ICOT Technical Memorandum: TM-0731

TM-0731

完備化アルゴリズムの並列化と GHCによる実装

藤田　博

May, 1989

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

Institute for New Generation Computer Technology

完備化アルゴリズムの並列化と GHC による実装

藤田 博

(財) 新世代コンピュータ技術開発機構
東京都港区三田 1-4-28　三田国際ビル 21F
電話: 03-456-4365　電子メイル: fujita@icot.junet

概要

Knuth-Bendix の完備化アルゴリズムを並列化し，GHC で実装した.

1. はじめに

Knuth-Bendix アルゴリズム [1] は，与えられた等式の集合からこれと論理的に等価で完備な書き換え規則の集合を導くものである.

2 節では，Knuth-Bendix アルゴリズムを並列化した完備化アルゴリズムを与える．3 節では，並列完備化アルゴリズムの GHC による実現例を示す．4 節では，例題を与える.

2. 並列 Knuth-Bendix アルゴリズム

ここでは，Knuth-Bendix アルゴリズム [2] を並列化する．アルゴリズムは，等式の集合に対する処理と書き換え規則の集合に対する処理に大きく分けられるが，まず，これらが基本的に並列実行可能である.

このアルゴリズムは，パターンマッチ，簡約化 (reduction)，occur checktc つき同一化 (unification)，重像 (superposition)，などそれぞれ独立した並列性を有するアルゴリズム部品から構成されており，全体として多くの並列性を引き出せる可能性がある.

THE PARALLEL KNUTH-BENDIX COMPLETION ALGORITHM

Initial data: a (finite) set of equations $\mathcal{E}$, and a (recursive) reduction ordering $\succ$.

$\mathcal{Q}^{\mathcal{E}} := \mathcal{E}$; $\mathcal{Q}_0^{\mathcal{R}} := \phi$; $\mathcal{R}_0 := \phi$; $p := 0$;

do in parallel while the following 1 or 2 is applicable

1) *Reduce equation:*

Select equation $M = N$ in $\mathcal{Q}^{\mathcal{E}}$.

Let $M\downarrow$ (resp. $N\downarrow$) be an $\mathcal{R}_i$-normal form of M (resp. N) obtained by applying rules of $\mathcal{R}_i$ in any order, until none applies.

if $M\downarrow \succ N\downarrow$ **or** $N\downarrow \succ M\downarrow$ **then**

begin

if $M\downarrow \succ N\downarrow$ **then** $\lambda := M\downarrow$; $\rho := N\downarrow$ **else** $\lambda := N\downarrow$; $\rho := M\downarrow$ **endif**;

$p := p + 1$;

$\mathcal{R}_p := \mathcal{R}_{p-1} \cup \{p : \lambda \rightarrow \rho\}$;

$\mathcal{Q}_p^{\mathcal{R}} := \phi$;

Add $\lambda \rightarrow \rho$ into $\mathcal{Q}_0^{\mathcal{R}}$.

end

else if $M\downarrow = N\downarrow$ **then exitdo** (*failure*) **endif**;

2) *Reduce rule:*

Select rule $\lambda_j \rightarrow \rho_j$ in $\mathcal{Q}_i^{\mathcal{R}} (0 \leq i \leq p)$.

if λ_i is reducible to λ_i' by $\lambda_j \rightarrow \rho_j$ **then**

begin $\mathcal{R}_p := \mathcal{R}_p - \{i : \lambda_i \rightarrow \rho_i\}$; Add $\lambda_i' = \rho_i$ into $\mathcal{Q}^{\mathcal{E}}$ **end**

else

begin

if ρ_i is reducible to ρ_i' by $\lambda_j \rightarrow \rho_j$ **then**

begin

$\mathcal{R}_p := \mathcal{R}_p - \{i : \lambda_i \rightarrow \rho_i\} \cup \{i : \lambda_i \rightarrow \rho_i'\}$;

Let $\mathcal{CP}$ be all critical pairs computed between $\lambda_j \rightarrow \rho_j$ and $\lambda_i \rightarrow \rho_i'$

end

else Let $\mathcal{CP}$ be all critical pairs computed between $\lambda_j \rightarrow \rho_j$ and $\lambda_i \rightarrow \rho_i$ **endif**;

Add all the members in $\mathcal{CP}$ into $\mathcal{Q}^{\mathcal{E}}$

end;

if $i < p$ **then** Add $\lambda_j \rightarrow \rho_j$ into $\mathcal{Q}_{i+1}^{\mathcal{R}}$

enddo

($\mathcal{R}_p$ canonical)

3. GHC による実現

3.1 トップレベル

comp(Eqs,Res) は，等式の集合 Eqs を入力し，計算が終了したとき結果を Res に出力する．Res の値は，成功終了の場合は完備な書き換え規則の集合であり，失敗終了の場合は fail である．

プログラムは，主に等式を処理するプロセス eq_rule と書き換え規則を処理するプロセス rule_eq の二つから構成される．

eq_rule(ES,RS,_,_,_,_,_) は，ES を入力ストリーム，RS を出力ストリームとする．一方，rule_eq(RS,Feedback,_,_) は，RS を入力ストリーム，Feedback を出力ストリームとする．Feedback は，InitEqs と併合されて ES に戻される．こうして，eq_rule と rule_eq は，互いの入出力ストリームの結合によってループを形成している．

入力された等式集合の各等式には，スイッチ sw(A,Z) が埋め込まれる．これらのスイッチは全体でショートサーキットを形成するように結線される．このショートサーキットによって全体の処理の終了を知ることができる．

```
comp(Eqs,Res) :- true |
   init_eqs(Eqs,0,sw(Success,success),InitEqs),
   merge(InitEqs,Feedback,ES,Abort),
   eq_rule(ES,RS,0,[],FinalRls,OutS1,Abort),
   rule_eq(RS,Feedback,OutS2,Abort),
   merge_outs(OutS1,OutS2,OutS),
   outstream(OutS),
   comp_result(Success,Abort,FinalRls,Res).

   comp_result(_,abort(all),_,Res) :- true | Res=fail.
   comp_result(success,Abort,FinalRls,Res) :- true | Res=FinalRls,
      Abort=abort(success).

   init_eqs([(L=R)|Eqs],IDe,sw(A,Z),InitEqs) :- true |
      IDe1:=IDe+1,
      weight(L,0,WL), weight(R,0,WR), max(WL,WR,W),
      InitEqs=[eq(IDe1,(L=R),W,0,1,sw(A,Y))|InitEqs1],
      init_eqs(Eqs,IDe1,sw(Y,Z),InitEqs1).
   init_eqs([],_,sw(A,Z),InitEqs) :- true | A=Z, InitEqs=[].

   merge_outs([A-Z|S1],S2,S) :- true | S=A, merge_outs(S1,S2,Z).
   merge_outs(S1,[A-Z|S2],S) :- true | S=A, merge_outs(S1,S2,Z).
   merge_outs([],[],S) :- true | S=[].

   weight(var(_),W,W1) :- true | W1:=W+1.
   weight(T,W,W2) :- T\=var(_) | functor(T,_,N), W1:=W+1,
      weight_arg(N,T,W1,W2).

      weight_arg(N,T,W,W2) :- N>0 | N1:=N-1, arg(N,T,A),
         weight(A,W,W1), weight_arg(N1,T,W1,W2).
      weight_arg(0,_,W,W2) :- true | W=W2.

   max(N,M,Max) :- N<M | Max=M.
   max(N,M,Max) :- N>=M | Max=N.
```

Fig. 3.1 並列完備化プログラムのトップレベル

3.2 等式の処理

eq_rule(ES,RS,IDr,CurRls,FinalRls,_,Abort) は，打ち切り信号 abort(_) を受信した場合，部分完備化規則集合 CurRls を最終完備化規則集合 FinalRls に出力して，終了する．また，入力ストリーム ES に流れてくるデータに応じて以下の処理を行う．

i) 書き換え規則の変更指令 update_rule(IDr,Rule,SW) の場合，
部分完備化規則集合 CurRls を更新する．

ii) 等式 eq(IDe:Eq,W,Q,P,SW) の場合，
等式 Eq の両辺を部分完備化規則集合 CurRls によって簡約化した後，荷電 Q (-1,0) 及びパリティ P (1,0) に応じて以下の処理を行う．

ii-1) 荷電 Q が 0 パリティ P が 0 の等式の場合，
両辺が合流した場合はスイッチ SW を閉じた後，棄却する．合流はしないが簡約化された場合は，重み W を再評価し，荷電を +1 に変更して RS に出力する．簡約化されてない場合は，順序づけを試みる．順序づけが成立した場合は，その両辺から書き換え規則を生成して RS に出力すると同時に，部分完備化規則集合 CurRls に加える．

ii-2) 荷電 Q が 0 パリティ P が 1 の等式の場合，
両辺が合流した場合はスイッチ SW を閉じた後，棄却する．合流はしないが簡約化された場合は，重み W を再評価し，荷電を -1 に変更して RS に出力すると同時に，ES の前面に同名で反対荷電 Q=+1 の等式によるフィルタを生成する．

ii-3) 荷電 Q が -1 の等式の場合，
両辺が合流した場合は重み W を 0 にし，そうでない場合は再評価して RS に出力する．

```
eq_rule(_,_,_,CurRls,FinalRls,OutS,abort(_)) :- true |
   FinalRls=CurRls, OutS=[].
eq_rule([update_rule(IDr1,Rule,sw(A,Z))|ES],
        RS,IDr,CurRls,FinalRls,OutS,Abort) :- true | A=Z,
   update_rules(CurRls,IDr1,Rule,CurRls1),
   eq_rule(ES,RS,IDr,CurRls1,FinalRls,OutS,Abort).
eq_rule([eq(IDe,(L=R),W,Q,P,SW)|ES],
        RS,IDr,CurRls,FinalRls,OutS,Abort) :- true |
   reduce(L,CurRls,NewL,Yes,No-Ny),
   reduce(R,CurRls,NewR,Yes,Ny-no),
   eq_rule1(Yes,No,NewL,NewR,IDe,(L=R),W,Q,P,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,Abort),
   eq_rule(ES1,RS1,IDr1,CurRls1,FinalRls,OutS1,Abort).

   eq_rule1(_,_,R,R,IDe,_,_,0,_,sw(A,Z),ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- true |
      OutS=[[write('Reduced eq_'),write(IDe),write(' => '),
             write(R),nl|Oz]-Oz|OutS1],
      A=Z, ES=ES1, RS=RS1, IDr=IDr1, CurRls=CurRls1.
   eq_rule1(yes,_,L,R,IDe,_,_,0,0,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- L\=R |
      OutS=OutS1, ES=ES1, IDr=IDr1, CurRls=CurRls1,
      weight(L,0,WL), weight(R,0,WR), max(WL,WR,W),
      RS=[eq(IDe,(L=R),W,0,1,SW)|RS1].
   eq_rule1(_,no,L,R,IDe,_,_,0,0,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- L\=R |
      OutS=[[write('Select eq_'),write(IDe),nl|OutS2]-Oz|OutS1],
      ES=ES1,
      order(L,R,Order),
      new_rule(Order,L,R,SW,IDr,IDr1,RS,RS1,CurRls,CurRls1,
               OutS2-Oz,Abort).
   eq_rule1(yes,_,L,R,IDe,_,_,0,1,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- L\=R |
      OutS=OutS1, IDr=IDr1, CurRls=CurRls1,
      weight(L,0,WL), weight(R,0,WR), max(WL,WR,W),
      RS=[eq(IDe,(L=R),W,-1,0,SW)|RS1],
      eq_filter(ES,ES1,IDe,W,Abort).
   eq_rule1(_,no,L,R,IDe,Eq,W,0,1,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- L\=R |
      OutS=OutS1, IDr=IDr1, CurRls=CurRls1,
      RS=[eq(IDe,Eq,W,-1,0,SW)|RS1],
      eq_filter(ES,ES1,IDe,W,Abort).
   eq_rule1(_,_,R,R,IDe,_,_,-1,_,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- true |
      OutS=[[write('Reduced eq_'),write(IDe),write(' => '),
             write(R),nl|Oz]-Oz|OutS1],
      ES=ES1, IDr=IDr1, CurRls=CurRls1,
      RS=[eq(IDe,R,0,-1,0,SW)|RS1].
   eq_rule1(_,no,L,R,IDe,Eq,W,-1,_,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- L\=R |
      OutS=OutS1, ES=ES1, IDr=IDr1, CurRls=CurRls1,
      RS=[eq(IDe,Eq,W,-1,0,SW)|RS1].
   eq_rule1(yes,_,L,R,IDe,_,_,-1,_,SW,ES,ES1,RS,RS1,
            IDr,IDr1,CurRls,CurRls1,OutS,OutS1,_) :- L\=R |
      OutS=OutS1, ES=ES1, IDr=IDr1, CurRls=CurRls1,
      weight(L,0,WL), weight(R,0,WR), max(WL,WR,W),
      RS=[eq(IDe,(L=R),W,-1,0,SW)|RS1].
```

Fig. 3.2.1 等式の処理 (I)

eq_filter(ES,ES1,IDe,W,Abort)は，打ち切り信号 abort(_) を受信した場合，終了する．また，入力ストリーム ES に流れてくるデータに応じて以下の処理を行う．

i) 荷電 Q が 0 で重み W1 が W より小さい等式の場合，
そのまま ES1 に出力する．

ii) 荷電 Q が 0 で重み W1 が W より大きいか等しい等式の場合，
荷電 Q を -1 に変更し，パリティ P を 0 にして RS に出力すると同時に，ES の前面に同名で反対荷電 Q=+1 の等式によるフィルタを生成する．

iii) 異名で荷電 Q が -1 の等式の場合，
そのまま ES1 に出力する．

iv) 同名で荷電 Q が -1 の等式の場合，
荷電 Q を 0 に変更し，パリティ P を 0 にして RS に出力すると同時に，ここのフィルタを終了する．

v) 同名で重み W が 0 の等式の場合，
等式に付随するスイッチ SW を閉じ，ここのフィルタを終了する．

vi) 書き換え規則の変更指令 update_rule(IDr,Rule,SW) の場合，
そのまま ES1 に出力する．

```
eq_filter(_,_,_,_,abort(_)) :- true | true.
eq_filter([eq(IDe1,Eq,W1,0,P,SW)|ES],ES1,IDe,W,Abort) :- W1<W |
   ES1=[eq(IDe1,Eq,W1,0,P,SW)|ES2],
   eq_filter(ES,ES2,IDe,W,Abort).
eq_filter([eq(IDe1,Eq,W1,0,P,SW)|ES],ES1,IDe,W,Abort) :- W1>=W |
   ES1=[eq(IDe1,Eq,W1,-1,0,SW)|ES2],
   eq_filter(ES,ES3,IDe1,W1,Abort),
   eq_filter(ES3,ES2,IDe,W,Abort).
eq_filter([eq(IDe1,Eq,W1,-1,P,SW)|ES],ES1,IDe,W,Abort) :- IDe1\=IDe |
   ES1=[eq(IDe1,Eq,W1,-1,P,SW)|ES2],
   eq_filter(ES,ES2,IDe,W,Abort).
eq_filter([eq(IDe,Eq,W,-1,_,SW)|ES],ES1,IDe,_,_) :- W>0 |
   ES1=[eq(IDe,Eq,W,0,0,SW)|ES].
eq_filter([eq(IDe,_,0,_,_,sw(A,Z))|ES],ES1,IDe,_,_) :- true |
   A=Z, ES1=ES.
eq_filter([update_rule(IDr,Rule,SW)|ES],ES1,IDe,W,Abort) :- true |
   ES1=[update_rule(IDr,Rule,SW)|ES2],
   eq_filter(ES,ES2,IDe,W,Abort).

rename(var(V),D,NewV,Vi,Vo) :- true | rename_var(Vi,0,D,var(V),NewV,Vo).
rename(T,D,NewT,Vi,Vo) :- T\=var(_) | functor(T,F,N), rotcnuf(NewT,F,N),
   rename_arg(N,T,D,NewT,Vi,Vo).

   rename_arg(N,T,D,NewT,Vi,Vo) :- N>0 |
      N1:=N-1, arg(N,T,A), arg(N,NewT,B),
      rename(A,D,B,Vi,Vm), rename_arg(N1,T,D,NewT,Vm,Vo).
   rename_arg(0,_,_,_,Vi,Vo) :- true | Vo=Vi.

   rename_var([var(V)=NV|Vi],_,_,var(V),NewV,Vo) :- true |
      Vo=[var(V)=NV|Vi], NewV=NV.
   rename_var([var(U)=NU|Vi],I,D,var(V),NewV,Vo) :- V\=U |
      Vo=[var(U)=NU|Vo1], I1:=I+D,
      rename_var(Vi,D,I1,var(V),NewV,Vo1).
   rename_var([],I,D,var(V),NewV,Vo) :- true |
      Vo=[var(V)=NewV], I1:=I+D, NewV=var(I1).
```

Fig. 3.2.2 等式の処理 (II)

```
new_rule((>),L,R,SW,IDr,IDr1,RS,RS1,CurRls,CurRls1,OutS-Oz,_) :- true |
   IDr1:=IDr+1,
   rename((L->R),1,NewRule,[],_),
   RS=[new_rule(IDr1,(L->R),NewRule,SW)|RS1],
   OutS=[write('Add rule_'),write(IDr1),write(' : '),write(NewRule),nl|Oz],
   append(CurRls,[rule(IDr1,NewRule)],CurRls1).
new_rule((<),R,L,SW,IDr,IDr1,RS,RS1,CurRls,CurRls1,OutS-Oz,_) :- true |
   IDr1:=IDr+1,
   rename((L->R),1,NewRule,[],_),
   RS=[new_rule(IDr1,(L->R),NewRule,SW)|RS1],
   OutS=[write('Add rule_'),write(IDr1),write(' '),write(NewRule),nl|Oz],
   append(CurRls,[rule(IDr1,NewRule)],CurRls1).
new_rule((=),_,_,sw(A,Z),IDr,IDr1,RS,RS1,CurRls,CurRls1,
         OutS-Oz,_) :- true |
   A=Z, IDr=IDr1, RS=RS1, CurRls=CurRls1, OutS=Oz.
new_rule((?),L,R,_,_,_,_,_,_,_,OutS-Oz,Abort) :- true |
   OutS=[nl,write('!!! FAILURE !!!'),nl,nl,write(L),nl,write('and'),nl,
         write(R),nl,write('are not comparable.'),nl|Oz],
   Abort=abort(all).

update_rules([rule(IDr,Rule)|Rls],IDr1,Rule1,NewRls) :- IDr\=IDr1 |
   NewRls=[rule(IDr,Rule)|NewRls1],
   update_rules(Rls,IDr1,Rule1,NewRls1).
update_rules([rule(IDr,_)|Rls],IDr,NewRule,NewRls) :- NewRule\=delete |
   NewRls=[rule(IDr,NewRule)|Rls].
update_rules([rule(IDr,Rule)|Rls],IDr,delete,NewRls) :- true |
   NewRls=Rls.
```

Fig. 3.2.3 書き換え規則の生成 / 更新

3.3 書き換え規則の処理

rule_eq(RS,Feedback,_,Abort) は，打ち切り信号 abort(_) を受信した場合，終了する．また，入力ストリーム RS に流れてくるデータに応じて以下の処理を行う．

i) 等式 eq(IDe,Eq,W,Q,P,SW) の場合，
そのまま Feedback に出力する．

ii) 新規の書き換え規則 new_rule(IDr,Rule,NewRule,SW) の場合，
ヴァリアント Rule と NewRule の間で重像を計算してその結果の要対を Feedback に出力すると同時に，後続の書き換え規則による変更，および重像の計算を行うためのプロセス rule_filter を入力ストリーム RS の前面に生成する．

iii) 書き換え規則の更新指令 update_rule(IDr,Rule,SW) の場合，
そのまま Feedback に出力する．

```
rule_eq(_,_,OutS,abort(_)) :- true | OutS=[].
rule_eq([eq(IDe,Eq,W,Q,P,SW)|RS],Feedback,OutS,Abort) :- true |
   Feedback=[eq(IDe,Eq,W,Q,P,SW)|Feedback1],
   rule_eq(RS,Feedback1,OutS,Abort).
rule_eq([new_rule(IDr,(L->R),Rule,SW)|RS],Feedback,OutS,Abort) :- true |
   OutS1=[[write('Superpose '),write(IDr),write(' on '),write(IDr),nl
           |OutS1r]-Oz],
   functor(L,_,N),
   superpose_arg(N,L,[],rule(IDr,(L->R)),rule(IDr,Rule),SW,CPs),
   output_eqs(CPs,NewEqs,OutS1r-Oz),
   merge(NewEqs,Feedback1,Feedback,Abort),
   merge(OutS1,OutS2,OutS12,_),
   merge(OutS12,OutS3,OutS,_),
   rule_filter(RS,RS1,IDr,Rule,OutS2,Abort),
   rule_eq(RS1,Feedback1,OutS3,Abort).
rule_eq([update_rule(IDr,Rule,SW)|RS],Feedback,OutS,Abort) :- true |
   Feedback=[update_rule(IDr,Rule,SW)|Feedback1],
   rule_eq(RS,Feedback1,OutS,Abort).

rule_filter(_,_,_,_,_,OutS,abort(_)) :- true | OutS=[].
rule_filter([eq(IDe,Eq,0,Q,P,SW)|RS],RS1,IDr,Rule,OutS,Abort) :- true |
   RS1=[eq(IDe,Eq,0,Q,P,SW)|RS2],
   rule_filter(RS,RS2,IDr,Rule,OutS,Abort).
rule_filter([eq(IDe,(L=R),W,Q,P,SW)|RS],RS1,IDr,Rule,OutS,Abort) :- W>0 |
   rewrite1(L,Rule,NewL,Yes,No-Ny),
   rewrite1(R,Rule,NewR,Yes,Ny-no),
   update_eq(Yes,No,NewL,NewR,IDe,(L=R),W,Q,P,SW,RS1,RS2,OutS,OutS1),
   rule_filter(RS,RS2,IDr,Rule,OutS1,Abort).
rule_filter([new_rule(IDr1,Rule,NewRule,sw(A,Z))|RS],RS1,
            IDr,(OldL->OldR),OutS,Abort) :- true |
   RS1=[new_rule(IDr1,Rule,NewRule,sw(A,Y))|RS2],
   rewrite1(OldL,Rule,NewL,Yes,No-no),
   rule_filter1(Yes,No,IDr,(NewL=OldR),IDr1,Rule,sw(Y,Z),RS,RS2,OutS,Abort).
rule_filter([update_rule(IDr1,Rule1,SW)|RS],RS1,
            IDr,Rule,OutS,Abort) :- true |
   RS1=[update_rule(IDr1,Rule1,SW)|RS2],
   rule_filter(RS,RS2,IDr,Rule,OutS,Abort).
```

Fig. 3.3.1 書き換え規則の処理 (I)

rule_filter(RS,RS1,IDr,Rule,_,Abort) は，打ち切り信号 abort(_) を受信した場合，終了する．また，入力ストリーム RS に流れてくるデータに応じて以下の処理を行う．

i) 重み W が 0 の等式 eq(IDr,Eq,0,Q,P,SW) の場合，
そのまま RS1 に出力する．

ii) 重み W が 0 でない等式 eq(IDe,Eq,W,Q,P,SW) の場合，
Rule で書き換える．その結果，

ii-1) 荷電 Q が 0 のもので両辺が合流したものは棄却する．

ii-2) 荷電 Q が -1 のもので両辺が合流したものは，重み W を 0 に変更して RS1 に出力する．

ii-3) 書き換えが行われず両辺が合流していないものは，そのまま RS1 に出力する．

ii-4) 両辺が合流しなかったが，簡約化が行われたものは，重み W を再評価して RS1 に出力する．

iii) 新規の書き換え規則 new_rule(IDr,Rule1,NewRule,SW)
の場合，

iii-1) Rule1 が Rule の左辺を書き換えて，両辺を合流させる場合，
Rule の棄却指令を RS1 に出力する．

iii-2) Rule1 が Rule の左辺を書き換えるが，両辺が合流しない場合，
Rule の棄却指令，およびその両辺から荷電が 0 でパリティが 1 の等式を生成して RS1 に出力する．

iii-3) Rule1 が Rule の左辺を書き換えない場合，
右辺を書き換えるときは Rule の変更指令を RS1 に出力する．また，Rule1 と Rule の間で重像を計算してその結果の要対を RS に出力する．

iv) 書き換え規則の更新指令 update_rule(RlID:Rule,SW) の場合，
そのまま Feedback に出力する．

```
rule_filter1(yes,_,IDr,(R=R),_,_,SW,RS,RS1,OutS,_) :- true |
   OutS=[[write('Reduced rule_'),write(IDr),write(' => '),
          write(R),nl|Oz]-Oz],
   RS1=[update_rule(IDr,delete,SW)|RS].
rule_filter1(yes,_,IDr,(L=R),IDr1,_,sw(A,Z),RS,RS1,OutS,_) :- L\=R |
   OutS=[[write('Reduced rule_'),write(IDr),write(' => '),
          write(NewEq),nl|Oz]-Oz],
   weight(L,0,WL), weight(R,0,WR), max(WL,WR,W),
   rename((L=R),-1,NewEq,[],_),
   RS1=[update_rule(IDr,delete,sw(A,Y)),
        eq((IDr/IDr1),NewEq,W,0,1,sw(Y,Z))|RS].
rule_filter1(_,no,IDr,(OldL=OldR),IDr1,Rule,sw(A,Z),
             RS,RS1,OutS,Abort) :- true |
   rewrite1(OldR,Rule,NewR,Yes,No-no),
   update_rule(Yes,No,IDr,(OldL->NewR),sw(A,Y),RS1,RS2),
   rule_filter2(IDr1,Rule,sw(Y,Z),IDr,(OldL->NewR),RS,RS2,OutS,Abort).

rule_filter2(IDr1,(L1->R1),sw(A,Z),IDr,(L->R),
             RS,RS1,OutS,Abort) :- true |
   OutS=[[write('Superpose '),write(IDr),write(' on '),write(IDr1),nl
          |OutS1]-Oz|OutS2],
   superpose(L1,[],R,rule(IDr1,(L1->R1)),rule(IDr,(L->R)),sw(A,Y),CPs1),
   superpose(L,[],R1,rule(IDr,(L->R)),rule(IDr1,(L1->R1)),sw(Y,Z),CPs2),
   merge(CPs1,CPs2,CPs,Abort),
   output_eqs(CPs,NewEqs,OutS1-Oz),
   merge(NewEqs,RS2,RS1,Abort),
   rule_filter(RS,RS2,IDr,(L->R),OutS2,Abort).

   update_eq(_,_,R,R,IDe,_,_,0,_,sw(A,Z),RS,RS1,OutS,OutS1) :- true |
      OutS=[[write('Reduced eq_'),write(IDe),write(' => '),
             write(R),nl|Oz]-Oz|OutS1],
      A=Z, RS=RS1.
   update_eq(_,_,R,R,IDe,Eq,_,-1,_,SW,RS,RS1,OutS,OutS1) :- true |
      OutS=[[write('Reduced eq_'),write(IDe),write(' => '),
             write(R),nl|Oz]-Oz|OutS1],
      RS=[eq(IDe,Eq,0,-1,0,SW)|RS1].
   update_eq(_,no,L,R,IDe,Eq,W,Q,P,SW,RS,RS1,OutS,OutS1) :- L\=R |
      OutS=OutS1,
      RS=[eq(IDe,Eq,W,Q,P,SW)|RS1].
   update_eq(yes,_,L,R,IDe,_,_,Q,P,SW,RS,RS1,OutS,OutS1) :- L\=R |
      OutS=OutS1,
      weight(L,0,WL), weight(R,0,WR), max(WL,WR,W),
      RS=[eq(IDe,(L=R),W,Q,P,SW)|RS1].

   update_rule(_,no,_,_,sw(A,Z),RS,RS1) :- true | A=Z, RS=RS1.
   update_rule(yes,_,IDr,Rule,SW,RS,RS1) :- true |
      RS=[update_rule(IDr,Rule,SW)|RS1].

   output_eqs([eq(IDe,Eq,W,Q,P,SW)|Eqs],NewEqs,OutS-Oz) :- true |
      OutS=[write('   '),write(IDe),write(' : '),write(Eq),nl|Oy],
      rename(Eq,-1,NewEq,[],_),
      NewEqs=[eq(IDe,NewEq,W,Q,P,SW)|NewEqs1],
      output_eqs(Eqs,NewEqs1,Oy-Oz).
   output_eqs([],NewEqs,OutS-Oz) :- true | OutS=Oz, NewEqs=[].
```

Fig. 3.3.2 書き換え規則の処理 (II)

3.4 簡約化

reduce(T,Rls,NewT,Yes,No-Nz) は，入力項 T を書き換え規則の集合 Rls で簡約化し，結果 (normal form) を NewT に返す．T が簡約化可能 (T≠NewT) である場合，Yes は yes と，T が Rls に関して既に normal form である場合，No は Nz と同一化される．

rewrite(T,Rls,NewT,Yes,No-Nz) は，入力項 T を書き換え規則の集合 Rls で (一回限りの) 書き換えを試み，結果を NewT に返す．T が一回書き換えられれば，Yes は yes と，T が Rls に関して既に normal form である場合，No は Nz と同一化される．

rewrite(T,Rule,NewT,Yes,No-Nz) は，入力項 T を一つの書き換え規則 Rule で (一回限りの) 書き換えを試み，結果を NewT に返す．T が一回書き換えられれば，Yes は yes と，書き換えられなければ，No は Nz と同一化される．

substitute(T,Subst,NewT) は，入力項 T に変数置換 Subst を適用し，結果を NewT に返す．

```
reduce(var(V),_,NewT,_,No-Nz) :- true | NewT=var(V), No=Nz.
reduce(T,Rls,NewT,Yes,No) :- T\=var(_) |
   rewrite(T,Rls,T1,Y,N-no),
   reduce_result(Y,N,T1,Rls,NewT,Yes,No).

   reduce_result(_,no,T1,_,NewT,_,No-Nz) :- true | NewT=T1, No=Nz.
   reduce_result(yes,_,T1,Rls,NewT,Yes,_) :- true | Yes=yes,
      reduce(T1,Rls,NewT,_,_-_).

rewrite(T,[rule(_,Rule)|Rls],NewT,Yes,No) :- true |
   rewrite1(T,Rule,T1,Y,N-no),
   rewrite_result(Y,N,T1,Rls,NewT,Yes,No).
rewrite(T,[],NewT,_,No-Nz) :- true | NewT=T, No=Nz.

   rewrite_result(yes,_,T1,_,NewT,Yes,_) :- true | NewT=T1, Yes=yes.
   rewrite_result(_,no,T1,Rls,NewT,Yes,No) :- true |
      rewrite(T1,Rls,NewT,Yes,No).

rewrite1(var(V),_,NewT,_,No-Nz) :- true | NewT=var(V), No=Nz.
rewrite1(T,(Left->Right),NewT,Yes,No) :- true |
   match(Left,T,Res),
   rewrite1_result(Res,T,(Left->Right),NewT,Yes,No).

   rewrite1_result(Subst,_,(_->Right),NewT,Yes,_) :- Subst\=fail |
      substitute(Right,Subst,NewT), Yes=yes.
   rewrite1_result(fail,T,Rule,NewT,Yes,No) :- true |
      functor(T,F,N), rotcnuf(NewT,F,N),
      rewrite_arg(N,T,Rule,NewT,Yes,No).

      rewrite_arg(N,T,Rule,NewT,Yes,No-Nz) :- N>0 |
         N1:=N-1, arg(N,T,A), arg(N,NewT,B),
         rewrite1(A,Rule,B,Yes,No-Na),
         rewrite_arg(N1,T,Rule,NewT,Yes,Na-Nz).
      rewrite_arg(0,_,_,_,_,No-Nz) :- true | No=Nz.

substitute(X,[],Y) :- true | Y=X.
substitute(var(V),Subst,Y) :- Subst\=[] | substitute_var(Subst,var(V),Y).
substitute(X,Subst,Y) :- X\=var(_) | functor(X,F,N), rotcnuf(Y,F,N),
   substitute_arg(N,X,Subst,Y).

   substitute_arg(N,X,Subst,Y) :- N>0 | N1:=N-1, arg(N,X,A), arg(N,Y,B),
      substitute(A,Subst,B), substitute_arg(N1,X,Subst,Y).
   substitute_arg(0,_,_,_) :- true | true.

   substitute_var([V=T|_],     V,Out) :- true | Out=T.
   substitute_var([U=_|Subst],V,Out) :- U\=V | substitute_var(Subst,V,Out).
   substitute_var([],          V,Out) :- true | Out=V.
```

Fig. 3.4.1 簡約化

match(Pat,Tar,Res) は，パターン Pat を，ターゲット Tar にマッチさせることを試みる．成功すれば変数置換が，失敗すれば fail が Res に返される．

match(Pat,Tar,SW,OutBS,Abort) は，打ち切り信号 abort(_) を受信した場合，終了する．また，Pat と Tar に応じて以下の処理を行う．

i) Pat が変数でなく，Tar が変数の場合，
   マッチングは失敗し，打ち切り信号 abort(_) を発信する．

ii) Pat が変数の場合，
   OutBS に変数束縛 Pat=Tar を出力する．

iii) Pat も Tar も変数でない場合，
   主要構造を比較する．一致すれば，部分構造ごとにマッチングをとる．

match_sift(InBS,OutBS,Abort) は，打ち切り信号 abort(_) を受信した場合，終了する．また，InBS から変数 V の束縛を入力した場合，OutBS に出力すると同時，この束縛に付随しているスイッチを閉じ，変数 V によるフィルタを生成する．

match_filter(InBS,V,T,OutBS,Abort) は，打ち切り信号 abort(_) を受信した場合，終了する．また，InBS に流れてきた変数 V1 の束縛 T1 に応じて以下の処理を行う．

i) V1 が V に一致し，T1 が T と異なる場合，
   マッチングは失敗し，打ち切り信号 abort(_) を発信する．

ii) V1=T1 が V=T と一致する場合，
   変数束縛に付随しているスイッチを閉じる．

iii) V1 が V と異なる場合，
   そのまま OutBS に出力する．

```
match(Pat,Tar,Res) :- true |
   match(Pat,Tar,BS,sw(Success,success),Abort),
   match_sift(BS,OutBS,Abort),
   match_result(Success,Abort,OutBS,Res).

   match_result(_,abort(_),_,Res) :- true | Res=fail.
   match_result(success,Abort,OutBS,Res) :- true | Res=OutBS,
                        Abort=abort(_).

match(_,_,_,_,abort(_)) :- true | true.
match(T,V,_,_,Abort) :- V=var(_), T\=var(_) | Abort=abort(_).
match(V,T,OutBS,SW,_) :- V=var(_) | OutBS=[bind(V,T,SW)].
match(Pat,Tar,OutBS,SW,Abort) :- Pat\=var(_), Tar\=var(_) |
   functor(Pat,F,N), functor(Tar,G,M),
   match_functor(F/N,G/M,Pat=Tar,OutBS,SW,Abort).

   match_functor(F_N,G_M,_,_,_,Abort) :- F_N\=G_M | Abort=abort(_).
   match_functor(F/N,F/N,Pat_Tar,OutBS,SW,Abort) :- true |
      match_arg(N,Pat_Tar,OutBS,SW,Abort).

      match_arg(_,_,_,_,abort(_)) :- true | true.
      match_arg(N,Pat=Tar,OutBS,sw(A,Z),Abort) :- N>0 |
         N1:=N-1, arg(N,Pat,Pat1), arg(N,Tar,Tar1),
         match(Pat1,Tar1,BS1,sw(A,Y),Abort),
         merge(BS1,BS2,OutBS,Abort),
         match_arg(N1,Pat=Tar,BS2,sw(Y,Z),Abort).
      match_arg(0,_,OutBS,sw(A,Z),_) :- true | OutBS=[], A=Z.

match_sift(_,OutBS,abort(_)) :- true | OutBS=[].
match_sift([bind(V,T,sw(A,Z))|InBS],OutBS,Abort) :- true | A=Z,
   OutBS=[V=T|OutBS1],
   match_filter(InBS,V,T,InBS1,Abort),
   match_sift(InBS1,OutBS1,Abort).

match_filter(_,_,_,_,abort(_)) :- true | true.
match_filter([bind(V,T1,SW)|InBS],V,T,_,Abort) :- T1\=T | Abort=abort(_).
match_filter([bind(V,T,sw(A,Z))|InBS],V,T,OutBS,Abort) :- true | A=Z,
   match_filter(InBS,V,T,OutBS,Abort).
match_filter([bind(U,T1,SW)|InBS],V,T,OutBS,Abort) :- U\=V |
   OutBS=[bind(U,T1,SW)|OutBS1],
   match_filter(InBS,V,T,OutBS1,Abort).
```

Fig. 3.4.2 マッチング

3.5 重像

superpose(L1sub,Pos,L1_R2,Rule1,Rule2,SW,CPs) は，Rule1 の左辺の Pos で指定される部分項 L1sub1 に Rule2 の左辺を重ね合わせる．L1_R2 は Rule1 の左辺の Pos で指定される部分項 L1sub1 を既に Rule2 の右辺で置き換えたものとする．

superpose_arg(N,L1sub,Pos,Rule1,Rule2,SW,CPs) は，Rule1 の左辺の Pos で指定される部分項 L1sub のさらに N の位置の部分項において Rule2 の左辺を重ね合わせる．

重ね合わせが成立する場合には，要対が生成されて CPs に出力される．要対として得られた等式の荷電は 0 で，パリティは +1 である．この等式の ID は，重ね合わされた Rule1 と Rule2 の ID と重ね合わせの行われた位置 Pos とから一意に決められる．また，重み W が評価され，SW が埋め込まれる．

locate(Pos,L1,L1sub,L1_R2,R2) は，項 L1 の Pos で指定される位置の部分項 L1sub を同定すると同時に，L1 のコピーで L1sub の部分のみを項 R2 で置き換えた項 L1_R2 を生成する．

```
superpose(var(_),_,_,_,_,sw(A,Z),CPs) :- true | A=Z, CPs=[].
superpose(L1sub,Pos,L1_R2,Rule1,Rule2,sw(A,Z),CPs) :- L1sub\=var(_) |
   Rule1=rule(ID1,(_->R1)),
   Rule2=rule(ID2,(L2->_)),
   unify(L1sub,L2,MGU),
   superpose_result(MGU,L1_R2,R1,sw(A,B),ID1,ID2,Pos,CPs1),
   merge(CPs1,CPs2,CPs,_),
   functor(L1sub,_,N),
   superpose_arg(N,L1sub,Pos,Rule1,Rule2,sw(B,Z),CPs2).

   superpose_result(fail,_,_,sw(A,Z),_,_,_,CPs) :- true | A=Z, CPs=[].
   superpose_result(MGU,L1_R2,R1,SW,ID1,ID2,Pos,CPs) :- MGU\=fail |
      substitute(L1_R2,MGU,NewL1),
      substitute(R1,MGU,NewR1),
      weight(NewL1,0,WL), weight(NewR1,0,WR), max(WL,WR,W),
      CPs=[eq((ID1*Pos+ID2),(NewL1=NewR1),W,0,1,SW)].

   superpose_arg(_,var(_),_,_,_,sw(A,Z),CPs) :- true | A=Z, CPs=[].
   superpose_arg(N,L1sub,Pos,Rule1,Rule2,sw(A,Z),CPs) :-
         L1sub\=var(_), N>0 | N1:=N-1,
      append(Pos,[N],NewPos),
      Rule1=rule(_,(L1->_)),
      Rule2=rule(_,(_->R2)),
      locate(NewPos,L1,NewL1sub,L1_R2,R2),
      superpose(NewL1sub,NewPos,L1_R2,Rule1,Rule2,sw(A,B),CPs1),
      merge(CPs1,CPs2,CPs,_),
      superpose_arg(N1,L1sub,Pos,Rule1,Rule2,sw(B,Z),CPs2).
   superpose_arg(0,_,_,_,_,sw(A,Z),CPs) :- true | A=Z, CPs=[].

locate([I|Pos],L1,L1sub,L1_R2,R2) :- true |
   functor(L1,F,N), rotcnuf(L1_R2,F,N),
   locate_arg(N,I,Pos,L1,L1sub,L1_R2,R2).

   locate_arg(N,I,Pos,L1,L1sub,L1_R2,R2) :- N>0, N\=I |
      N1:=N-1, arg(N,L1,L1_N), arg(N,L1_R2,L1_N),
      locate_arg(N1,I,Pos,L1,L1sub,L1_R2,R2).
   locate_arg(N,I,[],L1,L1sub,L1_R2,R2) :- N=I |
      N1:=N-1, arg(N,L1,L1sub), arg(N,L1_R2,R2),
      locate_arg(N1,I,[],L1,L1sub,L1_R2,_).
   locate_arg(N,I,Pos,L1,L1sub,L1_R2,R2) :- N=I, Pos\=[] |
      N1:=N-1, arg(N,L1,L1arg), arg(N,L1_R2,L1_R2arg),
      locate(Pos,L1arg,L1sub,L1_R2arg,R2),
      locate_arg(N1,I,Pos,L1,L1sub,L1_R2,R2).
   locate_arg(0,_,_,_,_,_,_) :- true | true.
```

Fig. 3.5 重像 (superposition)

3.6 同一化

unify(S,T,Res) は，二つの入力項 S と T の同一化を試みる．成功したときには mgu が，失敗したときには fail が結果として Res に返される．

unify(S,T,OutBS,SW,Abort) は，S と T の同一化の結果生じる変数置換をストリーム OutBS に出力する．

S と T が既に同一であるとき，出力ストリームを閉じると同時に，スイッチ SW を短絡する．

S と T のいずれか一方が変数のとき，変数置換を

OutBS=[bind(var(V),Term,Term,FinalTerm,Vars,SW)]

のように出力する．ここで，Term は変数 var(V) に対する置換項，FinalTerm は全体の同一化が成功した時点で定まる予定の (dereference 後の) 置換項 (の予約席) である．Vars は，Term 中に現れる変数の集合である．ただし，Term 自身が変数の場合はこれを含まない．Vars が既に var(V) を含んでいるときは，(occur check 付き) 同一化は失敗である．

また，この変数置換が出力される場合にはスイッチ SW を埋め込んでおく．埋め込まれたスイッチは，あとの処理でこの変数置換が確定した時点で短絡されることになる．

S と T が同一でなく，かついずれも変数でないときは，構造を分解して部分項ごとに同一化を試みる．

unify_functor は，両項の主要構造 (primary functor の名前と arity) を見て，同一化可能性を判定する．もしこの時点で同一化不可能と判れば，打ち切り端子 Abort に打ち切り信号 abort(all) を発信する．この信号は，全体の同一化の失敗を示し，ほかのプロセスに即時打ち切りを促すためのものである．主要構造が一致した場合は，unify_arg において，各部分構造について unify/5 を起動する．各部分同一化の出力ストリームは，併合されてルートの出力ストリーム OutBS に一本化される．

```
unify(S,T,Res) :- true |
   unify(S,T,BS0,sw(Success,success),Abort),
   merge(BS0,Feedback,BS1,Abort),
   sift(BS1,OutBS,Feedback,Abort),
   unify_result(Success,Abort,OutBS,Res).

   unify_result(_,abort(all),_,Res) :- true | Res=fail.
   unify_result(success,Abort,OutBS,Res) :- true | Res=OutBS,
      Abort=abort(filters).

unify(_,_,_,_,abort(_)) :- true | true.
unify(T,T,OutBS,sw(A,Z),_) :- true | OutBS=[], A=Z.
unify(var(V),var(U),OutBS,SW,_) :- V\=U |
   OutBS=[bind(var(V),var(U),var(U),Tf,[],SW)].
unify(V,T,OutBS,SW,Abort) :- V=var(_), T\=var(_) |
   vars_in(T,[],Vars), member(Vars,V,Ans),
   safe_bind(Ans,Vars,V,T,OutBS,SW,Abort).
unify(T,V,OutBS,SW,Abort) :- V=var(_), T\=var(_) |
   vars_in(T,[],Vars), member(Vars,V,Ans),
   safe_bind(Ans,Vars,V,T,OutBS,SW,Abort).
unify(S,T,OutBS,SW,Abort) :- S\=var(_), T\=var(_) |
   functor(S,F,N), functor(T,G,M),
   unify_functor(F/N,G/M,S=T,OutBS,SW,Abort).

   unify_functor(_,_,_,_,_,abort(_)) :- true | true.
   unify_functor(F_N,G_M,_,_,_,Abort) :- F_N\=G_M | Abort=abort(all).
   unify_functor(F/N,F/N,S_T,OutBS,SW,Abort) :- true |
      unify_arg(N,S_T,OutBS,SW,Abort).

      unify_arg(_,_,_,_,abort(_)) :- true | true.
      unify_arg(N,S=T,OutBS,sw(A,Z),Abort) :- N>0 |
         N1:=N-1, arg(N,S,SN), arg(N,T,TN),
         unify(SN,TN,BS1,sw(A,Y),Abort),
         merge(BS1,BS2,OutBS,Abort),
         unify_arg(N1,S=T,BS2,sw(Y,Z),Abort).
      unify_arg(0,_,OutBS,sw(A,Z),_) :- true | OutBS=[], A=Z.

   safe_bind(yes,_,_,_,_,_,Abort) :- true | Abort=abort(all).
   safe_bind(no,Vars,V,T,OutBS,SW,_) :- true |
      OutBS=[bind(V,T,T,Tf,Vars,SW)].
```

Fig. 3.6.1 同一化 (I)

sift(InBS,OutBS,Feedback,Abort) の入力ストリーム InBS に流れてくる変数置換は，主出力ストリーム OutBS に送り出される．それと同時にその変数で後続の変数置換をふるいにかけるためにfilter プロセスを生成する．副出力ストリーム Feedback は，filter がその処理過程で新たに生成する変数置換を回収するためのものである．

filter(InBS,V,IniT,CurT,FinT,Vars,OutBS,Feedback,Abort) は，変数 V で InBS に流れてくる変数置換をふるいにかける．

流れてきたのが同じ V に対する置換であるとき，この filter で保持している最終置換値 FinT と今入力したものの最終置換値 FinT1 とを同一化しておく．

流れてきたのが V と異なる変数に対するものであるとき，この二つの変数置換を互いに相手の中間置換値に適用させる (deref)．

なお，filter プロセスは入力項に含まれる変数のそれぞれにつき高々一つしか生成されない．しかも変数置換の全ての対についての相互比較および dereference が，ただ一回ずつ試みられるようになっている．

```
sift(_,OutBS,_,abort(_)) :- true | OutBS=[].
sift([bind(V,Ti,Tc,Tf,Vars,sw(A,Z))|InBS],OutBS,Feedback,Abort) :- true |
   A=Z,
   OutBS=[V=Tf|OutBS1],
   filter(InBS,V,Ti,Tc,Tf,Vars,InBS1,Feedback1,Abort),
   merge(Feedback1,Feedback2,Feedback,Abort),
   sift(InBS1,OutBS1,Feedback2,Abort).

filter(_,_,_,VTc,VTf,_,_,_,_,abort(_)) :- true | VTf=VTc.
filter([bind(V,VTi1,_,VTf1,VVars1,SW)|InBS],
       V,VTi,VTc,VTf,VVars,OutBS,Feedback,Abort) :- true |
   VTf1=VTf,
   unify(VTi1,VTi,Feedback1,SW,Abort),
   merge(Feedback1,Feedback2,Feedback,Abort),
   append(VVars1,VVars,VVars2),
   filter(InBS,V,VTi,VTc,VTf,VVars2,OutBS,Feedback2,Abort).
filter([bind(U,V,_,UTf,UVars,sw(A,Z))|InBS],
       V,VTi,VTc,VTf,VVars,OutBS,Feedback,Abort) :- true |
   UTf=VTf,
   member([VTi|VVars],U,HasU),
   update_vars(yes,U,UVars,VVars,UVars1,sw(A,X),Abort),
   update_vars(HasU,V,VVars,UVars,VVars1,sw(X,Y),Abort),
   OutBS=[bind(U,VTi,VTc,VTf,UVars1,sw(Y,Z))|OutBS1],
   filter(InBS,V,VTi,VTc,VTf,VVars1,OutBS1,Feedback,Abort).
filter([bind(U,UTi,UTc,UTf,UVars,sw(A,Z))|InBS],
       V,VTi,VTc,VTf,VVars,OutBS,Feedback,Abort) :- U\=V, UTi\=V |
   deref(UTc,V=VTf,UTc1,Abort),
   deref(VTc,U=UTf,VTc1,Abort),
   member([UTi|UVars],V,HasV),
   member([VTi|VVars],U,HasU),
   update_vars(HasV,U,UVars,VVars,UVars1,sw(A,X),Abort),
   update_vars(HasU,V,VVars,UVars,VVars1,sw(X,Y),Abort),
   OutBS=[bind(U,UTi,UTc1,UTf,UVars1,sw(Y,Z))|OutBS1],
   filter(InBS,V,VTi,VTc1,VTf,VVars1,OutBS1,Feedback,Abort).

   update_vars(no,_,Vars,_,Vars1,sw(A,Z),_) :- true | Vars1=Vars, A=Z.
   update_vars(yes,V,Vars,Vars2,Vars1,SW,Abort) :- true |
      member(Vars2,V,HasV),
      safe_vars(HasV,Vars,Vars2,Vars1,SW,Abort).

      safe_vars(yes,_,_,_,_,_,Abort) :- true | Abort=abort(all).
      safe_vars(no,Vars,Vars2,Vars1,sw(A,Z),_) :- true | A=Z,
         append(Vars2,Vars,Vars1).
```

Fig. 3.6.2 同一化 (II)

```
deref(_,_,_,abort(all)) :- true | true.
deref(var(V),var(V)=S,NewT,_) :- true | NewT=S.
deref(var(V),var(U)=S,NewT,_) :- V\=U | NewT=var(V).
deref(T,Subst,NewT,Abort) :- T\=var(_) |
   functor(T,F,N), rotcnuf(NewT,F,N),
   deref_arg(N,T,Subst,NewT,Abort).

   deref_arg(_,_,_,_,_,abort(all)) :- true | true.
   deref_arg(N,T,Subst,NewT,Abort) :- N>0 |
      N1:=N-1, arg(N,T,T1), arg(N,NewT,NewT1),
      deref(T1,Subst,NewT1,Abort),
      deref_arg(N1,T,Subst,NewT,Abort).
   deref_arg(0,_,_,_,_,_) :- true | true.

vars_in(V,Vi,Vo) :- V=var(_) | member(Vi,V,Ans), vars_in_vars(Ans,V,Vi,Vo).
vars_in(T,Vi,Vo) :- T\=var(_) | functor(T,F,N), vars_in_arg(N,T,Vi,Vo).

   vars_in_vars(yes,_,Vi,Vo) :- true | Vo=Vi.
   vars_in_vars(no,V,Vi,Vo) :- true | Vo=[V|Vi].

   vars_in_arg(N,T,Vi,Vo) :- N>0 | N1:=N-1, arg(N,T,A),
      vars_in(A,Vi,Vm), vars_in_arg(N1,T,Vm,Vo).
   vars_in_arg(0,_,Vi,Vo) :- true | Vo=Vi.

append([H|X],Y,Z) :- true | Z=[H|Z1], append(X,Y,Z1).
append([],Y,Z) :- true | Z=Y.

member([X|_],X,Ans) :- true | Ans=yes.
member([Y|Z],X,Ans) :- Y\=X | member(Z,X,Ans).
member([],_,Ans) :- true | Ans=no.

merge(_,_,_,abort(_)) :- true | true.
merge([A|X],Y,Z,Abort) :- true | Z=[A|Z1], merge(X,Y,Z1,Abort).
merge(X,[A|Y],Z,Abort) :- true | Z=[A|Z1], merge(X,Y,Z1,Abort).
merge([],Y,Z,_) :- true | Z=Y.
merge(X,[],Z,_) :- true | Z=X.
```

Fig. 3.6.3 同一化 (III)

4. 例題

例題を示す前に，Fig. 4.1 に順序づけの一つの方法を与えておく．

order/3 は次のように定義されている．

i) 表現が一致する二つの項は順序づけ不可能である．
ii) 両方とも変数の場合も順序づけ不可能とする．
iii) 片方のみ変数の場合は変数の方が他方より小さいものとする．
iv) 両方とも変数でない場合は，まず双方に含まれる異なる変数の数を調べ，より少ない方が他方より小さいものとする．
v) この比較で順序がつかない場合は，ユーザが与える順序 def_order/3 に従うものとする．

このほか，ここではユーザが順序を定義する際のユーティリティとして，辞書式順序を決定する lex_order/2 が用意されている．

Fig. 4.2 に示す例題は，逆元の特性に関するもので，ただ一個の等式が与えられている．Fig. 4.3 に実行のトレースを示す．

```
order(X,X,Order) :- true | Order=(=).
order(var(X),var(Y),Order) :- X\=Y | Order=[].
order(var(_),T,Order) :- T\=var(_) | Order=(<).
order(T,var(_),Order) :- T\=var(_) | Order=(>).
order(X,Y,Order) :- X\=Y, X\=var(_), Y\=var(_) |
   vars_in(X,[],Xvars), vars_in(Y,[],Yvars),
   vars_order(Xvars,Yvars,VOrder),
   order2(VOrder,X,Y,Order).

   vars_order([_|Xvars],[_|Yvars],Order) :- true |
      vars_order(Xvars,Yvars,Order).
   vars_order([_|_],[],Order) :- true | Order=(>).
   vars_order([],[_|_],Order) :- true | Order=(<).
   vars_order([],[],Order) :- true | Order=[].

   order2(VOrder,_,_,Order) :- VOrder\=[] | Order=VOrder.
   order2([],X,Y,Order) :- true | def_order(X,Y,Order).

lex_order([(>)|Os],Order) :- true | Order=(>).
lex_order([(<)|Os],Order) :- true | Order=(<).
lex_order([(=)|Os],Order) :- true | lex_order(Os,Order).
lex_order([[]|Os], Order) :- true | lex_order(Os,Order).
lex_order([],      Order) :- true | Order=[].
```

Fig. 4.1 順序づけ

```
def_order(i(X),i(Y),Order) :- true | order(X,Y,Order).
def_order(i(_),_*_,Order) :- true | Order=(>).
def_order(_*_,i(_),Order) :- true | Order=(<).
def_order(X*Y,U*V,Order) :- true |
   order(X,U,O1), order(Y,V,O2),
   lex_order([O1,O2],Order).
def_order(X,Y,Order) :-
   (X,Y)\=(i(_),i(_)), (X,Y)\=(i(_),_*_), (X,Y)\=(_*_,i(_)),
   (X,Y)\=(_*_,_*_) |
   Order=[].

ex4(Res) :- true | comp([i(var(x))*(var(x)*var(y))=var(y)],
    Res).
```

Fig. 4.2 例題 (逆元の特性)

```
| ?- ghc ex4(R).
Select eq_1
Add rule_1 : i(var(2))*(var(2)*var(1))->var(1)
Superpose 1 on 1
   1*[2]+1 : i(i(var(2)))*var(1)=var(2)*var(1)
Select eq_1*[2]+1
Add rule_2 : i(i(var(2)))*var(1)->var(2)*var(1)
Superpose 1 on 2
   2*[]+1 : var(1)=var(-2)*(i(var(-2))*var(1))
   1*[]+2 : var(-2)*(i(var(-2))*var(1))=var(1)
   1*[2]+2 : i(i(i(var(-2))))*(var(-2)*var(-1))=var(-1)
Superpose 2 on 2
Reduced eq_1*[2]+2 => var(-1)
Select eq_2*[]+1
Add rule_3 var(2)*(i(var(2))*var(1))->var(1)
Reduced eq_1*[]+2 => var(-1)
Superpose 1 on 3
   3*[2]+1 : var(2)*var(1)=var(2)*var(1)
   1*[2]+3 : i(var(-2))*var(-1)=i(var(-2))*var(-1)
Reduced eq_3*[2]+3 => var(-2)*var(-1)
Reduced eq_3*[]+2 => var(-1)
Reduced eq_3*[2]+2 => var(-1)
Reduced eq_2*[]+3 => var(-1)
Superpose 2 on 3
   3*[]+2 : var(2)*(i(i(i(var(2))))*var(-1))=var(-1)
   3*[2]+2 : i(var(2))*(var(2)*var(1))=var(1)
   2*[]+3 : var(-1)=var(2)*(i(i(i(var(2))))*var(-1))
Superpose 3 on 3
   3*[2]+3 : var(-2)*var(1)=i(i(var(-2)))*var(1)
Reduced eq_3*[2]+1 => var(-2)*var(-1)
Reduced eq_1*[2]+3 => i(var(-2))*var(-1)

R = [rule(1,(i(var(2))*(var(2)*var(1))->var(1))),
     rule(2,(i(i(var(2)))*var(1)->var(2)*var(1))),
     rule(3,(var(2)*(i(var(2))*var(1))->var(1)))] ?
```

Fig. 4.3 例題 (逆元の特性) トレース

5. おわりに

Knuth-Bendix アルゴリズムの並列化を GHC で試みた．このアルゴリズムは，マッチング，簡約化，同一化，重像，の各部で並列性を引き出すことができる．

書き換え規則に転化される等式の選択の良し悪しは，アルゴリズムの停止性を損なうことはないが，全計算量を大きく左右する．複雑な左辺をもつ書き換え規則を選ぶと，多くの要対を生成し易い．生成される要対も複雑なものになり易い．また，これらの要対をそれぞれ方向づけして得られる書き換え規則は，冗長な(より簡単な書き換え規則の組によって置き換え得る)ものであることが多い．これが悪循環すると，不都合な要対の爆発的発生を招き，計算量を著しく増大させることになる．

従って，等式の書き換え規則への転化を完全に並列化するのは逆効果となる．ここではこれを避けて，最も簡単な等式を比較選択するために，等式の集合を一本のストリーム上に直列に配置するようにした．一方，獲得された書き換え規則の対から要対を生成するプロセスは完全に並列化できる．ただし，書き換え規則の対をつくるために，やはり書き換え規則の集合を一本のストリーム上に直列に配置する必要がある．こうして，等式の処理，書き換え規則の処理においては，ストリーム並列が実現されることになった．

参考文献

(1) D. Knuth and P. Bendix, Simple word problems in universal algebras, in J. Leech, (ed.) Computational Problems in Abstract Algebra, pp.263-297, Pergamon, Oxford, 1970.

(2) G. Huet, A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm, *Journal of Computer and System Sciences*, **23**, pp.11-21, 1981.