証明支援系 Coq の基礎を学ぶ

大町 誠也

仁川学院高等学校

要旨

専修コースにおいて Pierce 他著であるテキスト Software Foundations を用いて，基礎的な Coq の使用方法を学んだ．証明支援システムとは，命題及び証明の記述，証明に欠陥がないかの確認が行えるソフトウェアのことである．利用者はタクティクスと呼ばれるコマンドを用いてソフトウェアと対話方式で証明の記述を進めていく．本論文では，3 つの証明例を用いて Coq の基礎的証明手法を説明する．命題は，$(a+b+c+d)^2$ の展開，ド・モルガンの法則の一例，0 から n までの 2 乗の総和に関する公式の 3 つである．$(a+b+c+d)^2$ の展開に関しては，タクティクスとして intros, rewrite, reflexivity が用いられている．ド・モルガンの法則の一例は，bool に関する命題で destruct という場合分けのためのタクティクスが 3 回用いられている．そして 0 から n までの 2 乗の総和に関する公式は，数学的帰納法を用いるためのタクティクス induction を使った後，rewrite と assert を用いて書き換えることで証明されている．確かに Coq は厳密性に富むが，証明を理解するのは非常に難しいため，コメントを入れるべきである．

重要語句：Coq，命題，証明，rewrite，destruct，bool，induction，assert

序論

私は Coq [(1)] と呼ばれる証明支援システムと，テキストとして Software Foundations [(2)] を利用して，コンピュータ上においての証明の仕方を学んだ．

Coq はフランスにて，フランス国立情報学自動制御研究所（INRIA）によって開発された証明支援システムである．Gallina と呼ばれる言語を用いて操作者と Coq との相互的やり取りを行うことにより，数学的定義及び証明を記述していく．証明にはそれぞれ何らかの機能を携えたタクティクスと呼ばれるものを用いる．Coq を起動すると，図 1 の画面が表示される．操作者は左部に記述していく．それを左上部にある矢印を押すことで，Coq が読み込み，右上部に現在の証明目標が表示される．また，右下部には，証明終了時に，定義された旨が表示されるほか，定理名等を検索した際の結果が表示される．

命題 1

参考資料 1 参照.

$\forall a, b, c, d \in nat,$

$$(a+b+c+d)^2 = a^2 + b^2 + c^2 + d^2 + 2ab + 2ac + 2ad + 2bc + 2bd + 2da.$$

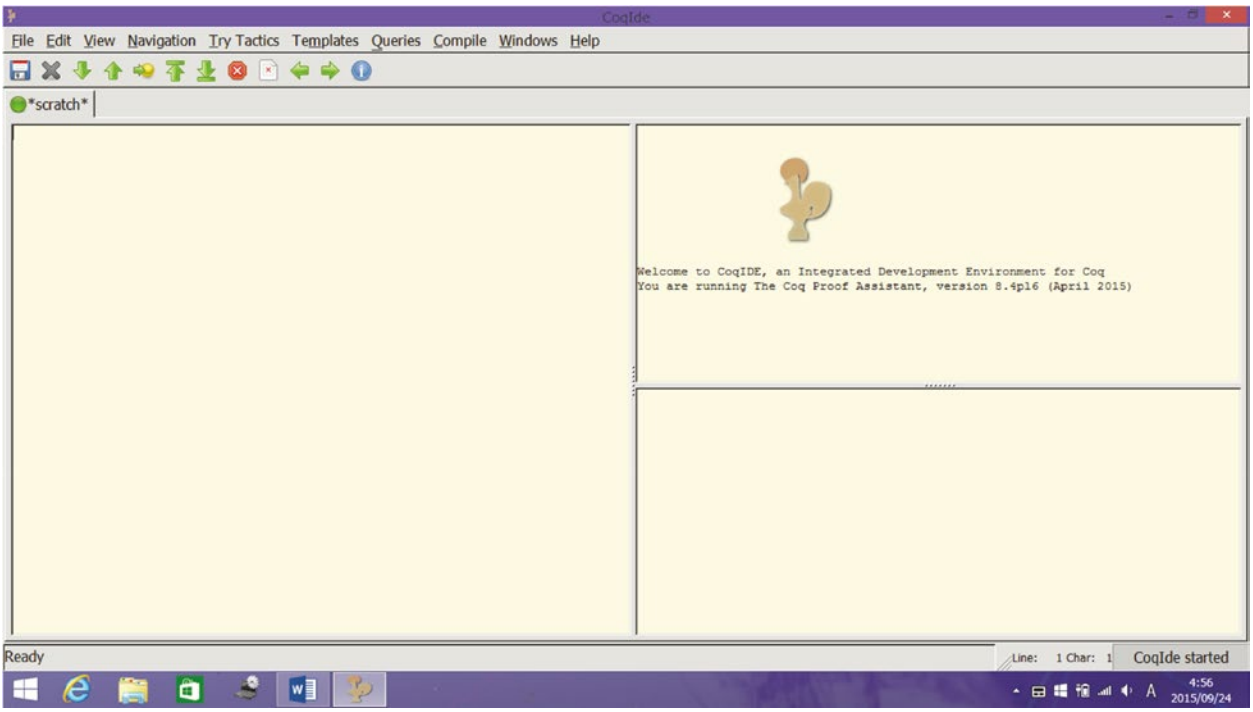

図 1. Coq 使用時画面

内容に関する連絡先：
五十嵐 淳（京都大学大学院情報学研究科）
igarashi@kuis.kyoto-u.ac.jp

使用タクティクス

intros（1 回），右向き rewrite（27 回），右向き変数指定型 rewrite（5 回），左向き変数指定型 rewrite（3 回），reflexivity（1 回）

nat とは，0 を含む自然数の集合で，Coq では，nat は次のように定義されている．

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
```

つまり，自然数は「自然数 O が存在し，また任意の自然数 a の後者 S a もまた自然数である」という定義がなされている．十進数に直すと，O が 0，S O が 1，S (S O) が 2，S (S (S O)) が 3……と S の数がその値となる．

intros は，変数及び仮定を，証明すべき命題から既知事項としてコンテクストと呼ばれる部分に移す役割を担う．

rewrite は今までに証明が完了している，あるいは Coq 内のライブラリ中の定理及び補題を用いることで命題を書き換えるために用いられるタクティクスである．rewrite は，書き換え方向及び変数の指定の有無という 2 つの観点からそれぞれ 2 通り，つまり総じて 4 通りに分類することが可能である．rewrite の直後には -> あるいは <- のいずれかの矢印を配置する．先述の通り，rewrite とは定理を用いることで書き換えを行う．例えば，ある定理 m = n に関して右向き矢印を用いると，目下，証明に取り組んでいる命題に含まれる m の部分を n に書き換える．対して，左向き矢印を用いて適用させると命題中の n の部分を m に書き換える．また，rewrite を使用する際，適用させる定理中における変数がいずれに相当するか指定することも可能である．

reflexivity は両辺が等しいときに用いる．ここでいうところの等しいとは，定義に沿った計算に基づくと，左右で完全に一致するということである．

本命題は，文字が 4 つの場合に関する 2 乗の展開である．紙面上の証明では，展開を行うことで証明できるが，Coq 上では，分配法則に基づき分解したのち，交換法則によって適切な順序に並び替えることが要される．

命題 2

参考資料 2 参照.

∀a, b, c, d ∈ bool,「a かつ b かつ c かつ d」でない＝
「a でない」または「b でない」
または「c でない」または「d でない」.

使用タクティクス

intros（1 回），destruct（3 回），reflexivity（4 回）

bool とは，true と false を要素としてもつ集合のことで Coq では以下のように定義される．

```
Inductive bool : Type :=
  | true : bool
  | false : bool.
```

これは，つまり bool は「true」と「false」の 2 要素からなるということである．

destruct とは場合分けを行うためのタクティクスである．

Coq において場合分けを行うときは，destruct というものを用いる．n という自然数に関して場合分けを施した際は，「O の場合」と「何らかの自然数の後者 S i の場合」の 2 通りに場合分けがされる．bool に関して場合分けを行うと，「true の場合」と「false の場合」の 2 通りに分けられる．

本命題は，ド・モルガンの法則の特殊な場合であるが，これは，a，b，c，d の 4 つに関してそれぞれ場合分けを行うことで解決される．1 度にこれらに関して場合分けを行うと，2 の 4 乗，つまり 16 通りについて考察せねばならない．ここで，「かつ」（andb4）と「または」（orb4）の性質について考えると，「かつ」（andb4）については，とる引数を前から調べていき，false が存在した時点で返り値が確定する．一方「または」（orb4）に関しては，とる引数を前から調べていき，true が存在した時点で返り値が確定する．従って，a, b, c, d について，前から見ていって false が存在した時点で命題の両辺は確定する．そこで，a, b, c について順番に場合分けを行い，「a，b，c が true の場合」，「a, b が true，c が false の場合」，「a が true，b が false の場合」，「a が false の場合」の 4 通りについて考察するだけで証明することに成功した．

命題 3

参考資料 3 参照.

$$\forall n \in \mathbf{N}, \qquad 6 \times \sum_{k=0}^{n} k^2 = n(n+1)(2n+1).$$

使用タクティクス

intros（1 回），induction（1 回），右向き rewrite（26 回），左向き rewrite（7 回），右向き変数指定型 rewrite（7 回），左向き変数指定型 rewrite（5 回），assert（6 回），simpl（2 回），reflexivity（8 回）（assert 内でのタクティクスを含む）

induction とは数学的帰納法を行うためのタクティクスである．本命題の場合，自然数 n に関して数学的帰納法を行っている．自然数について数学的帰納法を行うと，場合分け時同様，「O の場合」と「何らかの自然数の後者 S i の場合」の 2 通りに分かれる．ここで大きく異なるのは帰納法の仮定（IHi）を用いることが可能であるという点である．IHi を用いる場合には rewrite を用いる．

assert とは，命題中の変数を用いた式を別途証明して本題に適用させるために問題提起するタクティクスである．補題と異なり他の命題では使うことができず，また変数もその通りでないと適用することができないが，比較的手軽に用いることが可能である．

simpl とは，定義に沿って両辺を計算せよという指令である．例えば，和（plus, +）について考えてみよう．和（plus, +）は，Coq において以下のように定義づけされている．

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

つまり，n + m について n = O ならばその返り値は m であり，n = S n'（n' の後者）ならば，その返り値は S (n' + m)（n' と m の和の後者）であるということである．従って，n が O であるか，何らかの自然数の後者であるかによって，和の計算方法が異なってくるのである．従って，1 + n は simpl を用いると，S n と変形されるが，n + 1 は S n とは変形されず，そのままである．n + 1 が S n に等しいことを示すためには，「和の交換法則を用いてから simpl を行う」，「数学的帰納法を行う」のいずれかの手法をとる必要がある．また，命題 1 中で説明した reflexivity は，simpl を行うと両辺が同形となることを確認するタクティクスと言い換えることができる．したがって，1 + n = S n に reflexivity は適用可能であるが，n + 1 = S n には reflexivity は適用することができないのである．

定理

上記の命題で用いられた定理について記載する．

命題 1 中では，square_development，Plus.plus_comm，Plus.plus_assoc，Mult.mult_plus_distr_l，Mult.mult_plus_distr_r の 5 種の定理を用いた．

・square_development

自然数 m，n について，

$$(m + n)^2 = m^2 + 2mn + n^2$$

であることを主張する定理である．2 乗する関数の定義である「$n^2 = n \times n$」を用いて式を変形したのち，分配法則，積の交換法則，及び和の結合法則を用いて証明している．

・Plus.plus_comm

自然数 m，n について，

$$n + m = m + n$$

であることを主張する定理である．+ の定義は，その前にあるオペランドがゼロかそうでないかで場合分けされて定義されているので，n について数学的帰納法を行って証明する．また n+0=n という定理及び S (n+m)=n+(S m) という定理の二つを用いて，両辺を同じ式の形に変形する．

・Plus.plus_assoc

自然数 m，n，p について，

$$n + (m + p) = n + (m + p)$$

であることを主張する定理である．Plus.plus_comm 同様，そのままでは + の定義に従えないので n について数学的帰納法を行うことで証明できる．

・Mult.mult_plus_distr_l

自然数 k，m，n について，

$$k \times (m + n) = k \times m + k \times n$$

であることを主張する定理である．× は + 同様，演算子の前のオペランドがゼロかそうでないかによって定義されているため，k について数学的帰納法を行う．ただし，× の定義に従っただけでは求める結果と足し算の順序が異なるので，和の交換法則及び結合法則を用いる必要がある．

・Mult.mult_plus_distr_r

自然数 k，m，n について，

$$(m + n) \times k = m \times k + n \times k$$

であることを主張する定理である．上の分配法則及び，積の交換法則を用いて証明できる．

命題 2 中では，補題は用いていない．

命題 3 中では，Sum_square_Sn，Plus.plus_comm，Plus.plus_assoc，Mult.mult_1_l，Mult.mult_1_r，Mult.mult_comm，Mult.mult_assoc，Mult.mult_plus_distr_l，Mult.mult_plus_distr_r の 9 種の定理を用いた．

・Sum_square_Sn

自然数 n について，

$$\sum_{k=0}^{Sn} k^2 = \sum_{k=0}^{n} k^2 + (Sn)^2$$

であることを主張する定理である．$\sum k^2$の定義から証明される．

・Mult.mult_1_l

自然数 n について，

$$1 \times n = n$$

であることを主張する定理である．× の定義から証明される．

・Mult.mult_1_r

自然数 n について，

$$n \times 1 = n$$

であることを主張する定理である．上の定理と違い，このままでは × の定義が使えないので，n について数学的帰納法を用いて証明する．

・Mult.mult_comm

自然数 m，n について，

$$n \times m = m \times n$$

であることを主張する定理である．× の定義は，その前にあるオペランドがゼロかそうでないかで場合分けされて定義されているので，n について数学的帰納法を行って証明する．また Mult.mult_1_r 及び n+(S m)=n+n×m という定理の二つを用いて，両辺を同じ式の形に変形する．

・Mult.mult_assoc

自然数 n，m，p について，

$$n \times (m \times p) = (n \times m) \times p$$

であることを主張する定理である．Mult.mult_comm 同様，そのままでは × の定義に従えないので，n について数学的帰納法を用いる．形を合わせるため途中，分配法則を用いる．

考察

Coq での証明においては，紙面上での証明では通常省略される，因数分解・展開等の途中過程を和・積の交換法則・結合法則又分配法則等を用いて逐一，記述することが要される．このことから，Coq 上で証明を行えば，確かに煩雑さは増すが，紙面上での証明に比べ，遥かに誤りが減り，より数学に厳密に向き合うことが可能となるが分かる．また，Coq においての証明の正確さは読み込ますことによって判別することが出来るが，大方その証明には，証明の各時点での数式は示されておらず，証明の内容は非常に理解しがたい．そこで，証明中に適宜コメントを挿入することにより，その動作が何の目的で何を行っているのか明示する必要があると考えられる．

謝辞

本論文を作成するにあたり，多くのご指導を賜った五十嵐淳教授，中澤巧爾助教，奥村健太郎氏に感謝申し上げます．また基盤コース後期・専修コースで多くのご教授を頂いた五十嵐研究室の皆様に深謝いたします．この度，ELCAS という学びの場を与えてくださった京都大学の教員の方々，また多くの準備をしていただいた ELCAS 事務局の方々に深謝いたします．

参考文献

1. INRIA. Coq, <https://coq.inria.fr/>, [accessed 07 Sep. 2015]. (1989 onwards).
2. Pierce, B. C., C. Casinghino, M. Greenberg, V. Sjöberg & B. Yorgey. Software Foundations ver. 3.2, <http://www.cis.upenn.edu/~bcpierce/sf/current/index.html>, [accessed 07 Sep. 2015]. (2015).

Learning the Basics of the Proof Management System Coq

Masaya Ohmachi

Nigawa Gakuin High School

Abstract

I learnt how to use the formal proof management system, Coq, by using the textbook entitled “Software Foundations” by Pierce *et al*. A proof management system is a piece of computer software that allows users to write down mathematical propositions and proofs, and is able to mechanically check if proofs are correct. Users develop a proof by using commands called “tactics.” In this report, I explain the basic use of Coq with three example proofs. The three statements include the development of $(a+b+c+d)^2$, an instance of De Morgan’s laws, and the formula of the summation of squares of natural numbers from 0 to n. In the development of $(a+b+c+d)^2$, tactics called “rewrite,” “intros,” and “reflexivity” are used. In the instance of De Morgan’s laws, which concern Boolean algebra, a tactic called “destruct,” by which users perform case analysis in Coq, is used three times. The formula of the summation of squares is proven by making use of “rewrite” and “assert” after using the tactic of induction for mathematical induction. Although Coq is very rigorous, users should insert comments into their proofs because it is very difficult for other people to understand proofs written in Coq.

Key words: Coq, Propositions, Proof, Rewrite, Destruct, Bool, Induction, Assert

Correspondence Researcher:
Igarashi, A. (igarashi@kuis.kyoto-u.ac.jp)
Graduate School of Informatics, Kyoto University

参考資料

参考資料 1: 命題 1 の証明

```
Theorem square_a_b_c_d : forall (a b c d:nat),
square (a+b+c+d)=(square a)+(square b)+(square c)+(square d)+2*a*b+2*a*c+2*a*d+2*b*c+2*b*d+2*c*d.
Proof.
intros a b c d.
(* 両辺の平方部を展開する *)
rewrite->square_development.
rewrite->square_development.
rewrite->square_development.
(* 以下，両辺の形を揃える *)
rewrite->(Plus.plus_comm (square a + 2 * a * b + square b + 2 * (a + b) * c + square c +2 * (a + b + c) * d) (square d)).
rewrite->Plus.plus_assoc.
rewrite->Plus.plus_assoc.
rewrite->(Plus.plus_comm (square d + (square a + 2 * a * b + square b + 2 * (a + b) * c)) (square c)).
rewrite->Plus.plus_assoc.
rewrite->Plus.plus_assoc.
rewrite->Plus.plus_assoc.
rewrite->(Plus.plus_comm (square c+ square d+ (square a+2*a*b) ) (square b)).
rewrite->Plus.plus_assoc.
rewrite->Plus.plus_assoc.
rewrite->(Plus.plus_comm (square b + (square c + square d)) (square a)).
rewrite->Plus.plus_assoc.
rewrite->Plus.plus_assoc.
rewrite<-(Mult.mult_assoc 2 (a+b) c).
rewrite->Mult.mult_plus_distr_r.
rewrite->Mult.mult_plus_distr_l.
rewrite->Plus.plus_assoc.
rewrite->Mult.mult_assoc.
rewrite->Mult.mult_assoc.
rewrite<-(Mult.mult_assoc 2 (a+b+c) d).
rewrite->Mult.mult_plus_distr_r.
rewrite->Mult.mult_plus_distr_r.
rewrite->Mult.mult_plus_distr_l.
rewrite->Plus.plus_assoc.
rewrite->Mult.mult_plus_distr_l.
rewrite->Plus.plus_assoc.
rewrite->Mult.mult_assoc.
rewrite->Mult.mult_assoc.
rewrite->Mult.mult_assoc.
rewrite<-(Plus.plus_assoc (square a + square b + square c + square d + 2 * a * b + 2 * a * c ) (2*b*c) (2*a*d)).
rewrite->(Plus.plus_comm (2*b*c) (2*a*d)).
rewrite->Plus.plus_assoc.
reflexivity.
Qed.
```

参考資料 2: 命題 2 の証明

```
Theorem De_Morgan_law:forall (a b c d: bool),
negb (andb4 a b c d)=orb4 (negb a) (negb b) (negb c) (negb d).
Proof.
intros a b c d.
destruct a.
destruct b.
destruct c.
(*a, b, c が true の場合 *)
reflexivity.
(*a, b が true, c が false の場合 *)
reflexivity.
(*a が true, b が false の場合 *)
reflexivity.
(*a が false の場合 *)
reflexivity.
Qed.
```

参考資料 3: 命題 3 の証明

```
Theorem sum_square_to_n : forall (n:nat),
 6 *(sum_square n) = n * (n + 1)*(2*n + 1).
Proof.
intros n.
(*n について数学的帰納法を行う *)
induction n as[|i].
(*n = 0 の場合 *)
reflexivity.
(*n = i で成り立つとき，n = i + 1 について *)
(* Σを i までと i + 1 に分解 *)
rewrite->sum_square_Sn.
(* 帰納法の仮定を使えるよう，分配する *)
rewrite->Mult.mult_plus_distr_l.
(* 帰納法の仮定を用いる *)
rewrite->IHi.
(* 以下，両辺の形を揃える *)
rewrite->Mult.mult_assoc.
rewrite->(Mult.mult_comm (6*S i) (S i)).
rewrite->(Mult.mult_comm i (i+1)).
rewrite->(Plus.plus_comm i 1).
(*assert H1 開始 *)
assert(H1:1+i=S i).
reflexivity.
(*assert H1 終了 *)
rewrite->H1.
rewrite<-(Mult.mult_assoc (S i) i (2*i+1)).
rewrite<-(Mult.mult_plus_distr_l (S i) (i*(2*i+1)) (6*S i)).
rewrite->(Mult.mult_plus_distr_l i (2*i) 1).
rewrite->(Mult.mult_comm i (2*i)).
rewrite->Mult.mult_1_r.
(*assert H2 開始 *)
assert(H2:6*S i=6*i+6).
rewrite<-H1.
rewrite->Mult.mult_plus_distr_l.
rewrite->Plus.plus_comm.
rewrite->Mult.mult_1_r.
reflexivity.
(*assert H2 終了 *)
rewrite->H2.
rewrite->Plus.plus_assoc.
rewrite<-(Plus.plus_assoc (2*i*i) i (6*i)).
(*assert H3 開始 *)
assert(H3:i+6*i=1*i+6*i).
rewrite->Mult.mult_1_l.
reflexivity.
(*assert H3 終了 *)
rewrite->H3.
rewrite<-Mult.mult_plus_distr_r.
(*assert H4 開始 *)
assert(H4:S i+1=i+2).
rewrite<-Plus.plus_comm.
simpl.
rewrite<-Plus.plus_comm.
reflexivity.
(*assert H4 終了 *)
rewrite->H4.
(*assert H5 開始 *)
assert(H5:2*S i+1=2*i+3).
rewrite<-H1.
rewrite->Mult.mult_plus_distr_l.
rewrite->Plus.plus_comm.
rewrite->Plus.plus_assoc.
simpl.
rewrite->Plus.plus_0_r.
rewrite->(Plus.plus_comm (i+i) 3).
reflexivity.
(*assert H5 終了 *)
rewrite->H5.
rewrite<-(Mult.mult_assoc (S i) (i+2) (2*i+3)).
(*assert H6 開始 *)
assert(H6:(i+2)*(2*i+3)=2*i*i+7*i+6).
rewrite->Mult.mult_comm.
rewrite->Mult.mult_plus_distr_l.
rewrite->Mult.mult_plus_distr_r.
rewrite->Mult.mult_plus_distr_r.
rewrite->Plus.plus_assoc.
rewrite->(Mult.mult_comm (2*i) 2).
rewrite->Mult.mult_assoc.
rewrite<-(Plus.plus_assoc (2*i*i) (3*i) (2*2*i)).
rewrite<-Mult.mult_plus_distr_r.
reflexivity.
(*assert H6 終了 *)
rewrite->H6.
reflexivity.
Qed.
```