ACM Communications in Computer Algebra, Vol. 45, No. 4, Issue 178, December 2011

Ways to implement computer algebra compactly

David R. Stoutemyer*

Abstract

Computer algebra had to be implemented compactly to fit on early personal computers and hand-held calculators. Compact implementation is still important for portable hand-held devices. Also, compact implementation increases comprehensibility while decreasing development and maintenance time and cost, regardless of the platform. This article describes several ways to achieve compact implementations, including:

- Exploit evaluation followed by interpolation to avoid implementing a parser, such as in PicoMath^{tm}.
- Use contiguous storage as an expression stack to avoid garbage collection and pointer-space overhead, such as in Calculus Demon^{tm} and TI-Math-Engine.
- Use various techniques for saving code space for linked-storage representation of expressions and functions, such as in muMath^{tm} and *Derive*®

1 Introduction

"Inside every large program is a small program struggling to emerge."
– Hoare's law of large programs.

This article is a written version of a PowerPoint presentation at the 2008 Applications of Computer Algebra conference.

FORTRAN was my first programming language, around 1965. I was disappointed when I learned that it could only substitute numbers into formulas, rather than also do algebraic transformations. I was therefore delighted when I learned around 1973 that computer algebra programs existed, and I began using Reduce and Macsyma. I wanted my engineering students to learn to use them, but these systems tended to monopolize our time-shared university mainframe computer, quickly exhausting my course computing budget. This was the cause of my long-standing effort to make computer algebra available on personal computers that are free of time budgets and on robust inexpensive hand-held calculators that can be conveniently carried for spontaneous use in all classrooms, at home and while traveling.

At that time the available computer algebra systems barely fit on only the largest mainframes, and RAM was so expensive that personal computers and calculators had very little address space and often even less memory. Consequently, the most important design constraint was compactness of the computer algebra *implementation*.

Traditional student mathematics problems don't tend to entail enormous input or result expressions. Therefore compactness of the data representation is less important. Because of the modest

*dstout at hawaii dot edu

problem size, speed is also less important. For example, it isn't worth implementing an asymptotically faster algorithm if it is more complicated, and it is even less worthwhile combining several algorithms to achieve optimal speed all the way from the smallest through largest possible problem sizes. However, compact expression size and surprisingly fast execution can sometimes be obtained without sacrificing program compactness.

Along the way others and I discovered several quite different ways to implement computer algebra compactly. This article is an effort to collect in one place a description of these techniques and some references to relevant literature.

Although one purpose of this article is to record some history, this article is organized primarily by the type of implementation, then chronologically within each type.

Compactness is still important despite the declining cost of computer memory and the (more gradually) increasing speed of Internet connections, because:

- Maintenance cost and the time required to develop software grow super-linearly with program size.
- Program cache faults can be less frequent for small programs.
- Carbon footprints grow with silicon footprints, secondary storage footprints and transmitted file footprints. Estimates of electricity consumption associated with the Internet vary wildly from 1% for servers and their cooling worldwide [16], through 9.4% in the USA including also prorated client electricity costs [33]. Many programs and data files are stored on many computers, run on many computers, are transmitted many times, and are printed many times. Therefore the benefit of program compactness grows with the number of users.

Section 2 describes the typical constraints of early micro-computers and programmable calculators. Section 3 describes two different methods that are especially compact but suitable only for specialized classes of expressions. Section 4 describes three different methods that are better for general expressions.

2 Early micro-computers & programmable calculators

In 1976 Cioni and Miola [4] described implementing parts of the modular SAC-1 computer algebra system on a PDP-11 minicomputer having 64 kilobytes of memory with two hard disks, using overlays. Minicomputers with hard disks were too expensive for purchase by individuals, but it was a demonstration that useful computer algebra could be done in a relatively small amount of memory.

The first *mass-market* personal computers were the Apple II, Commodore Pet, Radio Shack TRS-80, and Atari 400 and 800, which all had 64 kilobytes of address space. Up to 25% of that was devoted to ROM that provided interactive terminal I/O, loading of programs stored on secondary storage, and an interpreted BASIC programming language that wasn't well suited to implementing computer algebra. However, if the BASIC wasn't exploited to help implement computer algebra, then the associated ROM was wasted address space, and precious RAM storage would have to be used for the entire implementation and data.

RAM was so expensive that the entry-level versions of these computers came with only 4 kilobytes to 16 kilobytes, and most users didn't buy more. Four kilobytes was typically only enough for about

100 lines of BASIC. Secondary storage was typically an audio cassette, which was too slow and unreliable to serve as virtual memory.

Programmable calculators were even more spartan. There was typically enough memory for about 100 steps of an interpreted assembly-like language that provided instruction mnemonics and some preassigned symbolic address labels such as LBL1, LBL2, etc. Typically the one-line display was only about 14 characters wide and incapable of displaying most characters other than what is necessary for floating-point numbers. Sometimes there was a slot for ROM cartridges containing programs purchased from the manufacturer or a third party. Usually programs and data could be stored to and loaded from secondary storage consisting of a magnetic strip or a miniature magnetic tape cartridge.

3 Special-purpose methods

A computer algebra system typically has a driver loop that repeatedly accepts inputs interactively entered by a user, producing and displaying a corresponding simplified result for each input. Usually there is:

- a parser that converts the user's human-oriented input expression into an internal tree-like form more suitable for mathematical transformations,
- a simplifier that transforms the parsed expression into a simplified equivalent internal form, and
- a displayer that transforms the simplified internal form into a string or a two-dimensional form more suitable for human comprehension.

The simplifier includes arithmetic, algebra, calculus, etc.

General expressions include at least rational expressions, fractional powers, exponentials, logarithms, trigonometric functions and their inverses, together perhaps with vectors and matrices having such expressions as entries.

Special-purpose methods are best suited for implementing only a narrow subset of general expressions, such as polynomials, univariate rational expressions, or Maclauren series. However, such a category is all that is needed for some applications, such as a web-based applet that exercises or tests students with problems in a particular category.

3.1 Dense univariate series and polynomial algebra

It is easy to implement univariate polynomials, Maclauren series, or Fourier series as a one-dimensional array of floating-point coefficients. Even the 1965 IBM FORTRAN Scientific Subroutine Package [14] had subroutines for operations such as adding, multiplying and integrating such polynomials with floating-point coefficients. The user was responsible for writing a main program to initialize or enter the input polynomial coefficients, then call an appropriate sequence of subroutines, then display the result polynomial coefficients.

In 1977 Henrici [9] describes an analogous set of subroutines for univariate Maclauren series on the HP 25 calculator, which had memory for only 49 steps, 8 direct-address registers, and a 4-register stack. The limited memory forced Henrici to use some ingenious indirect methods so that each coefficient of a result could be computed independently.

In 1977 Stoutemyer [25] described an analogous set of routines for the 224-step HP 67 and HP 97 calculators, using the more traditional algorithms described by Knuth [17]. All series were of degree 9. Table 1 lists the number of program steps and the execution times for the implemented operations, which are independent of the particular ten input coefficients for each polynomial.

Table 1: Size and speed of the HP-67 Maclauren-series operations

Operation	Number of steps	Seconds
subtraction	4	20
copying	5	10
negation	10	10
addition	14	10
substitute number	16	10
integration	16	10
multiplication	38	60
division	49	60
reversion	62	180

Knuth also gives algorithms for substituting one power series into another, raising a power series to a real power, and computing the exponential or logarithm of a power series. However, there isn't enough space to fit them simultaneously with the above routines, so they would have to be overlaid from secondary storage.

With a non-qwerty keyboard, entering coefficients alone in response to prompts is actually more convenient than also entering intervening sub-strings of the form "...*x^... +". Consequently there is no strong need for parsing such tightly-scoped mathematical expressions.

The technique could be extended to multivariate polynomials by using multidimensional coefficient arrays or mapping them into 1-dimensional arrays, However, such dense distributed representation is notoriously inefficient for most practical multivariate polynomial problems. Consequently, that extension would be well beyond the point of diminishing returns for this parser-less coefficient array method.

Though narrowly scoped, the two HP calculator series programs were interesting demonstrations that useful computer algebra could be done with floating point arithmetic and 49 or 224 steps of assembly language.

3.2 Computer algebra by evaluation and interpolation

It is all done by smoke and mirrors.
– Anne Brooke

With only four to sixteen kilobytes of RAM installed on most early commercial personal computers, there was no choice for this market except to use a built-in Basic for implementation even though those versions of Basic were not an attractive implementation language because:

- The entry-level Basic typically didn't support string variables: All variables were global floating point with non-mnemonic names A through Z.
- Subroutines couldn't have parameters or local variables.

- Function definitions were limited to unconditional one-line expressions.
- One-dimensional and perhaps two-dimensional arrays of floating-point numbers were the only non-scalar data structure.

Moreover, without string variables and string processing function such as substring(...), it is impractical to write even a parser. Nonetheless, we can still do computer algebra by finessing it with the evaluation-interpolation homomorphism:

BASIC already has a built-in parser, but its inseparable simplifier expects all of the variables in an expression to have assigned numeric values so that the final value and all intermediate values are numeric. In 1980 PicoMath [27] exploited this by evaluating a user's unsimplified expression at a set of points, then interpolating a simplified expression through those points. PicoMath then used character string constants such as "+" and "X" together with the interpolated coefficients to display the simplified result expression.

Programma International published PicoMath for the TRS80, Apple II, Commodore Pet and Atari 400 and 800. Texas Instruments published a ROM version for the TI 99/4. TI also planned to release a ROM version for their TI-59 calculator, which was discontinued just before that release. Arthur Norman adapted PicoMath for the Acorn BBS computer and Exidy Sorcerer, with the improvement of recovering simple exact fraction coefficients by repeated quotients, remainders and reciprocation with a tolerance. PicoMath also ran on the 1×7×17 cm Sharp PC-1211 or Radio-Shack Pocket computer, which had 2 kilobytes of RAM.

PicoMath is actually four separate programs for simplifying expressions to four different classes of results:

- The *rational* program can expand and reduce over a common denominator a univariate rational expression in x, such as

$$1+\frac{1}{x-1}-\frac{1}{x+1}+\frac{2x}{x^2-1} \;\rightarrow\; \frac{x+1}{x-1},$$

$$\frac{\dfrac{x^2-5x-6}{x^2-2x-15}\cdot\dfrac{x^2-7x+10}{x^2+5x+4}}{\dfrac{2x-12}{x^2+3}} \rightarrow \frac{x^2-2x}{2x+8}.$$

  To limit the effect of rounding errors, the maximum degrees of the numerator and denominator of the result are limited to about half the number of significant digits in the floating-point arithmetic.

- The *polynomial* program can expand, then optionally integrate or differentiate, with respect to x, polynomials in x, y and z such as

$$\begin{aligned}(x+1)^6+(x+y+1)^4+(x+y+z+1)^2 \\ \rightarrow x^6+6x^5+16x^4+4x^3y+24x^3+6x^2y^2+12x^2y+22x^2+4xy^3 \\ +12xy^2+14xy+2xz+12x+y^4+4y^3+7y^2+2yz+6y+z^2+2z+3,\end{aligned}$$

$$\int \left(\left(x^3-1\right)\left(x^3+1\right)+y^4+z^2\right)\,dx \;\;\rightarrow\;\; 0.142857x^7+xy^4+xz^2-x,$$
$$\frac{d}{dx}\left(x^6+xy^3+z^2\right) \;\;\rightarrow\;\; 6x^5+y^3.$$

As illustrated: In the simplified expression, integrand or differand, any term containing z is limited to degree 2, any term containing y is limited to total degree 4, and any term containing x is limited to total degree 6.

- The *trigonometric* program can simplify, then optionally integrate or differentiate with respect to x many trigonometric expressions in x and y. For example,

$$\int 2\sin\left(\frac{x+y}{2}\right)\cdot\cos\left(\frac{x-y}{2}\right)\,dx \;\;\rightarrow\;\; x\cdot\sin(y)-\cos(x),$$
$$\frac{\sin(x+3\pi/2)}{\cos(x+\pi)}+\frac{\cos(x-3\pi/2)}{\sin(x+\pi/2)} \;\;\rightarrow\;\; -\tan(x)+1,$$
$$\frac{d}{dx}\left(\frac{1+\cos(2x)}{\cos(x)}+\frac{\sin(2x)}{\sin(x)}\right) \;\;\rightarrow\;\; -4\sin(x).$$

  The simplified expression, integrand or differand can be any linear combination of the expressions 1, $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\csc x$, $\sec x$, $\sin x\cdot\cos x$, $\sin(x)^2$, $\sec(x)^2$, $\tan x\cdot\sec x$, $\cot x\cdot\csc x$, $\sin y$, $\cos y$, $\sin x\cdot\sin y$, $\sin x\cdot\cos y$, $\cos x\cdot\sin y$, $\cos x\cdot\cos y$. This set of basis functions covers a surprising portion of the results in textbook trigonometric examples and exercises.

- The *Fourier* program transforms polynomials in sines and cosines of x and of integer multiples of x into a linear combination of sines and cosines of x and integer multiples of x, then optionally integrates or differentiates with respect to x. It thus computes the spectrum or Fourier decomposition of univariate sinusoidal expressions. For example,

$$64\sin(x)^4\cos(x) \;\rightarrow\; 3\cos(x)-3\cos(3x)-\cos(5x)+\cos(7x),$$

$$\int -4\sin(2x)^2\cos(x)-4\cos(x)^3+5\cos(x)\,dx \;\rightarrow\; 0.2\sin(5x),$$
$$\frac{d}{dx}\left(-4\sin(2x)^2\cos(x)-4\cos(x)^3+5\cos(x)\right) \;\rightarrow\; 5\sin(5x).$$

Here is how it is done:

1. All four programs begin with the statements

2. ```
   10 GOTO 40
   20 A = expression to simplify, integrate or differentiate
   30 RETURN
   40 ...
   ```
   ```

3. The instruction manual tells users to modify the right side of the assignment statement on line number 20 to an expression that they want simplified to the implemented class. Except for the rational program, users are then queried whether they want to integrate or differentiate or simplify the expression on line number 20.

4. The program then executes the subroutine starting on line number 20 for different appropriately-chosen values of the independent variable or variables, depending on the program. From these samples it determines the coefficients of an interpolating expression in the class covered by the program.

5. It then evaluates the interpolant and the expression on line number 20 at a few extra points. If the magnitude of the relative difference or absolute difference exceeds tolerances at any of these points, then the user sees an error message such as, for the polynomial program:

   "Sampling suggests significant discrepancy. Perhaps line 20 is not equivalent to a polynomial in X, Y and Z, or the polynomial is of excessive degree or the expression is too sensitive to limitations of the BASIC arithmetic."

   "For comparison, LIST line 20. If desired, alter it to assign to A any expression equivalent to an expanded polynomial in X, Y and Z. The total degree of any expanded result term containing X, Y, and Z should not exceed 6, 4 and 2 respectively."

6. If instead all of the discrepancies are within tolerances, then for all but the rational program the user is asked to enter E for expand, I for integrate or D for differentiate. (For versions of BASIC that don't provide character or string variables, E, I and D are preassigned different numeric values that can be tested to determine the user's choice.)

7. The program then lists the coefficients of the interpolant expression (appropriately modified for the integration and differentiation choices) interspersed with character strings such as "x^2" or "cos(" and "+" to display the result expression. To avoid spurious terms caused by rounding errors, artificial underflow is used to replace by 0.0 any computed coefficients that have relatively small magnitude.

8. The program then lists the second paragraph of the above message beginning "For comparison ...", then stops. The user can then modify line 20 to do another problem for the same expression class, or the user can load one of the other three programs to treat a different expression class.

The polynomial and trigonometric programs use explicit fixed-basis formulas for the interpolant coefficients as linear combinations of the sample values, derived for the specific sample points. These formulas were derived by solving a set of linear equations having a symbolic right side $A_1, A_2, \ldots$, using Macsyma. The interpolation points were selected to make the resulting linear combinations sparse and simple. For example, x, y, and z were set to combinations of 0, 1, -1 and other small-magnitude integers for the polynomial program. For the trigonometric program, x and y were set to various simple multiples of π that avoid any poles of all the basis function. Moreover, computation of the coefficients is interleaved with display to recycle some of the 26 scalar variables, and common sub-expressions among the coefficients were exploited to reduce the program size.

The Fourier program uses an interpolant of the form

$$y = c_0 + s_1 \sin x + c_1 \cos x + s_2 \sin(2x) + c_2 \cos(2x) + \cdots + c_m \cos(mx)$$

together with sample points $x_j = j\pi/(2m+1)$ with $j = 0, 1, \ldots, 2m$, giving corresponding y_j.

The general closed-form solution is the discrete Fourier transform:

$$\begin{aligned} c_0 &= \sum_{j=0}^{2m} y_j, \\ c_k &= \sum_{j=0}^{2m} y_j \cos\left(\frac{2jk\pi}{2m+1}\right), \\ s_k &= \sum_{j=0}^{2m} y_j \sin\left(\frac{2jk\pi}{2m+1}\right). \end{aligned}$$

Integer m is set to 8, but it could be increased because this set of basis functions is orthogonal, making the interpolation relatively insensitive to rounding errors.

The rational program uses the method of inverted differences described by Hildebrand [10]. The maximum allowable numerator and denominator degree were determined by iteratively computing at run time a near-minimal $\varepsilon > 0$ such that $\arctan(1.0 + \varepsilon) = \arctan(\varepsilon)$. (Microsoft BASIC tended to use double precision for arithmetic but single precision for all exponentiation and for irrational functions such as sinusoids.) A tolerance was used to determine convergence or the lack thereof within this maximum degree. To reduce the expectation of having a sample abscissa near or on a zero of any denominator, the successive abscissas were taken as the transcendental numbers $x = \ln(2)$, $\ln(2) + 1$, $\ln(2) - 1$, $\ln(2) + 2$, $\ln(2) - 2$, ...

The evaluation-interpolation technique was pioneered by Brown and Collins [1] for computing polynomial gcds. Sparse interpolation pioneered by Zippel [32] has made it quite effective even for sparse multivariate polynomial gcds and for many other tasks. The novelty of Picomath was to use evaluation-interpolation to avoid the need for a parser. However, using evaluation-interpolation alone as a simplifier is suitable primarily only for narrowly-scoped applications such as a Web-based applet that tests or exercises students within narrow expression classes.

The four PicoMath programs could easily be combined, making the program try all four models then reject those that have unacceptable discrepancies at the extra sample points. However, even the scope of such a combined program is too narrow to be of widespread general interest, and the space required for the combined four programs would be comparable to the more flexible Calculus Demon BASIC program described in the next section.

4 General-purpose methods

Most of the time, most people want to treat general multivariate expressions composed without restrictions of at least the operators $+$, $-$, $\cdot$, $/$, and $\hat{}$, together with exponentials, logarithms, trigonometric functions and inverse trigonometric functions. They want optional transformations that include at least reduction over a common denominator, polynomial expansion, factoring and various trigonometric transformations. This section discusses implementation methods that are open ended — fully capable of accomplishing this and much more.

The scrollable history of input-result pairs of all the systems described in this section were inspired by the versions of Reduce and Macsyma that I started using in 1973.

4.1 String-based computer algebra

It is possible to do computer algebra directly on strings containing expressions in ordinary infix notation, as humans do. There were several early compact programs for doing differentiation using the pioneering string-processing language Snobol, which included string pattern matching. Many of these programs required the expression to be thoroughly parenthesized to avoid the need for parsing other than matching left and right parentheses.

There is some appeal to using the same representation internally as for I/O, but it is inefficient to repeatedly scan strings to delimit operands and locate operators between them. Also, although pattern matching is well suited to some operations such as differentiation, pattern matching is cumbersome for some other simplification algorithms such as the known efficient and thorough algorithms for integration, polynomial gcds, and factoring. Moreover, thorough exploitation of mathematical properties such as commutativity and associativity is cumbersome in a general purpose pattern matcher that doesn't have that capability built in.

4.2 Contiguous stack-based computer algebra

Mathematical expressions can be represented as trees. Computer algebra transforms a tree representing the input expression into a tree representing the corresponding result expression. Most computer algebra systems represent these trees using blocks of memory linked by pointers. They use either reference counters or garbage collection to recycle blocks of memory that are no longer referenced directly or indirectly by the program.

In contrast, Polish and reverse Polish provide a way of packing an expression tree into contiguous stack containing no internal pointers or parentheses. With *reverse* Polish, an operator is deeper than its operands, such as on many HP calculators. This is particularly appropriate for numeric computation where the result is always a number and where in most cases there is no need to know the operator until after the operands have been computed.

In contrast, ordinary Polish with operators shallower than their operands is more appropriate for computer algebra where the result can be a non-numeric expression: When transforming or combining simplified non-numeric expressions it is usually most convenient to consider the operators of the expressions before considering their operands. For example, to implement rule-based integration it is convenient to branch on the top-level operator of the simplified integrand. If the top-level operator is a sum, then we can first try distributing the integration over the summands. If the top-level operator is a product, then we can factor out factors that are independent of the integration variable, etc.

The items in the Polish representation could be the individual characters representing digits of numbers, characters of function names, etc. However, it is more efficient to *tokenize*: For example, variable-length integers can be represented as an integer *type tag* on top of a length field on top of a binary or binary-coded decimal number. As another example, built-in function names can be more efficiently represented by unique short tags rather than with strings containing their names. Because of the position of the tags and of any length fields at the shallower end of each subexpression, tokenized Polish can only be traversed starting from the top-level operator, rather than from the bottom-level operand as in reverse Polish.

In 1965 Smith [23] published an algorithm for symbolic differentiation using a Polish stack representation.

Computer algebra can be done entirely using one contiguous stack of tokenized Polish expressions

— an *expression stack*. In 1981 Calculus Demon was my first general purpose computer algebra program based on this technique:

- It is written in about 800 lines of BASIC, several statements per line, including about 100 lines of on-line help. It was published by Atari for their model 400 and 800 computers.
- It can represent, differentiate, modestly integrate and simplify general expressions written in traditional infix notation, including fractional powers, exponentials, logarithms, trigonometric and inverse trigonometric functions.
- Optional transformations include polynomial expansion together with transformation of integer powers and products of sinusoids to linear combinations of sinusoids.

The stack of expressions is represented as an array of floats. Particular number tags represent the variables A through Z, operators such as +, and functions such as ln. Floating-point constants are distinguished by having another particular number tag on top of them. The tags for numbers and variables, unary operators and functions, or binary operators or functions are each grouped together so that mere boundary checks on tag values can determine the corresponding number of operands or arguments.

The deepest 26 elements of the expression stack, indexed 0 through 25, are a symbol table of indices for values of user variables A through Z. Assigned values are stored contiguously above that, with the most recently assigned on top. The working stack where simplified results are developed is on top of the assigned values. A symbol table entry contains 0.0 if the corresponding variable has no assigned value. Otherwise the entry contains the index of the topmost element of the stored value.

If an assignment is made to a variable that already has an assigned value, then the previous value is deleted; and any assigned values above it are moved down by the size of the deletion, with the assignment indices of the corresponding variables decreased by the size of the deletion.

The parser is recursive descent, which is one of the most compact alternatives when there are only a few operators. To save code space, parsing and simplification are done in the same pass:

- Numbers simplify to themselves.
- Variables simplify to themselves if they are indeterminates. Otherwise they simplify to their stored values. There is a "re-simplify" postfix operator @ that enables intervening assignments to indeterminates in stored values to contribute their new values.
- Otherwise, depending on the operator or function, the top one or two simplified expressions on the expression stack are replaced with the result of applying the operator or function to them.

Combining parsing with simplification avoids having to branch on the tag value in two separate places. Moreover, simplification has fewer cases to consider than with separate parser and simplifier passes because, for example:

- Subtractions and negations don't occur in simplified internal form. They are represented using negative numeric coefficients, which are transformed back to subtractions and negations by the display subroutine.

- Ratios don't occur in simplified internal form. They are represented using multiplication and the -1 power, which are transformed back to ratios by the display routine.

- Other trigonometric functions are replaced by sines and cosines in simplified internal form. Obvious cases such as $\sin(x)/\cos(x)$ are transformed back to $\tan(x)$ by the display subroutine.

A disadvantage of having a combined parser-simplifier is that intervening expansions, etc. might cause a lengthy delay before encountering a syntax error near the right end of the input expression. However, the space savings are worth this disadvantage for this platform.

With a pure stack discipline, routines *using* the stack, as opposed to *implementing* the stack, have direct access to only the topmost element. However, for efficiency, the program and subroutines often store into scalar BASIC variables the indices of expressions and sub-expressions within the stack and use these for faster subsequent access. Thus the data structure is more accurately called a *remember stack*.

For computer algebra, most procedures are most compactly written recursively. At the expense of increased program size, some or all of the recursion can easily be replaced by loops over terms and over factors within procedures that merely *inspect* expressions to determine a property. Often the procedure can be *semi-recursive* — recurring on the left sub-tree but looping on the right sub-tree or *vice versa*. For example, this is convenient for a procedure that determines whether or not an expression is free of a particular variable.

However, pushing corresponding new terms or factors while looping over successively deeper terms or factors of a previous expression typically produces the terms or factors in reversed order. For example, suppose we use a loop to distribute negation over two terms of a sum: The loop will push the negative of the shallowest term, then the negative of the deeper term. Thus the negative of the given shallower term ends up deeper than the negative of the given deeper term. Consequently, we then need a second loop to push a reversed copy of the reversed negated terms, increasing the program size even more over that of a purely recursive procedure.

Most versions of BASIC permit recursive subroutines. Moreover, most versions also provide a large enough return-address stack so that the expression stack is more likely to avoid overflow first. However, the lack of subroutine parameters and local variables makes it awkward for a subroutine to remember indices into the expression stack past intervening calls on subroutines — particularly recursive calls.. This handicap can be overcome by pushing such indices onto the expression stack, which makes that stack a mixture of Polish expressions and indices of deeper Polish expressions.

Operators "+" and "∗" are treated as binary rather than n-ary, with associativity and commutativity being used to sort the operands into a canonical lexical order. Although n-ary "+" represents lengthy sums more compactly, the program would be more complicated: Each recursive routine that processed sums would have to invoke an auxiliary recursive routine that recurred over the operands to build a list of result terms, then the invoking routine would have to attach a "+" operator to the result if it was more than one term, or extract the first term from the list if there is only one term. There is even less reason to make products n-ary: Expanding polynomials doesn't generate new variables, so the number of factors in a fully-expanded term can't be more than 1 plus the number of indeterminates.

One of the most important Calculus Demon subroutines is one that, given the index of an expression or sub-expression, determines the index of the expression or sub-expression immediately below it. The algorithm is to initialize a global "sub-expression deficit" counter to 1, then decrement

it by 1 for a number or variable, or increment the counter by 1 for a binary operator or function, or leave the counter unchanged for a unary operator or function. For binary operators it recurs over the first operand and loops over the second operand. The loop is exited when the counter reaches 0.

As with assembly language, you can save code space, return-address stack space and execution time in BASIC by replacing a code sequence such as "GOSUB 90: RETURN" with "GOTO 90": The return from subroutine 90 then also returns from the subroutine that calls it.

Calculus Demon^{tm} is accompanied by a similar program named PolyCalc^{tm} that is specialized to polynomials and a program named AlgiCalc^{tm} that is specialized to univariate rational expressions in X, including an option for approximate fully-factored form. All three programs can be downloaded free from [2].

A major goal of PicoMath, Calculus Demon, PolyCalc and AlgiCalc was to demonstrate to calculator manufacturers the feasibility of building computer algebra into a widely marketed calculator. As a demonstration prototype, Calculus Demon was adapted to the BASIC on the HP 75C calculator [12], but never distributed. However, Hewlett-Packard did later build computer algebra into their HP 28C calculator [11] and some subsequent calculators.

Calculus Demon demonstrated the feasibility of using a Polish expression stack to implement general-purpose computer algebra.

However, using floats to represent variables and type tags and using BASIC as an implementation language was very suboptimal. It is a waste of memory and computing time to devote an entire floating-point number to representing one-letter variables, tags, integer indices etc. For this purpose, even assembly language is easier, with its symbolic labels and constants, integer variables, and built-in instructions for pushing data onto the return-address stack or popping it off.

The remember-stack mechanism was later implemented using the C programming language for the TI-92 introduced in 1995 and for some subsequent TI calculators, together with Windows and Macintosh computers [6]. Instead of floats, the units of storage on the expression stack are 8-bits on the Motorola 68000-based calculators, but they are 32 bits on the Macintosh version of TI-NSpire or 16 bits on the Windows version and on the ARM-based handheld version.

With its parametrized functions, local variables and integer arithmetic for implementing unlimited precision rational arithmetic, C is a luxurious programming language compared to BASIC. The TI-89/TI-92 Plus Developer Guide [30] gives some implementation details. The computer algebra capabilities are significantly more than Calculus Demon — roughly comparable to *Derive*, but with significant differences too.

Some advantages of Polish representation of expression trees are:

- No space is wasted on pointers within an expression.
- The data is contiguous, which improves speed for caches and virtual memory.
- The data is relocatable. (For this reason, and the fact that it doesn't require parsing, it is also more efficient than infix character strings for serialization in computer algebra systems that use linked storage for doing transformations.)
- It isn't necessary to implement garbage collection or reference counts.
- There are no annoying pauses for garbage collection during plotting, etc., making the technique particularly suitable for real-time applications.

- Unlike garbage collection, the percentage of time devoted to memory reclamation doesn't increase as the amount of reclaimed memory decreases.

To save some program space and execution time, the program uses pointers rather than indices that are added to the base address of an array. A pointer to the top of the expression stack is stored in a global variable named `top_estack`. When data that is no longer needed is on the top of the expression stack, that memory can be reclaimed in negligible constant time by simply assigning to `top_estack` an address that was saved earlier into a local variable.

When a block of memory strictly within the expression stack is no longer needed, it can be reclaimed by shifting the still-needed portion above it down in time $\Theta(n)$, where n is the number of memory units being shifted. This tends to require significantly less time than was required to create the contents of those memory locations.

Pointers to any expressions of remaining interest above the deleted portion must be decremented by the amount of shift, which requires time proportional to the number of such pointers. Such adjustments tend to be infrequent for recursive algorithms but more frequent for looping algorithms that push expressions onto the expression stack in an irregular sequence. Thus although there is an initial code space savings for not having to implement garbage collection or reference counts, the code size might grow faster per implemented feature. Moreover, storage reclamation is a distributed responsibility of the implementer rather than a larger initial programming time and program space investment followed by less subsequent concern.

Moenck [19] proposes a similar data structure, except that immediately below each binary operator is a displacement field that, when subtracted from the address of the displacement field, points to the deeper operand. Using displacements rather than pointer fields preserves the valuable relocatability. At the expense of requiring more space for expressions and more code to consistently manage the displacement fields, this technique might make the system faster by providing quicker access to the deeper operand. However, many functions that process the first operand can be written to return the address of the expression below that operand. Even when that is inconvenient, the time required to traverse the shallower operand to reach the deeper operand is often significantly less than the time required to process the shallower operand.

With a remember stack, any number of pointer variables from a running program can point to the same expression or sub-expression in the expression stack. However, the expression stack itself ordinarily doesn't contain pointers to expressions or sub-expressions. In this sense there is no sharing *within* the expression stack of common sub-expressions therein.

In contrast, a system implemented with linked storage can more thoroughly share common sub-expressions. For example, Lisp programs can fortuitously share some common sub-expressions — particularly if the data structure and algorithms are implemented with that as one of the goals. For example when recurring over the terms of two polynomials to add them, when one polynomial operand becomes empty, the partial result can be the pointer to the remaining terms in the other polynomial, thus sharing between that operand and the result those terms together with the CONS cells that glue them together. As another example, when multiplying two terms such as xy^2 times xz, a pointer to the existing y^2 can be used in the result x^2y^2z rather than forming a separate duplicate y^2 by computing y^{2+0}. Moreover, the implementation can more forcefully share some sub-expressions. For example in the Reduce computer algebra system, there is a protocol so that functional forms such as $\ln(x)$ and $\sin(\ln(x)+y)$ are stored uniquely, which saves space for duplicate instances and makes fast address comparison sufficient for recognition of EQUAL functional forms.

A system such as Maple [3] or HLisp [8] that also hashes sub-expressions can, at the expense of some initial time investment, share even more common sub-expressions and make recognition of identical expressions faster. Thus a remember stack and hashing with pointers are at the opposite ends of a spectrum with respect to the sharing of common sub-expressions. The amount of possible sharing depends strongly on the data structures and specific examples. Table 2 in [28] implies that for Lisp the number of Cons cells ranges from 2% to 100% of the number required if there were no sharing, with an average of 67% over all of the examples and alternative data structures. The break-even point for the relative space efficiency of an expression stack versus pointers with sharing depends on the relative sizes of tokens and atom representations in the expression stack *versus* pointers and atoms for Lisp.

4.3 Implementing linked-data computer algebra compactly

The early computer algebra systems MathLab, Reduce, ScratchPad and Macsyma were all implemented in Lisp. This helped inspire the use of Lisp as an implementation language for muMath and *Derive*.

The Lisp programming language and close relatives such as Scheme come with some important support for computer algebra already built in:

- They include a built-in garbage collector — one that tends to be fast even for very small blocks of memory such as a small-magnitude integer or a Cons cell consisting of two pointers.
- They include arbitrary-precision rational arithmetic that automatically adapts to whatever memory block sizes are necessary for each individual number.
- They include a built-in association list that makes it easy and fast to store and query information about an indeterminant, operator or function, such as its type, parser precedence, arity, associativity, commutativity, linearity, and symmetries.
- They include an interpreter that permits run-time definition of new functions, and the interpreter can be bootstrapped into interpreting mathematical expressions and accepting definitions of new mathematical functions.
- They include efficient powerful mapping functions that factor out common processing overhead such as looping or recurring over lists, thus reducing computation time and program size while increasing program comprehensibility.
- They are typically quite efficient at executing recursive function calls, which are prevalent for implementing most computer algebra.

Lisp is more suitable than any of the previously discussed methods for implementing computer algebra. Logic programming languages such as Prolog might be even better matched to the task — particularly for the highly desirable better integration of computer algebra with logic.

For languages such as C or C++ that lack most of these features, the features must be programmed by the computer algebra implementers.

4.3.1 *muLisp*^{tm} for the Intel 8080 and Zilog Z-80

Full implementations of the Common Lisp standard entail a very large run-time footprint, making them less suitable for compact computer algebra. However, Common Lisp includes many features that are not crucial for computer algebra. Before Common Lisp was defined, Albert Rich wrote a very compact Lisp interpreter named *muLisp* — initially for the Intel 8080 micro-processor [24]. The initial 2 kilobyte version was written in machine language and was entered into a S-100-bus computer built from a kit [15], using front-panel toggles for each bit of the address and each bit of the instructions.[1] At that time the computer had 4 kilobytes of RAM. A monochrome character-oriented monitor also built from a kit was used for the interactive Lisp I/O.

Subsequent versions added audio-tape cassette backup of the Lisp interpreter and Lisp programs. The cassette drive was later replaced with a dual-floppy drive storing about 128 kilobytes per floppy disk. At that time we also began using the CP/M^{tm} operating system, which included an assembler, loader, machine-language debugger, and line-oriented editor. The Imsai 8080 RAM was gradually increased to the 64 kilobyte address space using hand-soldered circuit boards. The interpreter was rewritten in assembly language and enhanced to contain arbitrary-precision rational numbers.

muMath was the first computer algebra system implemented in *muLisp*. Several unique features of *muLisp* contributed significantly to the compactness of muMath and its successor, *Derive*:

1. In most Lisps, evaluating the CAR or CDR of an atom provokes a run-time error. Consequently these most-frequently invoked functions incur the speed penalty of run-time checks. Instead, in *muLisp* it is allowable to evaluate the CAR or CDR of an atom because:

   (a) The CAR of a number points to itself, and the CDR points to a symbol designating its sign and type (small integer, big integer or reduced fraction). Numbers have two additional fields containing a small integer or containing the number of bytes and a pointer to the contiguous bytes of a large unsigned binary integer, or containing pointers to two atoms that are the numerator and denominator of a rational number. However, these two additional fields aren't accessible via CAR or CDR. Moreover, small integers are hashed to avoid duplicate instances of small magnitude integers that commonly occur in expressions and to make the EQ and EQUAL functions faster for them.

   (b) The CAR of a symbol points to its value (perhaps itself), and the CDR points to the symbol's property list (perhaps NIL). There are two additional fields pointing to its function definition (or to NIL) and to its print-name string, but these two fields aren't accessible via CAR or CDR.

   (c) This *closed pointer universe* makes it impossible to crash the interpreter or operating system by inadvertently taking the CAR or CDR of an atom, despite the lack of costly run-time checks.

2. Lisp has an interactive driver loop that prompts for an input expression, reads it, evaluates it, then displays the resulting value. In most Lisps if you attempt to evaluate a symbol such as x that hasn't been assigned a value, it provokes an error interrupt. With such Lisps you must use a quote macro, such as in 'x to avoid that error interrupt. In computer algebra, if a symbol doesn't have an assigned value, then we want the atom to evaluate to its own name. *muLisp*

[1] The IMSAI 8080 was a clone of the pioneering MITS Altair.

uses this auto-quoting method. Therefore the machine-language *muLisp* APPLY and EVAL functions can be used directly for computer algebra. There is no need to write additional slower versions merely to obtain the auto-quoting needed for computer algebra. This also saves space and execution time throughout the computer algebra program because the quote can usually be omitted for symbols.

3. In *muLisp*, symbols and strings are the same thing. For example, there are functions for concatenating and extracting sub-strings from the names of symbols. This makes the *muLisp* interpreter more compact, and it can make *muLisp* programs more compact because only one copy is stored for duplicate instances of symbols, hence also for strings.

4. Excess trailing formal parameters for which corresponding arguments aren't provided are treated as local variables, initialized to NIL. This avoids expanding and interpreting a separate LET macro, and it is an efficient method of providing a variable but limited number of arguments. (There are also *no-spread* functions that bundle an unlimited number of arguments into one parameter as a list of arguments.)

5. In computer algebra implementations almost every function body is a sequence of statements containing assignments, conditional statements and loops that themselves contain such sequences. In Lisp, PROGN is comparable to braces in C, and COND is comparable to the C construct "if (...){...} else if(...){...},... else{...}". In *muLisp*, function bodies are implicit PROGNs that optionally contain implicit CONDs, making these functions rarely needed: Every function body is a sequence of one or more forms: $form_1, form_2, \ldots form_n$. The returned value is the last value that is computed before exiting the function. Beginning with $form_1$ and proceeding sequentially:

   (a) If $form_i$ is an atom, then it is simply evaluated.

   (b) If instead the CAR of $form_i$ is an atom, then the function having that name is applied to the values of the remaining elements in $form_i$.

   (c) If instead the CAAR of $form_i$ is an atom, then it is an implicit COND: The function having that name is applied to the values of the remaining elements in the CAR of $form_i$. If the result is NIL, then the remaining elements of $form_i$ are ignored and execution proceeds to $form_{i+1}$. Otherwise $form_{i+1}$ through $form_n$ are abandoned, and execution proceeds to the remaining elements of $form_i$ as the alternative to those abandoned forms.

   (d) If instead the CAAR of $form_i$ isn't an atom, then the forms in $form_i$ are those of a nested implicit PROGN, treated as described here for forms $form_1$ through $form_n$, after which $form_{i+1}$ is evaluated instead of leaving the outer implicit PROGN that is the function body.

   Another way to regard a function body is that whenever $form_i$ begins with two left parentheses, then $form_i$ is a conditional branch that will abandon $form_{i+1}$ through $form_n$ if $form_i$ is non-NIL. Whenever $form_i$ begins with three left parentheses, then $form_i$ is a similar sequence of forms that will be executed or partly executed, after which execution continues with $form_{i+1}$. For example, here is a traditional Lisp recursive factorial function followed by a faster more compact one that exploits the *muLisp* implicit COND:

```
(Defun Factorial (N)
  (Cond
    ((Zerop N) 1)
    (T (* N (Factorial (Sub1 N)))) ) )

(Defun Factorial (N)
  ((Zerop N) 1)
  (* N (Factorial (Sub1 N))) )
```

muLisp also has PROGN and COND for compatibility with other Lisps, but these functions are rarely used in the implementation of muMath and *Derive*.

muLisp has an efficient general looping function (LOOP $form_1, form_2 \ldots, form_n$), where any number of the forms beginning with two left parentheses are implicit CONDs that conditionally exit the loop. For example, here is a traditional Lisp iterative factorial function and a faster more compact *muLisp* alternative that uses LOOP together with implicit PROGN and implicit COND:

```
(Defun Factorial (N)
  (Prog (M)
    (Setq M 1)
    A (Cond
        ((Zerop N) (Return M)) )
      (Setq M (* M N))
      (Decq N)
      (Go A) ) )

(Defun Factorial (N
                  M)
  (Setq M 1)
  (Loop
    ((Zerop N) M)
    (Setq M (* M N))
    (Decq N) ) )
```

There are also rarely-used macros that define the Common Lisp DO, DO*, DOLIST and DOTIMES looping functions in terms of LOOP.

1. *muLisp* uses dynamic shallow binding rather than lexical deep binding. Dynamic shallow binding is faster and more appropriate for interpreted programs. Although there was a compiler for the Intel 8080 version of *muLisp*, it was used only for a few appropriate speed critical functions because it typically increases code size by a factor of 2 to 3 while increasing speed of those functions by a factor of at most 2 to 3. The reason is that the *muLisp* interpreter is unusually space and speed efficient. The optionally loadable Flavors package provides lexical binding if desired.

2. CONS cells, symbols, their print names, numbers and their binary data are each stored in a separate contiguous area so that type checking can be done by mere comparisons with the boundary addresses between these regions. This is fast and it avoids wasting space in each number, symbol and CONS cell for a type tag.

3. The garbage collector compacts each of these data types toward one or the other end of each region, and pairs of data types grow toward each other, as do the return-address and argument stacks. Therefore garbage collection isn't triggered unless both members of a pair collide. Moreover, if collection results in relatively much more free space in one of the areas than another, the amount of memory allocated to each area is reapportioned to balance the free space proportionate to the employed space of each type. This way, computation can proceed until almost all of the memory is being used. The data that 16-bit pointers address all align on an even byte address, so the least significant bit of the address is used as the temporary marker bit for the mark-and-sweep garbage collection, avoiding the need for separate mark-bit space while allowing pointers to all of the even addresses in the full 64-kilobyte address space.

4. Function definitions ordinarily remain unchanged after they are debugged. Therefore it is worth spending some effort to make the stored form of function definitions more compact and faster to execute. Consequently:

   (a) Function definitions are CDR-coded: With the exception of an *in situ* quoted dotted pair for which the CDR points to a non-NIL atom, function definitions are comprised of atoms and nested lists. Therefore these lists in function bodies are represented as arrays of pointers rather than as linked CONS cells. Rather than having an extra pointer to NIL at the end of the array or having a field containing the number of elements in an array, the least significant bit of the last pointer is 1 if and only if it points to the last element. (Of course that bit is masked out before following the pointer.) This CDR coding approximately halves the storage space for the function, and it executes faster too because access to the CDRs is faster.[2]

   (b) Starting from the innermost expressions, if two or more pointers in a function definition point to syntactically identical code fragments, then only one copy of that data is stored, to be pointed to by all of the instances. For example even if the code fragment (EQ (CAR ARG1) COS) occurs more than once within a function definition, only one copy is stored, and recursively, if (CAR ARG1) occurs in other contexts in the function, then those instances are also shared. This *condensing* saved significant additional space. Moreover, the sharing isn't limited to entire lists: Lists having different head elements can share identical tails.

   (c) While the control variable *CONDENSE* is nonNIL, this condensing process is done *between* functions as well as within them, saving additional space if the same code fragment occurs in more than one function definition. Also, functions defined under these circumstances are excluded from garbage collection, making the collection faster. For this reason and the additional time required to load functions in this mode, it is

[2]Bytecode is an oft-used competing way to encode algorithms compactly. It has been used for many programming languages from BCPL on, including Lisp and Java.

used only after a file of functions is debugged, hence ordinarily subject to no further redefinition.

5. (ZAP-STRING *symbol*) frees the memory required to store a *symbol*'s print name by converting its print name to the null string. Thus a significant amount of space can be saved by zapping the print-names of internal functions and control variables as the last step in a program definition.

6. Early versions of the interpreter used some of the Intel 8080 one-byte reset instructions for some of the most commonly-occurring Lisp functions, such as CAR and CDR. Unfortunately, later versions of operating systems and other co-resident programs rudely clobbered these memory locations in a disorganized free-for-all without saving then restoring them after use. Consequently this idea was abandoned in later versions of *muLisp*.

7. Macros are used to conditionally exploit the more compact Z-80 relative jumps when assembling for the TRS-80 computer.

4.3.2 muMath

muMath [22, 26] also employs some techniques that make the code faster and/or more compact:

- The first file loaded during a build bootstraps a Pratt parser by defining the PARSE function, then storing the right and/or left binding powers of operators, etc. on their property lists. These definitions are co-mingled with using those newly-defined operators, delimiters, and control constructs in subsequent functions and expressions in the file as soon as they are available. Pratt Parsers [5, 21] are quite compact and particularly suitable for adding syntactic extensions during run time. The beginning of the file is muLisp, which steadily morphs into the the more Algol-like muSimp surface programming language by the end of the file. For example, the implicit COND is invoked by the explicit muSimp syntax

  WHEN *condition*; $expression_1$; $expression_2$; ...; EXIT;

  Once parsed, function definitions and mathematical expressions are just as space and speed efficient as *muLisp*. There is still only one level of interpretation, or not even that for the few functions that are compiled.

- The implementation uses a sort-of poor man's object-based style: Many algorithms that operate on mathematical expressions are implemented incrementally by putting on the property list of the function or operator name several small functions, each for handling a different kind of operand or combination of operands. For example to implement differentiation there is a function for differentiating numbers, another function for differentiating variables, another function for differentiating sums, etc. This makes it easy for implementers and users to enhance functions and operators without doing delicate surgery on a monolithic function that handles all cases. For example, a user can easily supplement the differentiation algorithms with differentiation of Bessel functions. The dispatching is handled by *muLisp*, and it is quite fast because it uses the assembly-language *muLisp* ASSOC function rather than interpreted implicit CONDs to determine which case is applicable. There are separate functions for putting new information at either the near or far end of an association list so that the most frequently used information is near the beginning of the list.

- Each of the *muLisp* arithmetic operators invokes an associated trap function to optionally catch error throws for non-numeric arguments. Using the above object-based style, the corresponding muSimp trap functions dispatch to various functions such as adding a variable to a number, a sum to a product, a logarithm to a logarithm, etc. Because of this organization, the most common case of combining numbers with numbers is tried first in machine language rather than after an interpreted test.

muMath has no approximate arithmetic, hence no function plotting capability. However, Microsoft licensed muMath for the Radio Shack TRS-80, for which they added function plotting by tying into their Microsoft BASIC and its floating-point arithmetic in ROM.

Microcomputer algebra became luggable in 1981 when muMath was bundled with the Osborne 1 "suitcase" computer [20].

4.3.3 *muLisp* for the Intel 8086

Implementing *muLisp* for the Intel 8086 was more challenging because the 1-megabyte address space was segmented into 64-kilobyte segments. The MS-DOS operating system reserved 360 kilobytes of that address space for its purposes, leaving 640 kilobytes for implementing and using *muLisp*. Awkward segmented architecture is becoming less common as memory costs decline. However, techniques for overcoming the limitations of short pointers are still relevant for implementing linked programs and data compactly.

For generating addresses there are four 16-bit *segment registers* whose contents are shifted left 4 bits then added to a 16-bit offset. One of these segment registers is used for instructions, one for two stacks, one for data, and one for an "extra" segment that is effectively a second data segment. *muLisp* uses one of the stacks in the stack segment for return-address offsets and the other for argument-address offsets, with the two stacks growing toward each other. This is a cleaner design than co-mingling return addresses and argument offsets in one stack.

Following a suggestion of Zippel [31], the 16-bit CARS are stored in an area pointed to by the Data segment register and the 16-bit CDRS are stored in an area pointed to by the Extra segment register, thus doubling the number of cells that would be available if the CAR and CDR were stored adjacent to each other in the same segment. The CARs and CDRs of numbers are below those of symbols, which are below those of CONS cells, to permit fast compact address typing, with the numbers and symbols growing toward each other. The data for the other two fields in number table and symbol table entries are in different segments.

There can be up to 6 code segments containing a maximum of about 55 kilobytes each. The first offset in each function definition is an offset to the atom LAMBDA, and the interpreter can detect that the current code segment is no longer appropriate by testing for this. When such a code-segment fault is detected, the interpreter tries each of the other code segments until LAMBDA occurs. The interpreter inserts gaps between function definitions if necessary so that only one code segment has LAMBDA at that offset.

An effort was made to sequence the function definitions according to locality of reference to reduce the frequency of code-segment faults and to put the most frequently used functions in the segments that are tried first after a code-segment fault. This effort also includes a function that can force a new code segment at a desired point in the source code, rather than letting it happen where it became unavoidable.

4.3.4 *Derive* for MS-DOS

muMath has a teletype style interface similar to that of the MS-DOS operating system under which the Intel 8086 version ran. The window and mouse based interface of the Macintosh made computing attractive and accessible to a far larger audience. Accordingly, to reach a larger audience, Soft Warehouse released *Derive* as an MS-DOS successor to muMath in 1988. *Derive* included its own windowing system implemented in *muLisp*.

Microcomputer algebra achieved compact cordless portability in 1991 when a special ROM version of *Derive* was released for the HP 95LX palmtop computer [13].

Derive also included function-plot graphing. We needed approximate arithmetic to do this. Since we already had arbitrary-precision rational arithmetic, the most compact way to implement approximate arithmetic was to round rational numbers that had excessively lengthy numerators and denominators to simpler rational numbers, using the mediant rounding described by Matula and Kornerup [18]. There was also the option of having magnitudes less than a user-set value underflow to 0. For very little additional code this technique gave us adjustable-precision approximate arithmetic as a mode. Moreover, mediant rounding has the nice property for computer-algebra that the capture interval around simple fractions is larger than around more complicated fractions, thus increasing the chances of recovering a simple exact rational result despite intermediate approximations. Also, this approximate rational arithmetic kept the implementation more compact because there wasn't a separate approximate-number type to test for and manage. It would be more informative to keep the information that a rational number might not be exact and optionally indicate that to the user, but we didn't.

For this mediant-rounding arithmetic:

- Rounded multiplication and division average about three times as slow as exact rational arithmetic using the same rational operands.
- Rounded multiplication and division average several times slower than for variable-precision binary floating-point arithmetic at comparable precision.
- Unlike variable-precision floating-point, addition is about the same speed as multiplication using the same operands, and grows quadratically rather than linearly with the number of words of precision.

The approximate arithmetic entailed also implementing algorithms for approximate fractional powers and elementary functions. For this purpose it was helpful to develop algorithms that, as much as practical, separately developed an approximate integer numerator and denominator of a result, perhaps shifting them both right by the same amount several times during the process as a fast approximate rounding, with one mediant rounding only at the end. This was typically much faster than simply using the usual floating-point algorithms but with many mediant roundings along the way. In fact, unlike addition, multiplication and division, the speed of fractional powers and the elementary functions was typically slightly *faster* than for variable-precision floating-point arithmetic. This technique of working mostly with large integers and shift-rounding pairs of them during intermediate steps might benefit implementation of irrational operations in arbitrary-precision floating-point arithmetic too.

Installation of the Intel 8087 floating-point co-processor was rare at that time, and about 6 significant digits was sufficient for most plotting purposes. Therefore the plot speed compared

acceptably to the then-prevalent 8 to 16 digit binary or binary-coded decimal fixed-precision software floating point.

Another difference between muMath and *Derive* is that although the one-level evaluation of Lisp and *muLisp* is sufficient for computer algebra, it isn't ideal. One-level evaluation works well for computing the values of formal parameters and local variables, but infinite evaluation is more natural for global variables. For example, in muSimp, if you enter the assignment $z := x + 5$; then the assignment $x := 7$; then enter z, it will still have the value $x + 5$, rather than 13. There was a way to force extra evaluation levels, but users expected that to be done automatically.

For functions such as differentiation, users prefer a sequence such as

$$x := 7;$$

$$\frac{d}{dx}x^2;$$

to produce 14 rather than an error caused by differentiating 49 with respect to 7. In other words, they want delayed substitution of assigned values in some circumstances.

Therefore, *Derive* uses special computer-algebra oriented EVAL and APPLY function that are written in *muLisp* rather than assembly language.

muLisp and muSimp both had separate optional windowing, editor and debugging environments, and analogous functions had different names. For example, CAR in *muLisp* was FIRST in muSimp. It was a burden to maintain these separate but parallel development environments, so *Derive* was written in *muLisp* rather than muSimp.

As indicated in the introduction, compact and fast data representation is secondary to a compact computer-algebra implementation for the limited address space targets of muMath and the Intel 8086 version of *Derive*. However, *Derive* also uses particularly compact data representations that facilitate fast processing. Reference [29] describes the recursive partially-factored semi-fraction form predominantly used by *Derive*, but without details about the representation of the recursive multinomial factors therein. Reference [28] compares the speed and space requirements for several alternative multinomial data structures, with observations about the relative compactness of corresponding polynomial co-distribution algorithms. Shortly after writing that article, I implemented two variants of the sparse recursive representation that were usually faster and more space efficient than the alternatives reported there. Through version 5, *Derive* used only the following variant:

- Expressions whose top-level function or operator is anything other than "+"', "·" and "^" are represented as *functional forms* that are lists starting with the function or operator name followed by arguments that are general expressions.

- A *power* is a non-zero rational numeric exponent dotted with a general expression that is the base. It requires only one CONS cell beyond any in the base, and it is easily distinguished from functional forms, for which the CAR is a symbol rather than a number. The exponents can be negative (representing denominators) and/or fractional.

- A *product* is a power dotted with a non-zero general expression. It is easily recognized as a product because it is not an atom and its CAR is not an atom, but its CAAR is numeric. The most main of the two factors is the CAR of the dotted pair. The product requires only one CONS cell beyond those of the leading power and its cofactor.

- A *sum* is a leading term that is a product, dotted with a non-zero reductum that is a general expression. It is easily recognized as the only remaining possibility for a general expression or as a non-atomic expression whose CAR and CAAR aren't atomic. It requires only one CONS cell beyond those of the reductum and the product that is the leading term.

This representation can be categorized as sparse recursive with implicit binary "+", "·", "^", and *variables in*. Reference [29] describes the ordering of terms and factors, together with how partial factoring and semi fractions are accommodated in this form. This is completely general and quite flexible, which is needed for the partially-factored semi-fraction form because it tends to have relatively few terms in any one sum, with powers, products and sums occurring almost anywhere.[3]

To make the implicit "+" and "·" recognizable, we must pad a base with an exponent of 1 if the base is a factor in a product and isn't the last factor therein; and we must pad a non-product term with a coefficient of 1 if it isn't the last term in a sum. However, this is significantly less padding than is usually required to use implicit operators. In contrast, for example, if a top-level expression isn't a number or a variable, the closest representation in [28] must construct a list with a POL tag followed by the non-atomic polynomial data structure in which at every recursive level we:

- always force variables to have an exponent even if it is 1;
- always force non-numeric terms to have a coefficient even if it is 1; and
- always force non-numeric polynomials to have a reductum even if it is 0.

At the deepest recursive levels where most of the CONS cells occur, polynomials often don't have a numeric term or are only one term that is perhaps merely a variable or a functional form rather than a non-unit power thereof. Therefore the space savings and associated time savings are significant for the *Derive* representation.

A dense univariate representation is also used for the special purpose of computing approximate univariate polynomial zeros.

In version 6, an even faster expanded sparse recursive representation was used for a few special purposes. Besides implicit operators, it has *stratified* factoring out of the variables, with one representation of the variable at each recursive level. It is similar to but not identical to the Maxima Canonical Rational Expression form [7].

4.3.5 *muLisp* for the Intel 386

The Intel 386 was the first processor in the x86 family that supported a full unsegmented 32-bit address space. Therefore the architecture of the 386 version of *muLisp* was closer to the simpler Intel 8080 version, but with 32-bit addresses rather than 16-bit addresses.

4.3.6 32-bit *Derive* for Windows

The 32-bit version of the Microsoft Windows operation system was finally becoming a widespread competitor to the much earlier Macintosh windows interface.

Accordingly, Albert Rich and Theresa Shelby designed a user-interface of *Derive for Windows*, which Theresa implemented in C++. For mathematical services such as simplifying, solving, or

[3] A *semi-fraction* is a sum of an optional extended polynomial and any number of proper ratios of extended polynomials having denominators whose mutual gcds are numeric. Unlike partial fractions, which semi-fractions include, the denominators do not need to be square free or irreducible.

integrating mathematical expressions, the interface called the *Derive* math engine, which was written in the 32-bit version of *muLisp*. The interface required more code space than the computer algebra, so compact computer algebra became less relevant for that platform.

By that time IEEE floating-point hardware was becoming widespread, and *Mathematica*(R) made apparent the marketing appeal and mathematical value of fast high-resolution 3D surface plots. Consequently, we wrote a *muLisp* function that produces a contiguous *reverse* Polish representation of an expression, together with a C evaluator function that can evaluate that representation for floating-point values of the variables therein. The Plot functions all invoke this translator from Lisp to reverse Polish, then repeatedly invoke the C evaluator function to quickly approximate the reverse Polish expression at plot points.

Adjustable-precision approximate rational arithmetic is still used for all other approximate arithmetic purposes in *Derive*, such as quadrature and approximate equation solution.

5 Summary

Despite decreasing memory costs, there are still good reasons for implementing programs compactly. Compact computer algebra has a long and varied history, with ideas that are relevant to implementing other kinds of programs compactly too.

Special-purpose methods include

- implementing univariate polynomial and series algebra using dense coefficient arrays, and
- finessing computer algebra via evaluation and interpolation.

General-purpose methods include

- string-based computer algebra,
- contiguous stack-based computer algebra, and
- various methods for making linked-data computer algebra more compact.

Acknowledgments

Albert Rich is the primary author of *muLisp*. He and David Stoutemyer are the primary co-authors of muMath. They and Theresa Shelby are the primary co-authors of *Derive*.

References

[1] W.S. Brown, On Euclid's algorithm and the computation of polynomial greatest divisors, *Journal of the ACM* 18(4), pp. 478-504, 1971.

[2] Calculus Demon, PolyCalc and AlgiCalc, 1981, `http://www.atariarchives.org/APX/showindex.php`

[3] B.W. Char, K.O. Geddes, W.M. Gentleman and G.H. Gonnet, The design of Maple: A compact, portable and powerful computer algebra system, in *Computer Algebra*, Lecture Notes in Computer Science 162, Springer-Verlag, pp. 101-115, 1983.

[4] G. Cioni and A. Miola, Using minicomputers for algebraic computations, *ACM SIGSAM Bulletin* 10(4), pp. 50-52, November 1976.

[5] D. Crockford, Top down operator precedence, 2007, `http://javascript.crockford.com/tdop/tdop.html`

[6] TI-92, TI-89, Voyage 200, TI Nspire, DataMath^{tm} Calculator Museum, 2008, `http://www.datamath.org/`

[7] Free Maxima download, http://maxima.sourceforge.net/download.html

[8] E. Goto and Y. Kanada, Hashing lemmas on time complexities with applications to formula manipulation., *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pp. 154-158., 1976.

[9] P. Henrici, *Computational Analysis with the HP 25 Pocket Calculator*, Wiley. 1977.

[10] F.B. Hildebrand, *Introduction to Numerical Analysis*, 2nd edition, McGraw-Hill, pp. 494-502, 1974.

[11] HP 28C calculator photograph and history, 1987, `http://www.hpmuseum.org/hp28c.htm`

[12] HP 75C calculator photograph and history, 1982, `http://oldcomputers.net/hp75.html`

[13] HP 95 LX photograph and history, 1991, `http://en.wikipedia.org/wiki/HP_95LX`

[14] IBM Scientific Subroutine Package, 1965, `http://www-03.ibm.com/ibm/history/exhibits/1130/1130_facts4.html`

[15] IMSAI 8080 photograph and history, 1975, `http://en.wikipedia.org/wiki/IMSAI_8080`

[16] J.G. Koomey, Worldwide electricity used in data centers, *Environmental Research Letters*, 3, 2008, and `http://uowblogs.com/digcjew994/files/2010/04/Koomey.pdf`

[17] D.E. Knuth, *The Art of Computer Programming, Volume 2*, Addison-Wesley, Sections 4.6 and 4.7, 1969.

[18] D.W. Matula and P. Kornerup, Finite precision rational arithmetic: slash number systems, IEEE Transactions on Computers, C-34 (1), pp. 3-18, 1985.

[19] R. Moenck, Is a linked list the best storage structure for an algebra system?, *ACM SIGSAM Bulletin* 12 (3), pp. 20-24, 1978.

[20] Osborne 1 computer photograph and history, 1981, `http://en.wikipedia.org/wiki/Osborne_1`

[21] V.R. Pratt, Top down operator precedence, *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 41-51, 1973.

[22] A.D. Rich, and D.R. Stoutemyer, Capabilities of the muMath-78 computer algebra system, *Symbolic and Algebraic Computation*, Lecture Note 72, Springer-Verlag, pp. 241-248, 1979

[23] P.J. Smith, Symbolic derivatives without list processing, *Communications of the ACM* 8(8), pp. 494-496, 1965.

[24] Soft Warehouse Inc., *muLisp 90 Reference Manual*, 1990. (Out of print.)

[25] D.R. Stoutemyer, Symbolic mathematics on a programmable hand-held calculator, An unpublished talk presented at the annual SIAM Meeting, 1977.

[26] D.R. Stoutemyer, The muMath symbolic math system, *Creative Computing Magazine*, July and August, 1979.

[27] D.R. Stoutemyer,, PicoMath-80, an even smaller computer algebra package, *ACM SIGSAM Bulletin* 14 (3), August, Issue 55, pp. 5-7, 1980.

[28] D.R. Stoutemyer, Which polynomial representation is best? Surprises abound!, *Proceedings of the 1984 Macsyma User's Conference*, General Electric, Schenectady N.Y., pp. 221-243, 1984.

[29] D.R. Stoutemyer, Ten commandments for good default expression simplification, *Journal of Symbolic Computation* 46, pp. 859-887, 2011.

[30] Texas Instruments, *TI-89/TI-92 Plus Developer Guide*, 2001, `http://education.ti.com/educationportal`

[31] R.E. Zippel, A MACSYMA Chip?, *Proceedings of the 1979 Macsyma Users Conference*, pp. 215-221, 1979.

[32] R.E. Zippel, Probabilistic algorithms for sparse polynomials, in *Symbolic and Algebraic Computation*, editor E.W. NG, Lecture Notes in Computer Science 72, Springer-Verlag, pp. 227-239, 1979

[33] Zonk, Internet uses 9.4% of electricty in the US, `http://hardware.slashdot.org/story/07/09/27/2157230/Internet-Uses-94-of-Electricity-In-the-US`