Coq ユーザが Lean を調べてみた

田中　哲

産業技術総合研究所 インテリジェントプラットフォーム研究部門

2024-06-28 Proof Summit 2025

発表の概要

本発表では、依存型の場合分けの基礎と使い方について Coq と Lean を比較しつつ説明します

- 依存型の場合分け
  Coq は match 式、Lean は recursor
- 場合分けの使い方
  単一化問題の生成と解決
  Coq と Lean の公理の扱いの違い

カリー＝ハワード同型対応と CIC

カリー＝ハワード同型対応:

- 命題は型、証明は項 (プログラム) に対応
- 「A ならば B」という命題は「$A \to B$」型と対応
- その証明は「A の証明を受け取って B の証明を返す関数」
- Coq や Lean では、証明を項として記述し、型検査によってその正しさを確認

CIC (Calculus of Inductive Constructions) は帰納型を導入

- Coq や Lean は CIC に基づく
- Coq は CIC にかなり忠実
- Lean は拡張的

CIC における論理の表現

命題論理:

- 真: 引数のないコンストラクタをひとつ持つ帰納型
- 偽: コンストラクタを持たない（値のない）帰納型
- $P \wedge Q$: ひとつのコンストラクタを持つ帰納型
- $P \vee Q$: ふたつのコンストラクタを持つ帰納型
- $P \to Q$: P の証明を受け取って Q の証明を返す関数型
- $\neg P$: 偽（帰納型）を返す関数型

述語論理:

- $\forall x{:}T, P(x)$:（依存）関数型
- $\exists x{:}T, P(x)$: ひとつのコンストラクタを持つ帰納型

多くは帰納型で表現される
また、述語論理で現れる普通の値 $(x{:}T)$ も帰納型で表現されることが多い

帰納型の定義

```
Inductive T1 (a1 : A1) ⋯ (ap : Ap) :
      ∀ (b1 : B1) ⋯ (bn : Bn), sort :=
⋯
| C : ∀ (x1 : X1) ⋯ (xm : Xm), T1 a1 ⋯ ap y1 ⋯ yn
⋯
with T2 (a1 : A1) ⋯ (ap : Ap) : ⋯
⋯
with Tk (a1 : A1) ⋯ (ap : Ap) : ⋯
⋯
```

- `T1 ⋯ Tk` は相互再帰な帰納型（相互再帰でなければ $k = 1$）
- パラメータ `(a1 : A1) ⋯ (ap : Ap)` や
  インデックス `(b1 : B1) ⋯ (bn : Bn)` がある場合、
  型族 (type family) を定義する
  例: list A は具体的な型 A それぞれについて別の型 (集合)
- 無限再帰を禁止する positivity condition という条件がある

Lean では相互再帰な帰納型は単一の帰納型に変換されるようだが、だいたい同じ
今日の発表では相互再帰は扱わず、インデックスは高々ひとつ

帰納型の定義

```
Inductive T (a1 : A1) ⋯ (ap : Ap) :
      ∀ (b1 : B1) ⋯ (bn : Bn), sort :=
⋯
| C : ∀ (x1 : X1) ⋯ (xm : Xm), T a1 ⋯ ap y1 ⋯ yn
⋯
```

- `(a1 : A1) ⋯ (ap : Ap)` はパラメータ
- `(b1 : B1) ⋯ (bn : Bn)` はインデックス
- T は引数としてパラメータとインデックスを受け取る

  `T a1 ⋯ ap b1 ⋯ bn`
- コンストラクタは引数としてパラメータを最初に受け取る
  return type では、パラメータ `a1 ⋯ ap` はそのまま

  `C a1 ⋯ ap x1 ⋯ xm : T a1 ⋯ ap y1 ⋯ yn`
- 定義において `y1 ⋯ yn` はインデックスを式で指定し、その中で引数 `a1 ⋯ ap x1 ⋯ xm` を利用できる

帰納型の例

```
Inductive bool : Set := true : bool | false : bool.
Inductive nat : Set := O : nat | S : nat → nat.
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.
Inductive EvenOddList (A : Type) : bool → Type :=
| nil : EvenOddList A true
| cons (even : bool) :
    A → EvenOddList A even → EvenOddList A (negb even).
Inductive vec (A : Type) : nat → Type :=
| vnil : vec A O
| vcons : A → ∀ (n : nat), vec A n → vec A (S n).
Inductive eq (A : Type) (x : A) : A → Prop :=
  eq_refl : x = x.
Inductive JMeq (A:Type) (x:A) : ∀ B:Type, B → Prop :=
  JMeq_refl : JMeq x x.
```

帰納型の使い方

- コンストラクタで値を生成
- match 式で（コンストラクタ毎に）場合分け
  コンストラクタに渡した引数が得られる
- Coq は再帰関数の停止性を保証するために、再帰するたびに小さくなる（部分項になる）帰納型引数が必要
  (停止性がないと無限再帰で値を返さない項を作れる。値を返さない項は任意の型をつけられるので、なんでも証明できてしまう)

OCaml の match 式

```
match item with
... | C vars ⇒ exp | ...
end
```

- `item` がコンストラクタ `C` で作られた値ならば、match 式の値は `exp` になる
- `exp` の中では `item` を作ったときに `C` に渡した引数が `vars` として使える

Coq の依存マッチ

T はパラメータ無、インデックス 1 個の帰納型

```
match item as u in T v return A with
... | C vars ⇒ exp | ...
end
```

- as-in-return 節が追加されている
- `item` のコンストラクタで分岐が選ばれて、対応する `exp` の値が match 式の値になるのは普通の match 式と同様
- match 式全体の型と分岐 `exp` の型は（同じ値の型なので）同じ型で `A` で示されるが、内部の変数 `u` と `v` は置換される
- match 式全体の型では `u` が `item` に置換され、`v` が `index` に置換される（ただし `item : T index`）
- `exp` の型では `u` が `C vars` に置換され、`v` が `index'` に置換される (ただし `C vars : T index'`)

正直、どう使ったらいいかよく分からない

Coq の再帰

Coq で再帰を表現するには fix 項を使う

```
fix f args {struct arg} := exp
```

- OCaml の let rec f args = exp in f に相当する
- `{struct arg}` は decreasing argument を指定する
- decreasing argument は再帰呼び出しで部分項になっていなければならない
- この制約が termination checker により検査され、停止性が保証される

Lean の依存マッチと再帰

- Lean の match 式には as-in-return 節はない
- Coq のプリミティブは match 式と fix 式だが、Lean は recursor
- 帰納型を定義すると、自動的に recursor という関数が提供される
  Lean 3 の The Lean Language Manual: 4.5. Inductive Families に形式的な定義がある
- recursor はプリミティブであり、定義を展開できるものではない

Lean の帰納型定義と recursor

```
inductive EvenOddList (α : Type u) : Bool → Type u where
  | nil : EvenOddList α true
  | cons {isEven : Bool} : α → EvenOddList α isEven →
          EvenOddList α (not isEven)
```

EvenOddList を定義すると EvenOddList.rec が定義される

```
EvenOddList.rec.{u_1, u} {α : Type u}
  {motive : (a : Bool) → EvenOddList α a → Sort u_1}
  (nil : motive true EvenOddList.nil)
  (cons :
    {isEven : Bool} →
    (a : α) →
    (a_1 : EvenOddList α isEven) → motive isEven a_1 →
    motive (!isEven) (EvenOddList.cons a a_1))
  {a : Bool} (t : EvenOddList α a)
  : motive a t
```

Lean の recursor と Coq の対応

Coq:

```
match nil as x in EvenOddList _ b return bool with
| nil => true
| cons ev h t => false
end
```

Lean:

```
@EvenOddList.rec Nat (fun b x => Bool)
  true
  (fun {ev} h t H => false)
  true EvenOddList.nil
```

- Lean の motive の `fun b x => Bool` が Coq の `as x in EvenOddList _ b return bool` の部分に対応
  (ちなみに CIC では、Lean のように関数になっている)
- `(fun {ev} h t H => false)` の H は再帰の部分で Coq の match にはない

Lean の recursor の形式的定義

Lean 3 の The Lean Language Manual: 4.5. Inductive Families motive が Coq の as-in-return 節に対応し、分岐の型と recursor の結果の型を変えられる

```
inductive foo (a : α) : Π (c : γ), Sort u
| constructor₁ : Π (b : β₁), foo t₁
| constructor₂ : Π (b : β₂), foo t₂
...
| constructorₙ : Π (b : βₙ), foo tₙ
```

the eliminator foo.rec, which takes arguments (a : α) (the parameters) C : Π (c : γ), foo a c → Type u (the motive of the elimination) for each i, the minor premise corresponding to constructor$_i$ (x : foo a) (the major premise) and returns an element of C x. Here, The ith minor premise is a function which takes (b : β_i) (the arguments to the constructor) an argument of type Π (d : δ), C s (b$_j$ d) corresponding to each recursive argument (b$_j$: β_{ij}), where β_{ij} is of the form Π (d : δ), foo s and returns an element of C t$_i$ (constructor$_i$ a b).

Coq での recursor

- Coq でも帰納型の定義で recursor が得られる（ `nat_ind`, `nat_rect` など）
- match/fix を使った定義 (プリミティブではない)

```
Inductive EvenOddList (A : Type) : bool → Type :=
| nil : EvenOddList A true
| cons (ev : bool) : A → EvenOddList A ev
                     → EvenOddList A (negb ev).

EvenOddList_ind =
fun (A : Type) (P : ∀ b : bool, EvenOddList A b → Prop) (f : P
  (f0 : ∀ (ev : bool) (a : A) (e : EvenOddList A ev), P ev e →
fix F (b : bool) (e : EvenOddList A b) {struct e} : P b e :=
  match e as e0 in (EvenOddList _ b0) return (P b0 e0) with
  | nil _ ⇒ f
  | cons _ ev y e0 ⇒ f0 ev y e0 (F ev e0)
  end
     : ∀ (A : Type) (P : ∀ b : bool, EvenOddList A b → Prop),
       P true (nil A) →
       (∀ (ev : bool) (a : A) (e : EvenOddList A ev), P ev e →
       ∀ (b : bool) (e : EvenOddList A b), P b e
```

場合分けと再帰について Coq と Lean の比較

Coq の match の as-in-return 節と fix 項

利点 OCaml と類似、extract で似た形を生成可能、call-by-value での効率 [Pierre-Marie Pédrot, A Kernel of Truth, TPP 2025]

欠点 termination checker が必要

Lean の recursor

利点 termination checker が不要

欠点 recursor を直接使うのは難しく、普通の match や再帰から recursor に変換する equation compiler が必要
素朴に実行すると非効率なので実行するときはいろいろ変形する必要がある

依存マッチの難しさ

- 機構を理解しても、証明でどう使ったらいいか分からない
- 型の一部を置換できますといわれてもどこを置換すればいいのか?
- どうなればうまくいったことになるのか?

match 式 を生成する tactic

Coq:

- case : 単に match 式を生成する
- case_eq : match 対象の等式も生成する
- destruct : コンテキストも巻き込んで置換する
- inversion : インデックスの等式を生成して整理する
- dependent inversion : inversion の等式生成に加えて match 対象も置換する
- dependent destruction : JMeq を使う
- depelim (Equations パッケージ) : JMeq を使わずにがんばる

Lean:

- cases : これだけで済む

Coq には悪戦苦闘の跡が見える

依存マッチでどんな情報が得られるのか

```
match item as u in T v return A with
… | C vars ⇒ exp | …
end
```

item が C で構築されていた場合、以下の等式が成り立つ

- item = C vars
- index = index'
  ただし item : T index であり、C vars : T index' とする

一般に インデックスは 0 個以上存在する
インデックスそれぞれに等式が成り立ち、さらに item の等式が成り立つ
依存マッチは概念的にコンストラクタ毎に連立方程式を生成する

連立方程式の分類

- 変数とコンストラクタのみ: 単一化で解けるかも
- （コンストラクタ以外の）関数を含む: がんばれ

単一化問題と解法

単一化問題: コンストラクタと変数だけからなる項の等式の集合 $\{t_1 = t'_1, t_2 = t'_2, \ldots\}$

単一化子: 適用すると等式の両辺が等しくなるような（変数をなんらかの項に置き換える）代入 $\{x_1 = u_1, x_2 = u_2, \ldots\}$

解法: 以下の変形を繰り返す（上のほうの変形が優先、左右非対称な規則は逆も行う）

- deletion: 同じ項の等式 $t = t$ を取り除く
- injectivity: 先頭が同じコンストラクタの等式は、各要素毎の等式に分解する: $C\ t_1 \ldots t_n = C\ t'_1 \ldots t'_n$ を $t_1 = t'_1, \ldots, t_n = t'_n$ に置き換える
- discrimination: 先頭が異なるコンストラクタの等式があったら、解は存在しないと判定: $C\ \ldots = C'\ \ldots$
- solution: ある等式 $x = t$ で t に x が出現せず、他の等式に x が出現している場合、その出現を t で置き換える: $x = t$ があったとき $P(x) = Q(x)$ を $P(t) = Q(t)$ にする
- occur-check: ある等式 $x = t$ の t の内部に x が出現するなら、解は存在しないと判定: $x = P(x)$

解が存在しないと判定されずに変形できなくなったら、それが解

証明のステップとしての単一化問題の解決（概念）

依存マッチで（コンストラクタ毎に）単一化問題を生成し、それを解く:

- 解が存在しなければ前提がありえないということで証明完了
- 解（代入）が存在するならそれを適用してゴール中の変数を置換する

これによって依存マッチで得られるすべての情報がゴールに反映される

証明のステップとしての単一化問題の解決（現実）

素直にはいかない

- 依存マッチで等式を生成: 両辺で型が異なる等式が発生
- deletion: 公理が必要になることがある
- injectivity: no confusion という補題が必要

依存マッチで等式を生成する

単なる match 式で得られるのは、コンストラクタの引数だけ
以下の S m の分岐では m を使えるが、n = S m という等式（の証明）は得られない

```
match n with
| O ⇒ true
| S m ⇒ false
end
```

等式を生成するには、等式を受け取る関数を分岐に記述して、等式を引数として即座に呼び出す
その際に等式の片方だけを置換する

```
match n as n' return n = n' → bool with
| O ⇒ fun (H : n = O) ⇒ true
| S m ⇒ fun (H : n = S m) ⇒ false
end eq_refl
```

注: eq_refl は eq 型のコンストラクタで

```
@eq_refl nat n : n = n
```

インデックスがある場合、型が異なる等式が発生

```
match item as u in T v return A with
⋯ | C vars ⇒ exp | ⋯
end
```

item が C で構築されていた場合、以下の等式が得られる

- item = C vars
- index = index'
  ただし item : T index であり、C vars : T index' とする

ここで、item と C vars の型はインデックスが異なる
そのため item = C vars という等式は eq 型では表現できない

eq 型

x = y という式は eq x y の notation
(型引数も省略しないで書くと @eq A x y)

```
Inductive eq (A : Type) (x : A) : A → Prop :=
  eq_refl : x = x.
```

x と y はどちらも A 型でなければならない
そのため、item : T index と C vars : T index' という異なる型の項は eq 型の等式にはできない

型が異なる等式をどう表現するか

- JMeq を使う (Lean では HEq)
- 型を置換して合わせる
- sigma 型の等式を使う

型が異なる等式に JMeq を使う

```
Inductive JMeq (A:Type) (x:A) : ∀ B:Type, B → Prop :=
  JMeq_refl : JMeq x x.
```

- eq 型と異なり、両辺の型が異なっていてもよい
- item と C vars の等式を表現できる:
  @JMeq (T index) item (T index') (C vars)
- しかし、JMeq を eq に変換するには一般に公理 (UIP など) が必要になる
- dependent destruction tactic は JMeq を使う

Lean には JMeq と同様な HEq がある

```
inductive HEq : {α : Sort u} → α → {β : Sort u} → β →
    Prop where
  /-- Reflexivity of heterogeneous equality. -/
  | refl (a : α) : HEq a a
```

Lean の cases は HEq を使う

JMeq による等式の生成

```
Require Import JMeq.
Check fun (A : Type) (b : bool) (x : EvenOddList A b) ⇒
  match x as x' in EvenOddList _ b'
  return ∀ (eb : b = b') (ex : JMeq x x'), nat with
  | nil _ ⇒
      fun (eb : b = true) (ex : JMeq x (nil A))
        ⇒ 0
  | cons _ even h t ⇒
      fun (eb : b = negb even) (ex : JMeq x (cons A even h t))
        ⇒ 1
  end eq_refl JMeq_refl.
```

型が異なる等式では型を置換して合わせる

- 型が異なるなら、rewrite で型を変えてしまえば良い
- rewrite は項としては `eq_rect` であり、rew という記法がある
- 依存マッチで生成する index = index' の証明 e を利用して item の T index 型を T index' 型に置換する
- 後で e を `eq_refl` に具体化できると、rew は計算が進んで item に簡約できる
- e を `eq_refl` にするのに公理が必要になるかどうかは index と index' の内容による

rew 記法を使うと連立方程式は以下となる

- e : index = index'
- (rew e in item) = C vars

rew を使った等式の生成

```
Import EqNotations.
Check fun (A : Type) (b : bool) (x : EvenOddList A b) ⇒
  match x as x' in EvenOddList _ b'
  return ∀ (eb : b = b') (ex : (rew eb in x) = x'), nat with
  | nil _ ⇒
      fun (eb : b = true)
          (ex : (rew eb in x) = (nil A))
        ⇒ 0
  | cons _ even h t ⇒
      fun (eb : b = negb even)
          (ex : (rew eb in x) = (cons A even h t))
        ⇒ 1
  end eq_refl eq_refl.
```

型が異なる等式に sigma 型の等式を使う

等式の型が異なる部分を隠して型を合わせる

```
Inductive sigT (A : Type) (P : A → Type) : Type :=
  existT : ∀ x : A, P x → sigT P.
```

existT A P x px : sigT P

- `existT A P x px` は `x : A` と `px : P a` のペアであるが、型は `sigT P` で `x` は含まれない
- `existT T index item = existT T index' (C vars)` という等式を作れる
- `EqdepFacts.eq_sigT_fst` で `index = index'` を得られる
- `EqdepFacts.eq_sigT_snd` で `(rew eq_sigT_fst H in item) = C vars` を得られる

インデックスが複数ある場合は最初に sigma 型の等式を生成した方が便利だと思う
Equations パッケージではこのような形の等式を生成する

sigma 型を使った等式の生成

```
Check fun (A : Type) (b : bool) (x : EvenOddList A b) ⇒
  match x as x' in EvenOddList _ b'
  return ∀ (e :   existT (EvenOddList A) b x
                    = existT (EvenOddList A) b' x'), nat with
  | nil _ ⇒
      fun (e :   existT (EvenOddList A) b x
                 = existT (EvenOddList A) true (nil A))
        ⇒ 0
  | cons _ even h t ⇒
      fun (ex :   existT (EvenOddList A) b x
                  = existT (EvenOddList A)
                      (negb even) (cons A even h t))
        ⇒ 1
  end eq_refl.
```

型が異なる等式の生成の得失

- JMeq を使う (Lean では HEq)：公理が必要になることが確定
- 型を置換して合わせる：公理を使わない方法が残っているかもしれない
- sigma 型の等式を使う：型を置換する場合に変換できるので同様

公理 UIP (Uniqueness of Identity Proofs)

UIP は eq 型の証明はひとつしかないという公理

```
UIP: ∀(U : Type) (x y : U) (p1 p2 : x = y), p1 = p2
```

eq 型にはコンストラクタがひとつしかないので、それは単に事実と思われるが、なぜか CIC では証明できないので、必要なら公理として導入する（公理は中身のない定数）
UIP と等価な公理がいろいろある

- Invariance by Substitution of Reflexive Equality Proofs.
- Injectivity of Dependent Equality
- Uniqueness of Identity Proofs
- Uniqueness of Reflexive Identity Proofs
- Streicher's Axiom K

JMeq を eq に変換する `JMeq_eq` は UIP から導ける

```
JMeq_eq : ∀ (A : Type) (x y : A), JMeq x y → x = y
```

Coq と Lean と UIP

- Coq: UIP が必要なら明示的に導入する
- Lean: UIP よりも一般的な proof irrelevance が型規則に入っている

UIP と proof irrelevance

- UIP: eq 型の証明はどれも同じ
- proof irrelevance: ひとつの命題の証明はどれも同じ

Coq でこれらを使うには、以下のような型の公理を導入する

```
UIP: ∀(U : Type) (x y : U) (p1 p2 : x = y), p1 = p2
proof_irrelevance : ∀(P : Prop) (p1 p2 : P), p1 = p2
```

UIP は proof irrelevance の特殊形
Lean では proof irrelevance が型システムに組み込まれている

> *It also features proof irrelevance: any two proofs of the same proposition are definitionally equal.*
> (The Lean Language Reference: 4. The Type System)

型検査の中でふたつの項の unification が必要になった場合、それらが証明なら unification が常に成功する、となっていると想像される
つまり proof irrelevance を外すことはできない

公理 UIP (や proof irrelevance) の問題

- 公理を導入してしまうと、その公理を認めない場合の証明ができなくなる
- そして、HoTT (Homotopy type theory) では同じ命題に証明が複数あって UIP があると困る（らしい）
- つまり、UIP を導入すると HoTT を扱えなくなるなどの問題が発生する

Coq では、可能なら公理の導入は避けたい (CIC をそのまま使いたい)
JMeq を使うと公理が必須になるので、rew を使った等式のほうが好ましい

解が存在する場合に必要になる規則

- deletion: 同じ項の等式 $t = t$ を取り除く
- injectivity: 先頭が同じコンストラクタの等式は、各要素毎の等式に分解する: $C\ t_1 \ldots t'_n = C\ t'_1 \cdots t'_n$ を $t_1 = t'_1, \ldots t_n = t'_n$ に置き換える
- solution: ある等式 $x = t$ で t に x が出現せず、他の等式に x が出現している場合、その出現を t で置き換える: $x = t$ があったとき $P(x) = Q(x)$ を $P(t) = Q(t)$ にする

deletion には公理が必要な場合がある

以下で e を `eq_refl` に具体化して rew を取り除くには、一般には UIP が必要

- $e : x = x$
- $(\text{rew } e \text{ in item}) = C\ \text{vars}$

ただし、x の型に decidable equality があれば、公理がなくても取り除ける

injectivity と no confusion

- injectivity には no confusion という補題を利用する
- no confusion は、帰納型の値同士の等式と、その要素同士の等式を変換できる
- no confusion は帰納型ごとに定義するが、常に可能ではない
- 少なくとも、Prop な帰納型ではできない

nat の no confusion

```
From mathcomp Require Import all_ssreflect.

Definition Noconf_nat (m n : nat) : Prop :=
  match m, n with
  | 0, 0 ⇒ True
  | m'.+1, n'.+1 ⇒ m' = n'
  | _, _ ⇒ False
  end.

Definition noconf_nat {m n} : Noconf_nat m n → m = n.
  case: m; case: n ⇒ //; by move⇒ m n <-.
Defined.
Definition noconf_nat_inv {m n} : m = n → Noconf_nat m n.
  case: m; case: n ⇒ //; by move⇒ m n <-.
Defined.
Definition noconf_natK {m n} (H : m = n) :
    noconf_nat (noconf_nat_inv H) = H.
  case: m H ⇒ [|m] H; by subst n.
Defined.
```

(ここからは SSReflect 前提。`t.+1` は `S t` の記法)

長さを型に持つリスト: vec 型

```
Inductive vec (A : Type) : nat → Type :=
| vnil : vec A 0
| vcons : A → ∀ (n : nat), vec A n → vec A n.+1.
```

ここで、長さが 1 以上の vec は vcons で作られたものである、という定理を証明したい

```
Lemma vec_caseS :
  ∀ (A : Type) (n : nat) (v : vec A n.+1)
  (P : vec A n.+1 → Type),
  (∀ (h : A) (t : vec A n), P (vcons A h n t)) → P v.
```

vec_caseS の証明

```
Import EqNotations.
Lemma vec_caseS : ∀ (A : Type) (n : nat) (v : vec A n.+1)
        (P : vec A n.+1 → Type),
        (∀ (h : A) (t : vec A n), P (vcons A h n t)) → P v.
Proof.
  move⇒ A n v P H.
  (* 連立方程式の生成 *)
  refine (
    match v as v' in vec _ m return
      ∀ (Hn : n.+1 = m) (Hv : rew Hn in v = v'), P v
    with
    | vnil ⇒ _
    | vcons x n' t ⇒ _
    end erefl erefl).

    (* vnil の場合の証明 *)
    by [].

  (* vcons の場合の証明 *)
  move⇒ Hn.

  (* injectivity で n.+1 = n'.+1 を n = n' にする *)
  rewrite -(noconf_natK Hn).

  (* solution で n = n' の n' を除去する *)
  destruct (noconf_nat_inv Hn).

  simpl.
  move⇒ →.
  by apply H.
Defined.
```

match で生成したふたつの連立方程式

```
n : nat
v : vec A n.+1
P : vec A n.+1 → Type
H : ∀ (h : A) (t : vec A n), P (vcons A h n t)
______________________________________(1/2)
∀ Hn : n.+1 = 0, rew [vec A] Hn in v = vnil A → P v
______________________________________(2/2)
∀ Hn : n.+1 = n'.+1, rew [vec A] Hn in v = vcons A x n' t → P v
```

- vnil の場合は以下の連立方程式が生成 (矛盾しているので簡単に証明可能)

  ```
  Hn : n.+1 = 0
  rew [vec A] Hn in v = vnil A
  ```

- vcons の場合は以下の連立方程式が生成

  ```
  Hn : n.+1 = n'.+1
  rew [vec A] Hn in v = vcons A x n' t
  ```

vcons の場合のゴール

```
n : nat
v : vec A n.+1
P : vec A n.+1 → Type
H : ∀ (h : A) (t : vec A n), P (vcons A h n t)
x : A
n' : nat
t : vec A n'
Hn : n.+1 = n'.+1
______________________________________(1/1)
rew [vec A] Hn in v = vcons A x n' t → P v
```

- rew の計算を進めるために Hn を場合分けして `eq_refl` にしたい
- しかし destruct Hn で Hn を `eq_refl` にすると、同時に起こるインデックスの置換が n'.+1 を n.+1 に置き換えるものなので、n' が残ってしまう (そのため型エラーになる)

no confusion をつかう

rewrite -(noconf_natK Hn) で Hn を noconf_nat (noconf_nat_inv Hn) に書き換える

```
n : nat
v : vec A n.+1
P : vec A n.+1 → Type
H : ∀ (h : A) (t : vec A n), P (vcons A h n t)
x : A
n' : nat
t : vec A n'
Hn : n.+1 = n'.+1
______________________________________(1/1)
rew [vec A] noconf_nat (noconf_nat_inv Hn) in v
  = vcons A x n' t → P v
```

- ここで noconf_nat_inv Hn : n = n' である
  destruct (noconf_nat_inv Hn) により noconf_nat_inv Hn が eq_refl になると同時に n' が n に置換される（すべての n' が置換されて消える）
- noconf_nat eq_refl は計算すると eq_refl になるので、rew も消える

依存マッチで証明を進める方法

依存マッチで、各コンストラクタ毎に連立方程式を生成する

- 連立方程式に関数が含まれる → がんばれ
- 解いている途中に $C \cdots = C \cdots$ という形が出てきて no confusion がない → がんばれ
- 解いている途中に $x = x$ の形が出てきて、decidable equality がない → がんばれ（あるいはあきらめて公理を使う）

ぜんぶクリアできたら公理を使わずに済む

Coq と Lean の違い

Coq:

- match と fix がプリミティブで、termination checker が必要
- extraction が容易
- CIC そのままが基本で、公理はなるべく導入しない
- 依存マッチは面倒
- JMeq を使う dependent destruction があるが、公理が必要
- 公理を使わない場合分けをサポートするパッケージもある (Equations)
- HoTT を扱える

Lean:

- recursor がプリミティブで、termination checker が不要
- 普通の match や再帰の形のコードとの変換が複雑
- proof irrelevance が型システムに組み込まれている
- cases は HEq を使う
- HEq を eq にするために公理が必要になるが、proof irrelevance が型システムに組み込まれているのでそれで済む (公理を避ける意味がない)
- HoTT は扱えない

参考

- Leonardo de Moura, Gabriel Ebner, Jared Roesch, Sebastian Ullrich. The Lean Theorem Prover. POPL 2017 Tutorial.
- Matthieu Sozeau and Cyprien Mangin. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. ICFP 2019.
- Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. ICFP 2014.
- Pierre-Marie Pédrot. A Kernel of Truth. TPP 2025
- The Lean Reference Manual (Lean 3)
- The Lean Language Reference (Lean 4)
- Rocq Reference Manual

まとめ

- Coq にくらべて Lean は公理に寛容だと思う (proof irrelevance だけかも?)
- 依存マッチは難しく、Coq には悪戦苦闘の跡がみえる
- Lean は依存マッチが簡単でよい（ただし公理が前提）
- Coq の Equations パッケージは依存マッチに便利（でも自動的に変数名が生成されるのが残念）