CONF-781052--5

MASTER

EXTENDING RELIABILITY: TRANSFORMATIONAL TAILORING OF ABSTRACT MATHEMATICAL SOFTWARE

by

James M. Boyle

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Prepared for

ACM/SIGNUM Conference on

Programming Environment for Development of Numerical Software

Pasadena, CA

October 18-20, 1978

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

U of C-AUA-USDOE

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Prepared for the U. S. DEPARTMENT OF ENERGY
under Contract W-31-109-Eng-38

5

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Extending Reliability: Transformational Tailoring of Abstract Mathematical Software*

James M. Boyle
Applied Mathematics Division
Argonne National Laboratory

EXTENDED ABSTRACT

INTRODUCTION

Over the past decade, mathematical software libraries have matured from small, usually locally-assembled, collections of subroutines to large, commercially-provided libraries which are approaching the status of standards [Aird; Du Croz; Fox]**. Despite the high quality of such libraries and the obvious economic advantages of using routines whose development cost has been shared with many other users, applications programmers, when asked: "Why don't you use routine XYZ from IMSL, or from NAG, or from PORT?" frequently reply that library routines are too general, that they need a routine which takes advantage of special features of their problem, and that since they could not use a library routine without modifying it, they might as well write their own routine from scratch.

In many, if not most, instances, the latter assertion could be easily refuted by a simple competition on selected test problems. However, the need for a routine adapted, or tailored, to a particular problem is more difficult to dismiss. It usually arises from considerations of efficiency, which may range from the perceived inefficiency of the presence of unused options in a routine to the practical impossibility of using a routine whose data representation is utterly incompatible with that needed in the rest of the applications program.

How, then, can mathematical software developers answer this need for mathematical algorithms tailored to individual applications? One approach especially applicable to complicated problems, such as solution of PDE's, is to preprocess a specification of the problem into code which uses a particular software package, as is done in ELLPACK [Rice]. (In some sense, this approach tailors the problem to the software.) For library routines in simpler problem areas, however, it seems necessary to tailor the routine to the problem, since such routines constitute only a small part of the application program, and several routines with possibly conflicting requirements may need to be included. In order for this to be practical, *tailored versions of such routines must be constructed mechanically from very general library routines.* Such mechanical program generation is necessary both to insure that the reliability of the library routine is preserved in its tailored versions and to insure that their construction is not prohibitively expensive [6].

For some time, the TAMPR system has been in use to construct multiple versions, or realizations, of prototype programs for inclusion in mathematical software packages themselves [4,5]. For the LINPACK package, a single prototype routine was used to construct the eight versions representing the combinations of complex or real arithmetic, single or double precision, and calls to Basic Linear Algebra subroutines or in-line code replacements for them [5].

Recent research with TAMPR has focussed on determining the properties a prototype program should have in order to maximize the number and diversity of realizations which can be constructed from it.

ABSTRACT PROGRAMS

The most important property of a prototype program is its *abstractness*. Intuitively, an abstract program captures the essence of a numerical algorithm without cluttering it with irrelevant detail. The presence of irrelevant detail in a prototype program hampers the construction of diverse realizations precisely because a great deal of analysis must be done to verify that it is indeed irrelevant.

The research discussed here has not progressed far enough to characterize abstract programs in general, but examples from the area of linear algebra have been studied sufficiently to illustrate the ideas involved. Consider the code fragment (1):

```
for i = n,1,-1
   for j = i+1,n
      y(i) = y(i)-U(i,j)*y(j)
   end
   y(i) = y(i)/U(i,i)
end
```

and the code fragment (2):

```
for i = n,1,-1
   y(i) = y(i)/U(i,i)
   for j = 1,i-1
      y(j) = y(j)-U(j,i)*y(i)
   end
end
```

Now, both of these fragments actually perform the same computation, the solution of an upper-triangular system of linear equations Uy=x (the final step in the solution of a general linear system whose matrix has been factored into triangular matrices L and U). Fragment (1) is the usual method of solving such a system; it refers to the matrix U by rows. Fragment (2) refers to the matrix U by columns, and is therefore more efficient than fragment (1) on machines with virtual memory or with buffer memory, when the language in which the program is written stores matrices by columns (as does Fortran), see Moler [8] and Smith [10].

Considerable effort is required to see that these two fragments actually perform the same computation; even more effort would be required to devise a way to transform (1) into (2) automatically in order to be able to make the (possibly) more efficient version available to a user. Thus

*Work performed under the auspices of the U.S. Department of Energy.

**Numbers in brackets designate References at end of paper; names in brackets designate authors of other abstracts in this Proceedings.

(1) is not a suitable prototype for tailoring, since it contains difficult-to-discard information about the row-oriented version of the program, which has nothing to do with the specification of the algorithm for the solution of the linear system (see also Smith [10].

At what level of abstraction, then, is such irrelevant information about how storage is referred to absent from the specification of the algorithm? It is absent when the algorithm is specified in terms of matrix (rather than matrix element) operations. Thus the abstract representation of this algorithm is (3):

$$x = U^{-1}y$$

(Note that this representation of the algorithm is abstract not only with respect to row or column orientation, but with respect to all aspects of the representation of the data; e.g., the elements of U could be given by a function.)

TRANSFORMATIONAL SYNTHESIS OF CONCRETE PROGRAMS

This abstract statement of the triangular solution algorithm can be converted into a concrete, executable program by first augmenting it with a specification of the properties of the concrete representations of U,x, and y. The augmented abstract program can then be transformed according to various program-algebraic rules which incorporate the properties into the abstract program and then simplify it where possible (see Boyle [4] and Green [9]).

This process can be illustrated by a sketch of the synthesis of fragment (2) from the abstract program (3). This sketch omits numerous small steps and to avoid introducing unfamiliar notation is presented in terms of quasi-programs. In the actual TAMPR implementation, the transformations are carried out on a representation of the program in an applicative (i.e., expression) language until the final stage, at which Fortran code is generated. As discussed by Backus in his 1977 Turing Lecture [1] such applicative languages have a richer and simpler associated "algebra" than do conventional languages in part because the scope of values is indicated clearly by functional application. (Experience with TAMPR strongly supports this simplicity of applicative languages, which is also well known to LISP programmers.)

The synthesis of fragment (2) depends on some identities from matrix algebra, including:

(4) $I = \sum_{i=1}^{n} e_i e_i^T$

where e_i is the i-th unit vector;

(5) $D + \sum_{i=1}^{n} U' e_i e_i^T = \prod_{i=n}^{1} (I + De_i e_i^T - Ie_i e_i^T + U' e_i e_i^T)$

where D is an nxn diagonal matrix and U' is upper triangular with zero diagonal; and

(6) $\prod_{i=n}^{1} (I + De_i e_i^T - Ie_i e_i^T + U' e_i e_i^T)$

$= \prod_{i=1}^{n} (I + D^{-1} e_i e_i^T - Ic_i c_i^T - U' c_i c_i^T D^{-1} c_i e_i^T)$

The idea of a matrix A being "stored by columns" is thus expressed as

$A = AI = A\sum_{i=1}^{n}(e_i e_i^T) = \sum_{i=1}^{n} ((Ae_i)e_i^T);$

Ae_i is the i-th column of A.

The synthesis begins with (3) augmented to indicate that U is an upper-triangular nxn matrix stored by columns, that y is an n-vector, and that x is to be identified with y:

```
    y = U**-1*y
--> y = (diag(U) + uppersubtri(U))**-1*y
now U is expanded by columns:
--> y = (diag(U) + $\sum_{i=1}^{n}$ (uppersubtri(U)$e_i e_i^T$))**-1*y
--> y = ($\prod_{i=n}^{1}$(I+(diag(U)$e_i$)$e_i^T$-I$e_i e_i^T$
        +(uppersubtri(U)$e_i$)$e_i^T$))**-1*y
--> y = $\prod_{i=1}^{n}$(I+(invdiag(U)$e_i$)$e_i^T$-I$e_i e_i^T$
        -(uppersubtri(U)$e_i$)($e_i^T$ invdiag(U)$e_i$)$e_i^T$)*y
--> for i = n,1,-1
        y = y + (invdiag(U)$e_i$)($e_i^T$y)-$e_i$($e_i^T$y)
        -(uppersubtri(U)$e_i$)($e_i^T$ invdiag(U)$e_i$)($e_i^T$y)
    end
```

(Note that the above program is the point of departure for a "vector" solution of the triangular system, although this problem is not particularly well suited to vector computation.) Now expand the remaining vectors to components (the assignment is still a vector one):

```
--> for i=n,1,-1
        $\sum_{j=1}^{n} e_j(e_j^T y) = \sum_{j=1}^{n} e_j(e_j^T y) + e_i(e_i^T$ invdiag(U)$e_i$)*
        $(e_i^T y) - e_i(e_i^T y) - \sum_{k=1}^{n}$ {$e_k(e_k^T$ uppersubtri(U)$e_i$)*
        ($e_i^T$ invdiag(U)$e_i$)($e_i^T$y)}
    end
```

After a number of steps which include determining that a vector temporary is not required in converting the vector assignment to a scalar one, the component-level is reached:

```
--> for i = n,1,-1
        for j = 1,i-1
            ($e_j^T$y) = ($e_j^T$y) - ($e_j^T$ uppersubtri(U)$e_i$)*
            ($e_i^T$ invdiag(U)$e_i$)($e_i^T$y)
        end
        ($e_i^T$y) = ($e_i^T$ invdiag(U)$e_i$)*($e_i^T$y)-
        ($e_i^T$ uppersubtri(U)$e_i$)*($e_i^T$ invdiag(U)$e_i$)($e_i^T$y)

        for j = i+1,n
```

```
            $(e_j^T y) = (e_j^T y) - (e_j^T \text{ uppersubtri}(U)e_i)*$
            $(e_i^T \text{ invdiag}(U)e_i)(e_i^T y)$
        end
    end
```

Uppersubtri(U) implies e_k^T uppersubtri(U)$e_j = 0$ for $k \geq j$, assignments of the form x = x need not be carried out, and common subexpressions can be computed once, so that the program becomes:

```
--> for i = n,1,-1
        $t = (e_i^T \text{ invdiag}(U)e_i)(e_i^T y)$
        for j = 1,i-1
            $(e_j^T y) = (e_j^T y) - (e_j^T \text{ uppersubtri}(U)e_i)*t$
        end
        $(e_i^T y) = t$
    end
```

The temporary can be eliminated by reordering and the component references converted to conventional form to obtain fragment (2), above. Thus the transformational synthesis of a program takes place in a large sequence of small steps, each effected by a transformation based on a relatively simple mathematical theorem or axiom.

CORRECTNESS OF CONCRETE PROGRAMS

As discussed in [4], transformationally-constructed concrete programs inherit the correctness of the abstract prototype program provided the transformations themselves are correct. A correct transformation may add information to the abstract program, but this information must be consistent with the properties assumed for the abstract program. (In this sense, the process is rather like constructing the integers as a "concrete" instance of a ring, by augmenting the ring axioms with additional axioms consistent with the original set.) Thus anything provable about the abstract program remains true for any of the concrete realizations of it.

The proof that an arbitrary set of transformations is correct may be difficult in general. However, as discussed in [7], if each transformation in the set is itself "semantics-preserving" (i.e., replaces a part of a program with program text which does not contradict the meaning of the original text), the correctness of the transformational process is guaranteed (if it terminates). Usually it is quite easy to see that an individual transformation is semantics-preserving, especially when it is based on a mathematical property.

Finally, the fact that the abstract program is closer to the mathematical formulation of a problem than is an ordinary program means that its correctness is much easier to prove. In the present example (but not in general) the abstract program and its specification are almost identical; about the only thing which must be verified is that the product and assignment involve consistently-dimensioned arrays.

Incidentally, the fact that the concrete realizations of (3) do not cause out-of-bounds subscript references when executed follows from the fact that (3) involves consistently dimensioned arrays and the fact that those transformations which introduce subscripts also simultaneously introduce the index sets for them based on the array dimensions. (See Backus [1], section 5, for some discussion of the significance of this.) This two-stage proof is much easier than showing directly that (2) does not execute out-of-bounds subscript references. The difference is even more dramatic for an abstract Gaussian elimination algorithm and a realization of it employing implicit pivoting; the subscripts in the latter program are themselves subscripted variables, and it is very difficult to prove directly from the code that they are always in bounds.

WHY TRANSFORMATIONS?

It is perhaps interesting to conclude by posing the questions: Why use a program transformation system to construct concrete realizations of abstract programs? Why not simply devise an extended language for specifying abstract programs and a processor to translate it into an existing language (e.g., EFL [Feldman] and Bayer and Witzgall's Complete Matrix Calculi [3]) or directly into machine language? Or, why not implement by hand a relatively fixed ensemble of routines for different data representations and call them as appropriate to a particular user's needs?

Clearly, these alternatives are not completely distinct, for the "processor" for an extended language might consist of a collection of transformations, while some transformations insert code which could be thought of as very small subroutines. However, what I call a program transformation system is distinguished from the other two approaches primarily because it provides a high-level notation for specifying and applying source-to-source program transformations and because it can manipulate any programming-language construct (not just subroutines and arguments). Transformation systems of this type include not only TAMPR, but also those proposed by Bauer [2], and by Loveman and Standish (see [6]).

In my experience, the idea of providing for abstract program specification through a fixed extended language is too static an approach to be effective. The work discussed here is far from complete, yet already it has undergone numerous revisions. Had a particular notation been fixed, or had the transformations been implemented in a fixed processor, they would have been very difficult to modify. Moreover, emphasis on Designing a Language tends to cause one to get lost in a tangle of language-related issues which are not very germane to abstract program specification; indeed the expression and function notation available in slightly modified Fortran or in Algol seems quite adequate for experimentation. Finally, even extensible languages, which permit the definition of new data types and operators (e.g., Algol 68), do not usually provide a means for easily specifying optimizations (especially global ones) for these extensions. As we have seen, such optimizations are both fundamental and rather specific (e.g., the row analog of (6), which shows that n instead of n(n+1)/2 divisions suffice) and it is unreasonable to expect them to be built into a general-purpose language processor. Specifying these optimizations by transformations not only allows them to be easily tested and modified, it also permits them to be selectively applied to classes of programs which may reasonably be expected to use them.

Similarly, the implementation, by hand, of a set of subroutines tailored to various properties is also static and not very reliable; moreover, the set needed is very large, being the product of the number of variables, the number of representations of each, etc. In a transformational formulation, the number of transformations needed behaves more like the sum (plus some initial overhead). Thus, use of transformations enables one to manage the complexity of the problem and thereby greatly enhances reliability.

CONCLUSION

I have sketched how various concrete executable programs can be constructed automatically from an abstract prototype program by applying transformations based on theorems of matrix algebra and on "algebraic" properties of programming languages. Although this research has just begun, it offers the hope of being able to provide a user with highly efficient programs tailored to his environment while maintaining the advantages of high reliability and low cost associated with routines from the best mathematical software libraries. Moreover, the transformations which produce such programs themselves represent a valuable resource: a formal codification of rules for writing linear algebra programs.

ACKNOWLEDGMENTS

Work on the derivation of row and column oriented programs from an abstract prototype was begun by Brian Smith in conjunction with Janet Bentley while Brian was on sabbatical at the NAG Central Office, Oxford. This work was supported by NAG; preliminary results are reported in [10]. I am indebted to Brian for numerous discussions which helped the work discussed here to evolve into its present form.

REFERENCES

1. J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Comm. ACM 21, 8, Aug. 1978, 613-641.
2. F. L. Bauer, Programming as an Evolutionary Process, Proc. 2nd Int'l Conf. on Software Engineering, San Francisco, 1976, 223-234.
3. R. Bayer and C. Witzgall, Some Complete Calculi for Matrices, Comm. ACM 13, 4, April 1970, 223-237.
4. J. M. Boyle, Mathematical Software Transportability Systems -- Have the Variations a Theme? in Portability of Numerical Software, Lecture Notes in Computer Science, No. 57, Springer-Verlag, 1977.
5. J. M. Boyle and K. W. Dritz, three papers on the TAMPR system in J. R. Bunch, Ed., Cooperative Development of Mathematical Software (available from the authors).
6. J. M. Boyle, K. W. Dritz, O. B. Arushanian, and Y. V. Kuchevskiy, Program Generation and Transformation -- Tools for Mathematical Software Developement, Information Processing 77, North Holland 1977, 303-308.
7. J. M. Boyle and M. Matz, Automating Multiple Program Realizations, Proc. of the MRI Symposium, XXIV: Computer Software Engineering, Polytechnic Press, 1977, 421-456.
8. C. B. Moler, Matrix Computations with FORTRAN and Paging, Comm. ACM 15, 4, April 1972, 268-270.
9. C. C. Green, The Design of the PSI Program Synthesis System, Proc. 2nd Int'l Conf. on Software Engineering, San Francisco, 1976, 4-18.
10. B. T. Smith, Portability and Adaptability -- What are the Issues? in D. Jacobs, Ed., Numerical Software -- Needs and Availability, Academic Press, 1978, 21-38.