定理証明支援系 Coq での論理的なリーゾニング

集中講義 千葉大学大学院

アフェルト レナルド

産業技術総合研究所

2017 年 01 月 25 日 (1/2)

目的

- Coq [CDT16] は型理論の実装である．型理論で命題論理と述語論理を表現しやすい．
- Coq でタクティック [CDT16, Chapitre 8] を用いてリーゾニングを行う．タクティックは自然演繹のルールに近い．
- Coq のタクティックを用いて, ([DNR03] による) 自然演繹のルールを説明する．
- Coq は型付きラムダ計算であることは重要である:型は論理式であり, 項は証明である．
- 式が成り立つというのは, その型を持つ項があるということである．Coq は **Curry-Howard 同型対応**の一例である．

アウトライン

Coq の概要

Coq での命題論理
- 公理と含意
- 論理積と論理和
- 矛盾と否定

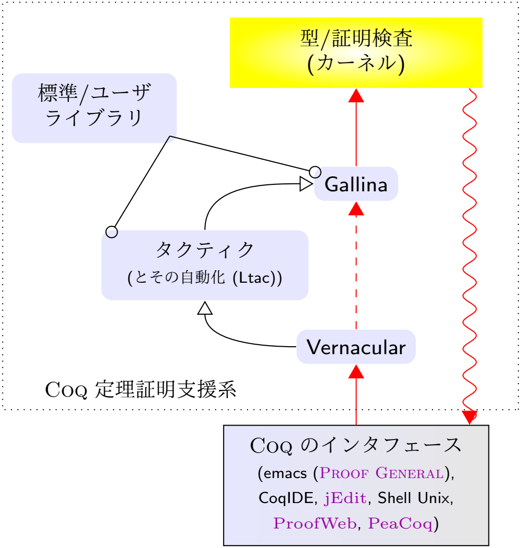
Coq での述語論理

補足

定理証明支援系 Coq

- INRIA で T. Coquand と G. Huet が 1984 年に開発を始めた
- 基本的に, Gallina というラムダ計算である
  - Calculus of Inductive Constructions (CIC) [CP90, PM92]
  - Calculus of Constructions の拡張 [CH84, CH85, CH86, CH88]
- たくさん使われている
  - 世界中で, 研究と教育で
  - 賞: ACM SIGPLAN Programming Languages Software 2013 award, ACM Software System 2013 award
  - 産業への応用の試み (CC EAL7 認証)
- 成功例: 四色定理 [Gon05, Gon08], C のコンパイラ [Ler09, BL09], 奇数位数定理 [GAA$^+$13], 等

Coq システムの概要



→: コマンド言語を用いて，補題/データ構造の追加 Vernacular

- - ▶: Gallina を用いて，項の直接な記述

〜▶: 型/証明検査の失敗の際の情報/エラーメッセージ

—▷: 実際に，タクティクを用いて，項を間接に記述する

—○: 標準ライブラリはデータ構造，タクティクや証明済みの補題を提供する

Coq のインタフェース

入力

コマンドとタクティク

```
Require Import
  Classical.
```

環境 (E)

```
Goal forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
intros P Q.
```

⋮

証明のスクリプト

⋮

出力

ゴール

```
P : Prop
Q : Prop
```

ローカルコンテキスト (Γ)

```
=====================
```

水平線

$\underbrace{\underbrace{\texttt{(P -> Q)}}_{\text{トップ}} \texttt{ -> P ->}}_{\text{仮定のスタック}} \underbrace{\texttt{Q}}_{\text{ゴール}}$

エラーメッセージ,
検索の結果, 等

Coq の出力をシークエントとして読む

- ```
  p : P
  q : Q
  ======
  R
  ```
  は $P, Q \vdash R$ として読む
- `P`, `Q`, `R` は命題である
- 仮定 (`P`, `Q`) の証明 (`p`, `q`) は明確である
- 結論 `R` の (構築中の) 証明は `Show Proof` コマンドで表示できる
  ```

アウトライン

Coq の概要

Coq での命題論理

- 公理と含意
- 論理積と論理和
- 矛盾と否定

Coq での述語論理

補足

自然演繹ルール「公理」

- 「証明があれば, 命題を証明できる」
- タクティク exact によって実装されている
  - 型検査しか呼ばないので, もっとも基本的なタクティクである

$$\frac{}{\Gamma, A \vdash A}\,\mathsf{ax}$$

```
A : Prop
a : A
==========
A
```

→exact a.→ No more subgoals.

含意 ($\to$) の導入

- 「$A \to B$ を証明するために, A を仮定してから B を証明するのは十分である」
- タクティク intros/revert を用いて, $\vdash$ の前/後に仮定を動かすだけ
  - ローカルコンテキストの仮定の証明に, 名前を付けなければならない

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to_i$$

```
A, B : Prop
===========
A -> B
```

$\to$`intros a.`[1]$\to$

$\leftarrow$`revert a.`[2]$\leftarrow$

```
A, B : Prop
a : A
===========
B
```

[1]一つのパラメータだけなら, intro も使える

[2]revert はローカルコンテキストから仮定を消す. generalize はコピーを行う.

含意 ($\to$) の除去

- 「$A \to B$ と A を知ると, B を推論できる」 (modus ponens)
- 含意の除去は関数の適用として理解する
  - もし `ab` は `A -> B` の証明であり, `a` は `A` の証明であれば, `ab a` は `B` の証明である
- 一つのサブゴールしか生成しない `apply` タクティクを使用する[3]

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to_e$$

```
A, B : Prop                              A : Prop
ab : A -> B                              ab : A -> B
a : A          →apply ab.→               a : A
===========                              ===========
B                                        A
```

[3] `cut` は $\to_e$ にもっと近いが, 不便である

証明の例 (1/7)

ゴールの記述

$$\vdash (A \to B \to C) \to (A \to B) \to A \to C$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
```

証明の例 (2/7)

一番目の含意の導入

$$\dfrac{\overbrace{A \to B \to C}^{\texttt{abc}} \vdash (A \to B) \to A \to C}{\vdash (A \to B \to C) \to (A \to B) \to A \to C} \to_i$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc.
```

証明の例 (3/7)

三つの含意の導入

$$
\dfrac{\dfrac{\dfrac{\overbrace{A \to B \to C}^{\texttt{abc}}, \overbrace{A \to B}^{\texttt{ab}}, \overbrace{A}^{\texttt{a}} \vdash C}{A \to B \to C, A \to B \vdash A \to C} \to_i}{A \to B \to C \vdash (A \to B) \to A \to C} \to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C} \to_i
$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
```

証明の例 (4/7)

含意の一番目の除去

$$
\dfrac{\dfrac{\dfrac{\dfrac{A \to B \to C, A \to B, A \vdash A \qquad A \to B \to C, A \to B, A \vdash B}{\overbrace{A \to B \to C}^{\texttt{abc}}, A \to B, A \vdash C}\to_e}{A \to B \to C, A \to B \vdash A \to C}\to_i}{A \to B \to C \vdash (A \to B) \to A \to C}\to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\to_i
$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
```

($A \to B \to C$ の結論と C をマッチし, 二つのサブゴール A と B を生成する)

証明の例 (5/7)

「公理」ルールを適用

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \to C, A \to B, \overbrace{A}^{\mathtt{a}} \vdash A}\,\mathsf{ax} \qquad A \to B \to C, A \to B, A \vdash B}{A \to B \to C, A \to B, A \vdash C}\,{\to_e}}{A \to B \to C, A \to B \vdash A \to C}\,{\to_i}}{A \to B \to C \vdash (A \to B) \to A \to C}\,{\to_i}}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,{\to_i}
$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
```

証明の例 (6/7)

含意の二番目の除去

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \to C, A \to B, A \vdash A}\,\mathrm{ax} \qquad \dfrac{A \to B \to C, A \to B, A \vdash {\color{red}A}}{A \to B \to C, \overbrace{{\color{red}A \to}{\color{green}B}}^{\mathtt{ab}}, A \vdash {\color{green}B}}\to_e}{A \to B \to C, A \to B, A \vdash C}\to_e}{A \to B \to C, A \to B \vdash A \to C}\to_i}{A \to B \to C \vdash (A \to B) \to A \to C}\to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\to_i
$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
- apply ab.
```

($A \to B$ の結論と B をマッチし, サブゴール A を生成する)

証明の例 (7/7)

公理と Qed

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \to C, A \to B, A \vdash A}\,\mathrm{ax} \qquad \dfrac{\dfrac{}{A \to B \to C, A \to B, \overbrace{A}^{\mathtt{a}} \vdash A}\,\mathrm{ax}}{A \to B \to C, A \to B, A \vdash B}\,{\to_e}}{A \to B \to C, A \to B, A \vdash C}\,{\to_e}}{A \to B \to C, A \to B \vdash A \to C}\,{\to_i}}{A \to B \to C \vdash (A \to B) \to A \to C}\,{\to_i}}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,{\to_i}
$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
- apply ab.
  exact a.
Qed.
```

Qed (Quod Erat Demonstrandum) を hilbertS の識別子を用いて, 証明を記録する

証明と証明のスクリプトの違い

- 実は, 書いたものは, 証明のスクリプト
- 証明自体は次のように表示できる:

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
- apply ab.
  exact a.
Show Proof.
```

```
(fun (A B C : Prop) (abc : A -> B -> C) (ab : A -> B) (a : A) =>
 abc a (ab a))
```

- タクティクはラムダ計算の項を書くための間接な方法だけである. ただし, その項は証明として見る.
- Coq は Curry-Howard 同型対応の実装の一例である

証明項の例

Hilbert の S 公理

- 途中のゴールの裏に隠れている (構築中の) 証明項をすべて表示する:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash a : A} \qquad \dfrac{\dfrac{}{\Gamma \vdash a : A}}{\Gamma \vdash ab\,a : B}}{\Gamma^{4} \vdash abc\,a\,(ab\,a) : C}}{abc : A \to B \to C, ab : A \to B \vdash \begin{array}{c}\lambda a : A.\\ abc\,a\,(ab\,a)\end{array} : A \to C}}{abc : A \to B \to C \vdash \begin{array}{c}\lambda ab : A \to B.\lambda a : A.\\ abc\,a\,(ab\,a)\end{array} : (A \to B) \to A \to C}
$$

$$
\overline{\vdash \begin{array}{c}\lambda abc : A \to B \to C.\lambda ab : A \to B.\lambda a : A.\\ abc\,a\,(ab\,a)\end{array} : (A \to B \to C) \to (A \to B) \to A \to C}
$$

- どうして Coq のインタフェースが証明項を毎回表示しないことが分かるでしょう...

[4] $\Gamma = abc : A \to B \to C, ab : A \to B, a : A$

アウトライン

Coq の概要

Coq での命題論理

- 公理と含意
- 論理積と論理和
- 矛盾と否定

Coq での述語論理

補足

論理積 (∧) の導入

- 「$A \wedge B$ を証明するために, A と B を証明するのは十分である」[5]
  - 証明木の中に枝を追加すること
- その論法はタクティク `split` によって実装されている

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i$$

```
A, B : Prop                     A, B : Prop           A, B : Prop
a : A                           a : A                 a : A
b : B             →split→       b : B                 b : B
==========                      ==========            ==========
A /\ B                          A                     B
```

[5] はい, 当たり前のように聞こえる

論理積の (∧) の除去

- 「A (resp. B) を証明するために, $A \wedge B$ を証明すれば十分である」 (sic)
  - アイデア: 論理積を特化できる
- タクティク `exact` と `proj1`/`proj2` 補題の組み合わせ

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_e^g \qquad \left(\text{resp.} \ \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_e^d \right)$$

```
A, B : Prop
ab : A /\ B
==========
A
```

→`exact (proj1 ab)`[6]→ `No more subgoals.`

[6] resp. `proj2`

どうやって論理積 (∧) は定義されている?

論理積は証明のペアの型として定義されている:

型　パラメータ　ソート

```
Inductive and (A B : Prop) : Prop :=
  conj : A -> B -> A /\ B.
```

構成子

構成子の型

- A と B は命題 (Prop) であれば, A /\ B も命題である
- A /\ B は and A B のための記法である
- a は A 証明であり, b は B の証明であれば, conj a b は A /\ B の証明である

論理積 (∧) の使用に当たるテクニカルな話

- タクティク split は関数として見た構成子 conj の適用に当たる:

  split ≈ apply conj

- proj1/proj2 は標準の補題である:

```
Theorem proj1 : A /\ B -> A.
Theorem proj2 : A /\ B -> B.
```

- 実際に, proj1 と proj2 を使うより, タクティク destruct の方が便利である:

```
A, B : Prop                                     A, B : Prop
ab : A /\ B                                     a : A
==========   →destruct ab as [a b]→             b : B
A                                               ==========
                                                A
```

論理積を使う証明項の例

タクティク destruct は Gallina(Coq の言語) でのパターンマッチの間接な方法である:

```
Lemma myproj1 (A B : Prop) : A /\ B -> A.
Proof.
exact (fun ab : A /\ B =>
  match ab with conj a b => a end).
Qed.
```

論理和 (∨) の導入

- 「$A \vee B$ を証明するために, A または B を証明するのは十分である」
  - アイデア: このルールを用いて、ゴールを特化する
  - $\wedge_e^g / \wedge_e^d$ と比較
- この論法はタクティック `left` と `right` によって実装されている

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_i^g \quad \left(\text{resp.} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_i^d \right)$$

```
A , B : Prop                      A , B : Prop
a : A                             a : A
===========     →left[7]→         ===========
A \/ B                            A
```

[7] resp. `right`

論理和 ($\vee$) の除去

- 論理和を用いて, 場合分けを行う
  - 証明木の中に, 枝を追加する
  - $\wedge_i$ と比較
- タクティク destruct を使う

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e$$

```
A,B,C : Prop
ab : A \/ B
==========
C
```

→destruct ab as [a|b]→

```
A,B : Prop
a : A
==========
C
```

```
A,B : Prop
b : B
==========
C
```

([a|b], 論理積の場合の [a b] と違う—slide 25)

どうやって論理和は定義されていますか?

論理和は二つの構成子を持つ型として定義されている:

```
                 型       パラメータ          ソート
Inductive or (A B : Prop) : Prop :=
構成子 ──→ | or_introl : A -> A \/ B
           | or_intror : B -> A \/ B
                                  構成子の型
```

- A と B は命題 (Prop) であれば, A \/ B も命題である
- A \/ B は or A B のための記法
- a は A の証明であると, or_introl B a は A \/ B の証明になる
- b は B の証明であると, or_intror A b は A \/ B の証明になる

論理和 (∨) の使用に当たるテクニカルな話

タクティク `left` は関数として見る構成子 `or_introl` の適用である:

$$\texttt{left} \approx \texttt{apply or\_introl}$$

タクティク `right` は関数として見る構成子 `or_intror` の適用である:

$$\texttt{right} \approx \texttt{apply or\_intror}$$

論理和を使う証明項の一例

論理和の除去のルール

メモ: $$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e$$

```
Lemma or_elim (A B C : Prop) :
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
```

論理和を使う証明項の一例

タクティクを用いた証明

```
Lemma or_elim_tactique (A B C : Prop) :
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
intros ac bc.
destruct 1 as [a | b].
- apply ac.
  exact a.
- apply bc.
  exact b.
Qed.
```

論理和を使う証明項の一例

Gallina 関数を用いた証明

```
Lemma or_elim_gallina (A B C : Prop) :
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
exact (fun (ac : A -> C) (bc : B -> C) (ab : A \/ B) =>
match ab with
| or_introl a => ac a
| or_intror b => bc b
end).
Qed.
```

実際に, 両方のスクリプトは 同じ証明 を構築する.

アウトライン

Coq の概要

Coq での命題論理

- 公理と含意
- 論理積と論理和
- 矛盾と否定

Coq での述語論理

補足

Coq での矛盾と否定

- 矛盾 (紙上の記法: $\perp$) は Coq で `False` と書く
  - `False` は基本的な概念でない．定義されている (slide 39)
- A の否定 (紙上の記法: $\neg A$) は Coq で `~ A` (「tilda A」と読む) と書く
  - 「`~`」も基本的な結合子ではない．定義されている:
    - 紙上, $\neg A$ は $A \to \perp$ として定義さている
    - Coq で, $A \to \perp$ は `A -> False` となる
    - 見えやすくするように, `A -> False` の代わりに, `~ A` と書く

否定 (¬) の導入と除去

メモ [DNR03]:

$$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\neg_i \qquad \dfrac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\neg_e$$

上記のルールで, $\neg A$ の代わりに, その定義 $A \to \bot$ と書く:

$$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash A \to \bot}\neg_i \qquad \dfrac{\Gamma \vdash A \to \bot \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\neg_e$$

そうすると, 否定のルールは含意のルールのインスタンスであることが分かる (下記のの B の代わりに, $\bot$ と書いただけ):

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\to_i \qquad \dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\to_e$$

従って, 否定のために, 新しいタクティクを加える必要はない

Coq での否定 (¬) の導入

「$\neg A$ をするために, A が成り立つコンテキストで $\bot$ を証明するのは十分である」

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i$$

```
A  :  Prop
==========
~  A
```

→`intros a.`→

```
A  :  Prop
a  :  A
==========
False
```

COQ での否定 (¬) の除去

「⊥ を証明するために, A と $\neg A$ が成り立つ A を見つかるのは十分である」

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg_e$$

```
A : Prop
na : ~ A
a : A             →apply na.[8]→  No more subgoals.
===========
False
```

[8] または: `exact (na a)`

Coq で, どうやって矛盾 (⊥) が定義されている?

- ⊥ は証明の作れない命題である
- Coq で, ⊥ は要素のない型として定義されている
- 具体的に, `False` は構成子のない帰納的型として定義する. 次のコマンドで実現する:

$$\texttt{Inductive}\ \underbrace{\texttt{False}}_{\text{Type}}\ :\ \underbrace{\texttt{Prop}}_{\text{Sort}}\ :=\ .$$

矛盾 ($\perp$) の除去のルール

- 実際に, コンテキストで $\perp$ があれば, 除去するだけで, ゴールを証明する
- *ex falso quodlibet*: 矛盾から, 何でも証明できる

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash A} \perp_e$$

```
A : Prop
abs : False
==========    →destruct abs.→   No more subgoals.
A
```

COQ での古典論理

- COQ は**構成的論理**: 古典論理の性質を使えないように定義されている
  - 例: **排中律** (「命題は成り立つまたは成り立たない」)
- ただし, 古典論理を導入しても, 矛盾を導かない
  - COQ で古典論理は**公理** (証明のない補題) として用意されている

「証明できる」古典論理の論法の低:

「もし $\neg A$ から $\bot$ を証明できれば, A は成り立つ」

```
Require Import Classical.

Lemma bottom_c (A : Prop) : ((~A) -> False) -> A.
Proof. ... Qed.
```

古典論理による矛盾

古典論理の矛盾のルールは前のスライド (slide 41) の bottom_c 補題を用いて証明できる:

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \bot_c$$

```
A : Prop
==========
A
```

$\rightarrow$

```
apply bottom_c.
intros na.
```

$\rightarrow$

```
A : Prop
na : ~ A
==========
False
```

自然演繹の「Weakening」ルール

全てのルールを列挙するように...

- 不要な仮定は出てくることがある
- この論法はタクティク `clear` を用いて実現できる

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \mathrm{weak}$$

```
A , B : Prop
a : A
b : B
===========
A
```

→`clear b.`→

```
A , B : Prop
a : A
===========
A
```

アウトライン

全称記号の導入

- 「$\forall x.A$ を証明するために, 任意の x に対して A を証明するのは十分である」
- 楽なところ:
  - $\forall (x : X), A$ は $X \to A$ のただの一般化 (特に, 同じタクティク)
  - Coq が識別子の管理を自動的に行う

$$\frac{\Gamma \vdash A \qquad x \text{ は } \Gamma \text{ の式の自由変数ではない}}{\Gamma \vdash \forall x.A}\ \forall_i$$

```
X : Type
A : X -> Prop
x : X
==========
forall x : X, A x
```

→intros x0.→

←revert x0.[9]←

```
X : Type
A : X -> Prop
x : X
x0 : X
==========
A x0
```

[9] generalize x0 は新しい変数 x1 を作成する

全称記号の除去

- 全称記号の除去は関数の適用として理解する
- 含意の除去に似ている (特に, 同じタクティク)

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x := t]}\ \forall_e$$

```
X : Type
A : X -> Prop
Ax : forall x : X, A x
t : X
==========
A t
```

→`apply Ax.`[10]→

```
No more
  subgoals.
```

[10] または `exact (Ax t)`

存在記号の導入

- 「$\exists x.A$ を証明するために, $A\,t$ が成り立つように witness t を見つけるのは十分である」
- この論法は `exists` タクティクとして実装されている

$$\frac{\Gamma \vdash A[x := t]}{\Gamma \vdash \exists x.A}\ \exists_i$$

```
X : Type
x : X
A : X -> Prop
t : X
At : A t
==========
exists x0 : X, A x0
```

→exists t→

```
X : Type
x : X
A : X -> Prop
t : X
At : A t
==========
A t
```

存在記号の除去

- $\exists x.A$ は witness t と証明 $A\,t$ のペアとして見る
- タクティク `destruct` は witness とその証明を明確にする
  - Coq が問題を起こさない識別子を選ばれることを確認する

$$\frac{\Gamma \vdash \exists x.A \qquad \Gamma, A \vdash C \qquad x\ \text{は}\ \Gamma\ \text{の式と}\ C\ \text{の自由変数ではない}}{\Gamma \vdash C}\ \exists_e$$

```
C : Prop
X : Type
x : X
A : X -> Prop
At : exists t : X,
       A t
==========
C
```

→`destruct At as [t At]`→

```
C : Prop
X : Type
x : X
A : X -> Prop
t : X
At : A t
==========
C
```

どうやって存在記号が定義されている?

存在記号はペアとして実装されている:

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
| ex_intro : forall x : A, P x -> exists x, P x
```

witness　証明　= ex P
(witness はない, 証明もない)

- exists x, P x は ex P のための記法である
- タクティク exists t は構成子のただの適用である: apply (ex_intro _ t)

同値関係の導入

- タクティク `reflexivity` は同値関係が反射関係であることを表現する

$$\frac{}{\Gamma \vdash t = t} =_i$$

```
X : Type
t : X
==========
t = t
```

→reflexivity→ `No more subgoals.`

同値関係の除去

- 同値関係の除去は書き換えとして実装されている

$$\frac{\Gamma \vdash A[x := t] \qquad \Gamma \vdash t = u}{\Gamma \vdash A[x := u]} =_e$$

```
X : Type
t, u : X
A : X -> Prop
At : A t
tu : t = u
==========
A u
```

→`rewrite <-tu`[11]→

```
X : Type
t, u : X
A : X -> Prop
At : A t
tu : t = u
==========
A t
```

[11]逆向き: `rewrite ->tu` あるいは `rewrite tu`(もう少し短い)

どうやって同値関係が実装されている?

Coq では同値関係は index を持つ帰納的型として定義されている:

arity

パラメータ　index　ソート

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x
```

family argument　index argument

- x = y は eq _ x y のための記法である
- reflexivity = apply eq_refl

rewrite は何をする?

帰納法の一例

- 帰納的型を定義する際, Coq が帰納法 (とその証明!) を生成する:

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
  P x -> forall y : A, x = y -> P y
```

- eq_ind は, 基本的に, 任意の P に対して, P x を P y に変換する (**Leibniz 同値関係と言う**)
- 実は, 全スライドの rewrite <-tu は次のタクティクと同じ:

```
apply (eq_ind _ _ At _ tu).
```

- 結局, 今回の重要なタクティクは少ない: intros/revert, apply, destruct ... as [...] (, clear)

アウトライン

Coq の概要

Coq での命題論理
- 公理と含意
- 論理積と論理和
- 矛盾と否定

Coq での述語論理

補足

演習問題

- 論理の授業 ([Pie16], [DNR03]) の例を形式化する
- 帰納的型を使わない (つまり, CIC より CoC を使う) 論理結合子の形式化を検討する
- 本スライドと演習問題: https://github.com/affeldt/ssrcoq-chiba2017

本スライドのタクティクのまとめ

参考文献 I

Sandrine Blazy and Xavier Leroy, *Mechanized semantics for the Clight subset of the C language*, J. Autom. Reasoning **43** (2009), no. 3, 263–288.

The Coq Development Team, *The Coq proof assistant reference manual*, INRIA, 2016, Version 8.5.

Thierry Coquand and Gérard Huet, *A theory of constructions (preliminary version)*, International Symposium on Semantics of Data Types, Sophia-Antipolis, 1984, Jun. 1984.

______, *Constructions: A higher order proof system for mechanizing mathematics*, Proceedings of the European Conference on Computer Algebra, Linz, Austria EUROCAL 85, April 1–3, 1985, Linz, Austria, vol. 1 (Invited Lectures), Apr. 1985, pp. 151–184.

______, *Concepts mathématiques et informatiques formalisés dans le calcul des constructions*, Tech. Report 515, INRIA Rocquencourt, Apr. 1986.

______, *The calculus of constructions*, Information and Computation **76** (1988), 95–120.

Thierry Coquand and Christine Paulin, *Inductively defined types*, Proceedings of the International Conference on Computer Logic (COLOG-88), Tallinn, USSR, December 1988, Lecture Notes in Computer Science, vol. 417, Springer, 1990, pp. 50–66.

René David, Karim Nour, and Christophe Raffalli, *Introduction à la logique*, 2ème ed., Dunod, 2003.

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry, *A machine-checked proof of the odd order theorem*, Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013, Rennes, France, July 22–26, 2013, Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 163–179.

参考文献 II

Georges Gonthier, *A computer-checked proof of the four colour theorem*, Tech. report, Microsoft Research, Cambridge, 2005, Available at: http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf. Last access: 2014/08/04.

———, *Formal proof—the four-color theorem*, Notices of the American Mathematical Society **55** (2008), no. 11, 1382–1393.

Xavier Leroy, *A formally verified compiler back-end*, J. Autom. Reasoning **43** (2009), no. 4, 363–446.

Thomas Pietrzak, *Logique—logique propositionnelle—*, Licence Informatique, Université de Lille 1 Sciences et Technologies, 2016, http://www.thomaspietrzak.com/download.php?f=CoursLogique0.pdf.

Christine Paulin-Mohring, *Inductive definitions in the sytem coq rules and properties*, Tech. Report 92–49, LIP, École Normales Supérieure de Lyon, Dec. 1992.