Short Note

Proposal of a Scheme for Linking Different Computer Languages —from the Viewpoint of Algebraic-Numeric Computation—

TATEAKI SASAKI*, YOSHINARI FUKUI**, MASAYUKI SUZUKI*
and MITSUHISA SATOU***

A simple and promising scheme for linking different computer languages is proposed. The scheme makes few assumptions about the operating system on which it is used, yet it is so promising that even programs written in quite different kinds of languages can be linked together. In particular, we discuss linking FORTRAN and Lisp (that is, a Lisp-based algebraic language), giving some proposals on the operating system and FORTRAN.

1. Introduction

Although computers are widely used, current computer languages are so specialized that each is used only for restricted kinds of computation, and a single language system (including its library) is not enough for many practical types of computation. For such computations to be performed satisfactorily, it is desirable that programs written in different computer languages should be executable in a single run as if they were executed as a single program written in a single language. For brevity, we call this computation style *hybrid computation*.

Many scientists and engineers have expressed a strong desire for a hybrid algebraic-numeric system [1, 2, 3]. For these users, it is inconvenient to split computation into algebraic and numeric types, because most of their computations include both. Hybrid algebraic-numeric computation is also desired by numerical analysts and computer algebrists [1, 2, 3]. In mathematical computation, algebraic and numeric methods are often complementary, and efficient and powerful algorithms can be designed by using hybrid methods. Furthermore, hybrid systems are sure to enhance computational flexibility greatly. For example, it is easy to generate FORTRAN programs by using Lisp. Hence, with an advanced hybrid system, it is possible to execute FORTRAN programs that are generated in run time. We call this operation *dynamic program generation*.

*Institute of Physical and Chemical Research, Wako-shi, Saitama 351, Japan.
**Total Information & Systems Division, Toshiba Corporation, Horikawa-cho, Saiwai-ku, Kawasaki-shi, 210, Japan.
***Department of Information Science, University of Tokyo, Hongo, Bunkyo-ku, Tokyo 113, Japan.

Many hybrid systems for algebraic-numeric computation have been constructed, but none of them can be said to be satisfactory. In this paper, we propose a new scheme for a hybrid system that is quite simple but seems to be very promising.

2. Desirable Hybrid Computation Scheme and Criticism of Previous Schemes

A different scheme for hybrid computation may be desirable if the languages concerned are different. However, so far as algebraic and numeric computations are concerned, the desirable scheme is almost unique for two reasons: one is that a huge number of library programs have been accumulated in these fields, and nobody wants to rewrite them in a new language; the other is that users have a very conservative attitude toward modifying programs that they have already written and debugged.

Thus, we assume that a desirable algebraic-numeric hybrid scheme should satisfy the following conditions:

(1) A FORTRAN system and its library as well as an algebraic system and its library can be fully used in such a way that each system can be used separately;

(2) FORTRAN and algebraic programs can be linked together, easily and with minimum changes to the programs, so that the algebraic system can be called from FORTRAN programs and the FORTRAN system from algebraic programs;

(3) If the realization of a hybrid system requires modifications and/or extensions of the languages and systems concerned, these modifications and/or extensions must be as few as possible.

Hybrid algebraic-numeric computation has been considered since about 1965, when many users began to use

general-purpose algebraic systems. Since then, several hybrid schemes have been proposed and implemented. However, these schemes are not satisfactory in terms of the above conditions. We will explain this by briefly surveying conventional algebraic-numeric hybrid schemes.

The algebraic-numeric hybrid schemes proposed so far may be summarized as follows:

(a) Constructing a file handler to transport common data via files;

(b) Implementing an algebraic system as a set of FORTRAN subroutines;

(c) Implementing a FORTRAN-to-Lisp translator in a Lisp system;

(d) Implementing an algebraic system in FORTRAN or a language that has high familiarity with FORTRAN.

Many users who want to perform hybrid computation now employ scheme (a), in which the file handling is often done by the users themselves [4]. According to our definition of hybrid computation, this scheme cannot be viewed as hybrid.

Scheme (b) was adopted by FORMAC [5]. This was the first successful general-purpose formula manipulation system, and was designed as an extended feature of FORTRAN. However, the manipulation of formulas covers a large number of operations, and including FORMAC into FORTRAN necessitated a drastic extension of FORTRAN. A moment's thought will show that this is rather obvious.

Scheme (c) was adopted by MACSYMA [6]. MACLISP, in which MACSYMA is written, has many facilities for driving FORTRAN, such as a translator from FORTRAN to Lisp [7], a compiler for fast execution of numerical programs written in Lisp [8], and a facility for calling FORTRAN subroutines [9]. Since facilities and data types in Lisp are much richer than those in FORTRAN, and Lisp can also be used for numerical computation, scheme (c) will be enough for performing algebraic-numeric hybrid computation. However, implementing this scheme requires a tremendous amount of programming labor. Furthermore, the system is not easy to maintain since it becomes quite large.

Scheme (d) was adopted by SMP [10], a rather new algebraic system developed at CalTech. SMP is written in C, and a subset of C for numeric computation is quite familiar with FORTRAN. Therefore, if a preprocessor is written to translate FORTRAN to C, FORTRAN programs can be executed easily on the C system. However, this scheme does not allow a large number of algebraic programs written in Lisp to be used, and they must be rewritten in C.

3. New Hybrid Scheme

Figure 1 illustrates a desirable hybrid computation on language systems A and B: given a problem for which a

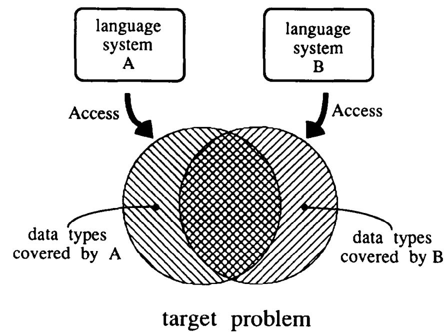

Fig. 1.

solution cannot be obtained by a single language system, we solve it by applying the facilities of other languages. We assume that, in Fig. 1, the data types covered by systems A and B overlap; otherwise, linking A and B is meaningless. However, we admit that the overlapped data types may be differently represented in systems A and B, and this is actually so. Therefore, one essential problem in linking different languages is how to adjust the discrepancy between internal data representations. Note that all the hybrid schemes listed in the previous section eliminate this problem by implementing the system in a single language.

The hybrid scheme we propose has the following four features:

(i) The hybrid program is a mixture of programs written in A and B, with commands specifying transition from one language to another, and is executed as sequential processes, that is, corountines;

(ii) The data used commonly in A and B are declared before execution, and are allocated separately to systems A and B. However, they are defined for the user as if they were allocated to a single area common to both A and B;

(iii) When the program execution is moved from A to B, for example, the common data allocated to A are copied to the common data area for B by converting their internal data representations;

(iv) The programs generated dynamically are written into a file shared commonly and transported between systems A and B.

Let us explain the above points by means of figures. For definiteness, we assume that command **ENTER B (ENTER A)** causes a transition from system A (B) to system B (A). Furthermore, command **GCOMMON** declares the data used commonly in systems A and B. The identifiers declared by this command can be used in both A and B. In addition to these commands, we need the command **EXIT** which, when used together with an **ENTER** command, specifies the end of a program block. (Without the **EXIT** command, we cannot recognize the execution flow correctly in nested subprograms.)

Figure 2 illustrates a hybrid program in our scheme.

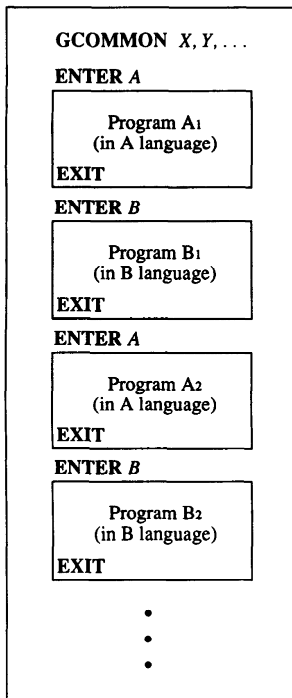

Fig. 2.

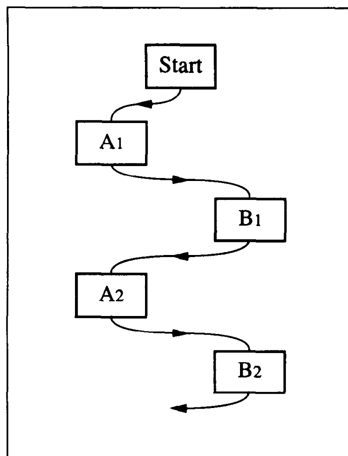

Fig. 3.

The hybrid program is first separated into A and B programs by a preprocessor. During this separation, **GCOMMON**, **ENTER**, and **EXIT** commands are replaced by corresponding codes in languages A and B. The programs separated are compiled by their respective compilers, if necessary, and executed as shown in Fig. 3.

We may emphasize the simplicity of our scheme: only a few extra commands are introduced to link A and B. That the programs written in A and B are executed sequentially, as weakly coupled processes, is one of the key features of our scheme. Such an execution style is already familiar in some operating systems, such as UNIX, and requires only minor modifications even in earlier-generation operating systems.

One might worry that the representation conversion and copying of the common data will be very costly. This concern is justified if the transition between A and B occurs very frequently. However, the main target of hybrid systems is actually scientific computation composed of large tasks whose execution takes much more time than the exchange of common data. Furthermore, operating systems in the near future will provide a facility for shared memory. When this happens, the overhead for exchanging common data will greatly decrease.

4. Linking FORTRAN and Lisp

In the preceding section, we stated our hybrid scheme in a general form to show that it can be applied to various languages. In this section, we discuss the hybrid scheme by confining ourselves to linking FORTRAN and Lisp (or a Lisp-based algebraic language). We select FORTRAN and Lisp because they are the most important languages for numeric and algebraic computations, respectively. For convenience of explanation, we call our hybrid system ANS (*A*lgebraic *N*umeric *S*ystem).

4.1 Data Types

The manner of treating precision is quite different in FORTRAN and Lisp, which causes a serious problem in connecting these languages. The precision of a number in FORTRAN is strictly fixed throughout the computation, while it will be changed automatically in Lisp. Thus, a number n that is defined as a fixed-precision integer in FORTRAN may be changed to an arbitrary-precision integer in Lisp. When this happens, it cannot be converted back to a fixed-precision integer in FORTRAN. If the ANS, not the user, is responsible for this precision change, it is necessary to extend the FORTRAN syntax. We propose the following extension of FORTRAN as a simple answer to this problem:

> *Introduce a hybrid data type of integer-real numbers in which a number is an integer so long as it is representable as a fixed-precision integer, and is automatically converted to a fixed-precision floating-point number when the integer precision overflows.*

Note that introduction of this hybrid integer-real number does not conflict with the static memory allocation scheme in FORTRAN. Note further that, although dynamic data type checking is necessary for this number, there is no need to check the data types of other numbers dynamically in FORTRAN. Hence, the computation cost will not be greatly increased by this in-

troduction.

There are many other differences between data types in FORTRAN and Lisp, but most can be adjusted by conversion of data representations. However, we insist strongly on the unification of external representations of strings and arrays, because they are important practically.

4.2 On Dynamic Program Generation and Loading

Dynamic program generation is an indispensable facility in hybrid algebraic-numeric computation. This facility requires an operating system to allow dynamic linking of FORTRAN subroutines. However, most current operating systems are not equipped with this facility.

Fortunately, most FORTRAN subprograms that are generated dynamically in algebraic-numeric computation are function subroutines whose names and usage are known before generation. Such cases of dynamic program generation can be processed so long as the OS allows dynamic loading of FORTRAN subroutines. That is, with the dynamic loading facility, it is possible to load dynamically and execute the body of a subroutine whose name has already been registered and linked with other subroutines. Dynamic loading is much simpler than dynamic linking.

```
;;
;; Common variables declaration
;;
$GCOMMON ((int CO 10))                ;; Vector for coeffs of polynomial.
$GCOMMON_FUNC                         ;; Functional variable holding
     ((double RECREL ((double X))))   ;; Newton's recurrence relation.

;;
;; Enter language C mode (Numerical language)
;;
$ENTER (C)

# define EPS 1.0e-6                   ;; Accuracy of coeffs.
# define Max 20                       ;; Maximum count of iteration
main()
{
     double x0, x1, fabs();
     int i;

     for (i=0; i<10; i++)             ;; Read coeffs. and set them to
          scanf("%i", &CO[i]);        ;; common variables.
;;
;; Enter REDUCE
;;
$ENTER (REDUCE) {
     F := 0;
     FOR I := 0:9 DO                  ;; Make an equation with
          F := (F*X + CO(9-i));       ;;  coeffs. given by common variables.
     RECREL := X - F /DF(F, X);       ;; Calculate recurrence relation.
$} EXIT
;;
;; Generate new function to calculate the recurrence relation.
;;
$LOAD_FUNC (RECREL)                   ;; compile and load
;;
;; Newton's iteration in C
;;
     x0 = 0.6;
     for ( i=0; i<Max; i++ ) {
          x1 = (*RECREL) (&x0);       ;; Calculate the next approximation.
          if ( fabs(x1-x0)<EPS )      ;; Test of convergence.
               break;
          x0 = x1;
     }
     printf("Result = %lf\n", x1);
     exit_ans();
}
$EXIT
```

Fig. 4 Example of an ANS program.

4.3 Preliminary Implementation and an Example

The above investigation of our hybrid scheme for ANS leads us to believe that realization of ANS is easy so long as the operating system provides us with some necessary facilities. In order to confirm this, we have implemented ANS preliminarily on the UNIX operating system. In the preliminary implementation, we have linked REDUCE with C instead of FORTRAN, because we have no FORTRAN system on our UNIX operating system. Note that numerical computation in C is almost the same as in FORTRAN, and that we have realized most of the facilities of ANS in our implementation. We found that the implementation was quite easy. (See Suzuki et al. [11] for a detailed description of the implementation; in particular, they describe how the internal data representations are converted by a simple mechanism). We found also that the inter-process communication facility of UNIX [12] played an essential role in this implementation; this facility allowed us to realize coroutine processing easily. On the other hand, it is rather difficult to implement ANS on an operating system that does not allow communication between different processes. Hence, in the next section, we make several proposals to operating system designers from the viewpoint of hybrid computation.

It should be mentioned that the notion of linking software by a process communication facility has already been proposed by Purtilo [13].

Figure 4 shows a sample ANS program in the current preliminary version. Given $c_9, c_8, \ldots, c_0$ as input data of numbers, the program solves a univariate equation

$$c_9X^9+c_8X^8+\cdots+c_0X^0=0$$

by Newton's method. In the program, the GCOMMON variable *CO*, which is declared by the command **$GCOMMON** (every ANS command in the preliminary version begins with $), is an integer array of size 10 and is used for inputting polynomial coefficients. The variable *RECREL* is also declared to be common (in this case, to be a function name). A common variable declared by **$GCOMMON_FUNC** is, when defined in REDUCE and transferred to C, converted automatically to a C function subroutine. The C subroutine thus generated dynamically is compiled and loaded by the command **$LOAD_FUNC**.

The program is executed as follows: first, the polynomial coefficients are read in by the C system, then the REDUCE system calculates the function RECREL algebraically. Finally, the C system executes Newton's numeric iteration procedure. Note that the numerical evaluation of RECREL is not executed by REDUCE but done as a C subroutine.

5. Proposals on OS, FORTRAN, and Lisp

We have proposed a simple hybrid computation scheme in which programs written in different languages are executed sequentially as weakly-coupled processes. The scheme makes it possible to construct a flexible and advanced computation system with a minimum of implementation labor.

When applied to FORTRAN and Lisp, our algebraic-numeric hybrid scheme requires the operating system to be equipped with the following facilities:

(O1) Execution of FORTRAN and Lisp programs as coroutines;

(O2) Data handling (mostly, conversion of representations and copying) between data allocated to FORTRAN and Lisp systems;

(O3) File sharing between FORTRAN and Lisp systems;

(O4) Dynamic loading of FORTRAN object codes (dynamic linking of FORTRAN program is better but dynamic loading is almost sufficient).

We think that the above requirements will be easily acceptable to operating system designers.

In addition to the above requirements for an operating system, the following extensions of FORTRAN and Lisp are necessary or desirable:

(E1) Unification of the user-level definitions of arrays and strings in FORTRAN and Lisp;

(E2) Definition of hybrid integer-real numbers in FORTRAN. (For the hybrid numbers, see Section 4.1.)

References

1. Brown, W. S. and Hearn, A. C. Applications of Symbolic Algebraic Computation, *Comput. Phys. Commun.*, **17**, 207-215.
2. Ng, E. W. Symbolic-Numeric Interface—A review, in *Lec. Notes Comput. Sci.*, **72** (*Proc. EUROSAM '79*) (1979), 330-345.
3. Mitsui, T. Interface between Algebraic and Numerical Computations, *Johoshori* (*Bulletin of Inf. Process. Soc. Japan*), **27** (in Japanese) (1986), 422-430.
4. Ojika, T., Watanabe, S. and Mitsui, T. Hybrid Manipulations for the Solution of a Large-Scale System of Nonlinear Algebraic Equations, *J. CAM. Intern'l Congress on Computation and Applied Mathematics*, Univ. of Leuvan, Belguim (1984).
5. Sammet, J. E. and Bond, E. Introduction to FORMAC, *IEEE Trans. Electron. Computers*, EC-**13** (1965), 386-394.
6. The MATHLAB group, *MACSYMA Reference Manual, version 9, Lab. Comput. Sci. MIT* (1979).
7. Pitman, K. M. A FORTRAN→LISP Translator, *Proc. 1979 MACSYMA User's Conf.* (*MIT*) (1979), 200-214.
8. Steel, G. L. Jr. Fast Arithmetic in MACLISP, *Proc. 1977 MACSYMA User's Conf.* (*NASA*) (1977), 215-224.
9. Lanam, D. H. An Algebraic Front-End for the Production and Use of Numerical Programs, *Proc. SYMSAC '81* (1981), 223-227.
10. Cole, C. A., Wolfram, S., et al. *SMP Handbook, version 1*, CALTEC (1981).
11. Suzuki, M., Sasaki, T., Fukui, Y. and Sato, M. A Hybrid Algebraic-Numeric System ANS and Its Preliminary Implementation, *Proc. EUROCAL '87* (*Lec. Note Comput. Sci.* **378**), 163-171.
12. Lettler, S. J., Fabry, R. S. and Joy, W. N. *A 4.2BSD Interprocess Communication Primer* (1983).
13. Purtilo, J. M. Application of a Software Interconnection System in a Mathematical Problem-Solving Environment, in *Proc. SYMSAC '86* (1986), 16-23.

(Received June 11, 1987; revised May 10, 1989)