Jacques Garrigue, 2019 年 6 月 12 日

命題論理の意味論

1 先週の課題

```
Theorem insert_ok a l : sorted l -> sorted (insert a l).
Proof.
  elim: l => //= b l IH /andP [lebl sl].
  case: ifPn => /= leab.
  - by rewrite leab (le_seq_trans _ leab) lebl.
  - by rewrite IH // le_seq_insert // le_total.
Qed.
Theorem isort_ok l : sorted (isort l).
Proof. by elim: l => //= a l /insert_ok. Qed.
Theorem isort_perm l : perm_eq l (isort l).
Proof.
  elim: l => //= a l pli.
  apply /perm_eq_trans /insert_perm. by rewrite perm_cons.
Qed.

Lemma leq_total a b : ~~ (a <= b) -> b <= a.
Proof. case /boolP: (a <= b) => //=; by rewrite -ltnNge => /ltnW. Qed.
Lemma isortn_ok l : sortedn (isortn l) && perm_eq l (isortn l).
Proof.
  have lto := leq_total. rewrite isort_perm /sortedn isort_ok //.
  by move=> x y z; apply leq_trans.
Qed.

Theorem even_odd n : even n = odd n.+1.      Proof. done. Qed.
Theorem odd_even n : odd n = even n.+1.      Proof. by rewrite /= negbK. Qed.
Theorem even_not_odd n : even n = ~~ odd n.  Proof. done. Qed.
Theorem even_or_odd n : even n || odd n.     Proof. by case: (odd n). Qed.
Theorem odd_odd_even m n : odd m -> odd n = even (m+n).
Proof.
  move=> Hm.
  have: even n.+1 = even (m.+1 + n.+1) by rewrite -even_plus /= negbK.
  by rewrite addnS /= !negbK.
Qed.
```

2 命題論理

第 2 回目の授業で見たように，命題論理は以下のように定義される．ただし，今回は古典論理の意味論を考えたいので，否定を構文に残すことにし，その代わり，$\underline{\text{False}}$ は $(P \wedge \neg P)$ と言う形で書けるので，省略した．

論理式 論理式は以下の結合子から定義される．

$$
\begin{array}{rll}
P, Q ::= & A & \text{論理変数} \\
| & \neg P & \text{否定} \\
| & P \wedge Q & \text{論理積} \\
| & P \vee Q & \text{論理和} \\
| & P \supset Q & \text{含意}
\end{array}
$$

論理値の割り当て　関数 $v : \mathcal{V} \to \{\mathsf{true}, \mathsf{false}\}$ は各論理変数に論理値を割り当てる．

論理式の評価　ある割り当て v の元で論理式を次のように評価できる．

$$
\begin{array}{rcl}
[\![A]\!]_v & = & v(A) \\
[\![\neg P]\!]_v & = & \text{if } [\![P]\!]_v \text{ then } \mathsf{false} \text{ else } \mathsf{true} \\
[\![P \wedge Q]\!]_v & = & \text{if } [\![P]\!]_v \text{ then } [\![Q]\!]_v \text{ else } \mathsf{false} \\
[\![P \vee Q]\!]_v & = & \text{if } [\![P]\!]_v \text{ then } \mathsf{true} \text{ else } [\![Q]\!]_v \\
[\![P \supset Q]\!]_v & = & \text{if } [\![P]\!]_v \text{ then } [\![Q]\!]_v \text{ else } \mathsf{true}
\end{array}
$$

恒真式　この意味論的な枠組みでは，恒真式は任意の割り当てで true になるものを指す．

$$P \text{ は恒真式} \overset{def}{\Leftrightarrow} \forall v, [\![P]\!]_v = \mathsf{true}$$

否定標準形　論理式の中で否定を論理変数にしか許さない形をを否定標準形という．任意の論理式を否定標準形に変換できる．変換は De Morgan の法則を使う．

$$
\begin{array}{rcl@{\qquad}rcl}
nnf(A) = A & & & nnf(\neg A) = \neg A & & \\
nnf(P \wedge Q) & = & nnf(P) \wedge nnf(Q) & nnf(\neg(P \wedge Q)) & = & nnf(\neg P) \vee nnf(\neg Q) \\
nnf(P \vee Q) & = & nnf(P) \vee nnf(Q) & nnf(\neg(P \vee Q)) & = & nnf(\neg P) \wedge nnf(\neg Q) \\
nnf(P \supset Q) & = & nnf(\neg P) \vee nnf(Q) & nnf(\neg(P \supset Q)) & = & nnf(P) \vee nnf(\neg Q)
\end{array}
$$

論理積標準形　論理積標準形では外側に論理積があり，その中に論理和があり，一番奥に論理変数とその否定があるという三層の標準形である．

否定標準形から，以下の変換を繰り返せば良い．

$$P \vee (Q \wedge R) \longrightarrow (P \vee Q) \wedge (P \vee R) \qquad (P \wedge Q) \vee R) \longrightarrow (P \vee R) \wedge (Q \vee R)$$

恒真式の判定　論理積標準形に変換すると、恒真式の判定が簡単になる．具体的には，各式は集合の集合（あるいはリストのリスト）に変換されると思っていい．

$$\{\{P \vee Q \vee R\} \wedge \{\neg Q \vee \neg P \vee Q\}\}$$

そのとき，論理和をなす集合にある論理変数は正と否定という形で 2 回出ていれば，その論理和は true になる．全ての論理和がそうなっていれば，恒真式である．

3　便利なコマンド

「`constructor`」 Inductive の構成子を自動的に `apply` する．

「`econstructor`」 その `eapply` 版．

「`destruct` x」 `case` と同じだが，仮定においたままでも使える．

「`induction 1`」 `elim` と同じだが，新しい変数や仮定を自動的におく．名前が選びにくいのが難点だが，場合が多いときは便利．

「`case/boolP: (式)`」 「式」と「`~~`式」に場合分けをする．特に,「`~~(a == b)`」は「`a != b`」と,「 `~~(a \in b)`」は「`a \notin b`」と表示されるので，そういう式を扱うときは便利．

4 実装

今日の実装は http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2019_SS/ においてある.

```
(* 命題論理 *)

From mathcomp Require Import all_ssreflect.
Set Implicit Arguments.

(* 命題の定義 *)
Inductive prop : Set :=
  | Atom of nat
  | Neg of prop
  | Conj of prop & prop
  | Disj of prop & prop.

Definition Imp p1 p2 := Disj (Neg p1) p2.                (* 含意は定義できる *)
Notation tt := (Disj (Atom 0) (Neg (Atom 0))).                (* 透明な定義 *)
Notation ff := (Conj (Atom 0) (Neg (Atom 0))).

(* 推移律 *)
Definition sp :=                          (* (A -> B) -> (B -> C) -> (A -> C) *)
  Imp (Imp (Atom 1) (Atom 2))
    (Imp (Imp (Atom 2) (Atom 3)) (Imp (Atom 1) (Atom 3))).

(* 比較を自動的に生成 *)
Scheme Equality for prop.
Check prop_eq_dec.

(* bool の関数を作る *)
Definition prop_eq_decb x y :=
  match prop_eq_dec x y with left _ => true | right _ => false end.

Lemma prop_eqP : Equality.axiom prop_eq_decb.
Proof.
  move=> x y.
  rewrite /prop_eq_decb.
  case: prop_eq_dec => H; by constructor.
Qed.

(* 等価述語として登録 *)
Canonical prop_eqMixin := EqMixin prop_eqP.
Canonical prop_eqType := Eval hnf in EqType _ prop_eqMixin.

Goal ff == ff.
done.
Abort.

(* 意味論 *)

Definition valuation := nat -> bool.

Fixpoint eval (V : valuation) (p : prop) :=
  match p with
```

```
  | Atom n => V n
  | Neg q => ~~ eval V q
  | Disj q1 q2 => eval V q1 || eval V q2
  | Conj q1 q2 => eval V q1 && eval V q2
  end.

Definition sv x := if x == 3 then true else false.
Eval compute in eval sv sp.

(* 恒真式の定義 *)
Definition tautology p := forall v, eval v p = true.

(* 古典論理なので、っは要らない  *)
Lemma imp_disj p1 p2 v : eval v (Imp p1 p2) = eval v p1 ==> eval v p2.
Proof. move=> /=. by case: (eval v p1). Qed.

(* 否定標準形の定義 *)
Fixpoint is_nnf p :=
  match p with
  | Atom _ => true
  | Neg (Atom _) => true
  | Neg _ => false
  | Conj p1 p2 => is_nnf p1 && is_nnf p2
  | Disj p1 p2 => is_nnf p1 && is_nnf p2
  end.

(* 変換関数 *)
Fixpoint nnf neg p :=
 match p with
 | Atom a => if neg then Neg p else p
 | Neg p => nnf (~~ neg) p
 | Conj p1 p2 =>
   let p1' := nnf neg p1 in let p2' := nnf neg p2 in
   if neg then Disj p1' p2' else Conj p1' p2'
 | Disj p1 p2 =>
   let p1' := nnf neg p1 in let p2' := nnf neg p2 in
   if neg then Conj p1' p2' else Disj p1' p2'
 end.

Eval compute in nnf false sp.

(* 変換の正しさの証明 *)

Theorem nnf_is_nnf neg p : is_nnf (nnf neg p).
Proof.
  elim: p neg => [n | p IH | p1 IH1 p2 IH2 | p1 IH1 p2 IH2] [] //=;
  by rewrite IH1 /=.
Qed.

Theorem nnf_correct neg p v : eval v (nnf neg p) = neg (+) eval v p.
Proof.
  elim: p neg => [n | p IH | p1 IH1 p2 IH2 | p1 IH1 p2 IH2] [] //=;
  rewrite ?IH ?negbK ?(IH1,IH2) //=; by case: eval.
Qed.
```

```
(* Conjunctive normal form: 論理積標準形の定義 *)

(* リテラル *)
Definition lit : Set := nat * bool. (* false: Neg *)

(* Clause = リテラルの論理和 *)
Definition clause := list lit.

Definition eval_lit (v : valuation) (l : lit) :=
  if l.2 then v l.1 else ~~ v l.1.

Definition eval_clause (v : valuation) (cl : clause) : bool :=
  has (eval_lit v) cl.                                    (* 1つがtrue *)

(* 論理積標準形 = Clause の論理積 *)
Definition eval_cnf (v : valuation) (l : list clause) : bool :=
  all (eval_clause v) l.                                  (* 全てがtrue *)

(* 変換関数 *)
Fixpoint neg_lit (x : lit) := (x.1, ~~ x.2).

Definition disj (l1 l2 : list clause) : list clause :=
  [seq c1 ++ c2 | c1 <- l1, c2 <- l2].              (* 2つのリストの直積 *)

Fixpoint cnf (p : prop) : list clause :=
 match p with
 | Atom a => [:: [:: (a, true)]]
 | Neg (Atom a)  => [:: [:: (a,false)]]
 | Neg _ => [::]                                   (* nnfなので有り得ない *)
 | Conj p1 p2 => cnf p1 ++ cnf p2
 | Disj p1 p2 => disj (cnf p1) (cnf p2)
 end.

(* 補題 *)
Lemma eval_clause_cat v c1 c2 :
  eval_clause v (c1 ++ c2) = eval_clause v c1 || eval_clause v c2.
Proof. by rewrite /eval_clause has_cat. Qed.

Lemma eval_cnf_cat v l1 l2 :
  eval_cnf v (l1 ++ l2) = eval_cnf v l1 && eval_cnf v l2.
Admitted.

orb_andl : left_distributive orb andb
orb_andr : right_distributive orb andb
orbT : forall b : bool, b || true
Lemma disj_correct v l1 l2 :
  eval_cnf v (disj l1 l2) = eval_cnf v l1 || eval_cnf v l2.
Admitted.

(* 論理積標準形は意味を変えない *)
andbT : right_id true andbCheck andbT.
andbF : right_zero false andb
Theorem cnf_correct v p : is_nnf p -> eval_cnf v (cnf p) = eval v p.
Admitted.
```

```
(* 恒真式の機械的な判定 *)

Fixpoint tauto_clause (c : clause) :=
 if c is a :: c' then (neg_lit a \in c') || tauto_clause c' else false.

Definition tauto_cnf (l : list clause) := all tauto_clause l.

Eval compute in tauto_cnf (cnf (nnf false sp)).

(* 補題 *)
Lemma eval_neg_lit v a : eval_lit v (neg_lit a) = ~~ eval_lit v a.
Admitted.

Lemma eval_clause_true v (a : lit) (c : clause) :
  a \in c -> eval_lit v a -> eval_clause v c.
Admitted.

Lemma tautology_notin a c :
  neg_lit a \notin c ->
  (forall v, eval_clause v (a :: c)) -> (forall v, eval_clause v c).
Proof.
  move=> /= Hna Hv v.                        (* a を否定する valuation を作る *)
  set v' := fun n => if n == a.1 then ~~ a.2 else v n.
  move: {Hv}(Hv v').
  case /boolP: (eval_lit v' a) => /=.
    rewrite /eval_lit /v' eqxx.
    by case: a.2.
  move=> Ha; elim: c Hna => //= b c IH.
  rewrite inE negb_or => /andP [Hb Hc] /orP [] H; last by rewrite IH // orbT.
  suff : eval_lit v' b -> eval_lit v b by move ->.
  rewrite /eval_lit /v'.
  case Ha2: (~~a.2); move: Ha2; case: ifP; case: ifP => // /eqP Hb1 <- Ha2;
  by rewrite (surjective_pairing a) (surjective_pairing b) Hb1 -Ha2 eqxx in Hb.
Qed.

(* Clause の恒真判定の正しさ *)
Lemma tauto_clause_ok c : tauto_clause c <-> (forall v, eval_clause v c).
Admitted.

Lemma tauto_cnf_lem l : tauto_cnf l <-> forall v, eval_cnf v l.
Proof.
  split. move=> /allP H v; apply/allP => c /H /tauto_clause_ok. by apply.
Admitted.

(* tauto_cnf は恒真式を正しく判定している *)
addFb : left_id false addb
Theorem tauto_cnf_ok p : tautology p <-> tauto_cnf (cnf (nnf false p)).
Admitted.
```

練習問題 4.1 証明の中の `Admitted` を `Qed` に変えよ.