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．

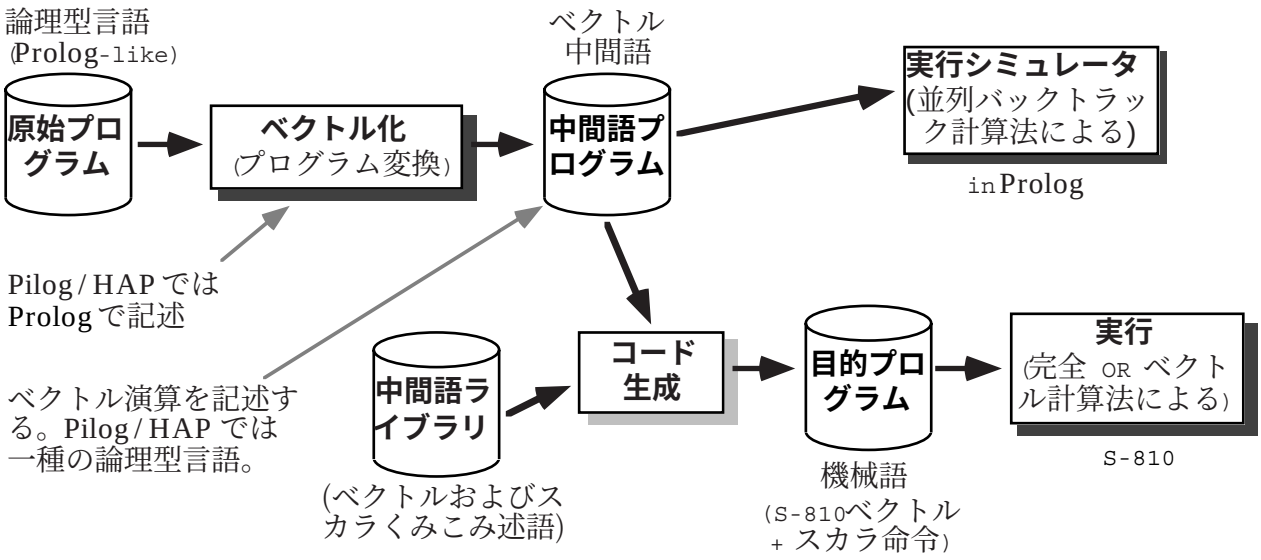

図 8.1　ベクトル計算機のための論理型言語試作処理系 Pilog/HAP における処理方法

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命

[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．

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

[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．

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．

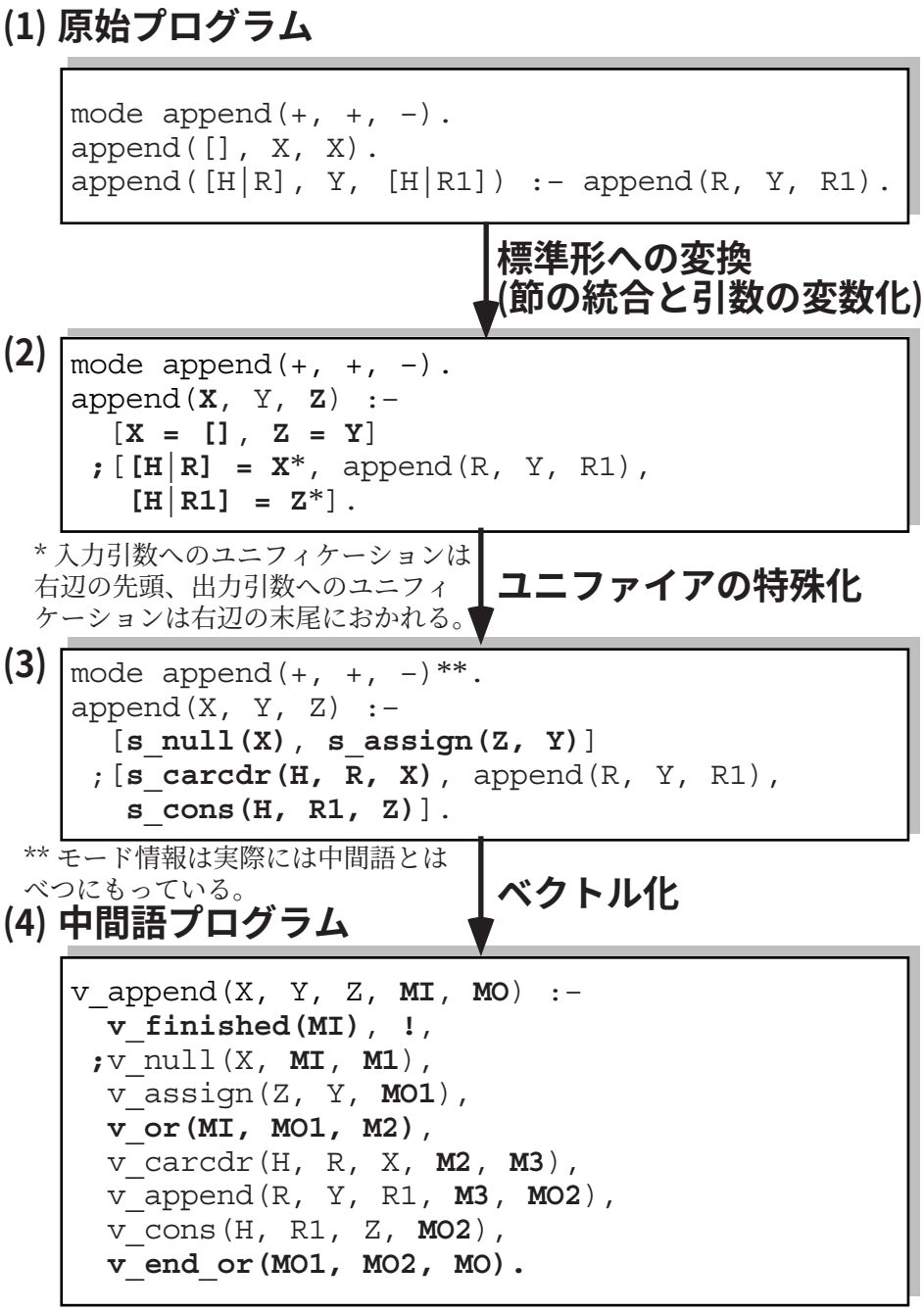

図 8.3　手続き append の決定性のばあいのベクトル化過程

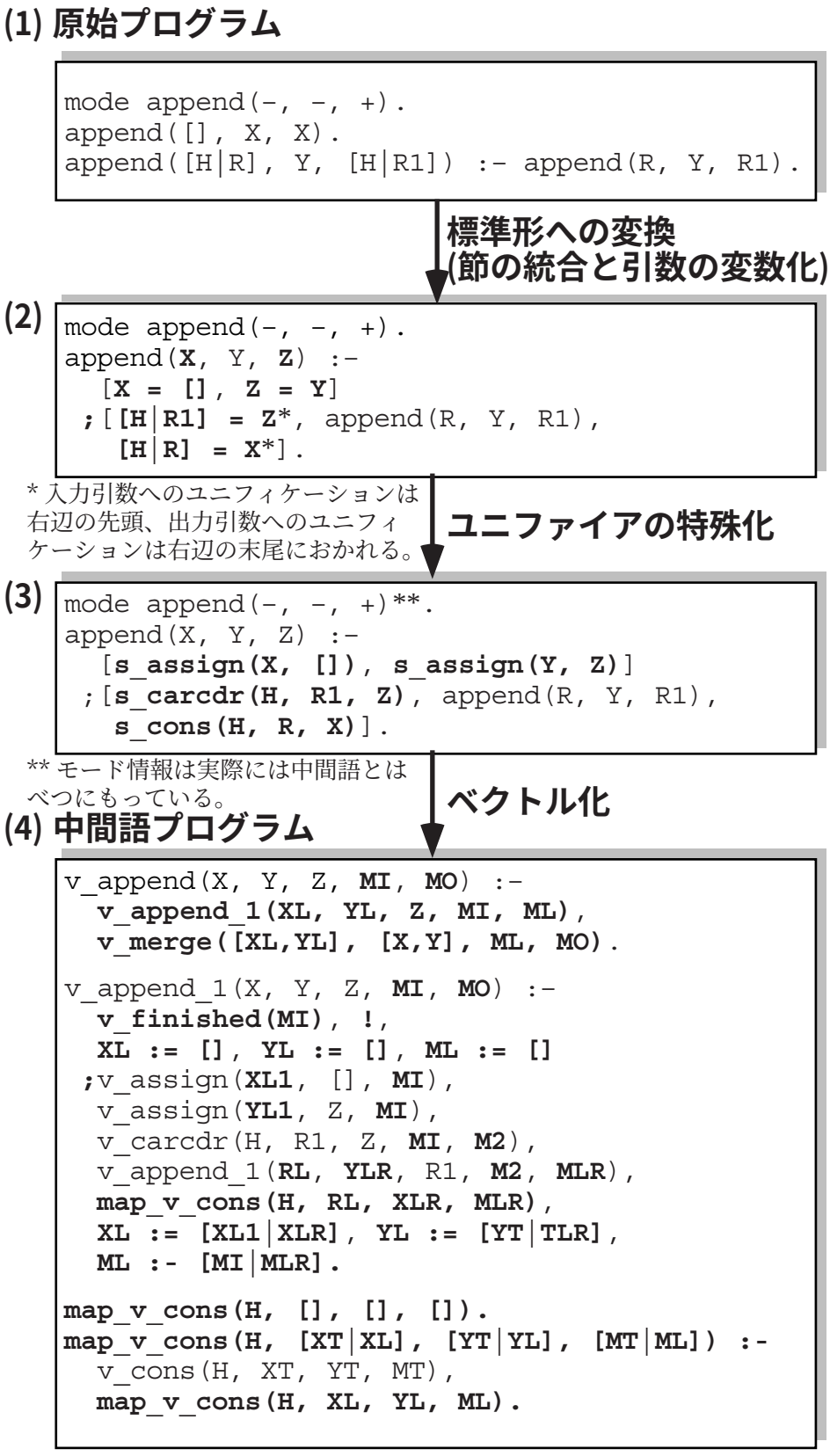

図 8.4　手続き append の非決定性のばあいのベクトル化過程

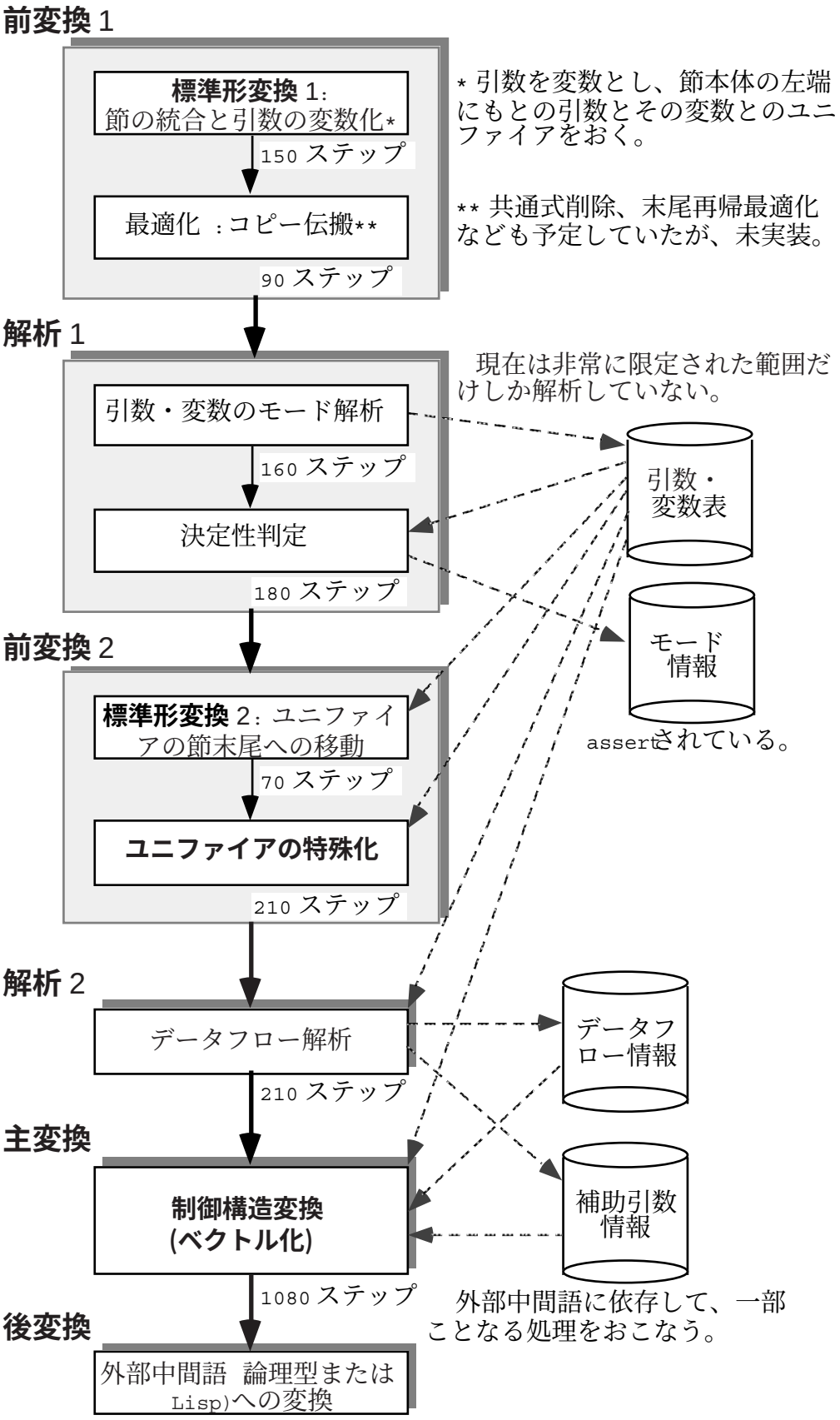

図 8.5　Pilog/HAP におけるベクトル化処理手順

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

[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．

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．

[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．

❑ **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．