数理解析研究所講究録
第 486 巻 1983 年 154-165

12 DESIGN OF A GENERAL COMPUTER ALGEBRA SYSTEM[+)]

(佐々木建昭) (渡辺隼郎) (古川昭夫)
T. Sasaki,[*)] S. Watanabe,[**)] and A. Furukawa[***)]
(Tateaki) (Shunro) (Akio)

*) The Institute of Physical and Chemical Research
Wako-shi, Saitama 351, Japan

**) Department of Mathematics, Tsuda College
Kodaira-shi, Tokyo 187, Japan

***) Department of Mathematics, Tokyo Metropolitan University
Setagaya-ku, Tokyo 158, Japan

Abstract

The authors are developing a general computer algebra system named GAL. The purposes of GAL are 1) to provide for users in various fields a powerful and efficient system for manipulating as general mathematics as possible and 2) to provide for algorithm implementors a general and efficient language for computer algebra. The system is decided to be of three-layer structure, the core, the intermediate layer, and the surface layer, so as to attain data abstraction and data type clarification. Furthermore, concepts of global data types and local data types are introduced so as to ease the data type handling. This paper considers a design principle of general computer algebra systems and gives a conceptual description of GAL.

+) Work supported in part by The Kurata Foundation.
+) Revised version of the paper presented at Programming Symposium, Jan.11-13, 1983, Hakone, Japan.

§1. Introduction

At SYMSAC '81, Cannon[1] and, independently, Jenks and Trager[2] proposed modernization of computer algebra systems. Cannon claimed the modern algebra systems to be such that which "enable a user to compute in a wide range of different structures" and which "permitt a user to define the structure in which he wishes to compute." His claim is nothing but a generalization of algebra systems. On the other hand, Jenks and Trager proposed a computer algebra language with "extensible parametrized types and generic operators." The approach of these authors is apparently influenced by recent theory on programming languages,[3,4] in particular by abstraction and parameterization of data types. However, it should be commented that their approach originates, via MODLISP,[5] from MOD-REDUCE[6,7] by Hearn in 1974. One of the main purposes of MOD-REDUCE is to enable system programming using only abstract data structures. A design of computer algebra language based on abstract data structures was also proposed by Ausiello and Mascari[8] in 1979, although their language handles very low level data structures.

Anyway, computer algebra is rapidly invading a wide area of mathematics, hence it now necessitates systems which can handle as general mathematics as possible, and several ideas have been presented for constructing such systems. This paper presents another idea. The idea itself is rather traditional compared with those proposed in recent theory on programming language but the implementation is easy and it seems to make the structure of the system simple. According to Hoare,[9] simplicity is a very important condition for system complex.

§2. Requirements on general computer algebra systems

We first consider requirements on general computer algebra systems from not only the viewpoint of generalness and efficiency of the system but also the viewpoint of large-scale complicated system. Analysis of such requirements is indispensable for designing a general algebra system. We think the followings are the most important

properties of the general computer algebra system:

1) capability of manipulating general mathematics,
2) easiness of system programming,
3) integrability of algorithm modules,
4) extensibility and maintainability,
5) accessibility from casual users,
6) efficiency.

Let us explain each of these properties in some details.

Someone may say that the existing computer algebra systems such as REDUCE,[10] MACSYMA,[11] or SCRATCHPAD[12] are enough general to manipulate many kinds of formulas appearing in science and engineering. However, they are never called general computer algebra systems. For example, if one wants to manipulate logical formulas asserting a theorem on geometry, none of the above-mentioned systems is available. A general computer algebra system should be able to manipulate as general mathematics as possible, including sets, logical formulas, and so on.

A general computer algebra system will cover a wide area of mathematics, hence collaboration of many experts in various fields of mathematics and algorithms is required to construct the system. These experts are not always familiar with whole algebra system, rather they know only about their respective areas of mathematics and algorithms. Furthermore, one algorithm module is usually programmed by using many other modules programmed by others. Therefore, easiness of system programming and integrability of algorithm modules are crucially important.

A general computer algebra system will become very large-scale, and construction of a computer algebra system usually requires a long period. For example, MACSYMA was begun to construct more than ten years ago and it is still growing actively. Furthermore, algorithms are rapidly advancing, hence algorithm modules should be continuously renewed. Therefore, extensibility and maintainability are also very important factors. Many of the computer algebrists do not care about this point

although they are quite enthusiastic for efficiency. This paper discusses, however, little about this point.

As for accessibility and efficiency, we have already enough experience and ideas. One notice is that efficiency often contradicts with generalness of the system. Our viewpoint on this dilemma is that generalness is more important than efficiency and the system is satisfactory only if it is efficient in manipulating large formulas which are often encountered in applications.

§3. Three-layer structure of GAL

In making a general computer algebra system satisfy the properties 2, 3, 4, and 5 presented above, data abstraction and data type clarification seem to be crucially necessary.

Data abstraction[3] is such that the internal data structures are masked from the users as much as possible and only well-defined abstract operations are prepared for the users to make access to the data structures. Hence, the programmers are maximumly free from the complication of internal data structures. Consequently, programming becomes easy and compatibility of programs by different programmers is attained so far as the programmers do not define their own data structures globally. Furthermore, integrity of the system increases because the programmer cannot manipulate internal data structures arbitrarily.

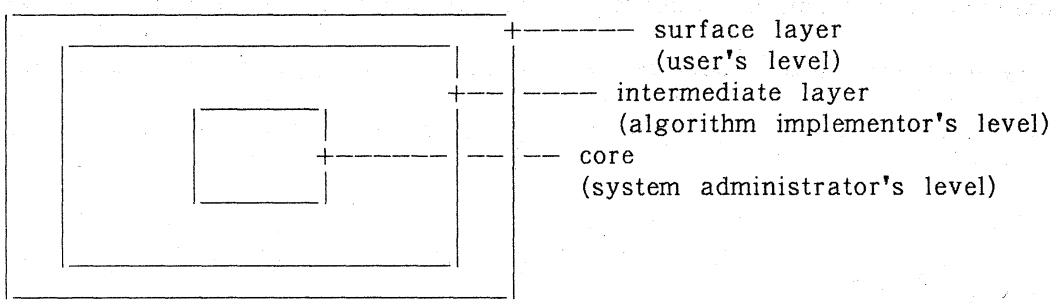

Fig.1. Three-layer structure of GAL.

Considering the point discussed above, GAL is designed to be of three-layer

structure, the core, the intermediate layer, and the surface layer (see Fig.1).

The core of GAL constitutes an algebraic programming system: it prepares many predicates and operators on algebraic data types defined in the core. The data structures defined in the core are completely confined to the core and masked from the outside. The core is programmed by several system administrators, and other system implementors (algorithm implementors) and users can have access to the internal data structure only through the programming language defined in the core. The language is, however, detailed enough to program efficient algorithm modules. With this design of the system, the system administrators, who are responsible to the most complications of system construction and maintainance, may have only to worry about the core as well as interfaces to other layers and nothing else. The followings are examples of procedures for handling explicit data types:

```
PLUS:ALGNALGN(U,V)  : <ALGNumber> + <ALGNumber>,
PLUS:ALGNPOLY(U,V)  : <ALGNumber> + <POLYnomial>,
PLUS:ALGNRATF(U,V)  : <ALGNumber> + <RATFunction>,
PLUS:ALGNALGF(U,V)  : <ALGNumber> + <ALGFunction>,
etc.
```

The intermediate layer consists of an assembly of many algorithm modules, such as polynomial factorizer or differential equation solver. This layer will be constructed by collaboration of many experts in different areas of mathematics and algorithms. Because the modules are programmed in a common programming language, the modules may become universal if the language is standardized. Standardization of a language for computer algebra is eagerly desired because most modules in algebra systems are quite laborious to program. Note that, in our design, we can change data structures in core without damaging the modules in the intermediate layer.

The surface layer provides an application-oriented language for the users in various fields. The language should be simple enough to use even by a casual user. Current GAL adopts an extended REDUCE language, because it is elegant as well as popularlized and suited for this purpose.

§4. Global and local data types

A general computer algebra system is one of the systems which handle so many complicated data types. Data types in algebra systems are in general nested multiply, and single datum is often given more than one data type. Hence, in order not to make the users/algorithm implementors confused in handling data types, the data types should be well-classified and clearly defined and they should be handled easily by every programmer, i.e., data type clarification is required. Without data type clarification, programming by several algorithm implementors will soon go to self-inconsistency of the whole system.

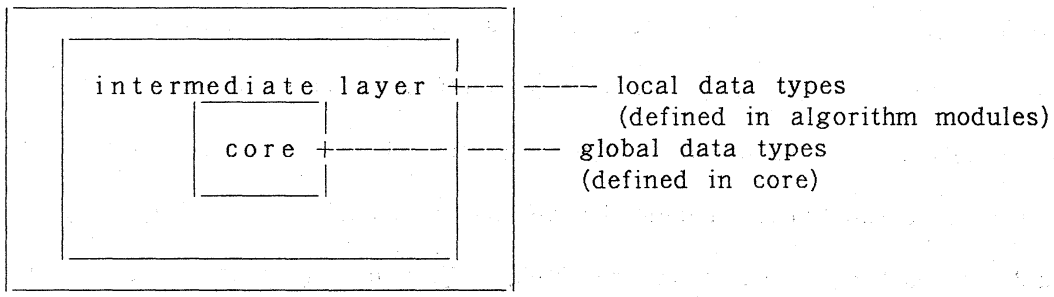

Fig.2. Two kinds of data types.

In order to attain the data type clarification, GAL introduces new concepts on data types, global and local. The global data types are defined by the system administrator in the core of GAL, and any user/algorithm implementor can use them. Global data types are used not only in application programs but also in algorithm modules and for communication among different modules. Therefore, every algorithm implementor should be familiar with the global data types. On the other hand, local data types are defined by implementors of algorithm modules and confined to the modules concerned. Local data types are defined only for constructing algorithm modules such as those for group theory, and most users and algorithm implementors need not know each of local data types. Furthermore, different algorithm modules may define the same local data types. Figure 2 illustrates two kinds of data types in GAL.

If we define all the necessary data types as global, specification of the global data types becomes very complicated. Such a complicated scheme of data type specification is hard to handle for most users and algorithm implementors. In the GAL, we define only a minimum number of basic and absolutely necessary data types as global, and other detailed data types are defined as local (see, §5). This policy will make the handling of data types quite simple: a algorithm implementor has only to know a minimum number of global data types and he may define his own data types without considering others.

For example, the current GAL defines data types of sets as global but no data types for groups. The data types of sets are absolutely necessary in implementing group theory. However, group theory itself is complicated in structure and the implementation requires a number of program modules. Hence, data types which are peculiar to group theory should be defined in modules for group theory and they should be used only within the modules. That is, the data types for groups are treated as local. With a similar reason, the core of GAL does not contain many of the generic data types such as ring or field.

Note that, in the above scheme of data type definition, the global data types should be enough general to allow good communications among algorithm modules. Without this property, integrability of the algorithm modules cannot be attained. Therefore, we must specify the global data types most cleverly.

§5. Internal representation in GAL

Data type clarification, extensibility, and efficiency are leading principles in designing internal data structures in GAL. Following these principles, we decided the internal data structures to be as follows.

1) We make the classification of data structures as simple as possible. The GAL is being implemented in LISP and all the data structures in core are classified by the following simple scheme:

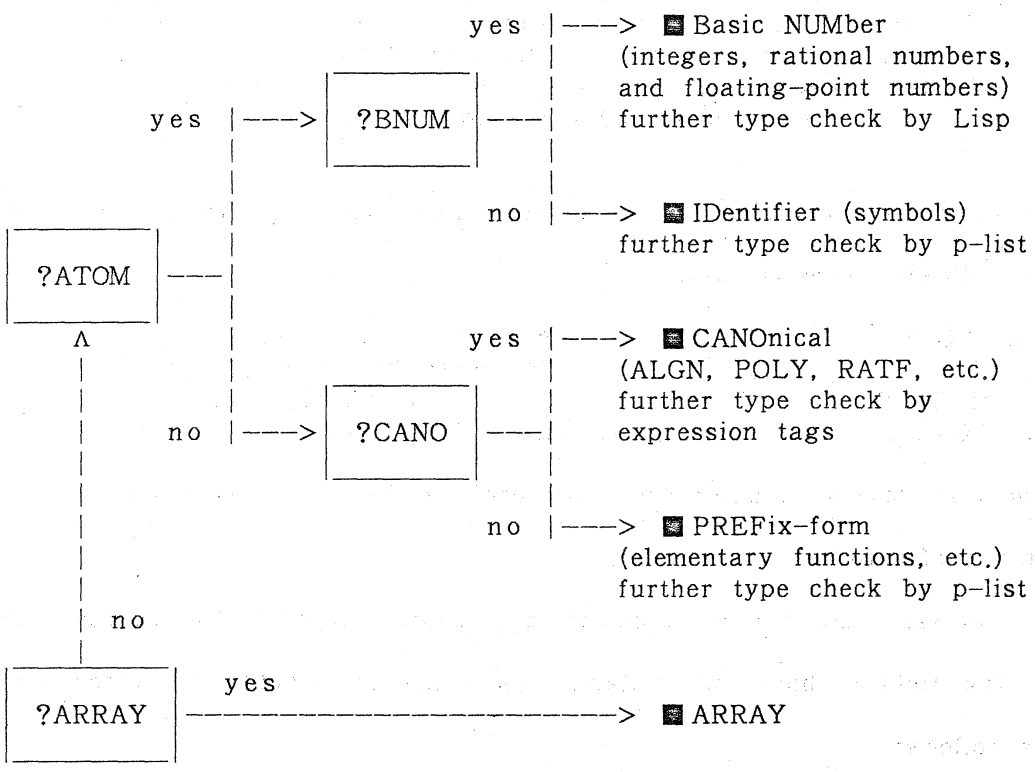

Fig.3. Classification of data structures.

Here, CANOnicals and PREFix-forms are of the form

CANO : (expression-tag . actual data structure),

PREF : (operator-name . list of operands),

respectively.

2) Expression representation in GAL is simple in that every expression is represented by one scheme and the user has no option of choosing data representations. This is consistent with the principle of data abstraction.

The following BNFs give a brief description of the data structures:

```
<Gal EXPRession> ::= <Basic NUMber> | <IDentifier(symbol)> |
                     <CANOnical> | <PREFix form> ;

<CANO> ::= <unstructured CANO> | <structured CANO> ;
      <unstructured CANO> ::= <ALGNumber> | <POLYnomial> | ... ;
      <structured CANO> ::= <VECTor> | <MATrix> | ... ;

<PREF> ::= <FUNCtion> | <Constructor EXPRession> ;
```

```
<FUNC> ::= <function name> . <operand list> ;
      <operand> ::= (<GEXPR> - <structured CANO>) ;
<CEXPR> ::= <CONStructor> . <term list> ;
      <term> ::= (<GEXPR> - <structured CANO>) ;
<CONS> ::= <rational CONS> | <set CONS> |
           <relational CONS> | <logical CONS> ;
```

Here, constructors are identifiers to construct expressions, and the current GAL prepares the following constructors:

```
rational CONS :      C+ (plus),  C* (commutative times), etc.,
set CONS :           S+ (union),  S* (intersection), etc.,
relational CONS :    R= (equal),  R> (greater than), etc.,
logical CONS :       L+ (logical plus),  L* (logical times), etc.
```

In our scheme, noncommutative elements are prefixed by the constructor NC* and classified into prefix-forms.

3) The expression-tag for a canonical is a short integer. The current GAL assumes the tag field to have 24 available bits which are divided into three parts and used as follows:

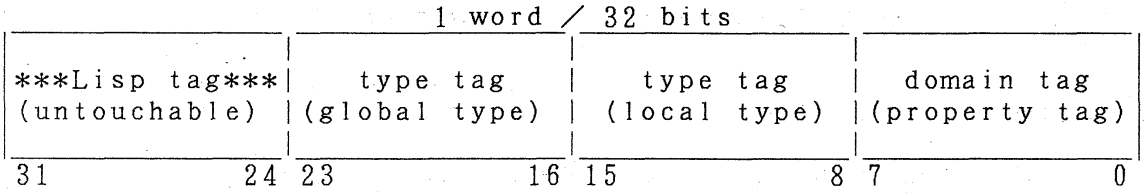

Fig.4. Specification of expression-tag.

The first part (23rd to 16th bits) is used for the global type tag. The second part (15th to 8th bits) is released to the algorithm implementors for defining local types. The third part (7th to 0th bits) is used for the domain/property tag. The first byte (31st to 24th bits) is used for a Lisp tag and GAL cannot handle it.

4) The followings are global data types defined in the current GAL:

```
ALGN    (algebraic number),        #ALGN# ← 1,
POLY    (polynomial),              #POLY# ← 2,
RATF    (rational function),       #RATF# ← 3,
ALGF    (algebraic function),      #ALGF# ← 4,
VECT    (vector),                  #VECT# ← 5,
MAT     (matrix),                  #MAT#  ← 6,
```

TENS	(tensor),	#TANS# ← 7,
INTVL	(interval),	#INTVL# ← 8,
FSET	(finite set),	#FSET# ← 9,
FOSET	(finite ordered-set),	#FOSET# ← 10,
INFSET	(infinite set),	#INFSET# ← 11,
INFOSET	(infinite ordered-set),	#INFOSET# ← 12.

The domain/property tag for INTVL is one of OO, OC, CO, and CC denoting ( ), ( ], [ ), and [ ], respectively. Domain/property tags for other data types are

INT,	(integer),	#INTDOM# ← 1,
BNUM,	(basic number),	#BNUMDOM# ← 2,
ALGN,	(algebraic number),	#ALGNDOM# ← 3,
POLY,	(polynomial),	#POLYDOM# ← 4,
RATF,	(rational function),	#RATFDOM# ← 5,
ALGF,	(algebraic function),	#ALGFDOM# ← 6,
GEXPR,	(GAL expression),	#GEXPRDOM# ← 7.

The domain/property tags specify the canonical expressions in details. For example, domain/property tag "INT" for polynomial means that the polynomial is of integer coefficients and that for finite set means that the elements of the set are integers. Current specification of type tags and domain/property tags favors polynomial and rational function arithmetic. However, we have enough space of introducing new type tags and domain/property tags.

Integer tags have many advantages over the name tags such as @MATRIX or @SET. First, integer tags can contain much more information than name tags: type, domain and more information can be represented by only a short integer. Second, integer tags are naturally ordered according to their largeness, and the ordering is useful in classifying data types. For example, data types of global type tags greater than or equal to #VECT# are structured canonicals in the current GAL. Third, by tabling every procedure for each data type, we can efficiently call the procedure by specifying only the type tag.

§6. Discussions

The largest weakness of our design is lack of completeness of the data type specification. In our design, requirements on global data types are quite severe:

the global data types should be enough simple for most users/algorithm implementors and, simultaneously, they should be enough general to allow implementation of as general mathematics as possible and to allow good communication among algorithm modules. The authors themselves do not think that the global data types in the current GAL are satisfactory. We must improve the specification of global data types again and again.

In order to accomplish completeness of data type specification, the most desirable way is to define the data types axiomatically. In this sense, the approach of Jenks and Trager[2] seems to be promising. However, the more mathematical the system is, the more difficult to handle the system is for nonmathematicians. Generally speaking, we may classify the algebra systems into two types, axiomatic and pragmatic. Which type of the system is better? Perhaps, this is the current largest problem in designing a general computer algebra system.

Acknowledgements

The authors would like to thank Dr. Y. Kanada and Mr. H. Murao for valuable discussions.

References

1) J. J. Cannon, "The basis of a computer system for modern algebra," Proc. 1981 ACM Symp. on Symbolic and Algebraic Computation, ACM, 1981, pp.1-5.

2) R. D. Jenks and B. M. Trager, "A language for computational algebra," ibid., pp.6-13.

3) B. H. Liskov and S. N. Ziles, "Programming with abstract data types," Proc. ACM Conf. on Very High Level Languages (SIGPLAN Notices Vol.9, No.4, 1974), pp.50-59.

4) J. N. Thatcher, E. G. Wagner, and J. B. Wright, "Data type specification: Parameterization and the power of specification techniques," Proc. SIGACT 10th Annual Symp. on Theory of Computing, 1978, pp.119-132.

5) R. D. Jenks, "MODLISP: an introduction," Proc. EUROSAM 79 (Lecture Notes in Computer Science No.72, Springer-Verlag, 1979), pp.466-480.

6) A. C. Hearn, "A mode analyzing algebraic simplification program," Proc. ACM 74, ACM, 1974, pp.722-724.

7) M. L. Griss, "The definition and use of data structures in REDUCE," Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation, ACM, 1976, pp.53-59.

8) G. Ausiello and G. F. Mascari, "On the design of algebraic data structures with the approach of abstract data types," Proc. EUROSAM 79 (Lecture Notes in Computer Science No.72, Springer-Verlag, 1979), pp.514-530.

9) C. A. R. Hoare, "The emperor's old clothes," Comm. ACM Vol.24, No.2, pp.75-83 (1981).

10) A. C. Hearn, "REDUCE-2 user's manual," 2nd Edition, Dept. Computer Science, University of Utah, 1973.

11) The MATHLAB group, "MACSYMA reference manual," 9th Version, Laboratory for Computer Science, MIT, 1977.

12) R. D. Jenks, "The SCRATCHPAD language," Proc. ACM Conf. on Very High Level Languages (SIGPLAN Notices Vol.9, No.4, 1974).