Computing 39, 93–110 (1987)

Computing

FORTRAN-SC
A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH

J. H. Bleher and S. M. Rump, Böblingen
U. Kulisch, M. Metzger, Ch. Ullrich and W. Walter, Karlsruhe

Received June 16, 1987; revised August 7, 1987

Abstract — Zusammenfassung

FORTRAN-SC. A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH. A new programming language called FORTRAN-SC is presented which is closely related to FORTRAN 8x. FORTRAN-SC is a FORTRAN extension with emphasis on engineering and scientific computation. It is particularly suitable for the development of numerical algorithms which deliver highly accurate and automatically verified results. The language allows the declaration of functions with arbitrary result type, operator overloading and definition, as well as dynamic arrays. It provides a large number of predefined numerical data types and operators. Programming experiences with the existing compiler have been very encouraging. FORTRAN-SC greatly facilitates programming and in particular the use of the ACRITH subroutine library [14], [15].

Key words: Programming languages, FORTRAN, compiler, computer arithmetic, numerical computation, verified numerics.

FORTRAN-SC. Eine FORTRAN-Erweiterung für naturwissenschaftlich-technisches Rechnen mit Zugriff auf ACRITH. FORTRAN-SC ist eine neue Programmiersprache, welche mit FORTRAN 8x eng verwandt ist. Es handelt sich um eine FORTRAN-Erweiterung für Anwendungen im naturwissenschaftlich-technischen Bereich. Insbesondere eignet sich FORTRAN-SC für die Entwicklung von numerischen Algorithmen, welche hochgenaue und automatisch verifizierte Ergebnisse liefern. Die Sprache ermöglicht die Vereinbarung von Funktionen mit allgemeinem Ergebnistyp, das Überladen und Definieren von Operatoren, sowie dynamische Felder. Außerdem wird eine große Zahl vordefinierter numerischer Datentypen und Operatoren zur Verfügung gestellt. Die bisherigen Programmiererfahrungen mit dem existierenden Compiler sind sehr vielversprechend. FORTRAN-SC vereinfacht das Programmieren und insbesondere die Benutzung der ACRITH-Unterprogrammbibliothek wesentlich [14], [15].

1. Introduction

In electronic computers the elementary arithmetic operations are these days generally approximated by floating-point operations of highest accuracy. This is one of the essential intentions of the ANSI/IEEE Floating-Point Arithmetic Standard 754 [5]. This arithmetic standard also requires the four basic arithmetic operations

$+, -, *, /$ with directed roundings. A large number of processors is already on the market which provide these operations. So far, however, no common programming language allows access to these operations.

On the other hand, there is a noticeable shift from general purpose machines to vector processors and parallel computers in scientific computation. These so-called super-computers provide additional arithmetic operations such as "multiply and add", "accumulate" or "multiply and accumulate" [12]. All of these hardware operations should always deliver a result of highest accuracy for all possible combinations of data. As far as we know, no processor which fulfills this requirement is as yet available. From the point of view of programming, there exists no standard language which permits direct access to these operations.

There is a continuous effort to enhance the power of programming languages. New powerful languages like ADA have been designed, and the development of existing languages like FORTRAN is constantly in progress. However, since these languages still lack a precise definition of the arithmetic, the same program may produce different results on different processors.

During the development of FORTRAN 8x, proposals were made as to how real and complex operations with directed roundings, interval and complex interval arithmetic, and vector/matrix operations for all these types could be incorporated into that language [8], [9]. Many useful concepts which are necessary for their realization have been adopted in recent drafts of the proposed FORTRAN 8x standard. Such concepts are: derived data types, dynamic arrays, functions with arbitrary result type, operator overloading and definition, and modules.

In this paper, we refer to the programming language described by the latest draft of the proposed FORTRAN 8x standard simply as FORTRAN 8x [4].

In 1983, a group of scientists[1] worked out the first draft of a programming language which is closely related to FORTRAN 8x. In particular, it is a superset of FORTRAN 77 [3]. The new language was given the name FORTRAN-SC. In this paper, we will give an informal description of this language. We will also make some remarks on its current implementation. FORTRAN-SC pursues the same goals as PASCAL-SC [10], [17], another programming language for scientific computation which was designed and implemented at the Institute for Applied Mathematics at the University of Karlsruhe.

With respect to scientific computation the newly designed language surpasses FORTRAN 8x. For example, FORTRAN-SC provides more than 1000 predefined arithmetic operators for all kinds of numerical data types. All of these operators are required to deliver a result of at least 1 ulp accuracy. This means that the error is less than 1 *u*nit in the *l*ast *p*lace (1 ulp). In other words, there is no floating-point number between the computed and the exact result. Since our interest is mainly directed towards scientific computation, several features of FORTRAN 8x which are of minor importance for that purpose were not incorporated into FORTRAN-SC.

---

[1] J. H. Bleher, H. Böhm, G. Bohlender, A. T. Gerlicher, K. Grüner, E. Kaucher, R. Klatte, U. W. Kulisch, W. L. Miranker, M. Neaga, S. M. Rump, Ch. Ullrich, J. Wolff von Gudenberg.

In FORTRAN-SC, the mathematical definition of the arithmetic is part of the language. This definition includes the vector/matrix operations. The elementary arithmetic operations $+, -, *, /$ with directed roundings are axiomatically defined and directly accessible in the language. All other arithmetic operations, in particular for intervals, are defined according to the rules of semimorphism [18].

During the process of implementation, the new language was further developed and completed. The compiler consists of a complete front end performing full analysis of the source language and a code generator. To achieve high portability, the code generator produces FORTRAN 77 code. The extensive runtime library contains the predefined operators and intrinsic functions of FORTRAN-SC. All implemented arithmetic operators are accurate to at least 1 ulp.

Programming experiences in FORTRAN-SC have been very encouraging. As a result of the operator notation in expressions for all arithmetic data types, programs become clearer and more concise. FORTRAN-SC programs are easy to read, write and understand.

One of the main goals of the development of FORTRAN-SC was to facilitate the use of the ACRITH subroutine library. FORTRAN-SC makes all ACRITH subroutines which provide arithmetic operations available as predefined operators. Furthermore, the large number of ACRITH mathematical standard functions for real, complex, interval and complex interval data may be referenced by their specific or generic names in FORTRAN-SC.

Through the availability of the interval, vector and matrix data types, the use of the ACRITH problem solving routines is greatly simplified. It is a common characteristic of all ACRITH routines that whenever they deliver a result, it is verified to be correct by the computer.

The language FORTRAN-SC was developed and implemented in a collaboration of the IBM Development Laboratory in Böblingen, Federal Republic of Germany, with the Institute for Applied Mathematics at the University of Karlsruhe.

2. The Language FORTRAN-SC

In traditional programming languages like FORTRAN, ALGOL or PASCAL, each vector/matrix operation such as matrix multiplication or vector addition requires an explicit loop construct or a call to an appropriate procedure. The same is true for all operations with a non-scalar (or structured) result, e.g. for interval arithmetic. To avoid such difficulties, the language FORTRAN-SC provides all vector and matrix operations in the commonly used linear spaces (the real and complex numbers, real and complex vectors, real and complex matrices as well as all the corresponding interval spaces) as predefined operators. All of these operations are accessible through their usual operator symbol. Thus, expressions of these types can be written in the usual mathematical notation.

Additionally, a general operator concept is available in FORTRAN-SC. It enables the user to define his own operators for old and new data types. Other modern

programming languages like ADA and FORTRAN 8x also provide an operator concept and the possibility to write non-scalar expressions. Thus, all of these languages provide the necessary tools for writing expressions in the common linear spaces in mathematical notation. The difference is, however, that FORTRAN-SC requires all predefined operators to deliver results of 1 ulp accuracy for all possible combinations of the data. The implemented runtime library satisfies this requirement.

Moreover, FORTRAN-SC provides means for computing certain classes of vector and matrix expressions with 1 ulp accuracy. In contrast, vector/matrix operations and expressions evaluated in the traditional manner will not, in general, deliver high accuracy.

It is an essential idea of FORTRAN-SC that the user need not perform an error analysis for any basic vector/matrix operation provided by the language. As a matter of fact, this should be a natural requirement for any modern programming language. Whenever a language provides a higher-dimensional operation by an operator symbol, the result should be of 1 ulp accuracy (as required in FORTRAN-SC). Otherwise, an error message should be given.

2.1. Standard Data Types, Predefined Operators and Functions

The following scalar data types are available in FORTRAN-SC:

INTEGER, REAL, DOUBLE REAL, COMPLEX, DOUBLE COMPLEX, INTERVAL, DOUBLE INTERVAL, COMPLEX INTERVAL, DOUBLE COMPLEX INTERVAL, LOGICAL, CHARACTER.

For these basic data types, arrays can be declared in the usual manner. For a large number of numerical array types, operators are predefined which always deliver 1 ulp accuracy. This means that the traditional arithmetic operators $+, -, *, /$ are to be implemented with the rounding to the nearest machine-representable element. The newly introduced arithmetic operators $+<, -<, *<, /<$ and $+>, ->, *>, />$ are to be implemented with the monotone downwardly directed and upwardly directed rounding, respectively. For real and double real scalar data the latter operations are also provided by the ANSI/IEEE Arithmetic Standard 754 [5].

Tables 1 and 2 show all predefined arithmetical and relational operators. We use the following abbreviations for the basic numerical data types:

R – REAL
DR – DOUBLE REAL
C – COMPLEX
DC – DOUBLE COMPLEX
I – INTERVAL
DI – DOUBLE INTERVAL
CI – COMPLEX INTERVAL
DCI – DOUBLE COMPLEX INTERVAL

Table 1. *Predefined arithmetic operators*

right operand / left operand	INTEGER	R DR C DC	I DI	CI DCI	RVEC DRVEC CVEC DCVEC	IVEC DIVEC CIVEC DCIVEC	RMAT DRMAT CMAT DCMAT	IMAT DIMAT CIMAT DCIMAT
[1])	{+, −}	{+, −}	{+, −}		{+, −}	{+, −}	{+, −}	{+, −}
INTEGER	λ	Λ	λ		Π	{*}	Π	{*}
R DR C DC	Λ	Λ			Π		Π	
I DI	λ		Γ			{*}		{*}
CI DCI			ζ					
RVEC DRVEC CVEC DCVEC	Θ	Θ			Ω			
IVEC DIVEC CIVEC DCIVEC	{*, /}		{*, /}		Φ			
RMAT DRMAT CMAT DCMAT	Θ	Θ			Π		Ω	
IMAT DIMAT CIMAT DCIMAT	{*, /}		{*, /}			{*}		Φ

[1]) The operators in this row are monadic (i.e. no left operand).

$\lambda := \{+, -, *, /, **\}$
$\Lambda := \{+, +\langle, +\rangle, -, -\langle, -\rangle, *, *\langle, *\rangle, /, /\langle, /\rangle, **\}$
$\Pi := \{*, *\langle, *\rangle\}$
$\Theta := \{*, *\langle, *\rangle, /, /\langle, /\rangle\}$
$\Omega := \{+, +\langle, +\rangle, -, -\langle, -\rangle, *, *\langle, *\rangle\}$
$\Phi := \{+, -, *, .IS., .CH.\}$
$\zeta := \{+, -, *, /, .IS., .CH.\}$
$\Gamma := \{+, -, *, /, .IS., .CH., **\}$

.IS.: Intersection of two intervals
.CH.: Convex hull of two intervals

Table 2. *Predefined relational operators*

right operand / left operand	INTEGER	R DR	C DC	I DI CI DCI	RVEC DRVEC CVEC DCVEC	IVEC DIVEC CIVEC DCIVEC	RMAT DRMAT CMAT DCMAT	IMAT DIMAT CIMAT DCIMAT
INTEGER	Ψ	Ψ	Σ	Ξ				
R DR	Ψ	Ψ	Σ	Ξ				
C DC	Σ	Σ	Σ	Ξ				
I DI CI DCI				Δ				
RVEC DRVEC CVEC DCVEC					Σ	Ξ		
IVEC DIVEC CIVEC DCIVEC						Δ		
RMAT DRMAT CMAT DCMAT							Σ	Ξ
IMAT DIMAT CIMAT DCIMAT								Δ

$\Xi := \{.IN.\}$
$\Sigma := \{.EQ., .NE.\}$
$\Delta := \{.EQ., .NE., .SB., .SP., .DJ.\}$
$\Psi := \{.EQ., .NE., .LT., .LE., .GT., .GE.\}$

.SB.: Subset for two intervals
.SP.: Superset for two intervals
.DJ.: Disjoint intervals
.IN.: Membership of a point in an interval

The suffixes VEC and MAT are abbreviations for one-dimensional and two-dimensional arrays, respectively.

Tables 1 and 2 are very compact. For instance, the symbol Ω may be substituted by any operator listed in the set Ω. Furthermore, each operator of the set Ω may be

applied to all type combinations listed in the corresponding rows and columns. So each occurrence of Ω in Table 1 represents 144 ($=9 * 4 * 4$) predefined operators.

Compared to FORTRAN 77, FORTRAN-SC provides an extended set of mathematical standard functions (see Table 3). All these functions are available for the basic data types real, complex, interval and complex interval in single and double precision. They can be referenced by their specific or their generic name. FORTRAN-SC requires the mathematical standard functions with a point result to be accurate to within 1 ulp. The interval functions must be accurate to within 2 ulps. In the implemented runtime library, the actual error bounds are usually only half as large. Only in rare cases will the error be slightly greater – but always within the prescribed bounds.

Table 3. *Mathematical standard functions*

	Function	Generic Name
1	Natural Logarithm	LOG
2	Common Logarithm	LOG10
3	Exponential	EXP
4	Sine	SIN
5	Cosine	COS
6	Tangent	TAN
7	Cotangent	COT, COTAN
8	Arcsine	ASIN
9	Arccosine	ACOS
10	Arctangent	ATAN
11	Arccotangent	ACOT
12	Arctangent (x1/x2)	ATAN2
13	Hyperbolic Sine	SINH
14	Hyperbolic Cosine	COSH
15	Hyperbolic Tangent	TANH
16	Hyperbolic Cotangent	COTH
17	Inverse Hyperbolic Sine	ARSINH
18	Inverse Hyperbolic Cosine	ARCOSH
19	Inverse Hyperbolic Tangent	ARTANH
20	Inverse Hyperbolic Cotangent	ARCOTH
21	Square Root	SQRT
22	Square	SQR
23	Absolute Value	ABS
24	Argument of a Complex Number	ARG

Besides the mathematical standard functions, FORTRAN-SC provides all the necessary type transfer functions for conversion between the numerical data types. They exist for scalar and array types.

2.2. Dynamic Arrays

As an extension to FORTRAN 77, the concept of dynamic arrays is introduced. This greatly extends the capabilities supplied by conventional FORTRAN arrays, called static arrays.

Dynamic arrays provide the user with the capability of allocating or freeing storage space for an array during execution of a program. Thus, the same program may be used for arrays of any size without recompilation. Furthermore, storage space can be employed economically since only the arrays currently needed have to be kept in storage and since they always use exactly the space required in the current problem. Also, type compatibility and full storage access security are offered for dynamic arrays. Note that the concepts of assumed size arrays and adjustable arrays become obsolete. Dynamic arrays offer the same functionality while being much more versatile.

In FORTRAN 77, arrays whose dimensions are unknown a priori are implemented via pseudo-dynamic mechanisms. This means that a sufficently large work area must be provided by the main program to handle the pseudo-dynamic objects, like vectors and matrices.

These and many other disadvantages are avoided through the use of dynamic arrays. We list a few advantages of dynamic arrays:

- storage space used only when needed,
- array size may change during execution,
- no recompilation for arrays of different sizes,
- complete type and index checking,
- no extra arguments for array dimensions,
- no user-defined array workspace,
- no module space for dynamic array storage.

The DYNAMIC statement is used to declare named array types and/or to declare dynamic arrays.

An array type is characterized by the (scalar) data type of the array elements and the number of dimensions of the array. We call this information (i.e. element type and number of dimensions) the *array form* or simply the *form* of a (dynamic or static) array. Note that the size of an array is not part of this information.

An array form can be given a name or several distinct names, each identifying a different named array type. The type of a dynamic array may simply be specified as an array form, or it may be specified by an array type name.

Example: Declaration of named array types and dynamic arrays:

```
DYNAMIC/REAL (:)/A, B
DYNAMIC/VECTOR=REAL (:)/V, W, /MATRIX=COMPLEX (:, :)/
DYNAMIC/MATRIX/M, /POLYNOMIAL=REAL (:)/P, Q
```

These statements declare A, B, V, W, P, Q as real one-dimensional dynamic arrays and M as a complex two-dimensional dynamic array. Note that VECTOR and POLYNOMIAL are two different named array types even though they are used for arrays of the same form. Thus, A, V, and P all have different data types.

In order to obtain storage space for a dynamic array, an ALLOCATE statement can be executed which specifies the index range for each dimension of the array. The storage space of a dynamic array is deallocated by a FREE statement.

Example: Allocation and deallocation of dynamic arrays:

```
DYNAMIC/DOUBLEMATRIX=DOUBLE REAL (:, :)/ A, B, C
READ(*,*) I
ALLOCATE A, B (I : 2 * I, 10)
...
C=A+B
FREE A
ALLOCATE A (20, 20)
```

An existing (allocated) dynamic array may be reallocated by an ALLOCATE statement without prior execution of a FREE statement. Thus, in the above example, the FREE statement is optional. In this manner the same array variable can be changed in size during execution. Note that its contents are lost when doing this. Deallocating a non-allocated array has no effect.

Furthermore, allocation of a dynamic array occurs automatically when assigning the value of an array expression to a non-allocated array (e.g. in the statement C=A+B in the example above).

The storage of a dynamic array which is local to a subprogram is automatically released before control returns to the calling program unit unless the array name occurs in a SAVE statement. Obviously, a static array may neither be allocated nor deallocated.

Array inquiry functions facilitate the use of static and dynamic arrays. In particular, the functions LB and UB provide access to the lower and upper index bounds of an array.

2.3. Array-Valued Functions and User-Defined Operators

In most programming languages the result of a function has to be a single scalar value. In addition, FORTRAN-SC allows functions which return a dynamic array as result. Thus, the user is no longer forced to write a subroutine instead of an array-valued function.

This concept allows functions with a result array whose size is unknown to the calling program even at the time it is calling the function. In general, only the function itself knows the size of its result. It is therefore always the function's responsibility to allocate the dynamic result array. Of course, allocation of the result may be taken care of by array assignment inside the function (as in the example below).

The type of an array function is defined by declaring the function name like a dynamic array.

Example:

```
C   This function multiplies the real R with the vector W and
C   substracts the resulting vector from the vector V.
    FUNCTION RVFUN (R, W, V)
    REAL R
    DYNAMIC /REAL(:)/ V, W, RVFUN
    RVFUN=V-R * W
    END
```

In the calling program unit, the function name RVFUN must be declared as a real one-dimensional dynamic array in a DYNAMIC statement. In addition, the function *must* be declared to be an external routine. Thus, in the calling unit, the function name must appear in an EXTERNAL statement or in an OPERATOR statement as the implementing function of a user-defined operator.

For some applications it may be useful and more convenient to introduce operators. In FORTRAN-SC, an operator is defined by an operator symbol or name, the number and type(s) of its operand(s) and the implementing function. The OPERATOR statement is used to declare such user-defined operators. In this way, an external function with one or two arguments can be called as a monadic or dyadic operator, respectively.

In an expression, an operator is uniquely determined by the operator symbol or name, by its appearance as a monadic or dyadic operator, and by the type(s) of its operand(s).

Example: Definition and usage of an operator for the dyadic product of two real vectors:

```
PROGRAM MAIN
INTEGER DIM
DYNAMIC /REAL(:)/ A, B, /REAL(:, :)/ C
OPERATOR .MUL.=DYPROD (REAL(:), REAL(:)) REAL(:, :)
READ(*, *) DIM
ALLOCATE A, B (1 : DIM)
...
C=A .MUL. B
...
END
```

```
      FUNCTION DYPROD (X,Y)
      DYNAMIC /REAL(:)/ X, Y, /REAL(:,:)/ DYPROD
      ALLOCATE DYPROD (LB(X):UB(X), LB(Y):UB(Y))
      DO 1Ø i=LB(X), UB(X)
         DO 1Ø j=LB(Y), UB(Y)
1Ø          DYPROD(i,j)=X(i)*Y(j)
      END
```

All standard operator symbols and names may be overloaded and/or redefined in this way. In the example above, if the operator symbol * were to be used instead of the user-defined operator name .MUL., then the predefined multiplication operator for two real vectors (the inner product) would no longer be accessible within the program unit MAIN.

Table 4 summarizes the intrinsic operator symbols and names and displays the priorities of all FORTRAN-SC operators. Note that the user is free to invent his own operator names (enclosed in periods as in FORTRAN 8x [4]).

Table 4. *Precedence of intrinsic and user-defined operators*

Priority		Operators
high	12	user-defined monadic operators
	11	**
	10	* / *< /< *> /> .IS.
	9	monadic + monadic −
	8	+ − +< −< +> −> .CH.
	7	//
	6	.LT. .LE. .EQ. .GE. .GT. .NE. .SB. .SP. .DJ. .IN.
	5	.NOT.
	4	.AND.
	3	.OR.
	2	.EQV. .NEQV.
low	1	user-defined dyadic operators

Overloading or redefining intrinsic operator symbols and names does not change their priority. Note that the operator priorities in Table 4 are the same as in FORTRAN 8x [4].

The possibility to introduce different named array types for the same array form allows the definition of different operators with the same operator symbol (or name) for operands of the same form.

Example: Replacing the DYNAMIC and the OPERATOR statement in program MAIN in the preceding example by

```
DYNAMIC /COLUMN=REAL(:)/ A
DYNAMIC /ROW=REAL(:)/ B, /REAL(:,:)/ C
OPERATOR *=DYPROD (COLUMN, ROW) REAL (:,:)
```

will have the effect of overloading the operator * for a new type combination. The standard multiplication operator for two real one-dimensional arrays (the inner product) will then still be accessible.

2.4. Evaluation of Expressions with High Accuracy

FORTRAN-SC provides a large number of predefined numerical operators and intrinsic functions. Although all of these primitives are highly accurate, expressions composed of several such elements do not necessarily yield results of high accuracy. However, techniques have been developed to evaluate numerical expressions with high and guaranteed accuracy.

A simple class of such expressions are the so-called dot product expressions. We distinguish three kinds which differ in their result form: scalar, vector and matrix dot product expressions. Each such expression consists of a sum where the terms are single elements of this form or single products which deliver results of this form. Examples of such expressions are:

s1+s2*s3−v1*v2	of scalar form
v1+m1*v2−s1*v3	of vector form
m1−m2*m3+s1*m4	of matrix form

where s1, s2, s3 are scalars, v1, v2, v3 are vectors and m1, m2, m3, m4 are matrices with matching dimensions. The element types may be REAL, DOUBLE REAL, COMPLEX and DOUBLE COMPLEX.

The language FORTRAN-SC provides a special notation which indicates that a dot product expression is to be evaluated with 1 ulp accuracy. To obtain the unrounded or correctly rounded result of a dot product expression, the user has to parenthesize the expression and precede it by the symbol # which may optionally be followed by a symbol for the rounding mode.

The possible rounding modes for dot product expressions are:

Symbol	Expression form	Rounding mode
#*	scalar, vector or matrix	nearest
#<	scalar, vector or matrix	downwards
#>	scalar, vector or matrix	upwards
##	scalar, vector or matrix	smallest enclosing interval
#	scalar only	exact, no rounding

In order to be able to store the unrounded result of a dot product expression, FORTRAN-SC provides the new data types DOT PRECISION and DOT PRECISION COMPLEX. Such results are produced by a dot product expression where no rounding is specified (see last row in the table above). The DOT PRECISION types are scalar data types of restricted accessibility. Variables of

these types can only be added, subtracted and compared. They may appear as summands within any scalar dot product-expression. A dot precision variable may only be assigned a dot precision value of the same type.

Example:

```
      DOT PRECISION D
      DYNAMIC /REAL(:)/ X, Y, Z, /REAL(:,:)/ A, B
      REAL R
      INTERVAL V
      READ (*,*) n
      ALLOCATE A(n,n), B (=A), X(n), Y, Z(=X)
      ...
      X= #*(Y-A * X)
      V= # #(X * Y-Y * Z+R)
      A= #*(A * B-B * A)
      ...
      D= #(Ø)
      DO 1Ø j=1, n
         D=D+ #(A(j,j) * B(j,j))
1Ø    CONTINUE
      R= #*(D)
      V= # #(D)
```

In practice, dot product expressions may contain a large number of terms, making an explicit notation very cumbersome. In mathematics the symbol Σ is used for short. For instance, if A_i, B_i are scalars or vectors or matrices for each $i=1,\ldots,k$, then the sum

$$\sum_{i=1}^{k} A_i B_i$$

represents a dot product expression. FORTRAN-SC provides the equivalent shorthand notation SUM for this purpose. In the example above, the last six lines could be replaced by:

```
      D= #(SUM(A(j,j) * B(j,j),j=1,n))
      R = #*(D)
      V = # #(D)
```

This shows that a result involving n multiplications and n−1 additions can be produced with a single rounding operation. In the last statement the exact dot product is rounded to the smallest possible interval enclosing the exact value of the expression. Thus, the bounds of the interval V will either be the same or two adjacent floating-point numbers.

Dot product expressions play a key role in numerical analysis. Iterative refinement or defect correction methods for linear and nonlinear problems usually lead to dot product expressions. Exact evaluation of these expressions eliminates cancellation.

Information that has been lost by rounding effects during an initial computation can often be recovered by defect correction. Such corrections can deliver results of full floating-point accuracy. In principle, there is no limit to the accuracy that can be obtained by these methods.

3. The Implementation of FORTRAN-SC

Since 1984, a FORTRAN-SC compiler has been developed for the IBM/370 architecture. First programming experiences have demonstrated the usefulness and effectiveness of the language and the reliability of the implementation.

The FORTRAN-SC compiler is essentially a 2-pass compiler. Its front end performs complete lexical, syntactical and semantical analysis of the source program. In order to achieve high portability, the code generator produces FORTRAN 77 code. For easy debugging, the FORTRAN-SC source code can optionally be merged as comments into the generated FORTRAN 77 code. The extensive runtime library provides the predefined operators, the intrinsic functions and some auxiliary routines (e.g. for array management). Error handling is integrated into every routine.

The guiding principle of FORTRAN-SC is to achieve higher accuracy and more reliable results in scientific computation. These ideas had a profound influence on both the language and its implementation. Several new concepts (new data types, dynamic arrays and dot product expressions) required new compilation techniques.

As mentioned earlier, FORTRAN-SC is closely related to FORTRAN 8x. In particular, it is a superset of FORTRAN 77. In contrast to FORTRAN 77, however, the current implementation of FORTRAN-SC does not support statement functions and entry statements (use separate routines instead), assumed size arrays and adjustable arrays (use dynamic arrays instead).

On vector machines, many runtime routines could be vectorized. In particular, the speed of array operations which work elementwise could be greatly increased. However, special care must be taken because the language FORTRAN-SC requires that all predefined operators deliver results of 1 ulp accuracy.

4. FORTRAN-SC Sample Program

The following program assumes that a function (APPINV) for the computation of an approximate inverse of a square matrix exists. After preliminary inversion, the solution of the linear system is enclosed in an interval vector by successive interval iterations. For details about this method, see [21].

Note that in FORTRAN-SC lower case letters are interpreted as upper case and that identifiers and operator names may be up to 31 characters in length.

```
      PROGRAM LINSYS
C     Verified solution of the linear system of equations
C        A · x = b
      DYNAMIC /REAL(:, :)/ A, R, UNIT, IDENTITY
      DYNAMIC /INTERVAL(:, :)/ E
      DYNAMIC /REAL(:)/ B
      DYNAMIC /INTERVAL(:)/ X, Y, Z
      INTEGER dim, i, j, iter
C     UNIT is an EXTERNAL function which delivers the identity
C     matrix of the given dimension
      EXTERNAL UNIT

C     The following operator declaration overloads the intrinsic operator .IN.
C     for a new operand type combination (2 interval vectors).
      OPERATOR .IN. = INCL (INTERVAL(:), INTERVAL(:)) LOGICAL
      OPERATOR .EXPAND. = EXPAND (INTERVAL(:)) INTERVAL(:)
C     APPINV is an EXTERNAL function which delivers an
C     approximate inverse of a real matrix
      OPERATOR .APPROXIMATE INVERSE. =
     &                 APPINV (REAL(:, :)) REAL(:, :)
      WRITE(*, *) 'Please enter the dimension of the linear system'
      READ(*, *) dim
      ALLOCATE A(dim, dim), B(dim)
      WRITE(*, *) 'Please enter the matrix A'
      READ(*, *) ((A(i, j), j = 1, dim), i = 1, dim)
      WRITE(*, *) 'Please enter the right-hand side B'
      READ(*, *) (B (i), i = 1, dim)

      R = .APPROXIMATE INVERSE. A
C     R · b is an approximate solution of the linear system.
C     Z is a maximally accurate inclusion of R · b. It does not
C     usually include the true solution.
      Z = # #(R * B)
      IDENTITY = UNIT(dim)
C     A maximally accurate inclusion of I − R · A is computed.
      E = # #(IDENTITY − R * A)
      X = Z

      DO 2Ø iter = 1, 1Ø
C     To obtain a true inclusion, the
C     interval vector X is slightly inflated.
         Y = .EXPAND. X
C     The following expression contains interval vectors and an
C     interval matrix.
         X = Z + E * Y
         IF (X .IN. Y) GOTO 1Ø
2Ø    CONTINUE
```

```
      WRITE(*,*) 'No solution found!'
      STOP
10    WRITE(*,*) 'The given matrix is non-singular and the',
     &           'solution of the linear system is contained in:'
      WRITE(*,*)X
      END

      FUNCTION EXPAND (X)
      DYNAMIC /INTERVAL(:)/ X, EXPAND
      INTERVAL IEPS
      INTEGER i
      DATA IEPS /-1.0D-75, 1.0D-75/
      ALLOCATE EXPAND (=X)
C     EXPAND now has the same index bounds as X.
      DO 10 i=LB(X), UB(X)
         EXPAND(i)=X(i)+IEPS
10    CONTINUE
      RETURN
      END

      FUNCTION INCL (X, Y)
C     Is X a subset of the interior of Y?
      LOGICAL INCL
      DYNAMIC /INTERVAL(:)/ X, Y
      INTEGER i
      INCL=.TRUE.
      DO 10 i=LB(X), UB(X)
         IF (INF(Y(i)) .GE. INF(X(i)) .OR.
     &       SUP(Y(i)) .LE. SUP(X(i))) THEN
            INCL=.FALSE.
            RETURN
         END IF
10    CONTINUE
      RETURN
      END
```

5. Conclusion

Several modern programming languages provide a large number of basic arithmetic operations by their usual mathematical symbol. ADA, FORTRAN 8x and other languages appropriate for vector machines provide vector and matrix operations. It is certainly the most natural requirement that these basic operations be executed with highest accuracy for all possible combinations of data. If this is not possible, an error message should be given. In FORTRAN-SC, all vector and matrix operations deliver a result of at least 1 ulp accuracy.

Finally, FORTRAN-SC greatly simplifies the use of the ACRITH library. ACRITH provides routines for a large number of vector and matrix operations as well as for elementary functions. All of these can be accessed by their usual mathematical notation in FORTRAN-SC. The availability of additional higher data types as well as dynamic arrays and array-valued functions provide additional advantages. The problem-solving routines of ACRITH and other libraries may be called with a reduced list of parameters. All of these concepts improve the readability of programs and facilitate debugging considerably.

References

[1] Agarwal, R. C., Cooley, J. W., Gustavson, F. G., Shearer, J. B., Slishman, G., Tuckerman, B.: New Scalar and Vector Elementary Functions for the IBM System/370. IBM Journal of Research and Development *30/2*, 126–144 (1986).

[2] Alefeld, G., Herzberger, J.: Introduction to Interval Analysis. New York: Academic Press 1983.

[3] American National Standards Institute: American National Standard Programming Language FORTRAN. ANSI X3.9-1978.

[4] American National Standards Institute: American National Standard Programming Language FORTRAN. Draft S8, Version 104, ANSI X3.9-198x (1987).

[5] American National Standards Institute / Institute of Electrical & Electronic Engineers: A Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985, New York (Aug. 1985).

[6] Arithmos (BS 2000) Benutzerhandbuch. Siemens, U2900-J-Z87-1 (Sept. 1986).

[7] Bohlender, G., Kaucher, E., Klatte, R., Kulisch, U., Miranker, W. L., Ullrich, Ch., Wolff v. Gudenberg, J.: FORTRAN for Contemporary Numerical Computation. IBM Research Report RC 8348 (1980). Computing *26*, 277–314 (1981).

[8] Bohlender, G., Böhm, H., Grüner, K., Kaucher, E., Klatte, R., Krämer, W., Kulisch, U., Rump, S. M., Ullrich, Ch., Wolff v. Gudenberg, J., Miranker, W. L.: Proposal for Arithmetic Specification in FORTRAN 8x. Proceedings of the International Conference on: Tools, Methods and Languages for Scientific and Engineering Computation, Paris 1983, North Holland (1984).

[9] Bohlender, G., Böhm, H., Braune, K., Grüner, K., Kaucher, E., Kirchner, R., Klatte, R., Krämer, W., Kulisch, U., Miranker, W. L., Ullrich, Ch., Wolff v. Gudenberg, J.: Application Module: Scientific Computation for FORTRAN 8x. Modified Proposal for Arithmetic Specification According to Guidelines of the X3J3-Meetings in Tulsa and Chapel Hill. Report of the Institute for Applied Mathematics, University of Karlsruhe (March 1983).

[10] Bohlender, G., Rall, L. B., Ullrich, Ch., Wolff v. Gudenberg, J.: PASCAL-SC: Wirkungsvoll programmieren, kontrolliert rechnen. Mannheim-Wien-Zürich: Bibliographisches Institut – Wissenschaftsverlag 1986.

[11] Braune, K., Kraemer, W.: High-Accuracy Standard Functions for Intervals. Computer Systems: Performance and Simulation (Ruschitzka, M., ed.). North-Holland (1986).

[12] Buchholz, W.: The IBM System/370 Vector Architecture. IBM Systems Journal *25/1* (1986).

[13] Gal, S.: Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance. IBM Technical Report 88.153 (1985).

[14] IBM High-Accuracy Arithmetic Subroutine Library (ACRITH): General Information Manual, GC 33-6163-02, 3rd Edition (April 1986).

[15] IBM High-Accuracy Arithmetic Subroutine Library (ACRITH): Program Description and User's Guide, SC 33-6164-02, 3rd Edition (April 1986).

[16] IBM System/370 RPQ: High-Accuracy Arithmetic. SA 22-7093-0 (1984).

[17] Kulisch, U. (ed.): PASCAL-SC: A PASCAL Extension for Scientific Computation. Information Manual and Floppy Disks, Version IBM PC/AT, Operating System DOS, B. G. Teubner, Stuttgart. Chichester: John Wiley & Sons 1987.

[18] Kulisch, U., Miranker, W. L.: Computer Arithmetic in Theory and Practice. New York: Academic Press 1981.

[19] Kulisch, U., Miranker, W. L. (eds.): A New Approach to Scientific Computation. New York: Academic Press 1983.
[20] Moore, R. E.: Interval Analysis. Englewood Cliffs, N.J.: Prentice Hall 1966.
[21] Rump, S. M.: Solving Algebraic Problems with High Accuracy. In: [19], pp. 58–62.

J. H. Bleher and S. M. Rump
Entwicklung und Forschung
IBM Deutschland GmbH.
Schönaicher Strasse 220
D-7030 Böblingen
Federal Republic of Germany

U. Kulisch, M. Metzger,
Ch. Ullrich and W. Walter
Institut für Angewandte Mathematik
Universität Karlsruhe
Postfach 6980
D-7500 Karlsruhe
Federal Republic of Germany