

# 「計算と論理」

## Software Foundations

### その8

五十嵐 淳

`cal16@fos.kuis.kyoto-u.ac.jp`

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

January 24, 2017

# 本日のメニュー

ProofObjects.v

- 証明スクリプト
- 量化子, 含意, 関数
- タクティックによるプログラミング
- 帰納的な型としての論理結合子
- 等しさ

# Show Proof コマンド

証明途中で証明オブジェクトを見る

```
Theorem ev_4'' : ev 4.
```

```
Proof.
```

```
  Show Proof.
```

```
    apply ev_SS.  Show Proof.
```

```
    apply ev_SS.  Show Proof.
```

```
    apply ev_0.   Show Proof.
```

```
Qed.
```

- ?Goal → (サブ)ゴールの証明が入る穴ボコ
- 穴がなくなったら証明完了
- Qed. で証明オブジェクトに名前をつける。

以下も、4 が偶数であることの「証明」

```
Definition ev_4'''' : ev 4 :=  
  ev_SS 2 (ev_SS 0 ev_0).
```

# 本日のメニュー

## ProofObjects.v

- 証明スクリプト
- 量化子, 含意, 関数
- タクティックによるプログラミング
- 帰納的な型としての論理結合子
- 等しさ

# 関数型 vs 含意・量化

- 関数型のデータ:
  - ▶ コンストラクタ (e.g., cons)
  - ▶ 関数 (fun)
- 量化・含意の証明:
  - ▶ コンストラクタ (e.g., ev\_SS)
  - ▶ 関数!

# 例

Theorem `ev_plus4` : forall n,  
 ev n -> ev (4 + n).

Proof.

```
intros n H. simpl.
```

```
apply ev_SS.
```

```
apply ev_SS.
```

```
apply H.
```

Qed.

`ev_plus4` の証明オブジェクトとは？

```
Definition ev_plus4' :  
  forall n, ev n -> ev (4 + n) :=  
  fun (n : nat) => fun (H : ev n) =>  
    ev_SS (S (S n)) (ev_SS n H).
```

Definition

```
ev_plus4'' (n : nat) (H : ev n) : ev (4 + n)  
  ev_SS (S (S n)) (ev_SS n H).
```

- 第2引数 H の型に第1引数が現れている! → 依存型 (dependent types)



# 含意は量子子の特殊ケース

- forall n, ev n -> ev (4 + n) は,
- forall (n:nat) (H : ev n), ev (4 + n) の略記

# 本日のメニュー

## ProofObjects.v

- 導入
- 証明スクリプト
- 量子子, 含意, 関数
- タクティックによるプログラミング
- 帰納的な型としての論理結合子
- 等しさ

証明をプログラムとして(直接)書いてよいなら、プログラムを証明のようにタクティックで書いてもよいのでは？

```
Definition add1 : nat -> nat.
```

```
  intro n.   Show Proof.
```

```
  apply S.   Show Proof.
```

```
  apply n.
```

```
Defined.
```

- 定義を `:=` を使わずに `.` で終わると、証明モードに入る
- 最後は `Qed.` ではなく `Defined.` で締めると、証明オブジェクトを(関数として)計算に使える。

# 本日のメニュー

## ProofObjects.v

- 導入
- 証明スクリプト
- 量子化子, 含意, 関数
- タクティックによるプログラミング
- 帰納的な型としての論理結合子
- 等しさ

# Coq における論理結合子

- 量化子 (と含意) 以外, 全てライブラリ定義されている
- `nat` などのデータ型を追加したのと同じように `Inductive` を使う!

# 論理結合子の追加方法

- 1 結合子の名前を決める
- 2 結合子で作られる命題が成立する条件 (導入規則) を与える
  - ▶ 導入規則  $\doteq$  証明オブジェクトのコンストラクタ!
- 3 除去規則は自動生成される

## c.f. 型の追加方法

- 1 型の名前を決める
- 2 その型に属する値の作り方 ( $\doteq$ コンストラクタの型) を決める
- 3 その型についての帰納法の原理が自動生成される

# 連言 (conjunction)

```
Inductive and (P Q : Prop) : Prop :=
```

```
  conj : P -> Q -> (and P Q).
```

```
Notation "P /\ Q" := (and P Q) : type_scope.
```

- 命題をパラメータとする命題定義
- 直観:  $\text{and } P \ Q$  ( $P \wedge Q$ ) の証拠は  $P$  の証拠と  $Q$  の証拠から ( $\text{conj}$  を付けることで) 構成される
  - ▶  $\text{conj}$  が導入規則に相当している
- 逆に  $P \wedge Q$  の証拠があれば, そこから  $P$  の証拠と  $Q$  の証拠が取り出せる
  - ▶ 除去規則相当 (定義から自動生成される)

## 参考: 直積型の定義と比較

```
Inductive and (P Q : Prop) : Prop :=  
  conj : P -> Q -> (and P Q).
```

```
Inductive prod (X Y : Type) : Type :=  
  pair : X -> Y -> (prod X Y).
```



# 交換律のプログラムっぽい証明

```
Definition and_comm_aux P Q (H : P /\ Q) :=  
  match H with  
  | conj HP HQ => conj HQ HP  
  end.
```

c.f.

```
Definition swap_pair X Y (p : X * Y) : Y * X :=  
  match p with  
  | (x,y) => (y,x)  
  end.
```

# 選言 (disjunction)

```
Inductive or (P Q : Prop) : Prop :=  
  | or_introl : P -> or P Q  
  | or_intror : Q -> or P Q.
```

Notation "P  $\vee$  Q" := (or P Q) : type\_scope.

- 直観— $\text{or } P \text{ } Q$  ( $P \vee Q$ ) の証拠を構成する方法は二通り:
  - ▶  $P$  の証拠から構成
  - ▶  $Q$  の証拠から構成

# 直和型との比較

```
Inductive or (P Q : Prop) : Prop :=  
  | or_introl : P -> or P Q  
  | or_intror : Q -> or P Q.
```

```
Inductive sum (X Y : Type) : Type :=  
  | inl : X -> sum X Y  
  | inr : Y -> sum X Y.
```

# 特称量化

$\exists x : X.P \dots$  「型  $X$  の要素  $x$  が存在して  $P$ 」

```
Inductive ex {X:Type} (P : X->Prop) : Prop :=
  ex_intro : forall (witness:X),
    P witness -> ex X P.
```

- 直観:  $\text{ex } X \ P$  の証拠は型  $X$  の要素 *witness* と  $P \ \text{witness}$  (*witness* が述語  $P$  を満たすこと) の証拠の組

```
Definition some_nat_is_even : exists n, ev n :=
  (* "exists n, ev n" expands to "ex ev" *)
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

- $P = \text{ev}$
- $\text{witness} = 4$
- $(\text{ev\_SS } 2 (\text{ev\_SS } 0 \text{ ev\_0}))$  の型は  $\text{ev } 4$

# 真

Inductive True : Prop :=  
 I : True.

- 唯一のコンストラクタ I
- True の証明はひと通り (なので, つまらない)

# 偽 (falsehood)

「偽」(⊥とも書かれる)の定義

Inductive False : Prop := .

- コンストラクタが存在しない定義!
- 偽の証明は存在しない
- 導入規則も存在しない

# 本日のメニュー

## ProofObjects.v

- 導入
- 証明スクリプト
- 量子子, 含意, 関数
- タクティックによるプログラミング
- 帰納的な型としての論理結合子
- 等しさ



# 「等しさ」の定義

(ライブラリの定義は実は少し違う)

```
Inductive eq {X:Type} : X -> X -> Prop :=  
| eq_refl : forall x, eq x x.
```

```
Notation "x = y" := (eq x y)  
                (at level 70, no associativity)  
                : type_scope.
```

- 簡約を通じて結びつく項 (例: 2 と 1 + 1) を (型・命題レベルでは) そもそも区別しない
- `eq_refl 2` の型は `eq 2 2` でもあるが, `eq (1+1) 2` でもある.

Definition four' :  $2 + 2 = 1 + 3 :=$   
eq\_refl 4.