Coq'Art 読書会

# Coq入門を駆け足で

@tmiya_

August 17, 2013

# 内容

今日話すことは

- Coq とは何か (何が出来るか) 知る
- Coq についての参考資料
- Coq の操作方法について慣れる
- Coq 簡単な証明をしてみる

です。

# 形式手法

ソフトウェアの検証とは「実装が仕様を満たす」事を保証すること

- 前提：仕様は曖昧さが無いように形式的記述されるべき
- Lightweight な形式手法 (例: VDM++)
  - 保証 (証明、網羅性) より手軽さを重視
  - 仕様を形式記述するだけでも効果がある
  - 仕様アニメーションにて動作確認
- モデル検査 (例: SPIN)
  - 手軽だが状態爆発を避けるノウハウが求められる場合も
  - 実装コードと別にモデル作成必要な場合も
  - 反例探しなど人間が解けない問題でも答えが得られる
  - 実際に適用可能な対象は案外限定的 (状態爆発)
- 対話的定理証明 (例: Coq)
  - 訓練が必要 $\Rightarrow$ Coq を使った証明に慣れよう
  - 証明出来る事柄には何にでも使える
  - 証明出来ない場合、大抵は問題設定 (=仕様) が間違っている。
  - 実装コード生成も可能 or 実装コードへの注釈

# Coq とは

- http://coq.inria.fr/
  - Coq 8.4 が最新
  - 実行可能バイナリ (Windows, MacOSX) か、rpm や ports など (Linux, BSD)
- LGPL ライセンスにて配布。
- 対話的定理証明の為の言語処理系。
- 理論的背景：CIC = Calculus of Inductive Construction。
- 依存型を用いた関数型プログラミングで開発可能。
- 高階述語論理を用いて仕様記述が可能。
- tactic と呼ばれる命令を用いて証明を構成する。
- 各種自動証明 tactic やライブラリを利用可能（自分で開発も可能）。
- 正しさを機械的検証可能な形で客観的に示す事が出来る。
- 証明済みの OCaml, Haskell, Scheme のプログラムを生成可能。

# Coq の使用例

- 数学、計算機科学の証明：例）四色問題の証明 (Microsoft) 他、論文での使用多数
- プロトコル検証：例）TLS の定式化と検証 (産総研)、など
- IT プランニング社の実績例：組込用 D-Bus ⇔ JSON 変換
  - 任意の D-Bus データが JSON に変換出来ることを証明
- 証明済みコンパイラ：CompCert Verified C Compiler(INRIA)
  - 生成されたコードが C プログラムと振る舞い等価であることを証明
  - 生成されたコードは gcc -O1 程度の実行速度

# 例：Monad

**型クラス Monad を定義した例。Haskell では Monad の具体的なインスタンスが Monad 則を満たすようにするのは実装者の責任で、Unit-test で確認するしか出来ない。Coq なら Monad 則を満たすことを示さないとコンパイルが通らない。**

```
Class Monad (M:Type -> Type):Type := {
  return' : forall {A}, A -> M A;
  bind : forall {A B}, M A -> (A -> M B) -> M B;
  left_unit : forall A B (a:A)(f:A -> M B),
    bind (return' a) f = f a;
  right_unit : forall A (ma:M A),
    bind ma return' = ma;
  assoc : forall A B C (ma:M A)(f:A->M B)(g:B->M C),
    bind (bind ma f) g = bind ma (fun a => bind (f a) g)
}.
```

`M:Type->Type` (**例**：M = option) **は 型** `A` **からモナド** `M A` **を作る。型クラス定義の中にはメソッドとして型を定義出来る。公理も命題なので型である。**

# Coq に適さないこと

- 有るのか無いのか判らない解を探す：Coq は具体的な解が解であることを示す、あるいは論理的帰結として解が存在しないことを示すのには使える。総当たりで解を探すのはモデル検査器の仕事。
- 人間にも解けない問題は駄目：Coq は証明の場合分けを網羅したり、証明間違いを指摘するが、考えるのは最後は人間。
- 証明出来ない時に何が悪いか判らない可能性：証明出来ないのは、自分の技量不足か解けない問題なのか判らない。大抵の場合は、仮定が弱過ぎるか結論が強過ぎるから、つまり仕様が間違っていることに気付く。

# 勉強用教材

**他にも色々ありますがメジャーどころを。**

- *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Yves Berrot and Pierre Casteran (Springer-Verlag) : **唯一の書籍。Coq の解説書。分量が多く、今となっては多少古く、高価。**
- *Certified Programming with Dependent Types*, Adam Chlipala (`http://adam.chlipala.net/cpdt/`) : **Coq を用いてプログラムを書く事を目標とした実用的な教科書。無料。良著だが** crush **に頼り過ぎるので初心者向けではないかも。**
- *Software Foundation*, Benjamin Pierce, et. al. (`http://www.cis.upenn.edu/ bcpierce/sf/`) : **Coq を通して単純型付き λ 計算と操作的意味論を学ぶ。Coq 入門用教材としては良い。旧版和訳あり** (`http://www16.atwiki.jp/sf_j/`)**。**
- **「**2010 **年度後期・数理解析・計算機数学 III」** (`http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2010_AW/index.html`) : **名古屋大学の** Garrigue **先生の授業テキスト** PDF**。無料、日本語。**2010 **年版に限らず自習教材としてお薦め。**

# Coq 標準対話環境の Coqtop, CoqIDE

CUI 対話環境 Coqtop の起動/停止方法は、

```
% coqtop
Welcome to Coq 8.4pl2 (April 2013)

Coq < Eval compute in (2+3).
      = 5
      : nat

Coq < Quit.
%
```

Coq ではコマンドの最後に必ずピリオドが必要。
CoqIDE はコマンド `coqide` あるいはアイコンから起動。[Preferences]
→ [Fonts タブ] で、日本語表示可能なフォントを選択。

# Proof General : emacs 上の対話環境

Proof General をインストールした後に、`~/.emacs` に

```
(load-file "***/ProofGeneral/generic/proof-site.el")
```

のように、`proof-site.el` へのパスを設定。
Coq ファイル (`*.v`) を開くかセーブすると Proof General が起動。

以下、CoqIDE か Proof General を使用するとして説明。“>> ” で始まる行は出力域に表示される内容のつもり。

# Hand (1)

coqart_20130817.v を CoqIDE か ProofGeneral で開く。後は矢印で進める。

```
(* ３つの値を持つ型を定義 *)
Inductive Hand : Set := Gu | Tyoki | Pa.
Check Hands. (* Hands の型は Set *)
Print Hands. (* Hands の定義の値を調べる *)
Print Hands_ind. (* Hands に対する帰納法 *)
(* ２引数の述語を定義 *)
Inductive win : Hand -> Hand -> Prop :=
| gu_win_tyoki : win Gu Tyoki
| tyoki_win_pa : win Tyoki Pa
| pa_win_gu : win Pa Gu.
Hint Constructors win. (* 自動証明用に登録 *)
```

# Hand (2)

簡単な定理を証明してみる。

```
(* ある手に勝つ手は唯一である。
手 c に手 a,b が勝つならば、a=b である。 *)
Theorem winning_hand_unique : forall a b c,
  win a c -> win b c -> a = b.
Proof.
intros a b c Hac HBc. (* 変数や前提条件を仮定に移す *)
induction c. (* c に対する帰納法 *)
 inversion Hac. (* Hac を満たす条件を win の定義から探させる *)
 inversion Hbc. (* 同様 *)
 reflexivity. (* 右辺と左辺が等しい *)

 inversion Hac. inversion Hbc. auto.
inversion H; inversion H0; auto.
Qed.
```

# Hand (3)

依存型プログラミングの例。関数は値を返すだけでなく、値と証明の組を返すことが可能。

```
(* 引数の手に対して、勝てる手とその証明の組を返す *)
Definition winning_hand(s:Hand) : { a | win a s }.
Proof.  (* 関数の定義を証明モードで開始する *)
induction s.  (* s についての場合分け *)
 exists Pa.   (* Gu に勝つ手を具体的に構成する *)
 apply pa_win_gu.   (* 既にある定理 pa_win_gu を適用 *)

 exists Gu. auto.   (* 後半は自動化 *)

 exists Tyoki; auto.   (* 一気に処理 *)
Defined.  (* 関数を定義した *)
```

この関数は Coq 内で評価出来る。また OCaml などのプログラムとして出力出来る。

# 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 の更に上の型である。

```
Require Import List.  (* List モジュールのインポート *)
Print list. (* list 型の定義を確認 *)

Inductive list (A : Type) : Type := (* A 型の要素のリスト *)
    nil : list A                    (* 空リスト *)
  | cons : A -> list A -> list A    (* cons セル *)

Check (1::2::nil). (* 要素の例 *)
1 :: 2 :: nil : list nat

Check list_ind. (* list に対する帰納法 *)
```

# 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。

```
Fixpoint append{A:Type}(xs ys:list A):=  (* 再帰関数の定義 *)
match xs with                          (* 第１引数に対してパターンマッチ *)
| nil => ys                                  (* 空なら ys そのまま *)
| x::xs' => x::(append xs' ys)  (* 自分を再帰的に呼び出す *)
end.                                   (* match は end で閉じる *)

(* 動作確認 *)
Eval compute in (append (1::2::nil) (3::4::nil)).
```

第１引数 xs が再帰呼び出しの度に１つづつ小さくなっている。それ故に、append 関数はいつか停止する。Coq では必ず停止性を示さねばならない。

# 余談：チューリングマシンの停止性判定と Coq

「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 はチューリング完全であることを捨てた代わりに色々なものを得ている。
※ 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。

# 自然数

Coq では自然数を nat 型という「Peano の自然数」としてエンコードしている。

```
Inductive nat : Set :=  (* nat 型の定義 *)
  O : nat               (* アルファベットの O。ゼロを表す *)
| S : nat -> nat.        (* 引数に+1 された数を表す *)

(* 足し算。n について帰納的定義 *)
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O => m (* O+m => m *)
| S p => S (plus p m)  (* (S p)+m => S (p+m) *)
end.
```

# 定理証明

```
Theorem len_append: forall (A:Type)(xs ys:list A),
  len (append xs ys) = plus (len xs) (len ys).
Proof.
(* "intro A." で A を仮定に移します（この行省略可） *)
(* "induction xs."で xs に対する帰納法 *)
 (* 1: xs=nil の場合 *)
 (* "simpl."で式を簡単化 *)
 (* "intro. auto."で証明を終わる *)

 (* 2: xs で成り立つと仮定して、a::xs の場合 *)
 (* "intro ys. simpl."する *)
 (* "erewrite IHex."で IHxs を使って書き換え *)
 (* "auto." *)
Qed.
```

# 実習課題

**リストを逆順にする関数**

```
Fixpoint reverse{A:Type}(xs:list A):list A :=
match xs with
| nil => nil
| x::xs' => append (reverse xs') (x::nil)
end.
```

**を考えます。(末尾再帰に書かれてないが、まずは一番簡単な実装で証明。)**
**目標は下記の定理の証明です。**

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

# 実習課題

この定理を証明するには先に下記の補題を証明する必要があります。

```
Lemma append_right_nil: forall (A:Type)(xs:list A),
  append xs nil = xs.
Lemma append_append : forall (A:Type)(xs ys zs:list A),
  append (append xs ys) zs = append xs (append ys zs).
Lemma reverse_append : forall (A:Type)(xs ys:list A),
  reverse (append xs ys) = append (reverse ys) (reverse xs).
```

どれもここまでに説明した tactic だけで証明出来ます。順に証明して目標の定理を証明しましょう。
＃実際の問題を解く時は、目標の定理の証明を行うと、足りない補題に気付くのですが。

# まとめ

- Coq は定理証明系。依存型を使える関数型言語で、高階述語論理を用いて仕様を記述し、直観主義論理に基づいた証明を行うことが可能。
- Coq を学ぶ為の教材を紹介した。
- Coq の使用方法について習熟した。
- 簡単な定理証明を行った。(list について、今日定義した関数や証明した定理は、既に別名でライブラリに存在します。今回は証明の練習です。)