飽和節の構造分析に基づいた再帰的論理プログラムの帰納

犬塚信博[†] 古澤光枝[††]
世木博久[†] 伊藤英則[†]

本論文では，論理プログラムを事例から帰納するためのボトムアップ方式の帰納学習アルゴリズム，MRIを提案する．MRIは，非常に少数の事例から，非再帰節1つと，1つあるいはそれ以上の再帰節を含む論理プログラムを帰納することができる．この方法は事例の飽和節の分析に基づいている．Idestam-Almquistは，ILPシステムTIMのアルゴリズムにおいて，述語で処理され受け渡される値の組の流れを表すため，経路構造と呼ぶ概念を用い，経路の中の繰返し構造を分析した．本論文では，経路構造の延長および経路構造の差分の概念を新たに導入し，再帰呼び出しを延長された経路構造に対応づけ，再帰節を経路間の差分に対応づけた．アルゴリズムでは，まず飽和節から作られる事例の経路構造を選択し，それによって，非再帰節で処理されるまでの項の流れが表現されると仮定する．次に，これを順次短縮することで，再帰節で処理されるまでの経路構造を求め，対応する差分から各再帰呼び出しをする再帰節を求める．これにより，従来ボトムアップ法で求められなかった複数の再帰節を持つプログラムを帰納することが可能となった．論文では，本方法に基づいたアルゴリズムとその実装について述べ，他のシステムとの比較実験の結果を示す．

Induction of Recursive Logic Programs Based on Structural Analysis of Saturations

NOBUHIRO INUZUKA,[†] MITSUE FURUZAWA,[††] HIROHISA SEKI[†] and HIDENORI ITOH[†]

In this paper we present a bottom-up algorithm called MRI to induce logic programs from their examples. This method can induce programs with a base clause and one or more recursive clauses from a very small number of examples. MRI is based on the analysis of saturations of examples. An ILP system TIM, developed by Idestam-Almquist, introduced a path structure to express a stream of terms processed by predicates. In this paper, we introduce the concepts of extension and difference of path structures. A recursive call corresponds with an extension of path structure and a recursive clause can be expressed as a difference between a path structure and its extension. First, MRI assumes a path structure that expresses a last recursive call, which should be processed by a non-recursive clause. Then MRI generates path structures of which the path structure of the last call is an extension. A recursive clause corresponds with a difference of them. MRI is the first bottom-up ILP algorithm that can induce programs with more than one recursive clause. The paper describes the algorithm and its implementation. Experimental results are shown comparing with another system.

1. はじめに

帰納論理プログラミング（Inductive Logic Programming: ILP）は，与えられた事例と背景知識から，事例を説明する理論を論理プログラムとして導出する枠組みである．このとき，少数の事例から論理プログラムを帰納することは，自動プログラミングへの応用の点から興味ある問題である．なぜなら，簡単なプログラムを導くために多くの事例が必要では，実際上プログラミングの補助に利用することはできない．再帰的なプログラムを扱うことも，ILPの1つの話題となってきた．FOIL[1),2)]，FOIL-I[3)]，Progol[4)]，GOLEM[5)]などのいくつかのILPシステムでは，実際，再帰的なプログラムを帰納することができる．しかし，このとき必要とする事例の数は少なくない．たとえば，リストに要素が帰属することを判断する再帰的述語memberのプログラムを帰納する場合，無作為に選んだ事例から70%以上の割合で正しいプログラムを得るには，FOILでは事例全体の半数以上を，また，Progolでは8割以上を与えなければならないこ

† 名古屋工業大学
Nagoya Institute of Technology
†† 三菱電気エンジニアリング株式会社
Mitsubishi Electric Engineering Company Limited

とが実験で確認されている[3]☆.

少数事例から再帰的なプログラムを帰納する問題においては，事例の構造的な分析に基づいたボトムアップ手法が成功をおさめてきた．LOPSTER[6]とその後継システムCRUSTACEAN[7),8]は，少数の事例から効率良く再帰的なプログラムを導き出すことができる．またTIM[9]は，飽和節（後述）の構造分析に基づいて，やはり効率良く再帰プログラムを導く．しかしながら，これらのシステムが帰納できる再帰プログラムのクラスは，非常に限られている．たとえば，LOPSTERとCRUSTACEANは，単位節1つと，頭部リテラル以外には再帰的な本体リテラルが1つあるのみの再帰節の，合計2節からなる再帰プログラムしか帰納できない．FORCE2[10]は，1つの非再帰節と1つの末尾再帰節からなるプログラムを帰納できるが，しかし，与えられた事例のうち，どれが非再帰節のみによって導き出せるものであり，どれが再帰節も必要としているのかを，システムに提示しなければならない．TIMは，そうした情報を与えなくても，FORCE2と同じクラスのプログラムを帰納することが可能である．

1つの非再帰節と1つの末尾再帰節からなるFORCE2やTIMのクラスは，計算論的に重要であるが，実用上あるいは柔軟性の点からは不十分である．本論文では，1つの非再帰節と，1つあるいはそれ以上（複数）の再帰節からなる再帰的論理プログラムを帰納することのできる，効率の良い方法を与える．

次章では，まず準備として必要な定義を与える．3章では，例を使って事例の構造分析の方法を詳しく述べる．4章では構造分析の方法に基づいた帰納アルゴリズムMRIを説明し，その実装について述べる．いくつかのプログラムを帰納させる実験の結果を5章で述べ，最後に6章でアルゴリズムの扱うことができるクラスについて考察し，計算複雑さについての結果を述べる．

2. 準　　備

ある目標述語 p に対して，p の正事例と負事例が，変数を含まない単位節として各々いくつか与えられる．また，背景知識として，いくつかの述語の定義が論理プログラムとして与えられる．このとき，正事例を満たし，負事例を満たさない p の定義を論理プログラムとして導くことがILPアルゴリズムの目的である．ただし本論文では，他の多くのILPアルゴリズムと同様，目標述語，背景知識はともにホーン節で記述されるものと仮定する．

☆ 要素a，b，cが重複せずに現れる長さ3までのリストに限定し，事例総数が64の場合．

表1 背景知識
Table 1 Background knowledge.

述語	type	説明
dec$(+A,-B,-C)$	p	$A=[B\|C]$
nil$(+A)$	c	$A=[]$
eq$(+A,+B)$	c	$A=B$
last$(+A,-B,-C)$	p	C は A の最後の要素，B は C を除いたリスト
large$(+A,+B)$	c	$A\geq B$
father$(+A,-B)$, etc.	p	A は B の父親

type：c = 判定述語，p = 経路述語

本論文では，述語のクラスとして2種類のものだけを扱う．第1のクラスは，すべての引数が入力引数である述語である．つまり，この述語を持つリテラルが呼ばれたとき，すべての引数に値が束縛されていなければならない．もう一方のクラスの述語は，ただ1つの入力引数を持つdeterminate述語☆☆である．つまり，1つの引数が入力引数で，それ以外に少なくとも1つの引数があり，それらがすべて出力引数である．さらに，入力引数に値が束縛されてこの述語が呼ばれたとき，すべての引数に唯一の値が束縛されなければならない．たとえば，**表1**のdecは1つのリストを入力とし，その頭部とそれ以外に分けて出力する述語であり，この条件を満たす．第1のクラスの述語を**判定述語**，第2のクラスの述語を**経路述語**，またこれらからなるリテラルをそれぞれ**判定リテラル**，**経路リテラル**と呼ぶ．経路リテラルは，ある項を入力することで別の項を出力するものであり，後述の経路を構成する．

本論文では飽和節（saturation），経路（path），経路構造（path structure）の各用語を文献9)に習って用いるが，若干の変更をともなう．正事例 E の理論 T に関する**飽和節**（saturation）とは，次の(1)，(2)を満たす確定節 F である．

(1) $T\wedge E\equiv T\wedge F$,
(2) 節 F' が $T\wedge E\equiv T\wedge F'$ を満たすならば $F'\Rightarrow F$.

E の飽和節は，理論 T と合わせることで意味が変わらない（条件(1)）節のうち，最も制限のきつい節（条件(2)）である．

上の条件に加えて本論文では，文献9)同様に，飽

☆☆ 入力引数の値に対して，真となる出力引数の値が1通りだけであるリテラルをdeterminateリテラルといい，determinateリテラルを作る述語をdeterminate述語と呼ぶ．determinateリテラルの概念は，GOLEM[5]ではじめて導入され，FOIL[1),2)]などの多くのシステムで使われている．

和節 F と E は同じ頭部リテラルを持ち，F の本体リテラルは E に関連するものに限定する．ここで，E そのものは E に関連し，また，飽和節の本体リテラルのうち，その入力引数のすべてが，E に関連するリテラルに現れるものも，E に関連すると定義する．

たとえば，理論 T が述語 dec, eq, nil を定義する論理式（表1参照）からなる場合，事例 $\mathsf{rem}(a,[a,b,a],[b])$ の T に関する飽和節は次のとおりである．

$$\begin{aligned}\mathsf{rem}(a,[a,b,a],[b]) \leftarrow\ & \mathsf{dec}([a,b,a],a,[b,a]), \quad (1)\\ & \mathsf{dec}([b,a],b,[a]),\mathsf{dec}([a],a,[]),\mathsf{dec}([b],b,[]),\\ & \mathsf{eq}([a,b,a],[a,b,a]),\mathsf{eq}([b,a],[b,a]),\mathsf{eq}([a],[a]),\\ & \mathsf{eq}([],[]),\mathsf{eq}([b],[b]),\mathsf{eq}(a,a),\mathsf{eq}(b,b),\mathsf{nil}([]).\end{aligned}$$

正事例 E とその飽和節 F に対して，F に含まれるリテラルの列 $(l_1,l_2,\cdots,l_t)$ が次の条件（1）～（3）を満たすとき，この列を E の T に関する**経路**といい，

$$P=[p_1\text{–}b_1,p_2\text{+}a_2\text{–}b_2,\cdots,p_t\text{+}a_t\text{–}b_t] \quad (2)$$

と書く．ここで p_i はリテラル l_i の述語記号である．

（1） l_1 は F の頭部リテラル，$l_2,\cdots,l_t$ は本体リテラルである．

（2） l_i $(i=2,\cdots,t)$ は経路リテラルであり，その a_i 番目の引数は入力引数，b_i 番目の引数は出力引数である．

（3） l_i と l_{i+1} $(i=1,\cdots,t-1)$ は，l_i の第 b_i 引数と l_{i+1} の第 a_{i+1} 引数に共通する項を持つ．

リテラルの列 $(l_1,l_2,\cdots,l_t)$ が経路であるとは，l_i で出力される項が，l_{i+1} で入力引数に渡され，l_1 から l_t までつながっていることを表す．上の表記は，各リテラルにおいて，項の受け渡しの起こる引数番号を合わせて記述している．

上述の例では，飽和節 (1) から選ばれたリテラルの列 ($\mathsf{rem}(a,[a,b,a],[b])$, $\mathsf{dec}([a,b,a],a,[b,a])$, $\mathsf{dec}([b,a],b,[a])$, $\mathsf{dec}([a],a,[])$) は，経路となる．第1リテラルの第2引数 $[a,b,a]$ が第2リテラルの第1引数に入力され，その第3引数から出力される項 $[b,a]$ が第3リテラルの第1引数に入力され，さらに第4リテラルへ続く．この経路は次のように記述される．

$$P_1=[\mathsf{rem}\text{–}2,\mathsf{dec}\text{+}1\text{–}3,\mathsf{dec}\text{+}1\text{–}3,\mathsf{dec}\text{+}1\text{–}3]$$

経路 (2) の最後のリテラル l_t の第 b_t 引数にある項は，最終的にこの経路で出力される項であり，これをこの**経路の値**といい，$P[E]$ と書く．上の例で経路の値は，$P_1(\mathsf{rem}(a,[a,b,a],[b]))=[]$ である．

n 個の経路の順序集合を **n-経路組**，あるいは単に**経路組**という．$\mathcal{P}=(P_1,\cdots,P_n)$ が n-経路組，q が n 引数述語であるとき，$q(\mathcal{P}[E])$ は，q の各引数に各経路の値を代入してできるリテラル $q(P_1[E],\cdots,P_n[E])$ を意味する．

述語記号 p を持つ k 引数の正事例 E に対して，k-経路組 $\mathcal{PS}=(P_1,\cdots,P_k)$ の各経路 P_i $(i=1,\cdots,k)$ が p–i で始まる経路であるとき，この経路組 $\mathcal{PS}$ を E の**経路構造**という．このときも $p(\mathcal{PS}[E])$ は，リテラル $p(P_1[E],\cdots,P_k[E])$ を表す．

経路組は，単に経路の組を表すのに対し，経路構造は k 引数の事例に対し，各引数の項に対応する経路を組にして表すために用いる．この2つの区別は本研究で新たに導入した．これらについての例は次章で詳しく述べる．

次に，本論文で提案するアルゴリズムを明確に述べるため，新たに経路の差分，および経路構造の差分の概念を提案する．

2つの経路 P と P' が次のように書かれるとき，P' は P の**延長**である．

$$P=[p_1\text{–}b_1,p_2\text{+}a_2\text{–}b_2,\cdots,p_t\text{+}a_t\text{–}b_t] \quad (3)$$

$$\begin{aligned}P'=[&p_1\text{–}b_1,p_2\text{+}a_2\text{–}b_2,\cdots,p_t\text{+}a_t\text{–}b_t, \quad (4)\\ &p_{t+1}\text{+}a_{t+1}\text{–}b_{t+1},\cdots,p_{t+s}\text{+}a_{t+s}\text{–}b_{t+s}]\\ &(s\geq 0),\end{aligned}$$

また，このとき次の D を P' の P からの（**経路の**）**差分**と呼ぶ．

$$\begin{aligned}D=[&p_1\text{–}b_1,p_{t+1}\text{+}a_{t+1}\text{–}b_{t+1},\cdots\\ &\cdots,\ p_{t+s}\text{+}a_{t+s}\text{–}b_{t+s}].\end{aligned}$$

経路 P が，経路構造 $\mathcal{PS}$ のある経路の延長であるとき，P は $\mathcal{PS}$ を延長するという（**経路構造に対する延長**）．さらに，経路組 $\mathcal{P}$ に含まれるすべての経路が，経路構造 $\mathcal{PS}$ を延長するとき，$\mathcal{P}$ は $\mathcal{PS}$ を延長するという（**経路組の延長**）．また，経路構造 $\mathcal{PS}$ が，別の経路構造 $\mathcal{PS}'$（すなわち経路組）を延長するとき，$\mathcal{PS}$ を $\mathcal{PS}'$ の延長（**経路構造の延長**），$\mathcal{PS}'$ を $\mathcal{PS}$ の**部分経路構造**と呼ぶ．

経路組 $\mathcal{P}=(P_1,\cdots,P_n)$ が，経路構造 $\mathcal{PS}=(P'_1,\cdots,P'_k)$ を延長する場合を考える．経路 P_i によって延長される経路構造 $\mathcal{PS}$ に含まれる経路を $P'_{i'}$ とし，D_i を P_i の $P'_{i'}$ からの差分 $(i=1,\cdots,1)$ としたとき，$(D_1,\cdots,D_n)$ を $\mathcal{P}$ の $\mathcal{PS}$ からの差分といい（**経路組の差分**），$\mathcal{P}-\mathcal{PS}$ と書く．

これらの概念を用いて事例の分析を行う方法を，次章で例を用いて説明する．

3. 実行時トレースの経路構造による分析

まず，1つの非再帰節と2つの再帰節を持つ，次の典型的な論理プログラムを考える．

$$\mathsf{rem}(A,B,C) \leftarrow \mathsf{nil}(B),\mathsf{nil}(C). \quad (5)$$

$$\mathsf{rem}(A,B,C) \leftarrow \mathsf{dec}(B,B_1,B_2), \quad (6)$$

```
1    ← rem(a,[a,b],[b])
2        ← dec([a,b],a,[b])
3        ← eq(a,a)
4        ← rem(a,[b],[b])
5            ← dec([b],b,[])
6            ← dec([b],b,[])
7            ← eq(b,b)
8            ← rem(a,[],[])
9                ← nil([])
10               ← nil([])
```

図1 remove の実行時トレース
Fig. 1 An execution trace of remove.

$$\mathsf{rem}(A,B,C) \leftarrow \begin{array}{l}\mathsf{eq}(A,B_1),\mathsf{rem}(A,B_2,C).\\ \mathsf{dec}(B,B_1,B_2),\\ \mathsf{dec}(C,C_1,C_2),\\ \mathsf{eq}(C_1,B_1),\mathsf{rem}(A,B_2,C_2).\end{array} \quad (7)$$

このプログラムで使用されている述語，nil，dec，eq の定義（表1参照）が背景知識に含まれているとする．表にある記号 'c' と 'p' はそれぞれ判定述語，経路述語を示している．eq と nil は判定述語であり，dec は経路述語である．

図1 に，次のゴールの実行時トレースを示す．

$$\leftarrow \mathsf{rem}(a,[a,b],[b]) \quad (8)$$

ここで，ゴール $\mathsf{rem}(a,[a,b],[b])$ を rem の正事例 E と見るならば，この事例 E およびトレースに現れるリテラル $\mathsf{rem}(a,[b],[b])$，$\mathsf{rem}(a,[],[])$ は，E の経路構造を使って次のように表すことができる．

$$\mathsf{rem}(a,[a,b],[b]) = \mathsf{rem}(\mathcal{PS}_1[E]) \quad (9)$$

$$\mathsf{rem}(a,[b],[b]) = \mathsf{rem}(\mathcal{PS}_2[E]) \quad (10)$$

$$\mathsf{rem}(a,[],[]) = \mathsf{rem}(\mathcal{PS}_3[E]) \quad (11)$$

ここで，各経路構造は次のとおりである．

$$\mathcal{PS}_1 = ([\text{rem–1}],[\text{rem–2}],[\text{rem–3}])$$

$$\mathcal{PS}_2 = ([\text{rem–1}],[\text{rem–2, dec+1–3}],[\text{rem–3}])$$

$$\mathcal{PS}_3 = ([\text{rem–1}],[\text{rem–2, dec+1–3, dec+1–3}],[\text{rem–3, dec+1–3}])$$

たとえば，式 (10) では，$\mathsf{rem}(a,[b],[b])$の各引数をゴール (8) から作る方法が，$\mathcal{PS}_2$ によって記述されている．$\mathsf{rem}(a,[b],[b])$の第1引数 a は，ゴール (8) の第1引数そのものであり，このことは，[rem–1] と記述される．また，第2引数 $[b]$ は，ゴール (8) の第2引数 $[a,b]$ を dec の第1引数に与えたとき第3引数に出力されるものであり，[rem–2, dec+1–3] と記述できる．第3引数も同様である．

このとき $\mathcal{PS}_1$ は $\mathcal{PS}_2$ の部分経路構造であり，また $\mathcal{PS}_2$ は $\mathcal{PS}_3$ の部分経路構造である．そこでこれらの差分 $\mathcal{PS}_2 - \mathcal{PS}_1$ および $\mathcal{PS}_3 - \mathcal{PS}_2$ を考えよう．

$$\mathcal{PS}_2 - \mathcal{PS}_1 = ([\text{rem–1}],[\text{rem–2, dec+1–3}],[\text{rem–3}]) \quad (12)$$

$$\mathcal{PS}_3 - \mathcal{PS}_2 = ([\text{rem–1}],[\text{rem–2, dec+1–3}],[\text{rem–3, dec+1–3}]) \quad (13)$$

すると，再帰リテラルはこれらの差分を使って下のように表現できる．

$$\mathsf{rem}(a,[b],[b]) = \mathsf{rem}((\mathcal{PS}_2 - \mathcal{PS}_1)[E])$$

$$\mathsf{rem}(a,[],[]) = \mathsf{rem}((\mathcal{PS}_3 - \mathcal{PS}_2)[E'])$$

ここで $E' = \mathsf{rem}(\mathcal{PS}_2[E]) = \mathsf{rem}(a,[b],[b])$ である．一般に，もし $\mathcal{PS}$ と $\mathcal{PS}'$ が述語 p を持つ事例 E の経路構造で，$\mathcal{PS}'$ が $\mathcal{PS}$ を延長するならば，次の式が成り立つ．

$$p(\mathcal{PS}'[E]) = p((\mathcal{PS}' - \mathcal{PS})[\mathcal{PS}[E]])$$

つまり，事例 E から経路構造 $\mathcal{PS}'$ で記述される操作によって得られるリテラル $p(\mathcal{PS}'[E])$ は，$\mathcal{PS}'$ の部分経路構造 $\mathcal{PS}$ を先に施した後 $(\mathcal{PS}[E])$，差分 $\mathcal{PS}' - \mathcal{PS}$ を施すことでも得られる．すなわち，$\mathcal{PS}[E]$ に対応する再帰呼び出しの後，$\mathcal{PS}'[E]$ に対応する再帰呼び出しまでの間に，項に施された履歴が差分 $\mathcal{PS}' - \mathcal{PS}$ に記述されている．実際，差分 (12) には，再帰節 (6) において再帰リテラルに渡される項を得るための情報がすべて含まれており，差分 (13) には再帰節 (7) の情報が含まれている．

次にトレースに含まれる判定リテラル（トレースの3，7，9，10行）について考える．リテラル $\mathsf{eq}(b,b)$ (7行目) は次のように表すことができる．

$$\mathsf{eq}(b,b) = \mathsf{eq}(\mathcal{P}[E]) \quad (14)$$

ただし，

$$\mathcal{P} = ([\text{rem–2, dec+1–3, dec+1–2}],[\text{rem–3, dec+1–2}])$$

であり，$\mathcal{P}$ は $\mathcal{PS}_2$ を延長する．ここで，$\mathcal{PS}_2$ はリテラル $\mathsf{eq}(b,b)$ を呼び出す再帰リテラル $\mathsf{rem}(a,[b],[b])$ (トレースの4行目) を表すものであった．このことは，$\mathsf{eq}(b,b)$ が再帰呼び出し $\mathsf{rem}(a,[b],[b])$ の後に呼ばれることを示す．また，これらの間の差分

$$\mathcal{P} - \mathcal{PS}_2 = ([\text{rem–2, dec+1–2}],[\text{rem–3, dec+1–2}])$$

は，この判定リテラルに渡される値を準備するために必要な，すべての情報を含んでいる．トレースにおいて，ある再帰リテラルに引き続いて呼び出されるすべての判定リテラルは，その再帰リテラルを表現する経路構造を延長する経路組によって表すことができる．

図2 はこれまでに説明してきたことを示す．

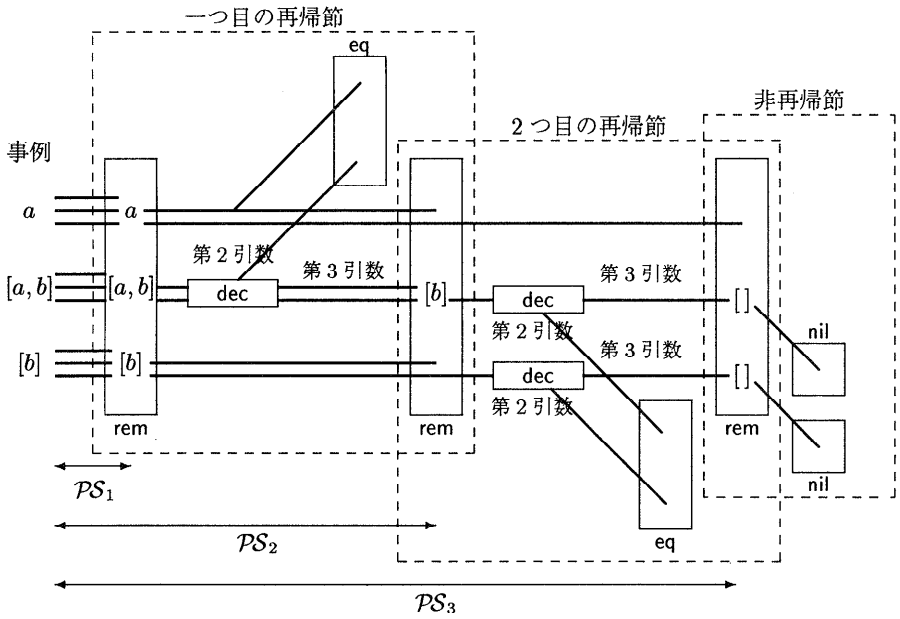


図2 項組の流れと経路構造
Fig. 2 Stream of terms and path structures.

4. 帰納アルゴリズム MRI

前章で分析したように，経路構造を用いることで，ゴールの実行時トレースを分析することができる．そこで逆に，ある与えられた事例に対して，その飽和節から可能な経路構造を選び出すことによって，そのゴールを真とする論理プログラムの実行時トレースを推測できる．アルゴリズム TIM[9]の方法は，このように実行時トレースを経路構造として再現するものであった．しかしながら TIM では，経路構造からプログラムを導出するために，その中に繰返し現れるリテラルの列を取り出し，その部分を再帰節に対応させるのみであった．したがって，複数の再帰節によって実行されたトレースのように，同じリテラル列の繰返しとならない場合は，プログラムを帰納することはできない．

MRI は繰返し構造の発見とは異なった方法で，プログラムを帰納する．このことを説明するために，3章の分析でこれまでに明らかになったことを次にまとめる．

(1) 再帰リテラルの列 rem(a,[a,b],[b]), rem(a,[b],[b]), rem(a,[],[]) のように，次々と再帰的に呼び出される再帰リテラルは，この列の順序で順に延長される経路構造によって表すことができる．

(2) このとき，列中のある再帰リテラルの経路構造 $\mathcal{PS}$ と，それより1つ手前の再帰リテラルの経路構造 $\mathcal{PS}'$ を考える．すると，差分 $\mathcal{PS}-\mathcal{PS}'$ は，前者の再帰リテラルから，後者へ渡される項を作り出すための，すべての経路リテラルの情報を含む．

(3) $p(\mathcal{PS}[E])$ をある再帰リテラルとしたとき，これから呼び出される n 引数の判定リテラルは，$\mathcal{PS}$ を延長するある n-経路組 $\mathcal{P}$ と n 引数判定述語 q により，$q(\mathcal{P}[E])$ と書くことができる．

(4) このとき，経路組 $\mathcal{P}$ の経路構造 $\mathcal{PS}$ からの差分 $\mathcal{P}-\mathcal{PS}$ は，その判定リテラルで使われる値を用意するために必要なすべての情報を含む．

上にまとめたように，経路構造を延長する経路組の，その経路構造からの差分は，この経路組に対応するリテラルのための情報をすべて含んでいる．この情報から，ある再帰節の中にある，再帰リテラル（または判定リテラル）のために必要な経路リテラルを生成できる．たとえば，式(12)にある差分 $\mathcal{PS}_2-\mathcal{PS}_1$ に対応して再帰節の本体に含まれるべきリテラルは

$$\mathsf{dec}(x_2,x_4,x_5),\mathsf{rem}(x_1,x_5,x_3)$$

である，ただし，節の頭部リテラルは $\mathsf{rem}(x_1,x_2,x_3)$ であると仮定した．実際これらのリテラルは再帰節(6)に含まれている．同様に式(13)の差分 $\mathcal{PS}_3-\mathcal{PS}_2$ に対応するリテラルは

$$\mathsf{dec}(x_2,x_4,x_5),\mathsf{dec}(x_3,x_6,x_7),\mathsf{rem}(x_1,x_5,x_7)$$

であり，やはり再帰節 (7) の本体に含まれる．

次に，この対応を形式的に与える．$\mathcal{PS}$ を事例 E の経路構造とし，$\mathcal{P}$ を $\mathcal{PS}$を延長する経路組とする．このとき $q(\mathcal{P}[E])$ が E の飽和節に含まれるならば，差分 $\mathcal{P}-\mathcal{PS}$ があり，次のように書くことができる．

$$
\begin{aligned}
(\ &[p\text{–}i_1, p_{11}{+}a_{11}\text{–}b_{11}, p_{12}{+}a_{12}\text{–}b_{12}, \cdots \\
&\qquad \cdots, p_{1n_1}{+}a_{1n_1}\text{–}b_{1n_1}], \\
&\qquad \cdots, \\
&[p\text{–}i_l, p_{l1}{+}a_{l1}\text{–}b_{l1}, p_{l2}{+}a_{l2}\text{–}b_{l2}, \cdots \\
&\qquad \cdots, p_{ln_l}{+}a_{ln_l}\text{–}b_{ln_l}]\ )
\end{aligned}
$$

このとき，$\mathcal{P}-\mathcal{PS}$ に対応するリテラルの集合は次のとおりである．

$$
\begin{aligned}
\{\ & p_{11}(\cdots \overset{\text{第}\,a_{11}\,\text{引数}}{\overset{\vee}{x_{i_1}}} \cdots \overset{\text{第}\,b_{11}\,\text{引数}}{\overset{\vee}{y_{11}}} \cdots), p_{12}(\cdots \overset{\text{第}\,a_{12}\,\text{引数}}{\overset{\vee}{y_{11}}} \cdots \overset{\text{第}\,b_{12}\,\text{引数}}{\overset{\vee}{y_{12}}} \cdots), \cdots \\
&\qquad \cdots, p_{1n_1}(\cdots y_{1(n_1-1)} \cdots \overset{\text{第}\,a_{1n_1}\,\text{引数}}{\overset{\vee}{y_{1n_1}}} \cdots \overset{\text{第}\,b_{1n_1}\,\text{引数}}{\overset{\vee}{y_{1n_1}}} \cdots), \\
&\qquad \cdots, \\
& p_{l1}(\cdots \overset{\text{第}\,a_{l1}\,\text{引数}}{\overset{\vee}{x_{i_l}}} \cdots \overset{\text{第}\,b_{l1}\,\text{引数}}{\overset{\vee}{y_{l1}}} \cdots), p_{l2}(\cdots \overset{\text{第}\,a_{l2}\,\text{引数}}{\overset{\vee}{y_{l1}}} \cdots \overset{\text{第}\,b_{l2}\,\text{引数}}{\overset{\vee}{y_{l2}}} \cdots), \cdots \\
&\qquad \cdots, p_{ln_l}(\cdots y_{l(n_l-1)} \cdots \overset{\text{第}\,a_{ln_l}\,\text{引数}}{\overset{\vee}{y_{ln_l}}} \cdots \overset{\text{第}\,b_{ln_l}\,\text{引数}}{\overset{\vee}{y_{ln_l}}} \cdots), \\
& q(y_{1n_1}, \cdots, y_{ln_l})\ \}
\end{aligned}
\tag{15}
$$

この集合を $q(\mathcal{P}-\mathcal{PS})$ と書く．

この観察に基づいて，次の手順でプログラムを帰納することができる．

I　事例 E の飽和節から，ある経路構造 $\mathcal{PS}$ を選ぶ．

II　$\mathcal{PS}$ が E の実行時トレースを表すと仮定して，最終の呼び出しである $\mathcal{PS}[E]$ から非再帰節（終了条件）を作る．

III　E の実行にかかわった再帰節を作る．

- i　$\mathcal{PS}$ のある部分経路構造 $\mathcal{PS}'$ を選び，これが $\mathcal{PS}$ の最終の1つ手前の再帰呼び出しを表すと仮定する．
- ii　差分 $\mathcal{PS}-\mathcal{PS}'$ の情報から再帰節を1つ作る．
- iii　$\mathcal{PS}'$ をさらに短くできるときは，$\mathcal{PS}'$ を $\mathcal{PS}$ として，III-i に戻る．

IV　II, III で作られた節から必要なもののみを残す．

上の III では，上述のとおり，差分をそこに含まれるリテラルに対応づけることで，再帰節を構成することができる．また，この方法は TIM と異なり，繰返し構造の発見ではなく，経路構造の短縮によってできる差分の抽出に基づくため，複数の異なった再帰節を含むプログラムも構成できる．

この手順に基づいた MRI アルゴリズムを図 **3** に示す．アルゴリズムでは，各節を構成するリテラルを，その役割から2つに分けて扱っている．1つは，節を適用するための条件のリテラル（条件部）であり，もう1つは，再帰リテラルへ値を用意するリテラル（経路部）である（経路部には経路リテラルのみが，条件部には判定リテラルと経路リテラルが含まれる）．再帰リテラル l が経路構造 $\mathcal{PS}$ によって表現される場合を考える．このとき，l を処理する節の条件部には，ある判定述語 q と $\mathcal{PS}$ を延長するある経路組 $\mathcal{P}$ に対して，式 (15) で与えられるリテラルの集合 $q(\mathcal{P}-\mathcal{PS})$ が含まれる．また，節の経路部は $\mathcal{PS}$ を延長するある経路構造 $\mathcal{PS}'$ に対して $p(\mathcal{PS}'-\mathcal{PS})$ である．ここで，p は事例の述語，つまり目標述語である．

非再帰節は条件部のみを持つ．アルゴリズムではこれを $\mathsf{condition}^{\mathsf{base}}$ と表す．再帰節は条件部と経路部の両方を持ち，各々 $\mathsf{condition}^{\mathsf{recur}}$, $\mathsf{pass}^{\mathsf{recur}}$ と表されている．

アルゴリズムは初期化の後，3つの部分（上述の II, III, IV に対応）からなる．アルゴリズムに現れる **Select** は，多くの選択肢の中から非決定的に1つを選択する非決定的な命令である．

初期化 (I) では，まず与えられた正事例の集合の中から数個の事例を選び出し，それらの飽和節を計算する．それに続く部分（II: 5〜18 行）では，事例を処理する最後の再帰呼び出しに対応する経路構造（$\mathcal{PS}_i$）を仮定（6 行）し，これに対応する節として，目標プログラムの非再帰節（終了条件に対応する）を求める．この非再帰節の条件部は，生成された経路構造を延長する経路組（$\mathcal{P}$）と，すべての判定述語の組合せを枚挙することで計算される（8〜9 行）．すべての事例に対してこの計算を行った後，それらに共通して含まれる条件部リテラルが空でなければこれを非再帰節として採用する．空である場合は，この経路構造が誤っていたと見なして5行に戻り，他の経路構造を選択する．非再帰節がただ1つであると仮定していることから，これによって非再帰節を得ることができる．

続いて III では，再帰節の候補の抽出にかかる（19〜30 行目）．まず，すでに生成された非再帰節に対応する経路構造（$\mathcal{PS}_i$）に対して，その部分経路構造（$\mathcal{PS}'_i$）を選ぶ（22 行）．$\mathcal{PS}'_i$ に対応する再帰呼び出しが，$\mathcal{PS}_i$ に対応する再帰呼び出しの直前に起きたと仮定することで，$\mathcal{PS}'_i$ に対応する再帰を含んだ $\mathcal{PS}_i$ に対応する再帰節を作り出すことができる．この再帰節の経路部は 23 行で計算される．また，条件部は，非再帰節の場

```
Input   E+, E-          : k 引数述語 p の正事例/負事例の集合
        T               : 背景知識
Output  base ∪ recursion : プログラム

# I 初期化
1   Select E+ から数個の正事例 {E1+, ..., En+} を選ぶ
2   For i = 1 to n do
3       Fi := Ei+ の T に関する飽和節
4       usedi := ø

# II 非再帰節を 1 つ選択する
5   For i = 1 to n do
6       Select Ei+ の T に関する経路構造 PSi を選択      # PSi が Ei+ の最終再帰呼び出し
7       conditionibase := ø                              #     を表現すると仮定.
8       For each q : T に含まれる判定述語,
        and each P : PSi を延長し q(P[Ei+]) ∈ Fi を満たす経路組 do
9           conditionibase := conditionibase ∪ {q(P − PSi)}  # q(P[Ei+]) がこの再帰呼び出しの終了条件に含
10          usedi := usedi ∪ {(q, P)}                        #     まれるとして, conditionibase に加える.
11  If conditon1base ∩ ··· ∩ conditonnbase = ø  go to 5   # 各事例からの終了条件の積が非空ならば, こ
12  conditionbase := ø                                    #     れを採用. 空ならば別の経路構造を調べる.
13  For each L ∈ condition1base ∩ ··· ∩ conditionnbase do  #
14      conditionbase := conditionbase ∪ L
15  For i = 1 to n do                                     # 使わなかったリテラルの情報を usedi から消す.
16      For each (q, P) ∈ usedi do
17          If q(P − PSi) ∉ condition1base ∩ ··· ∩ conditionnbase then usedi := usedi − {(q, P)}
18  base' := "p(x1, ···, xk)← conditionbase"              # 採用された非再帰節の候補.

# III 再帰節の候補を生成する
19  candidates := ø
20  For i := 1 to n do                                    # PS'i が PSi の再帰呼び出しの直前の再帰呼び出しに
21      Repeat                                            #     対応すると仮定.
22          Select PSi の部分経路構造 PS'i を選択
23          passrecur := p(PSi − PS'i)                    # 差分 PSi − PS'i から経路部を作る.
24          conditionrecur := ø
25          For each q : T の判定述語 and each P : PS'i を延長し
            q(P[Ei+]) ∈ Fi, (q, P) ∉ usedi を満足する経路組 do
26              conditionrecur := conditionrecur ∪ q(P − PSi)  # PS'i の再帰呼び出しに対する終了条件を作る.
27              usedi := usedi ∪ {(q, P)}
28          candidates := candidates                      # 採用された再帰節を candidates に加える.
              ∪{"p(x1, ···, xk)←conditionrecur ∪ passrecur"}
29          PSi := PS'i                                   # PS'i を新たに PSi としてさらに短い部分経路構造
30      Until p(PSi[Ei+]) = Ei+                           #     を調べる.
                                                          # PSi が事例と一致したら終了.
# IV 事例に無矛盾な再帰節を選択する
31  リテラル集合 base' の部分集合で, 事例集合 (E+, E-) に無矛盾な極小集合 base を欲張り法で求める.
32  再帰節集合 candidates の部分集合で事例集合 (E+, E-) に無矛盾な極小集合 recursion を欲張り法で求める.
```

図 3 MRI アルゴリズム
Fig. 3 MRI algorithm.

合と同様に, PS'_i を延長する経路組と, その項数と引数の数が一致する述語に対して式 (15) で計算される.

最後に IV では, 欲張り法 (greedy method) により非再帰節から必要なリテラルのみを残す. また, 同じく欲張り法を使って, 生成された再帰節候補の中から必要な節のみを取り出す. これは再帰節候補の中から, 与えられた事例を最も多く説明できるものを順に選んでゆくことで行っている.

アルゴリズム中, 同じ経路組が重複して何度も計算されることを避けるため, 変数 $used_i$ が使われている.

5. アルゴリズムの実装と実験

MRI アルゴリズムは Quintus Prolog Release 3.2 を使って実装されている. アルゴリズムは非決定的であるため, 可能な解を列挙するように実装した. ただし, すべての解を求めることはせず, はじめて解を得

表2 目標述語とその定義
Table 2 Target predicates and their definitions.

目標述語の定義	事例集合
memb(A, B)←dec(B,C,D),eq(A,C).	+(a, [c, a])
memb(A, B)←dec(B, C, D),	+(c, [c, b], [b])
memb(A, D).	
del(A, B, C)←dec(B,D,C),eq(A,D).	+(c, [c, d], [d])
del(A, B, [C|D])	+(a, [b, a], [b])
←dec(B, C, E),del(A, E, D).	
app(A, B, B)←nil(A).	+([ ], [ ], [ ])
app(A, B, [C|D])	+([a, b], [ ], [a, b])
←dec(A, C, E),app(E, B, D).	
last-of(A, B)←dec(A, B, C),nil(C).	+([d, e, a], a)
last-of(A, B)	+([d, b], b)
←dec(A, C, D),last-of(D, B).	−([x, y, d], x)
reverse(A, A)←nil(A).	+([3,1,2], [2,1,3])
reverse(A, [B|C])←last(A, D, B),	+([a,a,b], [b,a,a])
reverse(D, C).	−([a, y], [a, y])
merge(A, B, B) ←nil(A).	+([d], [ ], [d])
merge(A, B, [C|D])	+([d], [b], [b, d])
←dec(B, C, E),merge(A, E, D).	−([d], [e], [d,e,f])
merge(A, B, [C|D])	−([a], [ ], [ ])
←dec(A, C, E),merge(E, B, D).	
rem(A, B, B) ←nil(B).	+(c, [d], [d])
rem(A, B, [C|D])	+(1, [1, 1], [ ])
←dec(B, C, E),rem(A, E, D).	
rem(A, B, C)←dec(B, D, E),rem(A, E, C),eq(A, D).	
max(A, B)←dec(A, B, C),nil(C).	+([3, 4, 2], 4)
max(A, B)←dec(A, C, D),	+([5, 2], 5)
max(D, B),large(B, C).	−([3, 4, 5], 2)
max(A, B)←dec(A, B, C),max(C, D),large(B, D).	
ancestor(A, B)←eq(A, B).	
ancestor(A, B)←father(A, C),ancestor(C, B).	
ancestor(A, B)←mother(A, C),ancestor(C, B).	
antr-in-law(A, B)←eq(A, B).	
antr-in-law(A, B)←father(A, C),antr-in-law(C, B).	
antr-in-law(A, B)←mother(A, C),antr-in-law(C, B).	
antr-in-law(A, B)←ftr-in-law(A, C),antr-in-law(C, B).	
antr-in-law(A, B)←mtr-in-law(A, C),antr-in-law(C, B).	

事例はMRIが各プログラムを正しく帰納することのできた正事例（+ を付加）と負事例（− を付加）

ることのできた非再帰節候補に対してのみ処理を行っている．

飽和節を求めるために，述語の引数の入力/出力モードと型が与えられている．これにより，飽和節に含まれるリテラルを制限することができる．

ここまで，説明を簡略にするため，目標述語の引数はすべて与えられているとして説明してきた．しかし，帰納されたプログラムを使う段階では各引数には出力引数のものもあることを考慮しなければならない．本アルゴリズムの実装ではリスト，数などの項を出力引数としても利用できるよう考慮されている．また，本実装では出力引数は1つのみに限定されている．

表2 にMRIが帰納することのできる述語と，それを帰納するために必要とした事例を示す．背景知識の事例をあげなければ意味がないancestorとantr-in-lawについての事例は省略した．これらの定義に含まれる述語は，表1に示されている．目標述語のうち，merge, rem, max, ancestor, antr-in-lawは，2つ以上の再帰節を持っている．特に，antr-in-lawは人工的な述語ではあるが，4つの再帰節を持つ．ボトムアップ手法でこうしたプログラムを導出できるものはこれまでに知られていない．

表3 ランダムに生成された事例集合によるMRIとCRUSTACEANの比較
Table 3 Comparison of MRI and CRUSTACEAN with randomly selected examples.

	正/負	MRI			crustacean		
	事例数	C	T	N	C	T	N
memb	2 / 1	57	47	1.1	7	33	1.4
	3 / 2	93	67	1.1	47	87	1.2
	5 / 3	93	135	1.0	84	828	1.6
del	2 / 1	73	109	1.3	0	76	6.3
	3 / 2	87	74	1.0	7	170	5.9
	5 / 3	97	186	1.0	10	1758	5.0
app	2 / 1	100	248	2.8	33	83	3.8
	3 / 2	97	494	2.5	37	253	4.6
	5 / 3	100	9348	2.5	37[1]	1541	2.8
last-of	2 / 1	97	45	1.9	30	33	1.4
	3 / 2	100	60	1.6	60	74	1.0
	5 / 3	100	111	1.5	80	654	1.3
reverse	2 / 1	100	92	2.0	100	156	5.5
	3 / 2	100	145	2.0	100	425	3.2
	5 / 3	100	823	2.0	20[2]	523	2.0
merge	2 / 1	83	340	2.6			
	3 / 2	80	576	2.4			
	5 / 3	97	8658	2.2			
rem	2 / 1	90	1097	1.5			
	3 / 2	100	1687	1.3			
	5 / 3	100	1425	1.1			
max	2 / 1	43	144	1.4			
	3 / 2	67	215	1.1			
	5 / 3	77	400	1.0			

C = 30回の実験中，正しく帰納された割合（%）
T = 平均実行時間（秒）
（オーバーフローによる失敗の場合を除く）
N = 帰納されたプログラムの数の平均
[1] 失敗のうち，53%はオーバーフローによる失敗
[2] 残りの80%はすべてオーバーフローによる失敗

表3 には，ランダムに選んだ事例集合から，述語の定義を導出させる実験を各々30回行った結果を示す．事例数を，2つの正事例と1つの負事例，3つの正事例と2つの負事例，5つの正事例と3つの負事例，の3通りに変えて実験した．表に示されている正答率は，MRIが出力した複数の解の中に正しいプログラムが含まれていた割合である．結果はILPシステムCRUSTACEANと比較した．TIMと比較するべきだが，TIMは公開されていないため，代表的なボトムアップシステムであり，公開されているCRUSTACEAN[7),8)]との比較

を行った．CRUSTACEANは，ランダムに選ばれた少数事例からプログラムを帰納する能力において，優れていると評価されてきているシステムであり，MRIの正答率をこれと比較することは価値がある．MRIは正しいプログラムを導出する割合において，多くの場合CRUSTACEANを上回った結果を得た．また，CRUSTACEANが，正事例数2，3の場合低い正答率にとどまっているのに対して，MRIは高い率で帰納に成功した（MRIはCRUSTACEANに比べてreverse以外の述語で，各々40%以上高い割合で成功した）．導出に要した時間は大差がない．

これらのシステムは非決定的であるため，一般に複数の解を導く．MRIでは負事例を使って極力誤った節を排除している．表3にはいくつの解が平均して得られたかも示した．正しいプログラムを含む，少ない候補プログラム集合を出力するアルゴリズムが，優秀なものであると考えられるが，この点でも，MRIはCRUSTACEANと同等以上の能力を持つことが示された．

また，MRIではCRUSTACEANが導出することのできない2つ以上の再帰節を持ったプログラムについても，1つの再帰節の場合と同程度の正答率，実行時間，候補数を得られることが示された．

6. おわりに

本論文では，事例の飽和節を構造的に分析することで帰納的に論理プログラムを導出する新しいアルゴリズムを提案した．経路構造に基づいて飽和節の構造を解析する手法は文献9)がはじめである．本論文の新たな提案は，経路構造の差分の概念を導入し，これが再帰節に対応することを示し，さらにその再帰節に含まれるべきリテラルを対応づけたことにある．これによって，複数の再帰節を含む論理プログラムを帰納することができるようになった．このクラスの論理プログラムを分析的なボトムアップ手法で導出することができるようになったのは，これがはじめてである．

ここでMRIの能力について検討する．MRIはちょうど1つの非再帰節と，いくつかの再帰節からなるプログラムを帰納することができる．このとき再帰節は1つの再帰リテラルからなり，そのリテラルの第i入力引数は頭部リテラルの第i引数のみに依存して値が決まらなければならない．背景知識に含まれるすべての述語は，判定述語か，あるいは経路述語でなければならない．これらが満たされるとき，MRIが正しいプログラムを解の1つとして導出することができる事例の集合が存在する．実際，表2に示したとおり，目標となるプログラムのすべての節を用いる事例集合が与えられることで，プログラムを帰納することができる．

最後にMRIの計算複雑さに関する結果を述べる．本システムはTIM[9]と同様な枠組みを使っており，文献9)に述べられている結果のいくつかを用いることができる．この結果によると，飽和節に含まれるリテラルの数は$O((kjm)^{j^i})$でおさえられることが分かっている．また，可能な経路構造の数はたかだか$O((kjm)^{j^i ik})$である．ただし，kは正事例のアリティ，jは背景知識に含まれる述語のアリティの最大値，mは背景知識に含まれる述語の数，そしてiは可能な経路の最長の長さである．

ここで飽和節内のリテラルの数をs，経路構造の数をpとする．またnをアルゴリズムで用いられた正事例の数とする．アルゴリズム中，最も時間複雑さの大きな箇所は非再帰節の条件部に含まれるリテラルをすべて求める箇所（アルゴリズムの5～11行）と，再帰節の候補を集める箇所（21～32行）であり，ともに経路構造の間のマッチングをたかだかsn回行っている．また，1回のマッチングには$O(i)$だけかかるので，これらの箇所の時間複雑さは$O(ins) = O(in(kjm)^{j^i})$である．これがMRIの1回の動作の時間複雑さを決めており，MRIはこれをたかだかp回繰り返す．したがって，アルゴリズム全体でかかる時間コストは$O(in(kjm)^{j^i} \cdot p) = O(in(kjm)^{j^i} \cdot (kjm)^{j^i ik}) = O(in(kjm)^{j^i(1+ik)})$である．もし$i$，$j$，$k$を定数と仮定するならば$m$と$n$に関する多項式時間以内にアルゴリズムは停止する．

TIMアルゴリズムでは，MRIと同様に経路構造を順に調べる．各経路構造に対して，事例数n分の経路が共通した繰返し構造を持つことをマッチングにより調べるため，MRIよりもわずかに少ない$O(inp) = O(in(kjm)^{j^i ik})$の時間コストが必要となる．

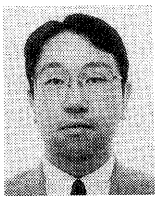
MRIアルゴリズムが行った方法は，実行時トレースを推測することで，そのトレースを実行できるプログラムを帰納するものである．したがって，本システムの帰納できる論理プログラムのクラスは，拡張できる可能性がある．たとえば，非再帰節が1つであることを仮定することにより，可能な経路構造を限定しているが，効率を落とさずにこの限定を外すことは今後の課題である．

参考文献

1) Quinlan, J.R.: Learning logical definitions

from relations, *Machine Learning*, Vol.5, pp. 239–266 (1990).
2) Quinlan, J.R. and Cameron-Jones, R.M.: FOIL: A midterm report, *Proc. 6th Europcan Conf. on Machine Learning*, Lecture Notes in Artificial Intelligence, Vol.667, pp.3–20. Springer-Verlag (1993).
3) Inuzuka, N., Kamo, M., Ishii, N., Seki, H. and Itoh, H.: Top-down Induction of Logic Programs from Incomplete Samples, *Proc. 6th Int'l Inductive Logic Programming Workshop*, pp.119–136 (1996).
4) Muggleton, S.: Inverse entailment and progol, *New Generation Computing*, Vol.3+4, pp.245–286 (1995).
5) Muggleton, S. and Feng, C.: Efficient induction of logic programs, *Proc. 1st Int'l Workshop on Algorithmic Learning Theory*, pp.368–381, Ohmsha, Tokyo (1990).
6) Lapointe, S. and Matwin, S.: A tool for efficient induction of recursive programs, *Proc. 9th Int'l Conf. on Machine Learning*, pp.273–281, Morgan Kaufmann (1992).
7) Aha, D.W., Lapointe, S., Ling, C.X. and Matwin, S.: Inverting Implication with Small Training Sets, *Proc. 7th European Conf. on Machine Learning* (*ECML '94*), pp.31–48, Springer-Verlag (1994).
8) Aha, D.W., Lapointe, S., Ling, C.X. and Matwin, S.: Learning Recursive Relations with Randomly Selected Small Training Sets, *Proc. 11th Int'l Conf. on Machine Learning*, pp.12–18, Morgan Kaufmann (1994).
9) Idestam-Almquist, P.: Efficient Induction of Recursive Definitions by Structural Analysis of Saturations, *Advances in Inductive Logic Programming*, De Raedt, L. (Ed.), pp.192–205 (1996).
10) Cohen, W.W.: PAC-learning a restricted class of recursive logic programs, *Proc. 3rd Int'l Workshop on Inductive Logic Programming*, J. Stephen Institute, Ljubljana, Slovenia (1993).

(平成 9 年 6 月 26 日受付)
(平成 10 年 10 月 2 日採録)



犬塚 信博（正会員）

1987 年名古屋工業大学工学部情報工学科卒業．1992 年同大学院工学研究科博士課程電気情報工学専攻修了．博士（工学）．同年，同大学電気情報工学科助手．1993 年同大学知能情報システム学科助手．1998 年同学科講師．現在に至る．1994～1996 年日本学術振興会海外特別研究員として英国インペリアルカレッジへ出張．人工知能，特に機械学習，知識表現に関する研究に従事．電子情報通信学会，人工知能学会，AAAI 各会員．

古澤 光枝

1997 年名古屋工業大学知能情報システム学科卒業．同年三菱電機エンジニアリング（株）入社．現在同社産業システム事業部において，データベースシステムの開発に従事．在学中帰納論理プログラミングアルゴリズムの研究に従事．

世木 博久（正会員）

1979 年東京大学工学部計数工学科卒業．1981 年同大学院工学系研究科修士課程修了．同年 4 月より三菱電機（株）中央研究所に勤務．1985～1989 年（財）新世代コンピュータ技術開発機構に出向．1992 年 4 月より名古屋工業大学工学部知能情報システム学科助教授．1997 年 4 月同学科教授．工学博士．論理プログラミング，演繹データベース等に興味を持つ．電子情報通信学会，人工知能学会，ACM，IEEE Computer Society 各会員．

伊藤 英則（正会員）

1974 年名古屋大学大学院工学研究科博士課程電気・電子専攻満了．工学博士号取得．同年日本電信電話公社入社，横須賀研究所勤務．1985 年（財）新世代コンピュータ技術開発機構に出向．1989 年より名古屋工業大学教授．知能情報システム学科．これまでに数理言語理論とオートマトン，計算機ネットワーク通信 OS，知識ベースシステム，人工知能等の研究と開発に従事．電子情報通信学会，日本ファジィ学会，人工知能学会各会員．