

Rapport

MARYEM HAJJI, LÉA Riant, RYAN LAHFA, IVAN HASENOHR

Table des matières

1	Introduction	1
1.1	Courte histoire des assistants de preuve et du rêve d'Hilbert	1
1.2	Principe d'un assistant de preuves	2
1.3	Enjeu d'un assistant de preuves et exemples d'usages	2
1.4	Éléments de théorie des assistants de preuves	2
1.5	Objectifs de ce projet	3
2	Détail des exercices du « Number Games » de Kevin Buzzard	3
2.1	Tactiques de base en Lean	3
2.1.1	intro	3
2.1.2	have	3
2.1.3	refl	3
2.1.4	rw	3
2.1.5	simp	4
2.1.6	exact	4
2.1.7	apply	4
2.1.8	induction	4
2.2	Addition World	4
2.2.1	Niveau 5	4
2.3	Multiplication World	5
2.3.1	Niveau 4	5
2.4	Power World	5
2.4.1	Niveau 7	5
2.5	Function World	6
2.5.1	Niveau 6	6
2.6	Proposition World	6
2.6.1	Niveau 1	6
2.6.2	Niveau 8	7
2.7	Advanced Proposition World	7
2.7.1	Niveau 8	7
2.8	Advanced Addition World	7
2.8.1	Niveau 10	7
2.9	Advanced Multiplication World . . .	8
2.9.1	Niveau 4	8
2.10	Inequality World	9
2.10.1	Niveau 15	9
3	Excursion dans le formalisme des espaces métriques	10
3.1	Avant-propos	10
3.2	Tactiques	10
3.2.1	set	10

3.2.2	use	10
3.2.3	obtain	10
3.2.4	cases	10
3.2.5	rcases	10
3.3	cauchy_est_bornee	10
3.3.1	Définition	10
3.3.2	Énoncé du lemme	10
3.3.3	L'idée principale de la preuve	11
3.4	cauchy_admet_une_va	12
3.5	\mathbb{R} est un espace complet	13
3.5.1	Structure du code	13
3.5.2	Bolzano-Weierstrass	14
3.6	Interlude : Continuité séquentielle .	14
3.7	Continuité Séquentielle	14
3.8	Complété d'un espace métrique . .	15
3.8.1	Détails du fichier custom/cauchy.lean	15
3.8.2	Détails du fichier metric.lean	16

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 [9]) 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 [7] 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 [14], HOL-Isabelle [12] et Coq [8] 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 : ZFC¹, la théorie des types dépendants [6], la théorie des types homotopiques [15] 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.²) [1]

À noter, que Lean implémente **partiellement** la procédure de décision **omega** tiré de l'arithmétique de Presburger. [4], [13]

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.

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

²L'arithmétique de Presburger est connu pour être intégralement décidable.

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.

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.

On mentionne aussi un intérêt pour la version homotopique aussi, qui est plus délicate à manipuler, mais un groupe de travail interne à Lean est intéressé en raison de la puissance du principe d'univalence qu'on peut vulgariser en $(x \equiv y) \equiv (x = y)$, autrement dit, « l'équivalence est l'égalité », cela permet de simplifier énormément beaucoup de preuves qui travaillent « à isomorphisme (unique) près » où il convient de manipuler certains objets formellement et de les remplacer par un de leurs isomorphismes, citons par exemple : $R[1/f]g \equiv R[1/f][1/g]$, en théorie des anneaux. Lean avait autrefois dans sa version 2, le support pour la théorie des types homotopiques, mais depuis Lean 3, celui-ci a été retiré. Ajoutons que Coq possède aussi une librairie de théorie des types homotopiques [5].

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. [16]

Dans une certaine mesure, Lean [3] est capable d'assister à trouver des morceaux de preuve par lui-même à l'aide de tactiques qui peuvent être aussi écrites 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 [3], l'assistant de preuve de Microsoft Research qui sera utilisé pour ce projet, ses concepts à travers le « Number Games » de Kevin Buzzard [10] 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 [11] construit les espaces topologiques, uniformes, métriques avec des notions de suites généralisées et de filtres.

Enfin, un projet plus ambitieux mais si le temps le permet est de donner une démonstration formalisée du théorème d'Ostrowski³ en posant la théorie des valuations d'Artin [2].

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 (Σ^*, \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 .⁴

2.1 Tactiques de base en Lean

2.1.1 intro

Les tactiques suivantes permettent la manipulation de fonction en Lean, une fonction $f : A \rightarrow B$ pour A et B deux types étant simplement un élément de type $A \rightarrow B$, 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 $A \rightarrow B$, 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 $f : A \rightarrow P$ 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**

³Dont le livre d'Artin fournit une démonstration

⁴i.e. tous les mots sur Σ

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 $\text{rw} \leftarrow H$, 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 $\text{rw } T$ (respectivement $\text{rw} \leftarrow T$) 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 $\{\text{simp}, \}$

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 $f : P \rightarrow Q$, 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 $f : P \rightarrow Q$. 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 quantifiée 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 :

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

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

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

```

1 theorem succ_eq_add_one (n : mynat) :
  ↪ succ n = n + 1 :=
2 begin [nat_num_game]
3 rw one_eq_succ_zero, -- c'est plus
  ↪ facile de manipuler le chiffre 0 que
  ↪ le chiffre 1.
4 --On réécrit donc 1 en succ(0), puisque
  ↪ 1=succ(0) ( la preuve de cette
  ↪ égalité est

```

```

5 --one_eq_succ_zero). On obtient
  ↪ succ(n)=n+succ(0)
6 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
7 rw add_zero, -- utilisation de ce
  ↪ théorème pour réécrire n+0 en n
8 refl,
9 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 $\forall a, b, t \in \mathbb{N}, t \times (a + b) = t \times a + t \times b$

```

1 lemma mul_add (t a b: mynat): t*(a+b) =
  ↪ t*a+t*b :=
2 begin [nat_num_game]
3 induction a with d hd, -- Dans
  ↪ l'induction, a est renommé en d qui
  ↪ varie inductivement
4 --et hd est l'hypothèse d'induction sur
  ↪ d (cas de base: d=0, cas
  ↪ d'induction: on suppose hd,
5 -- on démontre h(succ(d)))
6 --Cas de base: montrons que t*(0+b) =
  ↪ t*0+t*b
7 rw zero_add, -- on remplace 0+b par b,
  ↪ on obtient t*b = t*0+t*b
8 rw mul_zero, -- on remplace t*0 par 0,
  ↪ on obtient t*b = 0+t*b
9 rw zero_add, -- on obtient t*b = t*b
10 refl,

```

```

11 --Cas d'induction: supposons hd :
  ↪ t*(d+b) = t*d + t * b
12 --et montrons h(succ(d)): t*(succ(d)+b)
  ↪ = t*succ(d)+t*b
13 rw succ_add, -- une solution serait de
  ↪ se ramener à une équation où l'un
  ↪ des deux membres
14 --est égal
15 --à un membre de hd.
16 --Pour faire cela, on utilise succ_add
  ↪ qui s'applique uniquement sur une
  ↪ quantité
17 --de la forme succ(d)+b (d et b étant
  ↪ deux entiers naturels quelconques),
  ↪ nous permettant ainsi
18 --de la remplacer par succ(d+b)
19 rw mul_succ, -- on utilise mul_succ (a
  ↪ b: mynat): a*succ(b) = a*b+a
20 rw hd, -- on remplace t*(d+b)+t par
  ↪ t*d+t*b+t en utilisant hd,
21 -- on obtient t*d+t*b+t =
  ↪ t*succ(d)+t*b
22 rw add_right_comm, -- on applique la
  ↪ commutativité de l'addition pour
  ↪ remplacer t*b+t par t+t*b
23 rw → mul_succ, --on utilise rw ← pour
  ↪ remplacer t*d +t (qui est le membre
  ↪ droit
24 -- de l'égalité qui correspond au
  ↪ théorème mul_succ) par t*succ(d),
25 -- on obtient t*succ(d)+t*b =
  ↪ t*succ(d)+t*b
26 refl
27 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 : $(a+b)^n = \sum_{k=0}^n \frac{n!}{k!(n-k)!} a^k b^{n-k}$, pour $n = 2$)

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

```

1 lemma pow_pow (a m n: mynat): (a^m)^n =
  ↪ a^(m*n) :=
2 begin [nat_num_game]

```

```

3  -- on simplifie les puissances, en
   ↪ réécrivant les puissances 2 en
   ↪ fonction de 0
4  rw two_eq_succ_one, -- on utilise la
   ↪ preuve de succ(1)=2 pour réécrire
   ↪ le chiffre 2 en succ(1)
5  rw one_eq_succ_zero, -- on réécrit 1 en
   ↪ succ(0), on obtient donc
6  --(a + b)succ(succ(0)) = asucc(succ(0)) +
   ↪ bsucc(succ(0)) + succ(succ(0)) × a × b
7  repeat {rw pow_succ}, -- on obtient
   ↪ (a + b)0 × (a + b) × (a + b) =
   ↪ a0 × a × a + b0 × b × b + succ(succ(0)) × a × b
8  repeat {rw pow_zero}, -- on obtient
   ↪ 1 × (a + b) × (a + b) =
   ↪ 1 × a × a + 1 × b × b + succ(succ(0)) × a × b
9  simp, -- on obtient (a + b) × (a + b) =
   ↪ a × a + (b × b + a × (b × succ(succ(0))))
10 --donc simp, dans ce cas, applique le
   ↪ théorème one_mul (m: mynat):
   ↪ m × 1 = m
11 repeat {rw mul_succ}, -- on obtient
   ↪ (a+b) × (a+b) = a × a + (b × b + a × (b × 0 + b + b))
12 simp, -- on obtient
   ↪ (a+b) × (a+b) = a × a + (b × b + a × (b+b))
13 -- donc simp, dans ce cas, applique les
   ↪ théorèmes mul_zero (a: mynat):
   ↪ a × 0 = 0
14 -- et zero_add (n: mynat): 0 + n = n
15 -- on développe (a + b) × (a + b) :
16 rw mul_add,
17 -- on développe (a + b) × a
18 rw mul_comm,
19 rw mul_add,
20 -- on développe (a + b) × b
21 rw mul_comm (a + b) b,
22 rw mul_add,
23 simp, -- on met les termes du membre de
   ↪ gauche dans le bon ordre
24 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))
25 rw add_right_comm,
26 rw add_comm (a * b) (b * b),
27 rw add_assoc (b * b) (a * b) (a * b),
   ↪ -- on obtient
28 -- a × a + (b × b + (a × b + a × b)) =
   ↪ a × a + (b × b + a × (b + b))
29 -- on factorise par a :
30 rw ← mul_add a b b, -- on obtient
   ↪ a × a + (b × b + a × (b + b)) =
   ↪ a × a + (b × b + a × (b + b))
31 refl
32 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 :

$$(P, Q, R : \text{Type}) : \\ (P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$$

La preuve est de fait succincte :

```

1  intros f g p, -- On introduit les
   ↪ différents éléments/fonctions pour
   ↪ créer la fonction demandée
2  apply f p, -- On modifie le but à l'aide
   ↪ de la fonction curryfiée
3  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 $A \rightarrow B$.

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 $P \rightarrow Q$), trouvons un élément de type Q (montrons que Q est vrai). Ce qui se résout tout aussi succinctement :

```

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) :

```

1 lemma : (P → Q) → (¬Q → ¬P).
2
3 -- En Lean, on demande donc de créer une
4   ↪ fonction qui prend une preuve que
5   ↪ P → Q et renvoie une preuve de
6   ↪ ¬Q → ¬P.
7 intro f, -- On dispose d'une preuve de
8   ↪ P → Q
9 -- Lean nous demande alors de créer un
10  ↪ élément de type ¬Q → ¬P, qui
11  ↪ serait l'image de la fonction qu'il
12  ↪ nous est demandé de créer.
13 repeat{rw not_iff_imp_false}, -- On
14   ↪ retranscrit la définition de ¬P :
15   ↪ ¬P ≡ P → false.
16 -- Le but est alors réécrit en
17   ↪ (Q → false) → P → false, ce qui
18   ↪ revient à créer une fonction
19   ↪ curryfiée des éléments de type
20   ↪ (Q → false) × P vers les preuves de
21   ↪ false. On réapplique la même
22   ↪ technique d'introduire un élément de
23   ↪ chacun des ensembles de départ :
24 intros h p,
25 -- On dispose alors d'un élément p de P,
26   ↪ d'une fonction f de P dans Q et
27   ↪ d'une fonction h de Q dans false, et
28   ↪ il nous faut créer une preuve de
29   ↪ false, qui est facilement trouvable
30   ↪ avec :
31 exact h(f(p)), -- Ce qui conclut la
32   ↪ 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 conjonctions 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 simi-

laire. 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 :

```

1 intro h, -- h de type P ∧ (Q ∨ R)
2 cases h with p qor, -- p de type P ,
3   ↪ qor de type Q ∨ R
4 cases qor with q r, -- On sépare en
5   ↪ deux cas en fonction de la
6   ↪ disjonction :
7
8 -- Premier cas : q de type Q
9 left, -- On choisit de montrer P ∧ Q
10 split, -- On sépare en deux buts
11 exact p,
12 exact q,
13
14 -- Deuxième cas : r de type R
15 right, -- On choisit de montrer
16   ↪ P ∧ R
17 split, -- On sépare en deux buts
18 exact p,
19 exact r, -- Ce qui conclut la preuve de
20   ↪ l'implication directe.

```

2.8 Advanced Addition World

À ce niveau, nous avons déjà montré que $(\mathbb{N}_{\text{my nat}}, +)$ est un monoïde commutatif. Dans ce monde, nous allons prouver 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 $(\mathbb{N}_{\text{my nat}}, +)$ qu'on a prouvé au niveau 6 de ce monde. On a démontré que :

$$(a \ b \ c : \text{mynat}) : a + b = a + c \rightarrow b = c$$

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 $b = 0$.

```

1 lemma add_left_eq_zero {{a b : mynat}} :
  ⇨ a + b = 0 → b = 0 :=
2 begin [nat_num_game]
3   intro H,
4   --On fait une distinction de cas
5   --Soit b = 0 soit il existe d : mynat
  ⇨ tel que b = succ(d) :
6   cases b with d,
7
8   --Cas b = 0, le but devient 0 = 0, la
  ⇨ résolution est triviale :
9   refl,
10
11  --Cas b = succ(d), le but devient
  ⇨ succ d = 0
12  --Ce qui est impossible, cela
  ⇨ contredit l'axiome de Peano
  ⇨ zero_ne_succ
13  --On va donc faire une preuve par
  ⇨ l'absurde :
14  rw add_succ at H, --on fait rentrer a
  ⇨ dans le succ donc H : succ(a + d) = 0
15  exfalso, -- le but est impossible à
  ⇨ prouver donc on le change en faux
16  --Et on a le théorème succ_ne_zero
  ⇨ (n : mynat) : succ n = 0 → faux, donc
  ⇨ on sait que l'hypothèse H
  ⇨ implique faux
17  exact succ_ne_zero H,
18  --On a donc prouver que l'hypothèse
  ⇨ ∃ d ∈ ℕmynat, b = succ(d) est est
  ⇨ contradictoire avec nos axiomes
19 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 ins-

tinutive mais la preuve nécessite en réalité beaucoup de distinctions de cas et l'utilisation d'une nouvelle tactique, `revert`.

```

1 theorem mul_left_cancel (a b c : mynat)
  ⇨ (ha : a = 0) : a * b = a * c → b = c
  ⇨ :=
2 begin
3   revert b,
4   --On ne considère plus b comme une
  ⇨ hypothèse,
5   --à la place, on rajoute un
  ⇨ ∀(b : mynat) au but
6   --Ce sera utile plus tard, dans
  ⇨ l'hypothèse d'induction
7
8   -- On fait une induction sur c :
9   induction c with n hn,
10
11  --Le cas de base
  ⇨ ∀(b : mynat), a * b = a * 0 → b = 0 :
12  rw mul_zero, --On simplifie
13  intros b h, --On introduit un b et
  ⇨ l'hypothèse h : a * b = 0
14  rw mul_eq_zero_iff a b at h, --h est
  ⇨ équivalent à a = 0 ∨ b = 0 donc on
  ⇨ la réécrit
15
16  --On casse le a = 0 ∨ b = 0 en deux cas:
17  cases h with hha hhb,
18
19  --Si a = 0 (but : b = 0) :
20  --On a a ≠ 0 en hypothèse donc on sait
  ⇨ que ce cas est impossible
21  --On va donc faire une preuve par
  ⇨ l'absurde :
22  exfalso, --but = false
23  apply hha, --but = a = 0
24  exact hha, --Il n'y a plus qu'à
  ⇨ appliquer l'hypothèse de
  ⇨ disjonction de cas
25
26  --Si b = 0 (but : b = 0), c'est trivial
  ⇨ :
27  exact hhb,
28
29  --Le cas d'induction (but : ∀(b :
  ⇨ mynat), a * b = a * succ n → b = succ n)
  ⇨ :
30  intros b h, --On introduit un b et
  ⇨ l'hypothèse h : a * b = a * succ n
31  --Le but est juste b = succ n maintenant
32  --On fait une distinction de cas sur b
  ⇨ :
33  cases b with c,

```



```

34
35 --Cas b = 0 (but : 0 = succ n) :
36 --Cela contredit l'axiome de Peano
   ⇨ zero_ne_succ, on va donc passer
   ⇨ par l'absurde :
37 rw mul_zero at h, --On simplifie h
   ⇨ pour obtenir h : 0 = a * succ n
38 ex falso,
39 apply mul_pos a (succ n), --On a
   ⇨ besoin de démontrer les hypothèses
   ⇨ de mul_pos :
40 --Hypothèse a ≠ 0 :
41 exact ha,
42 --Hypothèse succ n ≠ 0 :
43 exact succ_ne_zero n,
44 --Retour à la preuve par l'absurde :
45 symmetry,
46 exact h,
47
48 --Cas b = succ c (but : succ c = succ n) :
49 repeat {rw succ_eq_add_one},
50 rw add_right_cancel_iff, --On
   ⇨ simplifie le but pour lui
   ⇨ appliquer l'hypothèse d'induction
51 --C'est là que le revert b prend tout
   ⇨ son importance car le but est
   ⇨ c = n
52 --On n'aurait pas pu appliquer
   ⇨ l'hypothèse d'induction si elle
   ⇨ prenait un b particulier
53 apply hn,
54 --Il ne reste plus qu'à simplifier
   ⇨ l'hypothèse h :
55 repeat {rw mul_succ at h},
56 rw add_right_cancel_iff at h,
57 exact h,
58 end

```

2.10 Inequality World

Dans ce monde, nous allons définir et prouver des propriétés 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).

```

1 lemma lt_aux_one (a b : mynat) : a ≤ b
   ⇨ ∧ ¬ (b ≤ a) → succ a ≤ b :=
2 begin
3   --On commence par transformer
   ⇨ a ≤ b ∧ ¬(b ≤ a) en 2 hypothèses :
4   intro h,
5   cases h with hab htba,
6   --On introduit c tel que b = a + c
7   cases hab with c hc,
8
9   --On ne peut rien faire à ce niveau
   ⇨ car le cas b = 0 pose problème
10  --On fait donc une distinction de cas
   ⇨ sur b :
11  cases b,
12
13  --Cas b = 0 (impossible donc par
   ⇨ l'absurde) :
14  ex falso,
15  apply htba,
16  exact zero_le a,
17
18  --Cas b = succ b (but : succ a ≤ succ b) :
19  --Là, c'est le cas a = 0 qui nous pose
   ⇨ problème
20  cases a,
21
22  --Cas a = 0 (trivial vu que b ≠ 0) :
23  apply succ_le_succ,
24  exact zero_le b,
25
26  --Cas a = succ a (but :
   ⇨ succ(succ a) ≤ succ b) :
27  rw hc,
28  rw succ_add,
29  apply succ_le_succ, --but :
   ⇨ succ a ≤ a + c
30
31  --Cette inégalité est impossible si
   ⇨ c = 0
32  cases c,
33
34  --Cas c = 0
35  rw add_zero at hc,

```

```

36   exfalse,
37   apply htba,
38   use 0,
39   rw add_zero,
40   symmetry,
41   exact hc,
42
43   --Cas  $c = \text{succ } c$  (but :  $\text{succ } a \leq a + \text{succ } c$ )
44   ↪ :
45   --Les +1 se simplifient
46   rw add_succ,
47   apply succ_le_succ,
48   use c,
49   refl,
50 end

```

3 Excursion dans le formalisme des espaces métriques

3.1 Avant-propos

Cette partie du projet réalise plus de 1370 lignes de Lean, il est préférable d'aller à l'URL suivante : <https://github.com/RaitoBezarius/projet-maths-lean/tree/master/src/espaces-metriques> afin de pouvoir apprécier les preuves correctement coloriées, plutôt que dans ce rapport, dont les contraintes d'espaces nous ont forcé à présenter un nombre réduit de preuves et de détails.

3.2 Tactiques

3.2.1 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 .

3.2.2 use

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

3.2.3 obtain

Cette tactique est une combinaison de deux tactiques : `have` et `rcases`. `obtain $\langle \text{patt} \rangle : \text{type}$` est équivalent à `have $h : \text{type}$, rcases h with $\langle \text{patt} \rangle$` .

Si `type` n'est pas prouvé, la syntaxe à utiliser sera `obtain $\langle \text{patt} \rangle : \text{type} := \text{proof}$`

3.2.4 cases

On dit qu'un système possède des types inductifs s'il permet de créer un nouveau type à l'aide des constantes et des fonctions qui créent des termes de même type. Ces termes autorisent la récursion structurelle et l'auto-référence.

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 $\text{succ}(d) = d + 1$.

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 $\text{succ}(d)$ avec l'hypothèse $h : A$ $\text{succ}(d)$.

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 deux hypothèses $p : P$, $q : Q$.

3.2.5 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 $h : (a \rightarrow b) \wedge (c \rightarrow d)$, alors `rcases h with $\langle A, B \rangle \mid \langle C \rangle$` remplace h par $A : a$, $B : b$, $C : c \rightarrow d$.

Donc $\langle A, B \rangle \mid \langle C \rangle$ divise l'hypothèse en deux constructeurs inductives, récupère les deux premiers paramètres du premier constructeur en A et B et récupère le deuxième constructeur en C .

3.3 cauchy_est_bornee

3.3.1 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 :

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

3.3.2 Énoncé du lemme

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

3.3.3 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 deux ensembles :

$$E_1 = \{U_n \mid n \leq n_0\} \text{ et } E_2 = \{U_n \mid n > n_0\}.$$

Soit Y un élément quelconque de la suite (U_n) .

- $\{U_n \mid n \leq n_0\}$ 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 $D_2 = \{d(U_n, Y) \mid n > n_0\}$. 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** $\langle N, H \rangle : \exists N, \forall p \geq N, \forall q \geq N, (d(x_p)(x_q)) < 1$

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 de Cauchy sur x et le fait que 1 soit un entier strictement positif.

À ce niveau, le **goal** est : $\exists (M : \mathbb{R}), M > 0 \wedge \forall (n : \mathbb{N}) d(x_n) y \leq M$.

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

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).

Afin de majorer cet ensemble, on peut 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 et retourne la distance entre y et x_n .

On montre que l'ensemble `Limage` est l'image de l'ensemble $\{i : \mathbb{N} \mid i \leq N\}$ 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 $\{i : \mathbb{N} \mid i \leq N\}$ 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 $d(x_0) y \in \text{Limage}$ 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`.

Donc, il existe un entier $n \leq N$ tel que $d(x_n) y = \max(\text{Limage})$.

- **use** $(\max(d(x_n) y, 1 + d(x_N) y))$

De cette façon le **goal** devient de démontrer que $\max(d(x_n) y, 1 + d(x_N) y)$ est un majorant de la suite de Cauchy x en question. C'est à dire qu'il faut montrer que $\max(d$

$(x_n) y, 1 + d(x_N) y) > 0$ (ce qui est immédiat),
et que $\forall n_1 : \mathbb{N}, d(x_{n_1}) y \leq \max(d(x_n) y, 1 + d(x_N) y)$

- `intro p, by_cases (p N)`

C'est à dire qu'on prend un entier p et on montre la propriété dans deux 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 $N \geq N$, $d(x_p)(x_N) < 1$.

Alors $d(x_p)(x_N) + d(x_N) y \leq 1 + d(x_N) y$.

Donc à fortiori $d(x_p)(x_N) + d(x_N) y \leq \max(d(x_n) y, 1 + d(x_N) y)$.

Or d'après l'inégalité triangulaire $d(x_p) y \leq d(x_p)(x_N) + d(x_N) y$.

Donc $d(x_p) y \leq \max(d(x_n) y, 1 + d(x_N) y)$.

Dans le cas où $p < N$:

On applique l'hypothèse `sup_atteint` pour majorer $d(x_p) y$ par $d(x_n) y$, puisque $d(x_n) y$ est la borne supérieure de l'ensemble `Limage` et $d(x_p) y \in \text{Limage}$.

Donc $d(x_p) y \leq \max(d(x_n) y, 1 + d(x_N) y)$.

3.4 cauchy_admet_une_va

Ici nous allons détailler la preuve de l'unicité de la valeur d'adhérence de toute suite de Cauchy, qui, en Lean, s'énonce comme ceci :

lemma `cauchy_admet_une_va` ($x : \mathbb{N} \rightarrow \mathbb{R}$) : `cauchy x` $\rightarrow \forall l_1 l_2 : X$, `adhere x l_1` \wedge `adhere x l_2` $\rightarrow l_1 = l_2$

Pour cela, on a besoin d'une autre preuve, à savoir :

lemma `eq_of_dist_lt` ($xy : X$) : ($\forall \varepsilon > 0, dxy < \varepsilon$) $\rightarrow x = y$

Nous l'admettrons ici (la preuve est dans le code) pour nous focaliser sur `cauchy_admet_une_va`, dont nous exposerons le principe de la preuve ci-dessous.

```
1 lemma cauchy_admet_une_va
  ⇨ (x : ℕ → ℝ) : cauchy x → ∀ l₁ l₂ :
  ⇨ X, adhere x l₁ ∧ adhere x l₂ → l₁ = l₂ :=
2 begin
3   intros cauch l1 l2 h,
4   apply eq_of_dist_lt,
5   intros ε hε,
6   have hε3 : ε/3 > 0 := by linarith,
7   obtain n , h_cauchy := cauch (ε/3)
     ⇨ hε3,
```

```
8   obtain p₁, hp₁, hl₁ := h.1 (ε/3)
     ⇨ hε3 (n),
9   obtain p₂, hp₂, hl₂ := h.2 (ε/3)
     ⇨ hε3 (n),
10  rw espace_métrique.sym l2 (x p₂) at
     ⇨ hl₂,
11  have Tr := espace_métrique.triangle (x
     ⇨ p₁) (x p₂) l2,
12  have Hc := h_cauchy p₁ hp₁ p₂ hp₂,
13  calc
14    d l1 l2 ≤ d l1 (x p₁) + d (x p₁) l2
     ⇨ : espace_métrique.triangle _ _ _
15    ... < ε/3 + d (x p₁) l2 :
     ⇨ add_lt_add_right hl₁ (d (x p₁)
     ⇨ l2)
16    ... ≤ ε/3 + (d (x p₁) (x p₂) + d
     ⇨ (x p₂) l2) : add_le_add_left
     ⇨ Tr (ε/3)
17    ... = d (x p₁) (x p₂) + (ε / 3 + d
     ⇨ (x p₂) l2) : by ring
18    ... < ε/3 + (ε/3 + d (x p₂) l2) :
     ⇨ add_lt_add_right Hc (ε/3 + d
     ⇨ (x p₂) l2)
19    ... = ε/3 + ε/3 + d (x p₂) l2 : by
     ⇨ ring
20    ... < ε/3 + ε/3 + ε/3 :
     ⇨ add_lt_add_left hl₂ (ε/3 + ε/3)
21    ... = ε : by ring,
22 end
```

Après avoir introduit les différentes variables, on utilise `eq_of_dist_lt` pour dire qu'il suffit de montrer que $\forall (\varepsilon : \mathbb{R}), \varepsilon > 0 \rightarrow dxy < \varepsilon$. On introduit les variables, utilisons la tactique `linarith` qui résout les inégalités triviales pour la propriété qui nous intéresse.

Ensuite, à l'aide de la tactique `obtain`, on dispose d'abord de n_0 tel que $\forall p, q \geq n_0, d(xp)(xq) < \varepsilon/3$, ce qui est possible car x est de Cauchy, puis p_1 et p_2 tels que $p_1, p_2 \geq n_0$ et $d(xp_i)l_i < \varepsilon/3, i \in \{1, 2\}$, ce qui est garanti par le fait que l_1 et l_2 sont des valeurs d'adhérence de x .

Ensuite il ne reste que du calcul formel à effectuer, que l'on fait grâce à la tactique d'environnement `calc`, qui permet ici de partir d'un côté du résultat, et par suite d'égalités ou d'inégalités strictes ou larges - toujours par rapport à l'étape précédente - pour arriver à l'autre côté du résultat, en accolant la preuve de chaque (in)égalité à droite de celle-ci. `calc` est une tactique qui fonctionne pour n'importe quelle relation qui vérifie la transitivité. Ici on a donc utilisé l'inégalité triangulaire, la symétrie de la distance et quelques lemmes fondamentaux

d'inégalité, ainsi que la tactique `ring` qui résoud des égalités simples dans un anneau.

3.5 \mathbb{R} est un espace complet

Nous allons détailler sans rentrer dans les preuves, comment démontrer que \mathbb{R} est un espace complet, qui repose axiomatiquement seulement sur l'axiome de la borne supérieure et le fait que \mathbb{R} soit archimédien :

- Un lemme type Bolzano-Weierstrass qu'on retrouvera dans `custom/bolzano_weierstrass.lean`
- Un lemme qui nous donne la convergence d'une suite de Cauchy sous réserve d'existence d'une valeur d'adhérence : `converge_of_va_for_cauchy` dans `custom/cauchy.lean`

Le second lemme repose sur des plus petits lemmes : une suite de Cauchy admet au plus une valeur d'adhérence, décrit sous le nom de `cauchy_admet_une_va`.

Le premier lemme requiert aussi un lemme de bornitude des suites de Cauchy, sous le nom de `cauchy_est_bornee`, décrit précédemment.

3.5.1 Structure du code

3.5.1.1 `custom/sequences.lean` On a des lemmes autour des suites, on a les définitions élémentaires de convergence, être de Cauchy, adhérence, bornitude, sous-suites et stricte croissance.

On y démontre une forme d'équivalence entre l'adhérence et sa version séquentielle, des lemmes autour de l'affaiblissement dénombrable du quantificateur réel devant les $\forall \varepsilon > 0$.

3.5.1.2 `custom/si_sequences.lean` On définit des lemmes autour de la construction des suites strictement croissantes dans un ensemble infini complètement linéairement ordonné⁵ (qu'on appliquera donc principalement à \mathbb{R}).

Petit point sur ce qui se passe, car il s'agit de preuves non classiques, on exploite une autre forme d'infinitude, celle qui indique qu'un ensemble fini c'est exactement un ensemble pour lesquelles toutes ses parties non vides ont un plus grand et plus petit élément.

⁵Cette hypothèse est probablement trop forte, on peut sûrement exploiter des notions fines comme la topologie de l'ordre, mais nous n'avons pas eu le temps d'explorer cela.

Cela nous fournit une preuve efficace pour tirer une suite strictement croissante, qu'on utilisera dans un raisonnement par l'absurde pour Bolzano-Weierstrass.

Un élément intéressant de ce fichier c'est la première utilisation de `well_founded.fix` avec l'ordre bien fondé de $\mathbb{N} : \text{nat.lt_wf}$, Lean permet de construire des objets donc par « récurrence »⁶.

Cela fonctionne comme ceci : `well_founded.fix` effectue un calcul de point fixe, cela prend en premier argument un ordre bien fondé, et en second un argument une fonction qui prend un élément, disons n , de l'ensemble bien fondé et la fonction partiellement construite pour tout n au sens de l'ordre bien fondé, on doit retourner le nouveau terme induit.

Cependant, cela ne fait que construire la suite ou l'objet désiré, cela ne donne pas les propriétés sur lui qu'on veut.

Pour cela, c'est quelque chose qu'on retrouve classiquement dans Lean, on utilise un théorème type « spec »⁷, en l'occurrence : `well_founded.fix_eq` qui prouve que le calcul de point fixe vérifie l'équation du point fixe.

De là, on construit notre suite par opérations ensemblistes, on pourrait explorer s'il était faisable de faire ça avec des types et des sous-types plutôt que des vrais ensembles.

3.5.1.3 `custom/sups.lean` On y démontre des lemmes autour des bornes supérieures et inférieures, et des lemmes de type topologiques.

Notamment, le fait que les sups et infs sont des points limites dès lorsqu'ils ne sont pas des max/min.

On exploite `linarith` principalement et des morceaux de la `mathlib` sur les ordres qui ne reposent pas sur \mathbb{R} .

3.5.1.4 `custom/topology.lean` Des définitions topologiques :

- Complétude
- Adhérence à un ensemble
- Point limite

3.5.1.5 `custom/negative_sets.lean` On y démontre des lemmes autour des ensembles opposés, i.e. pour S un ensemble, on regarde $-S := \{x \mid -x \in S\}$.

⁶On est plutôt dans l'induction bien-fondée, en réalité.

⁷`classical.spec`, etc.

On y démontre principalement des lemmes autour des sups/infs et de leurs adhérences.

3.5.1.6 custom/cauchy.lean On y démontre des lemmes autour des suites de Cauchy, on détaillera plus ce fichier dans la partie sur les complétés.

3.5.1.7 custom/bolzano_weierstrass.lean
La preuve de Bolzano-Weierstrass en deux versions :

- Version 1 : Toute suite bornée a une valeur d'adhérence.
- Version 2 : Un ensemble infini borné a des points limites.

La version 2 entraîne la version 1, nous allons montrer en quoi.

3.5.2 Bolzano-Weierstrass

Démontrer Bolzano-Weierstrass proprement et efficacement en partant de l'axiome de la borne supérieure est difficile en raison de la quantité de lemmes requis, nous retrouverons donc la plupart des constructions nécessaires dans les fichiers précédents.

Donnons la feuille de route pour le théorème final (Bolzano-Weierstrass version 2) :

- On a besoin du lemme fondateur : `lemme_fondateur_de_bw` : qui prouve la version d'infinitude dont on a parlé dans le fichier des suites strictement croissantes, par contraposée.
- Ensuite, on prouve `bolzano_weierstrass_v2` en utilisant la nouvelle définition de l'infinitude, par l'absurde, en supposant que l'ensemble n'a pas de point limite, puis en montrant que l'ensemble est fini, puisqu'il n'a pas de point limite, son sup/inf qui en seraient sont nécessairement des max/min, d'où, ceci étant vrai pour toutes les parties, on en tire la finitude, donc l'absurdité.
- Ensuite, on prouve la version 1 : `bolzano_weierstrass`, en commençant par une distinction sur la cardinalité des valeurs prises par la suite, si c'est fini, on se contente d'utiliser le principe des tiroirs, sinon, on recourt à la version 2, on s'en tire en démontrant qu'on a une adhérence séquentielle en utilisant la tactique `choose`.

3.6 Interlude : Continuité séquentielle

3.7 Continuité Séquentielle

Ici on va démontrer que la continuité séquentielle est stable par composition, i-e :

theorem `comp_seq_continue` $(f : X \rightarrow Y)(g : Y \rightarrow Z) : \text{seq_continue } f \wedge \text{seq_continue } g \rightarrow \text{seq_continue } (g \circ f)$

Tout d'abord définissons la continuité séquentielle d'une fonction, en un point puis en tous points :

def `seq_continue_en_l` $(f : X \rightarrow Y)(L : X) := \forall (x : \mathbb{N} \rightarrow X), \text{converge } x L \rightarrow \text{converge } (\text{suite_image } f x) (f L)$

def `seq_continue_en_l` $(f : X \rightarrow Y) := \forall X : L, \text{seq_continue_en_l } f L$

Ensuite il faut définir une fonction qui renvoie la suite composée par deux fonctions, ce qu'on fait dans :

theorem `comp_seq_continue` $(f : X \rightarrow Y)(g : Y \rightarrow Z)(x : \mathbb{N} \rightarrow X) : \text{suite_image } g (\text{suite_image } f x) = \text{suite_image } (g \circ f) x$

On ne détaillera pas la preuve - rapide - de cette fonction.

On définit ensuite la continuité séquentielle en un point de la composée :

def `comp_seq_continue_ponctuel` $(f : X \rightarrow Y)(l : X) : \text{seq_continue_en_l } f l \wedge \text{seq_continue_en_l } g (f l) \rightarrow \text{seq_continue_en_l } (g \circ f) l$

Pour cette preuve, après introduction des variables, on crée une variable, on prouve que la suite image par g de la suite image par f de x converge vers $g(f(l))$, ce qu'on modifie grâce à la tactique `conv` en "la suite image par $g \circ f$ de x converge vers $g(f(l))$ ", ce qui correspond exactement au but.

Finalement, pour prouver la continuité séquentielle en tous points de la composée, il suffit d'introduire les variables, appliquer la tactique `apply` au théorème précédent, puis montrer grâce aux hypothèses que pour tout point $l : X$, f est continue en l et g en $f(l)$.

3.8 Complété d'un espace métrique

Soit (X, d) un espace métrique.

À présent, nous allons essayer de construire un complété de l'espace métrique X en tant que quotient par la relation d'équivalence $x \sim y \iff \lim d(x_n, y_n) = 0$ où x, y sont des suites de Cauchy sur l'espace métrique X .

Tout d'abord, on se donne quelques lemmes :

- l'unicité de la limite ;
- une inégalité triangulaire pour le pré-écart ;
- la preuve qu'un pré-écart est une suite de Cauchy ;

À ce stade-là, puisque le pré-écart est de Cauchy, par complétude \mathbb{R} , il a toujours une limite.

Donc, on définit `cauchy.limit` la fonction qui associe l'unique limite d'une suite de Cauchy à valeurs réelles.⁸

On remarquera qu'on utilise `classical.some` qui est l'incarnation de l'axiome du choix, afin de choisir une limite.

Puis, après, on peut la rendre unique par les lemmes précédents.

On définit des lemmes utiles :

- Une suite constante est de Cauchy ;
- Limite d'une suite constante ;
- Passage à la limite dans les inégalités ;

Enfin, on définit la distance « de Cauchy » qui est $d_C(x, y) = \lim_n d(x_n, y_n)$, bien définie.

On montre que cette distance induit un espace pré-métrique sur les suites de Cauchy à valeurs dans X .

Dernière étape pour la complétion : montrer que $x \sim y \iff d_C(x, y) = 0$ est bien une relation d'équivalence.

Pour cela, on définit `cauchy.cong` comme étant le prédicat désiré, puis on démontre les trois lemmes classiques : réflexivité, symétrie, transitivité, ce qui nous donne le lemme d'équivalence : `equivalence (cauchy.cong X)`.

Enfin, on instancie `setoid (cauchy_seqs X)`, avec tout ce qui précède.

Pour terminer, on définit la complétion comme quotient (`pre_ecart.setoid`), i.e. la complétion est exactement le quotient par la relation

⁸On pourrait généraliser à des espaces complets arbitraires. Cela coûte trois fois rien.

d'équivalence sur la « distance de Cauchy » ou la limite du pré-écart.

Il reste à prouver que la complétion induit bien un espace métrique, pour cela, il faut procéder en plusieurs étapes :

- Il faut transporter la distance de Cauchy, variable sur les suites de Cauchy aux classes d'équivalences (suites de Cauchy quotientées par la distance)
- Il faut transporter la structure pré-métrique acquise : cela se fait en utilisant `quotient.lift_2`, en montrant qu'elles sont compatibles pour la relation d'équivalence (appelé `soundness` en général).
- Il faut démontrer la séparabilité, en invoquant le fait que si une distance s'annule, c'est exactement le fait que $x \sim y$, i.e. $\bar{x} = \bar{y}$.

Pour aller plus loin, il faudrait continuer à transporter toutes les lois compatibles dans le complété, rajouter un lemme de coercion afin de pouvoir injecter tout élément de X dans \bar{X} , le complété.

Une fois ceci fait, on aura un structure d'espace métrique complété élémentaire mais utilisable.

3.8.1 Détails du fichier `custom/cauchy.lean`

On y retrouve les lemmes suivants :

- `cauchy_est_bornee` : déjà expliqué.
- `dist_lt_of_ne` : on prouve que si la distance est non nulle, elle est strictement positive.
- `eq_of_dist_lt` : on prouve que si la distance entre deux éléments est aussi petite qu'on veut, alors ces éléments sont égaux.
- `cauchy_admet_une_va` : déjà expliqué.
- `converge_of_va_for_cauchy` : si une suite de Cauchy a une valeur d'adhérence, elle converge vers celle-ci.
- `converge_of_unicite_va` : sous réserve de complétude, existence et unicité de la valeur d'adhérence entraîne convergence (**non terminé**).
- `unicite_limite` : prouve l'unicité de la limite dans un espace métrique.
- `R_is_complete` : prouve la complétude de \mathbb{R} .
- `dist_ineg_utile` : prouve un lemme utile pour prouver que le pré-écart est de Cauchy à base d'inégalité triangulaires.
- `pre_ecart_cauchy` : prouve que le pré-écart est de Cauchy.
- `pre_ecart_triangle` : prouve que le pré-écart satisfait l'inégalité triangulaire.

- `cauchy.limit` : renvoie **une**⁹ limite d'une suite de Cauchy.
- `cauchy.converge_of_limit` : prouve que toute suite de Cauchy x converge vers `cauchy.limit x`
- `cauchy.limit_ge_of_seq_ge` : passage à la limite à gauche dans une inégalité \geq avec une constante (**non terminé**).
- `cauchy.limit_le_of_seq_le` : passage à la limite à gauche dans une inégalité \leq avec une constante (**non terminé**).
- `cauchy.le_of_limit_le` : passage à la limite à droite dans une inégalité \leq avec une suite (**non terminé**).
- `cauchy.limit_le_of_add_seq_le` : passage à la limite dans une inégalité du type : $\forall n, x_n \leq y_n + z_n$ dans toutes les suites (**non terminé**).
- `cauchy.cauchy_of_constant_real_seq` : une suite constante est de Cauchy.
- `cauchy.converge_of_constant` : une suite constante converge vers sa constante.
- `cauchy.constant_limit` : une limite de Cauchy d'une suite constante est sa constante.
- `cauchy.pre_ecart_sym` : symétrie du pré-écart.

3.8.2 Détails du fichier `metric.lean`

- `cauchy.dist` : la définition de la distance de Cauchy en tant que limite de Cauchy du pré-écart.
- `cauchy.d_pos` : positivité de la distance de Cauchy.
- `cauchy.pre_ecart_self_eq_zero_seq` : le pré-écart est nul pour la même suite, i.e. $(d(x_n, x_n))_n = 0$.
- `cauchy.cong` : le prédicat : être de distance de Cauchy nul.
- `cauchy.cong_xxx` : la preuve de réflexivité, symétrie, transitivité et d'équivalence de `cauchy.cong`.
- `pre_ecart.setoid` : l'instance de la relation d'équivalence précédente.
- `completion.dist_soundness` : la propriété de compatibilité de la distance de Cauchy sous la relation d'équivalence¹⁰
- `completion` : définition de la complétion en tant que quotient.
- `completion.dist` : distance de Cauchy sous la complétion, qui devient une vraie distance

⁹L'unicité provient de `unicite_limite`.

¹⁰Contient un bug bizarre ou difficile à comprendre sur les typeclasses et `espace_pre_metrique` au niveau de la réécriture symétrique. En cours de résolution...

pour le (futur) espace métrique.

- `completion.dist_self_eq_zero` : annulation de la distance pour les mêmes arguments.
- `completion.d_pos` : positivité de la distance complétée.
- `completion.metric_space` : l'instance de l'espace métrique pour le complété (**non terminé**).

Références

- [1] Behzad Akbarpour and Lawrence Charles Paulson. Metitarski : An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3) :175–205, 2010.
- [2] Emil Artin. *Algebraic numbers and algebraic functions*, volume 358. American Mathematical Soc., 2005.
- [3] Jeremy Avigad. The lean theorem prover. *Microsoft Research, Carnegie Mellon University*, 2014.
- [4] Seulkee Baek. *Reflected Decision Procedures in Lean*. PhD thesis, Master's thesis. Carnegie Mellon University. <http://www.andrew.cmu.edu> ..., 2019.
- [5] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library : a formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172, 2017.
- [6] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer Science & Business Media, 2013.
- [7] 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.
- [8] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
- [9] Jean-Yves Girard. *Le point aveugle : cours de logique*. Hermann, Paris, 2006.
- [10] Buzzard Kevin. Natural number games. https://github.com/ImperialCollegeLondon/natural_number_game, 2019.

- [11] 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.
- [12] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
- [13] William Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing'91 : Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. IEEE, 1991.
- [14] Andrzej Trybulec and Howard A Blair. Computer assisted reasoning with mizar. In *IJCAI*, volume 85, pages 26–28, 1985.
- [15] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- [16] Matthew Wampler-Doty. A complete proof of the robbins conjecture. *The Archive of Formal Proofs*. <http://afp.sf.net/entries/Robbins-Conjecture.shtml>, 2010.