

「計算と論理」 Software Foundations その1

五十嵐 淳

cal2014@fos.kuis.kyoto-u.ac.jp
igarashi@kuis.kyoto-u.ac.jp

京都大学

October 7, 2014

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- 書き換えによる証明
- 場合分けによる証明

Coq の基本要素 (復習)

- 数学的対象 (数, リスト, 木など) 定義とその対象を操作する **データとプログラムの記述言語**
 - ▶ Scheme のような関数型プログラミング
 - ▶ ただし静的に型がついている
 - ▶ そして文法がちょっと変わっている
- 対象に対する性質を述べる **判断の記述言語**
- 判断の **証明の記述言語**
- 証明の検査機能
- (自動証明機能)

新しい型の定義: 曜日

- 型 データの集合
- 型に属するデータの列挙による定義
 - ▶ 型: `day`
 - ▶ データ: `monday` など

```
Coq < Inductive day : Type :=  
Coq <   | monday : day  
Coq <   | tuesday : day  
Coq <   | wednesday : day  
Coq <   | thursday : day  
Coq <   | friday : day  
Coq <   | saturday : day  
Coq <   | sunday : day.
```

型定義の構文 (ver.1)

Inductive $\langle \text{型名} \rangle$: Type :=
| $\langle \text{データ名}_1 \rangle$: $\langle \text{型名} \rangle$
 :
| $\langle \text{データ名}_n \rangle$: $\langle \text{型名} \rangle$.

- 末尾のピリオド (Coq での入力終了の区切り) に注意

関数定義: 次の平日

- 場合分け (match 式) による定義
 - ▶ データの種類が7つあるので, 7通りの場合分け

```
Coq < Definition next_weekday (d:day) : day :=  
Coq <   match d with  
Coq <   | monday => tuesday  
Coq <   | tuesday => wednesday  
Coq <   | wednesday => thursday  
Coq <   | thursday => friday  
Coq <   | friday => monday  
Coq <   | saturday => monday  
Coq <   | sunday => monday  
Coq <   end.  
next_weekday is defined
```

関数定義の構文 (ver.1)

Definition $\langle \text{関数名} \rangle (\langle \text{仮引数名} \rangle : \langle \text{引数型} \rangle) : \langle \text{返値型} \rangle := \langle \text{式} \rangle$

```
 $\langle \text{式} \rangle ::= \langle \text{変数} \rangle | \langle \text{データ名} \rangle | \langle \text{match 式} \rangle$   
 $\langle \text{match 式} \rangle ::= \text{match } \langle \text{式} \rangle \text{ with}$   
    |  $\langle \text{パターン} \rangle \Rightarrow \langle \text{式} \rangle$   
     $\vdots$   
    |  $\langle \text{パターン} \rangle \Rightarrow \langle \text{式} \rangle$   
    end  
 $\langle \text{パターン} \rangle ::= \langle \text{データ名} \rangle$ 
```

プログラムの実行(式の計算)

```
Coq < Eval simpl in next_weekday friday.  
      = monday  
      : day
```

```
Coq < Eval simpl in next_weekday (next_weekday saturday).  
      = tuesday  
      : day
```


今日のメニュー

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- 書き換えによる証明
- 場合分けによる証明

言明 (判断・命題) と証明

- 言明 (判断・命題): 成立すると期待する「こと」

```
Coq < Example test_next_weekday:
```

```
Coq <   next_weekday (next_weekday saturday) = tuesday.
```

- 証明: その「こと」がなぜ成立するのかを説明したプログラム

```
Coq < Proof. simpl. reflexivity. Qed.
```

言明と証明の構文 (ver.1)

Example $\langle \text{名前} \rangle : \langle \text{命題} \rangle.$

Proof. $\langle \text{証明} \rangle$ Qed.

$\langle \text{命題} \rangle ::= \langle \text{式} \rangle = \langle \text{式} \rangle$

- ふたつの式 (の値) の等しさを述べることができる

Coq プログラムの主要な要素

- 型の定義 (Inductive)
- 関数の定義 (Definition)
- 定義に関する性質の言明とその証明 (Example)
 - ▶ その他, Theorem, Lemma など

真偽値型の定義

```
Coq < Inductive bool : Type :=  
Coq <   | true : bool  
Coq <   | false : bool.
```

真偽値関数

```
Coq < Definition negb (b:bool) : bool :=  
Coq <   match b with  
Coq <   | true => false  
Coq <   | false => true  
Coq <   end.
```

```
Coq <  
Coq < Definition orb (b1:bool) (b2:bool) : bool :=  
Coq <   match b1 with  
Coq <   | true => true  
Coq <   | false => b2  
Coq <   end.
```

orb の定義の正しさの証明

- 真理値表

```
Coq < Example test_orb1: (orb true false) = true.
```

```
Coq < Proof. simpl. reflexivity. Qed.
```

```
Coq < Example test_orb2: (orb false false) = false.
```

```
Coq < Proof. simpl. reflexivity. Qed.
```

```
Coq < Example test_orb3: (orb false true ) = true.
```

```
Coq < Proof. simpl. reflexivity. Qed.
```

```
Coq < Example test_orb4: (orb true true ) = true.
```

```
Coq < Proof. simpl. reflexivity. Qed.
```

練習問題 (nandb)

以下の *nandb* の定義を完成させ, *Example* にある *nandb* の正しさに関する言明を証明せよ.

```
Coq < Definition nandb (b1:bool) (b2:bool) : bool :=  
Coq <   admit.
```

```
Coq < Example test_nandb1: (nandb true false) = true.
```

```
Coq < Admitted.
```

具体的には, 定義右辺の `admit` をあるべき式で置き換え, 証明については, `Admitted.` を消して,

```
Proof.   simpl.   reflexivity.   Qed.
```

を書きこむ.

Check コマンド

式の型を調べる

```
Coq < Check (negb true).
```

```
negb true  
      : bool
```

```
Coq < Check negb. (* 関数の型の表記 *)
```

```
negb  
      : bool -> bool
```

```
Coq < Check orb. (* 二引数関数 *)
```

```
orb  
      : bool -> bool -> bool
```

今日のメニュー

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- 書き換えによる証明
- 場合分けによる証明

自然数 (nat 型) の定義

要素が無限にある型の定義

```
Coq < Inductive nat : Type :=  
Coq <   | 0 : nat (* 大文字のオー *)  
Coq <   | S : nat -> nat.
```

- 0 はそれだけで自然数
- S は, 自然数から自然数を作る **コンストラクタ**
 - ▶ n が自然数ならば $S\ n$ も自然数

帰納的集合 (inductively defined set)

「なにがその集合の元なのか」に関する規則を以て定義される集合

- Inductive は型 (データ集合) を帰納的に定義する
- `day`, `bool`, `nat` は帰納的型の例

自然数の集合の帰納的定義

以下のふたつの規則に従うもののみ `nat` の元である

- `0` は `nat` の元である
- `n` が `nat` の元ならば `S n` は `nat` の元である

```
Coq < Check 0.
```

```
0
```

```
  : nat
```

```
Coq < Check (S 0).
```

```
S 0
```

```
  : nat
```

```
Coq < Check (S (S 0)). (* S 0 のまわりに括弧が必要! *)
```

```
S (S 0)
```

```
  : nat
```

```
Coq < Check (S true).
```

```
Toplevel input, characters 69-73:
```

```
> Check (S true).
```

```
>           ^^^^
```

```
Error: The term "true" has type "bool"  
while it is expected to have type "nat".
```

前者関数

```
Coq < Definition pred (n : nat) : nat :=  
Coq <   match n with  
Coq <     | 0 => 0  
Coq <     | S n' => n'  
Coq <   end.  
pred is defined
```

- 適用パターン $S\ n'$: もし n が, ある式 n' に対して $S\ n'$ という形をしていたら, ...
- $\langle \text{パターン} \rangle ::= \langle \text{データ名} \rangle \mid \langle \text{データ名} \rangle \langle \text{変数} \rangle$

入れ子パターンと自然数のアラビア数字表記

```
Coq < Definition minustwo (n : nat) : nat :=
Coq <   match n with
Coq <     | 0 => 0
Coq <     | S 0 => 0
Coq <     | S (S n') => n'
Coq <   end.
minustwo is defined
Coq < Check (S (S (S (S 0)))).
4
      : nat
Coq < Eval simpl in minustwo 4.
      = 2
      : nat
```

教科書補足: 自然数の表記について

- 本当は `nat` 型は標準ライブラリで用意されている
- アラビア数字と `0`, `S` 表記は相互自動変換される

```
Coq < Check S(5).
```

```
6
```

```
    : nat
```


教科書補足: Module ... End について

Module A と End A で囲まれた部分は「箱庭」

- 箱庭の中の定義は外からそのまま見えない
 - ▶ A. をつければ見える
- 「名前空間」をわけするための機構
 - ▶ 例: 自然数の足し算, 整数の足し算, ...
- 教科書ではライブラリにある定義を何らかの理由で一時的に上書きしたい時に使っている
 - ▶ 話の途中で End が出てきている
 - ▶ 以降の話は標準ライブラリを使っており, 突然アラビア数字が使えるように見える

関数定義の構文 (ver.2)

Definition

$\langle \text{関数名} \rangle (\langle \text{仮引数名}_1 \rangle : \langle \text{引数型}_1 \rangle) \dots : \langle \text{返値型} \rangle := \langle \text{式} \rangle$

$\langle \text{式} \rangle ::= \langle \text{変数} \rangle | \langle \text{データ名} \rangle | \langle \text{式} \rangle \langle \text{式} \rangle | \langle \text{match 式} \rangle$
 $\langle \text{match 式} \rangle ::= \text{match } \langle \text{式} \rangle \text{ with}$
 $\quad | \langle \text{パターン} \rangle \Rightarrow \langle \text{式} \rangle$
 $\quad \vdots$
 $\quad | \langle \text{パターン} \rangle \Rightarrow \langle \text{式} \rangle$
 $\langle \text{パターン} \rangle ::= \langle \text{データ名} \rangle | \langle \text{変数} \rangle | \langle \text{データ名} \rangle \langle \text{パターン} \rangle$

関数とコンストラクタ

- S のような引数をとるコンストラクタは関数型を持つ

Coq < Check S.

S
: $nat \rightarrow nat$

- 関数は計算を伴う

再帰的関数定義

Definition ではなく Fixpoint を使う

```
Coq < Fixpoint evenb (n:nat) : bool :=  
Coq <   match n with  
Coq <   | 0           => true  
Coq <   | S 0         => false  
Coq <   | S (S n') => evenb n'  
Coq <   end.
```

evenb is recursively defined (decreasing on 1st argument)

```
Coq < Definition oddb (n:nat) : bool := negb (evenb n).
Coq < Example test_oddb1:      (oddb (S 0)) = true.
Coq < Proof. simpl. reflexivity. Qed.
Coq < Example test_oddb2:
Coq <      (oddb (S (S (S (S 0)))))) = false.
Coq < Proof. simpl. reflexivity. Qed.
```

複数引数の再帰関数: 足し算

```
Coq < Fixpoint plus (n : nat) (m : nat) : nat :=  
Coq <   match n with  
Coq <     | 0 => m  
Coq <     | S n' => S (plus n' m)  
Coq <   end.
```

plus is recursively defined (decreasing on 1st argument)

```
Coq < Eval simpl in plus (S (S (S 0))) (S (S 0)).  
      = 5  
      : nat
```

複数引数の再帰関数: かけ算・引き算

```
Coq < Fixpoint mult (n m : nat) : nat :=  
Coq <   match n with  
Coq <     | 0 => 0  
Coq <     | S n' => plus m (mult n' m)  
Coq <   end.
```

```
Coq < Fixpoint minus (n m:nat) : nat :=  
Coq <   match n, m with  
Coq <     | 0 , _      => 0  
Coq <     | S _ , 0    => n  
Coq <     | S n' , S m' => minus n' m'  
Coq <   end.
```

- 仮引数宣言の略記と複数の値の同時マッチング
- 何でもマッチするワイルドカードパターン (`_`)

Notation コマンド

```
Coq < Notation "x + y" :=  
Coq <   (plus x y)  
Coq <   (at level 50, left associativity)  
Coq <   : nat_scope.  
  
Coq < Check ((0 + 1) + 1).  
0 + 1 + 1  
      : nat
```

- 「記法」を定義するコマンド (おまじない)
 - ▶ 優先度, 結合の仕方, 有効範囲 (スコープ) を指定

自然数の比較 (1)

```
Coq < Fixpoint beq_nat (n m : nat) : bool :=
Coq <   match n with
Coq <     | 0 => match m with
Coq <         | 0 => true
Coq <         | S m' => false
Coq <       end
Coq <     | S n' => match m with
Coq <         | 0 => false
Coq <         | S m' => beq_nat n' m'
Coq <       end
Coq <   end.
```

自然数の比較 (2)

```
Coq < Fixpoint ble_nat (n m : nat) : bool :=
Coq <   match n with
Coq <   | 0 => true
Coq <   | S n' =>
Coq <       match m with
Coq <       | 0 => false
Coq <       | S m' => ble_nat n' m'
Coq <       end
Coq <   end.
```

今日のメニュー

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- 書き換えによる証明
- 場合分けによる証明

単純化による証明

今までに定義した関数についての性質をいろいろ証明しよう!

- 今までの Example も定理と証明の一例
 - ▶ 証明: 「両辺を計算すると等しくなる」
- もっと一般的な性質?

定理: 0 は足し算の(左)単位元

```
Coq < Theorem plus_0_n : forall n:nat, 0 + n = n.
```

```
Coq < Proof.
```

```
Coq < simpl. reflexivity. Qed.
```

- Theorem コマンド
- 全称量化子 forall: 「任意の~について」
- plus の定義を見ると第二引数 e の形に関わらず $0 + e$ は e になる
- 実は reflexivity だけでも単純化 (simpl) をしてくれる

言明の構文 (ver.2)

{Example, Theorem} \langle 名前 \rangle : \langle 命題 \rangle .

Proof. \langle 証明 \rangle Qed.

$$\begin{aligned} \langle \text{命題} \rangle &::= \langle \text{式} \rangle = \langle \text{式} \rangle \\ &| \text{forall } \langle \text{変数} \rangle : \langle \text{型} \rangle, \langle \text{命題} \rangle \end{aligned}$$

タクティック

証明記述に使う「おまじない」・証明すべき命題を変化させるコマンド

- `simpl`: 証明すべき命題中の式の単純化
- `reflexivity`: 「 $=$ の両辺は等しい . よって題意は示された .」

今日のメニュー

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- 書き換えによる証明
- 場合分けによる証明

全称量化された命題の証明と...

Theorem (任意の自然数 n について
 $0 + n = n$ である)

全称量化された命題の証明と...

Theorem (任意の自然数 n について
 $0 + n = n$ である)

n を自然数とする． $+$ の定義より， $0 + n$ は計算すると n になる．ゆえに ($=$ の反射性より)， $0 + n = n$ である． n は任意に取ったので，題意は証明された．□

- n という (名前の) 自然数の存在を仮定
 - ▶ 以降で n は，具体的な自然数 ($0, 1, \dots$) と同じような文脈で使える
- n については自然数であること以外何も仮定していないので，得られた結論は「任意の n について...」
とあってよい

...intros タクティック

仮定 (assumption) を導入するためのタクティック

- 示すべき性質が、全称量化されている時に使える
- 導入された仮定は「文脈」(context) に移動する
 - ▶ 文脈...仮定の列

```
Coq < Theorem plus_0_n'' : forall n:nat, 0 + n = n.  
1 subgoal
```

```
=====
```

```
forall n : nat, 0 + n = n
```

```
Coq < Proof.  
1 subgoal
```

```
=====
```

```
forall n : nat, 0 + n = n
```

```
Coq < (* n を仮定 (nat であることは命題から明らか) *)
```

```
Coq < intros n.
```

```
1 subgoal
```

```
  n : nat
```

```
=====
```

```
  0 + n = n
```

```
Coq < reflexivity.
```

```
No more subgoals.
```

```
Coq < Qed.
```

```
intros n.
```

```
reflexivity.
```

```
plus_0_n'' is defined
```

今日のメニュー

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- **書き換えによる証明**
- 場合分けによる証明

書き換えによる証明

定理「 n と m が等しい自然数ならば、 $n + n = m + m$ 」

```
Coq < Theorem plus_id_example : forall n m:nat,
```

```
Coq <   n = m ->
```

```
Coq <   n + n = m + m.
```

```
Coq < Proof.
```

- \rightarrow は「ならば」(含意, implication)

```
Coq < intros n m.
```

```
Coq < intros H.
```

```
1 subgoal
```

```
n : nat
```

```
m : nat
```

```
H : n = m
```

```
=====
```

```
n + n = m + m
```

- 「ならば」の証明にも仮定の導入 `intros` を使う
 - ▶ 「AならばB」は，Aが成立することを仮定して Bを示す
 - ▶ 仮定に名前をつける必要あり

```
Coq <  rewrite -> H.
```

```
1 subgoal
```

```
n : nat
```

```
m : nat
```

```
H : n = m
```

```
=====
```

```
m + m = m + m
```

- 仮定 **H** の等式の左辺から右辺へ (\rightarrow) の書き換えを施す
 - ▶ 右辺から左辺に書き換えただければ `rewrite <-` という
- あとはいつもと同じ

```
Coq <  reflexivity.  Qed.
```


ちょっとした謎

- \rightarrow が関数の型の記号だったり「ならば」だったり rewrite に使われたりするのなぜ？紛らわしい!
- `intros` を「任意の～」にも「ならば」にも使うのなぜ？紛らわしい!

実は「関数」「ならば」「任意の～」は互いに深く関係する概念なのだ!

() rewrite の \rightarrow や \leftarrow は単なる方向を示す注釈で関係ない

今日のメニュー

Basics.v

- 簡単なデータ型と関数定義
- 簡単な言明と証明
- 自然数の定義と再帰的関数定義
- 単純化による証明
- 全称量化子
- 書き換えによる証明
- 場合分けによる証明

場合分け: 単純化による証明の限界

変数を含む式は(最後まで)計算できないことがある

```
Theorem plus_1_neq_0_firsttry : forall n : nat,  
  beq_nat (n + 1) 0 = false.
```

Proof.

```
  intros n. simpl. (* does nothing! *)
```

場合分け: 単純化による証明の限界

変数を含む式は(最後まで)計算できないことがある

```
Theorem plus_1_neq_0_firsttry : forall n : nat,  
  beq_nat (n + 1) 0 = false.
```

Proof.

```
intros n. simpl. (* does nothing! *)
```

- $+$ (plus) は左側の数 (第1引数) についての場合分けで定義されているので, $n + 1$ の計算はこれ以上進まない
 - ▶ $S\ n$ と $n + 1$ の違いに注意

場合分けによる証明

n が具体的にどんな形をしようかを考えると計算が進む(場合がある)!

- ($n = 0$ の場合): $n + 1$ は単純化で 1 になる
- ($n = S(\dots)$ の場合): $n + 1$ は単純化で $S(S(\dots))$ になる

いずれの場合も, $+$ はおろか, `beq_nat` の計算も完了する!

destruct タクティックによる場合分け

```
Theorem plus_1_neq_0 : forall n : nat,  
  beq_nat (n + 1) 0 = false.
```

Proof.

```
intros n. destruct n as [| n'].  
  reflexivity. (* n = 0 の場合 *)  
  reflexivity. (* n = S(...) の場合 *)
```

Qed.

- 場合の数がふたつなのでゴールがふたつに増える
 - ▶ それぞれのサブゴールは reflexivity 一発撃破
- 場合分け対象の定義に従ってゴール中の n が変化
 - ▶ 0 の場合: $\text{beq_nat } (0 + 1) 0 = \text{false}$
 - ▶ S の場合: $\text{beq_nat } (S \ n' + 1) 0 = \text{false}$

イントロパターン

```
Theorem plus_1_neq_0 : forall n : nat,  
  beq_nat (n + 1) 0 = false.
```

Proof.

```
intros n. destruct n as [| n'].  
  reflexivity. (* n = 0 の場合 *)  
  reflexivity. (* n = S(...) の場合 *)
```

Qed.

- “...” 部分に名前をつける
 - ▶ [] 内に，変数列を | で区切って並べる
 - ▶ 変数列の数 = 場合分けの数
- 省略するとシステムが勝手に名前をつけてくれる
 - ▶ が，証明の可読性低下のもと

場合分けの原理

$P(n)$ を自然数 n の性質について述べた命題とする

自然数に関する場合分けの原理

「任意の自然数 n について $P(n)$ 」は以下と同値

- $P(0)$ かつ
 - 任意の自然数 n' について $P(S n')$
-
- 生成されるサブゴールふたつはこれらと対応
 - 二番目の命題は内容的に
任意の自然数 $n \geq S 0$ について $P(n)$
と同じであることに注意

ここまでのおさらい

- Coq ファイルの主要な要素
 - ▶ Inductive による (帰納的) データ型定義
 - ▶ Definition, Fixpoint による (再帰) 関数定義
 - ▶ Theorem, Example による命題の宣言とタクティックによる証明
- 型
- simpl, reflexivity タクティック
- 全称量化子 forall , 含意 \rightarrow と intros
- 仮定した等式による書き換え: rewrite タクティック
- 場合分けによる証明: destruct タクティック

宿題：10/21 午前10:30 締切

- Exercise の `nandb`, `andb3`, `factorial`, `blt_nat`, `plus_id_exercise`, `mult_S_1`, `zero_nbeq_plus_1`
- 解答を書き込んだ `Basics.v` をまるごとオンライン提出システムを通じて提出
- 以下をコメント欄に明記 (採点対象です):
 - ▶ 講義・演習に関する質問/要望, わかりにくいと感じたこと, その他気になること. (「特になし」はダメです.)
 - ▶ 友達に教えてもらったなら、その人の名前, 他の資料 (web など) を参考にした場合, その情報源 (URL など).

宿題のやり方

- (アカウント作成)
- 教科書の該当する章のファイルを Proof General もしくは CoqIDE で読み込む。
- 練習問題に従ってファイルを書き換える (解答を埋める)。
- ファイル全体を Coq に読み込ませエラーが出なかったら, 解答は正しい。
 - ▶ 但し Admitted. など解答を避けた部分は解いたとはみなされない。
- ファイル全体をアップロード。

Proof General のキーバインディング

C-c C-n	Next Step
C-c C-u	Undo
C-c RET	カーソル位置まで処理を進める (戻す)
C-c C-p	証明すべき命題 (ゴール) を表示
C-c C-t	既になされた証明・定義を表示