Program Generation through Symbolic Processing

J.A. van Hulzen

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Computer algebra systems can be useful when attempting to automate mathematics. One can use these facilities to assist in the construction of programs for numerical purposes, i.e., to assist in making the problem dependent parts of the software needed to solve a problem numerically. We discuss aspects of the symbolic-numeric interface required to accomplish this. Special attention is given to program generation aspects and to code optimization.

1. Introduction

Powerful computing resources are available today on a personal basis and for limited costs. It is therefore predictable that the use of personal computers to perform scientific computations will considerably increase. This in turn will enlarge the interest in a further and better integration of various mathematical software tools, such as computer algebra features and numeric and graphic facilities. It is therefore also expectable that computer algebra will slowly shift from a seemingly exotic and expensive hobby to an essential toolkit in problem solving, assuming adequate symbolic-numeric and symbolic-graphic interfaces are created. Computer algebra systems require dynamic development and dynamic storage of the mathematical expressions they allow to elaborate. But a language like FORTRAN, frequently employed to solve problems numerically, is used in a compile, load and execute fashion. The solution strategy is often based on the use of library subroutines in a problem defined context. A computer algebra system can be used for the construction of the mathematical expressions, which together define a specific problem and how to utilize the library facilities for its solution. Hence a symbolic-numeric interface is needed to transport information from one world to another, from a symbolic processing environment to a numeric scene. This, of course, must be worth the effort, i.e., the information to be transported must be extensive. In fact so extensive, that producing it by hand is not only error prone and impractical, but almost impossible. The symbolic-numeric interface will therefore ideally cover possibilities for program construction, code optimization and (a priori) error analysis (features). We discuss some of the aspects of such an interface, which are needed for
program generation. The type of generation we are interested in concentrates on "easy" construction of efficient and reliable programs. But such a discussion needs as a prerequisite some insight in the overall functioning of a computer algebra system. We hope to provide this knowledge in Section 2, using REDUCE to illustrate our assertions. Relevant aspects of and approaches to a symbolic-numerical interface are then presented in Section 3. Section 4 is dedicated to program generation. We mention some ideas about code optimization, intended for the production of more reliable and more efficient programs in Section 5, before some conclusions are given. Our contributions to the development of a symbolic-numerical interface are, in fact, realized as extensions of REDUCE. This is an additional reason to pay attention to REDUCE.

2. Computer Algebra

The differentiation programs of Kahrimanian [35] and Nolan [44], presented in 1953, are often considered as the first attempts to employ a digital computer to perform formal mathematical operations. We now know a rich diversity of computer algebra systems [63]. Some of these systems are frequently and routinely used to assist in solving non-trivial problems in science and engineering [2,13,48]. Well known are MACSYMA [48], MAPLE [16], muMATH [53,58], REDUCE [21] and SCRATCHPAD [34]. This list is certainly not exhaustive. We only mentioned some of the intended general purpose systems, which are either widely used, like REDUCE, or have a noteworthy design, like SCRATCHPAD. Introductory surveys of computer algebra are given in [47,70]. Recent summaries of the state of the art can be found in [11,15,19]. The mathematical capabilities of the better systems of today are of course strongly correlated to the early successes of computer algebra in such areas as integration, celestial mechanics, general relativity and quantum electro dynamics. These applications tended to shape the classes of mathematical expressions, to be formulated and manipulated in the various systems. Polynomial and rational function algebra was considered as a basic requirement. All of the well-known elementary transcendental functions, naturally entering in the description of our (approximate) models of physical reality, were and are considered as intriguing objects. A classification of computer algebra systems can be based on the class of mathematical expressions they allow to operate on. The impact of such a system is largely related to the class of transformations, which it allows to perform on its expressions, either automatically or via user control. Examples of such transformations are differentiation, integration and substitution. Portability, maintenance and ergonomic aspects, such as ease of interactive use, comprehensibility of output and performance, are additional criteria for judging such a system.

The mathematical criteria are strongly related to the quality of the algebraic simplification algorithms implemented in the system.
Simplification was once qualified as the most pervasive process in algebraic manipulation [42]. Much of the controversy around it is due to differences between the desires of a user and a designer, because the notion of simplicity is not context-free. Simplification has two aspects. An important issue is to be able to obtain an equivalent but simpler representation of a mathematical object, either internally or externally. Another aspect deals with the question how to compute a unique representation for equivalent objects. Finding equivalent but simpler objects requires an effective procedure $S : T \rightarrow T$, where $T$ is a class of mathematical objects, such that for all $t$ in $T$ holds that $S(t) \sim t$ and $S(t) \equiv t$, if $\sim$ is an equivalence relation on $T$ and if $\equiv$ connotes "simplicity". Obtaining a unique representation requires, in addition, that for all $s, t$ in $T$ holds $s \sim t \Rightarrow S(s) = S(t)$. Hence $S$ is meant to single out a unique representative for each equivalence class. $S(t)$ is therefore called the canonical form of $t$. However, it is proven that a canonical simplifier can not always be found for an equivalence relation on a given set of (mathematical) objects [12,14]. Therefore — in practice — weaker notions are employed, such as zero-equivalence and regular simplification. Zero-equivalence can be defined when a given set of expressions contains a zero-element $0$. We then call $S : T \rightarrow T$ a zero-equivalence (or normal) simplifier if for $\sim$ on $T$ holds that for all $t$ in $T$: $S(t) \sim t$ and $t \sim 0 \Rightarrow S(t) = S(0)$. Regular simplification is used in the context of expressions involving transcendental functions. It guarantees that transcendentals occurring in an expression are algebraically independent, a requirement which is for instance needed in the design of symbolic integration facilities, based on the Risch–Norman algorithm [45,46]. Simplification can be used as a ("political") instrument to produce a classification of computer algebra systems, as once done by Moses [42]:

- Radical systems can handle a single well defined class of expressions (polynomials, rational functions, for instance), by using a canonical simplifier to get all expressions into their internal canonical form. This implies that the task of the manipulating algorithms is well defined. But it can lead to inefficiencies. On input a user can present an expression as a string over a certain alphabet in any desired, syntactically correct form. This string is just one possible external representation of one internally unique object, being the representative of a whole equivalence class. To obtain such a unique object we need a set of rules, defining term ordering via degree ordering or — alternatively — ordering of the (irreducible) factors of an expression, in combination with some lexicographical ordering of the variable-symbols, occurring in the alphabet. This can imply that the output, being a reflection of the internal ordering, can surprise a user.

- New Left systems arose in response to some of the difficulties with radical a systems, such as caused by automatic expansion
(think of \((x + y)^{1000}\)) or factorization (for instance \(x^{1000} - y^{1000}\)). Expansion is brought under user control. Such systems usually can handle a wide variety of expressions with greater ease by using labels for non-rational (sub)expressions. REDUCE is such a system.

- Liberal systems give more freedom to a user and are therefore in general slower than new left systems.
- Catholic systems, finally, can use more than one internal representation and know different approaches to simplification. They tend to be large. A well-known example is MACSYMA.

Most computer algebra systems are interactive. The system reaction, an output expression, is of course just one of many possible visual representations of an internally stored expression. Other striking aspects of the use of such a system are time and space requirements. Intermediate expression swell is a well-known phenomenon. It can be caused by temporary fill-in. Factorization for instance requires expansion. Differentiation is another example of possible intermediate expression explosion. Since the purpose of computing can be qualified as an attempt to increase insight, it is obvious that we are interested in obtaining the most simple form of an expression. This is often also the shortest representation of an expression. Hearn, who designed and implemented most of REDUCE, has been studying these problems since he started making this system. He recently [29,30] gave a nice classification of simplification approaches, when considered as expression structuring activities:

- Structure preserving techniques are concerned with maintaining structure in an expression as long as possible in a given computation.
- Structure determining techniques cover attempts to induce structure on otherwise unstructured expressions.
- Structure reducing techniques are those which can be employed to reduce an expression using a set of side relations.
- Structure displaying techniques allow to present the output in a form that makes its structure more apparent to the user.

Structure preserving techniques are based on the reasonable presumption that the initial formulation of most scientific problems has a natural structure. Most simplifiers are based on this structure preservation philosophy. For instance taking an expression like

\[(x + 1)^2 - 2x^2\]

we immediately see that it has the simpler form

\[x^2 + 1\].

We want our algebra system to produce this result as well. Input
expansion easily allows to get this result, if we collect terms of equal
degree and employ ordering considerations. However, a form like

$$(x + 1)^{100} + 1$$

can better not be expanded at all. Brown [7] was the first who
observed that more flexibility was needed, against the price of drop-
ing a canonical representation. He proposed to guarantee a normal
form by representing a polynomial as a product of expanded factors in
the form:

$$\prod_{i}^{N}(\sum_{n_i=0}^{} u_{n_i} x^{n_i})^{k_i}$$

Simplification is straightforward. When multiplying polynomials,
given in such a form, one simply maintains the existing factor struc-
ture. For addition one starts collecting equal factors, before adding the
expanded remaining portions to produce a new factor. Hearn imple-
mented similar facilities in REDUCE. An implication is the need to
allow two internal forms, i.e., an expanded as well as a factored form.
The user operates by default with expanded forms, thus using a cano-
ical simplifier. He can employ a factored form on request, implying
that the non-expanded form construction is based on normal
simplification. But this does not always result in a factored form.
Internally a comparison is always made between the two alternatives.
The shortest is stored. But how? In REDUCE a recursive polynomial
definition is used [27]. The system is implemented using Standard LISP
[41], to guarantee a degree of portability. Thus the internal representa-
tion is always in the form of lists. The recursive definition implies
that a polynomial is stored as a pair consisting of a leading term and a
reductum, formed by the remaining terms of the polynomial, ordered
in some system dependent way, with (of course) the possibility of user
influence. A leading term is considered to be a pair again. This time
formed by a leading coefficient and a leading power. The coefficient can
again be a polynomial. The leading power also consists of a pair, now
formed by a main variable and its leading degree. The leaves of this
binary tree are either integer coefficients, non-zero integral powers or
variables, of which the ordering can be determined either via the object
list, or by user defined alternatives. To resolve the problem of
undesirable expansions, like for

$$(x + 1)^{100} + 1.$$ 

Hearn generalized the variable-concept [28]. In stead of the notion
variable REDUCE utilizes the kernel-concept. A kernel can either be a
variable in the traditional sense or a polynomial. So in the above given
example $(x + 1)$ acts like a variable and expansion can thus be avoided.
Once the parser knows of transcendental functions, like sine and
cosine, lists of the form (sine argument), for instance, can also be used
as a kernel. And again the argument can be a recursively defined polynomial. The REDUCE simplifier assumes all input to be the quotient of two polynomials (again a pair). When the input is really a polynomial the denominator-part of this so called Standard Quotient is simply 1. In summary:

\[
\begin{align*}
  \text{<Standard Quotient>} & ::= \text{<Numerator>} / \text{<Denominator>} \\
  \text{<Numerator>} & ::= \text{<form>} \\
  \text{<Denominator>} & ::= \text{<polynomial>} \\
  \text{<form>} & ::= \text{<integer>} | \text{<Leading Term>} + \text{<Reductum>} \\
  \text{<polynomial>} & ::= \text{<Leading Term>} | \text{<Leading Coefficient>} \\
  \text{<Leading Term>} & ::= \text{<form> * <Leading Power>} \\
  \text{<Leading Power>} & ::= \text{<Kernel>} \uparrow \text{<Leading Degree>} \\
  \text{<Leading Coefficient>} & ::= \text{<polynomial>} \\
  \text{<Kernel>} & ::= \text{<variable>} | \text{<polynomial>} | \text{<operator>} ( \text{<list of operands>} )
\end{align*}
\]

In addition it ought to be mentioned that the rich output repertoire of REDUCE can assist a user in influencing the visual version of the internal representation of the result of a computation, always being the transformation of an expression. Worth mentioning are tools to change the variable precedence or to display a partly factored form. Another facility which allows to modify output is formed by certain structure displaying commands, as mentioned by Hearn. The expression, subjected to such a command, is cut into obvious pieces which are renamed and separately shown. The renaming allows to list repeatedly occurring subexpressions only once.

The main reason to explain the overall functioning of REDUCE in some detail is to simplify our discussion of the symbolic-numeric interface. It might be illustrative to give

**Example 2.1.** Let us assume that we are interested in the determinant \( DM \) of the symmetric matrix

\[
M = \begin{bmatrix}
t_0 & t_1 & t_2 \\
t_1 & t_3 & 0 \\
t_2 & 0 & t_4
\end{bmatrix}
\]

So obviously we have

\[
DM = t_0 t_3 t_4 - t_1 t_4 - t_2 t_3.
\]

Let us now assume that the entries of \( M \) have the following values (This matrix was derived in the course of research reported in [3]):
\[ M(1,1) := - \left\{ \left( (9p \cdot M30 + J30y - J30z) \cdot \sin(Q3) \right)^2 - (18 \cdot M30 + M10) \right\}^2 - 
18 \cdot \cos(Q3) \cdot \cos(Q2) \cdot p \cdot M30 - J1Oy - J1Oy \]

\[ M(2,1) := M(1,2) := - \left\{ \left( (9p \cdot M30 + J30y - J30z) \cdot \sin(Q3) \right)^2 - 9 \cdot \cos(Q3) \right\}^2 
\cos(Q2) \cdot p \cdot M30 - 9 \cdot p \cdot M30 - J30y \]

\[ M(3,1) := M(1,3) := - 9 \cdot \sin(Q3) \cdot \sin(Q2) \cdot p \cdot M30 \]

\[ M(2,2) := - \left\{ \left( (9p \cdot M30 + J30y - J30z) \cdot \sin(Q3) \right)^2 - 9 \cdot p \cdot M30 - J30y \right\}^2 \]

\[ M(3,2) := M(2,3) := 0 \]

\[ M(3,3) := 9 \cdot p \cdot M30 + J30x \]

Neglecting the above given structure and using the facility REDUCE offers to compute the determinant of a given matrix, can lead to a number of different visualizations of one and the same object.

1. The result is presented in expanded form.
2. We turn off the expansion and get a normal form.

$$
\begin{align*}
(9P + M30 + J30Y + J30Z) \times \sin(Q3) & - (18M10 + M10) \times P - 18 \times \cos(Q3) \\
\cos(Q2) \times P \times M30 + J30Y - J10Y & \times ((9P + M30 + J30Y + J30Z) \times \sin(Q3) - 9P \\
P \times M30 - J30Y & \times (9P \times M30 + J30X) \\
((9P \times M30 + J30Y + J30Z) \times \sin(Q3) & - 9 \times \cos(Q3) \times \cos(Q2) \times P \times M30 - 9P \\
M30 + J30Y - J30Z & \times \sin(Q3) + \sin(Q2) \times \sin(Q2) \times P \times M30 \\
\end{align*}
$$

3. We use the possibility to get the structure of the determinant displayed for the unexpanded form of $DM$, and now denoted by $S7$:
WHERE

\begin{align*}
S7 &= S3 \cdot S4 \cdot S5 - S6 \cdot S5 + 81 \cdot S4 \cdot \sin(Q1) \cdot \sin(Q2) \cdot P \cdot M30^2 \\
S6 &= 81 \cdot \sin(Q3)^2 - 9 \cdot \cos(Q1) \cdot \cos(Q2) \cdot P \cdot M30^2 + 9 \cdot P \cdot M30 - J30Y \\
S5 &= 9 \cdot P \cdot M30 + J30X \\
S4 &= 81 \cdot \sin(Q3)^2 - 9 \cdot P \cdot M30 - J30Y \\
S3 &= 81 \cdot \sin(Q3)^2 - S2 \cdot P - 18 \cdot \cos(Q1) \cdot \cos(Q2) \cdot P \cdot M30 - J10Y - J30Y \\
S2 &= 18 \cdot M30 + M10 \\
S1 &= 9 \cdot P \cdot M30 + J30Y - J30Z
\end{align*}

4. Finally we display this \( DM \)-structure in FORTRAN-notation.

\begin{align*}
S1 &= 9 \cdot P \cdot P \cdot 2 \cdot M30 + J30Y - J30Z \\
S2 &= 18 \cdot M30 + M10 \\
S3 &= 81 \cdot \sin(Q3)^2 - S2 \cdot P - 18 \cdot \cos(Q1) \cdot \cos(Q2) \cdot P \cdot P \cdot 2 \cdot M30 - J30Y \\
S5 &= 9 \cdot P \cdot P \cdot 2 \cdot M30 + J30X \\
S6 &= 81 \cdot \sin(Q3)^2 - 9 \cdot \cos(Q1) \cdot \cos(Q2) \cdot P \cdot P \cdot 2 \cdot M30 - 9 \cdot P \cdot P \cdot 2 \\
S7 &= S3 \cdot S4 + S5 - 2 \cdot S5 + 81 \cdot \sin(Q3) \cdot \sin(Q2) \cdot P \cdot P \cdot 2 \\
S &= S7
\end{align*}

None of these forms is as compact as the originally given one using the \( t \)'s. The conclusion is that much room for improvement of output presentation exists and that the results, although easily obtained, can be far from optimal, especially when a numerical value for \( DM \) is required for a given set of input values for the different variables occurring in \( DM \).

In a numerical setting methods for solving systems of linear equations and determinant calculations are polynomial time-bounded operations, both in time and space. In a computer algebraic setting however, the algorithms show an exponential behaviour [31], although we have to remark that in such a setting problem size is always moderate in comparison with "numerical" problems. This limited size is related to core consumption during intermediate stages in the computations and to storage requirements for the final result. This example is quite illustrative. It clearly suggests what might happen when the matrix-size is enlarged and expansion is not turned off, for instance. \hfill \Box

The example also serves to stress that simplification, although algorithmic in nature, is not context-free. One has to try to avoid
undesirable side effects quite carefully. This is the main reason Hearn began considering the possibility of using structure determining techniques, i.e., heuristic tools to find structure in an expression, which otherwise would remain unchanged. Hearn’s presumption is that many physical problems have enough structure to allow user-controlled regrouping, based on expansion or factorization and applied at some lower levels inside an expression, and using knowledge about the “weighted”, physical meaning of the various variables used to build the given expression. What can be done is considering an expression to be a (multivariate) polynomial in (a) certain variable(s), factorize its coefficients or searching the different terms in these coefficients for common subexpressions to be factored out. These regrouping techniques can lead to remarkable compressions as is for instance shown by the following

**Example 2.2.** We show the effect of compression when applied on the expanded form of $DM$, taken from Example 2.1.1. Application of the same compression command on the unexpanded version of $DM$, as shown in Example 2.1.2, does not lead to an improvement.

\[
\]

\[
\]

\[
2 4 2
SIN(Q2)*P*M30 + 81*(9*M30 + M10)*P*M30 + J30Y*J10Y*J30X -
\]

\[
J30Z*J10Y*J30X)*SIN(Q3) - (9*(J10Y + J30X)*J30Y + J10Y*J30X)*
\]

\[
M30 + J30Y*M10*J30X)*P - 81*(J30Y - J30Z) + 9*P*M30)*
\]

\[
4 2 4 2
SIN(Q3)*SIN(Q2)*P*M30 - 9*(9*(J30Y + J10Y + J30X)*M30 + (J30Y + J30X)*M10)*P*M30 + 81*(9*P*M30 + J30X)*COS(Q3)*
\]

\[
2 4 2
COS(Q2)*P*M30 - 81*(9*M30 + M10)*P*M30 - J30Y*J10Y*J30X
\]

Also quite illustrative is the result of performing some compression experiments on the expression EXPR, given below. It shows why it is important that algebra systems can be used interactively and it stresses again that simplification is not context-free.
Program Generation through Symbolic Processing

on exp 5
expon := expr;

\[
\text{EXPO} := \frac{2}{2} \text{C*D*E} - \frac{2}{2} \text{C*D*F} \cdot \text{G} - \frac{2}{4} \text{C*D*F} \cdot \text{G*H*K} - \frac{2}{4} \text{C*D*F} \cdot \text{G*H*L} - \frac{2}{2} \text{C*D*F} \cdot \text{H*K} - \frac{2}{4} \text{C*D*F} \cdot \text{H*K*L} - \frac{2}{2} \text{C*D*F} \cdot \text{H*L} - \frac{2}{2} \text{H} + \frac{2}{2} \text{F*G} + \frac{2}{2} \text{F*F*G*H*L} + \frac{2}{2} \text{F*H*K} + \frac{2}{2} \text{F*H*K*L} + \frac{2}{2} \text{F*H*L} + \frac{2}{2} \text{F}
\]

TIME: 833 MS

off exp5

expoff := expr;

\[
\text{EXPOFF} := - (2(2(2(2(2(2(K + 2(K + L) + L) + H) + 2(K + L) + G*H + G) + F - E) + C*D - 2(K + 2(K + L) + L) + H + 2(K + L) + G*H + G) + F)) + 2(K + 2(K + L) + L) + H + 2(K + L) + G*H + G) + F) - E) + C*D - (2(K + L) + H + G) + F
\]

TIME: 901 MS

nfac expoff,c,f,(lfactr);

\[
- (2(2(2(2(2(K + L) + H + G) + (2(C*D*F - 1)*F - 2*C*D*E)) + 2(K + L) + H + G) + F - E) + C*D - (2(K + L) + H + G) + F)
\]

TIME: 1700 MS

nfac expoff,c,lfactr,(f);

\[
- (2(2(2(2(2(K + L) + H + G) + F - E) + C*D - (2(K + L) + H + G) + F)) + 2(K + L) + H + G) + F)
\]

TIME: 1904 MS

Citing Hearn [26], the present simplification algorithms in REDUCE, being used when the expansion is turned off, are a "moving target". This is mainly due to the fact that he is experimenting with the above indicated compression facilities, since the first experimental facilities were made by Hulshof, when visiting Hearn. Details about these facilities, as implemented for REDUCE, can be found in [33].

The above introduced structure-determining techniques can contribute to a reduction of the arithmetic complexity of an expression, which is needed in further numerical calculations. Another structure-determining technique — certainly in Hearn's view — is formed by the code optimization techniques, which are discussed in Section 5. They are based on reduction in arithmetic occurring in a given (set of) expression(s) by heuristically searching for common (sub)expressions. Hearn hopes that such heuristic techniques can be made instrumental for algorithmic methods to assist in a further reduction of the structure of an expression to a more simple form. The key idea is, that such common subexpression searches can lead to information about possibly
occurring side relations, which can be used for such a reduction. Another interesting thought of Hearn is to use a Gröbner base algorithm to assist in determining if these candidate side relations are consistent, by investigating their algebraic interrelations [9,10,29,30]. We only made these last remarks to underline that both heuristics and algorithmics have an important role to play in future developments in computer algebra, directed towards improving the quality of the output, certainly also when needed for further numeric work.

We gave a capsule view of some of the output features of computer algebra systems. The main intention in doing so is to provide a view on or perhaps a feeling for the rich variety of output possibilities — and thus of unwished inefficiencies — allowed by algebra systems. Although illustrated by REDUCE, similar remarks can be made for other computer algebra systems. These considerations also play a role in the next section.

3. The Symbolic-Numeric Interface

Aspects of the symbolic-numeric interface are discussed in some detail by Brown and Hearn [8] and complementary to them by Ng [43]. The apparent need for such an interface suggests, as already indicated in the introduction, that certain communication problems exist, related to information exchange between computer algebra systems and programming facilities, more specifically designed for numerical purposes. Brown and Hearn distinguished two problem sources: Numerical evaluation of symbolic results and Hybrid problems. The latter category demands for solution methods which are a mixture of numeric and symbolic techniques, implying that at some stage numerical evaluation of symbolic results might be needed as well. For numerical evaluation one can choose between, say interpretative evaluation, using a computer algebra system for both symbolic and numeric processing, and generation of arithmetic statements in an existing language for numeric processing. Both alternatives have certain drawbacks and implications.

Interpretation might be convenient for "one shot" applications (citing Ng [43]), if big float facilities, such as Sasaki’s package [54], can be used and if problem size is moderate. Kanada and Sasaki [36] found their Standard LISP-package to be half as fast as Brent’s well-known FORTRAN package [5], if they guarantee portability. Steele [56] and Pitman [49] came to similar conclusions concerning the use of MACSYMA for numerical evaluation. Pitman made FORTRAN to LISP translation facilities, thus creating the possibility of using the IMSL (International Mathematics and Statistics Library) in a LISP context. A drawback might however be that error analysis, and thus control over the precision of the big float calculations, is still left to the user.
When only differentiation is needed one can use instead of a computer algebra system special software tools, which in addition allow to utilize interval arithmetic to obtain reliable results [18,37,51,52]. These tools essentially use augmented FORTRAN or PASCAL compilers, which allow to produce subprograms, defining derivatives, created by making use of expression flow graphs, reflecting some form of intermediate 3-address code [1]. These approaches, however, do not provide simplification and thus can severely suffer from inefficiencies or limit the applicability to problems of moderate size.

The alternative — generation of arithmetic statements — is not perfect either. Many computer algebra systems offer users the possibility to obtain output in the form of assignment statements in FORTRAN notation. If the user decides to employ such a facility the obvious intend is to construct in some way or another complete programs and/or subroutines, which contain this arithmetic in some meaningful order. We discuss in this context strategies, which have been developed to assist users in producing such code in the next section.

Expression size might be an additional problem. Applications and application strategies, as for instance described by Cook [17], Smit and van Hulzen [55], Steinberg and Roache [57], Van den Heuvel et al. [60] and Wang et al. [65,66,67] clearly illustrate that computer algebra systems have to be used carefully. Often a form of lazy evaluation is employed to reduce or delay simplification activities. These applications illustrate Hearn's warning [29] that we have to learn to deal effectively with structure, which for instance might have been imposed by symmetry or by additional physical knowledge. Wang [65,66] recently showed how profitable this can be for the generation of finite element analysis software. Hearn also stated, as explained in the previous section, that the output we obtain is just one of a large number of possible representations and that structure determining techniques, such as code optimization, to be discussed in Section 5, can have a dramatic influence on reducing the arithmetic reflected by computer algebra output. This is also illustrated in most of the just mentioned papers on applications.

Once we are able to effectively generate efficient code for performing numerical computations it would be quite helpful if we are also able to guarantee the reliability of these calculations. In view of the existence of multiple precision floating point arithmetic packages it might be attractive to employ the power of a computer algebra system to determine, prior to the actual computations, how the precision has to be chosen during (parts of) the real computations, as to avoid unnecessary loss of significant digits. Our experiments with REDUCE and using Sasaki's big float package suggest that, in principle, this is possible [32]. The augmented compiler approach concentrates on creating limited symbolic facilities in a numerical context, as to allow to perform reliable computations, requiring at some stage derivatives.
without looking at the efficiency of the production of these derivatives. The creation of symbolic-numeric interfaces is still in development and has not yet resulted in completed facilities, which can be utilized to obtain reliable results.

The above outlined aspects for the construction of programs for numerical purposes are obviously related to the more traditional sequential view of programming and program execution. However, we believe that our ongoing research, based on variations and deviations of this theme, will lead to the development of similar facilities for vector- and parallel architectures, slowly on entering the market. In addition we - at least - indicated, that the already available tools do not cover the whole spectrum of instruments needed to automatize the process of solving a problem reliably.

4. Program Generation
The only tool for program generation, until recently provided by computer algebra systems such as REDUCE and MACSYMA, was the facility to switch from normal to FORTRAN-coded output. Hence to produce complete and executable FORTRAN programs directly from these systems was not possible. This left the user with the necessity to shorten output-expressions whenever required, to meet the limitations given by the 20-lines rule in FORTRAN, and to find a way (text editing or the use of write statements) to complete the FORTRAN program. The first packages to assist in this programming task are MACTRAN [69] and VAXTRAN [39]. MACTRAN, running under MACSYMA, allows to construct complete FORTRAN subroutines based on user-supplied template files. Such a file contains an outline of a FORTRAN program, the so-called passive parts of the file, and active parts, consisting of MACSYMA commands. MACTRAN processes such a file by simply copying the passive parts on the actual output file and by executing the active portions, which of course ought to result in meaningful arithmetic assignment statements in FORTRAN notation on the same output file. Hence the passive parts of such a template file define in fact the control structure of the FORTRAN program or subroutine which ought to be produced in this way. VAXTRAN, implemented in Franz LISP to run under VAXIMA, is similar to and based on MACTRAN. In addition to MACTRAN it provides a more general interface between symbolic and numerical computing techniques. Although VAXTRAN compiles generated code from VAXIMA, using an augmented compiler, and interfaces the resulting compiled code to make it callable directly from VAXIMA, it still relies on the MACSYMA FORTRAN switch only. This might effectively limit its use to moderate problems.

More recently GENTRAN [68], a code GENERation and TRANslational package became available, originally implemented in Franz LISP to
run under VAXIMA. Although specifically created to generate RATFOR-subprograms for use with an existing FORTRAN-based finite element package [66], it has the flexibility required to handle most code generation applications. A second more recent version of GENTRAN is written in RLISP to run under REDUCE [24,25]. This version transforms REDUCE prefix forms into formatted FORTRAN, RATFOR or C code. GENTRAN does not only allow generation of arithmetic expressions or assignment statements, but also of control structures, subprogram headings and type declarations. A consequence of this is that template file processing, although possible in GENTRAN, is not longer required under all circumstances. This implies that a user can generate complete (sub)programs for numerical purposes through a series of interactive (REDUCE or MACSYMA) commands. Besides a variety of flexible file handling commands, also allowing recursively performed template file processing, GENTRAN has some additional facilities which are notably interesting for the generation of numerical code: automatic expression segmentation and suppression of simplification through the generation of temporary variables. The latter facility is for instance quite attractive, as we show below in Example 4.1, to produce efficient code for the determinant $DM$, introduced in Example 2.1.

GENTRAN provides very powerful tools for the construction of efficient programs for numerical purposes, certainly when combined with code optimization facilities, to be discussed in the next section. We therefore give a short survey of the essentials of GENTRAN and conclude this section with an illustrative example. GENTRAN, viewed as a REDUCE extension, contains code generation and file handling commands, mode switches and global variables, all of which are accessible from both the algebraic and symbolic mode of REDUCE. The algebraic mode is the normal user interface with the system, while the symbolic mode — in fact a LISP-like system level — is meant for system modification and extension. Hence when the package is loaded, REDUCE can be considered to be brought in a new state. All REDUCE commands preceded by the keyword GENTRAN are now processed according to the GENTRAN rules. After conversion of the command into REDUCE prefix form it is transformed into formatted FORTRAN, RATFOR or C code, depending on the value of the global variable GENTRANLANG*. The whole transformation process is done in three stages: Between in pre- and postprocessing the translation phase is performed. During this phase either the prefix forms are translated into semantically equivalent code strings in the target language or an error message is generated. In addition subprogram headings, declarations and the like are produced. Hence, prior to translation, REDUCE evaluations have to be performed. They are actually done during the preprocessing phase. Although this strategy is similar to processing passive and active parts of template files, noteworthy differences exist.
The passive parts of a template file ought to consist of syntactically correct code strings in the target language. GENTRAN accepts translatable REDUCE commands. The active parts can be dealt with in different ways. Partly or full evaluation is under user control, either in algebraic or in symbolic mode, through some simple facilities.

For instance EVAL EXP, where EXP is any REDUCE expression or statement, causes EXP to be evaluated before translation takes place. So, assuming F stands for

\[ 2x^2 - 5x + 6 \]

and GENTRANLANG!* has the value 'FORTRAN, the command

\[
\text{GENTRAN } Q := \text{EVAL(F) / EVAL(DF(F,X))} \]

will result in

\[ Q = (2x^2 - 5x + 6)/(4x - 5) \]

GENTRAN also has three additional assignment operators, being \( :=; \) and \( :=: \). These operators are constructed out of the usual REDUCE assignment operator \( := \) by adding (an) extra colon(s). If the extra \( ':' \) is given on the left it means that the indices occurring in the matrix or array element of the left hand side have to be evaluated before translation is carried out. An extra colon to the right means that the right hand side has to be evaluated before translation into the target language is performed. So if \( M(2,2) := A \) and if \( M(3,3) := B \) then the command

\[
\text{FOR } j := 2:3 \text{ DO GENTRAN } M(j,j) := j*\text{M}(j,j)\]

will result in, again assuming FORTRAN is the target language,

\[
\begin{align*}
M(2,2) &= 2*A \\
M(3,3) &= 3*B
\end{align*}
\]

During the translation phase prefix forms of those arithmetic statements which are longer than a specified length can be replaced by equivalent sequences, which assign subexpression values to temporary variables whose values are gradually combined. This is simply achieved by assigning values to the globals MAXEXPPRINTLEN!* and FORTLINELEN!* (or RAT- or CLINELEN!* and by turning on the GENTRANSEG switch. This segmentation requires a facility to generate temporary variable names, which can be stored in a symbol table, like other names, to guarantee to obtain adequate declarations. A combination of these name generation facilities and the use of the special GENTRAN features for evaluation provides a powerful tool for effectively reducing simplification. Through the command VAR :=
TEMPVAR()$ a temporary variable name is assigned to VAR. The command MARKVAR VAR$ serves to further protect VAR for a too early reuse as temporary variable name. Thus it guarantees that the atom VAR can represent a significant value until further notice. Therefore the commands

VAR := TEMPVAR()$
MARKVAR VAR$
M(1,3) := VAR$

\text{GETRAN} \text{EVAL(VAR)} := M(1,3)$

result in the REDUCE setting

$M(1,3) = T0$

and in the FORTRAN assignment

\text{T0}=M(1,3)$

assuming the value of VAR is T0. Observe that M(1,3) is assigned a new value in the REDUCE-context, while T0=M(1,3) is only an output string. If all matrix entries are treated similarly the code to be produced for the computation of a determinant or an inverse matrix can be made much more efficient. We show the effect of these nice facilities, applied on the matrix M, introduced in Section 2, in

\textbf{Example 4.1.}

$M(1,1) = -(9*SIN(Q3)**2*P**2*M30) - (SIN(Q3)**2*J30Y) + SIN(Q3)**2*P**2*M30$

$M(1,2) = -(9*SIN(Q3)**2*P**2*M30) - (SIN(Q3)**2*J30Y) + SIN(Q3)**2*P**2*M30$

$M(1,3) = -(9*SIN(Q3)**2*P**2*M30) - (SIN(Q3)**2*J30Y) + SIN(Q3)**2*P**2*M30$

$M(2,1) = 9*P**2*M30 + J30Y$

$M(2,2) = 9*P**2*M30 + J30Y$

$M(2,3) = 0$

$M(3,1) = 9*P**2*M30 + J30X$

$T0=M(1,1)$

$T1=M(1,2)$

$T2=M(1,3)$

$T3=M(2,1)$

$T4=M(2,2)$

$D0=T0*T3*T4-(T1**2*T4)-(T2**2*T3)$

The above given piece of FORTRAN code is the result of executing the following mixture of GEOMETRAN and REDUCE commands:
GENTRANLANG* := 'FORTRAN $ 
FORTLINELEN* := 70 $ 
GENTRANOUT "M.OUT" $ 

for j:=1:3 do 
  for k:=j:3 do 
    GENTRAN M(j,k) := M(j,k) $ 

for j:=1:3 do 
  for k:=j:3 do 
    if M(j,k) neq 0 then 
      << VAR := TEMPVAR1 $ 
        MARKVAR VAR $ 
        M(j,k) := VAR $ 
        M(k,j) := VAR $ 
        GENTRAN EVAL(VAR) := M(EVAL(j), EVAL(k)) 
    >> $ 

GENTRAN DM := det(M) $ 
GENTRANSHUT "M.OUT"$ 

It is obvious that the possibility of generating temporary variables after having saved all relevant information — the real values of the matrix elements — can in principle lead to an enormous efficiency increase by computing — in fact only — the skeletal structure of the determinant. □ 

Further examples can for instance be found in [24]. Further details about GENTRAN are given in [22,23,24].

5. Code Optimization 

The level of sophistication of computer algebra systems easily allows generation of output of a size which is far beyond human understanding. The examples in the previous sections show that structure displaying techniques eventually combined with compression methods, based on heuristics, can largely improve the compactness and comprehensibility of output expressions. But when producing sets of output expressions, which are going to form the arithmetic parts of programs for numerical purposes, tools for reducing the computational complexity of such sets can be attractive. It is however only relevant to consider reducing this complexity if the arithmetic is extensive or when the solution strategy requires repetitions of identical sequences of arithmetic operations. The programs resulting from attempts to solve such, computationally intensive, problems do not meet Knuth’s conclusion that for an average FORTRAN program the extension of the compiler, with features for optimization of the arithmetic, is overdone [38]. This might explain why optimization of arithmetic code can not be qualified as a popular research area.

We recall that we do not concentrate on average programs. The elements of our sets of expressions are viewed as definitions of computational processes. Hence such a set can be seen as a block of straight
line code. The arithmetic complexity is defined as the number of elementary arithmetic operations required to obtain the results of these computations for a set of permissible inputs. This view can be refined by associating weights with the various elementary operations. These weights reflect computational costs. Attempts to optimize the description of such basic blocks can be considered as techniques for minimizing, or at least reducing, the arithmetic complexity of the given sets of expressions. However, a reduction is only possible when redundancy occurs. Redundancy is a needless form of repetition, i.e. the presence of common (sub)expressions. This is certainly true for the traditional sequential view on program execution. Other architectures demand for other notions, like "not sufficiently vectorized in a reasonable way" or "insufficiently decomposed in (sub)sets which can be processed in parallel without causing deadlock problems when combining the results to obtain the final answers". We only consider here the traditional sequential processes.

When attempting to minimize arithmetic, defined through expressions producible with a computer algebra system, we ought to know what the structure of these expressions can be before we are able to design methods allowing to discover eventually existing redundancy. As suggested in Section 2 a user can produce almost everything, efficient or not, of almost arbitrary size and depth of nesting. Therefore the design of algorithms for searching for common (sub)expressions (cse’s for short) ought to be based on the presumption that the elements of the input set of which the description ought to be optimized, can have an arbitrary structure. As a consequence such algorithms ought to be designed to allow finding cse’s of an equally arbitrary structure. The representation of cse’s inside a set of expressions is certainly normal, if we presume the commutative, associative and distributive laws to hold. With the REDUCE Standard Quotient form for expressions in mind, we can describe expressions, what ever their structure might be, in a prefix notation, as pairs of the form (operator . list of operands). Here "operator" stands for PLUS, TIMES or "something else". PLUS and TIMES, denote the usual commutative operations of addition and multiplication, respectively. Hence any desirable permutation of the elements of the "lists of operands" will in principle be allowed, when the operator is PLUS or TIMES. The lists of operands are again formed by such expressions. When for a while excluding the "something else" alternative, the expressions are multivariate polynomials over ZZ. Such a polynomial can be viewed as a sum (product) of primitive and/or composite terms (factors). We call a term primitive if it is an integer, a variable or an integer multiple of a variable. These primitives form together an (eventually empty) linear expression. Hence the composite terms are products. It depends on the ordering considerations of the algebra system, where the primitive and composite terms are located in the "list of operands". A primitive factor is a constant, a variable or a
power of a variable. Hence a product of primitive factors is simply a monomial. The composite factors are obviously sums. Every polynomial can be thought of as being built up by linear expressions and monomials only, what ever its (un)nested structure might be. When searching for cse's we — in principle — use these "primitive" information carriers, linear expressions and monomials. As soon as a new cse is found, it can be replaced by a new, system selected, variable name, assuming its description is added to the set of expressions. To obtain a correct basic block we ought to assume our set of expressions to be (partly) ordered. Every cse-description has to be inserted correctly in this sequence, i.e., before its first occurrence. When replacing a cse by a new variable it might happen that composite terms or factors collapse and become a primitive. Hence when basing the search for cse's on primitives the overall process becomes obviously iterative.

In contrast to the usual dag models for the representation of arithmetic expressions [1], we employ, following Breuer [6,61,62], sparse extendible matrices, albeit in a slightly more sophisticated way. We therefore have to (re)parse the internally stored list representations of the elements of our sets of expressions, multivariate polynomials, in more transparent and multi-accessible matrix structures. The columns of the matrices are associated with the variables and the rows with the (sub)expressions. Although merged in practice, we store the linear expressions and the monomials in separate matrices, which are interconnected via hierarchic information about the structure of the expressions, subjected to our cse-search. When a variable occurs in a linear (sub)expression its coefficient is stored as matrix entry. Similarly the exponent is stored when the variable occurs in a monomial. Therefore the validity of the commutative law ought to be presumed. To be able to retrieve the original structure of the expressions involved in the search, additional information about the hierarchy ought to be associated with the rows of the matrices. What kind of additional information is of importance? Of interest is a list of so-called children, i.e., a list of indices of rows where the descriptions of the composite terms or factors of the present row are stored. Also important is a name field, used to store the name associated with an expression or if we are dealing with a subexpression the index of the so-called father of this subexpression. We further mention an operator field and an ordered list of cse-indicators, allowing to obtain correct evaluation sequences, when translating the result of a cse-search into, for instance, FORTRAN code. The operator field is not only important for distinguishing between PLUS and TIMES, because we use merged structures internally, but also for effectively using the "something else" alternative. These "strange" operators are "removed" so as to get back to the multivariate polynomial scheme. We again distinguish between primitives and composites. When all elements in the "list of operands" are primitives, i.e., constants and/or variables, the pair (operator . list of
operands), a kernel in the REDUCE-setting, is replaced by a new variable, such that all identical primitive kernels share the same name. Kernels with (partly) composite operands are treated like sums. The operator field has a different value and the searches for identical operands are slightly different since commutativity is not longer valid. To avoid complicating our discussion we further neglect the "something else" alternative. Let us now try to visualize the data structures we employ temporarily to obtain an optimized version of a set of expressions via

**Example 5.1.** Let us assume to have as set of input expressions:

\[ E_1 := (2A + 4B + 3C) * A * C * D \]
\[ E_2 := (4A + 6C + 5D) * A * B * C \]

This set is — when oversimplifying reality — parsed and stored in the following way:

**Sumscheme:**

\[
\begin{array}{cccc}
1 & A & B & C \\
2 & 4 & 3 & D \\
3 & 4 & 6 & 5 \\
& 1 & 0 & 2 \\
\end{array}
\]

**Productscheme:**

\[
\begin{array}{cccc}
1 & A & C & D \\
0 & 4 & 6 & 5 \\
2 & 3 & 4 & 1 \\
& E_1 & E_2 & \\
\end{array}
\]

More detailed examples are given in [62]. A cse-search will result in:

\[ S_0 := 2A + 3C \]
\[ S_1 := A * C \]
\[ E_1 := S_1 * D * (S_0 + 4B) \]
\[ E_2 := S_1 * B * (2 * S_0 + 5 * D) \]

This set is reconstructed from the matrices, resulting from the cse-search:
So initially cse's are either linear expressions or monomials. To discover them the integer matrices are heuristically searched for sub-matrices of rank 1 of maximal size. A basic scan is used, which can be qualified as "test whether the determinant of a (2,2)-matrix of non-zero entries is zero". Its use is based on information about row weights, which allow to locate completely dense submatrices. The row-weight is a reflection of the arithmetic complexity of the primitive defined by that row. Further details are given in [61,62]. Since we want to reduce the arithmetic complexity, say the pair $AC = (np, nm)$, a cse-detection ought to contribute to a reduction of the number of additions (np) and/or the number of multiplications (nm). This is only possible if a cse occurs at least twice and contains at least one addition and/or multiplication. Other less detailed criteria are conceivable. Another category of cse's is formed by repeatedly occurring constant multiples of variables and by single powers, delivering addition chain problems. This category can be enlarged during the optimization process. This can be illustrated by

**Example 5.2.** The result shown in Example 5.1 is in fact intermediate. The real result, given by the present version of the Optimizer, is:
Number of operations in the input is:

Number of (+,-)-operations : 4
Number of (*)-operations : 12
Number of integer exponentiations : 6
Number of other operations : 0

\[ S_0 := 2*A + 3*C \]
\[ S_3 := A^A \]
\[ S_7 := C*C \]
\[ S_4 := C*S_7 \]
\[ S_1 := S_3*S_4 \]
\[ S_2 := S_1*S_1 \]
\[ S_8 := D*D \]
\[ S_7 := S_8*S_8 \]
\[ S_5 := D*S_7 \]
\[ E_1 := S_2*S_5*(S_0 + 4*B) \]
\[ S_7 := B*B \]
\[ S_6 := S_7*S_7 \]
\[ E_2 := S_1*S_6*(2*S_0 + 5*D) \]

Number of operations after optimization is:

Number of (+,-)-operations : 3
Number of (*)-operations : 19
Number of integer exponentiations : 0
Number of other operations : 0

First a repeated search for cse's with at least two operands is performed. Then the optimization is completed with a finishing touch.

The first step consists of four subsearches:

1— Application of the commutative law when looking for linear (sub)sums and (sub)monomials, respectively. The strategy is based on an extension of Breuer's grow factor algorithm. Cse's are replaced by new names and their description is added to the matrix, implying that composite operands can be reduced to (new) primitives.

2— A kernel search, to discover if composite kernels can now be viewed as primitives, followed by update operations.

3— Merging activities based on the assumption that composites are possibly reducible to primitives, i.e., a composite factor, defined in the sum matrix and reduced to a primitive, can be migrated, in its new form, to the product scheme and visa versa.

4— Application of the distributive law, i.e., replacement of an expression like \( a*b + a*c \) by \( a*(b + c) \) by adequate information migration.

Although the basic scans are always performed on primitive structures the cse's can have an arbitrary complex structure, because information is continuously migrated through the matrices.

The finishing touch consists of factoring out contents of integer coefficients in sums, detection of repeatedly occurring integer multiples of variables and addition chain operations so as to replace all exponentiations by multiplication sequences. This phase is characterized by one
row (or one column) operations, in contrast to the first, where mainly completely dense submatrices are examined.

Example 5.3. The determinant $DM$ of the matrix $M$, introduced in Section 2, can be computed quite efficiently, when using the possibility of introducing temporary variables, the $t_i$'s, via simple GENTRAN-commands. This is even more striking when the inverse matrix is required; see [24]. However we can further reduce the arithmetic complexity by optimizing the set of expressions formed by the different entries of $M$, leading to:

Number of operations in the input($T_0, T_1, T_2, T_3$ and $T_4$) is:

Number of $(\ast, \ast)$-operations : 17
Number of $(\ast)$-operations : 29
Number of integer exponentiations : 13
Number of other operations : 9

$S_0 := \text{SIN}(Q3)$  
$S_8 := S_0 \ast S_8$  
$S_1 := \text{COS}(Q3)$  
$S_2 := \text{COS}(Q2)$  
$S_7 := P^P$  
$S_5 := S_7 \ast M_{30}$  
$S_4 := S_5 \ast S_1 \ast S_2$  
$S_6 := - J_{30Y} \ast J_{30Z}$  
$S_{13} := 9 \ast S_5$  
$S_{10} := - S_{13} + S_6$  
$S_{11} := S_{10} \ast S_8$  
$T_0 := S_{11} + 18 \ast S_4 \ast J_{30V} \ast J_{30Y} \ast S_7 \ast (18 \ast M_{30} + M_{10})$  
$S_9 := S_{13} + J_{30Y}$  
$T_3 := S_{11} + S_9$  
$T_1 := T_3 + 9 \ast S_4$  
$S_3 := \text{SIN}(Q2)$  
$T_2 := - S_{13} \ast S_0 \ast S_3$  
$T_4 := S_{13} + J_{30X}$

Number of operations after optimization is:

Number of $(\ast, \ast)$-operations : 11
Number of $(\ast)$-operations : 13
Number of integer exponentiations : 0
Number of other operations : 4

Optimization of the various forms of $DM$, introduced earlier, leads to different results, as shown in Table 5.1. Observe that the arithmetic complexity of the optimized version of the expanded form of $DM$ is compatible with the arithmetic complexity of the not optimized version of the unexpanded form of $DM$. It is obvious that a slight increase of the size of $M$, without increasing the complexity of its entries, will result in more drastic differences.

As stated before expression size can impose problems. Our Optimizer allows to handle extensive input piece wise, but such that the results of previous optimization activities are taken into account on user request. This offers a possibility to handle expressions via partitioning. The user interface is simple. Only a few commands are needed, in combination with a number of mode switches and flag settings to influence output-notation or to obtain additional information
Program Generation through Symbolic Processing

<table>
<thead>
<tr>
<th>Form of DM</th>
<th>Status</th>
<th>Number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+,−</td>
<td>*</td>
</tr>
<tr>
<td>expanded</td>
<td>input</td>
<td>36 82 30</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>27 51 −</td>
</tr>
<tr>
<td>compressed, after expansion</td>
<td>input</td>
<td>35 63 23</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>27 42 −</td>
</tr>
<tr>
<td>unexpanded</td>
<td>input</td>
<td>24 40 21</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>13 19 −</td>
</tr>
</tbody>
</table>

Table 5.1.

about the optimization process. Details will be given in [64].

6. Some Conclusions

Ideally, as said before, the symbolic-numeric interface ought to provide user friendly facilities to allow to produce efficient and reliable numerical programs in a natural way. This requires a further integration of program generation and code optimization facilities and an extension of optimization techniques as to be able to optimize structured programs in stead of only local blocks of straight line code. A priori error analysis for such programs is an additional need and more far reaching than the present possibilities. It would also be of tremendous help if the Optimizer could be extended with a module which allows to discover automatically patterns in the problem formulation, which, for instance, are due to symmetries. This is certainly useful if extensive differentiation or integration of the code, to be optimized, is an additional need. We expect to witness such extensions during the coming years. In fact, our work in progress covers some of these items. We, not only work on variations related to program construction for vector and parallel architectures, but also on improvements of the symbolic-numeric interface and certain aspects of simplification. Worth mentioning are:

- Bottom-up structure recognition facilities [20], to be used to develop methods, which allow to discover symmetries, for instance. Such algorithms can also be helpful in improving differentiation procedures.

- A combined use of unification and simplification. We created already an environment to perform experiments in a REDUCE context [50].

- Improvements of the symbolic-numeric interface by investigating classes of problems, which can largely profit from such facilities for their solution. The design and implementation of programs for user friendly generation of Jacobians and Hessians, which ought to allow to simply connect their output with NAG library routines, learned that a further integration of a package like
GENTRAN with our Optimizer is not too complicated and certainly most profitable [59].

Thus far we limited the Optimizer activities to expressions with integer coefficients and exponents. But expressions over other domains are conceivable [4], implying that an extension of the Optimizer ought to be considered.

Acknowledgements. Over the past years I had many valuable and pleasant discussions about the above indicated aspects of computer algebra. I like to mention especially Johan de Boer, Barbara Gates, Victor Goldman, Tony Hearn, Ben Hulshof, Arthur Postmus, Jaap Smit, Pim van den Heuvel and Paul Wang. The continued interest of Leo Verbeek has been most stimulating.

References
5. R. Brent: A FORTRAN multiple-precision arithmetic package (1976), Computer Centre, Australian National University, Canberra, Australia.
Advanced Methods in Theoretical Physics (1977) 1-15, St. Maximin, France.


60. P. van den Heuvel, J.A. van Hulzen & V.V. Goldman: Automatic
generation of FORTRAN-coded Jacobians and Hessians, in: 


62. J.A. van Hulzen: Code optimization of multivariate polynomial
schemes: a pragmatic approach, in: J.A. van Hulzen (ed.): Proceed-


64. J.A. van Hulzen & B.J.A. Hulshof: A code optimization package
for REDUCE. In preparation.

65. P.S. Wang: Taking advantage of symmetry in the automatic gen-
eration of numerical programs for finite element analysis, in: 
Comp. Sci. 204 (1985) 572-582, Springer-Verlag, Berlin - Heidel-
berg - New York.

of numerical programs in finite element analysis, J. Symbolic Com-
put. 2 (1986) 305-316.

67. P.S. Wang, T.Y.P. Chang & J.A. van Hulzen: Code generation and
optimization for finite element Analysis, in: J. Fitch (Ed.): 

68. P.S. Wang & B.L. Gates: A LISP-based RATFOR code generator, 
in: V.E. Golden (Ed.): Proceedings 1984 MACSYMA User's

69. M.C. Wirth: On the Automation of Computational Physics, Ph.D. 

70. D.Y.Y. Yun & D.R. Stoutemyer: Symbolic mathematical computa-
tion, in: J. Belzer, et al. (Eds.): Encyclopedia of Computer Science 
York - Basel.