# 第 6 章
# 論理プログラムへの応用 — 1
# OR 並列性をふくむプログラムのベクトル化

## 要旨

第 2 〜 4 章でしめした解探索のベクトル化技法および制御構造変換技法の応用として，ベクトル計算機 (スーパコンピュータ) を使用して論理型言語プログラムを並列処理するための一種のプログラム変換法 (ベクトル化法) を開発した．この方法を OR ベクトル化法とよぶ．OR ベクトル化法は，OR 並列性があり，引数の入出力モードが確定しているプログラムを対象とする．この方法では，制御構造変換技法をもとにして，原始プログラムの論理変数ごとに，それが探索木上のことなる位置でとる複数の値を各要素とするベクトルをつくり，それをベクトル処理するプログラムを生成する．

この方法を $N$ クウィーン問題のプログラムに適用して手動変換と自動変換の両方をこころみ，生成されたプログラムがただしく動作することを確認するとともに，ベクトル計算機 S-810 において手動で 4.5 MLIPS，自動で 2.6 MLIPS という高い実行速度をえた．この章では抽象的なレベルでこの変換法を説明するとともに結果をしめす．自動変換のために必要な処理手順や中間語などについては次章でのべる．

また，この章では論理型言語の OR ベクトル化の範囲を引数の入出力モードが確定していないばあいにまでひろげるための方針についてもふれる．

# 6.1 はじめに

Prolog で代表される論理型言語は，ユニフィケーションと自動バックトラックという 2 つの機能をそなえている．ユニフィケーションは一種のパタン・マッチングであり，それによってリスト処理などのプログラムを容易に記述することを可能にしている．また，自動バックトラックは，解探索のプログラムを非常に容易に記述することを可能にしている．これらの機能によって論理型言語は，リスト処理，言語処理，知識ベース処理などの記号処理に非常に適したプログラミング言語となっている．したがって，今後，論理型言語の応用がひろがっていくとかんがえられる．

論理型言語の応用がひろがれば，その高速処理の必要性も増加してくる．逐次計算機上での高速化をめざす研究 [Komatsu 86, Yamaguchi 87]も重要だが，飛躍的な高速化のためには並列処理が不可欠である．論理型言語プログラムの並列処理法はつぎの 2 つに大分類される．

(1) OR **並列処理**

複数の解 (部分解) を並列に計算する．

(2) AND **並列処理**

1 つの解をもとめる計算の部分計算どうしを並列に計算する．

論理型言語プログラムの高速処理の方法は，また，使用するハードウェアによって，MIMD 型並列計算機を使用する方法と，S-820，Cray-2 などのベクトル計算機 (スーパコンピュータ) や SIMD 型並列計算機を使用する方法とにわけることができる．以下，ベクトル計算機と SIMD 型並列計算機とをあわせて SIMD 型計算機とよぶ．MIMD 型並列計算機による高速処理をめざした研究としては ICOT の PIM (Parallel Inference Machines) に関する研究 [Goto 86] などがあり，SIMD 型計算機による高速処理をめざした研究としては Nilsson ら [Nilsson 88a, Nilsson 88b]，辰口ら [Tatsuguchi 87] などの研究がある．

上記の MIMD 型並列計算機や Nilsson らによる並列処理法は，柔軟性がたかいという利点がある．しかし，前者は実行時に負荷分散などのためのオーバヘッドがおおきい．また，後者は SIMD 型計算機の特性をうまくいかせず，むだがおおい．すなわち，SIMD 型計算機は複数のデータに同種の演算を適用することすなわちベクトル演算を得意としているが，Nilsson らの実行方式では異種の演算をまとめて処理しようとしている．SIMD 型計算機の特性をいかすには，極力，ベクトル演算が適用できるようにコンパイル時にプログラムを変換しておくのがよいとかんがえられる．

ベクトル計算機の Fortran コンパイラは，ベクトル化とよばれる一種のプログラム変換によって，ベクトル計算機による実行を可能にしている．すなわち，DO ループ内の

配列計算をベクトル演算に変換し，ベクトル演算命令を生成している．しかし，論理型言語プログラムには DO ループもないし配列もあらわれない．そのかわりに再帰よびだしやリストがあらわれる．したがって，Fortran コンパイラにおけるベクトル化技法は論理型言語プログラムには適用できない．このようなばあいに適用できるベクトル化技法の基本は第 3 章でしめしたとおりである．

この章では，第 3 章のベクトル化技法の応用として，コンパイル時のプログラム変換にもとづいてベクトル計算機における論理型言語の実行を可能にするための方法をしめす．6.2 節ではそのプログラム変換の方針をしめす．6.3 〜 6.5 節では，その方針にもとづいて論理型言語の決定的な手続き (述語) および非決定的な手続きを変換する方法をしめす．とくに，6.5 節では解探索における記憶消費量のくみあわせ論的爆発をふせぐことができる並列バックトラック計算法を実現するための技法をしめす．6.6 節では，6.3 〜 6.4 節の方法を $N$ クウィーン問題のプログラムに適用して，ベクトル計算機 S-810 において実測した結果をしめす．6.7 節では， OR ベクトル化の範囲を引数の入出力モードが確定していないばあいにまでひろげるための方針についてふれる．6.8 節では関連研究についてのべる．

## 6.2 OR ベクトル計算法にもとづくベクトル処理法

この章でしめす論理型言語プログラムのベクトル処理法は，第 2 章でのべた OR ベクトル計算法にもとづいている．このベクトル処理法においては，論理型言語プログラムをつぎのような手順で翻訳・実行する．

(1) **ベクトル化**

論理型言語で記述されたプログラムを，OR ベクトル計算法によって計算するプログラムに変換する．このプログラム変換を OR **ベクトル化**とよぶ[注1]．ベクトル化は，第 3 章でのべたくりかえし構造の交換，くりかえし構造の一重化という 2 つの変換戦略にもとづいている．変換結果のプログラムは，ベクトルがあつかえる言語で表現する必要がある．この章では，Prolog にくみこみ手続き (くみこみ述語) を追加してベクトルがあつかえるようにした論理型言語を使用する．この言語を LIL (Logical Intermediate Language) とよぶ．LIL は論理型言語の一種ではあるが，6.3 ～ 6.4 節においては LIL のすべての手続きは決定的であり，大域的なバックトラックはおこらない．すなわち，OR ベクトル化においては，6.8 節で関連研究としてしめす上田，Codish ら，佐藤らによる非決定的な手続きの決定的な手続きへの変換におけるように，ベクトル化前のプログラムにおいては OR 関係にあった計算が AND 関係に変換される．

(2) **実行**

OR ベクトル計算法にもとづいて実行する．ただし， (1) で生成されるのが LIL のような高水準言語のプログラムのばあいには，コンパイルして機械語におとしてから実行する．

この節では OR ベクトル計算法についてかんたんに説明するとともに，OR ベクトル計算法による計算過程と論理型言語プログラムとのおよその対応づけをしめす．詳細な対応は 6.3 ～ 6.4 節でしめす．

OR ベクトル計算法は，第 2 章でのべたように，エイト・クウィーン問題で代表される解探索問題において，複数の解候補を要素とするベクトルをつくり，それに対する計算をベクトル処理によっておこなう計算法である．4 クウィーン問題の全解探索を例として，OR ベクトル計算法を説明する．OR ベクトル計算法では，全解候補を 1 つのベクトルに蓄積して，その各要素に対して並列処理をおこなう．4 クウィーンのばあいには，各解候補はクウィーンがのせられたチェス盤である．

---

[注1] これに対して AND ベクトル計算法へのプログラムの変換を AND **ベクトル化**とよぶ．AND ベクトル化に関しては第 7 章でのべる．

**図** 6.1 に，論理型言語による 4 クウィーンのプログラムをしめす[注2]．OR ベクトル計算法による 4 クウィーンの実行過程で生成されるベクトルとその内容を**図** 6.2 にしめす．各ベクトルはリストを要素としてふくんでいる．図 6.2 には，リストを Prolog の記法とチェス盤のイメージの両方でしめしている．また，図 6.1 のプログラムと対応づけるために，実行される手続き名を記述している．ただし，図 6.1 のプログラムを OR ベクトル計算法で実行するためには，プログラム変換をへる必要があるから，これは正確な対応づけとはいえない．

```
?- put([1,2,3,4], [], Q).                          — 4 クウィーン問題をとく.

put([], B, B).
put(Qs, B, Q) :-
     select(Qs, Q1, R), not_take(B, Q1), put(R, [Q1|B], Q).
             — put が再帰よびだしされるたびに 1 個のクウィーンが
             — チェス・ボードにおかれる.

select([A|L], A, L).
select([A|L], X, [A|L1]) :- select(L, X, L1).
             — select は第 1 引数のリストから 1 個の要素を選択して第 2 引数とし,
             — のこりのリスト要素からなるリストを第 3 引数とする手続き.

not_take(R, Q) :-
     Qa is Q + 1, Qs is Q - 1, not_take1(R, Qa, Qs).
             — not_take は，えらんだクウィーンがすでにチェス・ボードに
             — おいたクウィーンによってとられないかどうかをしらべる手続き.

not_take1([], Qa, Qs).
not_take1([Q|R], Qa, Qs) :-
     Q =\= Qa, Q =\= Qs,
     Qaa is Qa + 1, Qss is Qs - 1,
     not_take1(R, Qaa, Qss).     — not_take1 は not_take のしたうけの手続き.
```

**図 6.1　4 クウィーン問題の Prolog プログラム**

[注2] 中島 [Nakashima 83] のエイト・クウィーンのプログラムを一部かきかえたものである．

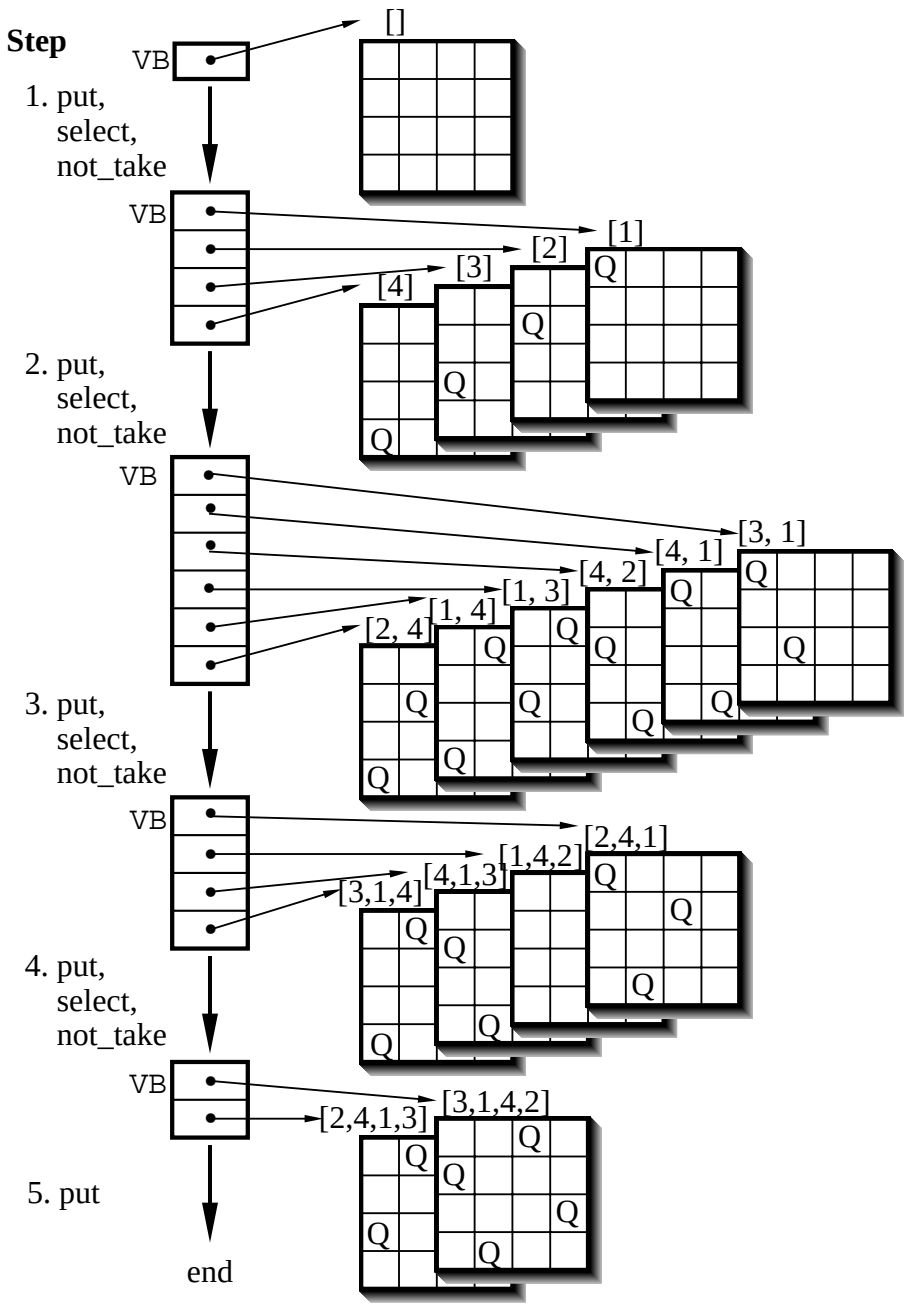



**図 6.2　OR ベクトル計算法による 4 クウィーンの実行過程**

手続き put は再帰よびだしをふくめて 5 回くりかえしてよびだされる (ステップ 0 から 4 まで)．put を構成する 2 個の節のうち，つねに 1 回には 1 個だけが実行される．最初の 4 回では第 2 節が実行され，手続き put からさらに手続き select と not_take がよびだされる．最後のよびだしでは第 1 節が実行される．

たとえばステップ 1 においては，つぎのような 4 個の同一の手続きのよびだしを並列処理するのと等価な処理がベクトル処理によっておこなわれる．

```
?- put([2,3,4], [1], Q1).
?- put([1,3,4], [2], Q2).
?- put([1,2,4], [3], Q3).
?- put([1,2,3], [4], Q4).
```

OR ベクトル計算法においては，このように，ことなる解候補に関する同一の手続きの実行を複数回まとめてベクトル処理する．論理変数や引数ごとに，それらがとりうる複数の値を各要素とするベクトルをつくり，それをベクトル処理する．図 6.2 にしめしたデータは，手続き put の第 2 引数に対応するベクトルとその内容である．他の引数や変数に対応するベクトルの値は図 6.2 にはしめしていない．

手続きが決定的か非決定的かによってベクトル化の方法およびベクトル化後の手続きのインタフェースはことなる．それらのインタフェースについて順にのべる．

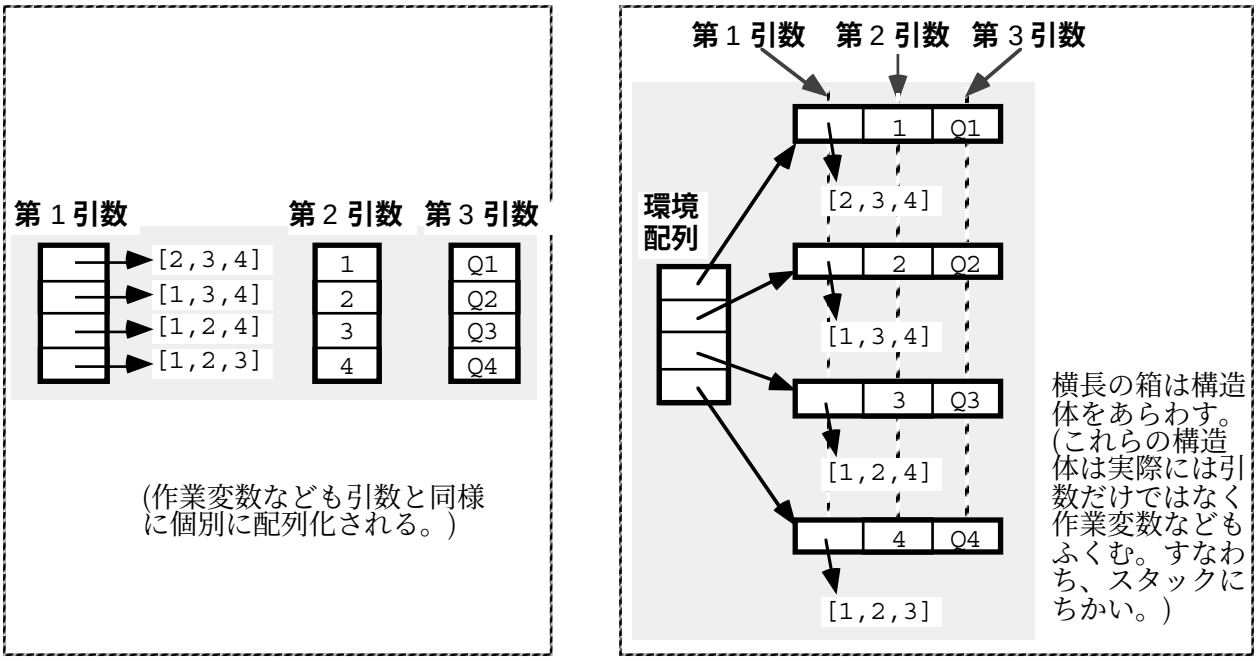
まず，手続き put は実際には非決定的だが，仮にそれが決定的だとしたばあいのそのベクトル化後の入出力インタフェースを**図** 6.3 (a) にしめす．ベクトル化後の手続きにわたされる第 1 引数はベクトル化前のそれにわたされる第 1 引数を配列化したものであり，第 2 引数，第 3 引数に関しても同様である[注3]．以上のほかに，図にはしめしていないが，引数である各ベクトルの要素が有効であるかどうかをしめす 2 個のマスク・ベクトルすなわち論理値をふくむベクトルが引数として入出力される．入力マスク・ベクトルは入力時，出力マスク・ベクトルは出力時の引数の有効性をしめす．実行が失敗するばあいがある手続きにおいては，出力マスク・ベクトルのほうが *true* である要素の個数がすくない．マスク・ベクトルに関してはあとでくわしく説明する．

つぎに，実際の手続き put がそうであるように，非決定的な手続きのばあいのベクトル化後の入出力インタフェースについて説明する．このばあいも第 1 引数，第 2 引数は図 6.3 と同一だが，第 3 引数のベクトルは手続きの実行前にはその要素数がわからないため，それは手続き内で記憶わりあてするほかはない．マスク・ベクトルを入力する点は決定的な手続きのばあいとおなじだが，入力ベクトルと出力ベクトルとで要素数がことなるために，要素の対応づけのための引数が必要なばあいがある．すなわち，その手続きの実行前に値が定義された論理変数がその手続きの実行後に参照されるばあいには，その値をふくむベクトルを “対応づけ引数” として入出力する．入力する対応づけ引数のベクトル要素数は他の入力ベクトルの要素数とひとしく，出力する対応づけ引数のベクトル要素数は他の出力ベクトルの要素数とひとしい．対応づけ引数に関してもあとでよりくわしく説明する．

以上でしめした方法のように引数ごとにデータを配列化する方法を**個別配列方式**とよぶ．個別配列方式においては，**部分解ベクトル**すなわち解の候補からなるベクトルを使

[注3] ただし，第 3 引数に関しては手続き内でベクトルを記憶わりあてする方法もかんがえられる．

用する．6.3 節で例をしめすが，通常，個別配列方式においては複数の部分解ベクトルを使用し，その要素数はひとしく，同一添字をもつ要素が対応する．図 6.3 (b) には，個別配列方式の代案である**環境配列方式**による入出力インタフェースをしめす．われわれは実現の容易さをおもな理由として個別配列方式を採用し，環境配列方式に関してはふかい検討をおこなっていない．なお，図 6.2 のデータの処理にはリストを要素とするベクトルの処理が必要になるが，その方法については第 3 章でのべたとおりである．




(a) **個別配列方式 (採用案)**　　(b) **環境配列方式 (代案)**

**図 6.3　手続き `put` の入出力インタフェース**

第 2 章でのべたように，OR ベクトル計算法はさらに完全 OR ベクトル計算法と並列バックトラック計算法とにわけることができる．上記の説明は完全 OR ベクトル計算法にしたがっている．並列バックトラック計算法の実現のためには完全 OR ベクトル計算法が基礎となるので，まず 6.3 〜 6.4 節において完全 OR ベクトル計算法の実現法をしめす．そして， 6.5 節において並列バックトラック計算法を実現する方法をしめす．

# 6.3 決定的な手続きのベクトル化法

論理型言語の手続きは決定的な手続きと非決定的な手続きとに分類することができる．この節では決定的な手続きを定義し，そのベクトル化法についてのべる．

## 6.3.1 決定的な手続きの定義

つぎのような $m$ 個の節から構成される手続き P のベクトル化についてかんがえる．

$$P_1 \text{ :- } S_{11}, \ldots, S_{1s1}, U_{11}, \ldots, U_{1u1}.$$
$$\ldots$$
$$P_m \text{ :- } S_{m1}, \ldots, S_{msm}, U_{m1}, \ldots, U_{mum}.$$

ここで，各 $S_{ij}$ はただ 1 つしか解をもたないくみこみ手続きのよびだしであり，各 $U_{i1}$ は第 $i$ 節の最初のユーザ定義手続きのよびだしだとする．また，各 $U_{ij}$ $(j > 1)$ はくみこみ手続き，ユーザ定義手続きのうちのいずれかのよびだしである．

手続き P のすべてのよびだしに対してつぎの条件がともになりたつとき，手続き P を決定的な手続きとよぶ.

(1) そのよびだしにおいて，頭部 $P_i$ とのユニフィケーションが成功し，かつ $S_{i1}, \ldots, S_{isi}$ のすべてが成功するような節が 1 個または 0 個しかない.

(2) $U_{11}, \ldots, U_{1u1}, \ldots, U_{mum}$ は決定的な手続きのよびだしまたは手続き P の再帰よびだしである．

決定的な手続き以外の手続きを非決定的な手続きとよぶ.

たとえば $N$ クウィーン問題のプログラムにおける手続き `not_take` および `not_take1` は決定的な手続きである．なぜなら，第 1 に，$N$ クウィーンのプログラムのなかでは，これらの手続きには第 1 引数に基底項 (ground term) いいかえれば定数がわたされるため，いずれか一方の頭部としかユニフィケーションが成功しないし，第 2 にこれらの節の本体には `not_take1` 以外の手続きよびだしは存在しないからである．

この節では決定的な手続きだけをあつかう．

## 6.3.2 手続きよびだしのベクトル化

6.2 節でのべたように，決定的な手続きをベクトル化してえられるプログラムにおいては，もとの手続きにおける複数のよびだしをまとめてベクトル処理する．そこで，$N$ クウィーンのプログラムを構成する手続き `not_take1` をベクトル化した手続きを `v_not_take1` とし，つぎの 2 個の手続きよびだしに対応する `v_not_take1` のよびだし

のベクトル化をかんがえる (**図** 6.4 (a) 参照).

```
?- not_take1([2,4,1], 4, 2).                    …… (6.3.2.1)
?- not_take1([4,1,3], 3, 1).                    …… (6.3.2.2)
```

これらの手続きよびだしを実行すると，(6.3.2.1) は失敗し，(6.3.2.2) は成功する．

以下，$e_1$, … , $e_n$ を要素とするベクトルを #($e_1$, … , $e_n$) とあらわす．よびだし (6.3.2.1) および (6.3.2.2) をベクトル化した結果は，LIL でつぎのように表現することができる (図 6.4 (b) 参照).

```
?- v_not_take1(#([2,4,1], [4,1,3]), #(4,3), #(2,1),
                #(true,true), MO).                  …… (6.3.2.3)
```

この変換は，not_take1 がよびだされるときに，そのよびだしの外側にバックトラックによるくりかえし構造があることを仮定して，そのくりかえし構造に関するいわゆるループ分散 (3.2 節参照) に相当する変換をおこなっていることになる．すなわち，変換前のプログラムのバックトラックによるおおきなくりかえし構造が，(ベクトル命令を単位とする) こまかいくりかえし構造に分割される．Fortran プログラムのベクトル化においてはユーザによって明に記述された DO ループがあってはじめてループ分散が可能になるが，非決定的な手続きの OR ベクトル化においてはベクトル化対象となる手続き内にあらかじめまったくくりかえしがないばあいでもベクトル化の対象となりうる点が特徴的である．なお not_take1 のばあいには，上記の変換をおこなうためには，第 3 章でのべたようにくりかえし構造の交換が必要になる．

(6.3.2.3) における第 1 〜 3 引数は，それぞれ (6.3.2.1) 〜 (6.3.2.2) における第 1 〜 3 引数とひとしい値を要素とするベクトルである．OR ベクトル化においては，変換前の複数の手続きよびだし (この例では (6.3.2.1) と (6.3.2.2)) はひろい意味での OR 関係にある．それに対して変換後はそれらがひろい意味での AND 関係になる．すなわち，変換前のプログラムにおいてはひとつの解をもとめるにはいずれか一方だけを計算すればよいが，変換後のプログラムにおいては全体を計算する必要がある．

第 4 引数は v_not_take1 の実行開始時の第 1 〜 3 引数の有効性をしめす論理型ベクトルである．また，第 5 引数は実行終了時の第 1 〜 3 引数の有効性をしめす論理型ベクトルである．このような, ベクトルの各要素の有効性をしめす論理型ベクトルをマスク・ベクトルという (第 3 章参照)．実行の結果，(6.3.2.1) が失敗するのと対応して，各ベクトルの第 1 要素は無効になる．また，(6.3.2.2) が成功するのと対応して，各ベクトルの第 2 要素は有効なままである．したがって，MO の値はつぎのようになる．

```
MO = #(false,true).
```

このようにベクトル要素の有効性をしめすのにマスク・ベクトルをつかうベクトル処理方式をマスク演算方式とよぶ.

例をつかって説明してきたが，他の手続きでもベクトル化の方法はかわらない．ただし，つねに成功する手続きのばあいは，追加する引数は 1 個でよく，その手続きのよびだしの前後の手続きよびだしで同一のマスク・ベクトルを使用すればよい．not_take1 のように失敗することがある手続きのばあいは，2 個の引数を追加して，そのよびだしのまえではそのうちの第 1 の引数をマスク・ベクトルとして使用し，よびだしのあとでは第 2 の引数をマスク・ベクトルとして使用する.

引数の有効性をあらわす方法としては，ほかに，有効な要素のインデクスをふくむベクトルを使用する方法がある．このベクトルはインデクス・ベクトルとよばれる (第 3 章参照)．インデクス・ベクトルをつかって上記の手続きよびだしを LIL で表現すると，つぎのようになる (図 6.4 (c) 参照).

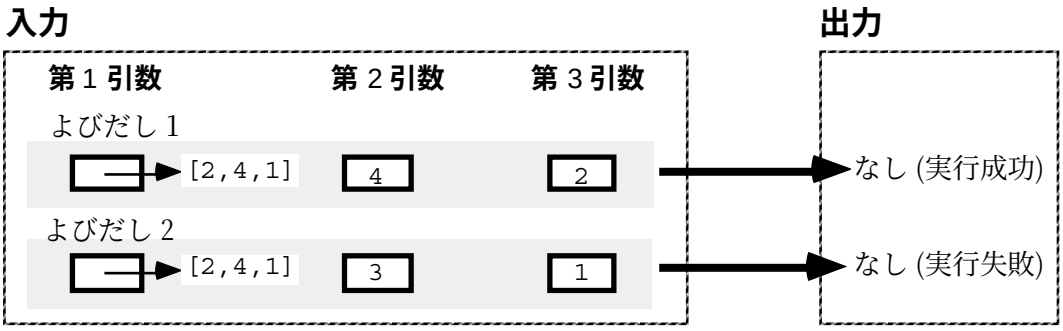
```
?- v_not_take1(#([2,4,1], [4,1,3]), #(4,3), #(2,1),
                #(1,2), XO).                                  …… (6.3.2.4)
```

第 4 引数は実行開始時のインデクス・ベクトルであり，第 1 〜 4 引数である各ベクトルの第 1，第 2 要素がともに有効であることをあらわしている．第 5 引数は実行終了時のインデクス・ベクトルであり，つぎのような 1 要素のベクトルに束縛される.

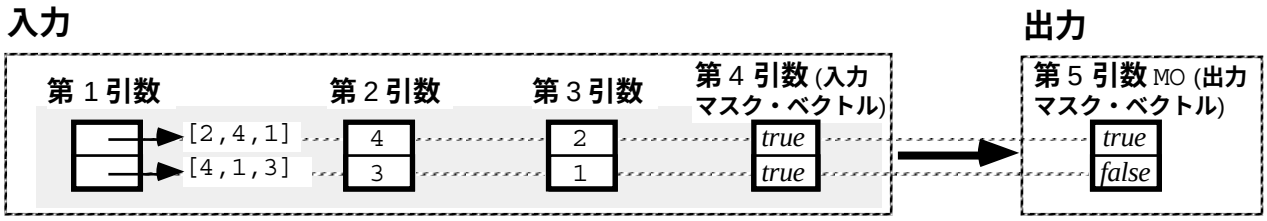
```
XO = #(2).
```

このインデクス・ベクトルは，各ベクトルの有効な要素が第 2 要素だけであることをしめしている．インデクス・ベクトルをつかう方式をインデクス方式とよぶ.

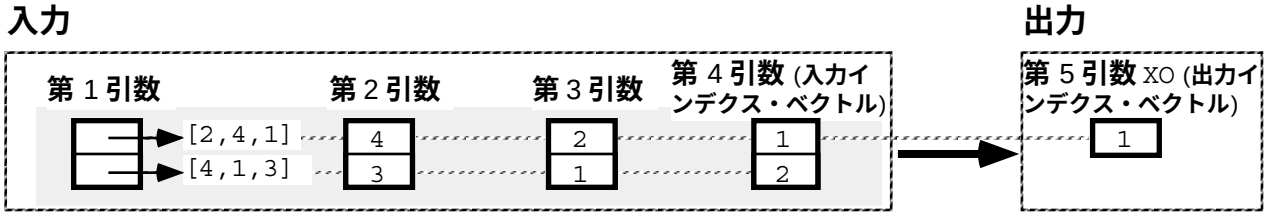
また，有効な要素だけを入出力する方式を圧縮方式とよぶ．圧縮方式では 6.4.1 節でのべるようなベクトル要素の対応づけが必要になる (他の方式と単純に比較できないため，図 6.4 には図示していない).




(a) **変換前の `not_take1` の入出力データ**




(b) **マスク演算方式による変換後の `v_not_take1` の入出力データ**




(c) **インデクス方式による変換後の `v_not_take1` の入出力データ**

**図 6.4　ベクトル化前とベクトル化後の手続き `not_take1` の入力データ**

### 6.3.3 手続き定義のベクトル化

この節では，前節でしめしたようなベクトル化された手続きよびだしでよびだされるべき手続き定義のベクトル化法をしめす．

この節でのべるベクトル化手順を適用するためには，対象となる手続き P はつぎの 2 つの条件をともにみたさなければならない．

**条件 1**　P は決定的な手続きである．

**条件 2**　P の定義は `'!'` (カット), `'\+'` (not) などの論理性をくずす手続きよびだしや，`';'` (すなわち `'or'`) をふくまない．

これらの条件のうち**条件 1** がなりたたない手続きの一部は 6.4 節でのべる方法でベクトル化することができる．**条件 2** については，手続き `';'` は手順をかんたんにするためにのぞいたのであり，これをふくんでよいように手順を拡張することは可能である．

'!', '\+' などについては，適当な仮定をおけば拡張可能だとかんがえられるが，十分に検討していない (これらのうちカット '!' については付録でかんたんにのべる).

決定的な手続きのベクトル化は，およそつぎのような手順でおこなうことができる．

**ステップ 1　決定的かどうかの判定**

ベクトル化すべき手続きが決定的な手続きかどうかを，6.3.1 節でしめした定義にしたがって判定する．決定的な手続きと判定されたときだけ，以下のステップを実行することができる．

**ステップ 2　標準化**

ベクトル化を容易にするため，プログラムを標準形に変換する．標準形については後述する．

**ステップ 3　本変換 (ベクトル化)**

各手続き名をベクトル化された手続きの名称で置換するとともに，マスク・ベクトル，インデクス・ベクトルなどの引数を追加する．

これらのステップについて，さらにくわしく説明する．

まず，決定的かどうかの判定についてのべる．決定的かどうかの判定の結果はしばしば引数のモード，すなわち手続きの引数が入力であるか出力であるかに依存する．したがって，引数のモードをしらべる必要がある．この解析をモード解析という．自動モード解析は上田 [Ueda 86] などでのべられているのと同様の方法でおこなえばよい．しかし，自動解析だけでは十分でないばあいがあるので，ユーザ・オプションというかたちでプログラマから情報をうけとる手段をもうける必要があるとかんがえられる．たとえば，つぎのようなモード宣言を手続きの直前に記述する方法が，自然でのぞましいとかんがえられる．

```
mode not_take1(+, +, +).
```
…… (6.3.3.1)

このモード宣言は，not_take の引数がすべて入力モードであることをしめしている．

つぎに，標準化について説明する．標準化によって，ベクトル化対象の手続きを構成する各節はつぎのような形式の標準形に変換される．

$$\mathtt{p}(X_1, \ldots, X_n) \ \mathtt{:-}\ I_1, \ldots, I_k, Q_1, \ldots, O_l, O_{l+1}, \ldots, O_m . \quad \cdots\cdots \text{(6.3.3.2)}$$

ここで $X_1, \ldots, X_n$ はすべてことなる論理変数である．また，$I_1, \ldots, I_k$ および $O_{l+1}, \ldots, Q_m$ はこれらの変数と変換前の節の仮引数とのユニフィケーションをおこなう手続きよびだしである．$O_1, \ldots, O_l$ は変換前の節の本体の手続きよびだしである．効率上は，

出力モードの仮引数とのユニフィケーションは変換前の本体よりあとにおき，それ以外の仮引数とのユニフィケーションは変換前の本体よりまえにおくのがのぞましい．しかし，モードが不明のときはそれらすべてを変換前の本体のまえにおけばよい．

本体に関しても，つぎのような変換をおこなう．手続き is のよびだしを手続き '+', '-' などの算術演算をおこなう手続きよびだしの列に置換する．

例を 2 つあげる．

**例 1　手続き not_take1 の標準形**

各節の標準形はつぎのとおりである．

```
not_take1(T1, Qa, Qs) :- T1 = [].                    …… (6.3.3.3)
not_take1(T1, Qa, Qs) :-
    T1 = [Q | R], Q =\= Qa, Q =\= Qs,
    '+'(Qa, 1, Qaa), '-'(Qs, 1, Qss),
    not_take1(R, Qaa, Qss).                          …… (6.3.3.4)
```

各節の第 1 仮引数は論理変数ではなかったため，T1 で置換されている．not_take1 の引数はすべて入力モードであるから，これらの引数に対するユニフィケーションは節の先頭におかれている．■

**例 2　手続き append の標準形**

リストを連接する手続き append のモード宣言つき原始プログラムをしめす．

```
mode append(+, +, -).                                …… (6.3.3.5)
append([], X, X).                                    …… (6.3.3.6)
append([A | D], Y, [A | D1]) :- append(D, Y, D1).    …… (6.3.3.7)
```

標準形はつぎのとおりである．

```
append(T1, X, T2) :- T1 = [], T2 = X.                …… (6.3.3.8)
append(T1, Y, T2) :-
    T1 = [A | D], append(D, Y, D1), T2 = [A | D1].   …… (6.3.3.9)
```

標準形第 1 節の第 3 引数は原始プログラムにおいても変数であるにもかかわらず置換されているが，これは，同一の変数が引数として 2 回あらわれているからである．append の第 1 引数は入力モードであるから，T1 に対するユニフィケーションは各節の先頭でおこなっている．また，append のの第 3 引数は出力モードであるから，T2 に対するユニフィケーションは各節の末尾でおこなっている．■

最後に，本変換について説明する．かんたんにするため，2 つの節から構成される手続きのマスク演算方式への変換にかぎってのべる．1 個または 3 個以上の節から構成される手続きについては後述する．インデクス方式への変換についても後述する．

変換すべき手続き P の標準形がつぎのとおりだとする．

$P(X_1, \ldots, X_l)$ :- $Q_{11}, \ldots, Q_{1n_1}$. …… (6.3.3.10)

$P(X_1, \ldots, X_l)$ :- $Q_{21}, \ldots, Q_{2n_2}$. …… (6.3.3.11)

ここで，各節の対応する引数は同名の変数としている．

変換後の手続き名を VP とする．本変換によって，つぎのような LIL の手続きが出力される．

```
VP(_, ..., _, MI, MI) :- v_finished(MI), !.                    …… (6.3.3.12)
VP(X1, ..., Xl, MI, MO) :-
    Q11', ..., Q1n1', v_or1(MI, MO1, MI2),
    Q21', ..., Q2n2', v_or2(MO1, MO2, MO).                     …… (6.3.3.13)
```

手続き VP の実行中には，`v_finished` 以外の部分では深いバックトラックはおこらない．すなわち，VP からよばれる手続きからバックトラックでとびだしたり，そこへとびこんだりすることはない．

第 1 節 (6.3.3.12) は変換前の手続きには対応する部分がない．この部分は，むだな計算をはぶくとともに，VP の無限再帰をふせいでいる．手続き `v_finished` は LIL のくみこみ手続きであり，入力されたマスク・ベクトルのすべての要素の値が *false* ならば成功し，手続き VP の実行を終了させる．VP が再帰よびだしされないばあいは第 1 節がなくてもただしく動作するが，第 2 節 (6.3.3.13) の実行がむだに実行されることがありうる．VP が再帰よびだしされるばあいは，第 1 節がなければ再帰が無限にくりかえされ，手続き VP の実行は停止しない．

$Q_{11}'$, …, $Q_{1n_1}'$ を第 1 節対応部，$Q_{21}'$, …, $Q_{2n_2}'$ を第 2 節対応部とよぶ．第 1 節対応部および第 2 節対応部は，変換前の第 1 節本体および第 2 節本体における手続きの名称をベクトル化後のそれに置換するとともに，マスク・ベクトルを引数として追加したものである．$Q_{ij}$ が引数としてマスク・ベクトルを 1 個とるばあいは，$Q_{ij}$ の前後で同一のマスク・ベクトルを使用する．また，$Q_{ij}$ が引数としてマスク・ベクトルを 2 個とるばあいは，$Q_{ij}$ のまえでは第 1 のマスク・ベクトル，$Q_{ij}$ のあとでは第 2 のマスク・ベクトルを使用する．

手続きよびだし $Q_{ij}$ がベクトル化できないばあいについてのべる．このばあいは，$Q_{ij}'$ を，その引数のベクトル長の回数だけ $Q_{ij}$ をくりかえしよぶ手続きとすることによって，

他の部分をベクトル化することができる．

手続きよびだし $Q_{ij}$ がくみこみ手続きであるばあいについてのべる．原始プログラムのくみこみ手続きに対しては，それぞれ対応するくみこみ手続きを LIL に用意する．この章では，LIL のくみこみ手続きの名称は，原始プログラムのそれの名称に 'v_' を接頭したものとする．たとえば，'=\=' に対しては 'v_=\=' を用意する．ただし，現在のベクトル計算機では論理型言語のくみこみ手続きのすべての機能を効率よく実装することは困難なので，かぎられた範囲で使用できる手続きを用意するのが，効率上のぞましい．たとえば，リストの合成と分解 (Lisp でいえば cons と car, cdr) は論理型言語ではともに C=[H|T] によっておこなわれるが，モード解析の結果にしたがって，合成のばあいは v_cons，分解のばあいは v_carcdr を使用する．また，リストと空リスト [] とのユニフィケーションは，同様にモード解析の結果にしたがって，空リストかどうかの判定のばあいには v_null，空リストの代入のばあいには v_nullify を使用する．モードが不明のばあいあるいは確定しないばあいは，タグ判定をおこなってマスクをつくり，それにしたがって複数種類の手続きを実行する必要がある．すなわち，リストの合成と分解のばあいには v_cons と v_carcdr とを条件制御下でともに実行する．このようなばあいにはベクトル化による加速率が低下し，1 以下になる，すなわち逐次実行より低速になる可能性がより高くなる．

(3.3.13) における v_or1 および v_or2 はマスク・ベクトルの値を合成する LIL のくみこみ手続きであり，これらの引数はすべてマスク・ベクトルである．

v_or1(MI,MO1,MI2) は手続きの開始前のマスク・ベクトル MI と，原始プログラムの第 1 節対応部が出力する MO1 とから第 2 節実行開始時のマスク・ベクトル MI2 を合成する．MO1 は $Q_{1n_1}$' が出力するマスク・ベクトルであり，MI2 は $Q_{21}$' が入力するマスク・ベクトルである．また，v_or2(MO1,MO2,MO) は MO1 と第 2 節対応部から出力される MO2 とから，手続き VP が出力すべきマスク・ベクトルを合成する．

v_or1 および v_or2 は論理値を入力するから，論理演算をおこなう．したがって，これらの手続きの機能はつぎのようにあらわされる．

$$MI2 = MI \wedge \tilde{} MO1. \quad \cdots\cdots (6.3.3.14)$$

$$MO = MO1 \vee MO2. \quad \cdots\cdots (6.3.3.15)$$

ここで '~' は論理否定，'∧' は論理積，'∨' は論理和をあらわす．これらの手続きの意味はつぎのとおりである．決定的な手続きにおいては，第 1 節が失敗したときだけ第 2 節が実行される．したがって，v_or1 は LIL の手続きの入力マスクが *true* かつ第 1 節対応部の出力マスクが *false* のときだけ第 2 節対応部の入力マスクを *true* とする．また，第 1 節対応部，第 2 節対応部のうちのいずれかが成功すれば手続きの実行が成功したこ

とになるので，v_or2 はこれらが出力するマスクの論理和をとったものを結果とする．

例として not_take1 をとりあげる．前記の標準形に本変換をほどこした結果はつぎのようになる．

```
v_not_take1(_, _,  _,  MI, MI) :- v_finished(MI), !.  …… (6.3.3.16)
v_not_take1(B, Qa, Qs, MI, MO) :-
    v_null(B, MI, MO1), v_or1(MI, MO1, M1),
    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),
    v_not_take1(R, Qaa, Qss, M4, MO2),
    v_or2(MO1, MO2, MO).                          …… (6.3.3.17)
```

このプログラムの実行過程の例や変換後のプログラムにあらわれる各手続きの説明はすでに第 3 章でしめしたので，ここでは実行に関するこまかい説明は省略する．ただ，上記の変換でえられた手続き (上記の例では v_not_take1) は決定的であるという点を指摘しておく．すなわちこの手続きにおいてバックトラックが生じうるのは最初の節にあらわれる手続き v_finished にかぎられ，しかもこのバックトラックはここではその手続きの第 2 節への局所的なバックトラックであり，大域的なバックトラックが生じることはない．決定的な手続きの変換に関してここでは他の例はしめさないが，金田ら [Kanada 88a] では，手続き append のベクトル化過程と LIL によるプログラムをしめしている．

原始プログラムの節の個数が 2 個以外のばあいも，節の本体の変換方法は，節が 2 個のばあいと基本的にかわらない．マスク・ベクトル合成は，節が 1 個のばあいは必要がない．節が 3 個以上のばあいは，**図** 6.5 にしめすようにマスク・ベクトルを逐次的に合成する．

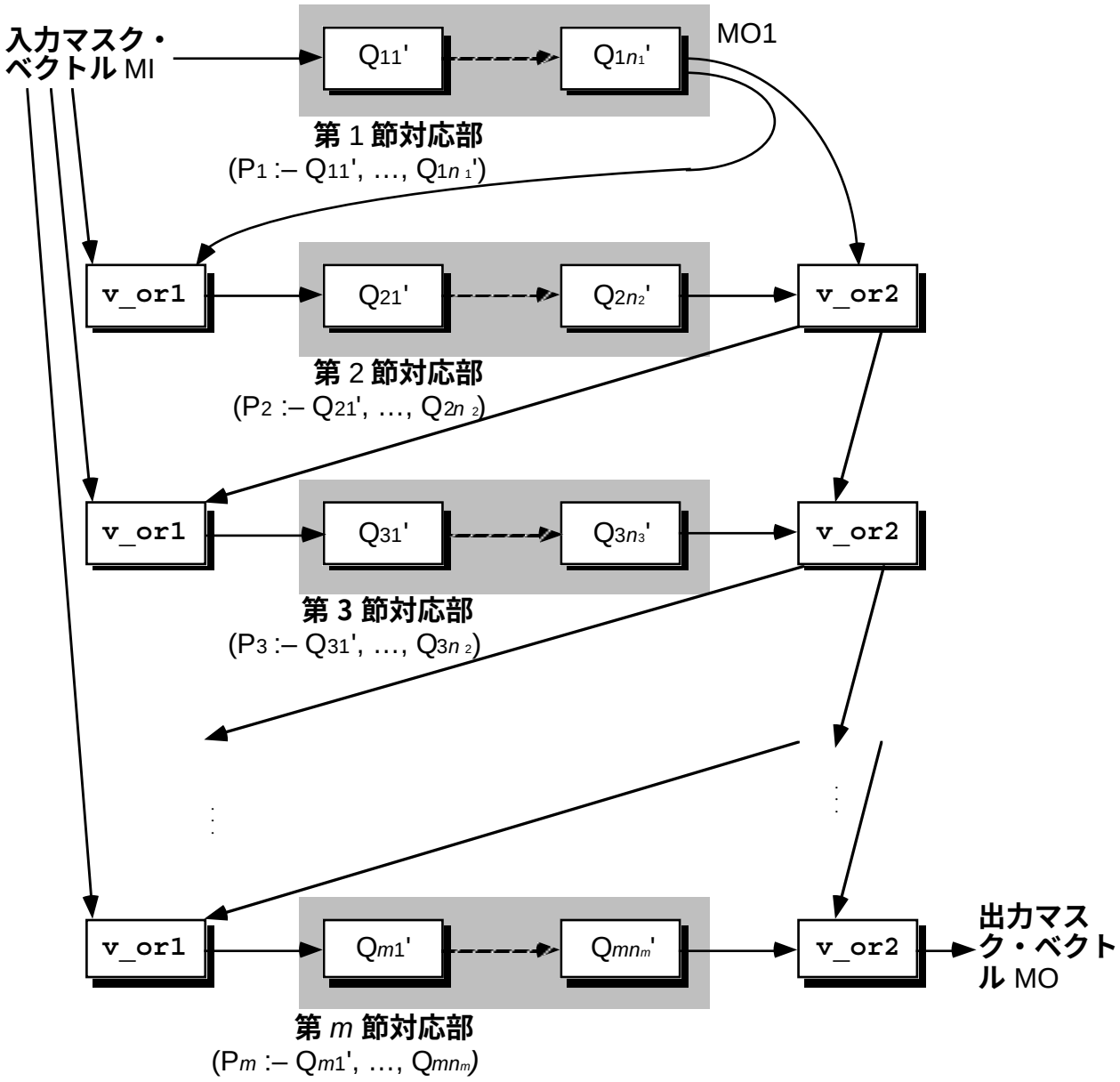



**図 6.5　マスク演算方式におけるマスク・ベクトルの合成**

最後に，マスク演算方式以外の条件制御方式への変換についてのべる．

インデクス方式のばあいは，節の本体の変換方法はほぼおなじである．ただし，マスク・ベクトルではなくインデクス・ベクトルを引数として追加し，マスク・ベクトルの合成のかわりにインデクス・ベクトルを合成する．すなわち，各節本体から出力されるインデクス・ベクトルの全要素からなるベクトルを手続きの出力インデクス・ベクトルとする．手続き `not_take_1` をインデクス方式によってベクトル化したプログラムをつぎにしめす．

```
x_not_take1(_, _,  _,  II, #()) :- x_finished(II), !.
          % II が入力インデクス・ベクトル，#() が空の出力インデクス・ベクトル.
x_not_take1(B, Qa, Qs, II, IO) :-
          % II が入力インデクス・ベクトル，IO が出力インデクス・ベクトル.
     x_null(B, II, IO, IO1),
          % インデクス・ベクトル II によってさされるベクトル B の各要素が空リストであるか
          % どうかを判定し，結果が真であるインデクスを IO1 の要素とする.
          % (以下こまかい説明は省略する.)
     x_carcdr(Q, R, B, II, I2),
     'x_=\='(Q, Qa, I2, I3),
     'x_=\='(Q, Qs, I3, I4),
     'xs_+'(Qa, 1, Qaa, I4),
     'xs_-'(Qs, 1, Qss, I4),
     x_not_take1(R, Qaa, Qss, I4, IO1).
```

手続き x_not_take1 にあらわれる II, IO, IO1 などの論理変数は，インデクス・ベクトルに束縛される. 各ベクトルくみこみ手続きにマスク・ベクトルのかわりにインデクス・ベクトルがわたされている点以外には，マスク演算方式によるプログラムと形式上おおきな差はない.

圧縮方式のばあいは，無効なベクトル要素が発生するたびにつめあわせをおこなう. しかし，命令レベルで圧縮方式をサポートする命令は現在のパイプライン型ベクトル計算機にはない．また，仮にあらたなハードウェアをつくるとしても，通常はつめあわせるべきベクトルは複数個あるから，命令レベルで圧縮方式をサポートするのは困難である．したがって，折衷的な方法をとる．すなわち，局所的には，いったんマスク・ベクトルやインデクス・ベクトルをマスク・レジスタあるいはベクトル・レジスタ上につくったのち，それをつかってベクトルの圧縮をおこなう．手続き not_take_1 を圧縮方式によってベクトル化したプログラムをつぎにしめす．このプログラムにおいては，局所的にはマスク演算方式を使用している.

```
c_not_take1(QR, Qa, Qs, B-B, R-R, Q-Q) :-
        % B, R, Q は c_not_take1 の本来の出力データとともに圧縮するべきベクトル.
    v_finished(QR), !.
c_not_take1(QR, Qa, Qs, BI-BO, RI-RO, QI-QO) :-
        % BI, RI, QI は c_not_take1 の本来の出力データとともに圧縮するべきベクトル.
        % BO, RO, QO がそれらに対応する出力ベクトル.
    v_null(QR, MI, M1),
    c_compress(BI, M1, BO-BOe),          % ベクトル BI を圧縮して BO にいれる.
    c_compress(RI, M1, RO-ROe),          % ベクトル RI を圧縮して RO にいれる.
    c_compress(QI, M1, QO-QOe),          % ベクトル QI を圧縮して QO にいれる.
    v_carcdr(Q, R, QR, M, M2),
    'v_=\='(Q, Qa, M2, M3),
    'v_=\='(Q, Qs, M3, M4),
    c_compress(R, M4, R1-#()),           % ベクトル R を圧縮して R1 にいれる.
    c_compress(Qa, M4, Qa1-#()),         % ベクトル Qa を圧縮して Qa1 にいれる.
    c_compress(Qs, M4, Qs1-#()),         % ベクトル Qs を圧縮して Qs1 にいれる.
    c_compress(BI, M4, BI1-#()),         % ベクトル BI を圧縮して BI1 にいれる.
    c_compress(RI, M4, RI1-#()),         % ベクトル RI を圧縮して RI1 にいれる.
    c_compress(QI, M4, QI1-#()),         % ベクトル QI を圧縮して QI1 にいれる.
    'vs_+'(Qa1, 1, Qaa, M4),
    'vs_-'(Qs1, 1, Qss, M4),
    c_not_take1(R1, Qaa, Qss, BI1-BOe, RI1-ROe, QI1-QOe).
        % 圧縮されたベクトルを引数として c_not_take1 を再帰よびだしする.
```

なお，M-680H IDP のばあいには圧縮方式をサポートする命令 (Sequential Search 命令) があるので，より直接的な実装が可能である．M-680H IDP における Sequential Search 命令を使用した解探索の実行法については鳥居ら [Torii 88a, Torii 88b] にかんたんに記述されている．

インデクス方式，圧縮方式による上記のプログラムの意味とその動作の詳細に関しては金田 [Kanada 87] で説明している．

# 6.4 非決定的な手続きのベクトル化法 (完全 OR ベクトル化)

この節では非決定的な手続きのベクトル化法を，再帰的な手続きと非再帰的な手続きとにわけてのべる．ただし，手続きよびだしのベクトル化法は，両者に共通である．6.3.2 節でのべたように OR ベクトル化においては広義の OR 関係が広義の AND 関係に変換されるが，とくに，この節でのべる非決定的な手続きのベクトル化において，全解探索プログラムの決定的なプログラムへの変換 [Ueda 85b, Codish 85, Tamaki 87] と類似の変換をおこなっているということができる．

## 6.4.1 手続きよびだしのベクトル化

*N* クウィーンのプログラムを構成する手続き `select` をベクトル化した手続きを `v_select` とし，つぎのような 2 個の非決定的な手続きのよびだしに対応する `v_select` のよびだしのベクトル化をかんがえる (**図** 6.6 (a) 参照).

```
?- select([2,4], X, Y).                                  …… (6.4.1.1)
?- select([1,3], X, Y).                                  …… (6.4.1.2)
```

これらの手続きよびだしを実行すると，(6.4.1.1) からは (6.4.1.3) と (6.4.1.4)，(6.4.1.2) からは (6.4.1.5) と (6.4.1.6) とが解としてえられる．

```
X = 2, Y = [4].                                          …… (6.4.1.3)
X = 4, Y = [2].                                          …… (6.4.1.4)
X = 1, Y = [3].                                          …… (6.4.1.5)
X = 3, Y = [1].                                          …… (6.4.1.6)
```

よびだし (6.4.1.1) および (6.4.1.2) をベクトル化した結果は，LIL でつぎのように表現することができる (図 6.6 (b) 参照).

```
?- v_select(#([2,4], [1,3]), VX, VY,
             #(true,true), BI, BO).                      …… (6.4.1.7)
```

決定的な手続きのばあいと同様に，`select` がよびだされるときに，この変換においてもそのよびだしの外側にバックトラックによるくりかえし構造があることを仮定している．ただし，決定的な手続きのばあいとはちがって，非決定的な手続きのベクトル化においてはその手続きの内部にもバックトラックのもどり点すなわちくりかえし構造の開始点が存在する．そのため，非決定的な手続きのベクトル化においては，第 3 章での

べたようにくりかえし構造の 1 重化が必要になる．

ここで，v_select のよびだしの第 1 引数は，select の各よびだしにおける第 1 引数とひとしい値を要素とするベクトルである．第 4 引数は入力マスク・ベクトルであり，第 1 引数の各要素の有効性をしめす．第 1 引数の 1 個の要素から複数の解が生成されうるので，第 2 〜 3 引数のベクトルの各要素は第 1 引数の各要素とは対応しない．たとえば，(6.4.1.7) をよびだした結果はつぎのようになる．

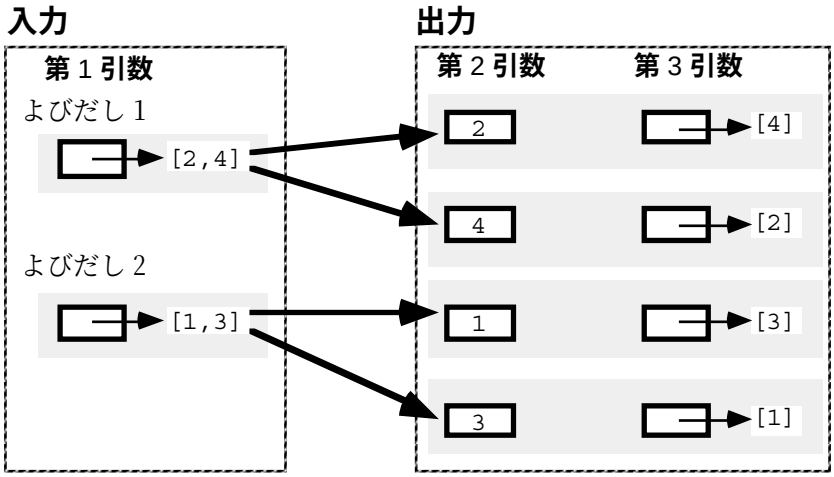
```
VX = #( 2,  4,  1,  3 ),
VY = #([4],[2],[3],[1]).                                    …… (6.4.1.8)
```

VX と VY の要素数はひとしく，それらの要素は対応している．ただし，要素の順序は上記のとおりだとはかぎらない．ところで，この例においては 2 要素の入力をあたえているが，もし 1 要素の入力をあたえたばあいには，手続き v_select は Prolog のくみこみ手続き bagof にちかい機能を実現することになる．

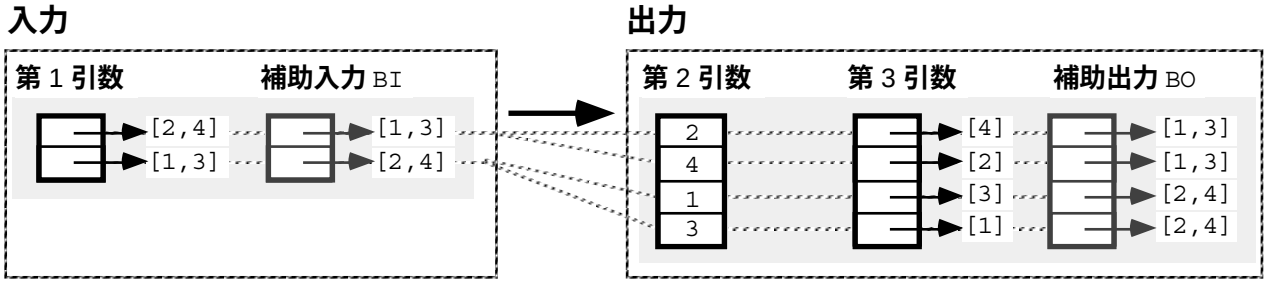
つぎに第 5 〜 6 引数 BI，BO について説明する．これらの引数は，ベクトルの要素の対応づけのためにもうけている．したがって，対応づけ引数とよぶ．図 6.1 の手続き put の第 2 節において，put に入力されたデータ Qs，B のうち手続き select のよびだし後に Qs はもはや使用されないが，B は使用される．したがって，B についてだけは X, Y との要素の対応をつける必要がある．たとえば，(6.4.1.1) においては B = [1,3]，(6.4.1.2) においては B = [2,4] だとする．

```
BI = #([1,3], [2,4]).
BO = #([1,3], [1,3], [2,4], [2,4]).
```

上記の説明からあきらかなように，対応づけ引数の個数は，非決定的な手続きのよびだし点前後のデータフローに依存する．したがって，原始プログラムの手続き定義だけがあたえられただけでは何個の対応づけ引数が必要であるかがわからない．




(a) **変換前の `select` の入出力データ**




(b) **変換後の `select` の入出力データ**

**図 6.6　ベクトル化前とベクトル化後の手続き `select` の入力データ**

### 6.4.2 非再帰的手続き定義のベクトル化

この節では，非決定的な手続き定義のベクトル化法をしめす．この節の方法は再帰よびだしをふくむ手続きのベクトル化にも適用できるが，再帰よびだしをふくむ手続きには次節でしめすより最適化された方法のほうがのぞましいとかんがえられる．したがって，この節の方法はおもに再帰よびだしをふくまない手続き定義のベクトル化に適用される．

この節のベクトル化手順を適用するためには，対象となる手続き P はつぎのすべての条件をみたさなければならない．

**条件 2**　P の定義は `'!'`, `'\+'` などの論理性をくずす手続きよびだしや，`';'` をふくまない．

**条件 3**　P のすべての引数は入力モード，出力モードのうちのいずれかである．ここで，ある引数が入力モードであるとは，その仮引数が基底項であることを意味する．また，ある引数が出力モードであるとは，その仮引数が論理変数であることを意味する．

**条件 4**　P のよびだし前で定義され，よびだし後に使用されるすべての変数の値は P のよびだし時に基底項である．

**条件 2** は 6.3.3 節におけるのと同一である．**条件 3**，**条件 4** がなりたたない手続きは，ベクトル化不能である．ただし，この節でのべる手順の拡張などによって**条件 3**，**条件 4** をゆるめることによって一部の手続きはベクトル化可能になるが，その方法はこの章ではのべない．なお，非決定的な手続きであることは，条件ではない．決定的な手続きをベクトル化するのにこの節の手順を使用すれば，6.3.3 節の手順で生成されるのより効率がわるい手続きが出力されるだけである．

非決定的な手続きのベクトル化は，およそつぎのような手順でおこなうことができる．

**ステップ 1　モード解析**

ベクトル化すべき手続きの各引数が入力モードであるかどうか，また，出力モードであるかどうかを判定する．解析の結果，すべての引数が入力モードか出力モードかに分類されたときだけ，以下のステップを実行することができる．

**ステップ 2　標準化**

ベクトル化を容易にするため，プログラムを標準形に変換する．

**ステップ 3　本変換 (ベクトル化)**

各手続き名をベクトル化された手続きの名称で置換するとともに，マスク・ベクトル，インデクス・ベクトルなどの引数を追加する．

モード解析については 6.4.3 節でのべた．また，標準化は，決定的な手続きとまったく同様におこなえばよい．

本変換について説明する．かんたんにするため，つぎのような制限をもうける．まず，2 つの節から構成される手続きのマスク演算方式への変換にかぎってのべる．また，引数のモードは第 1 〜 $k$ 引数が入力モード，第 $k$+1 〜 $l$ 引数が出力モードだとする．さらに，対応づけ引数は 1 組 (2 個) とする．これらの制限をゆるめるのは容易である．

変換すべき手続き P の標準形は (6.3.3.10) 〜 (6.3.3.11) のとおりだとする．変換後の手続き名を VP とすれば，本変換の結果，つぎのような LIL の手続きが出力される．

```
VP(_,  ..., _,  MI, _,  _) :- v_finished(MI), !.          ..... (6.4.2.1)
VP(X1, ..., Xl, MI, BI, BO) :-
    Q11', ..., Q1n1', Q21', ..., Q2n2',
    v_merge([[X1k+1,X2k+1],[X1k+2,X2k+2], ...,
             [X1l,X2l], [B1O,B2O]], [Xk+1,Xk+2,...,Xl,BO],
             [M1,M2]).                                    ..... (6.4.2.2)
```

第 1 節 (6.4.2.1) のはたらきは，決定的な手続きのばあいと同様にむだな計算をはぶき，無限再帰をふせぐことである．(6.4.2.2) におけるデータのながれを**図** 6.7 にしめす．第 1 節対応部 $Q_{11}$', … , $Q_{1\,n1}$' および第 2 節対応部 $Q_{21}$', … , $Q_{2\,n2}$' は，変換前の第 1 節，第 2 節がすべて決定的な手続きのよびだしで構成されていれば，決定的な手続きの本変換と同様に変換すればよい．非決定的な手続きのよびだし $Q_{ij}$' があらわれるときはつぎのようにする．変換前の手続き $Q_{ij}$ よりまえすなわち $Q_{11}$', … , $Q_{1\,j-1}$' で定義され，$Q_{ij}$ のあとすなわち $Q_{i\,j+1}$', … , $Q_{i\,ni}$' で使用される変数は，$Q_{ij}$' の対応づけ引数として入出力するようにする．

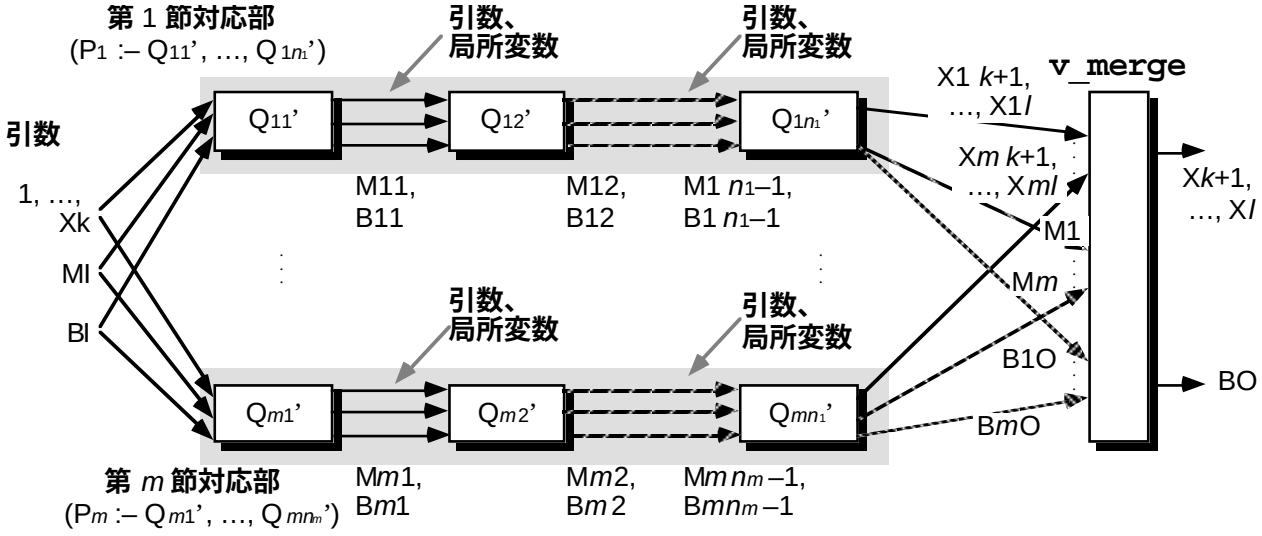



**図** 6.7　**ベクトル化後の非決定的な手続きの実行におけるデータのながれ**

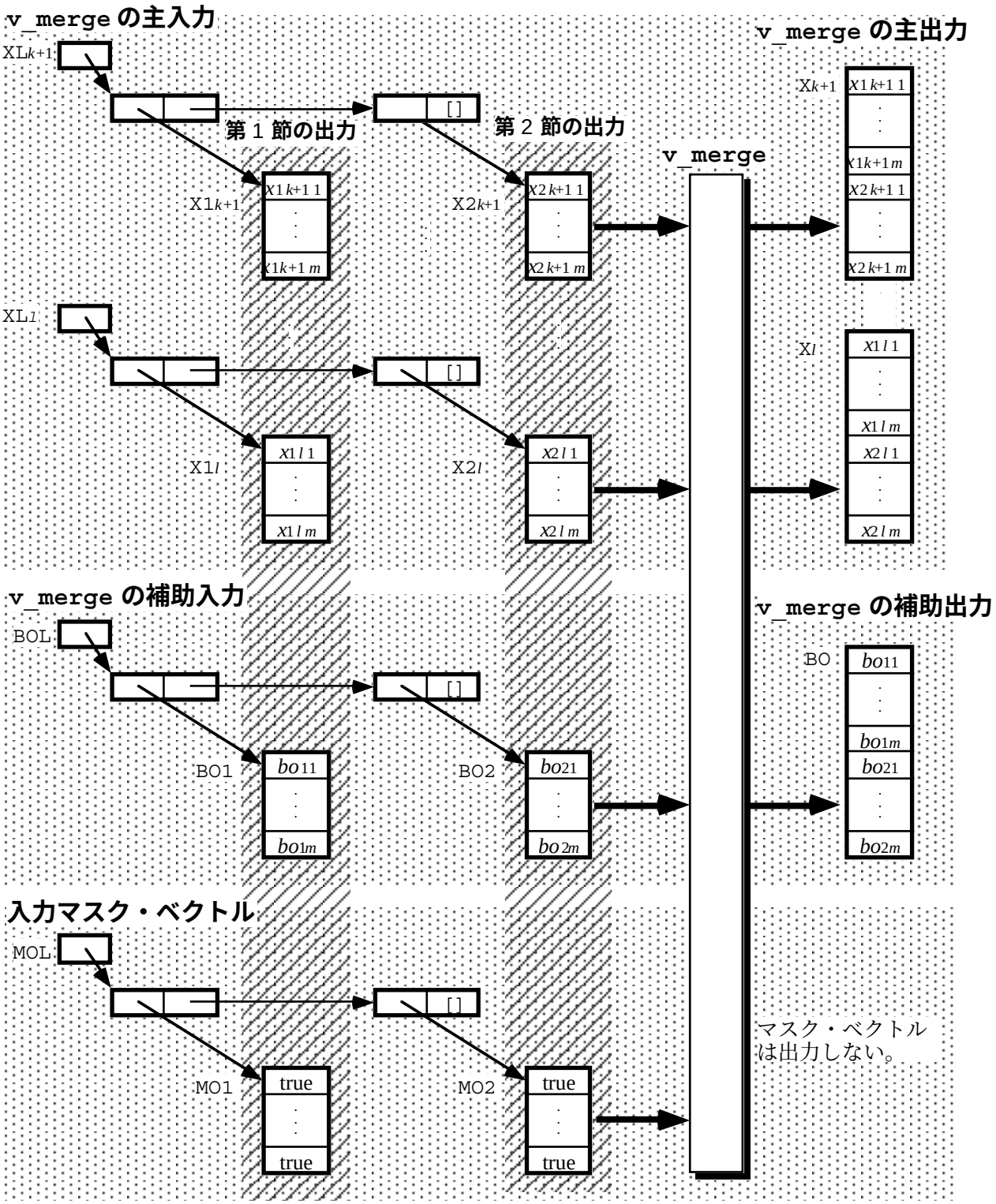



**図 6.8　ベクトルを併合する手続き v_merge の機能**

上記の説明では，変換前の各論理変数をどのように置換することによって変換後の各論理変数をえるかをしめさなかった．正確な変数の置換方法をしめすと煩雑になるので，かわりに後述の例において説明する．

LIL のくみこみ手続き v_merge の機能を，**図** 6.8 を使用して説明する．v_merge は，ベクトルを併合する．すなわち，複数のベクトルにふくまれるすべての要素を 1 つのベクトルの要素にする．ベクトル計算機においては，ベクトル長が数 10 以上にならないとパイプライン演算器の性能をいかしきれないので，v_merge によってベクトル長をのばしている．v_merge に入力する複数のベクトルはリストにしている．(6.4.2.2) でいえば，$X_{1k+1}$ と $X_{2k+1}$，B1O と B2O などをそれぞれリストにしている．このようなリストをマルチ・ベクトルとよぶ．v_merge は，ベクトルの併合を複数のマルチ・ベクトルのそれぞれについて実行する．

以下，例をしめす．手続き p のつぎのような標準形を本変換する．p の第 1 引数は入力モード，第 2 引数は出力モードだとする．

```
p(X,Y) :- q11(Y), q12(X,Y), q13(Y).                    …… (6.4.2.3)
p(X,Y) :- q2(X,Y).                                     …… (6.4.2.4)
```

手続き q12, q2 は決定的な手続きであり，手続き q11, q13 は非決定的な手続きだとする．また，p のよびだし以前に定義された 1 個の値がよびだし後に使用されるとする．ベクトル化後の手続きはつぎのようになる．

```
v_p(_,  _,  MI, _,  _) :- v_finished(MI), !.           …… (6.4.2.5)
v_p(XI, YO, MI, BI, BO) :-
    v_q11(Y11, MI, XI, X11, BI, B11),
    v_q12(X11, Y11, _, M12),
    v_q13(Y11, M12, Y11, Y12, B11, B12),
    v_q2(XI, YI, MI, M21),
    v_merge([[Y12,YI], [B12,BI]], [YO,BO], [M12,M21]).
                                                       …… (6.4.2.6)
```

v_q11，v_q13 にはそれぞれ 2 組の対応づけ引数がある．v_p の第 1 引数は入力であるため，第 1 仮引数が v_q11 のよびだしにおける対応づけ引数としてつかわれている．v_q11 のよびだし以後は，変換前の変数 X は X11 で置換されている．また，v_p の第 2 引数は出力であるため，v_merge の出力がそのまま第 2 仮引数とされている．v_q11 のよびだし以前では，変換前の変数 Y は Y11 で置換され，v_q11 のよびだしから v_q13 のよびだしのあいだでは Y12 で置換されている．'_' は全要素が *true* であるマスク・ベ

クトルをあらわしている．

v_merge は，ベクトル Y12 と YI とを併合して YO をつくり，B12 と BI とを併合して BO をつくるために使用されている．v_merge で入力引数 YI，BI をそのまま使用しているのは，変換前の第 2 節 (6.4.2.4) には決定的な手続きだけがあらわれるからである．なお，v_merge によってベクトルを併合することを基本とする非決定的な手続きの実行方法の代案を**付録**でしめす．

なお，v_q13 においては変数 Y11 が 2 つの引数でわたされているが，このむだをさけるために，インタフェースを変更して引数を 1 個へらすことも可能である．

つぎに，上記のプログラム変換の 3 つの性質についてのべる．

第 1 に，解がえられる順序は，一般には保存されない．すなわち，解をふくむベクトルにおいて，要素がならぶ順がふかさ優先順であることは保証できない．また，幅優先順であることも保証できない．ただし，特定の条件がなりたてば，解の順序を保証することができる．*N*クウィーン問題のプログラムに関する解の順序の保証に関しては後述する．

第 2 に，もとのプログラムが全解探索で停止しないばあいは，ベクトル化されたプログラムは停止しない．なぜなら，変換後のプログラムにおいては，非決定的な手続きにおいてすべての解がベクトルに蓄積されるまでは，その実行が終了しないからである．

第 3 に，第 2 の性質と同様の原因から，もとのプログラムが例外をおこすばあいは，ベクトル化後のプログラムの動作は保証されないということがいえる．

### 6.4.3 再帰的手続き定義の最適化されたベクトル化

この節では，再帰よびだしをふくむ非決定的な手続き定義のベクトル化法をしめす．すでにのべたように前節の方法を再帰よびだしをふくむ手続き定義に適用することもできるが，この節の方法を適用するほうが効率のよい目的プログラムを生成することができる．

この節でのべるベクトル化手順を適用するためには，対象となる手続き P は 6.4.2 節の条件 2 〜 4 とともにつぎの**条件** 5 をみたさなければならない．

**条件** 5　P の定義を構成する節のうちの 1 つが，1 個の自己再帰よびだしをふくむ．他の節は再帰よびだしをふくまない．

**条件** 5 はゆるい条件とはいえないが，実際にみられるほとんどの非決定的な再帰手続きはこの条件をみたしている．必要ならば，v_merge にかわる適当なくみこみ手続きを LIL にとりいれて使用することによって，**条件** 5 をゆるめることができる．

ベクトル化の概略手順は 6.4.2 節でのべたのとかわらない．標準化まではまったく同

一である．したがって本変換についてのべる．かんたんにするため，2 つの節から構成され，第 2 節が再帰よびだしをふくむ手続きのマスク演算方式への変換にかぎってのべる．変換すべき手続き P の標準形はつぎのとおりだとする．

```
P(X1, …, Xl) :- Q11, …, Q1n1.                                  …… (6.4.3.1)
P(X1, …, Xl) :-
    Q21, …, Q2n21, P(X1', …, Xl'), Q2n21+1, …, Q2n22.          …… (6.4.3.2)
```

ここで，引数のうち $X_1, \ldots, X_i$ が入力，$X_{i+1}, \ldots, X_l$ が出力だとする．変換後の手続き名を VP とすれば，本変換の結果，つぎのような LIL の手続きが出力される．

```
VP(X1, …, Xl, MI, BI, BO) :-
    VP_1(X1, …, Xi, XLi+1, …, XLl, MI, ML),
      v_merge([XLi+1, …, XLl],
              [Xi+1, …, Xl], [BI], [BO], ML).                  …… (6.4.3.3)

VP_1(_, …, _,  [], …, [], MI, []) :-
    v_finished(MI), !.                                         …… (6.4.3.4)
VP_1(X1, …, Xi, [XHi+1 | XLi+1], …, [XHl | XLl],
     MI, [MH | ML]) :-
    Q11', …, Q1n1', Q21', …, Q2n21',
    VP_1(X1', …, Xi', XLi+1', …, XLl', MI', ML'),
    MQ2n21+1', MQ2n22'.                                        …… (6.4.3.5)
```

手続き VP_1 においては，第 $i$+1 ～ $l$ 引数および最後の引数はマルチ・ベクトルである．VP_1 の再帰よびだしのたびに，これらの各マルチ・ベクトルに 1 個の要素がくわえられていく．手続き VP は VP_1 をよびだしたあと，v_merge をつかってこれらのマルチ・ベクトルを併合して，それぞれ 1 個のベクトルにする．

第 1 節対応部と $Q_{21}, \ldots, Q_{2n21}$ の変換法は 6.4.2 節におけるのと同様である．これらがベクトルに作用するのに対して，$Q_{2n21+1}, \ldots, Q_{2n22}$ は，マルチ・ベクトルに作用する．ただし，VP_1 のよびだし以前の部分で定義される値はマルチ・ベクトルではないので，これらのベクトルの要素をマルチ・ベクトルの要素ベクトルの要素と対応づけて計算する必要がある．

非決定的な再帰的手続きのベクトル化の例として，$N$ クウィーンの手続き v_select をとりあげる．まず標準形をしめす．2 つの節の対応する仮引数の変数名を統一するために，一部の変数名は原始プログラムとはかえてある．

```
select(T1, Y, T2) :- T1 = [Y|T2].                          ····· (6.4.3.6)
select(T1, Y, T2) :- T1 = [A|L],
    select(L, Y, X), T2 = [A|X].                           ····· (6.4.3.7)
```

本変換の結果はつぎのようになる．

```
v_select(AL, X, Y, MI, BI, BO) :-
    v_select_1(AL, X1L, Y1L, MI, ML),
    v_merge([X1L, Y1L], [X, Y], [BI], [BO], ML).           ····· (6.4.3.8)

v_select_1(_, [], [], MI, []) :-
    v_finished(MI), !.                                     ····· (6.4.3.9)
v_select_1(AL, [A' | X1L], [L' | Y1L], MI, [M1' | ML]) :-
    v_carcdr(A', L', AL, MI,  M1'),
    v_carcdr(A, L, AL, MI, M1),
    v_select_1(L, X1L, L1L, M1, ML1),
    map_v_cons(A, L1L, Y1L, ML1, ML).                      ····· (6.4.3.10)

map_v_cons(_, [], [], [], []).                             ····· (6.4.3.11)
map_v_cons(A, [XH|XL], [YH|YL], [MIH|MIL], [MOH|MOL]) :-
    v_cons(A, XH, YH, MIH, MOH),
    map_v_cons(A, XL, YL, MIL, MOL).                       ····· (6.4.3.12)
```

変換前の `T2 = [A|X]` をマルチ・ベクトルの全要素について実行するのが手続き `map_v_cons` である．`map_v_cons` では，(6.4.3.10) における変数 `A` の値をくりかえし使用している．

なお，6.4.2 節でのべたように一般には解がえられる順序はベクトル化によって保存されないが，*N* クウィーンにおいては順序を保存することが可能である．その方法と順序が保存される理由についてのべる．`v_select_1` には非決定的な手続きよびだしがふくまれないため，これらの手続きをマスク演算方式で実装すれば，`v_select_1` が出力する各マルチ・ベクトルを構成する要素ベクトルのベクトル長はひとしくなる．さらに，要素ベクトル $V_1, \ldots, V_m$ の各第 *i* 要素は，`v_select` のよびだしにおける入力ベクトルの第 *i* 要素からえられたものである．したがって，これらを `v_merge` でつぎのような順にならべれば，原始プログラムのふかさ優先順と一致する．

$$V_1[1], \ldots, V_m[1], V_1[2], \ldots, V_m[2], \ldots$$

`v_put` においては一般には解の順序は保存されないが，*N* クウィーンのばあいには解

が v_put の $N+1$ 回めのよびだしにおいて一度にえられるため，保存される[注4]．

金田ら [Kanada 88a] では，非決定的なのばあいの手続き append のベクトル化過程と LIL プログラムもしめしている．

なお，上記のように**条件** 2 〜 4, 5 がみたされれば変換をおこなうことはできるが，変換可能なすべてのプログラムにおいて高速実行が実現されるわけではない．すなわち，この変換によって十分なベクトル長がえられることが必要である．マクロにみれば，この処理方式は，少数の節が集中的に実行されるプログラムに適している．

この節の最後に**図** 6.9 に，以上のような OR ベクトル化 (プログラム変換) の結果として生じるプログラムの動作を，リストを分解する手続き append の実行を例として，変換前のプログラムと比較してしめす．

[注4] ただし，これはひとつの $N$ についての解だけをもとめるばあいにかぎる．ことなる $N$ について同時に解をもとめようとすれば解がいちどにはもとめられないため，解の順序は保存されない．

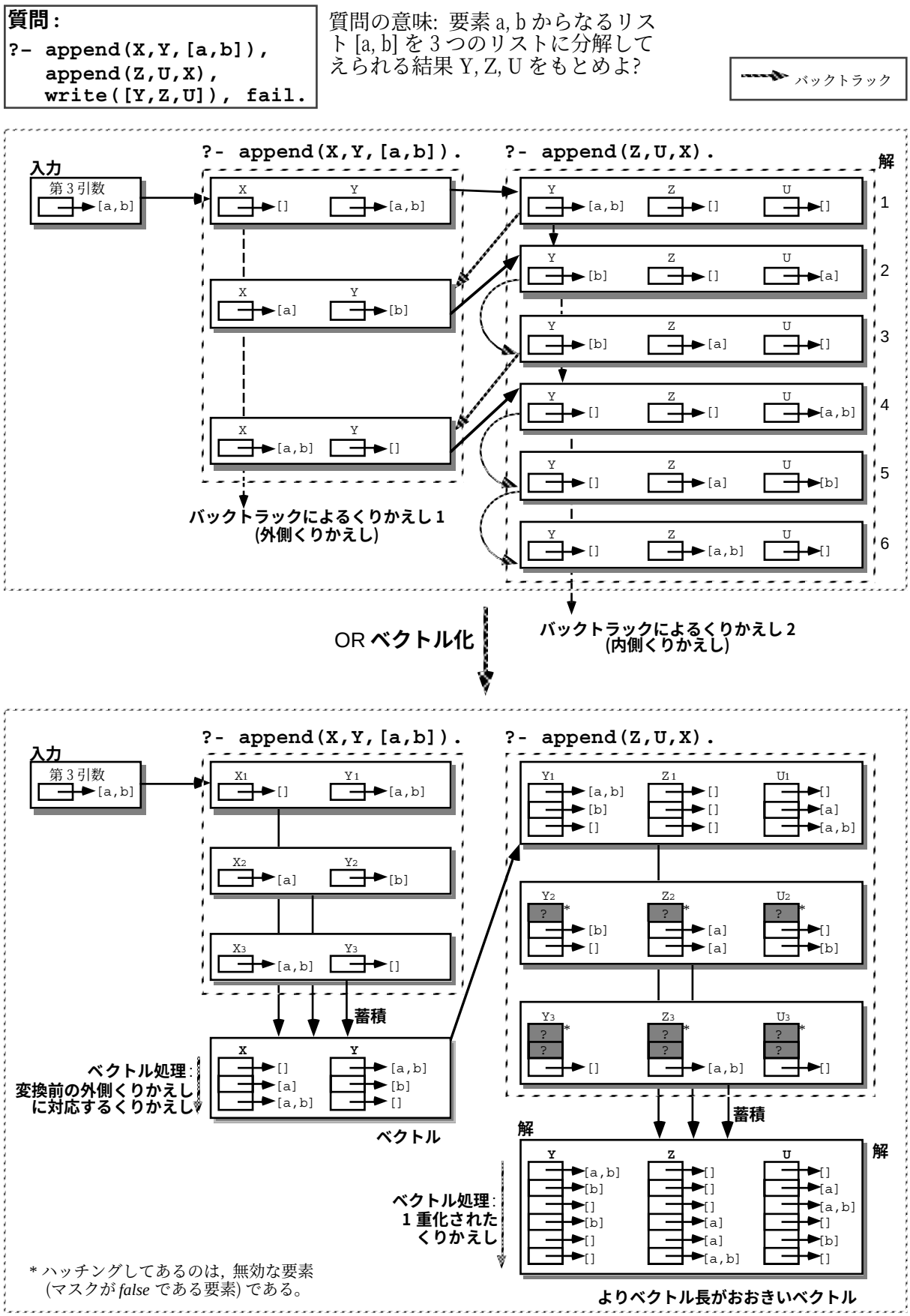



**図 6.9　OR ベクトル化前および OR ベクトル化後のプログラムの動作**

## 6.5 並列バックトラックの実現 (並列バックトラック化)

この節では，6.3 〜 6.4 節でのべた方法で生成される中間語プログラムに変更をくわえることなく，並列バックトラック計算法による実行を実現する方法についてのべる．

並列バックトラック計算法による実行を実現するためには，プログラム上のどの点で解候補ベクトルを分割するかを決定しなければならないが，われわれが採用したのは手続き v_merge の内部で分割する方法である．並列バックトラック計算法においては，部分解ベクトル (解候補からなるベクトル) を分割して，それぞれのベクトルに対する処理を逐次的におこなうが，そのために部分解ベクトルを分割しなければならない．分割をともなう v_merge の機能を**図** 6.10 にしめす．(a) にはベクトルの分割をおこなわないばあいすなわち完全 OR ベクトル計算法における v_merge の機能を例示する．(b)，(c) にはベクトルの分割をおこなうばあいすなわち並列バックトラック計算法における例をしめす．ここで (b) は (a) とは逆にベクトルの併合をまったくおこなわないばあいの例であり，(c) は中間的なばあいすなわち部分的な併合をおこなうばあいの例である．(a) においては v_merge の実行は決定的であり，v_merge のなかには選択点は存在しない．中間語 LIL には v_merge 以外に非決定的な手続きは存在しないので，これにより LIL の実行そのものが決定的になる．しかし，(b)，(c) においては v_merge の実行は非決定的であり，v_merge のなかに選択点が存在するためにバックトラックが発生すると v_merge のなかから実行が再開される．

v_merge を分割点とした理由は，つぎの 2 点である．

第 1 に，6.3 〜 6.4 節でしめした OR ベクトル計算法においてベクトル長が変化する唯一の点が v_merge であり，そこで分割するのがもっともオーバヘッドがすくないからである．他の点で分割しようとすれば，あらたなベクトルの記憶わりあてと内容の複写が必要になるが， v_merge でおこなえばそれをさけることができる．しかも，6.4 節の方法では v_merge において複数のベクトルをひとつのベクトルに併合していたが，この併合をやめる，あるいは部分的な併合にとどめることによって目的を達することができる．

第 2 に，v_merge において分割すれば，LIL の構文を変更することなく 並列バックトラック計算法による実行を実現することができる．すなわち，v_merge の意味を変更することによって 6.3 〜 6.4 節においては使用していなかった大域的バックトラックを LIL に導入し，それを並列バックトラックを実現するために使用している．大域的バックトラックのつかいかたは図 6.10 (b)，(c) にしめしたとおりである．このような LIL の意味の変更をおこなうことにより，LIL の構文にはまったく変更をくわえることなしに並列バックトラックを実現している．

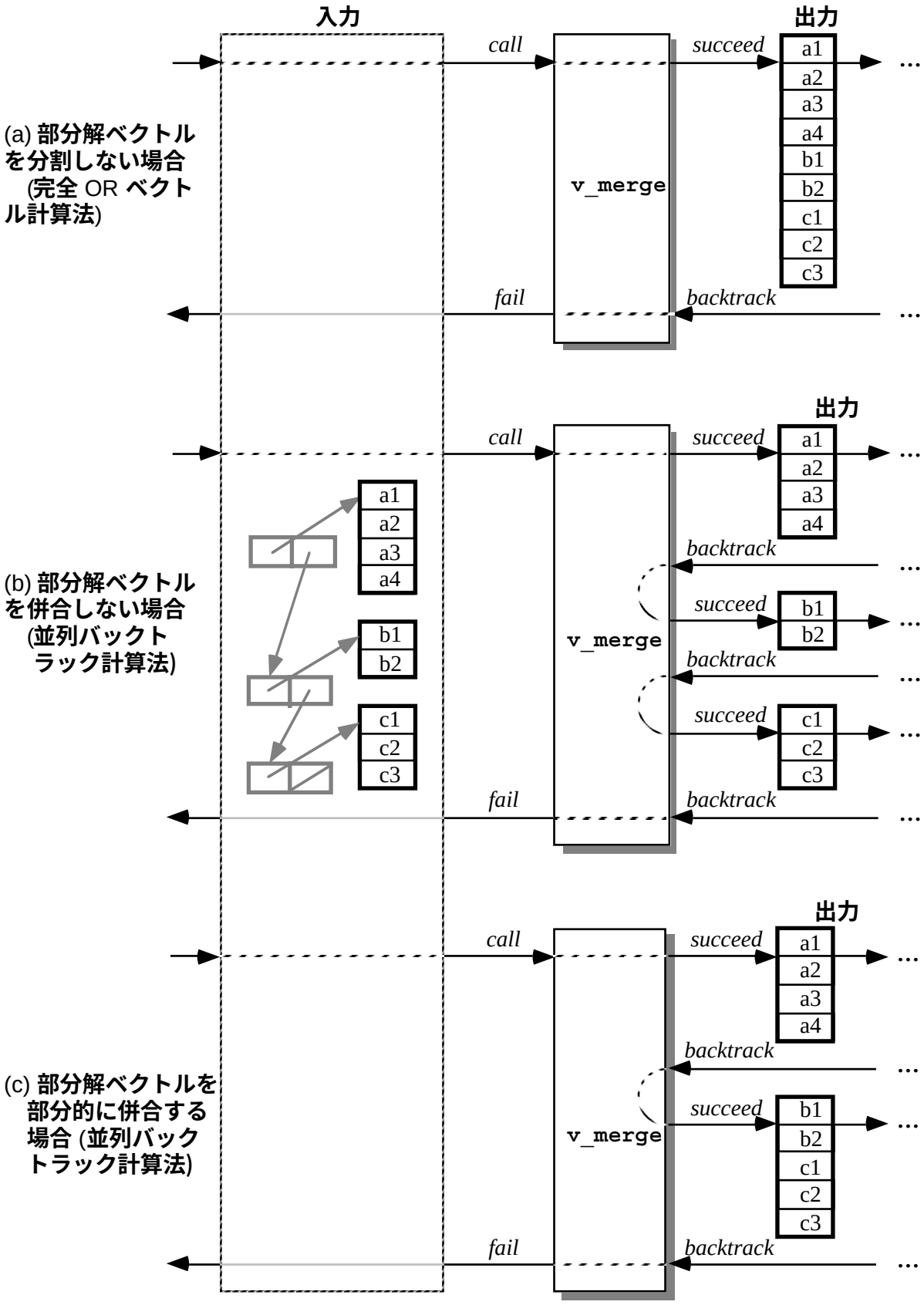



**図 6.10　並列バックトラック計算法における `v_merge` の機能**

# 6.6 評価

まず *N* クウィーンのプログラムを手動で変換して評価し，その結果をふまえてさらに自動ベクトル化処理系を開発して評価した．6.6.1 節では手動ベクトル化による評価結果，6.6.2 節では自動ベクトル化による評価結果をしめす．

## 6.6.1 手動ベクトル化による評価

われわれは，6.3 〜 6.4 節でしめした論理型言語 LIL のインタプリタを Prolog で作成して，*N* クウィーンのプログラムなどのベクトル化後のプログラムの動作をたしかめた．また，*N* クウィーンのプログラムを Fortran と Pascal とに手動で変換し，Fortran 部分をベクトル・コンパイラでコンパイルしたうえ実行時間を測定した．なお，この実行系にはガベジ・コレクタは実装していない．測定結果はすでに第 3 章でしめしたが，**表** 6.1 に再掲する．18 〜 19 ms の実行時間は，推論速度に換算すると 4.5 MLIPS (Million Logical Inference Per Second) となる[注5]．

表 6.1 からつぎのようなことがわかる．

(1) マスク演算方式，インデクス方式，圧縮方式のいずれも，エイト・クウィーン問題のプログラムの実行においては性能上ほとんど差はない．

(2) この性能は第 2 章でしめした配列を基本データ構造とするエイト・クウィーンのプログラムの約半分である．

(3) いずれの方式も S-810 においては，同一プログラムのスカラ処理に対するベクトル処理の加速率は 8 〜 9 倍である．

**表** 6.1　**ベクトル計算機** S-810 **むきにハンド・コンパイルした エイト・クウィーン問題のプログラムの実行性能**

| **プログラムの版 (主要な条件制御方式)** | S-810 **ベクトル処理時間** (ms) | S-810 **スカラ処理時間** (ms) | **加速率** |
|---|---|---|---|
| マスク演算版 | 18 | 167 | 9.3 |
| インデクス版 | 18 | 140 | 7.8 |
| 圧縮版 | 19 | 160 | 8.4 |

エイト・クウィーンの実行における実行時間のうちわけを**図** 6.11 にしめす．図 6.11 からつぎのようなことがわかる．

(1) いずれの方式においても，ベクトル要素が空リスト [] かどうかの判定 (図中では nil

[注5] なお，推論速度の計算においてくみこみ述語はかぞえていない．

判定) とリストの分解 (car, cdr), 合成 (cons) の実行比率がたかい (ベクトル処理のばあいで 40 % 程度).

(2) いずれの方式においてもリスト合成の加速率はひくい．これは，第 3 章でしめした結果にも一致している．

(3) ベクトルの複写・併合の実行比率は，マスク演算方式とインデクス方式においては数 % 程度，圧縮方式においては 20% 程度である．

このように実行時間のうちわけが各方式によってことなることから，結果としていずれの方式も実行時間の総和がほぼひとしくなったのは，エイト・クウィーン問題のプログラムにおける偶然にすぎないことがわかる．

これらの結果をもとにして考察すると，つぎのようなことがいえる．

(1) 上記の結果には，Fortran で記述したことなどによるオーバヘッドがふくまれているので，目的プログラムの最適化をはかれば，エイト・クウィーン問題の全解探索が S-810 において 10 ms 以下で実行できると期待することができる[注6]．

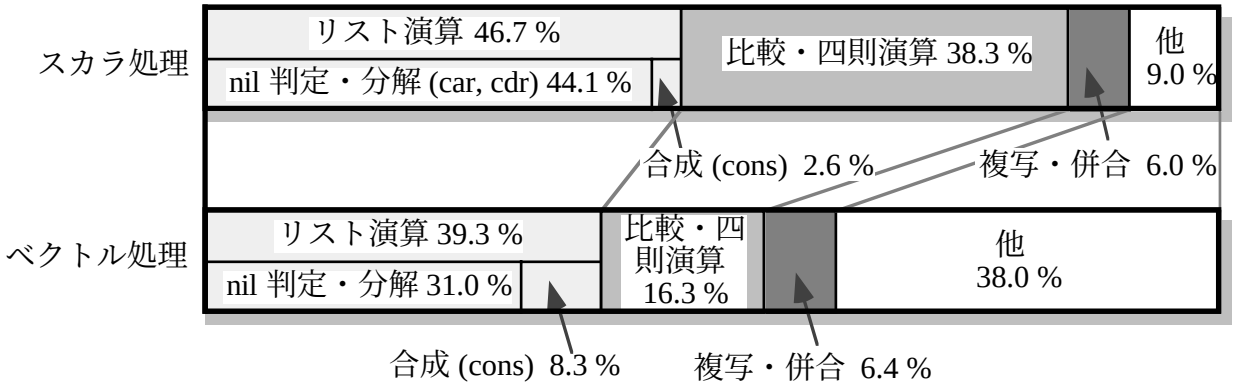
(2) リスト版 *N* クウィーンの性能が第 2 章でしめした配列版 *N* クウィーンの約半分にとどまっているのは，上記のオーバヘッドがあること, リスト・ベクトル (インデクス・ベクトル) を多用しているために，逐次アクセスのばあいとくらべて主記憶とベクトル・レジスタとのスルー・プットが減少していることなどがおもな理由だとかんがえられる．

(3) マスク演算方式とインデクス方式とにおいては，*N* クウィーンにおいてはベクトルの複写・併合は顕著なオーバヘッドになっていない．しかし，圧縮方式においてはオーバヘッドになっている．第 2 章における配列版のプログラムにおいては配列複写・圧縮の実行比率が 40 % 以上であり，オーバヘッドになっている．これにくらべて，マスク演算方式とインデクス方式とにおいては，データ構造のよりおおくの部分をプロセス間で共有するようにしているため，オーバヘッドとなっていないのだとかんがえられる．

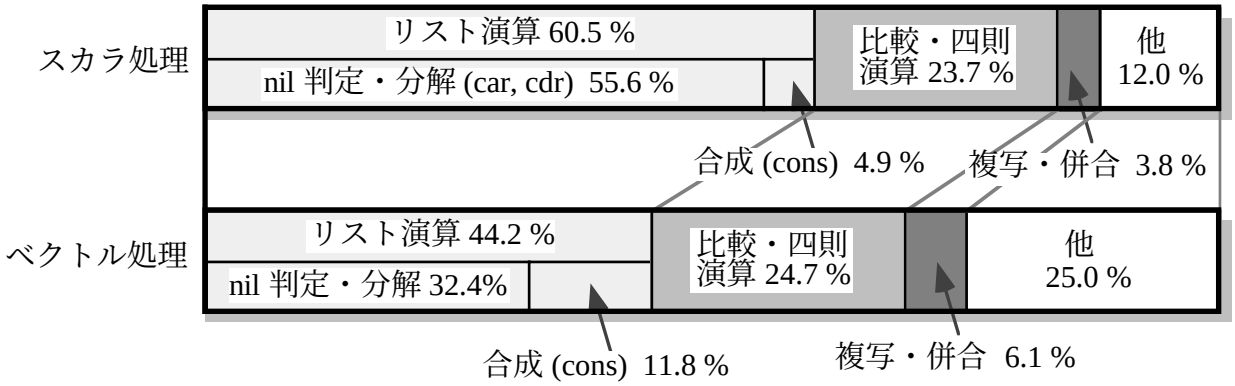
なお，リスト合成の加速率に関しては第 2 章ですでに考察したので，ここではくりかえさない．

[注6] Fortran で記述したためのオーバヘッドの例としては，シフト命令を生成することをうまく指示してやれないためにリストの分解においてベクトル除算命令が実行されてしまうことなどがあげられる．

(1) **マスク演算方式**




(2) **インデクス方式**




(3) **圧縮方式**

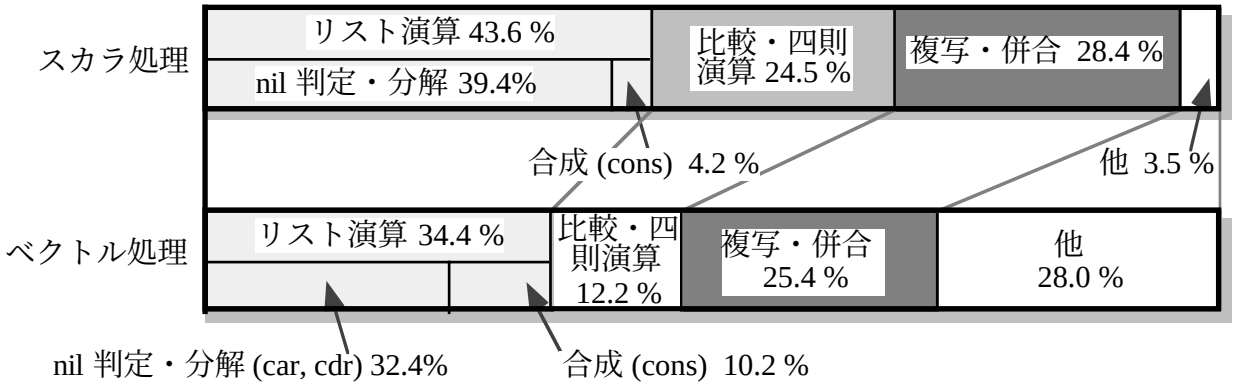



**図 6.11　ベクトル化後のエイト・クウィーンの実行における各種の演算の実行比率**

### 6.6.2 自動ベクトル化処理系の試作による評価

さらに，この章でしめした手順にしたがってベクトル化をおこない，マスク演算方式によるベクトル化後のプログラムを (LIL ではなく) Lisp の構文・意味にしたがって出力する自動ベクトライザを作成した．また，自動ベクトライザが出力する Lisp プログラムを，ベクトル計算機 S-810 のベクトル命令をふくむライブラリ・プログラムをよびだす目的プログラムに変換するコード生成系およびその実行系を Lisp 上に作成した．処理系

じたいについては第 8 章でのべる．

図 6.1 にしめした *N* クウィーンの Prolog プログラムをこの処理系で翻訳して実行し，動作を確認した．その結果，エイト・クウィーンで 2.6 MLIPS の実行速度をえた．最適化が不十分なため，自動変換したプログラムは手動変換したものにくらべて低速だが，改良すれば，後者をうわまわる実行速度をえることも可能であろう．

なお，6.5 節でしめした並列バックトラック化により生成されたプログラムの実測による性能評価はおこなっていない[注7]．しかし，みつもりによると，完全 OR ベクトル計算法にくらべた並列バックトラック計算法のオーバヘッドは約 1 % という結果がえられている．みつもりの根拠はつぎのとおりである．まず，エイト・クウィーン問題のプログラムの完全 OR ベクトル計算法による実行時間は 18 ms である．並列バックトラックのオーバヘッドを概算した結果，1 回のバックトラックにつき 20 μs となった．エイト・クウィーンのばあいのバックトラック回数は10 回以下であるから，オーバヘッドの総計は約 0.2 ms とみつもられる．これは全実行時間の約 1 % である．したがって，並列バックトラック化においても完全 OR ベクトル化と同程度の高い性能がえられるとかんがえられる．

[注7] そのかわり，並列バックトラック計算法にもとづいて中間語を解釈して逐次実行するシミュレータを Prolog 上に作成した．

## 6.7 引数モード不確定のばあいへの適用拡大

これまでのべてきたベクトル化法は，単に OR 並列性があるだけでなく，引数のモードが確定していなければならないこと，カットや否定などの手続きをふくんでいないことなど，さまざまな条件をみたしていなければならなかった．これらのなかでも，引数のモードに関する仮定は，論理プログラミングの利点をそこなうきびしい制約だということができる．そこで，ここでは引数モードが不確定のばあいに OR ベクトル化を拡大適用するための方法について考察する．

まず，引数モードが不確定のばあいに生じる問題について考察する．引数モードが確定しているばあいには，ベクトル化された手続きに入力されるデータは複写することなくそのままつかうことができ，その手続きから出力するデータはまったくあらたにベクトルをつくって出力すればよい．ところが，**図** 6.12 (a) に例示するように，非決定的な手続きに変数をふくむデータが入力されるばあいには，その変数に対して複数の値があたえられうるため，複写せずに使用すると，それらのあいだで矛盾が生じる可能性がある．この問題をさけるには，図 6.12 (b) のように，非決定的な手続きの実行前に入力データを複写しておけばよい．この原理にもとづくベクトル化の例を**図** 6.13 にしめし，その動作を**図** 6.14 にしめす．図 6.13 においては手続き append をリストを分解する非決定的な手続きとして使用しているが，このベクトル化手続きは入力データのモードに依存しないため，図 6.13 (b) にしめしたベクトル化後の手続きは，このままでリストの合成のためにも使用することができる．

ところが，この方法では複雑なデータが入力されるばあいには入力データ複写のオーバヘッドがおおきく，そのためにスカラ処理より時間がかかる可能性がたかい．データ複写のオーバヘッドがおおきいのは，入力データ量に比例する時間がかかるという理由のほかに，入力データに同一の変数が複数回あらわれるばあいにもただしく複写するために，複写手順が複雑になるという理由もある．したがって，上書きラベル・フィルタ法を応用したデータ複写の高速ベクトル処理アルゴリズムを開発するなどして，複写のオーバヘッドをへらすことが重要な課題となる．

また，モードがきまっているばあいにくらべると，ユニフィケーションの手順も複雑化する．このようなばあいのユニフィケーションに関しては金田 [Kanada 85, Kanada 88a] において考察している．

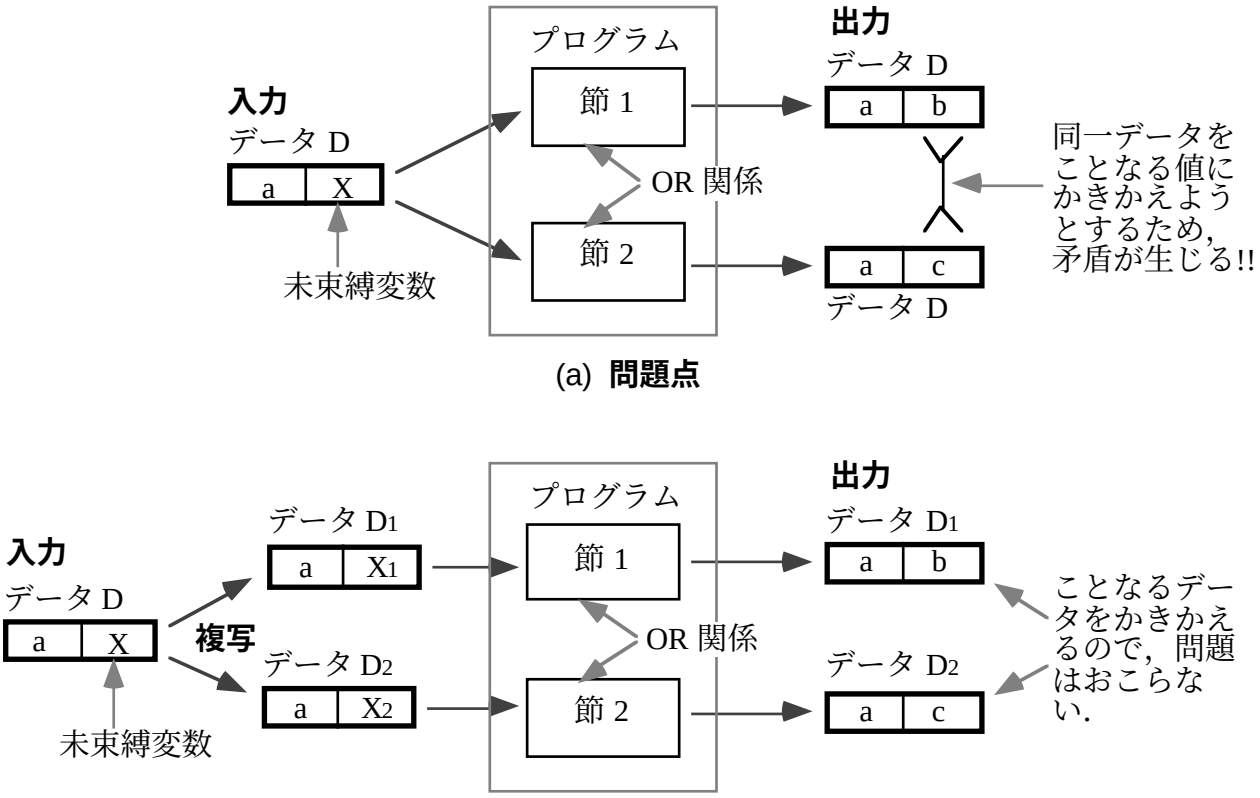



**図 6.12　モード不確定の引数に対する非決定的な処理の問題点とその解決策**

前節までにしめしたベクトル化法も図 6.13 にしめした方法を最適化した方法となっている．ベクトル化の適用範囲をひろげるには，図 6.13 の方法をもとにしてむだがすくなくかつ適用範囲がひろい方法を開発することが必要である．また，まだインプリメントされたことがない図 6.13 の方法をインプリメントして，そのオーバヘッドがどの程度おきいかをたしかめることも必要だとかんがえられる．入力データ複写にもとづくモードによらないベクトル化法に関しては，**付録**でさらにくわしく説明する．

```
append([],Z,Z).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
?- bagof((XX,YY),append(XX,YY,[1,2,3]),S).
```

**(1) ベクトル化前の全解探索プログラム**

```
v_append(Out,In) :-
      v_finished(In) -> Out = {}                    % 再帰よびだしの終了判定 / 処理.
  ;   v_copy(In1,In),                                % 入力データを複写する.
      v_append_1(Out1,In1),
      v_copy(In2,In),                                % 入力データを複写する.
      v_append_2(Out2,In2),
      v_merge(Out,Out1,Out2).          % ベクトル Out1, Out2 を併合して Out とする.

v_append_1(Out,In) :-  v_filter(Out,In,append_1).
          % 手続き append_1 を入力ベクトル In の全要素について実行し，えられ
          % た解からなるベクトルを Out とする (v_filter は，引数としてわた
          % された任意の手続きをベクトルの全要素に適用するための手続きである.
          % v_filter の定義は省略した).

append_1(Env,#([],Z,Z|Env)).                        % append の第 1 節に対応.

v_append_2(Out,In) :-
      v_filter(T1,In,append_2),
          % 手続き append_2 を入力ベクトル In の全要素について実行する.
      v_append(Out,T1).                             % v_append の再帰よびだし.

append_2(#(X,Y,Z|Env),#([A|X],Y,[A|Z]|Env)).       % append の第 2 節に対応.

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

**(2) ベクトル化後のプログラム**

**図 6.13　入力データ複写にもとづく非決定的な手続き append のベクトル化**

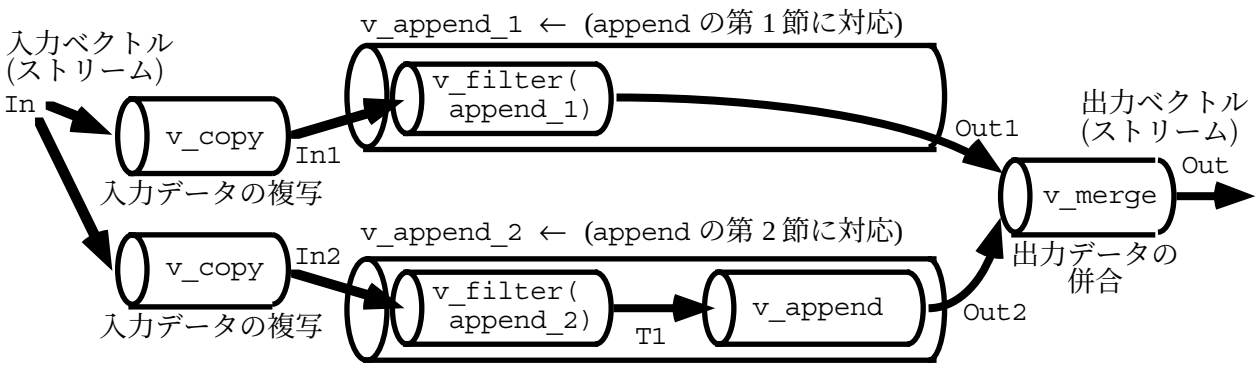



**図 6.14　v_append の動作のパイプラインによる表現**

## 6.8 関連研究

この研究に関連する研究の一部については 6.1 節でかんたんにのべたが，ここでは他の研究もふくめてよりくわしくのべる．

### (1) プログラム変換による論理型言語の OR 並列処理

ベクトル計算機に関する研究ではないが，上田，Codish ら，佐藤らによる，プログラム変換による論理型言語の OR 並列処理に関する研究は，この研究と関連がふかい．

上田 [Ueda 85b, Ueda 86] は非決定的な論理型言語プログラムの決定的な論理型言語プログラムへの変換を研究している．上田の研究の目的は Prolog のプログラムを，上田によって設計され KL1 の核となった並列論理型言語 (Flat) GHC に変換することにある．すなわち， Prolog のプログラムは大域的な OR 並列性をふくんでいるが，Flat GHCでは局所的な OR 並列処理しか記述することができないので，プログラム変換によって OR 並列処理を AND 並列処理にかきかえる必要がある．上田の研究はこのプログラム変換に関するものである．上田の方法は continuation を陽に導入して逐次論理プログラムを並列論理型言語でシミュレートする．したがって，パイプライン的な処理をめざすこの研究には適さない変換法だとかんがえられる．

Codish と Shapiro による研究 [Codish 85] も同様の目的の研究だが，Codish らは Concurrent Plolog (CP) から CP への変換をおこなっている．Concurrent Prolog (CP) は Shapiro によって設計された並列論理型言語である．CP は大域的な OR 並列を記述することができるので，この研究では CP のよりおおきなサブセットから大域的な OR 並列性をふくまないよりちいさなサブセットへの変換をおこなっているなどの特徴がある．

玉木 [Tamaki 87] も同様の目的の研究だが，彼らの方法においては，変換後のプログラムはパイプライン的な処理をおこなう．その点でこの研究と共通点があるが，ベクトル処理をめざしたものではなく MIMD 的並列処理のための変換である．

### (2) ベクトル計算機による MIMD 的並列処理の研究

FLENG Prolog [Nilsson 86] は GHC のサブセットであり，セマンティクスを単純化することによってベクトル計算機での実行を容易にしようとしている．パイプラインで実行するべきベクトルの各要素はそれぞれ FLENG のひとつの節に対応する．複数の節のデータをひとつのベクトルにまとめることによってベクトル実行を可能にしようとしている．そのため，ひとつのベクトルの各要素に対してそれぞれことなる演算をおこなわなければならない．VELVET [Nagashima 86] のようにベクトルの各要素にことなる論理演算がおこなえるようなベクトル命令をそなえたベクトル計算機アーキテクチャも存在するが，より多様な演算を同時におこなうことをかんがえると，現在のベクトル計算機

アーキテクチャに改良をほどこしたとしても，このようなベクトル演算を実現することは困難である．一方，このようなアプローチは，パイプライン型ベクトル計算機より柔軟な演算が可能な SIMD 型並列計算機においては成功する可能性があるとかんがえられる．実際，Nilsson もその後 Connection Machine による MIMD 的並列処理を研究している [Nilsson 87c, Nilsson 88e].

### (3) 専用関数による Lisp のベクトル処理

阿部ら [Abe 90a, Abe 90b] および小林ら [Kobayashi 91] は Lisp に vmap 関数および vmap マクロというベクトル処理専用の機能をくみこみ，これらをつかって記述されたプログラムのベクトル処理を可能にしている．この研究では，通常の Lisp 関数のベクトル化はおこなっていないかわりに，map 関数の意味に最小限の修正をくわえた vmap 関数と vmap マクロという自然な意味をもった機能によって，ユーザにとっても処理系にとっても無理のないベクトル処理を可能にしている．

ベクトル処理専用の機能を言語にくみこむという点では，CM-Lisp (Connection Machine Lisp) [Hillis 85, Hillis 90] のアプローチも共通している．ただし，CM-Lisp は SIMD 型並列計算機である Connection Machine のための言語であり，パイプライン型ベクトル計算機を目的としているわけではない．

### (4) その他の研究

平井ら[Hirai 86] による富士通 VP-200 上の処理系，島崎ら[Shimazaki 89]，日立 TO 武宮ら[Takemiya 90a, Takemiya 90b] などによる，Backus の FP 言語 [Backus 78] のベクトル処理に関する研究がある．

## 6.9 まとめ

ベクトル計算機を使用して OR 並列性がある論理型言語プログラムを並列処理するためのプログラム変換法 (ベクトル化法) を開発した．この方法においては，原始プログラムの論理変数ごとに，それがとりうる複数の値を各要素とするベクトルをつくってベクトル処理する．

この方法を *N* クウィーン問題のプログラムに適用して自動変換し，生成されたプログラムの動作のただしさを確認するとともに，ベクトル計算機 S-810 によって実行して 2.6 MLIPS という高速度をえた．現在の方法は，引数の入出力モードが確定していなければならない，高速化されるプログラムの範囲がかぎられているなどの限界がある．したがって，ベクトル化法を改良して適用範囲の拡大をはかることが今後の課題である．とくに，入力データの複写オーバヘッドがすくなく，かつ入力データのモードによらないベクトル化法を開発することが重要だとかんがえられる．その実現のためには，現在は使用していない共有部分があるデータのベクトル処理方法の応用も必要になるとかんがえられる．また，最適化による速度向上も重要な課題である．