ICOT Technical Memorandum: TM-0138

TM-0138

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

上田和紀

October, 1985

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

Making Exhaustive Search Programs Deterministic

上田　和紀　（（財）新世代コンピュータ技術開発機構）

全解探索を目的とした Horn 節プログラムないしは逐次 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 (Guarded Horn Clauses) プログラムないしは逐次 Prolog プログラムに変換する方法を述べる．逐次Prolog処理系上の実験では，本変換によって，全解探索の効率が順列生成の場合で６倍向上した．さらに本変換は，GHC のAND 並列性を利用して全解探索の並列実行を可能にするという点でも重要である．

１．動機

Horn節プログラムを，与えられたゴールのすべての解（＝ゴールを満足させるような変数の値の組）を求めるために用いることがしばしばある．Prologで探索型の問題を解く場合がこれにあてはまる．しかし，求めた全解を同じまな板の上にのせて，総数の計算，比較，類別などの処理をさらに施そうとすると，これが簡単にはゆかない．得られた解のそれぞれが，探索木の別々の枝に対応する，本来独立なものなので，それらを解のリストのようなひとつのデータ構造にまとめることが，Horn節プログラムの範囲では容易にはできないからである．このため多くのProlog処理系には解のリストを作る'setof'，'bagof'という組込述語が用意してあるが，このようなprimitive なしで効率的な全解収集はできないものだろうか．

また，GHC [Ueda 85] のような全解探索を直接には支援していない言語の上で，全解探索を行なうにはどうすればよいであろうか．特にGHC のような並列言語の場合，その並列性を探索に活かしたい．

これらの要求に答えるひとつの方法は，“ＳはN-queens問題の全解のリストである”というような関係を表わす述語をいきなり書き下すことであろう．このような述語がHorn節論理の中で記述できることはほぼ自明だが，実際に書こうとすると，ひとつの解をみつけるプログラムを書くよりはるかに難しい．そこで，このようなプログラムを得るための気のきいたサポートがほしくなる．

２．方針

ここで考える方法は，全解探索を目的としたHorn節プログラムを，同じ結果を返すようなGHC プログラムないしは決定的なPrologプログラムに変換する方法である．ここで“決定的な”とは，正確には一度生成した変数と値の結合は解除されないということであり，Prolog処理系の用語を借りるとtrail stack がいらないということである．この制限は，GHC がHorn節に対して加えた意味的制限ときわめてよく一致しており，それゆえ，変換したプログラムは，“｜”とcut 記号の間の読みかえによって，GHC プログラムとしてもPrologプログラムとしても解釈できる．

このプログラム変換は二重の見方と意義をもっている．ひとつは，Horn節プログラムからGuarded Horn節プログラムへの変換という見方である．これには，OR並列性のAND並列性への変換によって，並列探索に一般に必要な多重環境機構が不要になるという意義がある．もうひとつの見方は，Prologにおけるプログラム変換というものである．これにはバックトラックと'bagof' 機能の消去という単純化の意義があるほか，後で示すように，プログラムによっては効率が著しく改善するという意義もある．

このプログラム変換にはもうひとつ重要な意義がある．それは，解探索を単一環境で行なうようにするので，プログラムの中に探索の制御の機構を組みこむ可能性が生まれるということである．つまり，本変換は，無制御の探索を，より知的な探索にしてゆくための出発点を与えてくれる．

変換後のプログラムをGHC プログラムとして見ると，もとのプログラムのOR並列AND 逐次実行を模擬するものになっている．OR並列性は上述のようにGHC のAND 並列性に変換され，AND でつながったゴールの逐次実行は，継続(continuation)の持ち歩きによって実現される．

３．従来の方法とその問題点

並列論理型言語でHorn節プログラムの全解探索を行なう試みは，[Hirakawa et al. 84]や[Clark & Gregory 84]が行なっている．これらの試みは，Concurrent Prolog やPARLOGでHorn節プログラムのインタプリタを記述するものであるが，次のふたつの問題がある．

① インタプリタを介するので低速である
② 探索木を並列に調べるのに動的なvariant 作成によって環境の多重化を行なっている

①は，部分評価の手法によって解決できよう．あるいはこれらのインタプリタに対応するコンパイラをいきなり書き下しても大した手間ではない．問題は②である．

多重環境が必要なのは，あるゴールについて複数の節による導出を同時に行なうと，一般に異なるユニファイアが生成するからである．そこでインタプリタで解探索をする場合，解こうとしたゴールに対して複数の候補節が現われたときに，導出に先立って，その時点のゴールの集合と集めるべき解（の部分的に完成したもの）とのvariant （項の中のすべての変数を新しい変数に組織的におきかえてできる項）を作成する．上記のインタプリタも，variant 作成をへらす工夫を施してはいるが，基本は同じである．

しかし，動的にきまる項のvariant の作成には次の問題がある．variant を作る述語（'copy'と名づける）は，実行のタイミングが意味をもつので，'copy(X,Y)' を，これと論理的に等価な'X=Z, copy(Z,Y)'に書きかえることができない．そこで，この両者を同一のものとみなすGHC では，'copy'の意味が規定できない．逐次Prologを目的言語にするときでも，このようなextralogicalな述語はできるだけ

避けるべきである．プログラムの意味づけやプログラミング・システムによる支援が困難になるからである．

上の研究のほかに，Prologプログラムを非決定性のない言語にコンパイルする試みが，［元吉82］，［玉木83］でふれられている．しかし具体的にどのような目的プログラムが得られるかについては報告されていない．

4．簡単な例

従来の方式と本論文で提案する方式の差異をみるため，'append'を使ってリストを分割する例を考える．

```
:- append(U, V, [1,2,3]).
append([],     Z, Z      ).                    --①
append([A | X], Y, [A | Z]) :- append(X, Y, Z).
                                               --②
```

②からはU=[1 | X]という結合が得られ，その後再帰呼出しによって X=[], X=[2], X=[2,3] という３通りの結合が求まるが，これらの解は '[1|' という共通のprefixを共有できない．これが[1 | X]という部分解のvariant を作る理由であった．

これに対した本方式では，②のように解が徐々にきまる節は，まず

```
append(X,Y,[A | Z]) :- append(X2,Y,Z), X=[A | X2].
                                               --②'
```

と等価変形する．②とちがって②'は，ゴールを左から実行することにすると，解を基底項(ground term) の組合せでbottom-up に作るプログラムになっている．出力引数Xの値は再帰呼出しの後まで不定のままだから，部分解のvariant を作る必要がない．そのかわり②'では，再帰呼出しの際，残りの仕事，つまりX2にAを付加してXを得る仕事を継続としてスタックしておく必要がある．しかし，Aは整数値を持っているから，この継続は基底項として表現でき，再帰呼出しが枝分かれを起こしても，枝ごとにコピーを作る必要がない．

これでバックトラック除去の準備ができた．図１に，ゴール

```
:- ..., bagof((X,Y), append(X,Y,Z), S), ...
```

と等価な出力を返すGHC プログラムを示す．'append'のふたつの節①，②'の探索は，AND でつながったふたつのゴール'ap1', 'ap2'がそれぞれ担当する．その引数は，(i) もとのプログラムの入力引数，(ii)継続，(iii) 結果の差分リストの頭，(iv)その尾　の順である．①は単位節なので，'ap1' の本体部では，継続が示す“残りの仕事”を実行しにゆく．この際①のふたつの出力（[]と入力引数自身）を継続処理の述語に与える．'ap2' は，入力引数が[A | Z] の形をしていれば，あとの単一化に必要な情報を継続に付加し，最初のゴール（再帰呼出し）を実行する．さもなければ入力引数の単一化は失敗であり，空の差分リストを返す．'cont'は継続処理用の述語で，継続が'L1'(A,Cont)の形であればAをXの前に付加して，さらにContを処理する．継続が'L0'ならば，そのとき'cont'が受けとっている出力を差分リストに挿入する．継続を構成する関数記号'L0'は，もとのプログラムのトップレベルの呼出しの直後を，'L1'は②'の再帰呼出しの直後を表わすラベルと考えてよい．

```
:- ..., ap(Z, 'L0', S, []), ...

ap(Z, Cont, S0, S2) :- true |
    ap1(Z, Cont, S0, S1), ap2(Z, Cont, S1, S2).
ap1(Z, Cont, S0, S1) :- true |
    cont(Cont, [], Z, S0, S1).
ap2([A|Z], Cont, S0, S1) :- true |
    ap(Z, 'L1'(A,Cont), S0, S1).
ap2(Z,     _,    S0, S1) :- otherwise | S0=S1.

cont('L1'(A,Cont), X, Y, S0, S1) :- true |
    cont(Cont, [A|X], Y, S0, S1).
cont('L0',         X, Y, S0, S1) :- true |
    S0=[(X,Y)|S1].
```

Fig.1　List Decomposition Program

このプログラムでは'ap1' と'ap2' が返す結果を差分リストの連結によってまとめているが，この収集法はfairでない．Fairな収集をする必要があれば，差分リストの連結ではなく，リストのfairな併合によって解を集める必要がある．

なお，このプログラムをPrologプログラムにするには，各節の“|”を“!”に変え，'otherwise'（他のすべての候補節が失敗すると成功する述語）の呼出しを消し，それをもっていた節を述語の最後の節とすればよい．

5．一般的な変換手続き

本節では，４．で示したような変換が容易に（機械的に）できるようなクラスを与え，変換手続きを手短に示す．例題には順列生成プログラム（図２）を用いる．

まず，あとに示す変換手続きにかかるようなHorn節プログラムのクラスを示す．それは，節の中のゴールを左から右に実行したときに下の条件をみたすというものである．

- プログラム中に現れるすべてのゴールの引数は，入力引数と出力引数に分類できるものとする．入力引数とは，そのゴールが呼ばれるとき，基底項になっているものであり，出力引数とはそのゴールが成功したら，基底項に具体化するものである．

Multiple writer （６．参照）を用いたプログラムと差分リストを扱うプログラム以外は，通常この条件を満たすようにできるであろう．Multiple writer を用いている場合は６．に述べる変換が必要になる．差分リストを扱うプログラムに対しては，入出力概念の簡単な拡張で対処できる．

入出力モードについては，プログラマにすべて宣言させるのも一法だが，トップレベルのゴールだけ宣言させ，あとは次の規則にしたがって決める方が現実的であろう．

(a) 呼出し時に基底項の引数：　入力引数とみなす
(b) それ以外の引数：　出力引数とみなす

このモード解析は，プログラムが上記クラスに属するかの検査と同時に容易にできる．具体的には，頭部の入力引数に現われる変数は基底項であり，本体部のゴールに現われた変数はそれ以降では基底項であると見なして，左から順に上記(a)，(b)の判断をしてゆく．図３にその結果を示す．

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

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

Fig.2　Permutation Program

```
Given Declaration:  perm(+, -).
('+': input, '-': output)

        +     -
perm( [],   []).
        +     -                   +    - -          + -
perm([H|T],[A|P]) :- del([H|T],A,L), perm(L,P).
       +    -    -
del([H|T],H,  T).
       +    -    -                 + - -
del([H|T],L,[H|T2]) :- del(T,L,T2).
```

Fig.3 Mode Analysis of the Permutation Program

さて変換手続きであるが，次のような段階を経る．

① プログラムの中に複数の異なる入出力モードで呼んでいる述語があったら，モードごとに別の述語にする．
② プログラムの各節を標準形に直す．
③ プログラムの各述語を変換する．

①は，多モードの述語をなくすためである．この変換によって，モードは述語に付随する概念となる．
②は，次のような操作からなる．

(i) 単位節以外の節について，頭部の出力引数T1～Tnを相異なる新しい変数V1～Vnにおきかえ，その節の最後にゴールT1=V1, ..., Vn=Tn を置く．
(ii) 本体部の各ゴールについて，その出力引数T1～Tnが相異なる新しい変数でなければ，それらを相異なる新しい変数V1～Vnにおきかえ，そのゴールの直後にゴールV1=T1, ..., Vn=Tnを置く．

順列プログラムの標準形は図４のようになる．
次に③の概要を示す．順列プログラムにおける具体例は図５に示す．下の説明に現れる各概念が，順列プログラムの例で何に相当するかを，〔〕の中に示しておいた．

a. 変換後の述語は，もとの述語の入力引数と，継続と，解を返す差分リストの頭と尾からなる．個々の述語は，変換前の述語の仕事をして，さらに継続の示す仕事を行なう．
b. 導出に使える節が高々１個しかないような述語〔'perm'〕については，変換後の述語は，もとのプログラムの各節を変換した節〔<1>, <2>〕から構成する（i.参照）．そうでない述語〔'del'〕については，各節を変換した節〔<5>, <7>〕に別々の述語名〔'd1', 'd2'〕を与え，それらをすべて試みて結果を集める述語〔'd'〕を対応させる．
c. 単位節を変換した節〔<1>, <5>〕のボディ（"|"の右側）では，継続処理用のゴール〔'contp', 'contd'〕を呼ぶ．このゴールには，当単位節が出力として返す値（の組）〔[], (H,T)〕も与える．
d. 非単位節を変換した節〔<2>, <7>〕のボディでは，最初のゴール〔'del'〕に対応する述語を呼ぶ（e., j. 参照）．
e. ある節のｉ番目のゴール G_i 〔たとえば'del'の再帰呼出し〕に対応する（変換後の）述語を〔<7>のボディで〕呼出す際は，それが最後のゴール G_n でない限り，ゴール G_{i+1} を示すラベル〔L3〕と，G_{i+1} ～ G_n で用いる入力データ〔H〕をpushする．またトップレベルのゴールに対応する述語を呼ぶ際は，全体の終了を示すラベル〔L0〕と，集めるべき項を作るのに必要なデータ〔なし〕とを継続の初期値として与える．
f. 継続処理用の述語は，e.でpushした各ラベルに対応する節〔<9> ～<12>〕からなる．これらの節は，各ラベルの直前の述語によって分類し，それぞれ別の述語〔'contp', 'contd'〕に分ける．
g. 継続処理用の述語の節〔たとえば<12>〕は，各ラベル〔L1〕と共に積まれた情報〔なし〕と直前のゴールの出力〔A, L〕から，そのラベルを表わすゴール〔perm(L,P)〕への入力〔L〕を作り，それを処理する述語〔'p'〕を呼ぶ（c., j. 参照）．
h. 全体の終了を示すラベル〔L0〕を処理する節〔<10>〕は，トップレベルのゴールの出力〔P〕と，ラベルと共に積んであった情報〔なし〕とから，集めるべき項〔P〕を生成し，それを要素とする差分リストを返す．
i. 変換後の述語のうち，入力引数の単一化が失敗する可能性のあるもの〔'p', 'd1', 'd2'〕については，その場合に以後の処理をせずに空の差分リストを返すための節〔<3>, <6>, <8>〕を追加生成する．
j. 上の原則にかかわらず，もとのプログラム中の"="や組込述語の呼出しについては，対応する述語を作り出すことなく，"その場で"処理して次の処理に行く〔<9>, <11>〕．

```
perm([],[]).
perm([H|T],X) :- del([H|T],A,L),
    /*L1*/ perm(L,P), /*L2*/ X=[A|P].

del([H|T],H,T).
del([H|T],L,X) :- del(T,L,T2), /*L3*/ X=[H|T2].
```

Fig.4 Standard Form of the Permutation Program

```
<1> p([],    Cont, S0,S1) :- true |
        contp(Cont, [], S0,S1).
<2> p([H|T], Cont, S0,S1) :- true |
        d([H|T], 'L1'(Cont), S0,S1).
<3> p(L,     _,    S0,S1) :- otherwise |
        S0=S1.

<4> d(L, Cont, S0,S2) :- true |
        d1(L, Cont, S0,S1), d2(L, Cont, S1,S2).
<5> d1([H|T], Cont, S0,S1) :- true |
        contd(Cont, H, T, S0,S1).
<6> d1(L,     _,    S0,S1) :- otherwise |
        S0=S1.
<7> d2([H|T], Cont, S0,S1) :- true |
        d(T, 'L3'(H,Cont), S0,S1).
<8> d2(L,     _,    S0,S1) :- otherwise |
        S0=S1.

<9> contp('L2'(A,Cont), P, S0,S1) :- true |
        contp(Cont, [A|P], S0,S1).
<10>contp('L0',         P, S0,S1) :- true |
        S0=[P|S1].
<11>contd('L3'(H,Cont), L, T2, S0,S1) :- true |
        contd(Cont, L, [H|T2], S0,S1).
<12>contd('L1'(Cont),   A,  L, S0,S1) :- true |
        p(L, 'L2'(A,Cont), S0, S1).
```

Fig.5 Permutation Program Transformed

6. 対象プログラムの制限について

本論文の方式が応用面から有効であるためには，５．で示したHorn節プログラムのクラスが実用上十分大きくなけ

表１　全解探索プログラムの性能（単位msec.）

プログラム	変換前 'bagof'	変換前 探索のみ	変換後	解の数
リスト分割	836	4	27	51
順列生成	354	34	57	120
5-queens	45	20	28	10
6-queens	90	75	106	4
7-queens	441	325	446	40
8-queens	1796	1484	1964	92

ればならない．この意味で問題なのは，multiple writer を用いたプログラムである．mutlipie writer とは，共有データをもち，早い者勝ちでその各部の値を具体化しようとする多数のゴールのことである．Prologでは，このようなデータの表現と操作に非基底項と単一化をそのまま利用するのが好都合で，構文解析プログラムにおける解析情報の抽出等にこの技法が好んで用いられているが，本論文に示した変換技法の観点からは，次のような問題がある．

① データ構造のどの部分をどのゴールが具体化するかが一般には静的に解析できない．
② 最終結果が基底項になるとは限らない．

そこで，multiple writer を用いたプログラムを本変換にかけるためには，次のような前変換をHorn節プログラムの段階でする必要がある．それは，共有データを非基底項ではなく，各ゴールの出力する結合情報のリストとして表現し，各ゴールは現在のリストを受けとって新しいリストを返すように変換するのである．新たな結合情報をリストに追加して返すときは，既存の情報との無矛盾性を検査しなければならない．これは，最も安直には，もとの表現のデータが作れるかを試せばよいが，結合情報のデータ構造を工夫すれば，もっと効率的な検査が可能になるはずである．この結合情報は基底項で表現できるから，解の並列探索の場合でもvariant を作る必要がない．

multiple writer の上記の二方式を比較すると，まずプログラムの書きやすさの点で，新しい方式が一見不利であるが，これは抽象データの表現の違いにすぎず，本質的な困難は全くない．次に効率の点だが，たしかに逐次Prologにおいては，非基底項を用いるほうが無矛盾性の検査に有利である．しかし，この方式はバックトラックによる実行には向いているものの，並列実行に移行しようとすると，バックトラック機構にかえて多重環境の生成・管理機構を導入しなければならず，その機構のオーバーヘッドが新たにかかる．基底項による表現に変換すれば，無矛盾性の検査が高くなるものの，並列実行への移行が容易になる．

７．効率比較

表１に，変換前後のプログラムの性能比較を示す．例題は，本稿で示した例題およびN-queens (N=5,6,7,8)である．N-queensプログラムは，５．に示したクラスで記述したものを用いた．

変換前のプログラムも変換後のプログラムも，DEC2065上のDEC-10 Prolog 処理系にかけて測定した．変換前のプログラムについては，'bagof' で解を収集する場合のほかに，強制バックトラックによって解空間を全部調べはするが，解の収集は行なわない場合のデータも示した．各プログラムは，Prologの節の検査が逐次的であることを利用した高速化を施してから測定した．

表が示すとおり，提案したプログラム変換を行なうと，順列生成で６倍，リストの分割では３０倍以上の効率向上がみられる．この著しい高速化は，解を集める作業を'bagof' で行なう場合よりも特殊化して，Horn節論理の枠内で表現するようにしたため，コンパイラが良いコードを出せるようになったことによる．N-queensのように探索空間に比べて解の数が少ない場合は，このような著しい効率向上はなく，解が相対的に多い5-queensの場合を除いて，逆にわずかに遅くなっている．しかし，変換後のプログラムを人手で最適化すれば，8-queensの場合でも，'bagof' による解収集の効率を上回る．

上の結果でもうひとつ注目すべきことは，本変換によるプログラムが，解収集を行なわず，しかも自動バックトラックという探索問題むきの機構を利用したプログラムと比べても，8-queensで25% 程度しか遅くなっていないことである．これは，変換後のプログラムの大幅な改良は，探索方式を変えない限りむずかしいことを示唆している．

８．まとめと今後の課題

全解探索のためのHorn節プログラムを，決定的なGHC またはPrologのプログラムに変換する方法を述べた．本稿では述べなかったが，継続を用いる手法は，全解探索ばかりでなく解をひとつ見つければよい場合にも適用できる．解探索を単一環境で行なうようにしたので，プログラムの中に探索の制御の機構を組込むことも可能になった．

変換にかかるプログラムのクラスを制限したが，プログラムをこのクラスの中で記述することに大きな困難はない．むしろ，効率を失わず，extralogicalな機能も用いない変換が簡単にできるようなHorn節プログラムのクラスを呈示したことの実用上の意味は大きいと考える．

自動バックトラック等の探索問題向きの機構を用いないことによる性能低下は軽微であった．逆に，解の数が探索空間に比べて相対的に十分多ければ，'bagof' による解収集の効率を飛躍的に改善できることがわかった．

本変換は，多重環境を不要にするという意味で，探索の並列処理を容易にはするが，それ以外の資源管理の問題を解決するわけではない．資源管理が，並列探索を実現する上での重要課題である点に変わりはない．

今のところ自動変換プログラムは未完成だが，ここに示した変換を行なうだけなら何も問題はない．実用的観点からは，通常のプログラム変換技術によってさらに高速化したり，逆に類似の述語をまとめる等の工夫をして，効率とひきかえにコード量を減らすことも考える必要があろう．

参考文献

[玉木83]　Prologの関数サブセットPとそれ自身による処理系記述，Proc. Logic Prog. Conf. '83, ICOT.

[元吉82]　Lingolコンパイラ，情報処理学会記号処理研究会資料21-3.

[Hirakawa et al. 84]　Eager and Lazy Enumerations in Concurrent Prolog, Proc. 2nd Int. Logic Programming Conf., Uppsala.

[Clark and Gregory 84]　Notes on the Implementaion of PARLOG, J. Logic Programming, Vol.2, No.1.

[Ueda 85] Guarded Horn Clauses, ICOT Tech. Report TR-103.