証明支援系 LEAN に入門しよう

梅崎直也@unaoya

2024 年 10 月 20 日 第 6 回すうがく徒のつどい

目次

資料について

今回使用するソースコード、スライドは以下の GitHub リポジトリにあります。
https://github.com/unaoya/Tsudoi6

X のアカウント@unaoya 固定ポストにリンクがあります。
https://x.com/unaoya

はじめてのLEAN

はじめに

集合のことは忘れてください

型と項

項 a が型 A を持つことを a : A と表す。項は必ず型をもつ。

はじめての証明

$1 = 1$ を証明する。

FirstTheorem.lean

```
def my_first_thm : 1 = 1 := Eq.refl 1
```

項を定義する

`n` という `Nat` 型の項を `1` という項により定義する。

FirstTheorem.lean

```
def n : Nat := 1
```

項を定義する

`n` という `Nat` 型の項を `1` という項により定義する。

FirstTheorem.lean

```
def n : Nat := 1
```

`my_first_thm` という `1=1` 型の項を `Eq.refl 1` という項により定義する。

FirstTheorem.lean

```
def my_first_thm : 1 = 1 := Eq.refl 1
```

関数型

項 `f` が型 `A` から型 `B` への関数型を持つことを、`f : A -> B` と書く。

FirstTheorem.lean

```
def twice : Nat -> Nat :=
  fun n => n + n
```

関数適用

関数型の項 `f : A -> B` と項 `a : A` に対して、`f a` は型 `B` の項である。この記法を利用して関数型の項を定義することもできる。

関数適用

関数型の項 `f : A -> B` と項 `a : A` に対して、`f a` は型 `B` の項である。この記法を利用して関数型の項を定義することもできる。

FirstTheorem.lean

```
def twice (n : Nat) : Nat := n + n
def n : Nat := twice 3
def my_first_thm : 1 = 1 := Eq.refl 1
```

改めてはじめての証明

FirstTheorem.lean

```
def my_first_thm : 1 = 1 := Eq.refl 1
```

改めてはじめての証明

FirstTheorem.lean

```
def my_first_thm : 1 = 1 := Eq.refl 1
```

my_first_thm という項は 1=1 という型を持ち、項 Eq.refl 1 により定義される。Eq.refl 1 という Nat 型の項は Eq.refl という Nat -> Nat 型の項に Nat 型の項 1 を適用したものである。

証明とプログラムの関係

色々な型

自然数の型 `Nat` が真偽値の型 `Bool` などが標準で用意されている。型 `A`, `B` から関数の型 `A -> B` が作れる。また、直和型 `A x B` や直積型 `A + B` を作ることもできる。

色々な型

自然数の型 `Nat` が真偽値の型 `Bool` などが標準で用意されている。型 `A`, `B` から関数の型 `A -> B` が作れる。また、直和型 `A x B` や直積型 `A + B` を作ることもできる。

型の型 `Type` がある。（型全体の型ではない。）型も項になる。 `0 : Nat` であり `Nat : Type` である。

色々な型

自然数の型 `Nat` が真偽値の型 `Bool` などが標準で用意されている。型 `A`, `B` から関数の型 `A -> B` が作れる。また、直和型 `A x B` や直積型 `A + B` を作ることもできる。

型の型 `Type` がある。（型全体の型ではない。）型も項になる。 `0 : Nat` であり `Nat : Type` である。

重要な型に `Prop` がある。これは命題全体の型で、 `P : Prop` は「*P* が命題である」と解釈する。同時に P は型でもあり、 `h : P` を「*h* は命題 *P* の証明である」と解釈する。

色々な型

自然数の型 `Nat` が真偽値の型 `Bool` などが標準で用意されている。型 `A`, `B` から関数の型 `A -> B` が作れる。また、直和型 `A x B` や直積型 `A + B` を作ることもできる。

型の型 `Type` がある。（型全体の型ではない。）型も項になる。 `0 : Nat` であり `Nat : Type` である。

重要な型に `Prop` がある。これは命題全体の型で、 `P : Prop` は「*P* が命題である」と解釈する。同時に P は型でもあり、 `h : P` を「*h* は命題 *P* の証明である」と解釈する。

`1=1 : Prop` であり、 `Eq.refl 1 : 1=1` である。これを「Eq.refl 1 が 1=1 の証明である」と解釈し、 `Eq.refl` は等式の公理に対応する関数と考える。

証明とプログラム

命題 p に対応する型 P があり、 P 型の項 h を「h が P の証明である」と解釈する。

「命題 P を証明する」ことは、プログラム上では P 型の項を作ることであると解釈する。「命題 P を仮定する」ことは、P 型の項が与えられていることであると解釈する。

証明とプログラム

命題 p に対応する型 P があり、 P 型の項 h を「h が P の証明である」と解釈する。

「命題 P を証明する」ことは、プログラム上では P 型の項を作ることであると解釈する。「命題 P を仮定する」ことは、P 型の項が与えられていることであると解釈する。

証明とはおおよそ主張と理由の列と思える。理由となるのは論理規則または公理や定義またはすでに証明した命題。

プログラムとはおおよそ関数の列と思える。

プログラムの例

ある商店ではりんご、みかん、いちごの三種類の果物を売っている。1 個あたりの値段はりんご 100 円、みかん 50 円、いちご 200 円である。

りんごの個数とみかんの個数といちごの個数を引数に取り、合計金額を返すプログラムを作ろう。ただし、レジ袋（1 枚だけ）が必要な場合は合計金額に 10 円を加える。

コーディング

SecondTheorems.lean

```
def total (a b c : Nat)
    (bag : Bool) : Nat :=
  a * 100 + b * 50 +
    c * 200 + if bag then 10 else 0
```

`+ : Nat -> Nat -> Nat` や `* : Nat -> Nat -> Nat`、
`if then else : Bool -> Nat` を合成し、

項 `total : Nat -> Nat -> Nat -> Bool -> Nat` を作った。

関数とならば

命題 P, Q に対応する型（`Prop` 型の項）`P`, `Q` に対し、関数型の項 `f : P -> Q` は P の証明 `h : P` を与えると Q の証明 `f h : Q` を返す関数と解釈する。

プログラムは関数の組み合わせであり、証明はならばの組み合わせである。

ただし、通常のプログラムにおける関数は実際の値の対応関係が重要になる一方で、証明は項の対応関係は気にしないで型だけを気にしている。証明は区別せず、項があるかないかが重要。

定理 A とその証明

Theorem
a, b, c を自然数とし、$a = b$ と $b = c$ ならば $a = c$ である。

Proof.
等式の推移律により、$a = b$ と $b = c$ から $a = c$ を導くことができる。 □

定理 A の証明のコーディング

SecondTheorems.lean

```
theorem A (a b c : Nat)
    (h0 : a = b) (h1 : b = c) :
    a = c :=
  Eq.trans h0 h1
```

定理 A の証明のコーディング

SecondTheorems.lean

```
theorem A (a b c : Nat)
    (h0 : a = b) (h1 : b = c) :
    a = c :=
  Eq.trans h0 h1
```

関数 `Eq.trans` : `a = b -> b = c -> a = c` は `a=b` 型の項と `b=c` 型の項を受け取り `a=c` 型の項を返す関数である。

これは命題 $a = b$ の証明と命題 $b = c$ の証明から命題 $a = c$ の証明を与える、つまり等号の推移律の証明と解釈できる。

定理 B とその証明

Theorem
a, b を自然数とする。$(a+1) \times b = a \times b + b$ が成り立つ。

Proof.
分配法則より $(a+1) \times b = a \times b + 1 \times b$ である。

掛け算の定義より $1 \times b = b$ である。

関数と等号の定義より $a \times b + 1 \times b = a \times b + b$ である。

等号の推移律より $(a+1) \times b = a \times b + b$ である。 $\square$

定理 B の証明のコーディング

SecondTheorems.lean

```
theorem B (a b : Nat)
    : (a + 1) * b = a * b + b :=
  Eq.trans (Nat.right_distrib a 1 b)
    (congrArg
      (fun x => a * b + x)
      (Nat.one_mul b))
```

定理 C とその証明

Theorem
a を自然数とする。$(a+1)\times(b+1)=a\times b+a+b+1$ が成り立つ。

Proof.
定理 B を使うと、$(a+1)\times(b+1)=a\times(b+1)+(b+1)$ である。$a\times(b+1)=a\times b+a$ である。 $\square$

定理 C の証明のコーディング

SecondTheorems.lean

```
theorem C (a b : Nat) :
(a + 1) * (b + 1) = a * b + a + b + 1 :=
  Eq.trans (Eq.trans (B a (b + 1))
    (congrArg (fun x => x + (b + 1))
      (Eq.trans (Nat.left_distrib a b 1)
        (congrArg (fun x => a * b + x)
          (Nat.mul_one a)))))
  (Eq.symm
    (Nat.add_assoc (a * b + a) b 1))
```

ここまでの整理

P を仮定すると Q が成り立つという定理の証明は、P ならば P_1、P_1 ならば P_2、. . .、P_{n-1} ならば Q という命題とその証明の列で与えられる。

P から Q への関数を作るには、項 h1 : P -> P1, h2 : P1 -> P2, . . ., hn : P(n-1) -> Q を合成すればよい。

プログラムでは項の対応が重要である一方、定理の証明では型さえ合っていれば項の具体的な対応は気にしない。

述語論理と依存型

直積/直和とかつ/または

命題 P, Q から命題 P かつ Q、P または Q を作ることができる。これに対応して、型 P , Q から直積型 P x Q、直和型 P + Q を作ることができる。

直積/直和とかつ/または

命題 P, Q から命題 P かつ Q、P または Q を作ることができる。これに対応して、型 P , Q から直積型 P x Q、直和型 P + Q を作ることができる。

関数 $R \to P \times Q$ を作ることは関数 $R \to P$ と関数 $R \to Q$ を作ることと同じ。これは R ならば（P かつ Q）の証明は、R ならば P の証明と R ならば Q の証明の組であることに対応する。

直積/直和とかつ/または

命題 P, Q から命題 P かつ Q、P または Q を作ることができる。これに対応して、型 P , Q から直積型 P x Q、直和型 P + Q を作ることができる。

関数 $R \to P \times Q$ を作ることは関数 $R \to P$ と関数 $R \to Q$ を作ることと同じ。これは R ならば（P かつ Q）の証明は、R ならば P の証明と R ならば Q の証明の組であることに対応する。

関数 $P + Q \to R$ を作ることと関数 $P \to R$ と関数 $Q \to R$ を作ることは同じ。これは（P または Q）ならば R の証明は、P ならば R の証明と Q ならば R の証明の組であることに対応する。

依存型

型の族（型でパラメータ付けられた型、型 A から型の型 Type への関数）を考えることができる。例えば、自然数 n に対して n 次元実数ベクトルの型 $\mathbb{R}^n$ を考える。型の族から、族の直積型と直和型を考えることができる。これが依存積型と依存和型である。

依存型

型の族（型でパラメータ付けられた型、型 `A` から型の型 `Type` への関数）を考えることができる。例えば、自然数 n に対して n 次元実数ベクトルの型 $\mathbb{R}^n$ を考える。型の族から、族の直積型と直和型を考えることができる。これが依存積型と依存和型である。

命題は型であった。命題の族は述語 $P(n)$ だと思える。例えば、自然数 n に対して $0 < n$ であるという命題 $P(n)$ を考える。 `P` は `Nat -> Prop` という型を持つ。

依存型

型の族（型でパラメータ付けられた型、型 `A` から型の型 `Type` への関数）を考えることができる。例えば、自然数 n に対して n 次元実数ベクトルの型 $\mathbb{R}^n$ を考える。型の族から、族の直積型と直和型を考えることができる。これが依存積型と依存和型である。

命題は型であった。命題の族は述語 $P(n)$ だと思える。例えば、自然数 n に対して $0 < n$ であるという命題 $P(n)$ を考える。 `P` は `Nat -> Prop` という型を持つ。

先ほどまでに出てきた `a`, `b` : `Nat` に対する `a=b` という型も、パラメータ a, b に依存する型の族である。

述語論理と依存型

この型の族 P の直積型の項 h は何か？ すべての n に対して h n という P n 型の項を並べたものと思うことができる。P n 型の項 h n は命題 $P(n)$ の証明であるから、h はすべての n に対する命題 $P(n)$ の証明を並べたもの、つまり、$\forall n, P(n)$ の証明である。

述語論理と依存型

この型の族 `P` の直積型の項 `h` は何か？ すべての `n` に対して `h n` という `P n` 型の項を並べたものと思うことができる。`P n` 型の項 `h n` は命題 $P(n)$ の証明であるから、`h` はすべての n に対する命題 $P(n)$ の証明を並べたもの、つまり、$\forall n, P(n)$ の証明である。

直和型の項は？ ある `n` に対して `h n` という `P n` 型の項を選んだものと思うことができる。`P n` 型の項 `h n` は命題 $P(n)$ の証明であるから、`h` はある n に対する命題 $P(n)$ の証明を選んだもの、つまり、$\exists n, P(n)$ の証明である。

全称命題の証明例

Theorem
任意の自然数 n に対して $0 \leq n$ である。

Proof.
不等号の定義と帰納法よりよい。 □

全称命題の証明例

Theorem
任意の自然数 n に対して $0 \leq n$ である。

Proof.
不等号の定義と帰納法よりよい。 □

族の直積は、族が定数族の場合関数型とみなせる。これの拡張で、依存積は依存関数型とも呼ばれ、関数型の拡張と考えられる。

全称命題の証明例

Theorem
任意の自然数 n に対して $0 \leq n$ である。

Proof.
不等号の定義と帰納法よりよい。 □

族の直積は、族が定数族の場合関数型とみなせる。これの拡張で、依存積は依存関数型とも呼ばれ、関数型の拡張と考えられる。

全称命題の証明は対応する依存積型（依存関数型）の項を作ればよく、それには関数を定義すればよい。ただし、値の型が引数によって変わるところがポイント。

全称命題のコーディング例

Quantifier.lean

```
theorem zero_is_min :
    ∀ n : Nat, 0 ≤ n :=
  fun n ↦ Nat.zero_le n
```

Quantifier.lean

```
theorem zero_is_min (n : Nat):
    ∀ n : Nat, 0 ≤ n :=
  Nat.zero_le n
```

存在命題の証明例

Theorem
1 より大きな自然数が存在する。

つまり $\exists m, 1 < m$ を証明する。

Problem
$m = 2$ とすると、$1 < 2$ なのでこれが条件を満たす。

存在命題のコーディング例

Quantifier.lean

```
theorem one_not_max :
    ∃ m, 1 < m :=
  <2, Nat.one_lt_two>
```

項 one_not_max は依存和型を持つ。これは、 m : Nat で添字づけられた型 n<m に対する依存和型。すなわち m を変数とする述語 $P(m) = n < m$ に対する存在命題。

量化子の組み合わせ

量化子を組み合わせることもできる。依存和や依存積を繰り返すこともできる。

Quantifier.lean

```
theorem no_max :
    ∀ n : Nat, ∃ m : Nat, n < m := 
  fun n =>
    <(n + 1), (Nat.lt_succ_self n)>
```

ここまでのまとめ

パラメータ付けられた型の族（`f : A -> Type`）を考えることができる。これに対する族の直積型と直和型を考えることができる。

特にパラメータ付けられた命題の族（`P : A -> Prop`）を考えることができる。これに対しても族の直積型と直和型を考えることができる。これらが全称命題 $\forall a : A, fa$ と存在命題 $\exists a : A, Pa$ に対応する。

証明を支援する仕組み

証明の記述

- 引数の省略やドット記法
- calc モード
- タクティクによる証明。
- ライブラリを使う。

calc モード

等式や不等式の変形や命題の同値変形、より一般に推移的な関係による変形を書くためのモード。

Assist.lean

```
theorem A (a b c : Nat) (h0 : a = b)
    (h1 : b = c) : a = c :=
  calc
    a = b := h0
    _ = c := h1
```

タクティク

これまでは、証明したい命題に対応する型の項を直接記述した。タクティクという仕組みを用いることで、直接項を書くより簡単に証明が書ける。また、ごく一部だが、証明の自動化や探索ができる。

タクティクは関数ではない。タクティクは項だけでなく状態を扱う。同じ項でも状態によって返ってくる結果が違う。

タクティクを使った証明 A

先ほどの定理 A, B, C をタクティクを使って証明してみる。

Assist.lean

```
theorem A (a b c : Nat) (h0 : a = b)
    (h1 : b = c) : a = c := by
  simp [h0, h1]
```

タクティクを使った証明 B

Assist.lean

```
theorem B (a b : Nat) :
    (a + 1) * b = a * b + b :=
  calc
    (a + 1) * b = a * b + 1 * b :=
      Nat.right_distrib _ _ _
    _ = a * b + b := by
      simp [Nat.one_mul]
```

タクティクを使った証明C

Assist.lean

```
theorem C (a : Nat) :
    (a + 1) * (b + 1) =
        a * b + a + b + 1 :=
  calc
    (a + 1) * (b + 1) =
        a * (b + 1) + 1 * (b + 1) := by
      apply Nat.right_distrib
    _ = a * b + a + b + 1 := by
      simp [Nat.left_distrib, Nat.add_assoc]
```

ライブラリ

普通のプログラミング言語と同様、LEAN にもさまざまなライブラリが存在する。普通は便利なデータ構造やそれらを扱う関数がライブラリにある。LEAN では数学的構造やそれらに関するの定理もライブラリにある。

mathlib というライブラリが一番大きな lean の数学の定理に関するライブラリ。ライブラリにあるかないかは誰かが書いたかどうかが重要で、数学的な難しさなどはそこまで関係ない。より便利なタクティクもライブラリにある。

ライブラリを使った証明

mathlib には微積分学の基本定理がすでに形式化されてある。

微積分学の基本定理を証明する

目標の定理

$f : \mathbb{R} \to \mathbb{R}$ が連続であるとする。

$$\frac{d}{dx} \int_a^x f(t)dt = f(x)$$

今回はあえてライブラリを一切使わずに証明する。実数については公理だけを使う。（参考、杉浦解析入門）

証明(杉浦解析入門より引用)(1/4)

実数 x に対し、

$$F(x) = \int_a^x f(t)dt$$

とおく。(向きつき積分。f の連続性より、f は可積分)

任意の実数 x, y に対して等式

$$F(x) - F(y) = \int_y^x f(t)dt$$

が成り立ち、また三角不等式

$$\left| \int_y^x f(t)dt \right| \leq \left| \int_y^x |f(t)|dt \right|$$

が任意の x, y に対して成り立つことから、

証明 (2/4)

任意の実数 $h \neq 0$ に対し、

$$\begin{aligned}\left|\frac{1}{h}(F(x+h)-F(x))-f(x)\right| &= \left|\frac{1}{h}\int_x^{x+h} f(t)dt - f(x)\right| \\ &\leq \left|\frac{1}{h}\int_x^{x+h} |f(t)-f(x)|dt\right|\end{aligned}$$

となる。

証明 (3/4)

いま f は x で連続であるから、任意の $\epsilon > 0$ に対して、$\delta > 0$ が存在し、

$$|t - x| < \delta$$

ならば

$$|f(t) - f(x)| < \epsilon$$

となる。

証明 (4/4)

そこで、$0 < |h| < \delta$ のとき上式の右辺は $\leq \epsilon$ となる。これは、

$$\lim_{h \neq 0, h \to 0} \frac{1}{h}(F(x+h) - F(x)) = f(x)$$

を意味する。すなわち F は x で微分可能で、$F'(x) = f(x)$ である。

必要な定義や定理の形式化

- 実数の公理や諸性質。
- 関数の極限、関数の連続性、微分、積分の定義。
- 積分の三角不等式、単調性、定数関数の積分、線形性、区間に関する加法性など。

証明の形式化

証明に対応するコーディングをしよう。

参考資料

- 公式
  https://lean-lang.org/documentation/
- Mathematics in lean
  https://leanprover-community.github.io/mathematics_in_lean/index.html
- Theorem Proving in Lean 4 日本語訳
  https://aconite-ac.github.io/theorem_proving_in_lean4_ja/title_page.html
- 数学系のための Lean 勉強会
  https://haruhisa-enomoto.github.io/lean-math-workshop/