

「計算と論理」

Software Foundations

その8

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

January 7, 2025

ふりかえって，Coqとは

- Coq の機能
 - ▶ プログラミング機能
 - ▶ 証明記述・検査機能
- 両者は別の機能のようであり、重なる部分もある
 - ▶ データ型定義と述語定義のための Inductive
 - ▶ 関数型と「ならば」のための \rightarrow
- 実は，プログラミングと証明は「コインの表裏」

カリー・Howard¹対応

$a : A$ のふたつの読み方

- a が型 A を持つ
- a は命題 A の証明である

論理の世界	計算の世界
命題	型
証明	プログラム

¹Haskell B. Curry と William A. Howard にちなむ

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 が (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).
```

(Agda² という証明支援系では、タクティックではなく、
上のようなプログラムの形式で証明を書きます。)

²<https://github.com/agda/agda>

本日のメニュー

ProofObjects.v

- 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

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nが現れている! → 依存型 (dependent types)

含意は量化子の特殊ケース

- $\text{forall } (n:\text{nat}), \text{ ev } n \rightarrow \text{ ev } (4 + n)$ は,
- $\text{forall } (n:\text{nat}) (H : \text{ ev } n), \text{ ev } (4 + n)$ の略記
 - ▶ forall 以降にその量化された変数が現れない場合は \rightarrow になる

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
 - ▶ `Qed.` で締めると `add1` は定義の中身が見えない「曇った(opaque)」定数になってしまう

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

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 (proofOfP : P) (proofOfQ : 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 (conj を付けることで) 構成される
 - ▶ 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 (proofOfP : P) (proofofQ : Q).
```

```
Inductive prod (X Y : Type) : Type :=  
  pair (x : X) (y : Y).
```

- 教科書の定義

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

はコンストラクタの型を指定するのに別構文を使っているが同じ意味.

and の交換律のプログラムっぽい証明

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

```
Definition and_comm' P Q : P /\ Q <-> Q /\ P :=
  conj (and_comm'_aux P Q) (and_comm'_aux Q P).
```

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 (proofOfP : P).  
  | or_intror (proofOfQ : Q).
```

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

- 直観— $\text{or } P \ Q$ ($P \vee Q$) 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 (proofOfP : P)  
  | or_intror (proofOfQ : Q).
```

```
Inductive sum (X Y : Type) : Type :=  
  | inl (x : X)  
  | inr (y : Y)
```

存在量化

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

```
Inductive ex {A:Type} (P : A->Prop) : Prop :=  
  ex_intro (x:A) (H:P x) : ex P.
```

- 直観: $\text{ex } P$ の証拠は
 - ① 型 A の要素 x と
 - ② $P x$ (x が述語 P を満たすこと) の証拠 H 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)).
```

- $P = \text{ev}$
- $a = 4$
- $(\text{ev_SS } 2 (\text{ev_SS } 0 \text{ ev_0}))$ の型は $\text{ev } 4$

真

```
Inductive True : Prop :=  
  I.
```

- 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ので、`=`ではなく、
違う記号 `==` を使っている)

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

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

- Coq は簡約を通じて結びつく項 (例: 2 と $1 + 1$) を、
型・命題レベルで区別しない
- `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.
```

```
Definition singleton :  
  forall (X:Type) (x:X), [] ++ [x] == x :: [] :=  
  fun (X:Type) (x:X) => eq_refl [x].
```

Coq の「信頼の基盤」

- Coq はどうして信頼できる？
 - ▶ Coq 自身にバグがあるかも
- 究極的には「証明の正しさの検査」に集約される
- できあがった証明の正しさの検査 = プログラムの型検査
- わりと単純なプログラムで実現できる
- その意味で、Coq の「信頼の基盤」はそれほど大きくない
 - ▶ タクティックは「信頼の基盤」の外側
 - ▶ タクティックのバグは最後の証明検査でわかる

型検査アルゴリズムの基本

- 「関数の引数型」と「実引数の型」の等しさのチェック
- match のパターンが同じ Inductive 型か
- match の分岐の型が全て同じか
- ⋮

型検査において少し面倒なところ

- 型の正規化
- match のパターンの網羅性チェック
- 再帰関数の停止性検査
 - ▶ 健全だが完全ではない

おかしな定理

網羅していないパターンマッチ

Definition or_bogus

: forall P Q, P \setminus / Q \rightarrow P :=

fun (P Q : Prop) (A : P \setminus / Q) =>

match A with

| or_introl H => H

end.

無限ループを使った「偽」の証明

Fixpoint infinite_loop

{X : Type} (n : nat) : X :=

infinite_loop n.

Definition falso : False := infinite_loop 0.

この本のつづき

本講義では第1巻 “Logical Foundations” の80%くらいをカバー

- 残り：第2巻への入門，自動証明機能について
- 第2巻 “Software Foundations”
 - ▶ この講義の Coq 以外の部分，を Coq を使って行う
 - ▶ つまり，他のプログラミング言語 (簡単な命令型言語，ラムダ計算) の構文・意味論を記述，その性質について色々証明する

- 第3巻 “Verified Functional Algorithms”
 - ▶ 整列, 2分探索木, 赤黒木などの基本的なアルゴリズムとその正当性証明を Coq で行う
- 第4巻 “QuickChick: Property-Based Testing in Coq”
 - ▶ 仕様に基いたランダム・テストと証明を Coq で組み合わせる

- 第5巻 “Verifiable C”
 - ▶ Cプログラムの検証を Coq で行う
- 第6巻 “Separation Logic Foundations”
 - ▶ 分離論理と呼ばれる，命令型プログラムを検証するための枠組みを Coq 上に実装する