人 口 知 能 84－8
（1992. 9. 10）

一般化論理プログラムのunfoldingによる変換ルールの合成

馬淵 浩司　赤間 清　宮本 衛市

北海道大学　工学部

本論文では、一般化論理プログラムのunfoldingを用いて変換ルールを合成する手法について述べる。unfoldingとは、Horn節のHeadをBodyに展開する方法である。GLPのunfoldingの理論は、より広い範囲のプログラムを扱うために、Prologのunfoldingの理論を改善し拡張したものである。GLPのunfoldingによる合成では、unfoldingによって、もとの2つのHorn節から複数の生成節を得ることがある。本論文では、複数の生成節の中から、与えられた具体例に対応した適切な生成節を選び出す手法についても述べる。また、GLPのunfoldingによる合成の有用性を明らかにするために、Prologのunfoldingによる合成と比較する。

Synthesis of Conversion-Rules by unfolding of generalized logic program

Hiroshi Mabuchi　Kiyoshi Akama　Eiichi Miyamoto

Dept. of Information Eng., Faculty of Eng., Hokkaido Univ.

Kita 13,Nishi 8,Kita-ku,Sapporo,060,Japan

In this paper,we describe a method for synthesizing Conversion-Rules by unfolding of generalized logic program(GLP). Unfolding is a method which unfolds Head of Horn clause to Body. Unfolding of GLP is a method that improves and extends ones of Prolog for dealing with programs of more extensive field. Synthesis by unfolding of GLP is possible to get plural generated clauses from original 2 Horn clauses. In addition, we describe a method for selecting appropriate generated clauses which correspond to given example from plural generated clauses. And to make clear availability of synthesis by unfolding of GLP, we compare with ones by unfolding of Prolog.

1 まえがき

変換ルールは、ある状態をより簡単化された別の状態へ変換するものである。複数の入力変換ルールから、新しい合成ルールを求める合成という技術は非常に重要である [1]。

本論文では、一般化論理プログラム (generalized logic program : GLP)[3] の unfolding を用いて変換ルールを合成する手法について述べる。unfolding とは、Horn 節の Head を Body に展開する方法である。GLP の unfolding の理論は、より広い範囲のプログラムを扱うために、Prolog の unfolding の理論を改善し拡張したものである。

GLP は、集合オブジェクトを含むさなざまなデータ構造の表現を可能にする。GLP の unfolding による合成では、unfolding によって、もとの 2 つの Horn 節から複数の生成節を得ることがある。本論文では、複数の生成節の中から、与えられた具体例に対応した適切な生成節を選び出す手法についても述べる。また、GLP の unfolding による合成の有用性を明らかにするために、Prolog の unfolding による合成と比較する。

2 変換ルールの合成問題

2.1 変換ルールの合成の問題の定義

本節では、変換ルールの合成の問題を定義する。変換ルールを節で表現するために述語 next を考える。述語 next は、

next (変換ルール名 p , 状態 s , 状態 t)

の形をとり、「状態 s から状態 t へ、オペレータ p によって到達可能である」を意味するものとする。r1,r2 という 2 つの変換ルールが next 節で次のように定義されているものとする。ここで、T_A から T_F は状態を表すパターンである。これ以降、プログラムに付いている N_1 から N_3 と C_1,C_2 ･･･は、節の番号を示す。

next (r1 , T_A , T_B)←body$_1$. ...(N_1)

next (r2 , T_C , T_D)←body$_2$. ...(N_2)

これから

next (r3 , T_E , T_F)←body$_3$. ...(N_3)

で示される r1 と r2 を合成した変換ルール r3 を求めることが変換ルールの合成問題である。

これを 8 パズルの例で説明する。ここで、状態を表現するために、各位置に対して番号を次のように割り当てる。

1	2	3
4	5	6
7	8	9

状態は、例えば、次のようなリストで表す。

(4 b 7 6 1 8 2 3 5)

ここで、b はブランクを示す。これは、1 番の位置は 4 、2 番の位置はブランク、3 番の位置は 7 、･･･ である状態を示し、次のような配置になる。

4	b	7
6	1	8
2	3	5

UP は、1 を上に移動する変換である。

next (UP,
(4 b 7 6 1 8 2 3 5),
(4 1 7 6 b 8 2 3 5))←.

RIGHT は、6 を右に移動する変換である。

next (RIGHT,
(4 1 7 6 b 8 2 3 5),
(4 1 7 b 6 8 2 3 5))←.

UP と RIGHT の合成ルール UpRight は次のようになる。

next (UpRight,
(4 b 7 6 1 8 2 3 5),
(4 1 7 b 6 8 2 3 5))←.

2.2 変換ルールと変換プログラム

本節では、前節の問題を論理の言葉で定式化する。前節の next を意味づけるために次に示す path を導入する。

```
C1: path([Rule | Path],A,C) ←
        next(Rule,A,B),
        path(Path,B,C).

C2: path([],A,A) ←.
```

path は、path (変換ルール列 q , 状態 s , 状態 t) の形をしており、「状態 s から状態 t へ、オペレータ列 q によって到達可能である」を意味する。このとき、C_1,C_2に前節の合成する 2 つの next 節 (N_1,N_2) を加えたプログラムを P とする。このとき、P は次のような節の集合である。

$$P = \{N_1, N_2, C_1, C_2\}$$

そして、P の論理的帰結である

$$path([r1, r2], T_E, T_F) \leftarrow body.$$

の形の節を考える。この述語 path を next に置き換え、第 1 引数 $[r_1, r_2]$ を合成ルール r_3で置き換えて

$$next(r3, T_E, T_F) \leftarrow body.$$

を得る。このようにして得られるのが合成ルールである。

2.3 GLP による回路変換プログラム

本節では、論理回路を変換する 2 つの変換ルール (andAnd 変換ルールと noConnection 変換ルール) の例を挙げる。これらの変換ルールは、GLP によるものであり、後に合成に用いる。

GLP による andAnd 変換ルールを次に示す。

```
C3: next(andAnd,
     {[and,A,R1]
      [and,B,{A | R2}] | RR},
     {[and,A,R1][and,B,R3] | RR}) ←
        union(R1,R2,R3).
```

next は 3 つの引数を持っている。第 1 引数はルール名を示し、第 2 引数は変換前の回路を示し、第 3 引数は変換後の回路を示す。回路は、GLP のみが可能な集合オブジェクトの形で書かれている。第 2 引数は、2 つの and 素子と残りの回路である。最初の and 素子は、入力が R1 で出力が A である。2 つめの and 素子は、入力が [A | R2] で出力が B である。このプログラムの Body は、union のみである。union は、

$$union(X,Y,Z) \cdots X \cup Y = Z$$

を意味し、次のようなプログラム (C_4,C_5) である。

```
C4: union({X | U},Y,{X | V}) ←
        union(U,Y,V).

C5: union({},Y,Y) ←.
```

次に、GLP による noConnection 変換ルールのプログラムを示す。この変換ルールは、他につながっていない中間端子を持つ素子は、論理回路から削除できるというルールである。

```
C6: next(noConnection,
        {[ELEMENT,AA,P] | RR},
          RR)  ←
        notExist(AA,RR),
        mid(AA).
```

mid (AA) は、AA が中間端子であることを意味する。notExist (AA , RR) は、端子 AA が回路 RR の中で入力端子として使われていないことを意味する。notExist の定義節を C_7、mid の定義節を C_8とする。

3 GLP の unfolding による合成

変換ルールを合成するために GLP の unfolding を用いる。本章では、まず、GLP の unfolding を定義し、それに基づいて、GLP の unfolding によるルール合成と GLP の unfolding による例に対応したルール合成を行なう。

3.1 GLP の unfolding の定義

unfolding とは、プログラムの等価変換の中で最も重要な変換の 1 つであり、節の Head を Body に展開する方法である。これは一般化論理プログラム $P_1 = \{C_j \mid j \in J\}$ から節 C_kとその Body アトム A_iを選んで、unifier の集合 U_j(j ∈ J) を用いて、unfolding して P_2を得るというものである。これは、次のように定義されている [5] 。

1. $P_1 = \{C_j \mid j \in J\}$ から節 $C_k = (H \leftarrow A_1, \cdots, A_i, \cdots, A_a)$ を選ぶ。
2. 節 C_kの Body から A_iを選ぶ。
3. 一般化論理プログラム P_1の各節 $C_j = (K \leftarrow B_1, \cdots, B_b)$ (j ∈ J) に対して、$(A_i;C_k)$ と $(K;C_j)$ の健全かつ完全な unifier の集合の任意の 1 つを U_j (j ∈ J) とする。ここで、K は C_jの Head である。

4. U_j (j ∈ J) を用いて、

$R_j = \{(\mathrm{H}\ \theta \leftarrow A_1\ \theta, \cdots, A_{i-1}\ \theta,$
$B_1\ \sigma, \cdots, B_b\ \sigma,$
$A_{i+1}\ \theta, \cdots, A_a\ \theta) \mid (\theta,\sigma) \in U_j\}$

$\mathrm{R} = \cup_{j \in J}\ R_j$

$P_2 = P_1 - \{C_k\} \cup \mathrm{R}$

として P_2を得る。

3.2 GLP の unfolding によるルール合成

本節では、2 つの変換ルールを unfolding によって合成する方法を提案する。今、r1,r2 という 2 つの変換ルールが next 節で次のように定義されているものとする。

next (r1 , T_A , T_B)←body$_1$.
next (r2 , T_C , T_D)←body$_2$.

これから

next (r3 , T_E , T_F)←body$_3$.

で示される $r1$ と $r2$ を合成したルール $r3$ を求めることが合成である。

GLP の unfolding による合成法を提案する。まず、次のようなトートロジーを考える。

```
path([r1,r2],A,C) ←
    path([r1,r2],A,C).
```

この節を C_1による unfolding によって展開すると次のようになる。

```
path([r1,r2],A,C) ←
    next(r1,A,B),
    path([r2],B,C).
```

そして、Body の path を C_1によって展開すると、次のようになる。

```
path([r1,r2],A,C) ←
    next(r1,A,B),
    next(r2,B,C),
    path([],C,C).
```

さらに、Body の path を C_2によって展開すると、次のような節 C_Uを得る。

```
Cu:  path([r1,r2],A,C) ←
         next(r1,A,B),
         next(r2,B,C).
```

そして、C_Uの Body の 2 つの next を C_3と C_6によって展開して、Body には next が入らない次のような形の節を得る。

path ([r_1 , r_2] , T_A , T_C) ← body.

ここで、A は T_A、C は T_Cになる。

path を next に置き換え、[r_1,r_2] を合成ルール r_3で置き換えて、合成ルール

next (r_3 , T_A , T_C) ← body.

を得る。

3.3 GLP の unfolding による例に対応したルール合成

GLP の世界では、mgu(most general unifier) が複数存在しうるので、unfolding によって、もとの 2 つの節から複数の節が生成され、合成ルールも複数ありうる。それら複数の合成ルールの中には、教師の意図に合うものもあれば合わないものもありうる。教師の意図 (具体的な変換列) が例によって与えられると仮定する。複数の合成ルールは、与えられた例によって絞れる可能性がある。本節では、複数の合成ルールの中から変換列の具体例に対応した一般的な合成ルールを選び出す手法について述べる。

まず、例を定義する。例とは、(s_1, s_2, s_3) のような状態の 3 つの組であり、s_1が変換ルール r_1によって s_2に移され、さらに s_2が変換ルール r_2によって s_3に移されるものである。

ここで、例に対応したルール合成で中間状態をチェックする必要があるために、前節の C_Uに中間状態 B を含む新たな引数 [A,B,C] を増やしてやり、述語名を PATH に置き換えると次のような節 C'_Uを得る。

```
C'u:  PATH([r1,r2],A,C,[A,B,C]) ←
          next(r1,A,B),
          next(r2,B,C).
```

そして、C'_Uの Body の 2 つの next を C_3と C_6によって展開して、Body には next が入らない次のような形の節を得る。

path ([r_1 , r_2] , T_A , T_C , [T_A , T_B , T_C]) ← body.

ここで、A は T_A、B は T_B、C は T_Cになる。

ここで、それぞれの引数 (T_A,T_B,T_C) をθ倍したものが、例 (s_1,s_2,s_3) に対応していない節は、例に対応した合成ルールだとは言えない。そこで、次の3つの条件を満たすθが存在するルールを選択する。

(1) $T_A\ \theta = s_1$

(2) $T_B\ \theta = s_2$

(3) $T_C\ \theta = s_3$

4 GLPによる変換ルールの合成例

4.1 GLPのunfoldingによる合成例

本節では、提案した方法でandAnd変換ルールとnoConnection変換ルールを合成する。

一般化論理プログラム P_0を、次のような節の集合で定義する。

$P_0 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$

ここで、C_1からC_8の節は、2章で示した節と同一のものである。

GLPのunfoldingによる合成は、次に示す節C_9から出発する。

```
C9: path([andAnd,noConnection],A,C) ←
        path([andAnd,noConnection],A,C).
```

そして次のような一般化論理プログラム P_1を作る。

$P_1 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$

C_9はトートロジーなので P_0と P_1は等価である。

まず初めに節C_9を選ぶ。C_9のBodyは1つなのでpath([andAnd,noConnection],A,C)が選ばれる。プログラム P_1の各節C_1からC_8のHeadの中でC_9のBodyとunifyするのはC_1のHeadだけである。C_1とC_9より、新たに次の節C_{10}を得る。

```
C10: path([andAnd,noConnection],A,C) ←
         next(andAnd,A,B),
         path([noConnection],B,C).
```

これによって、

$P_2 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_{10}\}$

が得られる。これはunfoldingの健全性と完全性より P_1と等価である。

この後の処理は、同じ手続きを繰り返すために次のFig.1にまとめる。

selectされたBody	unifyするHead	生成された節	プログラム
			$P_1 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$
C_9のBody	C_1のHead	C_{10}	等価
			$P_2 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_{10}\}$
C_{10}の2つめのBody	C_1のHead	C_{11}	等価
			$P_3 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_{11}\}$
C_{11}の3つめのBody	C_2のHead	C_{12}	等価
			$P_4 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_{12}\}$
C_{12}の1つめのBody	C_3のHead	C_{13}	等価
			$P_5 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_{13}\}$
C_{13}の2つめのBody	C_6のHead	C_{14} C_{15} C_{16}	等価
			$P_6 = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_{14}, C_{15}, C_{16}\}$

Fig.1 GLPのunfoldingによる合成

Fig.1において、C_{11},C_{12},C_{13}はそれぞれ次のようになる。

```
C11: path([andAnd,noConnection],A,C) ←
         next(andAnd,A,B),
         next(noConnection,B,C),
         path([],C,C).
```

ここで、3.3節と同様に、引数[A,B,C]を増やしてやり、pathをPATHに置き換えると次の節C_{12}のようになる。

```
C12: PATH([andAnd,noConnection],A,C,
                                [A,B,C]) ←
         next(andAnd,A,B),
         next(noConnection,B,C).

C13: PATH([andAnd,noConnection],A,C,
                                [A,B,C]) ←
```

```
        union(R1,R2,R3),
        next(noConnection,B,C).

    A = {[and,a,R1]
         [and,b,{a | R2}] | RR}
    B = {[and,a,R1]
         [and,b,R3] | RR}
```

union,notExist,mid は展開しないので、一般化論理プログラム P_6から unfolding によって新しいプログラムは得られない。

ユニファイは、節C_{13}の 2 つめの Body の next を展開すときに、C_{12}の 1 つめの next の第 2 引数 (これは C_3の next の第 3 引数と同じである。これを X と置く。) と 2 つめの next の第 1 引数 (これは C_6の next の第 2 引数と同じである。これを Y と置く) によって達成される。このとき、$X\theta = Y\sigma$を満たすのは、次の 3 つのユニフィケーションが考えられる。ここだけ便宜上、andAnd 変換ルールの残りの回路 RR を RR1 、noConnection 変換ルールの残りの回路 RR を RR2 と置き換える。

```
(1) [and,a,R1] θ = [ELEMENT,AA,P] σ
    {[and,b,R3] | RR1}θ = RR2 σ
(2) [and,b,R3] θ = [ELEMENT,AA,P] σ
    {[and,a,R1] | RR1}θ = RR2 σ
(3) RR1 θ = [ELEMENT,AA,P] σ
    {[and,a,R1][and,b,R3]}θ = RR2 σ
```

これらにより、健全かつ完全な unifier set は、(1),(2),(3) の場合それぞれ次のようになる。

```
(1) θ={a/AA,R1/P}
    σ={ELEMENT/and,RR2/{[and,b,R3] | RR1}}

(2) θ={b/AA,R3/P}
    σ={ELEMENT/and,RR2/{[and,a,R1] | RR1}}

(3) θ={RR1/[ELEMENT,AA,P]}
    σ={RR2/{[and,a,R1][and,b,R3]}}
```

これらより、次に示す 3 つの節を得る。

```
C14: PATH([andAnd,noConnection],TA,TC,
                              [TA,TB,TC]) ←
        union(R1,R2,R3),
        notExist(a,{[and,b,R3] | RR1}),
        mid(a).

    TA = {[and,a,R1]
          [and,b,{a | R2}] | RR1}
    TB = {[and,a,R1] [and,b,R3] | RR1}
    TC = {[and,b,R3] | RR1}


C15: PATH([andAnd,noConnection],TA,TC,
                              [TA,TB,TC]) ←
        union(R1,R2,R3),
        notExist(b,{[and,a,R1] | RR1}),
        mid(b).

    TA = {[and,b,{a | R2}]
          [and,a,R1] | RR1}
    TB = {[and,b,R3] [and,a,R1] | RR1}
    TC = {[and,a,R1] | RR1}


C16: PATH([andAnd,noConnection],TA,TC,
                              [TA,TB,TC]) ←
        union(R1,R2,R3),
        notExist(AA,
                 {[and,a,R1][and,b,R3]}),
        mid(AA).

    TA = {[and,b,{a | R2}]
          [and,a,R1]
          [and,ELEMENT,AA,P]}
    TB = {[and,a,R1] [and,b,R3]
          [and,ELEMENT,AA,P]}
    TC = {[and,a,R1] [and,b,R3]}
```

このように、複数のユニフィケーションが存在することから、GLP の unfolding による合成から 3 つの生成節を得る。

4.2 例に対応した合成

次のような例を与えることで、例に対応しているかどうかをチェックする。

(1) 初期状態に対する例

{[and, o, {mp1 mp2}]
 [and, mp2, {mq1 mq2}]}

(2) 中間状態に対する例

{[and, mp2, {mq1 mq2}]
 [and, o, {mq1 mq2 mp1}]}

(3) 最終状態に対する例

{[and, o, {mq1 mq2 mp1}]}

前節で求めた 3 つの節のうち、まず、C_{14}について見てみる。θを

θ = {a / {mp2},R1 / {mq1 mq2},b / o,
R2 / {mp1},RR1 / (),R3 / {mq1 mq2 mp1}}
とすれば、$T_A\theta$=s_1、$T_B\theta$=s_2、$T_C\theta$=s_3となり、また Body に対する条件をチェックすると True となる。C_{15},C_{16}に関してはこのようなθはない。故に、C_{14}だけを選ぶことになる。この合成結果を図で示すと、Fig.2 のようになる。

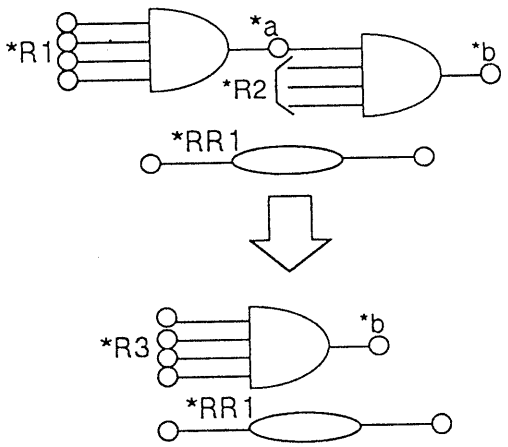

Fig.2　合成結果(GLP)

5　Prolog の unfolding による合成

2,3,4 章では、GLP の unfolding による合成について述べたが、この方法の有用性を明らかにするために Prolog の unfolding による合成と比較する。故に、この章では Prolog の unfolding による合成について説明する。

5.1　Prolog による回路変換プログラム

Prolog の場合も GLP の場合と同様に、andAnd 変換ルールと noConnection 変換ルールが与えられたとき、それらを合成したルールを作る例について説明する。

変換プログラム path に関しては、2.1 節の C_1,C_2と同じであるので、それぞれ C_1',C_2'とする。

Prolog による andAnd 変換ルールを次に示す。

```
C'3: next(andAnd,CIRCUIT1,CIRCUIT2) ←
        member_rest([and,b,IN],
                CIRCUIT1,RESTCIRCUIT),
        member_rest([and,a,R1],
                        RESTCIRCUIT,RR),
        member_rest(a,IN,R2),
        member_rest([and,b,R3],
                        R,RESTCIRCUIT),
        union(R1,R2,R3).
```

1 つめの member_rest 以下が節の Body である。Prolog のプログラムでは、Body が非常に長くなり情報を把握することが簡単ではない。このプログラムの述語 next は 3 つの引数を持っている。第 1 引数は、andAnd で、これはルール名を示す。第 2 引数は、変形前の回路を示し、第 3 引数は、変形後の回路を示す。このプログラムの中の述語は次の意味である。

member_rest(X,Y,Z) $\cdots$ Y = {X}$\cup$Z

union(X,Y,Z) $\cdots$ X$\cup$Y = Z

member_rest,union のプログラムを次に示す。

```
C'4: member_rest(A,{A | R},R) ←
          member_rest(A,{X | R},{X | rr}),
          member_rest(A,R,RR).

C'5: union({X | U},Y,{X | V}) ←
          union(U,Y,V).

C'6: union({},Y,Y) ←.
```

Prolog による noConnection 変換ルールは、次のようになる。

```
C'7: next(noConnection,CIRCUIT,RR) ←
        member_rest([ELEMENT,AA,P],
                        CIRCUIT,RR),
        notExist(AA,RR),
        mid(AA).
```

member_rest 以下が節の Body である。ここで、notExist の定義節を C_8'、mid の定義節を C_9'とする。

5.2 Prologの unfolding

合成するときに、Prologの unfoldingを用いるので、本節では、Prologの unfoldingを定義する。

1. $P_1 = \{C_j \mid j \in J\}$ から節 $C_k = (H \leftarrow A_1, \cdots, A_i, \cdots, A_a)$ を選ぶ。
2. 節 C_kの Body から A_iを選ぶ。
3. renaming ρを適切に選んで、$P_1\ \rho = \{C_j\ \rho \mid j \in J\}$ を作り、P_1 ρの各節 $C_j\ \rho$が、C_k と変数を共有しないようにする。
4. プログラム P_1 ρの節のうち、A_iと unifyする Headを持つ節をすべて選ぶ。それらの節の集合を P_ρとする。
5. P_ρの各節 $C_j\ \rho$の Head と A_iの mgu(most general unifier) の任意の1つをθ_jとする。
6. $P_2 = P_1 - \{C_k\} \cup$
$\{(H\ \theta \leftarrow A_1\ \theta, \cdots, A_{i-1}\ \theta,$
$B_1\ \theta, \cdots, B_b\ \theta,$
$A_{i+1}\ \theta, \cdots, A_a\ \theta) \mid C_j\ \rho \in P_\rho\}$
として P_2を得る。

ここで、renaming,mguは次のように定義されている。

renamingとは、変数を変数に1対1に置き換える代入、即ち、$X_1, X_2, \cdots, X_n$が互いに異なる変数で、$Y_1, Y_2, \cdots, Y_n$もまた互いに異なる変数であるとき、$\{X_1/Y_1, X_2/Y_2, \cdots, X_n/Y_n\}$ の形の代入である。

Θがアトム x とアトム y の mgu であるとは、Θが x と y の unifier であり、しかも x と y の任意の unifierθに対して、ある代入σが存在して、$\Theta\ \sigma = \theta$ が成り立つことである。

5.3 Prologの unfoldingによる変換ルールの合成例

本節では、Prologの unfoldingによる変換ルールの合成を行ない、その合成例について述べる。

プログラム P_0を、次のような節の集合で定義する。

$P_0 = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6, C'_7, C'_8, C'_9\}$

ここで、C'_1から C'_9の節は、5.1節で示したものと同一である。

Prologの unfoldingによる合成は、次に示す節 C'_{10}から出発する。

```
C'10: path([andAnd,noConnection],A,C) ←
          path([andAnd,noConnection],A,C).
```

そして次のようなプログラム P_1を作る。

$P_1 = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6, C'_7, C'_8, C'_9, C'_{10}\}$

C'_{10}はトートロジーなので P_0と P_1は等価である。

P_1から unfoldingによって $P_2, \cdots, P_6$と変化する。これを次の Fig.3 にまとめる。

selectされたBody	unifyするHead	生成された節	プログラム
			P_1 = {C'1,C'2,C'3,C'4,C'5,C'6,C'7,C'8,C'9,C'10}
C'10のBody	C'1のHead	C'11	等価
			P_2 = {C'1,C'2,C'3,C'4,C'5,C'6,C'7,C'8,C'9,C'11}
C'11の2つめのBody	C'1のHead	C'12	等価
			P_3 = {C'1,C'2,C'3,C'4,C'5,C'6,C'7,C'8,C'9,C'12}
C'12の3つめのBody	C'2のHead	C'13	等価
			P_4 = {C'1,C'2,C'3,C'4,C'5,C'6,C'7,C'8,C'9,C13}
C'13の1つめのBody	C'3のHead	C'14	等価
			P_5 = {C'1,C'2,C'3,C'4,C'5,C'6,C'7,C'8,C'9,C'13,C'14}
C'13の2つめのBody	C'7のHead	C'15	等価
			P_6 = {C'1,C'2,C'3,C'4,C'5,C'6,C'7,C'8,C'9,C'15}

Fig.3 Prologのunfoldingによる合成

但し、Fig.3において、C'_{11},C'_{12},C'_{13},C'_{14},C'_{15}はそれぞれ次のようになる。

```
C'11: path([andAnd,noConnection],A,C) ←
```

```
        next(andAnd,A,B),
        path(noConnection,B,C).

C'12: path([andAnd,noConnection],A,C) ←
        next(andAnd,A,B),
        next(noConnection,B,C),
        path([],C,C).

C'13: path([andAnd,noConnection],A,C) ←
        next(andAnd,A,B),
        next(noConnection,B,C).

C'14: path([andAnd,noConnection],
                          CIRCUIT1,C) ←
        member_rest([and,b,IN],
                CIRCUIT1,RESTCIRCUIT),
        member_rest([and,a,R1],
                    RESTCIRCUIT,RR),
        member_rest(a,IN,R2),
        member_rest([and,b,R3],
                    R,RESTCIRCUIT),
        union(R1,R2,R3),
        next(Rule,B,C).

C'15: path([andAnd,noConnection],
                          CIRCUIT1,RR) ←
        member_rest([and,b,IN],
                CIRCUIT1,RESTCIRCUIT),
        member_rest([and,a,R1],
                    RESTCIRCUIT,RR),
        member_rest(a,IN,R2),
        member_rest([and,b,R3],
                    R,RESTCIRCUIT),
        union(R1,R2,R3),
        member_rest([ELEMENT,AA,P],
                                R,RR),
        notExist(AA,RR),
        mid(AA).
```

member_rest,union,notExist,mid は展開しないので、プログラム P_6から unfolding によって新しいプログラムは得られない。故に、Prolog の unfolding による合成から C'_{15}が得られる。

6 GLP と Prolog の unfolding による変換ルールの合成の比較

本章では、GLP の unfolding による変換ルールの合成例と、Prolog の unfolding による変換ルールの合成例を比較することで、GLP の合成における有用性を述べる。分かり易いように、Fig.4 と照らし合わせて見てみる。

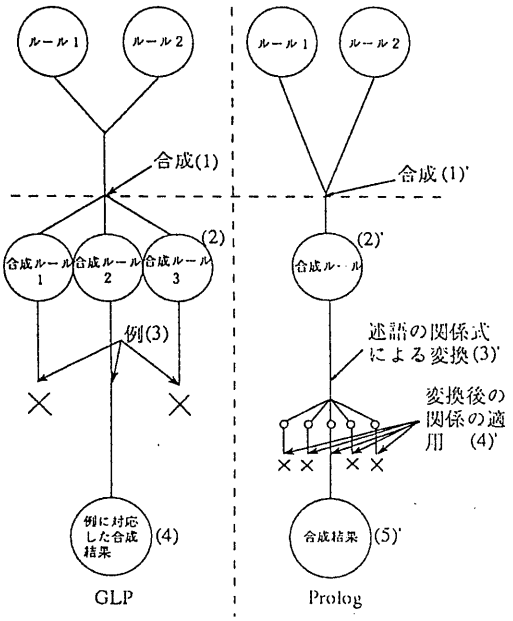

Fig.4 GLPとPrologの合成の比較

GLP の unfolding による合成例では、変換ルールを集合オブジェクトとして扱ったために、もとの 2 つの節をユニファイした結果、複数のユニフィケーションが自動的に現れ、それに例を与えることで、例に対応した合成結果を選択できた。一方、Prolog の unfolding による合成の場合、5.3 節で述べたように、合成結果は C'_{15}のように単数になる。この状態は、GLP の場合と同様に、複数の合成結果を含んだものになっている。しかし、分離することは不可能である。また、合成結果 C'_{15}は C'_3と C'_6のそれぞれの Body を並べただけにすぎない。これでは合成によるメリット (高速化等) は得られない。これらの状態は Fig.4 の (2)' に相当する。

Prolog の unfolding による合成では、C'_{15}の結果以上どうすることもできず、GLP の unfolding による合成と同じ結果を得ることはできない。そこで GLP の unfolding による合成結果を参照することで、この合成結果と同じ結果を得るための方法がありうるだろうか。実は C'_{15}の結果から GLP の場合と同じ結果を得るには、次に述べ

るように余分な労力を必要とし、極めて困難である。

1. 5.3 節の C'_{15} を図示することによって 4.2 節の Fig.2 の合成結果より、どの端子が中間端子に相当するかを推測することが必要になる。これによって、C'_{15} の [ELEMENT,AA,P] とユニファイすればよさそうなことが分かる。[Fig.4,(3)']

2. 合成結果をより良いものにするために、最適な Body の中の述語の関係式を考えれば良いと思われる。これは C'_{15} のユニファイ後の述語 member_rest は 5 つあるので、これらの関係式より述語の数を減らすことを考えるということである。member_rest の数は 5 つあるので、例えば 3 つの member_rest の関係を 1 つにすることを考えるとき、$5 \times 4 \times 3 = 60$ 組の member_rest の組合せが考えられる。これらより最適な組合せを選び出し、述語の関係式にする。しかし、この選択は非常に困難である。[Fig.4,(3)',(4)']

3. さらに、2 に対して理論的裏付けを与えるためには、2 で考えた述語の関係式を、理論的に証明することが必要になる。

以上より、GLP の unfolding による合成結果と同等の結果を得るには、余分な労力を必要とし、極めて困難であることが言える。故に、GLP の unfolding による変換ルールの合成の方が有用性が高いことが明らかである。

7 むすび

本論文では、変換ルールの合成法について論じた。ポイントは、次の通りである。

- GLP の unfolding によるルール合成法を提案し、それに基づいて合成を行なった。
- GLP の unfolding による例に対応したルール合成を提案し、それに基づいて例に対応した合成を行なった。
- GLP の unfolding によるルール合成の有用性を示すために、Prolog の unfolding によるルール合成を行ない、両者を比較することで GLP の unfolding によるルール合成の有用性を明らかにした。

GLP は、Prolog の領域を含み、さらに Prolog の領域を拡張した適用範囲の広い論理プログラムである。故に、Prolog でカバーできる処理は勿論のこと、カバーできない範囲までも表現し、処理できる。さらに、オブジェクトに対して、適切な表現ができるので、プログラム変換にとってとても有効的である。

本論文の説明では、比較的簡単な変換ルールの合成を扱ったが、複雑な変換ルールの合成では、GLP と Prolog の差は、ますます広がることが予想される。

参考文献

[1] 馬淵, 赤間, 青木: 一般化論理プログラムの部分計算による論理回路の変換ルールの合成法, 情報処理学会, 人工知能研究会資料, PP.11-19 (1991)

[2] 沼尾 正行: 説明に基づく学習, 一領域固有の知識を用いたアプローチー, 人工知能学会誌, vol.3, No.6, pp.704-711 (1988)

[3] Akama,K.: Sufficient Conditions of Two Inference Rules for Generalized Logic Programs The Logic Programming Conference '91 pp.161-170 (1991)

[4] 山田 誠二, 安部 憲広, 辻三郎:問題解決における戦略知識学習システム:PiL, 人工知能学会誌 ,vol.3, No.2, pp.76-85 (1988)

[5] Akama,K.: Unfolding of Generalized Logic Programs, Preprints Work.Gr. for Artificial Intelligence, IPSJ83-3-AI, pp.43-52 (1992)