Formal Methods Forum

Coq入門 (1)

@tmiya_

Janauary 10, 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
  - 問題の分割, 再帰の使用や副作用の回避、証明付きの値の使用、...
  - 証明出来ない場合、仕様の間違いに気付く

です。青字部分が今回の目標で、残りは第２回以降にお話しします。

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 8.3 が最新 (2012/1)
  - 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
- 証明済みコンパイラ：CompCert Verified C Compiler
  - 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

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。

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 8.3pl1 (December 2010)

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。[Edit] → [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。

定義の方法と定義の確認方法

tutorial1.v を CoqIDE か ProofGeneral で開く。後は矢印で進める。

```
Definition x := 1.          (* x を定義 *)
>> x is defined             (* 定義は := を用いる *)
Check x.                    (* x の型を調べる *)
>> x
>>      : nat                (* 型は nat 自然数 *)
Coq < Print x.              (* x の定義の値を調べる *)
>> x = 1
>>      : nat
Definition x := 2.              (* x を再定義 *)
>> Error: x already exists      (* --> 再定義出来ない *)
```

名前の衝突を避ける手段として Module (本入門コース範囲外) などがある。

関数の定義の方法

関数の定義や適用にはカッコを必要とせず、「関数名 引数 1 引数 2」などのように書く。

```
Definition f x y := x - y. (* 関数 f を定義 *)
>> f is defined
Check f.
>> f
>>      : nat -> nat -> nat      (* nat->(nat->nat) *)
Definition f' := f 3.      (* f' y = f 3 y *)
>> f' is defined
Check f'.
>> f'
>>      : nat -> nat      (* nat を与えると nat を返す *)
Eval compute in (f' 1).    (* f' 1 = f 3 1 = 2 *)
>>      = 2
>>      : nat
```

多引数関数に一部の変数を渡すと部分適用されて、戻り値は関数を返す関数になる。

無名関数の定義方法★

関数を無名関数として定義することも出来る。

```
Check (fun x => 2*x).                    (* 無名関数 *)
>> fun x : nat => 2 * x
>>       : nat -> nat

Eval compute in ((fun x => 2*x) 3).
>>      = 6
>>      : nat

Definition double := (fun x => 2*x).
>> double is defined          (* 無名関数で double を定義 *)
```

高階関数の定義方法

関数を引数に取る関数を高階関数という。

```
Definition twice(f:nat->nat):nat->nat :=
  fun x => f (f x).
>> twice is defined

Definition add5(x:nat) := x + 5.
>> add5 is defined

Definition twice_add5 := twice add5.
>> twice_add5 is defined

Eval compute in (twice_add5 2).
>>      = 12
>>      : nat
```

Coq 文法まとめ

■定義 : Definition 名前:型 := 内容.

```
Definition x := 1.
Definition f x y := x - y.
Definition twice(f:nat->nat):nat->nat :=
  fun x => f (f x).
```

■型を調べる : Check 名前.

```
Check twice.
```

■定義を表示する : Print 名前.

```
Print twice.
```

■式を計算して答えを表示 : Eval compute in (式).

```
Eval compute in ((fun x => 2*x) 3).
```

■型とか

nat : 自然数を表す型

nat -> nat : 自然数を受け取って自然数を返す関数の型

ユーザ定義の型

多くのプログラミング言語では int, float などがデフォルトで用意されているが Coq では自然数 nat などもライブラリとして定義されている。(よく使われるものは起動時に読み込まれる。)
ユーザ定義型 Weekday を定義してみる。

```
Inductive Weekday : Set :=
  Sun | Mon | Tue | Wed | Thr | Fri | Sat.

Check Sun.
>> Sun                (* Sun の型は *)
>>       : Weekday    (* Weekday *)

Check Weekday.
>> Weekday            (* 型の型は *)
>>       : Set        (* Set *)
```

値を表す型の型は Set に属する。

ユーザ定義の型への関数

関数定義の場合分けをする場合はパターンマッチ構文が使える。パターンマッチは全ケースを網羅しないとエラーになる。(網羅性によってバグが減る。)

```
Definition nextday d :=
match d with
| Sun => Mon
       : (* 中略 *)
| Sat => Sun
end.
>> nextday is defined
Check nextday.
>> nextday  (* 引数、戻り値の型を推論 *)
>>        : Weekday -> Weekday
Eval compute in (nextday Mon).
(* 結果を試してみよ *)
```

同様に prevday を定義してみよ。

Bool を定義してみる

```
Inductive Bool : Set :=
| tru : Bool
| fls : Bool.

Definition And(b1 b2:Bool):Bool :=
match b1,b2 with
| tru,tru => tru
| _,_ => fls
end.
```

パターンマッチは上から順に合致するか検査される。_ は全ての値がマッチする。
同様にして `Or`, `Not` も定義し、結果を `Eval` を用いて確認せよ。

ライブラリの bool 型

標準ライブラリには (当然ではあるが) `bool` という型が予め定義されている。下記コマンドを入力して確認してみよ。

```
Coq < Print bool.
Coq < Print andb.
Coq < Print orb.
Coq < Print negb.
```

`bool` には値が２通り存在したが、値が１つしかない型に意味はあるか？下記について試し、用途について考えてみよ。

```
Coq < Print unit.
```

Coq 文法まとめ

■ユーザ定義型の定義 : Inductive 型名 : Set := 値1 | ... | 値n .

```
Inductive Bool : Set :=
| tru : Bool
| fls : Bool.
```

■パターンマッチ式 : match 式 with | パターン => 値 | ... end

```
Definition And(b1 b2:Bool):Bool :=
match b1,b2 with
| tru,tru => tru
| _,_ => fls
end.
```

■型とか

Set : 値を表す型の型（階層が1つ上）

bool : true と false を持つブール値の型

andb, orb, negb : bool についての関数

unit : tt という1つの値を持つ型（あまり使わない）

De Morgan 則を証明してみる (1)★

ここまで定理証明の話をしなかったが簡単な定理を証明してみる。証明の操作の詳細は今は判らなくても良い。

```
Coq < Theorem De_Morgan_1 : forall b1 b2,
Coq < Not (And b1 b2) = Or (Not b1) (Not b2).
1 subgoal

  ============================
   forall b1 b2 : Bool,
   Not (And b1 b2) = Or (Not b1) (Not b2)

De_Morgan_1 < intros.
1 subgoal  (* b1 b2 を仮定に移した *)

  b1 : Bool
  b2 : Bool
  ============================
   Not (And b1 b2) = Or (Not b1) (Not b2)
```

De Morgan 則を証明してみる (2)★

```
De_Morgan_1 < destruct b1; destruct b2.
4 subgoals  (* b1 b2 について総当たりの場合分け *)

  ============================
   Not (And tru tru) = Or (Not tru) (Not tru)

subgoal 2 is:
 Not (And tru fls) = Or (Not tru) (Not fls)
subgoal 3 is:
 Not (And fls tru) = Or (Not fls) (Not tru)
subgoal 4 is:
 Not (And fls fls) = Or (Not fls) (Not fls)
```

De Morgan 則を証明してみる (3)★

```
De_Morgan_1 < simpl.
4 subgoals  (* 式を評価して変形 *)

  ============================
   fls = fls
(* 以下略 *)
De_Morgan_1 < reflexivity.
3 subgoals  (* 左辺=右辺 の時に使うと証明完了して次課題へ *)

  ============================
   Not (And tru fls) = Or (Not tru) (Not fls)

subgoal 2 is:
 Not (And fls tru) = Or (Not fls) (Not tru)
subgoal 3 is:
 Not (And fls fls) = Or (Not fls) (Not fls)
```

De Morgan 則を証明してみる (4) ★

残りの３通りについても証明する。auto. は simpl. と reflexivity. の動作を含んでいる。

```
De_Morgan_1 < auto.
De_Morgan_1 < auto.
De_Morgan_1 < auto.
Proof completed.

De_Morgan_1 < Qed.
intros.
destruct b1; destruct b2.
 simpl.
 reflexivity.
 auto.
 auto.
 auto.
De_Morgan_1 is defined
```

同様に Not (Or b1 b2) = And (Not b1) (Not b2) も試してみよ。

課題１：３値論理

1. `Yes`, `Maybe`, `No` の３つの値を持つ型 `Bool3` を定義せよ。
2. `Bool3` に対する `And3`, `Or3`, `Not3` を適切に定義せよ。
3. (★) 上の定義が適切であれば、やはり De Morgan 則が成立するはずである。同様に証明してみよ。今回は何通りを総当たりすることになるか？

ヒント：`destruct b1; destruct b2.` は、`destruct b1.` の各場合に対してそれぞれ `destruct b2.` を実行している。つまり `destruct b1; destruct b2; auto.` とすると、全ての場合に対して `auto.` が実行される。

自習用資料

- *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。**
- *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。**
- **「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**。日本語。自習教材としてお薦め。**
- *Coq in a Hurry*, Yves Bertot (`http://cel.archives-ouvertes.fr/docs/00/47/58/07/PDF/coq-hurry.pdf`) : **簡潔にまとめられたチュートリアル**
- **「プログラミング** Coq**」** (`http://www.iij-ii.co.jp/lab/techdoc/coqt/`) : 2011 **年に** Coq **が有名になったのはこのチュートリアルの影響。**

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。モデル検査ツールの Alloy のチュートリアルや定理証明系イベント ProofSummit 2011 を開催したり、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。