「計算と論理」
Software Foundations
その6

五十嵐 淳

cal24@fos.kuis.kyoto-u.ac.jp

http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/

京都大学
大学院情報学研究科
工学部情報学科計算機科学コース

December 10, 2024

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

命題

Coq では，命題も (プログラムと同じく) 項の一種.
“Prop” 型を持つ.

```
Coq < Check 3=3.
3 = 3
     : Prop

Coq < Check (forall n m :nat, n + m = m + n).
forall n m : nat, n + m = m + n
     : Prop

Coq < Check (forall (n:nat) (b:bool), n = b).
Toplevel input, characters 36-37:
> Check (forall (n:nat) (b:bool), n = b).
>                                     ^
Error:
In environment
n : nat
```

命題

ただし,

項が Prop 型を持つ $\not\Leftrightarrow$ その項が表す命題が証明可能

```
Coq < Check 2=3.
2 = 3
     : Prop

Coq < Check (forall (n:nat), n = 2).
forall n : nat, n = 2
     : Prop
```

命題を定義する

```
Definition plus_claim : Prop := 2 + 2 = 4.

Theorem plus_claim_is_true : plus_claim.
Proof. reflexivity.  Qed.
```

パラメータ化された命題

述語 (predicate) とは，引数を受け取って命題を返す関数

```
Coq < Definition is_three (n : nat) : Prop := n = 3.
is_three is defined
```

```
Coq < Check is_three.
is_three
     : nat -> Prop
```

n = m は eq n m の構文糖衣:

```
Coq < Check @eq.
@eq
     : forall A : Type, A -> A -> Prop
```

eq の型引数 ***A*** は省略可能

述語「f は単射である」

```
Definition injective {A B} (f : A -> B) :=
  forall x y : A, f x = f y -> x = y.

Lemma succ_inj : injective S.
Proof.
  intros n m H. injection H as H1. apply H1.
Qed.
```

Logic.v

- 命題
- 論理結合子
  - 連言 (「かつ」)
  - 選言 (「または」)
  - 偽 (矛盾) と否定 (「〜でない」)
  - 真
  - 論理的同値 (if and only if)
  - 存在量化 (「ある x が存在して〜」)
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

連言 (conjunction)

$A \wedge B$ …「A かつ B」

- $A \wedge B$ を証明するためには A と B をそれぞれ証明する
- `split` タクティックでゴールをふたつに分割

```
Example and_example : 3 + 4 = 7 /\ 2 * 2 = 4.
Proof.
  split.
  - (* 3 + 4 = 7 *) reflexivity.
  - (* 2 + 2 = 4 *) reflexivity.
Qed.
```

連言の導入

```
Lemma and_intro : forall A B : Prop,
  A -> B -> A /\ B.
Proof.
  intros A B HA HB. split.
  - apply HA.
  - apply HB.
Qed.
```

- 「任意の命題 A, B について」は (おそらく) 初出
  - でも意味はわかりますね？
- `split` と `apply and_intro` は実質同じ.

連言を使って何かいう

仮定にある $A \wedge B$ は destruct でふたつの仮定 A と B に分解できる.

```
Lemma and_example2 : forall n m: nat,
  n = 0 /\ m = 0 -> n + m = 0.
Proof.
  intros n m H.
  destruct H as [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

連言を経由して何かいう

```
Lemma and_example3 :
  forall n m : nat, n + m = 0 -> n * m = 0.
Proof.
  (* WORKED IN CLASS *)
  intros n m H.
  apply and_exercise in H.
  destruct H as [Hn Hm].
  rewrite Hn. reflexivity.
Qed.
```

不要な conjunct を捨てる

```
Lemma proj1 : forall P Q : Prop,
  P /\ Q -> P.
Proof.
  intros P Q H.  destruct H as [HP _].
  apply HP.  Qed.
```

Logic.v

- 命題
- 論理結合子
  - 連言 (「かつ」)
  - 選言 (「または」)
  - 偽 (矛盾) と否定 (「〜でない」)
  - 真
  - 論理的同値 (if and only if)
  - 存在量化 (「ある x が存在して〜」)
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

選言 (disjunction)

A \/ B … 「A または B」
「または」から何かを言うには destruct で場合分けをする:

```
Lemma factor_is_0 : forall n m: nat,
  n = 0 \/ m = 0 -> n * m = 0.
Proof.
  intros n m H.  destruct H as [Hn | Hm].
  - (* Here, n = 0 *)
    rewrite Hn. reflexivity.
  - (* Here, m = 0 *)
    rewrite Hm. rewrite <- mult_n_O.
    reflexivity.
Qed.
```

選言の導入

タクティック left と right

```
Lemma or_intro_l :
  forall A B : Prop, A -> A \/ B.
Proof.
  intros A B HA. left. apply HA. Qed.

Lemma zero_or_succ :
  forall n : nat, n = 0 \/ n = S (pred n).
Proof.
  intros [|n].
  - left. reflexivity.
  - right. reflexivity.
Qed.
```

ところで

- $P \wedge Q$ は *and P Q* の略記
- 同様に $P \vee Q$ は *or P Q* の略記
- *and*, *or* は命題ふたつを受けとって命題を返す関数型を持つ

```
Coq < Check and.
and
     : Prop -> Prop -> Prop

Coq < Check or.
or
     : Prop -> Prop -> Prop
```

Logic.v

- 命題
- 論理結合子
  - 連言 (「かつ」)
  - 選言 (「または」)
  - 偽 (矛盾) と否定 (「〜でない」)
    - ★ 不等号 (等しくない)
  - 真
  - 論理的同値 (if and only if)
  - 存在量化 (「ある x が存在して〜」)
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

否定と矛盾

「P ではない」($\neg P$, Coq では $\sim P$) の定義

```
Definition not (P:Prop) := P -> False.
 (* False:  「矛盾」「偽」を表す特殊な命題 *)
 (*  「P ではない」 = P を仮定すると矛盾する *)
 (* not : Prop -> Prop
    命題を受け取って命題を返す関数! *)

Notation "~ x" := (not x) : type_scope.
```

爆発則

矛盾からは何でもいえる:

```
Theorem ex_falso_quodlibet : forall (P:Prop),
  False -> P.
Proof.
  intros P contra.
  destruct contra.  Qed.
```

- ここで destruct は discriminate と同様な働き
  - discriminate は矛盾した等号に使う
  - 微妙な違いの理由はもう少し先 (IndProp.v) に進むとわかる
- 定理名はラテン語で「偽からは何でも導ける」の意

不等号(否定の証明(1))

$x <> y$ は $\neg(x = y)$ のこと:

```
Notation "x <> y" := (~ (x = y)) : type_scope.

Theorem zero_not_one : 0 <> 1.
Proof.
  unfold not. (* expands to (0 = 1) -> False *)
  intros contra. discriminate contra.
Qed.
```

否定の証明(2)

否定の証明は(Falseを導くことになるので)少しコツが必要なことも.

```
Theorem not_False : ~ False.
Proof.
  unfold not. intros H. destruct H.  Qed.

Theorem contradiction_implies_anything :
  forall P Q : Prop, (P /\ ~P) -> Q.
Proof.
  intros P Q [HP HNP].
  unfold not in HNP.
  apply HNP in HP. destruct HP.  Qed.
```

否定の証明(3)

```
Theorem double_neg : forall P : Prop,
  P -> ~~P.
Proof.
  intros P H. unfold not.
  intros G. apply G. apply H.  Qed.
```

```
Theorem not_true_is_false : forall b : bool,
  b <> true -> b = false.
Proof.
  intros [] H.  (* implictly destruct b *)
  - (* b = true *)
    unfold not in H.
    apply ex_falso_quodlibet.
      (* or use tactic exfalso. *)
    apply H. reflexivity.
  - (* b = false *)
    reflexivity.
Qed.
```

Logic.v

- 命題
- 論理結合子
  - 連言 (「かつ」)
  - 選言 (「または」)
  - 偽 (矛盾) と否定 (「〜でない」)
  - 真
  - 論理的同値 (if and only if)
  - 存在量化 (「ある x が存在して〜」)
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

真

- 真 (自明な命題) を表す命題: `True`
- 公理 `I : True`

```
Lemma True_is_true : True.
Proof. apply I. Qed.
```

「使い道なさそうですが…」→教科書に「あとで使い道がでてきます」と書いてあるけど，(少なくともこの章では) 出てこない…

異なるコンストラクタは等しくない

```
Definition disc_fn (n: nat) : Prop :=
  match n with
  | O => True | S _ => False
  end.

Theorem disc_example : forall n, ~ (O = S n).
Proof.
  intros n H1.
  assert (H2 : disc_fn O).  simpl. apply I.
  rewrite H1 in H2. simpl in H2. apply H2.
Qed.
```

(≒ discriminate が内部でやっていること)

Logic.v

- 命題
- 論理結合子
  - 連言 (「かつ」)
  - 選言 (「または」)
  - 偽 (矛盾) と否定 (「〜でない」)
  - 真
  - 論理的同値 (if and only if)
  - 存在量化 (「ある x が存在して〜」)
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

(論理的) 同値

同値 (if and only if) は，両方向の含意の連言:

```
Definition iff (P Q : Prop) :=
  (P -> Q) /\ (Q -> P).
Notation "P <-> Q" := (iff P Q)
  (at level 95, no associativity) : type_scope.
```

同値性に関する性質

対称性

```
Theorem iff_sym : forall P Q : Prop,
  (P <-> Q) -> (Q <-> P).
Proof.
  intros P Q [HPQ HQP].
  split.
  - (* -> *) apply HQP.
  - (* <- *) apply HPQ.  Qed.
Qed.
```

反射性，推移性もいえる，つまり iff は命題上の同値関係

apply で <-> 命題を使うと,「ならば」の方向をいい感じに選んでくれる.

```
Lemma apply_iff_example1:
  forall P Q R : Prop, (P <-> Q) -> (Q -> R) ->
  intros P Q R Hiff H HP. apply H.  apply Hiff.
Qed.
```

命題上の同値関係としての iff

いくつかのタクティック (`reflexivity`, `rewrite`) では iff を $=$ と同じように扱うことができる
要おまじない (ライブラリのロード):

```
Require Import Coq.Setoids.Setoid.
```

setoid ··· 同値関係が備わった集合

```
Lemma mul_eq_0 : forall n m,
  n * m = 0 <-> n = 0 \/ m = 0.
Lemma or_assoc : forall P Q R : Prop,
  P \/ (Q \/ R) <-> (P \/ Q) \/ R.

Lemma mul_eq_0_ternary : forall n m p,
  n * m * p = 0 <-> n = 0 \/ m = 0 \/ p = 0.
Proof.
  intros n m p.
  rewrite mul_eq_0. rewrite mul_eq_0.
  rewrite or_assoc.
  reflexivity.
Qed.
```

Logic.v

- 命題
- 論理結合子
  - 連言 (「かつ」)
  - 選言 (「または」)
  - 偽 (矛盾) と否定 (「〜でない」)
  - 真
  - 論理的同値 (if and only if)
  - 存在量化 (「ある x が存在して〜」)
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

存在量化 a.k.a 特称量化

$\exists x : X.P$…「型 X の要素 x が存在して P」

```
Definition Even x := exists n : nat, x = double

Lemma four_is_even: Even 4.
Proof.
  unfold Even. exists 2. reflexivity.
Qed.
```

- 「存在の証拠」(witness) を指定する `exists`
- 引き続き，その証拠が性質 P を満たすことを証明する

存在量化から何かいう

文脈に存在量化 $\exists x.P$ がある時は destruct を使う

- (正体はわからない)「存在の証拠」x と
- それが性質 P を満たす，という仮定

が得られる

```
Theorem exists_example_2 : forall n,
  (exists m, n = 4 + m) ->
  (exists o, n = 2 + o).
Proof.
  intros n H.  destruct H as [m Hm].
  (* witness に intro パターンで名前をつける *)
  (* could be "intros n [m Hm]." *)
  exists (2 + m).  apply Hm.  Qed.
```

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

命題を返す再帰的関数

「x はリスト l の要素である」ことを表す再帰的命題

```
Fixpoint In {A : Type}
           (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \/ In x l'
  end.
```

- x は第一要素と等しい，または，
- x は第二要素と等しい，または…

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
  simpl. right. right. right. left.
  reflexivity.
Qed.
```

```
Example In_example_2 : forall n,
  In n [2; 4] -> exists n', n = 2 * n'.
Proof.
  simpl.
  intros n H.  destruct H as [H1 | [H2 | []]].
  - exists 1. rewrite <- H1. reflexivity.
  - exists 2. rewrite <- H2. reflexivity.
Qed.
```

- ネストした intro パターン
- 矛盾を処理するパターン: []

```
Theorem In_map :
  forall (A B : Type)
    (f : A -> B) (l : list A) (x : A),
    In x l ->
    In (f x) (map f l).
Proof.
  intros A B f l x.
  induction l as [|x' l' IHl'].
  - (* l = nil, contradiction *)
    simpl. intros [].
  - (* l = x' :: l' *)
    simpl. intros [H | H].
    + rewrite H. left. reflexivity.
    + right. apply IHl'. apply H.
Qed.
```

再帰を使った述語定義は便利だが欠点もある

- 「明らかに止まる」再帰しか書けない
- 次の章で扱う「帰納的な述語定義」を待て
  - これにも長所・短所がある

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

証明も第一級データ

Check コマンドの挙動

```
Coq < Check 1.
1
     : nat

Coq < Check add_comm.
add_comm
     : forall n m : nat, n + m = m + n
```

- 定理の文面があたかも何かの型であるかのよう？
- 実は…
  - add_comm は「証明オブジェクト」というデータ (の名前)
  - 証明オブジェクトの型は命題

型 = 命題!?

型はデータの使い方を規定する

- `nat -> nat -> nat`
  - ふたつの `nat` を引数として与えると，`nat` が得られる
- $\forall X : Type, X \to X$
  - 型 T を引数として与えると，$T \to T$ 型の関数が得られる

命題をムリヤリ型っぽく読む

- $n = m \to n + n = m + m$
  - $n = m$ の証明を引数として与えると，
    $n + n = m + m$ の証明が得られる!?
- $\forall \mathrm{n} : nat, 2 * n = n + n$
  - 自然数，例えば 5 を引数として与えると
    $2 * 5 = 5 + 5$ の証明が得られる!?

```
Coq < Check (add_comm 3).
add_comm 3
     : forall m : nat, 3 + m = m + 3

Coq < Check (add_comm 3 5).
add_comm 3 5
     : 3 + 5 = 5 + 3
```

足し算の交換則を使った証明ふたたび

```
Lemma add_comm3_take3 :
  forall n m p, n + (m + p) = (p + m) + n.
Proof.
  intros n m p.
  rewrite add_comm.
  rewrite (add_comm m p).
  reflexivity.
Qed.
```

assert を使うよりエレガントでしょ？

(参考: assert を使った証明)

```
Lemma add_comm3_take2 :
  forall n m p, n + (m + p) = (p + m) + n.
Proof.
  intros n m p.
  rewrite add_comm.
  assert (H : m + p = p + m).
  { rewrite add_comm. reflexivity. }
  rewrite H.
  reflexivity.
Qed.
```

別の (人工的な) 例

```
Theorem in_not_nil :
  forall A (x : A) (l : list A),
    In x l -> l <> [].

(* x = 42 に特化した文言 *)
Lemma in_not_nil_42 :
  forall l : list nat, In 42 l -> l <> [].
Proof.
  intros l H.
  apply in_not_nil.  (* doesn't work! *)
```

4つの解決法

どれでも好きなものを使ってください.

```
apply in_not_nil with (x := 42).  apply H.
apply in_not_nil in H.  apply H.
apply (in_not_nil nat 42).  apply H.
apply (in_not_nil _ _ _ H).
```

最後のは型推論で紹介したのと同じ仕組み (推論されるのが型だけではないですが)

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理

命題と真偽値

性質を記述するふたつの方法: 真偽値 (`bool`) と命題 (`Prop`)
例: 自然数の等しさ $n1 =? n2$ と $n1 = n2$

	bool	*Prop*
決定可能?	yes	no
match で使える？	yes	no
rewrite で使える？	no	yes

真偽値，命題と決定可能性

- (空の文脈での) bool 型の式を計算すると必ず *true*/*false* のいずれかが得られる
- 決定不能な性質を判定する関数は書けない
- Prop 型は決定不能な性質も含む
- Prop 型の命題の真偽を判定することは (一般には) できない

決定可能な述語

例 1: n は偶数である

- $even\ n$ が $true$ を返す

```
Example even_42_bool : even 42 = true.
Proof. reflexivity. Qed.
```

- ある k が存在して，$double\ k = n$

```
Example even_42_prop : Even 42.
Proof. unfold Even. exists 21. reflexivity. Qed
```

ふた通りの表し方がある.

ふたつの表し方の等価性

```
Lemma even_double : forall k, even (double k) =
Lemma even_double_conv : forall n, exists k,
  n = if even n then double k else S (double k)
Theorem even_bool_prop : forall n,
  even n = true <-> Even n.
```

真偽値 ***even n*** は命題 ***Even n*** (の真偽) を
反映 (reflect) している，という

例 2: 自然数 n_1 と m_1 は等しい

- 関数呼出し $n_1 =? m_1$ が $true$ を返す
- $n_1 = m_1$

```
Theorem eqb_eq : forall n1 n2 : nat,
  (n1 =? n2) = true <-> n1 = n2.
```

$eqb_eq\ n1\ n2$ は命題 $n1 = n2$ を反映している

Prop は match で使えない

```
(* これはダメ！ *)
Definition is_even_prime n :=
  if n = 2 then true
  else false.
```

- Coq の関数は全て停止する (ことが要請されている)
- 一般に命題の真偽を有限時間で決定する方法はない

$\Longrightarrow$ 命題 (Prop) の真偽で条件分岐できない

決定可能な命題は常に bool で表すべき？

- 命題の定義の簡単さ $\neq$ 真偽を決定するアルゴリズムの簡単さなので話はそう単純ではない
  - e.g. 「文字列 s が，正則表現 re にマッチするかどうか」
- が，もし bool で表せるとある種の「証明の自動化」につながる (Proof by reflection)

Proof by reflection

命題を，それを反映する真偽値式に置き換えて，計算によって証明する.
ふつうの (？) 証明

```
Example even_1000 : Even 1000.
Proof. unfold Even.
 exists 500. (* まず k を見つける *)
 reflexivity. Qed.
```

reflection による証明

```
Example even_1000'' : Even 1000.
Proof. apply even_bool_prop. reflexivity. Qed.
```

一般に reflection による証明の方が，かなり単純になる

```
Example not_even_1001 : even 1001 = false.
Proof. reflexivity. Qed.
```

```
Example not_even_1001' : ~(Even 1001).
Proof. unfold Even.
  rewrite <- even_bool_prop.
  unfold not.
  simpl.
  intro H.
  discriminate H.
Qed.
```

even_bool_prop を使わない証明は難しそう…

逆に命題の “=” が嬉しい例

```
Lemma plus_eqb_example : forall n m p : nat,
    n =? m = true -> n + p =? m + p = true.
Proof.
  intros n m p H.
  rewrite eqb_eq in H.  rewrite H.
  rewrite eqb_eq.       reflexivity.
Qed.
```

- 状況によって命題の “=” と真偽値を行ったり来たりするのがミソ
- 大規模証明で役に立つテクニック (Coq の ssreflect 拡張によるサポート)

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理
  - 関数の外延性
  - 古典論理 vs. 構成的論理

集合 ≒ 述語？

- x が集合 X の要素である
  - 2 は偶数の集合の要素である
- x は性質 X を満たす (命題 $X(x)$ が成立する)

```
Definition ev (n:nat) : Prop :=
  even n = true.
Check (ev 2).
  ev 2 : Prop  (* 2 は偶数である *)
```

Coq の論理と集合論は似ているが重要な違いもある

関数の等しさ

Coqでの関数の等しさの定義

f, g : X -> Y が等しい $\overset{def}{\Longleftrightarrow}$ $f \leftrightarrow g$

- $\leftrightarrow$ は簡約による等しさ

```
Example function_equality_ex1 :
  (fun x => 3 + x) = (fun x => (pred 4) + x).
Proof. reflexivity. Qed.
```

集合論での関数の等しさの定義

$f, g : X \to Y$ が等しい $\overset{def}{\iff} \forall x \in X, f(x) = g(x)$

- 入出力の関係だけみる
- 関数の外延性 (extensionalitiy) 原理ともいう.

```
Example function_equality_ex2 :
  (fun x => x + 1) = (fun x => 1 + x).
Proof.
   (* Stuck *)
Abort.
```

外延性の公理の追加

Axiom コマンド (証明なしで使える命題の追加)

```
Axiom functional_extensionality :
  forall {X Y: Type} {f g : X -> Y},
    (forall (x:X), f x = g x) -> f = g.

Example function_equality_ex2 :
  (fun x => x + 1) = (fun x => 1 + x).
Proof.
  apply functional_extensionality. intros x.
  apply add_comm.
Qed.
```

補足

Q `rewrite add_comm` で解決できなかったの？

A `fun x` の外側から見ると，`x + 1` は (型がついた) 式ではないのでダメなのなのです (`x` が文脈にないので).

公理の追加について

- 何でもかんでも追加してよいわけではない
- 体系が矛盾する (何でも証明できる) 危険性
- 矛盾しないことを示すのは大変
- 関数の外延性公理は追加しても矛盾しないことが知られている
- `Print Assumptions 定理名.` で，定理の証明に使った公理がわかる

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- 決定可能な性質
- Coq の論理
  - 関数の外延性
  - 古典論理 vs. 構成的論理

Logic.v

- 命題
- 論理結合子
- 命題を使ったプログラム
- 定理の引数への適用
- Coq vs. 集合論
  - 関数の外延性
  - 古典論理 vs. 構成的論理

排中律

「どんな命題でもその肯定か否定のどちらかは成立する」

```
Definition excluded_middle :=
  forall P : Prop, P \/ ~ P.
```

は Coq では証明できない!

- 証明する立場に立つと，肯定・否定どちらを証明するか `left`，`right` で選ばなければいけない
- …がどちらを使えばよいかは `P` に依存するので一般にはわからない
  - これも，与えられた命題の真偽を判定する一般的な方法 (アルゴリズム) がないことと関係

限定された排中律(1)

真偽値で reflect できる場合:

```
Theorem restricted_excluded_middle :
  forall P b,
    (P <-> b = true) -> P \/ ~ P.
Proof.
  intros P b H.  destruct b.
  - left. rewrite H. reflexivity.
  - right. rewrite H.
    intros contra. destruct contra.
Qed.
```

限定された排中律(2)

等号についての排中律:

```
Theorem restricted_excluded_middle_eq :
  forall (n m : nat), n = m \/ n <> m.
Proof.
  intros n m.
  apply
    (restricted_excluded_middle (n = m)
                                (n =? m)).
  symmetry.
  apply eqb_eq.
Qed.
```

構成的論理 (constructive logic)

排中律がないおかげ・せいで,

- 存在 $(\exists x, P\ x)$ の証明ができたら,「何が P を満たすのか」を証明中に見つけ出すことができる.
  - 存在証明は「P を満たす何か」を作る (構成する) 必要がある
  - (存在証明を「存在しないわけがない」で済ませられない — 解析学の「中間値の定理」など)
- 逆に，いくつかの証明が困難 (場合によっては不可能) になる

- 排中律を認めない論理：構成的論理
- 排中律を (任意の命題に) 認める論理：古典論理 (classical logic)

構成的でない存在証明の例

a^b が有理数であるような無理数の組 a, b が存在する

(証明) $\sqrt{2}$ は無理数である．もし，$\sqrt{2}^{\sqrt{2}}$ が有理数ならば，$a = b = \sqrt{2}$ とすればよい．そうでなければ，$a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}$ とすると，

$a^b = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^2 = 2$ と，有理数になる．

(どこで排中律を使ったかわかりますか？)

構成的論理では認めない他の原理

2 重否定の除去 (狭義の背理法)

任意の命題 P について，P の否定を仮定して矛盾が導けたら P である，といってよい

```
Theorem classic_double_neg : forall P : Prop,
  ~~P -> P.
Proof.
  intros P H. unfold not in H.
  (* But now what? There is no way to
     "invent" evidence for [P]. *)
  Abort.
```

構成的論理では成立しない古典論理公理

- パース則 (Peirce's law): $((P \to Q) \to P) \to P$
- 排中律: $P \lor \neg P$
  - ただし，$\neg\neg(P \lor \neg P)$ は成立
- ド・モルガン則 (の一部):
  - $\neg(\neg P \land \neg Q) \to P \lor Q$
- $(P \to Q) \to (\neg P \lor Q)$

宿題： / 午前10:30 締切

- Exercise: `and_assoc` (2), `contrapositive` (2), `not_both_true_and_false` (1), `or_distributes_over_and` (3), `dist_exists_or` (2), `In_app_iff` (2), `logical_connectives` (2)
- 解答が記入された `Logic.v` を origin/main に push
- PandA のテスト・クイズに回答: