一階の単一化を証明する

Jacques Garrigue

http://www.math.nagoya-u.ac.jp/~garrigue/home-j.html

単一化とは

- ◇ 元々は一階述語論理の証明論の完全性を証明するために Jacques Herbrand先輩が考えたアルゴリズム
- ◇ 二つの項を等価にする変数の割り当てを見つける
- ◇ MLなどの型推論の基礎になっている
- ◇ Coqなどの高階論理に基づいたシステムでは高階単一化を使う
- ◇ パターンマッチングは変数が片方にしかない単一化として見ることができる
- ◇ [Paulson85] で既にコンピューターで証明されている定番

単一化の例

a, b, cは定数記号、f, g, hは関数記号、x, y, zは変数

$$t_1 = f(x, g(a, y), y)$$

$$t_2 = f(z, z, b)$$

単一化の解は変数を割り当てる**代入**

$$\theta = \begin{bmatrix} y \mapsto b \\ z \mapsto g(a, b) \\ x \mapsto g(a, b) \end{bmatrix}$$

$$\theta(t_1) = \theta(t_2) = f(g(a, b), g(a, b), b)$$

単一化の性質

一階の単一化は良い性質を持っている

◇ 最適な割り当てを見つける**完全なアルゴリズム**が存在する

◇ そのアルゴリズムが**最も一般的**な代入を返す

◇ 効率もよくて、ほとんどの場合では**線形的**

代入θが代入θ'より一般的であるとは

$$\exists \theta_1,\ \theta' = \theta_1 \circ \theta$$

単一化のアルゴリズム

等式の集合Eと今まで作られた代入θを以下の三つの規則で書き換える

$$
\begin{array}{lcll}
(E \cup \{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\}, \theta) & & & \\
 & \rightarrow & (E \cup \{t_1 = t'_1, \ldots, t_n = t'_n\}, \theta) & \\
(E \cup \{x = t\}, \theta) & \rightarrow & ([x \mapsto t]E, [x \mapsto t] \circ \theta) & x \notin \mathsf{vars}(t) \\
(E \cup \{x = x\}, \theta) & \rightarrow & (E, \theta) &
\end{array}
$$

$(\{t_1 = t_2\}, id)$ から始めて、$(\emptyset, \theta)$に書き換えることができれば、θはt_1とt_2の最も一般的な単一子である

途中で書き換えることができなくなれば、単一子が存在しない

単一化の例（つづき）

$$
\begin{aligned}
& (\{f(x, g(a, y), y) = f(z, z, b), id) \\
\rightarrow\ & (\{x = z, g(a, y) = z, y = b\}, id) \\
\rightarrow\ & (\{g(a, y) = z, y = b\}, [x \mapsto z]) \\
\rightarrow\ & (\{y = b\}, [z \mapsto g(a, y)] \circ [x \mapsto z]) \\
\rightarrow\ & (\emptyset, [y \mapsto b] \circ [z \mapsto g(a, y)] \circ [x \mapsto z]) \\
=\ & (\emptyset, [y \mapsto b] \circ [z \mapsto g(a, y), x \mapsto g(a, y)]) \\
=\ & (\emptyset, [y \mapsto b, z \mapsto g(a, b), x \mapsto g(a, b)])
\end{aligned}
$$

アルゴリズムの正しさの証明

停止性

各規則が$\langle E$の変数の数, Eの文字数$\rangle$を減らしている(辞書式順序)

健全性

各規則$(E_1, \theta_1) \to (E_2, \theta_2 \circ \theta_1)$について，もしも$\theta$が$E_2$の単一子ならば、$\theta \circ \theta_2$ が E_1 の単一子になる

完全性

もしもE_1の単一子θが存在し、θ_1がθより一般的ならば、ある規則$(E_1, \theta_1) \to (E_2, \theta_2)$が存在し、$\theta_2$ もθより一般的である

紙の上ではどれも簡単に証明できる

Coqでの形式化の難しさ

アルゴリズムの定義

◇ 前述のアルゴリズムは関数にはなっていない

◇ 構造的再帰ではないので、定義を工夫しなければならない

◇ 項の定義で引数の数が自由だと帰納法の原理が作れない

アルゴリズムの証明

◇ 代入の冪等性を仮定してもいいが、それをすると性質の表現が自由になる反面、証明が長くなる

◇ 停止性が集合の元の数を議論する必要があり、実は最も難しい

データ構造

```
Definition var := nat.                                          (* 変数 *)
Notation "x == y" := (eq_nat_dec x y) (at level 70).

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

Inductive tree : Set :=                                       (* 項は木構造 *)
  | Var : var -> tree
  | Sym : symbol -> tree
  | Fork : tree -> tree -> tree.                          (* 分岐を2に限定 *)

Definition t1 :=                                        (* f(x,g(a,y),y) *)
  Fork (Sym "f") (Fork (Var 0)
    (Fork (Fork (Sym "g") (Fork (Sym "a") (Var 1))) (Var 1))).
Definition t2 :=                                              (* f(z,z,b) *)
  Fork (Sym "f") (Fork (Var 2) (Fork (Var 2) (Sym "b"))).
```

代入

```
(* 代入: [x ↦ t]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.

(* 代入の合成をてきようする *)
Fixpoint subs_list (s : list (var * tree)) t : tree :=
  match s with
  | nil => t                                    (* 先頭からかけて行く *)
  | (x, t') :: s' => subs_list s' (subs x t' t)
  end.
```

再帰関数としての単一化

$$
\begin{aligned}
\mathcal{U}(E \cup \{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\} & \\
&= \mathcal{U}(E \cup \{t_1 = t'_1, \ldots, t_n = t'_n\}) \\
\mathcal{U}(E \cup \{x = x\}) &= \mathcal{U}(E) \\
\mathcal{U}(E \cup \{x = t\}) &= \mathcal{U}([x \mapsto t]E) \circ [x \mapsto t] \quad \textbf{if } x \notin \mathsf{vars}(t) \\
\mathcal{U}(E) &= \bot \qquad\qquad\qquad\qquad \textbf{otherwise}
\end{aligned}
$$

合成の順番に気をつける

単一化

```
Notation unify_subs unify x t r :=
  (if occurs x t then None else
   may_cons x t (unify (map (subs_pair x t) r))).

Function unify (l : list (tree * tree)) {wf lt_pairs l}
  : option (list (var * tree)) :=
  match l with
  | nil => Some nil
  | (Var x, Var x') :: r =>
    if x == x' then unify r else unify_subs unify x (Var x') r
  | (Var x, t) :: r => unify_subs unify x t r
  | (t, Var x) :: r => unify_subs unify x t r
  | (Sym a, Sym b) :: r =>
    if symbol_dec a b then unify r else None
  | (Fork t1 t2, Fork t1' t2') :: r =>
    unify ((t1, t1') :: (t2, t2') :: r)
  | l => None
  end.
```

Functionの問題点

◇ **整礎な順序** `lt_pairs` **を定義しなければならない**

◇ **停止性を関数定義と同時に証明しなければならない**

◇ **条件に依存型が使えない**

`symbol_dec (a b:string) : {a=b}+{a<>b}` **を**

`beq_symbol : string -> string -> bool` **に変更**

証明が冗長になる

◇ `unify_subs` **を** `Definition` **にできない**

ダミー引数を使う構造的再帰

```
Definition unify_subs unify x t r :=
  if occurs x t then None else
  may_cons x t (unify (map (subs_pair x t) r)).

Fixpoint unify (h : nat) (l : list (tree * tree)) {struct h}
  : option (list (var * tree)) :=
  match h with 0 => None                    (* h は最大ステップ数 *)
  | Some h' =>
  match l with
  | nil => Some nil
  | (Var x, Var x') :: r =>
    if x == x' then unify h' r else
    unify_subs (unify h') x (Var x') r
  | ...
  end end.
```

しかし、`l` に必要なステップ数が分からない

二重ダミー引数の使い方

```
Fixpoint unify1 (unify : list (tree*tree) -> option (list (var*tree))
  (h : nat) (l : list (tree*tree)) : option (list (var*tree)) :=
  match h with                                  (* h > (l の木の大きさ) *)
  | 0 => None
  | Some h' =>
  match l with
  | nil => Some nil
  | (Var x, Var x') :: r =>
    if x == x' then unify1 h' r else
    unify_subs unify x (Var x') r
  | ...
  end end.

Fixpoint unify2 (h : nat) l :=
  match h with                              (* h > (l に含まれる変数の数) *)
  | 0    => None
  | S h' => unify1 (unify2 h') (size_pairs l + 1) l
  end.
```

健全性の証明

```
Lemma unify_subs_sound : ∀ h v t l s,
  (∀ 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).

Theorem unify2_sound : ∀ h s l,                          (* 健全性 *)
  unify2 h l = Some s -> unifies_pairs s l.
Proof.
  induction h; simpl; intros.                  (* 変数の数に関する帰納法 *)
    discriminate.
  remember (size_pairs l + 1) as h'.
  clear Heqh'.
  revert l H.
  induction h'; simpl; intros.                   (* 大きさに関する帰納法 *)
    discriminate.
  ...                                  (* unify_subs_sound を含めて70行 *)
Qed.
```

健全性の証明（Function版）

```
Lemma unify_subs_sound : ∀ v t r s,
  let l := map (subs_pair v t) r in          (* unify_subs がない *)
  occurs v t = false ->
  (∀ s, unify l = Some s -> unifies_pairs s l) ->
  may_cons v t (unify l) = Some s ->
  unifies_pairs s ((Var v, t) :: r).

Theorem unify_sound : ∀ l s,
  unify l = Some s -> unifies_pairs s l.
Proof.
  intros l.
  functional induction (unify l);                    (* 整礎帰納法 *)
    intros; try discriminate.
  ...                                (* unify_subs_sound を含めて60行 *)
Qed.
```

完全性の証明

```
Lemma not_unifies_occur : ∀ v t s,                    (* 解のない等式 *)
  Var v <> t -> In v (vars t) -> ~unifies s (Var v) t.
Lemma unifies_extend : ∀ s v t t',                    (* 単一子の保存 *)
  unifies s (Var v) t -> unifies s (subs v t t') t'.
Lemma unifies_pairs_extend : ∀ s v t l,                    (* 拡張 *)
  unifies_pairs s ((Var v, t) :: l) ->
  unifies_pairs s (map (subs_pair v t) l).

Definition moregen s s' :=              (* s が s' より一般的である *)
  ∃s2, ∀t, subs_list s' t = subs_list s2 (subs_list s t).
Lemma moregen_extend : ∀ s v t s1,
  unifies s (Var v) t -> moregen s1 s -> moregen ((v, t) :: s1) s.

Parameter vars_pairs_decrease : ∀ x t l, ~In x (vars t) ->
  length (vars_pairs (map (subs_pair x t) l))
  < length (vars_pairs ((Var x, t) :: l)).  (* 代入が変数を減らす *)

Theorem unify2_complete : ∀ s h l,                    (* 完全性 *)
  h > length (vars_pairs l) -> unifies_pairs s l ->
  ∃s1, unify2 h l = Some s1 ∧ moregen s1 s.
```

まとめ

◇ 冪等性を使った証明では、細かいことを考えなくてもいいが
健全性までで550行程度

◇ 定義に気をつけると、単一化の健全性が簡単に証明できる
定義を含めて283行

◇ 完全性も難しくないが、停止性は準備が大変
今回は公理で対応 合計558行

◇ Functionは便利だが、この場合では自由度が減る
整礎順序の定義が要るので、合計612行

おまけ：構造的再帰による単一化

◇ Conor McBride が提案した手法
First-order unification by structural recursion.
JFP 13 (6), 2003.

◇ 変数の型を有限集合にすることで、型システムが減少を保障する

◇ AGDA などで書きやすいが、Coq だと依存型の等式を全部手で扱わないと行けないので断念

◇ 代わりにOCamlで書いてみた

変数の数を制限した項の定義

```
type zero = Zero                          (* ZeroとSuccはデータ型にするため *)
type _ succ = Succ

type _ fin =                              (* n fin は n より小さい自然数の型 *)
  | FZ : 'a succ fin                              (* ∀ n > 0, FZ : n fin *)
  | FS : 'a fin -> 'a succ fin        (* x : n fin ⊢ FS x : (n + 1) fin *)

type _ is_succ = IS : 'a succ is_succ             (* 0でない自然数 *)
(* n fin が空でなければ、n ≠ 0 *)
let fin_succ : type n. n fin -> n is_succ = function
  | FZ -> IS
  | FS _ -> IS

type 'a term =                                                (* 単純な項 *)
  | Var of 'a fin
  | Leaf
  | Fork of 'a term * 'a term
```

代入で変数の数が減る

```
(* m ≠ nならば、thick m n = if m < n then n − 1 else n *)
let rec thick : type n. n succ fin -> n succ fin -> n fin option =
  fun x y -> match x, y with
  | FZ, FZ   -> None
  | FZ, FS y -> Some y
  | FS x, FZ -> let IS = fin_succ x in Some FZ
  | FS x, FS y ->
      let IS = fin_succ x in bind (thick x y) (fun x -> Some (FS x))

let subst_var x t' y =                                  (* 変数に対する代入 *)
  match thick x y with
  | None -> t'                                                  (* x = y *)
  | Some y' -> Var y'                                           (* x ≠ y *)
val subst_var : 'a succ fin -> 'a term -> 'a succ fin -> 'a term

let subst x t' = pre_subst (subst_var x t')              (* 項に拡張 *)
val subst : 'a succ fin -> 'a term -> 'a succ term -> 'a term
```

単一化

```
type (_,_) alist =                          (* 変数の数をmからnに減らす代入 *)
| Anil : ('n,'n) alist
| Asnoc: ('m,'n) alist * 'm term * 'm succ fin -> ('m succ,'n) alist
type _ ealist = EA : ('m,'n) alist -> 'm ealist

let rec amgu: type m. m term -> m term -> m ealist -> m ealist option =
  fun s t acc -> match s, t, acc with
  | Leaf, Leaf, acc -> Some acc                        (* 代入を遅らせる *)
  | Leaf, Fork _, _ -> None
  | Fork _, Leaf, _ -> None
  | Fork (s1, s2), Fork (t1, t2), acc ->
      bind (amgu s1 t1 acc) (amgu s2 t2)                    (* m' = m *)
  | Var x, Var y, EA Anil ->          (* 変数のときに代入を先にかける *)
      let IS = fin_succ x in Some (flex_flex x y)
  | Var x, t, EA Anil -> let IS = fin_succ x in flex_rigid x t
  | t, Var x, EA Anil -> let IS = fin_succ x in flex_rigid x t
  | s, t, EA(Asnoc(d,r,z)) ->              (* 代入をかけ、m' = m - 1 *)
      bind (amgu (subst z r s) (subst z r t) (EA d))
           (fun (EA d) -> Some (EA(Asnoc(d,r,z))))
```