Coq によるプログラムの検証入門

集中講義 千葉大学大学院

アフェルト レナルド

産業技術総合研究所

2017 年 01 月 25(2/2)

目的

定理証明支援系 Coq を用いたプログラムの検証入門

1. Coq のプログラミング言語を紹介する. 型を用いて定義した論理述語で, 入力と出力を制限する. そうすると, 詳細な仕様を記述できる. 最終的に, 検証済み, 実行可能の OCaml プログラムを生成できる.
2. 仕様が複雑な場合, タクティクを用いて, 間接にプログラムを書くことができる. 関数型プログラムに関する論理式を証明することになる.
3. また, Coq で, 任意のプログラミング言語のシンタクスをデータ構造として形式化ができ, その意味論も論理述語として形式化できる. そうすると, タクティクを用いて, 命令型言語に関するリーゾニングもできる. 最小なホーア論理を用いて命令型言語の検証を行ってみる.

アウトライン

pred(ecessor) 関数

- Coq で自然数は帰納的型として定義されている (他の帰納的型も後で説明する):

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

- Coq での pred 関数:

```
Definition prec (n : nat) : nat :=
  match n with
  | O => O
  | S m => m
  end.
```

- 出力した OCaml の関数:

```
let prec = function
| O -> O
| S m -> m
```

```
type nat =
| O
| S of nat
```

- 「完全 (total)」な関数である. 特に, `prec O` は O となる...

部分的 (partial) な pred 関数に向けて

関数の入力が正であるように制限する. そのため, 入力が正であるという証明をパラメータとして追加する.

- pprec は関数を返す. 返す関数は nat を返す. ただし, 返す関数の入力は n とすると, 0 < n の証明も入力として渡さなければ成らない (<の定義は スライド 10 で調べる):

```
Definition pprec (n : nat) : 0 < n -> nat :=
  match n with
  | O => fun H => ???
  | S m => fun _ => m
  end.
```

- n は O である場合, 何を返す?

どうやって矛盾の証明を使う?

- 矛盾は型として定義されている. ただし, その型を持つものを構成できない (具体的に, 矛盾の帰納的型は構成しがない) (スライド 8 で改めて帰納的述語を調べる):

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

- 矛盾の証明から, 何でも構成できる. 例えば, 下記の関数を用いて何でもの自然数を構成できる:

```
Definition false_nat (abs : False) : nat :=
  match abs with end.
```

- 一方, Coq の標準ライブラリで次の補題がある (今のところはその証明を無視する):

```
Nat.lt_irrefl : forall x : nat, x < x -> False
```

実際に, `0 < 0` の証明から, 何でも作れる (*ex falso quodlibet*).

部分的な pred 関数

スライド 6 の false_nat[1] を使う:

```
Definition pprec (n : nat) : 0 < n -> nat :=
  match n with
  | O => fun H => false_nat (Nat.lt_irrefl _ H)
  | S m => fun _ => m
  end.
```

(自動的に推論できる引数は_と書いても良い.)
出力される OCaml 関数:

```
let pprec = function
| O -> assert false (* 矛盾の場合 *)
| S m -> m
```

[1] 一般化: False_rec

帰納的型の述語

- Coq の標準ライブラリでは, $\leq$ は帰納的型の述語として定義されている:

$$\frac{}{n \leq n}\ \mathsf{le\_n} \qquad \frac{n \leq m}{n \leq m+1}\ \mathsf{le\_S}$$

- Coq で次のように記述する:

```
Inductive le (n : nat) : nat -> Prop :=
  le_n : n <= n
| le_S : forall m : nat, n <= m -> n <= S m
```

  - `n <= m` は `le n m` の代わりの記法である
  - 型の引数の中に, **パラメータ**と **index** を区別する

- つまり, 証明は通常のデータ構造のようなものである

不等式を証明すること

- 不等式の証明の例:
  - `le_n 1` は $1 \leq 1$ の証明である,
  - `le_S _ _ (le_n 1)` は $1 \leq 2$,
  - `le_S _ _ (le_S _ _ (le_n 1))` は $1 \leq 3$, 等.
- 下記の関数[2] は `1 <= S n` の証明を構成する (その関数型から読み取れる):

```
Fixpoint spos (n : nat) : 1 <= S n :=
  match n with
  | O => le_n 1
  | S m => le_S _ _ (spos m)
  end.
```

[2]帰納的関数の場合, `Definition` の代わりに `Fixpoint` を使う.

部分的な pred 関数を使うこと

- `a < b` は `a + 1 <= b` として定義されている
- 従って, spos 関数 (スライド 9) を用いて, `0 < S n` の証明を構成できるし, 部分的な pred 関数も次のように実行できる:

```
> Compute pprec 5 (spos _).
  = 4
  : nat
```

同値関係ん定義

- Coq で, 同値関係は元々の言語の機能ではなく, 帰納的型で定義されている:

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

  - `x = y` は `eq x y` のための記法
  - 引数 `A : Type` は暗黙である: `_`を使わなくても自動的に推論される

- 証明の例:
  - `eq_refl 0` は $0 = 0$ の証明である,
  - `eq_refl true` は $\mathit{true} = \mathit{true}$, 等.
- `0 = 1` 書けるが, 証明はできない
- `eq_refl 4` は $4 = 4$ の証明であり, $2 + 2 = 4$ の証明でもある. 従って, 型の中で, 計算 (β-簡約) ができる.
  - 計算に当たる証明項はない (Poincaré 法則)
  - 「リフレクション (reflection)」と言う

ブール不等式を用いる部分的な pred 関数

- Coq の標準ライブラリでは, 不等式を決定する関数がある. ただし, その関数は `true` または `false` を返す. つまり, ブール関数である:

```
Nat.ltb : nat -> nat -> bool
```

- ブール関数を用いて, 前のスライドの pred 関数を書き直す:

```
Definition pprecb (n : nat) : Nat.ltb 0 n = true -> nat :=
  match n with
  | 0 => fun H => false_nat
    (Nat.lt_irrefl _ (proj1 (Nat.ltb_lt _ _) H))
  | S m => fun _ => m
  end.
```

(`Nat.ltb_lt` は証明である. `ltb` と `lt` は同じことを決定することを表す.)

- 不等式の証明は最小のし証明項になる:

```
Compute pprecb 5 eq_refl.
```

アウトライン

pred 関数の完全な仕様を書くことができる

- さらに詳細な型を書くことができる:

```
Definition pprec_interactif (n : nat) :
  0 < n -> {m | n = S m}.
```

- その場合, 返される型は存在の証明である:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> {x : A | P x}
```

  - 例えば, `exist (fun x =>x =0) 0 eq_refl` は 0 に等しい自然数が存在する証明である

- 証明項を提供せずに, 型を宣言すると, 対話的なモードに入る.
- このような関数を直接に記述するためには, 技が要る (特に, 同値関係の証明の移動). Coq の `Program` [CDT16, 24 章] 拡張は一部を自動化する: 関数の大雑把な形を定義してから, 証明を追加できる. Coq のタクティクを用いて, 間接に関数を記述もできる.

対話的な構築

タクティク destruct n as [|m]

次の形の項を導入することと同じ:

`match n with 0 =>... |S m =>... end`

入力 (対話的なモード)

```
Definition pprec_interactif
  (n : nat) : 0 < n -> {m | n = S m}.
```

```
destruct n as [|m].
```

構築される関数 (表示されない)

```
fun n : nat => ?
```

```
fun n : nat =>
 match n as n0 return
 (0 < n0 -> {m : nat | n0 = S m})
 with
 | 0 => ?0
 | S m => ?1
 end
```

(サブゴール?0 と?1 が生成される. それぞれを順番で処理する.)

対話的な構築

タクティク intros x

fun x =>... と同じ

タクティク generalize t

項 t を引数として導入する

入力 (対話的なモード)　　構築される関数 (表示されない)

(現在のゴール: 0 < 0 -> {m : nat |0 = S m})

```
- intros abs.
```

```
fun n : nat =>
 match n as ... with
 | 0 => fun abs :  0 < 0 => ?0
 | S m => ?1
 end
```

```
  generalize (Nat.lt_irrefl _ abs).
```

```
fun n : nat =>
 match n as ... with
 | 0 => fun abs : 0 < 0 =>
          ?0 (Nat.lt_irrefl 0 abs)
 | S m => ?1
 end
```

対話的な構築

タクティク `destruct 1` は「トップ」仮定に適用される
タクティク `intros _` は自分の引数を無視する関数を構築する

入力 (対話的なモード)　　構築される関数 (表示されない)

(現在のゴール: `False ->{m : nat |0 = S m}`)

```
destruct 1.
```

```
fun n : nat => match n as
  n0 return (0 < n0 -> {m : nat | n0 = S m}) with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False =>
     match H return m :  nat | 0 = S m with end)
     (Nat.lt_irrefl 0 abs)
 | S m => ?1
 end
```

(次のゴール: `0 < S m ->{m0 : nat |S m =S m0}`)

```
- intros _.
```

```
fun n : nat => match n as ... with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False => match H return ... with end)
     (Nat.lt_irrefl 0 abs)
 | S m => fun _ :  0 < S m => ?1
 end
```

対話的な構築

タクティク apply f

関数 f を適用する

入力 (対話的なモード)　　構築される関数 (表示されない)

(現在のゴール: {m0 : nat |S m =S m0})

```
apply (exist _ m).
```

```
fun n : nat => match n as ... with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False => match H return ... with end)
     (Nat.lt_irrefl 0 abs)
 | S m => fun _ : 0 < S m =>
   exist (fun m0 :  nat => S m = S m0) m ?1
 end
```

```
apply eq_refl.
```

```
fun n : nat =>
 match n as ... with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False => match H return ... with end)
     (Nat.lt_irrefl 0 abs)
 | S m => fun _ : 0 < S m =>
   exist (fun m0 : nat => S m = S m0) m eq_refl
 end
```

関数を書くこと = 補題を証明すること

Curry-Howard 同型対応の一例である

```
Definition pprec_interactif (n : nat) :
  0 < n -> {m | n = S m}.
...
Defined.
```

→ 構築された関数は見えるし, 実行できる

```
Lemma pprec_interactif (n : nat) :
  0 < n -> {m | n = S m}.
Proof.
...
Qed.
```

→ 構築された関数は表示されない. 同じ補題の二つの証明を同じにしても良い.
→ ->というシンボルを理論的な含意として読める

構成的証明 (つまり, 排中律を使わない証明) から OCaml のプログラムを出力できる

Coq のタクティク

- たくさんある (オプションも多い)[CDT16, 8 章] が, 重要なタクティクは限られている
  - オンラインのまとめ:
    `https://coq.inria.fr/refman/tactic-index.html`
- 重要なタクティク:
  - `induction`: 帰納的帰納的型の解析 (帰納法による論法に当たる)
  - `rewrite ->`/`rewrite <-`: 書き換え (実は, タクティク `apply` の適用)
  - `simpl`: β-簡約
  - `unfold`: `Definition` の展開
- 自動的なタクティクの例:
  - `auto`: 当たり前のとき
  - `tauto`: 直観主義論理の命題論理の決定
  - `omega`: Presburger 算術の決定

アウトライン

ホーア論理のまとめ

- プログラム c の実行は, 停止する場合, P を満たす状態から Q を満たす状態を導くことを $\{P\}c\{Q\}$ と書く.
  - P と Q はプログラムの状態に対するブール関数と考えても良い
- ホーア論理は三つ組に関する推論規則の集合である
  - それぞれの文法の要素に対して, 一つのルールがある (スライド 23 に参考)
- ホーア論理を意味論として理解しても良い
  - 操作的意味論との同値関係をよく証明する (**健全性と相対的完全性**)

ホーア論理の規則

- 最小の規則の集合:

$$\frac{}{\{Q\{e/v\}\}v \leftarrow e\{Q\}}\ \text{assign}$$

$$\frac{\{P\}c\{Q\} \quad \{Q\}d\{R\}}{\{P\}c; d\{R\}}\ \text{seq} \qquad \frac{P \to P' \quad \{P'\}c\{Q'\} \quad Q' \to Q}{\{P\}c\{Q\}}\ \text{conseq}$$

$$\frac{\{P \wedge t = \mathit{true}\}c\{P\}}{\{P\}\texttt{while}(t)\{c\}\{P \wedge t = \mathit{false}\}}\ \text{while}$$

  assign 規則を理解するよう, 例を書いてみる. while 規則の条件は**不変式**と呼ぶ.

- 事前/事後条件と不変式から, 検証を自動化できるはず
- 実際に, 対話的な証明によく頼る. その場合, 定理証明支援系はよく使われる (現実的な応用例:[WKS+09])

ホーア論理による証明の例

- プログラム $\mathit{while}(x \neq 0)\{\mathit{ret} = \mathit{ret} * x; x = x - 1\}$ が $x!$ を計算することを次のように示す:

$$\dfrac{\dfrac{\dfrac{}{\{Q\{\mathit{ret} * x/\mathit{ret}\}\}\mathit{ret} = \mathit{ret} * x\{Q\}}\ \text{assign}}{\{\mathit{ret} * x! = X! \wedge x \neq 0\}\ \mathit{ret} = \mathit{ret} * x\ \{\underbrace{\mathit{ret} * (x-1)! = X! \wedge 0 \leq x - 1}_{Q}\}}\ \text{stren} \qquad \dfrac{\dfrac{}{\{Q\{x-1/x\}\}x = x - 1\{Q\}}\ \text{assign}}{\{\mathit{ret} * (x-1)! = X! \wedge 0 \leq x - 1\}\ x = x - 1\ \{\underbrace{\mathit{ret} * x! = X!}_{Q}\}}\ \text{stren}}{\dfrac{\{\mathit{ret} * x! = X! \wedge x \neq 0\}\ \mathit{ret} = \mathit{ret} * x; x = x - 1\ \{\mathit{ret} * x! = X!\}}{\dfrac{\{\mathit{ret} * x! = X!\}\ \mathit{while}(x \neq 0)\{\mathit{ret} = \mathit{ret} * x; x = x - 1\}\ \{\mathit{ret} * x! = X! \wedge x = 0\}}{\{x = X \wedge \mathit{ret} = 1\}\ \mathit{while}(x \neq 0)\{\mathit{ret} = \mathit{ret} * x; x = x - 1\}\ \{\mathit{ret} = X!\}}\ \text{conseq}}\ \text{while}}\ \text{seq}$$

- 上記のリーゾニングを Coq で表現できるよう, 必要なインフラの形式化を行う

算術とブール表現の言語

- 変数を自然数として表現する:

```
Definition var := nat.
```

- 算術の表現は変数, 自然数, 掛け算, 引き算, 等である:

```
Inductive exp :=
| exp_var : var -> exp
| cst : nat -> exp
| mul : exp -> exp -> exp
| sub : exp -> exp -> exp.
```

例えば, $ret * x$ を次のように書く:
`mul (exp_var ret) (exp_var x).`

- ブール表現は同値関係, 否定, 等である:

```
Inductive bexp :=
| equa : exp -> exp -> bexp
| neg : bexp -> bexp.
```

最小の命令型言語

- プログラムは, 変数の割り当て, 順序の実行, ループからくる:

```
Inductive cmd : Type :=
| assign : var -> exp -> cmd
| seq : cmd -> cmd -> cmd
| while : bexp -> cmd -> cmd.
```

- 例えば, 次のプログラムは

$$while(x \neq 0)\{ret = ret * x; x = x - 1\}$$

次のように記述する:

```
while (neg (equa (exp_var x) (cst 0)))
  (seq
     (assign ret (mul (exp_var ret) (exp_var x)))
     (assign x (sub (exp_var x) (cst 1)))).
```

- ここで, 抽象的なシンタクスを使う. 実際に, Notation, Coercion, 等を用いて, 見やすくする.

表現の意味論 (1/3)

- 状態は変数から自然数への関数として表現する:

```
Definition state := var -> nat.
```

- 変数の割り当ては状態を定義する関数へのケースの追加に当たる:

```
Definition upd (v : var) (a : nat) (s : state) : state
  fun x => match Nat.eq_dec x v with
           | left _ => a
           | right _ => s x
           end.
```

ただし, `Nat.eq_dec` は自然性の同値関係が決定的であるという証明である:

```
Nat.eq_dec
     : forall n m : nat, {n = m} + {n <> m}
```

({...} + {...}は論理和として読める)

表現の意味論 (2/3)

- ある状態で表現を評価するのために, パーズを行う. ただし, 変数の場合, その状態に当たる関数を呼び出す:

```
Fixpoint eval e s :=
  match e with
  | exp_var v => s v
  | cst n => n
  | mul v1 v2 => eval v1 s * eval v2 s
  | sub v1 v2 => eval v1 s - eval v2 s
  end.
```

表現の意味論 (3/3)

- 例えば, *ret* と x という変数を作ってみる:

```
Definition ret : var := 0.
Definition x : var := 1.
```

- *ret* は 4 と x は 5 という状態を仮定する:

```
Definition sample_state : state :=
  fun x =>
    match x with
    | 0 => 4
    | 1 => 5
    | _ => 0
    end.
```

- 上記の状態で $ret * x$ という表現を評価すると, 20 を得る:

```
> Compute eval (mul (exp_var ret) (exp_var x)) sample_state.
  = 20 : nat
```

事前/事後条件の形式化

- 事前/事後条件を状態に対する関数として形式化する (*shallow encoding*):

```
Definition assert := state -> Prop.
```

  例えば, $ret = 1$ は次の関数になる:

```
fun s => eval (exp_var ret) s = 1
```

  この時点, 紙上では暗黙が多いと気がつくでしょう...

- また, 条件の間の含意等を話せるようになるよう, 論理結合子を「リフト」をしなければ成らない. 例えば:

```
Definition imp (P Q : assert) :=
  forall s, P s -> Q s.
```

ホーア論理の形式化 (1/2)

ホーアの三つ組を帰納的述語として定義される. その述語は, 事前/事後条件とプログラムのシンタクスの間の関係を表す:

```
Inductive hoare : assert -> cmd -> assert -> Prop :=

| hoare_assign : forall Q v e,
  hoare
   (fun s => Q (upd v (eval e s) s))
   (assign v e)
   Q

| hoare_seq : forall Q P R c d,
  hoare P c Q ->
  hoare Q d R ->
  hoare P (seq c d) R
```

$$\frac{}{\{Q\{e/v\}\}v \leftarrow e\{Q\}}$$

$$\frac{\{P\}c\{Q\} \quad \{Q\}d\{R\}}{\{P\}c; d\{R\}}$$

ホーア論理の形式化 (2/2)

```
| hoare_conseq : forall P' Q' P Q c,
  imp P P' -> imp Q' Q ->
  hoare P' c Q' ->
  hoare P c Q
```

$$\frac{P \to P' \quad \{P'\}c\{Q'\} \quad Q' \to Q}{\{P\}c\{Q\}}$$

```
| hoare_while : forall P b c,
  hoare
   (fun s => P s /\ beval b s)
   c
   P ->
  hoare
   P
   (while b c)
   (fun s => P s /\ ~ (beval b s)).
```

$$\frac{\{\lambda s.P\,s \wedge \mathtt{eval}(b,s)\}\ c\ \{P\}}{\{P\}\ \mathtt{while}(b)\{c\}\ \{\lambda s.P\,s \wedge \neg\mathtt{eval}(b,s)\}}$$

(/\, ~, 等は Coq での命題論理のための記法である)

応用: 階乗

- 紙上の記法:

$$\{x = X \wedge ret = 1\}$$
$$while(x \neq 0)\{ret = ret * x; x = x - 1\}$$
$$\{ret = X!\}$$

- Coq では (プログラムは スライド 26 に参考):

```
Lemma facto_fact x X ret : x <> ret ->
  hoare
    (fun s => eval (exp_var x) s = X /\
              eval (exp_var ret) s = 1)
    (facto x ret)
    (fun s => eval (exp_var ret) s = fact X).
```

最初のステップ

- メモ:

$$\frac{\{\overbrace{ret * x! = X!}^{P'}\}\,while(x \neq 0)\{ret = ret * x; x = x - 1\}\{\overbrace{ret * x! = X! \wedge x = 0}^{Q'}\}}{\{x = X \wedge ret = 1\}\,while(x \neq 0)\{ret = ret * x; x = x - 1\}\{ret = X!\}}$$

$$\frac{P \to P' \quad \{P'\}c\{Q'\} \quad Q' \to Q}{\{P\}c\{Q\}}\ \text{conseq}$$

- Coq では:

```
set (P' := fun s : state => ...).
set (Q' := fun s : state => ...).
apply (hoare_conseq P' Q').
```

三つのサブゴールが生成される. 今後, それぞれを証明しなければならない...

アウトライン

ライブラリの利用

- Coq の標準ライブラリに既に多くの補題がある:
  `https://coq.inria.fr/library`
- 新しい補題とタクティクをコンテキストに加えることができる. 例えば:

```
Require Import Arith.
Require Import Omega.
Require Import Factorial.
```

コンテキストにある補題をパターンを用いて検索できる. 例えば:

```
SearchPattern (_ + S _ = _).
SearchRewrite (_ + S _).
```

Coq によるプログラム検証に関する研究例

- 停止性の証明: 停止性は「当たり前」でない (つまり, 関数の再帰呼び出しは構造的でない) 場合, 先端な技術が要る ([BC04, 15 章] に参考)
- 分離論理:ポインターを扱うホーア論理の拡張 (Coq での形式化の例: [MAY06])
- 現実的な言語を扱うように今回のホーア論理を拡張できる (アセンブリ: [Aff13], C: [AS14])
- 今回のホーア論理を関数呼び出しで拡張できる (Coq での詳細な例: [Aff15])

参考文献


Reynald Affeldt, *On construction of a library of formally verified low-level arithmetic functions*, Innovations in Systems and Software Engineering **9** (2013), no. 2, 59–77.


———, *Proving properties on programs—from the Coq tutorial at ITP 2015—*, Coq tutorial @ ITP'15: `https://coq.inria.fr/coq-itp-2015`, Aug. 2015, Available at: `https://coq.inria.fr/files/coq-itp-2015/course-5.pdf`.


Reynald Affeldt and Kazuhiko Sakaguchi, *An intrinsic encoding of a subset of C and its application to TLS network packet processing*, Journal of Formalized Reasoning **7** (2014), no. 1, 63–104.


Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development—Coq'Art: The calculus of inductive constructions*, Springer, 2004.


The Coq Development Team, *The Coq proof assistant reference manual*, INRIA, 2016, Version 8.5.


Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa, *Formal verification of the heap manager of an operating system using separation logic*, Proceedings of the 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1–3, 2006, Lecture Notes in Computer Science, vol. 4260, Springer, 2006, pp. 400–419.


Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish, *Mind the gap*, Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009, Munich, Germany, August 17–20, 2009, Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 500–515.