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．

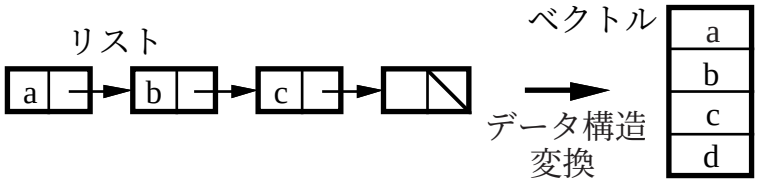


図 7.1　ベクトル処理可能なデータ構造への変換

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．

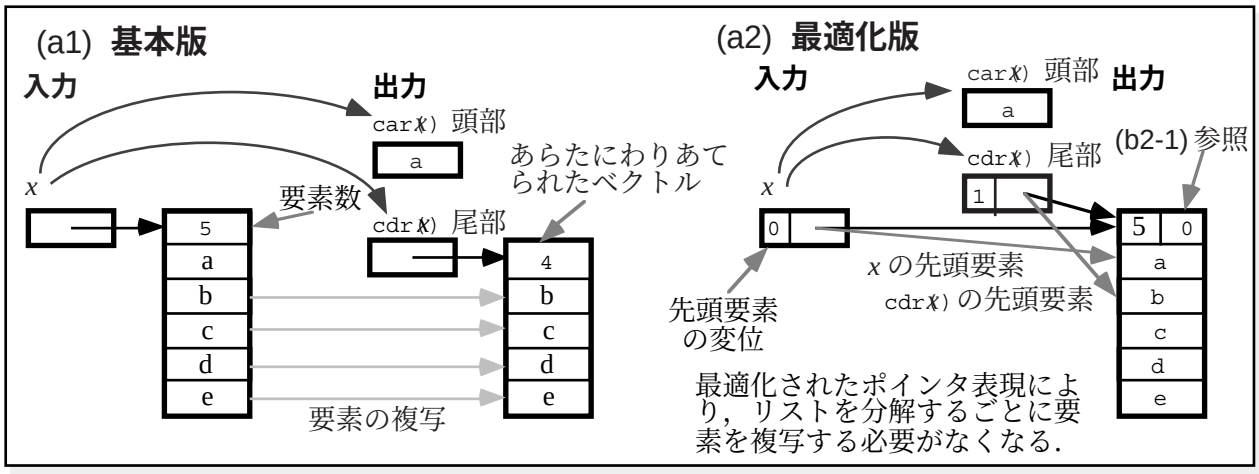
[2,3,4, ... ,1535] をベクトルとして生成する手続きである．v_iota は算術数列を生成する手続きであり，ベクトル中間語のくみこみ手続きとして用意されるべきものである．v_iota はベクトル計算機がもつ数列生成命令 (S-810 / S-820 においては VINC 命令) を使用することによってベクトル処理される．

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．

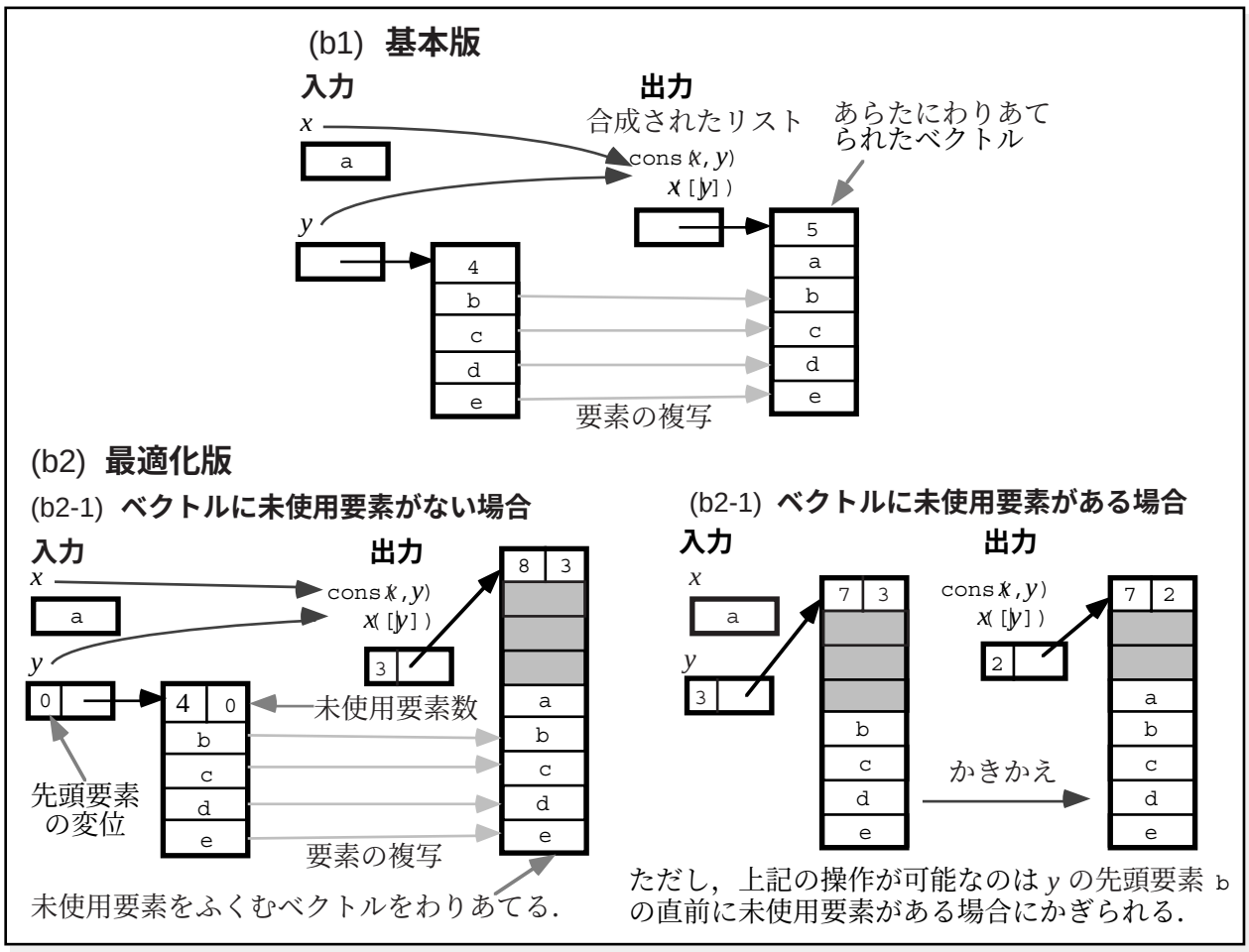
手続き v_sift の出力には変換前のプログラムと同様にリストを使用する．これは，この出力データに関してはベクトル処理がおこなわれない (ベクトル処理によって高速化されない) からである．

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．

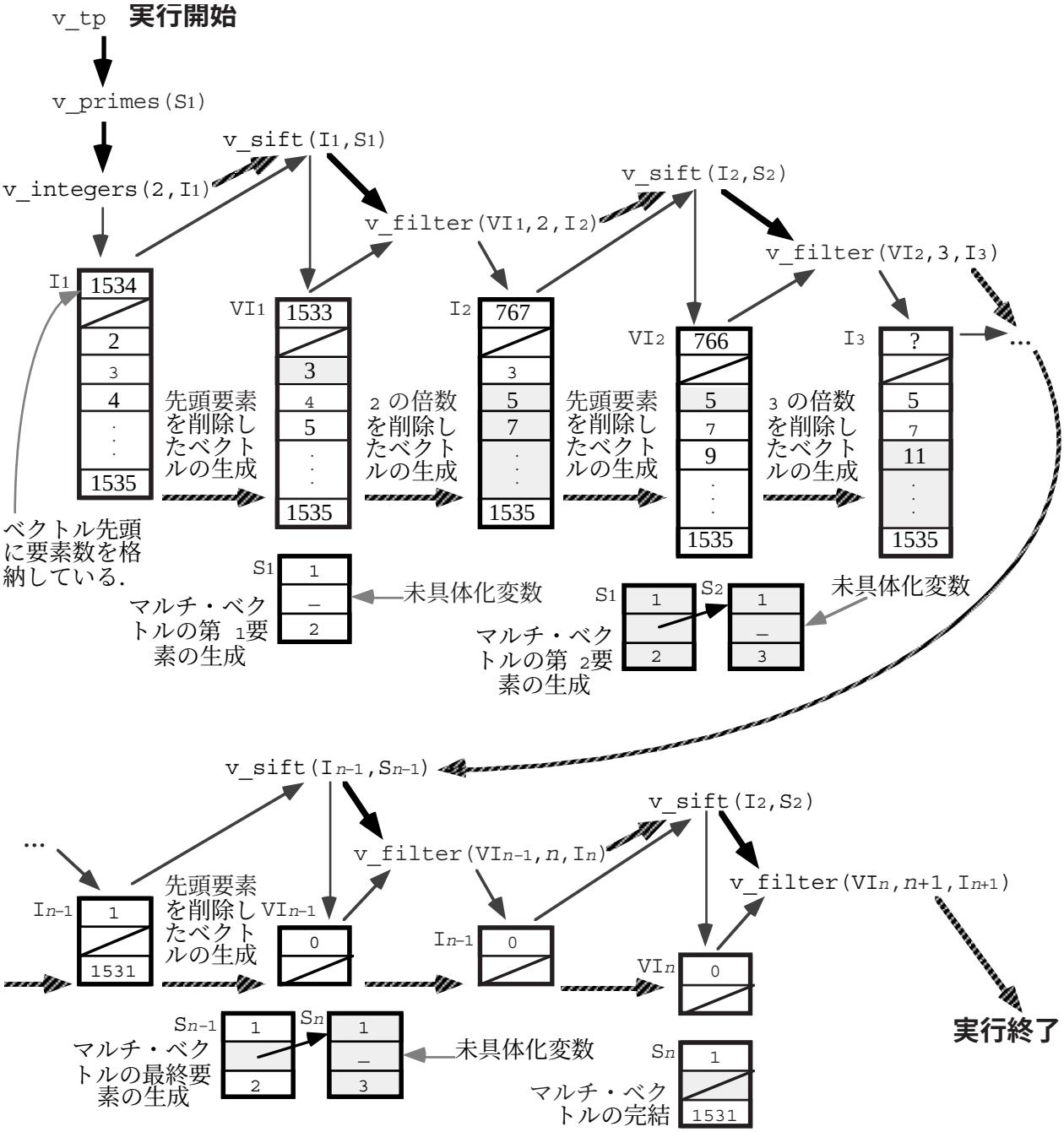


図 7.3　手動ベクトル化後の素数生成の Prolog プログラムの実行

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．

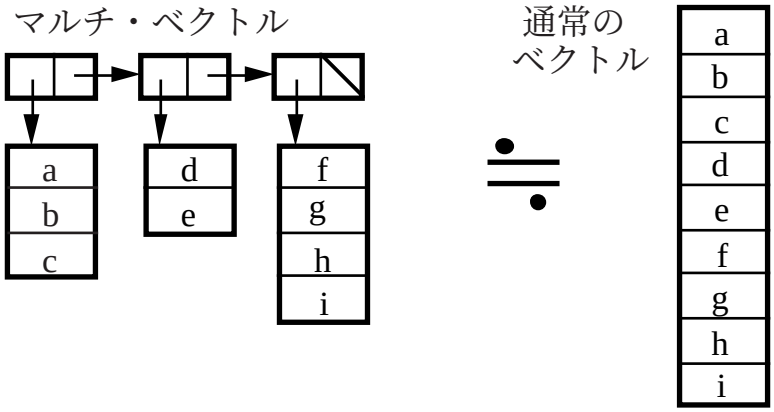


図 7.4　**マルチ・ベクトルおよびそれと等価なベクトル**

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．