ICOT Technical Memorandum: TM-0237

TM-0237

全解探索論理プログラムの
決定的論理プログラムへの変換(II)

上田和紀

October, 1986

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

全解探索論理プログラムの決定的論理プログラムへの変換（II）

Making Exhaustive Search Programs Deterministic (II)

上田　和紀　（（財）新世代コンピュータ技術開発機構）

前回大会で，全解探索を目的としたPrologプログラムを，バックトラックや多重環境を用いないＧＨＣないしはPrologプログラムに系統的に変換する方法を示した．今回は，より自然に書かれた generate and test型の論理プログラムのコルーチン的実行を模擬する決定的プログラムを，系統的に得る方法について述べる．変換可能なプログラムのクラスは少なくとも教科書的例題をカバーしており，また得られたプログラムの効率は十分高い．

１．はじめに

本論文は，［上田85］，［Ueda 86b］の続編である（以下これらを，前論文という）．前論文では，バックトラックまたはＯＲ並列探索を用いて全解の収集を行なうHorn節プログラムを，同じ結果を返すようなＧＨＣ［Ueda 86a, 86c, 86d］プログラムないしは決定的(deterministic) なPrologプログラムに系統的に変換する方法を示した．また，変換前後のプログラムの効率を同じ逐次Prolog処理系の上で比べると，順列生成のように解の総数が相対的に多い場合は大幅に改善し，8-queensのように少ない場合でもわずかな低下で済むことを明らかにした．さらに，変換したプログラムからはＡＮＤ並列性を容易に取り出すことができる．これらの結果の意義は，次のようにまとめられよう．

① ＧＨＣのような汎用並列言語［上田86］をベースにしても，探索問題の並列実行が，（最善ではないにせよ）かなりの効率で行なえる見通しがついた．
② Horn節論理を，ＧＨＣのひとつのユーザ言語として位置づけられることがわかった．

さて，前論文の方法によって得られる決定的プログラムは，もとのプログラムのＯＲ並列，ＡＮＤ逐次実行を模擬する．だが，ＡＮＤ逐次実行 ── すなわち各節のゴールを左から順に実行すること ── を仮定してしまうと，Horn節プログラムのレベルで，記述の自然さと効率が両立しなくなってしまうことがある．たとえば図１のプログラムは，一般に“素朴な”8-queensプログラムと呼ばれているものである．このプログラムは，generate and test 型の問題に対して，generator とtesterとが明確に分離して記述してあるので大変わかりやすいが，これを逐次的に実行すると，きわめて効率が悪い（５．参照）．すべてのqueen を置いてみたあとで，それが解になっているかを判定するからである．４．２に紹介するグラフの経路探索プログラムではさらに事情が悪く，generator が無限個の解候補を生成するので，逐次実行では，解の総数が有限個であるにもかかわらずプログラムが止まらなくなる．このため，現実のPrologプログラミングでは，図２のように，generatorとtesterを融合してしまうのが普通である．図２の8-queensプログラムも決して複雑ではないが，順列生成が8-queensの部分問題であることが不明確になってしまっている．実行戦略のためにプログラムのスタイルが犠牲になるのは，高級言語の立場からは望ましいことでない．

```
:- bagof(X, eightqueens(X), B).

eightqueens(X) :-
   perm([1,2,3,4,5,6,7,8], X) // nocheck(X).

perm([H|T], [A|P]) :-
   del([H|T], A, L), perm(L, P).
perm( [],    []  ).

del([H|T], H,   T   ).
del([H|T], A, [H|T2]) :- del(T, A, T2).

nocheck([H|T]) :- qsafe(H, T, 1), nocheck(T).
nocheck( []  ).

qsafe(U, [H|T], N) :-
   H+N=\=U, H-N=\=U, M is N+1, qsafe(U, T, M).
qsafe(_,  [],   _).
```

Fig. 1　Naive 8-Queens Program

```
eightqueens(X) :-
   eightq([1,2,3,4,5,6,7,8], [], X).

eightq([H|T], R, P) :-
   del([H|T], A, L), qsafe(A, R, 1),
   eightq(L, [A|R], P).
eightq( [],   R, P) :- rev(R, [], P).

del([H|T], H,   T   ).
del([H|T], A, [H|T2]) :- del(T, A, T2).

qsafe(U, [H|T], N) :-
   H+N=\=U, H-N=\=U, M is N+1, qsafe(U, T, M).
qsafe(_,  [],   _).

rev([A|X], Y, Z) :- rev(X, [A|Y], Z).
rev( [],   Y, Y).
```

Fig. 2　8-Queens Program Optimized for Sequential Execution

2．問題設定

そこで，本論文では，generate and test 型の自然な論理プログラムから，ＧＨＣやPrologを実行メカニズムとして良い効率で動く全解収集プログラムを，系統的に導く方法について検討する．この要件を満たすには，まずgeneratorとtesterのコルーチン的実行を模擬する必要がある．たとえば図１のプログラムで，ゴールperm([1,...,8],X)がＸの値の候補として[1,2|X2]という構造を生成したら，nocheck(X)を起動して，X2をどう具体化しても[1,2|X2]が解になりえないことを検出しなければならない．またこれと同時に，'bagof' のような，Horn節論理の枠外の機能を消去することも必要である．

図１のようなプログラムのコルーチン実行を模擬して，全解をバックトラックによって生成するPrologプログラムを得る方法は，［Gallagher 82］や［Bruynooghe 86 ］で提案されている．しかし，残念ながら，得られたPrologプログラムは，前論文の方法で変換可能なクラスに属さない．Generate and test 型のプログラムでは，トップダウンに値のきまる共有変数によって両者が通信しあうが，［Gallagher 82］や［Bruynooghe 86 ］の方法で変換したプログラムはこの通信方式をそのまま利用する．これが前論文で用いたモード解析の手法では解析できないからである．

一方，図２のプログラムは前回の論文の方法で変換可能なので，図１のプログラムから図２のプログラムを系統的に得る方法を開発するのもひとつの解決法である．これはエレガントな方法ではあるが，本論文では，前論文の変換手法を拡張することによって，図１のようなプログラムから一挙に全解収集プログラムを導く方法をとる．その方が，変換技法の系統化という観点から早道であると考えたからである．興味ある読者は，図１と図２の等価性，および図１から図２への変換技法の一般化の問題に挑戦していただきたい．

3．方法と変換例

本節では，コルーチン的実行の実現のために，どのように前論文の方法を拡張するかについて，図１の8-queensプログラムを例にとって述べる．

まず，コルーチン実行を行なうべき generatorとtesterを，g(... X ...) // t(... X ...)の形で書くことにする．ここでＸは両者間の共有変数で，gがそれを基底項のリストに頭の方から具体化し，tがその妥当性を検査するものとする．このゴールの組は，t(... X ...)という制約を与えられたg(... X ...)の呼出しと解釈でき，通常の（逐次的に実行される）ゴールのひとつとして現れることができる．Ｘ以外の引数の値は，この組の実行開始時には基底項に具体化しているものとし，Ｘはこのゴールの組の実行前は不定であるものとする．

我々の目的は，gがＸの値を少しきめるたびにtを少しずつ動かして，失敗をなるべく早く検出することである．図１のプログラムを追ってみよう．

perm([1,...,8],X) を実行すると，permの第１節のゴールdel の実行が終わったとき，Ｘの第１要素が１～８のいずれかにきまる．（del は，このプログラムの中で唯一，複数の節によって導出ができる述語であり，これが複数解を生みだすもとになっているが，ひとつの解の生成過程に注目すれば，Ｘの第１要素は特定のひとつの値をとる）．

さてＸの第１要素がきまると，nocheck(X)を走らせることができる．導出原理の上からは，Ｘが不定のうちにnocheck を走らせることももちろん可能だが，ここではnocheck(X)は外から与えられた値の検査にのみ用いることにしているので，Ｘが変数の間は，nocheck のいずれの節によっても導出を行なわない．

Ｘが[1|X1] になったと仮定しよう．すると，nocheckの第１節を用いて，nocheck(X)からqsafe(1,X1,1) とnocheck(X1) を導出することができる．しかし，これらのゴールは，X1の値がきまるまで待たなければならない．そこでnocheck(X)の実行は中断し，perm([1,...,8],X) の実行に戻ることになる．その後permの再帰呼出しによって，del が呼ばれ，Ｘがたとえば[1,5|X2] の形になったとする（ほかの値になる可能性もあるが，ここではひとつの場合に注目している）．すると，qsafe(1,X1,1) から 5+1≠1, 5-1≠1,N is 1+1, qsafe(1,X2,2) が導出できる．このうち最初のみっつのゴールは直ちに実行でき，成功するから，結局 qsafe(1,X2,2)だけが残る．一方 nocheck(X1)からは，qsafe(5,X2,1) とnocheck(X2) とが導出できる．

このような作業をくり返してゆくと，最終的に次のいずれかが起きる：

① qsafe から導出された '≠' の呼出しが失敗する
② Ｘが完全なリストに具体化する

①の場合，注目していた特定の解生成過程が失敗したことになる．②は，Ｘがたとえば[1,5,8,6,3,7,2,4|X8] に具体化したあと，X8がpermの第２節によって[]（空リスト）になるような場合である．このとき，nocheck(X)から導出されたゴールとしては

```
qsafe(1,X8,8), qsafe(5,X8,7), .... qsafe(4,X8,1),
nocheck(X8)
```

が存在しているが，これらはX8が[]になれば直ちに成功する．こうして，perm([1,...,8],X) とnocheck(X)の双方が成功し，解がひとつ見つかったことになる．

さて，上記の過程を模擬する上で重要なことは，次のことが，プログラムの静的解析だけでわかるということであ

る:

① 共有変数Xが表わすリストの“次の要素”の値がきまるのは，permの第１節の中のdel の実行が成功した時である．またリストXが[]によって閉じるのは，permの第２節による導出が成功した時である．

② Xの“次の要素”の値がわかると，それを持っていたnocheck の呼出しから，qsafe と新たなnocheck のふたつのゴールが導出できる．またそれを持っていたqsafe の呼出しがあれば，よっつのゴールが導出でき，そのうち qsafeの再帰呼出し以外は直ちに実行できる．またXの表わすリストの終端が[]によって閉じられると，終了していなかったnocheck およびqsafe の呼出しが成功する．

しかも上の性質は，permの第１引数が[1,...,8] であるという情報がなくても，それが基底項であるという情報があれば導くことができる．このことは，入出力モードの概念を少し拡張することによって，コルーチン実行に関する静的解析が，モード解析という形で，ある程度一般的に，しかもアルゴリズミックに行なえることを示唆している．その一般的手法は４．で述べることとして，ここでは上の①，②の性質を利用して，図１のプログラムのコルーチン実行を模擬する8-queensの全解収集プログラムを導いてしまおう．

まず，前回の論文のときと同様，もとのプログラムに名札を立てる（図３）．この名札が，継続(continuation)を構成するのに使われる．L2とL4は，del が成功したときに，２箇所ある帰り先のどちらに帰るかを示すのに用いる．L4につづく仕事（AとT2からXとYと構成する）は，前論文の順列生成プログラムと全く同じである．一方，L2につづく仕事は，前論文のときとは異っている．前回のときは，まずpermの再帰呼出しを実行してAとPの両方が基底項になってから[A|P]を出力引数と単一化していたが，ここではAが基底項になったらすぐ，nocheck(X)（から導出されたゴール）を起動し，可能な導出を行なう．このとき新たな要素Aが適当ならば導出は成功し，L3以下の仕事を行ない，不適当ならば導出は失敗し，L3には到達しない．

```
  eightqueens(X) :-
     perm([1,2,3,4,5,6,7,8], X) // nocheck(X)
{L1: construct X}.

  perm([H|T], X) :-
     del([H|T], A, L),
{L2} X=[A|P],
    {send A to nocheck(X) and invoke it}
{L3} perm(L, P).
  perm([],   []) {:-}
    {X is terminated, so nocheck(X) succeeds}.

  del([H|T], H, T).
  del([H|T], X, Y) :-
     del(T, A, T2), {L4} X=A, Y=[H|T2].
```

Fig. 3 Labeled 8-Queens Program

permの第２節は，共有変数を閉じる節であるが，このときはnocheck(X)から導出されたゴールは単に捨ててよい．なぜなら，

① リストが閉じるときにnocheck(X)のサブゴールとして残っているゴールは，qsafe(... Xn ...) とnocheck(Xn) の形のもの（Xnは，空リストに具体化しようとしているXのサブリスト）だけであり，

② これらはXnが空リストに具体化すれば成功する

ことが静的にわかるからである．

L1は，このpermの第２節からの唯一の帰り先である．L1に到達すれば，図３のプログラムの上では解がひとつ求まったことになるが，我々の変換技法では，共有変数Xの要素の値を逆順に蓄積してゆくので，L1に到達したときにそれを正順に反転して，ゴールeightqueens(X)の結果として返すという作業が必要になる．逆順に蓄積するのは，Xの値に関する部分解を，常に基底項によって表現することにより，多重環境の生成を回避したいからである．

さて，図４が，図３のプログラムから導いた全解収集プログラムである．上に示したgenerator (perm)側の制御の流れは，前論文のときと全く同じ継続の管理・操作手法によって実現できる．述語e, p, d, d1, d2, cont0, cont1, cont24, cont3 はこの意味で前回と共通の手法で作られている（変換前の述語に対応する変換後の述語は，変換前の述語名の頭文字をとっている）．唯一異なる点は，以下に説明するように，tester (nocheck)に関する情報とXの部分解とを，新たな形でpとpから呼ばれる述語が持ち歩くことにすぎない．

これに対し，testerに関する制御の流れは，全く新しい形で実現される．nocheck に関する“残りの仕事”，すなわち継続は，それ以外のゴールのための継続（主継続と呼ぶ）とは別に，perm用の局所データとして管理する．図４では，Contが主継続，Contn がnocheck(X)から派生した仕事のための継続（副継続という）を表わしている．その初期値n（nocheck ゴールがひとつあることを示す）は，eの第１節で設定している．Contn によって表現される仕事は，nresume によって処理される．nresume は，処理がL2に達したことをcont24の第２節が認識したときに起動され，共有変数の“次の要素の値”を用いて現在のContn から可能な導出を行ない，新しい副継続を作り出す．Contn は，nまたはq(U,N,Contn') の形をしている．nは，nocheck(Xn) というゴールを，q(U,N,Contn') はqsafe(U,Xn,N) およびContn'によって表現されるゴールを表わす．ここでXnは，図３における共有変数Xのサブリストの，いままさに[H|Xn']の形に具体化した部分を表わす．

```
:- e('L0',B,[]).

e(Cont,S0,S1) :- true |
   p([1,2,3,4,5,6,7,8],n,[],'L1'(Cont),S0,S1).

p([],    Contn,SR,Cont,S0,S1) :- true |
   cont1(Cont,SR,S0,S1).
p([H|T],Contn,SR,Cont,S0,S1) :- true |
   del([H|T],'L2'(Contn,SR,Cont),S0,S1).
p(L,     _,      _,    _,    S0,S1) :-
   otherwise | S0=S1.

d(L,Cont,S0,S2) :- true |
   d1(L,Cont,S0,S1), d2(L,Cont,S1,S2).

d1([H|T],Cont,S0,S1) :- true |
   cont24(Cont,H,T,S0,S1).
d1(L,    _,    S0,S1) :- otherwise | S0=S1.

d2([H|T],Cont,S0,S1) :- true |
   d(T,'L4'(H,Cont),S0,S1).
d2(L,    _,    S0,S1) :- otherwise | S0=S1.

nresume(q(U,N,Contn),SR,ContnR,Cont,H,S0,S1) :-
   H+N=\=U, H-N=\=U |
   M is N+1,
   nresume(Contn,SR,q(U,M,ContnR),Cont,H,S0,S1).
nresume(n,           SR,ContnR,Cont,H,S0,S1) :-
   true |
   rev2(ContnR,q(H,1,n),NewContn),
   cont3(Cont,NewContn,[H|SR],S0,S1).
nresume(_,           _,  _,     _,   _,S0,S1) :-
   otherwise | S0=S1.

cont0('L0',_,S0,S1) :- true | S0=[S|S1].

cont1('L1'(Cont),SR,S0,S1) :- true |
   rev(SR,[],S), cont0(Cont,S,S0,S1).

cont24('L4'(H,Cont),       A,T2,S0,S1) :- true |
   cont24(Cont,A,[H|T2],S0,S1).
cont24('L2'(Contn,SR,Cont),A,T2,S0,S1) :- true |
   nresume(Contn,SR,n,'L3'(T2,Cont),A,S0,S1).

cont3('L3'(T2,Cont),Contn,SR,S0,S1) :- true |
   perm(T2,Contn,SR,Cont,S0,S1).

rev([A|X],Y,Z) :- true | rev(X,[A|Y],Z).
rev([],   Y,Z) :- true | Y=Z.

rev2(q(A,B,X),Y,Z) :- true | rev2(X,q(A,B,Y),Z).
rev2(n,       Y,Z) :- true | Y=Z.
```

Fig. 4 Transformed 8-Queens Program

nresume の第１節は，qsafe(U,[H｜Xn'],N) から導出されるよっつのゴールのうちのみっつを直ちに実行し，次回まわしになる再帰呼出しを第３引数ContnR（Rは reversedを表わす）にスタックする．失敗する可能性のある‘≠’の呼出しはガードで実行し，それらが失敗したら第３節によって解収集用の差分リストを短絡する．第２節は，nocheck([H｜Xn']) から導出されるふたつのゴール qsafe(H,Xn',1)およびnocheck(Xn')と，ContnRに（逆順に）スタックしてきたゴールとから，新たな副継続NewContnを作り，名札L3から先の処理のためにcont3 を呼び出す．

共有変数の要素の値は，変数SRに逆順にスタックしてゆく．SRはｅの第１節で[]に初期化され，nresume の第２節でL3に制御を移すときに，次の要素の値Hをスタックしている．SRは，cont1 が，L1における仕事として正順に戻している．

共有変数の表わすリストを閉じると，Contn によって表現されている“残りの仕事”は，さきに述べた理由ですべて成功する．そこで，permの第１節に対応するｐの第１節ではContn は単に捨てて，L1に制御を移している．

４．一般的な変換手順

前論文のときと同様，全解収集プログラムへの変換は，

① モード解析
② 前変換
③ 本変換

の３段階からなる．自動変換を可能にするためにまず重要なことは，変換ができるための十分条件が静的に判定できる形で与えられていることである．この判定は，４．１に述べるモード解析時に行なう．本節では，このモード解析を中心に述べ，変換の詳細は省略する．前節の変換例から帰納・類推していただきたい．

４．１．モード解析

モード解析の目的は，プログラムを実行したときに起きる情報の流れを静的に把握し，変換に役立てることである．モード解析においては，次のよっつのモードのいずれかを各節の本体部のゴールの各引数に対して与える：

- ＋（入力）　そのゴールを実行するときに基底項になっている
- －（出力）　そのゴールが成功したときに基底項に具体化する
- ？（ストリーム入力）　基底項のリストが，対応する‘↑’モードの引数をもつ他のゴールからいずれ与えられる．この引数が[H｜T]の形に具体化したとき，Hは基底項になっており，Tは再び‘？’モードの性質をもつ
- ↑（ストリーム出力）　その述語が成功したときに基底項のリストに具体化する．その引数を[H｜T]の形に（中から）具体化するとき，Hが基底項であり，Tが再び‘↑’モードの性質をもつように述語の定義を修正できる．

‘↑’と‘？’は，一対のgenerator and testerによって具体化される共有変数に対して与えるモードである．

さてプログラムのモード解析とは，トップレベルのゴールのモード宣言をもとに，それから直接間接に呼ばれうるすべての述語の各節のモード解析を行なうことによって，

プログラム全体の情報の流れをモードづけという形で明らかにする作業である．解析に成功すれば，そのモードづけが正しいことが保証され，全解収集プログラムへの変換ができる．失敗すれば，そのプログラムは本論文の方法のままでは変換できない．各節のモード解析は，次のように行なう：

① '+' モードの頭部の引数中に現れる変数にgroundとマークする．

② '?' モードの頭部の引数が
   a. []ならば何もしない．
   b. 変数ならばstreamとマークする．
   c. [H | T]の形ならば，H中の各変数にgroundとマークし，Tは再びこのa.～d.に従って処理する．
   d. 上記以外ならば解析を失敗させる．

③ 本体部のゴールを左から順に調べ，それらのモードづけを行なう．各ゴールのモードは，次のように定める：
   a. 関数記号とgroundとマークされた変数だけからなる引数は '+' とみなす．
   b. streamとマークされた変数だけからなる引数は，'?' とみなす．streamとマークされた変数を含む非変数項の引数があれば解析を失敗させる．
   c. generate and test 型のゴールで，
      (i) generator の引数にただひとつ '+' でないものがあり，
      (ii) それがマークされていない単純変数で，
      (iii) しかもtesterのただひとつ '+' でない引数として単独で現れていれば，
      前者を '↑'，後者を '?' とし，この変数をgroundとマークする．この条件を満たさないgenerate and test型のゴールであれば解析を失敗させる．
   d. 上記以外（マークされていない変数またはそれを含む項）の引数は '−' とし，その中に現れる変数をgroundとマークする．

④ '?' モードの引数を頭部にもつ節の場合，次のことを確認する．できなければ解析を失敗させる：
   a. 本体部に現れる '?' モードの引数は，各ゴールにつきたかだかひとつである．
   b. '?' モードの引数をもつ本体部のゴールの他の引数はすべて '+' である．

⑤ '−' モードの頭部の引数の中に現れる変数がすべてgroundとマークしてあるか調べ，していなければ解析を失敗させる．

⑥ '↑' モードの頭部の引数が
   a. []
   b. groundとマークされた変数
   c. [H | T]の形の項，ここでHの中に現れる変数はすべてgroundとマークしてあり，Tは再びこのa.～c.のいずれかである
   のいずれかであることを確める．それ以外であれば，解析を失敗させる．さらに，b.に分類された変数（c.による再帰的検査で分類されるものを含む）がもしあれば，それが本体部にただ1回，単独の引数として現れるかどうか調べる．そうであれば，その引数のモードを '↑' に訂正する．さもなければ解析を失敗させる．

```
Declared Mode:   eightqueens(-).

          -
eightqueens(X) :-
                +            ^              ?
   perm([1,2,3,4,5,6,7,8], X) // nocheck(X).

       +      ^
perm([H|T], [A|P]) :-
          +    -  -        +  ^
   del([H|T], A, L), perm(L, P).
       +      ^
perm( [],     [] ).

      +    -    -
del([H|T], H,   T   ).
      +    -    -              +  -  -
del([H|T], A, [H|T2]) :- del(T, A, T2).

          ?              +  ?  +           ?
nocheck([H|T]) :- qsafe(H, T, 1), nocheck(T).
         ?
nocheck( [] ).

      +    ?    +
qsafe(U, [H|T], N) :-
   + +   +  + +   +  -    +           +  ?  +
   H+N=\=U, H-N=\=U, M is N+1, qsafe(U, T, M).
      +    ?    +
qsafe(U,  [],   V).
```

Fig. 5 Mode Analysis of the 8-Queens Program

図5に，モード解析を行なった8-queensプログラムを示す．なお，トップレベルのゴールのモード宣言に現れる '↑' と '?' モードは，上記③c.に示した条件を満たさなければならない．

4．2．制限の目的と一般性

以上のようないささか複雑な制限は，次のことを保証するために課したものである：

① generator とtesterは，ただひとつの共有変数によって通信しあう．そしてそれを唯一の出力とする．

② generator は共有変数の値をトップダウンに，徐々にきめてゆく．しかも共有変数の各要素がそれぞれ単一のゴールによって決定されてゆくようにし，複数の単一化が共有変数の具体化をめぐって衝突を起こさないようにする．

③ testerは，共有変数の値の検査をトップダウンに行な

う．ただし，図３のnocheck の第１節のように，複数のゴールが同じ共有変数を独立に検査することは許す．また共有変数またはそのサブリストを引数にとるゴールは，その検査のみを目的とし，他に出力引数をもたない．

モード解析が複雑になるのは，制御が複雑になった代償であり，やむをえない．しかもモード解析を行なうのはプログラマではないから，プログラマの負担がふえるわけではない．

ここで考察しておくべきことは，このモード解析による制限をみたすプログラムのクラスが十分大きいかどうかということである．これについては，コルーチン実行を前提とした論理プログラムの蓄積が乏しいので直ちには判断しかねるが，応用範囲という点からは，リスト，つまり一次元的系列を生成するような探索問題のクラスは十分な一般性をもっているといえよう．たとえば，土人と宣教師，グラフの経路探索，積木の問題など，多くの教科書的例題は，“操作の列”を生成する問題という共通の特徴をもっている．そしてこれらは，列のgenerator とtesterを分離することによってわかりやすく記述でき，①～③の制約も自然に満たすことができる．例として，教科書［古川86, p. 151］の中でProlog-II のfreeze機能を用いて記述して

```
path(Start,Goal,Path) :-
   path1(Start,Goal,Path) // good_list(Path).

path1(X,X, [X]    ).
path1(X,Y,[X|Path]) :-
   neighbor(X,Z), path1(Z,Y,Path).

neighbor(X,Y) :- nb(X,Y).
neighbor(X,Y) :- nb(Y,X).

nb(a,b).  nb(a,c).  nb(b,d).  nb(b,e).
nb(c,f).  nb(d,g).  nb(e,g).  nb(e,h).
nb(f,j).  nb(h,i).  nb(h,j).

good_list( [] ).
good_list([X|L]) :- out_of(X,L), good_list(L).

out_of(X, [] ).
out_of(X,[Y|L]) :- dif(X,Y), out_of(X,L).
```

Fig. 6 Path-Finding Program

いたグラフの経路探索プログラムを本論文の記法で書き直し，モード解析をした結果を図６に与える．

本論文で許した拡張を，さらにある程度一般化することも技術的には可能である．たとえば，‘↑’モードの引数の値が，基底項のリストに（徐々にでなく）一挙に具体化するようなプログラムを許すこともできる．が，そのような一般化の意義は，まさに応用プログラム側の必要性に依存する．

コルーチン的制御が有用だが，ここに示した手法は適用できない教科書的例題として，［古川86, p. 150］にある覆面算(SEND + MORE = MONEY) がある．図７が，これを本論文の記法に直したものである．このプログラムのデータの流れは，8-queensプログラムに比べてはるかに複雑だが，幸い解くべき問題がプログラムに組み込んであるので，トップレベルの節が部分評価できて図８のようになる．このプログラムを前論文の手法に従ってモード解析し，各不等号ゴールを，その両引数のきまる直後の位置まで移動する

```
test([S,E,N,D,M,O,R,Y]) :-
   add([D,N,E,S],[E,R,O,M],[Y,E,N,O,M]),
   S=\=0, M=\=0, different([S,E,N,D,M,O,R,Y]).

add(Xs,Ys,Zs) :- addc(Xs,Ys,0,Zs).

addc([],[],0,[]).
addc([],[],1,[1]).
addc([],[Y|Ys],C,Zs) :- addc([0],[Y|Ys],C,Zs).
addc([X|Xs],[],C,Zs) :- addc([X|Xs],[0],C,Zs).
addc([X|Xs],[Y|Ys],C,[Z|Zs]) :-
   addc99(X,Y,C,C1,Z), addc(Xs,Ys,C1,Zs).

addc99(0,Y,0,0,Y).  addc99(X,0,0,0,X).
addc99(1,1,0,0,2).
...
addc99(9,9,0,1,8).
addc99(0,0,1,0,1).  addc99(0,1,1,0,2).
...
addc99(9,9,1,1,9).

different([X|Xs]) :-
   out_of(X,Xs), different(Xs).
different([]).

out_of(X,[Y|Ys]) :- X=\=Y, out_of(X,Ys).
out_of(X,[]).
```

Fig. 7 "SEND + MORE = MONEY"

```
test([S,E,N,D,1,O,R,Y]) :-
   addc99(D,E,0, C1,Y),
   addc99(N,R,C1,C2,E),
   addc99(E,O,C2,C3,N),
   addc99(S,1,C3,1, O),
   S=\=0,
   S=\=E, S=\=N, S=\=D, S=\=1, S=\=O, S=\=R,
   S=\=Y,
   E=\=N, E=\=D, E=\=1, E=\=O, E=\=R, E=\=Y,
   N=\=D, N=\=1, N=\=O, N=\=R, N=\=Y,
   D=\=1, D=\=O, D=\=R, D=\=Y,
   1=\=O, 1=\=R, 1=\=Y,
   O=\=R, O=\=Y,
   R=\=Y.
```

Fig. 8 Unfolded Top-Level Clause

```
test([S,E,N,D,1,O,R,Y]) :-
            - - +  -  -
    addc99(D,E,O, C1,Y),
    D=\=1, E=\=D, E=\=1, E=\=Y, D=\=Y, 1=\=Y,
            - - +  -  +
    addc99(N,R,C1,C2,E),
    E=\=N, N=\=D, N=\=1, N=\=Y,
    E=\=R, N=\=R, D=\=R, 1=\=R, O=\=R, R=\=Y,
            + - +  -  +
    addc99(E,O,C2,C3,N),
    E=\=O, N=\=O, D=\=O, 1=\=O, O=\=Y,
            - + +  +  +
    addc99(S,1,C3,1, O),
    S=\=0, S=\=E, S=\=N, S=\=D, S=\=1, S=\=O,
    S=\=R, S=\=Y.
```

Fig. 9 Top-Level Clause with Reordered Goals

作業を行なうと，図9のようなプログラムになる．このプログラムから前論文の手法で全解収集プログラムを導けばよい．ただ，多数の単位節の集まりであるaddc99（足し算の表）が4通りのモードで呼ばれるので，その変換に関しては一工夫必要であろう．なお，この覆面算プログラムの変換の計算量は，全解収集の計算量よりは複雑度が小さい．

4．3．前変換・本変換

前論文では，全解収集プログラムへの変換の前に次のふたつの前変換を行なっていた：

① 複数のモードで呼ばれる述語は，モードごとに別の述語名にする．
② 節の頭部で行なわれる出力引数の単一化を，本体部の最後に移す．また本体部のゴールの出力引数を単純化する．

ここではさらに，次のことが必要である．

③ 節頭部の‘↑’モードの引数が$[H_1, \ldots, H_n \mid T_n]$（$n \geq 1$）の形のとき，頭部の引数を新しい変数$T_0$におきかえ，ゴール$T_0 = [H_1 \mid T_1]$, $T_1 = [H_2 \mid T_2]$, …, $T_{n-1} = [H_n \mid T_n]$を，この順序で本体部の適当な場所に挿入する．一般に，ゴール$T_{i-1} = [H_i \mid T_i]$の適当な場所とは，H_iが基底項になっている場所で，上記の順序をくずさない範囲でできるだけ左の場所である．
④ 節頭部の‘?’モードの引数が$[H_1, \ldots, H_n \mid T_n]$（$n \geq 2$）の形のとき，補助述語(n-1)個を用意して，共有変数の分解が導出1回につき1要素ずつ行なわれるようにする．

本変換は，generate and test 型ゴールのtesterの起動と管理以外に関しては前回のときと同様である．Testerは，副継続の形でgenerator に管理させる．Generator は，新しい共有変数の要素を決定するたびにそれを起動し，可能な導出を行ない，残ったゴールを新たな副継続としてgeneratorの実行に戻る．

5．性能測定

変換した全解収集プログラムは，決定的なPrologプログラムとみなして実行できる［前論文］．そこで，図4の8-queensプログラムのPrologプログラムとしての効率を，他の8-queensプログラムと比較した．処理系はDEC2065 上のDEC-10 Prolog で，局所的な最適化を各プログラムに施して測定した．なお下記の時間は廃品回収の時間を含まない．

① 図1のプログラム（逐次実行）と'bagof' の組合せ …… 24765 msec.
② 図4のプログラム（コルーチンを模擬する全解収集プログラム）…… 2045 msec.
③ 図2のプログラムと'bagof' の組合せ‥ 1798 msec.
④ 図2のプログラムから導いた全解収集プログラム …… 1938 msec.

このように，変換後のプログラム（②）は，もとのプログラムをコルーチン制御を行なわずに実行したときと比べ，10倍以上の効率になっている．Generator とtesterを融合したプログラムから導いた全解収集プログラム（④）と比べるとわずかに遅いが，それに非常に近い性能を達成している．また，②のプログラムには，並列ＧＨＣ処理系［宮崎86］によって容易にAND 並列性を取り出すことができるという特徴がある．

図6の経路探索プログラムに関する結果も，参考に示す．下記の結果は，gからjへの経路（よっつある）を求めるのにかかった時間である：

① 図6のプログラム（逐次実行）と'bagof' の組合せ …… 停止しない
② 図6のプログラムから導いた，コルーチンを模擬する全解収集プログラム…… 28 msec.
③ 図6のgenerator とtesterを融合したPrologプログラムと'bagof' の組合せ…… 36 msec.

なお，'bagof' による全解収集とプログラム変換による全解収集の時間の単純比較は，'bagof' 側にとって若干不利であることを注意しておきたい［Hermenegildo 86 ］．それは，バックトラックと'bagof' による処理時間は，バックトラック時のスタック操作に伴う廃品回収の時間を含むのに対し，プログラム変換による処理時間は廃品回収時間を含まないからである．

6．まとめ

Generate and test 型の素朴なHorn節プログラムから，

最適に近い効率の全解収集プログラムを導く自動化可能な方法を述べた．重要なことは，一般にリストを生成するようなプログラムのコルーチン的制御は，（少なくとも教科書的例題に関しては）静的に解析できるということである．このことは，Prolog-II［Colmerauer 82］やＥＳＰ［Chikayama 84］におけるゴールの遅延実行機能も，プログラム変換で消去できる場合があることを示している．

ここに示したコルーチン実行への拡張は，特に一般性を意図したものではない．しかしながら，何かの系列をgenerate and testによって生成するという枠組は，十分な応用範囲をもっていると期待できる．この種の変換技法をさらに発展させるには，より大きく複雑で，しかも厳密な意味で論理プログラムと呼べるような探索プログラムが蓄積されてゆくことが重要である．

謝辞

ICOT第１研究室の諸氏との討論は有益であった．ここに感謝の意を表する．

参考文献

［Bruynooghe 86］Bruynooghe, M., De Schreye, D., Krekels, B.: Compiling Control, Proc. 1986 Symp. on Logic Programming, pp.70-77 (1986).

［Chikayama 84］Chikayama, T.: Unique Features of ESP, Proc. Int. Conf. on Fifth Generation Computer Systems 1984, ICOT, pp.292-298 (1984).

［Colmerauer 82］Colmerauer, A.: Prolog II Reference Manual and Theoretical Model, Internal report, Groupe Intelligence Artificielle, Universite Aix-Marseille II (1982).

［Hermenegildo 86］Hermenegildo, M. V.: private communication (1986).

［Gallagher 82］Gallagher, J.: Simulating Coroutining for the 8-Queens Problem, Logic Programming Newsletter, L. M. Pereira (ed.), Universidade Nova de Lisvoa, No.3, pp.10-11 (1982).

［Ueda 86a］Ueda, K.: Guarded Horn Clauses, Proc. Logic Programming '85, LNCS 221, Springer, pp.168-179 (1986).

［Ueda 86b］Ueda, K.: Making Exhaustive Search Programs Deterministic, Proc. 3rd Int. Conf. on Logic Programming, LNCS 225, Springer, pp.270-282 (1986).

［Ueda 86c］Ueda, K.: Guarded Horn Clauses: A Parallel Programming Language with the Concept of a Guard, ICOT Tech. Report TR-208, ICOT (1986).

［Ueda 86d］Ueda, K.: Introduction to Guarded Horn Clauses, ICOT Tech. Report TR-209, ICOT (1986). Also to appear in NEC Research & Development, No.84 (1987).

［上田85］上田和紀：全解探索論理プログラムの決定的論理プログラムへの変換，日本ソフトウェア科学会第２回大会論文集，pp.145-148 (1985).

［上田86］上田和紀：並列プログラミング言語，情報処理，Vol.27, No.9, pp.995-1004 (1986).

［古川86］古川康一：Prolog入門，オーム社(1986).

［宮崎86］宮崎敏彦，瀧和男：multi-PSIにおけるFlat GHC の実現方式，Proc. the Logic Programming Conference '86, ICOT, pp.83-92 (1986).