# 12　推論と単一化 (続)

### SLD 融合アルゴリズム

前回は命題論理式に対して融合による反駁の証明を説明した．それを述語論理に拡張すると，以下のような働きになる．

知識 (論理プログラム) はホーン節の集りとして与えられる．

$$
\begin{array}{l}
A_1 \\
\ldots \\
A_{k-1} \\
A_k \leftarrow A_{k1} \wedge \ldots \wedge A_{1m_k} \\
\ldots A_n \leftarrow A_{n1} \wedge \ldots \wedge A_{nm_n}
\end{array}
$$

$A_{k-1}$ までのホーン節は仮定を持たないので，帰結が無条件に導ける．

その知識 (プログラム) の元で，目標 $G = B_1 \wedge \ldots \wedge B_l$ が常になりたつことを証明したい．そのために，$\neg G$ が知識と矛盾することを反駁によって証明する．

まずは，$G$ が変数を含まない場合を考える．

1. $G = B_1 \wedge \ldots \wedge B_l$ に分ける．
2. $G$ が空だったら，反駁が成功する．
3. 選択関数 $S$ によって $B_i$ を選ぶ．(通常は $B_1$)
4. $B_i = [t_1 ... t_{v_j}/x_1 ... x_{v_j}]A_j$ となるような節 $A_j$ と $t_1 ... t_{v_j}$ を求める．
5. 見付からなければ，反駁が失敗する．
6. $j < k$ ならば，$B_i$ を目標から削り (1) に戻る．

$$G = B_1 \wedge \ldots \wedge B_{i-1} \wedge B_{i+1} \wedge \ldots \wedge B_l$$

7. $j \geq k$ ならば，$B_1$ を $A_j$ の仮定に置き換えてから (1) に戻る．

$$G = B_1 \wedge \ldots \wedge B_{i-1} \wedge [t_1 ... t_{v_j}/x_1 ... x_{v_j}](A_{j1} \wedge \ldots \wedge A_{jm_j}) \wedge B_{i+1} \wedge \ldots \wedge B_l$$

この融合アルゴリズムが SLD resolution と呼ばれる．Selective(選択): 目標の中から述語を選び，Linear(線型的): 一個だけ置き換える，そして Definite(定義的) な節が使われる．

ステップ (4) では可能な $A_j$ が複数ありうるので，このアルゴリズムは本質的に非決定的である (途中での決定によって違う結果がありうる)．$G$ が変数を含まない場合，成功する導出が一つでもあれば十分．

$G$ が変数を含むとき，定義がもう少し複雑になる．

1. 最初の変数割り当てを恒等関数にする: $\sigma = id$
2. $G = B_1 \wedge \ldots \wedge B_l$ に分ける．
3. $G$ が空だったら，反駁が成功するので，$\sigma$ を返す．
4. 選択関数 $S$ によって $B_i$ を選ぶ．

5. $\sigma'(\sigma(B_i)) = \sigma''(A_j)$ となるような節 $A_j$ と最も一般的な $\sigma'$ と $\sigma''$ を求める．

6. 見付からなければ，反駁が失敗する．

7. 見付かれば，$\sigma$ の定義を更新する．

$$\sigma := \sigma' \circ \sigma$$

8. $j < k$ ならば，$B_i$ を目標から削り (2) に戻る．

$$G := B_1 \wedge \ldots \wedge B_{i-1} \wedge B_{i+1} \wedge \ldots \wedge B_l$$

9. $j \geq k$ ならば，$B_1$ を $A_j$ の仮定に置き換えてから (2) に戻る．

$$G := B_1 \wedge \ldots \wedge B_{i-1} \wedge \sigma''(A_{j1} \wedge \ldots \wedge A_{jm_j}) \wedge B_{i+1} \wedge \ldots \wedge B_l$$

ステップ (5) では別々の $\sigma'$ と $\sigma''$ を使っているが，あらかじめある代入 $\delta$ で $A_j$ の中の全ての変数を $\sigma$ に現われないものに変えると，解きたい式が $\sigma'(\sigma(B_i)) = \sigma'(\delta(A_j))$ になり，単一化が使える．そうすると最後のステップの中の $\sigma'$ が $\delta$ に変更できる．

変数がある場合，節の選択で変わるのは成功か失敗だけでなく，成功したときの変数の割り当てでもある．場合によって，成功してもそれ以外の割り当ても知りたいこともある．

こういう非決定性を必要とするようなアルゴリズムを実装するとき，7 章で見た検索問題を解く方法が有効である．特に，以下の関数 flat_map が役に立つ．

```
let rec flat_map f = function
    [] -> []
  | a :: l -> f a @ flat_map f l
val flat_map : ('a -> 'b list) -> 'a list -> 'b list
```

例えば，友達のリストを与えられ，“F$(x, y)$” という形の項を作ることを非決定的な選択を仮定すると

```
(* 非決定的に要素を選ぶ関数 (OCaml には当然ない) *)
val choose : 'a list -> 'a
let make_friends friends =
  let x = choose friends in let y = choose friends in
  if x = y then [] else [F[C x;C y]]
val make_friends : string list -> term
```

になるが，OCaml には非決定性がないので，全ての途中結果をリストに変える必要がある．そのために非決定的な **let** $x = E_1$ **in** $E_2$ を `flat_map` (**fun** $x$ `->` $E_2$) $E_1$ に変える．そうしたら `choose` は恒等関数になる．

```
let make_friends friends =
  flat_map
    (fun x -> flat_map (fun y -> if x = y then [] else [F[C x;C y]]) friends)
    friends
val make_friends : string list -> term list
# make_friends ["Ikura"; "Tarao"; "Wakame"];;
- : term list =
[F [C "Ikura"; C "Tarao"]; F [C "Ikura"; C "Wakame"];
 F [C "Tarao"; C "Ikura"]; F [C "Tarao"; C "Wakame"];
 F [C "Wakame"; C "Ikura"]; F [C "Wakame"; C "Tarao"]]
```

ちなみに，以下のような演算子を定義すると，もっとエレガントに書ける．

```
let (>>) l f = flat_map f l

let make_friends friends =
  friends >> fun x -> friends >> fun y -> if x = y then [] else [F[C x;C y]]
```

### 論理型プログラミング

ある目標がプログラムの帰結であるかどうかが調べられることは既に例で見せている．しかし，これでは一般的なプログラミングができるのだろうか？

実は，多くのことが直接的にできる．例えば，自然数の上の足算や掛け算が以下のように定義できる．

$$
\begin{aligned}
&add(Zero, x, x)\\
&add(Succ(x), y, z) \leftarrow add(x, Succ(y), z)\\
&mul(Zero, x, Zero)\\
&mul(Succ(x), y, z) \leftarrow mul(x, y, t), add(y, t, z)
\end{aligned}
$$

しかし，このままでは一つの弱みがある．否定が表現できない．それを解決するのに，二つのやりかたが導入されて来た．

一つの方法は定数に対してのみ不等号を導入すること．

$$
\begin{aligned}
&if0(0, x, y, x)\\
&if0(z, x, y, y) \leftarrow z \neq 0
\end{aligned}
$$

ただし，比較が成功するのは $z$ 値が決っているときだけで，非決定的な検索が使えない．異る方法で $z$ の具体的な値を生成することになる．

もう一つの方法は，特定の節の一部の仮定が成功すれば，同じ述語に関するそれ以降の項を利用しないようにする．選択関数を左から右のものにし，節にも順番を入れることになる．

$$
\begin{aligned}
&if0(t, x, y, z) \leftarrow t = Zero \wedge ! \wedge z = x\\
&if0(t, x, y, z) \leftarrow y = z
\end{aligned}
$$

この定義では，$t$ が $Zero$ と単一化できるときは，前の節が選ばれ，後の節が無視される．逆に $t$ の単一化が失敗すれば，後の節が使われる．この！を「カット」と読む．検索範囲が制限できたりするのでとても便利だけど，論理的な意味があまりないので純粋でないと批判されることもある．