Jacques Garrigue, 2014 年 6 月 16 日

命題論理の意味論

1 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，$False$ 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．

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

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．

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

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．

2 実装

```
(* 命題論理 *)
Require Import Bool List Arith Recdef Omega.

Definition atom := nat.                                          (* 論理変数 *)
Definition atom_eq_dec : forall a b : atom, {a=b} + {a<>b} := eq_nat_dec.
Notation "x == y" := (atom_eq_dec x y) (at level 70).       (* 演算子にする *)

(* 命題の定義 *)
Inductive prop : Set :=
  | Atom : atom -> prop
  | Neg : prop -> prop
  | Conj : prop -> prop -> prop
  | Disj : prop -> prop -> prop
  | Imp : prop -> prop -> prop.

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))).

(* 意味論 *)
Definition valuation := atom -> bool.

Fixpoint eval (v : valuation) (p : prop) : bool :=
  match p with
  | Atom a => v a
  | Neg p1 => negb (eval v p1)
  | Conj p1 p2 => eval v p1 && eval v p2
  | Disj p1 p2 => eval v p1 || eval v p2
  | Imp p1 p2 => if eval v p1 then eval v p2 else true
  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.

(* 古典論理の意味論なので、つは要らない *)
Parameter imp_disj : forall p1 p2 v,
  eval v (Imp p1 p2) = eval v (Disj (Neg p1) p2).

(* 否定標準形の定義 *)
Inductive is_nnf : prop -> Prop :=
  | nnfAtom : forall a, is_nnf (Atom a)
  | nnfNegAtom : forall a, is_nnf (Neg (Atom a))
  | nnfConj : forall p1 p2, is_nnf p1 -> is_nnf p2 -> is_nnf (Conj p1 p2)
  | nnfDisj : forall p1 p2, is_nnf p1 -> is_nnf p2 -> is_nnf (Disj p1 p2).

(* 変換のための測定 *)
Fixpoint nnf_sz (p : prop) :=
  match p with
  | Atom a     => 1
  | Neg p1     => 1 + nnf_sz p1 + nnf_sz p1
  | Conj p1 p2 => 2 + nnf_sz p1 + nnf_sz p2
  | Disj p1 p2 => 2 + nnf_sz p1 + nnf_sz p2
  | Imp p1 p2  => 3 + nnf_sz p1 + nnf_sz p1 + nnf_sz p2(* Disj (Neg p1) p2 *)
  end.
```

```
(* 変換関数 *)
Function nnf (p : prop) {measure nnf_sz p} : prop :=
  match p with
  | Atom a           => Atom a
  | Neg (Atom a)     => Neg (Atom a)
  | Neg (Neg p1)     => nnf p1
  | Neg (Conj p1 p2) => nnf (Disj (Neg p1) (Neg p2))
  | Neg (Disj p1 p2) => nnf (Conj (Neg p1) (Neg p2))
  | Neg (Imp p1 p2)  => nnf (Conj p1 (Neg p2))
  | Conj p1 p2       => Conj (nnf p1) (nnf p2)
  | Disj p1 p2       => Disj (nnf p1) (nnf p2)
  | Imp p1 p2        => Disj (nnf (Neg p1)) (nnf p2)
  end.
  simpl; intros; omega.  simpl; intros; omega.
  simpl; intros; omega.  simpl; intros; omega.
  simpl; intros; omega.  simpl; intros; omega.
  simpl; intros; omega.  simpl; intros; omega.
  simpl; intros; omega.  simpl; intros; omega.
Defined.

Eval compute in nnf sp.

(* 変換の正しさの証明 *)
Hint Constructors is_nnf.
Parameter nnf_is_nnf : forall p, is_nnf (nnf p).
Parameter nnf_correct : forall p v, eval v (nnf p) = eval v p.

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

(* リテラル *)
Inductive lit : Set :=
  | Posi : atom -> lit
  | Nega : atom -> lit.

Fixpoint eval_lit (v : valuation) (l : lit) := ...

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

Fixpoint eval_clause (v : valuation) (l : clause) : bool :=
  match l with
  | nil     => false
  | x :: l' => eval_lit v x || eval_clause v l'
  end.

(* 論理積標準形 = Clause の論理積 *)
Fixpoint eval_cnf (v : valuation) (l : list clause) : bool := ...

(* 変換関数 *)
Fixpoint neg_lit x :=
  match x with
  | Posi a => Nega a
  | Nega a => Posi a
  end.

Fixpoint disj (l1 l2 : list clause) :=
  match l1 with
  | nil => nil
  | c1 :: l => map (fun c2 => c1 ++ c2) l2 ++ disj l l2
  end.
```

```
Fixpoint cnf (p : prop) : list clause :=
  match p with
  | Atom a => (Posi a :: nil) :: nil
  | Neg p1 => map (map neg_lit) (cnf p1)                    (* 一点集合のみ *)
  | Conj p1 p2 => cnf p1 ++ cnf p2
  | Disj p1 p2 => disj (cnf p1) (cnf p2)
  | Imp p1 p2 => nil                                        (* nnf にはない *)
  end.

Eval compute in (cnf (nnf sp)).

(* 補題 *)
Check orb_assoc.
Parameter eval_clause_app : forall v c1 c2,
  eval_clause v (c1 ++ c2) = eval_clause v c1 || eval_clause v c2.

Check andb_assoc.
Parameter eval_cnf_app : forall v l1 l2,
  eval_cnf v (l1 ++ l2) = eval_cnf v l1 && eval_cnf v l2.

Check orb_andb_distrib_l.
Check orb_andb_distrib_r.
Parameter disj_correct : forall v l1 l2,
  eval_cnf v (disj l1 l2) = eval_cnf v l1 || eval_cnf v l2.

Parameter cnf_correct : forall v p, is_nnf p -> eval_cnf v (cnf p) = eval v p.

(* 恒真式の機械的な判定 *)
Definition lit_eq_dec : forall a b : lit, {a=b}+{a<>b}.
  pose atom_eq_dec.                                  (* 定義を仮定に置く *)
  decide equality.
Defined.
Print lit_eq_dec.

Fixpoint tauto_clause (c : clause) :=
  match c with
  | nil => false
  | a :: c' =>
    if in_dec lit_eq_dec (neg_lit a) c then true else tauto_clause c'
  end.

Fixpoint tauto_cnf (l : list clause) :=
  match l with
  | nil => true
  | c :: l' => tauto_clause c && tauto_cnf l'
  end.

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

Parameter tauto_clause_ok :
  forall c, tauto_clause c = true <-> (forall v, eval_clause v c = true).

Parameter tauto_cnf_ok :
  forall p, tautology p <-> tauto_cnf (cnf (nnf p)) = true.
```

練習問題 2.1　*1.* eval_lit と eval_cnf を定義せよ.

2. 証明の中の Parameter を Theorem に変えよ．ただし、tauto_clause_ok は後回しにするべきだ.