Expression Optimization using High-Level Knowledge

Mark P.W. Mutrie
Bruce W. Char
Richard H. Bartels

Research Report CS-87-09
February 1987

Expression Optimization using High-Level Knowledge

Mark P.W. Mutrie
Bruce W. Char
Richard H. Bartels
*

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

February 18, 1987

Abstract

Combining symbolic algebra with numerical computation has become an effective way of solving many scientific and engineering problems. One of the difficulties in practice is producing concise, efficient, stable code from the large expressions generated in symbolic algebra. Most of the existing optimization techniques are applied after an operation or algorithm has been performed. We introduce techniques using high-level knowledge which are to be applied while the output expressions are generated.

*This work was supported by grants A5471 and A5471 of the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

1.1 Rationale for Needing Optimization

Symbolic algebra and numerical computing techniques can be combined to solve problems in engineering and physics [Wir80,Wan86], but expressions generated using symbolic algebra systems for subsequent use in numeric codes tend to be very large [SR84,Bre84]. Translating these expressions directly into a computing language can lead to disappointing results. The code may be inefficient and numerically unstable. Furthermore, improving the code by hand is impractical because of its size. Consequently, an interface between symbolic and numerical computing facilities should include procedures for optimizing the number of arithmetic operations in the formulae and tools for analyzing the numerical stability of the generated code. The term optimization is used in the sense of *improving* the arithmetic operation count and compactness of the code rather than "the best possible".

Most of the existing optimization techniques are applied to the large output expressions *after* a procedure has been performed. The process applied to the input to produce the output expression is usually well-understood, although perhaps not by the user. We discuss an approach to expression optimization which uses knowledge about the operation or algorithm being performed, and the structure of the input expressions to generate concise output formulae that can be evaluated efficiently.

In Section 3 we present three examples in which optimization using high-level knowledge has been applied. The examples involve differentiation, Taylor series approximation, and integration operations. Within the process of this knowledge-based approach, we use some of the existing structure-determining and structure-reducing optimization techniques on small components of the larger problem. A brief summary of existing optimization techniques with references is given in Section 2.

2 Survey of Existing Optimization Techniques

Using Built-in Procedures The built-in procedures in symbolic algebra languages like Reduce, Macsyma and Maple can be used interactively to improve the quality of many codes. Steinberg and

Roache [SR84] and Cahill and Reeder [CR85] used built-in Macsyma procedures to interactively analyze and produce improved FORTRAN code for specific applications in physics.

Expression Analysis and Compression Hulshof and van Hulzen developed a Reduce package for interactively analyzing and restructuring expressions [vHH82] and a package for expression compression [HvH85]. After analyzing the expression(s), compression techniques using local factorization [Coo82] and controlled expansions based on work done by Brenner can be applied to produce a more compact representation.

Common-Subexpression Searching van Hulzen developed a common-subexpression searching procedure in Reduce [vH81,vH83], based on Breuer's algorithm [Bre69], that performs a common-subexpression search on multivariate polynomials. This approach to common-subexpression searching has also been implemented in Maple.

The `optimize` procedure in the Maple (version 4.0) library, developed by M. Monagan, makes use of "remember" tables [CGGW85, page 186], which are based on hashing, to identify subexpressions which are complete subtrees of an expression's parse tree. Only subexpressions that Maple simplifies as identical syntactic structures are recognized as common. In $(x+y+z)(x+y+f)$, $x+y$ is not recognized as a common component since it is only a part of the sum structure, whereas it is recognized in $(x+y)\exp(5(x+y))$.

Macsyma's `OPTIMIZE` command also employs a heuristic based upon hashing. During the descent of an expression tree, a hash code is formed for all subexpressions at a given level. A collision in the hash codes triggers a check for a common subexpression. Thus `OPTIMIZE` makes two passes through the expression – one to find and mark common subexpressions, and another to form the list of expressions for the result.

Macsyma's and Maple's optimization procedures, which we will refer to as "syntactic-optimization procedures", both make use of hashing to discover occurrences of common subexpressions in linear time. If expression-analysis and compression techniques are applied to an expression first, the result of a common-subexpression search is usually more compact than if only a common-subexpression search is performed.

Evaluation of Polynomials Given the polynomial $p(x) = a_o + a_1 x + a_2 x^2 + \ldots + a_n x^n$, Horner's rule is suggested as an economical way of evaluating the polynomial. Horner's rule is not always optimal [Fat69]; methods for preconditioning the coefficients can further improve the operation count. For example, preconditioning based on polynomial chaining, an extension of addition chaining, is given in Knuth [Knu81, pages 446-505]. These methods extend to multivariate polynomials. Horner's rule and preconditioning methods very often give good results, but their numerical stability depends on the polynomial's coefficients [MW67].

Other Techniques for Optimization The procedure `FORT`, due to Harten [Har84], uses the built-in functions `OPTIMIZE` and `HORNER` to produce improved FORTRAN code.

Brenner [Bre84] describes techniques for simplying large algebraic expressions which include segmenting the calculation in parts, and analyzing the structure so that rational simplification can be applied at appropriate points in the computation. By segmenting, he is able to reuse some intermediate results in the symbolic simplification process. His approach is implemented in Macsyma in a procedure called `LTAB`.

Paul Wang [Wan86] describes several techniques for improving the code produced in his symbolic system for the automatic generation of numerical programs for finite element analysis. These techniques, which include automatic intermediate-expression labelling, exploiting the symmetry of a problem using generated functions and subroutines, and interleaving formula derivation with code generation, could be useful in other applications. Intermediate-expression labelling is an illustrative example of Wang's methods. This method generates machine-created labels for certain intermediate expressions. These intermediate results are saved on an association list, which is checked before each computation to avoid reevaluating expressions. For an expression not on the association list, a label is generated and the expression-label pair is added to the list. For an expression already on the association list, the label is returned. This technique is much like Maple's option remember [CGGW85, page 169], but a label representing the result is returned instead of the actual result.

3 An Alternative Approach Using High-Level Knowledge

We present an approach which uses knowledge about the operations, and the structure of the input expressions to these operations, to produce a compact representation of the output. This technique does not necessarily preserve the original structure of the problem, but does exploit it to a great extent. Since the approach depends on knowledge about the operations being performed, we apply it to three examples – differentiation, Taylor series approximation, and integration – to illustrate how it works. These are representative of the (small number of) basic operations that all symbolic algebra systems use to create and transform expressions.

3.1 Differentiation

Solving systems of nonlinear equations is just one problem where differentiation can be an important part of the solution process. The generation of a Jacobian using a symbolic algebra system sometimes produces large output expressions that are difficult to understand, and must be transformed into a more concise and efficient form for use in numeric code [Spe80,Ked80,NC79].

Differentiation will very likely generate common subexpressions. Certain differentiation rules, such as the chain rule, also generate particular structures in the solution expressions. Rather than generating the solution, and then searching for a concise representation of the output, we could take advantage of the knowledge we have about the operation being performed and about the input expressions such as their syntactic structure and sparsity.

Consider computing the Jacobian of the following nonlinear problem [Ric83, page 241]:

$$\begin{aligned} h1 &:= \sin(xy+1)/(1+x+y) - 1 \\ h2 &:= \tan(xy+2)/(1-x+y) - 2. \end{aligned}$$

The Jacobian is:

$$\begin{bmatrix} \frac{\cos(xy+1)y}{1+x+y} - \frac{\sin(xy+1)}{(1+x+y)^2} & \frac{\cos(xy+1)x}{1+x+y} - \frac{\sin(xy+1)}{(1+x+y)^2} \\ \frac{\sec^2(xy+2)y}{1-x+y} - \frac{\tan(xy+2)}{(1-x+y)^2} & \frac{\sec^2(xy+2)x}{1-x+y} - \frac{\tan(xy+2)}{(1-x+y)^2} \end{bmatrix}.$$

We could then perform a Breuer-algorithm based common-subexpression search on the elements of this matrix which would find 16 common structures. As an alternative we could exploit our knowledge of the differentiation process, and the structure of the problem. In the given problem

we can use our knowledge of the division rule

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v\frac{du}{dx} - u\frac{dv}{dx}}{v^2} = \frac{1}{v}\frac{du}{dx} - \frac{u}{v^2}\frac{dv}{dx}.$$

The denominator v is a common subexpression in the final result.

In the given problem, the following structures were identified using Maple's op procedure to extract the components of the internal representation of the input expressions.

$$\begin{aligned} t1 &= xy \\ u1 &= \sin(t1+1) \\ v1 &= 1+x+y \\ u2 &= \tan(t1+2) \\ v2 &= 1-x+y \end{aligned}$$

The component $x+y$ could also be identified if a Breuer-algorithm-based common-subexpression search was used.

For lack of space, we will just consider calculating the $(1,1)$ and $(2,1)$ elements of the Jacobian corresponding to the derivatives of $h1$ and $h2$ with respect to x. We first calculate the derivatives of the components that we identified above. For assignment variable *var*, its derivative with respect to x is assigned to the variable *varp*.

$$\begin{aligned} t1p &= y \\ u1p &= \cos(t1+1)t1p \\ v1p &= 1 \\ u2p &= \sec^2(t1+2)t1p \\ v2p &= -1 \end{aligned}$$

The $(1,1)$ and $(2,1)$ elements of the Jacobian are given by

$$\begin{aligned} j11 &= \frac{1}{v1}u1p - \frac{u1}{v1^2} \\ j21 &= \frac{1}{v2}u2p + \frac{u2}{v2^2} \end{aligned}$$

where simple intermediate expressions representing simple constants have been eliminated. The components, their derivatives, and the result can be saved as elements of an expression table. The above approach could be used to generated the entire Jacobian; this approach produces an improved description of the output expressions, finding a similar number of common subexpressions as a Breuer-algorithm-based search, without requiring a search upon completing the differentiation process.

To further illustrate how this strategy can be used, consider taking the derivative of a product of 3 terms u, v, and w which are functions of the variable x. Using the product rule, the derivative is

$$\frac{d(uvw)}{dx} = uv\frac{dw}{dx} + uw\frac{dv}{dx} + vw\frac{du}{dx}.$$

By identifying the operation we are performing and the existing factors in the product, we see that the result contains the common subexpressions u, v and w. If the number of factors is greater than two, the product rule for differentiation will produce common subexpressions (as long as the components are independent of each other). The derivative of each factor can be determined and a common-subexpression search can be performed on the smaller problem, the set of three expressions $\{\frac{dw}{dx}, \frac{dv}{dx}, \frac{du}{dx}\}$, rather than doing a systematic search on the complete result afterwards.

If we consider a product of 4 terms, we can determine a more efficient representation than would be expected from just applying the product rule directly. Using the product rule:

$$\frac{d(uvwy)}{dx} = uvw\frac{dy}{dx} + uvy\frac{dw}{dx} + uwy\frac{dv}{dx} + vwy\frac{du}{dx}$$

where the original factors in the product u, v, w, and y are now common subexpressions. uv and wy could be identified as common subexpressions *afterwards* using a search based on Breuer's algorithm, but would not be found using syntactic optimization. On the other hand, we can take advantage of the structure during the differentiation process and eliminate the need for a common-subexpression search on the final result. Consider

$$\frac{d(uvwy)}{dx} = \frac{d(uv)(wy)}{dx} = uv\frac{d(wy)}{dx} + uwy\frac{d(uv)}{dx} = uv\left(w\frac{dy}{dx} + y\frac{dw}{dx}\right) + wy\left(u\frac{dv}{dx} + v\frac{du}{dx}\right).$$

uv and wy were identified as significant structures from the beginning of the process using knowledge about the operation and the expression. Assuming the components u, v, w, and y are evaluated

once, the direct use of the product rule requires 12 multiplications and 3 additions; using knowledge-based optimization, 8 multiplications and 3 multiplications are needed. This improvement has been achieved without the need for a Breuer-algorithm-based common-subexpression search. Greater improvements in efficiency and compactness of the code would be seen in large problems where one uses knowledge about several differentiation rules, and all the structures identified in the input expressions.

Similar strategies can be established for other differentiation rules involving exponentials, integer powers, and powers which are functions of x. Using the differentiation operation with this code improvement strategy includes steps to identify existing common subexpressions, factors in a product, the number of such factors, and the use of division and exponentiation, and then, steps to take advantage of these structures *within* the differentiation process. The derivatives of the components can be calculated, a common-subexpression search performed on this set of derivatives, and the required derivatives can be constructed from assigned variable names representing common subexpressions.

3.2 Taylor Series Approximation

The standard form of the Taylor series for $f(x)$ about the point $x = a$ is given by

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \frac{(x-a)^3}{3!}f'''(a) + \ldots + \frac{(x-a)^n}{n!}f^{(n)}(a) + \ldots$$

There is two sources for improvement. Firstly, we can take advantage of the common subexpression $\theta = (x - a)$. Secondly, we can exploit the common subexpressions produced by the differentiation operation. For example, in the Taylor series expansion of $\exp(y/(1-x) + a)$ about $x = 0$, each term in the result contains the expression $\exp(-y + a)$. Both sources of common subexpressions are identified without the need for a common-subexpression search.

Subsequent operations on the Taylor series approximation will have to deal with the common subexpressions repeatedly. In a symbolic context, Maple already uses this fact by only storing the common components once in its internal data representation of the expression, and using pointers to the single representation of the common component. Furthermore, Maple's `taylor` procedure uses a remember table [CGGW85, page 186] to avoid recomputing the common pieces of the Taylor expansion.

The proposal that is being made here goes further; in order to describe large output expressions effectively to the user in the symbolic algebra system, and in order to generate efficient code for a numeric environment, the output should be generated using, for example, the high-level-knowledge approach described above, in a form where the common subexpressions are described only once as a side relation [Hea84].

3.3 Integration

Consider applying this strategy to integrals calculated using the Risch algorithm [Ris69]. Liouville's theorem states that if the integral of an elementary function $f(x) \in K(x, \theta_1, \theta_2, \ldots \theta_n)$ is an elementary function then it can be expressed in the form

$$\int f(x)dx = v_0 + \sum_{i=1}^{k} c_i \log v_i$$

where θ_i, c_i and v_i belong to a tower of algebraic, logarithmic or exponential extension fields, or their algebraic closure, of the base differential field, $K(x)$.

Consider two examples; in the first integral, I_1, the extension field used in the Risch algorithm is $\phi = \ln(x-1)$, and in the second integral, I_2, the extension field is $\omega = \ln(x)$.

$$\begin{aligned} I_1 &= \int \frac{(x^2-x)\ln^5(x-1) - (x^2-x)\ln^4(x-1) + 2\ln^3(x-1) - 8\ln(x-1) + 8}{(x-1)\ln^4(x-1) - (x-1)\ln^3(x-1)} dx \\ &= \frac{4}{(x-1)\phi^2} + \ln(\phi-1) + \ln(\phi) + \frac{x^2-1}{2}\phi - \frac{x^2+2x}{4} \qquad \text{where } \phi = \ln(x-1) \\ I_2 &= \int \frac{x\ln^2(x)+1}{x\ln(x)} dx \\ &= x\ln(x) - x + \ln(\ln(x)) \\ &= x\omega - x + \ln(\omega) \qquad \text{where } \omega = \ln(x). \end{aligned}$$

The extension fields act as common subexpressions in the indefinite integral. The final result can be expressed in terms of variables, θ_i representing the extension fields, and a set of side relations describing the θ_i subexpressions. We observe that the integral can be described more concisely simply by taking advantage of our knowledge about the Risch algorithm without having to do a common-subexpression search on the result that is produced. Further investigation is needed to determine if this optimization approach can be used to identify common subexpressions that are not algebraic extensions and if this approach can be used with other procedures used for integration.

4 Concluding Remarks

We have proposed an approach to expression optimization that differs from most existing techniques, which are applied after the output has been generated. We take advantage of our knowledge of the operation and the structure of the input to this operation to produce concise, efficient code. The approach was illustrated using three operations as examples. It should not be necessary to apply this approach to a large number of operations; most of the preprocessing occurring in symbolic algebra systems for numeric codes appears to use only a few key algebraic operations such as differentiation, integration, and power series. The proposed optimization scheme need only be used in a small number of operations, in which the optimization strategies that can be exploited are easily identifiable, in order to achieve code improvement for a significant range of problems.

Future research includes determining the class of operations with which this approach is useful and comparing the cost and effectiveness of this approach with existing optimization techniques. Since all the optimization techniques improve the representation of the large expressions but do not necessarily generate the true optimum, the investigation should determine how much improvement is achieved by applying each method, for what types of problems each method is best suited, and when the use of several techniques in sequence is effective. Another issue that must be addressed is the generation of expressions that are numerically stable as well as efficient to evaluate.

References

[Bre69] Melvin A. Breuer. Generation of Optimal Code for Expressions via Factorization. *Communications of the ACM*, 12(6):333–340, June 1969.

[Bre84] Richard L. Brenner. Simplifying Large Algebraic Expressions by Computer. In V. Ellen Golden, editor, *Proceedings of the 1984 Macsyma User's Conference*, pages 50–109, General Electric, Schenectady, New York, July 1984.

[CGGW85] B. W. Char, K. O. Geddes, G. H. Gonnet, and S. M. Watt. *Maple User's Guide.* WATCOM Publications Ltd., Waterloo, Ontario, Canada, 4 edition, 1985.

[Coo82] Cook Jr., G. O. *Development of a Magnetohydrodynamic Code for Axisymmetric, High-Beta Plasmas with Complex Magnetic Fields.* PhD thesis, Lawrence Livermore National Laboratory, University of California, Livermore, 1982. Also available as report from Lawrence Livermore National Laboratory, Livermore.

[CR85] Kevin Cahill and Randolph Reeder. *Using MACSYMA to Write Long FORTRAN Codes for Simplicial-Interpolative Lattice Gauge Theory.* Research Report DOE-UNM-85/3, Department of Physics and Astronomy, University of New Mexico, Albuquerque, New Mexico 87131, July 1985.

[Fat69] Richard J. Fateman. Optimal Code for Serial and Parallel Computation. *Communications of the ACM*, 12(12):694–695, December 1969.

[Har84] Leo P. Harten. Using Macsyma to Generate (Somewhat) Optimized FORTRAN Code. In V. Ellen Golden, editor, *Proceedings of the 1984 Macsyma User's Conference*, pages 524–544, General Electric, Schenectady, New York, July 1984.

[Hea84] Anthony C. Hearn. Structure: The Key to Improved Algebraic Computation. In *The Second Symposium on Symbolic and Algebraic Computation by Computers, RIKEN, Japan, 21-22 August 1984*, pages 18–1 — 18–16, RSYMSAC, 1984.

[HvH85] B. J. A. Hulshof and J. A. van Hulzen. An Expression Compression Package for REDUCE based on Factorization and Controlled Expansion. 1985. Twente University of Technology, Department of Computer Science, Enschede, the Netherlands.

[Ked80] G. Kedem. Automatic Differentiation of Computer Programs. *ACM Transactions on Mathematical Software*, 6(2):150–165, June 1980.

[Knu81] D. E. Knuth. *The Art of Computer Programming.* Volume 2, Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1981. Seminumerical Algorithms.

[MW67] C. Mesztenyi and C. Witzgall. Stable Evaluation of Polynomials. *Journal of Research of the National Bureau of Standards – B. Mathematics and Mathematical Physics*, 71B(1):11–17, January-March 1967.

[NC79] E. Ng and B. W. Char. Gradient and Jacobian Computation for Numerical Applications. In V. Ellen Golden, editor, *Proceedings of the 1979 Macsyma User's Conference*, pages 604–621, NASA, Washington, D.C., June 1979.

[Ric83] John R. Rice. *Numerical Methods, Software, and Analysis*, chapter 3, pages 33–58. McGraw-Hill, Inc., New York, IMSL Reference edition, 1983.

[Ris69] R. H. Risch. The Problem of Integration in Finite Terms. *Transactions of the American Mathematical Society*, 139:167–189, May 1969.

[Spe80] Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, January 1980.

[SR84] Stanly Steinberg and Patrick Roache. Using Vaxima to Write Fortran Code. In V. Ellen Golden, editor, *Proceedings of the 1984 Macsyma User's Conference*, pages 1–22, General Electric, Schenectady, New York, July 1984.

[vH81] J. A. van Hulzen. Breuer's grow factor algorithm in computer algebra. In *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 100–104, 1981.

[vH83] J. A. van Hulzen. Code Optimization of Multivariate Polynomial Schemes: A Pragmatic Approach. In J. A. van Hulzen, editor, *Computer Algebra, Eurocal 83 (European Computer Algebra Conference, London, England, March 1983*, pages 286–300, Springer-Verlag, Berlin, 1983.

[vHH82] J. A. van Hulzen and B. J. A. Hulshof. An Expression Analysis Package for REDUCE. *SIGSAM Bulletin*, 16(4):32–44, 1982.

[Wan86] Paul S. Wang. FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis. *Journal of Symbolic Computation*, 2(3):305–316, September 1986.

[Wir80] Michael C. Wirth. *On the Automation of Computational Physics.* PhD thesis, University of California, Davis, October 1980. Also available as Lawrence Livermore National Laboratory, Livermore, Report UCRL-52996 (October 1980).