Proof Summit 2011

Coq入門 (2)

@tmiya_

February 2, 2012

本 Coq 入門コースの目的

この Coq 入門コースの目的は

- Coq とは何か (何が出来るか) 知る
- Coq の操作方法について慣れる
  - Coq のコマンドや文法について知る
  - tactic を用いた Coq での証明について慣れる
- Coq を用いてのプログラミングについて知る
  - 関数型言語的なコーディングに慣れる
  - Coq を用いて簡単な定理を証明する
  - 仕様を述語論理で記述する
  - 開発したコードが仕様を満たす事を証明する
- 更なる Coq に関する情報の入手先を知る
- 証明し易さを意識してプログラミングするようになる

です。青字部分が今回の目標で、残りは第３回以降にお話しします。

自然数 nat について

自然数 nat は下記の様に定義されている。(見やすい様に整形)

```
Print nat.
>> Inductive nat : Set :=
>>   O : nat
>> | S : nat -> nat
```

O (大文字の O) が 0 を表し、S は引数に nat を取って、1 大きな nat を返す関数、と考える。この自然数の定義方法を Peano の自然数という。

```
Eval compute in (S (S (S O))).
>>      = 3
>>      : nat
```

S (S (S O)) では人間に不便な為、Coq が 3 と表示してくれている。

自然数の加法

nat の加法を定義してみよう。第 1 引数の n についての再帰関数として定義する。

```
Fixpoint add(n m:nat):nat :=
match n with
| O => m
| S n' => S (add n' m)
end.
>> add is recursively defined (decreasing on 1st argument)
```

再帰関数を定義する際はキーワード Fixpoint を用いて定義する。

```
Eval compute in (add (S (S O)) (S O)).
>>      = 3
>>      : nat
```

上記の計算過程 (call-by-value) を考えてみよ。

自然数の比較

nat の値を比較する関数を定義してみよう。_,_ の部分はどんなパターンがマッチするか、なぜこの定義で OK なのか、考えよう。

```
Fixpoint eq_nat(n m:nat):bool :=
match n,m with
| O,O => true
| S n', S m' => eq_nat n' m'
| _,_ => false
end.

Eval compute in (eq_nat 3 3).
```

同様に le_nat を定義せよ。le_nat n m は $n \leq m$ の真偽値を返す。再帰関数を定義するとき最初は末尾再帰を考えなくて良い。出来るだけ簡単かつ仕様に忠実に実装する事をまず考える (複雑なものは証明も複雑になりやすいので)。将来必要になってから末尾再帰版と非末尾再帰版が等価である事を証明すれば良い。

再帰関数の停止性★

Coq では再帰関数について停止性 (無限ループにならないこと) を保証する必要がある。実はここまで定義した再帰関数は再帰呼び出しの度にどれか一つの引数の構造が必ず小さくなることによって、再帰呼び出しの停止性を保証している。

　下記の例は明示的に第 1 引数 n の構造についての再帰と宣言した例 ({struct n} は Coq が推論するため省略可)

```
Coq < Fixpoint add'(n m:nat){struct n} :=
Coq < match n with
Coq < | O => m
Coq < | S n' => S (add' n' m)
Coq < end.
add' is recursively defined (decreasing on 1st argument)
```

例：add' 2 3 であれば、add' の第 1 引数が毎回簡単になっている。

```
  add' (S (S O)) 3
= S (add' (S O) 3)
= S (S (add' O 3)) = S (S 3) = 5.
```

チューリングマシンの停止性判定★

「停止性判定が不可能」というのは「チューリングマシンで動く全てのプログラムに対して」停止性を判定することは出来ない (そのようなアルゴリズムがあると矛盾する) ということ。
Coq では停止性を保証したプログラムしか書けない (※) (停止すると判らないプログラムは処理系が受理しない) ため、Coq のプログラムが必ず停止する事は上記と矛盾しない。(つまりチューリングマシンで許される全てのプログラムが Coq で許される訳では無い。)
同様にライスの定理から「『全てのプログラムについて』それが仕様を満たすか判定する方法は無い」ことが言えるが、Coq が証明付きで受理したプログラムにバグが無い (証明結果に反した動作をしない) ことは保証出来る。
※ Coq で書かれた C コンパイラは存在する。Coq でチューリング完全な言語を記述や検証出来ない訳では無い。

Coq 文法まとめ

■再帰関数の定義 : Fixpoint 名前 (引数:型):型 := 内容.

```
Fixpoint eq_nat(n m:nat):bool :=
match n,m with
| O,O => true
| S n', S m' => eq_nat n' m'
| _,_ => false
end.
```

■ nat を扱う基本 : n を O と S n' にパターンマッチする。

課題２：自然数の関数

1. 掛け算を行う関数 mul を add を参考に定義せよ。
2. mul を用いて階乗を計算する関数 fact を定義せよ。
   $(\mathrm{fact}(n+1) = (n+1) \times \mathrm{fact}n,\ \ \mathrm{fact}0 = 1)$
3. 引き算を行う関数 sub を定義してみよ。但し答えが負になる場合は 0 を返せば良いものとする。
4. 次の関数 div3 は何を計算する関数か考えよ。また Eval を用いて動作を確認してみよ。

```
Fixpoint div3(n:nat) :=
match n with
| S (S (S n')) => S (div3 n')
| _ => O
end.
```

注意：通常の言語では引き算を行う関数 sub から任意の割り算を行う div n m を定義するのは簡単。Coq の場合は再帰関数の停止性を保証するため整礎帰納法を用いる必要があり、今日は説明しない。

多相型とは

3引数の関数 cond c vt vf を考える。第1引数 c:bool が true なら vt を、false なら vf を返す。vt, vf の型を任意の Set の型 A として定義したい。

```
Coq < Definition cond{A:Set}(c:bool)(vt vf:A) :=
Coq < match c with
Coq < | true => vt
Coq < | false => vf
Coq < end.

Coq < Eval compute in (cond true 2 3).
     = 2 : nat
Coq < Eval compute in (cond false false true).
     = true : bool
```

{A:Set} も cond の引数だが、呼び出し時に A が推測出来るならば省略出来る事を示す。(必要なら明示的に指定可)

```
Coq < Eval compute in (@cond nat false 2 3).
```

option 型

手続き型言語では (適切な) 値が無いことを示す為に null 値などを多用するが、関数型言語では同様の目的で option 型 (Haskell では Maybe 型) を用いる。(※但し Coq では値が無い理由の証明を含めて表現可能な sumor 型を用いる方が便利な事も多い。)

```
Coq < Print option.
Inductive option (A : Type) : Type :=
  Some : A -> option A
| None : option A

Definition option_map {A B:Type} (f:A->B)(o:option A) :=
  match o with
    | Some a => Some (f a)
    | None => None
  end.
Coq < Eval compute in (option_map (fun x => x + 1) (Some 1)).
```

prod 型と sum 型★

prod A B 型は A 型と B 型の２つの値を組にしたものである。
prod A B を A * B と略記する。(x , y , .. , z) というタプルとしての表記は (pair .. (pair x y) .. z) の略記法である。

```
Coq < Check (2,true,3).
(2, true, 3) : nat * bool * nat
```

prod の第１成分、第２成分を取得する為の関数 fst, snd があるが、実際はパターンマッチで分解する事が多い。
sum A B 型は、値が A 型あるいは B 型の値のどちらかであることを示す型である。下記の様に使う。

```
Coq < Definition test_sum (s:sum nat bool) :=
Coq < match s with
Coq < | inl n => n
Coq < | inr true => 1
Coq < | inr false => 0
Coq < end.
```

prod, sum 型については Curry-Howard 対応のところで再度触れる。

List 型

多相型なデータ構造の例として List を調べる。List は標準ライブラリに定義されている。ライブラリをインポートすれば各種関数や定義が使用出来る。:: は cons を中置演算子に定義したもの。Type は Set の更に上の型 (Check Set. してみよ) である。

```
Coq < Require Import List.
Coq < Print list.
Inductive list (A : Type) : Type :=
    nil : list A
  | cons : A -> list A -> list A
Coq < Check (1::2::nil).
1 :: 2 :: nil : list nat
```

関数型言語では、再代入を避けたり、再帰呼び出しとの相性などの事情から、データ列を扱う際は配列ではなくリストを多用する。

List に対する再帰関数

List に対する再帰関数を定義する場合は、List を nil と x::xs とにパターンマッチで場合分けすることを考える。

```
Coq < Fixpoint append{A:Type}(xs ys:list A):=
Coq < match xs with
Coq < | nil => ys
Coq < | x::xs' => x::(append xs' ys)
Coq < end.
Coq < Eval compute in (append (1::2::nil) (3::4::nil)).

Coq < Fixpoint olast{A:Type}(xs:list A):option A :=
Coq < match xs with
Coq < | nil => None
Coq < | a::nil => Some a
Coq < | _::xs' => olast xs'
Coq < end.
Coq < Eval compute in (olast (1::2::3::nil)).
```

課題３：List への関数

1. リストの長さを与える関数 `len{A:Type}(xs:list A):nat` を定義せよ。`Eval compute in (len (1::2::3::nil)).` などで確認せよ。
2. `list bool` の入力を受け取り、全要素が `true` の時 `true` を返す関数 `all_true(xs:list bool):bool` を定義せよ。但し `nil` に対しては `true` を返すとせよ。
3. リストの先頭要素 `x` があれば `Some x` を、空リストに対しては `None` を返す、関数 `ohead{A:Type}(xs:list A):option A` を定義せよ。
4. 自然数 `s, n` に対して `s :: s+1 :: ... :: (s+n-1) :: nil` を返す関数 `nat_list(s n:nat):list nat` を定義せよ。
5. リストを反転する関数 `reverse{A:Type}(xs:list A):list A` を定義せよ。必要なら `append` を使え。

真理値表を使った証明

前回、Bool 型についての De Morgan 則を証明した。その際に行った事は下記の様な真理値表の全ケースについて一致する事を確かめることであった。

P	Q	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$
F	F	T	T
F	T	T	T
T	F	T	T
T	T	F	F

Bool, nat 型のような `Inductive` を用いて定義した型については、全ての値を列挙して場合分けする事が可能である。(`nat` については `O` と `S n` に列挙して場合分けしていると考える。)
しかし Coq が採用する「直観論理」という論理体系では、`Prop` に属する「命題の型」については真偽で場合分け出来ない、と考える。(実際、直観論理では上記の２つの命題が等価である事を示せない。)

排中律について

排中律というのは「どんな命題 P についても、P か $\neg P$ のどちらかは成り立つ」という (古典) 論理の公理である。通常の数学の証明では排中律を自由に使うが、プログラムを書く立場では排中律が役に立たない場合もある。排中律の説明でよく使われる例を示す。
「定理：a^b が有理数となる様な 無理数 a, b が存在する」

1. $a = b = \sqrt{2}$ とする。a, b は無理数であり、$a^b = \sqrt{2}^{\sqrt{2}}$ がもし有理数ならば証明終わり。
2. 上の a^b が無理数の場合は、$a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}$ とする。a, b は無理数であり、$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$ は有理数。

よって $\sqrt{2}^{\sqrt{2}}$ が、有理数 (P) か無理数 (~P) かのどちらかが成り立つならば、a^b が有理数となる様な 無理数 a, b が存在する。
ではこの証明は「仕様：a^b が有理数となる様な 無理数 a, b を計算する」というプログラム開発の役に立つだろうか？
直観主義論理はこの排中律を認めない立場の論理体系である。

公理と推論規則に基づく証明

直観主義論理では真理値表による証明が使えない。そこで定式化として自然推論と呼ばれる、公理と推論規則による証明を用いる。公理は「正しいと認めて良い命題」であり、推論規則は「(幾つかの) 正しい命題を組み合わせて正しい命題を導く規則」である。
推論規則の例としていわゆる三段論法の Modus Ponens と呼ばれる規則がある。命題「A」が成り立つ、命題「A ならば B である」が成り立つならば、命題「B」が成り立つ、というものである。(Γ = 仮定 (公理含む) の集合。「仮定 $\vdash$ 結論」という記法。)

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash A \to B}{\Gamma \vdash B} \ (\to \text{除去})$$

上記は (上から下の向きに考えると)$\to$ 除去規則となっている。自然演繹ではこのような推論規則を用いて公理から様々な「正しい命題」を構成する。
Coq の証明は逆にこの証明図を下から上へ辿る形で行われる。ゴールに B、仮定に `Hab : A → B` が存在する時に、`apply Hab.` という tactic を実行するとゴールが A に変化する。ここで仮定あるいは公理に `Ha : A` が存在すれば、`exact Ha.` を実行すると証明が完了する。

ゴールと仮定が一致する場合

Coq の証明で必要な tactic を示す。まず証明のゴールと同じものが、仮定の中にある場合は、tactic の `assumption.` を使用すると証明が完了する。`exact H.` あるいは `trivial.` も同じ目的に使用出来る。

```
...
H : A
...
------------------------
A
```

仮定の中に無いが既存の定理と一致する場合は `exact 定理名.` とすれば良い。
これは自然演繹の推論規則としては、

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\text{仮定})$$

に相当する。

ゴールに → を含む場合

ゴールが $A \to B$ (CUI 端末では `A -> B` と表示) の様な形をしている場合は、A を仮定に持って行く為に `intro Ha.` という tactic を用いる。
すると仮定に `Ha : A` が追加され、ゴールが `B` となる。
Coq の証明では上記の様にゴールを徐々に簡単な形に変形して進める事が多い。
これは自然演繹の推論規則としては、

$$\dfrac{\dfrac{\cdots}{\Gamma, A \vdash B}}{\Gamma \vdash A \to B}\ (\to \text{導入})$$

に相当する。
ゴールが $H_1 \to H_2 \to \cdots \to H_n \to B$ の場合は `intros H1 H2 ... Hn.` と同時に複数 intro することも出来る。また `intro.` や `intros.` とすると Coq が自動的に仮定に名前を付けてくれる。
逆の働きをする → 除去規則は、先に述べた Modus Ponens であり、対応する tactic は `apply 仮定名.` である。

例題：→ のみを含む命題 (1)

ここまで学んだ tactic だけで証明出来る例題を解いてみよう。値は「値の型の型」Set に属していた。命題 P は型であり「命題の型」Prop に属している。命題 P は型であり、 P に属する、P の証明の実体 p : P が存在する。(型階層としては Set, Prop は同じレベルで、その上に Type が存在する。)

```
  Section imp_sample.
  Variables P Q R : Prop.
  Theorem imp_sample : (P -> (Q -> R)) -> (P -> Q) -> P -> R.
1 subgoal

  ============================
   (P -> Q -> R) -> (P -> Q) -> P -> R
```

二重線 === の上が仮定、下がゴールである。このゴールを tactic を用いて変形していく。ゴールが → を含む場合は intro(s) を用いる。

例題：→ のみを含む命題 (2)

ゴールの → を消す為に intros を行う。大体の証明の最初の tactic は intro(s) である。

```
imp_sample < intros pqr pq p.
1 subgoal

  pqr : P -> Q -> R
  pq : P -> Q
  p : P
  ============================
   R
```

仮定の中で R を導けそうなものを探すと pqr が良さそうである。従って apply pqr. を入力する。

例題：→ のみを含む命題 (3)

ゴールの R を消す為に `apply pqr.` をを行う。すると `pqr` の `P -> Q ->` の部分から、ゴール `P` と `Q` が生成される。

```
imp_sample < apply pqr.
2 subgoals

  pqr : P -> Q -> R
  pq : P -> Q
  p : P
  ============================
   P

subgoal 2 is:
 Q
```

ここでゴール `P` は仮定にそのままあるので `assumption.` を実行する。

例題：→ のみを含む命題 (4)

ゴールの P を消す為に `assumption.` を行う。すると 残りのゴール `Q` が証明課題となる。

```
imp_sample < assumption.
1 subgoal

  pqr : P -> Q -> R
  pq : P -> Q
  p : P
  ============================
   Q
```

ゴール `Q` を導くにはどうすればよいか考えてみよう。

例題：→ のみを含む命題 (5)

```
imp_sample < apply pq.
1 subgoal

  pqr : P -> Q -> R
  pq : P -> Q
  p : P
  ============================
   P

imp_sample < assumption.
Proof completed.

imp_sample < Qed.
```

証明が完了したら最後に `Qed.` を入力する。

∧ を含む場合 (1)

仮定が $P \wedge Q$ という形の場合、P と Q という２つの仮定に分解したい。仮定 `pq : P /\ Q` を分解するには `destruct pq as [p q].` という tactic を使う。すると `p : P` と `q : Q` の２つの仮定に分解される。単に `destruct pq.` とした場合は `p,q` ではなく Coq が適当な名前を付ける。
ゴールが $P \wedge Q$ という形をしている場合は、P と Q の両方を示す事が出来れば $P \wedge Q$ を示せる。`split.` という tactic を用いると現在のゴール `P /\ Q` を２つのサブゴール `P, Q` に分割する。
例として次の定理を証明する。

```
Coq < Variable P Q R:Prop.
Coq < Theorem and_assoc : (P/\Q)/\R -> P/\(Q/\R).
1 subgoal

  ============================
   (P /\ Q) /\ R -> P /\ Q /\ R
```

∧ を含む場合 (2)

まず intro してゴールの → を除去し、次いで仮定の ∧ を分解する。

```
and_assoc < intro pqr.
1 subgoal

  pqr : (P /\ Q) /\ R
  ============================
   P /\ Q /\ R
and_assoc < destruct pqr as [[p q] r].
1 subgoal

  p : P
  q : Q
  r : R
  ============================
   P /\ Q /\ R
```

∧ を含む場合 (3)

次いでゴールを分解し、個々の仮定と一致したら assumption を使用する。; で tactic を連結可能で、split で生成された２つのゴールの両方に assumption を使用する。

```
and_assoc < split.
2 subgoals
  ============================
   P
subgoal 2 is:
 Q /\ R
and_assoc < assumption.
1 subgoal
  ============================
   Q /\ R
and_assoc < split; assumption.
Proof completed.
and_assoc < Qed.
```

∨ を含む場合 (1)

仮定が $P \vee Q$ という形の場合、P が成立する場合と Q が成立する場合との両方について証明をする必要がある。仮定 `pq : P \/ Q` を分解するには `destruct pq as [p|q].` という tactic を使う。すると仮定に `p : P` が含まれる場合と `q : Q` の場合の２つのサブゴールに分解される。単に `destruct pq.` とした場合は `p,q` ではなく Coq が適当な名前を付ける。
ゴールが $P \vee Q$ という形をしている場合は、P と Q のどちらかを示す事が出来れば $P \vee Q$ を示せる。`left.` あるいは `right.` という tactic を用いて証明出来そうなゴール `P` あるいは `Q` のどちらかを選択する。
例として次の定理を証明する。

```
Coq < Variable P Q R:Prop.
Coq < Theorem or_assoc : (P\/Q)\/R -> P\/(Q\/R).
1 subgoal

  ============================
   (P \/ Q) \/ R -> P \/ Q \/ R
```

∨ を含む場合 (2)

まず intro **してゴールの** → **を除去し、次いで仮定の** ∨ **を分解する。**

```
and_assoc < intro pqr.
or_assoc < destruct pqr as [[p|q]|r].
3 subgoals

  p : P
  ============================
   P \/ Q \/ R

subgoal 2 is:
 P \/ Q \/ R
subgoal 3 is:
 P \/ Q \/ R
```

∨ を含む場合 (3)

ゴールを分解し、個々の仮定と一致したら assumption **を使用。**

```
or_assoc < left.
3 subgoals
  p : P
  ============================
   P
or_assoc < assumption.
2 subgoals
  q : Q
  ============================
   P \/ Q \/ R
or_assoc < right; left.
2 subgoals
  q : Q
  ============================
   Q
```

以下、同様に証明すれば OK。

¬ を含む場合 (1)

直観論理では $\neg P$ は $P \to$ `False` にて定義される。`False` は

```
Inductive False : Prop :=
```

で定義される、値が存在しない型である。
ゴールが `~P` の場合は、`intro p.` すると仮定に `p : P` が追加され、ゴールが `False` になる。
仮定に `H : False` が存在する場合は、ゴールが何であれ `elim H.` を実行すると証明が終わる。
仮定が `np : ~P` の形をしている時に、`elim np.` を行うとゴールが何であれ `P` に変わる。
直観論理では二重否定の除去 ($\neg\neg P$ を P にする) が使えない ($\because$ 排中律と等価なので) といった制約もあり、否定の入った命題の証明は多少面倒であり、またそういう定理も後から使いにくかったりする。

¬ を含む場合 (2)

例として次の定理を証明する。

```
Coq < Theorem neg_sample : ~(P /\ ~P).
1 subgoal
  ============================
   ~ (P /\ ~ P)

neg_sample < intro.
1 subgoal
  H : P /\ ~ P
  ============================
   False
```

¬ を含む場合 (3)

```
neg_sample < destruct H as [p np].
1 subgoal
  p : P
  np : ~ P
  ============================
   False

neg_sample < elim np.
  p : P
  np : ~ P
  ============================
   P

neg_sample < assumption.
Proof completed.
```

tactic まとめ

記号	仮定 H に記号がある	ゴールに記号がある
$P \to Q$	`apply H.`	`intro.`
$P \wedge Q$	`destruct H as [p q].`	`split.`
$P \vee Q$	`destruct H as [pq].`—	`left.` か `right.`
¬	`elim H.`	`intro.`

■ゴールと同じものが仮定にある ⇒ `assumption.` あるいは `exact H.`
Curry-Howard 対応を考えると「コンパイラを型チェックを通る関数を書く」＝「証明を書く」である。複雑な関数を人間が書き下すのは困難なので、Coq は対話的証明という形でそれを支援する。

課題４：命題論理の証明

証明せよ。

```
Variable A B C D:Prop.
Theorem ex4_1 : (A -> C) /\ (B -> D) /\ A /\ B -> C /\ D.
Theorem ex4_2 : ~~~A -> ~A.
Theorem ex4_3 : (A -> B) -> ~B -> ~A.
Theorem ex4_4 : ((((A -> B) -> A) -> A) -> B) -> B.
Theorem ex4_5 : ~~(A\/~A).
```

Curry-Howard 対応 (1)

先の定理を別のやり方で証明してみよう。定理の名前を少し変えて入力する。

```
Theorem imp_sample' : (P -> (Q -> R)) -> (P -> Q) -> P -> R.
imp_sample' < intros pqr pq p.
imp_sample' < Check pq.
pq
     : P -> Q
imp_sample' < Check (pq p).
pq p
     : Q
```

命題 `pq` の型は `P -> Q` であることが判る。これは型 `P` の引数を受け取って型 `Q` の値を返す関数である。従って、`pq` に引数 `p` を与えると結果の型は `Q` になる。

Curry-Howard 対応 (2)

pqr, pq, p を使って型 R の値を作る方法を考えると下記の様になる。これを exact tactic を用いて与えると証明が完了する。

```
imp_sample' < Check (pqr p (pq p)).
pqr p (pq p)
     : R


imp_sample' < exact (pqr p (pq p)).
Proof completed.
```

これをみて判る様に証明とは、仮定を引数、ゴールを戻り値、とした関数を書くことに他ならない。実際、定理 imp_sample を Print してみると判る。

```
Coq < Print imp_sample.
imp_sample =
fun (pqr : P -> Q -> R) (pq : P -> Q) (p : P) => pqr p (pq p)
     : (P -> Q -> R) -> (P -> Q) -> P -> R
```

Curry-Howard 対応 (3)

先ほどの例を観た様に証明とプログラムの間には

証明	プログラム
命題 P	型 `P`
命題 $P \to Q$	関数 `P -> Q`
$\dfrac{\Gamma \vdash P \qquad \Gamma \vdash P \to Q}{\Gamma \vdash Q}$ ($\to$ 除去)	関数適用 `pq p`
$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}$ ($\to$ 導入)	関数 `pq (p:P):Q`
命題 $P \wedge Q$	prod 型 `(P,Q)`
命題 $P \vee Q$	sum 型 { `inl P` / `inr Q`
証明の正しさの検証	コンパイラの型チェック

という対応がある。これを Curry-Howard 対応という。
Curry-Howard 対応を考えると「コンパイラを型チェックを通る関数を書く」＝「証明を書く」である。複雑な関数を人間が書き下すのは困難なので、Coq は対話的証明という形でそれを支援する。

Curry-Howard 対応 (4)

命題論理の証明の検証であれば、Coq でなく多相型 (ジェネリクス) のある静的型付き言語であれば可能である。例えば Java だと $(P \to Q \to R) \to (P \to Q) \to P \to R$ の証明は

```
interface Fun<A,B> {
  public B apply(A a);
}
public class P {}
public class Q {}
public class R {}
public class Proof {
  public R imp_sample(Fun<P,Fun<Q,R>> pqr, Fun<P,Q> pq, P p) {
    return pqr.apply(p).apply(pq.apply(p));
  }
}
```

のコンパイルが通る事で確認出来る。(Java が得意な人は prod 型、sum 型 をどのように表現すれば良いか考えてみよ。また $\neg$ を Java でどう表現するか？)

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 < | O => 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).
```

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。