Interval Computations
No 4(6), 1992

PROGRAMMING LANGUAGE SUPPORT FOR SCIENTIFIC COMPUTATION

Jürgen Wolff von Gudenberg

The necessary prerequisites for scientific computation, e.g. operators with directed roundings, operator and function overloading, dynamic arrays and accurate scalar product computation have been implemented in PASCAL-XSC, ACRITH-XSC, MODULA-SC, C-XSC and Ada.

This paper gives an overview on currently existing language extensions and comments on several trends for future development.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ ДЛЯ НАУЧНЫХ ВЫЧИСЛЕНИЙ

Ю.Вольфф фон Гуденберг

В языках PASCAL-XSC, ACRITH-XSC, MODULA-SC, C-XSC и ADA были реализованы такие необходимые для научных вычислений средства, как операции с направленным округлением, динамические массивы, высокоточное скалярное произведение, совмещение имен для операций и функций. Дан обзор существующих в настоящий момент расширений языков программирования и обсуждаются некоторые тенденции их дальнейшего развития.

1. Prerequisites

The datatypes occurring in numerical computation include the real and complex numbers, vectors, or matrices. Furthermore interval spaces are considered to provide guaranteed results.

The mathematician uses the same infix operators to denote the arithmetic operations in all of these spaces. Also identical function names identify the standard functions for different data types. Matrix or vector algorithms are usually formulated for arbitrary dimension.

Furthermore the accuracy of the basic operations shall be optimal for all spaces, i.e. the relative error is less than or equal to 1/2 ulp (unit of the last place) in case of rounding to the nearest computer representable element, or less than 1 ulp in case of other monotone roundings.

Algorithms and hardware which fulfill these accuracy requirements even in the case of arbitrarily dimensioned matrices are available [10,11].

All these features ought to be supported in a programming language for scientific computation. For this purpose a language may be extended by a huge set of additional standard operators or functions for each of the relevant data types. The main drawback of this approach, however, is that it is not extendable. The programmer is not able to define his own set of types and operations.

New problem specific datatypes like grids or gradients can not be handled the same way as the standard types. Therefore the programming language should be extended by powerful concepts which enable the programmer to write and implement new arithmetic operations or even new expression evaluation in the language. For this purpose the following concepts are recommended.

- user-defined operators at least by overloading the standard operators or further by allowing new operator identifiers or symbols
- structured result types of operators and functions
- overloadable function identifiers
- dynamic arrays
- access to directed roundings for interval arithmetic

For the accuracy requirement three arithmetic properties are mandatory.

- Floating-point arithmetic is optimally defined.
- At least three roundings (towards + or $-\infty$, and to the nearest) are accessible.
- An optimal scalar product with 1/2 ulp accuracy is available. (The

scalar product may be simulated by integer arithmetic.)

More features will further support the ease of programming.

A module concept is very helpful to facilitate the management of a lot of different arithmetic libraries.

Generic, type-parameterized subroutines reduce the size of source code and thus enhance the readability of the program by combining routines which perform identical operations for different data types. The object code size may also be decreased.

A subtype concept may allow the automatic conversion of parameters to match the interface of a function or operator.

Alternative expression concepts, e.g. for accurate evaluation or symbolic computation may be supplied.

Simultaneous polymorphism increases the information obtained. A complex interval may be regarded as a circle or rectangle or an expression may be kept in its symbolic form and evaluated as floating-point number and interval.

2. Comparison of existing languages

In the following table we compare the standard languages FORTRAN 90 (F90), C++, Ada and several extensions with respect to the facilities they support for scientific computation. PASCAL-XSC (PXSC) [3, 8], ACRITH -XSC (FXSC) [1] and C-XSC [12] are extensions of Pascal, Fortran or C developed at the Institute for Applied Mathematics in Karlsruhe, FXSC on behalf of IBM Deutschland. C-XSC actually is no new compiler but a set of C++ classes which appear as a simple extension of C. Modula-SC [4] is a Modula-2 extension developed of Institute for Computer Science in Basel. Ada packages were implemented in the ESPRIT project DIAMOND by NAG Oxford, CWI Amsterdam, Siemens München and Karlsruhe University. [9]

	operator concept	result type	overload functions
F90	overload new idents	structure	generic interface
FXSC	overload new idents	structure	generic interface
PXSC	overload new idents	structure	yes
MSC	overload new idents	structure	no
CXSC	predefined	class	yes
C++	overload	class	yes
Ada	overload	structure	yes

	array concept	module concept	genericity	subtype relation
F90	dynamic expl.alloc.	yes	no	no
FXSC	dynamic autom.alloc.	yes	no	no
PXSC	blockdyn.	yes	no	int,real
MSC	static	yes	no	no
CXSC	dynamic predefined	class	no	yes predef.
C++	free inherit	class	yes	predef. inherit
Ada	dynamic	yes	yes	no

	directed rounding	altern. express.	float arith.
F90	no	no	not spec.
FXSC	yes	dot	KA
PXSC	yes	dot	KA
MSC	yes	no	KA
CXSC	yes	dot	KA
C++	no	no	not spec.
Ada	no	no	Brown DIAMOND

Comments:

All SC languages fulfill the accuracy requirements (KA) for all numerical datatypes. From the standard languages only Ada specifies its floating-point accuracy by Brown's model numbers. This complies to our requirement for the floating-point data type only. The same holds for the IEEE Standard 754 which is more and more used as floating-point hardware.

C-XSC on its own is only a collection of data types and operations, which may be sufficient for a modest user. A sophisticated programmer will use C-XSC together with C++ and thus may join the features of both languages.

Block structured languages like PASCAL-XSC and Ada provide dynamic arrays as local variables, parameters, and results of functions where the scope extends the defining block. In FORTRAN 90 and ACRITH-XSC explicit allocate and deallocate statements control the scope. The left hand side of an assignment has to be allocated in F90, but is automatically created in FXSC. In C++ dynamic arrays may be implemented as contiguous space accessed via pointers and therefore specific constructors and destructors are necessary.

Overloading of functions in FORTRAN 90 works via the definition of a generic interface, which means that specific names must be provided.

3. Alternative expression concepts

Although all basic operators for all numerical data types have maximum accuracy, this is not sufficient to guarantee the result of an arbitrary expression or algorithm. But there are methods which evaluate complete expressions with maximum accuracy or enclose the result of a full algorithm into sharp bounds [2, 5]. The procedure for expression evaluation is straightforward. All intermediate results are kept, and a defect correction iteration using the optimal scalar product is performed until the enclosure of the exact result is sharp enough (1 or 2 ulp).

This procedure may be integrated into a compiler. But then two different ways to evaluate an expression exist. Therefore we need to mark such accurate expressions. We propose to put an expression which is to be evaluated accurately into parentheses, preceded by #<, #>, #*, or ## to indicate the rounding downwardly directed, upwardly directed, to the nearest, or to the smallest enclosing interval, respectively.

Since the accurate evaluation of expressions and the sharp enclosure of results of algorithms usually use one very specific kind of expression, the generalized dot product, i.e. a sum of variables (constants) or two-factored products of variables (constants), only this feature has been supported by several language extensions. This implementation can easily be obtained by using a type which is large enough to store intermediate results of scalar product computations without loss of information. Such type is called dotprecision.

Dotprecision expressions for all numerical datatypes are available in CXSC, PXSC and FXSC.

Their implementation via the universal operator concept is not possible, since operators are identified by their name or symbol, and their operand type(s) and because it is not possible to overload an operator twice for the same parameter profile.

If we define a second overloading of the $*$ operator, by using a new operator identifier, e.g. dotmul and a conversion function dot from real to dotprecision, a dotprecision expression looks like

```
A dotmul B + dot(D) - C dotmul E
```

where + and - are overloadings for the type dotprecision. We further need rounding functions from dotprecision to real or interval.

The overhead for the addition of two dotprecision values in contrast to the addition of a floating-point number may be neglected in the scalar case. In the matrix case, however, simple operator overloading requires the allocation of a full matrix of dotprecisions whereas one single value is sufficient to compute the whole expression componentwise. Therefore the operators dotmul and + in this case shall collect the structure of the expression in a linked list or tree whose evaluation is initiated by the call of the rounding function or operator. A static or class variable may serve to store the structure of the expression in an object oriented language like C++. If this list is kept visible for the sophisticated user symbolic manipulation and optimization of the expression is possible, before it is evaluated using some basic runtime system routines. Hiding the list for the normal user, however, is strongly recommended.

Expressions with named operators and lot of conversion functions do not look very familiar, therefore we suggest a small extension to the operator concept which allows multiple overloading. For PASCAL-XSC the syntax may look like follows

```
prefix <prefix symbol> operator <op symbol>
<parameter list> <result name>:  <result type>
```

Valid prefix symbols are combinations of characters starting with #, %, ! or another symbol which does not occur in standard operators. The prefix symbol itself is introduced as a monadic operator with highest priority.

Applying these extensions the dotprecision expression from above looks like

```
#<(A*B + D - C*E)
```

where the prefix operator #< denotes the downwardly directed rounding from dotprecision to real, e.g. This is exactly the syntax of dotprecision expressions in PASCAL-XSC and ACRITH-XSC, but in these languages the prefix operator overloading is not accessible by the user and therefore not extendable.

4. Expression data types

Using the principles of data encapsulation or information hiding, we can extend the manipulation of expressions and give the user access to the symbolic structure without letting him know the internal details of the representation.

Since on the one hand symbolic expression manipulation usually does not depend on the operand types, on the other hand static type checking shall be performed, we suggest a kind of restricted polymorphic expression data type.

The operand types of a symbolic expression may be chosen from a specified set of types.

Two different syntax extensions have been proposed in [17,18] (expression datatype), and [19] (multiaspectness). We shortly report on these two approaches.

Expression data type:

A new type is introduced by the keywords 'expression of' followed by a list of possible operand types.

Symbolic operators for this type just build up the structure of the expression. They represent a set of possible operators one of which is chosen for evaluation.

To identify those operators not only the operand types but also the left hand side of an assignment, the possible actual argument type of a function (or operator), if the expression occurs in a function call and accessing array or record components are considered.

A standard set of symbolic expression manipulation routines is provided.

Multiaspectness:

- A standard type formula which is not related to specific operand types is provided.
- Assignment compatibility is declared by allowing a variable to be represented in several types.
- More than one simultaneous value for a variable is possible.
- If only one representation shall be valid at each time, precedence rules or order relations between types have to be specified.
- If more representations are valid, we have a kind of simultaneous poly-

morphism which may be used to obtain more information about the true value. The typical example here is the representation of a complex interval as a rectangle and a circle.

Standard operator overloading is sufficient.

5. Genericity

Genericity means the declaration of a function or module (class, type) templates which are parameterized by a type and its instantiation with an actual type. The FORTRAN term generic function only means a simple way of overloading and is therefore not treated here.

There are various ways of implementing genericity. The simplest is to copy the source text and include the proper actual type name where appropriate. This can be done by a preprocessor, as is shown in C and early C++ versions. But this does not save any code and disables the debugging of such routines, it further is error prone and may by no means considered as a proper language extension.

Genericity may be restricted in the sense that the generic type parameter needs to import some operations. In Ada these operations have to be specified as generic parameters, since types correspond to structures and carry no operations with them. The compiler checks the availability of the actual operations at instantiation time. In C++ classes, which include operations, are used as generic parameters and the correctness of an instantiation is checked at linking time. Due to the simple concept of separate compilation existing compilers need to have the source code of a template definition at hand when creating an instantiation.

Is it possible to simulate genericity with other concepts introduced above ?

Generic operators or functions may be declared for the expression data type [17,18]. But this only means that a fixed number of operators are implicitly declared. All possible instantiations can be created after the declaration of the generic routine. Therefore this concept can only be considered as an abbreviation rather then a language extension.

In object-oriented languages genericity may to a certain degree be simulated by inheritance. But if the result type is generic, we need the assignment from supertype to subtype which is forbidden for reasons of type consistency. Additional syntax and semantics like declaration by association have to be introduced [14].

We suggest to use a newly defined abstract generic base type which only specifies the signatures of the operations allowed on this type as a parameter type for generic routines. A derivation from this type automatically instantiates the proper routines. Partial instantiation is possible, if a routine or type depends on more than one generic parameter. [13]

We define the syntax and semantics in a C++ like style of generic classes as follows.

The keyword generic preceding an abstract class definition means that in derivations from this class all virtual functions or operators are instantiated by accordingly replacing parameters or results of the generic class type itself by the derived class type.

Structures and arrays of generic classes remain generic.

A class or type which is derived from a generic class must provide appropriate definitions of all virtual functions or operators, either by default, or by generic instantiation or by explicit code. Standard types or earlier defined classes may be made descendants of a generic class, if the corresponding functions and operators match.

6. Summary

A programming language for scientific computation is based on a correctly defined arithmetic and provides features which enable the sophisticated user to write the runtime system which includes standard functions, arithmetic for various data types, accurate expression evaluation, and self-validating algorithms completely in the language. We discussed some of these features in this article.

References

1. Bleher, J.H., Kulisch, U., Metzger, M., Rump, S.M., Ullrich, Ch. and Walter, W. *FORTRAN-SC: a study of a FORTRAN extension for engineering/scientific computation with access to ACRITH.* Computing **39** (1987), pp. 93–110.

2. Böhm, H., Rump, S. and Schumacher, G. *E-methods for nonlinear problems.* In [7], pp. 59–80.

3. Bohlender, G., Rall, L.B., Ullrich, Ch. and Wolff von Gudenberg, J. *PASCAL-SC:*

a computer language for scientific computation. In: "Perspectives in Computing, vol. 17". Academic Press, Orlando, 1987.

4. Falco-Korn, C., Gutzwiller, S., Künig, S. and Ullrich, Ch. *Modula-SC, motivation, language definition and implementation.* In [6], pp.161–180.

5. Fischer, H.C., Schumacher, G. and Haggenmßller, R. *Evaluation of arithmetic expressions with guaranteed high accuracy.* Computing Supp. **6**, pp. 149–158.

6. Kaucher, E., Markov, S. and Mayer, G. *Computer arithmetic – scientific computation and mathematical modelling.* IMACS annals on computing and applied mathematics **12** (1992).

7. Kaucher, E., Kulisch, U. and Ullrich, Ch. (eds) *Computer arithmetic – scientific computation and programming languages.* Teubner Verlag, Stuttgart, 1987.

8. Klatte, R., Kulisch, U., Neaga, M., Ratz, D. and Ullrich, Ch. *PASCAL-XSC, sprachbeschreibung mit Beispielen.* Springer, Berlin,1991.

9. Kok, J. *The embedding of accurate arithmetic in Ada.* In [16], pp. 99–120.

10. Kulisch, U. and Miranker, W.L. *Computer arithmetic in theory and practice.* Academic Press, New York, 1981.

11. Kulisch, U. and Miranker, W.L. *The arithmetic of the digital computer: a new approach.* SIAM Review **28** (1) (1986).

12. Lawo, C. *C-XSC a programming environment for eXtended Scientific Computation* In: "Proceedings of the 13th World congress on computation and applied mathematics, IMACS, Dublin, 1991".

13. Marggraff, H. *Objektorientierte Erweiterung von PASCAL-SC und Verfahren zur Implementierung.* Diplomarbeit, Universität Karlsruhe, 1988.

14. Meyer, B. *Object-oriented software construction.* Prentice Hall, 1988.

15. Ullrich, Ch. (ed.) *Contributions to computer arithmetic and self-validating numerical methods.* IMACS annals on computing and applied mathematics **7** (1990).

16. Wallis, P.J.L. *Improving floating-point programming.* J. Wiley, Chichester, 1990.

17. Wolff von Gudenberg, J. *Arithmetische und programmiersprachliche Werkzeuge für die Numerik.* Computer Theoretikum und Praktikum für Physiker **5** (1990) pp. 15–42.

18. Wolff von Gudenberg, J. *A symbolic generic expression concept.* In [15], pp.459–464

19. Yakovlev, A.G. *Multiaspectness in programming of localizational (interval) computations.* In: "Proceedings of Seminar on interval mathematics, Saratov, May 29–31, 1990", pp. 113–120 (in Russian).