TM-0716

プログラムを最適化するには

藤田　博、古川康一

May, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456 3191-5
Telex ICOT J32964

プログラムを最適化するには

― プログラム変換と並列プログラミング ―

藤田 博 ・ 古川 康一

1. はじめに

プログラマが書き易く書けたプログラムは，作成コストが安くついて良いプログラムである．さらに，それが誰が読んでも分かり易ければメンテナンスコストが安くついてもっと良いプログラムになる．こうして，人にとって最も都合のよいプログラムというものが考えられる．

一方，同じ働きをするプログラムが幾つかあるとき，実際に計算機上で走らせてみると計算時間や記憶容量の点でほかにはるかに優るものがある．即ち，機械にとって最も都合のよいプログラムというものが考えられる．

人にとって最良のプログラムが，常に同時に機械にとっても最良のプログラムとなるならば，問題は簡単である．我々はひたすら，分かり易いプログラムを書くように努めていればよいのだから．しかし，多くの場合そうではない．分かり易さを犠牲にして最大効率を目的としたプログラムや，場合によっては，効率は少し犠牲にしても分かり易さを優先したプログラムが作られるのが普通である．(分かりにくく効率も悪いばかりか，とんでもない虫が潜んでいるというようなプログラムが多いのが実情ではあるまいか?)

しかし，分かり易いプログラムから，その機能を保ちつつ最大効率のプログラムを機械的手段で導けるとすれば，プログラマ側の問題は再び簡単になる．そのような機械的手段を提供するのがプログラム変換の目的である．

プログラム変換には二つのアプローチがある．展開／たたみ込み変換と部分計算である．

展開／たたみ込み変換はプログラム変換の基本であって，狭義のプログラム変換といえば展開／たたみ込み変換のことだと思ってもよい．

今，プログラム $\mathcal{P}$ は手続きの集合で，各手続きは $p \leftarrow a, b, \ldots, z$ の形で与えられ，p を頭部，手続き本体 $a, b, \ldots, z$ を幾つかの手続き呼び出しの並びとしよう．このとき，一般に展開／たたみ込み変換とは手続きに対する次のような書き換え規則のことである．

(i) 展開規則：$\mathcal{P}$ 中に

$$p \leftarrow \varphi\ q\ \psi \qquad \cdots ①$$

$$q \leftarrow a, b, \cdots, z \qquad \cdots ②$$

があるとき，①を次の③で置き換える．

$$p' \leftarrow \varphi\ a, b, \cdots, z\ \psi \qquad \cdots ③$$

(ii) たたみ込み規則：$\mathcal{P}$ 中に

$$p \leftarrow \varphi\ a, b, \cdots, z\ \psi \qquad \cdots ①$$

$$q \leftarrow a, b, \cdots, z \qquad \cdots ②$$

があるとき，①を次の③で置き換える．

$$p' \leftarrow \varphi\ q\ \psi \qquad \cdots ③$$

ただし，φ, ψ は 0 個以上の手続き呼び出しの並びとする．また，手続き呼び出しと手続き頭部の間での引き数の受け渡しに関して条件が課せられて，実際の規則はもっと複雑になっている．

一方，部分計算は，あるプログラムにその入力が与えられる前に実行できる計算が含まれているとき，これを実行して簡単化したプログラムを得るものである．今，プログラム $\mathcal{P}$ の入力を，K(Known：既知) と U(Unknown：未知) の二つに分けられるとしよう．このとき，部分計算 PE は次のように表現される．

$$PE[\![\mathcal{P}, K]\!] = \mathcal{P}_K \qquad \cdots ①$$

ここで，$\mathcal{P}_K$ は剰余プログラムと呼ばれる．このプログラムに残りの U を与えて実行し，結果 A を得ることを，

$$E[\![\mathcal{P}_K, U]\!] = A \qquad \cdots ②$$

と表現することにしよう．これに対して，K と U が揃っているときの $\mathcal{P}$ の普通の実行を全計算といい，

$$E[\![\mathcal{P}, (K, U)]\!] = A' \qquad \cdots ③$$

と表現する．

部分計算の正しさを議論するとき，健全性と完全性の二つが問題となる．ここで，①の計算が結果 A を与えるならば③の計算が結果 $A' = A$ を与えるとき，剰余プログラム $\mathcal{P}_K$ は $\mathcal{P}$ と K, U に関して健全であるといい，逆も言えれば完全であるという．

部分計算の正しさは当然の要件であるが，①での計算コストが③のそれに比べて安価になっていることが実用上もう一つの必須要件である．

本稿では，これら二つの技術，すなわち展開 / たたみ込み変換および部分計算，を並列論理型言語 GHC で書かれたプログラムに如何に適用されるかについて述べてみたい．

2. 展開 / たたみ込み変換によるプロセス融合

Prolog プログラムの場合，展開 / たたみ込み変換が元のプログラムの意味を変えないことは，玉木 - 佐藤[8] によって示された．しかしながら，GHC プログラムの場合は実行のタイミングの問題があるので，話はそう簡単ではない．GHC の場合，ボディ部のゴールを展開する時，そのゴールをただ単にそのゴールと同一化可能なヘッドをもつ節の右辺で置き換える訳にはいかない．というのは，そうするとコミット・オペレータが一つの節に二つ以上現れてしまうからである．

今，ボディ・ゴールの展開において，ガード部は元の節のガード部に追加し，ボディ部も同様に元の節のボディ部に追加することにしよう．ところで，このような展開操作を不用意に行うと，デッド・ロックが起こってしまうことを，例によって示そう．今，図 1 のように首の所が細くなっている壷の中にバナナが二本あった時，猿がバナナを二本とも取り出す操作を GHC のプログラムで記述すると，図 2 のようになる．この図で，節 M1 の monkey(X,Y) は，X=[Hand1|X1] ならば猿が手 Hand1 によってバナナを一本掴み，Y=grasped になることを表わし，節 W1 の withdraw(Y,X1) は，Y=grasped ならば，手 X1 が二本目のバナナを掴める状態 (X1=[Hand2]) になることを表わす．実際に二本目のバナナを掴むのは，ゴール monkey1(X1) の実行によってである．この時，節 M2 のガード条件から X1=[Hand2] でなければならない．プログラムの実行は，ゴール G1 によって起動される．このゴールは，初め猿が空の手の状態になっていることを示している．

このプログラム {M1,M2,W1,G1} は，デッド・ロックを起こさずに成功して止まる．さて，ここで，節 M1 のボディ・ゴール monkey1(X1) を展開してみよう．そうすると，節集合 {M1,M2} の代わりに次のような節 M12 が得られる．

```
M12.  monkey(X,Y) :-
          X=[Hand1|X1],X1=[Hand2] |
          Hand1=banana1,Hand2=banana2,Y=grasped.
```

ところが, プログラム {M12,W1,G1} は, デッド・ロックしてしまう. それは, 節M12のガードが, X=[Hand1|X1] かつ X1=[Hand2], すなわち X=[Hand1,Hand2] のように二つの空の手を同時に要求しているからである. すなわち, 節M12は, 二本のバナナを同時に取ろうとする欲張りな猿を表わしている.

我々は, このような不都合が起こらないようなGHCの展開 / たたみ込み変換アルゴリズムを開発することに成功した[6)9)]. そのエッセンスは, 出力変数に対する同一化がない時にのみガードにまたがる展開を許す, という考えである. また, その条件に合う形にプログラムを変換するために, たたみ込みを利用する. そのアルゴリズムリズムは, 次の四つの変換ルールからなる.

(1) 正規化規則　　同一化ゴールを実行して, 単純化する.
(2) 即時実行規則　　ボディに現れる即時実行可能なゴールを展開する.
(3) 場合分け展開規則　出力変数への同一化がないことを条件に,
　　ガードにまたがる展開を行う.
(4) たたみ込み規則　　与えられた述語によって, たたみ込みを行う.

この展開 / 畳み込み変換アルゴリズムを用いて, 与えられた整数列の二乗和を求める2プロセスのプログラムを1プロセスのプログラムに変換する過程を示そう. 初めに, 元の2プロセスGHCプログラムを示す.

```
squareSum(Ns,S) :- true | square(Ns,Qs), sum(Qs,0,S).    ①

square([N|Ns],Qs) :- true |
    Q := N**2, Qs=[Q|Qs1], square(Ns,Qs1).               ②

square([],Qs) :- true | Qs=[].                           ③

sum([N|Ns],T,S) :- true |
    NewT := T+N, sum(Ns,NewT,S).                         ④

sum([],T,S) :- true | S=T.                               ⑤
```

今, 二つのプロセスを融合して一つのプロセスにした手続きを fusedSquareSum(Ns,T,S) としよう. それは, square と sum を用いて, 次のように定義される.

```
fusedSquareSum(Ns,T,S) :- true |
```

```
square(Ns,Qs), sum(Qs,T,S).                                    ⑥
```

元の squareSum 手続きは, fusedSquareSum を用いて, 次のように表わすことができる.

```
squareSum(Ns,S) :- true | fusedSquareSum(Ns,0,S).              ⑦
```

さて, fusedSquareSum の定義⑥は, 依然として 2 プロセスである. これを 1 プロセスからなるプログラムにするには, ⑥を fusedSquareSum の再帰呼び出しを行うプログラムにしなければならない. また, そのプログラムは, 1 回の再帰呼び出しで square および sum の両者の 1 回の再帰呼び出しと同じことをするはずなので, ⑥の右辺を, square および sum について各 1 回展開してみる. 初めに, square ゴールを展開する. このゴールは即時実行可能ではない (Ns が具体化されていない) ので, 場合分け展開規則を用いる. すなわち, square プログラムのガード部の条件に着目し, Ns=[N|Ns1] および Ns=[] の二つの場合に分けて展開を行う. それ以降の変換は, それぞれの場合毎に, 独立に進める.

(a) Ns=[N|Ns1] の場合

```
fusedSquareSum([N|Ns1],T,S) :- true |
    Q:=N**2, Qs=[Q|Qs1], square(Ns1,Qs1), sum(Qs,T,S).         ⑧
```

(正規化規則による同一化の実行)

```
fusedSquareSum([N|Ns1],T,S) :- true |
    Q:=N**2, square(Ns1,Qs1), sum([Q|Qs1],T,S).
```

(即時実行規則による sum ゴールの展開)

```
fusedSquareSum([N|Ns1],T,S) :- true |
    Q:=N**2, square(Ns1,Qs1), NewT:=T+Q,
    sum(Qs1,NewT,S).                                           ⑨
```

(たたみ込み規則)

```
fusedSquareSum([N|Ns1],T,S) :- true |
    Q:=N**2, NewT:=T+Q, fusedSquareSum(Ns1,NewT,S).            ⑩
```

ここで得られた(10)式は，確かに fusedSquareSum の再帰プログラムになっている．次に，Ns=[] の場合について，変換を行う．

(b) Ns=[] の場合

```
fusedSquareSum([],T,S) :- true |
    square([],Qs), sum(Qs,T,S).                    ⑪
```

↓ (即時実行規則による square の展開)

```
fusedSquareSum([],T,S) :- true | Qs=[], sum(Qs,T,S).    ⑫
```

↓ (正規化規則による同一化の実行)

```
fusedSquareSum([],T,S) :- true | sum([],T,S).          ⑬
```

↓ (即時実行規則による sum の展開)

```
fusedSquareSum([],T,S) :- true | S=T.                  ⑭
```

場合分けのそれぞれの変換結果⑬式および⑭式を集めると，fusedSquareSum プログラムの再帰プログラムが得られていることが分かるであろう．これらの二つの式とその定義を元の squareSum とを結び付ける⑦式からなる集合が求めるプログラムである．得られたプログラムは，元のプログラムと異なり，再帰的なプロセスは唯一つで，プロセス間の通信に用いられた無限長のバッファ Qs を必要としていない．これは，二つのプロセスを同期を取って走らせるようにしたために，不要になったものである．

3. 部分計算による最適化

部分計算は，汎用プログラムを個別の用途毎に最適なプログラムに特化するための技法と考えることができる．例えば，前に述べた $\langle P, K, U \rangle$ として具体的に，

(i) ⟨汎用構文解析プログラム，文法，解析される文章⟩，

(ii) ⟨メタレベルプログラム，オブジェクトレベルプログラム，実行時入力⟩，

```
parse_count(Sentence,Grammar,N):-true|
   parse(Sentence,Grammar,[begin],OutS),
   count(OutS,0,N).

parse([Word|Rest],Grammar,InS,OutS):-true|
   leftmost(Word,Grammar,Grammar,InS,InS1),
   parse(Rest,Grammar,InS1,OutS).
parse([],_,InS,OutS):-true|OutS=InS.

count([end|Rest],I,N):-true|I1:=I+1,count(Rest,I1,N).
count([S|Rest],I,N):-S\=end|count(Rest,I,N).
count([],I,N):-true|N=I.
```

図 3. Meta-PAX プログラム (parse_count, parse, count)

(iii) ⟨汎用定理証明系，公理，証明すべき命題⟩

などの場合が有効であることがわかっているが，ほかにもプログラム $\mathcal{P}$，既知情報 K と未知情報 U の切り分け次第で，面白い応用があるに違いない．

なかでも特に興味深いのは，$\mathcal{P}$ をある言語 L のインタプリタ I^L とおいた場合である．この場合，PE が汎用のコンパイラの役割を果たすことが示される．次に，$\mathcal{P}$ を PE 自身，K を I^L とおけば，PE_{IL} は L 専用のコンパイラとなる．さらに，$\mathcal{P}$ も K も PE 自身とおいた場合，PE_{PE} はコンパイラコンパイラとなるだろう．このアイデアは二村射影[4] として有名である．

ここでは，(i) の例として本誌別稿でも詳しく説明されている並列パーサをとりあげてみよう．ここで扱うのは Meta-PAX と呼ばれるインタプリタ方式の汎用構文解析プログラムで，入力文が予め与えられた文法に照らして正しい文かどうかを判定するものである．図 3 から図 6 に並列論理型言語 GHC で記述された Meta-PAX のプログラムを示す．

Meta-PAX は，ボトムアップ型の構文解析を行う．即ち，入力文を左から右へ見ていきながら適合する文法規則を選び，単語から始めて次第に上の階層の文法要素 (カテゴリ) へと文の構造 (構文木) を決定していく．例えば，名詞から名詞節へ，次に動詞から動詞節へ，最後に名詞節と動詞節から文へ，という具合である．また，複数の解釈ができる文に対しては，Meta-PAX はそのあらゆる可能性を並行して試みるようになっている．そこで，レイヤードストリーム[7] という探索問題の全解を並列に効率よく求めるためのプログラミング技法が用いられている．

```
leftmost(Cat,[(Cat1-->Cat,Rest)|Rules],Grammar,InS,OutS):-true|
   OutS=[(Rest->Cat1)*InS|OutS1],
   leftmost(Cat,Rules,Grammar,InS,OutS1).
leftmost(Cat,[(Cat1-->Cat)|Rules],Grammar,InS,OutS):-true|
   middle(Cat1,Grammar,InS,OutS1),
   leftmost(Cat1,Grammar,Grammar,InS,OutS2),
   leftmost(Cat,Rules,Grammar,InS,OutS3),
   merge(OutS1,OutS2,OutS12),
   merge(OutS12,OutS3,OutS).
leftmost(Cat,[(_-->RHS)|Rules],Grammar,InS,OutS):-
       RHS\=Cat, RHS\=(Cat,_) |
   leftmost(Cat,Rules,Grammar,InS,OutS).
leftmost(_,[],_,_,OutS):-true|OutS=[].
```

図 4. Meta-PAX プログラム (leftmost)

図 3 の parse_count(Sentence,Grammar,N) は，入力文 Sentence と文法規則 Grammar を入力して，可能な解釈の数 N を返す．この数は，parse の出力ストリーム OutS に流れてくる end の数に等しい．

図 4 の leftmost(Cat,Rules,Grammar,InS,OutS) の基本的役割は，文法規則の列 Rules のうち，文法カテゴリ Cat を右辺の最左端にもつようなもの Cat1-->Cat,Rest を選び，その規則の残りの断片を Rest->Cat1 として，ストリーム OutS に出力することである．この意味は，Cat の後，Rest が残りの入力文と適合できれば上位のカテゴリ Cat1 が認識できるということを，後の解析部に通知するということである．特に，文法規則が Cat1-->Cat で，直ちに上位カテゴリ Cat1 に遷移できるときは，middle(Cat1,Grammar,InS,OutS1) によって，Cat1 を中間カテゴリとするような解析，および，leftmost(Cat1,Grammar,Grammar,InS,OutS2) によって，Cat1 を最左端カテゴリとするような解析を新たに始める．

図 5 の middle(Cat,Grammar,InS,OutS) の基本的役割は，ストリーム InS に流れてくる文法規則の断片のうち，文法カテゴリ Cat をその最左端にもつようなもの Cat,Rest->Cat1 を選び，さらに残った断片を Rest->Cat1 として，ストリーム OutS に出力することである．流れてきた文法規則の断片が Cat->Cat1 で，直ちに上位カテゴリ Cat1 に遷移できるときは，leftmost(Cat1,Grammar,Grammar,IInS,OutS1) によって，Cat1 を最左端カテゴリとするような解析，および，middle(Cat1,Grammar,IInS,OutS2) によって，Cat1 を

```
middle(Cat,Grammar,[(Cat,Rest->Cat1)*IInS|InS],OutS):-true|
   OutS=[(Rest->Cat1)*IInS|OutS1],
   middle(Cat,Grammar,InS,OutS1).
middle(Cat,Grammar,[(Cat->Cat1)*IInS|InS],OutS):-true|
   leftmost(Cat1,Grammar,Grammar,IInS,OutS1),
   middle(Cat1,Grammar,IInS,OutS2),
   middle(Cat,Grammar,InS,OutS3),
   merge(OutS1,OutS2,OutS12),
   merge(OutS12,OutS3,OutS).
middle(s,Grammar,[begin|InS],OutS):-true|
   OutS=[end|OutS1],
   middle(s,Grammar,InS,OutS1).
middle(Cat,Grammar,[S|InS],OutS):-
      S\=(Cat->_)*_, S\=(Cat,_->_)*_, (Cat,S)\=(s,begin) |
   middle(Cat,Grammar,InS,OutS).
middle(_,_,[],OutS):-true|OutS=[].
```

図 5. Meta-PAX プログラム (middle)

```
merge([H|InS1],InS2,OutS):-true|
   OutS=[H|OutS1],merge(InS1,InS2,OutS1).
merge(InS1,[H|InS2],OutS):-true|
   OutS=[H|OutS1],merge(InS1,InS2,OutS1).
merge([],InS2,OutS):-true|OutS=InS2.
merge(InS1,[],OutS):-true|OutS=InS1.
```

図 6. Meta-PAX プログラム (merge)

中間カテゴリとするような解析を新たに始める. InS に流れてきたのが begin で, 現在のカテゴリが最上位カテゴリの s であるときは, 一つの文の解釈の完成を示す end をストリーム OutS に出力する.

図 6 の merge(InS1,InS2,OutS) は, 二本のストリーム InS1 と InS2 を一本のストリーム OutS に併合する.

図 7 に入力文と文法の例を示す.

Meta-PAX は高度な並列処理により, それ自身極めて効率のよいプログラムであるということができるが, 部分計算を行うことによって, さらに効率を向上させることができる. 即ち, 文法を固定入力と考えて, 汎用の Meta-PAX を一旦その文法専用のパーサに特化しておけば, あとで与えられる入力文に対する処理が効率よく行えるわけである.

```
example(N) :- true | parse_count([time,flies,like,an,arrow],
          [(s --> np, vp),
           (np --> n),
           (np --> det, n),
           (np --> adj, np),
           (adj --> n),
           (vp --> v),
           (vp --> v, np),
           (vp --> v, adv),
           (adv --> prep, np),
           (n --> time),
           (n --> flies),
           (v --> flies),
           (v --> like),
           (prep --> like),
           (det --> an),
           (n --> arrow)],
          N).
```

図 7. Meta-PAX プログラム (example)

上の例について部分計算を実行した結果の一部を図 8 に示す．ここでは，GHC プログラムに対する部分計算法[3] を適用した．文法中に現れる各文法カテゴリごとに特化された leftmost のプログラム節が生成されている．特化された leftmost では，それぞれのカテゴリから生じ得る文法規則の断片のすべてが既に分かっているので，直ちに関連するすべての middle の呼び出しおよび文法規則の断片の出力が行われるようになっている．文法規則 Grammar は既に使われてしまっているので，leftmost および middle の引き数から除かれている．

さらに，middle についても最適化を行うと，最終的にこの部分計算で得られたプログラムは，手書きによる PAX トランスレータで作られたものにほぼ匹敵する効率のよいものになる．このように，部分計算による方法は，コンパイラやトランスレータに比べてインタプリタの方がはるかに簡単な場合，プロトタイピング，機能拡張の際の融通性と効率を同時に満足する優れた手法となる．

4. 今後の課題

プログラムを最適化する機械的手段として，プログラム変換と部分計算についてその研

```
leftmost(time,InS,OutS) :- true |
   OutS=[(vp->s)*InS,(np->np)*InS|OutS1_354],
   middle(n,InS,OutS1_38),
   middle(np,InS,OutS1_58),
   middle(adj,InS,OutS1_102),
   merge(OutS1_58,OutS1_102,OutS1_338),
   merge(OutS1_38,OutS1_338,OutS1_354).

leftmost(flies,InS,OutS) :- true |
   OutS=[(np->vp)*InS,(adv->vp)*InS,(vp->s)*InS,
         (np->np)*InS|OutS1_585],
   middle(n,InS,OutS1_42),
   middle(np,InS,OutS1_62),
   middle(adj,InS,OutS1_126),
   merge(OutS1_62,OutS1_126,OutS1_522),
   middle(v,InS,OutS1_50),
   middle(vp,InS,OutS1_178),
   merge(OutS1_50,OutS1_178,OutS1_526),
   merge(OutS1_42,OutS1_522,OutS1_554),
   merge(OutS1_554,OutS1_526,OutS1_585).
```

図 8. 特化された Meta-PAX プログラム (部分)

究成果の一端を紹介した.

展開／たたみ込み変換は，それ自体は単にプログラム変換の一ステップを行うための道具にすぎない．どの手続きを選んで，どういう順序で展開／たたみ込みの各規則を適用すればよいか，また，展開／たたみ込み規則をうまく適用するためにほかにどんな補助的な規則があればよいか，など，変換戦術や戦略に関する最も人を悩ます問題がまだ完全には解決されていない．人のプログラムを最適化する機械的手段としてのプログラム変換は，まだ実用的な域には達していないのが現状なのである.

部分計算については，一昨年デンマークで国際会議が開催され，現在行われている研究の成果について多くの発表があったほか，今後の課題についての全体討議の結果が会議資料[1]に報告されている．そこには，二村射影のアイデアが現実的なコンパイラ生成技法としていかにして実用化できるか，また，部分計算で計算量のオーダーを変える程のプログラムの最適化が可能かどうか，といった問題が大きなテーマとして挙げられている.

近年，人の思考の基本となる論理をベースとしつつ，並列処理を記述できる高水準プロ

グラミング言語GHCが開発された．これにより，分かり易いプログラムが同時に最大効率のプログラムになるという理想に一歩近づいたといえるかもしれない．しかしながらなお，ここで示したようにプログラム変換や部分計算が，さらに最適なプログラムを作り出すために有用な技術であることに変わりはなく，今後一層の研究の進展が望まれている．

参考文献

1) D. Bjørner, A.P. Ershov and N.D. Jones (eds.), Partial Evaluation and Mixed Computation, North-Holland, 1988, 論理プログラミング分野については， *New Generation Computing*, Vol.6, Nos.2,3, 1988.

2) 淵 一博 監修, 古川康一・溝口文雄 共編， プログラム変換, 知識情報処理シリーズ第7巻, 共立出版, 1987.

6) K. Furukawa, A. Okumura, M. Murakami, Unfolding Rules for GHC Programs, *New Generation Computing*, Vol.6, pp.143-157, 1988

3) H. Fujita, A. Okumura and K. Furukawa, "Partial Evaluation of GHC Programs Based on the UR-set with Constraints," In R.A.Kowalski and K.A.Bowen (eds.) *Proc. of the Fifth International Conference and Symposium on Logic Programming*, pp.924-941, 1988.

4) Y. Futamura, "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler," *Systems, Computers, Controls*, 2(5) pp.45-50, 1971.

5) Y. Matsumoto, "A Parallel Parsing System for Natural Language Analysis," In E. Shapiro (ed.) *Proc. of the Third International Conference on Logic Programming*, pp.396-409, 1986.

7) 奥村 晃, 松本裕治， レイヤードストリームを用いた並列プログラミング, The Logic Programming Conference '87 論文集， pp.223-232, 1987

8) H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs," In *Proc. of the Second International Logic Programming Conference*, pp.127-138, 1984.

9) K. Ueda and K. Furukawa, Transformation Rules for GHC Programs, *Proc. of the International Conference on Fifth Generation Computer Systems*, Vol.2, pp.582-591, 1988