拡張可能配列の一実現方式と Pascal 言語への組込み†

都司達夫†† 井口雅彦††† 渡辺勝正††

プログラム実行時に配列容量が飽和した際，動的にその大きさを拡張できる配列は拡張可能配列 (extendible array) と言われる．動的配列の場合にはまったく新たな配列領域が確保されるのに対して，拡張可能配列の場合には拡張以前の配列領域はそのまま使用され，したがって蓄積データは失われることはない．この利点のために拡張可能配列の応用範囲は非常に広いと考えられる．拡張可能配列を実現するときには配列要素の再配置をすることなくすべての次元方向に拡張可能であることが要求される．拡張可能配列の実現の難しさは，配列の任意の次元方向に拡張可能であることと，メモリの利用率および配列要素のアクセス速度を低下させないアドレス関数を構成することとが互いに背反し一般には両立しないことにある．本論文では，"index-array" のモデルに基づいた，新たな拡張可能配列の実現モデルを設定する．そこでは，拡張の経歴を記録することにより動的メモリ割付け機能とのインタフェースを新たに考慮している．本実現モデルに従って現実のプログラミング言語 (Pascal) に拡張可能配列を支援させるための方式を提案した．また，本方式の採用により整合配列機能が拡張可能配列として自然な形で実現できることを示した．さらに，実現した処理系について配列要素のアクセス速度を実測し，配列の長所である迅速なランダムアクセスをそれほど損なわないことを確認した．

1. まえがき

プログラム実行時に配列容量が飽和した際，動的にその大きさを拡張できる配列は拡張可能配列 (extendible array) と言われる．動的配列が，割付けの要求があった時点で配列全体が新たに確保されるのに対して，拡張可能配列の場合には，それ以前に割り付けられた記憶領域はそのまま使用され，拡張される部分が新たに動的に割り付けられる．したがって拡張以前の蓄積データは失われることはない．この利点のために，あらかじめ配列の必要サイズが予測し難いような種々の応用において非常に有効である．拡張可能配列を実現するときには配列要素の再配置をすることなくすべての次元方向に拡張可能であることが要求される．拡張可能配列の実現の難しさは，配列の任意の次元方向に拡張可能であることと，メモリの利用率および配列要素のアクセス速度を低下させないアドレス関数を構成することとが互いに背反し一般には両立しないことにある．現在まで拡張可能配列の実現については種々の理論的な研究が行われてきたが[1)~3)]，これらはいずれも抽象的なレベルのものであり，現実のプログラミング言語に組み込むことを前提として，処理系レベル，生成コードレベルで妥当な効率を発揮するような方法を示した例は見当たらない．動的に大きさを変更できる配列を支援しているプログラミング言語としては CLU[4)] があるが，CLU における配列型 v はその成分の型がさらに配列型 v' であるとき，v' は可変であり得，したがって，v は通常の意味での配列とは言えない．ここで "通常の意味" とは，配列成分の占める記憶領域の大きさが一定であり，したがって簡単な計算で配列成分にランダムにアクセスできることを言う．

言語レベルで拡張可能配列としての機能を十分発揮させ，しかも妥当なメモリの利用率と実行速度を保証するような実現方式を見いだすためには，従来の実現モデルに対して，動的メモリ割付けの機能などの現実の実行環境との関わりを考慮した新たなモデルを設定するとともに，組み込むべき言語の仕様とその処理系のデータ構造取扱い方法に調和した方式を考える必要があると思われる．

本論文では文献 3) で提案されている index-array による実現モデルに対して，拡張の経歴を記録することにより動的メモリ割付け機能とのインタフェースを考慮した新たな実現モデルを設定し，それに基づいて現実のプログラミング言語(ここでは Pascal 言語)に拡張可能配列を組み込むための方式を提案する[8)]．また，"整合配列 (conformant array) 引数*の欠如" は

† An Implementation Scheme of Extendible Arrays and Its Application to Pascal Language by TATSUO TSUJI (Department of Information Science, Faculty of Engineering, Fukui University), MASAHIKO IGUCHI (Hokuriku Nichiden Software Co., Ltd.) and KATSUMASA WATANABE (Department of Information Science, Faculty of Engineering, Fukui University).

†† 福井大学工学部情報工学科
††† 北陸日電ソフトウエア(株)

* 標準 Pascal では，引数として配列を手続きに受け渡す場合，実引数の添字の型が対応する仮引数の添字の型と異なることは許されない．すなわち，仮引数とまったく "同型" の配列しか受け渡せない．Pascal 固有のこのような強い型づけによる制限を緩和して，『実引数の添字の型が仮引数のそれに整合" すれば，手続きに受渡し可能である』としたとき，この仮引数を整合配列仮引数と言う．

Pascal に対する批判の一つであるが，本方式の採用により整合配列が拡張可能配列として自然な形で実現できることを示す．

2. 拡張可能配列のモデル

ここでは，まず，本論文で提案する拡張可能配列のモデルを Pascal 言語風に定義した後，拡張可能配列の論理構造について議論する．ここでの定義はそのまま Pascal 言語における拡張可能配列に関連する構文を定義している．

[定義 1] n 次元拡張可能配列型 EA は次のように宣言される．

type EA = array [$I_1, \cdots, I_n$] of E;

ここで，I_k は k 次元目の添字の型であり，任意のスカラ型である．ただし，整数型と実数型を除く．I_k を，特に $a \ll b$ (a, b は定数) と書いた場合または $a \ll b$ を定義している型名の場合には EA は次元 k について拡張可能であると言う．a は拡張の下限，b は上限を表す．$a \ll b$ なる表記は Pascal における部分範囲型 $a..b$ と等価な意味を持つ．拡張可能配列型は少なくとも一つの次元について拡張可能でなければならない．拡張可能な次元を含まないときには通常の（固定）配列型の定義になる．E は要素の型を表し，拡張可能配列型を除く任意の型である．

[定義 2] EA を n 次元拡張可能配列型 array [$I_1, \cdots, I_n$] of E とするとき，型 EA をもつ変数 v は次のように宣言される．

var v: $EA(I'_1, \cdots, I'_n)$;

EA が次元 k について拡張可能であるときには I'_k は I_k と同じ親のスカラ型をもつ部分範囲型であり，I'_k は通常の表記どおりに，例えば $l..h$ (l, h は定数) と書く．I_k が $a \ll b$ とすると ord (a) ≤ ord (l) ≤ ord (h) ≤ ord (b) でなければならない．ここで ord (x) は Pascal における標準関数で x の順序数を値として返す．EA が次元 k について拡張可能でないときには I'_k は # と書き I_k の範囲をそのまま受け継ぐ．($I'_1, \cdots, I'_n$) で規定される拡張可能配列 v の部分を v の初期配列と言う．

定義 1，定義 2 による 2 次元の拡張可能配列の一般的なモデルを**図 1** (b) に示しておく．

[例 1] type week=(sun, mon, tue, wed, thu, fri, sat);

eaw = array [1..100, sun≪sat] of integer;

により次元 2 について拡張可能な 2 次元拡張可能配列型 *eaw* が宣言される．また，

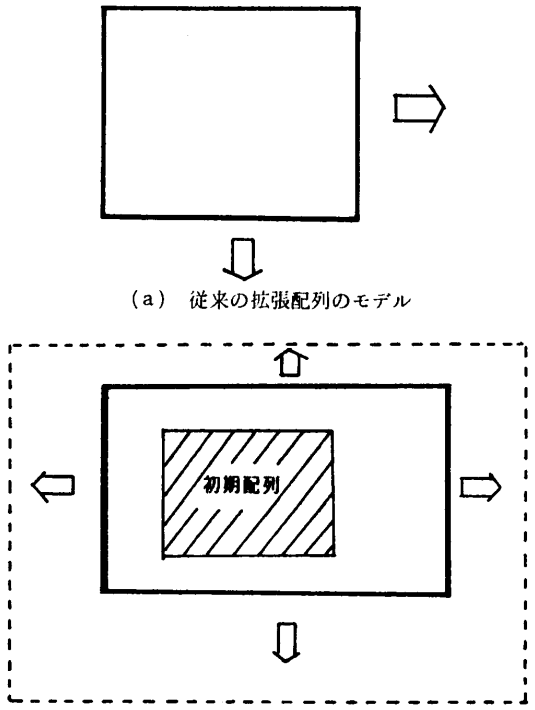
var *vw*: *eaw*(#, tue..thu);

により型が *eaw* で初期配列の形が (1..100, tue..thu) である拡張可能配列型変数 *vw* が宣言される．

[例 2] type *eac* = array ['*a*'≪'*z*', −1000≪2000] of boolean;

var *vc*: *eac*('*a*'..'*k*', −100..200);

により 2 つの次元ともに拡張可能な拡張可能配列型 *eac* と型が *eac* で初期配列の形が ('*a*'..'*k*', −100..200) である変数 *vc* が宣言される．*vc* の概念図を**図 2** に示す．



(a) 従来の拡張配列のモデル

(b) 本論文における拡張可能配列のモデル

図 1 拡張可能配列のモデル
Fig. 1 Models of an extendible array.

従来の拡張可能配列のモデル[1)~3)]は図 1 (a) のように図示される．以下に，本論文で提案する拡張可能配列のモデルと従来のモデルの相違をまとめて説明しておく．

(i) 従来のモデルでは初期配列の考え方はなく大きさ 0 から拡張される．初期配列の考え方を持ち込むことは，論理的に妥当であり，つぎの 3 章で述べるようにメモリの利用率の点でも有利である．

(ii) 提案モデルでは図 1 (b) のように添字の下限方向の拡張についても考慮している．これにより，(C 言語等とは違って) Pascal 言語のように配列の添字

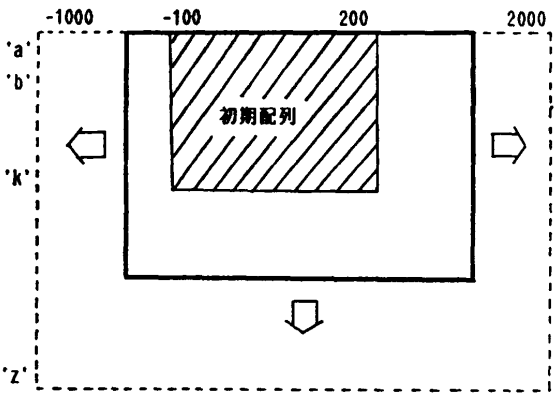


図 2 拡張可能配列変数 *vc* の概念図
Fig. 2 Illustration of an extendible array.

の型が一般に部分範囲型である言語に対して，添字の下限方向の拡張も支援できるようになる．

(ⅲ) 提案モデルの場合には，拡張の上下限が存在し指定する必要がある．これは応用範囲を狭めるものではなく現実的には妥当な制限と言えよう．ユーザは必要と予想されるサイズの配列を初期配列として指定する．

定義1で与えた宣言の形は Pascal における多次元配列の省略形（多次元形）に対応する．これを本来の定義として例えば，例1を

type *eaw* = array [1..100] of array [sun≪sat] of integer;

とした場合，*eaw* は拡張可能配列型

array [sun≪sat] of integer;

を成分にもつ（固定）配列型を意味することになる．この場合，配列の成分サイズが一定しないことになり，通常の意味での配列の必要条件を満たさない．したがって，通常の配列のように効率良いメモリ割付けと迅速な成分のランダムアクセスを実現することが困難である．場合によってはリストを多用せざるを得ないことになり，大容量の配列に対しては実用的でないことも考えられる．このような理由から拡張可能配列型の宣言は定義1のように多次元形の定義のみ許し，要素の型として拡張可能配列型を除外している．また，構造型（通常の固定配列型を含めて）の成分として拡張可能配列型を指定することも許されない．

ここで与えた拡張可能配列とよく似たものとして Ada における無制約配列[5]があるが，無制約配列の場合には，拡張可能ではなく，変数宣言時に指定される添字の範囲に従ってその大きさは固定になることと，すべての次元方向に無制約でなければならないことなどの点で拡張可能配列とは本質的に異なる．

なお，拡張可能配列の拡張は新たに用意された標準手続き extend によりプログラムにおいて明示的に指示するものとする．この extend を含めて，拡張可能配列に対する諸操作を支援するためにつぎのような3つの標準手続き，標準関数を増設した．

(標準手続き extend (var v: EA; l_1, h_1: I_1; …; l_n, h_n: I_n)) n次元拡張可能配列の大きさを一度に ($l_1..h_1, \cdots, l_n..h_n$) に拡張する．ここで，$I_1, \cdots, I_n$ は拡張可能配列型 EA の拡張限界を指定するスカラ型である．l_i や h_i と書かずに # と書いたときにはその次元の大きさは現在の大きさのままである．

(標準関数 *lb* (var v: EA; i: integer), *ub* (var v: EA; i: integer)) *lb* は拡張可能配列 v の次元 i についての現在の大きさの下限を返す．*ub* は上限を返す．

3. 拡張可能配列の実現方式

例えば，変数 v: array [$a..b, c..d$] of E; を Pascal における通常の2次元配列型の変数として，$p(x, y) = \mathrm{ord}(y) - \mathrm{ord}(x)$ とする．配列要素 $v[i, j]$ (ord $(a) \leq \mathrm{ord}(i) \leq \mathrm{ord}(b)$, ord $(c) \leq \mathrm{ord}(j) \leq \mathrm{ord}(d)$) に対して，翻訳系 (compiler) は通常，よく知られているように例えばアドレス関数，

$$\mathcal{A}(i, j) = \mathrm{address}(A[a, c]) + \{p(a, i) + (p(a, b) + 1) \times p(c, j)\} \times \mathrm{size}(E)$$

によって，静的変数領域を割り付ける．ここで，address ($v[a, c]$) は配列の先頭要素 $v[a, c]$ に対するメモリ領域の先頭番地であり，また，size (E) は配列要素のサイズである．これらのアドレス関数を使用すれば，メモリの利用率は100%であり，迅速なランダムアクセスも可能であるが，プログラムの実行時に配列の大きさを任意の次元方向に拡張することは不可能である．例えば，配列 v を上記のアドレス関数によりメモリ上に割り付けた場合，実行時に i 方向に1行分拡張が必要である（例えば，**図3**において，大きさ (3, 4) の状態から (4, 4) に拡張する場合）とすると，拡張分を同じアドレス関数で割り付けることは不可能である．アドレス関数を組み直して v のすべての要素を再配置する必要がある．一般に n 次元配列について，ただ一つの次元方向についてのみ拡張以前のアドレス関数を使用して拡張可能であり，他の次元方向の拡張については拡張以前の配列要素の再配置なしでは拡張できない．ただし，一つの次元方向に拡張可能であるとはいっても，拡張に必要なメモリアドレスはすでに他

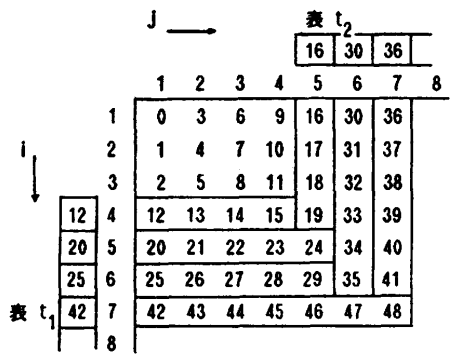


図 3 2次元拡張可能配列の拡張
Fig. 3 Extensions of a two dimensional extendible array.

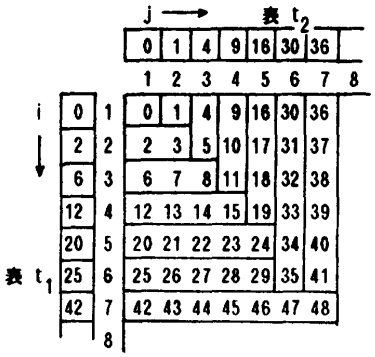


図 4 従来の index-array による拡張可能配列の実現
Fig. 4 Realization of an extendible array using the usual 'index-array' method.

の静的変数によって占められているのが普通である．

n 次元拡張可能配列変数 v の現在の形状が $(l_1..h_1, \cdots, l_k..h_k, \cdots, l_n..h_n)$ であり，初期配列が $(p_1..q_1, \cdots, p_k..q_k, \cdots, p_n..q_n)$ であるとする．$s_i=\mathrm{ord}(h_i)-\mathrm{ord}(l_i)+1$ とすれば，v のサイズは $s_1\times\cdots\times s_k\times\cdots\times s_n$ となる．次元 k の上限または下限方向に一つだけ拡張が起こったときには形状が $(l_1..h_1, \cdots, l_{k-1}..h_{k-1}, l_{k+1}..h_{k+1}, \cdots, l_n..h_n)$ で，したがって大きさが $s_1\times\cdots\times s_{k-1}\times s_{k+1}\times\cdots\times s_n$ の通常の $n-1$ 次元部分配列 v' が動的に割り付けられる．v' の要素は v' の先頭番地と要素の大きさがわかれば v' に対する通常のアドレス関数を計算することによってその番地が与えられる．そこで，図3に示すように初期配列の部分は除いて，v のすべての拡張可能な次元について，拡張があるたびに動的割付けルーチンにより割り付けられる部分配列の先頭番地を書き込む表を次元ごとに用意する．もし動的割付けルーチンがメモリ領域を要求順に番地の高い向きあるいは低い向きに単調に割り付けるならば v の要素 $v(i_1, \cdots, i_n)$ の番地は次のようにして求まる．任意の $1\leq r\leq n$ に対して，$p_r\leq i_r\leq q_r$ のとき，すなわち要素が初期配列内にある時には，初期配列を通常どおりメモリに割り付けるとして，そのアドレス関数を計算する．要素が拡張部分にある時には，v の拡張可能な次元ごとに存在する前述の先頭番地の表の集合を T とし，$t_k\in T$ が次元 k に対する表とする．このとき，$\max\{t_k(i_k)\mid t_k\in T, 1\leq k\leq n\}$ がこの要素を含む部分配列 v' の先頭番地を与える．これと，v' のアドレス関数を構成する情報（v' の各次元についての添字の下限と成分のサイズであり，部分配列ごとに与えられる）を知り，計算すればよい．例えば，図3において，要素 $v(5,6)$ に対しては，$t_1(5)$ と $t_2(6)$ を比較して，$t_1(5)<t_2(6)$ であるので，部分配列 v' の先頭番地は $t_2(6)=30$ と決定する．これに部分配列内の変位 5−1 を加えて得られる 34 が求める番地となる．

このような方法により，簡単な補助表を用意するだけで，従来の固定サイズの配列のアドレス計算法ほど高速ではないにしろ，実用に耐え得る程度のシンプルな計算方法が得られることは真に注目に値する．他の方法と比較して，この方法の優位性が定量的に示されている[3]．ただし，文献3)における拡張可能配列のモデルでは，初期配列の考え方はなく，大きさ0から拡張されるので添字の全範囲について補助表を用意する必要がある（図4参照）．これは一次元の拡張可能配列の場合にはメモリの利用率を極端に悪化させる．ここでは，初期配列の部分は静的に割り付け，通常のアドレス関数を適用するので，初期配列の添字の範囲については補助表が不要である．

文献3)における方法も含めて，上記の方法における大きな問題点は動的割付けルーチンが必ずしも単調にメモリ領域を割り付けるとは限らず，この方法は一般には適用できないことである．そこで，ここでは上記の表とは別に，拡張が起こった経歴を記録する表も同時に用意する（図5参照，図4と比較されたい）．上記の表の代わりにこの表を使って同様に最大の経歴を調べ対応する先頭番地を求めればよい．実際にはこの先頭番地は各々の部分配列のアドレス関数を構成するのに必要な情報を格納するためのヘッダを指示しており，部分配列本体はそれに引続いて割り当てられる．拡張可能配列の各次元ごとに持たせる情報として，これら二つの表以外にその次元において必要な番地計算のための情報がある．以後，二つの表とこれらの情報を合わせて表 t_i（i は次元番号）と言う．また，各次

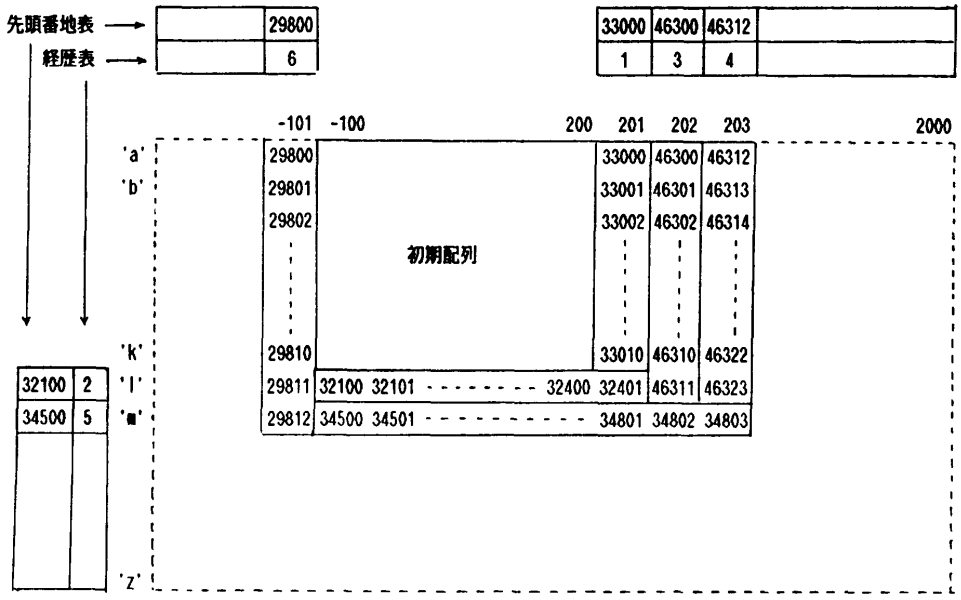


図 5 拡張可能配列の実現
Fig. 5 Implementation of an extendible array.

元に対する表 t_i と経歴を与えるカウンタ，初期配列の先頭番地の情報などを含めて表Tと言う．表Tと部分配列のヘッダの内容を**図 6** に示す．

4. Pascal 処理系への拡張可能配列の組込み

ここでは，文献 6)で与えられている Pascal P 4 の処理系を UNIX machine (HP-9000 model 310: cpu 68010) に移植した後，配列に関する P 4 の仕様を上述のように拡張し，それをインプリメントする部分を P 4 の処理系に組み込んだ．P 4 の処理系はソースプログラムを読み込み，Pascal の仮想機械に対するニーモニックコード（pコード）を生成する翻訳系とニーモニックコードをバイナリコードにアセンブルする直訳系 (assembler)，およびバイナリpコードを解釈実行する通訳系 (interpreter) から成る．

4.1 インプリメント上の問題点

次の諸点が問題となった．

(i) 表Tをどこにおくか．

(ii) 初期配列をどこにおき，その番地付けをどうするか．

(iii) 拡張部分は heap 領域に確保するが，Pascal の動的割付け手続き new との関連をどうするか．

(iv) 拡張可能配列が手続きに引数として受け渡されたときの処理方法をどうするか．

拡張可能配列型変数は，有効範囲 (scope) や可視性 (visibility) 等の点において Pascal の他の変数と同じ扱いを受ける．したがって，実行時スタック上の変数割当てと解放は他の変数と同じように行う必要がある．ここでは，手続きが起動されたときには，それに局所的に宣言されている拡張可能配列に対応する表Tが実行時スタック上に確保され，初期化される．手続き終了後は他の局所変数と同じように解放される．また，先にも述べたように，ここでは，初期配列は実行時スタック上に置くことにより，手続き終了後，動的に解放され，使用可能メモリを圧迫しないようにする．初期配列の要素の番地付けは通常のアドレス関数で行える．ただし，要素の参照に際して，それが初期配列に含まれるかどうかを判定するオーバヘッドが必要である．初期配列を拡張領域に置くとすれば，手続きの実行直前に初期配列の大きさにまで動的に拡張する必要がある．また要素の番地付けは通常のアドレス関数で行えない．(iii)については拡張領域を含む heap 領域の有効な屑集めの問題とも関連するが，ここでは動的割付け領域を一本化して効率的に利用したいことから new と割付けポインタを共有させることにする．(iv)については整合配列の実現としてつぎの 5 章で述べる．

4.2 翻訳系の拡張

拡張可能配列を支援するために，主に次のような翻訳系の拡張を行った．

(i) 型と変数名の属性表

型の処理部で作成される部分範囲型の属性として，

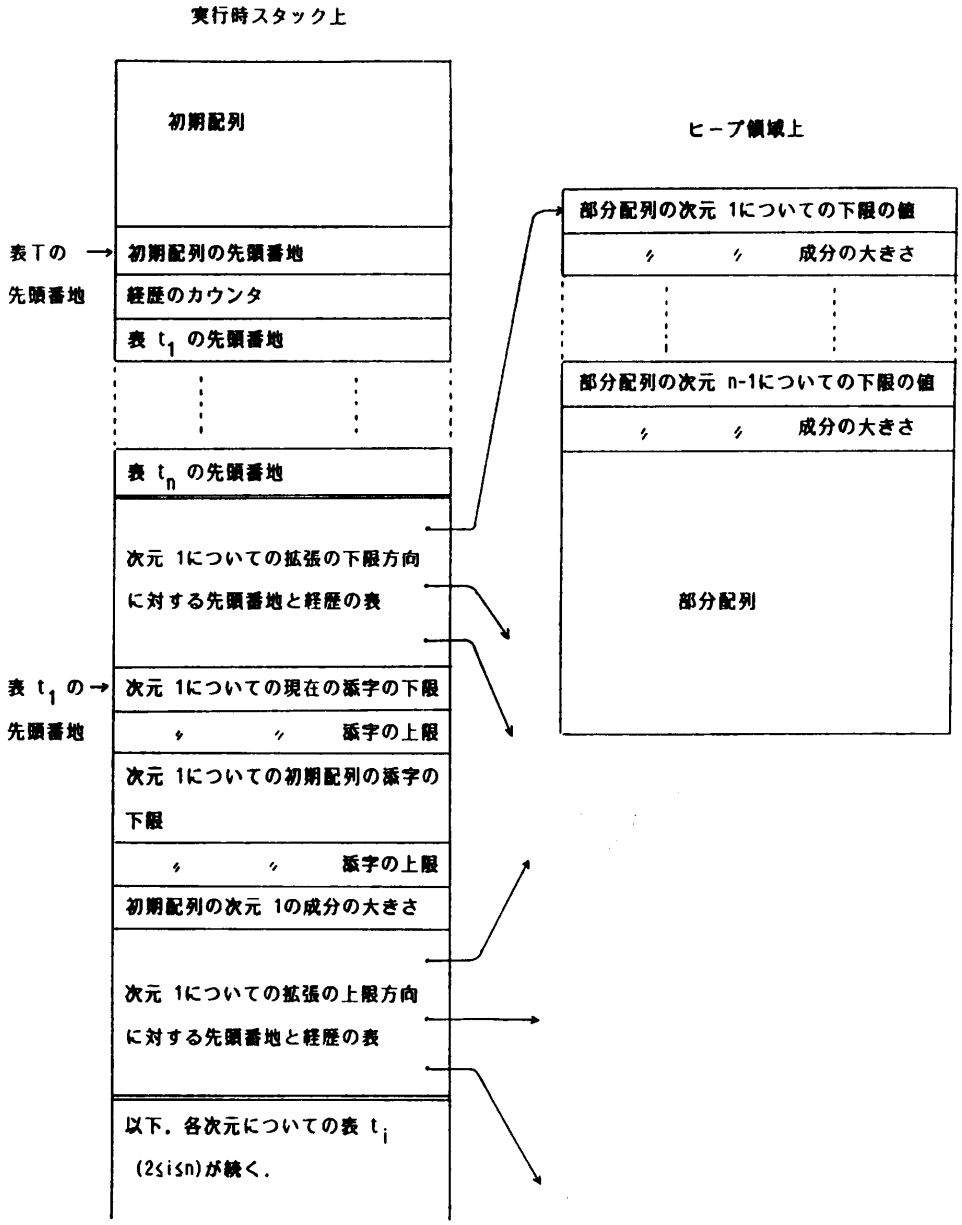


図 6 表 T と部分配列のヘッダの内容
Fig. 6 Contents of table T and a header of partial array.

拡張可能な上下限値を表す $a \ll b$ タイプのものであるかどうかを指示するフラグを設ける．また，配列型の属性表は表Tに対して関連する情報を与えることができるように拡充される．さらに初期配列の属性は拡張可能配列の変数名に付随して確保される必要があり，配列型変数名の属性が増補される．

（ii） 配列要素参照のためのコード生成

実行文の翻訳において，配列型変数の要素参照が現れると翻訳系は，その添字を処理して，要素参照のためのコードを生成する．拡張可能配列の要素参照も可能にするため，コード生成部を拡充・修正する．

（iii） 表Tの初期化

手続き呼出し時に局所変数領域に置かれる表T中の諸情報はスタック枠を作成した直後，初期化する必要がある．この初期化のためのコードを実行文のコードの直前に生成するように拡充する．

（iv） 標準手続き extend と標準関数 *lb*, *ub* の呼出しに対するコード生成

5. 整合配列[5),7)] への応用

拡張可能配列は引数として手続きに受け渡すことができる．手続きの仮引数の型指定は拡張可能配列型の型名のみを指定する．このとき，実引数である拡張可能配列変数の初期配列の添字の範囲にかかわらず，その型（の構造）が，仮引数の型と一致しておれば，仮引数に整合する．**図7は無向グラフの隣接行列を生成するプログラムである．拡張限界の範囲内の任意の形状の拡張可能配列を手続きに受け渡すことができる．**なお，拡張可能配列を用いたために，このプログラムのユーザは new による動的割付けを使用するときのようにサイズを気にせずにグラフを成長させていくことができる．しかも，その検索は配列のランダムアクセス性により迅速に行うことができる．

整合配列引数を処理するときには，実引数である拡張可能配列変数の要素の番地計算に必要な情報を実行時に，動的に引き渡す必要がある．このとき，配列としての諸操作の実行効率を落さないようにすることが肝要となる．これについては，次の6章で説明するように増設・修正したpコードの引数により静的あるいは動的引渡しの区別をきめ細かくし，可能なかぎりpコードの引数として静的に引き渡すようにした．ただし，その分翻訳系の負担は増加することになる．動的引渡しの対象となる情報は図6に示されている表T中の諸情報であり，これらは表Tの先頭アドレスや各々の次元についての表 t_i の先頭アドレスを実行時スタック上に積むことによって受け渡される．拡張可能配列変数が値呼びで手続きに受け渡されるときには，通常の配列の場合よりも，オーバヘッドを伴う．すなわち，配列要素のデータのコピー以外に表Tの確保とその初期化を必要とするからである．翻訳系は値呼びの仮引数を認めたときには，その引数の型より表Tのサイズを計算する．しかし，表Tのサイズは変数宣言時の初期配列の指定が決定してからでないと決定できない．したがって，実行時に実引数が結び付いた時点で実引数の表Tから決定されることになり，静的な決定はできない．にもかかわらず，仮引数のための表Tの領域をスタックの静的変数領域に確保する必要がある．そこで，ここでは，実引数の初期配列の各次元の添字の範囲を $l_1..h_1, \cdots, l_n..h_n$ としたとき，仮引数の初期配列の要素を $[l_1, \cdots, l_n]$ の1要素のみからなる配列として，翻訳時に仮引数の表Tのサイズを確定しておく．手続きが呼び出されたときには，実引数の表Tから，$[l_1, \cdots, l_n]$ を確定し，現在の大きさが1要素の初期配列であるように，表Tの諸情報をセットした後，実引数の配列の大きさにまで内部的に拡張する．その後に，実引数の要素をすべて仮引数の要素にコピーする．

```
program buildgraph(input,output);
const max = 80; size1 = 15; size2 = 50;
type graph = array[1<<max,1<<max] of boolean;
var g1: graph(1..size1,1..size1);
    g2: graph(1..size2,1..size2);

procedure init(n: integer; var g: graph);
var i,j,q: integer;
begin
  q := ub(g,1);
  for i:=1 to q do
    for j:=n+1 to q do
      if i = j then g[i,j] := true
        else g[i,j] := false;
  for j:=1 to n do
    for i:=n+1 to q do g[i,j] := false
end;

procedure build(var g: graph);
var v,w,p: integer;
begin
  writeln('vertex ?'); readln(v);
  while v <> 0 do
    begin p := ub(g,1);
      if v > p then
        begin extend(g,1,v,1,v);
          init(p,g) end;
      read(w);
      while (w >= 1) and (w <= ub(g,1))  do
        begin g[v,w] := true; g[w,v] := true;
          read(w) end;
      writeln('vertex ?'); readln(v)
    end;
end;

begin init(0,g1); build(g1);
  init(0,g2); build(g2)
end.
```

図7 無向グラフの生成プログラム
Fig. 7 A program generating undirected graphs.

手続き内において，仮引数に結び付いている実引数である拡張可能配列の各次元の現在の上下限値が必要なときには，標準関数 *lb*, *ub* を使用して知ることができる．

通常の配列に対して，このような整合配列の機能を支援するときには，それが整合配列仮引数に受け渡されることを考慮して，アドレス計算のために必要な情報を実行時レベルで個々の配列ごとに持つ必要がある．ここで行ったように，整合配列として機能する配列を拡張可能配列に限定することにより，通常の配列に対して必要な，このようなオーバヘッドは回避できることになる．

6. pコードの増設・修正と標準手続き・関数の実現

従来の固定サイズの配列の操作の実行効率を落とさ

```
lao       9   aの先頭番地をスタックトップへ.
ldoi  10009   1次元目の添字をスタックトップへ.
chki 1  100   添字の値が上下限内にあるか検査.
deci      1   添字の値から下限を引き変位を求める.
ixa  0  100   変位に成分の大きさ 100をかけて先頭番地
              に加える.
ldoi  10010
chki 1   10   2次元目について同様の操作をする.
deci      1
ixa  0    1   実行後，求める番地がスタックトップに.
ldci   9999   代入する値をスタックトップへ.
stoi          代入.
```

(a) 図10(a) の代入文

```
lao       9   初期配列の先頭番地をスタックトップへ.
ldoi    420   1次元目の添字をスタックトップへ
lao     221   表 t1 の先頭番地をスタックトップへ
cix  1  100   表 t1 より添字の値が上下限内にあるか検査.
              さらに添字の値が初期配列の範囲内なら下限
              からの変位を求めてスタックトップへ.
ixa  1    1   添字の値が初期配列の範囲内なら変位に初期
              配列の 1次元目の成分の大きさをかけて先頭
              番地に加える.
ldoi    421 ┐
lao      18 │ 2次元目について同様の操作をする.
cix  1  100 │
ixa  2    1 ┘
eix       2   要素が拡張部分にあるなら部分配列内の変位を
              計算して先頭番地に加えてスタックトップへ.
ldci   9999   代入する値をスタックトップへ
stoi          代入
```

(b) 図10(c) の代入文

```
loda 0    5   表Tの先頭番地をスタックトップへ.
inda          間接参照して初期配列の先頭番地を求める.
ldoi  10023   1次元目の添字をスタックトップへ
loda 0    5
indi      2   間接参照して表 t1 の先頭番地をスタックトップへ
cix  max  0   表 t1 より初期配列の範囲を求めるほかは
              (b) の場合と同じ.
ixa  1    0   表 t1 より成分の大きさを求めるほかは
              (b) の場合と同じ.
ldoi  10024 ┐
loda 0    5 │
indi      3 │ 2次元目について同様の操作をする.
cix  max  0 │
ixa  2    0 ┘
eix       2   (b) の場合と同じ.
ldci   9999   代入する値をスタックトップへ
stoi          代入
```

(c) 図10(e) の代入文

図 8 図10のプログラム (a), (c), (e) の代入文に対する生成pコード列
Fig. 8 Generated p code sequences for the assignment statements in Fig. 10.

ないようにするために拡張可能配列に関する操作に対しては新たなpコードを増設したり，従来のpコードを修正した．それに伴い，通訳系を増補した．

比較のために図8(a), (b), (c)にそれぞれ図10(a), (c), (e)のテストプログラム中の代入文に対するpコード列を示して説明しておく．

cix *p q* 拡張可能配列の添字を検査するコードであり，対象とする拡張可能配列が手続きの仮引数でない時 p, q は初期配列の対応次元の上下限値である．仮引数の時には，pの値は max (cix 命令の引数として取り得る最大値として定義している) である．いずれの場合にも，実行以前に，検査すべき添字の値と添字の次元 i に対する表 t_i の先頭番地がスタック上に置かれており，仮引数の場合には表 t_i を通じて実行時に初期配列の上下限値を引き当てる．

(i) 添字の値が，現在の配列の上下限を越える場合は実行時誤りとする．

(ii) 添字の値が初期配列の範囲にあるならば添字の値から初期配列の下限を引いた値をスタックトップに置く．仮引数ならば表Tの先頭番地から下限を引き当てる．

(iii) 添字の値が拡張部分にあり，経歴が最新であれば，通訳系内にもっている番地計算のための関連情報を更新する．

従来のコードとは異なり，後に eix コードにより拡張部分の要素のアドレスを計算するために，cix コード実行後，検査すべき添字の値はスタックに残している．

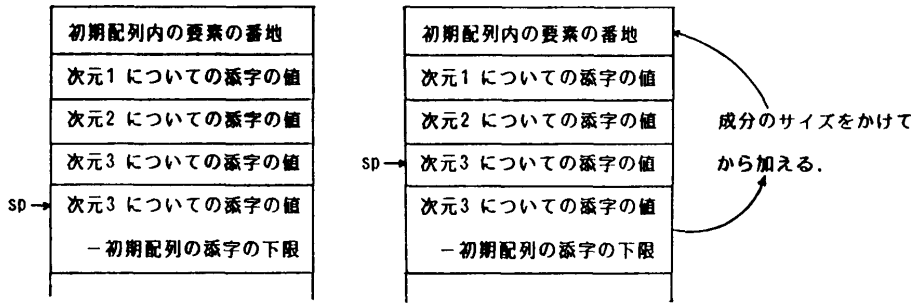
ixa *p q* 従来の配列または拡張可能配列の初期配列の要素のアドレス計算を対応する次元について計算するコードであり，従来の ixa コードを修正して使用する．$p=0$ の時は従来の配列である．$p \neq 0$ の時 p は拡張可能配列の次元番号を表す．また仮引数の時には q は0であり，そうでない時には，成分のサイズを表す．仮引数の時には，表 t_i により，実行時に成分サイズを引き当てる．

(i) スタックポインタを一つ戻し，pの値が0なら従来どおりのアドレス計算をする．

(ii) $p \neq 0$ ならば，通訳系内の情報を参照してすでにこの次元を含む上位次元において拡張部分の添字が現れたかを調べる．

(iii) まだ現れていない (つまり，すべて初期配列の範囲内) なら，スタックトップにある添字の変位に成分の大きさをかけて，スタックポインタを pの分，下げた位置にあるアドレスに加える (図9参照)．このアドレスには，最初，初期配列の先頭番地が入っている．

eix *q* 拡張可能配列の拡張部分の配列要素のアドレスを計算するコードであり，qは拡張可能配列の次元数である．このコード実行以前には各次元について cix コードによりスタック上に要素の添字の値が積上げられている．(i) 配列要素が初期配列内にあれば，スタックポインタを q だけもどして終る．この時，スタックトップに求めるアドレスが得られている．



(a) 実行直前のスタック (b) ixa命令の実行

図 9 まだ拡張部分の添字が表れていない場合の ixa 命令の実行
Fig. 9 Execution of 'ixa' instruction in the case where any extended index has not appeared yet.

(ii) 拡張部分にあるならば，通訳系内の情報により，拡張部分の部分配列のヘッダにアクセスし，部分配列の各次元の添字の下限と成分サイズを得る．これらの情報とスタック上の添字値列を使用して部分配列内の変位を計算する．その結果と部分配列の先頭アドレスを加えて求めるアドレスを得る．

標準手続き extend 指定された引数に従って拡張範囲を決定し，各次元について図5に示すように，一つずつ部分配列のサイズを計算しそれをヒープ上に順次割り付け，その位置と経歴の値を表 T に書き込む．例えば，後出の図 10(c)のプログラムのように，一度に拡張したとしても，extend は内部的には逐次拡張していく．

lb, ub 指定された次元 i についての表 t_i より添字の現在の下限および上限を返す．

7. 評　　価

(a) 従来の固定サイズの2次元配列の場合，

(b) 初期配列の大きさを拡張限界まで指定した場合，

(c) 初期配列の大きさを最小にして拡張限界まで拡張した場合，

(d) 初期配列を適度な大きさに指定して拡張限界まで拡張した場合，

(e) 拡張可能配列を整合配列として，手続きの変数仮引数に受け渡した場合，

(f) 拡張可能配列を整合配列として，手続きの値仮引数に受け渡した場合，

のそれぞれについて，アクセス速度と所要メモリに関する評価に使用したテストプログラムを**図 10**(a)～(f)に示し，評価の結果を**図 11** に示す．評価の要点は拡張可能配列要素のアクセス速度であり，以下 a～f と書いた場合，図 11(a)～(f)のプログラムの実行時間をそれぞれ表すものとする．従来の固定配列を拡張可能配列とした時の速度の低下の下限は，$(b-a)/a$ で与えられ，11% 程度である．上限は $(c-a)/a$ で与えられ 28% 程度である．また，プログラム (d) のような平均的な使用においては，速度の低下は，$(d-a)/a$ で与えられ，17% 程度である．初期配列を静的領域におくことの効果は $(c-b)/c$ で与えられ，13% 程度向上する．さらに整合配列として手続きの番地仮引数に受け渡す場合のオーバヘッドは $(e-d)/d$ で与えられ 11% 程度である．また，手続きの値仮引数に受け渡す場合には，かなり遅くなっているが，主に表 T の初期化と仮引数に実引数の拡張可能配列全体をコピーするのに要する時間が原因であり，これは 19.2 sec であった．以上の結果より，配列の最大の利点である迅速なランダムアクセスの速度を，それほど損なわないで拡張可能配列を支援できることがわかる．

また，所要メモリの増加分は，図6に示した表 T と部分配列のヘッダに要する分であり，配列本体 ($100\times100=10000$ ワード) に対する増加の割合は，最大の場合 (プログラム(c)) で8%程度であり，平均的な使用 (プログラム(d)，(e)，(f)) においては1%以下である．

最後に拡張可能配列支援のための生成コードのオーバヘッドであるが，コードサイズの増加の原因として，4.2 節でも述べたように次の二つがある．

1) 手続き呼出し直後の表 T の初期化のためのコード
2) 拡張可能配列要素の参照のためのコードのオーバヘッド

```
program test0;
type ear =
  array[1..100,1..100] of integer;
var a: ear; i,j: integer;
begin
  for i:=1 to 100 do
    for j:=1 to 100 do
      a[i,j] := 9999
end.
```

(a)

```
program test1;
type ear =
  array[1<<100,1<<100] of integer;
var a: ear(1..100,1..100);
  i,j: integer;
begin
  for i:=lb(a,1) to ub(a,1) do
    for j:=lb(a,2) to ub(a,2) do
      a[i,j] := 9999
end.
```

(b)

```
program test2;
type ear =
  array[1<<100,1<<100] of integer;
var a: ear(1..1,1..1);
  i,j: integer;
begin
  extend(a,1,100,1,100);
  for i:=lb(a,1) to ub(a,1) do
    for j:=lb(a,2) to ub(a,2) do
      a[i,j] := 9999
end.
```

(c)

```
program test3;
type ear =
  array[1<<100,1<<100] of integer;
var a: ear(1..80,1..80);
  i,j: integer;
begin
  extend(a,1,100,1,100);
  for i:=lb(a,1) to ub(a,1) do
    for j:=lb(a,2) to ub(a,2) do
      a[i,j] := 9999
end.
```

(d)

```
program test4;
type ear =
  array[1<<100,1<<100] of integer;
var a: ear(1..80,1..80);
  i,j: integer;
procedure p(var a: ear);
begin
  for i:=lb(a,1) to ub(a,1) do
    for j:=lb(a,2) to ub(a,2) do
      a[i,j] := 9999
end;
begin extend(a,1,100,1,100); p(a) end.
```

(e)

```
program test5;
type ear =
  array[1<<100,1<<100] of integer;
var a: ear(1..80,1..80);
  i,j: integer;
procedure p(a: ear);
begin
  for i:=lb(a,1) to ub(a,1) do
    for j:=lb(a,2) to ub(a,2) do
      a[i,j] := 9999
end;
begin extend(a,1,100,1,100); p(a) end.
```

(f)

図 10 テストプログラム
Fig. 10 Test programs.

プログラム	(i)	(ii)	(iii)	(iv)
(a)	41.8	1	10000	11
(b)	46.5	1.11	10014	12
(c)	53.5	1.28	10809	12
(d)	49.1	1.17	10089	12
(e)	54.6	1.31	10089	15
(f)	68.6	1.64	10089	12

(i) プログラム実行時間
(ii) プログラム(a)の実行時間を1としたときの割合
(iii) 最大拡張時の拡張可能配列が占めるメモリ領域の大きさ
(iv) 代入文のコードサイズ（pコード命令数）

図 11 テストプログラム（図 10）の評価
Fig. 11 Evaluations of the test programs in Fig. 10.

このうち 1) については手続きの冒頭で一度だけ挿入されるコードであり，拡張可能配列一つについて 28 pコード命令の大きさである．コードの増加の原因として重要なのはむしろ 2) のほうであり，従来の固定配列の要素に対する代入文（11 pコード命令）に対して，直接に参照する場合（(b), (c), (d), (f)）で1命令の増加，間接に参照する場合（(e)）には4命令の増加である．

8. おわりに

本論文では，拡張の履歴を記録することにより動的メモリ割付け機能とのインタフェースを考慮した拡張

可能配列の新たな実現モデルを設定し，それに基づいて現実のプログラミング言語 (Pascal) に拡張可能配列を支援させるための方式を提案した．また，本方式の採用により整合配列機能が拡張可能配列として自然な形で実現できることを示した．さらに，実現した処理系について配列要素のアクセス速度を実測し，配列固有の迅速なランダムアクセスの長所を損わないことを確認した．

拡張可能配列を手続きの局所変数としたとき，手続き実行中にヒープ領域上に拡張した部分配列は手続き終了後，解放されることが望ましい．また，拡張可能配列は同時に縮小可能であることも望ましく，縮小分は解放される必要がある．これらのためのヒープ領域の有効な層集めを実現する方策を検討することは今後の課題である．

参考文献

1) Rosenberg, A. L.: Allocating Storage for Extendible Arrays, *JACM*, Vol. 21, pp. 652-670 (1974).
2) Rosenberg, A. L. and Stockmeyer, L. J.: Hashing Schemes for Extendible Arrays, *JACM*, Vol. 24, pp. 199-221 (1977).
3) Otoo, E. J. and Merrett, T. H.: A Storage Scheme for Extendible Arrays, *Computing*, Vol. 31, pp. 1-9 (1983).
4) Liscov, B. H. et al.: CLU Reference Manual, Springer-Verlag, Berlin (1981).
5) 情報処理新興事業協会編：最新 Ada 基準文法書，共立出版，東京 (1984).
6) Pemberton, S. and Daniels, M.: *PASCAL IMPLEMENTATION—The P4 Compiler—*, Ellis Horwood Limited, Chichester (1982).
7) 石畑，筧，安村(訳著)：Pascal の標準化—ISO 規格全訳とその解説—，共立出版，東京 (1984).
8) 都司，井口，渡辺：拡張可能配列の実現方式とその Pascal 言語への組込み，第 34 回情報処理学会全国大会論文集，2V-1 (1987).

(昭和 62 年 11 月 24 日受付)
(昭和 63 年 7 月 15 日採録)

都司 達夫（正会員）

昭和 24 年生．昭和 48 年大阪大学基礎工学部電気工学科卒業．昭和 53 年同大学院工学研究科博士課程修了．工学博士．同年福井大学工学部情報工学科講師．現在，同助教授．プログラミング言語，データ構造論に関する研究を行っており，AI に興味を持つ．電子情報通信学会会員．

井口 雅彦

昭和 37 年生．昭和 61 年福井大学工学部情報工学科卒業．同年北陸日本電気ソフトウエア(株)入社．現在，同基本技術開発部にて汎用コンピュータの通信系 RAS 技術の開発に従事．

渡辺 勝正（正会員）

1940 年生．1968 年京都大学大学院博士課程数理工学専攻単位修得退学．京都大学工学博士．京都大学工学部を経て，現在，福井大学工学部情報工学科教授．計算機言語，言語処理プログラム，計算機アーキテクチャ，並列アルゴリズムの研究に従事．著書「アルゴリズムと計算機械」，訳書「コンピュータアーキテクチャの設計」．