ALGEBRAIC SIMPLIFICATION
A GUIDE FOR THE PERPLEXED

by
Joel Moses

Project MAC, MIT

Abstract

Algebraic simplification is examined first from the point of view of a user needing 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 non-existence of algorithms for certain classes of expressions are included.

Table of Contents

1.0 Introduction
2.0 Simplification for the Sake of Comprehension - The Needs of Users
  2.1 Conventional Lexicographic Ordering of Expressions
  2.2 Substitution as an Aid to Comprehension
3.0 Simplification for the Sake of Efficient Manipulation - What Designers Provide
  3.1 The Politics of Simplification
    3.1.1 The Radicals
    3.1.2 The New Left
    3.1.3 The Liberals
    3.1.4 The Conservatives
    3.1.5 The Catholics
  3.2 Intermediate Expression Swell
  3.3 Canonical Simplifiers and Theoretical Results - The Radicals Revisited
    3.3.1 Simplification Algorithms for Expressions with Nested Exponentials
    3.3.2 Expressions involving Exponentials and Logarithms
    3.3.3 Roots of Polynomials
    3.3.4 Unsolvability Results
4.0 Prospects for the Future
  References
  Figures

1.0 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, a requirement which usually means that the expressions presented to the user should be small. The designer wants expressions which can be manipulated with great ease and efficiency, a requirement which translates to a desire for a uniform representation of expressions utilizing a minimum number of functions. 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 the exponentials, sines and cosines, or just exponentials with both real and complex arguments, or nothing but rational functions.

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, but not its value. Changes of representation occur in many problem solving domains. In fact, in the field of Artificial Intelligence one speaks of the Problem of Representation which can be stated roughly as "how does one transform the statement of the problem into a form which is more readily solved." Thus an ideal, but not very helpful, way to describe simplification is that it is the process which transforms expressions into a form with which the remaining steps of the problem can be taken most efficiently.

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 non-trivial problem to recognize the equivalence. For example, it is rare that we do not want to recognize that an expression is equivalent to 0. However, many of us have difficulty in recognizing the following identities.

$$\log(e^{2x} + 2e^{x} + 1) - 2\log(e^{x} + 1) = 0$$

or

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

or

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

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. Thus a bad representation of the product or one of its factors is propagated and magnified n-fold.

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,

$$\frac{x^7}{x^{12} + 1}$$

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

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

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. Designers can take comfort in the fact that while the simplified form of an expression is not the same in every context, it is so in many contexts. Nonetheless, the task of deriving a compromise between the wishes of users and the requirements of an efficient system is likely to keep designers talking to themselves and to each other for quite some time.

A related aspect of 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, one 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

$$a + e^{x^2} + 2x^2\, e^{x^2}$$

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 manipulating system. In the remainder of this paper our discussion of simplification will range from mundane topics (such as whether one writes x+a or a+x) to sublime ones (whether $e+\pi$ is a rational number). We shall concentrate in section 2 on the users' need for comprehension of expressions, and then in section 3 on the designers' facilities for manipulating expressions.

In discussing simplification for the sake of comprehension, we shall describe the technique of substituting labels for subexpressions in simplifying large expressions. We shall also describe principles of conventional lexicographic ordering of expressions. A system can hinder the comprehension of a user by displaying expressions in unconventional order. Unfortunately, no current system pays sufficient heed to this point.

In section 3 we describe five different approaches to the design of simplification facilities in algebraic manipulation systems. We describe the facilities which are usually offered, and indicate the advantages and disadvantages in them. The section ends with a discussion of very powerful algorithms based on formalizations of the concepts of simplification.

2.0 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 such a form as to optimize the use of space and time in the calculation. We do not wish to underestimate the difficulties in the simplification problem for such users. However, in this section we will be mainly concerned with the needs of users who do care about the form of expressions which result from a symbolic calculation.

The latter class of users include 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 symbolically manipulating a 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. He could, of course, plot the value of the expression for several values of the variables, but this method may not be very useful if there are many variables in the expression. Other users might need to examine an expression in order to know what the next manipulatory step should be. A simple instance of such a situation occurs when the next step in the calculation depends on whether the expression is linear or quadratic in a given variable.

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

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.

simplification should be to produce small expressions. In fact, most of the usual simplification transformations such as collecting terms in sums ($x + 2x \rightarrow 3x$), collecting exponents in products

$$(xy^2 \frac{1}{x} \rightarrow y^2) ,$$

removing 0 terms in sums ($x + 0 \rightarrow x$), and factors of 1 in products ($1 \cdot x \rightarrow x$), produce smaller expressions. In fact, transformations which produce larger expressions (e.g., expanding integral powers of sums

$$(x + 1)^3 \rightarrow x^3 + 3x^2 + 3x + 1)$$

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

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

is a lot 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]. Furthermore, many descriptions which are of value to humans are incomplete descriptions of an expression (e.g., the expression has f's whenever a y occurs except for the first term). Most programs are unable to deal with incomplete descriptions. Both of these drawbacks mean that we have to depend on the user to obtain a description for his own use. The process of "massaging an expression" which people use in pencil-and-paper calculations, can be characterized as a trial-and-error attempt to decrease the size of an expression and to transform it into a form for which a short description is apparent. As we have remarked, it is not possible now to fully automate this massaging process. Rather, it should be the goal of a good system to provide machinery which will aid the user in his massaging efforts. The remainder of this section will be devoted to describing such machinery.

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

A major reason why computers are not 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 (preferably single literals) for large subexpressions which either occur frequently or possess some meaning. We shall examine this technique in section 2.2.

Working with an algebraic manipulation system can frequently be annoying because such systems do not display expressions in the way to which the user has become accustomed. A user who is presented with a quadratic in x written as $c + x^2a + bx$ is not only annoyed, but is incapable of understanding such expressions as well as those written in a more conventional form. In section 2.1 we present an attempt to model the rules which govern the conventional ordering of expressions.

2.1 Conventional Lexicographic Ordering of Expressions

As noted above, most systems do not make a very satisfying attempt to produce expressions in a conventionally ordered form. Fenichel's FAMOUS [14], and PL/1 FORMAC [44] make some attempt to supply conventional lexicographic ordering. However, each system provides an incomplete solution to the problem. We have also not encountered in the mathematical literature an analysis of what constitutes conventional ordering of expressions. The rules that people implicitly use are apparently not hard and fast ones. For example, $x + e^x$ seems not much preferable to $e^x + x$. In addition, context plays a role in modifying very strongly held views about ordering of expressions. One writes $\underline{am}$ rather than $\underline{ma}$ except in cases like $F = \underline{ma}$ in which $\underline{m}$ is relatively constant and $\underline{a}$ varies.

We believe that systems should make greater effort to avoid highly unconventional and therefore confusing ordering of expressions such as in $\sin(-x + 1)/2$. The following discussion is presented with that goal in mind. We do not claim that our discussion of lexicographic ordering is complete, or that a rather different set of rules and principles would not serve equally well to describe conventional mathematical notation. If we have done our job well, the reader will not find many surprises in our analysis.

Difficulties in ordering expressions arise when one deals with commutative operations. The sole commutative operators in most algebraic manipulation systems are PLUS and TIMES. As a first approximation, the following principles apply to ordering terms in a sum and factors in a product.

Principle for ordering products: Factors increase in complexity in a left-to-right scan.

Examples: $2xe^x$, $2x^2y^3\sin(y)$

Principle for ordering sums: Terms decrease in complexity in a left-to-right scan.
Examples: $3x^2 + 5x - 6$, $2x^2 - 3ax + a + 1$

The principles stated above beg a definition of complexity. A handle on such a definition is obtained by classifying expressions into three groups: constants, variables, and functions. Constants are less complex than variables which are in turn less complex than functions of variables. Simple constants are either numbers or literals, with numbers less complex than literals (e.g., $2a$, $(3/2)\pi$, $a + 1.4$). Functions of simple constants are more complex than constants (e.g., $3\sqrt{2}$, $3ae^a$), but less complex than variables (e.g., $a^{3/2}x$).

The usual convention regarding literal constants and variables is that constants occur early in the alphabet (e.g., a, b, c) and variables late in the alphabet (e.g., x, y, z). Letters in the middle of the alphabet play many roles, (e.g., f frequently represents a function, and n an integer) but they can be considered, for our purposes, to be intermediate between constants and variables. Within each class (i.e., constants, intermediates, and variables) alphabetic ordering appears to determine the complexity. In products, literals early in the alphabet in each class have the lowest complexity rank. This means that literals in a product are ordered purely alphabetically (e.g., $abxz$, $2amn$). The complexity rank within each class is reversed in sums. Thus we get $x + y$ rather than $y + x$, and $x + a + b$ rather than $x + b + a$. In sums, moreover, the class rank is not adhered to very strictly. Thus, we frequently see $a + b + x$ rather than $x + a + b$ or $1 + x - y$ rather than $x - y + 1$. Since attitudes about ordering of products are more strictly held than those about sums, we shall confine most of our examples to products.

Functions of variables are more complex than the variables themselves. However, in products, positive integral powers of an expression A possess the same rank as A (e.g.,

$$x^2y,\ x^2\sin x).$$

In sums, a term having a higher exponent of the most complex factor of an expression is ranked higher than that expression, but possesses the same rank relative to different expressions (e.g.,

$$3x^2 + 2x,\quad x \sin^2x + x \sin x).$$

One could rank the complexity of the usual functions[3], but surely any such ranking must be somewhat arbitrary. It seems that sin x cos x is preferable to cos x sin x, but is sin x log x preferable to log x sin x?

Because of our grouping convention, positive integer powers of variables are ranked lower than any other functions of variables (e.g.,

$$x^2ye^x\sin y).$$

We believe that this is a better model of conventional notation than an extension of the alphabetic ranking of variables which would read: any function of a given variable, say x, is less complex than any expression containing a variable more complex than x. Using the latter convention, the above example would be written

$$x^2e^xy \sin y.$$

Expressions which are more involved than the ones we have been examining up to now can be ranked by a recursive test on their most complex subexpressions. We do not wish to give a detailed discussion of such a ranking procedure because there does not seem to be strong feeling about the ordering of very large or involved expressions except when patterns clearly appear among the subexpressions (e.g., polynomials, power series, Fourier series). Most of the common situations should be handled correctly by a recursive ranking procedure, but the ordering of some expressions would involve global pattern recognition which is beyond the capabilities of existing algebraic manipulation systems (e.g.,

$$(a + b)(a + \tfrac{1}{2}b)(a + \tfrac{1}{6}b)(a + \tfrac{1}{24}b)\quad).$$

There are additional conventions for ordering expressions which algebraic manipulation systems should follow. One rule involves eliminating the leading minus sign in a sum. Thus given an expression of the form $-A + B$ one reverses the terms to eliminate the leading minus sign (e.g., $1 - 2x$ rather than $-2x + 1$, $y - x$ rather than $-x + y$). One could conceivably extend the rule to include sums with two or more terms, but the force of the rule seems to diminish considerably with an increase in the number of terms (e.g., $-3x + 4y + 5z$ rather than $4y - 3x + 5z$ or $5z + 4y - 3x$ or $4y + 5z - 3x$).

Conventional notation tends to avoid the use of parentheses in functions of a single argument (e.g., sin x, log 2). As a result of this convention and also in order to avoid confusion with the scope of special signs such as the square-root, integral and summation sign, one sometimes reverses the normal order (e.g., $2i\sqrt{2}$ rather than $2\sqrt{2}\ i$, $3y \sin a$ rather than $3 \sin(a)y$). This convention is not crucial for algebraic manipulation systems since such systems rarely display expressions with ambiguous scopes of functions or signs.

The constant π has lower rank in products than literals which represent angles (e.g., $2\pi\theta$ rather than $2\theta\pi$). However, expressions which evaluate to integers or rational numbers rank lower than π (e.g.,

$$(\frac{3n+1}{2})\pi \text{ rather than } \pi(\frac{3n+1}{2})\quad).$$

Few constants lead the charmed life of π as exemplified by the expression $2n\pi i\theta$.

Frequently, certain literals are used as implicit functions of other variables. Such literals should be ranked as functions. This convention accounts for the subexpression $y + x + 1$ in $y' + y + x + 1 = 0$.

Some systems allow users to declare dependencies such as that of y on x in the expression above. One should extend this machinery to allow users to override the conventional classification of literals. By declaring m constant and a variable, one would be able to obtain the usual formula $F=ma$.

Our final remark about conventional ordering regards instances of general patterns.

3. In PL/I FORMAC [44], the following ranking is employed in products; $e^x <$ erf(x) $<$ log(x) $<$ sin(x) $<$ cos(x) $<$ atan(x) $<$ sinh(x) $<$ cosh(x) $<$ atanh(x) $<$ x!

If one is given a general pattern of certain expressions, such as a + bi for complex numbers, then instances of that pattern will follow the order given in the pattern, rather than the usual rules (e.g., -1 + i rather than i - 1). Systems should accommodate definitions of such patterns.

2.2 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

$$\frac{\left(\frac{\partial f}{\partial x}\right)^2 + 2E^2}{f^3}$$

Then one attempts to simplify the expression which results. 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.

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 same function that f, y, and z had in forming the expression above.

An artificial example which points out the value of substitution to the comprehension of an expression occurs in [41]. 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 those subexpressions which are good 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 subexpressions 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 non-artificial example of the use of substitutions which is borrowed from Hearn [19] is given in figure 2.

A technical problem arises when one makes substitutions for expressions other than atomic or literal ones. Consider the problem of substituting $\underline{a}$ for xy^2 in the expression

$$x^2y^3.$$

Some possible results of this substitution are

1) x^2y^3
2) axy
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. However, no system, until recently, gave the user much choice in the result of substitution. The REDUCE system of Hearn [18], 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 [26] 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 α_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 to 1) not contain a sum, 2) be polynomials (and not rational), and 3) 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 substitution made with Fateman's routines are given in figure 3. In each case we substitute A for B in C.

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

$$\sin^2(x) + \cos^2(x) \rightarrow 1.$$

One of the most popular uses of substitution occurs in the differentiation of products. This kind of substitution really addresses itself to the problem of decreasing the size of an expression during a computation. However, the techniques used can effect an increase in the intelligibility of the final expression. Moreover, these techniques are of a fairly general nature, and can be used to improve intelligibility in other cases.

Consider the general form of the first derivative of a product of four factors f, g, h, k:

$$(fghk)' = f'ghk + fg'hk + fgh'k + fghk'$$

In the general form f, g, h, and k each appear three times. In certain cases the general form will simplify because f', g', h' or k' may be 0. Frequently this situation will not occur. In cases where f, g, h, and k are complex expressions, considerable savings in space will result if we use labels for the factors which would appear in the derivative. A disadvantage of labelling the factors is that some simplifications which would occur due to the presence of the factor in the expression would not occur when its label is used. For example, the difference of a label and the expression it is labelling is not zero.

Problems arise when we have to take derivatives of labelled expressions. We can use the strategy of not evaluating all derivatives of labels until some later time. We might wish to evaluate certain derivatives of labels whenever they occur by replacing the labels by the expressions they represent. Clearly, several strategies are possible. A technical problem here involves the ability of a system to produce the labelled expressions on demand. This can be done by substitutions. However, the following statement in MACSYMA [26] avoids making explicit substitutions and is very selective in its effect. The statement has the general form

```
WHEN conditional DO label = expression
```

An instance of such a WHEN statement is

```
WHEN SHOWF DO F = SIN(X)/(COS(X) + 1).
```

As a result of executing such a statement, the variable F will appear as F inside expressions as long as the variable SHOWF is FALSE. When SHOWF is changed to TRUE then F will be replaced by the expression SIN(X)/(COS(X) + 1) when it is encountered in a computation. When SHOWF is changed back to FALSE all remaining F's will appear as F once again.

In summary, the technique of labelling can be used whenever an algorithm tends to duplicate expressions. The labelled expression yields to structural analysis more easily than the expression it replaces. However, the technique has its drawbacks since it prevents certain simplifications from taking place.

Another situation where labelling can be used occurs when one is unable to display an expression on one page. Suppose the expression is a sum. Then one could replace as many of the terms in the sum by labels as are needed to allow the labelled expression to be displayed in one page. The labelled terms can be displayed independently. This technique does not decrease the size of an expression. However, it affords a simple way of breaking up the large problem of analyzing the whole expression into a number of smaller problems. Success with this technique depends on clever decompositions of an expression. Automatic routines for introducing labels into expressions by Baker [43] and Martin [24] cannot be considered great successes. At the present time, we require an interacting user to break the expression up into chunks which he can conveniently manage and usefully comprehend.

3.0 Simplification for the Sake of Manipulation - What Designers Provide

3.1.0 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 approach to simplification.

Four years ago, when we last surveyed the scene [29], 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 of representation of an expression given 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 <u>radical</u> 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 <u>conservative</u> system. A system which will make certain transformations automatically, but will leave others to the discretion of a user is called a <u>liberal</u> system. The <u>new left</u> 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.

In the remainder of this section we shall describe the facilities offered by the different systems. We shall then consider a major problem in the manipulation of expressions, that is, the tremendous growth in the size of intermediate expressions in a computation. Finally, we shall consider the design of simplification algorithms based on canonical forms which is the most theoretical topic in algebraic simplification.

In reading an essay such as this, the reader should bear in mind that the author, as an interested party, will tend to bias the discussion toward his point of view. Our attitude is best described as a catholic one. Such a position means that we see the merit in the other approaches for some situations. However, it is probable that our discussion of any view other than our own will be less positive than that of a strong adherent to that view.

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 in a unique way internally. This means that the system stands ready to make a major change in the representation of an expression 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. Such systems do not appear to have specialized simplification machinery since the process of generating expressions in canonical form which is automatically 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 radical designers.

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 [9, 10]. 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 [3], is the expanded representation. The first polynomial is written in expanded form.

Situations in which there is wide-spread 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, 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.

A similar situation occurs in radical rational function systems. The canonical representation in such systems is a quotient of a numerator written as a polynomial in canonical form and a denominator which is likewise written in canonical form. One must, for the sake of canonicalness, combine sums of quotients into a single quotient and divide the resulting numerator and denominator by their greatest common divisor. Suppose we wanted the partial fraction decomposition of

$$\frac{2x + 3}{x^2 + 2x + 5} - \frac{1}{3x + 2} + \frac{3x}{x^2 + 2x + 6}$$

The problem is straight-forward (in fact, solved) if one leaves the expression as it stands. However, a radical system would first combine the quotients and proceed to rederive the expression above.

One can claim that radical systems can handle only a small subset of the expressions that are commonly found in applied mathematics. However, theoreticians have been chipping away at this problem so that radical systems can now handle a wide variety of expressions which include exponentials, trigonometric functions, roots of polynomials, etc. The idea is to introduce labels for a minimal number of nonrational expressions in such a way that the labelled expression is in canonical form. (See section 3.3). For example, in order to deal with rational functions of trigonometric functions in x one can represent sin x and cos x in the complex exponential form (e.g.,

$$\sin x = \frac{e^{ix} - e^{-ix}}{2i}).$$

Then substitute $y = e^{ix}$ in the expression to obtain a rational expression in y. Now consider integrating $\sin x \cos^{10} x$ with respect to x. This becomes, after appropriate translation,

$$\int \frac{1}{2i} (y - \frac{1}{y}) \frac{1}{2^{10}} (y + \frac{1}{y})^{10} \frac{1}{iy} dy$$

The original problem is trivial to integrate, and produces a concise integral. The translated problem is more expensive to solve and the solution is not very comprehensible. This technique (or the similar technique of substituting

$$y = \tan \frac{1}{2}x)$$

fits eminently into the radical way of solving problems. It has the advantage that it will work and give some result when less general, more heuristic techniques will fail.

Because the expressions and the manipulatory algorithms of radical systems are so well defined, there is a great likelihood that theoretical research will find ways to improve the algorithms. This has, in fact, been the case. 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. As a result, systems which do not transform expressions into a canonical form do not boast algorithms as powerful or efficient as those of radical systems.

A radical system, in effect, forces a user to tailor the problem to fit the system. When such tailoring is clearly out of the question, the radical solution is to build a new system expressly for the problem or class of problems that the user has. This accounts for the number of distinctly different radical systems which have been written for different problem areas.

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 a radical system would. The user of a new left system is given the ability to decide when expansion is most appropriate, a facility which is, of course, not present ir a radical system.

Systems which allow unexpanded terms in an expression are Hearn's latest version of Reduce [20], and the latest version of ALTRAN [16].

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, once again, is to use labels for non-rational expressions. Thus

$$xe^x + x^2$$

might be rewritten as

$$xy + x^2z, \text{ where } y = e^x, \text{ and } z = \sin x.$$

The expression

$$\frac{e^{2x} + e^x}{e^x}$$

would probably be expressed as

$$\frac{y + z}{z}, \quad y = e^{2x}, \quad z = e^x$$

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

Some systems permit one to represent non-rational expressions in a way similar to that indicated above, but force expansions to be made once the translation pass is over. Such systems should probably be considered to be more canonical than new leftish. Examples of these systems are Hearn's early versions of Reduce [18], MATHLAB's rational function subsystem [23], and MACSYMA's rational function subsystem [26].

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 non-rational 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

$$\sqrt{1-x^2}\ .$$

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 were, the gcd 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 for them than for radical or new left systems.

As was indicated in our discussion of radical systems, certain problems can be solved more efficiently in the flexible environment provided in liberal systems. A liberal designer wants to minimize the transition from the user's usual techniques. He would pay relatively great attention to making his displayed expressions intelligible to a user. Radical designers are usually interested in solving large computational problems of a specialized nature. New left designers can be said to aim for a large proportion of the users of both liberal and radical systems. Such competition should benefit all users.

Most liberal systems will reorder terms in sums and factors in products based on some lexicographic ordering scheme. As was mentioned in 2.2, such schemes frequently produce rather unnatural orderings. Thus a user writing $x + y + z + w$ can expect to get any permutation of the terms as a response, depending on the nature of the ordering being used. Some systems, such as MATHLAB, minimize the use of simplification of expressions partly in order to avoid an unnatural ordering as much as possible. Other liberal systems do not use a lexicographic ordering at all, and prefer to use the ordering originally presented by the user. Such an ordering will, of course, be modified when the expression is manipulated. Martin [25] used a unique hash-coding scheme with which to tag expressions in order to be able to recognize when to collect terms or factors. Martin's hash-code was powerful enough to recognize many identities. However, the cost of hash-coding, and the fact that ordered expressions can be manipulated more efficiently than unordered ones, probably led to some inefficiency in his system.

Martin's hash code assigned to an algebraic expression an element in the finite field formed by the integers modulo some large prime. The prime was chosen so that certain elements in the field had useful properties. For example, one element acted like i since its square was -1. Given random values for the variables, the hash code assigned numbers to expressions so that equivalent expressions usually had the same code. Due to the finiteness of the field, non-equivalent expressions could be assigned the same code, but the probability of this event is extremely low.

Martin's hash code was utilized in an experimental program designed to teach freshmen how to integrate symbolically [31]. Another technique due to Oldehoeft [33], was also intended for a CAI environment. Oldehoeft examined the problems associated with determining the equivalence of expressions by evaluating them at random points. He discusses problems due to round-off, overflow, accidental coincidence and the effects of dealing with non-analytic functions (e.g., absolute value).

Simplification in liberal systems is performed by a program usually called the simplifier. Some of the earliest projects in algebraic manipulation involved the design of liberal simplifiers. The earliest simplifier was written in the LISP Assembly Program by Goldberg in 1959 [15], and was used in Slagle's Symbolic Automatic Integrator (SAINT) [42]. Other LISP-based simplifiers were written by Hart (1961) [17], Russell and Wooldridge (1963) [47], and Korsvold (1965) [22]. The Korsvold simplifier is used in MATHLAB and in SCRATCHPAD [2]. A highly modified version of it is also used in MACSYMA. The FORMAC system's simplifier is called AUTSIM. The philosophy behind AUTSIM is given in [45].

Liberal systems offer a user the ability to affect the representation of expressions through two kinds of mechanisms. One way of changing the expression is by using commands (such as EXPAND in FORMAC or MACSYMA) to perform the transformation. Another way is to modify certain switches whose value the simplifier checks to guide its operation. A switch might determine if a function such as log should evaluate to a floating point number if its argument is a number. Another switch might determine if certain indicated operations such as differentiation should, in fact, be carried out. This last example is not strictly an example of simplification; however, it does point out the fact that a simplifier is close to being the heart of a system.

3.1.4 The Conservatives

Designers of conservative systems claim that one cannot design simplification rules which will be best for all occassions. 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 [14], and FORMULA ALGOL [34].

The importance of conservative systems lies in the philosophy they represent, which is most clearly given by Fenichel [14], 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 pattern

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

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:

$$\text{Pattern} \rightarrow \text{Replacement.}$$

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 would 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 should apply a whole set of rules to the replaced expression. Only when no rule is applicable to a given expression is the algorithm complete.

Conservative systems offer variations of the Markov algorithm technique with which a user can generate his own simplification and manipulatory algorithms. Such rules are most easily written when making local transformations of an expression[4]. 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 transformation of an expression.

Several designers have added a capability for writing Markov algorithms to their systems, 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 [13] 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. The MATHLAB system is best described in this fashion. The MACSYMA system 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 is a conglomerate made up of several LISP-based systems. It has a total of four simplifiers.

Catholic systems emphasize the range of problems that can be solved by them. 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.

A catholic organization is only one way to obtain the advantages of the conservative, liberal, and radical approaches to simplification. A system such as REDUCE, can be viewed as a compromise system offering many of these advantages. REDUCE, however, uses only a single representation. Radical systems, as we noted earlier, use different representations for expressions which occur in different problem areas (e.g., polynomials and truncated Poisson series). While it is advisable to limit the number of distinct representations as much as possible, it appears likely that a system which tries to handle a large number of applications efficiently will require several representations. A designer of a catholic system is willing to accept such a situation. Other designers might not be so willing.

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 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" [43]. In many cases the final result of a symbolic calculation is quite small, but in order to get that result one finds oneself generating 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 with polynomials as coefficients. These coefficients might be obtained from the determinant of the matrix, which can be several pages long.

Intermediate expressions swell can be caused by several different phenomena. In some cases, the problem the user is trying to solve is inherently explosive, and it is likely that no general method will decrease the size of the intermediate expressions. We claim that such a situation exists when one tries to solve systems of simultaneous polynomial equations by eliminating variables [28]. The number of solutions to such systems can be as high as the product of the degrees of each polynomial. If the intermediate equations do not factor, as is likely to be the case, one is forced to generate a polynomial of very high degree which would be very hard to solve numerically for all its roots.

In certain other cases, the particular algorithm that the user, in combination with the system, has chosen for solving a problem, is bad and a radically different approach is necessary. For example, the recently developed modular algorithm for computing the greatest

4. 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 [27].

common divisor of two polynomials [3] is a radically different and much more efficient algorithm than any of the previous algorithms. Major changes in algorithm design usually require extensive analysis so that one cannot make such modifications on a regular basis. Certainly it is not very fruitful to consider such modifications as a task of algebraic simplification.

The final set of cases which we shall consider is when small changes in the sequence of steps cause a nontrivial improvement in the utilization of space and time. We have already mentioned the idea of labelling subexpressions which would tend to be repeated in a calculation. Sometimes 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' first improvement to the Euclidean GCD algorithm [11] was the idea that one could predict how certain terms were automatically introduced into the intermediate expressions, and therefore these terms could be cancelled without affecting the final result. Before the appearance of this algorithm, several people, including this author, thought that the size of the coefficients in the intermediate steps of the algorithm had to grow exponentially. Collins showed that they need grow only 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 $\frac{x}{1-x}$. Suppose you wanted

$$\sum_{j=0}^{n} y^j .$$

The straight-forward 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 [12].

3.3 Canonical Simplification and Theoretical Results - The Radicals Revisited

In this section we shall discuss most of the theoretical results related to algebraic simplification. The algorithms we shall describe are either canonical or else possess a strong property, namely that they can determine if an expression is equivalent to zero. Almost all of the algorithms are incomplete in the sense that they depend on, as yet, unproved conjectures about expressions involving constants. For example, the conjecture by Brown [4] 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 conjecture 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 exponential and absolute value functions, then one cannot, in general, tell whether such expressions are equivalent to zero.

3.3.1 Simplification Algorithms for Expressions with Nested Exponentials

In [4] Brown describes a simplification algorithm for a class of expressions he calls Rational EXponential (REX) expressions. REX expressions are obtained recursively from the rational numbers, i, and π, and the variables $x_1, x_2, \ldots, x_n$ by the rational operations of addition, subtraction, multiplication and division and by forming exponentials of existing REX. Thus the expression

$$\frac{e^{\left(\frac{e^x}{e^x+1}\right)}}{e^{5x} + 3e^{2x} + xe^{4e^1+1}}$$

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 exponentials in order to reduce an 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 (i.e., it does not always reduce equivalent expressions into the same form). It is powerful, though, since if we assume a certain conjecture, then we can prove that the algorithm simplifies any REX expression equivalent to 0 into 0. Thus the algorithm can determine if any two REX expressions are equivalent. It should be noted that since the constants i and π are included, the REX expressions contain the trigonometric and hyperbolic functions in exponential form.

In generating labels for the algorithms one must pay great attention not to allow algebraically dependent exponentials to be assigned to different labels. Two expressions are algebraically dependent (over the rationals) if there exists a nonzero polynomial with rational coefficients in these expressions which is equivalent to 0. Thus,

$$e^x \text{ and } e^{2x}$$

are algebraically dependent since

$$(e^x)^2 - e^{2x} = 0.$$

Likewise,

$$e^x,\ e^{x^2},\ \text{and } e^{x+x^2}$$

taken together are algebraically dependent.

Our labelling scheme must be such that if we assign

$$y = e^x,$$

then e^{2x} is assigned y^2.

The algorithm proceeds by replacing innermost exponentials in the expressions 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

$$\frac{e^x + x}{e^{2x} + 2xe^x + x^2} \ .$$

Traversing the numerator from left to right, we first encounter e^x. Let

$$q_1 = x \text{ and } r_1 = e^{q_1} = e^x.$$

Thus our first label is r_1. By substituting it into the expression we obtain

$$\frac{r_1 + x}{e^{2x} + 2xr_1 + x^2} \ .$$

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. However, 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

$$q_1 - 2q_2 = 0.$$

Therefore, redefine

$$r_2 = r_1^2$$

and by substitution obtain

$$\frac{r_1 + x}{r_1^2 + 2xr_1 + x^2} \ .$$

Simplifying the above as a rational function reduces it to

$$\frac{1}{r_1 + x}$$

Since no more exponentials are to be found, replace the labels by the exponentials. The result is

$$\frac{1}{e^x + x}$$

which is indeed simpler than the expression we had originally.

Brown's conjecture is that if

$$\{q_1, q_2, \ldots, q_k, i\pi\}$$

is linearly independent over the rational numbers,

$$\{e^{q_1}, e^{q_2}, \ldots, e^{q_k}, x_1, x_2, \ldots, x_n, \pi\}$$

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 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

$$\frac{e^{2x} + e^x}{e^x}$$

Let $q_1 = 2x$, $r_1 = e^{q_1} = e^{2x}$.

Now $q_2 = x$, $r_2 = e^x$, and $q_2 = 1/2 \ q_1$.

We cannot let $r_2 = r_1^{1/2}$ since we want to obtain rational results. So we redefine r_1 as

$$r_2^2$$

and obtain

$$\frac{r_2^2 + r_2}{r_2} = r_2 + 1 = e^x + 1$$

Brown's algorithm is not canonical because the algorithm does not make an optimal choice for labels.

Consider

$$\frac{e^{x+x^2}}{e^x}$$

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,

$$\frac{r_1}{r_2}$$

is simplified as a rational expression. Hence, the simplified result is

$$\frac{e^{x+x^2}}{e^x}$$

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

$$\frac{e^{\frac{1}{x+x^2}} + e^{\frac{1}{x}} e^{-\frac{1}{x+1}}}{e^{\frac{1}{x}}}$$

we can get

$$\frac{2\ e^{\frac{1}{x+x^2}}}{e^{\frac{1}{x}}}$$

using one order of assigning labels, and

$$2e^{-\frac{1}{x+1}}$$

using a different order.

The last two examples are intended to show the difficulties that a canonical simplifier for REX expressions has to surmount. There is a simple proof that such a simplifier exists, using Brown's conjecture. A little reflection will show that we can produce a function of a single integer which for increasing values of its input will yield syntactically valid REX expressions, and which will yield each REX expression for some input. The canonical simplifier will, given a REX expression, get the function to generate REX expressions until one is found which Brown's algorithm determines is equivalent to the expression to be simplified. The first expression found in this manner is considered the simplified result. The algorithm is canonical, assuming Brown's conjecture, since all equivalent REX expressions would result in the same simplified expression. However, the scheme is utterly inefficient. It also suffers from the fact that the simplified expressions are not describable 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 [7], 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 assuming a conjecture similar to Brown's, Caviness shows how exponential polynomials (other than pure polynomials) can be transformed into the form

$$P_1(x)e^{S_1} + P_2(x)e^{S_2} + \ldots + P_k(x)e^{S_k},$$

where the S_i are distinct exponential polynomials which are also in this form, and the P_i are non-zero, canonically ordered polynomials. To get exponential polynomials into this form one has to apply the usual algebraic transformations (including expansion), and collect the exponentials via

$$e^a e^b \rightarrow e^{a+b}.$$

The following exponential polynomial is in Caviness' form

$$5e^0 + (-3+x^2)e^x + (5+x)e^{1+x} + 3e^{3e^0+xe^x}$$

In order to guarantee that putting an expression into his form yields a canonical simplifier, Caviness must decide how terms are to be ordered in sums. This can be done by some lexical ordering scheme. Given some ordering scheme, Caviness requires that the S_i in his form be in increasing order. To prove canonicalness, Caviness assumes that if

$$C_1, C_2, \ldots, C_k$$

are different constant exponential polynomials written in his form (e.g.,

$$e^{3e^0+e^5}),$$

then

$$\{e^{C_1}, e^{C_2}, \ldots, e^{C_k}\}$$

is linearly independent over the rational numbers. The proof makes use of the idea that equivalent expressions are equal for each value of the variable x. The conjecture implies, among other things, that the set

$$\{e, e^e, e^{e^e}, \ldots\}$$

contains only transcendental numbers which, like the e + π conjecture, is unknown at present.

While Caviness' algorithm is quite powerrul, it suffers from the weakness that it does not permit division. Brown's algorithm, while it does permit division, does not yield canonical results. In [30] we describe a canonical 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 canonicalness 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.

$$\frac{x^5+x^3+1}{x^4-x^2} = x + \frac{-1}{x^2} + \frac{-\frac{1}{2}}{x+1} + \frac{\frac{1}{2}}{x-1}$$

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,

$$\frac{e^{x^2+x}}{e^x}$$

is decomposed into

$$\frac{e^{x^2}\ e^x}{e^x}$$

With proper relabelling and simplification of the resulting rational expression we obtained the simplified result

$$e^{x^2} .$$

Note that

$$\frac{1}{x^2 + x} = \frac{-1}{x + 1} + \frac{1}{x}$$

Therefore

$$\frac{e^{\frac{1}{x^2+x}} + e^{\frac{1}{x}}\ e^{\frac{-1}{x+1}}}{e^{\frac{1}{x}}}$$

is transformed into

$$\frac{e^{\frac{-1}{x+1}}\ e^{\frac{1}{x}} + e^{\frac{1}{x}}\ e^{\frac{-1}{x+1}}}{e^{\frac{1}{x}}}$$

Then if

$$q_1 = \frac{1}{x},\ r_1 = e^{\frac{1}{x}},\ q_2 = \frac{-1}{x+1},\ r_2 = e^{\frac{-1}{x+1}}$$

we can determine that q_2 is linearly independent of q_1. In fact, we need not check for a full linear dependence in our algorithm, but only for the possibility that a new exponent is a rational number multiple of some previous exponent. This is a consequence of the linear independence of our partial fraction decomposition. Our example, therefore, reduces to

$$\frac{r_2 r_1 + r_1 r_2}{r_1} = 2r_2 = 2e^{\frac{-1}{x+1}}$$

Unfortunately, partial fraction decomposition must be used with great care in higher order exponentials; for example,

$$e^{\frac{1}{e^x+1}} = e^{\frac{\frac{1}{3}}{e^{1/3x}+1}}\ e^{\frac{-\frac{1}{3} e^{\frac{1}{3}x} + \frac{2}{3}}{e^{2/3x} - e^{1/3x} + 1}}$$

Therefore, if we let $r_1 = e^x$, a partial fraction decomposition of

$$\frac{1}{r_1+1}$$

is simply

$$\frac{1}{r_1+1} ,$$

thus missing the possibility for a decomposition in terms of

$$r_1^{1/3}$$

which might be crucial in some expresion. We are confident, however, that a fairly efficient exptension of our algorithm to higher order exponentials will, in fact, be found.

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 involving the logarithm function. A result close in spirit to those of 3.3.1 was obtained by Richardson [37] 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 functions sine, cosine and $\log|x|$. The three functions in addition to the exponential function of REX expressions can be nested to any depth. Richardson was interested in the problem of determining whether an expression was equivalent to 0 on some interval of the real line.

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 which involves only constants is equal to 0. This requirement is, of course, 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 essence, this requirement is that no subexpression could become unbounded in value at some finite point on the interval being examined.

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 comples than e^x because it is of higher degree. The complexity measure does not presume that

$$e^{2x} \quad \text{and} \quad (e^x)^2$$

are algebraically related. In fact, it does not matter very much which expression 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

$$a_n(x)y^n + a_{n-1}(x)y^{n-1} + \ldots + a_0(x)$$

which is equivalent to 0 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

$$a_{n-1}(x)y^{n-1} + \ldots + a_0(x)$$

is of lower degree in y 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

$$y^n + \frac{a_{n-1}(x)}{a_n(x)}\ y^{n-1} + \ldots + \frac{a_0(x)}{a_n(x)}$$

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

$$ny^{n-1}y' + \ldots + \frac{a_n a_0' - a_0 a_n'}{a_n^2}$$

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 E2 is equivalent to a constant. To complete the algorithm we must determine if the constant is 0. This is the way in which the constant problem arises in Richardson's algorithm. One could attempt to evaluate the expression at a point as Oldehoeft does [33]. The situation here is simpler than in Oldehoeft's cases since if the function is equivalent to a constant we need not worry about accidental values of 0 arising in the evaluation.

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. As we pointed out earlier, the logarithmic case of the algorithm hinges on the fact that the derivative of a logarithmic term is of lower rank in complexity than the term itself. Thus functions which are defined by integrals such as the error function

$$\operatorname{erf}'(x) = \frac{2}{\sqrt{\pi}}\, e^{-x^2}$$

and the exponential integral

$$E_i'(x) = \frac{e^x}{x}\ ,$$

can be included in the expressions to be tested for equivalence to 0. The key property of the exponential function used in the algorithm is that the derivative of an exponential of degree k is also of degree k in that exponential. Consider rational roots of polynomials which have the form

$$P(x)^{n/m},$$

where P is a polynomial and n and m are integers. (E.g.,

$$\sqrt{x},\ (x^2+2)^{2/3}).$$

In general,

$$[P(x)^{n/m}]' = \frac{n}{m}\, P(x)^{n/m\,-\,1}\ P'(x)$$

$$= \frac{n}{m}\,\frac{P'(x)}{P(x)}\ [P(x)^{n/m}]$$

Since

$$\frac{n}{m}\,\frac{P'}{P}$$

is a rational function and of lowest complexity in Richardson's ranking, we can say that rational roots of polynomials will behave like exponentials in Richardson's algorithms.

Johnson [21], using a somewhat different approach at deciding equivalence, is also able to handle a large class of expressions like the one we have just indicated.

It can further be shown [32] that Richardson's algorithm can be extended to accept any function 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 may, however, 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

$$\frac{E'}{E} = Q(x,y).$$

Therefore, integrating both sides

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

$$E = C_2 e^{\int Q(x,y)\ dx} \text{ where } C_1,\ C_2 \text{ are constants.}$$

Exponentials usually cannot have a zero value. In such cases E can only have a zero value 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.

The canonical simplification algorithms represent an extreme in the use of explicit knowledge of the simplification rules of a class of expressions. Equivalence matching algorithms need not explicitly know these rules. For example, Richardson's algorithm does not explicitly know that $\log|ab| = \log|a| + \log|b|$. As a result, equivalence matching algorithms are usually poor simplification algorithms. It is clear that in many situations explicit knowledge of general simplification rules is valuable in reducing the size of an expression. In fact, we would also like to know that a given set of simplification rules for a class of expressions is complete in the sense that no other general rules can be found which cannot be derived from the set by rational operations. Risch tackled these questions for a class of expressions similar to Richardson's [40]. His results, in effect, state that no general rules exist other than the familiar ones (e.g.,

$$e^{a+b} = e^a e^b,\ e^{m/n \log a} = a^{m/n},\ \log e^a = a,$$

$$\log ab = \log a + \log b + 2k\pi i,$$

where k, m, and n are integers). The method of attack he uses is to ask what relationship must exist between an exponential or a logarithmic term and a set of other exponentials and logarithmics for the former to be algebraically dependent on the latter. The proof relies on much of the machinery used in Risch's previous work on integration. As before, Risch's results require the solution of constant problems.

Thus we may speak of three varieties of theoretical algorithms. Zero-equivalence algorithms will guarantee that expressions equivalent to 0 will be identified. Regular algorithms guarantee that the exponential and logarithmic terms in an expression are algebraically independent of each other. Regular algorithms are zero-equivalence algorithms. Canonical algorithms which reduce equivalent expressions to a single form are always zero-equivalence and are usually regular. Caviness' algorithm is an exception to this rule in that it is not regular.

3.3.3 Roots of Polynomials

In [6], 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;

$$\frac{1}{x^{1/2} + x^{1/3}},$$

$$\frac{(4-x)^{5/3}}{(x^2+2)^{2/3}}.$$

The expression

$$(x + 3^{1/2})^{1/3}$$

is not in this class because it involves nested exponentiation by non-integers. Caviness shows that there exists a zero-equivalence simplification algorithm for this class of expressions. The algorithm is not canonical. Unfortunately, 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 [26] 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 having polynomial coefficients (i.e., $y^2 - x$). 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 even roots of unity, has the same strong property as Caviness', namely the zero-equivalence property. However, 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

$$\sqrt{x^2-1}$$

into $\sqrt{x-1}\ \sqrt{x+1}$

if $\sqrt{x-1}$ or $\sqrt{x+1}$ occurred elsewhere in the expression. However,

$$\sqrt{x^2+2}$$

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 zero-equivalence property is obtained without resorting to additional conjectures. Fateman's algorithm will simplify

$$\frac{\sqrt{x^2-4}}{\sqrt{x+2}}$$

to

$$\sqrt{x-2}$$

and incidentally, it will convert $\sqrt{98}$ to $7\sqrt{2}$. The algorithm can be made canonical by removing radicals from denominators in quotients through a generalization of the process of "rationalizing the denominator." Thus

$$\frac{1}{x+\sqrt{2}}$$

could be converted to

$$\frac{x-\sqrt{2}}{x^2-2}$$

in order to achieve a canonical form.

3.3.4 Unsolvability Results

The earliest of the theoretical results which we discuss in this section, and probably the best known one in the field of algebraic manipulation, is a negative result due to Richardson [35]. In 1965 Richardson, Risch and Moses were all working on integration. Richardson and Moses were pursuing theorems stating that integration was unsolvable (i.e., that no algorithm existed for determining whether the integral did or did not exist in closed form), and Risch was examining algorithms for solving the problem [38,39]. Richardson succeeded in obtaining an unsolvability result for integration by showing that there exists a class E of expressions for which no algorithm exists for deciding whether each expression in E was equivalent to 0. If we consider the class of integrals

$$\int Re^{x^2}\,dx,$$

where R is some specially chosen member of E, then if R is equivalent to 0, the integral exists in closed form (in fact, it is a constant). If R is not equivalent to 0, then the integral cannot be expressed in closed form due to the well-known properties of

$$e^{x^2}$$

and those of the chosen members of E. As it turns out, Richardson's major result was the demonstration of the existence of the class E, rather than the application of this result to integration.

The starting point for most unsolvability results in algebraic manipulation is Hilbert's Tenth Problem. This problem, which is 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. In 1965, it was known that a version of the problem, called the Exponential Diophantine Problem, was unsolvable. Today, one can use the unsolvability of Hilbert's Tenth Problem to claim that there exists a polynomial $P(y,x_1,x_2, \ldots ,x_n)$ such that the question of whether $P = 0$ had integer solutions for $x_1,x_2, \ldots ,x_n$ for varying integral values of y could not be solved by an algorithm. One can generalize this problem to determine whether P had real roots by asking whether

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

since $\sin \pi x_i = 0$ forces each x_i to be an integer.

By manipulating the equation above, Richardson was able to show that there exists a function G(y,x) such that as y varies over the integers one can not tell whether there exist real values of x such that $G(y,x) < 0$.

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 G to a function F(y,x) such that $F(y,x) \equiv 0$ could not be determined by an algorithm as y varied over the integers. This is Richardson's major unsolvability result.

Several corollaries of Richardson's theorem were derived by Fenichel [14] to show that, among other things, one could not determine the limit of every expression which possessed a limit.

Risch [38] used the fact that

$$\log(e^x) = x + 2k\pi i ,$$

for some integer k, to generate an unsolvable integration problem by using Hilbert's Tenth Problem and Richardson's device of integrating a multiple of

$$e^{x^2}.$$

In [29], we used the fact that the differential equation

$$y' + y^2 = 1 + \frac{p(p+1)}{x^2} , \quad p \text{ a constant,}$$

has a solution which is a rational function in x if and only if p is an integer. Thus we were able to generate systems of ordinary differential equations for which one cannot decide whether they possess rational functions as solutions.

4.0 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. Designers have made substantial improvements to their systems just in the last year. Much of what we discussed in this paper, the conventional lexicographic ordering of expressions, the variety of substitution techniques, and canonical simplifiers, is only available in systems in an experimental nature. Therefore, any predictions about the future state of algebraic simplification are made on shaky grounds. Nonetheless, we shall attempt some predictions, albeit fairly conservative ones.

Practitioners in the field of algebraic manipulation, just as in much of computer science, can be divided into three major categories: theoreticians, systems designers, and users. Theoreticians tend to design radical and new left systems since the algorithms in such systems are most easily defined. Users with substantial problems to solve also tend to design radical and new left systems since the needs of such users are very special and frequently do not require the flexibility offered by liberal, conservative, or catholic systems. Systems designers expect a great variety of users and therefore tend to build systems with liberal or conservative components. 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 the efficiency of the algorithms provided in radical systems or subsystems. It would not be surprising if many radical systems mature into new left systems when the disadvantages 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. Risch's results about relationships of exponentials and logs of different arguments will probably be extended to a number of other functions.

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 will be possible for a user to define a function as a solution to a differential equation and then employ that function 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, hopefully, continue to improve their creations.

Acknowledgements

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

References

1) Abrahams, P.W., "Applications of LISP to Sequence Prediction," CACM, vol.9, no.8, p.551, Aug. 1966.

2) Blair, F. et al., "SCRATCHPAD/1: An Interactive Facility for Symbolic Mathematics," these proceedings.

3) Brown, W.S., et al., "The ALPAK System for Non-Numerical Algebra on a Digital Computer-II," Bell System Technical Journal, vol.XLIII, no.2, pp.785-804, March, 1964.

4) Brown, W.S., "Rational Exponential Expressions and a Conjecture Concerning π and e," Amer. Math. Monthly, vol.76, pp.28-34, Jan. 1969.

5) Brown, W.S., "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," these proceedings.

6) Caviness, B.F., "On Canonical Forms and Simplification," PhD disser., Carnegie-Mellon U., Pitts. Pa., Aug. 1967.

7) Caviness, B.F., "On Canonical Forms and Simplification," Journal of the ACM, vol.17, no.2, April 1970, pp.385-396.

8) Christensen, C., and Karr, M., "IAM, A System for Interactive Algebraic Manipulation," these proceedings.

9) Collins, G., "PM, A System for Polynomial Manipulation," Comm. ACM, vol.9, no.8, Aug. 1966, pp.578-589.

10) Collins, G., "The SAC-I System: An Introduction and Survey," these proceedings.

11) Collins, G., "Subresultants and Reduced Polynomial Remainder Sequences, " Journal of the ACM, vol.14, Jan. 1967, pp.128-142.

12) Engeli, M., "User's Manual for the Formula Manipulation Language SYMBAL," University of Texas at Austin, Computer Center, 1968.

13) Fateman, R., "The User-Level Semantic Matching Capability in MACSYMA," these proceedings.

14) Fenichel, R., "An On-Line System for Algebraic Manipulation," PhD. dissert., Harvard U., Cambridge, Mass., 1966.

15) Goldberg, S.H., "Solution of an Electrical Network Using a Digital Computer," M.S. Thesis, MIT, Cambridge, Mass.,1959.

16) Hall, A., "The ALTRAN System for Rational Function Manipulation - A Survey," these proceedings.

17) Hart, T., "Simplify," Memo.27, Artificial Intelligence Group, Project MAC, MIT, Cambridge, Mass., 1961.

18) Hearn, A., "REDUCE: A User-Oriented Interactive System for Algebraic Simplification," In Interactive Systems for Experimental Applied Mathematics, Academic Press, New York, pp.79-90.

19) Hearn, A., "The Problem of Substitution," in Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation, IBM, Cambridge, Mass., pp.3-19.

20) Hearn, A., "REDUCE 2: A System and Language for Algebraic Manipulation," these proceedings.

21) Johnson, S., "On the Problem of Recognizing Zero," these proceedings.

22) Korsvold, K., "An On-Line Algebraic Simplification Program," Artificial Intelligence Project Memo. no.37, Stanford University, Stanford, California, Nov. 1965.

23) Manove, M. et al., "Rational Functions in MATHLAB," in Symbol Manipulation Languages and Techniques, North-Holland, Amsterdam, pp.86-102.

24) Martin, W., "Symbolic Mathematical Laboratory, "Report MAC-TR-36. Project MAC, MIT, Cambridge, Mass., Jan. 1967.

25) Martin, W., "Determining the Equivalence of Algebraic Expressions by Hash Coding," these proceedings.

26) Martin, W., and Fateman, R., "The MACSYMA System," these proceedings.

27) Minsky, M., and Papert, S., Perceptrons, MIT Press, Cambridge, Mass., 1969.

28) Moses, J., "Solutions of Systems of Polynomial Equations by Elimination," Comm. of the ACM, vol.9, no.8, Aug. 1966, pp.634-637.

29) Moses, J., "Symbolic Integration," Report MAC-TR-47, Project MAC, MIT, Cambridge, Mass., Dec. 1967.

30) Moses, J., "A Canonical Form for First Order Exponential Expressions," in preparation.

31) Moses, J., "Sarge - A Program for Drilling Students in Freshman Calculus Integration Problems," Project MAC memo., March 1968.

32) 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.

33) Oldehoeft, A., "Analysis of Constructed Mathematical Responses by Numeric Tests for Equivalence," ACM National Conference Proceedings, pp.117-124, Aug. 1969.

34) Perlis, A., et al., "A Definition of Formula Algol," Computation Center, Carnegie-Mellon Univ., Pitts., Pa., March 1966.

35) Richardson, D., "Some Unsolvable Problems Involving Functions of a Real Variable," PhD dissert., Univ. of Bristol, Bristol, England, 1966.

36) Richardson, R., "Some Unsolvable Problems Involving Elementary Functions of a Real Variable," Journal of Symbolic Logic, vol.33, 1968, pp.511-520.

37) Richardson, R., "A Solution of the Identity Problem for Integral Exponential Functions," Z. Math Logik u. Grundlagen Math, to appear.

38) Risch, R., "The Problem of Integration in Finite Terms," Trans. of the AMS, vol.139, May 1969, pp.167-189.

39) Risch, R., "On the Integration of Elementary Functions Which are Built up Using Algebraic Operations," Report SP-2801-002, Systems Develop. Corp., Santa Monica, Calif., June 1968.

40) Risch, R., "Further Results on Elementary Functions," Report RC 2402, IBM Corp., Yorktown Heights, NY, March 1969.

41) Simon, H., The Sciences of the Artificial, MIT Press, Cambridge, Mass., 1969.

42) Slagle, J., "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus," PhD dissert., MIT, May 1961.

43) Tobey, R., "Experience with FORMAC Algorithm Design," Comm. of the ACM, vol.9, no.8, Aug. 1966, pp.589-597.

44) Tobey, R., et al. "PL/I-FORMAC Symbolic Mathematics Interpreter," IBM, Cambridge, Mass., 1969.

45) Tobey, R., et al., "Automatic Simplification in FORMAC," in Proc of Fall Joint Computer Conference, vol.27, 1965, pp.37-52.

46) van der Waerden, B., Modern Algebra, vol.I, F. Ungar, New York, pp.77-78.

47) Wooldridge, D., "An Algebraic Simplify Program in LISP," Art. Intell. Project Memo. no.11, Stanford Univ., Stanford, Calif., Dec. 1963.

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

Let us call the array $\begin{vmatrix} AB \\ CD \end{vmatrix}$ a, the array $\begin{vmatrix} MN \\ OP \end{vmatrix}$ m, the array $\begin{vmatrix} RS \\ TU \end{vmatrix}$ r, and the array $\begin{vmatrix} HI \\ JK \end{vmatrix}$ h. Let us call the array $\begin{vmatrix} am \\ ma \end{vmatrix}$ w, and the array $\begin{vmatrix} rh \\ hr \end{vmatrix}$ x. Then the entire array is simply $\begin{vmatrix} wx \\ xw \end{vmatrix}$. While the original structure consisted of 64 symbols, it requires only 35 to write down its description:

$$S = \begin{matrix} wx \\ xw \end{matrix}$$

$$w = \begin{matrix} am \\ ma \end{matrix} \qquad x = \begin{matrix} rh \\ hr \end{matrix}$$

$$a = \begin{matrix} AB \\ CD \end{matrix} \quad m = \begin{matrix} MN \\ OP \end{matrix} \quad r = \begin{matrix} RS \\ TU \end{matrix} \quad h = \begin{matrix} HI \\ JK \end{matrix}$$

FIGURE 1.

```
PQ = M**2 - PROP1/2,

PR = QR + RT - RS,

PS = QS + RT - PROP1/2,

PT = QS - PR + RT,

QS = M**2 - PROP3/2,

QT = PS - QR - RT,

PROP2 = PROP1 - 2*RT + 2*RS
```

(b.) Relations Between Variables

```
((4*M**4 - (PROP1+PROP3)**2)*(- 2*M**2*QR - 4*QR*RT
       + 2*RT**2 - RT*(PROP1+PROP3)+(PR*PROP1+RS*PROP3)
       + 2*M**2*PR*RS/RT)
+ 4*M**2*QR*(PR + RS)*(2*M**2 + RT + (PROP1+PROP3))
+ 2*M**2*PR*RS*(2*QR - 6*RT - 3*(PROP1+PROP3))
+ 2*(QR - RT)*((PR*PROP1+RS*PROP3)*(M**2 - (PROP1+PROP3))
       + 2*QR*RT*(PROP1+PROP3))
+ 2*(QR**2 + RT**2)*(2*QR*RT - (PR*PROP1+RS*PROP3)
       + RT*(PROP1+PROP3)) + 6*M**2*RT**2*(PROP1+PROP3))
/ ( 4*PROP1*PROP3*RT*PR*RS)
```

(c.) Final Result Produced by Man and Machine

Figure 2

```
(        M**4 *

(2 * PROP1 * PR * RS    -    2 * PROP1 * PR * RT    -    4 * PR**2 * RS
 -   4 * PR**2 * RT    -    4 * PR * RS**2    +    14 * PR * RS * RT    +
  2 * PR * RS * PROP2    -    4 * PR * RS * PS    -    4 * PR * RS * PT    -
   4 * PR * RS * QT    -    4 * PR * RS * QS    -    4 * PR * RS * QR    -
  10 * PR * RT**2    -    2 * PR * RT * PROP2    +    4 * PR * RT * PS    +
   4 * PR * RT * PT    +    4 * PR * RT * QT    +    4 * PR * RT * QS    -
  4 * PR * RT * QR    -    6 * RS**2 * RT    -    4 * RS * RT**2    -    6 *
 RS * RT * QR    -    6 * RT**3    +    6 * RT**2 * QR)


   +    M**2 *

(    -    PROP1 * PR * RS * RT    +    PROP1 * PR * RT**2    +    PROP1 * P
R * RT * PROP2    +    PROP1 * RS**2 * RT    +    2 * PROP1 * RS * RT**2
 -   2 * PROP1 * RS * RT * PT    +    PROP1 * RT**3    +    2 * PROP1 * RT
**2 * PS    +    6 * PR**2 * RT * QT    -    2 * PR**2 * RT * QS    +    4 *
 PR**2 * RT * QR    -    4 * PR * RS * RT * PROP2    +    4 * PR * RS * RT
 * PS    +    8 * PR * RS * RT * PT    +    4 * PR * RS * RT * QT    +    2
* PR * RS * RT * QS    -    4 * PR * RS * RT * QR    +    8 * PR * RS * PS
 * QT    +    8 * PR * RS * PS * QR    -    4 * PR * RT**3    +    2 * PR *
RT**2 * PROP2    +    4 * PR * RT**2 * PT    +    6 * PR * RT**2 * QT    -
  4 * PR * RT**2 * QS    +    PR * RT * PROP2**2    -    2 * PR * RT * PRO
P2 * PS    -    2 * PR * RT * PROP2 * PT    -    2 * PR * RT * PROP2 * QT
 -    2 * PR * RT * PROP2 * QS    -    8 * PR * RT * PS * QT    -    2 * P
R * RT * PS * QR    -    2 * PR * RT * PT * QR    +    2 * RS**2 * RT * QT
  +    4 * RS**2 * RT * QR    -    4 * RS * RT**3    -    2 * RS * RT**2 *
 PROP2    +    4 * RS * RT**2 * PS    -    4 * RS * RT**2 * PT    +    6 * R
S * RT**2 * QT    -    2 * RS * RT**2 * QS    -    2 * RS * RT * PROP2 * P
T    +    RS * RT * PROP2 * QR    +    4 * RS * RT * PS * PT    +    2 * RS
* RT * PS * QR    +    4 * RS * RT * PT**2    +    4 * RS * RT * PT * QT
 +    4 * RS * RT * PT * QS    +    4 * RT**3 * PS    -    2 * RT**3 * QS
 +    4 * RT**3 * QR    +    2 * RT**2 * PROP2 * PS    -    RT**2 * PROP2 *
 QR    -    4 * RT**2 * PS**2    -    4 * RT**2 * PS * PT    -    4 * RT**2
* PS * QT    -    4 * RT**2 * PS * QS    +    4 * RT**2 * PS * QR    +    2
* RT**2 * PT * QR)


  -    2 * PROP1 * RS * RT**2 * PS    -    2 * PR**2 * RT * PROP2 * QT
 +    8 * PR * RS * RT**2 * QT    +    2 * PR * RS * RT * PROP2 * QT    -
   8 * PR * RS * RT * PS * QT    -    8 * PR * RS * RT * PS * QR    -    4
 * PR * RS * RT * PT * QT    -    4 * PR * RT**2 * PS * QT    +    4 * PR
* RT**2 * PS * QS    +    4 * PR * RT * PROP2 * PS * QT    +    2 * PR * R
T * PROP2 * PS * QR    +    4 * RS**2 * RT * PT * QT    -    4 * RS * RT**
2 * PS * QT    -    8 * RS * RT * PS * PT * QT    -    4 * RS * RT * PS *
PT * QR    +    8 * RT**2 * PS**2 * QT) /


 (  -    4 * PROP1 * PR * RS * RT**2 * PROP3)
```

(a.) Expression Initially Produced by Computer

Figure 2. Example of Reducing the Size of Output Expressions by Substitution

A	B	C	Alternative Results
a	xy^2	x^2y^3	axy, a^2/y, x^2y^3
a	$2x + 3y$	$3x + 4y + 1$	$3x + 4y + 1$, $\frac{x}{3} + \frac{4}{3}a + 1$, $-\frac{y}{2} + \frac{3}{2}a + 1$
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	$s^2 + c^2$	$\frac{s^3 - s}{c^2}$	$-s$

Figure 3.