Programming and Computer Software, Vol. 28, No. 6, 2002, pp. 333–341. Translated from Programmirovanie, Vol. 28, No. 6, 2002.

Refined Description of the C[] Language

A. Ya. Kalinov, A. L. Lastovetsky, I. N. Ledovskih, and M. A. Posypkin

Institute of System Programming, Russian Academy of Sciences,
ul. Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia
e-mail: posypkin@ispras.ru

Received August 15, 2001

Abstract—In the paper, an accurate and detailed description of the programming language C[] is given. Unlike previous publications on the language, this paper gives a formal definition of a vector, which is used for the description of the semantics of basic constructs. The language updates that appeared after the first publication are discussed. Other vector programming languages are surveyed and compared with C[].

1. INTRODUCTION

Operations on arrays are important components of the majority of computational mathematics algorithms. Therefore, a number of languages have been developed that support such operations. Programming languages supporting operations on arrays, which are further referred to as *vector operations*, can be classified into three groups. The first group includes languages of the MATLAB [1] and APL [2] types, in which these operations have been available from the very beginning. The second group includes languages like ZPL [3] and SAC [4], which have been developed on the basis of the existing languages (ZPL is based on Modula-2, and SAC, on C) but are not their extensions. They, for example, lack such an important construct of the prototype languages as pointers. Finally, the third group includes languages that are extensions of the earlier existing languages, such as, e.g., FORTRAN 90/95 [5]. We consider FORTRAN 90 as an extension of FORTRAN 77 since only obsolete constructs of the latter, which do not reduce its expressiveness, are removed.

In this paper, the current version of the C[] language [6, 7], which is classified among the third group, is described. The C[] language, in turn, is a subset of the parallel programming language mpC [8].

C[] is a strict extension of the programming language C. In C, an array in expressions is transformed to a pointer to the type of its elements. In C[], by means of the special grid and block operators, which are discussed in detail in Section 2, this transformation is prevented, and the array takes part in arithmetic operators as a whole.

C[] allows one to code various algorithms more compactly compared to C. On the other hand, additional semantic information included in vector expressions gives the compiler more possibilities for performing optimizing transformations. Studies carried out [9] show that such information can efficiently be employed, for example, for optimizing cache use.

In Section 2, a description of C[] is given. Syntax and semantics of vector operators are considered in detail. Differences of the current version of C[] from that published earlier are discussed. In Section 3, C[] is compared with other vector languages.

2. DESCRIPTION OF THE C[] LANGUAGE

2.1. Types

2.1.1. Vectors. In C, the notion of an *object* is introduced as a memory area the content of which can represent values [10]. The C[] language introduces the notions of a *vector of objects*, or simply *vector*, and a *vector type*, which is the type of a vector.

A *vector* is defined as an ordered sequence of objects or vectors of one type *T*. Here, *T* is any type of C[] different from functional. Elements of the sequence forming a vector are called *elements of the vector*, and the *i*th elements of a vector v is denoted as v_i. The number of elements in this sequence is referred to as the *vector length*, and the type *T*, the *type of the vector element.*

A vector type is specified by two attributes: the number of elements (or length) of vectors of this type and the type of the elements. If *T* is a vector type, then *Dim*(*T*) denotes its length, and *VET*(*T*), the type of its elements. All types of C[] that are not vector types are referred to as *nonvector types*.

As is known, an object in C can represent a value. Similarly, in C[], a vector represents a *vector value*. A value of a vector in C[] is said to be a vector value. A vector value is an ordered sequence of values of elements of the vector. A vector value has the same type as the vector that represents this value.

Let us introduce the notion of the *rank of a vector type.* Let *T* be a vector type. Then, its rank *Rank*(*T*) is defined by the formula

$$Rank(T) = \begin{cases} 1, \text{ if } VET(T) \text{ is a nonvektor type,} \\ Rank(VET(T)) + 1, \\ \text{if } VET(T) \text{ is a vector type.} \end{cases}$$

0361-7688/02/2806-0333$27.00

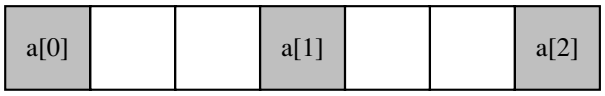

Fig. 1. Location of elements of a one-dimensional array with the step 3.

A *form of a vector type (vector)*[1] T is a sequence of integers $Form(T)$ of length $Rank(T)$ formed by the rule

$$Form(T) = \begin{cases} \{Dim(T)\}, & \text{if } VET(T) \text{ is a nonvektor type,} \\ \{Dim(T), Form(VET(T))\}, & \\ \text{if } VET(T) \text{ is a vector type.} & \end{cases}$$

Two vector types T_1 and T_2, such that $Rank(T_1) = n_1$, $Rank(T_2) = n_2$, and $n_1 < n_2$, are said to be *conformal* if the form $Form(T_1)$ is an initial subsequence of the form $Form(T_2)$, i.e., if

$$Form(T_2) = \{Form(T_1) f_{n_1+1}, \ldots, f_{n_2}\}.$$

For example, the vector types with the forms {5, 8} and {5, 8, 7, 3} are conformal.

By definition, any nonvector type is assumed to be conformal to any vector type.

If a vector (vector value) v has a vector type T, then the object (value) $v_{i_1, \ldots, i_{Rank(T)}}$, where $0 \le i_j < Form(T)_j$ for $1 \le j \le Rank(T)$, is called a *terminal element with index* $\{i_1, \ldots, i_{Rank(T)}\}$.

The *terminal type for a given vector type T* is defined as the type of its terminal element and denoted as $VTT(T)$.

2.1.2. Arrays. An array in C is defined as a set of objects of one type located in the memory in succession. C[] extends this notion by introducing a new attribute, step. The step is a positive integer that determines the distance between the initial addresses of successive elements of the array measured in the units equal to the element of the given array.

Thus, arrays in C[] are composed of objects of one type located in the memory at a certain distance, equal to a given step, one from another. Figure 1 shows an array with the step equal to 3.

2.1.3. Pointers. The notion of a pointer is extended in a similar way, such that a pointer in C[] may have a step. As in the case of arrays, a pointer step is a positive integer, which is used for defining various arithmetic operators with pointers. Ordinary C pointers are viewed as pointers with the unit step.

If an integer is added to or subtracted from a pointer, it is first multiplied by the product of the size of the object to which the pointer refers to and the pointer step.

In C[], the difference of pointers is determined only for pointers of one type, i.e., for pointers that have one step and refer to one type. When two pointers to elements of one array with the step equal to the step of the pointers are subtracted, their difference is divided by the size of the element of this array multiplied by the step. The result is the difference of values of indices of two elements of the array. If two pointers taking part in the operator do not refer to elements of one and the same array, the behavior is not determined.

2.1.4. Dynamic types. The C[] language permits the use of a nonconstant expression as a specifier of an array dimension, array step, or a pointer. Such arrays and pointers are further referred to as *dynamic*. Only arrays with automatic storage duration may be dynamic. An expression for array dimensions must not contain subexpressions with side effects and subexpressions of vector type. The same rules are used for pointers in C[].

Listing 2.1. Access to diagonal elements by means of a pointer to an array with a step.

```
typedef (* tDiag)[N + 1];
int A[N][N];
tDiag p;
…
p = (tDiag)A;
…
(*p)[i] = 1;
```

Arrays and pointers with a step are a convenient tool for accessing various collections of elements of a given array. For example, in the program fragment represented in Listing 2.1, the pointer to the array `p` makes it possible to access diagonal elements of the matrix stored in the array `A`. The type `tDiag` is the type of the pointer to the array with the step `N + 1`. Since arrays in C are stored in memory by rows, the diagonal elements of the array are located in memory with the step `N + 1`, and the expression `(*p)[i]` denotes the *i*th diagonal element.

2.2. Type Transformations

2.2.1. Classification of type transformations. In C[], transformations of types are classified into two groups: *scalar type transformations* and *vector type transformations.*

All transformations of nonvector types fall into the group of scalar transformations.

Vector transformations, in turn, are divided into two groups: *transformations of terminal types* and *conformal extensions.*

2.2.2. Transformations of terminal type. A *transformation of a terminal type to a scalar type S* is a type transformation applied to the value of an expression of vector type T that results in a vector of type T' the form

[1] The notion of a form of a vector type is defined in a similar way in the languages FORTRAN 90, C*, ZPL, as well as in some other vector languages.

of which coincides with the form of the vector type *T*, and the type of the terminal element is *S*.

The result of a transformation is determined as follows. Let *S* be a scalar type and *T* be a vector type of an expression *E* the value of which is a vector value v. Then, the result of the application to the expression *E* of a transformation of a terminal type to the type *S* is a vector value the elements of which are obtained from elements of the vector value v by applying either the transformation of a terminal type to the type *S* if *VET*(*T*) is a vector type or a scalar transformation to the type *S* otherwise.

2.2.3. Conformal extensions. Conformal extensions make it possible to transform vector values of different (but conformal) types to one type, as well as to transform scalar values to a vector type.

Let *a* be a value of a scalar type, and *T* be a vector type. Then, the *conformal extension of a to the type T* is a vector v of type *T*', where *Form*(*T*) = *Form*(*T*') and *VTT*(*T*) = *Type*(*a*). Note that all terminal elements of the vector v coincide with *a*.

Let *T* and *T*' be conformal types of rank *m* and *n*, respectively, $m < n$. Then, the *conformal extension of the vector value v of type T* to the *type T*' is a vector value v' of the form *Form*(*T*') with the terminal type equal to *VTT*(*T*). In this case, terminal elements of v' are calculated by the formula $v'_{i_1, \ldots, i_n} = v_{i_1, \ldots, i_m}$.

2.2.4. Transformations of types of operators operands. In this section, we consider implicit transformations of types and restrictions imposed on types of the C operators extended to the vector case.

C[] permits only conformal operands in any operators admitting vector operands (an exception is operators of access to elements of structures, see Section 2.3.11). Terminal types of the operands are subject to the same restrictions as the operand types in C. Types are transformed by the following scheme:

(1) operands are transformed to vector types of one form by means of a conformal extension;

(2) to the vector types obtained, transformations of terminal type are applied in accordance with the rules of the type transformation for the given operator, which are specified in the C standard.

The reverse order is also possible.

Specific features of type transformations for various operators are considered in detail in Section 2.3.

2.3. Expressions of the C[] Language

2.3.1. Lvectors. In C, the notion ***lvalue*** is used. It is defined as an expression denoting an object [11]. By analogy, in C[], the notion of an *lvector* is defined as an expression denoting a vector.

2.3.2. Vector construction. In addition to all C operators available in C[], operators on vectors are introduced. There are two operators designed for constructing a vector from an array or a pointer. These are *grid* and *block* operators.

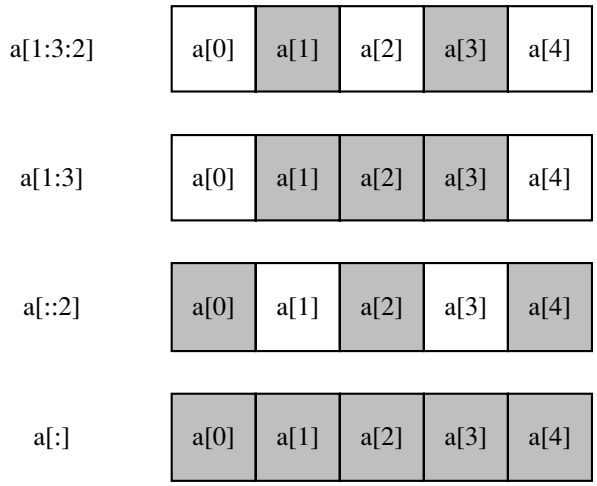

Fig. 2. Variants of the application of the grid operator to a one-dimensional array. The shaded squares correspond to the selected elements.

The grid operator has the following syntax:

$$expr[l : r : s],$$

where an expression *expr* has the type "array of elements of type *T*" or a "pointer to the type *T*" and operands *l*, *r*, and *s* are of integer type and denote the left boundary, the right boundary, and the step, respectively. The operands must satisfy the conditions: $l \le r$, $l \ge 0$, $r \ge 0$, and $s > 0$. The result of the expression is a vector v containing $\lfloor (r - l)/s \rfloor + 1$ elements of type *T*, with the value of the *i*th element of the vector v being the value of the expression $expr[l + s * i]$. If $e[l + s * i]$ is an address expression, then v_i denotes the same object in the memory as $expr[l + s * i]$.

The left and right boundaries of the grid, as well as the step, can be omitted. In this case, their values are taken by default. If the expression for the step is omitted, then the second colon in the notation for the grid operator must be omitted as well. The default value for the step is one; the default value for the left boundary is zero; and the default value for the right boundary either is equal to the number of the array elements minus one or, if the grid operator is applied to the pointer, is determined from the context.

Example. Figure 2 shows variants of the determination of the dropped parameters of the grid operator for an operand of the type "array."

If a grid operator is applied to a pointer, the default value *r* for the right boundary is determined from the context of the expression by the following rules:

(1) if the expression $expr[l : r : s]$ is one of the operands of a binary operator, and the other operand has the type "vector of length *N*," then *r* is set equal to $N - 1$;

(2) if the expression $expr[l : r : s]$ is one of the operands of a conditional operator, and one of the two

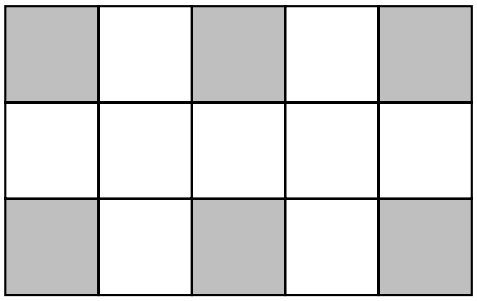

Fig. 3. Vector denoted by the expression `A[0:2:2][0:5:2]`, where `A` is described as `int A[3][5]`. The shaded squares correspond to the selected elements.

remaining operands has the type "vector of length *N*," then *r* is set equal to $N - 1$.

In all other cases, the default value of the right boundary cannot be calculated from the context, and the corresponding operand cannot be dropped.

The first operand of a grid operator may be an expression of a vector type. In this case, the grid operator is applied to elements of a vector value. This makes it possible to successively apply several grid operators and, thus, to extract different collections of elements from multidimensional arrays.

The expression *expr*[$l : r : s$] is an address vector. It is a modifiable address vector if all expressions *expr*[$l + s * i$] are modifiable address values or modifiable address vectors.

Example. Figure 3 shows an expression obtained by applying successively two grid operators to the array `A` defined as `int A[3][3]`.

The result of the expression `A[0:2:2][0:5:2]` is interpreted as follows. The expression `A[0:2:2]` denotes the vector consisting of two arrays of the type "array of five elements of the type `int`." This expression is a nonmodifiable *lvector*, since the expressions `A[0]` and `A[2]` are nonmodifiable *lvalues*. The second grid operator is applied to each element of this vector. Hence, the expression `A[0:2:2][0:5:2]` denotes a vector of two vectors that are obtained by applying the grid operator with the boundaries 0 and 5 and the step 2 to the array of five integers (i.e., both latter vectors consist of three integers). This expression is a modifiable address vector, since the result of the application of the grid operator to an array of five integers is a modifiable address vector.

Another operator of vector construction is the block operator with the syntax

expr[],

where the expression *expr* has an array or pointer type. If it has a pointer type, then the expression *expr*[] is equivalent to the expression *expr*[:]; if it has the array type, the expression *expr*[] is equivalent to the expression *expr*[:] … [:], where the grid operator with dropped expressions for the boundaries and step is applied the number of times that is equal to the rank of the array.

As in the case of the grid operator, an operand of the block operator may be an expression of vector type. In this case, the operator is applied element-wise to the elements of the vector value of the expression.

2.3.3. Cast operator. The syntax of the cast operator is given by

(*type_name*) *expression.*

The type *type_name* may be any scalar type of C[]. If the expression *expression* has a vector type, the cast operator is considered to be the cast of the terminal type of the expression value. By analogy with C, the cast expression is not an lvector.

2.3.4. Address operator. The operator & of getting an address is applicable to *lvectors* only. If an expression *expr* is an lvector of a type *T*, the value of the expression &(*expr*) is the value of the vector type with the form *Form*(*T*) and terminal type "pointer to *VTT*(*T*)." In this case, terminal elements of this vector value are pointers to the terminal elements of the vector denoted by the expression *expr.*

2.3.5. Indirection operator. An operand of the indirection operator * may be an expression of vector type. If an expression *expr* has a vector type *T* and *VTT*(*T*) is the type "pointer to the type *T*'," then the expression *(*expr*) is an address vector denoting the vector whose terminal elements are objects addressed by the terminal elements of the vector value of the expression *expr*. The expression *(*expr*) has the vector type with the form *Form*(*T*) and the terminal element *T*'.

2.3.6. Postfix increment and decrement operators. C[] permits the use of operands of vector type in the postfix increment and decrement operators. If the value of an expression *expr* is of vector type, the result of the expression *expr*++ is a vector value that coincides with the vector value of the expression *expr.* The expression *expr* must be lvector. Terminal elements of the vector denoted by the expression *expr* are increased by one. The postfix decrement operator is defined similarly.

2.3.7. Prefix increment and decrement operators. C[] permits the use of operands of vector type in the prefix increment and decrement operators. If the result of an expression *expr* has a vector type *T*, the result of the expression ++*expr* is a vector value of the form *Form*(*T*), terminal elements of which are obtained by applying the prefix operator ++ to the terminal elements of the vector value of the expression *expr*. The expression *expr* must be an address vector. Terminal elements of the vector denoted by the expression *expr* are increased by one. The prefix decrement operator is defined similarly.

2.3.8. Other unary operators. The unary operators +, –, ~, and ! may have vector operands. The result of the application of an operator *op* to an expression *expr*

of a vector type *T* is a vector value of the form *Form*(*T*) the terminal elements of which are obtained by applying the operator *op* to the corresponding terminal elements of the vector value *v* of the expression *expr*. On the value of the expression *expr*, transformations of terminal type defined in C for these operators are performed.

2.3.9. Assignment operators. The syntax of the assignment expression is given by

lexpr op rexpr, (1)

where *op* denotes one of the following assignment operators: =, *=, /=, %=, +=, –=, <<=, >>=, &=, ^ =, and |=. Operands of the assignment operator may have a vector type.

The left-hand side of the assignment expression is either a modifiable *lvalue* or a modifiable *lvector*. The expressions *lexpr* and *rexpr* must have conformal types, and the value type rank of the expression *lexpr* must be not less than the value type rank of the expression *rexpr*. The value of the expression *rexpr* is transformed to the type of the expression *lexpr* in accordance with the rules discussed in Section 2.2.4.

Let *l* be a vector denoted by the expression *lexpr* and *r* be a vector value of the expression *rexpr* transformed to the type of *lexpr*. As a result of a simple assignment operator, the vector value *r* replaces the value of the vector *l* in the memory. This replacement is implemented by elements; i.e., the values of the objects corresponding to the terminal elements of the vector *l* are replaced by the values corresponding to the terminal elements of the vector value *r*.

The expression for the complex assignment *lexpr op* = *rexpr* is equivalent to the expression *lexpr* = (*lexpr*)*op*(*rexpr*) with one exception that the expression *lexpr* is calculated only once.

Below is an example of the use of a binary assignment operator.

Listing 2.2. Elements of the *i*th row of the array A are assigned the value of the *i*th element of the array b.

```
int A[M][N];
int b[M];
…
…
A[:][:] = b[:];
```

2.3.10. Access to array elements. As in C, in C[], the expression $expr_1[expr_2]$ is equivalent to the expression $*((expr_1) + (expr_2))$, where the expressions $expr_1$ and $expr_2$ may have a vector type. This definition makes it possible to use vector values for addressing several array elements simultaneously.

Example. In the code presented in Listing 2.3, elements of the array A with indices 0, 1, and 3 are assigned the value of 1.

Listing 2.3. Elements of the array A with indices 0, 1, and 3 are assigned the value of 1:

```
int A[M];
int ind[3] = {0, 1, 3};
…
…
A[ind[:]] = 1;
```

2.3.11. Access to elements of structures and unions. The first operand of the operator "." may have a vector type the terminal type of which is a structure or union type, and the second operand is an identifier, a name of a structure or union member. The value of the expression *lexpr.rexpr* has a vector type of the same form as the type of *lexpr*, with the terminal type coinciding with the type of the named structure member.

The value of the expression *lexpr.rexpr* is a vector value the terminal elements of which are values of named members of the corresponding terminal elements of the vector denoted by *lexpr*. If the expression *lexpr* is lvector, then the expression *lexpr.rexpr* is an lvector the terminal elements of which are objects denoted by named members of the corresponding terminal elements of the vector denoted by *lexpr*.

The expression *lexpr*– > *rexpr* is equivalent to the expression (**lexpr*).*rexpr*.

2.3.12. Other binary operators. The set of C binary operators is extended in C[] through the addition of two new binary operators ?> and ?<. The result of the binary maximum operator ?> is the maximum of two operands. The restrictions on the operand types, the resulting type, and rules of the operand type transformations are the same as for the operator <.

Binary operators in C[] may have vector operands. Types of the operands must be conformal. If the type rank of one of the operands is less than the type rank of the other operand, the vector value of this operand is extended conformally to the type of the other operand. Then, to the values of both operands, the transformation of terminal types is applied in accordance with the C rules for the given binary operator.

A binary operator is applied element-wise to the corresponding elements of values of the vector operands. If *u* and *v* are results of the above-considered transformations of values of the operands *lexpr* and *rexpr*, the value of the expression *lexpr op rexpr* is a vector value *w* the terminal elements of which are obtained by applying the operator to the corresponding elements of the vector values *u* and *v*.

The application of binary operators to vector operands is exemplified by a code for the calculation of the element-wise sum of two arrays shown in Listing 2.4.

Listing 2.4. A fragment of the C[] code for the calculation of the element-wise sum of two arrays.

```
int a[M][N], b[M][N], c[M][N];
c[] = a[] + b[];
```

2.4. Reduction Operators

The binary C operators +, *, |, &, ^, ||, &&, ?>, and ?< are associated with the *reduction operators* in C[].

Table 1. Base operations of Single Assignment C

dim (A)	Dimension of the array ^{A}
shape (A)	Form vector of the array ^{A}
psi (v, A)	Subarray (or element) of the array ^{A} corresponding to the index vector ^{v}
take (v, A)	Subarray of the array ^{A} of form ^{v} consisting of elements of the array ^{A} taken for the index vectors that are not greater than ^{v}
drop (v, A)	drop (v, A) = take (v – shape (A), A)
reshape (v, A)	Array of form ^{v} consisting of elements of the array ^{A}
cat (m, A, B)	Concatenation of the arrays ^{A} and ^{B} along the dimension ^{m}
rotate (m, n, A)	Rotation of elements of the array ^{A} along axis ^{m} by ^{n} positions

If *op* is a binary operator, the reduction operator corresponding to it is denoted as [*op*]. It has the following syntax:

red_oper expr,

where *red_oper* stands for one of the reduction operators `[+]`, `[*]`, `[|]`, `[&]`, `[^]`, `[||]`, `[&&]`, `[?>]`, `[?<]`, and *expr* is an expression of vector type. If the expression *expr* has type *T*, then the reduction operator [*op*] is applicable to the expression *expr* if the corresponding binary operator *op* is applicable to operands of type *VET*(*T*). The type of the value of the expression [*op*]*expr* coincides with the type of the value of type *VET*(*T*). On a vector type *T*, the same transformation of terminal type is performed as for the type *VET*(*T*) in the case of a binary operator *op* with the operands of type *VET*(*T*).

The result of the application of an operator [*op*] to an expression *expr* is a value calculated by the formula $(\ldots(v_0\ op\ v_1)\ op\ v_2\ldots)op\ v_{n-1}$, where v is the result of the expression *expr*.

The reduction operators `[>?]` and `[<?]` corresponding to the maximum and minimum operators make it possible to find maximal and minimal values of the elements of a vector. A code fragment presented in Listing 2.5 illustrates the application of the combination of the element-wise multiplication and the reduction sum operator for the calculation of the scalar product of vectors.

Listing 2.5. Scalar multiplication of `a[]` and `b[]`.

```
int a[N];
int b[N];
int c;
…
…
c = [+](a[] * b[]);
```

2.5 Differences of the Current Version of the Language from That Published Earlier

During the time that passed after the first publication [6], C[] has been updated, with the updates being aimed at improving its expressiveness. The basic two of them are as follows. First, in the descriptions of arrays, the expression for the array dimension may now be of arbitrary integer type rather than only constant; i.e., dynamic arrays have been incorporated into the language. Second, the semantics of the grid operator has been changed; now, it is closer to that accepted in FORTRAN 90.

Operators of access to elements of structures and unions are extended to the vector case.

Notions of conformal vector types and conformal extensions have been introduced, which made it possible to more rigorously describe the semantics of expressions.

The operators *pipe* and *par* for explicit constructing vector values have been excluded from the language.

3. COMPARISON WITH OTHER VECTOR LANGUAGES

Unlike C and C[], which consider multidimensional arrays as “arrays of arrays,” all languages to be discussed in this section consider an array as an aggregate of a *data vector* and a *form vector.* The form vector is a vector consisting of integers the number of elements of which is equal to the number of the array dimensions, and the elements themselves are equal to the values of the corresponding dimensions. The data vector contains all array elements.

3.1. Single Assignment C

The SAC (Single Assignment C) language [4] is based on C but is not its strict extension. It does not have such important C constructs as pointers and global variables.

The notion of an array in the Single Assignment C is different from that in C. An array is considered as an aggregate of a data vector and a form vector. The description of an array in the SAC is syntactically different from the C description (see Listing 3.1).

The language includes a number of built-in operators on arrays (Table 1) that are used to describe various calculations on arrays.

The basic means to express calculations on arrays in the SAC are `with` loops. The heading of such a loop contains the description of the range of index variation, and the body contains one of three—`genarray`, `modarray`, and `fold`—statements, which are used to describe various expressions over arrays (Table 2).

The heading of a `with` loop has the syntax `with (a <= v <= b)step s width w` and specifies the

set of index vectors v such that $a_i <= v_i <= b_i$, where $(v_i - a_i) \bmod s_i < w_i$.

For an example of using with loops, we consider a SAC code calculating the element-wise sum of two arrays.

Listing 3.1. SAC code for array summation.

```
double[100] A;
double[100] B;
double[100] C;
C = with (. <= i_vec <= .)
      modarray(A, i_vec, A[i_vec] +
      B[i_vec]);
```

The Single Assignment C has a greater set of operators on arrays compared to C[]. In particular, built-in operators for array concatenation and rotation are not available in C[]. Moreover, the with construct is a more general means for the expression of calculations than just successive application of the reduction and binary operators used in C[]. For example, the construct

```
C = with (. <= i_vec <= .)
      modarray(A, i_vec, f(B[i_vec]));
```

which assigns to the elements of an array A the result of the application of a function f to the elements of an array B, cannot be replaced by one vector expression in C[].

The incorporation of many built-in operators on arrays and with loops made the developers of the SAC exclude such key elements of C as pointers and global variables. Unlike the SAC, C[] completely preserves the syntax and semantics of C, which allows one to use an earlier written C code and facilitates the learning of the language for the programmers working with C.

In addition, the existence of many built-in functions is inconvenient for the user, who must either remember them or permanently address the manual. It is for this reason that C[] contains a minimum number of new syntactic constructs, which have a natural form and, thus, are easy-to-remember.

3.2. ZPL

The ZPL language [3] is an extension of a subset of Modula-2. To simplify the compilation and facilitate the use of optimizing transformations, pointers and unconditional transfers are excluded from the base language.

The ZPL language supports traditional arrays used in Modula-2, which are referred to as *indexed arrays* in ZPL. In addition to the indexing operator, the so-called *absent array reference* operator, which is syntactically represented as square brackets with dropped index expression, is introduced in the language. Being applied the number of times that is equal to the number of the array dimensions, this operator makes it possible to address the array as a whole. Such an expression may be used as an operand of a binary or unary operators, which, in this case, are applied by elements. For operands of a binary operator, only arrays of the same form can be used, or one of the operands is to be a scalar. Listing 3.2 gives an example of a ZPL program for finding the sum of arrays.

Table 2. Single Assignment C statements

genarray (v, expr)	Returns an array of form ^{v} with the elements calculated by the expression ^{expr}
modarray (A, v, expr)	Returns a modified version of the array ^{A}: on the positions corresponding to ^{v}, elements of the vector are equal to the values of the expression ^{expr}
fold (fold_op, neutral, expr)	Calculates the reduction of all elements from the range of indices by means of the binary operator $^{fold\_op}$

Listing 3.2. ZPL code for the summation of arrays.

```
type array1 = array[1..10, 1..5] of
integer;
var a, b, c : array1;
c[][] := a[][] + b[][];
```

The support of calculations over arrays in the ZPL language is considerably poorer than in C[]: no reduction operators are available in ZPL, and element-wise operators can be applied only to arrays of the same form.

A wider set of operators are provided in ZPL for the so-called parallel arrays. Parallel arrays are distributed in the computational space such that each node of the computational space gets a certain number of the array elements and each element belongs to only one node. Parallel arrays also may be operands of arithmetic and assignment operators, as well as operands of reductions. Since parallel arrays are means for expressing program parallelism, we will not further discuss them: possibilities of ZPL related to its parallel implementation are reasonable to compare with those of mpC rather than C[], which suggests a one-processor target platform.

3.3. FORTRAN 90

One of the most well-known modern FORTRAN extensions, FORTRAN 90, supports various operations on arrays. This language seems to be conceptually closest to C[], and we consider it in a more detail.

FORTRAN 90 permits the use of arrays as operands of binary and unary operators, which are applied element-wise. Operands of a binary operator must have the same form. The result of an operator is an array of the same form as the operands.

FORTRAN 90 contains built-in functions for constructing arrays. For example, the function RESHAPE is used to obtain an array from elements of another array. The function SPREAD is designed for constructing an array whose dimension is greater than that of a given array by one, by creating the required number of

Table 3. FORTRAN 90 reduction statements

ALL	Returns true if all elements of the array are equal to true
ANY	Returns true if any of the elements of the array are equal to true
COUNT	The number of the array elements
MAXVAL	The maximum element of the array
MINVAL	The minimum element of the array
PRODUCT	Product of the array elements
SUM	Sum of the array elements

copies of the given array and placing them along the given dimension.

Reduction operators are realized through built-in functions listed in Table 3. The reduction functions can be applied either to the whole array to get a scalar result or along a given dimension to get an array whose rank is less than the rank of the operand by one.

FORTRAN 90 has facilities for addressing segments of arrays. Within one dimension, boundaries of the segment are indicated by means of the expression `l : r : s`, where `l` and `r` are the left and right boundaries and `s` is the selection step.

There are a number of other built-in functions for work with arrays, including the functions `DOT_PRODUCT` and `MATMUL` designed for the calculation of the scalar product of vectors and the product of a matrix and a vector, respectively.

Listing 3.3 presents a fragment of a code in FORTRAN 90 computing the sum of two arrays.

Listing 3.3. FORTRAN 90 code for the summation of arrays.

```
REAL DIMENSION (5,5) :: A
REAL DIMENSION (5,5) :: B
REAL DIMENSION (5,5) :: C
C = A + B
```

C[] and FORTRAN 90 are similar in many respects; for example, binary operators are defined for operands–arrays, it is possible to address segments of arrays, etc.

Still, there are a number of differences between them. In particular, reduction operators in FORTRAN 90 are implemented as built-in functions, whereas, in C[], they are denoted by the symbol of the corresponding binary operator enclosed in square brackets. The C[] approach seems to be more convenient, since the language is not overloaded by additional key words. Moreover, the number of the reduction operators in C[] is greater than that in FORTRAN 90.

4. CONCLUSION

The C[] language considered in the paper extends the ANSI C through the support of element-wise and reduction operators on vectors. These operators are convenient-to-use means for writing codes containing computations over array data. The approach based on the C[] language is original, and, to our best knowledge, is different from other approaches to the development of vector languages.

In the future, we plan to improve the compiler from C[] by adding to it optimizing transformations aimed at the minimization of time expenditures and more efficient use of interprocessor parallelism.

A free version of the C[] compiler and user's manual are available in the Internet at the address www.ispras.ru/~cbr. For the compilation of codes in C[], the compiler from mpC can also be used, the free version of which is also available in the Internet at the site www.ispras.ru/~mpc.

APPENDIX. EXAMPLES OF C[] CODES

A.1 Matrix Multiplication

In this example, the product of matrices `A` and `B` is calculated, and the result is written to matrix `C`. The algorithm used is based on the fact the *i*th row of matrix `C` is a linear combination of rows of `B` with the coefficients equal to the elements of the *i*th row of `A`.

Listing A.1. Matrix multiplication.

```
double A[M][L];
double B[L][N];
double C[M][N];

for(i = 0; i < M; i ++)
    C[i][] = [+](A[i][] * B[]);
```

A.2 LU Decomposition

The listing below presents a fragment of the code for the LU decomposition of a matrix `A` by the Gauss method.

Listing A.2. LU decomposition.

```
for(i = 0; i < N–1; i ++)
  for(j = i+1; j < N; j ++) {
    double t;
    t = A[j][i]/A[i][i];
    if(A[j][i]!= 0) A[j][i:N–1]–=
    t*A[i][i:N–1];
  }
```

REFERENCES

1. Higham, D. and Higham, N., *The Matlab Guide*, SIAM, 2000.
2. *ISO, Programming Language APL*, extended, 1996.
3. Lin, C. and Snyder, L., ZPL: An Array Sublanguage, in *Languages and Compilers for Parallel Computing*, 1993, pp. 96–114.
4. Scholz, S.-B., On defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities, *Proc. of APL'98*, ACM-SIGAPL, 1998, pp. 40–45.

5. *ISO & IEC, FORTRAN 95*, X3J3/95-007R1, Working Draft, 1996.
6. Gaissaryan, S.S. and Lastovetsky, A.L,, ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler, *J. C Language Translation*, 1994, vol. 5, no. 3, pp. 183–198.
7. Gaissaryan, S.S. and Lastovetsky, A.L., Extension of ANCI C for Vector and Superscalar Computers, *Programmirovanie*, 1995, vol. 21, no. 1.
8. Lastovetsky, A., Arapov, D., Kalinov, A., and Ledovskih, I., A Parallel Language and Its Programming System for Heterogeneous Networks, *Concurrency: Practice and Experience*, 2000, vol. 12, pp. 1317–1343.
9. Kalinov, A.Ya., Lastovetsky, A.L., Ledovskih, I.N., and Posypkin, M.A., Compilation of Vector Statements of C[] Language for Architectures with Multilevel Memory Hierarchy, *Programmirovanie*, 2001, vol. 26, no. 3, pp. 3–18.
10. *ISO & IEC, Programming Language C*, WG14/N843, Committee Draft, 1998.
11. Kernighan, B.W. and Ritchie, M., *The C Programming Language*, Englewood Cliffs (USA): Prentice–Hall, 1978. Translated under the title *Yazyk programmirovaniya Ci*, St. Petersburg: Nevskii Dialekt, 2001.