Proof Summit 2011

Coq入門 (3)

@tmiya_

February 9, 2012

Coq で何を証明するか？

「関数型言語は手続き型言語より生産性が高い」については (少なくとも参加者の皆様には) 異論は無いと思う。が、「Coq による証明駆動開発が、関数型言語でのテスト駆動開発より生産性が高いか」については一部の Coq 熟練者以外には No ということになるかと思う。
Coq を使って (= 証明の為の工数を費やして) メリットのある開発要素は、

- 高信頼性が要求されるプログラム (例えば鉄道や原子炉、医療機器などの制御系とか)
- 繰り返し共用される為、一度だけ大きな工数を払う価値が有る (コンパイラ、GC アルゴリズム、コレクションライブラリ、DSL の正しさ、など)
- そもそも定理証明系と相性が良い課題 (シリアライザやプロトコル変換の等価性証明)

ということになるだろう。近い将来、自動証明技術の発達が進めば定理証明系適用の閾は下がるが、それまでは我々が Coq に習熟する他は無い。

例：Coq のコレクションライブラリ (1)

Coq にもコレクションライブラリ (標準, user contribution) が存在し、様々なデータ構造を提供している。コレクションライブラリでは、

- 対象となるデータ構造 (例：`list`)
- そのデータ構造に対して使える関数 (例：`map`, `filter`, `length`, ...)
- そのデータ構造や関数の満たす性質 (例：`in_map`, `map_length`, ...)
- そのデータ構造に対する便利な tactic (例：`simpl_list`, `invList`, ...)

などが提供される。通常のプログラミング言語同様、将来の保守性を考え、Coq でも標準ライブラリで提供されるものは原則それを使用するのが望ましい。

例：Coq のコレクションライブラリ (2)

例えばこのような定理はリストに関する性質として必要になると思われるだろう。

```
Require Import List.
Goal forall (A:Type)(xs ys:list A), length (xs++ys) = length xs
Proof.
induction xs.
  intro ys. simpl. reflexivity.
  intro ys. simpl. erewrite IHxs. reflexivity.
Qed.
```

この定理はライブラリに app_length として予め用意されている。List ライブラリの関数や定理を再実装するのは勉強目的以外では必要無いが、自分で作成したコレクションの連結に関して同様の定理はおそらく必要になるだろう。

命題論理式をどう読むか

命題論理式も式である以上、木構造になっているが、命題論理記号の優先度に慣れるまでは、式の纏まりの単位が判りにくいかもしれない。一方で、例えばゴールに対する tactic である `intro`, `split`, `left/right` などは、ゴールの式の木の頂点の演算子に対して選択するものであるから、式を正しく把握出来る必要がある。

```
(A -> C) /\ (B -> D) /\ A /\ B -> C /\ D
--------------------------------    ------
(A -> C) /\ (B -> D) /\ A /\ B      C     D
--------    ------------------      -     -
 A     C      B -> D       A /\ B
 -     -      --------     ------
              B     D      A     B
              -     -      -     -
```

問題

```
Section ExProp.
Variable A B C D P Q R S:Prop.
Goal (P -> Q) -> (Q -> R) -> P -> R.
Goal ~False.
Goal P -> ~~P.
Goal (P -> Q) -> ~Q -> ~P.
Goal P /\ (Q /\ R) -> (P /\ Q) /\ R.
Goal P /\ (Q \/ R) -> (P /\ Q) \/ (P /\ R).
Goal P -> ~P -> Q.
Goal (A \/ B -> C) -> (A -> C)/\(B -> C).
Goal (A -> (B -> C)) /\ (A -> B) -> (A -> C).
Goal (~A /\ ~B) -> ~(A \/ (~A /\ B)).
Goal (A -> ~A) -> ~A.
Goal (~(A /\ A) -> (~A \/ ~A)).
Goal (A -> B) -> (A -> ~B) -> A -> C.
End ExProp.
```

述語論理

述語論理では命題論理に加えて、

- 「全ての $a : A$ について $P\ a$ $(\forall a : A, P\ a)$」: `forall (a:A), P a`
- 「ある $a : A$ が存在して $P\ a$ $(\exists a : A, P\ a)$」: `exists (a:A), P a`

という量化子を用いた記述が使える。
Coq では述語 `P` とは値 `a:A` に応じて命題 `P a:Prop` を返す関数 `A -> Prop` と考える。
Coq は一階述語論理だけではなく高階述語論理もサポートしているので、「全ての値 $a : A$ 」だけでは無く「全ての述語 $P : A \to \mathrm{Prop}$ 」「引数として述語を取る様な述語」なども記述出来る。
述語は例えば下記の様に定義出来る。

```
Coq < Definition iszero(n:nat):Prop :=
Coq < match n with
Coq < | 0 => True
Coq < | _ => False
Coq < end.
iszero is defined
```

∀ がある場合 (1)

ゴールに `forall` **がある場合は、** `intro(s)` **を行う。実は** `forall x:X` **は** `x:X ->` **と同じである。**

```
Coq < Theorem sample_forall : forall (X:Type)(P Q:X->Prop)(x:X),
  P x -> (forall y:X, Q y) -> (P x /\ Q x).
  ============================
   forall (X : Type) (P Q : X -> Prop) (x : X),
   P x -> (forall y : X, Q y) -> P x /\ Q x

sample_forall < intros X P Q x px Hqy.
  X : Type
  P : X -> Prop
  Q : X -> Prop
  x : X
  px : P x
  Hqy : forall y : X, Q y
  ============================
   P x /\ Q x
```

∀ がある場合 (2)

仮定に `forall y:X` がある場合は、`y` に任意の `X` 型の変数を代入したものを得る事が出来る。

```
sample_forall < split.  (* ゴールを P x と Q x とに *)
sample_forall < assumption. (* P x は仮定 px そのまま *)
1 subgoal

  X : Type
  P : X -> Prop
  Q : X -> Prop
  x : X
  px : P x
  Hqy : forall y : X, Q y
  ============================
   Q x

sample_forall < apply (Hqy x).  (* Hqy の y に x を代入 *)
Proof completed.
```

∃ がある場合 (1)

```
Coq < Theorem sample_exists : forall (P Q:nat->Prop),
Coq < (forall n, P n) -> (exists n, Q n) ->
Coq < (exists n, P n /\ Q n).

sample_exists < intros P Q Hpn Hqn.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  Hpn : forall n : nat, P n
  Hqn : exists n : nat, Q n
  ============================
   exists n : nat, P n /\ Q n
```

∃ がある場合 (2)

仮定に `exists` がある場合は、仮定に `destruct` を行う。

```
sample_exists < intros P Q Hpn Hqn.
  P : nat -> Prop
  Q : nat -> Prop
  Hpn : forall n : nat, P n
  Hqn : exists n : nat, Q n
  ============================
   exists n : nat, P n /\ Q n

sample_exists < destruct Hqn as [n' qn'].
  P : nat -> Prop
  Q : nat -> Prop
  Hpn : forall n : nat, P n
  n' : nat
  qn' : Q n'
  ============================
   exists n : nat, P n /\ Q n
```

∃ がある場合 (3)

ゴールに `exists x:X` **がある場合は、具体的な** `x:X` **を用いて** `exists x.` **を行う。**

```
sample_exists < destruct Hqn as [n' qn'].
  :
  Hpn : forall n : nat, P n
  n' : nat
  qn' : Q n'
  ============================
   exists n : nat, P n /\ Q n

sample_exists < exists n'.
  :
  Hpn : forall n : nat, P n
  n' : nat
  qn' : Q n'
  ============================
   P n' /\ Q n'  (* 以下証明してみよ *)
```

課題５：述語論理の証明

証明せよ。

```
Theorem ex5_1 : forall (A:Set)(P:A->Prop),
  (~ exists a, P a) -> (forall a, ~P a).
Theorem ex5_2 : forall (A:Set)(P Q:A->Prop),
  (exists a, P a \/ Q a) ->
  (exists a, P a) \/ (exists a, Q a).
Theorem ex5_3 : forall (A:Set)(P Q:A->Prop),
  (exists a, P a) \/ (exists a, Q a) ->
  (exists a, P a \/ Q a).
Theorem ex5_4 : forall (A:Set)(R:A->A->Prop),
  (exists x, forall y, R x y) -> (forall y, exists x, R x y).
Theorem ex5_5 : forall (A:Set)(R:A->A->Prop),
  (forall x y, R x y -> R y x) ->
  (forall x y z, R x y -> R y z -> R x z) ->
  (forall x, exists y, R x y) -> (forall x, R x x).
```

= を含む証明 (1)

最も重要な述語は値が等しい事を示す eq (= も使用可) である。

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : x = x
```

Coq では

- 型が等しい (nat と bool では駄目)
- コンストラクタが等しい (nat でも O と S n では駄目)
- コンストラクタ引数が等しい (S m と S n なら $m = n$ が必要)

の場合のみ、等号が成り立つ。
ゴールが

```
============================
 n = n
```

になったときは、apply (refl_equal n). としても良いが通常は tactic の reflexivity. を用いる。

= を含む証明 (2)

等号を含む簡単な式を証明する。`plus` の定義は `Print plus.` で確認可能。式を簡単にする為には tactic の `simpl.` を使う。

```
Coq < Theorem plus_0_l : forall n, 0 + n = n.
plus_0_l < intro n.
  n : nat
  ============================
   0 + n = n

plus_0_l < simpl.
  n : nat
  ============================
   n = n

plus_0_l < reflexivity.
Proof completed.
```

帰納法 (1)

同様に $\forall n : \mathtt{nat}, n + 0 = n$ を証明出来るだろうか？実は `simpl.` を使ってもうまくいかない。

```
  ============================
   n + 0 = n

plus_0_r < simpl.
  ============================
   n + 0 = n
```

これは `plus n m` の `n` の値で場合分けして再帰している為である。
この定理を証明する為には n に関する帰納法：

1. $n = 0$ の時、$n + 0 = n$ が成り立つ。
2. $n = n'$ の時、$n + 0 = n$ が成り立つならば、$n = \mathtt{S}\ n'$ でも $n + 0 = n$ が成り立つ。

を用いる。

帰納法 (2)

n に関する帰納法を使用する為には induction n as [|n']. または induction n. という tactic を使う。Coq 内部では次の定理 nat_ind が呼び出される。(P に現在のゴール)

```
Coq < Check nat_ind.
nat_ind : forall P : nat -> Prop, P 0 ->
  (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n
```

この nat_ind は nat のコンストラクタ 0 : nat と S : nat -> nat の形から自動的に生成される。実は Inductive を使って定義した型、例えば bool などにも bool_ind は存在する。

```
bool_ind : forall P : bool -> Prop,
  P true -> P false ->
  forall b : bool, P b
```

帰納法 (3)

induction n as [|n']. を使用すると、n が 0 と S n' の場合の証明課題が生成される。前者は reflexivity. で OK (simpl. は自動で実行)。

```
Coq < Theorem plus_0_r : forall n:nat, n + 0 = n.
plus_0_r < induction n as [|n'].
2 subgoals
  ============================
   0 + 0 = 0
subgoal 2 is:
 S n' + 0 = S n'

plus_0_r < reflexivity.
1 subgoal
  n' : nat
  IHn' : n' + 0 = n'
  ============================
   S n' + 0 = S n'
```

帰納法 (4)

n = S n' の証明課題では n = n' では成立するという仮定 IHn' が存在するので、これを使う事を考える。
S n' + 0 = S n' を証明するため simpl. を使うと plus の定義より S (n' + 0)= S n' になる。ここで IHn' を使って n' + 0 を n' に書き換えるには rewrite IHn'. と rewrite を使う。

```
  ============================
   S n' + 0 = S n'

plus_0_r < simpl.
  IHn' : n' + 0 = n'
  ============================
   S (n' + 0) = S n'

plus_0_r < rewrite IHn'.
  IHn' : n' + 0 = n'
  ============================
   S n' = S n'
```

課題6: $m + n = n + m$ の証明

コマンド `SearchAbout` **を使うと定義済みの定理を探す事が出来る。**

```
Coq < SearchAbout plus.
plus_n_O: forall n : nat, n = n + 0
plus_O_n: forall n : nat, 0 + n = n
plus_n_Sm: forall n m : nat, S (n + m) = n + S m
plus_Sn_m: forall n m : nat, S n + m = S (n + m)
mult_n_Sm: forall n m : nat, n * m + n = n * S m
```

定義済みの定理は仮定と同じ様に `rewrite` **で使用出来る。(例えば** `rewrite <- plus_n_Sm n' m.` **など。** `rewrite H.` **はゴールの中の** `H` **の左辺を右辺に書き換える。右辺を左辺に書き換える場合は** `rewrite <- H.` **)**
上記の適切な定理を用いて下記を証明せよ。

```
Theorem plus_comm : forall m n:nat, m + n = n + m.
```

帰納法 (5)

帰納法を使った証明は自然数 nat 以外の帰納型、例えば list A などにも使用する。下記の定理を証明せよ。

```
Theorem length_app : forall (A:Type)(l1 l2:list A),
  length (l1 ++ l2) = length l1 + length l2.
```

list のコンストラクタ cons は引数を２つ取るため、induction l1 as [|a l1']. などの様に２つ書く (あるいは induction l1.)。

課題 7 :リストに関する証明 (1)

List をインポートし、リストの連結 append とリストの反転 reverse を行う関数を定義する。

```
Require Import List.
Fixpoint append{A:Type}(l1 l2:list A):=
match l1 with
| nil => l2
| a::l1' => a::(append l1' l2)
end.
Fixpoint reverse{A:Type}(l:list A):=
match l with
| nil => nil
| a::l' => append (reverse l') (a::nil)
end.
```

ここで下記の定理を証明したい。

```
Theorem reverse_reverse : forall (A:Type)(l:list A),
  reverse (reverse l) = l.
```

課題 7 :リストに関する証明 (2)

下記補題を証明し、それを用いて `reverse_reverse` を証明せよ。

```
Lemma append_nil : forall (A:Type)(l:list A),
  append l nil = l.
Lemma append_assoc : forall (A:Type)(l1 l2 l3:list A),
  append (append l1 l2) l3 = append l1 (append l2 l3).
Lemma reverse_append : forall (A:Type)(l1 l2:list A),
  reverse (append l1 l2) = append (reverse l2) (reverse l1).
```

inversion (1)

induction の使用が難しい場合は、inversion tactic が便利。

```
Inductive even : nat -> Prop :=
| even_0 : even 0
| even_SS : forall n, even n -> even (S (S n)).

Theorem even_SS_inv : forall n, even (S (S n)) -> even n.
Proof.
intros n H. inversion H.

n : nat
H : even (S (S n))
n0 : nat        (* H を成立させる前提として *)
H1 : even n     (* n0, H1, H0 が生成された *)
H0 : n0 = n
______________________________________(1/1)
even n
```

仮定 H を成立させるケースが複数あれば、証明課題が複数生成される。

課題：inversion の練習 1

証明せよ。

```
Inductive odd : nat -> Prop :=
| odd_1 : odd 1
| odd_SS : forall n, odd n -> odd (S (S n)).

Theorem even_not_odd : forall n, even n -> ~odd n.
```

時間があれば下記も証明せよ。(inversion **を使うとは限らない。前の定理を後の証明で使うと良い。あるいは次の課題をやっても良い。**)

```
Theorem even_odd : forall n, even n -> odd (S n).
Theorem odd_even : forall n, odd n -> even (S n).
Theorem even_or_odd : forall n, even n \/ odd n.
Theorem odd_odd_even : forall m n,
  odd m -> odd n -> even (m + n).
```

課題：inversion の練習 2

証明せよ。(全てに inversion が必要とは限らない。)

```
Section List_inversion.
Require Import List.
Variable A:Type.
Theorem app_inv_l : forall l l1 l2:list A,
  l ++ l1 = l ++ l2 -> l1 = l2.
Check app_nil_r.
Check app_cons_not_nil.
Lemma app_cons_assoc : forall (a:A)(l1 l2:list A),
  l1 ++ a::l2 = (l1 ++ a::nil) ++ l2.
Lemma app_snoc_inv : forall (a:A)(l1 l2:list A),
  l1 ++ a::nil = l2 ++ a::nil -> l1 = l2.
Theorem app_inv_r : forall (l l1 l2:list A),
  l1 ++ l = l2 ++ l -> l1 = l2.
End List_inversion.
Check app_inv_head.
```

Section スコープの外で A はどのように扱われているか見よ。

型 = 仕様、という立場

静的型付け関数型言語では、型それ自体が関数の仕様という考え方がある。Coq においては、

- 依存型を用いることで、長さ n の vector なども型として表現出来る。
- Coq では引数に証明を用いたり、返り値に証明付きの値を用いるなどで、事前条件や事後条件を関数の型として表現出来る。
- 仕様と実装の分離に関しては

仕様	実装
値や関数の型	具体的な値や関数
公理	具体的な証明
Record 宣言, Module Type, Class	Record 値, Module, Instance

となる。(Class/Instance は次回解説予定)

関数の型で明らかではない仕様については、別に定理の形で関数の性質を証明すれば良い。

定義 = 仕様、という立場

計算方法を記述する必要のある手続き型言語と異なり、関数型言語では「関数の仕様通りコードを書けばそれが実装になる」という長所がある。例えばリストの `filter` 関数であれば、

```
Fixpoint filter (l:list A) : list A :=
match l with
| nil => nil
| x :: l => if f x then x::(filter l) else filter l
end.
```

が、関数の仕様そのままである、と考えることも出来る。
関数の定義で明らかではない仕様については、別に定理の形で関数の性質を証明すれば良い。

課題：`In` 関数の実装 (`Coq.Lists.List` 参照) を調べ、関数の仕様について考えてみよ。

帰納的定義

例えば偶数を表す述語を定義するとしよう。Coq の場合、

```
Definition even1(n:nat):Prop := exists m, 2*m = n.
Definition even2(n:nat):Prop := mod2 n = 0.
```

の用に内包的定義を使って記述することも可能である。しかし実際は下記の様に帰納的な定義を用いた方が、証明が不要/簡潔になり望ましい。

```
Inductive even:nat -> Prop :=
| even_O : even 0
| even_SS : forall n, even n -> even (S (S n)).
```

上記の様に定義することで、induction, inversion などの tactic が利用可能である。
帰納的 (inductive) な定義は

- 再帰を含まない基底となるケースの規則
- 再帰的な定義によって基底ケースから遠いケースを定義する規則

の生成規則の閉包 (基底ケース自体、あるいはそれに再帰的なケースを繰り返したもの、で構成される集合) を与える。また帰納的定義は定義を満たす条件を「場合分け」したものと言える。

決定可能な述語

前に Coq では「一般には」排中律 ($P \vee \neg P$) は使えないと述べた。しかし、命題の中には成立するか否か決定可能 (decidable) なものがある。例えば２つの `nat` が等しいか等しく無いか (`{n=m}+{~(n=m)}`) は決定可能と証明されている。

```
> Check eq_nat_dec.
eq_nat_dec
     : forall n m : nat, {n = m} + {n <> m}
```

このような定理があれば `destruct (eq_nat_dec x 3).` のように、$x = 3$ と $x \neq 3$ とを場合分けすることが出来る。
述語を定義する場合は、戻り値が bool の関数を定義するよりは、`{P x} + {~P x}` の形の戻り値を返すか、あるいは戻り値が Prop で、別途 `xxx_dec` という名前で `forall x:X, {P x} + {~P x}` の形の定理を証明するのが良い習慣である。

宿題：プログラミング課題

次の問題：
4 種類のアルファベット "A,C,G,T" から成る n 文字の文字列のうち、"AAG" という並びが含まれる文字列を全て列挙するプログラムを書きなさい。ただし、n は 3 以上の整数とし、文字列内に同じアルファベットが出現しても構わないものとし、出力順序は問わないものとします。
について、

1. 知っているプログラミング言語 (Coq か関数型言語、駄目ならばどの言語でも) で上記問題を解いて下さい。
   - 但し、正規表現ライブラリは使わないことが条件。
   - 出来る範囲で、ループではなく再帰を使ったコードを目指して下さい。
   - 単機能の小さな関数を組み合わせてプログラムを作って下さい。
2. 上記の小さな関数について、仕様を考えてみて下さい。自然言語の仕様でも構いませんが、出来れば Coq で関数の仕様を示す命題を書き下してみて下さい。

Coq チュートリアルの第４回の後半で、上記問題 (の一部？) を取り上げる予定です。

形式手法勉強会 Formal Methods Forum

Coq に限らず様々な形式手法についての勉強会を (ほぼ) 月１回、豆蔵セミナールーム (新宿) にて行っています。基本的には、参加者同士で自分が知っている話題に付いて話すという勉強会です。また Google group メーリングリストでも随時質問可能です。
Coq については、ほぼ毎回、何らかの話題について話をしています。
2010 年度は前述の *Certified Programming with Dependent Types* を皆で読みました。
まだ余り成果は多く有りませんが、Coq の証明付きの正規表現ライブラリなどを作成しました。
勉強会開催情報やメンバー間の情報交換は Google group (`http://groups.google.co.jp/group/fm-forum`) にて行っています。
参加希望の方は登録をお願いします。
参加の際は予め自分のノートパソコンに Coq を導入して参加する事をお勧めします。