Rapport

Maryem Hajji, Léa Riant, Ryan Lahfa, Ivan Hasenohr

Table des matières

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) 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 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, HOL-Isabelle et Coq 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 : ZFC, 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.

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 écrite par les utilisateurs afin d'améliorer l'intelligence de Lean dans certains contextes (chasse aux diagrammes en catégories par exemple).

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. [1]

Tactiques de base en Lean

intro Parfois, le but que nous cherchons à atteindre est une implication. Pour prouver que $A- > B$, on va prouver $BsachantAvrai$. En Lean, cela revient à inclure A dans les hypothèses et à changer le but en 'B'. C'est ce que fait la tactic '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. On entrera plus en détails dans la structure des implications dans le Function World.

have Pour déclarer une nouvelle hypothèse, on peut utiliser la tactic *have*. *have p : P* diviser le but en 2 sous-but : 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.

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 appliquant la tactique $\{$**refl,**$\}$.

rw : Le nom de cette tactique correspond au mot anglais *rewrite*. Elle s'applique dans deux cas distincts.

[1] i.e. tous les mots sur Σ

Soit $H : A = B$ une hypothèse d'égalité. [2] 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 (présents dans F) sont réécrits en B. De même, si F contient au moins un B et si on utilise $\{$**rw** $\leftarrow$ **H,**$\}$, alors le seul changement sera : tous les B (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** $\leftarrow$ **T,**$\}$) dérive un mot F' du mot F, en effectuant un seul changement : tous les As (resp. Bs) sont remplacés par des Bs (resp. As).

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 : $A \to B$ 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.

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 : $P \to Q$, 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 $P_1 \to P_2 \to ... \to P_n \to Q$, *intros* $p_1 p_2 ... p_n$ 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 : $P \to Q$, alors *have q : Q := f(p),* vous permet de renommer un élément $q = f(p)$. Le principe du Démiurge 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 : $P \to Q$. De ce fait, pour disposer d'un élément de Q, il vous suffit de disposer d'un élément de P, car f(p) sera 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 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 :

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

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

[2] ou une preuve de $A = B$, c'est la même chose.

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 :
Le 5^{me} niveau : succ_eq_add_one

$$\forall \text{ entier naturel } n,\ succ(n) = n + 1$$

Preuve

```
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
```

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 :

Le 4^{me} niveau : mul_add

La multiplication est distributive, c'est à dire ∀ entiers naturels a, b et t :

$$t \times (a + b) = t \times a + t \times b$$

```
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 |$\to$| 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
```

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 :
Le 7^{me} niveau : 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 $n = 2$)

$$\forall \text{ entiers naturels } a \text{ et } b : \\ (a+b)^2 = a^2 + b^2 + 2 \times a \times b$$

```
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
```

IV : 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 :
$(PQR : Type) : (P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$
La preuve est de fait assez simple :
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.

V : 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 $A \to B$.

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

Lemme : if P is true and $P \to Q$ is true, then Q is true.

Soit en Lean : *(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),*.

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) :

Lemme : $(P \to Q) \to (\neg Q \to \neg P)$.

En Lean, on demande donc de créer une fonction qui prend une preuve que $P \to Q$ et renvoie une preuve de $\neg Q \to \neg P$.
Pour cela, la première étape est de disposer d'une preuve de $P \to Q$:
intro f,
Lean nous demande alors de créer un élément de type $\neg Q \to \neg P$, qui serait l'image de la fonction qu'il nous est demandé de créer.
L'astuce est ensuite de revenir à la définition de $\neg P$: $\neg P \equiv P \to false$. On retranscrit cette définition :
repeat{rw not_iff_imp_false},
Le but est alors réécrit en $(Q \to false) \to P \to false$, ce qui revient à créer une fonction curryfiée des éléments de type $(Q \to false) \times P$ 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.

VI : 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 consjonctions et disjonctions logiques. Un exemple combinant la plupart des nouvelles méthodes est le Lemme suivant :

Lemme : Soient P,Q et R trois propositions.
ALors $P \wedge (Q \vee R) \iff (P \wedge Q) \vee (P \wedge R)$.

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 $P \wedge (Q \vee R)$
cases h with p qor, – p de type P , qor de type $Q \vee R$
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 $P \wedge Q$
split, – On sépare en deux buts
exact p,
exact q,

– Deuxième cas : r de type R
right, – On choisit de montrer $/P \wedge R$
split, – On sépare en deux buts
exact p,
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 :

```
(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 succd = 0
  --Ce qui est impossible, cela contredit
   ↪  l'axiome de Peano zero_n e_s ucc
  --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) : succn = 0−> faux, donc on
   ↪  sait que l'hypothèse H implique faux
  exact succ_ne_zero H,
  --On a donc prouver que l'hypothèse
   ↪  ∃d : mynattelqueb = succ(d) est est
   ↪  contradictoire avec nos axiome
end
```

Advanced Multiplication World : Level 4 Ce théorème consiste à prouver la régularité à gauche du monoïde (mynat, *). L'idée est instinctive mais la preuve nécessite en réalité beaucoup de distinctions de cas et l'utilisation d'une nouvelle tactic, *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 = 0orb = 0 donc on la
   ↪  réécrit

  --On casse le a = 0orb = 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 * succn → b = succn)
   ↪  :
  intros b h, --On introduit un b et
   ↪  l'hypothèse h : a * b = a * succn
  --Le but est juste b = succn maintenant
  --On fait une distinction de cas sur b :
  cases b with c,

  --Cas b = 0 (but : 0 = succn) :
```

```
  --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'HR 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
```

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

```
    a < b := a \leq b \land \lnot (b \leq
      a)
```

Mais la définition :

```
    a < b := succ a \leq b
```

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

```
a \leq b \land \lnot (b \leq a) → succ a
  \leq b
```

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

```
lemma lt_aux_one (a b : mynat) : a \leq b
  \land \lnot (b \leqa) → succ a \leq 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 \leq 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
```

Excursion dans le formalisme des espaces métriques