On the Design and Construction of Algebraic Manipulation Systems

Richard J. Fateman
University of California
Berkeley, California

ABSTRACT

We compare and contrast several techniques for the implementation of components of an algebraic manipulation system. On one hand is the mathematical-algebraic approach which characterizes (for example) IBM's Axiom. On the other hand is the more *ad hoc* approach which characterizes many other popular systems (for example, Macsyma, Reduce, Maple, and Mathematica). While the algebraic approach has generally positive results, careful examination suggests that there are significant remaining problems, especially in the representation and manipulation of analytical, as opposed to algebraic, mathematics. We describe some of these problems and some general approaches for solutions.

INTRODUCTION

Symbolic algebraic mathematics programs have been in use for several decades now. Some programs implement stand-alone algorithms which accomplish a particular task: say, factoring a univariate polynomial. Other programs appear as part of a monolithic system such as Macsyma, Mathlab '68, Axiom (formerly Scratchpad II), Reduce, Maple, Mathematica, MuMath, etc. (see Ref. 2). These systems attempt, in various ways, to resolve representation and abstraction problems for a large class of tasks and provide algorithms, user-interfaces and links to other kinds of programs (numerical, graphical), etc.

In this paper we will discuss our experiences in resolving some of the problems of system building. Most of these problems arise from the desire to build a nearly-autonomous system for mathematical problem representation and solution: the intent is for the system to make it unnecessary for the user to provide detailed programming at the level of data representation of basic mathematical concepts. We are *not* concerned here with projects which have been somewhat more limited in scope (when viewed as systems) and, in particular, those programs where the user – as opposed to the system – chooses representations and algorithms explicitly. A system such as SAC-2 requires that such choices be made and, as a consequence, at run-time there is nearly full isolation from linguistic and representational problems. This is not to imply that important algorithmic problems cannot be treated in such programs, just that SAC-2 does not concern itself with choosing the best representation for, say, 1/2.

We are, by contrast, discussing systems in which the representation and manipulation issues may be quite prominent. Whether $x^2/2 + x/6$ is a polynomial of degree two in x with rational coefficients, or should be converted to $(3x^2 + x)/6$, a ratio of two polynomials with integer coefficients, is important. Should this be a choice that is open to the user (or to an algorithm) at run-time? Should algorithms be available to (say) add in either representation? What if forms in two different representations are added? Other questions that may also have to be resolved include the nature of x: Is x a solution to an algebraic equation such as $x^2 + 1 = 0$? Is x a small quantity whose high powers are negligible? Identifying the proper decisions with respect to representation and computation for such items is a systems-design problem of great importance not only for calculational efficiency, but also correct semantics.

We describe two approaches to system building. We first describe what we call the prototype/hacker approach. This is not meant in a derogatory sense; as will be seen, we show that at least for the moment, it has some advantages. This approach emerged from the confluence of a number of historical streams: simple applications of the computer to symbolic problems in (say) physics or celestial mechanics, the availability of symbolic languages such as LISP, and the growing power of the computer to tackle problems (like indefinite integration) that seemed to require human intelligence.

The second approach is based on the realization that, by and large, a good deal of what we have required from symbolic mathematical systems can be described by "classical" modern algebra. This approach emerged from the confluence of these mathematical ideas with recent ideas from programming languages: abstract data types, procedures-as-data, strong typing, and hierarchies of data as used in "object-oriented" languages. The approach, in summary, is to construct a system based on mathematical categories, and write programs which have the maximum amount of generality possible, subject to the well-understood tenets of algebra.

The next two sections discuss the two approaches: their advantages and shortcomings. Then we discuss the prospects for overcoming some of the shortcomings of either type of system and makes some specific proposals for computing with analytic notions. We conclude with a brief summary.

THE PROTOTYPE/HACKER APPROACH

Historically, every early system for symbolic mathematics seems to have been a combination of struggling technology and uncertain algorithms. This approach has persevered, and can be characterized as roughly *get something of specific problem-solving interest to a working state, and extend it, while debugging, to be a more-or-less general system.*

In spite of various efforts to *systematically* design and execute a master plan for a general algebraic manipulation system, none has been entirely successful to date; thus, most distributed "practical" systems have been those built up using this prototype/hacker approach. Consider the case of Macsyma, a system which is about 20 years old now, and has precursors going back another 10 years. We can see that a number of fundamental problems which presented themselves along the way were not solved (Ref. 9). Speaking as a designer, implementor, and user of Macsyma, and an observer of other similar systems, I believe that we were unable to find a single, comprehensive, and effective, organizational structure to explain to users and programmers the system's mathematical and computational premises, and their consequences. Any such structure would have to accomodate all of the conflicting objectives of users, and maintain the viability of any facilities or algorithms that are useful in some context.

In order to give some food for thought, we'll mention just two system issues here.

- Syntax seems superficial, but is related to mathematical linguistic problems. Here is an example of a declarative "summation" notation used in mathematics: $\sum_{i=1}^{n} 1/2^i$. Here is an algorithmic language's similar but imperative notion "`s:=0; for i:=1 to n do s:=s+1/2**i; s`". These two examples are very similar in semantics *in some contexts*, but in others the semantics differ substantially: for example, the sum above for $n = \infty$ is quite reasonable as a clumsy way of writing 1, but the corresponding endlessly looping "`for`" loop is not meaningful. It's tempting (for a hacker) to use the first notation for the second.

- Equality is a notion deeply embedded in most systems. There is a necessity for determining when two expressions are equal (even when they "look" different). How much work should a system perform to ascertain that an expression is definitely not zero? Without such a determination, much of computer algebra is in principle impossible. Macsyma has numerous programs for simplification or testing for zero. Which should it use? Since the zero-equivalence problem is unsolvable over the class of all expressions representable in Macsyma, even the most powerful of the simplifiers will sometimes fail. It's tempting (for a hacker) to make some specific fallible choice.

While Macsyma is a prime example of the prototype/hacker approach, a number of more recent systems also reflect this development approach. This strategy characterizes Mathematica (Ref. 22), Maple (Ref. 5), SMP (Ref. 6), and MuMath (Ref. 20). I think it is accurate to characterize all of these systems as approaching symbolic mathematics through re-engineering of mostly pre-existing ideas and algorithms, with a modest attempt to provide better extension mechanisms for either the user or the system programmer.

One distinction between the prototype/hacking approach and the next approach we discuss, is the extent to which extension mechanisms permeate the system, providing a theoretical foundation for the mathematics, and a programming mechanism for system programmers and users.

THE MATHEMATICAL-HIERARCHY APPROACH

By contrast with the Prototype/Hacker approach, the fundamental idea of the mathematical-hierarchy approach is to structure an algebraic manipulation system (algorithms and data) along the lines of the mathematical algebraic hierarchy, which has developed in the last 50 years or so, under the name "Modern Algebra" (see, for example, Ref. 20). Examples of this approach include Axiom (need ref) Scratchpad II (Ref. 7), Mode-Reduce, Andante (Ref. 18), Newspeak/SCARAB (Ref. 8), Views (Ref. 1), and Capsules. With the exception of Scratchpad II, these systems have not been described extensively in the literature. Although they each have some unique characteristics, from our point of view in this paper they can generally be considered essentially similar.

We will discuss this approach by pointing first to successes, then to shortcomings in the next two sections.

Successes of Mathematical Hierarchy Systems

A major area of success in the mathematical hierarchy systems is in the clarification of issues which have, for the most part, been obscured by the prototype/hacker algebraic manipulation systems.

The advantages of a strongly-typed programming language framework are made available. These advantages: (a) more reliable programming as the result of carrying along redundant information that can be used for debugging, verification, etc.; (b) polymorphism or generic programming (assuming parametric types), so that one program can serve for several purposes; and (c) a consistent world view that can be reflected in documentation, the user's view, etc.

By contrast, in Macsyma there is nothing to prevent the user from specifying nonsensical arithmetic operations on an object, and obtaining results which (although deterministic) are unlikely to be meaningful or consistent. Macsyma can be commanded to simultaneously use algebraic numbers and finite field arithmetic, and is unable to keep domains clear. Thus, certain versions of Macsyma simplify $\sqrt{5}$ modulo 13 to -1, whereas there is, in fact, no square-root of 5 in Z_{13}. The transformation is roughly:

$$\sqrt{5} \bmod 13 = (5 \bmod 13)^{1/2 \mathrm{mod} 13} \bmod 13$$

and then

$$5^{-6} \bmod 13 = 15625^{-1} \bmod 13 = -1.$$

I have been advised that this is merely a "bug" fixed in later versions (Symbolics' Version 415.69 and later, 1986), but I suspect that if one "patches" systems as such anomalies are encountered, a system becomes more complex, slower and buggy.

Mathematical hierarchy systems also clarify the role of algebra in computation. For example, the Risch integration algorithm, even though it is apparently performing a calculation from analysis, is actually an algebraic algorithm. A systematic construction of a differential field and logarithmic extensions is followed by a solution of algebraic equations. Although this can be described algebraically (indeed it can be argued that otherwise it could not be programmed), it is probably more appropriate to describe its implementation in Macsyma as mashing around a collection of data objects by *ad hoc* simplification routines.

Probably the most obvious positive result of these new systems is their modularity of construction. The same discipline which requires a quotient field in order to permit division, also imposes a modularity whereby any of a number of quotient fields can be constructed and similar programs can be used for operations, regardless of the base structures. For example, polynomial operations over a ring will be defined once, and the algebraic "type" or "domain" required for the coefficients will be a parameter to the operations (e.g., polynomial multiplication). This kind of generic or polymorphic arithmetic is very attractive.

Our experience from supporting and extending Macsyma included the occasional request from a user that an existing facility (e.g., truncated power series) be used over a domain which had not been anticipated by the original programmer (e.g., square matrices as coefficients). Although some changes could be made by modest programmer effort in redefining some LISP macro-definitions, these could be not done by the typical user. Some other reasonable requests could be accomodated only by extensive system changes and, therefore, were not done at all. Our experience at Berkeley has been substantially more positive in that almost exactly the same transformation can be done at a rather modest alteration of existing code in SCARAB (Ref. 22).

Problems with Mathematical Hierarchy Systems

These comments are somewhat controversial and speculative. I hope they will prompt rebuttals and discussions.

The mathematically disallowed operations. In a strongly typed language, one forbids all operations not explicitly allowed. By constrast, in the hacker approach, any construction not forbidden is permitted.

Can we specify, using types, exactly all and only the allowed operations? I think there are substantial problems. Programmers who implement the "type" *algebraic field* know that they must allow for addition, multiplication and division. A (commutative) field is a ring in which there is always a unique solution for x, where $ax = b$ and a is non-zero. There is no consistent way of viewing any field element b/a as having a type while forbidding, in the type system itself, a division by zero. Thus one cannot describe division in a field F by saying it maps a pair of elements to a quotient. Does one define a new type, namely non-zero element of a field, to help out? This certainly is clumsy. Other dilemmas suggest that adjunction of an element ∞ would help, but this is just the first step into a morass. What is $\infty + \infty$? What is $0/0$?

Now that we have muddied the waters at the level of a field to accomodate a more accurate model of a useful computational structure, what can the algebraist build upon this? Can we invert a matrix of such pseudo-field elements including infinities? There are basically two solutions in such a system:

a. Disallowed operations raise an exception, as in conventional programming languages. This would be more acceptable if conventional programming languages solved the problem. In fact, some languages provide hardly more than a "core dump." The problem with this solution is that it drastically weakens the notion that we are building on a mathematical hierarchy system that hides the representation. Instead it thrusts the representation to the top level and gives up.

b. Operations are defined to return unions of types. This trick works reasonably well for matrix inverse: using Scratchpad II syntax, declare the signature: `inverse: SquareMatrix (n:PositiveInteger, F:Field)`$\rightarrow$ `Union($, "failed")`. Here, anyone using the result of inversion will have to check for the singular matrix result `"failed"`. Consider arithmetic, now. We could specify that `$/$` $\rightarrow$ `Union($,`∞`)`[1]. What this means is that the quotient operation will either return a member of the same field (denoted by `$`), or that the operation will return a special element, ∞. The difficulty with this approach is that it defers breaking everything by exactly one level. We have constructed something which is no longer a field, and cannot be used as one. Since we no longer have a field, much of the structure we have erected becomes unsteady; if we do not have a model for this *almost* field, then we must continue to explicitly write new procedures, one level at a time to deal with this structure. For example, polynomial arithmetic over the almost-field in question above will be different from polynomial arithmetic over a field. It also will have to be modified because such polynomials do not quite form a ring. This continues upward to rational functions, etc.

Can we adapt the solution used for matrix inversion to our almost-fields? It would be difficult for a programmer to check to see if the result of *each* arithmetic operation was a failure, especially when some of them require no special attention. The IEEE floating-point standard provides a model for what might be done with evaluation with ∞ in certain circumstances: $x := a + b/c$ where $c = \infty$, a, b finite is equivalent to $x := a$. This is very handy for evaluating continued fractions such as $x := a_1 + b_1/(a_2 + b_2/\ldots)$.

If the mathematical hierarchy systems cannot model this level of complexity, their advantages may not compensate sufficiently. The "hacker/prototype" models do not provide much of an alternative here, but they don't place a philosophical barrier between the system and the potential solution of introducing non-field elements into an algebraic structure that is otherwise a field.

We could retreat to a model which avoids division, but this would be a very lean algebraic manipulation system. Of course, directions of growth in other areas which are computationally interesting can be imagined: programs such as the Cayley system (Ref. 3) demonstrate some of these. Cayley does not, as a primitive, provide structures as complex as polynomials.

<u>The value-type dilemma.</u> Some of the problems are much more mundane, but face implementors who choose to use a mathematical hierarchy approach. We give some examples: consider a ring which is a set of elements with two operations, usually written in computing circles as + and *. A ring can be constructed of a set of elements viewed two different ways: each way as a monoid, but with different operations. This can lead to sticky linguistics: there are, for example, two different identities, one each for multiplication and addition.

This problem must be overcome by some programming technique (it has in Newspeak (Ref. 10)), but it remains a thorn.

[1] Actually, we would name a new domain which would be a field extended with "∞" and use it in this construction. Unfortunately, this domain is not a field anymore.

Another common problem is that of parameters in types. Consider a function I of one integer argument n which returns the size n square identity matrix over the integers. If this function is allowed at all, it returns a type which "includes" the value of n (we say that n is a *parameter*). Matrix addition, which ordinarily does not allow for addition of incompatible types, must check the types of its arguments, insuring, for example, that they have the same dimensions and compatible elements. But the value(s) of n may not be known at compile time; in fact, the difference between checking for type-compatibility and for special values (like division by 0) can become uncomfortably intertwined. Systems tend to become more *ad hoc* as these issues are faced.

The principal questions remain. Is there a foundation from modern algebra which suffices to include all of the operations which have been so convincingly programmed in the larger systems of today? Can we really build upon a solid foundation those algorithms that will make a system appealing for engineering computations? Can we clean up and adopt the heretofore muddy semantics of the hacker systems' impressive capabilities on this algebra, and make all the adjustments needed to make everything constructive?

Certainly, it would be pleasant if we could see unified and clear semantics. Could we anticipate new results such as the invention of the Risch integration algorithm proceeding from the establishment of foundations of such systems? It seem unlikely, since even the implementation and extension of the Risch algorithm has never been obvious, even from existing math-hierarchy systems of today.

The needs of the users. Another problem area which may have emerged (and perhaps was the death of Mode-Reduce (Ref. 12), is the level at which the user can communicate with the system. The programmer using a mathematics-hierarchy system is now dealing with a strongly-typed language, and the required coercion between types is either an enormous bother for the programmer (to handle explicitly) or a burden for the compiler. If it is left in the hands of the programmer (as it was in Mode-Reduce), there is a constant need to explicitly direct conversions between (say) the zero polynomial and the integer zero and the element in a finite field zero. This is annoying, especially when a programmer might observe that in the hacker system they might all be concretely represented by the same data structure (as is the case in Macsyma). If it is left in the care of a compiler, there is evidence from the SCARAB/Newspeak project here at Berkeley, and perhaps similar projects elsewhere, that the compiler and data structures must be fairly complex. There is hope that by careful language design and type inference (to the extent that it can be done at compile time at all), coercion can be performed with a linear-time unification algorithm. However, it appears to be, in general, not feasible.

To add to the discouragement, consider the case of a system which has finally been perfected as a mathematical hierarchy. There is no guarantee that its perfection can be maintained in the face of less-than-ideal demands placed on it. The realities that formed the prototype/hacker systems persist. These include ambiguous notations and the intrusion of real-world approximations, contexts and computational special cases into the world of algebra.

BEYOND ALGEBRA: WHAT DO WE NEED?

In a nutshell, it appears that we need to compute with mathematical functions and domains. We must be clear that we do not mean *computer procedural functions* and *data types*, but analytic functions (and non-analytic ones, too), and regions of real and complex n-dimensional space.

There are fewer solutions and more problems in this area than have been readily admitted in the past. This section discusses three approaches but none are effective solutions. We hope that by presenting them we can explicate some of the problems. (I am especially grateful for discussions with W. Kahan concerning the material of this section.)

Make Implicit Assumptions Explicit

One initially appealing way of dealing with functions and their singularities is the implicit extension of all expressions to some universal domain such as the complex plane. Most existing symbolic systems, without very much deliberation and without any explicit recognition of what this domain is or why transformations should be extended over it, in spite of branch cuts or singularities, nevertheless allow a number of transformations, which are valid only over parts of this domain.

Systems such as Macsyma contain such transformations as $\sqrt{z^2} \to z$. Their designers were relying on the consequences of the *Monodromy theorem*, a form of which we quote for simply connected regions:

> Let the region S be simply connected, let f be analytic at a point $z_0 \in S$, and let an analytic continuation exist along every path emanating from z_0 and lying in S. Then all these analytic continuations are continuations of one another and thus together define the analytic continuation of f into S. (see Ref. 11, p. 168, theorem 3.5c; also see Ref. 15, p. 105).

What this means, essentially, is that if at some point (in the complex plane) the transformation is correct, (say, at the point $z = 3$, $f(3) = \sqrt{3^2} - 3 = 0$), and if $f(z)$ is analytic (or analytically continuable) along a line that can be drawn from the point (3,0) to all points in some region of interest, then the transformation is correct in that region. The number of paths that must be tested in order to assure that all of the implicit "universal" domain is covered, depends on the connectedness of the complex plane subject to removal of certain points and lines which depend upon the branch cuts of the function f. In fact, in order to draw a line from $3 \to -3$, it is necessary to cross the Y-axis, where $f(z)$ is not regular, so the transformation does not (necessarily) hold at -3. However, our transformation based on $f(z) = 0$ holds for any point that can be reached by a continuous path from the point (3,0) without crossing the Y-axis.

Should a user of such a system perform a substitution or evaluation of -3 for z in the expression z formed by simplifying $\sqrt{z^2}$ under the transformation $f(z) = 0$, the system would have to object: the point $z = -3$ is not in the region of validity of the expression z. The system's view of all the functions involved, and their analytic continuations, does not allow for z in the left half plane.

Here is another example. Consider the transformation

$$\log(x - 1) + \log(x + 1) \to \log(x^2 - 1).$$

In the complex plane there is a branch cut of the logarithm for the locus of points for which the argument is non-positive. (This is by convention, and we would have to keep track of many such conventions for other functions.) Thus, for the first log, the cut runs from $x = -\infty\ (real)$ to $x = 1$; for the second log, from $x = -\infty\ (real)$ to $x = -1$; and for the "simplified" result, the locus of points for which $x^2 - 1 \leq 0$ is the branch cut: the line includes the whole imaginary axis plus the real axis between -1 and 1. If you draw all three sets of branch cuts, the complex plane is cut into three sections: the NW and SW quadrants, plus the East half plane, with a cut along the real axis up to 1.

What use can be made of this information? According to the Monodromy theorem, if you use the above transformation, in spite of its hazards, you need only check the transformation once in each of the three regions. If equivalence holds in each region, it holds throughout the complex plane, and you need not carry any restriction on domain along with the resulting expression.

With the exception of some tentative work by Adam Dingle (need ref) and Harlan Seymour (see Ref. 17) that was being conducted with the SCARAB algebra and analysis system at the University of California, Berkeley, no system we are aware of does any such checking, nor even makes an attempt to identify transformations for which this checking would be useful, nor even has a general means for defining branch cuts. The notion of analytic continuation on which this is based is a much neglected area of symbolic representation of mathematics (except, perhaps Ref. 11).

To conclude this section, we would like to issue a challenge to computer systems to solve the following problem, which presumably would fit into the class that would be solvable by the required programs. We have chosen an early problem from a complex variables text (Ref. 4, p. 24, exercise 13):

> Discuss the branch-cut and Riemann-surface situation for ...
>
> $$m(z) = \frac{1}{\sqrt{z^2+\alpha^2}} \ln \frac{z+\sqrt{z^2+\alpha^2}}{z-\sqrt{z^2+\alpha^2}}$$
>
> where α is a constant with Re $\alpha = k > 0$. In particular, verify that $m(z)$ can define a function which has no branch point in Im $z < k$, that it can also define a function which has no branch points in Im $z > -k$, and that one linear combination of the two functions so defined is $1/\sqrt{z^2+\alpha^2}$.

Computing Using Intersection of Domains

As an alternative to the approach of the previous section, it may seem that an appealing way of dealing more effectively with functions and their singularities is to consider, for purposes of algebraic manipulation, each function as a pair consisting of an expression (e.g., $(x+1)/(x-1)$) and a domain (e.g., $|x| > 1$).[2] Then, when combining two functions, one can assume that the combination is valid on the intersection of the two domains. The combination may be valid elsewhere by the Monodromy theorem, but this is hard and provably impossible to deduce in general.[3]. We might think that this cut-down requirement would be easier to deal with.

The next few paragraphs provide evidence of some problems that one must confront.

Consider the function h defined by $h(x) := \arctan x + \arctan(1/x)$, which is equal to $\pi/2$ for $x > 0$ and $-\pi/2$ for $x < 0$, (a step function). We might observe that for $x = 0$ and $x = \infty$, $h(x)$ is undefined. Thus, the domain we might choose for $h(x)$ is "x is finite and non-zero". Let $g(x) :=$ the derivative of $h(x)$. On the domain of definition of $h(x)$, we can see that $g(x) := 0$. (The domain of g is x finite and non-zero.) Now, we can construct a manifest constant (in particular, we have just constructed a kind of 0), so that the constant depends on one or more domains of invisible indeterminates in its representation, that is, $g(x) := \langle 0, (x \neq 0) \wedge (x \neq \infty)\rangle$. This pair-representation is starting to become complicated, even bizarre if the first part, the expression, is 0.

A more typical manifest constant would be the pair consisting of some simple integer, and a domain in which "any indeterminate can have any value". If we have a suitable shorthand for this situation, then we can deal naively, almost as we would in any other system, with integers, rational numbers and multivariate polynomials with "arbitrary" indeterminates. Yet, there are problems. For a starter, ratios of polynomials introduce holes in domains whenever denominators can be zero. Returning to one of our simple problems in the introduction (is an expression zero, and how much work are you willing to perform to deduce this fact): Are you willing to say that $(x^2-1)/(x+1)-(x-1)$ is $\langle 0, x \neq -1\rangle$?

How long a description of a domain can we get? If we return to our function $g(x)$ above, we can observe that even in a computational domain with only one indeterminate, domain descriptions can be arbitrarily long, even for an expression as simple as "0".

The Audit Trail

[2] In fact, we have written simple experimental systems which include considerably more information, including specification of dependencies, free vs. bound variables, names, etc. For our discussion here, the two items are all that is needed.

[3] This does not mean it is worthless to attempt to do this; all it means is that we will not always succeed. We may be successful almost all the time with a clever program.

A third, perhaps more practical alternative is a kind of "audit trail" algebraic computation system. The principle is that sufficient data is preserved to recompute (by the auditor) any result from first principles (e.g., input data). An auditor can, therefore, confirm the correctness of any calculation by repeating it.

This practice can be followed not only in bookkeeping, but in algebraic manipulation, given that associated with an answer (which could be symbolic or numeric), an auditor (or user) has the opportunity to question whether the results are, in fact, appropriate to a specific instance. Perhaps the simplest case to consider is a situation in which an algebraic manipulation system has deduced that some identity holds "in general" and the user wishes to test that deduction. For example, as we have previously mentioned, some algebraic manipulation systems will claim that

$$(x^a)^b \;=\; x^{ab}$$

and therefore

$$\sqrt{x^2} \;=\; (\sqrt{x})^2 \;=\; x.$$

Unfortunately, this fails to hold for $x \;=\; -3$. An auditor could check that.

Many other "obvious" truths which are embedded in conventional symbolic manipulation systems are, in general, false. $\log(\exp(x)) \;=\; x \;=\; \exp(\log(x))$ falls to pieces at 0 and many other places in the complex plane.

The basic way around this is to encode, in an answer, all possibly questionable transformations, until the user wishes to test that all hypotheses are satisfied. Thus, one would have to, as a minimum:

1. Reserve all fractional-power transformations (e.g.,implement square roots) for later evaluation at a test point;

2. Reserve all choices when principal-value considerations might apply. In particular, this may require the user to fix the meaning of non-rational "kernels" such as log, exp, inverse trigonometric functions, etc.

3. Preserve all non-constant common factors removed from numerators and denominators to check for zero-division.

Fortunately, some modern programming languages provide features for storing and computing with "suspensions", "closures", or "funargs" (Ref. 19). These are, in principle, at least, sufficient to provide an audit trail. It may be less expensive to save *computational paths* than *domain descriptions*. In case the audit "fails" one could get some specific indications of errors; for example, "at step k, a division by $x-1$ is invalid because you have subsequently asserted that $x = 1$."

CONCLUSIONS

Modern algebra and programming languages with strong typing are useful, but not panaceas in the realm of algebraic manipulation systems. It appears that we are neither sufficiently sophisticated in algebra, nor in programming languages to solve all the problems mentioned here.

While there are useful contributions that these ideas have made in the face of even simple notions from applied mathematics, current efforts lack sufficient modeling capabilities. In brief, representations for mathematics are tougher than one might be led to believe by the apparent partial successes of computer systems. To go much further, some additional break with tradition is necessary.

We suggest that approaches to analysis require computation with domains. This may be a fruitful area for study (Refs. 17 and 13). The methods indicated in this paper provide some particular directions.

ACKNOWLEDGMENTS

I thank David R. Barton, Paul Hilfinger, W. Kahan, Scott Morrison and C. J. Williamson for comments and suggestions. This work was supported in part by the following grants: National Science Foundation under grant number CCR-8812843 through the Center for Pure and Applied Mathematics, University of California at Berkeley; the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space and Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley; NSF grant CSD-8722788 (ERL); the IBM Corporation; a matching grant from the State of California MICRO Program.

References

[1] Abdali, S. K., Cherry, G. W. and Soiffer, N., "An Object-Oriented Approach to Algebra System Design," Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation, Char, B. W. (ed.), ACM, Waterloo, Ontario, July 1986, pp. 24-30.

[2] Buchberger, B., Collins, G. E., Loos, R., Albrecht, R. (eds.), Computer Algebra: Symbolic and Algebraic Computation, Springer-Verlag, 1982, 1983.

[3] Cannon, J., A Language for Group Theory (and Newsletters), Department of Pure Math, University of Sydney, Australia, Dec. 1982.

[4] Carrier, G. F., Krook, M. and Pearson, C. E., Functions of a Complex Variable: Theory and Technique, McGraw-Hill, 1966.

[5] Char, B. W., Geddes, K. O., Gonnet, G. H. and Watt, S. M., Maple Reference Manual, 4th ed. Symbolic Computation Group, University of Waterloo, Waterloo, Ontario, Canada, March 1985.

[6] Cole, C. A., Wolfram, S., et al., SMP a symbolic manipulation program, California Institute of Technology. Also see Ref. 23, July, 1981.

[7] Davenport, J. H., Gianni, P., Jenks, R. D., Miller, V. S., Morrison, S. C., Rothstein, M., Sundaresan, C. J., Sutor, R. S. and Trager, B.M., "New Scratchpad," Math. Sciences Dept., IBM T. J. Watson Research Center, Yorktown Heights, NJ.

[8] Fateman, R. J., et al., "Research in Algebraic Manipulation at the University of California at Berkeley," Proceedings of the 1984 Macsyma Users Conference, General Electric Research Lab., Schenectady, NY, July 1984, pp. 188-198.

[9] Fateman, R. J., "A Review of Macsyma," IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 1, March 1989, pp. 133-145.

[10] Foderaro, J. K., "The Design of a Language for Algebraic Computation Systems", Ph.D. dissertation, Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1983.

[11] Henrici, P., Applied and Computational Complex Analysis, Vol. 1, Wiley and Sons, 1974.

[12] Hearn, A. C., personal communication, 1984.

[13] Hilfinger, P. and Colella, P., FIDIL (Finite Difference Language) Reference Manual, Rev. 2.6, Internal Report, University of California at Berkeley, July, 1986.

[14] Jenks, R. D. and Trager, B. M., "A Primer: 11 Keys to New Scratchpad," Proceedings of Eurosam 84, Lectures Notes in Computer Science 174, Springer-Verlag, 1984, pp. 123-147.

[15] Knopp, K., Theory of Functions, Part I: Elements of the General Theory of Analytic Functions, Dover Publications, NY, 1945.

[16] The Mathlab Group, "Macsyma Reference Manual", Vols. 1 and 2, Version 10, Laboratory for Computer Science, Mass. Inst. of Tech., Cambridge, MA., Jan. 1983. [Also available from the National Energy Software Center (NESC), Argonne, IL. Similar manuals available from Symbolics, Inc. (for example, Version 11, Oct. 1985).]

[17] Seymour, H., "Conform: a Conformal Mapping System," Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation, Char, B. W. (ed.), ACM, Waterloo, Ontario, July 1986, pp. 163-168.

[18] Soiffer, N., "A Perplexed User's Guide to Andante", Unpublished Manuscript), University of California at Berkeley, 1983.

[19] Steele, G. L. Jr. Common LISP: The Language, Digital Press, 1984.

[20] The Soft Warehouse, "muSIMP/muMATH Reference Manual", P.O. Box 11174, Honolulu, HI 96828.

[21] van der Waerden, B. L., Algebra, 7th ed., F. Ungar Publishers, NY, 1970.

[22] Williamson, C. J., Jr. "Taylor Series Solutions of Explicit ODE's in a Strongly Typed Algebra System," ACM SIGSAM Bulletin, No. 1, Feb. 1984, pp. 25-29.

[23] Wolfram, S., et al., SMP Reference Manual, Computer Mathematics Group, Los Angeles, CA., 1983.

Reprinted by permission of ACM:
An essentially similar manuscript was published by the Association of Computing Machinery (ACM) under the title, "Advances and Trends in the Design and Construction of Algebraic Manipulation Systems" in the *Proceedings of ISSAC-90*, August 20-24, 1990, Tokyo, Japan.

Minor revisions have been made circa 1994; some of the problems expressed here have been solved in part by A. Dingle's thesis.