(2) and its derivative R' are polynomials which arose naturally in a recent application of PM, where it was desired to find whether R has any multiple factors. Both pairs of polynomials turn out to be relatively prime. Using the old algorithm, the computation (1) required 0.64 seconds; using the new algorithm, it required 0.22 seconds. The advantage of the new algorithm increases rapidly with the complexity of the polynomials to which it is applied. It did the computation (2) in 0.30 minutes, while the old algorithm required 21.11 minutes!

REFERENCES

1. Collins, G. E. A method for overlapping and erasure of lists. *Comm. ACM 3* (Dec. 1960), 655–657.
2. ——. REFCO III, a reference count list processing system for the IBM 7094. IBM Res. Rept. RC-1436, May, 1965.
3. McCarthy, J., et al. LISP I programmer's manual. Computation Center and Res. Lab. of Electronics, MIT, Cambridge, Mass., 1960.
4. Weizenbaum, J. Symmetric list processor. *Comm. ACM 6* (Sept. 1963), 524–543.
5. Glasner, Judith, et al. The NU-SPEAK system. NYO-1480-9, Courant Inst. of Mathematical Sciences, New York University, New York, N.Y., Nov. 1964.
6. Pope, David A., and Stein, Marvin L. Multiple precision arithmetic. *Comm. ACM 3* (Dec. 1960), 652–654.
7. Cox, Albert G., and Luther, H. A. A note on multiple precision arithmetic. *Comm. ACM 4* (Aug. 1961), 353.
8. Brown, W. S. The ALPAK system for non-numerical algebra on a digital computer—I: Polynomials in several variables and truncated power series with polynomial coefficients. *Bell Sys. Tech. J. 42* (Sept. 1963), 2081–2119.
9. ——, Hyde, J. P., and Tague, B. A. The ALPAK system for non-numerical algebra on a digital computer—II: Rational functions of several variables and truncated power series with rational function coefficients. *Bell Sys. Tech. J. 43* (March 1964), 785–804.
10. Hyde, J. P. The ALPAK system for non-numerical algebra on a digital computer—III: systems of linear equations and a class of side relations. *Bell Sys. Tech. J.*, 43, (July, 1964), 1547–1562.
11. Tarski, A. *A Decision Method for Elementary Algebra and Geometry*. U. of California Press, Berkeley, Calif., 2nd ed.
12. Collins, G. E. Polynomial remainder sequences and determinants. IBM Res. Rept. RC-1209, June, 1964. Also *Am. Math. Month.* to be published.
13. ——. Subresultants and reduced polynomial remainder sequences. *Notices of the Am. Math. Soc.*, to appear.
14. Uspensky, J. V., and Heaslet, M. A. *Elementary Number Theory*. McGraw Hill Co., New York, N. Y., 1939, pp. 43–45.

Experience with FORMAC Algorithm Design

R. G. Tobey

International Business Machines Corporation, Cambridge, Massachusetts*

Various facets of the design and implementation of mathematical expression manipulation algorithms are discussed. Concrete examples are provided by the FORMAC EXPAND and differentiation algorithms, a basic FORMAC utility routine, and an experiment in the extraction of the skeletal structure of an expression. One recurrent theme is the need to avoid excessive intermediate expression swell in order to minimize core storage requirements. Although many details from the FORMAC implementation are presented, an attempt is made to stress principles and ideas of general relevance in the design of algorithms for manipulating mathematical expressions.

Introduction

Shortly after the Formac experimental programming system first became operational in April, 1964, a small group of programmers and mathematicians began to experiment with the Formac object-time routines. One explicit goal was to seek improvements to particular Formac expression manipulation algorithms; another less provincial goal was to isolate and study general or theoretical problems—problems independent of a particular internal representation or implementation—which arise in the design of mathematical expression manipulation algorithms. Several highlights of this experimentation, which led to the improvement of Formac algorithms, and which have implication for algorithm design of a more general nature, are presented in this paper.

The Formac capability is described in increasing amounts of detail in [1, 2, 3]. Some details of Formac implementation are presented in [4]. The role of automatic simplification in Formac is sketched in detail in [5], and applications that have been made of the Formac system are described in [6]. Some familiarity with these papers is assumed in the discussion which follows.

This paper is divided into four sections. In the first section, problems encountered in design of expansion algorithms (the Formac EXPAND command) are discussed. The second section is devoted to alternative organizations of the differentiation algorithm. Examples contrast the output produced by the two different routines. The third section discusses a simple-minded idea concerning expression scanning and error checking; it leads to an order of magnitude increase in the efficiency of the Formac system. The last section deals with the problem of incomprehensible mathematical expressions; an experiment in

Presented at an ACM Symposium on Symbolic and Algebraic Manipulation, Washington, D.C., March 29–31, 1966.

* Systems Development Division.

the extraction of the skeletal structure of an expression is described. One implication is that certain techniques can be readily implemented for reducing enormous expressions to a comprehensible form.

Expansion Algorithms and Intermediate Swell

Expansion, the algebraic operation of multiplying out a product of sums, is a transformation of central importance in the manipulation of mathematical expressions. In Figure 1 three forms for p_3, the third Legendre polynomial, are displayed. The first form indicates how the polynomial looks when generated from the basic iterative relation (also displayed in Figure 1) without benefit of expansion.

$$p_1 = x$$

$$p_n = np_{n-1} + \frac{x^2 - 1}{n} \frac{d}{dx} [p_{n-1}]$$

$$p_3 = x(x^2 + \tfrac{1}{2}(x^2 - 1)) + x(x^2 - 1)$$
$$= x^3 + \tfrac{1}{2}x^3 - \tfrac{1}{2}x + x^3 - x$$
$$= \tfrac{5}{2}x^3 - \tfrac{3}{2}x$$

FIG. 1. Expansion and collapsing of the Legendre polynomial, p_3.

The second form for the polynomial is that obtained by expansion without collection of like terms. The third form has been obtained from the second by collection of like terms. This example indicates that the unexpanded form of a polynomial may require more space for its representation than the expanded form and much more space than the simplified form.

In some symbolic computations the expanded form—prior to the collection of like terms—requires a great deal more space than either the unexpanded form or the completely simplified form for the expression. Such is the case with the symbolic calculation of f and g series as reported in [7]. Figure 2 displays the space requirements for the 4th through the 11th coefficients, the f_i, of the f series. Four sets of data are displayed. The first ($\cdot$) is the size of the expression at input to the EXPAND command. The next two ($\emptyset$ +) represent the maximum space required by the expression as a result of the EXPAND algorithm. The fourth data curve ($\times$) gives the size of the expression after expansion and simplification. It is obvious from this display that expansion requires more space for expression representation before it requires less space. We call this phenomena *intermediate expression swell*. In the design of algorithms for symbolic expression manipulation, it is important to minimize intermediate expression swell. The need for minimization is apparent from Figure 2; if only 1000 units of memory space are available for expression manipulation, then execution of the program utilizing the original EXPAND algorithm would cease during the generation of the 10th iterate. F_{10} requires more than 1000 units of core storage for its generation due to intermediate expression swell. In this section the partial redesign of the FORMAC EXPAND algorithm to minimize intermediate expression swell is discussed.

Expansion of Products of Sums

The original FORMAC EXPAND algorithm (operational in April, 1964) generated all the terms in the expanded sum before attempting to collect like terms. This brute force approach to the organization of expansion led to the enormous intermediate swell indicated in Figure 2. The frequent exhaustion of work space while expanding expressions made it obvious that it was necessary to reorganize the communication between the EXPAND algorithm and AUTSIM—the FORMAC AUTomatic SIMplification routine [5]. The EXPAND routine was reorganized so that the EXPAND transformation was driven by the AUTSIM transfer table. This reorganization cut down intermediate swell as is indicated by the + data points in Figure 2. As is to be expected in the organization of computer algorithms, this decrease in space requirement was bought with execution time. In Figure 3, expression size is plotted against execution time in seconds for f_{11}. The solid line indicates expression size under the original EXPAND algorithm. Note that the time at

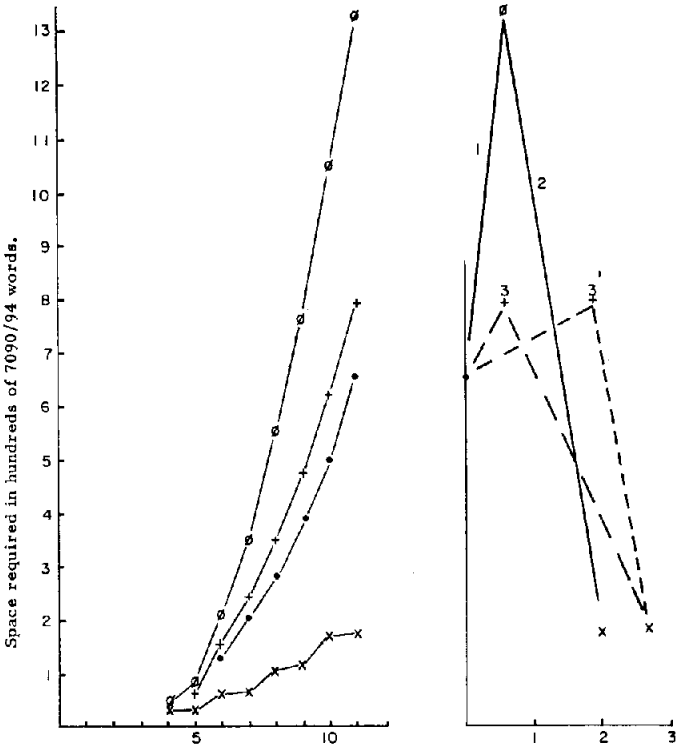

FIG. 2. Space required for 4th through 11th iterates

FIG. 3. Expansion of 11th iterate (execution time in seconds) for both algorithms

$\emptyset$ intermediate swell produced by first EXPAND algorithm.
+ intermediate swell produced by improved EXPAND (type III release).
. size of expression at input to EXPAND.
$\times$ size of expression after expansion and simplification.

which the maximum space is required is known exactly. The portion of this curve in which the space requirement for expansion is increasing (1) corresponds to the time in the EXPAND algorithm. The portion of the curve in which the space requirement is decreasing (2) corresponds to the time in the AUTSIM algorithm. The two slashed lines (3 and 3′) indicate a maximum and minimum expansion AUTSIM interaction. Since EXPAND and AUTSIM are interconnected, there is no meaningful separation of the two algorithms on these curves.

Figure 4 displays the conceptual difference between the two expansion algorithms. Figure 4A indicates how expansion and simplification are performed by the original EXPAND algorithm. Figure 4B indicates the approach taken in the improved expansion algorithm. The meshing of the EXPAND algorithm with the AUTSIM algorithm gives the EXPAND algorithm immediate access to LEXICO—short for LEXICOgraphical reordering (see [5])—the routine that collects like terms in a sum. As each term of the expanded product is generated, it is added to the intermediate results and an attempt is made, via LEXICO, to collect like terms in that sum. The algorithm bounces back and forth between term generation and collection of like terms in the sum. In this manner, intermediate expression swell is minimized.

$$(a + b - c)(a - b + c)$$

$$\xrightarrow{\text{EXPAND}} (a + b - c)a + (a + b - c)(-b) + (a + b - c)c$$

$$\xrightarrow{\text{EXPAND}} a^2 + ab - ac - ab - b^2 + bc + ac + bc - c^2$$

$$\xrightarrow[\text{SIM}]{\text{AUT}} a^2 - b^2 - c^2 + 2bc$$

(A) No intermediate collection of like terms (9 terms maximum swell)

$$\xrightarrow[\text{Generator}]{\text{Term}} a^2 + ab \xrightarrow{L} ab + a^2$$

$$\xrightarrow[\text{Generator}]{\text{Term}} ab + a^2 - ac \xrightarrow{L} ab - ac + a^2$$

$$\xrightarrow[\text{Generator}]{\text{Term}} ab - ac + a^2 - ab \xrightarrow{L} - ac + a^2$$

$$\xrightarrow[\text{Generator}]{\text{Term}} - ac + a^2 - b^2 \xrightarrow{L} - ac + a^2 - b^2$$

$$\xrightarrow[\text{Generator}]{\text{Term}} - ac + a^2 - b^2 + bc \xrightarrow{L} -ac + a^2 + bc - b^2$$

$$\xrightarrow[\text{Generator}]{\text{Term}} - ac + a^2 + bc - b^2 + ac$$

$$\xrightarrow{L} a^2 + bc - b^2 \xrightarrow[\text{Generator}]{\text{Term}} a^2 + bc - b^2 + bc \xrightarrow{L} a^2 + bc2 - b^2$$

$$\xrightarrow[\text{Generator}]{\text{Term}} a^2 + bc2 - b^2 - c^2 \xrightarrow{L} a^2 + bc2 - b^2 - c^2$$

(B) Intermediate collection of like terms while terms are generated (5 terms maximum swell). The LEXICO sorting order for expressions is described in [5].

FIG. 4. Contrast between the two expansion algorithms

There is an additional contrast between the two EXPAND algorithms. As is obvious from Figure 4A, the original EXPAND routine depended upon successive applications of the right distributive law; the intermediate expressions displayed in Figure 4A are actually generated as part of the expansion. The improved EXPAND algorithm generates terms directly by a nested iteration. It is possible to contrast the two EXPAND algorithms by saying that the original EXPAND algorithm only knew the right distributive law and applied it successively, whereas the improved algorithm is aware of the general theorem implied by the distributive law and applies that theorem directly.

Multinomial Expansion

If the algorithm described above were applied to a sum raised to an integral power in order to accomplish the expansion, then growth inefficiencies with respect to intermediate swell would occur. Consider the expression $(A_0 + A_1 + \cdots + A_n)^k$ where n and k are positive integers. The EXPAND algorithm described above would generate $(n + 1)^k$ distinct terms if applied to this expression. However, direct application of the generalized multinomial theorem will produce $\binom{n+k}{n}$ distinct terms. This is, in general, much smaller than $(n + 1)^k$. The algorithm implied by the multinomial theorem has one further advantage: if each of the variables which appear as terms in the above sum are distinct, then there can be no collapsing (or collecting) of like terms in the sum generated by the multinomial theorem. Hence, the multinomial theorem is a valuable tool in avoiding the problem of intermediate swell as encountered in the ordinary expansion algorithm. Suppose, however, that $A_i = a_i x^i$ with the a_i numeric. Then like terms will not have been collected in the sum produced by multinomial expansion. In fact, this case of a polynomial in a single variable is the worst possible case with respect to the contrast between the size of the expression after multinomial expansion and the size of the expression after the collection of like terms. In this particular case, the simplified polynomial will consist of $n \cdot k + 1$ terms. This is considerably smaller than $\binom{n+k}{n}$ terms. If the sum of the A_i is a polynomial in several variables, the contrast will not be so great.

In order to make concrete the contrast in the number of terms required, and hence the space required, in these various cases, consider the 5th degree polynomial $g(x) = 2 + 3x + 19x^2 + 7x^3 + 5x^4 + x^5$ in a single variable. Raise it to the 6th power, $h(x) = [g(x)]^6$. If the original EXPAND algorithm were applied, $h(x)$ would require $6^6 = 46656$ terms for the intermediate result. With the improved EXPAND algorithm, this would require much less space but would greatly increase the execution time. Straightforward multinomial expansion will generate $\binom{11}{5} = 462$ terms. But the simplified polynomial requires only

$5 \cdot 6 + 1 = 31$ terms. With respect to minimization of intermediate expression swell, multinomial expansion occupies middle ground; in one extreme case it provides the optimal result; in the other extreme it is highly inefficient.

The EXPAND algorithm currently operational within the FORMAC system performs multinomial expansion; however, this section of the algorithm has no contact with LEXICO until all the terms have been generated. Users of the experimental FORMAC system can expect to encounter great inefficiency in space utilization when raising polynomials to a power.

If one takes the basic unit of core storage to be that storage required to represent a term of a sum (this is admittedly variable and depends upon the nature of the term), it becomes obvious that the above remarks apply to the organization of EXPAND algorithms for any expression manipulation system. The concept of intermediate expression swell is independent of efficiencies in representation which may accrue due to a limited or well-structured data base; e.g., it applies equally well to polynomial manipulation systems which can make efficient use of storage space since the polynomial data structure is assumed. It is clear that disastrous intermediate swell while expanding expressions can be avoided by designing an expansion algorithm that utilizes the multinomial theorem and collects like terms simultaneously with the generation of those terms. How one organizes the communication between the term-generation function and the collection of like terms, so as to minimize execution time, is an intriguing problem which will, no doubt, be the subject of further study.

Differentiation in FORMAC

The two expressions displayed in Figure 5 are the third iterates generated by the FORMAC program cited in [7].

Original Differentiation Algorithm:

$$f_3 = ((0 - (((- (0\mu\sigma + 3(-3\mu\sigma)\sigma + 3\mu(\epsilon - 2\sigma^2)))0 + (-3\mu\sigma)0) + ((-3\mu\sigma)0 + \mu 0))) - ((-3\mu\sigma)(0 + 1) + \mu(0 + 0))) - \mu((0 + 0) + (0 - \mu 0))$$

Revised Algorithm:

$$f_3 = ((-(((-(3(-3\mu\sigma)\sigma + 3\mu(\epsilon - 2\sigma^2)))0))) - ((-3\mu\sigma)(0 + 1))) - \mu(0 + (0 - \mu 0))$$

The expression simplifies to $3\mu\sigma$.

FIG. 5. Contrast between differentiation algorithms

They have been generated without the benefit of AUTSIM, the FORMAC automatic simplification routine [5]. The first expression is the form generated by the original FORMAC differentiation program. The second expression was generated by a redesigned differentiation algorithm that does not generate as many redundant elements—the majority of the redundancies in this expression arise from sources other than the differentiation algorithm. The redundancies that were eliminated by the redesign are apparent from a careful comparison of the two expressions in Figure 5.

Hanson, Caviness and Joseph, in their work at the University of North Carolina [8], recognized that their method of generating derivatives introduced a large number of redundancies that required elimination. They observed that there were two ways to accomplish this: (1) Perform a second scan over the differentiated expression to clean up the redundancies, or (2) eliminate the redundancies as they are generated. They chose the second alternative. The original FORMAC differentiation algorithm implemented the first alternative with AUTSIM performing the second scan. The new FMCDIF subroutine implements a third alternative: (3) Perform a preliminary scan over the input expression so that dependency relations are known during the main scan; few redundant expression

TABLE I. TRANSFORMATIONS IN ORIGINAL DIFFERENTIATION ALGORITHM

1. $D - \lambda \rightarrow - D\lambda$
2. $D \exp \lambda \rightarrow * \exp \lambda\ D\lambda\ ]$
3. $D \log \lambda \rightarrow * \uparrow \lambda - 1\ D\lambda\ ]$
4. $D \sin \lambda \rightarrow * \cos \lambda\ D\lambda\ ]$
5. $D \cos \lambda \rightarrow - * \sin \lambda\ D\lambda\ ]$
6. $D \operatorname{atan} \lambda \rightarrow * \uparrow + 1 \uparrow \lambda\ 2\ ] - 1\ D\lambda\ ]$
7. $D \tanh \lambda \rightarrow * 4\ \uparrow + \exp \lambda \exp - \lambda\ ] - 2\ D\lambda\ ]$
8. $D \operatorname{fac} \lambda \rightarrow 0$
   (The gamma function representation for the factorial is not employed. The factorial function is assumed discrete.)
9. $D \operatorname{dfc} \lambda \rightarrow 0$
10. $D\ \uparrow \Delta \lambda \rightarrow + *\ \uparrow \Delta + \lambda - 1\ ]\ D\Delta\lambda\ ] * \log \Delta\ D\lambda\ \uparrow \Delta \lambda\ ]\ ]$
11. $D \operatorname{comb} \Delta \lambda \rightarrow 0$
12. $D + \lambda_1\lambda_2 \cdots \lambda_s\ ] \rightarrow + D\lambda_1\ D\lambda_2 \cdots D\lambda_s\ ]$
13. $D * \lambda_1\lambda_2 \cdots \lambda_t\ ] \rightarrow + * D\lambda_1\lambda_2 \cdots \lambda_t\ ] * \lambda_1 D\lambda_2 \cdots \lambda_t\ ] \cdots * \lambda_1\lambda_2 \cdots D\lambda_t\ ]\ ]$
14. If $\lambda \in A$, $D\lambda \xrightarrow{x} 1$ for $\lambda = x$.
    $D\lambda \xrightarrow{x} \frac{d\lambda}{dx}$ for λ dependent upon x (declared so in the FORMAC DEPEND statement).
    $D\lambda \xrightarrow{x} 0$ for λ independent of x.

TABLE II. TRANSFORMATIONS IN REVISED ALGORITHM

0′ Given $D\lambda$, $\lambda \in B$. If lead element of λ is independent of variable of differentiation, then $D\lambda \rightarrow 0$.
Otherwise, apply the application transformation from the remainder of table.
See transformations 1–7 of Table I.
Dependency scan eliminates need for transformations 8, 9 and 11.

10′ $D \dot{\uparrow} \dot{\Delta}\ \dot{\lambda} \rightarrow$ same as 10
$D \dot{\uparrow} \dot{\Delta}\ \dot{\lambda} \rightarrow * \uparrow \Delta + \lambda - 1\ ]\ D\dot{\Delta}\lambda\ ]$
$D \dot{\uparrow} \dot{\Delta}\ \dot{\lambda} \rightarrow * \log \Delta D\dot{\lambda}\ \uparrow \Delta\lambda\ ]$.

12′ $D + \lambda, \cdots \lambda_t\ ] \rightarrow + D\lambda_{i_1} \cdots D\lambda_{i_s}\ ]$ where only those λ_{i_j} occur which are dependent upon the variable of differentiation.

13′ $D * \lambda, \cdots \lambda_t\ ] \rightarrow + \cdots * \lambda \cdots D\lambda_{i_1} \cdots \lambda_t\ ] \cdots\ ]$ where products are generated only for λ_i which are dependent upon the variable of differentiation.

14′ If $\lambda \in A$, $D\dot{\lambda} \xrightarrow{x} 1$ for $\lambda = x$; $D\dot{\lambda} \xrightarrow{x} \frac{d\lambda}{dx}$ (the symbolic name of the derivative) for $\lambda \neq x$.

elements are generated. The design of the first FORMAC algorithm was simplified by the assumption that AUTSIM would clean up expressions—remove redundant elements —after the differentiation transformations had been performed. As is clear from a study of Tables I and II, this assumption eliminated the necessity to design transformations for specific subcases of the differentiation transformations of Table I. Several such subcases are handled by specific transformations in the redesigned algorithm (Table II). In this section we discuss the organization of both differentiation routines and indicate how sources of redundancy in expressions have been eliminated.

The elimination of redundant expression elements decreases intermediate expression swell. This is very significant for differentiation algorithms since symbolic differentiation generates deceptively complicated expressions quite rapidly. Reference [9] provides interesting documentation of this phenomena.

THE FORMAC ALGORITHM

The structure of the FORMAC differentiation routines is most easily presented by considering mathematical expressions in the FORMAC internal representation, delimiter Polish (see [5]). For our purposes, it is sufficient to define delimiter Polish as a formal mathematical system. The two differentiation algorithms are then defined in terms of formal operations upon elements of this formal system.

Let A be the set of primitive elements (constants and variables) in this system. An elementary function will be defined as a function that can be represented by an expression in the set B of expressions generated from the primitive elements by the rules listed below.

Basic operators:

Unary operators: $-$, exp, log, sin, cos, atan, tanh, fac, dfc
Binary operators: $\uparrow$ (**), comb
Variary operators: $+$, $*$
In addition, there is a delimiter,] .

Rules for generation of expressions in B:

1. If $\lambda \in A, \lambda \in B$;
2. Let $\eta \in B$. Then $-\eta$, exp η, log η, sin η, cos η, atan η, tanh η, fac η, dfc η are in B.
3. Let $\eta, \lambda \in B$. Then $\uparrow \eta\lambda$ and comb $\eta\lambda$ are in B.
4. Let $\eta_1, \cdots, \eta_s \in B$. Then $+\eta_1\eta_2 \cdots \eta_s]$ and $* \eta_1\eta_2 \cdots \eta_s]$ are in B.
5. These are the only expressions in B.

Sample expressions in B:

Infix Notation	*Deliminter Polish*
$A - B + C$	$+ A - BC]$;
$(\cos (y))^{2\sin(x-y)}$	$\uparrow \cos y * \sin + x - y]2]$.
A/B	$* A \uparrow B - 1]$

First Algorithm

Let D be the differential operator; it is unary. We define C as the set of expressions generated by the rules for B with the addition of the following rule:

6. If $\lambda \in B, D\lambda \in C$.

The transformations of Table I define an algorithm for differentiating any λ in the set B. Each transformation of Table I suppresses the operator D to a point further to the right in the delimiter Polish expression. Successive application of these transformations suppresses the occurrences of D so that it operates only on primitive elements; and finally, application of Rule 14 eliminates all occurrences of D from the expression. Such iterative application of these transformations is itself a transformation from the set C to the set B. The transform is a symbolic form for the derivative of the original operand of the operator D.

The transformations presented in Table I are sufficient to provide a differentiation algorithm. The organization of the expression scan to implement such an algorithm is clear. A simple left-to-right scan, which applies the transformations of Table I to suppress and hence remove all occurrences of the operator D, is all that is required. The results, however, contain redundant subexpressions—automatic simplification of the expression is required. Several illustrative examples are displayed below. In these examples, the variable appearing over the arrow is the variable of differentiation.

(a) $D \uparrow X\,2 \xrightarrow{x} + * \uparrow X + 2 - 1]\,D\,X\,2] * \log X\,D\,2 \uparrow X\,2]]$
$\xrightarrow{x} + * \uparrow X + 2 - 1]\,1\,2] * \log X\,0 \uparrow X\,2]]$
$\xrightarrow[\text{SIM}]{\text{AUT}} * X\,2]$

(b) $D \tanh 1 \xrightarrow{v} * 4 \uparrow + \exp 1 \exp - 1] - 2\,D\,1]$
$\xrightarrow{v} * 4 \uparrow + \exp 1 \exp - 1] - 2\,0]$
$\xrightarrow[\text{SIM}]{\text{AUT}} 0$

(c) $D * X\,Y\,V\,3] \xrightarrow{v} + * D\,X\,Y\,V\,3] * X\,D\,Y\,V\,3]$
$* X\,Y\,D\,V\,3] * X\,Y\,V\,D\,3]]$
$\xrightarrow{v} + * 0\,Y\,V\,3] * X\,0\,V\,3] * X\,Y\,1\,3] * X\,Y\,V\,0]]$
$\xrightarrow[\text{SIM}]{\text{AUT}} * X\,Y\,3].$

The above examples are extreme in that the action of AUTSIM produces a dramatic reduction in the size of the expression. They illustrate, however, the extent to which the original differentiation routine depended upon automatic simplification to clean up its tracks. It is obvious that expressions were generated with unnecessarily large intermediate swell.

Second Algorithm:

The redesign of the differentiation algorithm includes a preliminary scan, performed to determine which subexpressions are dependent upon the variable of differentiation. As a result, one can determine immediately whether a given operator heads an expression that is dependent or independent of the variable of differentiation. This information eliminates the need to generate such horrendous expressions as displayed in example (b) above; it is known immediately that the tanh operator heads an expression that is independent of the variable of differentiation.

The new set of transformations is displayed in Table II. The first transformation indicates an alternative to any of the first thirteen transformations in Table I. The

alternative is invoked only in the case of nondependency. The changes to the other transformations are self-explanatory. The initial dependency scan utilizes a pushdown in order to scan into the expression, recognize dependency relationships, and back out of the expression marking the operators that have dependent subexpressions.

The effect of these changes to the differentiation algorithm can be seen by looking at the results of the following three examples. The input expressions are displayed after the dependency scan, on the left. The dot above the various symbols indicates that the dependency bit on these operators or variables has been set.

(a) $D \dot{\uparrow} \dot{X}\ 2 \xrightarrow{x} * \uparrow X + 2 - 1]\ 1\ 2]$
$\xrightarrow[\text{SIM}]{\text{AUT}} * X\ 2]$

(b) $D \tanh 1 \rightarrow 0$

(c) $D \dot{*} X\ Y\ \dot{V}\ 3] \xrightarrow{v} + * X\ Y\ 1\ 3]]$
$\xrightarrow[\text{SIM}]{\text{AUT}} * X\ Y\ 3].$

Although some expression redundancies still remain to be cleaned up by the AUTSIM routine, intermediate expression swell has been greatly reduced by this revised algorithm. Moreover, the new algorithm represents about a 4 percent decrease in execution time requirements; it is slightly more efficient than the old algorithm.

Subexpression Scan-Off Algorithm in FORMAC

Several FORMAC object-time algorithms require a basic operation that is performed by the routine FMCSEX. It is frequently necessary to scan over complete subexpression elements embedded in the expression currently being manipulated. This task may be performed to obtain the beginning and end of an expression so that it may be deleted or repositioned, or simply to scan over the expression so that the next argument of the governing operator (see [5]) may be scanned. For example, consider the delimiter Polish expression $+ \underline{* A\ B\ 3\ ]} \uparrow \underline{* B\ C\ ]}\ 1.2\ D\ ]$. It may be necessary to scan over the first underlined product in order to look at the second argument under a sum. Or, it may be necessary to determine the form of the exponent under a power operator, $\uparrow$. This necessitates scanning over the second underlined product expression.

The efficiency with which this operation is performed is surprisingly crucial to the total efficiency of the FORMAC object-time system. Figure 6 graphically displays the time requirements for various FORMAC subroutines during the execution of a particular program. The length of the bars indicate the time spent in each routine. The higher routines call the lower routines directly if the routines are adjacent; indirectly, if not. The top display indicates the time spent in the original FMCSEX routine. It is apparent that over half the time is spent in this routine. It was a surprise to the implementers of the FORMAC object-time system that the FMCSEX routine represented this large a proportion of the work required to manipulate symbolic expressions.

A new and different approach to the scan-off algorithm was coded in order to avoid this excessive waste of time. The original algorithm did complete testing for well-formedness. It used a pushdown store and was quite slow. The new algorithm, coded using the compare instructions of the 7090 family of computers, required the same space (due to compare tables) but does no checking for ill-formedness in expressions. The bottom display in Figure 6 indicates the dramatic reduction in execution time due to this algorithm.

The contrast pictured in Figure 6 provides still another example of a basic principle related to the design and implementation of large systems. When debugging a complex system, routines that perform complete error checking are a necessity. Without them, it is impossible to locate system bugs. Once the system has been debugged and it is put into actual production use, error-checking subroutines may be replaced with much more rapid routines that enhance total performance but do no error checking.

Skeletal Structure Extraction

The development of formal mathematical expression manipulation systems such as ALPAK, Formula ALGOL, and FORMAC, presents the scientist with a new data editing problem. The problem is analogous to that faced by physicists ten years ago when computers were first used to generate large volumes of numerical data. The physicist soon learned that data editing prior to output was essential if he was to comprehend the implications of his data.

It is not abnormal for expressions generated by FORMAC to require 2 to 300 lines (120 characters each) of listing when output. Such expressions are incomprehensible to the human reader. The expression is composed of several thousand characters and the main mathematical operators, which determine the essential nature of the expression, are buried somewhere within the massive listing of symbols. This situation will be aggravated as the various space

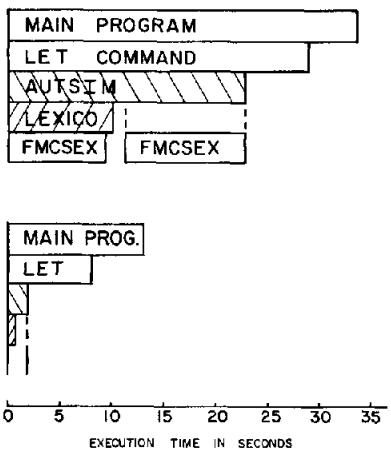

FIG. 6. Contrast in time required by the two FMCSEX routines during execution of a specific program.

and time limitations on computer systems for formal mathematics are reduced or eliminated. In the future it will be possible and perhaps common to generate and to manipulate expressions with hundreds of thousands or more elements. In such a situation the scientist or engineer will be swamped with overwhelming mathematical detail unless algorithms are developed for extracting and presenting the essential or skeletal structure of symbolic expressions. This problem is ever more acute in a time-shared, conversational environment.

In this section an experiment in skeletal structure extraction utilizing FORMAC is discussed. Skeletal structure extraction simplifies expressions by replacing subexpressions with single variables. This operation suppresses detail in an expression, thereby revealing the skeletal structure of the expression. This work was conceived and executed by James Baker, at the IBM Boston Programming Center.

The experiment was inspired by an attempt to establish that output from the North Carolina differentiation routine [8] was equivalent to the output of the FORMAC differentiation routine. Figure 7 displays the input expression

$$\begin{aligned}\text{INPUT} = {} & (g \sin^2 i + h \cos^2 i + f + ((f^2 + 2f(g \sin^2 i + h \cos^2 i) \\ & + g^2 \sin^2 i + h^2 \cos^2 i)(1 + 2t(g \sin^2 i + h \cos^2 i) \\ & + t^2(g^2 \sin^2 i + h^2 \cos^2 i)))^{\frac{1}{2}} + t(f(g \sin^2 i + h \cos^2 i) \\ & + g^2 \sin^2 i + h^2 \cos^2 i))/((1 - ft)(g \sin^2 i + h \cos^2 i \\ & + f + ((f^2 + 2f(g \sin^2 i + h \cos^2 i) \\ & + g^2 \sin^2 i + h^2 \cos^2 i)^{\frac{1}{2}})))\end{aligned}$$

FIG. 7. The input expression as published in [8]

as published in [8]. Figure 8 displays the output from the FORMAC differentiation routine and the North Carolina result in FORMAC notation. Both expressions have been transformed by AUTSIM. The initial attempt to check the equivalence of these expressions was made utilizing the FORMAC match-for-equivalence capability (MATCH EQ Command). This effort failed due to lack of adequate space in the computer. (The MATCH EQ algorithm employs several calls to the EXPAND algorithm. This work was carried out when the original EXPAND algorithm was the only one available; hence, simple matches for equivalence ran out of space due to excessive intermediate expression swell.) The equivalence of the two results was verified by hand. However, it was hoped that the equivalence could be shown using FORMAC by "shrinking" both outputs and then performing a match for equivalence on the shrunken forms.

Mathematical expressions are shrunk by the routine SHRINK. It is called with the name of a FORMAC expression and the name of two parallel lists. Upon each call to SHRINK, the bottom-level subexpressions of the expression are replaced by atomic variables, the elements of the first list. The corresponding elements of the second list are defined as the subexpressions that the atomic variables replace. The parallel lists (or dictionary), which were generated in the successive shrinking of the expressions in Figure 8 are displayed in Table III; the A-list contains the atomic variables (variables which name themselves) and the L-list contains the corresponding expressions. A glance at the L-array indicates what is meant by

---

North Carolina Differentiation Result in FORMAC Notation (Rearrangement is due to AUTSIM):

```
(F+(F*(G*FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0+
FMCSIN(I)**2.0+H**2.0*FMCCOS(I)**2.0)**5.0E-1+G*FMCSIN(I)**
2.0+H*FMCCOS(I)**2.0)**(-2.0)*(-F*T+1.0)**(-1.0)*(-(F+F*(G*
FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)*T+(F*(G*FMCSIN(I)**2.0+H*
FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)**2.0+H**2.0*
FMCCOS(I)**2.0)**5.0E-1*((G*FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)
*T+2.0+(G**2.0*FMCSIN(I)**2.0+H**2.0*FMCCOS(I)**2.0)*T**2.0+
1.0)**5.0E-1+G*FMCSIN(I)**2.0+(G**2.0*FMCSIN(I)**2.0+H**2.0*
FMCCOS(I)**2.0)*T+H*FMCCOS(I)**2.0)*((F*(G*FMCSIN(I)*FMCCOS(
I)*2.0+H*FMCSIN(I)*FMCCOS(I)*(-2.0))*2.0+G**2.0+FMCSIN(I)*
FMCCOS(I)*2.0+H**2.0*FMCSIN(I)*FMCCOS(I)*(-2.0))*(F*(G*
FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)
**2.0+H**2.0*FMCCOS(I)**2.0)**(-5.0E-1)*5.0E-1+G*FMCSIN(I)*
FMCCOS(I)*2.0+H*FMCSIN(I)*FMCCOS(I)*(-2.0))+(F+(F*(G*FMCSIN(
I)**2.0+H*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)**2.0+H
**2.0*FMCCOS(I)**2.0)**5.0E-1+G*FMCSIN(I)**2.0+H*FMCCOS(I)**
2.0)*(F*(G*FMCSIN(I)*FMCCOS(I)*2.0+H*FMCSIN(I)*FMCCOS(I)*(
-2.0))*T+(F*(G*FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)*2.0+F**2.0+G
**2.0*FMCSIN(I)**2.0+H**2.0*FMCCOS(I)**2.0)**(-5.0E-1)*((F*(
G*FMCSIN(I)*FMCCOS(I)*2.0+H*FMCSIN(I)*FMCCOS(I)*(-2.0))*2.0+
G**2.0+FMCSIN(I)*FMCCOS(I)*2.0+H**2.0*FMCSIN(I)*FMCCOS(I)*(
-2.0))*((G*FMCSIN(I)**2.0+H**2.0*FMCCOS(I)**2.0)*T**2.0+1.0)+(F*(G*
FMCSIN(I)**2.0+H**2.0*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)
**2.0+H**2.0*FMCCOS(I)**2.0)*((G*FMCSIN(I)*FMCCOS(I)*2.0+H*
FMCSIN(I)*FMCCOS(I)*(-2.0))*T*2.0+(G**2.0*FMCSIN(I)*FMCCOS(I
)*2.0+H**2.0*FMCSIN(I)*FMCCOS(I)*(-2.0))*T**2.0))*((G*FMCSIN
(I)**2.0+H*FMCCOS(I)**2.0)*T**2.0+(G**2.0*FMCSIN(I)**2.0+H*
2.0*FMCCOS(I)**2.0)*T**2.0+1.0)**(-5.0E-1)*5.0E-1+G*FMCSIN(I
)*FMCCOS(I)*2.0+(G**2.0*FMCSIN(I)*FMCCOS(I)*2.0+H**2.0*
FMCSIN(I)*FMCCOS(I)*(-2.0))*T+H*FMCSIN(I)*FMCCOS(I)*(-2.0)))
```

FORMAC Differentiation Result:

```
-(F+F*(G*FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)*T+(F*(G*FMCSIN(I)
**2.0+F*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)**2.0+H**
2.0*FMCCOS(I)**2.0)**5.0E-1*((G*FMCSIN(I)**2.0+H*FMCCOS(I)**
2.0)*T*2.0+(G**2.0*FMCSIN(I)**2.0+H**2.0*FMCCOS(I)**2.0)*T**
2.0+1.0)**5.0E-1+G*FMCSIN(I)**2.0+(G**2.0*FMCSIN(I)**2.0+H**
2.0*FMCCOS(I)**2.0)*T+H*FMCCOS(I)**2.0)*(F+(F*(G*FMCSIN(I)**
2.0+F*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)**2.0+H**2.0
*FMCCOS(I)**2.0)**5.0E-1+G*FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)
**(-2.0)*(-F*T+1.0)**(-1.0)*((F*(G*FMCSIN(I)*FMCCOS(I)*
2.0+H*FMCSIN(I)*FMCCOS(I)*(-2.0))*2.0+G**2.0+FMCSIN(I)*
FMCCOS(I)*2.0+H**2.0*FMCSIN(I)*FMCCOS(I)*(-2.0))*(F*(G*
FMCSIN(I)**2.0+H*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)
**2.0+H**2.0*FMCCOS(I)**2.0)**(-5.0E-1)*5.0E-1+G*FMCSIN(I)*
FMCCOS(I)*2.0+H*FMCSIN(I)*FMCCOS(I)*(-2.0))+(F+(F*(G*FMCSIN(
I)**2.0+H*FMCCOS(I)**2.0)*2.0+F**2.0+G**2.0*FMCSIN(I)**2.0+H
**2.0*FMCCOS(I)**2.0)**5.0E-1+G*FMCSIN(I)**2.0+H*FMCCOS(I)**
2.0)**(-1.0)*(F*(G*FMCSIN(I)*FMCCOS(I)*2.0+H*FMCSIN(I)*
FMCCOS(I)*(-2.0))*T+(F*(G*FMCSIN(I)*FMCCOS(I)*2.0+H*FMCSIN(I
)*FMCCOS(I)*(-2.0))*2.0+G**2.0*FMCSIN(I)*FMCCOS(I)*2.0+H**
2.0*FMCCOS(I)**2.0)*(F*(G*FMCSIN(I)*FMCCOS(I)*2.0+H*FMCSIN(I
)*FMCCOS(I)*(-2.0))*2.0+G**2.0*FMCSIN(I)*FMCCOS(I)*2.0+H**
2.0*FMCCOS(I)**2.0)*T**2.0+1.0)**(-5.0E-1)*5.0E-1+G*FMCSIN(I
)*FMCCOS(I)*2.0+(G**2.0*FMCSIN(I)*FMCCOS(I)*2.0+H**2.0*
FMCSIN(I)*FMCCOS(I)*(-2.0))*T+H*FMCSIN(I)*FMCCOS(I)*(-2.0)))
```

FIG. 8. The differentiation results as they appear in FORMAC notation after simplification.

the phrase "bottom-level subexpression." It is a subexpression with a single mathematical operator (as viewed in delimiter Polish) and only constants or atomic variables occurring at the level below that operator. Before creating a new item on the L-list and introducing the corresponding item from the A-list into the expression, SHRINK checks the previous items on the L-list to make sure that the expression it is about to substitute for has not already been assigned an atomic variable; hence, new atomic variables are introduced only for new subexpressions. If SHRINK is called with another expression as argument but with the same parallel lists, subexpressions in the new expression, that have already been assigned atomic variables, will be replaced by those atomic variables. This use of a common dictionary is a necessity if a match for equivalence is to be attempted on the shrunken forms.

Since subexpressions are replaced by atomic variables, the resultant expression requires less storage space. Moreover, the new expression can be shrunk again; the process can be iterated several times in order to achieve a workably small result. The parallel lists of Table III are the result of such iteration.

The results obtained by successively shrinking the output expressions of Figure 8 and attempting to match for equivalence at each level of the SHRINK are quite interesting. Table IV shows the result of attempting a MATCH EQ at each level. (Level 0 is the original expression, level 1 the expression after one call to SHRINK, and level n the expression after n calls to SHRINK.) At the 7th SHRINK, the expressions diverge: they no longer contain the same atomic variables. A simple example indicates how this can happen. Consider the two expressions $(x + y) \cdot (x - y)$ and $x^2 - y^2$; these have the same atomic variables initially, but one call to SHRINK will produce the expressions a_1a_2 and $a_3 - a_4$.

The possibility of matching two structurally different expressions via the shrinking method depends upon the level at which the two expressions still contain the same atomic variables. If the last common atomic level results in an expression that will match within the confines of available core storage, then the match for equivalence will take place. Otherwise, there is no further hope with this technique. As is indicated in Table IV, the match between the North Carolina differentiation results and the FORMAC differentiation results took place at the last level prior to divergence. The FORMAC MATCH EQ command just barely made it.

It should be noted that trigonometric identities will generally cause a structural divergence at a fairly low level. For example, adding $\sin(2\omega)$ to the North Carolina result and $2 \sin \omega \cos \omega$ to the FORMAC result would cause a divergence at level one; adding $\sin^2\omega + \cos^2\omega$ to the North Carolina result and 1 to the FORMAC result would cause a divergence at level zero (i.e., the original expressions would no longer have the same atomic variables). Thus, a primitive technique such as SHRINK is of limited use in a complicated context where functional identities apply.

The results of this experiment indicate the promise of very simple techniques for developing more powerful expression manipulating algorithms internal to expression

TABLE III. DICTIONARY ENTRIES OF PARALLEL LISTS GENERATED BY SHRINK

A-List	*L-List*	*A-List*	*L-List*
	1st Level		*5th Level*
A1	FMCSIN (*I*)	A35	A28 * *F* * *T*
A10	FMCCOS (*I*)	A36	A28 * *F* * 2.0
A100	*F* * * 2	A37	A28 * *T* * 2.0
A11	*G* * * 2	A38	A13 * A29
A12	*H* * * 2	A39	A29 * *T*
A13	*T* * * 2	A4	A2 + A20 + A30
A14	*F* * *T*	A40	A32 + A33
	2nd Level		*6th Level*
A15	A1 * * 2.0	A41	A100 + A23 + A24 + A36
A16	A10 * * 2.0	A42	A37 + A38 + 1.0
A17	−A14		*7th Level*
A18	A1 * A10 * *G* * 2.0	A43	A41 * * 5.0*E* − 1
A19	A1 * A10 * *H* * (−2.0)	A44	A42 * * 5.0*E* − 1
A2	A1 * A10 * A11 * 2.0	A45	A41 * * (−5.0*E* − 1)
A20	A1 * A10 * A12 * (−2.0)	A46	A42 * * (−5.0*E* − 1)
	3rd Level	A47	A4 * A42
A21	A15 * *G*	A48	A40 * A41
A22	A16 * *H*		*8th Level*
A23	A11 * A15	A49	A43 * A44
A24	A12 * A16	A5	A21 + A22 + A43 + *F*
A25	A17 + 1.0	A50	A4 * A45 * 5.0*E* − 1
A26	A18 + A19	A51	A4 * A44 * A45 * 5.0*E* − 1
A27	A2 + A20	A52	A40 * A43 * A46 * 5.0*E* − 1
	4th Level	A53	A47 + A48
A28	A21 + A22		
A29	A23 + A24		
A3	A25 * * (−1.0)		
A30	A26 * *F* * 2.0		
A31	A26 * *F* * *T*		
A32	A26 * *T* * 2.0		
A33	A13 * A27		
A34	A27 * *T*		

TABLE IV. RESULTS OF MATCH FOR EQUIVALENCE AND STORAGE REQUIREMENTS OF THE EXPRESSIONS AT EACH LEVEL OF SHRINKING

Level of SHRINK	*Storage requirement for North Carolina form in FORMAC*	*Storage requirement for FORMAC form in FORMAC*	*Result of MATCH EQ*
0	898	915	Blew up
1	695	709	Not tried
2	436	449	Not tried
3	276	285	Not tried
4	206	212	Not tried
5	134	140	Blew up
6	89	95	Match
7	62	68	No match (structure no longer same)
8	41	38	No match (structure no longer same)

manipulation systems. Possible drawbacks to such approaches are also implied. The value of SHRINK for producing comprehensible mathematical expressions is indicated in Figures 9 and 10. Here are displayed the two expressions, the results from the North Carolina differentiation algorithm, and the results from the FORMAC differentiation algorithm, as they appear after 6 and 7 calls to SHRINK. The essential expression structure is obvious from this display; indeed, the human being can now see at a glance that these two expressions are equivalent (Figure 9). (These two expressions were purposely translated from the linear FORTRAN notation. The need for two-dimensional output of mathematical expressions in order to make them legible to the mathematician or scientist is widely accepted, and work is being done in this area [10, 11]. The issue under discussion is independent of this problem.) As the expressions which man generates and manipulates by computer grow in size, the necessity for techniques far more sophisticated than SHRINK will become increasingly obvious.

Shrunken North Carolina Result:

$$\left[\left(A_{18} + A_{19} + A_{31} + A_{34} + \frac{1}{2}\frac{A_4 A_{42} + A_{40} A_{41}}{\sqrt{A_{41}}\sqrt{A_{42}}}\right)(A_{21} + A_{22} + \sqrt{A_{41}} + F) - \left(A_{18} + A_{19} + \frac{1}{2}\frac{A_4}{\sqrt{A_{41}}}\right)(A_{21} A_{22} + A_{35} + A_{39} + \sqrt{A_{41}}\sqrt{A_{42}} + F)\right] \cdot \frac{A_3}{(A_{21} + A_{22} + \sqrt{A_{41}} + F)^2}$$

Shrunken FORMAC Result:

$$\left(A_{18} + A_{19} + A_{31} + A_{34} + \frac{1}{2}\frac{A_4\sqrt{A_{42}}}{\sqrt{A_{41}}} + \frac{1}{2}\frac{A_{40}\sqrt{A_{41}}}{\sqrt{A_{42}}}\right) \cdot \frac{A_3}{(A_{21} + A_{22} + \sqrt{A_{41}} + F)} - \left(A_{18} + A_{19} + \frac{1}{2}\frac{A_4}{\sqrt{A_{41}}}\right) \cdot (A_{21} + A_{22} + A_{35} + A_{39} + \sqrt{A_{41}}\sqrt{A_{42}} + F) \cdot \frac{A_3}{(A_{21} + A_{22} + \sqrt{A_{41}} + F)^2}$$

FIG. 9

Shrunken North Carolina Result:

$$[(A_{18} + A_{19} + A_{31} + A_{34} + \tfrac{1}{2}A_{45} A_{46}(A_{47} + A_{48}))(A_{21} + A_{22} + A_{43} + 5) - (A_{18} + A_{19} + \tfrac{1}{2} A_4 A_{45})(A_{21} + A_{22} + A_{36} + A_{39} + A_{43} A_{44} + F)] \cdot \frac{A_3}{(A_{21} + A_{22} + A_{43} + F)^2}$$

Shrunken FORMAC Result:

$$(A_{18} + A_{19} + A_{31} + A_{34} + \tfrac{1}{2} A_4 A_{44} A_{45} + \tfrac{1}{2}A_{40} A_{42} A_{46}) \cdot \frac{A_3}{A_{21} + A_{22} + A_{43} + F} - (A_{18} A_{19} + \tfrac{1}{2}A_4 A_{45}) \cdot (A_{21} + A_{22} + A_{35} + A_{39} + A_{43} A_{44} + F) \cdot \frac{A_3}{(A_{21} + A_{22} + A_{43} + F)^2}$$

FIG. 10

Summary

The importance of avoiding intermediate expression swell in the design of algorithms for symbolic manipulation of formulas has been indicated by remarks concerning the design of expand algorithms and differentiation algorithms. As in all computer processing, this may boil down to the problem of designing algorithms that are slower but require less space. In the case of differentiation, however, the improved algorithm eliminated redundant processing as well as redundant data, resulting in slightly faster execution than the original algorithm. The old truism that speed can be bought by eliminating error-checking is again borne out by the discussion of the FMCSEX subroutine.

The basic problem addressed by the SHRINK routine promises to be a challenge to future investigators. Mathematicians, engineers and scientists will be using the computer to generate symbolic expressions much larger and more complicated than any they have seen before. Suitable techniques must be developed to make the essential meaning of these expressions comprehensible.

Acknowledgments. The experimentation and study of FORMAC was pursued by Marc Auslander, Jim Baker, Mathew Myszewski, and the author.

The code changes of FORMAC Programs were made by Marc Auslander and Patricia Cundall, under the direction of Robert Kenney.

REFERENCES

1. SAMMET, J. E., AND BOND, E. R. Introduction to FORMAC. *IEEE Trans. EC-13* (Aug. 1964), 386.
2. BOND, E. R., ET AL. FORMAC—an experimental FORmula MAnipulation Compiler. Proc. 19th ACM Nat. Conf., Aug. 1964, Paper K2.1.
3. FORMAC. SHARE General Program Library, 7090 R2 IBM 0016, IBM Program Inf. Dept., White Plains, N. Y.
4. BOND, E. R., ET AL. Implementation of FORMAC. IBM Tech. Rept. 00.1260, March, 1965.
5. TOBEY, R. G., BOBROW, R. J., AND ZILLES, S. N. Automatic simplification in FORMAC. Proc. AFIPS 1965 Fall Joint Comput. Conf., Pt. 1, Nov. 1965, p. 37.
6. ——. Eliminating monotonous mathematics with FORMAC. IBM Tech. Rept. 00.1365, Nov., 1965.
7. SCONZO, P., LESCHACK, A. R., AND TOBEY, R. Symbolic computation of f and g series by computer. *Astron. J. 70* (May 1965), 269.
8. HANSON, J. W., CAVINESS, J. S., AND JOSEPH. C. Analytic differentiation by computer. *Comm. ACM 5*, 6 (June 1962), 349.
9. GRILLIOT, T. J. Derivatives of composite functions. *Am. Math. Month. 69*, (Nov. 1962), 914.
10. KLERER, M., AND MAY, J. Two-dimensional programming. Proc. AFIPS Fall Joint Comput. Conf., Pt. 1, Nov. 1965, p. 63.
11. MARTIN, W. A. Syntax and display of mathematical expressions. Project MAC Memo MAC-M-257, M.I.T., Cambridge, Mass., July, 1965. (Unpublished)