プログラム変換にもとづくリストのベクトル処理方法とそのエイト・クウィーン問題への適用†

金田　泰[††]　菅谷正弘[††]

この論文では，ベクトル計算機を使用してリスト処理を高速に実行するための基本戦略を提案する．この戦略は，Fortran プログラムのベクトル化技法の拡張と考えることができる「くりかえし構造の交換」と「くりかえし構造の一重化」というプログラム変換技法にもとづいている．変換結果のプログラムにおいては，複数のリストを要素とするベクトルを使用する．それらに関する操作をベクトル計算機のリスト・ベクトル処理機能や数列生成機能などを用いて実現する．上記の戦略をエイト・クウィーン問題の Prolog プログラムに適用し，ベクトル計算機 S-810 においてスカラ処理の約9倍の実行速度をえた．

1. はじめに

リスト (linked list) は記号処理においてもっとも重要なデータ構造であり，Lisp や Prolog などの記号処理用言語において欠かせない．したがって，記号処理応用プログラムの高速化および記号処理用言語の高速処理のためには，リスト処理の高速化がもっとも重要な課題である．

リスト処理高速化の方法としては，最適化コンパイラによって逐次計算機用の高速なプログラムを生成するという方法もある．しかし，根本的な高速化のためには，並行処理むきハードウェアの使用が必須である．並行処理むきハードウェアを使用する高速化方法としては，つぎのようなものが考えられる．

（1） 実行時の負荷分散にもとづく方法

複数の処理装置をもつ並列計算機において，並行処理可能な部分処理を実行時にことなるプロセッサに負荷分散し，実行する．各処理装置がまったくことなる処理をおこなうので，MIMD 型並列計算機に適した方法である[1),2)]．

（2） コンパイル時のプログラム変換にもとづく方法

コンパイル時に並行処理可能な部分をみつけ，ベクトル計算機用または並列計算機用のプログラムを生成し実行する．MIMD 型並列計算機においてもこの方法は利点があるだろうが，逐次処理むきのプログラムをそのままでは実行できない日立 S-820 や Cray-2 のようなベクトル計算機や，SIMD 型並列計算機に適した方法である．実行時の負荷分散オーバヘッドがない点が(1)より有利だと考えられる．

（3） プログラミング時のアルゴリズム改良にもとづく方法

プログラマが並行処理むきのプログラムを記述する．どのようなアーキテクチャの計算機にも有効な方法だが，かなりアーキテクチャに依存したプログラミングが要求される．単純なリコーディングなどでは十分な性能はえられず，アルゴリズムの再検討が要求される[3)]．

これらの方法はいずれも研究途上にあり，どの応用においてどの方法が成功するかは現在のところわからない．この論文であつかうのは(2)の方法である．

これまで，ベクトル計算機はほとんど数値計算専用機と考えられてきた．また，現在でも Fortran がベクトル計算機で使用できるほとんど唯一の言語である．したがって，ベクトル計算機のためのコンパイル技術あるいはベクトル化技術は，数値計算プログラムにおいて，また Fortran において発展してきた．

ベクトル計算機は，ベクトル処理を高速におこなうことができる専用ハードウェアと，それに関する命令（ベクトル命令）とをもっている．ベクトル計算機の Fortran コンパイラにおいては，DO ループ内の演算の実行順序を変更して同種の演算をまとめることにより，それを1個のベクトル命令で実行するようにする．これによって，それらの演算をベクトル処理専用ハードウェアで実行することを可能にしている．この演算順序の変更をベクトル化とよんでいる．

しかし，Fortran において発展してきたベクトル化技術をそのまま記号処理プログラムおよび記号処理用言語に適用するだけでは，リスト処理プログラムのベクトル化は実現できない．その理由をのべる．

† A Vector Processing Method of Lists Based on Program Transformation and Its Application to the Eight-Queens Problem by YASUSI KANADA and MASAHIRO SUGAYA (Central Research Laboratory, Hitachi, Ltd.).

†† (株)日立製作所中央研究所

Fortran コンパイラにおいては，プログラム中のループを検出し，そのうちの可能なものをベクトル化する．Fortran プログラムにあらわれるデータ構造は配列であり，ベクトル化に適するかどうかの判定は配列添字の比較をベースにしておこなわれる．

一方，記号処理用言語で記述されたリスト処理プログラムにおいては，第1に制御構造としては，ループが存在せず，再帰よびだしが使われることが多い．また，第2にデータ構造としては，配列はあらわれず，再帰よびだしごとに独立のリストが処理されることが多い．したがって，配列添字の比較をベースにしたベクトル化法では，まったく対処できない．

しかも，リスト処理のプログラムは，もとのままでは本質的にベクトル処理に適さない構造のプログラムであることが多い．ベクトル処理に適さないのは，つぎのような理由による．各要素の処理のあいだにデータ依存性があるとベクトル処理はできないが，リスト処理プログラムの最内側ループは，通常，リストの各要素を順に処理していくため，データ依存性がある．また，最内側ループが短すぎるためにたとえベクトル化できても加速されない場合もある（第4章で示すエイト・クウィーン問題のプログラムにおける手続き not_take1 はその例のひとつである）．したがって，新たなベクトル化法を開発しなければ，ベクトル計算機によるリスト処理を実現することはできない．

第2章では，準備として，Fortran コンパイラで使われている3つの基礎的なベクトル化技術について説明する．第3章では，プログラム変換にもとづくベクトル・リスト処理の戦略についてのべる．第4章では，変換後のプログラムで使われるリスト処理基本演算のベクトル処理方法についてのべる．第5章では，エイト・クウィーン問題のプログラムにおける変換の適用例についてのべる．第6章でそのプログラムに関する実測結果を示し，最後に結論をのべる．

2. ベクトル化の基礎

この論文でのべるリスト処理プログラムのベクトル化においても，Fortran におけるベクトル化技術が基礎となる．したがって，この章ではベクトル化の基礎について簡単に説明する．

ベクトル化は一種のプログラム変換とみなすことができる[4]．ベクトル化はつぎのようなプログラム変換の組みあわせである．

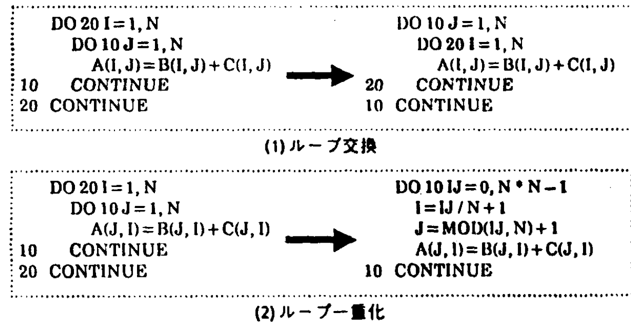


図 1 Fortran におけるループ交換とループ一重化
Fig. 1 Loop interchange and loop unrolling in Fortran.

（1） ループ分散 (loop distribution)

ベクトル命令は，ロード，加算，ストアなどの1個の基本操作だけを，あたえられたベクトルの各要素におこなう微小なループとみなすことができる．DO ループをこのような微小なループに分割するプログラム変換を，ループ分散という[5]．

（2） ループ交換 (loop interchange)

Fortran のプログラムのなかには，ループ分散だけではループ内の一部の操作をベクトル命令に変換できないプログラムもある．このような場合には，外側ループとの入れかえ（くりかえし方向の変更）をおこなって最内側ループの全体をベクトル化可能にするというプログラム変換がおこなわれることがある．この変換はループ交換とよばれる[6),7]．ループ交換は，後述するループ一重化と同様に，ベクトル長をのばすことを目的としておこなわれることもある．例を図1（1）に示す（簡単な例にするため，変換前にもベクトル化可能なプログラムを示している）．

（3） ループ一重化 (loop unrolling)

ベクトル計算機においては，最内側ループのくりかえし回数すなわちベクトル長をのばすことが性能向上につながる．したがって，多重ループを一重ループにする変換がおこなわれる．この変換はループ一重化とよばれる[8]．例を図1（2）に示す．

3. ベクトル・リスト処理の戦略

この章では，まず，我々が提案するリストのベクトル処理戦略におけるプログラムの処理手順を示す．つづいて，そこで使われる個別のプログラム変換技法について説明する．最後に，この戦略における可変長リストのあつかいについて，とくに説明する．

3.1 プログラムの処理手順

このベクトル処理戦略では，およそつぎのような手順でリスト処理をおこなう．

(A) プログラミング

プログラマが，記号処理用言語などで自然なリスト処理プログラムを記述する．

(B) プログラム変換

上記のプログラムを，プログラム変換によってベクトル計算機で処理可能なプログラムに変換する．変換はプログラマがおこなうか，コンパイラまたはプリプロセッサによっておこなう．

(C) 実行

変換後のプログラムをベクトル計算機で実行する．ただし，変換後のプログラムが原始プログラムとしてあたえられる場合は，実行のまえにコンパイルする必要がある．

(B)におけるプログラム変換は，第2章で説明した Fortran のベクトル化よりこみいっているが，その拡張と考えることができる．このプログラム変換はつぎのような2種類の変換の組みあわせである．

(1) くりかえし構造の交換

(2) くりかえし構造の一重化

これらについて，つづく2つの節で説明する．これらの節では，簡単のため固定長のリストをあつかう．可変長リストのあつかいについて 3.4 節でのべる．

3.2 くりかえし構造の交換

くりかえし構造の交換とは，多重のくりかえし構造における内側のくりかえしと外側のくりかえしとを入れかえることによってベクトル処理できないプログラムをベクトル処理可能にするプログラム変換のことである．図2(1)に例を示す．くりかえし構造の変換によって，図2(1.1) に PAD (Problem Analysis Diagram) を使って示したようなプログラムが図2(1.2) に示したようなプログラムに変換される．

対象となるくりかえし構造は，ループ，末尾再帰よびだし (tail recursion), あるいは，自動バックトラックをひきおこす一種のくりかえし構造である．Prolog プログラムに関していえば，再帰よびだしによって内側のくりかえしが形成され，再帰よびだしやバックトラックによって外側のくりかえしが形成されている場合に，くりかえし構造の交換の対象になりうる．

図3はくりかえし構造の交換前のプログラムの動作と，交換後のプログラムの動作とを対比して示している．図3において，細線であらわしたデータは操作対象である複数のリストであり，太線であらわしたデータは処理過程で一時的につくられるデータである．図3(1)のように，もとのプログラムの内側のくりかえしはリストの要素に関するくりかえしであって，リスト要素間のデータ依存性のためにベクトル処理できない．しかし，交換の結果，図3(2)のように，内側のくりかえしがことなる（データ依存性がない）リストに関するくりかえしとなり，ベクトル処理可能になる．

くりかえし構造の変換は，図1(1)に示したループ交換の拡張と考えることができる．基本演算のベクトル処理方法については第3章でくわしくのべ，くりかえし構造の交換の例は第4章で示す．

3.3 くりかえし構造の一重化

くりかえし構造の一重化とは，多重のくりかえし構造を一重のくりかえし構造に変換するプログラム変換のことである．図2(2)に2重のくりかえし構造の一重化を図示する．くりかえし構造の一重化によって，図2(2.1)に示したようなプログラムが図2(2.2)に示したようなプログラムに変換される．

Prolog プログラムに関していえば，再帰よびだしやバックトラックによって多重のくりかえしが形成さ

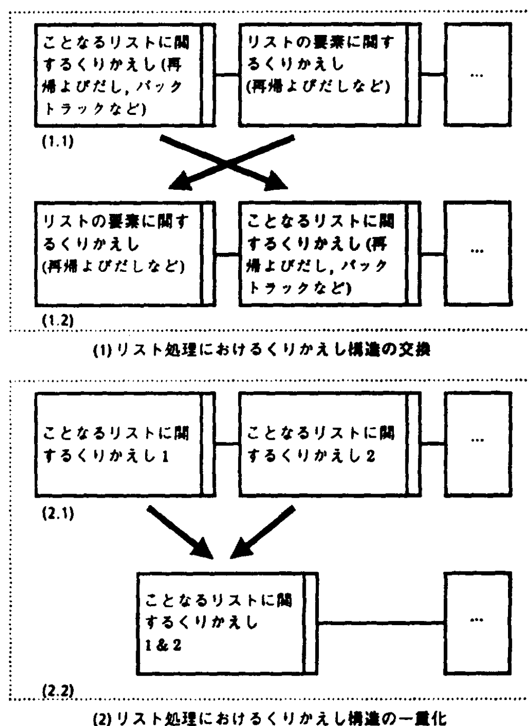


図2 くりかえし構造の交換と一重化
Fig. 2 Interchange and unrolling of repetitive structures.

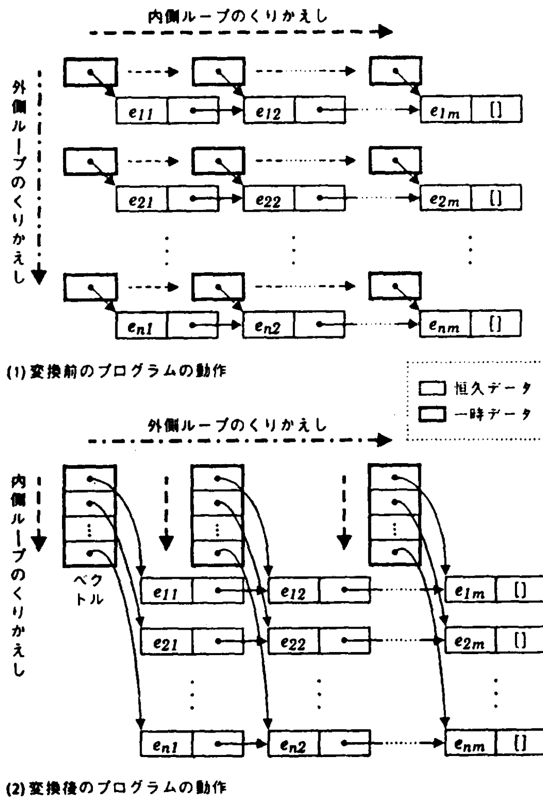


図 3 くりかえし構造の変換によるベクトルの生成とくりかえし方向の変化
Fig. 3 Generation of vectors and change of repetition direction by interchange of repetitive structures.

れる場合に，くりかえし構造の一重化の対象になりうる．この変換によってスカラ処理（ベクトル長が1のベクトル処理）をベクトル処理に変換したり，ベクトル長をのばしてベクトル計算機による実行速度を向上させたりすることができる．

この変換は，図1(2)に示した Fortran のベクトル化における多重ループの一重化の拡張と考えることができる．例は第4章で示す．

3.4 可変長リストのあつかい

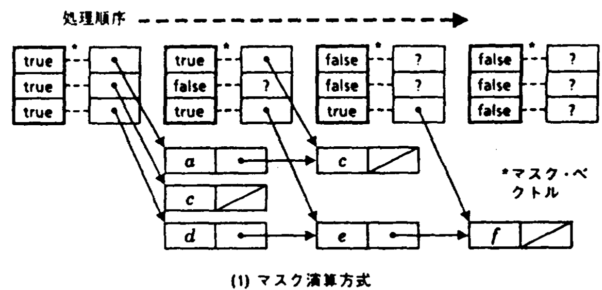
3.2 節の説明においては，簡単のためにベクトルの要素である各リストの長さを等しくした．しかし，リストは一般に可変長であるから，各リストの長さがことなる場合でも変換後のプログラムがただしく動作するようにしなければならない．

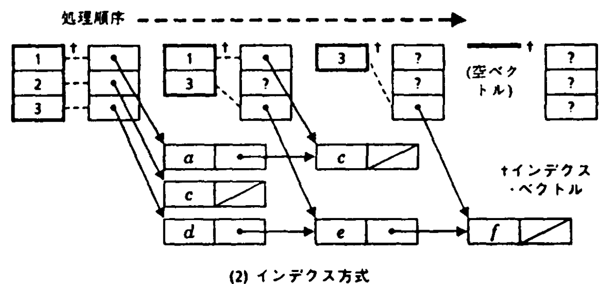
図2(1.2)の外側のくりかえしにおいて，長さのことなるリストをあつかう場合，すべてのベクトル要素について一定回数 n 回の処理をおこなうとすれば，つぎのような不都合が生じる．第1に，長さが n をこえるリストについては処理されない要素がのこる．第2に，長さが n 未満のリストについては空リスト (nil) を分解しようとして，あやまった処理がおこなわれる．

このような不都合をさけ，ちょうど必要なだけの回数の処理をおこなうようにするためには，ベクトル計算機に用意されている3種類の条件制御機構を使えばよい．それらは，マスク演算処理機構，リスト・ベクトル処理機構，圧縮・伸長処理機構である[9]．いずれの条件制御機構をおもに使用するかによって，可変長リストをあつかう方法もつぎの3種類にわけられる．

（1） マスク演算方式，
（2） インデクス方式，
（3） 圧縮方式．

これらの方法を図4を使って説明する．操作すべきリストへのポインタを要素とするベクトルをデータ・ベクトルとよぶ．





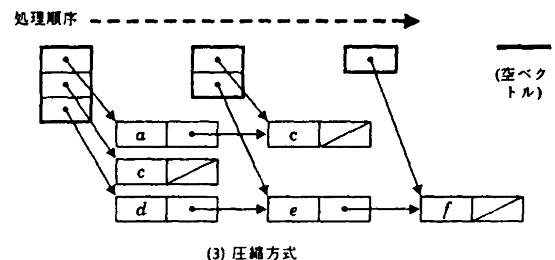


図 4 3条件制御方式によるリストのアクセス方法
Fig. 4 The access methods for lists by the three conditional control methods.

図4(1)はマスク演算方式によるリストのアクセス方法を示す．処理中につくられる全ベクトルのベクトル長は等しい．この方法では各データ・ベクトルの要素が有効かどうかをあらわす論理値をふくむベクトルを使用する．このベクトルをマスク・ベクトルとよぶ．最初はマスク・ベクトルの全要素の値を true とする．処理がすすむとデータ・ベクトルの第 i 要素に関する処理がリストの末尾に達するので，マスク・ベクトルの第 i 要素の値を false とする．マスク・ベクトルの対応する要素の値が true であるようなデータ・ベクトル要素に関してだけ処理をおこなう．処理がすすむにつれてマスク・ベクトル中には false を値とする要素が増加する．マスク・ベクトルの全要素の値が false になったとき，処理を終了する．

図4(2)はインデクス方式によるリストのアクセス方法を示す．この方式では，各データ・ベクトルを直接にアクセスするのではなく，データ・ベクトルの有効要素のインデクス（または変位）をふくむベクトルをつうじてアクセスする．このベクトルをインデクス・ベクトルとよぶ．インデクス・ベクトルから指示される要素だけが処理対象となる．すべてのデータ・ベクトルのベクトル長は等しいが，インデクス・ベクトルからは処理が終了した要素が除去されていく．したがって，最初は両者のベクトル長は等しいが，しだいにインデクス・ベクトルのベクトル長は短縮する．そのベクトル長が0になったとき，処理を終了する．

図4(3)は圧縮方式によるリストのアクセス方法を示す．この方式では，データ・ベクトルはつねに有効要素だけをふくむ．そのために，処理が終了した要素が生じるたびに，その要素はデータ・ベクトルから除去されていく．

圧縮方式にくらべると他の2方式は複雑にみえるが，金田[10]でのべているように，つぎのような利点がある．図4(3)においては各ステップでただ1つのデータ・ベクトルが使われているが，一般には複数のデータ・ベクトルが使われる．このような場合，圧縮方式による実行においては，全データ・ベクトルをいっせいに圧縮しなければならない．これに対して，マスク演算方式においては圧縮の必要はなく，また，インデクス方式においては1個のインデクス・ベクトルだけを圧縮すればよい．なぜなら，1個のマスク・ベクトル，また1個のインデクス・ベクトルによって，すべてのデータ・ベクトルの条件制御ができるからである．

上記の戦略はプログラム変換の方針をあたえているだけであり，具体的な変換方法はくりかえし構造がループ，再帰よびだし，バックトラックのいずれによって形成されているか，などによってことなり，それぞれ具体的な変換方法をみいだす必要がある．プログラム中に二重以上のくりかえし構造があることがこれらのプログラム変換を適用するための必要条件だが，それは十分条件ではないから，それをみたすすべてのプログラムがベクトル化可能な構造に変換可能なわけではない．たとえば，部分的に共有された複数のリストの一部をかきかえる処理などは，プログラム変換できない．しかし，多くのリスト処理のプログラムは多重のくりかえし構造をもっていて，外側のくりかえしごとにことなるリストを処理するので，この戦略の適用範囲はかなりひろいと考えられる．またこの戦略は，可変長リストだけではなく他の可変長データ構造にも適用することができる．

4. リスト処理基本演算のベクトル処理法

第5章で，上記の戦略にもとづく具体的なプログラム変換例を示すが，それにさきだって，変換後のプログラムにおけるリスト処理のベクトル処理方法について説明する．リスト処理における基本的な演算として，データ型の判定，リストの分解，リストの合成があげられる．Lisp でいえば null, car, cons などの演算である．変換後のプログラムではデータ・ベクトルの各要素に対する基本演算をまとめておこなう．すなわち，最内側のくりかえしは複数のリストに関する基本演算のくりかえしになる．このループは，ベクトル計算機 S-810 などにおいてはベクトル処理可能である．

これらの基本演算とそのベクトル処理法について順に説明する．すべての条件制御方式についてのべると煩雑になるので，マスク演算方式についてだけのべる．ほかの条件制御方式も同様に実現可能である．

4.1 データ型判定

Lisp, Prolog などの記号処理用言語においては，各データは型（タグ）をともなっている．そして，各データの処理にさきだって，型を判定する必要がある．たとえば，リスト処理においては，通常，空リスト判定（空リストかどうかの判定）をおこないながらリストをたどる必要がある．空リスト判定をおこなう Lisp 関数 null を例にとり，図5(1)を使用して，データ型判定の機能とそのベクトル処理の方法を説明

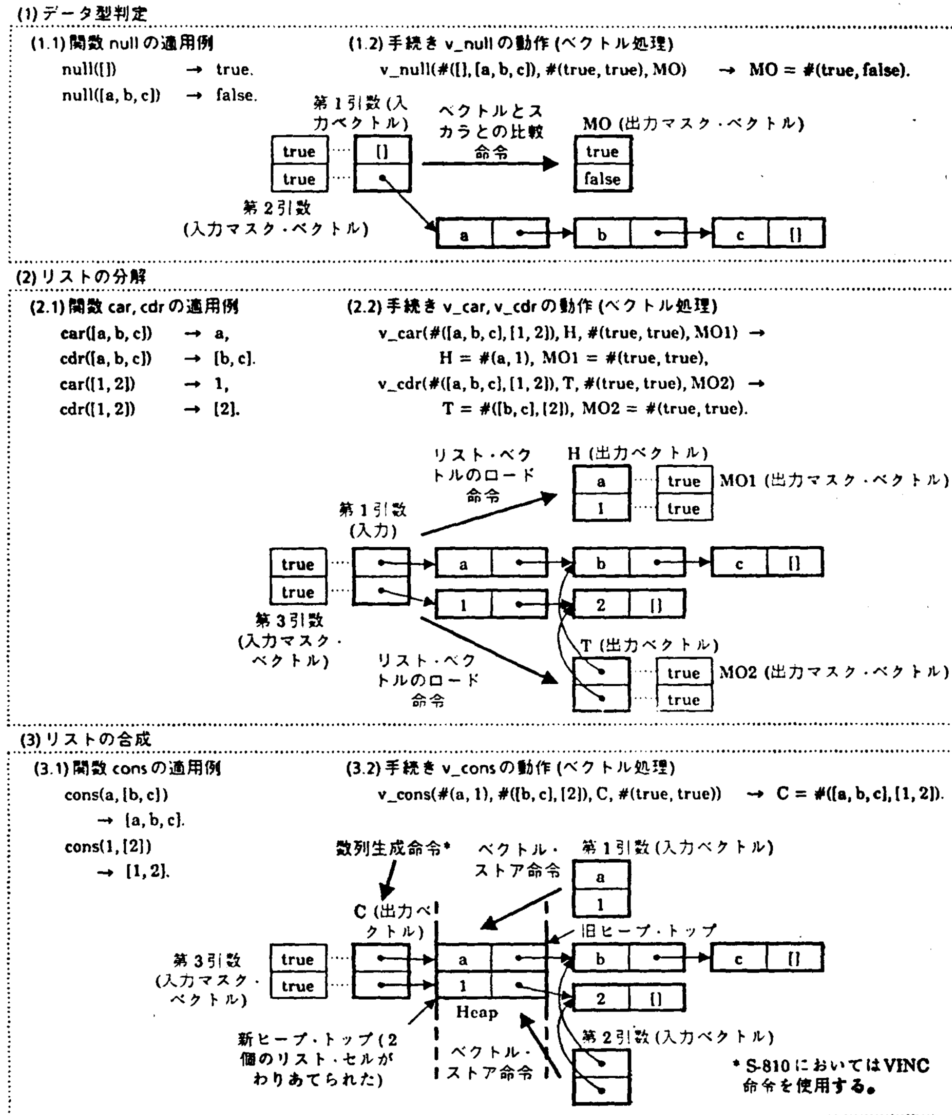


図 5 リスト処理基本演算とそのベクトル処理方法
Fig. 5 Basic operations for lists and their vector processing methods.

する．

関数 null は1個の引数をもち，引数が空リストなら true，そうでなければ false を結果とする．関数 null の適用の例を図5 (1.1) に示す．

複数のデータに関する空リスト判定をベクトル処理で実現するには，つぎのような機能をもつ手続き v_null を作成・使用すればよい．v_null (X, MI, MO) において，引数 X, MI は入力，引数 MO は出力であって，これらは要素数が等しいベクトルである．以下，データ・ベクトルVの第 i 要素を V[i] であらわす．MI, MO はマスク・ベクトルである．MI[i] が true ならば MO[i] に null(X[i]) ($1 \leq i \leq N$, N は要素数）の値を代入する．MI[i] が false ならば X[i] に関する比較はおこなわず，MO[i] も false とする．

リスト処理におけるデータ型判定として，ほかには，データがリストかそれ以外の型かを判定する関数 atom などが使われる．これらの演算も null と同様にベクトル処理できる．

データ型判定の実行後に，判定結果が true のときだけまたは false のときだけ実行可能なベクトル演算をデータ・ベクトルXの要素に対して実行する場合には，データ型判定において出力されるマスク・ベクト

ルをその演算においてマスク・ベクトルとして入力・使用する．たとえば，4.2 節でのべるリストの分解において，データ・ベクトルの要素のなかにリストとそれ以外のものとがまざっているときは，まず各要素がリストであるかどうかの判定をおこなって，その結果が true になった要素だけ分解をおこなう必要がある．

なお，ベクトルXの要素がつねにすべて演算対象であれば v_null の引数 MI は不要だが，v_null が条件制御のもとで実行される場合には MI が不可欠である．

v_null は，ベクトル計算機のもつベクトルとスカラとの比較命令を使うことによって，容易にベクトル命令で実現できる．例を図5 (1.2) に示す．

4.2 リストの分解

Lisp においてリストの要素をアクセスするには，あたえられたリストを分解する関数，すなわちその頭部をもとめる関数 car と尾部をもとめる関数 cdr とを使用する．Prolog においてはリストはユニフィケーションによって分解されるため，分解の機能は手続きというかたちで陽にはあらわれない．しかし，ユニフィケーションの実装のために関数 car, cdr に相当する機能が必要である．図5 (2) を使用して，これらの機能とそのベクトル処理の方法とを説明する．まず，関数 car, cdr の適用例を図5 (2.1) に示す．

複数のリスト分解をベクトル処理で実現するには，つぎのような機能をもつ手続き v_car, v_cdr を作成・使用すればよい．v_car (X, Y, MI, MO) において，引数 X, MI は入力，引数 Y, MO は出力であって，これらは要素数が等しいベクトルである．MI, MO はマスク・ベクトルである．MI[i] が true であってかつ X[i] が分解可能ならば Y[i]=car(X[i]) ($1\leqq i\leqq N$, N は要素数) である．MI[i] が false または X[i] が分解不能ならば X[i] に関する分解はおこなわず，MO[i] を false とする．関数よびだし v_cdr (X, Y, MI, MO) においても同様である．

v_car, v_cdr においては，データ・ベクトルから各要素の頭部または尾部をもとめて，それらを要素とするデータ・ベクトルをつくる．例を図5 (2.2) に示す．より具体的にいえば，つきのようになる．S-810 などのベクトル計算機は，ベクトル・インデクス（リスト・ベクトルとよばれている）つきのロード・ストア命令をもっている．ベクトル・インデクスつきロード命令の動作をC言語であらわすと，つぎのようになる．

```
for (i=1; i<n; i++) {
  vector1[i]=*(vector2[i]+vectorBase);
}
```

ベクトル化された Fortran プログラムにおいては，通常は vectorBase として配列の原点（先頭アドレス），vector2[i] としてその添字を指定して使用する．しかし，vector2[i] としてリスト・セルの先頭アドレス，vectorBase としてリスト・セルの先頭から頭部または尾部への変位を指定すれば，各リスト・セルの頭部または尾部を要素とするベクトルをもとめることができる．

4.3 リストの合成

Lisp においてリストを合成するには，関数 cons を使用する．Prolog においてはリストはユニフィケーションによって合成されるため，手続きというかたちで陽にはあらわれない．しかし，ユニフィケーションの実装のために関数 cons に相当する機能が必要である．図5 (3) を使用して，関数 cons の機能とそのベクトル処理方法を説明する．

関数よびだし cons (X, Y) は，リストYの先頭に要素Xをつけくわえたリストを結果とする．関数 cons の適用の例を図5 (3.1) に示す．

複数のリスト合成をベクトル処理で実現するには，つぎのような機能をもつ手続き v_cons を使えばよい．v_cons (X, Y, C, MI) において引数 X, Y, MI は入力，引数Cは出力であって，これらは要素数が等しいベクトルである．MI はマスク・ベクトルである．MI[i] が true ならば $C[i]=\mathrm{cons}(X[i], Y[i])$ ($1\leqq i\leqq N$, N は要素数) である．MI[i] が false ならば合成をおこなわない．図5 (3.2) に例を示す．

つぎのようにすれば，v_cons をベクトル命令で実行できる．まず，頭部とするべきデータを要素とするデータ・ベクトルHと，尾部とするべきデータを要素とするデータ・ベクトルTとを入力する．HとTとは要素数が等しい．そして，HとTの対応する要素の値を頭部および尾部の値とするリスト・セルをつくり，それらをさすポインタを要素とするデータ・ベクトルCをつくる（v_cons は1要素あたり3個のポインタを生成することに注意）．そしてCを結果とすればよい．リスト・セル長を Lc とし，リスト・セルをヒープに連続してわりあてるとすれば，ベクトルCの各要素のアドレス部は，リスト合成の実行前のヒープ・トップ・ポインタの値に 0, Lc, 2*Lc, … を加算してできた値をふくむ．このような値のベクトルは，

ベクトル計算機の数列生成命令（S-810 においては VINC 命令）を使用すれば高速に生成できる．

なお，MI[i] の要素のなかに true でないものがあるときには，true でない要素に対応するデータ・ベクトル要素に対してはリスト・セルをわりあてないようにするのが場所効率はよい．しかし，そうするとベクトルCのとなりあう要素のアドレスの差分が一定ではなくなるため，ベクトル計算機によってはベクトル化できなくなったり，ベクトル化できてもベクトルCの生成に要する時間が増加したりする．したがって，高速性を優先する場合には，MI[i] が true でない要素に対してもリスト・セルをわりあてるほうがよい．

5. エイト・クウィーンの Prolog プログラムへの適用

この章では，第2章でのべたプログラム変換の戦略を，Prolog で記述されたエイト・クウィーン問題のプログラムに適用した例についてのべる．5.1 節ではエイト・クウィーンのプログラムの一部である手続き not_take1 を例としてくりかえし構造の交換についてのべる．5.2 節では，エイト・クウィーンのプログラムの一部である手続き select を例としてくりかえし構造の一重化についてのべる．

5.1 くりかえし構造の交換

エイト・クウィーン問題をとく Prolog プログラムは，たとえば中島[11] で示されている．つぎに示す not_take1 はその一部であり，指定された1個のクウィーンがチェス・ボードの対角方向にすでにおかれたクウィーンと衝突するかどうかをしらべる手続きである．not_take1 のすべての引数は入力である．

（1） 原始プログラム

```
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).
```

第1引数はチェス・ボードをあらわすリストであり，第2～3引数はしらべるべき位置（行番号）をあらわす整数である．

not_take1 はその末尾が再帰よびだしになっていて，各くりかえしにおいて，チェス・ボードをあらわすリストの要素を1個ずつしらべていく．エイト・クウィーンの場合には，このリストの平均長は4以下と短い．したがって，この再帰よびだしをそのままループに変換したとしてもベクトル計算機の性能をいかすことはできない[12]．しかし，not_take1 は，その外部で発生する深いバックトラックによって，その全体がくりかえし実行される．これらの実行のあいだにはデータ依存関係がないので，このくりかえしを最内側のくりかえしとするようにプログラム変換すれば，ベクトル処理が可能になる．

変換後の手続き v_not_take1 を示す．

（2） ベクトル化後のプログラム

```
v_not_take1(_, _, _, MI, MI):-
    v_finished(MI), !.
v_not_take1(B, Qa, Qs, MI, MO):-
    v_null(B, MI, MO1),
    v_car(B, Q, MI, M1), v_cdr(B, R, 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_end_or(MO1, MO2, MO).
```

v_not_take1 の第1～3引数は not_take1 の第1～3引数に対応している．not_take1 はバックトラックによってくりかえしよびだされるが，各回における not_take1 の第1引数の値を要素とするデータ・ベクトルを v_not_take1 の第1引数とする．第2～第3引数も同様である．エイト・クウィーンのプログラムにおける not_take1 以前に実行される部分は，このインタフェースをみたすようにプログラム変換しなければならない．

変換後のプログラムにおける条件制御の方式として3種類があるが，いずれの場合もプログラムの基本構造はかわらないので，ここではマスク演算方式によるプログラムだけを示す．v_not_take1 の第4～第5引数は入力および出力のマスク・ベクトルである．引数であるすべてのベクトルの要素数は等しい．

（1）から（2）への変換の手順については金田ら[13]でのべているので，ここでは省略する．（1），（2）の各部分の対応を**表1**に示す．手続き v_null, v_car, v_cdr の機能は第3章でのべたとおりである．他の手続きの機能については表1を参照されたい．

すでにのべたように not_take1 の末尾再帰よびだしはリストの要素に関するくりかえしである．v_not_take1 の再帰よびだしは末尾再帰ではないが，データ・ベクトルBの要素であるリストの要素に関するく

表 1 手続き not_take1 における，原始プログラムとベクトル化後のプログラムとの対応
Table 1 The correspondence between the source program and the vectorized one for not_take1 procedure.

原始プログラム	ベクトル化後のプログラム	意　　味
_*	v_finished(MI)	手続きを実行するかどうか（再帰を停止させるかどうか）を判定する.
[ ]	v_null (B, MI, MO1)	ベクトルBの要素が空リストかどうかを判定する.
[Q|R]	v_car (B, Q, MI, M1), v_cdr (B, R, M1, M2)	ベクトルの要素であるリストを頭部と尾部とに分解する.
Q=\=Qa, Q=\=Qs	'v_=\='(Q, Qa, M2, M3), 'v_=\='(Q, Qs, M3, M4)	整数Qと Qa, Q と Qs とが等しくないかどうかをしらべる.
Qaa is Qa+1	'vs_+'(Qa, 1, Qaa, M4)	ベクトルの各要素とスカラ・データとの整数加算をする.
Qss is Qs+1	'vs_−'(Qs, 1, Qss, M4)	ベクトルの各要素とスカラ・データとの整数減算をする.
not_take1(R, Qaa, Qss)	v_not_take1(R, Qaa, Qss, M4, MO2)	再帰よびだしをする.
_*	v_end_or(MO1, MO2, MO)	原始プログラム第1節に対応する部分と第2節に対応する部分から出力されるマスクを合成する.

* 原始プログラムには対応する部分がない.

りかえしであり，not_take1 の末尾再帰よびだしと対応している．しかし，v_not_take1 においては，それがよびだす手続き v_null, v_car, v_cdr などの内部にループがあり，これらは not_take1 におけるバックトラックで形成されるくりかえしに対応している．したがって，not_take1 から v_not_take1 への変換においてくりかえし構造を交換しているということができる．

（2）のプログラムにつぎのような入力をあたえたときの実行過程を図 6 に示す．ただし，ここで #(e_1, e_2) は e_1, e_2 を要素とするベクトルをあらわす．

第 1 引数：#([2, 4, 1], [4, 1, 3]).

第 2 引数：#(4, 3).

第 3 引数：#(2, 1).

第 4 引数：#(true, true).

上記の入力をあたえて v_not_take1 をよびだすことは，つぎのような 2 つの手続きよびだしを実行すること

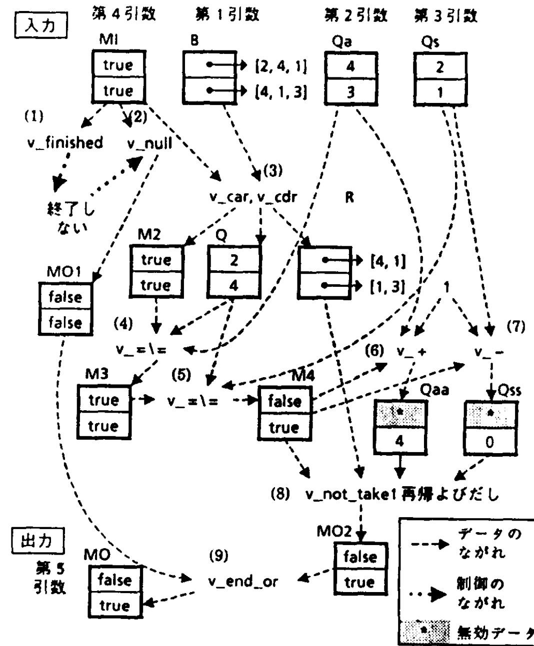


図 6 手続き v_not_take1 の実行例
Fig. 6 An execution example of procedure v_not_take1.

とほぼ等価である．

?- not_take1 ([2, 4, 1], 4, 2). (5.1)

?- not_take1 ([4, 1, 3], 3, 1). (5.2)

(5.1)を実行すると失敗し(5.2)を実行すると成功する．それに対応して，図 6 の実行の結果は #(false, true) となる．実行過程については金田[2] でよりくわしくのべている．

5.2 くりかえし構造の一重化

手続き select はリストを入力して，そのリストから 1 要素をえらびだした結果と，そのリストからその要素をのぞいたリストとを出力する手続きである．入力されるリストの長さを n とすれば，解の数も n 個だけある．手続き select またはそれと等価な手続き（しばしば delete という名称をあたえられる）は Prolog プログラムにおいてしばしば使われるが，エイト・クウィーン問題のプログラムにおいても，クウィーンのリストから 1 個のクウィーンをえらぶのに使われる．

select のプログラムを示す．

（1） 原始プログラム

```
select ([A|L], A, L).
select ([A|L], X, [A|L1]):-
   select (L, X, L1).
```

第1引数が入力のリスト，第2引数がえらばれた要素，第3引数がのこりの要素のリストである．

select にはその末尾に再帰よびだしがあって，その各くりかえしにおいて第1節から1個ずつ解を出力し，select のよびだしもとに復帰する．ただし，select の実行後にバックトラックが生じなければ第2節は起動されず，したがって再帰よびだしも起動されない．

すべての解が出力されるまでバックトラックがくりかえされることを前提とすれば，つぎのようなプログラム変換が可能になる．すなわち，1個の解がもとめられるごとに解を出力するかわりに，ベクトルを用意してそれに解を蓄積していく．すべての解がもとめられたところで，そのベクトルを出力する．この変換後のプログラムの機能は，Prolog の手続き bagof の機能に似ている．このようにことなる解を要素とするベクトルを使うインタフェースは，5.1 節の手続き v_not_take1 のそれと一致している．

変換後の手続き v_select のプログラムを示す．

（2） ベクトル化後のプログラム

```
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).
v_select_1 (_, [ ], [ ], MI, [ ]):-
  v_finished (MI), !.
v_select_1 (AL, [A'|X1L], [L'|Y1L],
          MI, [M1'|ML]):-
  v_car (AL, A', MI, MO'),
  v_cdr (AL, L', MO', M1'),
  v_car (AL, A, MI, MO),
  v_cdr (AL, L, MO, M1),
  v_select_1 (L, X1L, L1L, M1, ML1),
  mapcar (v_cons (A), L1L, Y1L, ML1, ML).
```

v_select の第1～3引数は select の第1～3引数に対応している．v_select の入力時にも，出力時と同様のベクトル・インタフェースをとる．すなわち，v_select の実行開始以前に複数の解をもつ手続きが実行されていれば，それらを要素とするベクトルが第1引数として入力される．入力する解が1個の場合も，唯一の要素をもつベクトルが第1引数として入力される．

v_select の第4引数は入力マスク・ベクトルである．すなわち，第1引数の各要素の有効性が第4引数によって示される．しかし，第4章の各手続きや手続き v_not_take1 とはちがって，第1，第4引数と第2～3引数の要素数は等しくなく，要素は対応しない．出力である第2～3引数の要素はすべて有効である．したがって，マスク・ベクトルは出力しない．

v_select の各出力ベクトルの要素は，各入力ベクトルの要素とは対応がくずれるため，対応をとるべきベクトルを第5引数として入力し，その要素を複写・対応づけしてえられたベクトル BO を第6引数として出力する．したがって，BO の要素数は第2～3引数の要素数と等しい．エイト・クウィーンのプログラムにおいては要素対応をとるべきベクトルが1個だけなので対応づけのための引数は1組だけだが，プログラ

表2 手続き select における，原始プログラムとベクトル化後のプログラムとの対応
Table 2 The correspondence between the source program and the vectorized one for non-deterministic select procedure.

原始プログラム	ベクトル化後のプログラム	意　味
_*	v_select_1(_, [ ], [ ], MI, [ ])	空のマルチ・ベクトルを生成する．
_*	v_finished(MI)	手続きを実行するかどうか（再帰を停止させるかどうか）を判定する．
_*	v_select_1(AL, [A'|X1L], [L'|Y1L], MI, [M1'|ML])	各マルチ・ベクトルに要素を追加する．
[A'|L']*1	v_car(AL, A', MI, M0'), v_cdr(AL, L', M0', M1')	ベクトルの要素であるリストを頭部と尾部とに分解する．
[A|L]*2	v_car(AL, A, MI, M0), v_cdr(AL, L, M0, M1)	ベクトルの要素であるリストを頭部と尾部とに分解する．
select (L, X, L, 1)	v_select_1(AL, X1L, L1L, M1, ML1)	再帰よびだしをする．
[A|L1]	mapcar(v_cons(A), L1L, Y1L, ML1, ML)	マルチ・ベクトル L1L の各部分ベクトルの要素に関してリストを合成する．
_*	v_merge([X1L, Y1L], [X, Y], [BI], [BO], ML)	マルチ・ベクトル X1L, Y1L をそれぞれ1個のベクトル X，Y に併合し，ベクトル BI の各要素を X，Y の要素と対応づけて BO を生成する．

* 原始プログラムには対応する部分がない．
*1 原始プログラム第1節に対応している．
*2 原始プログラム第2節に対応している．

ムによっては2組以上の引数が必要になる．手続き v_merge はこれらの引数をリストとして入出力するため，その場合でも使用することができる．

（1）から（2）への変換の手順は省略するが，その対応関係を表2に示す．

v_select の実行においては，まず v_select_1 をよびだす．v_select_1 の再帰よびだしごとに，その第2，3，5仮引数において，もとめた解ベクトルをリストの要素として蓄積する．v_select_1 からの復帰後に，v_merge においてこれらのリストの要素をそれぞれ1つのベクトル長の長いベクトルに格納しなおし，出力する．v_select_1 の定義にあらわれる mapcar (v_cons (A), L1L, Y1L, ML1, ML) は，Aを第1引数とし，リスト L1L, Y1L, ML1, ML のそれぞれの各要素であるベクトルを第2～5引数として手続き v_cons をくりかえし実行する．

すでにのべたように select は再帰よびだし1回ごとに解を出力するので，select のよびだし以降に実行されるプログラム部分はくりかえし実行される．したがって，not_take1 と同様の方法でベクトル化すれば，ベクトル化された select のよびだし以降に実行される部分はやはりくりかえし実行されることになる．それに対して v_select においては，もとめた解を蓄積することによってこのくりかえしをなくしている．したがって，v_select のよびだし以降に実行される部分のくりかえしを一重化しているということができる．

（2）のプログラムにつぎのような入力をあたえたときの実行過程を図7に示す．

第1引数：#([2, 4], [1, 3])．

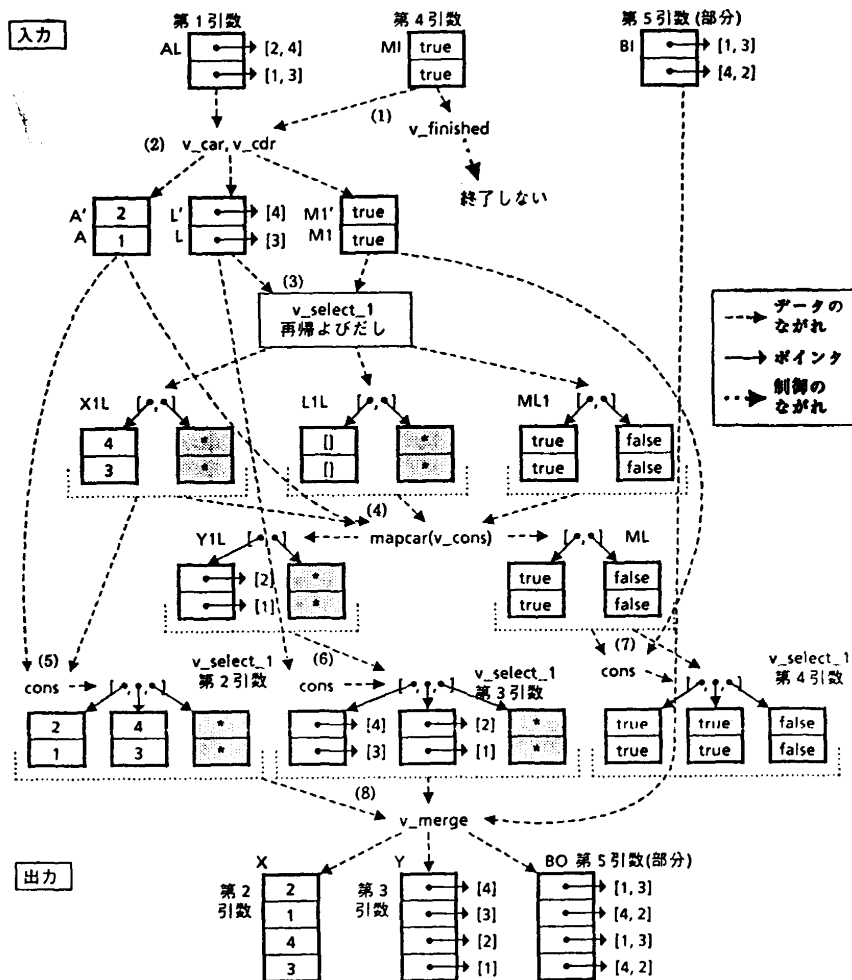


図 7 手続き v_select, v_select_1 の実行例

Fig. 7 An execution example of procedures v_select and v_select_1.

第4引数：#(true, true).

第5引数：#([1, 3], [4, 2]).

上記の入力をあたえて v_select をよびだせば，つぎのような2つの手続きよびだしを実行することとほぼ等価である．

?- select ([2, 4], X, Y). (5.3)

?- select ([1, 3], X, Y). (5.4)

(5.3)を実行するとつぎの(5.5)，(5.6)のような解がえられる．また，(5.4)を実行すると(5.7)，(5.8)のような解がえられる．

X=2, Y=[4]. (5.5)

X=4, Y=[2]. (5.6)

X=1, Y=[3]. (5.7)

X=3, Y=[1]. (5.8)

したがって，図7においては，(5.5)～(5.8)をベクトルに格納した形の解がえられている．

6. 評価

第4章で示した方法にしたがってエイト・クウィーン問題の Prolog プログラムを手動でベクトル化し，さらに手動で Fortran と Pascal とによるプログラムに変換したあとコンパイルした．Fortran 部分はベクトル・コンパイラでコンパイルした．Fortran コンパイラのベクトル化オプションをかえることによってベクトル処理用のコードとスカラ処理用のコードとを生成して，両者をベクトル計算機 S-810 で実行した．

実行時間を測定した結果を表3に示す．第5章で示したマスク演算方式だけではなく，3つの条件制御方式のそれぞれをおもに使用するプログラムをコーディングして測定した（各プログラムのなかで複数の方式をまぜて使用しているが，これは，1つの方式だけでは実現不可能な場合があるなどの理由による）．プログラム全体としては，ベクトル処理速度はスカラ処理速度の8～9倍となっている．

表3 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

Table 3 Performance of the Eight-Queens program hand-compiled for the Hitachi S-810.

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

表4 エイト・クウィーン問題のプログラムにおける演算種別ごとの加速率

Table 4 Acceleration rate of each kind of operations in the Eight-Queens program.

プログラムの版（主要な条件制御方式）	v_car, v_cdr, v_null における加速率	v_cons における加速率
マスク演算版	12.1	3.1
インデクス版	13.6	3.3
圧縮版	13.7	3.1

エイト・クウィーン問題のプログラムの各部分ごとの実行時間を測定することによって，リスト処理基本演算の加速率すなわちスカラ処理時間とベクトル処理時間の比をもとめ，表4に示した．v_car, v_cdr および v_null の加速率と，v_cons の加速率とを示している．前3者のそれぞれの時間を個別に示していないのは，測定したプログラムにおいては，高速化のためにこれらが1個のループのなかに共存していて，個別に測定することが困難だったためである．

前3者の加速率は10倍をこえていて，満足すべき加速率だと考えられる．しかし，v_cons は3倍程度であり，十分な加速率とはいえない．加速率がひくいおもな原因は，v_cons が1要素あたり3つのポインタのストアを必要とするのに対して，S-810 のストアのスループットが十分でないことだと考えられる．これは，現在のスーパ・コンピュータが数値計算むきに最適化されていることの結果だといえる．

7. 結論

ベクトル計算機を使用し，「くりかえし構造の交換」，「くりかえし構造の一重化」というプログラム変換にもとづいて，リスト処理を高速に実行するための戦略を提案した．この方式をエイト・クウィーンの Prolog プログラムに適用して，ベクトル計算機 S-810 においてスカラ処理の約9倍の実行速度をえた．かぎられた例題に適用しただけではあるが，この結果はこれらのプログラム変換方式の有効性を示すものと考えられる．今後，ほかのプログラムに適用範囲をひろげていきたい．

謝辞 この研究をすすめる過程でしばしば議論をしていただいた日立製作所中央研究所の小島啓二研究員および安村通晃主任研究員，この研究を支援していただいた同ソフトウェア工場 AI 応用プログラム部の高橋栄部長，同中央研究所第8部の吉住誠一主任研究員

ほかの方々に感謝します.

参考文献

1) 板井修一：並列計算機におけるスケジューリングと負荷分散, 情報処理, Vol. 27, No. 9, pp. 1031-1038 (1986).
2) 金田　泰：ベクトル計算機による論理型言語の高速実行をめざして—各種 OR ベクトル実行方式の実現と検討—, 情報処理学会プログラミング言語研究会, PL-87-12 (1987).
3) Shapiro, E.: Systoric Programming: A Paradigm of Parallel Processing, *Proc. Fifth Generation Computer Systems '84* (1984).
4) 安村通晃：ベクトル化とプログラム変換, 知識情報処理シリーズ7「プログラム変換」, pp. 121-134, 共立出版 (1987).
5) Kuck, D. J., Kuhn, R. H., Padua, D. H., Leasure, B. and Wolfe, M.: Dependence Graphs and Compiler Optimizations, *Proc. 8th ACM Symposium on Principles of Programming Languages*, pp. 207-218 (1981).
6) Allen, J. R. and Kennedy, K.: Automatic Loop Interchange, *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 19, No. 6, pp. 233-246 (1984).
7) Wolfe, M.: Advanced Loop Interchanging, *Proc. of the '86 International Conference on Parallel Processing*, pp. 536-543 (1986).
8) 津田孝夫, 国枝義敏, 二宮正和, 栗屋　徹：ループ間にまたがるデータ参照関係をもつ多重ループの自動ベクトル化, 情報処理学会論文誌, Vol. 26, No. 3, pp. 536-544 (1985).
9) Kamiya, S., Isobe, F., Takashima, H. and Takiuchi, M.: Practical Vectorization Techniques for the "FACOM VP", *Information Processing '83*, pp. 389-394 (1983).
10) 金田　泰：スーパー・コンピュータによる Prolog の高速実行, 第26回プログラミング・シンポジウム報告集, pp. 47-56 (1985).
11) 中島秀之：Prolog, 産業図書 (1983).
12) 金田　泰, 小島啓二, 菅谷正弘：ベクトル計算機のための探索問題の計算法「並列バックトラック計算法」, 情報処理学会論文誌, Vol. 29, No. 10, pp. 985-994 (1988).
13) Kanada, Y., Kojima, K. and Sugaya, M.: Vectorization Techniques for Prolog, *1988 ACM International Conference on Supercomputing* (1988).

(昭和63年10月13日受付)
(平成元年4月11日採録)

金田　泰（正会員）
1956年生. 1979年東京大学工学部計数工学科卒業. 1981年同大学院情報工学専門課程修了. 同年4月から（株）日立製作所中央研究所第8部. 入社後 Fortran コンパイラ開発に従事し, 現在はスーパ・コンピュータの論理型言語処理系の研究に従事. 1985年山内奨励賞受賞. プログラミング言語とその処理系に興味をもつ. ACM, ソフトウェア科学会各会員.

菅谷　正弘（正会員）
1958年生. 1982年電気通信大学電気通信学部機械工学科卒業. 1984年同大学大学院電気通信学研究科機械工学専攻修士課程修了. 同年4月（株）日立製作所入社. 中央研究所第8部勤務. 入社後, 並列推論マシンの研究に従事. 現在, スーパ・コンピュータの論理型言語処理系の研究に従事.