「計算と論理」 Software Foundations その1

五十嵐 淳
cal17@fos.kuis.kyoto-u.ac.jp

京都大学

October 3, 2017

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，木などのデータ) 定義とその対象を操作するプログラムの記述言語
  - OCaml, Scheme, Haskell 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
  - 文法は OCaml に近い (が微妙に違うので困る ;-)
- (対象の性質を述べる) 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 :=
        | monday : day
        | tuesday : day
        | wednesday : day
        | thursday : day
        | friday : day
        | saturday : day
        | sunday : day.
```

ピリオドや :day がいちいち冗長だけど我慢されたし.

型定義の構文 (ver.1)

```
Inductive ⟨型名⟩ : Type :=
  | ⟨データ名 1⟩ : ⟨型名⟩
        ⋮
  | ⟨データ名 n⟩ : ⟨型名⟩ .
```

- 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 :=
        match d with
        | monday => tuesday
        | tuesday => wednesday
        | wednesday => thursday
        | thursday => friday
        | friday => monday
        | saturday => monday
        | sunday => monday
        end.
next_weekday is defined
```

関数定義の構文 (ver.1)

`Definition` ⟨ 関数名 ⟩ `(`⟨ 仮引数名 ⟩ `:` ⟨ 引数型 ⟩`)` `:` ⟨ 返値型 ⟩ `:=` ⟨ 式 ⟩`.`

ただし

⟨ 式 ⟩ ::= ⟨ 変数 ⟩ | ⟨ データ名 ⟩ | ⟨match 式 ⟩

⟨match 式 ⟩ ::= `match` ⟨ 式 ⟩ `with`
| ⟨ パターン ⟩ ⇒ ⟨ 式 ⟩
⋮
| ⟨ パターン ⟩ ⇒ ⟨ 式 ⟩
`end`

⟨ パターン ⟩ ::= ⟨ データ名 ⟩

Compute コマンドによるプログラムの実行 (式の計算)

Compute ⟨式⟩. で⟨式⟩の計算をする.

```
Coq < Compute (next_weekday friday).
      = monday
      : day

Coq < Compute (next_weekday (next_weekday saturday)).
      = tuesday
      : day
```

- 関数適用の構文・括弧のつけかたは OCaml と同じ
- (一番外の括弧は不要)

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
- 計算による証明
  - 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:
        (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 ⟨名前⟩ : ⟨命題⟩.
Proof. ⟨証明⟩ Qed.

⟨命題⟩ ::= ⟨式⟩=⟨式⟩

- ふたつの式 (の値) の等しさについて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)
  - その他に Theorem, Lemma など

真偽値型の定義

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

真偽値関数

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

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

orb の定義の正しさの証明

実質，真理値表を書き下しているのと同じ

```
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.
```

- 括弧はなくてもよい

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 :=
        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
```

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
- 計算による証明
  - 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 :=
        | O : nat  (* 大文字のオー *)
        | S : nat -> nat.
```

- O 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

- `O` 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 O.
O
     : nat

Coq < Check (S O).
S O
     : nat

Coq < Check (S (S O)).  (* S O のまわりに括弧が必要! *)
S (S O)
     : 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
 "nat".
```

前者関数

```
Coq < Definition pred (n : nat) : nat :=
        match n with
          | O => O
          | S n' => n'
        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，…
- ⟨ パターン ⟩ ::= ⟨ データ名 ⟩ | ⟨ データ名 ⟩⟨ 変数 ⟩

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 :=
        match n with
          | O => O
          | S O => O
          | S (S n') => n'
        end.
minustwo is defined

Coq < Check (S (S (S (S O)))).
4
     : nat

Coq < Compute (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 O, 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
  - 名前に `A.` をつければ見える
- 「名前空間」をわけるための機構
  - 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`
⟨ 関数名 ⟩ (⟨ 仮引数名 $_1$⟩ : ⟨ 引数型 $_1$⟩) ... : ⟨ 返値型 ⟩ := ⟨ 式 ⟩.

ただし

⟨ 式 ⟩ ::= ⟨ 変数 ⟩ | ⟨ データ名 ⟩ | ⟨ 式 ⟩ ⟨ 式 ⟩ | ⟨match 式 ⟩
⟨match 式 ⟩ ::= match ⟨ 式 ⟩ with
| ⟨ パターン ⟩ ⇒ ⟨ 式 ⟩
⋮
| ⟨ パターン ⟩ ⇒ ⟨ 式 ⟩
⟨ パターン ⟩ ::= ⟨ データ名 ⟩ | ⟨ 変数 ⟩ | ⟨ データ名 ⟩ ⟨ パターン ⟩

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 -> nat
```

- 関数とコンストラクタの違い
  - 関数は引数が与えられると計算を引き起こす
  - コンストラクタは値に「タグ付け」をするだけだが,「タグ」についてパターンマッチできる

再帰的関数定義

n が偶数かどうかを判定する関数 *evenb*:

- **0** は偶数
- **1** は偶数ではない
- $n - 2$ が偶数ならば n も偶数

再帰の時は `Definition` ではなく `Fixpoint` を使う

```
Coq < Fixpoint evenb (n:nat) : bool :=
        match n with
        | O         => true
        | S O       => false
        | S (S n') => evenb n'
        end.
evenb is defined
evenb is recursively defined (decreasing on 1st argument
```

```
Coq < Definition oddb (n:nat) : bool := negb (evenb n).

Coq < Example test_oddb1:    (oddb (S O)) = true.

Coq < Proof. simpl. reflexivity.  Qed.

Coq < Example test_oddb2:
                  (oddb (S (S (S (S O))))) = 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 :=
        match n with
          | O => m
          | S n' => S (plus n' m)
        end.
plus is defined
plus is recursively defined (decreasing on 1st argument)

Coq < Compute plus (S (S (S O))) (S (S O)).
     = 5
     : nat
```

複数引数の再帰関数: かけ算・引き算

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

Coq < Fixpoint minus (n m:nat) : nat :=
        match n, m with
        | O   , _    => O
        | S _ , O    => n
        | S n', S m' => minus n' m'
        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" :=
        (plus x y)
           (at level 50, left associativity)
           : nat_scope.

Coq < Check ((0 + 1) + 1). (* plus (plus 0 1) 1 *)
0 + 1 + 1
     : nat
```

- 「記法」(マクロ) を定義するコマンド
  - 優先度 (数字が小さい方が結合が強い)，同優先度の記号の結合 (右・左) を指定

自然数の比較(1)

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

- b ··· 真偽値 (boolean) の b
- eq ··· EQuality

自然数の比較(2)

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

- le ··· “Less than or Equal”

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
- 計算による証明
  - 全称量化子
- 書き換えによる証明
- 場合分けによる証明

計算による証明

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

```
Theorem plus_O_n : forall n:nat, 0 + n = n.
```

- Theorem コマンド
- 全称量化子 forall: 「任意の〜について」
- 成立しそうな理由: plus の定義を見ると第二引数 m の形に関わらず 0 + m は m になる

言明の構文 (ver.2)

{Example,Theorem} ⟨名前⟩ : ⟨命題⟩.

ただし

⟨命題⟩ ::= ⟨式⟩=⟨式⟩
| forall ⟨変数⟩:⟨型⟩,⟨命題⟩

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のこと

- `simpl`: 証明すべき命題中の式の計算
- `reflexivity`: 「= の両辺は等しい．よって題意は示された.」
- `intros`: 文脈への仮定の導入 (次で説明)

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
- 計算による証明
  - 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 $\mathbf{0} + n = n$ 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 $(0, 1, \ldots)$ と同じ場所で使える
- 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…」といってよい

···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_O_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）*)
        intros n.
1 subgoal

  n : nat
  ============================
  0 + n = n

Coq <   simpl.
1 subgoal

  n : nat
  ============================
  n = n

Coq <   reflexivity.
No more subgoals.

Coq < Qed.
Proof
```

補足: simpl と reflexivity について

- 実は，reflexivity そのものに両辺を計算する機能が備わっている
- `simpl. reflexivity.` は `reflexivity.` に置き換え可能
- `simpl` は主に計算して「様子を伺う」のに使う (後述)

小まとめ

全称量化子の (証明論的な) 意味づけ

「任意の $x \in S$ について $P(x)$」を主張するためには,

- S の元をひとつとり (存在を仮定して), x という名前をつける
  - x については S の元であること以外, 何も仮定してはいけない
- その文脈のもとで, $P(x)$ を示す

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
- 計算による証明
  - 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,
        n = m ->
        n + n = m + m.
```

- -> は「ならば」(含意, implication)

```
Coq < Proof.

Coq <    intros n m.

Coq <    intros H.
1 subgoal

  n, 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 を示せばよい
  - 仮定に名前をつける必要あり
    - ★ H for hypothesis

仮定された等式を使ったゴールの書き換え

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

  n, m : nat
  H : n = m
  ============================
  m + m = m + m
```

- 仮定 **H** の左辺から右辺へ (->) の書き換えを施す
  - 右辺から左辺に書き換えたければ `rewrite <-`
- あとはいつもと同じ

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

rewrite タクティック

- 文脈にある等式を使って，ゴールを書き換える
  - 何箇所も一度に書き換わる
  - 書き換え箇所の制御が必要な場合あり (後述)
- 書き換えの方向を指定 (->, <-)
- `intros` で仮定した等式だけでなく，既に証明した定理を使ってもよい (教科書の `mult_0_plus` 定理)

ちょっとした謎

- `->` 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はなぜ？紛らわしい!
- `intros` を「任意の〜」にも「ならば」にも使うのはなぜ？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 `->` や `<-` は単なる方向を示す注釈で関係ない

小まとめ

「ならば」の (証明論的) 意味付け

「A ならば B」を主張するためには,

- A の成立 (A の証明の存在) を仮定し,
- その文脈のもとで B を示す

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
- 計算による証明
  - 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: 計算による証明の限界

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: 計算による証明の限界

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
  - 我々は ***n*** の形について何の知識もない!
  - (`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うるかを考えると計算が進む (場合がある)!

- (n = O の場合): n + 1 は計算で 1 になる
- (n = S(⋯) の場合): n + 1 は計算で S(S(⋯)) の形になる

いずれの場合も，+ はおろか，beq_nat の計算まで完了する!

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: `beq_nat (0 + 1) 0 = false`
  - `S` の場合: `beq_nat (S n' + 1) 0 = 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 Coq 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

“-” を使った証明の構造化

```
intros n. destruct n as [| n'].
- reflexivity.
- reflexivity.
```

destruct による場合分け後のハイフン:

- 各サブゴールの証明開始を示す記号
- あるサブゴールの証明が終わってもいないのに，次のハイフンを入れるとエラー
  - 違うサブゴールの証明が混ざるのを防止

$\Rightarrow$ 証明の可読性・メンテのしやすさの向上

入れ子の場合分け

証明によっては場合分けを何重にもすることがある

- 入れ子のレベルに応じてハイフン (“-”), プラス (“+”), アスタリスク (“*”) が使える
  - どの順序で使ってもよい
  - 同じ入れ子レベルの記号は揃える必要あり
- 中括弧 {...} で囲んでもよい

(教科書 `andb_commutative` 参照)

intros + destruct

- データの存在を仮定した直後にそれについての場合分けは頻出.
- intros にイントロパターンが使える.

nat の場合

```
intros x.  destruct x as [| y].
  ⇒ intros [| y].
```

bool の場合

```
intros b.  destruct b.
  ⇒ intros [].
```

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 $n \geq S\ O$ について $P(n)$

  と同じであることに注意

$P(b)$ を真偽値 b の性質について述べた命題とする

真偽値に関する場合分けの原理

「任意の真偽値 b について $P(b)$」は以下と同値

- $P(\mathit{true})$ かつ
- $P(\mathit{false})$

(教科書 `negb_involutive`, `andb_commutative` 参照)

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 `->` 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： / 午前10:30 締切

- Exercise の `nandb`, `andb3`, `factorial`, `blt_nat`, `plus_id_exercise`, `andb_true_elim2` (残りは随意)
- 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 Geneal 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. などで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.

- 自分でマニュアルを調べて，教科書・講義で紹介されていないタクティックを使っても構いません
  - ただし，自動証明系のタクティックは (講義では) 禁止
- 「証明の仕方がこれでよいのかわからない」
  - Coq がＯＫすればＯＫ，ですが，
  - 自分の証明が，どういう思考・推論と対応づいているかわかってない時は遅かれ早かれつまずくので，きちんと考えてみましょう

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 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