「計算と論理」
Software Foundations
その1

五十嵐 淳
igarashi@kuis.kyoto-u.ac.jp

京都大学

October 8, 2013

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

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
- **それらの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 ⟨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 :=
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 ⟨ **関数名** ⟩ (⟨ **仮引数名** ⟩ : ⟨ **引数型** ⟩) : ⟨ **返値型** ⟩ := ⟨ **式** ⟩

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

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(命題)と証明

- **言明(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)

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．

関数の型

- Check: **式の型を調べるコマンド**

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

Coq < Check negb.
negb
     : bool -> bool

Coq < Check orb.
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 <   | O : nat  (* 大文字のオー *)
Coq <   | 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 59-63:
> 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 <     | O => O
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，…**
- ⟨**パターン**⟩ ::= ⟨**データ名**⟩ | ⟨**データ名**⟩⟨**変数**⟩

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既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

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

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 ⟩

⟨ 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 <   | O         => true
Coq <   | S O       => false
Coq <   | S (S n') => evenb n'
Coq <   end.
evenb is recursively defined (decreasing on 1st argumen
```

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

Coq < Fixpoint minus (n m:nat) : nat :=
Coq <   match n, m with
Coq <   | O   , _    => O
Coq <   | S _ , O    => 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 <   | O => match m with
Coq <           | O => true
Coq <           | S m' => false
Coq <           end
Coq <   | S n' => match m with
Coq <             | O => 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 <   | O => true
Coq <   | S n' =>
Coq <       match m with
Coq <       | O => 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_O_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} ⟨名前⟩ : ⟨命題⟩.
Proof. ⟨証明⟩ Qed.

⟨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

- `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)

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，全称量化されたものの場合に使える**
- **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

  ============================
```

```
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_O_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.
```

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

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

ちょっとした謎

- `->` 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

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! *)
```

- + (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 は単純化で 1 になる
- (n = S(…) の場合): n + 1 は単純化で S(S(…)) 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**: `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

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： 10/15 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**:
  - **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

- 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.

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**