

Rapport

MARYEM HAJJI, LÉA Riant, RYAN LAHFA, IVAN HASENOHR

Table des matières

Introduction	1
Courte histoire des assistants de preuve et du rêve d'Hilbert	1
Principe d'un assistant de preuves . . .	1
Enjeu d'un assistant de preuves et exemples d'usages	1
Éléments de théorie des assistants de preuves	1
Objectifs de ce projet	2
Détail des exercices du « Number Games » de Kevin Buzzard	2
Tactiques de base en Lean	2
Addition World	4
Multiplication World	4
Power World	5
Function World	6
Proposition World	6
Advanced Proposition World	6
Excursion dans le formalisme des es- paces métriques	9

Introduction

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

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 [5]) 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[3] 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[8], HOL-Isabelle[7] et Coq [4] naissent pour devenir les assistants de preuve mathématiques que l'on connaît.

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, la théorie des types homotopiques 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).

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.

Éléments de théorie des assistants de preuves

Nous 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.

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 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.

Dans une certaine mesure, Lean 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).

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 qui consiste à redémontrer quelques théorèmes autour des entiers naturels en partant des axiomes de Peano.

Nous fournissons en XXX, 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 [6] 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'Ostrowski² en posant la théorie des valuations d'Artin [1].

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 .³

Tactiques de base en Lean

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 .

intro 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

²Dont le livre d'Artin fournit une démonstration

³i.e. tous les mots sur Σ

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** $h_1 h_2 \dots h_n$, pour introduire plusieurs hypothèses en même temps. On entrera plus en détails dans la structure des implications dans le *Function World*.

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.

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**.

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 A s (présents dans F) sont réécrits en B s. De même, si F contient au moins un B et si on utilise **rw** $\leftarrow H$, alors le seul changement sera : tous les B s (présents dans F) sont réécrits en A s.

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** $\leftarrow T$) dérive un mot F' du mot F , en effectuant un seul changement : tous les A s (resp. B s) sont remplacés par des B s (resp. A s).

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, }**

Exact, Intro, Have, Apply Ici nous allons présenter 4 techniques fondamentales pour l'utilisation de fonctions, 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 .

Exact

La première de ces tactiques est *exact*. Elle permet de dire à Lean que le but recherché correspond exactement à ce que vous lui indiquez. Par exemple, si le but est $\exists p$ de type 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 $\exists q$ de type 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.

Intro

Lorsque vous manipulez des fonctions, Lean peut vous demander de créer une fonction d'un type P vers un type Q . Une méthode est alors de d'émettre l'hypothèse qu'on dispose d'un p de type P à partir duquel vous fabriquerez un élément de Q . *intro* p , fait cela : vous disposerez alors d'une preuve p de P et votre but sera reformulé en Q .

De façon similaire, lorsque de le but est de la forme $P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_n \rightarrow Q$, *intros* $p_1 p_2 \dots p_n$ change le but en Q .

Have

Cette technique permet de renommer des variables : par exemple, si vous disposez de p de type P et de $f : P \rightarrow Q$, alors *have* $q : Q := f(p)$, vous permet de renommer un élément $q = f(p)$. Le principe du Démonstrateur nous permet en effet de renommer comme on veut ce que l'on veut, ce qui garantit la validité de la preuve dans le cas de l'utilisation de *have*.

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 dans Q . *apply f*, fait exactement ça, et donc changera le but de Q en P .

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.

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 :

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
```

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 :

Niveau 4 : `mul_add`

La multiplication est distributive, 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 :
    ↪  $t \times (d+b) = t \times d + t \times b$ 
12 --et montrons  $h(\text{succ}(d))$ :  $t \times (\text{succ}(d)+b)$ 
    ↪  $= t \times \text{succ}(d) + t \times b$ 
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  $\text{succ}(d)+b$  ( $d$  et  $b$  étant
    ↪ deux entiers naturels quelconques),
    ↪ nous permettant ainsi
18 --de la remplacer par  $\text{succ}(d+b)$ 
19 rw mul_succ, -- on utilise mul_succ ( $a$   $b$ 
    ↪ :  $\text{mynat}$ ) :  $a \times \text{succ}(b) = a \times b + a$ 
20 rw hd, -- on remplace  $t \times (d+b) + t$  par
    ↪  $t \times d + t \times b + t$  en utilisant hd,
21 -- on obtient  $t \times d + t \times b + t =$ 
    ↪  $t \times \text{succ}(d) + t \times b$ 
22 rw add_right_comm, -- on applique la
    ↪ commutativité de l'addition pour
    ↪ remplacer  $t \times b + t$  par  $t + t \times b$ 
23 rw → mul_succ, --on utilise  $\text{rw} \leftarrow$  pour
    ↪ remplacer  $t \times d + t$  (qui est le membre
    ↪ droit
24 -- de l'égalité qui correspond au
    ↪ théorème mul_succ) par  $t \times \text{succ}(d)$ ,
25 -- on obtient  $t \times \text{succ}(d) + t \times b =$ 
    ↪  $t \times \text{succ}(d) + t \times b$ 
26 refl
27 end

```

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 expliquons la démonstration du théorème suivant :

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  $\text{succ}(1)=2$  pour réécrire
    ↪ le chiffre 2 en  $\text{succ}(1)$ 
5 rw one_eq_succ_zero, -- on réécrit 1 en
    ↪  $\text{succ}(0)$ , on obtient donc
6  $-(a + b)^{\text{succ}(\text{succ}(0))} = a^{\text{succ}(\text{succ}(0))} +$ 
    ↪  $b^{\text{succ}(\text{succ}(0))} + \text{succ}(\text{succ}(0)) \times a \times b$ 
7 repeat {rw pow_succ}, -- on obtient
    ↪  $(a + b)^0 \times (a + b) \times (a + b) =$ 
    ↪  $a^0 \times a \times a + b^0 \times b \times b + \text{succ}(\text{succ}(0)) \times a \times b$ 
8 repeat {rw pow_zero}, -- on obtient
    ↪  $1 \times (a + b) \times (a + b) =$ 
    ↪  $1 \times a \times a + 1 \times b \times b + \text{succ}(\text{succ}(0)) \times a \times b$ 
9 simp, -- on obtient  $(a + b) \times (a + b) =$ 
    ↪  $a \times a + (b \times b + a \times (b \times \text{succ}(\text{succ}(0))))$ 
10 --donc simp, dans ce cas, applique le
    ↪ théorème one_mul( $m : \text{mynat}$ ) :
    ↪  $m \times 1 = m$ 
11 repeat {rw mul_succ}, -- on obtient
    ↪  $(a+b) \times (a+b) = a \times a + (b \times b + a \times (b \times 0 + b + b))$ 
12 simp, -- on obtient
    ↪  $(a+b) \times (a+b) = a \times a + (b \times b + a \times (b + b))$ 
13 -- donc simp, dans ce cas, applique les
    ↪ théorèmes mul_zero( $a :$ 
    ↪  $\text{mynat}$ ):  $a \times 0 = 0$ 
14 -- et zero_add( $n : \text{mynat}$ ):  $0 + n = n$ 
15 -- on développe  $(a + b) \times (a + b) :$ 
16 rw mul_add,
17 -- on développe  $(a + b) \times a$ 
18 rw mul_comm,
19 rw mul_add,
20 -- on développe  $(a + b) \times b$ 
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
    ↪  $a \times a + (a \times b + a \times b + b \times b) =$ 
    ↪  $a \times a + (b \times b + a \times (b + b))$ 
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  $- a \times a + (b \times b + (a \times b + a \times b)) =$ 
    ↪  $a \times a + (b \times b + a \times (b + b))$ 
29 -- on factorise par  $a :$ 
30 rw ← mul_add a b b, -- on obtient
    ↪  $a \times a + (b \times b + a \times (b + b)) =$ 
    ↪  $a \times a + (b \times b + a \times (b + b))$ 
31 refl
32 end

```

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.

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.

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 $A \rightarrow B$.

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

lemma : $(P, Q : \text{Prop}) (p : P) (h : P \rightarrow Q) : Q$

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),

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
  ↳ fonction qui prend une preuve que
  ↳  $P \rightarrow Q$  et renvoie une preuve de
  ↳  $\neg Q \rightarrow \neg P$ .
4 intro f, -- On dispose d'une preuve de
  ↳  $P \rightarrow Q$ 
5 -- Lean nous demande alors de créer un
  ↳ élément de type  $\neg Q \rightarrow \neg P$ , qui
  ↳ serait l'image de la fonction qu'il
  ↳ nous est demandé de créer.
6 repeat{rw not_iff_imp_false}, -- On
  ↳ retranscrit la définition de  $\neg P$  :
  ↳  $\neg P \equiv P \rightarrow \text{false}$ .
  ↳ Le but est alors réécrit en
  ↳  $(Q \rightarrow \text{false}) \rightarrow P \rightarrow \text{false}$ , ce qui
  ↳ revient à créer une fonction
  ↳ curryfiée des éléments de type
  ↳  $(Q \rightarrow \text{false}) \times P$  vers les preuves de
  ↳ false. On réapplique la même
  ↳ technique d'introduire un élément de
  ↳ chacun des ensembles de départ :
8 intros h p,
9 -- 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 :
10 exact h(f(p)), -- Ce qui conclut la
  ↳ preuve.
```

Advanced Proposition World

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

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 :

```

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

```

Advanced Addition World : Level 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 :

```

1 (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 $b = 0$.

```

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  $d : \text{mynat}$ 
   ↪ tel que  $b = \text{succ}(d)$  :
6   cases b with d,
7
8   --Cas  $b = 0$ , le but devient  $0 = 0$ , la
   ↪ résolution est triviale :
9   refl,
10

```

```

11 --Cas  $b = \text{succ}(d)$ , le but devient
   ↪  $\text{succ } d = 0$ 
12 --Ce qui est impossible, cela
   ↪ contredit l'axiome de Peano
   ↪  $\text{zero}_n \text{e}_s \text{ucc}$ 
13 --On va donc faire une preuve par
   ↪ l'absurde :
14 rw add_succ at H, --on fait rentrer a
   ↪ dans le succ donc  $H : \text{succ}(a + d) = 0$ 
15 exfalse, -- le but est impossible à
   ↪ prouver donc on le change en faux
16 --Et on a le théorème  $\text{succ\_ne\_zero}$ 
   ↪  $(n : \text{mynat}) : \text{succ } n = 0 \rightarrow \text{faux}$ , donc
   ↪ on sait que l'hypothèse  $H$ 
   ↪ implique faux
17 exact succ_ne_zero H,
18 --On a donc prouver que l'hypothèse
   ↪  $\exists d : \text{mynat} \text{ tel que } b = \text{succ}(d)$  est est
   ↪ contradictoire avec nos axiome
19 end

```

Advanced Multiplication World : Level 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`.

```

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
   ↪  $\forall (b : \text{mynat})$  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
   ↪  $\forall (b : \text{mynat}), a * b = a * 0 \rightarrow b = 0$  :
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  $a = 0 \text{ or } b = 0$  donc on
   ↪ la réécrit
15
16   --On casse le  $a = 0 \text{ or } b = 0$  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 ha, --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_n_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 exfalso,
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

```

Inequality World Level 15 Dans la suite, nous allons définir $>$ tel que :

```

1 def a < b := a ≤ b ∧ ¬ (b ≤ a)

```

Mais la définition :

```

1 def a < b := succ a ≤ b

```

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

```

1 (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  exfalso,
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  exfalso,
37  apply htba,
38  use 0,
39  rw add_zero,
40  symmetry,
41  exact hc,
42
43  --Cas c = succ c (but : succ a ≤ a + succ c)
   ↪ :
44  --Les +1 se simplifient
45  rw add_succ,
46  apply succ_le_succ,
47  use c,
48  refl,
49  end

```

- [5] Jean-Yves Girard. *Le point aveugle : cours de logique*. Hermann, Paris, 2006.
- [6] 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.
- [7] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
- [8] Andrzej Trybulec and Howard A Blair. Computer assisted reasoning with mizar. In *IJCAI*, volume 85, pages 26–28, 1985.

Excursion dans le formalisme des espaces métriques

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