Formal Methods Forum

Coq入門 (4)

@tmiya_

February 16, 2012

= 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。

プログラミング課題

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。

ここでは「どう Coq で仕様化するか」の例題とします。

文字列の表現方法

とりあえず下記でも OK。

```
Inductive Letter:Set := A | C | G | T.
Check A::C::G::T::nil.  -->  A :: C :: G :: T :: nil : list Lett
```

ここでは String **ライブラリを使うことに。**

```
Require Import Ascii.
Require Import String.
Check "c"%char.   ----> "c"%char : ascii
Check "c"%string.   ----> "c"%string : string
Inductive string : Set :=
  EmptyString : string
| String : ascii -> string -> string.
Inductive ascii : Set :=
    Ascii : bool -> bool -> bool -> bool ->
            bool -> bool -> bool -> bool -> ascii.
```

関数：append, length, get, ...

ACGT からなる n 文字の文字列、という述語 (1)

とりあえず下記でも OK。

```
Definition word1(n:nat)(s:string):Prop :=
  (length s = n) /\
  (forall (i:nat)(c:ascii), get i s = Some c ->
    c="A"%char \/ c="C"%char \/ c="G"%char \/ c="T"%char ).
```

ACGT からなる n 文字の文字列、という述語 (2)

実装＝仕様、という立場には近い。

```
Fixpoint word2(n:nat)(s:string):Prop :=
match n,s with
| O, EmptyString => True
| S n', String "A"%char s' => word2 n' s'
| S n', String "C"%char s' => word2 n' s'
| S n', String "G"%char s' => word2 n' s'
| S n', String "T"%char s' => word2 n' s'
| _,_ => False
end.
```

ACGT からなる n 文字の文字列、という述語 (3)

帰納的な定義で証明に使いやすい。

```
Inductive Word : nat -> string -> Prop :=
| Wempty : Word 0 EmptyString
| W_A : forall n s, Word n s -> Word (S n) (String "A"%char s)
| W_C : forall n s, Word n s -> Word (S n) (String "C"%char s)
| W_G : forall n s, Word n s -> Word (S n) (String "G"%char s)
| W_T : forall n s, Word n s -> Word (S n) (String "T"%char s).
```

AAG を含む文字列、という述語 (1)

この定義を元にすると、どうやって問題を解くべきか？

```
Definition hasAAG(s:string):Prop :=
 exists s1, exists s2, s = (s1 ++ "AAG" ++ s2)%string.
```

AAG を含む文字列、という述語 (2)

帰納的な定義

```
Inductive hasAAG : list Letter -> Prop :=
| cons_aag : forall (c:Letter) s,
  hasAAG s -> hasAAG (c::s)
| aag_cons : forall s, hasAAG (A::A::G::s).
```

整礎帰納法 (1)

Fixpoint で帰納的関数を定義出来るが、その為にはある引数の構造が次第に簡単になる必要がある。(リストであれば呼び出し毎に長さが短くなるなど。) 変数の構造が簡単にならない場合は、整礎 (極小元を持つ) 関係を用いた整礎帰納法を用いる。しばしば再帰の引数から自然数への関数を一つ考え、呼び出しの度にその自然数が必ず減少する、という形で定式化する。例えばクイックソートであれば

```
Require Import Recdef.   (* 整礎帰納法に必要 *)
Function qsort(l:list A){measure length l}:list A :=
match l with
| nil => nil
| x::xs => qsort (filter (gt_A x) xs)
  ++ (x::nil) ++ qsort (filter (le_A x) xs)
end.
Proof.
(* length (filter (le_A x) xs) < length l を示す *)
(* length (filter (gt_A x) xs) < length l を示す *)
Defined.
```

整礎帰納法 (2)

Fixpoint で関数を定義すると幾つかの定理などが定義される。これらは qsort を含む証明に使える。

```
qsort_ind (* ソート前とソート後の値に関する定理の証明に *)
     : forall P : list A -> list A -> Prop,
       (forall l : list A, l = nil -> P nil nil) ->
   中略 ->
       forall l : list A, P l (qsort l)

qsort_equation (* 普通に等式になっているので rewrite で使える *)
     : forall l : list A,
       qsort l =
       match l with
       | nil => nil
       | x :: xs =>
           qsort (filter (gt_A x) xs) ++
           (x :: nil) ++ qsort (filter (le_A x) xs)
       end
```

課題：整礎帰納法

証明せよ。

```
Theorem qsort_In1 : forall (l:list A)(x:A),
  In x l -> In x (qsort l).
Proof.
intros l a. eapply qsort_ind.
(* 以下証明せよ。補題 in_or_app を使うと良い。*)
```

余裕があれば以下も証明せよ。

```
Theorem qsort_In2 : forall (l:list A)(x:A),
  In x (qsort l) -> In x l.
Theorem length_of_qsort : forall l:list A,
  length l = length (qsort l).
```

値に依存した型 (1)

Coq では型に依存した型 (= 多相型) 以外に、値に依存した型 (= 依存型) を作れる。
その例として、型として長さ情報を持つ vector 型の例を示す。

```
Require Import Bool.Bvector.
Print vector.
Inductive vector (A : Type) : nat -> Type :=
  Vnil : vector A 0
| Vcons : A -> forall n : nat, vector A n -> vector A (S n)

Check (Vcons bool true 1 (Vcons bool false 0 (Vnil bool))).
Vcons bool true 1 (Vcons bool false 0 (Vnil bool))
  : vector bool 2      (* bool 値の長さ 2 の vector *)
```

値に依存した型 (2)

vector を連結する関数。match の中で型 vector A (n0 + p) を Coq が推測出来ないので明示的に書く必要がある。(Coq 8.4 では改善されるらしい)

```
Check Vextend.      (* vector の連結用関数 *)
Vextend =
fun A : Type =>
fix Vextend (n p : nat) (v : vector A n) (w : vector A p)
  {struct v} : vector A (n + p) :=
  match v in (vector _ n0) return (vector A (n0 + p)) with
  | Vnil => w
  | Vcons a n' v' => Vcons A a (n' + p) (Vextend n' p v' w)
  end
     : forall (A : Type) (n p : nat),
       vector A n -> vector A p -> vector A (n + p)
```

関数と証明の統合 (1)

ここまで紹介した話は、値を計算する関数の書き方と、関数の戻り値が仕様を満たす証明の方法、であった。Coq ではこの２つを統合した関数を定義出来る。
例として、nat の引き算について考えよう。minus(n m:nat) は $n \leq m$ の場合は 0 を返す仕様とする事が多い。しかしその場合
$n - m + m = n$ とはならない (理由を考えよ)。
ここで次の様な関数 sub を考える。

```
Definition sub(m n:nat)(H:le n m) : {x:nat|x+n=m}.
```

この関数は、

1. 引数 H として $n \leq m$ の証明を要求する。従って $n \leq m$ と証明出来る場合しか関数を呼び出せない。
2. 戻り値はとして値 $x = m - n$ だけではなく、$x + n = m$ である証明と組になっている。(戻り値は exists x:nat, x+n=m. と考えて良い。後続の計算で証明が必要な時に使用可能。)

であり、上記のチェックは実行時ではなく「コンパイル時に」行える。

関数と証明の統合 (2)

Coq では証明モードを利用して関数を定義する事が出来る。ゴールに返り値を要求される場合は exists を用いて具体的な値を与える。証明を楽にする為に nat に関するライブラリ Arith パッケージを使用。

```
Require Import Arith.  (* Arith をインポート *)
Definition sub(m n:nat)(H:le n m) : {x:nat|x+n=m}.
  m : nat
  n : nat
  H : n <= m
  ============================
   {x : nat | x + n = m}
```

まず m, n が既に intro されて都合が悪いので generalize dependent を使って戻す。

```
sub <   generalize dependent m.
sub <   generalize dependent n.
  ============================
   forall n m : nat, n <= m -> {x : nat | x + n = m}
```

関数と証明の統合 (3)

以下、これまで同様に証明。eapply, erewrite は引数を推論する apply, rewrite で便利。最後は Qed. ではなく Defined. を使う。引数 m n と H が矛盾する場合は値を返す必要は無い。

```
induction n as [|n'].
  (* n=0. m - n = m - 0 = m. exists m する。 *)
  intros m H.  exists m.  eapply plus_0_r.
  (* n=S n' *)
  induction m as [|m']; intro H.
    (* m=0. m - n = 0 - (S n') < 0 なので矛盾を示す *)
    assert(H': ~(S n' <= 0)). eapply le_Sn_0.
    elim H'. assumption.
    (* m=S m'. S m'- S n' = m' - n'. IHn' を使う。 *)
    assert(H': n' <= m'). eapply le_S_n. assumption.
    assert(IH: {x:nat|x+n'=m'}). eapply IHn'. assumption.
    destruct IH as [x IH]. exists x.
    erewrite <- plus_n_Sm. rewrite IH. reflexivity.
Defined.
```

関数と証明の統合 (4)

定義された sub は Print sub. などを行うと証明を含む複雑な式になっている。しかし sub から OCaml コードを抽出したものは簡単なコードになっており、よく見ると minus の定義とさほど違わない事が判る。

```
Coq < Extraction sub.
(** val sub : nat -> nat -> nat **)
let rec sub m = function
  | O -> m
  | S n0 ->
      (match m with
         | O -> assert false (* absurd case *)
         | S n1 -> let iH = sub n1 n0 in Obj.magic iH)
```

OCaml 版のコードでは、仮定 H が満たされない場合は例外を投げる振る舞いとなり、戻り値の証明は単に無視される。

関数と証明の統合 (5)

実際にこの関数で計算をしてみる。引数 H に与える証明を用意してから呼び出す。

```
Theorem le_2_5 : le 2 5.
Proof. repeat eapply le_n_S. repeat constructor. Qed.

Eval compute in (sub 5 2 le_2_5).  (* 長い証明付きの値 *)
Eval compute in (proj1_sig (sub 5 2 le_2_5)).
                                                (* = 3:nat *)
```

{x|x+n=m} から実際の値 x を取り出す為には、値と証明の組の第 1 成分を取り出す関数 proj1_sig を用いる。

課題：pred 関数

以下の関数を完成させよ。

```
Definition pred'(n:nat):nat+{n=0}.
Proof.
... 証明...
Defined.

Print pred'.
Eval compute in (pred' 3).
Eval compute in (pred' 0).
Extraction pred'.
```

nat+{n=0} という型は、nat 型の値、もしくは引数 n について n=0 の証明、のどちらかの値を示す sumor 型である。このように引数に (H:n<>0) を追加する代わりに sumor 型を用いてもよい。戻り値に n=0 の証明を渡すことで、エラー処理コード内で証明を利用出来る。Coq では option 型の代わりにしばしば sumor 型を使用する。

Program コマンドを使って定義 (1)

前述のような関数は Program コマンドを使うと簡単に実装出来る。値を計算する部分だけ実装すると証明に関わる部分は Coq が生成する。

```
Program Definition sub'(m n:nat)(H:le n m):{x:nat|x+n=m}
  := m - n.
sub' has type-checked, generating 1 obligation(s)
Solving obligations automatically...
1 obligation remaining
Obligation 1 of sub':
forall m n : nat, n <= m -> m - n + n = m.
Obligation 1. (* 自動生成された証明課題 *)
1 subgoal
  A : Type
  m : nat
  n : nat
  H : n <= m
  ============================
   m - n + n = m
```

Program コマンドを使って定義 (2)

```
Require Import Omega.
omega.
Defined.
Print sub'.  (* 作成された関数 *)
(* sub' =
fun (m n : nat) (H : n <= m) =>
exist (fun x : nat => x + n = m) (m - n)
  (sub'_obligation_1 m n H)
  : forall m n : nat, n <= m -> {x : nat | x + n = m}
*)
Extraction sub'.  (* 証明から生成された OCaml コード *)
(* (** val sub' : nat -> nat -> nat **)
  let sub' m n =
    minus m n *)
```

Program コマンドを使って定義 (3)

再帰関数も Program を使って定義することも出来る。

```
Program Fixpoint div2 (n : nat) :
  { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
match n with
| S (S p) => S (div2 p)
| _ => O
end.
Solving obligations automatically...
4 obligations remaining
```

Notation の定義の仕方

Coq では各種演算子記法を定義出来る。

```
Notation "A /\ B" := (and A B)
  (at level 80. right associativity).
Notation "'IF' c1 'then' c2 'else' c3 :=
  (IF_then_else c1 c2 c3).
Notation "[ x ; .. ; y ]" :=
  (@cons _ x .. (@cons _ y (@nil _)) .. ).
```

- 詳細はマニュアル参照のこと。
- 優先度は 40(*,/,&&), 50(+,-,||), 70(等号不等号) とかを参考に、Coq.Init.Notation 参照。
- Software Foundations にパーサコンビネータの do 記法を Notation を使って書いた例がある。
  (http://www.seas.upenn.edu/ cis500/cis500-s10/sf/html/ImpParser.html)

Type Class

Coq でも Haskell 同様に ad-hoc 多相を表現する為に Type Class を使用出来る。仕様となる Class には型を指定出来る、即ちその Class の満たすべき公理を指定出来る。実装となる Instance には具体的な値や証明を書く。証明が不足している場合は証明モードに入るので最後は `Defined.` で終わる。

```
Class Monoid(S:Set) : Type := {
  op : S -> S -> S;
  e : S;
  left_id : forall x, op e x = x;
  right_id : forall x, op x e = x;
  assoc : forall x y z, op (op x y) z = op x (op y z) }.
Notation "a :+: b" := (op a b) (left associativity, at level 50)
Instance nat_monoid:Monoid nat:= {
  op := plus; e := 0 }.
Proof. (*中略*) Defined.
Eval compute in (2 :+: 3).
```

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。