知識工学と人工知能　45－2
（1986・3・10）

対象分野の基本知識に基づく例題学習

沼尾正行　　志村正道
（東京工業大学工学部）

1．まえがき

人工知能における学習の研究には、学習すべき情報の与え方によって、助言に基づく学習や例題による学習などがある[MIC 83][BAR 82]。このうち例題による学習には、論理式やパターンなどで表わされた概念を学習するものと、プロダクションルールなどのルールを学習するものが考えられる。ここでは、例題に基づいてルールを学習する方法について考察する。

従来のルール学習システムの多くは、ルールの条件部を洗練するものである。すなわち、ルールの実行部をあらかじめ用意しておき、実行部を起動するのに適した条件部を概念学習の手法を使って洗練または生成する[BUN 85]。この方法では、推論の手順を意識した実行部をあらかじめ与えなければならない。

本論文では、ルールの両辺を同時に合成する方法を提案する。提案する方法では、最初に基本知識としてルールの構成要素となる基本ルールが与えられる。システムはまず、例題の問題と答えを合致させる基本ルールの組み合わせを探索によって見つけ出し、それをもとにして基本ルールからルールを組立てる。この学習法は、書き換えルールによってプログラムの翻訳を行なうシステム[NUM 85][MIY 86]のために考案されたものであるが、書き換えルールを使用してパーザを記述したり、VLSIの回路を生成することもできるので、広範囲に応用することが可能である。

2．基本ルールの結合に基づくルールの合成

ルールは、基本ルールを組み合わせることによって合成される。ここでは、そのための基本操作であるルールの結合について述べる。

2.1 ルールの形式

ルールの形式を図1に示す。ルールは項書き換えシステム[FUT 83]における等式にならったものとなっており、左辺(LHS)と右辺(RHS)が等しいことを表わしている。すなわち、〈オペレータ〉が ⇒ のときには左辺のパターンとマッチする表現を右辺のパターンに書き換えることができ、〈オペレータ〉が ⇐ のときには、右辺のパターンを左辺のパターンに書き換えることができることを表わす。また、＝のときには、両方向に書き換えることが可能なことを表わす。たとえば、次のルールは、それぞれ加算における結合法則と交換法則を示している。

```
〈ルール〉 :=
  (rule  〈rule名〉
         〈左辺〉 〈オペレータ〉 〈右辺〉
       { restrict: 〈マッチ条件リスト〉 } )
〈左辺〉 := 〈パターン〉
〈オペレータ〉 :=  ＝ | ⇒ | ⇐
〈右辺〉 := 〈パターン〉
〈マッチ条件リスト〉 := ( 〈マッチ条件〉 ･･･ )
〈マッチ条件〉 := (〈パターン〉 ･･･ 〈Lisp関数〉)
```

図1　プロダクションルールの形式

```
(rule assoc (*x + (*y + *z)) = ((*x + *y) + *z) )
(rule comm (*x + *y) = (*y + *x) )
```

2.2 ルールの結合

ルールは基本ルールを組み合わせることによって生成される。このための基本的な操作は次に述べるルールの結合である。

ルールの結合は一つのルールの片方の辺と他方のルールの対応する辺を単一化し、それらのルールの単一化しなかった方の辺を新しいルールの両辺とすることで行なわれる。すなわち、

```
(rule a  X = Y  restrict: Ra )
```

の右辺 Y の一部分 Ys に

```
(rule b  Z = U  restrict: Rb )
```

の左辺 Z を結合するとする。まず結合の準備として、ルール b 中の変数のうちルール a に含まれる変数と名前が重複しているものをユニークな名前に置き換えて、ルール b′ を得る。

```
(rule b′  Z′ = U′  restrict: Rb′ )
```

次にルール a の右辺 Y の一部分 Ys とルール b′ の左辺 Z′ の単一化を行なう。その際の変数の代入を θ とすれば、結合されたルール c は次のようになる。

```
(rule c  Xθ = Y[ U′θ/Ys ]  restrict: (RaRb′)θ )      ··· (*)
```

ただし、$X\theta$, $U'\theta$ は、それぞれ X, U′ の変数に代入θを行なったものであり、Y[U′θ/Ys] は Y においてその一部分 Ys を U′θで置き換えたもの、RaRb′ は Ra と Rb′ の要素を並べたものである。

例えば、前述のルール assoc にルール comm を結合する場合を考える。まず、ルール comm 中の変数 *x および *y は、ルール assoc 中の変数と重複しているので、それぞれ *x1, *y1 と置き換えたルール comm′ を得る。

```
(rule comm′  (*x1 + *y1) = (*y1 + * x1) ).
```

ルール assoc と comm′ から、式 (*) の各部分は次のようになる。

```
X = ( *x + (*y + *z) )
Y = ( ( *x + *y ) + *z )
Ys = ( *x + *y )
Z′ = ( *x1 + *y1 )
U′ = ( *y1 + *x1 )
θ = [ *y/*y1, *x/*x1 ].
```

ゆえに、

```
(rule assoccomm (*x + (*y + *z)) = ((*y + *x) + *z) )
```

2.3 基本ルールの役割

ルールを結合して新しいルールを生成した場合、生成されたルールはもとのルールに比べて特殊化される。すなわち、ルールの適用可能な表現のクラス、およびルールから生成可能な表現のクラスは共に小さくなる。したがって、基本ルールが正しい書き換えを行なうルールである場合には、ルールの結合は書き換え速度の向上をもたらすだけで、学習としての意味をもたない。しかし、基本ルールをルールの単なる構成要素ととらえれば、ルールの結合の結果、目的とするルールが得られることになる。

プログラム翻訳においては、基本ルールとして、プログラムの意味を保存するルールを用いることができる[NUM 85]。すなわち、原言語および目的言語の表示的意味を基本ルールとして与えれば、翻訳においてプログラムの意味を変えないことが保証される。その上で、プログラミングスタイルやプログラムの効率を考えて、ルールの結合により翻訳ルールを基本ルールから組立てればよい。

ＶＬＳＩの設計においては、基本ルールとして、

1) 左辺に設計の単位となる各ゲートを与え、右辺にそのゲートの論理表現を与えたルール、および、
2) 論理表現の簡約化ルール、すなわち、ド・モルガンの定理や二重否定の除去ルール、

を与える。これらのルールを組み合わせれば、論理的な仕様から回路表現を生成するルールを組立てることができる。これを基本ルールの結合ではなく、検証結果に基づく拘束条件の伝搬によって行なったのが、検証に基づく学習(Verification-Based Learning)[MIT 85][MAH 85]である。

基本ルール集合として、あらゆるルールが合成可能なものを選べば、学習によってあらゆるルールを生成することもできる。すなわち、次のルールを基本ルールとする。

```
(rule a1  (*x . *t) = *t )
(rule a2  (*x *y . *t) = (*y *x . *t) )
(rule a3  ( (*x . *t1) . *t2) = (*x *t1 . *t2) )
(rule ai  Atom = nil )
```

ただし、ルール ai はルールに出現しうるすべてのアトム(Atom)について作成しておく必要がある。前述のルール comm はルール a1,a2,a3,ai を用いて次のように組立てられる。

まず、アトム '+' および 'nil' についてルール ai を生成しておく。

```
(rule +   + = nil )
(rule nil  nil = nil)
```

ルール nil とルール a2 より、

```
(rule nila2  (*x *y) = (*y *x) )
```

ルール '+' とルール a1 より

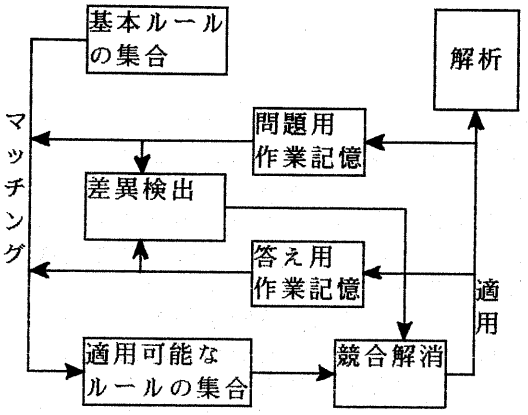


図２　Ｎａｔシステムの構成

(rule ＋a1　(+ . *t) = *t)

ルール nila2 の左辺の cdr である'(*y)' にルール ＋a1 の右辺を結合すれば、

(rule c1　(*x + *y) = (*y *x))

さらに右辺の cdr である '(*x)' にルール ＋a1 の右辺を結合すれば、ルール comm が得られる。

(rule comm　(*x + *y) = (*y + *x))

このように、上述の基本ルールを使用すれば、ルールの結合によって任意のルールが合成可能であるが、そのような基本ルールを用いると学習の効率が低くなってしまう。基本ルールを小さなものとすればきめ細かにルールを作りだすことができるし、逆に大きな基本ルールを用意すれば学習の効率を上げることができるので、対象分野の基本知識をうまく反映した基本ルールを用意することが重要となる。

３．例題からの学習

基本ルールを組合わせることによりルールが合成されるが、その組合わせ方は与えられた例題が正しく解けるように決定する。ここではまず、与えられた例題を解く基本ルールの組合せを探索によって見つけ出す方法を述べ、次にそれに基づいて合成すべきルールを決定する方法を考察する。

3.1 基本ルールによる探索

例題を解く基本ルールの組合せを求めるためには、例題として問題と答えを与え、基本ルールを用いたプロダクションシステムによって探索を行なう。このため、プロダクションシステムインタプリタ Ｎａｔ(Nonalgorithmic translator)を作成した。その構成を図２に示す。

プロダクションシステムの作業記憶には、問題と答え用のものがそれぞれあり、一種の双方向探索法[SIM 83][POH 71]により、問題側と答え側から双方向に探索を行なう。双方向に探索を行なうのは、ルールの適用方向によって探索範囲に大きな差が出てくる場合があるためである。例えば、ラムダカリキュラスのβ変換では、逆方向の変換は無数の変換が存在するため、ほとんど不可能である。差異検出部は、作業記憶が書き換えられるたびに、問題用作業記憶の内容と答え用作業記憶の内容を比較し、その差異を検出する。また、適用可能なルールの集合が、書き換えられた内容にしたがって更新される。競合解消部は、差異検出部で検出された差異を解消するのに効果的なルールを適用可能なルールの集合から選び、選ばれたルールが作業記憶に適用されて書き換えを行なう。

このようにして行なわれた探索の結果、問題から答えに到達する経路がルールの適用グラフとして得られる。その構文を図３に示す。探索結果がグラフになるのは、ルールが作業記憶中の任意の部分表現に適用でき、作業記憶中の各部分表現に独立にルールが適用されうるからである。適用グラフは、問題ノード problem 、答えノー

```
<適用グラフ> ::= (history <ノード> ... ).
<ノード> ::= (problem (nil (<適用名> <アドレス>) ... ))
           | (answer (nil (<適用名> <アドレス>) ... ))
           | (<適用名> (<左辺/右辺> (<適用名> <アドレス>) ... ) ... ).
<左辺/右辺> ::= L | R | neg-L | neg-R .
<適用名> ::= <ルール名>#<番号> .
<アドレス> ::= ( <符号> <セレクタ> ... ).
<符号> ::= + | -
<セレクタ> ::= car | cdr .
```

図３　適用グラフの構文

ド answer と適用ノードのリストとなっており、適用ノードは <ルール名>#<番号> の形式の適用名で識別される。各ノードには適用されたルールの左辺および右辺に隣接しているノードがアドレスと共に記述されている。アドレスの先頭の符号が＋の場合には、左辺または右辺の部分表現に対して隣接した適用が結合していることを示す。－の場合には、隣接した適用の部分表現に対してその適用が結合していることを示す。ルールは左辺と右辺が対称になっているので、適用グラフも適用方向に対して対称なものとなる。前述のルール comm とルール assoc を用いて、例題

```
問題： (a + (b + c)),
答え： ( (b + a) + c),
```

を解いた結果は次のように出力される。

```
(history
   (answer (nil (comm#5 (+ car))))
   (problem (nil (assoc#2 (+))))
   (assoc#2 (L (problem (-))) (R (comm#5 (+ car))))
   (comm#5 (L (answer (- car))) (R (assoc#2 (- car)))))
```

problem は問題 (a + (b + c)) を表わし、これにはルール assoc の左辺がマッチして適用 assoc#2 により書き換えが行なわれる。その結果の car にはルール comm の右辺がマッチして適用 comm#5 により answer すなわち答え ((b + a) + c) が生成されている。

合成すべきルールはこの適用グラフを部分グラフに分割したとき、各部分グラフについてルールの結合を行なったものであると考えられる。例えば、上の適用グラフの部分グラフ

```
((assoc#2 (R (comm#5 (+ car))))
 (comm#5 (R (assoc#2 (- car)))) )
```

についてルールの結合を行なうと、前述のルール assoccomm が得られる。このように、合成すべきルールを決定する問題は、適用グラフから部分グラフを選択する問題に帰着されるので、その方法を検討する。

3.2 ルール集合の半順序関係を用いた合成

ルール集合 S1, S2 について S1 が S2 の一般化であるというのは、任意の作業記憶内容 σ と σ' について、

(σ S2*⇒ σ') ならば (σ S1*⇒ σ')

が成立することである。ただし、σ S*⇒ σ' はルール集合 S 中のルールを繰り返し適用することにより、作業記憶内容 σ が σ' に書き換えられることを表す。S1 が S2 の一般化であることを半順序 S1 ≺ S2 で表す。また、S2 を S1 の特殊化という。

ルールの結合に関して次の関係が成立する。

ルール r1 が、ルール r2 と r3 を結合して得られるとき、
{r2, r3} ≺ {r1}. ・・・(α)
S1 ≺ S2 かつ S3 ≺ S4 ならば、 S1 ∪ S3 ≺ S2 ∪ S4. ・・・(β)
S1 ≽ S2 ならば、 S1 ≺ S2. ・・・(γ)

すなわち、ルールの結合によって生成されたルール集合は、もとのルール集合に比べて特殊化される。このことを用いるとルールの結合に基づいて、合成すべきルールの集合を正しい例と誤った例によって決定してゆくことができる。これを次のようなルールによる簡単な翻訳の学習を例として述べる。

```
(rule elim  (*x . *t) = *t )
(rule ex  (*x *y . *t) ⇒ (*y *x . *t) )
(rule I  (I . *t) p=a (私は . *t) )
(rule fire (fire . *t) p=a (火は . *t) )
(rule you (you . *t) p=a (あなたが . *t) )
(rule burns (burns . *t) p=a (燃える . *t) )
(rule like (like . *t) p=a (好きだ . *t) )
(rule period (.) p=a (。) )
```

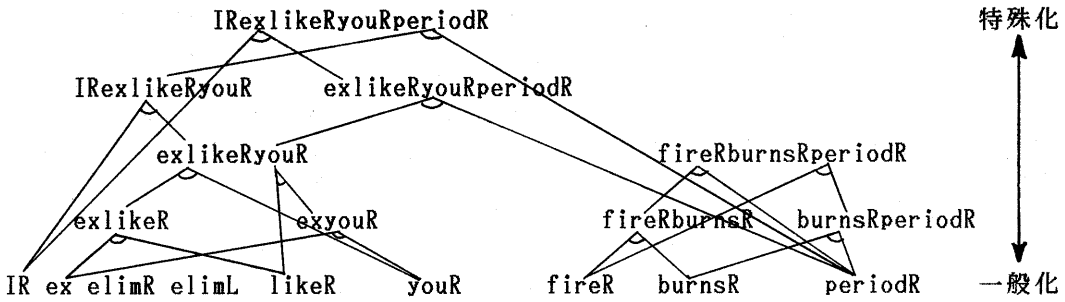
ただし、オペレータ p=a は左辺が問題用作業記憶とのみマッチし、右辺が答え用作業記憶とのみマッチすることを示す。これらのルールは、単語の対応する単語への置換、順序の変更、削除および追加により翻訳がおこなわれることを定義している。これらの基本知識と例題により、正しい翻訳ルールが以下のように学習される。

まず、次の例題により、基本ルールの適用グラフを得る。

例題1　問題：(fire burns .)
　　答え：(火は 燃える 。)

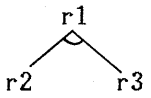
例題2　問題：(I like you .)
　　答え：(私は あなたが 好きだ 。)

それらの適用グラフから、合成しうるルールを図4に示す。ここで、



(rule IRexlikeRyouRperiodR (I like you .) ⇒ (私は あなたが 好きだ 。))
(rule IRexlikeRyouR (I like you . *t) ⇒ (私は あなたが 好きだ . *t))
(rule exlikeRyouRperiodR (like you .) ⇒ (あなたが 好きだ 。))
(rule fireRburnsRperiodR (fire burns .) ⇒ (火は 燃える 。))
(rule exlikeR (like *y . *t) ⇒ (*y 好きだ . *t))
(rule exyouR (*x you . *t) ⇒ (あなたが *x . *t))
(rule fireRburnsR (fire burns . *t) ⇒ (火は 燃える . *t))
(rule burnsRperiodR (burns .) ⇒ (燃える 。))
(rule IR (I . *t) ⇒ (私は . *t)) (rule ex (*x *y . *t) ⇒ (*y *x . *t))
(rule elimR (*x . *t) ⇒ *t) (rule elimL *t ⇒ (*x . *t))
(rule likeR (like . *t) ⇒ (好きだ . *t))
(rule youR (you . *t) ⇒ (あなたが . *t))
(rule fireR (fire . *t) ⇒ (火は . *t))
(rule burnsR (burns . *t) ⇒ (燃える . *t)) (rule periodR (.) ⇒ (。))

図４　ルール集合の半順序関係



は、 r2 と r3 を結合することによって r1 が得られることを示しており、図４と(α),(β),(γ)を用いれば、ルール集合の半順序関係を決定できる。ルール集合において、最も一般化されたルール集合は、合成されるルールに方向性をもたせるため、右向きと左向きのルールを区別した基本ルールの集合とする。最も特殊化されたルール集合は、例題の問題を左辺に答えを右辺に持つ右向きルールの集合とする。

半順序関係に基づいて、適切なルール集合を決定するためには、正しい例とともに誤った例を与える必要がある。誤った例を与えるためには、最も一般化されたルール集合を選び、例題の問題を解かせてみる。このとき、誤った答えが得られれば、それが誤った例となる。そこで得られた適用グラフの空でない部分グラフのうちで、正しい例により得られた適用グラフ中に現れず、かつなるべく小さなものを選ぶ。さらに、選んだグラフから合成しうるルールをルール集合の特殊化によって消去する。これらの操作を繰り返せば、誤った答えを生成せず、正しい答えだけを生成するルール集合を得ることができる。

誤った例を与える方法で得られたルール集合を用いると、探索の結果例題が正しく解けることが保証される。さらに、探索なしに問題を解くようなルール集合を得るた

めには、答えを求めるまでに生じた誤ったルール適用を正しい答えが得られた場合についても収集しなければならない。このためＮａｔには、無駄になったルールの適用をすべて追跡して適用グラフに追加する機能がある。この誤った適用グラフは図３において、<左辺/右辺> をneg-L または neg-R とすることにより表現される。

前述の例における誤った適用は、ルール ex, elimR, elimL の適用である。詳細は省略するが、ルール集合を特殊化することにより、これらのルールを消去すれば、例題を探索なしに解くことができるルール集合

{ IR, exlikeR, youR, fireR, burnsR, periodR }

が得られる。

４．あとがき

対象領域の基本知識と例題を与えることによって、ルールの両辺を同時に合成する方法を述べた。この方法では、対象領域の基本知識を基本ルールによって柔軟に与えることができる。そのため、学習の結果得られたルールを次の学習の基本ルールとして使用する多段階の学習を行なうことが有効となるであろう。

参考文献

[BAR 82] Cohen, P.R. and Feigenbaum, E.A.: The Handbook of Artificial Intelligence, Volume3, Pitman, pp.323-511(1982).

[BUN 85] Bundy, A., Silver, B. and Plummer, D.: An Analytical Comparison of Some Rule-Learning Programs, Artificial Intelligence, Vol.27, pp.137-181 (1985).

[FUT 83] 二木厚吉，外山芳人：項書き換え型計算モデルとその応用，情報処理，Vol.24, No.2, pp.133-146(1983).

[MAH 85] Mahadevan,S.: Verification-based Learning: A Generalization Strategy for Inferring Problem-Reduction Methods, IJCAI85, pp.616-623 (1985).

[MIC 83] Michalski, R.S., Carbonell, J.G. and Mitchell, T.M.: MACHINE LEARNING, Tioga(1983).

[MIT 85] Michell, T.M., Mahadevan, S. and Steinberg, L.I.: LEAP: A Learning Apprentice for VLSI Design, IJCAI85, pp.573-580(1985).

[MIY 86] 宮田俊介：ALGOL系言語からFORTRANへのプログラム翻訳，東京工業大学情報工学科修士論文(1986).

[NUM 85] 沼尾正行，志村正道：学習機能をもつプログラム翻訳システム，情報処理学会研究報告 85-AI-40(1985).

[POH 71] Pohl,I.: Bi-directional search, Machine Intelligence 6, pp.127-140 (1971).

[SIM 83] 志村正道：機械知能論，昭晃堂，pp.190-192(1983).