Manuel de nos commandes Lean

Glossaire

Un *but* est un énoncé mathématique à démontrer. Toute démonstration commence avec un unique but. Certains types de raisonnements augmentent le nombre de buts, par exemple la démonstration d'une équivalence par double implication. Lean affiche les buts après le symbole ⊢. Le *but courant* est le premier but affiché par Lean.

Le *contexte* d'un but est la collection des objets et hypothèses fixés par l'énoncé ou les étapes de démonstrations déjà effectuées.

Un *nom* est une suite de lettres, chiffres, prime ou indices qui commence par une lettre. Un tel nom peut désigner un objet mathématique (nombre, fonction, etc...), une hypothèse, une définition ou un lemme.

Un *nom annoté* est une suite de symboles de la forme `nom : T` où `T` est soit un type d'objet (entier naturel, nombre réel, fonction...) soit un énoncé. L'annotation annonce le type de l'objet ou de l'énoncé nommé, essentiellement pour rendre la démonstration plus lisible. On peut ajouter des annotations dans une phase intermédiaire entre la première démonstration complète avec Lean et la rédaction sur papier.

Un *nom existant* est un nom que Lean connait déjà, soit parce qu'il désigne un objet ou une hypothèse du contexte courant soit parce que c'est le nom d'une définition ou d'un lemme connu. Un *nouveau nom* est un nom choisi par l'utilisateur pour désigner un nouvel objet ou une nouvelle hypothèse.

Une *expression* est un objet ou énoncé formé en assemblant des objets connus ou en appliquant des fonctions ou lemmes connus. Par exemple si `x` et `y` sont des nombres réels du contexte courant alors `x`, `x+y`, `cos(x)` et `mul_comm x y` sont des expressions (ici `mul_comm` est un lemme affirmant la commutativité de la multiplication, de sorte que `mul_comm x y` est une démonstration de `x * y = y * x`). On notera que de nombreuses commandes incluent une partie `appliqué à` optionnelle qui permet d'appliquer un énoncé (ou une fonction). Cette syntaxe aide à rédiger sur papier mais n'est jamais indispensable. Ainsi on pourra démontrer `x*y = y*x` en utilisant au choix `On conclut par mul_comm appliqué à [x, y]` ou bien `On conclut par mul_comm x y`.

Une *expression annotée* est une suite de symboles de la forme `expr : T` où `T` est soit un type d'objet (entier naturel, nombre réel, fonction...) soit un énoncé. L'annotation annonce le type de l'objet ou de l'énoncé décrit par l'expression, essentiellement pour rendre la démonstration plus lisible. On peut ajouter des annotations dans une phase intermédiaire entre la première démonstration complète avec Lean et la rédaction sur papier.

Un *nom borné* est une suite de symbole de la forme `nom R e` où `R` est un symbole d'inégalité ou d'appartenance et `e` est une expression décrivant un nombre (dans le cas des inégalités) ou un ensemble (dans le cas de l'appartenance).

Une *liste* d'expressions est une collection d'expressions entourée de crochets et sont les éléments sont séparés par des virgules, par exemple `[a, ε > 0, hyp]`.

Une *juxtaposition* d'éléments de syntaxe (expressions, expressions annotées etc.) est une collection d'éléments séparés par des espaces. S'il y a plusieurs éléments, chaque élément composés de plusieurs mots doit être entouré de parenthèses, par exemple `n (ε : ℝ) (ε_pos : ε > 0) k`.

Égalité (=)

Démonstration

Pour démontrer une égalité `a = b`, on peut remplacer des morceaux du but en utilisant des égalités déjà connues (voir la partie utilisation de l'égalité plus bas) jusqu'à ce que le but devienne de la forme `a = a`, ce qui termine automatiquement la démonstration.

On peut utiliser `On calcule` pour démontrer des égalités qui découlent des propriétés usuelles des opérations (commutativité, associativité, distributivité etc.) sans utiliser d'hypothèse.

On peut mener un calcul en utilisant `calc`. Le calcul doit alors commencer par `a` et se finir par `b` et toutes les étapes doivent être des égalités. Chaque étape est justifiée par `: by demo` où `demo` est une commande ou une suite de commandes entourée d'accolades qui démontre cette étape. Il n'y a pas de virgule à la fin des lignes intermédiaires, seulement à la fin du calcul entier.

Si l'égalité but découle par combinaison linéaire à coefficients constants d'égalités et qu'on dispose d'expressions prouvant ces égalités, on utilise `On combine l` où `l` est la liste de ces expressions. Par exemple, si on a des hypothèses h_1 `: a = b + c` et h_2 `: b - a = c` alors on peut démontrer `c = 0` avec `On combine [h₁, h₂]`. Cette commande utilise aussi automatiquement la transitivité de l'égalité.

Les méthodes de démonstration des deux paragraphes précédents s'appliquent aussi aux démonstrations d'inégalités. La commande `calc` utilise automatiquement les diverses règles de transitivité (par exemple si $a \leq b$ et $b < c$ alors $a < c$).

Utilisation

Pour utiliser une expression `h : a = b` on peut remplacer `a` par `b` dans le but courant via `On réécrit via h`. Pour remplacer de la droite vers la gauche c'est-à-dire remplacer `b` par `a`, on utilise `On réécrit via ← h`. On peut effectuer plusieurs remplacements dans la même commande en fournissant une liste d'expressions, précédées de `←` le cas échéant. Par exemple `On réécrit via [h, ← mul_assoc]`.

Si on veut effectuer le remplacement dans une hypothèse nommée `hyp` du contexte courant, on utilise `On réécrit via h dans hyp`. Pour augmenter la lisibilité, on peut écrire `On réécrit via h dans hyp qui devient e`, où `e` est l'expression obtenue après remplacement.

Plus généralement, la commande `On réécrit via` peut utiliser comme argument une expression contenant une égalité cachée derrière un quantificateur universel. Dans ce cas, Lean essaie de spécialiser l'énoncé quantifié avant de réécrire. Par exemple, étant donnés `f : ℝ → ℝ` et des hypothèses `hf : ∀ x, f (-x) = f x` et `h : f(-1) = 0`, la commande `On réécrit via hf dans h` va transformer `h` en `f 1 = 0`.

L'égalité utilisée peut aussi apparaître en conclusion d'une implication. Lean va alors créer un but correspondant à la prémisse. Par exemple, étant donnés `f : ℝ → ℝ`, `x : ℝ` et une hypothèse `hf : x > 0 → f x = 0`, la commande `On réécrit via hf` va remplacer `f x` par `0` dans le but mais en créant un nouveau but `⊢ x > 0`.

Équivalence (↔)

Démonstration

Pour démontrer une équivalence `P ↔ Q`, on peut remplacer des morceaux du but en utilisant des égalités ou équivalences déjà connues (voir la partie utilisation de l'égalité plus haut et celle de l'équivalence plus bas) jusqu'à ce que le but devienne de la forme `P ↔ P`, ce qui termine automatiquement la démonstration.

On peut annoncer la démonstration de l'implication de la gauche vers la droite par `Montrons que P → Q`. Lean crée alors automatiquement un autre but pour `Q → P`. Lorsque le premier but est démontré, on peut annoncer `Montrons que Q → P` pour plus de clarté mais ce n'est pas nécessaire.

On peut mener un calcul en utilisant `calc`. Le calcul doit alors commencer par `P` et se finir par `Q` et toutes les étapes doivent être des équivalences. Chaque étape est justifiée par `: by demo` où `demo` est une commande ou une suite de commandes entourée d'accolades qui démontre cette étape. Il n'y a pas de virgule à la fin des lignes intermédiaires, seulement à la fin du calcul entier.

Utilisation

Pour utiliser une expression `h : P ↔ Q` on peut remplacer `P` par `Q` dans le but courant via `On réécrit via h`. Pour remplacer `Q` par `P`, on utilise `On réécrit via ← h`. On peut effectuer plusieurs remplacements dans la même commande en fournissant une liste d'expressions, précédées de `←` le cas échéant. Par exemple `On réécrit via [h, ← h']`.

Si on veut effectuer le remplacement dans une hypothèse nommée `hyp` du contexte courant, on utilise `On réécrit via h dans hyp`. Pour augmenter la lisibilité, on peut écrire `On réécrit via h dans hyp qui devient e`, où `e` est l'expression obtenue après remplacement.

Comme dans le cas des égalités, l'équivalence utilisée par `On réécrit via` peut être cachée derrière un quantificateur universel ou en conclusion d'une implication, mais ces cas de figure sont moins courants avec les équivalences.

Implique (→)

Démonstration

Lorsque que le but courant est de la forme `P → Q` pour des énoncés `P` et `Q`, on utilise `Supposons nv_nom` pour introduire l'hypothèse `nv_nom` que `P` est vrai. On peut annoter `nv_nom` , ce qui donne `Supposons nv_nom : P`.

Si `Q` est lui aussi une implication on peut combiner plusieurs introductions en juxtaposant des noms. Si ces noms sont annotés il faut les entourer de parenthèses.

Exemples : `Supposons h`, `Supposons h : n ≥ n₀`, `Supposons (hn : n ≥ n₀) (hn₀ : n₀ > 0)`.

Utilisation

Étant donnée une expression `ePQ : P → Q`, si le but courant est `Q`, on utilise `On applique ePQ` pour transformer le but en `P`.

Pour plus de clarté, on peut écrire plutôt `Par ePQ il suffit de montrer que P`. Il est commode d'écrire une première version de la démonstration avec `On applique` puis de peaufiner la démonstration en passant à cette deuxième forme avant de rédiger sur papier.

Si, en plus de `ePQ`, on dispose déjà d'une expression `eP` qui démontre `P`, on peut achever directement le but courant par `On conclut par ePQ appliqué à eP`.

Et (∧)

Démonstration

Pour démontrer un but de la forme `P ∧ Q`, on annonce la démonstration de `P` par `Montrons que P`. Lean crée alors automatiquement un autre but pour `Q`. Lorsque le premier but est démontré, on peut annoncer `Montrons Q` pour plus de clarté mais ce n'est pas nécessaire.

Utilisation

Étant donnée une expression `e` démontrant `P ∧ Q`, on peut créer deux hypothèses `hP : P` et `hQ : Q` dans le contexte local en écrivant `Par e on obtient hP hQ`. On peut annoter chaque morceau par souci de lisibilité : `Par (e : P ∧ Q) on obtient (hP : P) (hQ : Q)`.

On peut aussi accéder aux deux côtés par les expressions `e.1` et `e.2`. Cela est particulièrement commode lorsque `e` est simple (par exemple s'il s'agit simplement du nom d'une hypothèse) et qu'on a peu d'utilisations prévues.

Ou (∨)

Démonstration

Pour démontrer (directement) un but de la forme `P ∨ Q`, on annonce au choix que l'on va démontrer `P` par `Montrons que P` ou bien `Q` par `Montrons que Q`.

Utilisation

Étant donnée une expression `e` démontrant `P ∨ Q`, on peut remplacer le but courant `R` par deux branches de la démonstration, l'une expliquant pourquoi `P` implique `R` et l'autre expliquant pourquoi `Q` implique `R`. On utilise pour cela `On discute en utilisant e`. On peut annoter `e` pour plus de lisibilité.

Pour tout (∀)

Démonstration

Lorsque que le but courant est de la forme `∀ x : X, P x` pour un prédicat `P` sur un type `X` d'objet, on utilise la commande `Soit`. Chaque argument de `Soit` est, au choix, un nouveau nom, un nouveau nom annoté ou un nouveau nom borné si `P` s'y prête. Dans le dernier cas, la commande introduit à la fois le nouvel objet et une hypothèse d'inégalité ou d'appartenance nommée automatiquement.

On peut fournir plusieurs arguments juxtaposés pour effectuer plusieurs introductions sur la même ligne. Si ces arguments sont des noms annotés ou des noms bornés, il faut les entourer de parenthèses pour éviter les ambigüités.

Exemples : `Soit n`, `Soit n : ℕ`, `Soit ε > 0`, `Soit n k`, `Soit (n ≥ N) (x : ℝ)`,

Utilisation

Étant donnée une hypothèse de la forme `hyp : ∀ x : X, P x` pour un prédicat `P` sur `X`, on peut spécialiser `hyp` à une expression *expr* de type `X` par `On applique hyp à x` où `x` est une expression (ou une liste d'expressions si `P x` commence par un quantificateur universel ou une implication).

Si on veut aussi garder `hyp` pour plus tard, on peut utiliser `Par hyp appliqué à x on obtient nv_nom` (où `nv_nom` peut être annoté).

Si le but courant est de la forme `P x` alors on peut utiliser `On applique hyp` pour achever la démonstration de ce but. De façon plus explicite, on peut écrire `On conclut par hyp appliqué à x`.

Les explications concernant l'utilisation d'une égalité ou d'une équivalence contiennent aussi des références à l'utilisation d'un énoncé quantifié universellement dans un remplacement.

Il existe (∃)

Démonstration

Lorsque que le but courant est de la forme `∃ x : X, P x` pour un prédicat `P` sur un type `X` d'objet, on fournit un témoin, disons x_0, par `Montrons que x₀ convient` ou bien `Montrons que x₀ convient : P x₀`. La deuxième forme est principalement destinée à rendre la démonstration plus lisible mais peut aussi servir à reformuler un peu le but `P x₀` restant.

Utilisation

Étant donnée une expression `e` de la forme `∃ x : X, P x` pour un prédicat `P` sur un type `X` d'objet, on extrait un témoin `x` par `Par e on obtient x tel que nv_nom .`

Si `e` n'est pas directement de la forme `∃ x : X, P x` mais le devient après avoir été appliquée à une autre expression `e'` on écrit `Par e appliqué à e' on obtient x tel que nv_nom .` Si `e` doit être appliquée à plusieurs arguments alors il faut fournir la liste des arguments.

Dans tous les cas, on peut annoter le nouveau nom fourni.

Si l'énoncé `P x` est une conjonction (un « et ») on peut donner plusieurs arguments juxtaposés après `tel que`, entourés de parenthèses s'ils sont annotés. Cela s'applique même lorsque cette conjonction est cachée derrière un quantificateur borné, comme par exemple `∃ δ > 0, Q δ` qui est l'abréviation de `∃ δ : ℝ, δ > 0 ∧ Q δ` qu'on pourra utiliser via `Par hyp on obtient δ tel que (δ_pos : δ > 0) (hQδ : Q δ).`

Exemples :

```
Par h on obtient k tel que n = 2*k,
Par h on obtient k tel que (hk : k > 0) (hnk : n = 2*k),
Par hf appliqué à [ε, ε_pos] on obtient δ tel que (δ_pos : δ > 0) (H : ∀ x, |x - x₀| < δ → |f x - f x₀|
< ε)
```

Non (¬)

Démonstration

Lorsque que le but courant est de la forme `¬ P` pour un énoncé `P`, on utilise `Supposons nv_nom` pour introduire l'hypothèse `nv_nom` que `P` est vrai. Le but devient alors `false` puisque `¬ P` est simplement une notation pour `P → false`. On peut annoter `nv_nom` , ce qui donne `Supposons nv_nom : P`.

Alternativement, on peut utiliser la commande `On pousse la négation` pour pousser la négation vers la droite dans l'expression du but, en utilisant les règles de négation des opérateurs logiques, quantificateurs et opérateurs de comparaison. Par exemple la commande `On pousse la négation` transforme le but `¬ ∀ x, x ≤ 1` en `∃ x, 1 < x`.

Utilisation

Étant donnée une expression `e` de la forme `¬ P` pour un énoncé `P`, si le but courant est `false`, on utilise `On applique e` pour transformer le but en `P`. Pour plus de clarté, on peut écrire plutôt `Par e il suffit de montrer que P`. Il est commode d'écrire une première version de la démonstration avec `On applique` puis de peaufiner la démonstration en passant à cette deuxième forme avant de rédiger sur papier.

Si, en plus de `e`, on dispose déjà d'une expression `eP` qui démontre `P`, on peut montrer `false` directement par `On conclut par e appliqué à eP`.

Si `e` est le nom d'une hypothèse locale, on peut aussi utiliser `On pousse la négation dans e` pour pousser les négations vers la droite dans l'hypothèse `e`, en utilisant les règles de négation des opérateurs logiques, quantificateurs et opérateurs de comparaison.

Si le but courant n'est pas `false` mais que les hypothèses courantes sont incohérentes, par exemple si elles contiennent `hP : P` et `hnP : ¬ P`, on peut transformer le but courant en `false` par `Montrons une contradiction`.

La commande `On conclut`

La commande `On conclut` sert toujours à terminer la démonstration du but courant. Elle prend en argument une expression, éventuellement suivie de `appliqué à` suivi d'un argument ou d'une liste d'arguments, si l'expression de départ est quantifiée universellement ou bien est une implication.

Dans le cas le plus simple, l'expression fournie est directement une démonstration du but. Par exemple, on peut démontrer `x*y = y*x` en utilisant au choix `On conclut par mul_comm appliqué à [x, y]` ou bien `On conclut par mul_comm x y`.

On peut aussi ne pas spécifier les arguments et Lean essaiera de les deviner en examinant le but. Dans l'exemple précédent, on peut ainsi utiliser `On conclut par mul_comm`.

Enfin cette commande s'occupe aussi d'aplanir d'infimes variations entre l'expression fournie et le but lorsque celui-ci est une égalité ou une inégalité. Par exemple si on a une hypothèse `h : ε > 0`, la commande `On conclut par h` permet d'achever la démonstration de `ε > 0` mais aussi de `ε ≥ 0`, de `ε/2 > 0`, ou même de `ε ≥ -1`.

Faits intermédiaires

La commande `Fait nom : énoncé` permet de créer un nouveau but `énoncé`. Après démonstration de ce but, l'énoncé devient une hypothèse nommée `nom`.

Raisonnement par l'absurde et contraposition

Quel que soit le but courant `P`, on peut toujours initier un raisonnement par l'absurde en écrivant `Supposons par l'absurde nv_nom` qui introduit une nouvelle hypothèse `nv_nom : ¬ P` et transforme le but en `false`. Pour plus de lisibilité, on peut annoter `nv_nom` et écrire `Supposons par l'absurde nv_nom : ¬ P`.

Lorsque le but courant est une implication `P → Q`, on peut raisonner par contraposition en écrivant `On contrapose` qui transforme le but en `¬ Q → ¬ P` puis essaie de pousser les négations vers la droite. Par exemple, la commande `On contrapose` transforme le but `(∀ ε > 0, x ≤ ε) → x ≤ 0` en `0 < x → ∃ ε, ε > 0 ∧ ε < x`.

Si on souhaite simplement contraposer sans pousser les négations, il faut utiliser la commande `On contrapose simplement`. Dans l'exemple précédent, le but devient `¬ x ≤ 0 → ¬ ∀ ε > 0, x ≤ ε`.

Disjonctions de cas

Quel que soit le but `P` à démontrer, pour tout expression `e` formant un énoncé, on peut initier une disjonction de cas suivant que cet énoncé est vrai ou faux par `On discute selon que e`. La démonstration se scinde alors en deux branches, l'une de but `e → P` et l'autre de but `¬ e → P`.

Raisonnement par récurrence

Étant donné un prédicat `P` sur les entiers naturels, on peut démontrer `∀ n, P n` par récurrence via la commande `Montrons par récurrence nv_nom : ∀ n, P n`. Cette commande crée deux branches dans la démonstration, une branche dont le but est `P 0` et l'autre dont le but est `∀ n, P n → P (n+1)`. Si le but initial était de la forme `P k` pour un entier `k` alors il est automatiquement démontré après ces deux branches. Sinon la démonstration du but principal reprend avec en plus une hypothèse `nv_nom : ∀ n, P n`.

Reformulations, renommages et oubli

À tout moment, il est possible d'annoncer que le but courant est un énoncé `P` par la commande `Montrons que P`. Cela peut être fait pour aider le lecteur alors que `P` est exactement le but affiché par Lean. Mais c'est aussi l'occasion de reformuler le but est un but qui lui est équivalent par définition. Par exemple, si le but affiché est `pair n`, on peut écrire `Montrons que ∃ k, n = 2*k` qui change l'affichage du but en `∃ k, n = 2*k`, même si cela ne fait pas progresser la démonstration. L'opération analogue sur une hypothèse nommée `hyp` se fait par `On reformule hyp en e` où `e` est une expression équivalente par définition à l'énoncé affirmé par `hyp`.

On peut aussi utiliser `On déplie nom` pour déplier une définition dans le but ou dans une hypothèse. Par exemple, la commande `On déplie pair` transforme le but `pair n` en `∃ k, n = 2*k`. Pour effectuer cette opération dans une hypothèse `hyp`, on utilise `On déplie pair dans h`. On peut alors préciser ce que devient `h` pour augmenter la lisibilité. Par exemple `On déplie pair dans h qui devient ∃ k, n = 2*k`.

Ces reformulations ne sont presque jamais nécessaires pour Lean. Une source d'exception est que la commande `On réécrit via` ne voit pas à travers les définitions. De façon exceptionnelle, il est donc parfois utile d'utiliser une reformulation juste avant un remplacement. La même remarque s'applique à la commande `On pousse la négation`.

On peut aussi changer le nom d'une variable muette `x` en `y` dans le but par `On renomme x en y` (à condition que le nom de variable `y` soit disponible). Par exemple, `On renomme x en y` change le but `∀ x, f x ≤ g x` en `∀ y, f y ≤ g y`. Pour effectuer la même opération dans une hypothèse `hyp` on utilise `On renomme x en y dans hyp`.

Dans une démonstration longue, il est parfois commode de demander à Lean d'oublier une hypothèse `hyp` (ou plus rarement un objet) qui a déjà servi. Pour cela on utilise `On oublie hyp`.