Jacques Garrigue, 2013 年 7 月 26 日

導出原理 (Resolution)

1 述語論理の意味論

第 3 回に挙げた述語論理の構文を再掲する

$$
\begin{array}{lcll}
t & ::= & x & \text{項変数} \\
 & | & c & \text{項定数} \\
 & | & f(t_1, \ldots, t_n) & \text{項関数} \\
P, Q & ::= & \ldots & \text{命題} \\
 & | & p(t_1, \ldots, t_n) & \text{述語} \\
 & | & \forall x.P & \text{全称} \\
 & | & \exists x.P & \text{存在} \\
 & | & t_1 = t_2 & \text{等価性}
\end{array}
$$

述語論理の意味論は二つの部分に分けられる。

- ストラクチャー (構造)S は各述語と関数・定数記号の意味を与える。ある項の意味領域 U を決め，各述語 p に対して $U^n \to \{\mathsf{true}, \mathsf{false}\}$ の関数を与え，各関数記号 f に対して $U^n \to U$ の関数を与える。p の意味を $[\![p]\!]$、f や c の意味を $[\![f]\!], [\![c]\!]$ と書く。
- 項の自由変数に対して、U の値を割り当てる関数 $v : \mathit{Vars} \to U$。

S が与えられたら，意味を以下に定める。

$$[\![x]\!]_v = v(x) \qquad [\![c]\!]_v = [\![c]\!] \qquad [\![f(t_1, \ldots, \tau_n)]\!] = [\![f]\!]([\![t_1]\!]_v, \ldots, [\![t_n]\!]_v)$$

$$[\![p(t_1, \ldots, t_n)]\!]_v = [\![p]\!]([\![t_1]\!]_v, \ldots, [\![t_n]\!]_v) \qquad [\![t_1 = t_2]\!]_v = [[\![t_1]\!]_v = [\![t_2]\!]_v]$$

$$[\![P \supset Q]\!]_v = \text{if } [\![P]\!]_v \text{ then } [\![Q]\!]_v \text{ else } \mathsf{true} \qquad \text{他も命題論理と同じ}$$

$$[\![\forall x.P]\!]_v = [\{a \in U \mid [\![P]\!]_{v[x \mapsto a]} = \mathsf{true}\} = U] \qquad [\![\exists x.P]\!]_v = [\{a \in U \mid [\![P]\!]_{v[x \mapsto a]} = \mathsf{true}\} \neq \emptyset]$$

特定の S において v に依存せず $[\![P]\!]_v = \mathsf{true}$ ならば $S \models P$ と書く。全ての S において $S \models P$ ならば、P は恒真式といい、$\models P$ と書く。全ての S において、ある v で $[\![P]\!]_v = \mathsf{true}$ ならば、P が充足可能である。

2 Herbrand 領域

恒真式を決めるには、任意のストラクチャーを考えなければならず、一見扱いにくそうである。

しかし、S として Herbrand 領域 H を使えば、H で成り立つ命題は任意の S でも成り立つ。

Herbrand 領域の U は単一化で見た木構造を使い、述語と関数記号はそのままの木構造を作るだけである: $[\![p]\!](a_1, \ldots, a_n) = p(a_1, \ldots, a_n)$, $[\![f]\!](a_1, \ldots, a_n) = f(a_1, \ldots, a_n)$.

H においてある変数割当 v で $[\![P]\!]_v = \mathsf{true}$ ならば、P は充足可能。それを利用すると、充足可能性が半決定可能になる。理論的には、P の中の変数に対して割当を作って行けばいい。可能な割当の数は加算無限なので、成功するかどうか分からないが、探す戦略を工夫すれば、解があればいつか見つかる。

3 Horn Clause

今回は限られた設定を考える。Horn clause とは、正のリテラルが一つしかない Clause をいう。ここでリテラルは $p(t_1, \ldots, t_n)$ あるいは $\neg p(t_1, \ldots, t_n)$ のみである。ただし、変数は $\forall$ で閉じる。論理式として書くと

$$\forall x_1 \ldots x_m. \neg L_1 \vee \ldots \vee \neg L_n \vee L$$

あるいは

$$\forall x_1 \ldots x_m. (L_1 \wedge \ldots \wedge L_n) \supset L$$

(L_i は全て正のリテラル)

Horn clause の例: $\forall x. human(x) \supset mortal(x)$, $human(Socrates)$

そこで考える問題は、特定の Horn Clause の集合を仮定したときに、ある正のリテラル (の列) が充足可能かである。そのリテラルの列をゴール (goal) という。

$$(C_1 \wedge \ldots \wedge C_n) \supset (G_1 \wedge \ldots \wedge G_m)$$

Goal の例: $mortal(Socrates)$.

4 導出原理

この簡単な設定では、可能な証明方法は非常に限られている。具体的には、各 G_i はある C_j の結論と一致しなければならない。ただし、Clause の変数は量化されているので、instance(例) を取らなければならない。G_i を C_j の instance の前提に置き換えたゴールから元のゴールが導ける。

また、充足可能性を考えるときに G_i の自由変数が具体化できるので、G_i と C_j の結論 L を単一化できれば十分である。

Robinson の導出原理では、$\mathcal{U}(G_i, L) = \sigma$ とすると、新しいゴールは

$$\sigma(G_1 \wedge \ldots \wedge G_{i-1} \wedge L_{j1} \wedge \ldots L_{jn_j} \wedge G_{i+1} \wedge \ldots \wedge G_m$$

になる（Clause は何度でも使えるので変わらない)

さらに、複数の Clause が一致する可能性があるので、新しいゴールへの転移は非決定的に行う必要がある。

5 実装

今日の実装は `http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2013_SS/` においてある。`unify.v` と `coq14.v` をダウンロードして、端末で `coqc unify.v` で単一化をコンパイルしてから `coq14.v` で作業する。

```
(* unify.v *)
Require String.
Open Scope string_scope.

(* 定数・関数記号 *)
Notation symbol := String.string.
Definition symbol_dec := String.string_dec.

(* 記号を直接に文字列として書く *)
Definition atree := Fork (Sym "a") (Fork (Var 0) (Var 1)).
Definition asubs := (0, Fork (Var 1) (Sym "b")) :: (1, Var 2) :: nil.
Eval compute in subs_list asubs atree.
```

```
(* coq14.v *)
Require Import Arith List ListSet Omega.
Require Import unify. (* 単一化 *)

(* 述語 *)
Inductive pred : Set := Pred : symbol -> tree -> pred.
(* Horn clause *)
Inductive clause : Set := Clause : pred -> list pred -> clause.

Definition hmc := Clause (Pred "Mortal" (Var 0)) (Pred "Human" (Var 0) :: nil).

(* 代入 *)
Definition substi := list (var * tree).

(* 各ステップでのゴール．それまでの代入としよう変数を覚える必要がある *)
(* Record は場合が一つの Inductive と同じだが、射影関数も定義される *)
Record state : Set :=
  State { st_subs : substi; st_count : var; st_goal : list pred }.

(* Clause の名前を確認 *)
Definition clause_name_eq s cl :=
  match cl with Clause (Pred q _) _ =>
    if symbol_dec s q then true else false
  end.

(* 代入 *)
Definition subs_pred s (P : pred) :=
  match P with Pred p t => Pred p (subs_list s t) end.
Definition subs_clause s (c : clause) :=
  match c with Clause p pl => Clause (subs_pred s p) (map (subs_pred s) pl) end.

(* 変数 *)
Definition vars_pred p :=
  match p with Pred _ t => vars t end.
Definition vars_preds l :=
  fold_right union nil (map vars_pred l).
Definition vars_clause c :=
  match c with Clause p pl => vars_preds (p :: pl) end.

(* 長さ n の新しい変数のリスト *)
Fixpoint fresh v (n : nat) : list tree :=
  match n with
  | 0 => nil
  | S n' => Var v :: fresh (v+1) n'
  end.

(* 例を作る．使った変数も知らせる *)
Definition instance (v : var) (c : clause) : var * clause :=
  let vs := vars_clause c in
  let s := combine vs (fresh v (length vs)) in
  (v + length vs, subs_clause s c).

(* 例 *)
Eval compute in instance 3 hmc.

Section Resolve.
Variables clauses : list clause. (* clause の集合を固定する *)

(* 導出可能性の定義 *)
Inductive derivable : list pred -> Prop :=
  | D_true : derivable nil
  | D_conj : forall p1 pl1 pl2 s c,
    In c clauses ->
    subs_clause s c = Clause p1 pl1 ->
    derivable pl1 ->
    derivable pl2 ->
    derivable (p1 :: pl2).
```

```
Hint Constructors derivable.

Theorem derivable_app : forall pl1 pl2,
  (derivable pl1 /\ derivable pl2) <-> derivable (pl1 ++ pl2).
Admitted.

(* 特定の述語を結論に持つ clause を集める *)
Definition get_clauses s := filter (clause_name_eq s) clauses.

Fixpoint map_option {A B:Type} (f : A -> B) (a : option A) : option B :=
  match a with
  | None => None
  | Some x => Some (f x)
  end.

(* 述語の引数を単一化する．代入は後で使う *)
Definition unify_clause (t1 : tree) (c : clause) :=
  match c with Clause (Pred _ t2) pl =>
    map_option (pair pl) (unify t1 t2)
  end.

Definition make_state s pl v' (pls' : list pred * substi) :=
  let (pl', s') := pls' in
  State (s ++ s') v' (map (subs_pred s') (pl' ++ pl)).

(* 特定の clause を使ってみる *)
Definition expand_clause s v t1 pl (c : clause) :=
  let (v', c') := instance v c in
  map_option (make_state s pl v') (unify_clause t1 c').

(* 結果から None を消す *)
Fixpoint filter_some {A:Type} (l : list (option A)) : list A :=
  match l with
  | nil => nil
  | None :: l' => filter_some l'
  | Some a :: l' => a :: filter_some l'
  end.

(* 全ての clause を使ってみる *)
Definition expand_clauses s v t1 pl cls :=
  filter_some (map (expand_clause s v t1 pl) cls).

(* 一つの状態に対して導出原理を適用する *)
Definition expand_one (st : state) :=
  match st with
  | State s v nil => st :: nil
  | State s v (Pred p t1 :: pl) =>
    expand_clauses s v t1 pl (get_clauses p)
  end.

(* 全ての平行して扱っている状態に導出原理を適用する *)
Definition expand_all (sts : list state) : list state :=
  fold_right (@app _) nil (map expand_one sts).

(* n 回繰り返す *)
Fixpoint iter {A : Type} (n : nat) (f : A -> A) (a : A) : A :=
  match n with
  | 0    => a
  | S n' => iter n' f (f a)
  end.

Parameter expand_all_app : forall l1 l2,
  expand_all (l1 ++ l2) = expand_all l1 ++ expand_all l2.
Parameter iter_expand_all_app : forall n l1 l2,
  iter n expand_all (l1 ++ l2) = iter n expand_all l1 ++ iter n expand_all l2.
```

```
(* get_clauses の仕様 *)
Lemma get_clauses_ok : forall s,
  incl (get_clauses s) clauses /\
  forall p t1 pl, In (Clause (Pred p t1) pl) (get_clauses s) -> p = s.
Proof.
  unfold get_clauses.
  split.
    intros x Hx.
    apply -> filter_In in Hx.
    intuition.
Admitted.

Parameter subs_pred_app_map : forall s s' l,
  map (subs_pred (s ++ s')) l = map (subs_pred s') (map (subs_pred s) l).

Lemma expand_clause_sound : forall st s0 v s1 t pl s t' l',
  In (Clause (Pred s1 t') l') clauses ->
  expand_clause s0 v t pl (Clause (Pred s1 t') l') = Some st ->
  derivable (map (subs_pred s) (st_goal st)) ->
  exists s', st_subs st = s0 ++ s' /\
    derivable (map (subs_pred (s' ++ s)) (Pred s1 t :: pl)).
Proof.
  unfold expand_clause.
  unfold instance.
  intros.
  set (sa := combine _ _) in H0.
  unfold unify_clause in H0.
  simpl in H0.
  case_eq (unify t (subs_list sa t')); intros; rewrite H2 in *;
    try discriminate.
  simpl in H0.
Admitted.

Theorem expand_all_sound : forall s st' st,
  In st' (expand_all (st :: nil)) ->
  derivable (map (subs_pred s) (st_goal st')) ->
  exists s', st_subs st' = st_subs st ++ s' /\
    derivable (map (subs_pred (s' ++ s)) (st_goal st)).
Proof.
  destruct st as [s0 v pl].
  destruct pl.
    intros.
    simpl in H; destruct H.
      subst; simpl.
      exists nil.
      rewrite <- app_nil_end.
      auto.
    elim H.
  unfold expand_all.
  simpl; intros.
  destruct p.
  rewrite <- app_nil_end in H.
  destruct (get_clauses_ok s1).
  simpl.
  induction (get_clauses s1); simpl in *.
Admitted.

Lemma cons_app : forall {A:Type} (a:A) l, a :: l = (a :: nil) ++ l.
Proof. reflexivity. Qed.

Lemma iter_expand_inv : forall n st l,
  In st (iter n expand_all l) ->
  exists st', In st' l /\ In st (iter n expand_all (st' :: nil)).
Proof.
  induction l; simpl; intros.
    elimtype False.
    induction n; intuition.
```

```
  rewrite cons_app in H.
Admitted.

Theorem iter_expand_sound : forall n s st' st,
  In st' (iter n expand_all (st :: nil)) ->
  derivable (map (subs_pred s) (st_goal st')) ->
  exists s', st_subs st' = st_subs st ++ s' /\
    derivable (map (subs_pred (s' ++ s)) (st_goal st)).
Admitted.

Corollary soundness : forall n s v v' pl,
  In (State s v' nil) (iter n expand_all (State nil v pl :: nil)) ->
  derivable (map (subs_pred s) pl).
Admitted.
End Resolve.

Extraction "resolve.ml" iter expand_all.

Definition myclauses :=
  hmc :: Clause (Pred "Human" (Sym "Socrates")) nil :: nil.

Definition goal := State nil 0 (Pred "Mortal" (Sym "Socrates") :: nil) :: nil.

Eval compute in iter 1 (expand_all myclauses) goal.
Eval compute in iter 2 (expand_all myclauses) goal.

(* おまけ：余帰納法による解の定義 *)
Section Search.
Variable clauses : list clause.

Require Import Streams.
Print Stream.

Fixpoint get_solutions (ls : list state) : list substi * list state :=
  match ls with
  | nil => (nil, nil)
  | State s n nil :: rem =>
    let (ss, ns) := get_solutions rem in (s :: ss, ns)
  | st :: rem =>
    let (ss, ns) := get_solutions rem in (ss, st :: ns)
  end.

CoFixpoint search (l : list state) :=
  let (sols, next) := get_solutions l in
  Cons sols (search (expand_all clauses next)).

End Search.

Definition sols := search myclauses goal.
Eval compute in Str_nth 0 sols.
Eval compute in Str_nth 1 sols.
Eval compute in Str_nth 2 sols.

Extraction "resolve.ml" search iter.
```