論　文　　Technical Paper

Prologに関するいくつかの性質について
Some Properties of Prolog

阿久津　達也*　大須賀　節雄*
Tatsuya AKUTSU　Setsuo OHSUGA

*　東京大学工学部境界領域研究施設
Institute of Interdisciplinary Research, Faculty of Engineering, The University of Tokyo, Tokyo 153, Japan.

1986年11月25日　受理

Keywords: prolog, logic programming, SLD-resolution, inductive inference.

Summary

We have been making efforts to improve Shapiro's model inference system. We have not got answer yet, but we got three results from such consideration. The results are as follows.

(1) The inference system which inferences a source prolog program from trace data.

(2) Neither the limitation of amount of predicate symbols nor the limitation of amount of clauses used in prolog programs makes no hierarchy of programs which can be described.

(3) The program P′ which satisfies following properties can be constructed from any given program P and any given number k ($0 < k < 1$).

- If there exists an SLD-refutation of $p \cup \{<- A\}$ of length N, there exists an SLD-refutation of $P' \cup \{<- A\}$ of length under $\lceil kN \rceil$.
- Program P and P′ have the same meaning in the sense of least fixpoints.

1. はじめに

(Pure) Prologはたったひとつの，しかも簡単な推論規則しかもたないため，理論的にも種々の良い性質[1]を備え，帰納的推論[2]やプログラム変換[3]に関して良い研究成果も得られている．本論文では，主として文献(2)のモデル推論システム(MIS.)の改良を目的とした考察から得られた3つの結果を紹介する．それらは，

(1) トレース例からのプログラム推論アルゴリズム，

(2) 基本的には，述語の数，節の数の違いによって記述できるプログラムに階層が生じることはない，

(3) 任意のプログラムP，任意の整数k ($k \geq 1$) に対して，Pと等価で，同じ論理式を導出するのに必要なステップ数がPの場合の1/k以下であるプログラムを構成できる，

である．

なお，本論文中で導出とはSLD導出を指すものとし，また，PrologとSLD導出についての関係や細かい定義などは，文献(1)を参照してほしい．

2. トレース例からのプログラム推論

文献(4)では，最小汎化を用いてモデル推論システムの改良を試みているが，それよりヒントを得て，最小汎化を用いてPrologプログラムのトレース例からもとのプログラムを推論する方法を考えた．

Prologプログラムのトレース例からのプログラム推論は，たとえば，appendプログラムの(複数の)トレース例，

```
<- append ([a, b], [], [a, b]),
    <- append ([b], [], [b]),
<- append ([], [], []),□
<- append ([c], [d], [c, d]),
    <- append ([], [d], [d]), □
                ⋮
```

から，appendプログラム，

```
append ([], X, X) <-
append ([H|X], Y, [H|Z])
    <- append (X, Y, Z)
```

を推論することである．

2・1 Plotkinの最小汎化[5]

最小汎化とは一種の帰納的推論で，一言でいうと，リテラル（もしくは項）の集合について，その集合に属するすべてのリテラル（項）を例（変数に代入したもの）とすることのできる最も特殊な項のことである．以下，その概要を説明する．

［**定義2．1**］　リテラルまたは項を語と呼ぶ（本論文においては，V, W, V_1, W_1, …で表す）．

［**定義2．2**］　W_1, W_2を任意の語とする．このとき，ある代入σが存在して，$W_1\sigma=W_2$であるとき，$W_1 \leqq W_2$（「W_1はW_2より一般的である」と読む）と書く．

定義2．2より，$W_1 \leqq W_2$かつ$W_2 \leqq W_1$であることと，W_1とW_2がvariants（変数名だけが異なる語）であることは等価となる．

［**定義2．3**］　Wを語，Kを語の空でない集合とする．Wが，

(1) Kに属するすべてのVについて，$W \leqq V$，

(2) Kに属するすべてのVについて，および任意のW_1について，$W_1 \leqq V$ならば$W_1 \leqq W$，

を満たすとき，WをKの最小汎化という（今後，Kの最小汎化を lg (K) で表す）．

〔**定理2．1**〕(plotkin)　Kを語の空でない有限集合とするとき，Kの最小汎化が存在するための必要十分条件は，Kの任意の要素W_1, W_2に対し，W_1, W_2が共に項であるか，同じ述語記号とサイン（否定記号をもつかどうか）であることである．更に，そのとき，lg ({W_1, W_2}) を計算できる．

なお，最小汎化はあるとすればvariantsを除いて1つであり，K= {$W_1, W_2, \cdots, W_n$} の最小汎化は，lg ({W_1, lg ({W_2, … lg ({W_{n-1}, W_n}) …})}) で計算できる．

（**例2．1**）　K= {p (a, f (X)), p (b, f (a)), p (f (a), f (f (Y)))} の最小汎化は，p (U, f (V)) である．

2・2 方　　法

まず，最小汎化の定義を節について拡張する．なお，以下の定義は文献(5)の節についての最小汎化の定義とは異なるが，同じく最小汎化という言葉を用いることにする．

［**定義2．4**］　C_1, C_2を任意の節とする．このとき，ある代入σが存在して，$C_1\sigma = C_2$であるとき，$C_1 \leqq C_2$と書く．

［**定義2．5**］　Cを節，Sを節の空でない集合とする．Cが，

(1) Sに属するすべてのC′について，$C \leqq C'$．

(2) Sに属するすべてのC′について任意のC_1に対して，

$C_1 \leqq C'$ならば$C_1 \leqq C$

を満たすとき，CをSの最小汎化節という（今後，Sの最小汎化節についても最小汎化同様に lg (S) で表す）．

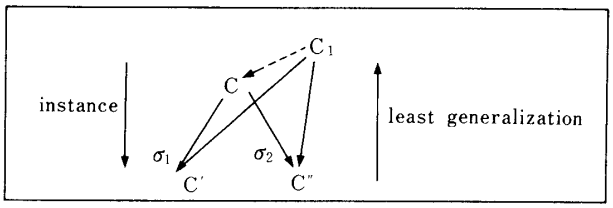

Fig. 1　Least generalization.

〔**定理2．2**〕　節集合S= {C_1, C_2, …, C_m} が，

(1) C_iがそれぞれ$A_{i0} <- A_{i1} \wedge A_{i2} \wedge \cdots \wedge A_{in}$という型をしている（body部のリテラル数が同じ），

(2) A_{ik}とA_{jk}の述語記号はすべて等しい，

を満たすときに（のみ）lg (S) が存在し，

$$\lg(S) = A_0 <- A_1 \wedge \cdots \wedge A_n$$

となる．ただし，A_0, A_1, …, A_nは，f_{n+1}をS中に現れない任意の$n+1$変数関数記号とするとき，

$$f_{n+1}(A_0, A_1, \cdots, A_n) = \lg(\{f_{n+1}(A_{10}, A_{11}, \cdots, A_{1n}), \cdots, f_{n+1}(A_{m0}, A_{m1}, \cdots, A_{mn})\})$$

を満たすものとする．

《証　明》

定理2．1および定義2．5から明らか．□

なお，最小汎化の場合と同様に，

$$\lg(\{C_1, \cdots, C_n\}) = \lg(\{C_1, \lg(\{C_2, \cdots, \lg(\{C_{n-1}, C_n\}) \cdots\})\})$$

となる．

（**例2．2**）　$p(X1) <- p(f(a)) \wedge q(f(Y1))$と$p(X2) <- p(f(b)) \wedge q(f(X2))$の最小汎化節は，$p(X) <- p(f(Y)) \wedge q(f(Z))$となる．

次に，トレース例を導出が成功した後に求めるものとして以下のように定義する．

［**定義2．6**］　プログラムPにおけるゴールG (= $G_0 = <- A_1 \wedge \cdots \wedge A_k$) のトレース例，TR (P, G) は，Gの導出が成功したときにのみ定義されるものとし，

$$TR(P, G) = [\, G_0\sigma_1 \cdots \sigma_n,\ G_1\sigma_2 \cdots \sigma_n,\ \cdots,\ G_i\sigma_{i+1} \cdots \sigma_n,\ \cdots,\ G_n(=\square)\,]$$

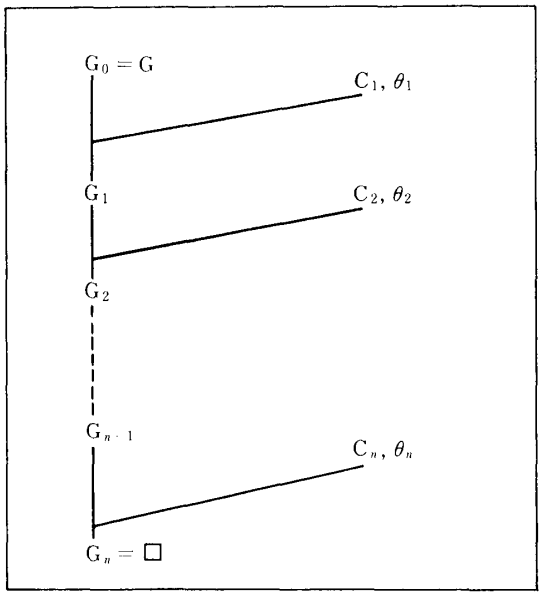

Fig. 2 An SLD refutation.

ただし，σ_{j+1}はゴールG_jと節C_{j+1}からゴールG_{j+1}を導出するのに使用されたmgu.

とする．

なお，SLD導出では任意のcomputation ruleを許しているが，本節ではProlog処理系を対象としているので，computation ruleとして，常に最も左側のリテラルが導出に使われるものとする．

(例2.3) appendプログラムにおいて，ゴールG_0が，<- append ([a, b], [c], X) の場合，

```
TR (P, G) = [ <- append ([a, b], [c],
                  [a, b, c]),
                <- append ([b], [c], [b, c]),
                <- append ([], [c], [c]),
                □]
```

であり，<- append ([d, e], X, Z) の場合は，

```
TR (P, G) = [ <- append ([d, e], X,
                  [d, e|X]),
                <- append ([e], X, [e|X]),
                <- append ([], X, X),
                □]
```

となる．

さて，いよいよ本節の主題であるトレース例からのプログラム推論アルゴリズムを示すことにする．

アルゴリズム (Fig. 3 参照) の要点は，トレース例から元のプログラム中の節の例を生成し，その例の最小汎化節を求めることによって元のプログラムを推論しようということである．

(例2.4) アルゴリズム (Fig. 3) の動作を示すために，トレース例

```
Input : A list of trace data.
Output : A sequence of programs.

P := {};
while (there exist trace data) {
    read the next trace data
          TR (= [G0', G1', ・・・, Gn']);
    i := 0;
    while (i < n) {
          suppose that
                Gi' = <- Bi1 ∧ ・・・ ∧ Bik,
              Gi+1' = <- B(i+1)1 ∧ ・・・ ∧ B(i+1)m
        then
      C := Bi1 <- B(i+1)1 ∧ ・・・ ∧ B(i+1)(m - k+1);
      if (there is a clause C' in P,
          with which there exists a lg ({C, C'})) {
        P := (P - {C'}) ∪ {lg ({C, C'})};
      }
      else {
        P := P ∪ {C};
      }
    }
    output P;
}
```

Fig. 3 Inference from trace data.

```
TR (P, G) = [ <- append ([a, b], [c],
                  [a, b, c]),                    (i)
                <- append ([b], [c], [b, c]),
                                                 (ii)
                <- append ([], [c], [c]),
                                                 (iii)
                □(iv) ]
```

を読み込んだ場合，アルゴリズム (Fig. 3) のC, Pがどのように変わっていくかを示そう．

(i), (ii) より，

```
C = append ([a, b], [c], [a, b, c]) <-
          append ([b], [c], [b, c])
P = {append ([a, b], [c], [a, b, c]) <-
          append ([b], [c], [b, c]) (v)}
```

(ii), (iii) より，

```
C = append ([b], [c], [b, c]) <-
          append ([], [c], [c])
```

(v) との最小汎化節を求めて，

```
P = {append ([H1|X1], [c], [H1|Z1]) <-
          append (X1, [c], Z1)}
```

(iii) と (iv) より，

```
C = append ([], [c], [c])
P = {append ([H1|X1], [c], [H1|Z1]) <-
          append (X1, [c], Z1),
      append ([], [c], [c]) <-}
```

更に，別の例を読み込めば，Appendix のように正しい append プログラムに収束する．

2・3 考　　察

残念ながらアルゴリズム（Fig. 3）は，すべてのクラスの Pure Prolog プログラムをトレース例から推論できるわけではない．しかしながら，あるクラスについては推論するプログラムが必ずしももとのプログラムと同じというわけではないが，正しく収束することが言える．そのことを定理として述べる．

＜補題 2．1＞　節C_1，C_2があって，$C\sigma_1=C_1$，$C\sigma_2=C_2$となる節Cが存在するとき，$\{C_1, C_2\}$の最小汎化節が存在し，$C\leqq lg(\{C_1, C_2\})$となる．

《証　明》

C_1，C_2が上の条件を満たすとき，定理 2．2 の条件も満たす．よって，定義 2．5 より補題は成立する．□

＜補題 2．2＞　節C_1と節C_2に対して，$C\sigma=C_1$，$C\theta=C_2$とできる節Cは，variants を除いて，たかだか有限個しか存在しない．

《証　明》

節C_1に対して，$C\sigma=C_1$となる節は，variants を除いて有限個しかないことを示せば十分である．そのため，代入σを考えることにより，

(1) 節中に現れる定数記号の数 (C)
　　≦ 節中に現れる定数記号の数 (C_1)
(2) 節中に現れる定数記号の集合 (C)
　　⊆ 節中に現れる定数記号の集合 (C_1)
(3) 節中に現れる関数記号の数 (C)
　　≦ 節中に現れる関数記号の数 (C_1)
(4) 節中に現れる関数記号の集合 (C)
　　⊆ 節中に現れる関数記号の集合 (C_1)
(5) 節中に現れる変数の数 (C) は，有限である．
(6) 節中に現れるリテラルの数 (C)
　　≦ 節中に現れるリテラルの数 (C_1)

であることがわかり，C_1に現れる定数記号，関数記号の数，種類ともに有限で，リテラルの数も有限であるから，Cに現れる定数記号，関数記号の数，種類ともに有限で，変数，リテラルの数も有限となり，C は variants を除いて有限個しかありえない．よって，補題が成り立つ．□

〔定理 2．3〕　プログラムP_1は，次の条件を満たすものとする．

- 任意のC_j, $C_k\in P_1$（ただし，$C_j\neq C_k$）について，$lg(\{C_j, C_k\})$は，存在しない．

このとき，P_1についてのトレース例の入力に対し，アルゴリズム（Fig. 3）における P は（トレース例の増大に対して）収束し，その収束点のプログラム P′ 中の節はすべてP_1中の節の例（ただし，variants も含む）になっている．

《証　明》

この証明において節C_{i+1}とC_{i+1}の variants を区別せずにC_{i+1}で表すものとする．

まず，アルゴリズム（Fig. 3）において使われるCという節が，P_1中の節の例になっていることを示そう．

G_iを$<-A_1\wedge\cdots\wedge A_k$，$G_i$との導出に使われる節を

$$C_{i+1}(=A<-B_1\wedge\cdots\wedge B_m)$$

とすると，$A_1\sigma_{i+1}=A\sigma_{i+1}$であり，

$$G_{i+1}=(<-B_1\wedge\cdots\wedge B_m\wedge A_2\wedge\cdots\wedge A_k)\,\sigma_{i+1}$$

である．ところで，Cは，

$$(A_1<-B_1\wedge\cdots\wedge B_m)\,\sigma_{i+1}\cdots\sigma_n$$

であるから，$C=C_{i+1}\sigma_{i+1}\cdots\rho_n$となり，C は$C_{i+1}$の例となる．

次に C がP_1中の節C_{i+1}の例となっているとき，C と最小汎化節を持つ P 中の節 C′ は，やはりC_{i+1}の例となっており，それゆえ，$lg(\{C, C'\})$もC_{i+1}の例となっていることを帰納法で示す．

(1) C′ が最小汎化の操作を 1 回も経ていないとき，C′ はアルゴリズム（Fig. 3）の else 文のところの操作によって得られたものであり，この証明中の最初の結果より明らかに成立する．
(2) C′ が最小汎化の操作を n 回経ているときに成立するものとする．
(3) C′ が最小汎化の操作を $n+1$ 回経ているとする．アルゴリズム（Fig. 3）を考慮すると，C′ は以前に（その時点の）C′ と C との最小汎化節として得られたものである．このときの C′，C をそれぞれ CC′，CC とする．このとき，帰納法の仮定より CC′ はC_{i+1}の例であり，P_1に対する条件とこの証明中の最初の結果から，CC はC_{i+1}の例であるから補題 2．1 より C′ はC_{i+1}の例となる．

更に，補題 2．2 より，節 C を含む集合の最小汎化節となりうる節は有限個しかありえないことがわかる．よって，そのことと最小汎化節の性質，

$$lg(\{C, \cdots\})\leqq C$$

より，アルゴリズム（Fig. 3）は，C_{i+1}の例となる P 中の節を（variants を除いて）有限回しか変更できない．

よって，P中の節はすべてP_1の節の例であり，P_1中の節は有限個であり，1つの節について有限回しか変更できないので定理が成り立つ．□

なお，定理 2．3 の条件を満たす場合，アルゴリズムの途中で得られる P だけで，それ以前の例についての導出が可能である．

さて，なぜ定理2．3程度にしかアルゴリズム(Fig. 3)が強力でないかについて補足したい．

まず，定理2．3の条件がなぜ必要かを考える．たとえば，元のプログラム中に，

p(a)<−

p(b)<−

という2つの節があり，<−p(a)と<−p(b)の2つのゴールについてのトレース例についてアルゴリズム(Fig. 3)を適用すると，

p(X)<−

という節が求まってしまう．つまり，元のプログラムより一般的すぎるプログラムが求まってしまう．そのことを避けるためには，アルゴリズム(Fig. 3)によって求まるCで最小汎化節を持つものの集合を分割しなくてはならなくなる．しかしながら，分割するためには何らかの情報が必要であるがトレース例だけから情報を取り出すのは難しい．可能性としては，導出に失敗したゴールの例(負のデータ)を用いることによって一般化のしすぎを防ぎ，分割するための情報とすることができよう．

次に，もとのプログラムと全く同じプログラムが必ずしも求まらない理由を説明する．たとえば，もとのプログラムPを，

P={ p([H|X])<−,
　　p([H|X])<−p(X)}

とする．このとき，十分トレース例を与えた後に求まるプログラムは，

P′={p([H|X])<−,
　　p([H1, H2|X])<−p([H 2|X])}

というものである．この理由は，p([H|X])<−p(X)において，p(X)の導出を考えると，導出が成功するためには，Xが[H′|X′]という型をしていなければならないからである．しかしながら，PとP′の意味(最小不動点)は同じものであり，一般的にいってもアルゴリズム(Fig. 3)によって求まるプログラムは，もとのプログラムと意味において同じものが求まるであろう．

さて，この方法とMIS.との関係について触れよう．この方法とMIS.との大きな違いは，与えられる例の違いである．MIS.では，基本的には基底アトムの真偽，つまりプログラムの正しい入出力例と誤った入出力例だけであり，プログラムの仕様に近いものだけを与えている．しかし，この方法ではプログラムのトレース例を与えており，もとになったプログラムの内部構造をかなり反映するものを与えているのである．MIS.より，より多くの情報を与えているわけである．また，この方法で推論を行うためには，もとになるプログラムが存在することが必要であるのに対して，MIS.では仕様だけがあればよいのである．よって，この方法の適用範囲は，MIS.よりはずっと少ないと思われ，単独で使用するよりはほかの方法と組み合わせて使用するのが適切であると思われる．応用の可能性としては，Prologプログラムのデバッグ支援や，帰納的推論システムの仮説生成部の一部としての使用などが考えられる．

3．述語，節の数によるプログラムの階層

モデル推論システムでは，最初に，

(1) 推論されるべきプログラムにおいて使われる可能性のあるすべての述語記号について，その述語記号の各引数のタイプ，入出力の区別，

(2) 推論されるべきすべての述語定義の節についてBody部に現れる可能性のある述語記号，

を与え，実行時に，

(3) 使用されるすべての述語記号に関する例やoracle(基底原子式の真偽を答える)，

を与える必要がある．このことは，推論されるべきプログラムについてかなりの情報(制限)を与えることになる．つまり，メインルーチンについての性質だけでなく，サブルーチンの構造や引数のタイプについてまでかなりの情報を与えることになる．

しかしながら，実際に例だけから推論できるようにするためには，少なくとも最初にゴールとして呼ばれる述語以外の情報は，入力しないですむほうが望ましい(もちろん，制約は別の形で与える)．このことを考えると，「使用される述語記号の数(種類)によって，記述できるプログラムに階層はあるか」という疑問が生じる．更に，「節の数，節中のリテラルの数によって記述できるプログラムに階層はあるか」という疑問も生じる．そして，この解答として得たのは，そのどれについても「基本的には階層は生じない」という結果である．

この結果の証明は非常に単純で，一言でいうと，インタプリタに相当するものを記述できるということである．もちろんPrologでPrologのインタプリタを記述できるという事実は有名であるが，それはclause述語を使用するものでPure Prologによるものでなく，また，プログラムをそのままの形式で引数として入力し実行するPure Prologのインタプリタは，Pure Prologでは記述できそうにない．ここで述べるのは対象とするプログラムごとにそのインタプリタが記述

できるということである.

ここで定理を述べるが，考える1階述語言語には，以下の議論が成立する程度の定数記号，関数記号，述語記号が十分あるものとする．更に，述語記号と関数記号に同じ記号を用いるが，実際は異なるものとする.

〔**定義3．1**〕　プログラムP_1とP_2が述語記号pについて等価とは，P_1，P_2の最小不動点をlfp (P_1)，lfp (P_2) とすると，

$$\{p(t_1, \cdots, t_n) \mid p(t_1, \cdots, t_n) \in \mathrm{lfp}(P_1)\} = \{p(t_1, \cdots, t_n) \mid p(t_1, \cdots, t_n) \in \mathrm{lfp}(P_2)\}$$

であることである.

〔**定理3．1**〕　任意のプログラムPとpについて等価で，使用される述語記号の数が3個，節の数が5個，節中のリテラルの数がたかだか3個であるプログラムP′を構成できる.

《証　明》

$P = \{C_1, C_2, \cdots, C_m\}$ とし，pをn変数の述語記号であるものとする.

ここで，節C_iを$A_0 <- A_1 \wedge A_2 \wedge \cdots \wedge A_k$とすると，項$C_i'$は，$[A_0, A_1, A_2, \cdots, A_k]$であるものとする.

このとき，

```
P′= {
  p(X1, ···, Xn) <- ip1(p(X1, ···, Xn),
    [C1′, ···, Cm′]),
  ip1(Head, [[Head|Body]|Rest]) <-
  ip2(Body),
  ip1(Head, [Clause|Rest]) <- ip1(Head,
      Rest),
  ip2([]) <- ,
  ip2([Goal1|Rest]) <- ip1(Goal1, [C1′, ···,
      Cm′]) ∧ ip2(Rest)
}
```

とすればよい，プログラムP′が，pについてPと等価であることは明らかである．□

上記のようにして，基本的には，述語記号，節，節中のリテラルの数によって記述できるプログラムのクラスに階層のないことがわかった．ところで，定理3．1では，述語記号，節，節中のリテラルの数が少なくなったかわりに，節中の項の長さが長くなっている．それでは，項の長さによって階層はあるだろうか？　しかしながら，項の長さによっても項の長さ単独の制限では階層がないことが予測される．証明はまだ行っていないが，その根拠を例で示そう.

たとえば，ある節が，

$$p(X) <- q(f(f(a, b), f(a, f(a, X))))$$

であったとしよう．このとき，もとのプログラムに使用されていない述語記号q1，q2，q3を用いて，その節のかわりに，

```
p(X) <-q(f(Y, Z)) ∧q1(Y) ∧q2(X, Z)
q1(f(a, b)) <-
q2(X, f(a, Y)) <-q3(X, Y)
q3(X, f(a, X)) <-
```

とすればよい.

しかしながら，述語記号，節，節中のリテラルの数，および項の長さすべてを有限のある値以下におさえて，すべてのプログラムと等価なプログラムを作ることはできない.

このことは，制限なしで作ることのできるプログラムの種類は無限個であるが，この制限のもとで作ることのできるプログラムの種類は有限であることから簡単にわかる.

ところで，本来の目的のMIS．の改良という点に戻るが，残念ながら定理3．1の結論はMIS．の改良にはつながらない．確かに，述語数などを減らすことはできるが，定理3．1中のip1，ip2といった述語のもつ情報は，すべてのサブルーチンのもつ情報と同じであり，ip1，ip2という述語を帰納的に推論することと，すべてのサブルーチンをも帰納的に推論することの大変さはほとんど同じだからである.

さて，この節の最後に定理3．1の述語数は1個までは減らせないことを示そう（2個までは減らすことができる）．そのためには反例を1つ挙げればよい.

例として，square (N) という述語を考える．この述語の意味はNは自乗数（ある自然数を自乗した数）であるという意味であるものとする．そのようなプログラムは，自然数を項s (s (…s (0) …)) で現すこととすると，何の制限もなければPure Prologでも簡単に書くことができる．しかし，squareという述語記号以外をつかわないで書くことはできない.

もし，そのようなプログラムが作れたとしよう．すると，自然数は無限にあり，プログラム中の節の数は有限でなければならないので，空節を導出するまでに使われる回数がいくらでも大きくなりうる.

$$\mathrm{square}(t_0) <- \mathrm{square}(t_1) \wedge \cdots \wedge \mathrm{square}(t_n) \qquad (*)$$

という節が少なくとも1つはあるはずである．ここで，項t_0は，変数を含んでいなければならず（そうでなければ，この節からは1つの自乗数を得ることしかできない），項t_0が含む変数（項は，s (s (…)) の形なので変数はただ1つである）は，body部にも現れなければならない（そうでなければ，自乗数でない数が導出できてしまう）．その変数を含むbody部の原子式を

square (t_i) とする．ある基底インスタンスにおいて，t_0の表す数をjとし，t_iの表す数をmとすると，$m=j+k$ (kはある整数) となる．なお，$k=0$となる場合は，別の節を考えることにする．ところで，t_0もt_iも自乗数を表していなければならないが，kが0以外のとき，自乗数m,jで，$m=k+j$となるものは有限個しかない．よって(*)式からは，有限個の自乗数しか導くことができないので矛盾する．

こうして，述語記号1個では記述できないプログラムがあることがわかった．

4．推論のステップ数に関する定理

この節では，任意の正数$k \leqq 1$，任意のプログラムPについて，Pと等価なプログラムacc (P) で，同じゴールについての空節までの導出に必要なステップ数がk倍 (端数は切り上げる) 以下であるものを構成できるという定理を示す．これは，チューリング機械の線形加速定理に相当するものと思われる．もちろん，この定理はチューリング機械の定理と同様に，導出ステップの定数倍の差というものが問題に対して本質的であるかないかということを議論するためのものであり，実際の効率向上とは直接関係ない．空節までの導出のステップ数にはバックトラックによって無駄になった導出はステップ数に含まれず，導出のステップ数が少なくなった分だけバックトラックや1回のユニフィケーションの手間が増大するので，必ずしも実際の実行時間が短くなるわけではない．

Pからacc (P) を構成する方法は簡単で，一言でいうとunfold変換 (文献 (3) 参照，ただし文献 (3) の変換とは少し異なる) を必要な分だけ繰り返すことである．

まず，appendプログラムの例をとって説明しよう ((例4．1) とする)．

```
P = {
  append ([ ], X, X) <- ,
  append ([H|X], Y, [H|Z])
    <- append (X, Y, Z)
}
```

とすると，acc (P) はPにappendをappend自身でunfoldingした節を加えて，

```
acc (P) = {
  append ([ ], X, X) <-,
  append ([H], X, [H|X]) <-,
  append ([H|X], Y, [H|Z])
    <- append (X, Y, Z),
  append ([H1, H2|X], Y, [H1, H2|Z])
    <- append (X, Y, Z)
}
```

とすれば，acc (P) はPの場合とくらべて，同じゴールに対する (最小の) 導出のステップ数は，もとのステップ数をnとすると$\lceil(1/2)n\rceil$ステップですむ．更に，acc (acc (…P…)) というようにこの作業を繰り返せば導出のステップ数はいくらでも短くできる．結局，unfoldingすることによって前もって推論を行っておくわけである．

原理は簡単であるが，定理の証明は面倒である．以下，いくつかの補題を用いて定理を証明するが，導出 (SLD導出) についての定義などが重要となるので，簡単に実際のPrologの処理系との違いを述べると，

(1) Prologでは縦型探索を行うが，SLD導出は横型探索を行うのに近いイメージであり，そのうちの1つでも探索に成功すればよい．また，解が求まるか，求まらないかだけが重要で重複して求まってもよい，

(2) Prologでは，ゴール中のリテラルのうち常に最も左のリテラルが導出に使用されるが，SLD導出ではどのリテラルが導出に使用されてもよい，

である．なお，SLD導出に関する定義などは，最初のゴール節中のリテラルを1個に限るという点を除いて文献 (1) に従うものとし，また，入力節の変数名のrenamingなどについてもそのたびにことわらないものとする．

まず，Pからacc (P) を構成する方法を述べる．

[**定義4．1**]　プログラムPを$P=\{C_1, \cdots, C_n\}$とする．

$$C_i = (A \leftarrow B_1 \wedge \cdots \wedge B_m)$$

とするとき，プログラムacc (P) は，

(1) Pを含む，

(2) C_iのbody部のすべてのリテラルについて1回以下unfoldingした節をすべて含む，

である．

acc (P) とPが等価 (最小不動点が同じ) であることは明らかであろう．

(例4．2)

```
P = {
  p (a) <-, p (b) <-,
  q (X, Y) <- p (X) ∧p (Y)
}
```

のとき，

```
acc (P) = {
  p (a) <-, p (b) <-,
```

q (a, a) <−, q (a, b) <−,
q (b, a) <−, q (b, b) <−,
q (a, Y) <−p (Y), q (b, Y) <−p (Y),
q (X, a) <−p (X),
q (X, b) <−p (X),
q (X, Y) <−p (X)∧q (Y)
}

である.

［**定義 4．2**］　導出$G_0=\{<-A\}, G_1, \cdots, G_n=\square$（入力節は$C_1, \cdots, C_n$, mgu は$\theta_1, \cdots, \theta_n$）において,

$$G_p=(<-A_1\wedge\cdots\wedge A_i\wedge\cdots\wedge A_m)$$

とすると，原子式A_iの子孫とは,

(1)　A_i自身,

(2)　$G_q=(<-A_{q1}\wedge\cdots\wedge A_{qh}\wedge\cdots\wedge A_{qj}\wedge\cdots\wedge A_{qk})$

$$G_{q+1}=<-A_{(q+1)1}\wedge\cdots\wedge(B_1\wedge\cdots\wedge B_r)\wedge\cdots\wedge A_{(q+1)j}\wedge\cdots\wedge A_{(q+1)k}$$

においてA_{qj}がA_iの子孫であるとき，$A_{(q+1)j}$, である．また，A_iがA_jの子孫であるとき，A_jをA_iの先祖という．

［**定義 4．3**］　節Cのリテラル数を lit (C) で表す.

＜**補題 4．1**＞　導出を，$G_0=\{<-A\}, G_1, \cdots, G_n=\square$（入力節は$C_1, \cdots, C_n$, mgu は$\theta_1, \cdots, \theta_n$）とする.

このとき,

$$T=\{i|\mathrm{lit}(C_i)\geq 3\},$$
$$S=\{i|\mathrm{lit}(C_i)=1\}$$

とすると,

$$\sum_{i\in T}(\mathrm{lit}(C_i)-2)=|S|-1$$

である.

《証　明》

G_kとG_{k+1}のリテラル数を考える.

(1)　$\mathrm{lit}(C_{k+1})=1$のとき

$$\mathrm{lit}(G_k)-1=\mathrm{lit}(G_{k+1})$$

(2)　$\mathrm{lit}(C_{k+1})=2$のとき

$$\mathrm{lit}(G_k)=\mathrm{lit}(G_{k+1})$$

(3)　$\mathrm{lit}(C_{k+1})\geq 3$のとき

$$\mathrm{lit}(G_k)+\mathrm{lit}(C_{k+1})-2=\mathrm{lit}(G_{k+1})$$

よって，$\mathrm{lit}(G_0)=1, \mathrm{lit}(G_n)=0$より,

$$\mathrm{lit}(G_n)-\mathrm{lit}(G_0)=\sum_{i\in T}(\mathrm{lit}(C_i)-2)-|S|=-1$$

よって,

$$\sum_{i\in T}(\mathrm{lit}(C_i)-2)=|S|=-1\quad\square$$

＜**補題 4．2**＞　$P\cup\{<-A\}$の導出，$G_0=\{<-A\}, G_1, \cdots, G_n=\square$（入力節は$C_1, \cdots, C_n$, mgu は$\theta_1, \cdots, \theta_n$）があるとき，次の性質を満たす$P\cup\{<-A\}$の導出，$G_0'=\{<-A\}, G_1', \cdots, G_n'=\square$（入力節は$C_1', \cdots, C_n'$, mgu は$\theta_1', \cdots, \theta_n'$）が存在する.

(1)　任意のゴール$G_p'=(<-A_1\wedge\cdots A_i\cdots\wedge A_m)$において，$A_i$の子孫が$G_q'$から$G_{q+1}'$を導出する際に使われる原子式とすると，もし，$C_{q+1}'$が単位節ならば，$C_{p+1}', \cdots, C_q'$の中には単位節以外の節はない.

(2)　$\theta_1\ \theta_2\cdots\theta_n$と$\theta_1'\theta_2'\cdots\theta_n'$は，variants である.

《証　明》

スイッチング補題(1)を繰り返し使用すればよい．□

補題 4．2 は，単位節が入力節となるとき，その単位節を，単位節とユニファイする原子式の先祖がゴール中に初めて現れる直後（の一続き）の入力節として持ってこれることを示している.

＜**補題 4．3**＞　$P\cup\{<-A\}$の導出，$G_1, \cdots, G_n=\square$（入力節は，$C_1, \cdots, C_n$, mguは，$\theta_1, \cdots, \theta_n$）において,

$$G_{p-1}=<-A_{(p-1)1}\wedge\cdots\wedge A_{(p-1)q}\wedge\cdots\wedge A_{(p-1)h}$$
$$G_p=<-(A_{(p-1)1}\wedge\cdots\wedge(B_1\wedge\cdots\wedge B_m)\wedge\cdots\wedge A_{(p-1)h})\theta_p$$
$$C_p=A<-B_1\wedge\cdots\wedge B_m$$

とし，$A_{(p-1)q}$が$G_{(p-1)}$において選択された原子式，G_pからG_{p+k}までの導出で，$B_i\theta_p, \cdots, B_j\theta_p$の子孫が選択されるものとする．このとき，節$C_p$において，$B_i, \cdots, B_j$を$C_{p+1}, \cdots, C_{p+k}$で unfolding した節を$C_p'$とすると，$C_p'$を用いて，$G_{p-1}$から$G_p'$を導出することができ（acc (P) ∪{<−A} の導出），G_p'はG_{p+k}の variant となる.

《証　明》

unfolding の性質，および unfolding した原子式に関する帰納法で容易に証明できる．□

＜**補題 4．4**＞　$P\cup\{<-A\}$のnステップの導出，$G_0=\{<-A\}, \cdots, G_n=\square$（ただし，$n>1$, 入力節は$C_1, \cdots, C_n$）があったとき，acc (P) ∪ {<−A} の導出で，$(n-|S|)$ステップの導出が存在する.

（ただし，$S=\{i|\mathrm{lit}(C_i)=1\}$.）

《証　明》

まず，補題 4．2 によって補題 4．2 の性質を満たす導出,

$$G_0'=\{<-A\}, \cdots, G_n'=\square$$
（入力節は，$C_1', \cdots, C_n'$）

を得ることができる.

次に,

$\mathrm{lit}(C_p')>1$かつ$\mathrm{lit}(C_i')=1\ (p<i\leq q)$かつ
$(\mathrm{lit}(C_{q+1}')>1$もしくは$q=n)$

である導出の一部, G_{p-1}', G_p', …, G_q'をC_p'をunfoldingした適当な節, C_p''を用いると, 補題4.3によって,

G_{p-1}', G_p''

という導出で置き換えることができる(正確には, G_q'とG_p''がvariantsになる).

この操作によって導出のステップ数は, $q-p$, すなわち使用された単位節の数だけ減らすことができる. この操作は, すべての単位節について行うことができるので, 補題が成立する. □

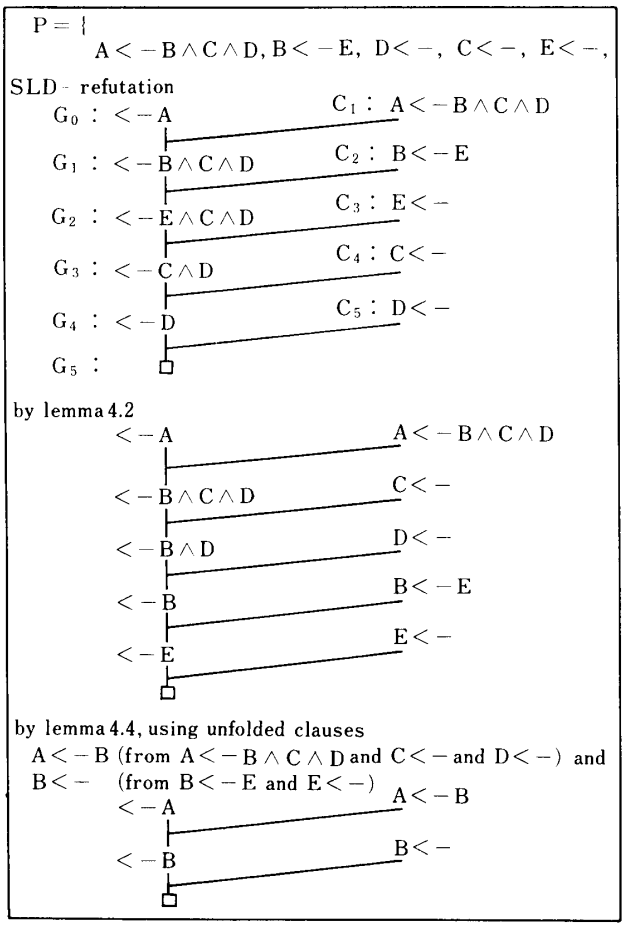

Fig. 4 An example of getting shorter resolution.

[**定理4.1**] P∪{<−A}の導出, $G_0=\{<-A\}$, G_1, …, $G_n=\square$(入力節は, C_1, …, C_n)が存在するとき, acc(P)∪{<−A}の導出で, $\lceil(5/6)n\rceil$ステップ以下であるものが存在する.

《証　明》

$n=1$の場合は, 明らかに成立するので, $n>1$の場合を考える.

まず,

$S=\{i \mid \mathrm{lit}(C_i)=1\}$

$R=\{i \mid \mathrm{lit}(C_i)=2\}$

$T=\{i \mid \mathrm{lit}(C_i)\geq 3\}$

とする.

(1) $|R|\leq 2(|T|+|S|)$の場合

まず, 補題4.4によって導出のステップ数を, $|S|$ステップ減らすことができる. ここで, 補題4.1より,

$|T|<|S|$. よって$|R|<4|S|$. よって,

$n=|T|+|R|+|S|<6|S|$

$|S|/n>1/6$

$(n-|S|)/n=1-(|S|/n)<5/6$

よって定理が成立する.

(2) $|R|>2(|T|+|S|)$の場合

まず, $i\in R$であるC_iについて補題4.2と同様の操作を行い, 導出G_0', …, G_n'(入力節は, C_1', …, C_n')を得る.

次に, 入力節の組, $<C_p', C_{p+1}'>$ $(p+1\in R)$を考える. このとき, 同じインデックスが2回現れないような$<C_p', C_{p+1}'>$を要素とする集合Qで, $|Q|>\llcorner|R|/2\lrcorner$である集合が存在する. このとき, 補題4.3より, $<C_p', C_{p+1}'>\in Q$に対応する導出の一部,

G_{p-1}', G_p', G_{p+1}'

をC_p'をunfoldingしたacc(P)中の適当な節C_p''を用いることによって,

G_{p-1}', G_p''

で置き換える(G_{p+1}'とG_p''がvariantsになる)ことができる. この操作は, すべてのQの要素について繰り返し適用できるので, 結局, 推論ステップは,

$\llcorner|R|/2\lrcorner$

ステップ減らすことができる.

このとき, $|R|\geq 3$, $n<2|R|$を考えると,

$(\llcorner|R|/2\lrcorner/n)>1/6$. よって,

$(n-\llcorner|R|/2\lrcorner)/n=1-(\llcorner|R|/2\lrcorner/n)<5/6$

よって定理が成立する. □

最後に定理4.1を繰り返し適用して, 定理4.2(線形加速定理)を得る.

[**定理4.2**] 任意のプログラムP, 任意の正数k (≤ 1)に対し, 次の性質を満たすプログラムP'を構成できる.

(1) P∪{<−A}の導出でステップ数がnのものが存在するとき, P'∪{<−A}の導出でステップ数が$\lceil kn\rceil$以下であるものが存在する.

(2) PとP'は等価である.

《証　明》

まず, 定理4.1の方法を繰り返し適用し, ある定数m(この数は, kから求めることができる)より大きいnについて定理が成り立つようなプログラムP''を得ることができる. 次に, n $(\leq m)$に対しては, プログラムP中の各述語記号pに対し, ゴール<−p(X,

Y, …, Z) のmステップ以内の導出 (そのときの answer substitution をθとする) をすべて求め, p (X, Y, …, Z) θ<−という節をすべてP″に追加してP′を得ればよい.

□

なお, 定理4.2はスイッチング補題を用いることにより, ゴール中の最左リテラルが導出に使用されるという制限をつけても設立するので, 横型探索 (OR並列処理) を行うProlog処理系についても成立することになる. また, 現在, 証明はしていないが, 定理4.2においてP ∪ {<−A} の導出の answer substitution とP′ ∪{<−A} の導出の answer substitution とは, variants になると思われる.

5. おわりに

Shapiroの帰納的推論システム (MIS.) の改良を動機として, 上述の3つの結果を得ることができたが, 残念ながら本来の目的であるシステムの改良については達成できなかったし, 画期的に向上させられる見込みも, 現在, 得られていない. 帰納的推論を行う方法で最も有力なものは, MIS. においても採用されている枚挙による推論であるが, だいたいの場合, 効率が本質的に悪い. この効率の悪さは, おそらく一般的な推論を行おうとする場合避けられないものであろう. しかしながら, 推論するべきクラスに対して適切な制限 (constraints) が与えられれば, その効率を大幅に改良できると思われる. 制限は,

(1) 推論すべきプログラムの長さ, パターン,

(2) 推論すべきクラスに関する知識,

などによって与えることができると思われる. しかしながら,「これらの制限をどのような知識表現で与えればよいのか」,「制限を与えた場合, どのようにすれば効率的な推論が行えるか」については, まだわかっていない. 今後は, この点を中心に帰納的推論の効率化にとりくんでいきたい.

謝　辞

日ごろ, お世話になっている大須賀研究室の皆様に感謝の意を表します.

◇参考文献◇

(1) Lloyd, J. W.: Foundations of Logic Programming, Springer-Verlag (1984).
(2) Shapiro, E. Y.: Algorithmic Program Debugging, The MIT Press (1982).
(3) 佐藤泰介, 玉木久夫: Prologに於けるプログラム変換, Proc. of The Logic Programming Conference, in Japanese (1983).
(4) 石坂裕毅: 汎化を用いたモデル推論の能力について, 第5回情報システムシンポジウム資料 (1985).
(5) Plotkin, G. D.: A note on inductive generalization, Machine Intell., Vol. 6, pp. 153-163 (1970).
(6) 阿久津達也: 学習と知識の獲得に関する研究, 東京大学工学系研究科航空学専門課程修士論文 (1986).

〔担当編集委員: 中川裕志〕

Appendix

An example of inference from trace data.

```
?-listing(append).
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

yes
?-pis.
***** Program Inference System from Trace Data *****

                                    by  T.Akutsu

Input Goal append([a,b],[c],X).
Traced Data is
          :-append([a,b],[c],[a,b,c])
          :-append([b],[c],[b,c])
          :-append([],[c],[c])
          :-true
```

```
+++

Inferred Clauses are
          append([X|Y],[c],[X,U|V]):-append(Y,[c],[U|V]).
          append([],[c],[c]):-true.


Input Goal append([c],[d],[c,d]).
Traced Data is
          :-append([c],[d],[c,d])
          :-append([],[d],[d])
          :-true


++

Inferred Clauses are
          append([X|Y],[Z],[X,V|W]):-append(Y,[Z],[V|W]).
          append([],[X],[X]):-true.


Input Goal append([a],[],X).
Traced Data is
          :-append([a],[],[a])
          :-append([],[],[])
          :-true


++

Inferred Clauses are
          append([X|Y],Z,[X|V]):-append(Y,Z,V).
          append([],X,X):-true.


Input Goal
Quit from PIS. .....

yes
```

著 者 紹 介

阿久津達也（正会員）

昭和59年東京大学工学部航空学科卒業．昭和61年同大学院修士課程修了．現在，同大学院博士課程情報工学専門課程在学中．論理型プログラミング，帰納的推論の研究に従事．情報処理学会，日本ソフトウェア科学会会員．

大須賀節雄（正会員）

昭和32年3月東京大学工学部航空学科卒業．同年4月富士精密工業（現日産自動車）株式会社入社．昭和36年1月東京大学航空研究所助手．昭和42年2月東京大学宇宙航空研究所助教授．昭和56年8月東京大学工学部教授．境界領域研究施設．工学博士．

研究テーマ：知識処理システム，データベース・システム，次世代情報処理方式，マン・マシン・コミュニケーション，CAD，情報処理基礎理論．