USER-DEFINED SYSTEMS FOR PURE MATHEMATICS

by John J Wavrik
Dept of Math
Univ of Calif - San Diego
La Jolla, CA

ABSTRACT

Some branches of mathematics require the ability to represent unusual types of data and algorithms. Some of the needs of workers in these areas can be met by providing them tools to construct their own special purpose software systems. Forth has properties that make it very useful for this purpose. It is possible to build such systems from re-usable parts and to create abstract types like "polynomial" and "matrix" which can, without writing new code, be specialized to polynomials with various kinds of coefficients and monomial parts, matrices with various kinds of entries, etc. -- and which can be combined to form compound structures. This paper discusses an approach resulting in systems which manipulate complex mathematical objects using standard Forth syntax and semantics.

INTRODUCTION

Most people associate mathematics with numerical computation -- an aspect of mathematics intimately tied to the early development of computers and computer languages. Most mathematics, however, manipulates symbols. A more recent development in computer systems is the appearance of programs like MACSYMA, Maple, Mathematica, etc. which are designed to perform symbolic computations. Both numerical and symbolic programs involve parts of mathematics which have a history of computation, which have a body of established algorithms, and which are of widespread use outside of mathematics itself. There are parts of mathematics, however, which have not been historically bound up with computation. Where experimentation is needed with data representation and algorithms. And which are not likely to receive the attention of commercial software developers. This paper is a survey of an approach to meet the needs of workers in these fields. It uses Forth to provide users with a toolkit for the building of small special purpose systems. In the spirit of object-oriented programming, these systems manipulate complex mathematical objects using standard Forth syntax and semantics. Program design emphasizes modular techniques so that component parts of the systems are reusable both by their author and others.

Problem-solving often benefits by the creation of a special purpose language designed for the problem domain. If a user is to create a problem-oriented language, the language used for implementation must allow lower level access to the computer than most conventional languages provide. The implementation language must allow the user to work at the level at which language features are made -- but must not restrict him to that level. In mathematics it is almost essential to have an interactive environment. Forth was selected for this project because it has these necessary characteristics.

MATHEMATICAL OBJECTS AND THEIR PROPERTIES

Mathematical objects have been and are studied independently of computer representation. An indication of the approach used by modern mathematics can be seen in the definition of the first and simplest mathematical objects: the natural numbers. The natural numbers are members of a class. The class has properties and operations which characterize it. (The natural numbers are characterized by having a distinguished number called 1 and an operation called "successor". Together they satisfy a set of assertions called the Peano Axioms.) New classes and properties are constructed from old classes and properties. The class of integers, for example, can be defined in terms of the class of natural numbers. Addition, subtraction, multiplication, and "long division" of integers are ultimately based on the successor operation. Even such a simple class as the integers has an amazingly large body of knowledge built up in the form of layers of theorems which ultimately stem from the axioms.

The idea of developing a body of mathematics from a small set of fixed axioms comes from the Ancient Greeks. The abstract point of view, which regards mathematical entities as members of a class of objects having certain properties, is an extension of the axiomatic approach of the Greeks. It developed gradually over several hundred years. It became the dominant viewpoint during the early part of this century. It is extremely useful because it allows the mathematician to think about objects in terms of their properties rather than in terms of the often very complicated ways in which they originally present themselves. Moreover, the mathematician can exploit similarities in the properties shared by classes of objects even if these objects are presented in very different ways. This point of view has become so pervasive that many mathematicians define mathematics as the subject which studies the consequences of a set of assertions about a certain class of objects having certain properties.

Treating things as objects, therefore, is quite natural to mathematicians. If the words "produced", "present", and "arise" in the previous paragraph are replaced by "implemented", the statement will almost sound like something said by a modern computer scientist about object-oriented programming. There are some important distinctions between object-oriented programming and the axiomatic view of mathematics: The computer scientist's "methods" act on objects in a class, while the mathematicians "operations" and "properties" can belong to the class as a whole as well as to individual objects. The computer scientist's "message passing" paradigm for triggering methods does not seem to match the mathematician's idea of applying an operation to a set of arguments. The mathematician has a broader set of techniques for creating new classes of objects -- the new classes reflect the properties of the old, but do not necessarily inherit them in the strong sense.

MATHEMATICAL OBJECTS AND THEIR REPRESENTATION

A naive view is to regard the computer as just a new medium for representing mathematics and regard the task of getting a computer to do mathematics as a simple problem of translation. The computer is, in fact, quite different from pencil and paper. Even in the traditional area of "number crunching" there are important differences between the computer's floating point numbers and the mathematician's real numbers. Moreover, conventional computer languages do not offer the facilities to perform computations even in parts of mathematics which have a heavy computational flavor. Until recently no computer language offered the ability to manipulate symbolic expressions and to do "exact arithmetic" with rational numbers and "arbitrary precision" integers. Efficient algorithms for machine computation are not always transcriptions of algorithms for hand computation (see, for example, Knuth's description of an algorithm for multiplying permutations in

"The Art of Computer Programming" 1.3.3). Many areas of mathematics are so new to machine computation that there are no conventional ways to represent the data and no body of algorithms. To make things worse, logicians have shown that some of the techniques used by mathematicians to construct new classes of objects (particularly the formation of quotient structures) can never be done algorithmically in general.

The use of computers in pure mathematics will require new approaches on both sides. Some thinkers, like Errett Bishop ("Foundations of Constructive Analysis"), suggest abandoning traditional mathematics in favor of a new "constructive" mathematics more in line with the algorithmic nature of machine computation. Others look to computer science to produce systems that more nearly approach what mathematicians currently do (e.g. "automated theorem proving"). A middle ground is to expect mathematics to be pursued in a more or less traditional way, but to have the computer serve as an experimental laboratory for examining examples and testing hypotheses -- this entails the exploration of data representation and algorithms. This paper describes an approach for producing software systems designed to facilitate experimentation. These systems are designed by users for specific problem areas -- providing the user with the ability to create a specialized data representation and vocabulary. In order to allow the time and effort involved in producing such systems to be recovered, they are built from packages or modules which are reusable and easily modified. The packages are constructed using a technique similar to object-oriented programming. They isolate representation dependence of the objects as much as possible from other parts of the package and from other packages. They allow for extension and modification of the features of a package while minimizing side effects. In this way the packages can be independently useful in other applications -- they constitute a toolkit for building systems.

This paper is based on the contention that, if done properly, there are advantages to allowing users to interact with a computer at a very low level. It is not a bad idea for people to think about how to make a chunk of computer memory behave like a mathematical object provided that (1) the language eventually allows them to forget the details, (2) the amount of code which specifically depends on representation can be kept small, and (3) mechanisms exist for building new objects from old without constant return to the representation details.

Let us examine a class of familiar objects: polynomials in one variable with integer coefficients. An object in this class looks like:

(1) $a_n x^n + ... + a_1 x + a_0$ (with the a_i integers)

Polynomials can be added, subtracted, multiplied and (sometimes) divided using algorithms taught in high school. A first attempt at a polynomial package treats the polynomial as an array of integers and defines the polynomial arithmetic operations explicitly in terms of this representation. This approach has several disadvantages. The polynomial package becomes tied to a particular type of coefficients and to a particular representation. Whenever an algorithm is implemented the mathematician must think in terms of representation rather than in terms of mathematical entities.

A better approach is to identify the essential properties of polynomials and define words which embody these properties. We may introduce a defining word POLY which sets aside storage for a polynomial. The most important task for a child word is to put the address of its storage block on the stack. (Child words often do only this -- but they can also define a referencing environment, put vocabularies in the search path, install values for system variables, etc.) The main features of polynomials in one variable are that they have coefficients and a degree. We define COEFF so that i P COEFF puts the ith coefficient of P on the stack. We define COEFF! to store a new value for a particular coefficient. We define DEG so that P DEG returns the degree of P. The definitions, of course, depend on the specific way we have chosen to represent polynomials. It may be, for example, that a polynomial occupies a block of memory in the dictionary and that the degree of a polynomial is stored at the base of its block -- so DEG is defined by : DEG @ ;. On the other hand, the access words can also be used to conceal the details of more complex memory allocation. In the case of a processor like the Intel 80x86, the coefficients for polynomials may reside in a segment outside the code segment which contains the dictionary. The polynomial P could put a segment address on the stack and COEFF and COEFF! would contain code for moving coefficients from the data segment to the stack. In this case, all of the manipulation involving segments takes place entirely within the access words and is transparent to the rest of the package. It might also be that a polynomial is not represented as an array, but rather as a linked list [although different access words and algorithms are usually used for "sparse" representations]. In any case, if we have been thorough enough in identifying the essential features of a class of objects, the implementation of these features will be the ONLY place where details of the representation are used. Should we change the representation, only the access words need to be changed. Aside from this practical advantage, we gain an important psychological advantage which should not be minimized: arithmetic of polynomials can be implemented in terms of polynomials, coefficients, and degrees rather than in terms of memory locations. We can conceptualize in mathematical rather than computer terms.

The virtue of this approach becomes even more evident when we generalize. In (1) above we can allow the coefficients to be rational numbers, or floating point numbers, or polynomials in another variable, or matrices. All of the algorithms for polynomial arithmetic can be extended to these coefficient domains and, on an abstract level, the algorithms are the same. We add two polynomials, for example, by adding corresponding coefficients. We may also make similar statements about another familiar data type: matrices and their arithmetic. "Polynomial" and "Matrix" are general (some say "abstract") data types. The approach taken in the paragraph above makes it possible to implement a general polynomial or matrix package which can be used with arbitrary coefficients. Here is how:

We examine the algorithms for polynomial (or matrix) arithmetic and discover that the polynomial package must know certain things about the coefficient domain. It must know how to add, subtract and multiply coefficients (and, for some algorithms, how to divide). We may also need the size of coefficients in bytes, words to swap and dup them on the stack, and words to store and fetch them from memory. The creation of a general polynomial package, therefore, begins with an analysis of what a polynomial package must know about its coefficient domain. The polynomial package will need to know that coefficients can be added, but does not need to know how the coefficients are represented or how the addition is defined. We build the package using words like X+ for the addition of coefficients; XSIZE for the size of a coefficient; X. for printing a coefficient; etc. Here is how polynomial addition looks:

```
: P+ ( poly1 poly2 -- sum )  GETRES IS P2 IS P1
     P1 DEG  P2 DEG  MAX  DUP RESULT DEG!
     1+ 0 DO  I P1 COEFF  I P2 COEFF  X+
              I RESULT COEFF!
          LOOP  RESULT SETDEG  RESULT ;
```

P1 and P2 are initializable constants. Removing the arguments from the stack allows the algorithm to be implemented without stack manipulations which depend on coefficient size. GETRES obtains a temporary storage location, RESULT, for the result. SETDEG recalculates the degree of the result (since cancellation of coefficients could have occurred). [The details of a mechanism for managing temporary storage locations are found in a paper "Handling Multiple Data Types in Forth" by J Wavrik to appear in the Journal for Forth Application and Research.]

There are, of course, computerisms in this definition. The mathematician must find a place to put the result; compute loop limits; reset degrees; etc. The important thing, however, is that in this (and also other algorithms in the package) specific details of representation of the coefficients and polynomials do not appear. Elements of the coefficient domain are being treated as objects and the algorithms of polynomial arithmetic are expressed in terms of these objects and their properties. The end result is a polynomial package which treats polynomials as objects (with properties and operations): If F and G are names for polynomials then F G P+ will add F and G putting the result on the stack. Syntactically and semantically polynomials are handled in the same way as other Forth objects (indeed even the stack operations work as expected since polynomials are represented on the stack by their addresses).

The object-oriented viewpoint we have described becomes particularly useful when we realize that polynomials are just low level building blocks for even more complex structures. Computer algebra specialists pose as a challenge to create systems with the ability to handle compounding of structures. We might wish, for example, to work with matrices whose entries are polynomials with integer coefficients. The challenge is to create a matrix package, a polynomial package, and a way of joining them so that this and other combinations can be easily formed. Early computer algebra systems could not do this. An unusual feature of Forth makes it possible to meet this challenge in a fairly simple way provided the components have been built using an object-oriented modular approach: Forth allows several definitions of the same word to coexist in the dictionary. When a word is redefined the earlier version is still the referent for prior words (and can, by use of vocabularies, still be accessible). Here is a Forth solution to the problem of producing matrices of polynomials:

(1) We define, as above, general matrix and polynomial packages using X+, etc for the entries (coefficients).

(2) We now load an interface module which defines the coefficient operations to apply to the integers (i.e. : X+ + ; etc).

(3) Now we load the polynomial package. (We have just produced polynomials with integer coefficients.)

(4) Next load another interface module which defines coefficient operations to apply to polynomials (i.e. : X+ P+ ; etc).

(5) Finally load the general matrix package. At the top level the matrices have entries which are polynomials with integer coefficients.

The process is easy to visualize: we are putting three modules (integers, polynomials, matrices) together in somewhat the way that children make chains from pop-beads. At the two joints we put a dab of glue (the interface modules). If we compile the polynomial package into a separate vocabulary we still have access to the operations of its coefficient domain. All that is needed for this scheme is that for each potential coefficient domain we also create an interface module which defines the generic domain operations in terms of the operations of this package. Here is an example from a package produced in this way:

```
2 2 matrix mm   2 2 matrix nn

(( 1 2 3 )} ({ 1 0}) {( 2 1 )) (( 1 0 –1 }} mm matinput

mm mat.

 2
x   + 2 x + 3                                x

                                              2
2 x + 1                                      x   - 1


{{ 1 1 1 )} (( 2 0 0 )} ({ 1 1 }} {( -1 0 1 0 }} nn matinput

nn mat.

 2                                              2
x   + x + 1                                  2 x

                                               3
1                                            -x + x

mm nn mat*

mat.

 4     3      2                               4      3     2
x  +3 x  + 6 x + 6 x + 3                     x  + 4 x  +7 x

   3     2                                    5      3      2
2 x + 4 x + 3 x                             -x  + 6 x  + 2 x  - x
```

Object-oriented design produces computer systems based on classes of objects which they manipulate (rather than on functions they perform). Generality is achieved by realizing objects as instances of abstract types. The design is modular, and the modules interface in specific and limited ways. Modules encapsulate the details of the representation and manipulation of a class of objects—providing the outside world with a specific set of tools to be used to act on the objects. Large programs are built by hierarchically linking modules. Ease in writing and correctness are enhanced by allowing modules to interact using only the features provided by their interfaces. Some object-oriented systems impose further requirements: detailing the mechanisms which can be used to act on objects and requiring that code internal to a module be inaccessible from outside. The goal of this type of design is to enhance the reusability of component parts; to enhance extendibility of systems; to improve program correctness; and to speed the creation of systems.

We have outlined an approach to system design in this paper which resulted from a blend of the mathematician's object-oriented viewpoint with features inherent in the Forth language. Forth has several features which makes it useful for designing mathematical systems. It is able to isolate the properties, features, and operations associated with a class of objects in the actions of individual words. Forth extensibility allows applications to be built in a layered fashion permitting a gradual ascent from the level of machine representation to the level of ultimate abstraction. The fact that Forth provides an integrated assembly language makes it possible to increase speed; employ unusual memory allocation schemes; and interface with hardware without disturbing the high-level appearance of a package. The user's ability to create defining words is used to create classes of objects. The user's ability to create control structures and manipulate the input stream can be used to shape the appearance of the language. We feel that the approach taken in this work is natural, both from the point of view of representing mathematics and from the point of view of the characteristics of the Forth language.

A comment should be made about "information hiding" which is a prominent feature of object-oriented programming. We have discussed the reasons why a programmer should not have to be constantly aware of details of implementation for low level features. Forth's vocabulary mechanism or even explicit manipulation of dictionary links can be used (as in Richard Poutain's book "Object-Oriented Forth") to produce sealed modules. Most of our systems, however, are relatively small and are dynamic. They are built in a "middle out" fashion. Packages start with the most obvious features for export rather than an exhaustive list. For our purposes it has been found useful not to render implementation details inaccessible.