Jacques Garrigue, 2023 年 7 月 19 日

単一化とその応用

1　単一化

単一化は二つの項を同じものにする代入を求めるアルゴリズムである．一階の項に関して，単一化は最適な代入を見つけてくれるという結果が知られている．

定義 1 ある代入 σ_1 が代入 σ_2 **より一般的**であるとは，σ_{12} が存在し，$\sigma_2 = \sigma_{12} \circ \sigma_1$ であることをいう．

定義 2 単一化問題 $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ に対して，$\sigma(t_i) = \sigma(t'_i)$ であるとき，σ はその単一化問題の**単一子**だという．

定理 1 任意の単一化問題に対して，解が存在するときには**最も一般的な単一子**を返し，存在しないときにはそれを報告するアルゴリズムが存在する．

具体的には，アルゴリズム $\mathcal{U}$ を以下の書き換え規則で定義できる．ここでは定数記号を 0 引数の関数記号と同一視する．

$$
\begin{array}{lllll}
1 & (E \cup \{f(t_1,\ldots,t_n) = f(t'_1,\ldots,t'_n)\}, \sigma) & \to & (E \cup \{t_1 = t'_1, \ldots, t_n = t'_n\}, \sigma) & \\
2 & (E \cup \{f(t_1,\ldots,t_n) = g(t'_1,\ldots,t'_m)\}, \sigma) & \to & \bot & f \neq g \\
3 & (E \cup \{x = t\}, \sigma) & \to & ([t/x]E, [t/x] \circ \sigma) & x \notin vars(t) \\
4 & (E \cup \{x = x\}, \sigma) & \to & (E, \sigma) & \\
5 & (E \cup \{x = t\}, \sigma) & \to & \bot & x \in vars(t) \wedge x \neq t
\end{array}
$$

(E, id) から始まり，書き換えを繰り返すことで，$(\emptyset, \sigma)$ になった場合，$\mathcal{U}(E) = \sigma$ は E の最も一般的な単一子になる．$\bot$ になった場合，単一子が存在しない．

停止性を確認するために，$\mu(E) = \langle E$ の変数の数, E の項の大きさの和 $\rangle$ の上の辞書式順序を考える:

$$\langle n, s\rangle < \langle n', s'\rangle \Leftrightarrow n < n' \vee (n \leq n' \wedge s < s')$$

各規則 $(E, \sigma) \to (E', \sigma')$ について $\mu(E) > \mu(E')$ ならば，順序の整礎性より，書き換えが必ず止まる．(1) と (4) では n が増えずに s が減るので，なりたつ．(3) では n が減り，(2) と (5) ではただちに止まるので，アルゴリズムが必ず止まる．

2　実装

今日の実装と前回の答えは `http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2023_SS/` においてある．

```
(* 単一化 *)

From mathcomp Require Import all_ssreflect.
Require String.
Import String.StringSyntax.
Open Scope string_scope.
```

```
(* 変数 *)
Definition var := nat.

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

(* 項は木構造 *)
Inductive tree : Set :=
  | Var : var -> tree
  | Sym : symbol -> tree
  | Fork : tree -> tree -> tree.

(* 自動的に等価性を生成する *)
Scheme Equality for tree.

(* tree を eqType として登録する *)
Lemma tree_eq_boolP : Equality.axiom tree_eq_dec.
Proof. move=> x y. case: tree_eq_dec => //= H; by constructor. Qed.
Definition tree_eq_mixin := EqMixin tree_eq_boolP.
Canonical tree_eqType := Eval hnf in EqType _ tree_eq_mixin.

(* [t/x] t' *)
Fixpoint subs (x : var) (t t' : tree) : tree :=
  match t' with
  | Var v => if v == x then t else t'
  | Sym b => Sym b
  | Fork t1 t2 => Fork (subs x t t1) (subs x t t2)
  end.

(* 代入は変数代入の繰り返し *)
Definition subs_list (s : list (var * tree)) t : tree :=
  foldl (fun t (p : var * tree)  => subs p.1 p.2 t) t s.

(* 単一子の定義 *)
Definition unifies s (t1 t2 : tree) := subs_list s t1 = subs_list s t2.

(* 例 : (a, (x, y)) [x := (y, b)] [y := z] = (a, ((z,b), z)) *)
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.

(* 和集合 *)
Fixpoint union (vl1 vl2 : list var) :=
  if vl1 is v :: vl then
    if v \in vl2 then union vl vl2 else union vl (v :: vl2)
  else vl2.

Lemma in_union_or v vl1 vl2 :
  v \in union vl1 vl2 = (v \in vl1) || (v \in vl2).
Proof. elim: vl1 vl2 => //= x vl IH vl2. Admitted.

(* 完全性のために必要 *)
Lemma uniq_union vl1 vl2 : uniq vl2 -> uniq (union vl1 vl2).
Proof. elim: vl1 vl2 => //=. Admitted.

(* 出現変数 *)
Fixpoint vars (t : tree) : list var :=
  match t with
  | Var x => [:: x]
  | Sym _ => nil
  | Fork t1 t2 => union (vars t1) (vars t2)
  end.

(* 出現しない変数は代入されない *)
Lemma subs_same v t' t : v \notin (vars t) -> subs v t' t = t.
Proof.
```

```
  elim: t => //= [x | t1 IH1 t2 IH2].
  - by rewrite inE eq_sym => /negbTE ->.
  - by rewrite in_union_or negb_or => /andP[] /IH1 -> /IH2 ->.
Qed.

Definition may_cons (x : var) (t : tree) r := omap (cons (x,t)) r.
Definition subs_pair x t (p : tree * tree) := (subs x t p.1, subs x t p.2).

(* 単一化 *)
Section Unify1.

(* 代入を行ったときの再帰呼び出し *)
Variable unify2 : list (tree * tree) -> option (list (var * tree)).

(* 代入して再帰呼び出し．x は t に現れてはいけない *)
Definition unify_subs x t r :=
  if x \in vars t then None else may_cons x t (unify2 (map (subs_pair x t) r)).

(* 代入をせずに分解 *)
Fixpoint unify1 (h : nat) (l : list (tree * tree))
  : option (list (var * tree)) :=
  if h is h'.+1 then
    match l with
    | nil                  => Some nil
    | (Var x, Var x') :: r =>
                  if x == x' then unify1 h' r else unify_subs x (Var x') r
    | (Var x, t) :: r      => unify_subs x t r
    | (t, Var x) :: r      => unify_subs x t r
    | (Sym b, Sym b') :: r => if symbol_dec b b' then unify1 h' r else None
    | (Fork t1 t2, Fork t1' t2') :: r
                           => unify1 h' ((t1, t1') :: (t2, t2') :: r)
    | _                    => None
    end
  else None.
End Unify1.

(* 等式の大きさの計算 *)
Fixpoint size_tree (t : tree) : nat :=
  if t is Fork t1 t2 then 1 + size_tree t1 + size_tree t2 else 1.

Definition size_pairs (l : list (tree * tree)) :=
  sumn [seq size_tree p.1 + size_tree p.2 | p <- l].

(* 代入したときだけ再帰 *)
Fixpoint unify2 (h : nat) l :=
  if h is h'.+1 then unify1 (unify2 h') (size_pairs l + 1) l else None.

Fixpoint vars_pairs (l : list (tree * tree)) : list var :=
  match l with
  | nil => nil                                          (* 集合を後部から作るようにする *)
  | (t1, t2) :: r => union (union (vars t1) (vars t2)) (vars_pairs r)
  end.

(* 変数の数だけ unify2 を繰り返す *)
Definition unify t1 t2 :=
  let l := [:: (t1,t2)] in unify2 (size (vars_pairs l) + 1) l.

(* 例 *)
Eval compute in unify (Sym "a") (Var 1).

Eval compute in
  unify (Fork (Sym "a") (Var 0)) (Fork (Var 1) (Fork (Var 1) (Var 2))).

(* 全ての等式の単一子 *)
Definition unifies_pairs s (l : list (tree * tree)) :=
  forall t1 t2, (t1,t2) \in l -> unifies s t1 t2.
```

```
(* subs_list と Fork が可換 *)
Lemma subs_list_Fork s t1 t2 :
  subs_list s (Fork t1 t2) = Fork (subs_list s t1) (subs_list s t2).
Proof. elim: s t1 t2 => //. Admitted.

(* unifies_pairs の性質 *)
Lemma unifies_pairs_same s t l :
  unifies_pairs s l -> unifies_pairs s ((t,t) :: l).
Proof. move=> H t1 t2; rewrite inE => /orP[]. Admitted.

Lemma unifies_pairs_swap s t1 t2 l :
  unifies_pairs s ((t1, t2) :: l) -> unifies_pairs s ((t2, t1) :: l).
Admitted.

Lemma unify_subs_sound h v t l s :
  (forall s l, unify2 h l = Some s -> unifies_pairs s l) ->
  unify_subs (unify2 h) v t l = Some s ->
  unifies_pairs s ((Var v, t) :: l).
Proof.
  rewrite /unify_subs.
  case Hocc: (v \in _) => // IH.
  case Hun: (unify2 _ _) => [s'|] //= [] <-.
Admitted.

(* unify2 の健全性 *)
Theorem unify2_sound h s l :
  unify2 h l = Some s -> unifies_pairs s l.
Proof.
  elim: h s l => //= h IH s l.
  move: (size_pairs l + 1) => h'.
  elim: h' l => //= h' IH' [] //.
  move=> [t1 t2] l.
  destruct t1, t2 => //.
  (* VarVar *)
- case: ifP.
    move/eqP => <- /IH'.
    by apply unifies_pairs_same.
  intros; by apply (unify_subs_sound h).
  (* VarSym *)
- intros; by apply (unify_subs_sound h).
Admitted.

(* 単一化の健全性 *)
Corollary soundness t1 t2 s :
  unify t1 t2 = Some s -> unifies s t1 t2.
Admitted.

(* 完全性 *)

(* s が s' より一般的な単一子である *)
Definition moregen s s' :=
  exists s2, forall t, subs_list s' t = subs_list s2 (subs_list s t).

(* 単一化の完全性 *)
Corollary unify_complete s t1 t2 :
  unifies s t1 t2 ->
  exists s1, unify t1 t2 = Some s1 /\ moregen s1 s.
Admitted.
```

練習問題 2.1 証明の中の Admitted を Qed に変えよ.
unify_subs_sound が最も重要な補題である.