(1) 引数の複写

本文中のプログラムでは手続き v_append_1 と v_append_2 のあいだのデータの共有をさけるために手続き v_copy によって陽に複写をおこなっているが，言語 SL では引数が値わたしなのでその必要はない．したがって，記述が簡潔になっている．そのかわり，言語 SL 上では複写をへらす最適化は記述できなくなっている[注21]．

(2) 結果のかえしかた

本文中のプログラムにおける第1引数は上記のプログラムでは関数値としている．前者では第1引数は差分ベクトルあるいは通常のベクトルだったが，後者ではそのかわりにストリームを使用している．

(3) 手続き間インタフェース形式のちがい

本文中のプログラムでは手続き間でうけわたすデータの形式として個別配列形式を採用していた．しかし，上記の関数型言語によるプログラムでは結果は関数値としてかえすしかなく，結果は1個のストリームなので，単一配列形式[注22]を採用している．

[注21] もちろん，機械語にちかいレベルに変換してからこの最適化をおこなうことはできる．

[注22] 単一ストリーム形式とよぶのがより正確であろう．

第7章
論理プログラムへの応用 — 2
AND 並列性をふくむプログラムのベクトル化

要旨

リスト処理においては，mapping のようにリストの全要素に対して同一の処理がほどこされることがおおい．これらの処理は一種の AND 関係にあるとみなすことができ，広義の AND 並列処理の対象となりうる．しかし，リストをたぐる処理には本質的な逐次性があり，パイプラインにのせることは困難である．このようなばあいに，リストに一種のデータ構造変換 (CDR コーディング) を適用することによってベクトルに変換しベクトル処理することによって高速処理をはかることがかんがえられる．この方法を論理プログラムに自動的に適用することをめざして研究をおこない，Prolog による素数生成のプログラムに適用するとともに，マルチ・ベクトルというデータ構造をつかうことによって GHC による素数生成のプログラムに適用することを可能にした．これまでのところ自動変換には成功していないが，第6章でしめしたのと同様の論理型中間語表現への変換をへて Fortran および Pascal のプログラムに手動で変換し，HITAC S-810 において Prolog プログラムにおいては約4倍，GHC プログラムにおいては約3倍の実行高速化をはかることができることが実測によりたしかめられた．

7.1 はじめに

第6章では，バックトラックがおおい Prolog (逐次論理型言語) プログラムをベクトル化して処理する方法をしめした．しかし，このベクトル化法で広義の OR 並列性がないプログラムをベクトル化しても高速化できない．すなわちバックトラックがすくない，またはバックトラックがないプログラムをベクトル化しても高速化できない．したがって，Prolog のベクトル処理可能な範囲をひろげる，あるいは GHC のような並列論理型言語のプログラムをベクトル化するためには，広義の AND 並列性をパイプライン化するベクトル化法を開発して使用する必要がある．

リスト処理においては，mapping すなわち Lisp の mapcar, mapcan などの関数のように，リストの各要素に共通の部分処理がおこなわれることがおおい．このような処理は mapping だけでなく，再帰よびだしや do, dolist のようなループとして記述されるばあいもある．これらの部分処理は，論理型言語のばあいには一種の AND 関係にあるとみなすことができ，広義の AND 並列処理の対象となりうる．しかし，リストをたぐる処理には本質的な逐次性があり，パイプライン型ベクトル計算機においても，また通常のパイプライン型計算機においても，このような処理をパイプラインにのせて高速化をはかることは困難である．

このようなばあいに各部分処理の並列化をはかってパイプラインにのせる方法としては，第3～4章でしめした制御構造の交換にもとづく方法がある．しかし第1に，制御構造の交換をおこなうには多重のくりかえし構造がプログラム中に存在し，かつ外側のくりかえしがベクトル化に適したものでなければならない．また第2に，自動ベクトル化を目標とすると，Fortran や論理型言語の OR ベクトル化 / 並列バックトラック化にくらべると AND ベクトル化における制御構造の交換には困難な問題がある．すなわち，おおくのばあいひとつの手続き中にあらわれる2重ループを解析するだけで交換ができる Fortran にくらべると，論理型言語プログラムにおいては複数の手続きにまたがる再帰よびだしの解析が必要である．また，OR ベクトル化のばあいには外側くりかえしがユーザによって陽に記述されていないバックトラックであるのに対して，AND ベクトル化のばあいには外側くりかえしもユーザが再帰よびだしとして陽に記述されていなければならない．このため AND ベクトル化における制御構造の変換のためには，大域的なプログラムの構造をかえるおおがかりな変換が必要であり，それを自動化するのは非常に困難である．

これに対して，リストのデータ構造変換によるベクトル化方法は，上記のような問題がないため，AND ベクトル化への第1歩としてより適当だとかんがえられる．リストのデータ構造変換によるベクトル化とは，あらかじめリストをベクトルに変換 (データ構造変換) しておくことによって，最内側くりかえしをそのままベクトル処理することを

可能にする方法である．この方法によれば，制御構造の交換ができるばあいにつねにこの方法が適用できるわけではないが，逆に制御構造の交換ができないばあいでもベクトル処理可能になりうる．この章では，このようなベクトル処理法についてのべる．

7.2 節ではリストのデータ構造変換にもとづくベクトル処理法の原理をしめす．7.3 節ではこの方法を Prolog および GHC によって記述された素数生成のプログラムへの適用例をしめす．7.4 節では 7.3 節のプログラムを Fortran と Pascal とで記述されたプログラムに手動で変換して S-810 で実行した結果をしめす．7.5 節では，この研究におけるベクトル処理方法と関連研究における方法とを比較する．

7.2 データ構造変換にもとづくリストのベクトル処理法

前節でのべたように，リストの全要素に対する処理は，リストをたぐるという本質的に逐次性がある処理をともなうためにベクトル処理することができない．しかし，図7.1にしめすようにリストがあらかじめベクトルにデータ構造変換されていれば，ベクトル処理可能である．この変換は一種のCDRコーディング[Bawden 77]だとかんがえることができる．

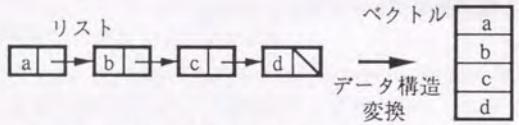


図7.1　ベクトル処理可能なデータ構造への変換

リストのデータ構造変換にもとづくベクトル処理法を，つぎにしめすプログラム7.1(リストの各要素の積和をもとめる手続き)を例として説明する．

プログラム7.1

```
mode muladd(+,+,+,-).
        %手続きmuladdの第1～3引数が入力，第4引数が出力である．
muladd([Ah | At], [Bh | Bt], [Ch | Ct], [Dh | Dt]) :-
   Dh is Ah * Bh + Ch, muladd(At, Bt, Ct, Dt).
muladd([], [], [], []).■
```

プログラム7.1をプログラム7.2のような中間語プログラムに変換すれば，部分的にベクトル処理可能になる．このようなAND並列処理のためのプログラムの変換をANDベクトル化とよぶ．プログラム7.2においては，ANDベクトル化後のプログラムを表現するためのベクトル中間語として，第6章で使用したのと同様の論理型言語中間語を使用している．

プログラム7.2

```
v_muladd(A, B, C, D) :-
   v_LtoV(A, VA), v_LtoV(B, VB), v_LtoV(C, VC),
   'v_*'(VA, VB, T), 'v_+'(T, VC, VD),
   v_VtoL(VD, D).■
```

ここで`v_LtoV`はリストをベクトルに変換する手続き，`v_VtoL`はベクトルをリストに変換する手続きであり，ベクトル中間語のくみこみ手続きとして用意されるべきものである．これらはベクトル処理できないので，スカラ処理(逐次処理)により実行される．

VA, VB, VCが変換結果のベクトルである．`'v_*'`と`'v_+'`とは，第6章でつかわれている同名の手続きと同様に，それぞれ第1引数および第2引数としてあたえられるベクトルの要素ごとの乗算，加算をおこなってその結果を要素とするベクトルを第3引数とする手続きであり，ベクトル処理される．

中間語プログラム7.2への変換ができれば，`'v_*'`と`'v_+'`とをベクトル命令に展開することは容易であり，自動的におこなうことができる．中間語プログラムへの変換のただしさは，リストの各要素に対する処理の独立性に依存している．プログラム7.1のばあいには各要素に関する処理が独立であることは容易にわかるから，この変換を強制ベクトル化オプションのようなユーザ指示なしに自動的におこなうことは可能である．しかし，処理の独立性を自動的に証明することは一般的には困難である[注1]．しかし，プログラム7.2という中間結果をへることによって，自動ベクトル化のための一歩はふみだすことができたとかんがえることができる．

プログラム7.2の実行の例として，入力が`A=[3,1,2], B=[1,3,2], C=[4,5,7]`のばあいをかんがえる．要素が$e_1, e_2, \ldots, e_n$であるベクトルを`#(`$e_1, e_2, \ldots, e_n$`)`とあらわすと，プログラム7.2の実行において`VA=#(3,1,2), VB=#(1,3,2), VC=#(4,5,7)`となる．`T=#(3,3,4), VD=#(7,8,11)`となり，したがって`D=[7,8,11]`となる．この結果はプログラム7.1の実行結果と一致する．

上記のようにしてリストの全要素に対する処理のベクトル化をはかることができるが，`v_muladd`のように各要素に対する処理が非常に短時間のばあいには，`v_LtoV`，`v_VtoL`のオーバヘッドのために実行時間はかえって増加してしまう．したがって高速化のためには，手続きをまたがるベクトル化をおこない，リストを入出力する手続きの入出力インタフェースを変更してベクトルを入出力する手続きに変換することが必須である．すなわち，`v_muladd`に関していえば，それをよびだす側でもデータ構造変換をはかることによって，`v_muladd`をよびだすごとにかかる変換オーバヘッドを1回ですませるようにすれば，高速化がはかれる可能性がある．この点に関しては，次節でさらにのべる．

[注1] この問題をさけるために，阿部ら[Abe 90b]はベクトル化のために`vmap`関数および`vmap`マクロという特別の機能をLispに用意している．阿部らの研究については7.5節でのべる．

7.3 Prolog による素数生成プログラムのベクトル処理

Prolog と GHC による素数生成プログラムを，7.2 節でしめした方針にしたがって手動ベクトル化した．まず Prolog 版の素数生成プログラムとそのベクトル化法について説明する．

2 以上 1536 未満の素数を生成して印刷するプログラムは Prolog によってプログラム 7.3 のように記述することができる．手続き tp をよびだすと，1536 未満のすべての素数からなるリストをつくってからそれを印刷する (1536 という数はインプリメントの都合できめたものであり，特別な意味はない)．したがって，すべての素数がもとまるまではいっさい印刷はおこなわれない．

プログラム 7.3：Prolog による素数生成プログラム

```
tp :- primes(S), write(S).                          % 主プログラム

primes(S) :- integers(2, I), sift(I, S).
        % integers によって整数列 I を生成し，sift で素数以外をフィルタ
        % する．

integers(From, []) :- From >= 1536, !.
integers(From, [From|Rest]) :- From < 1536, !,
    From1 is From + 1, integers(From1, Rest).

sift([], []) :- !.
sift([P | IR], [P | OR]) :- filter(IR, P, S), sift(S, OR).
        % filter によって，数列 IR にふくまれる P の倍数をフィルタし，
        % のこった整数の列を S とする．

filter([], _, []) :- !.
filter([H | IR], P, [H | OR]) :-
    H mod P =\= 0, !, filter(IR, P, OR).
filter([H | IR], P, OR) :-
    H mod P =:= 0, !, filter(IR, P, OR). ■
```

各手続きの意味の説明はプログラム中にしるした注釈をもってこれにかえる．プログラム 7.3 を手動ベクトル化してえられる論理型中間語プログラムはプログラム 7.4 のようになる．手続き v_integers, v_sift および v_filter は，変換前のプログラムにおけるリストのかわりに，いずれもベクトルを入出力する．上記のプログラムにおいては，これらの手続きの途中でデータをリスト構造にもどさないことによって，リスト－ベクトル間の変換オーバヘッドを減少させている．しかし，それ以外の点では手続きにまたがる変換はおこなわれておらず，上記の変換が基本的にもとのプログラムの大域的な構

造を保存していることに注意されたい．これは，リストからベクトルへの変換の正当性がしめされれば，上記の変換が自動化できる可能性がたかいことを示唆している[注2]．変換の正当性は自動的に証明することがのぞましいが，現在の技術のもとでは，これをユーザ・オプションとしてあたえるのが現実的である．

プログラム 7.4：手動ベクトル化後の Prolog 版素数生成プログラム

```
v_tp :- v_primes(VS), v_VtoL(VS, S), write(S).      % 主プログラム

v_primes :- v_integers(2, VI), v_sift(VI, VS).
        % integers をベクトル化したプログラムが v_integers, sift を
        % ベクトル化したプログラムが v_sift であり，VI, VS はベクトル
        % である．

v_integers(From, VI) :- v_iota(From, 1535, VI).

v_sift(#(), []) :- !.                    % 結果はリストとして出力する．
v_sift(#(P | IR), [P | OR]) :-
    v_filter(IR, P, VS), v_sift(VS, OR).
        % filter をベクトル化したプログラムが v_filter であり，VS は
        % ベクトルである．

v_filter(VI, P, VO) :- vs_mod(VI, P, T, _M),
    'vs_=:='(T, 0, _M, M), v_compress(VI, VO, M). ■
```

プログラム 7.4 においては，プログラム 7.3 における手続き tp, primes, integers, sift, filter をベクトル化してえられた手続きがそれぞれ v_tp, v_integers, v_sift, v_filter である．これらのうち，まず v_tp について説明する．プログラム 7.4 においては，前節でしめした手続き muladd とはちがって入力データのなかにリストは存在しない．したがって，手続き v_primes をよびだすまえにリストをベクトルに変換する必要はない．手続き v_primes をつかって素数列をもとめる．素数をもとめる過程ではベクトルを使用するが，それらのベクトルは出力されない．そして，素数列じたいはリストとしてもとめられる．それは，素数列がベクトル処理されることはなく，ベクトルに変換しても利点がないからである．したがって，もとめられた素数列をくみこみ手続き v_VtoL をつかってリストに変換する必要はない．このプログラムでは，最後に素数列をくみこみ手続き write をつかって印刷する．

手続き v_primes の構造は原始プログラムにおける手続き primes とかわらないので，その説明は省略する．

つぎに手続き v_integers について説明する．手続き v_integers は，素数列

[注2] ただし，手続き filter から v_filter への変換は容易でなく，このような変換の自動化は今後の研究課題である．

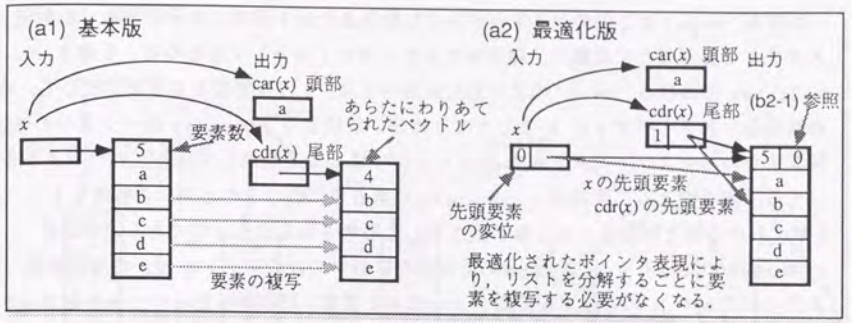
[2,3,4, ...,1535] をベクトルとして生成する手続きである．v_iota は算術数列を生成する手続きであり，ベクトル中間語のくみこみ手続きとして用意されるべきものである．v_iota はベクトル計算機がもつ数列生成命令 (S-810 / S-820 においては VINC 命令) を使用することによってベクトル処理される．

手続き v_sift の構造は原始プログラムにおける手続き sift とほとんどかわらないが，入力としてリスト [], [P|IR] などのかわりにベクトル #(), #(P|IR) などを使用している．ここで #() は空ベクトル，#(P|IR) は先頭要素が P でのこりの要素からなるベクトルが IR であるベクトルをあらわす．たとえば，#(P|IR) = #(2,3,5) のとき P = 2, IR = #(3,5) となる．素数生成のベクトル処理プログラムにおいては，このようにベクトルが先頭要素と後続要素からなるベクトルに分解される (もとのプログラムにおいてはリストの分解すなわち Lisp の car, cdr) が，その逆の演算 (もとのプログラムにおいてはリストの合成すなわち Lisp の cons) は存在しない．これらの AND ベクトル化されたリストの基本演算の処理方法を，素数生成ではつかわれないリスト合成までふくめて図 7.2 にしめす．図 7.2 には単純な方法と最適化された方法の両方をしめしている．

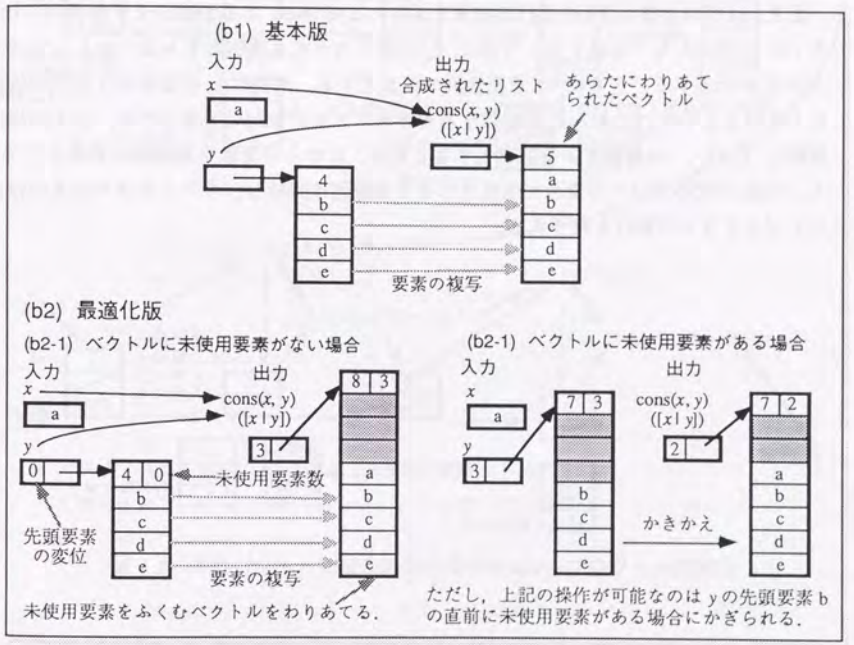
手続き v_sift の出力には変換前のプログラムと同様にリストを使用する．これは，この出力データに関してはベクトル処理がおこなわれない (ベクトル処理によって高速化されない) からである．

つぎに手続き v_filter について説明する．手続き v_filter は，第 2 引数である整数 P の倍数である要素を第 1 引数であるベクトルから削除したあらたなベクトルをつくってそれを第 3 引数 VO とする手続きである．ベクトル VI, P, T および _M のベクトル長はひとしく，各要素が対応している．手続きよびだし vs_mod(VI,P,T,_M) は，ベクトル VI の各要素を P でわり，そのあまりを要素とするベクトルをもとめて T とする．手続き vs_mod はベクトル除算命令を使用することによってベクトル処理される[注3]．手続き vs_mod のよびだしにおける第 4 引数 _M は入力マスク・ベクトルであり，vs_mod の実行開始前にはすべての要素が *true* であるマスク・ベクトルをあらわしている．

[注3] 手続きよびだし vs_mod(VI,P,T,_M) の実行においては，除数ベクトル VI のすべての要素の値がきまっていなければ 1 回のベクトル命令の実行で計算することができず，したがって効率的に処理することができない．VI はベクトルくみこみ手続きによって生成されるため，この条件がみたされる．そのため，7.4 節で実測結果をしめす手動コンパイルしたプログラムにおいては，部分的に未束縛のばあいは考慮せず，変数 VI じたいが未束縛のばあいと VI が完全に束縛されたベクトルに束縛されているばあいについてだけ処理をおこなっている．上記の処理方法は vs_mod だけでなく，'vs_=:=' などのベクトル中間語の他のすべてのくみこみ手続きについても同様である．プログラム 7.4 のばあいには，すべてのベクトルくみこみ手続きについて，その入力引数が完全に未束縛であるか完全に束縛されているかのいずれかであるという条件がなりたっているため，効率的にベクトル処理することができる．



(a) リストの分解



(b) リストの合成

図 7.2 CDR コーディングされたリストに対する基本演算

手続き 'v=:=' は2個のベクトルがふくむ数値または1個のベクトルがふくむ数値とスカラとを要素ごとに比較し，結果をマスク・ベクトルとしてもとめる．手続き v_filter における 'v=:=' のよびだしにおいては，T の各要素と 0 とを比較して，その結果をマスク・ベクトル M としてもとめる．手続き v_compress はベクトルの無効要素すなわちマスク・ベクトルの *false* に対応する要素を削除して圧縮されたベクトルをつくる手続きである．手続き v_compress は第6章におけるのと同一の機能をもち，第6章におけるのと同様にくみこみ手続きとして用意されるべきものである．手続き v_filter における v_compress のよびだしにおいては，ベクトル VI の有効要素 (マスク・ベクトル M の対応する要素が *true* である要素) だけからなるベクトルをつくり，VO とする．

図7.3はプログラム7.4の実行の様子をあらわしている．すなわち，まず整数列をベクトル I_1 のかたちで生成する．つぎに，I_1 の要素のうち2の倍数をふるいおとしてベクトル I_2 を生成するとともに，2を素数列の要素とする．同様に I_2 の要素のうち3の倍数をふるいおとしてベクトル I_3 を生成するとともに3を素数列の要素とする．以下同様に素数 5, 7, 11, ... の倍数をふるいおとすとともに，これらの素数を素数列の要素としていく．1536未満のすべての素数が生成されると素数列は完結し，すべてのプロセスが停止してプログラムの実行も終了する．

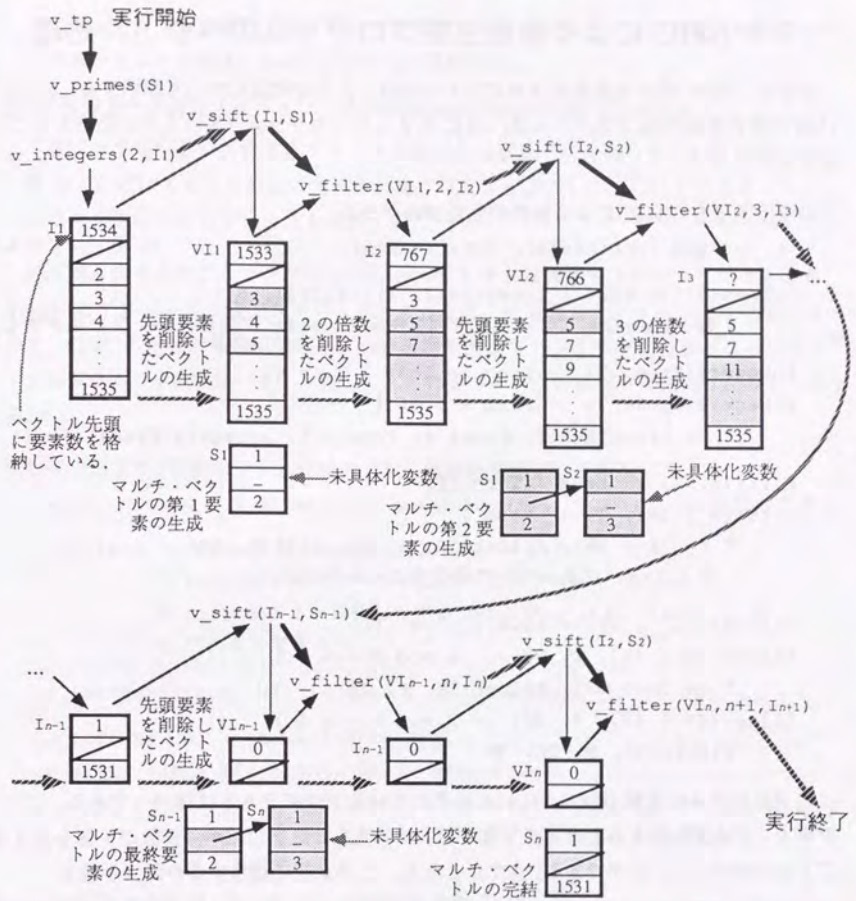


図7.3　手動ベクトル化後の素数生成の Prolog プログラムの実行

7.4 GHC による素数生成プログラムのベクトル処理

つぎに，GHC 版の素数生成プログラムとそのベクトル化法について説明する．2 以上 1536 未満の素数生成プログラムは，GHC によってプログラム 7.5 のように記述することができる．

プログラム 7.5：GHC による素数列生成プログラム

```
tp :- true | primes(S), outstream(S).                    % 主プログラム

primes(S) :- true | integers(2, I), sift(I, S).
      % integers によって整数列を生成し，sift で素数以外をフィルタする．

integers(From, T) :- From >= 1536 | T := [].
integers(From, T) :- From < 1536 |
     T := [From|Rest], From1 is From + 1, integers(From1, Rest).

sift([], T) :- true | T := [].
sift([P | IR], T) :- true |
     T := [P | OR], filter(IR, P, S), sift(S, OR).
      % filter によって P の倍数をフィルタする．

filter([], _, T) :- true | t := [].
filter([H | IR], P, T) :-  H mod P =\= 0 |
     T := [H | OR], filter(IR, P, OR).
filter([H | IR], P, OR) :- H mod P =:= 0 |
     filter(IR, P, OR). ■
```

このプログラムは文献 [Fuchi 87] にしめされているプログラムとほぼ同一である．プログラム 7.5 は素数列をもとめる点ではプログラム 7.3 とおなじだが，素数が 1 個もとまるごとに印刷する (ことができる) 点でことなる．このように動作するのは，手続き primes, outstream, sift, filter の各よびだし (ゴール) およびそれらの下請けの手続きよびだしのすべてが並行プロセスとして動作するためである．

GHC で記述されたプログラムを逐次計算機によって実行するばあい，並行プロセスとして生成された各手続きよびだしをただひとつのスケジュラが集中的に管理することになるが，そのスケジュリングの方法として，つぎのようなものがある [Fuchi 87].

(1) 幅優先スケジュリング (breadth-first scheduling)

スケジュラがつぎに実行するプロセス (手続きよびだし) を選択するばあいに，もっともふるく生成 (プロセス・キューに投入) されたプロセスを選択する．

(2) ふかさ優先スケジュリング (depth-first scheduling)

スケジュラがつぎに実行するプロセスを選択するばあいに，もっとも最近に生成 (プロセス・キューに投入) されたプロセスを選択する．

(3) N 有界ふかさ優先スケジュリング (N-bounded depth-first scheduling)

スケジュラがつぎに実行するプロセスを選択するばあいに，あらかじめきめられた回数 $N-1$ だけもっとも最近に生成されたプロセスを選択し，N 回以上になるともっともはやく生成されたプロセスを選択する[注4]．

N 有界ふかさスケジュリングにおいて $N=1$ とすれば幅優先スケジュリングとなり，$N=\infty$ とすればふかさ優先スケジュリングとなる．N 有界スケジュリングは効率がよいので，GHC の逐次計算機用処理系における標準的なスケジュリング方式となっている[注5]．

プログラム 7.5 を手動ベクトル化してえられるプログラムはプログラム 7.6 のようになる．

プログラム 7.6：手動ベクトル化後の GHC 版素数生成プログラム

```
v_tp :- true | v_primes(S), outstream(S).                % 主プログラム

v_primes :- true | v_integers(2, I), v_sift(I, S).
        % integers をベクトル化したプログラムが v_integers,
        % sift をベクトル化したプログラムが v_sift であり，I は
        % ベクトルである．

v_integers(From, VI) :- true | mv_iota(From, 1535, VI).

v_sift([], O) :- true | O := [].
v_sift([#(P | I1) | It], O) :- true |
     O := [P | OR], v_filter([I1 | IR], P, S), v_sift(S, OR).
        % filter をベクトル化したプログラムが v_filter であり，S は
        % ベクトルである．
v_sift([#() | I], O) :- true | v_sift(I, O).

v_filter(VI, P, VO) :- true |
     mv_newmask(VI, _M),                % 空のマスク・ベクトルを生成する．
     mvs_mod(VI, P, T, _M),
     'mvs_=:='(T, 0, _M, M), mv_compress(VI, VO, M). ■
```

GHC 版のプログラム (プログラム 7.6) と Prolog 版のプログラム (プログラム 7.4) とのちがいはつぎのとおりである．

[注4] ただし，リダクションによってプロセスが生成されないばあいは，もっとも最近に生成されたプロセスを N 回選択するよりもまえに，もっともはやく生成されたプロセスの選択にうつる．

[注5] もちろん N 有界ふかさ優先スケジュリングのさまざまな変種をかんがえることができる．

(1) マルチ・ベクトルの使用

プログラム 7.6 においては，ベクトルを要素とするリストを使用している．第 6 章でものべたように，このようなリストをマルチ・ベクトルとよび，マルチ・ベクトルを構成する各ベクトルを部分ベクトルとよぶ．マルチ・ベクトルは図 7.4 にしめすように，その部分ベクトルのすべての要素を要素とするベクトルと等価である．したがって，データ構造変換されたプログラムにおいては，マルチ・ベクトルはその部分ベクトルのすべての要素を要素とするリストを表現している．

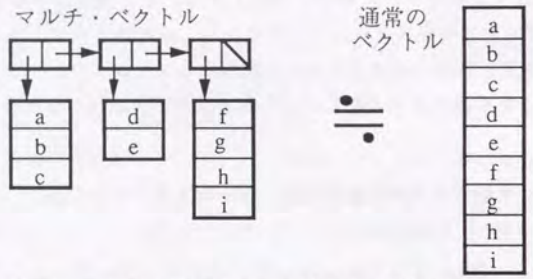


図 7.4　マルチ・ベクトルおよびそれと等価なベクトル

GHC プログラムの変換後のプログラムでは，Prolog プログラムの変換後のプログラム 7.4 で接頭辞 v_ または vs_ を冠した手続きを使用していたところで接頭辞 mv_ または mvs_ を冠した手続きを使用しているが，これはマルチ・ベクトルを入出力する演算である．これらの手続きの機能は，リストの表現としてマルチ・ベクトルを使用し，部分ベクトルを単位として計算をおこなうという点以外は，プログラム 7.4 で使用している対応する手続きとひとしい．たとえば mvs_mod の意味はつぎの GHC 風手続きで表現される．

```
mvs_mod([], _, O, []) :- true | O := [].
mvs_mod([X1 | Xt], S, O, [M1 | Mt]) :- true | O := [Z1 | Zt],
  vs_mod(X1, S, Z1, M1), mvs_mod(Xt, S, Zt, Mt).
```

この表現からわかるように，各部分ベクトルについての処理 (vs_mod のよびだし) は

非同期に (ただし実際には先頭から順に) おこなわれる[注6]．

マルチ・ベクトルがこのように第 6 章でのべた OR ベクトル処理と AND ベクトル処理の両方でつかわれることは，ベクトル記号処理におけるマルチ・ベクトルの重要性をしめしているとかんがえられる．したがって，マルチ・ベクトルに関しては第 8 章でそのよりひろい定義をあたえるとともに，よりくわしく考察する．

(2) **v_sift の構造と処理**

マルチ・ベクトルを使用したために，手続き v_sift の構造をかえる必要が生じている．すなわち，入力するリストが空のばあいの処理をおこなう節は (マルチ・ベクトルを部分ベクトルのリストで実装しているばあいには) もとのままでよいが，入力するリストが空でないばあいの処理をおこなう節は，つぎのように変換する必要がある．すなわち，リストの先頭要素 P をもとめるのにマルチ・ベクトル先頭の部分ベクトルの最初の要素をとる ([#(P|I1) | It]) ようにするとともに，マルチ・ベクトル先頭の部分ベクトルが空のばあいの処理をおこなう節を追加している．

(3) **v_filter におけるマスクの初期設定**

中間語プログラムも並列論理型言語の意味論にしたがうので，手続き v_filter においてはマスク・ベクトル _M の値の初期設定が必須である．そのため，くみこみ手続き mv_newmask をよびだしている．手続きよびだし mv_newmask(VI,_M) は VI と同一の構造のマルチ・ベクトルをつくり，各部分ベクトルの要素をすべて *true* にする．プログラム 7.5 は，ベクトル計算機においてつぎのように処理される．各部分ベクトルはパイプライン処理されるが，そのほかは逐次的に実行される．すなわち，真の並列処理はおこなわれない．GHC の逐次計算機用処理系は，前記した幅優先スケジュリングあるいは N 有界ふかさ優先スケジュリングをおこなうために，プロセスの列であるスケジュリング・キューを使用する．ベクトル長の最大値をきめ，これを上限ベクトル長とよぶ．上限ベクトル長を N (< 1536) とすると，くみこみ手続き mv_iota においてはまず N 個の整数をベクトルとして生成し，後続の整数を生成するプロセスは

[注6] 手続きよびだし mvs_mod(VI,P,T,_M) の実行においては，除数マルチ・ベクトル VI の先頭の部分ベクトルすべての要素の値がきまっていなければ 1 回のベクトル命令の実行でその部分ベクトルの計算を終了することができず，したがって効率的に処理することができない．VI はベクトルくみこみ手続きによって生成されるため，この条件がみたされる．そのため，7.4 節で実測結果をしめす手動コンパイルしたプログラムにおいては，部分ベクトルが部分的に未束縛のばあいは考慮していない．上記の処理方法は mvs_mod だけでなく，'mvs_=:=' などのベクトル中間語の他のすべてのくみこみ手続きについても同様である．プログラム 7.6 のばあいには，すべてのベクトルくみこみ手続きについて，その入力引数の部分ベクトルが完全に未束縛であるか完全に束縛されているかのいずれかであり，先頭から順に束縛されるという条件がなりたっているため，効率的にベクトル処理することができる．

スケジュリング・キューに投入される．すなわち，Prolog プログラムのばあいとはちがって，1535 までの整数を一度に生成するのではなく，N 個をこえる整数はおくれて生成される．このプロセス・スケジュリング戦略はほぼ N 有界ふかさ優先スケジュリングにしたがっている．すなわち，N 個の整数はふかさ優先で生成され，のこりの整数に関しては幅優先で生成される．整数の生成だけでなく，ふるいの処理も同様にほぼ N 有界ふかさ優先スケジュリングにしたがっておこなわれる．したがって，生成された整数に関するふるいの処理と後続の整数に対する処理は，混合されて実行される．ただし，ふるいがすすむにつれてみじかくなったベクトルを単位としてベクトル処理がおこなわれるため，ふかさ優先でのくりかえし回数は N よりみじかくなる．

ところで，上記のように素数生成においてははじめは十分なベクトル長をとっていても処理がすすむにつれて素数でない整数がベクトルからふるいおとされるため，ベクトル長が短縮する．そのためにベクトル処理性能が低下するという問題がある．この問題を解決するため，ベクトル化後の `filter` 手続きの末尾において，ベクトル長をしらべて，一定数以下のときはとなりあう部分ベクトルを併合する処理をおこなうようにした．この処理を部分ベクトル併合処理とよぶ[注7]．すなわち，ある部分ベクトルの処理をおこなう際に，ベクトル長がみじかいばあいには後続の部分ベクトルをしらべ，それがすでに具体化されているばあいには当該の処理をおこなう．まだその部分ベクトルが具体化されていないばあいには，その処理をおくらせる (すなわち，ふたたびスケジュリング・キューに投入する)．部分ベクトル併合処理をおこなう最大のベクトル長を下限ベクトル長とよぶ．部分ベクトル併合をおこなうばあいの最大ベクトル長は，素数列生成時における部分ベクトルのベクトル長をきめるのとおなじ上限ベクトル長で規定される．

[注7] GHC 版の中間語プログラムを Fortran および Pascal のプログラムに変換するときに，この処理を追加するようにした．

7.5 実行結果

この節では Prolog 版および GHC 版の素数生成プログラムの S-810 および M-680H IAP / IDP による実行結果をしめす．まず 7.5.1 節では測定のために開発したプログラムについてかんたんにのべる．7.5.2 節ではスカラ処理とベクトル処理による実行時間を比較し，7.5.3 節では部分ベクトル併合の効果をしめすための測定結果についてのべる．

7.5.1 測定に使用したプログラムについて

7.3 ～ 7.4 節でしめした手動ベクトル化した Prolog 版と GHC 版の素数生成プログラムをさらに手動で Pascal プログラムに変換し，Fortran, Pascal およびアセンブリ言語によって記述した実行支援系[注8]と静的に結合してベクトル計算機 S-810 と内蔵ベクトル処理機構 (IAP および IDP) つき汎用計算機 M-680H とでそれぞれ実行した[注9]．ベクトル処理する部分は，Fortran およびアセンブリ言語によって記述された部分だけであり，他の部分はすべてスカラ処理される．ベクトル中間語をもとにして生成したプログラムをできるかぎりベクトル処理したばあいの性能と，すべてスカラ処理したばあいの性能を両方測定して比較した[注10]．スカラ処理むきに最適化された実行方式に関しては，ベクトル処理方式と比較しうる測定をおこなっていないが，ベクトル処理むきのプログラムをスカラ処理しているため，スカラ処理むきに最適化された実行方式よりはこの測定におけるスカラ処理方式のほうがやや性能がわるいものとかんがえられる．

この測定で使用した実行系においては，部分ベクトルの併合をおこなっている．また，図 7.2 にしめした最適化されたリスト表現を使用している．

7.5.2 全体性能

1536 まで素数生成に要した時間と推論性能，スカラ処理方式と比較しての加速率を表 7.1 にしめす．ベクトル処理においてはスカラ処理にくらべて Prolog 版で 4.4 倍，GHC で 2.9 倍の性能がえられているが，十分な加速率がえられているとはいえない．その原因はベクトル化率が不十分であること，すなわちベクトル化後のプログラムにおいてもスカラ処理されている部分がおおいことにあるとかんがえられる[注11]．とくに GHC 版のばあいは，ベクトル化によって高速化がのぞめないスケジュリングがスカラ処理されて

[注8] 実行支援系は `vs_mod` などのベクトルくみこみ手続きの定義をふくんでいる．

[注9] アセンブリ言語を使用したのは，他の言語によってサポートされていない M-680H IDP を使用するためであり，それ以外の部分では使用していない．

[注10] S-810 版においては，ベクトル処理される部分はすべて Fortran によって記述されているため，ベクトル処理とスカラ処理との差は Fortran コンパイラでコンパイルするときのオプションのちがいだけである．M-680H IDP 版においては，Fortran とアセンブリ言語によって記述されたプログラムじたいもことなる．

[注11] 測定困難なために，ベクトル化率の測定はおこなっていない．

いること，スケジュラの最適化が十分でないことなどがProlog 版にくらべてベクトル化率をさげているとかんがえられる．

表 7.1　1536 までの素数生成の性能比較 (S-810/20)

プロセッサ	処理方式	Prolog 版			GHC 版		
		時間 (ms)	性能 (MLIPS*)	加速率	時間 (ms)	性能 (MLIPS*)	加速率
S-810	スカラ	71	0.5	-	84	0.4	-
	ベクトル	16	2.1	**4.4**	29	1.2	**2.9**
M-680H	スカラ	30	1.1	-	38	0.9	-
	IAP + IDP*	18	1.9	1.7	24	1.4	1.6

* MLIPS = Million Logical Inference Per Second．1秒あたりに実行されたリダクションの回数をあらわす．ただしくみこみ手続きの実行回数はふくまない．総リダクション数は Prolog 版，GHC 版ともに 33.8k．なお，印字に要する時間はのぞいてある．

7.5.3 部分ベクトル併合の効果

部分ベクトル併合による平均ベクトル長と実行時間の変化の測定結果を表 7.2 にしめす．7.4 節でものべたように，表 7.2 において上限ベクトル長とは `v_filter` 手続きでベクトルを生成する際のベクトル長のことであり，下限ベクトル長とは部分ベクトル併合をおこなう最大のベクトル長のことである．部分ベクトル併合をおこなうばあいの最大ベクトル長も上限ベクトル長で規定される．下限ベクトル長が 0 のばあいは，すなわち部分ベクトルの併合をおこなわないばあいである．

上限ベクトル長が 256 以下のときは部分ベクトルの併合によってベクトル長は3 ～ 7 倍になり，加速率も向上しているためかなり効果があるということができるが，上限ベクトル長が 512 のときはあまり効果がない．これは，部分ベクトル併合をしなくても平均ベクトル長がかなり長いからである．また，部分ベクトルの併合はプロセスのサスペンド回数をふやし (なぜなら，併合が可能になるまで待つ必要があるから)，したがってスカラ処理されるスケジュリング処理の比率をふやすため，ベクトル長がおおきくなったわりには性能が向上しない．これが，部分ベクトル併合の効果をさげる原因となっているとかんがえられる．しかし，より急速にベクトル長が短縮するプログラムにおいては，上限ベクトル長が512 以上でも効果があるとかんがえられる．

表 7.2 GHC 版における部分ベクトル併合の効果 (S-810 で測定)*

下限ベクトル長		上限ベクトル長		
		128	256	512
0	平均ベクトル長 [非併合のばあいとの比]	10.4 [1.0]	19.7 [1.0]	61.2 [1.0]
	時間 (加速率) [非併合のばあいとの比]	87.2 (1.2) [1.0]	51.9 (1.5) [1.0]	23.8 (2.2) [1.0]
50	平均ベクトル長 [非併合のばあいとの比]	66.4 [6.4]	68.7 [3.5]	- -**
	時間 (加速率) [非併合のばあいとの比]	25.3 (2.1) [0.29]	23.9 (2.2) [0.46]	- -
200	平均ベクトル長 [非併合のばあいとの比]	- -	- -	94.5 [1.5]
	時間 (加速率) [非併合のばあいとの比]	- -	- -	22.2 (2.3) [0.93]

* 表 7.1 とは測定条件が一部ことなるため，それと一致する測定値はこの表にはない．

** "-" をしるした点は測定をおこなっていない．

7.6 関連研究との比較

データ構造変換によるリストのベクトル処理をめざした研究としては阿部ら [Abe 90a, Abe 90b]，小林ら [Kobayashi 91] などがある．阿部らの方法では，ベクトル処理専用の vmap 関数，vmap マクロの使用によりデータ構造変換をおこなう点をユーザが指定する．したがって自動ベクトル化をめざした研究ではない．

阿部らとくらべたばあい，この研究における方法の特徴はつぎのようにまとめることができる．

(1) 対象言語

阿部らは Lisp を対象言語としているが，この研究は論理型言語を対象としている．とくに，GHC を対象言語としていることにより並列処理言語を対象としているが，阿部らは逐次処理言語を対象としている．

(2) 中間語の設定

阿部らはプログラムの変換法についてくわしくのべていないが，この研究におけるような中間語は設定していない．すなわち，原始プログラムを直接コードに変換することをかんがえている．阿部らの vmap 関数 / マクロを追加した Lisp にくらべるとこの研究における論理型中間語は低水準に設定されている．たとえば，ベクトルを圧縮するための手続き v_compress が陽にあらわれている．これにより低水準中間語レベルでの最適化が可能になり，したがって阿部らの方法より最適化が容易であり，たとえば v_compress の使用によって無効演算をへらせる可能性がある．

(3) マルチ・ベクトルの使用

この研究では，マルチ・ベクトルというデータ構造を使用している．これはおもに (1) に起因する相違点だとかんがえられる．マルチ・ベクトルを使用することにより，部分ベクトルの併合などという課題も生じた．

(4) 阿部らがすぐれている点

おもに対象言語のちがいにより，この研究と阿部らの研究の水準をいちがいに比較することはできないが，この研究にくらべて阿部らの研究がすぐれているのは，つぎのような点だとかんがえられる．阿部らは自動ベクトル化をめざさないかわり，ユーザにも vmap 関数 / マクロの使用というかたちで負担させることにより，無理のないベクトル化をはかっているということができる．この研究でもユーザ・オプションを利用したベクトル化をかんがえているが，ユーザ・オプションがプログラムを複雑化させるということは否定できない．

Connection Machine Lisp [Hillis 85, Hillis 90] は，SIMD 型並列計算機のための Lisp だが，ベクトル処理[注12]専用の関数を用意しているという点において，またその一部の機能において阿部らのアプローチにちかい．ただし，阿部らよりははるかにおおくの専用関数をもうけている．

また，GHC プログラムのベクトル計算機および SIMD 型並列計算機による実行という点で同一の目的をめざした研究として Nilsson による研究 [Nilsson 89] がある．Nilsson の実行方法は，リダクションや任意の演算をおこなうあらゆるプロセスからなるベクトルをつくり，多種類の演算を一度に実行するものである．特定の演算がほどこされるデータだけからなるベクトルをつくるこの研究の方法にくらべると適用範囲はひろい[注13]が，ベクトル計算機においては一度に一種類の演算しか実行することができないので，むだがおおく，たかい加速率をえることがむずかしいという問題点がある．これに対して，この研究は (第 6 章でのべた OR ベクトル化もふくめて) 適用範囲をかぎるかわりにたかい加速率をえることを可能にしている．

[注12] Connection Machine においてはデータ構造として Vector ではなく Xector とよばれるものがつかわれるから，より正確には Xector 処理とよぶべきであろう．

[注13] 理論的には任意の GHC プログラムをベクトル処理することができる．

7.7 まとめ

リストを CDR コーディングすることにより，論理型言語で記述されたプログラムをベクトル処理する方法をしめした．この方法によって，すくなくとも素数生成の Prolog プログラムおよび GHC プログラムにおいては実行性能を向上させることができることがわかった．この方法においては原始プログラムをまず論理型中間語表現に変換するが，この中間語表現からはベクトル処理プログラムに自動的に変換することができる．中間語プログラムへの変換はいまのところ自動化できないが，手動で変換することを可能にし適当な中間語をみいだしたことによって，自動化に一歩ちかづいたものとかんがえることができる．とくに，変換後のプログラムにおいて使用するマルチ・ベクトルというデータ構造ベクトル記号処理において重要なものであり，また素数生成プログラムにおいても効果がみられた部分ベクトルの併合処理は重要だとかんがえられる．この点については第 8 章でさらにのべる．さらに，この方法は Prolog や GHC 以外の言語で記述されたプログラムにも適用できるとかんがえられる．

今後の課題としては，第 1 に，素数生成のベクトル処理プログラムにおけるオーバヘッドを減少させて，加速率の向上をはかることがあげられる．第 2 に，リストの CDR コーディングにもとづくベクトル化を素数生成以外のプログラムにも適用をこころみることがあげられる．すなわち，素数生成のプログラムにおいては CDR コーディングが比較的容易だったが，より適用がむずかしい他のばあいにも適用をこころみ，それによってこの方法の発展をはかることである．容易ではないが，このような研究をつうじて自動ベクトル化をめざすことがより究極の目的である．

第 8 章
論理プログラムの自動ベクトル化処理系とその中間語

要旨

第 6 章における逐次論理型言語の OR ベクトル化方式にもとづいて，モード宣言つきの Prolog プログラムを機械語にコンパイルして HITAC S-810 上で実行する自動ベクトル化処理系を試作した．この章では試作処理系の構造および中間語の概要についてのべる．この処理系によってベクトル化できるプログラムの範囲はかぎられており，また Prolog 処理系としても不完全ではあるが，この試作によって *N* クウィーン問題をはじめとする一部の論理型言語プログラムが自動ベクトル化によって高速にベクトル処理可能な目的プログラムに変換できることが実証された．この処理系の基本構造および中間語は，OR ベクトル化だけではなく，AND ベクトル化のばあいにも，ほぼそのまま適用できるとかんがえられる．

8.1 はじめに

この章では Prolog にちかい逐次論理型言語のベクトル計算機のためのプロトタイプ処理系 Pilog/HAP についてのべる．すなわち，逐次論理型言語処理系における論理型言語プログラムのベクトル化法および実行方法を説明する．Pilog/HAP によって N クウィーン問題のプログラムをはじめとするいくつかの Prolog プログラムを自動ベクトル化することができることが実証された．しかし，Pilog/HAP は完全な論理型言語処理系として完成されてはいない．すなわち，すべての Prolog プログラムがベクトル化できるわけではもちろんないし，さらに，コンパイルおよび実行が可能なのはかぎられた範囲の Prolog プログラムだけである．なお，最後に逐次論理型言語の AND ベクトル化および並列論理型言語のばあいの補足をする．

8.1.1 ベクトル化法の分類

論理型言語のベクトル化方式の分類については第6章でもかんたんにのべたが，ここでもういちどまとめておく．

❑ OR 関係と AND 関係

OR 関係と AND 関係をつぎのように定義する[注1]．

◆ OR 関係

ひとつの手続きよびだし p によって起動されうることなる節 $p1$, $p2$ のなかから直接または間接によびだされておこなわれる計算 (手続きよびだしまたはユニフィケーション) は OR 関係にあるという．

◆ AND 関係

ひとつの手続きよびだし p によって起動されるひとつの節のなかのことなる部分から直接または間接によびだされておこなわれる計算 (手続きよびだしまたはユニフィケーション) は AND 関係にあるという．

❑ OR ベクトル化方式

上記の意味における OR 関係にある複数の演算の一部をベクトル処理に変換する方式を OR ベクトル化方式とよぶ．OR ベクトル化方式はさらにつぎの2つに分類される．

◆ 完全 OR ベクトル化方式

原始プログラムに存在するふかいバックトラックすなわち複数のユーザ定義手続きにまたがるバックトラックをすべてなくし，すべての解を同時にもとめるプログラムに変換する方式である．

[注1] この定義はかならずしも通常の定義と一致しない．

◆ 並列バックトラック化方式

原始プログラムに存在するふかいバックトラックの一部はのこし，解の一部だけを同時にもとめるプログラムに変換する方式である．第2章でのべたように，完全 OR ベクトル化方式のばあいに問題となるくみあわせ爆発 (combinatorial explosion) をなくし，完全 OR ベクトル化方式による単解探索の効率のわるさを改善することができる．

❑ AND ベクトル化方式

上記の意味における AND 関係にある複数の演算の一部をベクトル処理に変換する方式を AND ベクトル化方式とよぶ．

OR ベクトル化は一種の OR 並列性をひきだす変換であり，AND ベクトル化は一種の AND 並列性をひきだす変換である．

8.1.2 処理系の構造

試作したベクトル計算機のための論理型言語処理系における処理方法を図 8.1 にしめす．この処理方法によれば，論理型言語で記述された原始プログラムを2フェーズで目的プログラムにコンパイルしたのち，実行する．すなわち，ベクトル演算を記述することができる機械独立な中間語を設定しておき，まずフェーズ1では中間語プログラムを生成する．つぎに中間語プログラムからよびだされるベクトル処理およびスカラ処理のくみこみ手続きをあわせてコード生成し，目的プログラムをえる．この処理方式においてもっとも重要なのはフェーズ1のベクトル化 (プログラム変換) の機能と手順，およびその対象言語としてのベクトル中間語の設計である．なお，Pilog/HAP においては，このような機械語による実行のほかに，Prolog 上での中間語のシミュレーションという実行方法も用意している．シミュレータを用意した理由などについては 8.4 節でのべる．

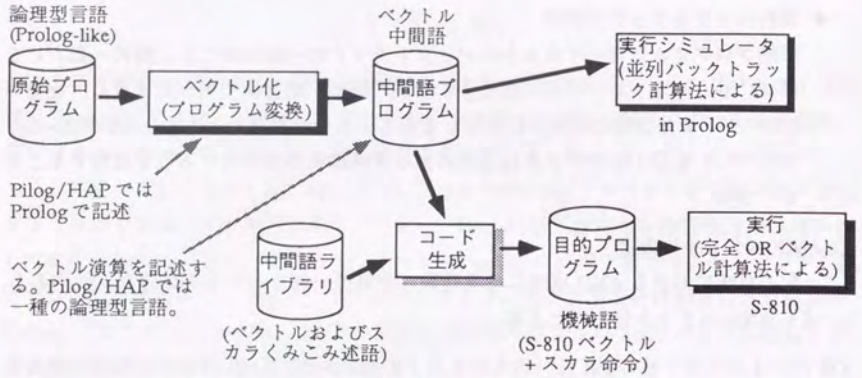


図 8.1　ベクトル計算機のための論理型言語試作処理系 Pilog/HAP における処理方法

8.2 ベクトル中間語

この節では，前節でしめした実装上の重要事項のうち，ベクトル中間語としてどのような言語を選択するかという問題についてのべる．また，Pilog/HAP で採用した論理型中間語を構成する要素すなわち手続きよびだし (述語よびだし) あるいは命令の機能について説明する．

8.2.1 ベクトル中間語の選択：論理型言語か関数型言語か

Fortran などの言語においては原始プログラムの意味を形式的に表現することも非常にむずかしく，したがってベクトル化をある種の等価変換として形式的にあらわすことものぞみえないことだとかんがえられる．しかし，この研究の対象となっている論理型言語のばあいには，カットなどをつかわないかぎりは原始プログラムの意味を形式的に表現することができる．したがって，ベクトル化をある種の等価変換として形式化できるのぞみもたかいとかんがえられる．現在はこのような形式化はできていないが，そこにちかづくためには，ベクトル化後のプログラムを表現するための中間語は，機械独立であって意味を簡潔かつ明確に規定しやすい形式であることがのぞましい．このような条件をみたし中間語の候補となるのは，論理型言語と関数型言語である．

第 6 章でのべたように中間語として FP [Backus 78] に類似した一種の関数型言語を使用することも一時は検討していたが，試作処理系 Pilog/HAP においては論理型言語を採用した．たとえば，図 8.2 a のような原始プログラムに対する関数型中間語は図 8.2 b，論理型言語中間語は図 8.2 c または d のとおりである (図 8.2 d が Pilog/HAP で採用した中間語である．図 8.2 c の中間語については 6.7 節および 6.10 節において説明した)．

(a) 原始プログラム (リスト連接のプログラム)

```
append([], Y, Y).
append([H | X], Y, [H | Z]) :- append(X, Y, Z).
```

(b) 関数型中間語

```
v_append = (v_append1 & (v_append ° v_append2)).
                                    —"&" はストリームの連接, "°" は関数合成.
v_append1 = map(#([], Y, Y), #([], Y, Y)).  —"map" は "apply all" の意味.
v_append2 = map(#([H | X], Y, [H | Z]), #(X, Y, Z)).
```

(c) 論理型中間語 ver. 1

```
v_append(Out:OutE, In) :-
   v_finished(In) -> Out = OutE
 ; v_copy(In1, In), v_append_1(Out:Out1, In1),
   v_copy(In2, In), v_append_2(Out1:OutE, In2).

v_append_1(DOut, In) :- v_filter(DOut, In, append_1).

append_1(Env, #([], Y, Y | Env)).

v_append_2(DOut, In) :-
   v_filter(T1:T1E, In, append_2), v_finished(T1E),
   v_append(DOut, T1).

append_2(#(X, Y, Z | Env), #([H | X], Y, [H | Z] | Env)).

?- v_append(#(S): #(#()), #(#(X), #(Y), #([1, 2, 3]), #((X, Y)))).
```

(d) 論理型中間語 ver. 2 (Pilog/HAP)

```
v_append(X, Y, Z, MI, MO) :-
   v_finished(MI), !
 ; v_null(X, MI, M1),
   v_assign(Z, Y, MO1),
   v_or(MI, MO1, M2),
   v_car(H, X, M2, M3),
   v_cdr(R, X, M3, M4),
   v_append(R, Y, R1, M4, MO2),
   v_cons(H, R1, Z, MO2),
   v_end_or(MO1, MO2, MO).
```

図 8.2　関数型中間語と論理型中間語の例

論理型中間語を採用した理由はつぎの4点である.

❑ 形式的とりあつかいの可能性

現時点では論理型言語のベクトル化を, fold / unfold 変換におけるように形式化されたかたちで記述することはできない. しかし, 将来これを形式的にあつかえるように準備しておくことは重要だとかんがえられる. 形式的なとりあつかいのためには, 手続き型言語や Lisp のような言語で記述するより, 論理型言語で記述するほうが有利だとかんがえられる.

❑ 並列バックトラックの簡潔な記述・中間語の不変性

論理型中間語ならば並列バックトラック技法にもとづくプログラムを簡潔に記述できる. なぜなら, 論理型中間語においては並列バックトラックが通常の OR 関係として, すなわち通常のバックトラックによって表現できるからである. しかも, 第6章でのべたように, 論理型中間語を採用すれば完全 OR ベクトル処理方式と並列バックトラック方式とでことなる中間語を生成する必要がなく, コード生成において共通の中間語プログラムからいずれの方式の目的プログラムでも生成することができる[注2]. 関数型言語においてはバックトラックを容易に記述する方法がないから, 並列バックトラックも容易に記述することができない.

❑ ベクトル・データの陽な記述

論理型中間語のほうがベクトル・データの陽な記述に適しているため, 目的にあっている. すなわち, 第1に FP のような関数型言語を中間語として採用するばあいにくらべてプログラムを簡潔に記述することができる. それは, FP のようにデータを陽に記述しない関数型言語においては, マスク・ベクトルや他のベクトルなどのデータが陽にあらわれないからである. しかし, これらのデータに関する情報はコード生成に必要であるから, これらが陽に記述される論理型言語のほうが中間語の目的にはあっている. 第2に Lisp を中間語として採用した場合と比較すれば, まず, 引数が値わたしにかぎられていて出力マスク・ベクトルを引数としてかえすのがむずかしいことが問題となる. また, 出力マスク・ベクトルを多値のうちのひとつとしてかえそうとすれば, ふたたびそれがプログラムに陽にあらわれないことが問題となる. したがって, FP においても Lisp においてもその関数型言語としての特性がベクトル中間語として採用するうえで問題をはらんでいるということができる. 論理型言語においてはこのような問題は存在しない.

❑ シミュレータの開発容易性

論理型中間語を採用すれば, 中間語プログラムを解釈実行するベクトル処理実行シミュレータを Prolog で記述することができるから, その開発が容易になる. Lisp を採用したばあいにも同様のことがいえるが, 自動バックトラックをつかうことはできなくなる. さらに, 手続き型言語やその他のバッチ処理的な言語を採用したばあいには Prolog や Lisp のばあいほど開発は容易ではない.

8.2.2 ベクトル中間語の機能

ベクトル中間語がもつべきくみこみ手続き (すなわち中間語を実行すべき仮想機械の

[注2] ただし, Pilog/HAP においては, 実装上の制約のため並列バックトラック方式のプログラムを生成することができない.

命令語) はつぎのように分類される.

❑ ベクトルくみこみ手続き

おもなベクトルくみこみ手続きの機能を表 8.1 にまとめた. これらをその分類項目ごとに説明する.

◆ 制御手続き

原始プログラムの各節 (手続きの構成単位) に対応する中間語プログラムの各部分からの出力を合成したり, 無効な要素を除去したり, 再帰よびだしを停止させたりするためのくみこみ手続きが制御手続きである. マスク演算方式, インデクス方式, 圧縮方式の各条件制御方式ごとにことなる機能をもった制御手続きが必要である. そのため, たとえばマスク演算方式のためには v_or1, v_or2, v_finished, 圧縮方式のためには v_compress などを用意している. また, 非決定的な手続きのコンパイル・コードにおいて使用する, 解の蓄積のためのくみこみ手続きとして v_merge を用意している. これらのくみこみ手続きのなかで v_or1, v_or2, v_merge の中間語プログラムにおけるつかわれかたはすでに図 6.5 にしめした. 他の重要な手続きに関しても第 6 章においてプログラム例のなかで説明した.

◆ リスト処理手続き

リスト処理は Prolog の原始プログラムではユニフィケーションの部分機能とかんがえられているが, 高速処理への要請がつよいために, Pilog/HAP のベクトル中間語においては専用化したくみこみ手続きを用意している. このくみこみ手続きはリスト処理専用であるというばかりではなく, 引数のモードすなわち計算の方向に関しても専用化されている[注3]. すなわち, リスト処理の基本演算である分解 (v_car, v_cdr), 合成 (v_cons), テスト (v_null など) をそれぞれくみこみ手続きとして用意している. これらの基本演算は, マスク演算方式, インデクス方式, 圧縮方式の各条件制御方式ごとにことなるくみこみ手続きとして用意する. これらの手続きの機能は第 3 章においてしめしたものとひとしい (図 3.6)[注4], また使用に関しては第 6 章において説明した.

[注3] 制御手続きのようなベクトル処理特有のくみこみ手続きの存在と, このような引数のモードに関する専用化のために, ベクトルくみこみ手続きの体系は Prolog の逐次処理系において一般的に使用されている Warren Abstract Machine などにおける命令体系とはまったくことなるものになっている.

[注4] ただし, 第3章でしめしたのはマスク演算方式だけである.

表 8.1　Pilog/HAP におけるおもなベクトルくみこみ手続き

種別	手続き名	引数	機能
制御手続き	v_finished	MI	マスク・ベクトル MI の要素がすべて偽ならば成功し, そうでなければ失敗する. マスク演算方式において再帰よびだしの終了判定のために使用する.
	x_finished	XI	インデクス・ベクトル XI が空ならば成功し, そうでなければ失敗する. インデクス方式および圧縮方式において再帰よびだしの終了判定のために使用する.
	v_merge	X,R,MI	フレーム X がふくむベクトルを併合して R に出力する. MI は X の各部分解ベクトルを支配するマスク・ベクトルである. 出力にはマスクが偽である要素はふくまれない.
	v_or1, v_or2	MI1,MI2,MO	マスク演算方式において, 原始プログラムの複数の節に対応するプログラム部分で生成されるマスク・ベクトルを合成する.
	c_compress	X,MI,R-Re	圧縮方式において, マスク・ベクトル MI に支配されたベクトル X からマスクが真である要素をえらびだして差分ベクトル R-Re に追加する (つぎに要素を追加する位置を Re とする).
リスト処理手続き	v_null, x_null	X,MI,MO または X,XI,XO	ベクトル X の要素が空リストかどうかを判定する. 結果はマスク・ベクトル (v_null のばあい) またはインデクス・ベクトル (x_null のばあい) として出力する.
	v_cons, x_cons	A,D,C,MI, MO または A, D,C,XI,XO	ベクトル A とベクトル D の各要素に関するリスト合成をおこなって, その結果をベクトル C の要素とする.
	v_carcdr, x_carcdr	A,D,C,MI, MO または A, D,C,XI,XO	ベクトル C の各要素に関するリスト分解をおこなって, その頭部をベクトル A の要素とし, 尾部をベクトル D の要素とする.
数値計算手続き	v_+, v_-, v_*, v_/, v_mod	X,Y,R,MI,MO	ベクトルどうしの四則演算と剰余演算. ベクトル X と Y の対応する要素を演算して R に格納する. MI は入力マスク・ベクトル, MO は出力マスク・ベクトルである.
	vs_+, vs_-, vs_*, vs_/, vs_mod	X,S,R,MI,MO	ベクトルとスカラとの四則演算と剰余演算. S がスカラである.
	sv_/, sv_mod	S,Y,R,MI,MO	スカラとベクトルとの四則演算と剰余演算. S がスカラである.
	v_minus	X,R,MI,MO	ベクトル要素ごとの符号反転.
比較手続き	'v_=:=', 'v_>', 'v_=\=', 'v_<', 'v_=<', 'v_>='	X,Y,MI,MO	ベクトル X, Y の要素である整数を比較して, 結果をマスク・ベクトル MO に出力する. MI は入力マスク・ベクトルである.
	'x_=:=', 'x_>', 'x_=\=', 'x_<', 'x_=<', 'x_>='	X,Y,XI,XO	ベクトル X, Y の要素である整数を比較して, 結果をインデクス・ベクトル XO に出力する. XI は入力インデクス・ベクトルである.
ユニフィケーション手続き	v_assign, x_assign	X,Y,MI または X,Y,XI	マスク・ベクトル MI またはインデクス・ベクトル XI にしたがってベクトル Y の各要素を未束縛変数であるベクトル X の対応する要素に代入する.
	vs_assign, xs_assign	X,Y,MI または X,Y,XI	マスク・ベクトル MI (または XI) にしたがってベクトル Y の各要素を未束縛変数であるベクトル X の対応する要素に代入する.

◆ 数値計算手続き

数値計算は Prolog の原始プログラムにおいては手続き is によっておこなう．しかし，手続き is の多機能性は高速処理には不利である．したがって，Pilog/HAP では演算の種類ごとにくみこみ手続きをもうけ，ベクトル化時にデータフロー解析にもとづいてこれらの手続きをくみあわせた中間語プログラムを生成するようにしている．ただし，プログラムによっては実行時にならなければ演算の種類がさだまらないばあいがある[注5]．このようなばあいにはプログラムをコンパイルすることができず，インタプリタによるスカラ処理で実行しなければならない．Pilog/HAP では，現在のところこのようなばあいはあつかっていない．数値計算用のくみこみ手続きには，オペランドがともにベクトルのばあいに使用する 'v_+'，'v_-' などと，オペランドの片方がスカラのばあいに使用する 'vs_+'，'vs_-' などとがある．また，等差数列を生成するくみこみ手続き v_iota がある．これらの数値計算用くみこみ手続きの主要機能は 1 個のベクトル命令で実現されるが，引数の型のチェックなどをおこなう必要があるため，1 個のくみこみ手続きに対して複数のベクトル命令が生成される．

◆ 比較手続き

数値の比較のためにくみこみ手続き 'v_=:='，'v_=\=' が用意されている．これらの手続きのベクトル要素に対する機能は Prolog の '=:=' および '=\=' にちかいが，それらとはちがって，両辺のいずれかまたは両方が演算子をふくむ式であるばあいにはその評価をおこなわない (エラーとみなされる)．すなわち，くみこみ手続き is の右辺とおなじあつかいとしている．したがって，Pilog/HAP においては Prolog の比較手続きのよびだしが数値計算手続きのよびだし列と比較とに変換される[注6]．Prolog はリストや他の種類のデータの比較のための手続きをもっているので，Prolog の機能の完全なサポートのためにはそれらに対応するくみこみ手続きも必要だが，試作処理系では用意していない．

◆ ユニフィケーション手続き (代入 他)

ユニフィケーションは比較手続き以上に多機能であり，それゆえに高速処理が困難になっている．したがって，中間語においては単機能化をはかっている．すなわち，まず論理変数への代入 (ユニフィケーションの部分機能) のためにくみこみ手続き v_assign と vs_assign とを用意している．リスト処理に関してはすでにのべたとおりである．これらの手続きでは処理できない場合があるため，ベクトル処理による Prolog の機能の完全なサポートのためにはより一般的なユニフィケーションのための手続き v_unify が必要だとかんがえられる[注7]．しかし，この手続きはまだ実装していない．したがって Pilog/HAP ではかぎられた場面におけるユニフィケーションしかベクトル処理することができない．v_unify の実現のためには手続き引数の入出力モードに依存しないユニフィケーションのベクトル化法が必要だが，いまのところ高速処理が可能な方法は開発されていないため実装していない．

❑ スカラくみこみ手続き

Fortran のばあいと同様に，かりに論理型言語のベクトル処理が実用化したとしても，プログラムのすべての部分がベクトル化できるとはかんがえられない．むしろ，現状の Pilog/HAP におけるベクトル化法によればたいていの論理型言語プログラムはベクトル化されないままとなる．したがって，論理型言語の完全なサポートのためには，通常の論理型言語がもつスカラ処理機能を実現するくみこみ手続き (スカラくみこみ手続き) はすべて必要である[注8]．

❑ インタフェース手続き

ベクトル処理部分とスカラ処理部分とではデータ構造もことなり，処理方法もことなる．すなわち，ベクトル処理部分ではマルチ・ベクトルがつかわれ，並列バックトラック以外のバックトラックは存在しない．これに対して，スカラ処理部分ではマルチ・ベクトルはあらわれず，OR 関係にある計算はすべて通常のバックトラックによって実行される．したがって，ベクトル処理部分とスカラ処理部分とをつなぐ，ベクトル合成・分解のためのくみこみ手続き v_singleton および v_decompose が用意されている．これらの手続きの機能を表 8.2 にしめす．

表 8.2　Pilog/HAP におけるインタフェースくみこみ手続き

種別	手続き名	引数	機能
インタフェースくみこみ手続き	v_sigleton	S,V	スカラ S を入力して 1 要素からなるベクトルをつくり，V に出力する．
	v_decompose	V,M,S	v_decompose がよびだされると V の有効要素 (マスク・ベクトル M の偽でない最初の要素に対応する V の要素) を S とユニファイして出力する．バックトラックが生じるたびに S に V の有効要素をユニファイして出力する．V の有効要素がなくなると実行は失敗する．

[注5] たとえば X is E のようなよびだしにおいて変数 E の値が実行時に 1+2 になるようなばあいである．

[注6] Prolog においては，実行時までかたちがさだまらない式を比較手続きの両辺にあたえることができるが，このベクトル中間語においてはそれはゆるされない．また，このような Prolog の手続きよびだしはベクトル中間語にコンパイルすることができない．

[注7] 現在ベクトル中間語としてサポートされていない範囲はすべてスカラ処理するようにすれば，処理系としてはとじる．しかし，性能的には満足できないだろう．

[注8] しかし，現在の試作処理系においてはスカラくみこみ手続きは部分的にしかサポートされていない．

8.3 ベクトル化の方法

試作処理系におけるベクトル化の方法は，基本的には第6章の方法にしたがっている．しかしながら，第6章においてはそれを手順としてはしめしていない．ここでは，試作処理系におけるベクトル化の手順をしめす．手順の抽象的な記述にさきだって，例題によって手順を説明する．

8.3.1 ベクトル化の例

決定的な手続きと非決定的な手続きのベクトル化手順を手続き append を例題としてしめす．図8.3は決定的なばあいすなわち第1引数および第2引数が入力で第3引数が出力であるばあいの例であり，図8.4は非決定的なばあいすなわち第3引数が入力で第1引数および第2引数が出力であるばあいの例である．いずれにおいても，まず解析と変形がより容易なプログラム表現形式である「標準形」に原始プログラムを変換する．そして高速実行が可能になるように，引数のモードに関する情報にもとづいて，特殊化(専用化)したユニファイアにおきかえる．そのうえで(狭義の)ベクトル化すなわち図8.3および8.4の第3ステップの処理をおこなう．すなわち，1個の入力から1個の出力をえるように構成されたプログラムを，複数個の入力から複数個の出力をえるようなプログラムに変換する．両者のあいだで標準化およびユニファイアの特殊化においても生成されるプログラムにはちがいがあるが，これは引数のモードがことなるためである．両者の本質的なちがいは(狭義の)ベクトル化の方法にある．このちがいについては第6章でのべたとおりであるから，ここではあらためて説明しない．

(1) 原始プログラム

```
mode append(+, +, -).
append([], X, X).
append([H|R], Y, [H|R1]) :- append(R, Y, R1).
```

標準形への変換
(節の統合と引数の変数化)

(2)

```
mode append(+, +, -).
append(X, Y, Z) :-
  [X = [], Z = Y]
 ;[[H|R] = X*, append(R, Y, R1),
   [H|R1] = Z*].
```

* 入力引数へのユニフィケーションは右辺の先頭、出力引数へのユニフィケーションは右辺の末尾におかれる。

ユニファイアの特殊化

(3)

```
mode append(+, +, -)**.
append(X, Y, Z) :-
  [s_null(X), s_assign(Z, Y)]
 ;[s_carcdr(H, R, X), append(R, Y, R1),
   s_cons(H, R1, Z)].
```

** モード情報は実際には中間語とはべつにもっている。

ベクトル化

(4) 中間語プログラム

```
v_append(X, Y, Z, MI, MO) :-
  v_finished(MI), !,
 ;v_null(X, MI, M1),
  v_assign(Z, Y, MO1),
  v_or(MI, MO1, M2),
  v_carcdr(H, R, X, M2, M3),
  v_append(R, Y, R1, M3, MO2),
  v_cons(H, R1, Z, MO2),
  v_end_or(MO1, MO2, MO).
```

図8.3　手続き append の決定性のばあいのベクトル化過程

(1) 原始プログラム

```
mode append(-, -, +).
append([], X, X).
append([H|R], Y, [H|R1]) :- append(R, Y, R1).
```

標準形への変換
(節の統合と引数の変数化)

(2)

```
mode append(-, -, +).
append(X, Y, Z) :-
  [X = [], Z = Y]
 ;[[H|R1] = Z*, append(R, Y, R1),
   [H|R] = X*].
```

* 入力引数へのユニフィケーションは右辺の先頭、出力引数へのユニフィケーションは右辺の末尾におかれる。

ユニファイアの特殊化

(3)

```
mode append(-, -, +)**.
append(X, Y, Z) :-
  [s_assign(X, []), s_assign(Y, Z)]
 ;[s_carcdr(H, R1, Z), append(R, Y, R1),
   s_cons(H, R, X)].
```

** モード情報は実際には中間語とはべつにもっている。

ベクトル化

(4) 中間語プログラム

```
v_append(X, Y, Z, MI, MO) :-
  v_append_1(XL, YL, Z, MI, ML),
  v_merge([XL,YL], [X,Y], ML, MO).

v_append_1(X, Y, Z, MI, MO) :-
  v_finished(MI), !,
  XL := [], YL := [], ML := []
 ;v_assign(XL1, [], MI),
  v_assign(YL1, Z, MI),
  v_carcdr(H, R1, Z, MI, M2),
  v_append_1(RL, YLR, R1, M2, MLR),
  map_v_cons(H, RL, XLR, MLR),
  XL := [XL1|XLR], YL := [YT|TLR],
  ML := [MI|MLR].

map_v_cons(H, [], [], []).
map_v_cons(H, [XT|XL], [YT|YL], [MT|ML]) :-
  v_cons(H, XT, YT, MT),
  map_v_cons(H, XL, YL, ML).
```

図 8.4　手続き append の非決定性のばあいのベクトル化過程

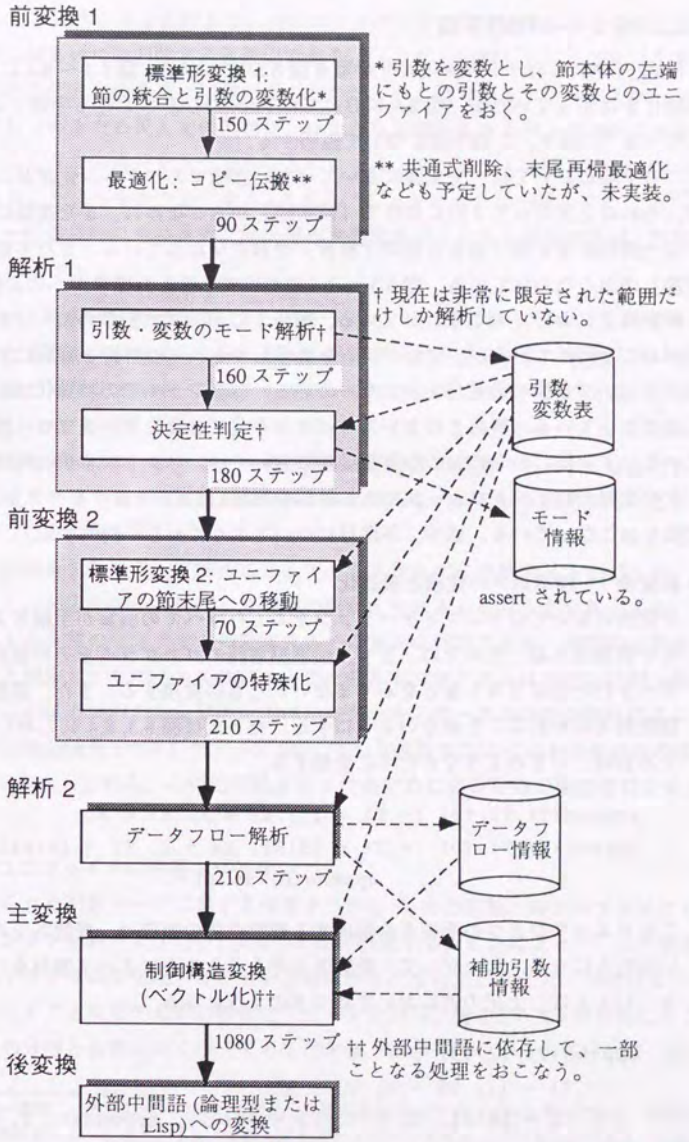


図 8.5　Pilog/HAPにおけるベクトル化処理手順

8.3.2 ベクトル化の手順

Pilog/HAP におけるベクトル化手順を図 8.5 にしめす．図 8.3 ～ 8.4 においては一部単純化をはかっていたが，図 8.5 においては Pilog/HAP における手順をより忠実にしめしている[注9]．以下，この手順について説明する．

この手順においては，主変換においてせまい意味でのベクトル化がおこなわれているが，それにさきだって 2 回にわけて「前変換」がおこなわれ，また変換に必要なデータフロー情報などをあつめるための「解析」がおこなわれている．また主変換のあとに「後変換」がおこなわれている．図 8.3 ～ 8.4 においてしめした標準形への変換は前変換 1 と前変換 2 にわけておこなわれている．解析 1 においては引数のモード解析と決定性判定がおこなわれているが，プログラムを整理してそれらの解析を容易にするために前変換 1 において多少の最適化をおこなっている．ユニファイアの特殊化は前変換 2 においておこなっている．解析 2 においてはベクトル化に必要なデータフロー解析をおこなっている．主変換ののちにおこなう後変換においては，ベクトライザの内部中間語からコード生成部とのインタフェースである外部中間語 (図 8.1 におけるベクトル中間語) への変換をおこなっている．以下，各部分についてよりくわしく説明する．

❑ 前変換 1：標準形への変換と最適化

中間語においてはユニフィケーションをふくむすべての演算が手続きよびだしのかたちで表現される．そのため，まず，論理型言語のプログラムをその頭部に変数以外のデータたとえばリストなどをふくまないかたちに変換する．また，頭部に同一変数が複数回あらわれることがないようにする．たとえば図 8.3, 8.4 にしめした append の例であれば，つぎのようなかたちに変換する．

```
append(T1,T2,T3) :- T1 = [], T2 = X, T3 = X.
append(T4,T5,T6) :- T4 = [H|R], T5 = Y, T6 = [H|R1],
                    append(R, Y, R1).
```

これによってひとつの手続きを構成する複数の節の頭部は，変数名をのぞいてひとしいかたちになる．したがって，変数名をそろえることによって頭部をひとつにまとめる．たとえば，上記の例においてはつぎのようになる．

```
append(T1,T2,T3) :-
      T1 = [], T2 = X, T3 = X.
    ; T1 = [H|R], T2 = Y, T3 = [H|R1], append(R, Y, R1).
```

つぎに，上記の変換の際に生じた冗長性をなくすためにコピー伝搬をおこなう．これは，X = Y というような変数どうしのユニフィケーションをおこなう手続きよびだしを，可能なばあいには削除する最適化である．「コピー伝搬」という名称は手続き型言語における代入文の最適化においてつかわれることばである．ユニフィケーションを対象としているため代入文の最適化とはことなる部分もあるが，基本的にはおなじである．ここまでの操作をおこなうことによって，プログラムは図 8.5 (2) および図 8.6 (2) にしめしたようなかたちになる (ただし，こまかい点では一致していない[注10])．なお，コピー伝搬以外にも共通式削除などの最適化をおこなうと効果的だが，現在はおこなっていない．

❑ 解析 1：モード解析と決定性判定

ユーザがあたえたモード宣言をつかって，引数のモードを決定する．本来はユーザがあたえた情報だけにたよらず，手続きをまたがる解析をおこなうべきである．しかし，現在の Pilog/HAP 処理系においてはそれをおこなっていない．そのひとつの理由は，強力な解析のためにはデータフロー解析の結果を利用する必要があるが，現在の処理系の構造ではデータフロー解析があとでおこなわれるためにそれを利用することはできないということである[注11]．

つぎに，手続き引数のモード解析にもとづいて手続きが決定的であるかどうか，すなわちひとくみの入力データに対して複数の解がえられうるかどうかを判定する．これは，ベクトル化後のプログラムの効率をきめる重要な処理である．決定的な手続きを非決定的と判定してもベクトル化によってえられるプログラムはただしいが，非効率になる．このばあいのおもな効率低下は，ベクトル・データのむだな複写によるオーバヘッドと無効演算 (マスクがすべて *false* である演算またはベクトル長が 0 の演算) によってもたらされる．この効率低下をさけるために必要なのが決定性判定である．

❑ 前変換 2：ユニファイアの移動と特殊化

解析 1 でえられた引数モードに関する情報をつかって，出力引数へのユニファイア (ユニフィケーション手続きよびだし) は節の末尾に移動する．これによって，より効率がよい目的プログラムを生成することを可能にする．また，おなじモード情報をつかってユニファイアを専用のものに特殊化する．すなわち，図 8.3 ～ 8.4 にしめしたようにリストの分解と合成とをくべつするなどする．このくべつができなければ，現在

[注9] 図 8.5 には Prolog で記述したベクトライザの各部分のコーディング・ステップ数もしめしている．

[注10] 一致していない点は，一時的に使用される変数名と，節の本体に [] がついている点とである．後者は，ベクトライザ内でのあつかいを容易にするために節本体をリストのかたちにしているためである．したがって，いずれも本質的な問題ではない．

[注11] 容易にかつ強力に各種の解析をおこなうことができ，かつその結果が必要な変換処理において参照でき，効率よく変換できるようにベクトライザを設計するのは容易ではない．

のベクトル化方法によってはベクトル化することができない．

❑ 解析2：データフロー解析

データフロー解析をおこなうおもな目的は，ベクトル化可能かどうかを判定するためと，非決定的な手続きのよびだしにおいて複写するべきベクトルを決定するためである．第6章においてしめしたように，非決定的な手続きの実行においては，それ以降で使用されるすべての部分解ベクトルを複写してつくりなおす必要がある．なぜなら，解の数が変化したときに部分解ベクトルをつくりなおしておかないと，部分解ベクトルの要素の対応がくずれてアクセス不能になってしまうからである．6.4節においては，対応づけのための引数に `BI` および `BO` というなまえをつけている．現在の方法では複写可能なベクトルは変数をふくまないものだけであり，基底項だけをふくんであることがしめせないベクトルの複写が必要なばあいはベクトル化不能である．データフロー解析をおこなわなければ多数のむだな複写をおこなうことになるためベクトル化によってかえって性能低下をまねく．また，上記の理由により原理的にもベクトル化の範囲をせばめることになる．

Pilog/HAPにおいてもとめているデータフローは，各よびだしの前後における各論理変数の生存 (liveness) と利用可能性 (availability) である．これらの解析は Aho, Sethi, and Ullman [Aho 86] などに記述されている手続き型言語における変数データフローの解析と同様の方法でおこなうことができるが，論理型言語においては制御構造が単純化されているために手続き型言語より解析は容易である．

ところで，上記のように手続き間のインタフェースをきめるためにデータフロー解析をおこなうのであるから，当然その解析範囲はプログラム全体すなわち手続き間にまたがる解析が必要である．しかし，手続き型言語にくらべると意味が単純な論理型言語が解析対象であることによって，この点でも手続き型言語にくらべて解析は単純化されている．

❑ 主変換：制御構造変換 (ベクトル化)

主変換はOR並列性をAND並列性に変換する変換と，ベクトル処理のために必要なくりかえし構造の交換と1重化という2種類の変換があわせられたものとかんがえることができる．前者を**決定化**とよび，後者を**制御構造変換**とよぶことにする．

まず決定化についてのべる．解探索のベクトル化すなわちORベクトル化および並列バックトラック化は様々なプログラム変換の複合された変換である．そのなかでもっとも重要な変換はOR並列性のAND並列性への変換であり，ここではこれを決定化とよぶことにする．決定化は，原始プログラムにおけるOR関係 (バックトラックによって逐次実行される節のあいだの関係) にあるプログラム部分を中間語においてはAND関係にするために必要な変換である[注12]．逐次論理型言語プログラムを並列論理型言語プログラムに変換する変換 [Ueda 85b, Ueda 86, Codish 86, Tamaki 87] がおこなっているのが決定化である．

つぎに制御構造変換についてのべる．ここでは，ベクトル処理のためにくりかえし構造の交換と1重化をおこなう．N クウィーンのプログラムに即していえば，変換前は再帰よびだしが内側のくりかえしでありバックトラックすなわちOR関係にある計算によるくりかえしが外側のくりかえしであったものを，変換により逆にする．この変換は，各手続きの入出力をベクトルとすることによって実現される．また，非決定的な手続きを変換したプログラムの実行の末尾で，変数ごとにそのOR関係にある値をひとつのベクトルに蓄積するように変換することによって，非決定的な手続きの実行ごとに導入されていたくりかえし (選択点) をなくす．同時に，くみこみ手続きとユーザ定義手続きの両方について，マスク演算方式，インデクス方式，圧縮方式のうちのいずれかの条件制御にしたがうかたちに変換する．たとえば，マスク演算方式のばあいは入力マスク・ベクトルと出力マスク・ベクトルとを引数として追加する．現在の処理系においては基本的にマスク演算方式だけをサポートしている．

❑ 後変換：外部中間語への変換

Pilog/HAPにおいては，図8.1にしめしたように2種類の外部中間語を生成することができる．すなわち，機械語の実行のためのLispによる中間語と，シミュレータによる実行のためのPrologによる中間語とである．前者をPilog/LLとよび，後者をPilog/ILとよんでいる．Pilog/ILは内部中間語と非常にちかいが，Pilog/LLにおいてはLispの制御構造をつくりだす変換が必要である．実際にPilog/HAPによって生成されたこれらの中間語の例を図8.6にしめす．図8.6にしめしたのは，第6章でも使用した N クウィーン問題のプログラムを構成する手続き `not_take1` をベクトル化してえられたプログラムである．また，図8.7には，図8.3～8.4にしめした決定的および非決定的な手続き `append` をベクトル化してえられたプログラムをしめす．

これらの図をみると，Pilog/LLにおいてはPilog/ILのばあいあるいは表6.2にしめしたのとはくみこみ述語の引数の数がことなっていることがわかる．それは，効率をあげるなどの目的のためにマスク・ベクトルが引数ではなく大域変数によってわたされているなどの理由による．現在はおこなっていないが，その大域変数をマスク・レジスタにわりつけることによって，オーバヘッドのすくないコードを生成することが可能である．

[注12] これにより図14においては，(a) においてOR関係にある2つの節が，(c) においては `v_append1` と `v_append2` というAND関係にある手続きに変換されている．(d) に関しても同様である．

```
not_take1(_, _, _, MI-MI) :-                          ……MIはマスク・ベクトル.
   v_finished(MI), !.
not_take1(B, Qa, Qs, MI-MO) :-          ……MI,MO,M1,...,M4などはマスク・ベクトル.
   v_null(B, MI, MO1),
   v_carcdr(Q, R, B, M1, M2),
   'v_='(Q, Qa, M2, M3), 'v_='(Q, Qs, M3, M4),
   'vs_+'(Qa, 1, Qaa, M4), 'vs_-'(Qs, 1, Qss, M4),
   not_take1(R, Qaa, Qss, M4-MO2),
   v_end_or(MO1, MO2, MO).
```

(a) Pilog/IL による表現

```
(defun not_take1 (_20909 _20825 _20827)
  (cond ((v_finished _20909))
        (t (s_let _20909
                  ……s_letはPilog/LLのくみこみマクロ(Lispでインプリメントされている).
                  (_26085 _21005 _21003 _20997 _20995)      ……局所変数のリスト.
                  nil
                (v_or0 _26085)
                        ……v_or0はPilog/LLのシステム関数(v_or1,v_or2なども同様).
                (v_null _20909)
                (v_or1 _26085)
                (v_carcdr _20995 _20997 _20909)
                (v_=\= _20995 _20825)
                (v_=\= _20995 _20827)
                (vs_+ _20825 1 _21003)
                (vs_- _20827 1 _21005)
                (not_take1 _20997 _21003 _21005)
                (v_or2 _26085)))))
```

(b) Pilog/LL による表現

図 8.6　述語 **not_take1** の OR ベクトル化後のプログラムの中間語表現


```
(defun append (_185 _99 _213)
  (cond ((v_finished _185))
        (t (s_let _185
                  (_275 _273 _271)
                  nil
                (prog (_3155)
                  (v_or0 _3155 _185)      ……第2節対応部のためのマスク・ベクトル退避.
                  (v_null _185)                          ……リストが空かどうかの判定.
                  (v_assign _213 _99)          ……第2、第3引数のユニフィケーション.
                  (v_or1 _3155)              ……第2節対応部のマスク・ベクトル計算.
                  (v_carcdr _271 _279 _185)                  ……リスト分解.
                  (append _273 _99 _275)                  ……再帰よびだし.
                  (v_cons _271 _275 _213)                  ……リスト合成.
                  (v_or2 _3155))))))                  ……出力マスク・ベクトルの計算.
```

(a) 決定性の場合の中間語プログラム

```
(defmacro append (_5329 _5243 _5357)
  `(s_let nil nil (*m _8455 _8489)
     (ap2_1 _8455 _8489 ,_5357 *m)
     (v_merge_2_0 _8455 ,_5329 _8489 ,_5243 *m)))     ……マルチ・ベクトルの併合.

(defun append_2 (_11383 _11483 _11583)
  (v_cons _5415 _11483 _11583)

(defun append_1 (_5329 _5243 _5357 _9079)
  (cond ((v_finished _5357)
         (close_multi_vector _5329)    ……マルチ・ベクトルの完結(末尾を[]にする).
         (close_multi_vector _5243)                          ……同上.
         (close_multi_vector _9079))                         ……同上.
        (t (s_let _5357
                  (_9661 _9497 _5419 _5415)
                  (_5417)
                (prog (_9379 _9381)
                  (v_save_mask _9379 _5357)
                                ……第2節対応部のためのマスク・ベクトル退避.
                  (v_assign _9661 _5357)
                  (vs_assign _9497 nil)
                  (v_change_mask _9381 _9379)……第2節対応部のマスク・ベクトル計算.
                  (v_carcdr _5415 _5419 _5357)
                  (append_1 _5417 _5243 _5419 _9079)
                  (v_map_2_1 #'append_2 _9079 _5243 _5417 _5329)
                  (s_add_multi_vector_element _9497 _5329)
                        ……マルチ・ベクトルに要素すなわち部分ベクトルを追加する.
                  (s_add_multi_vector_element _9661 _5329)          ……同上.
                  (s_add_multi_vector_element _9381 _9079)))))))    ……同上.
```

(b) 非決定性の場合の中間語プログラム

図 8.7　手続き **append** の Pilog/LL による中間語プログラム

8.4 実行方式

この章では，まずPilog/HAPにおける2つの実行方式についてのべ，つぎにベクトル計算機による論理型言語プログラムの実行方式に関する3つの問題についてそれぞれかんたんにのべる．実行方式の詳細は，Pilog/HAPが実装された環境すなわちS-810とLispに依存するところがおおきいので，記述を省略する．

8.4.1 2つの実行方式

8.1節でのもべたように，Pilog/HAPにおいては2種類の実行方法が用意されている(図8.1)．第1はS-810のベクトル命令をふくむ機械語プログラムを出力して実行する方法である．第2は論理型中間語をそのまま実行することができるシミュレータを使用する方法である．当然のことながら高速な実行のためには第1の方法をとる必要がある．

第1の方法においては，中間語における各くみこみ手続きにほぼ対応するアウトラインのくみこみ手続きが用意され，コンパイルされたプログラムからこれらの手続きがつぎつぎによびだされる．ベクトル命令がインラインに生成されることはない．コード生成はほとんどLispのコンパイラに依存している．図8.1にしめした「コード生成部」の実態は，図8.6, 8.7などにしめされた中間語プログラムにあらわれる`s_let`などのマクロや関数である．これらがくみこみ手続きをよぶための準備たとえばベクトルのわりあてなどをおこなう．より高速な実行のためにはくみこみ手続きのインライン化が必要だが，その実現のためにはLispのコード生成部を改造する必要がある．とくにベクトル・レジスタわりあてが必要になるために，かなりの量の開発が必要になる．それをさけるため，インライン化はおこなわなかった．*N*クウィーン問題のプログラムのばあい，第6章でしめしたように手動ベクトル化のばあいの性能が4.5 MLIPSであるのに自動ベクトル化のばあいの性能が2.6 MLIPSというようにかなりひくいのは，おもにインライン化とそれにともなう最適化がなされていないためだとかんがえられる．

第2の方法においては，中間語における各くみこみ手続きをPrologの手続きとして実現している．第2の方法におけるシミュレータを開発した目的を説明しておく．

(1) 並列バックトラック方式の実験

機械語による実行系は並列バックトラック方式による実行の機能をもっていない．したがって，シミュレータにおいてそれを実現することにより，並列バックトラック方式に関する実験と検討をおこなうことを可能にしている．

(2) 会話的な実行の必要

HITAC S-810においてはベクトル命令をふくむプログラムをTSSで実行することができなかったため，会話的な実行を可能にするためにシミュレータを開発した[注13]．なお，S-820においてはTSSによる実行が可能になったため，シミュレータの必要性もうすれたといえる．

8.4.2 実行方式における3つの問題

実行方式を設計する際に問題となる3つの問題についてのべる．

(1) 個別配列と環境配列

論理型言語プログラムをベクトル化するにあたって，第6章でのべたように，データの表現形式としては原始プログラムにおける各データ(論理変数の値)を個別に配列化する個別配列方式と，ORプロセスごとにデータをまとめたものを要素とするベクトルをつくる環境配列方式とがかんがえられる．いずれを採用するかによって，実行方法だけではなく，中間語の形式やベクトル化方法にもちがいが生じるとかんがえられる．環境配列方式における記憶わりあては，共有記憶があるMIMD型並列計算機による実行方式における記憶わりあてと共通部分がおおい．この研究においては，個別配列方式のほうが現在のベクトル計算機に適合性がたかく，また開発が容易だとかんがえられたため，ほとんど個別配列方式だけを検討してきた[注14]．

(2) 構造共有と構造複写 (Structure-Copying vs. Structure-Sharing)

論理型言語の処理系を作成する際に，逐次処理系，並列処理系をとわず問題になるのが，論理変数の値の保持のために構造共有方式を採用するか構造複写方式を採用するかという選択枝である[Kanada 85]．この研究におけるベクトル処理法においては，変数値の保持法も従来の論理型言語処理系におけるのとはおおきくことなっているが，構造複写方式にちかいデータ構造を採用している．そのおもな理由は，ベクトル計算機では逐次計算機にくらべるとデータ複写がおおきなオーバヘッドとならず，むしろ共有をおおくすると記憶アクセス衝突のために性能が極端に低下するばあいが生じやすいことである．

(3) 並列バックトラック方式による実行

Pilog/HAPにおいては，第6章でしめしたような，中間語プログラムを変更せずに完全ORベクトル処理方式と並列バックトラック方式とをきりかえる方法を採用してい

[注13] Pilog/HAPにおいて会話的な実行が必要とかんがえた第1の理由はデモンストレーションである．Pilog/HAPは実用的な処理系ではないためとくに必要はなかったが，実用的な処理系においてはデバグのために会話的な実行が必要であることはいうまでもない．

[注14] マルチ・プロセッサにおける論理型言語の並列処理をかんがえると，通常そのデータ表現形式は環境配列方式にちかい．個別配列方式のほうが有利であるようなアーキテクチャは，Connection Machineなどにかぎられるであろう．

る．この方法は，8.3節でのべたように，論理型中間語を採用することにより可能になった．すなわち，中間語のくみこみ手続き v_merge において(通常の意味の)バックトラックを発生させずにマルチ・ベクトルの併合をおこなえば完全ORベクトル処理が実現され，マルチ・ベクトルの併合をおこなわずにバックトラックを発生させれば並列バックトラック処理が実現される．

8.5 ANDベクトル化の自動化について

これまで逐次論理型言語のORベクトル化法についてのべてきた．これに対して逐次論理型言語および並列論理型言語のANDベクトル化においては，ベクトル処理のための一般的なデータ構造変換法は確立されていないため，いまのところ自動ベクトル化はできない．しかし，たとえば並列論理型言語が得意とするデータのフィルタリングをおこなうプログラム[注15]においては，おおくのばあい，手動でマルチ・ベクトルへの変換をおこなうことによってベクトル処理が可能になるとかんがえられる．そこでわれわれは第6章でのべたように，フィルタリングのプログラムの例として，エラトステネスのふるいによる素数生成のプログラムを手動でベクトル化し，性能を測定した．性能向上のためマルチ・ベクトルの併合をとりいれたが，加速率は1.7倍にとどまった．十分なベクトル長がえられているのに加速率がひくいのは，スカラ処理部分のオーバヘッドがたかいためだとかんがえられる．今後の研究によってこのオーバヘッドを減少させるとともに，ベクトル化手続きをアルゴリズム化することができる可能性がある．

[注15] 並列論理型言語においてはおおくの処理が(UNIXにおけるパイプのような)フィルタリングの形式で記述される．たとえば，オブジェクト指向プログラミングもメッセージ(すなわちデータ)のフィルタリングとして記述される．したがって，並列論理型言語プログラムのベクトル化においてまずフィルタリングをかんがえるのは妥当だとかんがえられる．ただしそのばあ，並列論理型言語における重要なプログラミング技法のひとつである不完全メッセージすなわち変数をふくんだメッセージ・ストリームのあつかいが重要な課題となるだろう．

8.6 まとめ

HITAC S-810 上のための自動ベクトル化処理系 Pilog/HAP を試作した．この処理系によってベクトル化できるプログラムの範囲はかぎられており，また Prolog 処理系としても不完全ではあるが，この試作によって *N* クウィーン問題をはじめとする一部の論理型言語プログラムが自動ベクトル化によって高速にベクトル処理可能な目的プログラムに変換できることが実証された．*N* クウィーン問題のばあい，自動ベクトル化による実行性能は 2.6 MIPS であり，手動ベクトル化による性能 4.5 MLIPS よりはかなりひくいが，この差はおもにくみこみ述語のインライン化がなされていないためだとかんがえられる．インライン化とそれにともなう最適化をおこなえば，手動ベクトル化にちかい性能あるいはそれ以上の性能がえられるとかんがえられる．この処理系の基本構造および中間語は，OR ベクトル化だけではなく，AND ベクトル化のばあいにも，ほぼそのまま適用できるとかんがえられる．

第9章
マルチ・ベクトル
— ベクトル記号処理のためのデータ構造

要旨

第6章でのべた逐次論理型言語プログラムのベクトル処理法と第7章でのべた並列論理型言語プログラムのベクトル処理方法の両方において，共通のデータ構造「マルチ・ベクトル」がつかわれることがわかった．この節ではマルチ・ベクトルということばによりひろい定義をあたえ，論理型言語以外の応用までふくめてマルチ・ベクトルの機能，マルチ・ベクトルに対する操作などについて考察する．すなわち，マルチ・ベクトルはベクトル記号処理の広範な応用において，ベクトル再生成オーバヘッド回避にやくだち，バックトラックや並列処理の処理単位としてはたらき，ベクトル・パイプラインの有効活用にやくだつとかんがえられる．したがって，マルチ・ベクトルはベクトル記号処理において非常に重要なデータ構造であると結論できるだろう．

9.1 はじめに

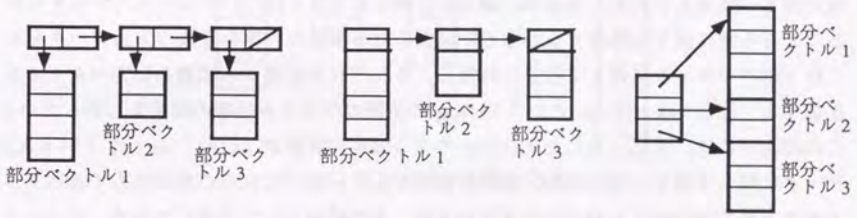
われわれは，Cray-1，HITAC S-810，HITAC M-680H IAP など，数値計算専用機として発展してきたパイプライン型ベクトル計算機をリスト，木，グラフなどの可変データ構造をあつかう記号処理 (非数値処理) に適用し，その高速実行をはかってきた．また，データベース処理用のベクトル計算機である M-680H IDP をデータベース以外の記号処理に応用する実験をおこなってきた．記号処理プログラムにおけるリスト，木，グラフなどの可変データ構造は，そのままではベクトル計算機によって実行できないばあいや高速には実行できないばあいがしばしば存在する．このようなプログラムを高速にベクトル処理できるようにするためには，ベクトル計算機に適合したデータ構造をみいだし，もとのデータ構造をその構造に変換することが重要である．われわれが開発した上記の論理型言語の OR 並列処理方式と AND 並列処理方式もこのようなデータ構造変換にもとづいている．

ところが，これらの実行方式はまったくことなるベクトル化方式にもとづいているにもかかわらず，その実行においてほとんどおなじデータ構造がつかわれ，そのデータ構造がよく似た操作をうけていることがわかった．また，類似のデータ構造が M-680H IDP におけるマージ・ソートにもつかわれていることがわかる．これらのデータ構造をわれわれはマルチ・ベクトルとよんでいる．マルチ・ベクトルの応用法をしめし，その機能を分析するのが，この章のおもな目的である．

9.2 節ではマルチ・ベクトルの定義をしめす．9.3 節ではマルチ・ベクトルがベクトル記号処理においてどのように使用されるかを例をつかってしめす．9.4 節では，9.3 節でしめした応用例においてマルチ・ベクトルがはたしている役割を分析する．9.5 節ではマルチ・ベクトルに対する操作としてどのようなものがあるかをしめす．

9.2 マルチ・ベクトル

複数個の可変長ベクトルまたは固定長ベクトルをなんらかの方法で連結したデータ構造をマルチ・ベクトルとよぶ．ただし，各ベクトルは同質の値をふくむことを仮定する[注1]．マルチ・ベクトルを構成する各ベクトルを部分ベクトルとよぶ．図 9.1 (a) にしめすようなベクトルを要素とするリストをマルチ・ベクトルの基本形とかんがえる．Prolog で容易に記述できるなどの理由によって，第 6 ～ 7 章では基本形のマルチ・ベクトルを使用している．しかし，より低水準の言語で記述するばあいには，部分ベクトルの先頭要素でつぎの部分ベクトルをさす図 9.1 (b) のようなかたちのほうが，時間，記憶のいずれにおいてもより効率がよいであろう．また，図 9.1 (c) にしめすように各部分ベクトルへのポインタを要素とするベクトルもマルチ・ベクトルの一種だとかんがえることができる．さらに，図 9.1 にしめした以外にもマルチ・ベクトルのさまざまな変形をかんがえることができる．なお，(a)，(b) のように可変長の部分ベクトルを連結したばあいには，処理の際に各部分ベクトル要素数を知る必要があるので，その値を部分ベクトル先頭などに格納する必要がある．



(a) 基本形　　(b) 最適化形　　(c) 変形

図 9.1　マルチ・ベクトルとその変形

[注1] ここで「同質」ということばにはあいまいさがあるため，マルチ・ベクトルの定義にもあいまいさがふくまれることがとりあえずはさけられない．しかし，どのような範囲のデータ構造をマルチ・ベクトルと定義するのが適切であるかがまだ明確でないので，あえて厳密な定義はあたえないことにする．

9.3 マルチ・ベクトルの応用

現在わかっている，ベクトル記号処理におけるマルチ・ベクトルないしそれに類似のデータ構造の使用法には3とおりある．第1はエイト・クウィーン問題のような解探索において部分解を格納するという用途である．第2は素数生成問題のようなストリーム処理においてストリームの要素を格納するという用途である．第3はマージ・ソートである．これらについて順に説明する[注2]．

9.3.1 解探索における使用

第6章では，Prologのような論理型言語で記述されたバックトラックをつかった解探索プログラムを，ベクトル化するすなわちベクトル処理できるようにかきかえるばあいにマルチ・ベクトルを使用し，エイト・クウィーン問題のベクトル化されたプログラムの実行性能をしめした．第6章でしめした方法においては，もとのプログラムにおける複数のことなる入力データまたは出力データ(すなわち解)を要素とするベクトルを使用し，もとのプログラムではバックトラックしながらくりかえし処理してもとめていた複数の解を一度にもとめる．そして，図9.2に例示するように，ひとつの入力からえられることなる解に関する計算すなわちOR関係にある複数の計算を複数の入力データをふくむ1個のベクトルに対して独立に計算し，それぞれの計算から複数の出力ベクトルを生成する．図9.2はcarcdrというPrologの述語のベクトル処理の様子をしめしている．このばあいには，図にしめしたcarcdrの2つの節の計算が[a|b]，[c|d]という入力データに関して独立に実行され，独立に結果をえている．これらの節が出力するベクトルをまとめてあつかうためにつなぎあわされ，その結果として自然にマルチ・ベクトルが生成される．

ただし，正確にいえば図9.2においては入力したベクトルもただひとつの部分ベクトルからなるマルチ・ベクトルの形式をしている．したがって，入力とくらべて出力においてはマルチ・ベクトルを構成する部分ベクトルの数がふえているだけである．このように部分ベクトルが1個のときにもマルチ・ベクトルを使用することにより，入力するベクトルが1個のときも複数個のときも同一の入出力インタフェースですむようになり，したがってひとつのプログラムで両方のばあいをあつかうことができるようになる．

6.4節でしめした完全ORベクトル化の方法においては，マルチ・ベクトルの生成がおわるとただちに図9.3のようにしてただひとつのベクトルに併合される(部分ベクトルの併合)．しかし，6.5節でしめした並列バックトラック化においてはマルチ・ベクトルはすぐには併合されず，バックトラックの単位として使用される．部分ベクトルの併合

[注2] なお，マルチ・ベクトルによく似たデータ構造として6.10節でしめしたフレームがある．しかし，フレームは部分ベクトルが同質の値をふくむとはいえないので，マルチ・ベクトルとはみなさないことにする．

に関しては9.6節でくわしく説明する．

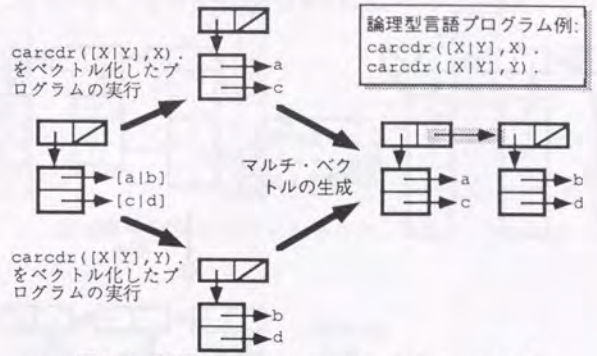


図9.2　解探索におけるマルチ・ベクトルの生成

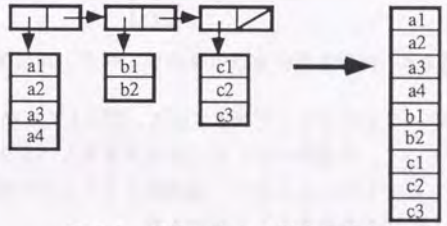


図9.3　部分ベクトル併合の基本

9.3.2 ストリーム処理における使用

第7章では，逐次論理型言語PrologおよびGHCによる素数生成のプログラムを手動で変換してベクトル処理可能なプログラムを生成し，その性能測定結果をしめした．このベクトル化法は任意の論理型言語によるプログラムに適用できるわけではない．しかし，論理型言語で記述されたプログラムにかぎらず，ストリーム処理，いいかえればフィルタリング処理においてはばひろくつかえるものだとかんがえられる．

素数生成を例として，ストリーム処理におけるマルチ・ベクトルのつかいかたを図9.4に例示する[注3]．ここでは，整数列をふくむマルチ・ベクトルに対する素数の倍数のフィルタリング処理のくりかえしの結果として素数が生成される．フィルタすべき整数列のながさNが非常におおきいばあいには，これらをひとつのベクトルにいれて一度に処理するのは得策でない．なぜなら，この処理には，入出力ベクトルのほかにもすくなくとも$O(N)$の膨大な作業記憶が必要だからである．また，整数列を複数のばらばらのベ

[注3] ただし，この図は第7章における処理をずっと簡略化したかたちでしめしている．

クトルにわけていれておくと，管理が容易でない．したがって，整数列をマルチ・ベクトルのかたちにするのがベクトル処理のために適切だということができる．

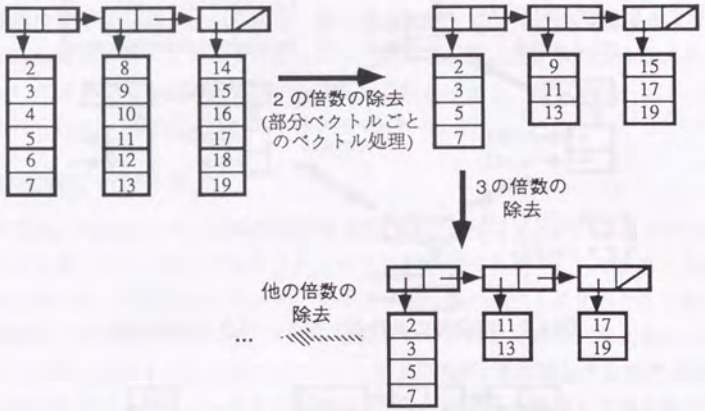


図9.4　素数生成におけるマルチ・ベクトルの使用

素数生成のばあいは，フィルタリングがすすむと，図9.4にしめしたように部分ベクトル要素数が減少していく．要素数がすくなくなりすぎるとベクトル処理性能が低下する．したがって，第7章でしめしたように，適当なタイミングで部分ベクトルの併合をおこなうことにより，性能を改善することができる．

9.3.3 ソートにおける使用

M-680H IDPにおいても，マージ・ソートをおこなう際にマルチ・ベクトルにちかいデータ構造が使用される(図9.5)．ただしこのばあいには，図9.5でハッチングをほどこした部分ベクトルの先頭をさすベクトルは実際には生成されない．なぜなら，各部分ベクトルの先頭アドレスは最初の部分ベクトルの先頭アドレスから計算することができるからである．各部分ベクトルの先頭アドレスが計算できるのは，全要素数が2のべき乗のばあいはこの疑似マルチ・ベクトルのすべての部分ベクトルの要素数が一定であり，2のべき乗でないでないばあいでも最後の部分ベクトルをのぞけば各部分ベクトルの要素数は一定になるからである．しかし，利点があるかどうかはべつとして，部分ベクトルの要素数が一定にならないソート方法は容易につくることができる．このようなソート方法をベクトル処理しようとすればハッチングした部分も必要になり，図9.1 (c) の形式のマルチ・ベクトルが必要になる．マージ・ソートにかぎらず，計算時間が $O(N \log N)$ であるソート方法をベクトル処理しようとすれば，マルチ・ベクトルまたはそれにちか

いデータ構造が必要になるとかんがえられる．

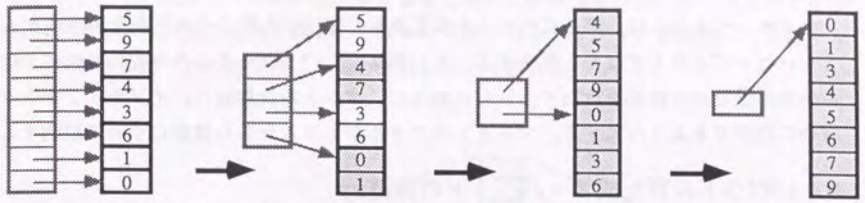


図9.5　M-680H IDPにおけるマージ・ソートの方法

9.4 マルチ・ベクトルの機能

マルチ・ベクトルは，第1にベクトルの要素数が変化する際にベクトルを再生成するオーバヘッドをなくすはたらきをする．また第2に，バックトラックや並列処理における処理単位という役割をはたす．さらに第3に，ベクトル計算機のパイプラインをより有効に利用できるようにする．マルチ・ベクトルがもつこれらの機能について説明する．

9.4.1 ベクトル再生成オーバヘッドの回避

解探索においてマルチ・ベクトルを使用するおもな理由は，ベクトル再生成による実行時間や記憶消費量のオーバヘッドをさけることにある．数値計算であつかうベクトルは，処理がすすんでも通常は要素数が一定である．これに対して，記号処理におけるデータ構造をベクトルに変換したばあい，結果としてえられるベクトルは要素数 (ベクトル長) が一定ではなく，処理をへるたびに要素数が変化するばあいがおおい．要素数が減少するばあいにはもとのベクトルにわりあてられた領域をそのまま使用することが可能だが，要素数が増大するばあいにはそのベクトルの領域の再わりあてが不可欠である．しかし，要素数が増大するたびにベクトル全体をつくりかえると，つぎのような問題点が生じる．第1に，ベクトル再わりあてのたびにもとのベクトルの要素をあたらしいベクトルにすべて複写しなければならないため，実行時間がふえる．第2に，ベクトル再わりあてによって記憶消費量がふえる．マルチ・ベクトルを使用すれば，これらのオーバヘッドをさけることができる．図9.2においても，複数のベクトルの複写をともなう併合によって生じるベクトル再生成オーバヘッドをへらすためにマルチ・ベクトルを生成しているとかんがえることができる．

9.4.2 バックトラック・並列処理の処理単位

マルチ・ベクトルはまた，並列バックトラックや並列処理における処理単位をきめるという点でも重要である．

第1に並列バックトラックについてのべる．並列バックトラック技法については6.5節で説明したが，これは解探索のベクトル処理において使用される，くみあわせ爆発をふせぐことをおもな目的とするプログラミング技法である．

図9.6にしめすように，並列バックトラック技法においてはマルチ・ベクトルを構成するひとつの部分ベクトルが処理単位となる．すなわち，並列バックトラック技法においては，ひとつの部分ベクトルの要素である部分解の数が膨大になり処理単位をわけたほうがよい状態になると，部分ベクトルの分割がおこなわれる．そして，それ以降の計算は部分ベクトルごとに逐次におこなわれる．図9.6においては，マルチ・ベクトル *m1* を構成する唯一の部分ベクトル *p1* が *p21* と *p22* とに分割され，それらを要素とするマルチ・ベクトル *m2* がつくられている．分割の結果としてえられたマルチ・ベクトルを構成するひとつの部分ベクトルに関する計算から解がえられなかったとき，またはそこから解がえられたがさらに解をもとめるばあいには，バックトラックが発生してそのマルチ・ベクトルのつぎの部分ベクトルの処理がおこなわれる．

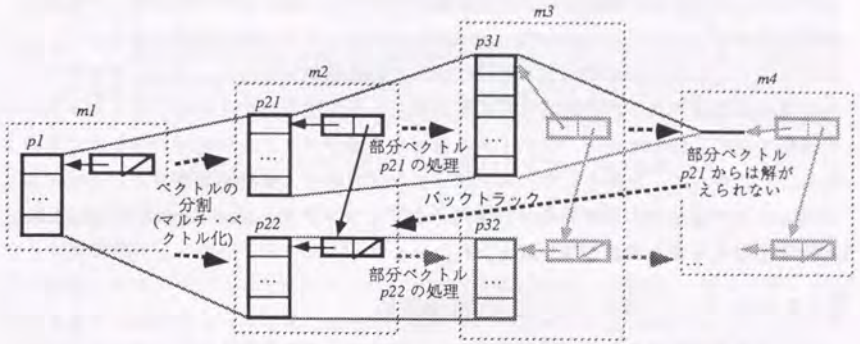


図9.6　並列バックトラック技法におけるマルチ・ベクトルの使用法

ひとつの部分ベクトルに関する計算でえられた解をふくむベクトルはもとの部分ベクトルに対応してつくられ，あらたなマルチ・ベクトルの部分ベクトルとなる．図9.6においてはマルチ・ベクトル *m2* の最初の部分ベクトル *p21* を入力してベクトル *p31* がつくられ，バックトラックのあとで部分ベクトル *p22* を入力してベクトル *p32* がつくられる．そして，*p31* および *p32* を要素とするマルチ・ベクトル *m3* がつくられる[注4]．マルチ・ベクトル *m4* も同様にしてつくられる．

上記のことから，並列バックトラック技法においてはマルチ・ベクトルが重要な役割をはたしているということができる．なぜなら，すべての部分解がひとつのベクトルにふくまれているばあいはこのようにマクロに処理をわけることは困難であるから複数のベクトルが必要であり，さらにそれらのベクトルをまとめてあつかうにはそれらを連結してマルチ・ベクトルのかたちにしておくのが適当だとかんがえられるからである．

第2に，素数生成などのストリーム並列処理のベクトル計算機による実行についてのべる．このばあいには，マルチ・ベクトルが並列処理の処理単位として使用されうる．並列論理型言語による素数生成プログラムの動作は図9.4にしめしたが，第7章ではこのようなプログラムをベクトル処理するために，部分ベクトルを単位とするスケジュリングをおこなっている．

すなわち，整数列をふくむマルチ・ベクトルに対するつぎのような素数の倍数のフィ

[注4] *p31* がつくられた時点ではマルチ・ベクトル *m3* の尾部は未定である．

ルタリング処理のくりかえしの結果として素数(素数を要素とするマルチ・ベクトル)が生成される．ひとつの部分ベクトルに関する素数の倍数のフィルタリング処理がまとめて一連のベクトル命令で実行される．この方法は，並列論理型言語のスケジュリング法としてはふかさ優先スケジュリング(depth-first scheduling)に対応する．しかし，ひとつのマルチ・ベクトルを構成することなる部分ベクトルに関するフィルタリング処理は，幅優先スケジュリング(breadth-first scheduling)によっておこなわれる．すなわち，ひとつのマルチ・ベクトルを構成する部分ベクトルの処理がおわると，そのマルチ・ベクトルの後続の部分ベクトルの処理はもっともひくい優先度をあたえられ，フィルタリングの過程で生じる他のマルチ・ベクトルを構成する部分ベクトルの処理が優先的に実行される．したがって，全体としてのスケジュリング戦略は，並列論理型言語の効率的なスケジュリング方法である有界ふかさ優先スケジュリング(bounded depth-first scheduling)に相当するやりかたでおこなわれる．

9.4.3 ベクトル・パイプラインの有効活用

パイプライン型ベクトル計算機は，ひとつのベクトルの各要素を短ピッチのパイプラインで計算する機構をもっている．このようなパイプラインをベクトル・パイプラインとよぶ．通常のベクトル計算機には，(疑似)マルチ・ベクトルの操作を支援するハードウェアは存在せず，マルチ・ベクトルの全部分ベクトルを一度にあつかえる命令は存在しない．したがって，複数の(部分)ベクトルのすべての要素に同一の演算をほどこすばあいには，ベクトルごとに1個の命令を実行する必要がある．このばあい，ベクトルの平均要素数が減少するとベクトル・パイプラインの使用効率は低下し，性能の低下をまねく．

しかし，もしマルチ・ベクトル支援機構をベクトル計算機に装備して，マルチ・ベクトル全体を処理するベクトル命令をつくれば，図9.7にしめすように，このような性能低下を部分的にふせぐことが可能になる．なぜなら，マルチ・ベクトルのデータ構造にあわせて，ハードウェアで最適なベクトル・パイプラインのスケジュリングをおこなうとともに，むだな処理をはぶくことができるからである．このような機構が存在するばあいにはマルチ・ベクトルを使用すれば，ひとつの命令であつかうことができないばらばらのベクトルを使用するのにくらべて，高速にベクトル処理をおこなうことが可能になる[注5]．

[注5] ただし，マルチ・ベクトル支援機構以外にこれに匹敵する高速化をする機構をつくることがまったくできないわけではない．また，このような機構においては，マルチ・ベクトルをつかわずばらばらのベクトルのままでも高速処理できる可能性がある．

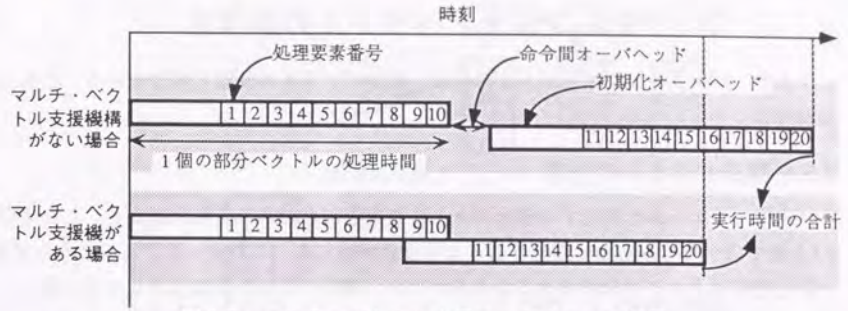


図9.7　マルチ・ベクトル支援ハードウェアの効果

図9.1 (a) や図9.1 (b) のようなマルチ・ベクトルを1命令であつかえるベクトル命令をもつ計算機はまだ存在しないとかんがえられるが，M-680H IDPは図9.5でしめしたような疑似マルチ・ベクトル(マルチ・ベクトルに類似したデータ構造)に関するマージ処理を支援する機構をもっている．M-680H IDPはこの機構があるためにマージ・ソートを高速に実行することができる．マージ・ソートのベクトル処理においては，その実行開始時の疑似マルチ・ベクトルを構成する各部分ベクトルすなわちソートされたデータ列の要素数は1であり，処理がすすむにつれて部分ベクトルが併合されていく．そのため，処理がすすんだあとは部分ベクトルを単位とするベクトル処理をおこなっても十分な性能がえられるが，開始直後はベクトル長(ベクトルの要素数)がみじかすぎるために，疑似マルチ・ベクトル全体を単位とするベクトル処理をおこなえる機構がなければ効率よく実行することができない．

また，2次元配列が1次元配列をならべたものだとみなせば，これはマルチ・ベクトルに類似したデータ構造だということができるが，ベクトル計算機ASC [Ramamoorthy 77]においては，2次元配列を処理する命令によってマルチ・ベクトル命令と同様のやりかたでベクトル・パイプライン使用効率をあげている．

マージ・ソートのばあいにかぎらず解探索やストリーム処理においても，(疑似)マルチ・ベクトル処理を支援する機構があればベクトル再生成オーバヘッドをさらにへらすことが可能になる．なぜなら，部分ベクトルの併合回数をへらすことによって部分ベクトルの平均要素数が減少しても，ベクトル処理性能を維持することができるからである．

9.5 マルチ・ベクトルの操作法

マルチ・ベクトルに関するおもな操作は，部分ベクトルの分割と併合である．これらの操作についてのべる．

9.5.1 部分ベクトルの分割

マルチ・ベクトルを構成する部分ベクトル要素数がおおきくなりすぎると，たとえば9.4.2節でのべたくみあわせ爆発のような不都合が生じる．したがって，このようなばあいには部分ベクトルの分割をおこなうのがよい．部分ベクトル分割法としては，図9.8にしめすような複数の方法がかんがえられる．すなわち，要素数が一定数をこえた部分ベクトルだけを分割し，他の部分ベクトルはもとのままとする単純分割法 (a) と，各部分ベクトルの要素数のバランスを考慮して併合をともなう分割をおこなう方法 (b) とがかんがえられる．しかし，(b) の方法は併合によるオーバヘッドがあることをかんがえると，通常は (a) のような単純な方法がよいであろう．

上記の分割法の選択よりも重要なのは，分割の条件をきめることである．すなわち，単純分割法を採用するとして，部分ベクトル要素数が何個以上になったときに分割するか，また1個の部分ベクトルを何個に分割するかをきめることがより重要である．どのような分割条件がよいかはハードウェアや応用に依存するので一概にいえず，ベンチマーク・テストをおこなったうえで決定するのがのぞましい．しかし，一般には要素数がベクトル・レジスタ長の数倍程度すなわち256～2048になったときに分割するようにし，必要以上に要素数が減少するのをさけるためには，部分解の増加がとくに急速でないかぎりは分割数は2とするのがよいであろう．

なお，部分ベクトル要素数の増大に対する対策としては，部分ベクトルの分割以外にも要素数の増大そのものの発生をなくす方法もある．並列バックトラック技法では，次節でのべる部分ベクトルの併合を意図的におこなわないかぎりは部分ベクトル要素数が増大することがない．それは，図9.2にしめしたように，部分解の数がふえるときにはもとの部分ベクトルと要素数がひとしいあらたな部分ベクトルを生成し，部分ベクトル要素数をふやすことはないからである．並列バックトラック技法においては，これにより部分ベクトルの分割と併合によるオーバヘッドをともにさけることが可能になっているということができる．

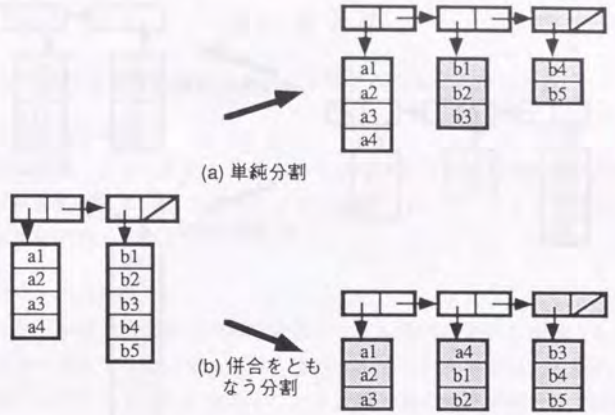


図9.8　部分ベクトルの分割法

9.5.2 部分ベクトルの併合

金田らの方法によるエイト・クウィーン問題のような解探索のプログラムや並列論理型言語による素数生成プログラムにおいては，計算がすすむにつれて部分ベクトル要素数が減少する．この条件は素数生成のばあいになりたつことはあきらかだが，第6～7章のベクトル処理方法においては，併合をおこなわないかぎりは部分ベクトル要素数は一定であるかまたは短縮するためになりたつ．このように部分ベクトルが短縮する一方なので，適当なところで図9.9に例示するような部分ベクトルの併合をおこなうのがよい．

部分ベクトルの併合法としては，**部分的併合** (図9.9 a) と**完全な併合** (図9.9 b) とがかんがえられる．完全な併合をおこなうと部分ベクトル要素数は最大になるので加速率は向上する．しかし単解探索のばあいには，部分ベクトル要素数をあまりおおきくすると不要な計算がふえるため，かえって効率が低下する (第2章参照)．また，部分ベクトル要素数がおおきくなると主記憶の消費がふえ，ばあいによってはデータが主記憶にはいりきらなくなって実行がつづけられなくなる．このようなばあいには，実行不能になるまえにあらかじめ部分的併合をおこなうようにするのがよい．

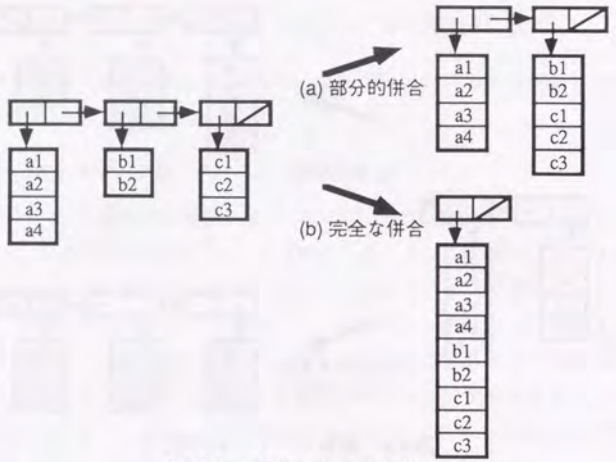


図9.9　部分ベクトルの併合

部分ベクトルの併合の一般的な効果を議論できるだけの経験は蓄積していない．しかし，素数生成プログラムにおける部分ベクトルの併合の効果に関するかぎりは第7章で測定結果をしめしている．それによれば，併合をおこなわないばあいにくらべて，ばあいによっては2倍以上の高速化を実現している．しかし，これは併合をおこなわないばあいの部分ベクトルの平均要素数が20以下のばあいであり，もとの平均要素数が61のばあいには併合してもほとんど高速化されていない．

どのような条件のもとで併合をおこなったらよいかは，分割のばあいと同様にハードウェアや応用に依存する．しかし，上記の結果からは，部分ベクトル要素数が50程度以下になったら併合をおこなうのがよいとかんがえられる．

9.6 まとめ

9.2 ～ 9.5 節における検討結果はつぎのようにまとめることができる．

❑ マルチ・ベクトルの応用

ベクトル記号処理，とくに並列バックトラック技法などによる解探索，素数生成のようなストリーム処理あるいはフィルタリング処理，マージ・ソートなどにおいてマルチ・ベクトルを使用することができる．

❑ マルチ・ベクトルの機能

マルチ・ベクトルは，解探索などの可変長ベクトル処理において問題となるベクトル再生成オーバヘッドを回避にやくだつ．またマルチ・ベクトルは，並列バックトラック技法におけるバックトラックやストリーム並列処理の処理単位としてはたらく．さらにマルチ・ベクトルは，ベクトル計算機にマルチ・ベクトル処理機構をもうけることによって，ベクトル・パイプラインの有効活用にやくだてることができる．

❑ マルチ・ベクトルの操作法

マルチ・ベクトルに対する重要な操作としては部分ベクトルの分割と部分ベクトルの併合とがあり，これらをおこなう条件を適切にきめることがベクトル処理性能をたかめるうえで重要である．

この研究の結果，ベクトル記号処理，とくに解探索，ストリーム処理，ソートなどにおいてマルチ・ベクトルを使用することができること，マルチ・ベクトルはベクトル再生成オーバヘッドを回避にやくだつこと，バックトラックや並列処理の処理単位としてはたらくこと，そしてベクトル・パイプラインの有効活用にやくだつことなどがわかった．これらの結果から，マルチ・ベクトルはリスト処理，データベース処理をはじめとするさまざまなベクトル記号処理において使用することができる非常に重要なデータ構造であると結論することができるだろう．

マルチ・ベクトルに関する今後の課題としては，この報告でしめした応用以外にもマルチ・ベクトルが有効であるかどうかをたしかめること，この報告でしめした以外の形式もふくめてどの形式のマルチ・ベクトルがより有効でありハードウェア支援に値するかをたしかめることなどがあげられる．また，ベクトル記号処理においてマルチ・ベクトル以外にも有用なデータ構造が存在するかどうか，それをどのようにつかうことができるかをしらべるのも今後の課題である．

第10章
結論

この章では，研究成果をまとめて結論をのべるとともに，今後の課題をまとめる．

10.1 研究成果

この節ではまずこの研究によってえられた成果の概要をしめし，その後，各項目について説明する．

10.1.1 研究成果の概要

この研究の目的は，Cray-1，HITAC S-810，HITAC M-680H IAP/IDP など，数値計算専用機として発展してきたパイプライン型ベクトル計算機を記号処理あるいは非数値処理に適用して応用範囲の拡大をはかるとともに，記号処理プログラムとくに論理型言語プログラムの実行の高速化をはかることであった．またこの研究においては，とくに Prolog で代表される論理型言語で記述された解探索などの AI プログラムの，パイプライン型ベクトル計算機による高速処理を可能にすることをおもな目標としてきた．この面では，Prolog にかぎらず Lisp をはじめとする他の言語で記述された AI プログラムに対しても適用することができる一般性のあるベクトル化技法すなわちベクトル処理可能なプログラムへのプログラム変換技法をめざして研究してきた．

この研究でえられた主要な成果はつぎのようにまとめることができる．

(1) ベクトル計算機の記号処理への適用可能性の実証
(2) 可変データ構造の変換にもとづくベクトル処理法開発
(3) 複雑なくりかえし制御構造のベクトル処理法開発
(4) 共有部分があるデータのベクトル処理法開発
(5) 論理型言語プログラムのベクトル化法開発

これらの各項目について，次節以降で説明する．

10.1.2 ベクトル計算機の記号処理への適用可能性の実証

現在のベクトル計算機は数値計算における汎用性を追求してきた．その結果，数値計算だけではなく，わずかなハードウェア拡張によってあるいは現在のままでも記号処理

に適したアーキテクチャをもっているとかんがえられ，ソフトウェアしだいで広範な記号処理に適用できるとかんがえられる．この事情はConnection Machineのような汎用の計算機構をそなえたSIMD型並列計算機においてもほぼおなじだとかんがえられる．このように，ソフトウェアしだいで汎用数値計算ベクトル計算機は記号処理に適用でき，極端にいえば「ベクトル計算機は記号処理も実行できる汎用計算機になる」という命題を実証することがこの研究のひとつの目的だった．これに対してこの研究は，従来は数値計算以外にほとんどつかわれることがなかったパイプライン型ベクトル計算機が，解探索などの記号処理においても有効であることをしめした．固定長の配列とDOループによって特徴づけられる数値計算のために開発されたベクトル計算機が，より複雑な可変データ構造とより複雑な制御構造をもつ記号処理にも使用することができ高速に実行することができることを，Nクウィーン問題などにおける実測をつうじて実証した．この研究でベクトル化可能になった記号処理プログラムとしては，つぎのようなものがある．

❑ 解探索問題

Nクウィーン問題のような解探索プログラム．ただし，バックトラックにもとづくかぎられた解法だけが対象となる．

❑ ある種の反復的なリスト処理

くりかえし構造の変換によってベクトル化できるリスト処理のプログラム．たとえば，リストを基本データ構造とするNクウィーン問題のプログラムなど．

❑ 複数データの登録・検索処理

複数データのハッシュ表への登録と検索，2分木への登録と検索など．

❑ 共有部分がある複数の構造データの処理

番地計算ソート，頻度ソートなどの番地計算をともなう配列のソート・アルゴリズム．リスト，2分木，グラフのかきかえなど．

❑ フィルタ処理

素数生成など．とくに，素数の無限列の生成のような，並列動作するフィルタ処理．

ベクトル処理が可能になった応用はいまのところおおいとはいえず，また十分なくりかえし処理がおこなわれるばあいにかぎって高速処理が可能になっただけである．この事実をかんがえれば，この研究がめざしていたように「汎用計算機」ということばをベクトル計算機に対してあたえるのは，現在の時点では過大評価であって適切でない．しかしながら，この研究によってベクトル計算機の記号処理への適用可能性をしめすという

目的は達成することができたとかんがえられる．

10.1.3 複雑なくりかえし構造の変換によるベクトル処理方法

この論文においては，くりかえし構造の変換すなわちFortranにおけるループ構造の変換にもとづくベクトル化技法を拡張したくりかえし構造の交換，くりかえし構造の1重化というプログラム変換戦略をしめし，多重のくりかえしの最内側のくりかえしがベクトル処理可能でない記号処理プログラムをベクトル化するためのくりかえし構造の交換法として最大回数反復法と残存要素検出法とを提案した．また，全解探索および単解探索のプログラムをベクトル計算機で高速実行するための並列バックトラック技法を提案した．これらについて順にのべる．

❑ くりかえし構造の変換

数値計算においては大半の処理はDOループで記述される．すなわち，くりかえし回数がループの実行開始前に確定する．また，ループ内に条件文が存在しないか，または比較的単純な条件制御がつかわれることがおおい．しかし，記号処理においてはさまざまな制御構造がつかわれる．くりかえし回数がループ実行中にきまるループ長可変ループ(**while**型ループ)，各種の再帰よびだし，バックトラックなどがつかわれる．これらのくりかえし構造がうまくベクトル計算機であつかえなかったことが，これまでベクトル計算機の記号処理への応用，あるいはSIMD型計算機の応用拡大をさまたげていたとかんがえられる．

これらの複雑なくりかえし制御構造をベクトル化するうえでとくに重要な技術はくりかえし構造変換法である．数値計算プログラムに関するもっとも重要なループ構造変換法は，ベクトル化の基本であるループ分散を除外すればループ交換とループ1重化である．これらの技法は，再帰よびだしなどのループ以外のくりかえし構造にも拡張することができ，もとのままの構造ではベクトル化できないもしくはもとのままの構造では十分な加速率がえられない記号処理プログラムをベクトル化可能にしたり加速率をたかめたりするためのくりかえし構造変換戦略として重要である．それらの戦略とは，くりかえし構造の交換とくりかえし構造の1重化である．

最内側くりかえしが可変長でしかもその最大値がわからないばあいは，くりかえし回数を決定する部分を分離してスカラ処理する以外に適当なベクトル化方法は存在しないが，2重のくりかえし構造の外側が固定長のばあいあるいは最内側くりかえしの最大値がわかっているばあいは，最大回数反復法または残存要素検出法によってベクトル化することができる．これらの方法は，**while**型のループにかぎらず，再帰よびだしやバックトラックで形成されたくりかえし構造にも適用することができる．

❑ 解探索のベクトル処理法

解探索をベクトル処理するために，完全ORベクトル計算法および並列バックトラック技法を提案した．完全ORベクトル計算法は，逐次処理による解探索でバックトラックをくりかえしながらおこなわれる計算を，すべての解候補をベクトルの要素とすることによって，解探索のベクトル処理を可能にする方法である．この方法は論理型言語における一種のOR並列計算をおこなうところから，ORベクトル計算法という名称をつけている．また，並列バックトラック技法は，完全ORベクトル計算法において問題になる記憶消費量の爆発すなわちベクトルの要素数がおおきくなりすぎるという問題を，ベクトルの分割と一種のバックトラック処理によって解決するための技法である．並列処理とバックトラック処理とをくみあわせたことから，この技法を並列バックトラック技法とよんでいる．これらの方法によって，Nクウィーン問題をはじめとする解探索問題をベクトル計算機をつかってとくことが可能になった．

10.1.4 可変データ構造の変換にもとづくベクトル処理法

この論文においては，ポインタを使用したリストなどの可変構造データのデータ構造変換によるベクトル化法すなわちプログラムの変換方法を提案し，そのベクトル化法とベクトル計算機のリスト・ベクトル処理機能などにもとづく記号処理プログラムのベクトル処理法を提案した．そして，これらのデータ構造のベクトル処理のために，3種の条件処理機能とともに，第2世代以降のベクトル計算機がもっているリスト・ベクトル処理機能がとくに重要であることを指摘した．また，変換後のプログラムにおけるデータ構造として，とくにマルチ・ベクトルが重要であることを指摘した．

この論文で提案したベクトル記号処理法はつぎのようにまとめることができる．

❑ リスト・ベクトル処理にもとづくポインタ・データ処理

くりかえし構造の交換をおこなったばあいには，線形リスト，木，グラフのようなポインタをつかったデータ構造は，インデクスまたはポインタを要素とするベクトルすなわちインデクス・ベクトルに変換される．そして，プログラム変換後のプログラムにおいては，ポインタをたぐる操作はベクトル計算機がもつリスト・ベクトル処理機能を使用することによって高速に実行される．ベクトル計算機における記号処理の性能をきめるもっとも重要な機能がこのリスト・ベクトル処理機能である[注1]．

❑ 3つの条件制御法にもとづく可変長データ処理

可変長データ処理のためには，条件制御が必要である．この論文においては，可変長データ処理法としてマスク演算方式，インデクス方式および圧縮方式を提案した．これらは，“日本製スーパコンピュータ”において実現されている条件制御の各方式[Kamiya 83]を可変長データ処理に応用したものである．これらのうちのいずれがより適しているかは，ばあいによる．たとえば，ORベクトル計算法によるNクウィーン問題のプログラムのS-810における実行に関するかぎりは，これらの可変長データ処理方式の間に有意な性能差はみられなかった．一方，配列線形検索においてはマスク演算方式にもとづく方式がよりよい性能をしめした．M-680H IDPにおいてはインデクス方式をサポートするハードウェアがあたえられているため，それが有利である．

❑ ベクトル処理による動的記憶管理

記号処理においては，リスト処理におけるcons操作(リスト・セルの生成)で代表される動的記憶わりあてが重要である．この論文においては，ベクトル処理によって複数の記憶要素の動的記憶わりあてを実行する方法をしめした．

データ構造をベクトルに変換しても，結果としてえられるベクトルは数値計算におけるようにベクトル長が一定ではなく，ひとつの処理をへるたびにベクトル長が変化するばあいがおおい．ベクトル長が変化するたびにベクトル全体をつくりかえることによるベクトル再生成オーバヘッドをさけるには，可変長ベクトルを連結したデータ構造であるマルチ・ベクトルを使用すればよいことをしめした．マルチ・ベクトルはまた，バックトラックや並列処理における処理単位をきめるという点でも重要である．

エイト・クウィーン問題のような解探索のプログラムや並列論理型言語による素数生成プログラムのようなばあいには，計算がすすむにつれて部分ベクトル長が短縮する[注2]．この部分ベクトル長短縮による性能低下をさける方法として部分ベクトル併合法を提案し，素数生成においてその効果をたしかめた．

10.1.5 共有部分があるデータのベクトル処理法

この論文においては，共有部分がある複数の(または1つの)データの更新に関するくりかえし処理(すなわちループや再帰よびだし)をベクトル化するための上書きラベル・フィルタ法を提案して，そのアルゴリズムをしめすとともに，そのいくつかの重要な性質をしめした．また，複数データのハッシング，番地計算ソートなどの応用において上書きラベル・フィルタ法の有効性を実証した．上書きラベル・フィルタ法によってベクトル処理すると，衝突検出という，ベクトル処理にともなうオーバヘッドが生じるが，衝突が十分すくなければスカラ処理にくらべて処理全体で5～11倍程度加速されうることがしめされた．また上書きラベル・フィルタ法は，各プロセス(単位処理)が1個ずつ

[注1] したがって，リスト・ベクトル処理機能をもたないベクトル計算機においては，ベクトル記号処理の可能性はいちじるしくひくいといわざるをえない．

[注2] この論文でのべたベクトル処理方法では，一般に，併合をおこなわないかぎりはマルチ・ベクトルの部分ベクトル長は一定であるかまたは短縮する．

の共有データ(正確には共有されている可能性があるデータ)を処理するばあいだけではなく，各プロセスが複数個の共有データを処理するばあいにも拡張できることをしめした．

10.1.6 論理型言語プログラムのベクトル化法

この論文においては，ORベクトル化にもとづいて逐次論理型言語プログラムを自動的にベクトル化するための方法を，非常に限定された範囲のプログラムに対してではあるが，しめした．そして，ORベクトル化にもとづく自動ベクトル化と実行をおこなう処理系をS-810上に試作して，Nクウィーン問題のプログラムなどにおいて高速化がはかられることを実証した．また，並列論理型言語で記述されたプログラムの手動変換による実行結果もしめした．ここでは開発した逐次論理型言語処理系を中心として論理型言語プログラムのベクトル化法および実行方法についてまとめ，最後に並列論理型言語のばあいの補足をする．

ベクトル計算機のための論理型言語処理系の処理手順においてもっとも重要なのはベクトル化であり，その対象言語としてのベクトル中間語の設計も非常に重要である．試作処理系においては中間語として論理型言語を採用した．ベクトル化以外の重要な機能としては，決定性判定，決定化，制御構造変換，データフロー解析などのステップがある．

ベクトル化後のプログラムの実行法に関していえば，完全ORベクトル処理方式と並列バックトラック方式とがあるが，これらに関しては，解探索などに関係してすでにのべた．論理型言語プログラムのベクトル処理に関して特記すべきことは，中間語プログラムを変更せずに完全ORベクトル処理方式と並列バックトラック方式とをきりかえる方法を開発したことである．この方法は，8.3節でのべたように，論理型中間語を採用することにより可能になった．一方，開発された試作処理系は，完全ORベクトル処理方式にもとづいている．実用的なベクトル処理のためには並列バックトラック処理方式による実行が不可欠だが，現在のところ並列バックトラック処理方式については実行シミュレータを開発するにとどまっている．

並列論理型言語においては，ベクトル処理のための一般的なデータ構造変換法は確立されていないため，いまのところ自動ベクトル化はできない．しかし，並列論理型言語が得意とするデータのフィルタリングをおこなうプログラムにおいては，おおくのばあい，手動でマルチ・ベクトルへの変換をおこなうことによってベクトル処理が可能になるとかんがえられる．そこでわれわれは，フィルタリングのプログラムの例として，エラトステネスのふるいによる素数生成のプログラムを手動でベクトル化し，性能を測定した．性能向上のためマルチ・ベクトルの併合をとりいれたが，加速率は1.7倍にとど

まった．十分なベクトル長がえられているのに加速率がひくいのは，スカラ処理部分のオーバヘッドがたかいためだとかんがえられる．

10.2 結論

この研究の目的は，ベクトル計算機の応用範囲を拡大することと，記号処理・論理型言語実行のベクトル計算機による高速化をはかることだった．これらに関して順にのべる．

まずベクトル計算機の応用範囲を拡大することに関していえば，いまだ実用化にはいたっていないものの，この論文で提案した可変データ構造のマルチ・ベクトルなどへの変換にもとづくベクトル処理法，並列バックトラック技法・くりかえし構造の交換や1重化などの複雑なくりかえし制御構造のベクトル処理法，共有部分があるデータのベクトル処理法などによって，またベクトル計算機がもつマスク演算機能をはじめとする各種の条件処理機能やリスト・ベクトル処理機能を使用することによって，これまでベクトル処理できなかったさまざまなリスト処理をはじめとする各種の記号処理をベクトル処理可能にすることができ，ある程度目的を達成することができたとかんがえられる．

応用範囲の拡大に関してさらにいえば，この研究によって**汎用数値計算ベクトル計算機は汎用記号処理計算機になる**という命題を部分的に実証することができたとかんがえている．数値計算とくらべて記号処理においては必要な機能としてもとめられるものが基本的にことなるということはなく，単に重点がことなるだけである．記号処理においては数値計算機能は相対的に重要度がひくくなり，リスト・ベクトル機能やマスク演算機能，ベクトル圧縮命令などの重要度がたかくなる．しかし，これらの機能は数値計算においても必要であり，したがって現在のベクトル計算機にはとりいれられている機能である．ただし，IAP のような汎用スカラ計算機の付加機構においては，パイプライン型ベクトル計算機におけるのとはちがって，実験対象とした記号処理においては十分な加速率をえることができなかった (第2章)．

つぎに記号処理・論理型言語実行のベクトル計算機による高速化をはかることに関していえば，まず，さまざまな記号処理アルゴリズムから手動でベクトル処理アルゴリズムを構成し，ベクトル計算機においてたかい加速率で実行することができたという点では目的を達成することができたとかんがえられる．一方，論理型言語プログラムについては，そのごく一部が自動ベクトル化によってベクトル計算機で処理できるようになったという点では成果があったが，このプログラム変換法の適用範囲のせまさのために，実用にはほどとおいところにあり，その点においては目的はまだ達せられていないということができる．

10.3 今後の課題

各成果項目に対応させて，今後の課題を説明し，最後にそれらをまとめる．

10.3.1 複雑なくりかえし構造の変換によるベクトル処理法

複雑なくりかえし制御構造の変換に関しては，つぎのような課題がある．

まず，くりかえし構造の変換に関する最大の課題は，第3～4章でしめした以外の種類のプログラムに関しても，自動ベクトル化が可能になるところまで変換手順をより精密にすることだとかんがえられる．また，これまでの研究では配列線形検索などの非常にかぎられた応用プログラムへの適用をこころみただけであるが，今後はほかのプログラムへも適用をはかり，どのようなばあいにいずれの方法を適用することがより適切かを決定することもひとつの課題である．

つぎに，解探索のベクトル処理に関しては，これまでに並列バックトラック技法の適用対象となったのは単純な全解探索や単解探索のプログラムであるが，この技術を実用化につなげるためには，こまかい枝刈りなどの制御を必要とする分岐限定法や各種の AI 近似最適解探索アルゴリズムのベクトル化をめざすべきだとかんがえられる．このような陽な制御をふくんだバックトラック・アルゴリズムをベクトル化するためには，かぎられた範囲の論理型言語プログラムからの変換だけでなく，C や Lisp などの手続き型言語あるいは関数型言語からの変換を可能にすることが必要だとかんがえられる．また，並列バックトラック計算法において短縮したベクトル長を増大させるための残留バックトラック方式を研究するべきだとかんがえられる．

10.3.2 可変データ構造の変換にもとづくベクトル処理法

可変データ構造の変換に関しては，つぎのような課題がある．まずマルチ・ベクトルに関しては，この報告でしめした応用以外にもマルチ・ベクトルが有効であるかどうかをたしかめること，この報告でしめした以外の形式もふくめてどの形式のマルチ・ベクトルがより有効でありハードウェア支援に値するかをたしかめることなどがあげられる．また，ベクトル記号処理においてマルチ・ベクトル以外にも有用なデータ構造が存在するかどうか，それをどのようにつかうことができるかをしらべるのも今後の課題である．

10.3.3 共有部分があるデータのベクトル処理法

共有部分があるデータのベクトル処理法に関しては，つぎのような課題がある．まずこれまでベクトル化できないとかんがえられていたさまざまなアルゴリズムに上書きラベル・フィルタ法を適用することがあげられる．とくに，木の再バランス (rebalancing) のアルゴリズムのように，ひとつのプロセスが複数のデータ要素をかきかえる処理への

適用をこころみるべきだとかんがえられる．また，第5章ではデッドロックをふせぐためのひとつの方法をしめしたが，そのためのよりよい方法をみつけることも今後の課題としてあげることができる．

10.3.4 論理型言語プログラムのベクトル化法

自動ベクトル化可能なプログラムの範囲を拡大することは，困難ではあるが重要な課題だとかんがえられる．自動 OR ベクトル化をめざすアプローチとしては2とおりがかんがえられる．

ひとつは，この論文で中心的にのべてきたボトム・アップの研究をつづけることである．すなわち，変換後のプログラムの性能を第1にかんがえ，つづいて性能をおとさずに適用範囲をひろげる方法をさぐるアプローチである．これはもっとも困難な課題であり，解決のためにはおおきなブレーク・スルーが必要だとかんがえられる．論理型言語のベクトル化にかぎらず，ベクトル記号処理におけるこのようなブレーク・スルーにいたるためのアプローチとして，特定の (小規模または大規模) 応用プログラムのベクトル化法に関する研究をつみかさねる地道なアプローチが重要だとかんがえられる．この研究をはじめるきっかけになったのが N クウィーン問題のベクトル化に成功したことであること，ハッシングのベクトル化法の研究が共有データのベクトル化法につながったことなどをかんがえると，このような研究をさらにべつのプログラムに関してもおこなっていくことがブレーク・スルーにつながる可能性がたかいとかんがえられる．

もうひとつは，6.7節および第6章の付録でしめしたトップ・ダウンの研究である．すなわち，変換できるプログラムの範囲を第1にかんがえ，つづいて変換されたプログラムをいかにして最適化するかをかんがえるアプローチである．研究の対象はいささかことなるが，Nilsson のアプローチはこれにちかい．OR ベクトル化あるいは並列バックトラック化に関してはこのトップ・ダウン・アプローチはまだふかめられていないため，今後の研究に期待することができる．しかし，うめられるべきボトム・アップ・アプローチとのあいだの溝は，いまもなお非常にふかい．

自動ベクトル・コンパイラに関していえば，完全 OR ベクトル化方式ではなく並列バックトラック技法にもとづく実行系の開発がひとつの課題である．

また，AND ベクトル化に関する今後の課題としては，まず，手動ベクトル化されたプログラムにおけるオーバヘッドを減少させて，加速率の向上をはかることがあげられる．また，リストの CDR コーディングにもとづくベクトル化をさまざまなプログラムに適用をこころみ，それによって第7章でしめした方法の発展をはかることである．容易ではないが，このような研究をつうじて AND ベクトル化に関しても自動ベクトル化をめざすことがより究極の目的である．

10.3.5 他の課題とまとめ

この節では，前節までに分類できなかった課題をしめしたのち，課題をまとめる．

第1に，この論文においては実測はすべて日立製ベクトル計算機によっておこなってきた．同様の機能をもつ他社のベクトル計算機において実測をおこなうことが，この論文でしめした各種の方法の適用範囲のひろさを実証するうえで必要であろう．第2に，序章で課題としてとりあげた Lisp やエキスパート・シェルなどへの記号処理ベクトル化技術の適用はそのまま課題としてのこされている．第3に，この研究の成果を，今後しだいにパイプライン型ベクトル計算機にとってかわるとかんがえられる SIMD 型並列計算機に適用することも重要な課題である．第4に，この研究を発展させるなかから，ベクトル計算機や SIMD 型並列計算機に付加すべき機能をあきらかにし，そのアーキテクチャへの提案をおこなっていくことが，重要な課題のひとつであろう．

これまで，比較的こまかい課題を列挙してきたが，いうまでもなく，これらの課題はベクトル計算機の応用範囲を拡大し，記号処理・論理型言語実行のベクトル計算機による高速化というこの研究の目的に，この研究をさらに発展させることによって，ちかづくための課題である．しかし，これらの目的にちかづくためには，この研究の成果をあらたな目でみなおしてあらたなアプローチをさぐることも，また必要だとかんがえられる．

参考文献

[Abe 90a] 阿部 一裕, 安井 裕: スーパーコンピュータ（ベクトル計算機）のための並列 LISP コンパイラ, *情報処理学会第 40 回全国大会報告集*, 1G-8, pp. 665–666 (1990).

[Abe 90b] 阿部 一裕, 安井 裕: スーパーコンピュータのためのベクトル化 Lisp コンパイラ, *情報処理学会記号処理研究会*, 54-2 (1990).

[Aho 74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D.: *The Design and Analysis of Computer Algorithms*, Addison Wesley (1974). 邦訳: 野崎 昭弘 他訳: アルゴリズムの設計と解析 I, II, サイエンス社 (1977).

[Aho 86] Aho, A. V., Sethi, R., and Ullman, J. D.: Compilers — Principles, Techniques, and Tools, Addison Wesley, Massachusetts (1986).

[Allen 84] Allen, J. R., and Kennedy, K.: Automatic Loop Interchange, *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction*, *SIGPLAN Notices*, Vol. 19, No. 6, pp. 233–246 (1984).

[Appel 89] Appel, A. W., and Bendiksen, A.: Vectorized Garbage Collection, *Journal of Supercomputing*, Vol. 3, pp. 151–160 (1989).

[Backus 78] Backus, J.: Can Programming Be Liberated from von Neumann Style? A Functional Style and its Algebra of Programs, *Comm. ACM*, Vol. 21, No. 8 (1978).

[Batcher 68] Batcher, K. E.: Sorting Networks and Their Applications, *1968 Spring Joint Computer Conference*, pp. 307–314 (1968).

[Bawden 77] Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D.: LISP Machine Progress Report, *MIT AI Memo* No. 444 (1977).

[Baudet 78] Baudet, G., and Stevenson, D.: Optimal Sorting Algorithms for Parallel Computers, *IEEE Trans. Computers*, Vol. C-27, pp. 84–87 (1978).

[Bitner 75] Bitner, J. R., and Reingold, E. M.: Backtrack Programming Techniques, *Comm. ACM*, Vol. 18, No. 11, pp. 651–656 (1975).

[Brock 81] Brock, H, K., Brooks, B. J., and Sullivan, F.: DIAMOND: A Sorting Method for Vector Machines, *BIT*, Vol. 21, pp. 142–152 (1981).

[Brooks 85] Brooks, R., and Lum, L.: Yes, An SIMD Machine Can Be Used For AI, *Proc. Int. Joint Conference on Artificial Intelligence*, Los Angeles, pp. 73–79 (1985).

[Buchnolz 86] Buchnolz, W: The IBM System/370 Vector Architecture, *IBM Systems*

Journal, Vol. 25, No. 1 (1986).

[Clark 86] Clark, K., and Gregory, S.: PARLOG: Parallel Programming in Logic, *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, pp. 1–49 (1986).

[Codish 86] Codish, M., and Shapiro, E. : Compiling OR-parallelism into AND-parallelism, *Third Internationsl Conference on Logic Programming, Lecture Notes in Computer Science*, No. 225, pp. 283–297, Springer-Verlag (1986). Also in *New Generation Computing*, Vol. 5, pp. 45–61 (1987).

[Conery 81] Conery, J. S., and Killer, D. F.: Parallel Interpretation of Logic Programs, *In Proc. ACM 1981 Conference on Functional Programming Languages and Computer Architecture*, pp.163–170 (1981).

[Dijkstra 72] Dijkstra, E. W.: Notes on Structured Programming, in Dahl, O. — J., Dijkstra, E., and Hoare, C. A. R., *Structured Programming*, Academic Press (1972).

[Flenders 84] Flanders, P. M., and Reddaway, S. F.: Sorting on DAP, *Parallel Computing 83*, pp. 247–252, Elsevier Science Publishers B. B., North-Holland (1984).

[Floyd 84] Floyd, R.: Nondeterministic Algorithms, *J. ACM*, No. 14, pp. 636–644 (1967).

[Fuchi 87] 淵 一博 監修 : *並列論理型言語 GHC とその応用*, 共立出版 (1987).

[Furumasa 84] 古勝 紀誠, 渡辺 貞, 近藤 良三 : 最大性能 1.3 GFLOPS, マシン・サイクル 6 ns のスーパコンピュータ SX システム, *日経エレクトロニクス*, 1984.11.19, No. 356, pp. 237–272 (1984).

[Gonnet 87] Gonnet, G. H.: *Handbook of Algorithms and Data Structures*, Addison-Wesley, 1984. 邦訳 : 玄 光男 他 訳 : *アルゴリズムとデータ構造ハンドブック*, 啓学出版 (1987).

[Goto 86] Goto, A., and Uchida, S.: Toward a High Performance Inference Machine — The Intermediate Stage Plan of PIM —, *Future Parallel Computers, Lecture Notes in Computer Science*, No. 272, Springer-Verlag (1986).

[Hillis 85] Hillis, D.: *The Connection Machine*, MIT Press, Cambridge, Massachusetts (1985).

[Hillis 90] Hillis, D., and Kitsuregawa, M.: コネクション・マシン, パーソナル・メディア (喜連川 優 訳) (1990).

[Hirai 86] 平井 他 3 : ベクトル処理機能を利用した FP 処理系 VFP システム, *日本ソフトウェア科学会第 3 回大会論文集*, pp. 109–112 (1986).

[Hirakuri 83] 平栗 俊男, 田畑 晃, 槌本 隆光, 田中 尚三: マシン・サイクル 7.5 ns を達成した並列パイプライン処理方式のスーパコンピュータ FACOM VP, *日経エレクトロニクス*, 1983.4.11, pp. 131–155 (1983).

[Horikoshi 83] 堀越 彌, 梅谷 征雄 : 汎用計算機のための内蔵ベクトル演算方式, *情報処理学会論文誌*, Vol. 24, No. 2, pp. 191–199 (1983).

[Ishiura 86] 石浦 菜岐佐, 安浦 寛人, 矢島 修三 : ベクトル計算機による高速論理シミュレーション, *情報処理学会論文誌*, Vol. 27, No. 5, pp. 510–517 (1986).

[Ishiura 88] 石浦 菜岐佐, 高木 直史, 矢島 脩三 : ベクトル計算機上でのソーティング, *情報処理学会情報処理学会論文誌*, Vol. 29, No. 4, pp. 378–385 (1988).

[Jones 70] Jones, B.: A Variation on Sorting by Address Calculation, *Comm. ACM*, Vol. 13, No. 2, pp. 105–107 (1970).

[Kamiya 83] Kamiya, S., Isobe, F., Takashima, H., and Takiuchi, M.: Practical Vectorization Techniques for the "FACOM VP," *Information Processing '83*, pp. 389–394 (1983).

[Kacsuk 87] Kacsuk, P., and Bale, A.: DAP Prolog: A Set-oriented Approach to Prolog, *Computer Journal*, Vol. 30, No. 5, pp. 393–403 (1987).

[Kanada 85] 金田 泰 : スーパー・コンピュータによる Prolog の高速実行, *第 26 回プログラミング・シンポジウム報告集*, pp. 47–56 (1985).

[Kanada 87] 金田 泰 : ベクトル計算機による論理型言語プログラムの高速実行をめざして — 各種 OR ベクトル実行方式の実現と性能 —, *情報処理学会プログラミング言語研究会*, PL-87-12 (1987).

[Kanada 88a] Kanada, Y., Kojima, K., and Sugaya, M.: Vectorization Techniques for Prolog, *1988 ACM International Conference on Supercomputing*, pp. 539–549, St. Malo (1988).

[Kanada 88b] 金田 泰, 小島 啓二, 菅谷 正弘 : ベクトル計算機のための探索問題の計算法「並列バックトラック計算法」, *情報処理学会論文誌*, Vol. 29, No. 10, pp. 985–994 (1988).

[Kanada 89a] 金田 泰, 菅谷 正弘 : OR 並列実行のための論理型言語プログラムのベクトル化法, *情報処理学会論文誌*, Vol. 30, No. 4, pp. 495–506 (1989).

[Kanada 89b] 金田 泰, 菅谷 正弘 : プログラム変換にもとづくリストのベクトル処理方法とそのエイト・クウィーン問題への適用, *情報処理学会論文誌*, Vol. 30, No. 7, pp. 856–868 (1989).

[Kanada 89c] Kanada, Y., and Sugaya, M.: Vectorization Techniques for Prolog without Explosion, *International Joint Conference on Artificial Intelligence '89*, pp. 151–156 (1989).

[Kanada 90a] Kanada, Y. : A Vectorization Technique of Hashing and its Application to

Several Sorting Algorithms, *PARBASE-90*, IEEE (1990).

[Kanada 90b] 金田 泰, 菅谷 正弘: リストのデータ変換にもとづく Prolog プログラムのベクトル処理法とその評価, *情報処理学会第 41 回全国大会* (1990).

[Kanada 91a] 金田 泰, 菅谷 正弘: 共有部分がある複数データのベクトル処理方法, *情報処理学会第 42 回全国大会*, 4M-2 (1991).

[Kanada 91b] 金田 泰, 菅谷 正弘: ベクトル記号処理のためのデータ構造「マルチ・ベクトル」とその応用, *ソフトウェア科学会 8 回大会* (1991).

[Kanada 91c] Kanada, Y.: A Method of Vector Processing for Shared Symbolic Data, *International Conference on Supercomputing '91*, Albuquerque (1991).

[Kawabe 78] Kawabe, S., Kobayashi, F., Murayama, H., et al: S-820 — 2 GFLOPS Peak Performance by a Simgle Processor, *日経エレクトロニクス*, No. 437, 1988, pp. 111–125 (1988).

[Knuth 73] Knuth, D. E.: *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley (1973).

[Kobayashi 91] 小林 一隆, 阿部 一裕, 安井 裕: vmap マクロ, vmap 関数を用いたプログラミング及びベクトル化 Lisp コンパイラでの実行とその考察, *情報処理学会第 42 回全国大会*, 4M-1 (1991).

[Kojima 87] Kojima, K., Torii, S., and Yoshizumi, S.: IDP — A Main Storage Based Vector Database Processor, *1987 International Workshop on Database Machines*, pp. 60–73 (1987).

[Kojima 90] 小島 啓二, 鳥居 俊一, 吉住 誠一: ベクトル型データベースプロセッサ IDP, *情報処理学会論文誌*, Vol. 31, No. 1, pp. 163–173 (1990).

[Komatsu 86] Komatsu, H., Tamura, N., Asakawa, Y., and Kurokawa, T.: An Optimizing Prolog Compiler, *The Logic Programming Conference '86*, pp.143–149, Japan, (1986).

[Kuck 81] Kuck, D. J., Kuhn, R. H., Padua, D. H., Leasure, B., and Wolfe, M.: Dependence Graphs and Compiler Optimizations, *Proc. 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218 (1981).

[Levin 90] Levin, Stewart A.: A Fully Vectorized Quicksort, *Parallel Computing*, Vol. 16, pp. 369–373, (1990).

[Melville 80] Melville, R., and Gries, D.: Controlled Density Sorting, *Information Processing Letters*, Vol. 10, No. 4, pp. 169–172 (1980).

[Miki 91] 三木 良雄, 鈴木 敬, 高嶺 美夫: ベクトル計算機を用いた迷路法の高速化, *情報処理学会第 42 回全国大会* (1991).

[Mine 89] 峯 亮太郎, 辰口 和保, 村岡 洋一: ベクトル計算機上における並列構文解析の一手法, *情報処理学会第 38 回全国大会報告集*, 5P-1, pp. 907–908 (1989).

[Mishina 89] 三科 雄介, 小島 啓二: ベクトル演算むきテキストサーチアルゴリズム, *電子情報通信学会データ工学研究会, DE89-45, 信学技報*, Vol. 89, No. 335, pp. 73–80 (1989).

[Nagashima 86] Nagashima, S., Nakagawa, T., Omota, K., Miyamoto, S., Kawabe, S., and Tsuchiya, Y., Hardware Implementation of VELVET on the Hitachi S-810 Computer, *IEEE International Conference on Computer-Aided Design*, pp.390–393 (1986).

[Nakashima 83] 中島 秀之: *Prolog*, 産業図書 (1983).

[Nilsson 86] Nilsson, M.: — FLENG Prolog — The Language which turns Supercomputers into Parallel Prolog Machines, *Proc. Japanese Logic Programming Conference '86*, pp. 209–216 (1986). Also in Wada, E. (Ed.): Logic Programming '86, *Lecture Notes in Computer Science*, No. 264, pp. 170–179, Springer-Verlag (1987).

[Nilsson 87a] Nilsson, M., and Takana, H.: Implementing Safe GHC the Easy Way — by Compilation into Guard-free Form, *情報処理学会全国大会*, pp. 773–774 (1987.3).

[Nilsson 87b] Nilsson, M., and Tanaka, H.: The Art of Building a Paralell Logic Programming System, *Proc. Japanese Logic Programming Conference '87*, pp. 155–163. Also in Furukawa, H., Tanaka, H., Fujisaki, T. (Eds.): Logic Programming '87, *Lecture Notes in Computer Science*, No. 315, pp. 95–104, Springer-Verlag (1988).

[Nilsson 87c] Nilsson, M., and Tanaka, H.: A Proposal for Implementing GHC on the Connection Machine, *Proc. IEEE Region 10 Conf.* pp. 821–825, Seoul (1987).

[Nilsson 88a] Nilsson, M., and Tanaka, H.: Conferting FGHC Clauses with Guards into Clauses without Guards, *情報処理学会プログラミング言語研究会*, 88-PL-17 (1988).

[Nilsson 88b] Nilsson, M., and Tanaka, H.: SIMD Architecture and Superparallel Logic Programming, *情報処理学会計算機アーキテクチャ研究会*, 88-ARC-71, Section 71-16 (1988).

[Nilsson 88c] Nilsson, M., and Tanaka, H.: A Flat GHC Implementation for Supercomputers, *Fifth International Symposium on Logic Programming*, pp. 1337–1350 (1988).

[Nilsson 88d] Nilsson, M., and Tanaka, H.: Graph Algorithms for Supercomputers, *Proc. Int. Computer Symposium*, Tamkang University, Tamkang, Taiwan, Vol. 2, pp. 913–917, Tamkang, Taiwan (1988).

[Nilsson 88e] Nilsson, M., and Tanaka, H.: Massively Parallel Implementation of Flat GHC on the Connection Machine, *International Conference on Fifth Generation Computer Systems*, pp. 1031–1040 (1988).

[Nilsson 88f] Niisson, M., 田中 英彦: A 1.1 MLIPS (i.e. Hz) Flat GHC Interpreter for the Hitachi Supercomputer S-820, *情報処理学会第37回全国大会報告集*, 7Y-3, pp. 687–688 (1988).

[Nilsson 89] Nilsson, M.: Parallel Logic Programming for SIMD Supercomputers and Massively Parallel Computers, *東京大学大学院工学系研究科情報工学専門課程学位論文* (1989).

[Odaka 83] 小高 俊彦, 小林 二三幸, 河辺 峻, 長島 重夫: 最大性能が630 MFLOPS で1Gバイトの半導体拡張記憶が付くスーパコンピュータ HITAC S-810, *日経エレクトロニクス*, 1983.4.11, pp. 159–184 (1983).

[Okuno 84] 奥乃 博: 第3回 Lisp コンテストおよび第1回 Prolog コンテストの課題案, *情報処理学会記号処理研究会資料*, 28-4 (1984) .

[Osaka 82] 大阪大学計算センタ・ニュース, Vol. 12, No. 1, pp. 59–72 (1982).

[Ramamoorthy 77] Ramamoorthy, C. V., and Li, H. F.: Pipelined Architectures, *ACM Computing Surveys*, Vol. 9, No. 1, pp. 61–102 (1977).

[Reif 83] Reif, J. H., and Valiant, L. G.: A Logarithmic Time Sort for Linear Size Networks, *Proc. Fifteenth Annual ACM Symposium on the Theory of Computing*, pp. 10–16 (1983).

[Roensch 87] Roensch, W., and Strauss, H.: Timing Results of Some Internal Sorting Algorithms on Vector Computers, *Parallel Computing*, Vol. 4, pp. 49–61 (1987).

[Sakai 86] 坂井 修一: 並列計算機におけるスケジューリングと負荷分散, *情報処理*, Vol. 27, No. 9, pp. 1031–1038 (1986).

[Sedgewick 83] Sedgewick, R.: Algorithms, Addison-Wesley (1983).

[Shapiro 84] Shapiro, E.: Systoric Programming: A Paradigm of Parallel Processing, *Proc. Fifth Generation Computer Systems '84* (1984).

[Shapiro 86] Shapiro, E. Y.: Concurrent Prolog : A Progress Report, *IEEE Computer*, August 1986, pp. 44–59 (1986).

[Shimazaki 89] 島崎 真昭: ベクトル計算機上の FP 型言語の処理系, *電子情報通信学会技術研究報告*, CPSY89-21, pp. 39–44 (1989).

[Stolfo 86] Stolfo, S. J.: On the Limitations of Massively Parallel (SIMD) Architectures for Logic Programming, Proc. US-Japan AI Symposium, *J. Logic Programming*, No. 1, ICOT, Tokyo, Japan (1987).

[Stone 78] Stone, H. S.: Sorting on STAR, *IEEE Trans. Software Engineering*, Vol. 4, No. 2, pp. 138–146 (1978).

[Takemiya 90a] 武宮 博, 布川 博士, 白鳥 則郎, 野口 正一: 関数型言語 FP のベクトルプロッセサ向きコンパイル手法, *情報処理学会第40回全国大会報告集*, 6J-7, pp. 982–983 (1990).

[Takemiya 90b] 武宮 博, 布川 博士, 白鳥 則郎, 野口 正一: 並列処理関数に着目した関数型言語 FP のベクトル処理方法, *情報処理学会第41回全国大会報告集*, 1E-7, pp. 5–12 - 5–13 (1990).

[Tamaki 87] Tamaki, H.: Stream-based Compilation of Ground I/O Prolog into Commintted- choice Languages, *ソフトウェア基礎論研究会資料21-4*, pp. 21–27 (1987).

[Tatsuguchi 87a] 辰口 和保, 村岡 洋一: ベクトル計算機上の並列論理型言語処理系, *情報処理学会第35回全国大会報告集*, 5Q-1, pp. 753–754, (1987).

[Tatsuguchi 87b] 辰口 和保, 村岡 洋一: Parallel Logic Programming Interpreters on Supercomputers, *情報処理学会プログラミング言語研究会*, No. 14 (1987).

[Tatsuguchi 88] 辰口 和保, 村岡 洋一: ベクトル数値計算向き Prolog の提案, *情報処理学会第37回全国大会報告集*, 6Y-9, pp. 681–682 (1988).

[Tatsuguchi 88] 辰口 和保, 村岡 洋一: スーパーコンピュータ上の並列論理型言語処理系 — 制限 AND 並列処理のための中間言語について —, *ソフトウェア科学会第5回大会論文集*, B8-4, pp. 361–364 (1988).

[Torii 87a] 鳥居 俊一, 小島 啓二, 吉住 誠一, 河辺 峻, 高橋 政美, 久代 康雄: リレーショナル・データベースの処理速度向上を図る CPU 内蔵型データベース・プロセサ, *日経エレクトロニクス*, 1987.2.9, No. 414, pp. 185–210 (1987).

[Torii 87b] Torii, S., Kojima, K., Yoshizumi, S., Sakata, A., Takamoto, Y., Kawabe, S., Takahashi, M., and Ishizuka, T.: A Relational Database System Architecture Based on A Vector Processing Method, *Proc. Third International Conference on Data Engineering*, pp. 182–189 (1987).

[Torii 88a] Torii, S., Kojima, K., Kanada Y., Sakata, A., Yoshizumi, S., Takahashi, M.: Accelerating Non-Numerical Processing by An Extended Vector Processor, *Proc. Fourth International Conference on Data Engineering*, pp. 194–201 (1988).

[Torii 88b] 鳥居 俊一, 小島 啓二, 金田 泰, 坂田 明治, 吉住 誠一, 高橋 政美: 拡張ベクトル演算による非数値処理高速化, *電子情報通信学会研究報告*, DE88-8, pp. 57-64 (1988).

[Tsuda 85] 津田 孝夫, 国枝 義敏, 二宮 正和, 栗屋 徹: ループ間にまたがるデータ参照関係をもつ多重ループの自動ベクトル化, *情報処理学会情報処理学会論文誌*, Vol. 26, No. 3, pp. 536–544 (1985).

[Ueda 85a] 上田 和紀: Guarded Horn Clauses, *Logic Programming Conference '85*,

pp. 225–236 (1985). Also in *ICOT Technical report*, TR-103, Institute for New generation Computer Technology (1985-7), and *New Generation Computing*, Vol. 5, pp. 29–44 (1987).

[Ueda 85b] 上田 和紀：全解探索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回大会報告集*, pp. 145–148 (1985).

[Ueda 86] Ueda, K.: Making Exhaustive Search Programs Deterministic, *Third International Conference on Logic Programming*, *Lecture Notes in Computer Science*, No. 225, pp. 270–282, Springer-Verlag (1986).

[Uematsu 90] 植松 尚士, 小林 一隆, 安井 裕：LISP からのベクトルプロセッサの利用，*情報処理学会第40回全国大会報告集*, 1G-9, pp. 667–668 (1990).

[Wolfe 86] Wolfe, M.: Advanced Loop Interchanging, *Proc. of the '86 International Conference on Parallel Processing*, pp. 536–543 (1986).

[Yamaguchi 87] Yamaguchi, S., Bandoh, T., Kurosawa, K., and Morioka, M.: Architecture of High Performance Integrated Prolog Processor IPP, *Fall Joint Computer Conference*, pp.175–182 (1987).

[Yasumura 87] 安村 通晃：ベクトル化とプログラム変換，*知識情報処理シリーズ7「プログラム変換」*, pp. 121–134, 共立出版 (1987).

[Yasumura 90] 安村 通晃, 小島 啓二：スーパーコンピュータ上でのグラフ問題の高速化，*第30回プログラミング・シンポジウム報告集*, pp. 105–116 (1990).