Algebraic Simplification: A Guide for the Perplexed

Joel Moses*
Project MAC, MIT, Cambridge, Massachusetts

Algebraic simplification is examined first from the point of view of a user who needs to comprehend a large expression, and second from the point of view of a designer who wants to construct a useful and efficient system. First we describe various techniques akin to substitution. These techniques can be used to decrease the size of an expression and make it more intelligible to a user. Then we delineate the spectrum of approaches to the design of automatic simplification capabilities in an algebraic manipulation system. Systems are divided into five types. Each type provides different facilities for the manipulation and simplification of expressions. Finally we discuss some of the theoretical results related to algebraic simplification. We describe several positive results about the existence of powerful simplification algorithms and the number-theoretic conjectures on which they rely. Results about the nonexistence of algorithms for certain classes of expressions are included.

Key Words and Phrases: algebraic manipulation, algebraic simplification, canonical simplification

CR Categories: 3.1, 3.2, 3.6, 4.9, 5.2, 5.9

1. Introduction

Simplification is the most pervasive process in algebraic manipulation. It is also the most controversial. Much of the controversy is due to the difference between the desires of a user and those of a system designer. The user wants expressions which he can comprehend. The designer wants expressions which can be manipulated with great ease and efficiency. Users tolerate, and in fact prefer, a certain amount of redundancy in an answer. For example, they usually desire to see expressions containing the twelve trigonometric and hyperbolic functions. Designers would prefer giving a user only sines and cosines or just exponentials with complex arguments.

There is one property of simplification about which both users and designers can agree. That is, that simplification changes only the form or representation of an expression, not its value. Thus an ideal, but not very helpful, way to describe simplification is that it is the process which transforms expressions into a form on which the remaining steps of a computation can be most efficiently performed.

The problem of representation for algebraic expressions is especially acute because there are so many equivalent ways to represent an expression. Frequently one of these equivalent forms is much more useful than another, and, just as frequently, it is a nontrivial problem to recognize the equivalence. For example, it

This is a revised version of the paper which appeared in the Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, March 23–25, 1971, pp. 282–304.

* Department of Electrical Engineering. Work reported herein was supported by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0001.

is rare that we do not want to recognize that an expression is equivalent to 0, but many of us have difficulty in recognizing the following identities:

$$(2^{\frac{1}{3}} + 4^{\frac{1}{3}})^3 - 6(2^{\frac{1}{3}} + 4^{\frac{1}{3}}) - 6 = 0$$

or

$$\log \tan (\tfrac{1}{2}x + \tfrac{1}{4}\pi) - \sinh^{-1} \tan x = 0.$$

Consider how much more difficult the problems become when we deal with expressions which are several pages long. Yet expressions of such size are quite common in algebraic manipulation! An additional difficulty is that the usual manipulatory algorithms can easily magnify a bad choice of representation. For example, the derivative of a product of n factors can be a sum of n terms each of n or more factors.

Another issue which arises in discussions of simplification is related to the local or global nature of the problem. If expression A is deemed simpler than its equivalent expression B in one context, then is A to be considered simpler than B in every context? A perfectly strict answer is no. For example, $x^7/(x^{12} + 1)$ is a more compact representation of the rational function it represents than $\frac{1}{4}(4x^3)x^4/[(x^4)^3 + 1]$. The former is usually easier to manipulate and comprehend. However, when integrating, the latter expression indicates a pattern which suggests the substitution $y = x^4$ which yields

$$\int \frac{\frac{1}{4}y}{y^3 + 1}\, dy,$$

a much simpler integration problem than that which is posed by the first expression. Designers would prefer a system in which the simplification steps are the same in every context. Users clearly would prefer a system which could take contextual information into account in deriving a simplified expression.

A related issue regarding simplification is the extent to which the concept can be formalized. The point that we made above is that the simplest form of an expression depends on one's goals or, in other words, on the context. One would be hard put to formalize the goals of all potential users. However, we can obtain theoretical results for simplification algorithms which have useful properties. One such property is that the algorithm simplifies to zero any expression equivalent to 0. A stronger property is that the simplifier reduces all equivalent expressions to a single (canonical) expression.

Historically, simplification was required in algebraic manipulation systems because the manipulatory algorithms produced sloppy results. For example, the unsimplified result of differentiating $ax + xe^{x^2}$ with respect to x is an expression such as

$$0 \cdot x + a \cdot 1 + 1 \cdot e^{x^2} + x \cdot e^{x^2} \cdot 2 \cdot x.$$

Simplifying the derivative above would yield an expression like $a + e^{x^2} + 2x^2e^{x^2}$.

With the ever growing use of algebraic manipulation, it has become increasingly apparent that simplification plays a much more complex role in the way one solves problems with an algebraic manipulation system. In the remainder of this paper we shall discuss some capabilities users wish to see in an algebraic manipulation system, and the variety of simplification facilities currently offered by such systems. The paper ends by discussing a number of formalizations of the concept of simplification and the algorithms corresponding to such formalizations.

2. Simplification for the Sake of Comprehension—the Needs of Users

One of the most common complaints of users of algebraic manipulation systems is that the expressions obtained as results of a calculation are incomprehensible and therefore essentially useless. In order to understand the importance of such a complaint we have to differentiate between two major classes of users. Some users are only interested in the value of a calculation. For example, those who use symbolic differentiation as a step in a numerical calculation do not care very much about the form of the symbolic derivative.[1] For such users the problem of simplification reduces to keeping the intermediate expressions in a calculation in a form which optimizes the use of space and time in the calculation.

The second class of users includes those who need to make "physical sense" of an expression. Perhaps such a user is studying a process and wants to learn about some property of the process by manipulating a mathematical model of it. For example, he might be interested in the manner in which the value of an expression varies as one of its variables increases in value.

It should be clear that a user is likely to comprehend and to answer questions about a small expression a lot better than about an equivalent but larger one. Thus a goal of simplification should be to produce small expressions. In fact, most of the usual simplification transformations such as collecting terms in sums (e.g. $x + 2x \rightarrow 3x$) produce smaller expressions. Moreover, transformations which produce larger expressions (e.g. expanding integral powers of sums $[(x + 1)^3 \rightarrow x^3 +$

[1] Such users should care a little about the form of the derivative because some forms of expressions yield a smaller round-off error in a numerical calculation than other forms.

[2] An exception to this rule is represented by programs that recognize the next number in a sequence [1]. Such programs would, in fact, recognize the pattern in the former expression.

$3x^2 + 3x + 1)]$ are controversial. Many systems will employ such transformations only if the user specifically demands them.

Of course the prevalence in algebraic manipulation systems of simplification transformations which produce smaller expressions is due mostly to the fact that small expressions are usually easier to manipulate than larger ones. This is an instance where the needs of simplification for the sake of improved comprehension and simplification for the sake of efficient manipulation coincide.

The requirements of simplification for the sake of comprehension are, however, more subtle than we have just indicated. It is not so much the small size of the expression which aids in comprehension, as the small size of a description of the expression. For example,

$$1 + 2x + 3x^2 + 4x^3 + \cdots + 11x^{10}$$

is less complex for many purposes than

$$1 + 3x + 4x^2 + x^3 - 9x^4 + 5x^5 + x^6 + 2x^7$$

because one can supply a small description of the former, [i.e. $\sum_{i=0}^{10} (i + 1)x^i$], but not of the latter. The way one usually obtains a small description is by recognizing a repeated pattern in an expression. Unfortunately, computer programs nowadays are not as good as humans at recognizing useful patterns.[2]

A major reason for computers not being as good as human users in simplifying an expression is that they lack knowledge of the context in which the expression was derived. To a physicist subexpressions like mc^2 contain a good deal of information not apparent to an algebraic manipulation system. For example, a physicist might be tempted to substitute E for mc^2 in order to reduce the size of the expression without destroying its information content to him. In fact, the major technique for simplifying large expressions is the substitution of small expressions for large subexpressions which either occur frequently or possess some meaning. We shall examine this technique in Section 2.1.

2.1 Substitution as an Aid to Comprehension

Many symbolic calculations take the following form: One starts with some equations such as

$$y = g(x),$$
$$z = h(x),$$
$$f = x^2 + y^2 + z^2,$$

and expressions such as

$$E: \sum_{i=0}^{5} c_i x^i.$$

Later one substitutes such equations and expressions into another expression such as $[(\partial f/\partial x)^2 + 2E^2]/f^3$. Then one attempts to simplify the resulting expression. In this section we are interested in the process of making intelligible large expressions such as the one which would result if we performed the substitutions and carried out the derivatives and expansions in the expression above.

Fig. 1

$$\begin{vmatrix} A & B & M & N & R & S & H & I \\ C & D & O & P & T & U & J & K \\ M & N & A & B & H & I & R & S \\ O & P & C & D & J & K & T & U \\ R & S & H & I & A & B & M & N \\ T & U & J & K & C & D & O & P \\ H & I & R & S & M & N & A & B \\ J & K & T & U & O & P & C & D \end{vmatrix}$$

Let us call the array $\left|\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right|$ a, the array $\left|\begin{smallmatrix} M & N \\ O & P \end{smallmatrix}\right|$ m, the array $\left|\begin{smallmatrix} R & S \\ T & U \end{smallmatrix}\right|$ r, and the array $\left|\begin{smallmatrix} H & I \\ J & K \end{smallmatrix}\right|$ h. Let us call the array $\left|\begin{smallmatrix} a & m \\ m & a \end{smallmatrix}\right|$ w, and the array $\left|\begin{smallmatrix} r & h \\ h & r \end{smallmatrix}\right|$ x. Then the entire array is simply $\left|\begin{smallmatrix} w & x \\ x & w \end{smallmatrix}\right|$. While the original structure consisted of 64 symbols, it requires only 35 to write down its description:

$$S = \left|\begin{smallmatrix} w & x \\ x & w \end{smallmatrix}\right|$$

$$w = \left|\begin{smallmatrix} a & m \\ m & a \end{smallmatrix}\right| \qquad x = \left|\begin{smallmatrix} r & h \\ h & r \end{smallmatrix}\right|$$

$$a = \left|\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right| \quad m = \left|\begin{smallmatrix} M & N \\ O & P \end{smallmatrix}\right| \quad r = \left|\begin{smallmatrix} R & S \\ T & U \end{smallmatrix}\right| \quad h = \left|\begin{smallmatrix} H & I \\ J & K \end{smallmatrix}\right|$$

Frequently the process of simplifying large expressions involves a reversal of the process which led to the expression above. That is, one substitutes small expressions (usually literals) for large subexpressions which occur more than once in the expression. The literals being substituted into the expression act as names or labels for the expressions that they replace. This is the role of f, y, and z in the expression above.

An artificial example which points out the value of substitution to the comprehension of an expression occurs in [31]. The example shows how to obtain a compact description of the matrix in Figure 1. We obtain a hierarchical description by recognizing patterns in the matrix and patterns in the matrix of literals that we substituted, etc. We finally reduce the 64 characters in the original matrix to 35 characters in the final matrix and all the associated equations. However, the hierarchical description seems to make the simplified result much clearer than is implied by the ratio 35/64.

The process of finding good subexpressions to replace usually involves some trial and error. It is useful to replace subexpressions which have some meaning in the context of the problem. In such cases we need not require that the subexpression being replaced occur more than once in the expression. Beyond such generalities, there does not seem to be much one can say at present which is frequently useful in the "massaging" process for large expressions. We should note that one often combines substitution with other manipulations (e.g. carrying out expansions or differentiations which have been delayed).

A technical problem arises when one makes substitutions for expressions other than atomic ones. Consider the problem of substituting a for xy^2 in the expression x^2y^3. Some possible results of this substitution are

(1) x^2y^3, (2) axy, and (3) a^2/y. One cannot say that there is a "correct" answer, because what is appropriate in one context need not be appropriate in another. Yet, no system, until recently, gave the user much choice in the result of substitution. The REDUCE system of Hearn [16, 17] has a good deal of machinery for making substitutions, but it does not give the user much control over the effects of its substitutions. Fateman [12, 20] has recently arrived at the following analysis of the problem. For simplicity, we shall make the analysis for polynomials, but it can be easily extended to more complex expressions.

Let us suppose we are trying to substitute A for B in C. We shall consider C to be represented as

$$\sum_{i=0}^{n} \alpha_i B^i.$$

Thus the substitution will yield $\sum_{i=0}^{n} \alpha_i A^i$. This representation of C is nonunique. We can make the representation precise by imposing constraints on the coefficients c_i. Let us assume that the variables in B are ranked in some way. Fateman's substitution programs usually provide that the degree of the main variable of B is lower in each α_i than in B itself. In addition, one can restrict the coefficients (1) to not contain a sum, (2) to be polynomials (and not rational), and (3) to have lower degree in all of the variables of B than the degree of those variables in B.

By varying these and other conditions, and by modifying the ranking of the variables, one can get a variety of results. One can then choose that result which seems most useful in the computation.

Some examples of substitutions made with Fateman's routines are given in Figure 2. The ability of Fateman's routines to obtain the results in the last four examples is due to the technique of continually dividing C by B. The last two examples indicate how this substitution mechanism provides for the application of the oft-discussed transformation $\sin^2(x) + \cos^2(x) \rightarrow 1$.

3. Simplification for the Sake of Manipulation—What Designers Provide

3.1 The Politics of Simplification

Simplification is such a central issue in algebraic manipulation that when a designer has decided how he will represent expressions, what changes of representation his system will perform automatically, which of these automatic transformations he will let the user override and modify, and what additional facilities for simplifying expressions his system will have, there are few major decisions remaining. As a result, one can classify algebraic manipulation systems by their approaches to simplification.

Four years ago, when we last surveyed the scene [23], we classified algebraic manipulation systems into three categories: conservatives, liberals, and radicals. In the meantime, there has been a slight change in the characteristics of some systems, and the characteristics of other systems have stabilized sufficiently so that we now claim the entry of two new parties, namely, the new left and the catholics.

The classification that we make of systems is based on a single criterion—the degree to which a system insists on making a change in the representation of an expression as provided by a user. A system which insists on radically altering the form of an expression in order to get it into its internal form is called a *radical* one in our scheme. A system which is so unwilling to make an inappropriate transformation that it essentially forces a user to program his own simplification rules is called a *conservative* system. A system which will make certain transformations automatically, but will leave others to the discretion of a user, is called a *liberal* system. The *new left* is mainly composed of variations of old radical systems which give certain additional choices to a user. Designers of *catholic* systems see the merit of each of the other approaches for some contexts. They design systems which offer several subsystems using different simplification techniques, and let the user switch among them as he pleases.

3.1.1 The Radicals. Radical systems can handle a single, well-defined class of expressions (e.g. polynomials, rational functions, truncated power series, truncated Poisson series). The expressions in this class are represented in a canonical form. That is, any two equivalent expressions in the class are represented identically, internally. This means that the system stands ready to make a major change in the representation of an expression as written by a user in order to get that expression into the internal canonical form. The advantage of this approach is that the task of the manipulatory algorithms is well defined and lends itself to efficient implementation. In fact, most of the major advances in algorithm design in the field of algebraic manipulation such as in the greatest common divisor algorithm, polynomial factorization, and integration have assumed expressions represented in canonical form.

Radical systems do not appear to have specialized simplification machinery since the process of generating expressions in canonical form, which is automatically

Fig. 2

Substitution of A for B in C

A	B	C	Alternative results
a	xy^2	x^2y^3	axy, a^2/y, x^2y^3
a	$2x + 3y$	$3x + 4y + 1$	$3x + 4y + 1$, $\frac{1}{3}x + \frac{4}{3}a + 1$, $-\frac{1}{2}y + \frac{3}{2}a + 1$
a	$x + y$	$bx + by + 1$	$ab + 1$
-1	i^2	$i^4 + 1$	2
1	$s^2 + c^2$	$s^4 + 2s^2c^2 + c^4$	1
1	$s^2 + c^2$	$(s^3 - s)c^{-2}$	$-s$

employed by the manipulatory algorithms (e.g. addition, multiplication, differentiation), is akin to simplification. An expression written in its canonical form is considered simplified, once and for all time. Any attempt to allow the user to modify the representation of an expression for his problem will likely cause a decrease in the efficiency of the manipulatory algorithms and is therefore eschewed or highly discouraged by designers of radical systems.

Excellent examples of radical systems are polynomial manipulation systems. One canonical representation of polynomials is the recursive representation used in Collins' PM and SAC-I systems [6, 7]. One assumes a ranking of the variables such as $x > y > z$. The polynomial is considered as a polynomial in the major variable with coefficients which are polynomials in the other variables and which are themselves represented in this recursive form. Thus

$$3x^2y^2 - 2x^2yz^3 + 5x^2z^2 + 4x - 6y^3z + y^3 + 3y^2 + z^4 + 1$$

would be represented as

$$(3y^2 - (2z^3)y + 5z^2)x^2 + (4)x + ((-6z + 1)y^3 + 3y^2 + z^4 + 1).$$

The other major representation of polynomials, popularized in the ALPAK system of Brown [2], is the expanded representation. The first polynomial is written in expanded form.

Situations in which there is widespread disagreement with the radical approach usually concern expressions which contain powers of sums. The radical systems would automatically expand such expressions in order to put them into the canonical form. Other designers would complain that $(x + 1)^{1000}$ should almost never be expanded. For example, the integral of $(x + 1)^{1000}$ with respect to x is trivially found if the integrand is not expanded. However, computing the integral of the expanded expression requires more time and space, and the final result appears atrocious to the human eye unless the pattern is recognized.

3.1.2 The New Left. The new left arose in response to some of the difficulties experienced with radical systems such as those caused by the automatic expansion of expressions. A new left system is usually a rational function system which does not necessarily expand products or integer powers of sums. A new left system will have all the usual machinery of a radical system, but the algorithms will be generalized to handle unexpanded expressions. The new left thus sacrifices canonicalness and some of the well-definedness of the manipulatory algorithms for the ability to solve some problems more efficiently than is possible in a radical system. The user of a new left system can specify when expansion will take place, a facility which is, of course, not present in a radical system.

Systems which allow unexpanded terms in an expression are Hearn's latest version of REDUCE [18], and the latest version of ALTRAN [15].

A new left system can usually handle a wide variety of expressions with greater ease, though with less power, than a radical system using a canonical form. The idea is to use labels for nonrational expressions. Thus $xe^x + x^2 \sin x$ might be rewritten as $xy + x^2z$, where $y = e^x$, and $z = \sin x$. The expression $(e^{2x} + e^x)/e^x$ would probably be expressed as

$$(y + z)/z, \qquad y = e^{2x}, \qquad z = e^x,$$

since no attempt probably would be made to write the expression in canonical form.

3.1.3 The Liberals. Liberal systems rely on a very general representation of expressions and use simplification transformations which are close in spirit to the ones used in paper-and-pencil calculations. Liberal simplifiers perform the usual simplifications of collecting terms in sums and exponents in products, applying the rules regarding 0 and 1, and removing redundant operators (e.g. $a + (b+c) \rightarrow a + b + c$). Frequently such systems will also know simplification rules for certain arguments of nonrational functions. Thus $\sin 2\pi$ might simplify to 0, $e^{2\log y + x}$ might simplify to y^2e^x, and $\cos(\arcsin x)$ to $(1-x^2)^{\frac{1}{2}}$.

Liberal systems differ from radical and new left systems in several important ways.

(1) Expansions are carried out only if the user so demands (new left systems, of course, offer this feature also).

(2) Sums of quotients are never put over a common denominator unless the user forces such a transformation, but even if they are, the greatest common divisor cancellations are likely to be missed.

(3) Expressions can usually be represented in "unsimplified" form. That is, $1 \cdot \sin(x) + 0 \cdot \cos(x)$ can be represented in such systems. This allows patterns to be represented. Most manipulatory algorithms will, however, require that all their arguments be simplified, thus destroying the patterns.

(4) Nonrational terms can be expressed with great ease. Terms such as e^x, $x!$, and $\sum_{i=0}^{n} c_i x^i$ would be explicitly present in the expression, and would not be replaced by a label whenever they occurred.

(5) The representation is local in the sense that a term $\sin(x)$ appearing in one part of the expression can be modified without affecting a $\sin(x)$ appearing in another part of the expression.

The major disadvantage a liberal system has relative to a radical or new left system is its inefficiency. The representation of information in a liberal system might require two or three times as much space as in a radical system, and manipulations can be a factor of ten slower (of course such figures might increase or decrease depending on the situation).

The advantage claimed for liberal systems is that one can express problems more naturally in them than in radical or new left systems. Examples of liberal systems are FORMAC [33], and most LISP-based algebraic simplification programs such as Korsvold's [19].

3.1.4 The Conservatives. Designers of conservative systems claim that one cannot design simplification rules which will be best for all occasions. Therefore conservative systems provide little automatic simplification capabilities. Rather, they provide machinery whereby a user can build his own simplifier and change it when necessary. A simplifier written in such a way is far slower than a liberal simplifier, and this fact presents a distinct disadvantage for conservative systems. In fact, one can point to only two major conservative systems, Fenichel's FAMOUS [13] and FORMULA ALGOL [27].

The importance of conservative systems lies in the philosophy they represent, which is most clearly given by Fenichel [13], and in the technique which they champion of using rules and advice to describe simplification transformations. Their philosophy presents an indictment of all the other systems which perform many simplification transformations automatically, without seriously considering the context. Designers of conservative systems emphasize that the simplified form of an expression is determined by context. They will point to situations where even the most obvious transformations $0 \cdot x \rightarrow 0$ and $1 \cdot x \rightarrow x$ will destroy useful information as in the expression

$$0 \cdot \sin x + 1 \cdot \cos x + 2 \cdot \tan x + 3 \cdot \cot x + 4 \cdot \sec x + 5 \cdot \csc x.$$

Therefore, they claim that one must be able to tune the system to the particular nature of the problem. The preferred technique of "tuning" is based on the theoretical concept of Markov algorithms. In a Markov algorithm one is given an ordered set of rules to apply to an expression. Each rule has the form:

Pattern $\rightarrow$ Replacement.

For example, one such rule applied to algebraic expressions might be $A \cdot X + B \cdot X \rightarrow (A + B) \cdot X$.

To make such a rule correspond to the usual notion of "collecting like terms," one might want to restrict A and B to be numbers, while X could represent any product of factors other than numbers. The rule just given does not necessarily yield a simplified result in cases such as $2 \cdot X + (-1) \cdot X \rightarrow 1 \cdot X$. One generally applies a whole set of rules to a given expression, and when there is no rule which is applicable then the algorithm is complete.

Such rules are most appropriate in indicating local transformations on an expression.[3] One would not wish to write a factoring program as a Markov algorithm. Conservative systems have tended to model liberal systems rather than radical ones, since the latter specialize in global transformations on expressions.

Several designers have added to their systems a capacity for writing Markov algorithms, thus allowing their systems to take on various degrees of conservatism. The main use of rules in such systems has been to add new simplification transformations [e.g. cos $n\pi \rightarrow (-1)^n$], rather than to override old transformations. Thus a user of REDUCE can define the simplification rules relating to general exponentiation (e.g. $x^y \cdot x^z \rightarrow x^{y+z}$), although he cannot override $x^0 \rightarrow 1$. Korsvold's simplifier and MACSYMA's pattern matching subsystem [11] also allow one to define simplification rules. The latter allows one to override many of the built-in rules. It also provides for the compilation of new rules which should yield a relatively efficient simplifier.

3.1.5 The Catholics. Catholic systems use more than one representation for expressions, and have more than one approach to simplification. The basic idea underlying catholicism is that if one technique does not work, another might, and the user should be able to switch from one representation and its related simplification facilities to another with ease. A catholic system might use a liberal simplifier for most calculations and have a radical subsystem in reserve for performing special calculations such as combining quotients, solving linear equations with rational coefficients, and factorization. MATHLAB [9] is best described as a catholic system. The MACSYMA system [20] goes further in that it allows the user to manipulate entirely with a radical rational function subsystem, as well as with a liberal-radical combination as just described. In addition, MACSYMA, as pointed out in 3.1.4, has a rule-defining facility which allows it to closely approximate a conservative system. The SCRATCHPAD system [14] is a conglomerate made up of several LISP-based systems. It has a total of four simplifiers.

The designers of catholic systems emphasize the ability to solve a wide range of problems. They would like to give a user the ease of working with a liberal system, the efficiency and power of a radical system, and the attention to context of a conservative system. The disadvantage of a catholic organization is its size. A catholic system is necessarily larger than any other type of system. The variety of the services provided by the system may force users to learn a larger number of conventions than in other systems. A catholic designer may also impose a number of system-wide conventions (e.g. on the data representation) which would not be present in a smaller system. Such conventions might slow down all of the component systems.

3.2 Intermediate Expression Swell

Users of numerical analysis programs have learned to anticipate problems due to round-off errors. Users of symbolic manipulation programs have encountered

[3] The author begs for forgiveness of the reader for not defining "local." That concept tends to be as context dependent as the concept of simplification. However, see [22].

a corresponding problem in the tremendous growth of intermediate expressions in some calculations. Such growth has caused many calculations to be aborted because the expressions filled the available computer memory. Tobey has described this phenomenon with the colorful phrase "intermediate expression swell" [32]. In many cases the final result of a symbolic calculation is quite small, but in order to get that result one generates very large intermediate expressions. For example, the eigenvalue of a matrix with polynomial entries can be as simple as a single number. However, in order to obtain that number, one is forced to factor a polynomial having polynomials as coefficients. These coefficients might be obtained from the determinant of the matrix, which can be several pages long.

In some problems one can apply one's knowledge of subsequent steps in a calculation in order to keep expressions in a form which will maximize utilization of space and time. At the heart of Collins' improvement to the Euclidean GCD algorithm [8] was the idea that one could predict how certain terms were automatically introduced into the intermediate expressions, and therefore these terms could be canceled without affecting the final result. Before the appearance of this algorithm, it was thought that the size of the coefficients in the intermediate steps of the algorithm had to grow exponentially. Collins showed that they need only grow linearly!

Of course, results such as Collins' would not be expected from the average user; however, improvements of a similar nature can be made in many applications of algebraic manipulation. For example, consider $y = \sum_{i=1}^{n} x^i$, which is an approximation for $x/(1 - x)$. Suppose you wanted $\sum_{j=0}^{n} y^j$. The straightforward application of expansion in the latter sum would yield a polynomial of degree n^2. However, since y is only accurate to degree n, all powers of x greater than n are worthless. What is called for is a truncation in the expansion of powers greater than n. Systems which allow the user to specify truncation (e.g. by declaring $x^m = 0$ for $m > n$) can probably save factors of 100 or 1000 in speed for $n = 20$ [10].

3.3 Canonical Simplifiers and Theoretical Results

In this section we shall discuss theoretical results related to algebraic simplification. Almost all of the algorithms we shall describe are incomplete in the sense that they depend on as yet unproved conjectures about expressions involving constants. For example, the conjecture by Brown [3] has, as a special case, the statement that $e + \pi$ is not a rational number. That statement is almost certainly true, but no proof of it exists, and certainly none exists of the full conjecture. Even if the conjectures were false, the average user will probably never obtain incorrect results from these algorithms.

All of the results deal with well-defined classes of expressions which are extensions of polynomials or rational functions. Some deal with exponentials, others with both exponentials and logarithms, and still others with roots of polynomials. We shall also discuss a negative result, due to Richardson, which says that when one deals with expressions involving the sine and absolute value functions, then one cannot, in general, tell whether such expressions are equivalent to zero.

The simplification algorithms fall into three categories: zero-equivalence, canonical, and regular. *Zero-equivalence* algorithms can determine whether an expression in a given class is equivalent to 0. Such algorithms need not simplify a nonzero expression in any way. *Canonical* simplification algorithms transform all equivalent expressions in a given class into the same (canonical) form. Canonical simplifiers are zero-equivalence algorithms. It is an elementary but surprising fact that if a class of expressions E possesses a zero-equivalence algorithm it also possesses a canonical simplification algorithm. We assume that there exists an algorithm which generates a sequence of members of E in which any given member of E can be found in a finite number of steps. In order to obtain the canonical form for an expression f in E, we generate members of E until one is shown to be equivalent to f by the zero-equivalence algorithm. This argument points out a weakness in the definition of canonical forms—the canonical form need not be a simpler expression than the original expression.

Regular simplification algorithms arise when one deals with transcendental functions (e.g. exp, log). A regular algorithm guarantees that all nonrational terms in the simplified expression are algebraically independent. A set of expressions is algebraically independent (over the rational numbers) if in its elements there exists no nontrivial polynomial with rational number coefficients which is equivalent to 0. Regular algorithms are zero-equivalence algorithms, but need not be canonical ones. Likewise, canonical algorithms need not be regular.

3.3.1 Simplification Algorithms for Expressions with Nested Exponentials. In [3] Brown describes a regular simplification algorithm for a class of expressions he calls Rational Exponential (REX) expressions. These REX expressions are obtained recursively from the rational numbers, i, and π, and the variables $x_1, x_2, \cdots, x_n$, by the rational operations of addition, subtraction, multiplication, and division, and by forming exponentials of existing REX. Thus the expression

$$e^{[e^x/(e^x+1)]}/(e^{5x} + 3e^{2x} + xe^{4e1+1})$$

is a REX expression if we agree to write x for x_1 when only one variable occurs. Brown's algorithm makes use of the technique frequently mentioned in this paper of substituting labels for subexpressions (in this case, exponentials) in order to reduce a REX expression to a rational expression in the variables and the labels. The major simplification work in the algorithm occurs when the resulting rational expression is transformed into a canonical form. We shall see, however, that Brown's algorithm is not canonical. It should be noted that since the constants i and π are included, the REX expres-

sions contain the trigonometric and hyperbolic functions in exponential form.

In generating labels for the algorithm one must pay great attention not to allow algebraically dependent exponentials to be assigned to different labels. For example, e^x and e^{2x} are algebraically dependent since $(e^x)^2 - e^{2x} = 0$. Likewise, e^x, e^{x^2}, and e^{x+x^2} taken together are algebraically dependent. The labeling scheme must be such that if we assign y to e^x, then e^{2x} is assigned y^2.

The algorithm proceeds by replacing innermost exponentials in the expression by labels, if such exponentials are not algebraically dependent on previously replaced exponentials. The algebraic dependency is determined with the help of the conjecture by testing whether the argument (of the exponential function) being examined is linearly dependent on previous arguments. The following is a simple example of the procedure, and incidentally shows its simplifying power.

Suppose we are given the REX expression $(e^x + x)/(e^{2x} + 2xe^x + x^2)$. Traversing the numerator from left to right, we first encounter e^x. Let $q_1 = x$ and $r_1 = e^{q_1} = e^x$. Thus our first label is r_1. By substituting it into the expression we obtain $(r_1 + x)/(e^{2x} + 2xr_1 + x^2)$.

By treating e^{2x} as an independent variable in the expression above, we can try for a simplification by determining the greatest common divisor of both numerator and denominator.

That attempt is unsuccessful in reducing the expression and we continue generating labels. We next encounter the exponential e^{2x}. Let $q_2 = 2x$, $r_2 = e^{q_2} = e^{2x}$. Now check to see if a linear dependence exists between q_1 and q_2 (and also with $i\pi$, it turns out). Such a relation does exist, since $2q_1 - q_2 = 0$. Therefore, redefine $r_2 = r_1^2$ and by substitution obtain $(r_1 + x)/(r_1^2 + 2xr_1 + x^2)$. Simplifying this as a rational function reduces it to $1/(r_1 + x)$.

Since no more exponentials are to be found, replace the labels by the exponentials. The result is $1/(e^x + x)$, which is indeed simpler than the expression we had originally.

Brown's conjecture is that if $\{q_1, q_2, \cdots, q_k, i\pi\}$ is linearly independent over the rational numbers, $\{e^{q_1}, e^{q_2}, \cdots, e^{q_k}, x_1, x_2, \cdots, x_n, \pi\}$ is algebraically independent over the rational numbers. Using the conjecture, Brown can easily prove that the only simplified REX expression equivalent to 0 is 0 itself. Note that since 1 and $i\pi$ are linearly independent, the conjecture states that e^1 and π are algebraically independent, a statement which is stronger than the currently unproved statement "$e + \pi$ is not a rational number."

An important aspect of the algorithm is the retracing of steps one must go through in some cases. Consider $(e^{2x} + e^x)/e^x$. Let $q_1 = 2x$, $r_1 = e^{q_1} = e^{2x}$. Now $q_2 = x$, $r_2 = e^x$, and $q_2 = \frac{1}{2}q_1$. We cannot let $r_2 = r_1^{\frac{1}{2}}$ since we want to obtain rational results. So we redefine r_1 as r_2^2 and obtain $(r_2^2 + r_2)/r_2 = r_2 + 1 = e^x + 1$.

Brown's algorithm is not canonical because the algorithm does not make an optimal choice for labels. Consider e^{x+x^2}/e^x. Let $q_1 = x + x^2$, $r_1 = e^{x+x^2}$, $q_2 = x$, $r_2 = e^x$. Note that $\{q_1, q_2, i\pi\}$ is linearly independent over the rational numbers. Thus $\{r_1, r_2, x, \pi\}$ is algebraically independent by the conjecture. Furthermore, r_1/r_2 is simplified as a rational expression. Hence, the simplified result is e^{x+x^2}/e^x, which differs from the equivalent expression e^{x^2} which is also simplified. So the algorithm is not canonical.

Brown's algorithm produces different results when the expressions are reordered. In

$(e^{1/(x+x^2)} + e^{1/x}e^{-1/(x+1)})/e^{1/x}$ we can get $2e^{1/(x+x^2)}/e^{1/x}$,

using one order of assigning labels, and $2e^{-1/(x+1)}$ using a different order.

The last two examples are intended to show the difficulties that a canonical and regular simplifier for REX expressions has to surmount. In 3.3 we pointed out that the existence of a regular and thus zero-equivalence simplifier implies that a canonical algorithm for REX expressions exists. The scheme proposed there for the canonical simplifier is utterly inefficient. It also suffers from the fact that the simplified expressions are not described by some simple pattern. For example, the simplified form of 1 might be quite different from "1" in this scheme. The next algorithm produces expressions which do satisfy general patterns. Such simplifiers are exceedingly useful since they can help us determine answers to global questions about an expression (e.g. Is it a constant? Is it linear in x?).

In [5], Caviness describes a canonical simplification algorithm for a class of expressions related to REX expressions. His expressions admit only one real variable, say x, but no π, and no division at all. Because division is not allowed, Caviness' expressions are exponential polynomials. By using a conjecture similar to Brown's, Caviness shows how exponential polynomials (other than pure polynomials) can be transformed into the form $P_1(x)e^{S_1} + P_2(x)e^{S_2} + \cdots + P_k(x)e^{S_k}$, where the S_i are distinct exponential polynomials which are also in this form, and the P_i are nonzero, canonically ordered polynomials.

In [24] we describe a canonical and regular simplifier for first order exponential expressions (i.e. no nesting of exponentials) which are REX expressions but do not involve i or π. The proof of the regularity of our simplification algorithm also depends on a conjecture which is very similar to Brown's and Caviness' conjectures.

The novel idea in our algorithm is to use a partial fraction decomposition of the exponents. The left-hand side of the equation below is in the usual canonical form for rational functions and the right-hand side represents a partial fraction expansion of it.

$$(x^5 - x^3 + 1)/(x^4 - x^2) = x + (-1/x^2) + [-\tfrac{1}{2}/(x + 1)] + \tfrac{1}{2}/(x - 1).$$

We require that the terms of the partial fraction decomposition be linearly independent (over the rational numbers) of each other. Such partial fraction decompositions lead to yet another canonical representation of rational functions. The simplification algorithm breaks up an exponential of a sum into a product of exponentials which are replaced by labels in a manner similar to that of Brown's algorithm.

Thus, e^{x^2+x}/e^x is decomposed into $e^{x^2}e^x/e^x$. With proper relabeling and simplification of the resulting rational expression we obtained the simplified result e^{x^2}.

3.3.2 Expressions Involving Exponentials and Logarithms. The functions of the calculus include logarithms as well as rational functions and exponentials. Therefore, there is much interest in results admitting the logarithm function. A zero-equivalence algorithm was obtained by Richardson [28, 30] for a class of expressions which differs from the REX expressions in that it involves no i, only a single variable x, but allows the function $\log | x |$. The logarithm function in addition to the exponential function of REX expressions can be nested to any depth.

His algorithm for determining the equivalence involves a reduction process in which one asks whether progressively less complex expressions are equivalent to 0. The algorithm is incomplete in that it relies, in some cases, on knowing whether a reduced expression composed entirely of constants is equal to 0. The requirement of the solution to this so-called "constant problem" is similar to the need for conjectures in the algorithms of 3.3.1. Richardson's algorithm is, furthermore, only applicable when the expression being examined is totally defined everywhere in the interval in which zero-equivalence is to be determined. In essence, this requirement implies that no subexpression is unbounded in value at some finite point on the interval in question.

Richardson's measure of the complexity of an expression is very lexicographic in nature and relies on very little knowledge of the algebraic properties of the functions involved. For example, e^{e^x} is considered more complex than e^x because of the greater depth of nesting of the exponential function, and $(e^x)^2$ is more complex than e^x because of its higher degree. The complexity measure does not presume that e^{2x} and $(e^x)^2$ are algebraically related. In fact, it does not matter very much which of these two expressions is considered more complex as long as the ranking is used consistently.

The reduction procedure of the algorithm assumes that the equivalence problem for rational functions is trivial. A more complex expression will force the algorithm to generate subproblems which will either end up as rational functions or constant problems.

Let us suppose that we wish to determine whether an expression E is equivalent to 0. Let y be the most complex exponential or logarithmic term in E. Let us further suppose that y is a logarithmic term. By multiplying out denominators, expanding products of sums, and collecting like terms, we can get a polynomial expression E^* in y of the form

$$E^*: a_n(x)y^n + a_{n-1}(x)y^{n-1} + \cdots + a_0(x),$$

which is equivalent to 0 if and only if the original expression E is equivalent to 0. Since $a_n(x)$ does not contain y, it is less complex than E or E^* and we can apply the algorithm recursively in order to determine if it is equivalent to 0. If a_n is equivalent to 0 then since the expression $E1$,

$$E1: a_{n-1}(x)y^{n-1} + \cdots + a_0(x),$$

is of lower degree in y than E^* and hence, less complex than E^*, we can test to check if it is equivalent to 0. If it is, E^* and therefore E are also equivalent to 0. If it is not, E^* and E are not equivalent to 0.

If a_n is not equivalent to 0, divide E^* by it resulting in the expression $E2$.

$$E2: y^n + [a_{n-1}(x)/a_n(x)]\, y^{n-1} + \cdots + a_0(x)/a_n(x).$$

Now differentiate, resulting in an expression, say $E3$, of the form

$$E3: ny^{n-1}y' + \cdots + (a_n a_0' - a_0 a_n')/a_n^2,$$

$E3$ is of lower degree in y than E^* since the derivative of a logarithmic term is of lower complexity than the term itself. (Note that this is essentially the only fact we need to know about logarithms except for cases where the constant problem arises.) If $E3$ is not equivalent to 0, then $E2$ and therefore E^* and E are not equivalent to 0. If $E3$ is equivalent to 0, then E^* is equivalent to a constant multiple of a_n. To complete the algorithm we must determine if the constant is 0. Thus the constant problem arises in Richardson's algorithm. One could attempt to evaluate the expression at a point as Oldehoeft does [26]. The situation here is simpler than in Oldehoeft's cases since if the function is equivalent to a constant multiple of a_n (which is not 0) we need not worry about accidental values of 0 arising in the evaluation. In many cases, it suffices to know the exact value of the logarithmic terms at only one or two points.

If the most complex term y is an exponential, then Richardson's algorithm involves division by a_0. Differentiation will then yield a low order term equal to 0. Since the derivative of y^k is of degree k in y, the rest of the derivative can be divided by y to yield an expression similar to $E3$ which is of lower degree than E^*.

At the heart of Richardson's reduction procedure is the idea that through differentiation we can obtain expressions which can be transformed in such a way as to yield simpler problems whose solution will determine the answer to the original equivalence problem. It turns out that this idea can be used to test expressions which involve functions other than exponentials and logarithms.

In [25] it is shown that Richardson's algorithm can be extended to accept functions defined by a differential equation of the form $y' = P(x, y)$, where P is a polynomial in y. When P is linear in y the extension is

straightforward. P's which are quadratic in y are of great importance in applied mathematics. Unfortunately, when a function is defined by a quadratic P, then its derivative is more complex than itself. Thus if we are testing $E(x, y(x))$ for equivalence to 0, we shall usually find that $E'(x, y(x))$ has a higher degree in y than E does. If $E \equiv 0$, then $E' \equiv 0$, and therefore, the greatest common divisor of E and E' is also equivalent to 0. Conversely if the gcd of E and E' is equivalent to 0, so is E. Hence, we may use the result of the equivalence test for the gcd. The gcd, however, may not be of lower degree in y than E itself is. In such cases it must possess the same degree in y as E does. Therefore, we may properly speak of E dividing E'. Let us say that $E'/E = Q(x, y)$. Therefore, integrating both sides

$$\log E = \int Q(x, y)\, dx + C_1$$

$$E = C_2 \exp\left(\int Q(x, y)\, dx\right)$$

where C_1, C_2 are constants.

Exponentials usually cannot have a zero value. In such cases E can have a zero value only if C_2 is identically zero. This determination is another constant problem of a special nature in that we are dealing with a function that is either always 0 or never 0. An exponential can have a zero value when the argument goes to $-\infty$. Such cases would be disallowed by Richardson's requirement that expressions be totally defined.

3.3.3 Roots of Polynomials. In [4], Caviness discusses a class of expressions which is obtained from the rational numbers, the variable x, the rational operations, and the operation of exponentiating to a rational number. The exponentiation in this class may not be nested. The following expressions are in this class:

$$1/(x^{\frac{1}{2}} + x^{\frac{1}{4}}),\ (4 - x)^{\frac{1}{3}}/(x^2 + 2)^{\frac{2}{3}}.$$

The expression $(x + 3^{\frac{1}{2}})^{\frac{1}{4}}$ is not in this class because it involves nested exponentiation by nonintegers. Caviness shows that there exists a zero-equivalence simplification algorithm foi this class of expressions. The algorithm is not canonical. Furthermore, it is also very time-consuming since it can easily force one to factor polynomials (over the integers) having a high degree, and factorization is still a very expensive operation.

Recently, Fateman [12] showed that factorization is usually not necessary if we modify the meaning of a radical expression. What Caviness means by a radical expression such as $\sqrt{x}$ is a symbol which represents the general root of a polynomial equation (i.e. $y^2 - x = 0$). That is, $\sqrt{x}$ can be either one of the roots normally written as $+\sqrt{x}$ and $-\sqrt{x}$. Fateman's algorithm assumes that the symbol $\sqrt{x}$ represents exactly one of the roots, and that $-\sqrt{x}$ represents the other.

Fateman's algorithm, except for expressions involving roots of -1, has the same property as Caviness', namely the zero-equivalence property. Yet all he needs to test is whether the integers and polynomials which occur inside the radicals are relatively prime to each other. He would decompose $(x^2 - 1)^{\frac{1}{2}}$ into $(x - 1)^{\frac{1}{2}}(x + 1)^{\frac{1}{2}}$ if $(x - 1)^{\frac{1}{2}}$ or $(x + 1)^{\frac{1}{2}}$ occurred elsewhere in the expression. But $(x^2 + 2)^{\frac{1}{2}}$ would be left unchanged since no other simplified radical expression could combine with it under the rational operations. In both Caviness' and Fateman's algorithms the proof that the simplification algorithm has the zero-equivalence property is obtained without resorting to additional conjectures.

Fateman's algorithm can be made canonical (again except for roots of -1) by removing radicals from denominators in quotients through a generalization of the process of "rationalizing the denominator." Thus $1/(x + \sqrt{2})$ could be converted to $(x - \sqrt{2})/(x^2 - 2)$ in order to achieve a canonical form.

3.3.4 Unsolvability Results. The best-known negative result in algebraic manipulation is a theorem by Richardson [5, 28] that shows that there exists a class of expressions E for which the zero-equivalence problem is recursively unsolvable.

The starting point for most unsolvability results in algebraic manipulation is Hilbert's Tenth Problem. This problem, also known as the Diophantine Problem, asks whether there exists an algorithm for telling whether polynomials in several variables with integer coefficients have solutions which are integers. This problem has been recently shown to be recursively unsolvable [21].

For any polynomial $P(x_1, x_2, \cdots, x_n)$, with integer coefficients, we can reduce the question of whether $P = 0$ has integer solutions to whether the equation

$$\sum_{i=1}^{n} \sin^2 \pi x_i + P^2(x_1, x_2, \cdots, x_n) = 0$$

has real solutions since each term $\sin^2 \pi x_i$ forces x_i to be an integer. By manipulating the equation above, Richardson was able to show the existence of a set of functions $G_i(x)$ derivable from any polynomial $P_i(x_1, x_2, \cdots, x_n)$ such that it was recursively unsolvable to decide whether $G_i(x) < 0$ for some value of x.

At this point we have an undecidability result for the class of expressions formed by the rational numbers and π, the variable x, the operations of addition and multiplication, and the sine function (which can be nested). By adding the absolute value function to this class, Richardson was able to modify the $G_i(x)$ to $F_i(x)$ such that the question "$F_i(x) \equiv 0$?" could not be solved recursively.

Richardson's unsolvability result is considered by some people to be an important limit to theoretical results in algebraic manipulation. On the other hand, the generality of the extensions possible to Richardson's zero-equivalence algorithm (see Section 3.3.2) give a much more optimistic outlook. In fact, the unsolvability problem may lie in Richardson's use of the absolute value function. When one adds the absolute value function to a class of functions which forms a field (e.g. the rational functions), then one introduces zero-divisors. For example, $(x + |x|)(x - |x|) = 0$, although

neither factor is 0. The lesson which Richardson's results taken *together* may hold for algebraic manipulation is that one should exercise great care in introducing functions other than the "natural" ones which are solutions to differential equations, but that with these natural functions, very powerful simplification procedures are possible.

4. Prospects for the Future

Although the field of algebraic manipulation can already claim a number of important advances in the design of algorithms, and a significant number of important applications, one cannot yet say that the field has stabilized. We have already witnessed the demise of purely conservative systems. In the next few years we may witness the demise of purely liberal systems. The reason for the diminished importance of such systems is their inefficiency when compared to the algorithms provided in radical systems or subsystems. It would not be surprising if many radical systems mature into new left systems when the restrictions of canonical forms become unbearable. This would leave the new left systems with a single representation, which is a compromise between the radical and liberal representations, and the catholic systems with their multiple representations. We believe that the theoreticians and the major users will tend to gravitate to the new left systems, and the systems designers to the catholic ones.

We expect to see theoretical results about simplification algorithms encompass increasingly larger classes of expressions. The generality of the extensions which can be made to Richardson's zero-equivalence algorithm lead one to expect that in the next few years it may be possible for a user to define a function as a solution to a differential equation and then employ that function and its simplification properties immediately in a calculation.

One thing that we do not expect is that the difficulties of using algebraic manipulation systems will disappear completely. Users will continue to complain, and designers will, we hope, continue to improve their creations.

Acknowledgments. The author wishes to acknowledge useful criticism of drafts of this paper by B.F. Caviness, C. Engelman, R. J. Fateman, J. P. Golden, W.A. Martin, L.P. Rothschild, and the editor.

References

1. Abrahams, P.W. Application of LISP to sequence prediction. *Comm. ACM 9*, 8 (Aug. 1966), 551.
2. Brown, W.S., et al. The ALPAK system for non-numerical algebra on a Digital Computer-II. *Bell Sys. Tech. J. 43*, 2 (Mar. 1964), 785–804.
3. Brown, W.S. Rational exponential expressions and a conjecture concerning π and e. *Amer. Math. Monthly 76* (Jan. 1969), 28–34.
4. Caviness, B.F. On canonical forms and simplification. Ph.D. diss., Carnegie-Mellon U., Pittsburgh, Pa., Aug. 1967.
5. Caviness, B.F. On canonical forms and simplification. *J. ACM 17*, 2 (Apr. 1970), 385–396.
6. Collins, G. PM, A system for polynomial manipulation. *Comm. ACM 9*, 8 (Aug. 1966), 578–589.
7. Collins, G. The SAC-I system: An introduction and survey. SYMSAM II, pp. 144–152.
8. Collins, G. Subresultant and reduced polynomial remainder sequences. *J. ACM 14*, 1 (Jan. 1967), 128–142.
9. Engelman, C. The legacy of MATHLAB 68. SYMSAM II, pp. 29–41.
10. Engeli, M. User's manual for the formula manipulation language SYMBAL. Computer Center, U. of Texas at Austin, 1968.
11. Fateman, R. The user-level semantic matching capability in MACSYMA. Proc. 2nd. Symp. on Symbolic and Algebraic Manipulation, ACM Headquarters, New York, pp. 311–323.
12. Fateman, R. Essays in algebraic simplification. Ph.D. diss., Harvard U., Cambridge, Mass., 1971.
13. Fenichel, R. An on-line system for algebraic manipulation. Ph.D. diss., Harvard U., Cambridge, Mass., 1966.
14. Griesmer, J.H., and Jenks, R.D. SCRATCHPAD/1: An interactive facility for symbolic mathematics. SYMSAM II, pp. 42–48.
15. Hall, A.D. The ALTRAN system for rational function manipulation—A survey. *Comm. ACM 14*, 8 (Aug. 1971), 517–521.
16. Hearn, A. REDUCE: A user-oriented interactive system for algebraic simplification. In *Interactive Systems for Experimental Applied Mathematics*. Academic Press, New York, 1968, pp. 79–90.
17. Hearn, A. The Problem of Substitution. Proc. 1968 Summer Inst. on Symbolic Math. Comput. IBM, Cambridge, Mass., pp. 3–19.
18. Hearn, A. REDUCE 2: A system and language for algebraic manipulation. Proc. 2nd Symp. on Symbolic and Algebraic Manipulation, ACM Headquarters, New York, pp. 128–135.
19. Korsvold, K. An on-line algebraic simplification program. Artif. Intell. Proj. Memo. no. 37, Stanford U., Stanford, Cal., Nov. 1965.
20. Martin, W., and Fateman, R. The MACSYMA system. SYMSAM II, pp. 59–75.
21. Matijasevic, J.V. Enumerable sets are diophantine. *Soviet Math. Dokl., 11*, 1970.
22. Minsky, M., and Papert, S. *Perceptrons*. MIT Press, Cambridge, Mass., 1969.
23. Moses, J. Symbolic integration. Report MAC-TR-47, Project MAC, MIT, Cambridge, Mass., Dec. 1967. (Available as AD #662–666, Clearinghouse, Springfield, Va. 22151.)
24. Moses, J. A canonical form for first order exponential expressions. In preparation.
25. Moses, J., Rothschild, L.P., and Schroeppel, R. A zero-equivalence algorithm for expressions formed by functions definable by first order differential equations. In preparation.
26. Oldehoeft, A. Analysis of constructed mathematical responses by numeric tests for equivalence. Proc. ACM 24th Nat. Conf., 1969, ACM, New York, pp. 117–124.
27. Perlis, A., et al. A definition of Formula Algol. Comput. Center, Carnegie-Mellon U., Pittsburgh, Pa., Mar. 1966.
28. Richardson, D. Some unsolvable problems involving functions of a real variable. Ph.D. diss., U. of Bristol, England, 1966.
29. Richardson, D. Some unsolvable problems involving elementary functions of a real variable. *J. Symb. Logic 33* (1968), 511–520.
30. Richardson, D. A solution of the identity problem for integral exponential functions. *Z. Math Logik u. Grundlagen Math*, to appear.
31. Simon, H. *The Science of the Artificial*. MIT Press, Cambridge, Mass., 1969.
32. Tobey, R. Experience with FORMAC algorithm design. *Comm. ACM 9*, 8 (Aug. 1966), 589–597.
33. Tobey, R., et al. Automatic simplification in FORMAC. Proc. AFIPS 1965 FJCC, Vol. 27, Pt. 1, Spartan Books, New York, pp. 37–57.