Jacques Garrigue, 2019 年 6 月 19 日

# 命題論理の証明論

## 1　健全性と完全性

先週見たとおり，証明論の重要な定理が二つある．今回は空でない前提を考える．

**定義 1 (充足)** ある割り当て $v$ が論理式 $P$ を充足するとは，$[\![P]\!]_v = \mathit{true}$ ということであり，$v \models P$ とも書く．$v$ が $\Delta$ を充足する ($v \models \Delta$) とは，$\forall P \in \Delta,\ v \models P$ である．

**定義 2 (論理的帰結)** $\Delta$ を充足する任意のの割り当て $v$ について $v$ が $P$ を充足するなら，$P$ は $\Delta$ の論理的帰結といい，$\Delta \models P$ と書く．

**定理 1 (健全性)** $\Delta \vdash P$ が証明できるなら，$\Delta \models P$ になる．$\Delta$ が空なら，$P$ が恒真式．

健全性の証明は比較的に容易である．$\Delta \vdash P$ の証明 (の深さ) に関する帰納法を使い，各推論規則について前提が論理的帰結ならば結論も論理的帰結であると証明するだけでいい．

**定理 2 (完全性)** $P$ が $\Delta$ の論理的帰結ならば $\Delta \vdash P$ が証明できる．

完全性は難しい．今回の方針は，まず論理積標準形について完全性を証明し (前回の恒真式の判定より，各項をみるだけでいいので簡単)，この証明から元の (標準形でない式の) 証明を構築する．そのために多くの補題が必要になる．

**補題 1 (弱化)** 任意の $P, \Delta, \Delta'$ について $\Delta \vdash P$ が証明できれば $\Delta, \Delta' \vdash P$ も証明できる．

**補題 2 (排中律)** 任意の $P$ について $\Delta \vdash P \vee \neg P$ が証明できる．

**補題 3 (分配律)** 任意の $P, Q, R$ について $\Delta \vdash (P \wedge Q) \vee R$ が証明できれば，$\Delta \vdash (P \wedge R) \vee (Q \wedge R)$ が証明でき，その逆も然り．

## 2　実装

```
From mathcomp Require Import all_ssreflect.
Set Implicit Arguments.

(* ... *)

(* 恒真判定の正しさ *)
Lemma tauto_clause_ok c : reflect (forall v, eval_clause v c) (tauto_clause c).
Proof.
  elim: c => //= [|a c IH].
    by constructor => /(_ (fun => true)).
  case /boolP: (_ \in _) => //= Hna.
    constructor => v. (* reflect ... true *)
    case Ha: eval_lit => //=.
    move/eval_clause_true: Hna; apply.
    by rewrite eval_neg_lit Ha.
  apply: (iffP IH) => Hv.                          (* reflect のブール値が同じ *)
```

```
    move=> v; by rewrite Hv orbT.
  by apply (tautology_notin a).
Qed.

Lemma tauto_cnf_ok l : reflect (forall v, eval_cnf v l) (tauto_cnf l).
Proof.
  apply: (iffP allP) => /= [H v | H c Hc].   (* tauto_cnf が all で定義される *)
    apply/allP => /= c /H /tauto_clause_ok; by apply.
  apply /tauto_clause_ok => v.
  move/allP: (H v); by apply.
Qed.

Theorem tauto_ok p : reflect (tautology p) (tauto_cnf (cnf (nnf false p))).
Proof.
  rewrite /tautology.
  apply: (iffP (tauto_cnf_ok _)) => H v; move: (H v);
  by rewrite cnf_correct (nnf_correct,nnf_is_nnf).
Qed.

(* 証明論 *)

(* 証明の判定 *)
Reserved Notation "h |- p" (at level 69).
Definition hypotheses := seq prop.

(* 証明の定義 *)
Unset Implicit Arguments.
Inductive provable : hypotheses -> prop -> Prop :=
  | AxiomI : forall (h : hypotheses) (p : prop), p \in h -> h |- p
  | ImpI   : forall h p1 p2, p1 :: h |- p2 -> h |- (Imp p1 p2)
  | ImpE   : forall p1 h p2, h |- p1 -> h |- Imp p1 p2 -> h |- p2
  | ConjI  : forall h p1 p2, h |- p1 -> h |- p2 -> h |- Conj p1 p2
  | ConjE1 : forall p2 h p1, h |- Conj p1 p2 -> h |- p1
  | ConjE2 : forall p1 h p2, h |- Conj p1 p2 -> h |- p2
  | DisjI1 : forall h p1 p2, h |- p1 -> h |- Disj p1 p2
  | DisjI2 : forall h p1 p2, h |- p2 -> h |- Disj p1 p2
  | DisjE  : forall p1 p2 h q,
      h |- Disj p1 p2 -> p1 :: h |- q -> p2 :: h |- q -> h |- q
  | NegI : forall q h p,   (* 結論にない引数を最初におく *)
      p :: h |- q -> p :: h |- Neg q -> h |- Neg p
  | NegE : forall h p, h |- Neg (Neg p) -> h |- p
where "h |- p" := (provable h p).  (* 先に定義した記法を使う *)

(* 仮定を充足する割り当て *)
Definition validates v (h : hypotheses) := {in h, forall p, eval v p}.
(* 論理的帰結 *)
Definition consequence h p := forall v, validates v h -> eval v p.
(* validates のための補題 *)
Lemma in_consP (A : eqType) (P : A -> Prop) a s :
  P a -> {in s, forall x, P x} -> {in a :: s, forall x, P x}.
Proof. move=> Ha Hs x; by rewrite inE => /orP [/eqP -> | /Hs]. Qed.

(* 証明論の健全性 *)
Theorem soundness p h : h |- p -> consequence h p.
Proof.
```

```
  induction 1 => v Hv /=.
  - by apply Hv.
  - case Hp1: (eval v p1) => //=.
    apply IHprovable.
    by apply in_consP.
  - move/IHprovable2: (Hv) => /=.                          (* Hv を残す *)
    by rewrite IHprovable1.
Admitted.

(* 完全性の準備 *)

Hint Constructors provable.                (* provable を auto のヒントにする *)
Lemma axiom_head p h : p :: h |- p.                  (* AxiomI の特別な場合 *)
Proof. by apply AxiomI, mem_head. Qed.
Hint Resolve axiom_head.

(* 前提を強くしても証明が可能 *)
Theorem weakening h1 h2 h3 p :
  h1 ++ h2 |- p -> h1 ++ h3 ++ h2 |- p.
Proof.
  move eqh: (h1++h2) => h H.
  move: h1 h2 eqh.
  induction H => h1 h2 eqh; subst h; auto.
  - apply AxiomI.
    move: H; rewrite !mem_cat.
    case/orP => -> //.
    by rewrite !orbT.
  - apply ImpI.
    by apply (IHprovable (p1::h1)). (* 単一化のヒント *)
  - apply /ImpE /IHprovable2; auto. (* 連続 apply *)
Admitted.

(* よく使う特殊な場合 *)
Corollary weakening_head p h p' : h |- p' -> p::h |- p'.
Proof. apply (weakening nil h (p::nil)). Qed.
Hint Resolve weakening_head.

Lemma excluded_middle h p : h |- Disj p (Neg p).
Proof.
  apply NegE, (NegI p). (* 連続 apply, /lem と大体同じ *)
    apply NegE, (NegI (Disj p (Neg p))); auto.
  apply (NegI (Disj p (Neg p))); auto.
  Show Proof.
Qed.
Hint Resolve excluded_middle.                          (* tt の証明にもなる *)

Lemma ex_falso p q h : h |- Conj p (Neg p) -> h |- q.
Admitted.                        (* p のせいで auto が効かないので, Hint なし *)

(* Clause に関する完全性 *)
Lemma true_clause h :
  tauto_clause h -> exists b, (b, true) \in h /\ (b, false) \in h.
Proof.
  elim: h => //= a c IH.
  case /boolP: (_ \in c) => /=.
```

```
    case: a => a [] /=.
Admitted.

Definition prop_of_lit (l : lit) :=
  if l.2 then Atom l.1 else Neg (Atom l.1).

Definition prop_of_clause :=
  foldr (fun x => Disj (prop_of_lit x)) ff.

Definition prop_of_cnf :=
  foldr (fun x => Conj (prop_of_clause x)) tt.

Theorem prop_of_cnf_ok v c :
  eval_cnf v c = eval v (prop_of_cnf c).
Proof.
  elim: c => /= [|a c ->].
    by case: v.
Admitted.

Lemma prop_of_cnfP l h :
  h |- prop_of_cnf l <-> {in l, forall c, h |- prop_of_clause c}.
Proof.
  split; elim: l => //= c l IH H.
    apply: in_consP.
    - by move/ConjE1: H.
    - apply IH; by move/ConjE2: H.
  apply ConjI.
Admitted.

Lemma provable_clause a c h :
  a \in c -> prop_of_lit a \in h -> h |- prop_of_clause c.
Admitted.

Lemma completeness_cnf p :
  (forall v, eval_cnf v p) -> nil |- prop_of_cnf p.
Proof.
  move=> Val; apply /prop_of_cnfP => /= c Hc.
  case: (true_clause c) => /=.
    apply/tauto_clause_ok => v.
    move: c Hc; apply/allP/Val.
  move=> b [Ht Hf].
  apply NegE, (NegI (Atom b)); auto.
  - apply NegE, (NegI (prop_of_clause c)); auto.
    apply (provable_clause (b, false)); by rewrite // !inE eqxx.
  - apply (NegI (prop_of_clause c)); auto.
    apply (provable_clause (b, true)); by rewrite // !inE eqxx.
Qed.

Lemma provable_cnf_cat c1 c2 h :
  h |- prop_of_cnf (c1 ++ c2) <-> h |- prop_of_cnf c1 /\ h |- prop_of_cnf c2.
Proof.
  split.
  - move/prop_of_cnfP => H.
    split; apply/prop_of_cnfP => c Hc; apply H; by rewrite mem_cat Hc // orbT.
Admitted.
```

```
Lemma provable_disj_conj a b c h :
  h |- Conj (Disj a c) (Disj b c) <->
  h |- Disj (Conj a b) c.
Proof.
  split=> H.
    apply (DisjE a c); auto.
      move: H; apply ConjE1.
    apply (DisjE b c); auto.
    move/ConjE2: H; auto.
Admitted.

Lemma provable_clause_cat_inv c1 c2 h :
  h |- prop_of_clause (c1 ++ c2) ->
  h |- Disj (prop_of_clause c1) (prop_of_clause c2).
Proof.
  elim: c1 h => //= [|a c1 IH] h H; auto.
  move/DisjE: H; apply; auto.
  apply (DisjE (prop_of_clause c1) (prop_of_clause c2)); auto.
Qed.

Lemma provable_cnf_disj c1 c2 h :
  h |- prop_of_cnf (disj c1 c2) ->
  h |- Disj (prop_of_cnf c1) (prop_of_cnf c2).
Proof.
  elim: c1 h => [|a c1 IH] h /=; auto.
  case/provable_cnf_cat => H1 /IH H2.
  apply provable_disj_conj, ConjI; auto.
  elim: c2 H1 {H2 IH} => /= [|b c2 IH] H1; auto.
  move/ConjE2/IH: (H1) => {IH} /DisjE; apply; auto.
  move/ConjE1/provable_clause_cat_inv in H1.
  apply (DisjE (prop_of_clause a) (prop_of_clause b)); auto.
Qed.

(* nnf への変換が証明可能性を保存する *)
Lemma provable_nnf p b h :
  h |- (if b then Neg p else p) <-> h |- nnf b p.
Proof.
  elim: p b h => [n | p IH | p1 IH1 p2 IH2 | p1 IH1 p2 IH2] [] h;
    split => //= H; try (apply IH || move/(IH _ _) in H); auto.
  (* 1 *)
  apply (NegI p); auto.
  (* 2 *)
  apply (DisjE (Neg p1) (Neg p2)); first last.
  - apply DisjI2, IH2; auto.
  - apply DisjI1, IH1; auto.
  apply NegE, (NegI (Conj p1 p2)); auto.
  apply ConjI.
  - apply NegE, (NegI (Disj (Neg p1) (Neg p2))); auto.
  - apply NegE, (NegI (Disj (Neg p1) (Neg p2))); auto.
  (* 3 *)
  move/DisjE: H; apply.
  - apply (NegI p1), (IH1 true); auto.
    apply (ConjE1 p2); auto.
  - apply (NegI p2), (IH2 true); auto.
```

```
    apply (ConjE2 p1); auto.
Admitted.

Lemma provable_cnf h p :
  is_nnf p -> h |- prop_of_cnf (cnf p) -> h |- p.
Proof.
  elim: p h => [n | p IH | p1 IH1 p2 IH2 | p1 IH1 p2 IH2] h /=.
  - move=> _ /ConjE1 /DisjE; apply; auto.
    apply (ex_falso (Atom 0)); auto.
  - case: p IH => //= n IH _ /ConjE1 /DisjE; apply; auto.
    apply (ex_falso (Atom 0)); auto.
  - move=> /andP [np1 np2] /provable_cnf_cat [H1 H2]; auto.
  - move=> /andP [np1 np2] /provable_cnf_disj /DisjE; apply; auto.
Qed.

Theorem completeness p : tautology p -> nil |- p.
Proof.
  move=> H.
  apply <- (provable_nnf p false); auto.
  apply provable_cnf.
    apply nnf_is_nnf.
  apply completeness_cnf => v.
  by rewrite cnf_correct (nnf_correct,nnf_is_nnf) // H.
Qed.
```