤ a) en 2 hypothèses :
  intro h,
  cases h with hab htba,
  --On introduit c tel que b = a + c
  cases hab with c hc,

  --On ne peut rien faire à ce niveau car le cas b = 0 pose problème
  --On fait donc une distinction de cas sur b :
  cases b,

  --Cas b = 0 (impossible donc par l'absurde) :
  exfalso,
  apply htba,
  exact zero_le a,

  --Cas b = succ b (but : succ a ≤ succ b) :
  --Là, c'est le cas a = 0 qui nous pose problème
  cases a,

  --Cas a = 0 (trivial vu que b ≠ 0) :
  apply succ_le_succ,
  exact zero_le b,

  --Cas a = succ a (but : succ(succ a) ≤ succ b) :
  rw hc,
  rw succ_add,
  apply succ_le_succ, --but : succ a ≤ a + c

  --Cette inégalité est impossible si c = 0
  cases c,

  --Cas c = 0
  rw add_zero at hc,
  exfalso,
  apply htba,
  use 0,
  rw add_zero,
  symmetry,
  exact hc,

  --Cas c = succ c (but : succ a ≤ a + succ c) :
  --Les +1 se simplifient
  rw add_succ,
  apply succ_le_succ,
  use c,
  refl,
end
```

3 Excursion dans le formalisme des espaces métriques

3.1 `set`:

`set` $a := t$ `with` h est équivalent à `soit` $a := t$. Cette tactique ajoute l'hypothèse $h : a = t$ au contexte local et remplace toutes les occurrences de t avec a.

3.2 `use:`

`use` x instancie le premier terme d'une existence avec x. On l'utilise quand le goal commence avec un $\exists$.

3.3 `obtain:`

Cette tactique est une combinaison de deux tactiques : `have` et `rcases`. obtain <patt>: type est équivalent à have h: type, rcases h with <patt>. Si `type` n'est pas prouvé, la syntaxe à utiliser sera obtain <patt>: type := proof

3.4 `cases:`

Soit x une variable dans le contexte local de type inductive, alors les hypothèses qui contiennent x et le goal (si il contient x) se divisent selon le nombre de constructeurs inductives de x. Par exemple, si x est un entier naturel, alors d'après l'axiome de Peano, l'induction qui permet de construire x est la suivante : $d = 0$ et $succ(d) = d + 1$.
Dans ce cas, si on prend une hypothèse $h : A\ x$ et un goal $B\ x$ alors `cases` x `with` d produit un goal $B\ 0$ avec l'hypothèse $h : A\ 0$, et un goal $B\ succ(d)$ avec l'hypothèse $h : A\ succ(d)$. un autre exemple de variable de type inductive est donné par une hypothèse sous la forme $P \wedge Q$ ou bien $P \leftrightarrow Q$. Si on prend $h1 : P \wedge Q$, `cases` $h1$ `with` $p\ q$ va remplacer l'hypothèse $h1$ par les 2 hypothèses $p : P$, $q : Q$.

3.5 `rcases:`

Cette tactique a le même principe que la tactique `cases`, la seule différence c'est qu'elle donne plus de flexibilité au niveau des constructeurs de l'induction correspondant à la variable à laquelle on applique `cases`.
Par exemple, on pose $h : (a \to b) \wedge (c \to d)$, alors `rcases h with`($<A, B>|<C>$) remplace h par $A : a$, $B : b$, $C : c \to d$. Donc $<A, B>|<C>$divise l'hypothèse en 2 constructeurs inductives, récupère les deux premiers paramètres du premier constructeur en A et B et récupère le deuxième constructeur en C.

4 Lemma Cauchy_est_bornee

4.1 Définition

Soit (X, d) un espace métrique. On dit qu'une suite des éléments de X, $(u_n)_{n\in\mathbb{N}}$ est de Cauchy si est seulement si :

$$\forall\varepsilon \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}, \forall n, m \in \mathbb{N}, n, m \geq n_0 \Rightarrow d(U_n, U_m) \leq \varepsilon$$

4.2 Énoncé du lemme

Dans un espace métrique, toute suite de Cauchy est bornée.

4.3 L'idée principale de la preuve

On veut montrer que toute suite de Cauchy est bornée. Une suite bornée dans un espace métrique $(W_n)_{n\in\mathbb{N}}$ est une suite qui vérifie la condition suivante :

$$\forall m \in \mathbb{N}, \exists M, \forall n \in \mathbb{N}, d(W_n, W_m) \leq M$$

Soit $(U_n)_{n\in\mathbb{N}}$ une suite de Cauchy. D'après la définition d'une suite de Cauchy, en prenant 1 comme valeur de ε, il existe un rang $n_0 \in \mathbb{N}$ tel que pour tout $n, m \geq n_0$, $d(U_n, U_m) \leq 1$. Donc on peut diviser les termes de la suite en 2 ensembles : $E_1 = \{U_n | n \leq n_0\}$ et $E_2 = \{U_n | n > n_0\}$. Soit Y un élément quelconque de la suite (U_n).

- $\{U_n | n \leq n_0\}$ est un ensemble fini dénombrable, donc pareil pour l'ensemble $D_1 = \{d(U_n, Y), \forall U_n \in E_1\}$ qui est également dénombrable et fini. Alors l'ensemble D_1 admet un maximum M_1.
- Soit $D_2 = \{d(U_n, Y) | n > n_0\}$. D'après l'inégalité triangulaire, $d(U_n, Y) \leq d(U_n, U_{n_0}) + d(U_{n_0}, Y)$. Or d'après le caractère Cauchy de la suite, $d(U_n, U_{n_0}) \leq 1$ pour tout $n \geq n_0$, on en déduit que D_2 est majoré par $d(U_{n_0}, Y) + 1$.

On en déduit que la suite (U_n) est majorée par $max(M_1, d(U_{n_0}, Y) + 1)$. Elle est donc bornée.

Dans la partie ci-dessous, nous allons expliquer l'utilisation de certaines tactiques dans la démonstration de ce théorème sur Lean.

- obtain $<N, H> : \exists N, \forall p \geq N, \forall q \geq N,$ ((d (x p) (x q)) < 1)

  on utilise cette tactique pour stocker dans H une formule mathématique (hypothèse) qui dépend d'un entier N, et dans N l'entier en question. Suite à l'utilisation de `obtain`, on doit démontrer l'existence d'un entier N qui vérifie H. Pour ce faire, on utilise l'hypothèse `cauch: cauchy x` et le fait que 1 soit un entier strictement positive.
  Á ce niveau, le **goal** est : $\exists (M : \mathbb{R}), M > 0 \wedge \forall (n : \mathbb{N})\ d\ (xn)\ y \leq M$

- set Limage := $\{M : \mathbb{R}, \exists n \leq N,$ M = d (x n) y $\}$

  Cet ensemble contient toutes les valeurs possibles de la distance entre un terme de la suite d'indice inférieur ou égal à N et y (qui est un terme fixe quelconque de la suite). On cherche à majorer cet ensemble. Une méthode pour le faire sera de démontrer qu'il est non vide et fini.

- have limage_finiteness : Limage.finite

  Cette tactique permet d'ajouter une hypothèse nommée `limage_finiteness` qui dit que l'ensemble `Limage` est fini. Elle est suivie par une démonstration dont la démarche est la suivante :

  - On a défini la fonction `fonction_distance` qui prend comme paramètres une suite X, un indice n et un terme Y de la suite et retourne la distance entre Y et X_n. On montre que l'ensemble `Limage` est l'image de l'ensemble $\{i : \mathbb{N}, i \leq \mathbb{N}\}$ par `fonction_distance`.
  - On utilise `apply set.finite_image`, pour appliquer le résultat suivant : l'image d'un ensemble fini est finie. Le `goal` devient de démontrer que l'ensemble $\{i : \mathbb{N}, i \leq \mathbb{N}\}$ est fini.
  - On utilise `exact set.finite_le_nat` N pour ce faire.

- have limage_nonempty : Limage.nonempty

  Cette tactique permet d'ajouter une hypothèse nommée `limage_nonempty` qui dit que l'ensemble Limage est non vide. Afin de démontrer ce résultat, on utilise le fait que $d\ (x0)\ y$ appartient à l'ensemble, puisque $0 \leq N$.

- have sup_est_atteint : Sup Limage   Limage

  Pour démontrer `sup_est_atteint` (c'est à dire que `Limage` admet un maximum), on utilise `set.finite.has_a_reached_sup` avec les hypothèses `limage_finiteness` et `limage_nonempty`. On note ce maximum $d(xn)y$.

- use $(max(d(xn)y)(1 + d(xN)y))$

  De cette façon le `goal` devient de démontrer que $max(d(xn)y)(1 + d(xN)y)$ est un majorant de la suite de Cauchy X en question. C'est à dire il faut montrer que $max(d(xn)y)(1 + d(xN)y) > 0$ (ce qui est trivial) et que $\forall n_1 : \mathbb{N}, d(xn_1)y \leq max(d(xn)y)(1 + d(xN)y)$

- intro p, by_cases (p   N)

  C'est à dire qu'on prend un entier p et on montre la propriété dans 2 cas : pour $p \geq N$ et pour $p < N$.
  Dans le cas où $p \geq N$, d'après l'hypothèse H, puisque $p \geq N$ et $N \geq N$, $d(xp)(xN) < 1$.
  Alors $d(xp)(xN) + d(xN)y \leq 1 + d(xN)y$
  Donc à fortiori $d(xp)(xN) + d(xN)y \leq max(d(xn)y)(1 + d(xN)y)$. Or d'après l'inégalité triangulaire $d(xp)y \leq d(xp)(xN) + d(xN)y$, donc $d(xp)y \leq max(d(xn)y)(1 + d(xN)y)$
  Dans le cas où $p < N$, on applique l'hypothèse `sup_atteint` pour majorer $d(xp)y$ par $d(xn)y$ puisque $d(xn)y$ est la borne supérieure de l'ensemble `Limage` et $d(xp)y$ appartient à `Limage`.
  Donc $d(xp)y \leq max(d(xn)y)(1 + d(xN)y)$.

Références

[1] Emil Artin. *Algebraic numbers and algebraic functions*, volume 358. American Mathematical Soc., 2005.

[2] Jeremy Avigad. The lean theorem prover. *Microsoft Research, Carnegie Mellon University*, 2014.

[3] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions.* Springer Science & Business Media, 2013.

[4] Nicolaas Govert De Bruijn. A survey of the project automath. In *Studies in Logic and the Foundations of Mathematics*, volume 133, pages 141–161. Elsevier, 1994.

[5] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[6] Jean-Yves Girard. *Le point aveugle : cours de logique.* Hermann, Paris, 2006.

[7] Buzzard Kevin. Natural number games. https://github.com/ImperialCollegeLondon/natural_number_game, 2019.

[8] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020,

page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[10] Andrzej Trybulec and Howard A Blair. Computer assisted reasoning with mizar. In *IJCAI*, volume 85, pages 26–28, 1985.

[11] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[12] Matthew Wampler-Doty. A complete proof of the robbins conjecture. *The Archive of Formal Proofs. http ://afp. sf. net/entries/Robbins-Conjecture. shtml*, 2010.