Effective reduction of arithmetical expressions

João B. Oliveira*
Instituto de Informática
Caixa Postal 1429 — PUCRS
90690-900 – Porto Alegre – Brasil

Ivan Santa Maria Filho†
Instituto de Computação
Caixa Postal 6176 — UNICAMP
13083-970 – Campinas – Brasil

26 de março de 2001

Resumo

This paper describes a sequence of steps to reduce the code used to evaluate arithmetical expressions, and these steps may be used either by a preprocessor or even when the code itself is being executed by the first time, hiding the process through class methods that intercept the evaluation of objects. The main steps involved are symbolical simplification, collected common subexpressions, static memory allocation (when used as preprocessor) and partial vectorization. This set of simple techniques obtains impressive results when the expressions are to be evaluated several times, and specially when methods like automatic gradient or slope evaluations are used, as every operation that is avoided makes a vector operation unnecessary. These methods may be applied to computations with real, interval or complex variables as well.

Introduction

Despite the computer evolution in the last decades, the need to reduce the number of instructions needed to evaluate arithmetical expressions is felt in many environments and many techniques have been used to generate better code for expressions that will be evaluated many times and thus consume considerable time in scientific systems. Although

*oliveira@inf.pucrs.br
†ifilho@dcc.unicamp.br

optimal methods are not always practical, a small collection of simple techniques that dramatically reduce the effort without guaranteeing optimality may be used for considerable gain in speed. In this paper we present a collection of such techniques, to be either used at runtime using an object called `Formula`, having special operations associated to it, or by a preprocessor that rewrites the source code to reflect the simplification. Thus, an object–oriented implementation based on extending the floating point (or complex, or interval) types can easily manage all calculations almost without user interference and keeping the implementation easy to use.

To reduce the code for functions defined by algorithms, we introduce the notion of factorable functions and their factorizations. This provides a formal background enabling us to evaluate such functions, and from these factorizations we are able to operate. Thereafter, we explain the use of a `Formula` data type, and the simplification operations embedded on it. Starting with symbolical simplification, we collect common subexpressions and optimize memory allocation, preparing for a possible vectorization. At the end, we explain the method that may be used to force `Formulas` to behave like usual mathematical objects, that is, sending values of parameters to them and getting back results.

1 Functions and factorizations

Usually, a function or subroutine that computes a mathematical function might be regarded as a sequence of steps written in an algebraic representation. Following [6], we assume:

- For every set of variables $x_1, \ldots, x_n \in R$ a finite sequence of execution steps will be executed.
- Most of the operations are piecewise differentiable.
- At each step only previously defined values are used.
- If we regard `IF`s, jumps and loop instructions (`for, continue, break`) as convenient tools for sequencing operations, almost all other operations are arithmetical expressions, attributions and library function calls. An analysis of the problems introduced by such control structures is a current research topic, but considerations of this kind will not be made in this paper.

As described above, a function will be a given block of code that possibly has some kind of internal control (`IF`s and loops) and computes a well defined value for every set of input values. We will not consider problems that may be introduced by such control structures in special environments, as is the case of `IF`s used in automatic differentiation programs.

A set of dyadic operators will be possibly necessary when defining a function. The set usually found in many computer languages is $\{+, -, *, /\}$. Another set composed by continuous monadic operators may also be necessary. A typical set composed by this kind of operators would be $\{\cos, \sin, \exp, \ln, \log, \ldots\}$.

Definition: Factorization, factorable function

Let $L \subset C^1$ be a set of monadic operators used in the definition of a function $f : D \subseteq R^n \to R$. The function f is said to be computed by some sequence $\langle f \rangle = (f_1, \ldots, f_k)$ of functions $f_i : R^n \to R$, with $1 \leq i \leq k$ satisfying

$$\begin{array}{llll} f_i(x) \equiv \alpha \in R & & & \text{or} \\ f_i(x) \equiv x_j & 1 \leq j \leq n & & \text{or} \\ f_i(x) = g(f_j(x)) & j < i & g \in L & \text{or} \\ f_i(x) = f_j(x) \circ f_k(x) & j, k < i & \circ \in \{+, -, *, \div\} & \end{array}$$

and if for all $x \in D$ the values $f_i(x)$ are well–defined and $f(x) = f_k(x)$.

Additionally, the sequence $\langle f \rangle$ is said to be a *factorization* of f, and has k steps $f_1, \ldots, f_k$. The function f is also said to be a *factorable* function.

For vector valued functions $f(x) = (f^1(x), \ldots, f^s(x))$, the definition of a factorable function can be extended in the following way: $f : D \subseteq R^n \to R^s$ is factorable if all component functions $f^k(x), 1 \leq k \leq s$ are factorable.

2 Formulas and their simplification

An object-oriented language like C++ can be easily used to support factorizations as above and preprocess them before they are evaluated several times inside a loop, recursive calls or other methods, by applying simplification rules and holding the resulting steps in the internal memory of the object. In fact, it is possible to build a C++ class representing a `Formula` in such a way that it completely masks all kinds of arithmetical expressions using operator overriding and appropriate constructor methods, and may comfortably be used as below[1]:

```
Formula Test(x, y, z)
Formula x, y, z;
{
Formula res = x^2 + x*y + x*z - 3*x^2*y*z / (x-1+1);
return res;
}
```

This would represent a clear mathematical expression, receiving three parameters and computing something from them, but we are not as much interested in computing values as in analyzing and reducing the number of operations. At this point, the use of objects brings its first advantages: the `Formula` class overrides the operations used with real numbers and every operation in this class produces a node in a tree representation of the expression being evaluated (this tree is shown in fig. 1). To build such tree, the `Formula` class has a stack of pointers at its disposal and every operator in the class gets operands from the stack and

[1] This expression has some redundancies embedded to demonstrate simplification facilities later.

pushes its result there. As operators in the class are called, the tree corresponding to the expression to evaluate is built.

At the end, the assignment operator starts the whole process of simplification and reduction, finishing with the storage of the enhanced code in the result, a shorter factorization from where the user may take numerical values through other operators. To continue with the example, the expression corresponding to `res` would be represented by a traditional compiler without simplification as the sequence

$$
\begin{array}{lcl@{\qquad}lcl}
f_1 & = & x & f_{10} & = & f_9 * f_2 \\
f_2 & = & f_1 * f_1 & f_{11} & = & f_3 * f_{10} \\
f_3 & = & y & f_{12} & = & f_6 * f_{11} \\
f_4 & = & f_1 * f_3 & f_{13} & = & 1 \\
f_5 & = & f_2 + f_4 & f_{14} & = & f_1 - f_{13} \\
f_6 & = & z & f_{15} & = & f_{14} + f_{13} \\
f_7 & = & f_1 * f_6 & f_{16} & = & f_{12}/f_{15} \\
f_8 & = & f_5 + f_7 & f_{17} & = & f_8 - f_{16} \\
f_9 & = & 3 & & &
\end{array}
$$

In this case, f_{17} will contain the last value computed, that is, the evaluation of `Test(x, y, z)` for some `x`, `y`, `z`.

2.1 The tree

To perform symbolical simplification on the expression, its necessary manipulate and change the expression to be simplified, and we begin with the classical representation of arithmetical expressions using trees, although a little modified to make some operations simpler. Even if analyzing input expressions and constructing trees is usually a minor nuisance, in our case the object-oriented language itself will assure that the sequence of operations describing the computation is analyzed properly with respect to priorities, so we simply have to declare each monadic and dyadic operator from the `Formula` class to get two operands from the stack, create a node with the necessary information and push its address back in the stack. Thus, at the end of the evaluation of the right side of the assignment, we have the tree in fig. 1.

This kind of tree is equally produced when a compiler generates code to be evaluated, but compilers do not go very far in optimizing the expression as they work rather near the instruction level and resource allocation. Thus, we insert simplification steps at an intermediate level, starting with symbolical simplification and going further down, so that a shorter sequence of steps can be either evaluated by an evaluating procedure if runtime reduction is being done, or delivered to the compiler if simplification is being done before the program actually runs.

For later convenience, as the tree is produced the `Formula` class also knows when a new variable or number is inserted, so it avoids multiple nodes for variables or numbers and groups references to them, and we really obtain the graph in fig. 2.

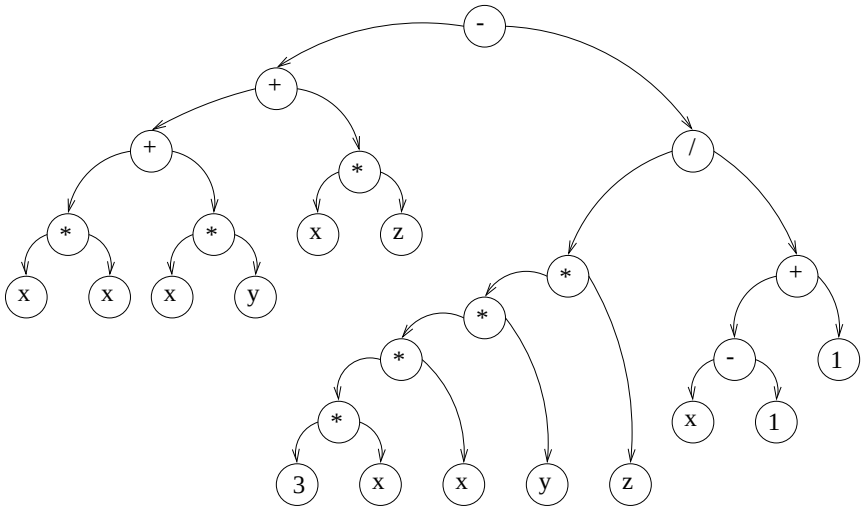

Figura 1: the tree initially used to represent the expression.

After powers are converted to successive multiplications (making non-integer exponents not directly treatable in this stage), we are ready to perform the assignment, associating the tree to the `res` variable. The assignment operation, however, plays a key role as it signs that the right side (the tree) is ready for simplification, and thereafter can be safely assigned to the left side. Thus, the assignment assumes the whole work, that can be resumed in the repeated application of the following steps:

- Structure maintenance
- Application of simplification rules

Finally, when no further maintenance or simplification can be performed, common subexpressions are collected and variables are assigned so to reduce the amount of memory used and allow memory to be allocated at once and the final factorization is created. These steps will be analyzed in more detail in the following sections.

2.2 Simplification rules

The set of reduction rules presented is fairly primitive and does not go beyond rules taught in basic school, but its repeated application leads sometimes to impressive results. Here is a description of the implemented rules, from the simpler ones to the more complex.

2.2.1 Grouping operations

The binary representation fromm figures 1 and 2 is too restrictive when aiming at work arithmetical simplification, but the adoption of general trees assures the simplicity of the rules that will be used later. So, the first step is the analysis of the tree to join nodes that share the same kind of operation, that is, successive multiplications are packed together,

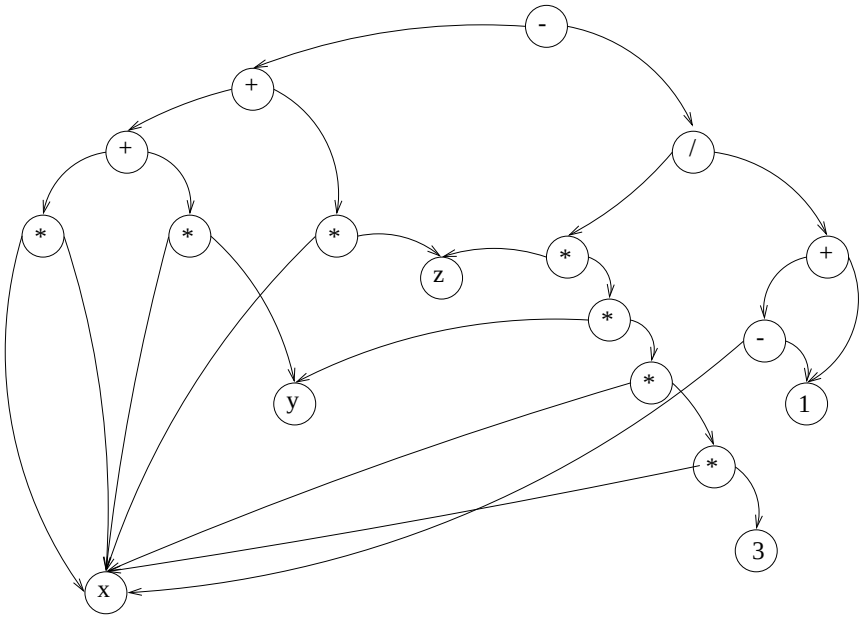

Figura 2: variables and numbers are uniquely represented.

sums are grouped and so on. The result is depicted in fig. 3. There, the major improvement is the transformation of four multiplication nodes into a single node. In addition, two sum nodes are also joined.

2.2.2 Joining compatible operations

Further, the separation between sum and subtraction, as well as between multiplication and division, introduces extra complexity in the simplification rules as they have to handle four different types of nodes. Trying to make the algorithms as simple as possible, one comes to the idea of joining these operations through a special kind of node having pointers that may either point to the left or to the right with opposite meanings, as shown in fig. 4.

Doing this, both subtractions and divisions disappear, substituted by sums and multiplications if we just assume that the pointers leaving a node by the left preserve the original operation, and nodes leaving through the right have its opposite meaning. Performing this whenever possible, we arrive at the graph shown in fig. 5.

Following this convention, both unary minus and $1/x$ fractions have obvious representations: unary minus is represented as a sum with a single operand at the right side, and a $1/x$ ratio is represented as a multiplication with its x operand at the right side and an explicit 1 at the left, although this may be left implicit. The explicit representation, however, has some advantages when simplifying, reducing the number of cases that must be tested later. Figures 6 and 7 show possible representations.

Up to now, the representation of the expression has changed to a simpler form, but its number of operations has not been reduced, and the time has come to develop the simplification procedures. These procedures operate on a bottom-up basis, continuously

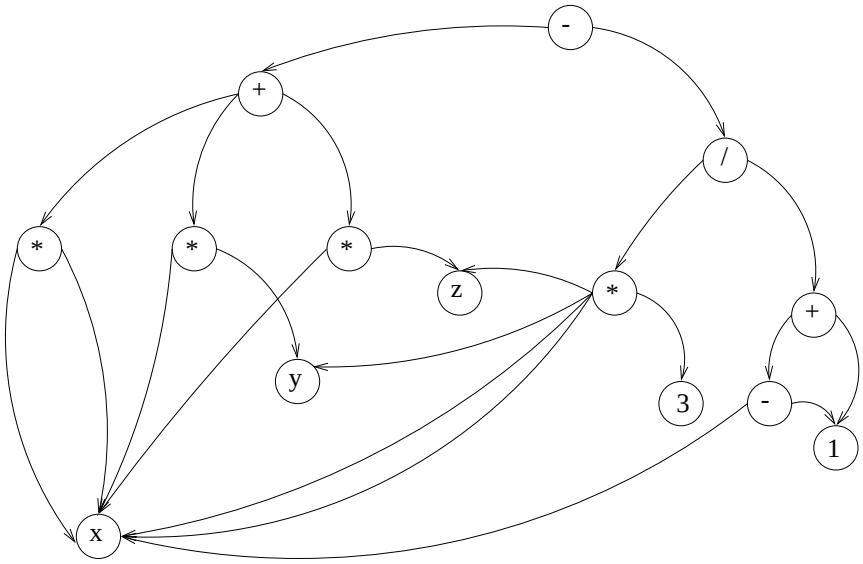

Figura 3: consecutive operations of the same type are grouped.

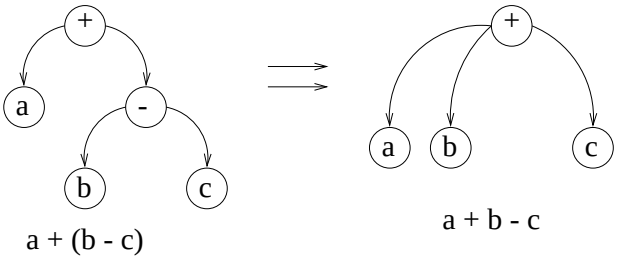

Figura 4: the form of the new nodes, with left and right "sides".

examining all nodes up to the root and trying to simplify them. If some node is simplified, all its sons are examined again, as a change may have introduced other possibilities for code reduction. So, a node may be visited several times until none of the operations can be applied on it.

The application of the rules may seem somewhat unusual at the first time: when a rule is successfully applied the algorithm does not try to apply it again, or to advance to another rule. In fact, as soon as any rule is applied it is safer to start from the first rule again, as these first rules perform simple maintenance operations, and largely reduce the work and complexity of the following rules.

In this sense, we build an exhaustive application of simplification rules, and this application is guaranteed to stop as every rule reduces the complexity of the expression represented on the tree, making a contraction-expansion loop impossible. The rules used after the previous operations, in order of complexity, are the following:

Join compatible operations: this maintenance must be repeated every time a node is visited for simplification, as other operations in previous steps may have left behind

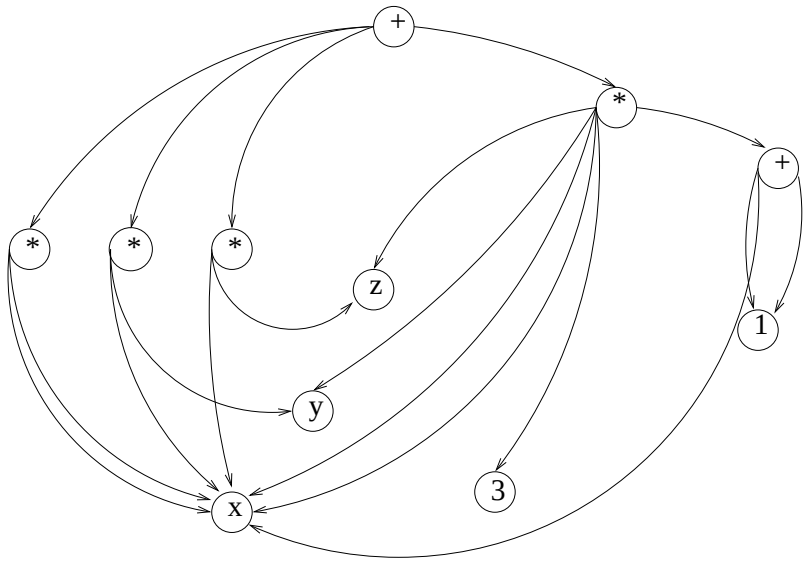

Figura 5: the new nodes are used in the tree.

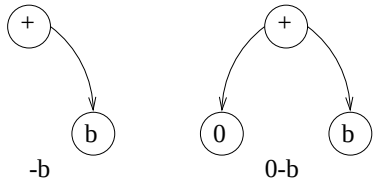

Figura 6: options for unary minus.

a "mergeable" situation.

Check for useless nodes: some operation in a previous simplification step may have left a node that is either useless or can be deleted without loss of information. As an example, see fig. 8. In the example, the computation of $6 - 2 = 4$ in a previous step would leave a sum node with a single left operand, and this sum can be safely deleted provided that its operands are inherited by its father. It is interesting to notice that if the sum were pointing to the value 4 through its right side, that is, representing an unary minus, the sum node would have to be kept.

Signs are normalized: to assure an uniform representation and more simplification pos-

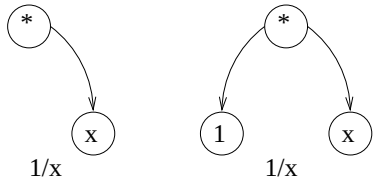

Figura 7: options for the inverse of a number.

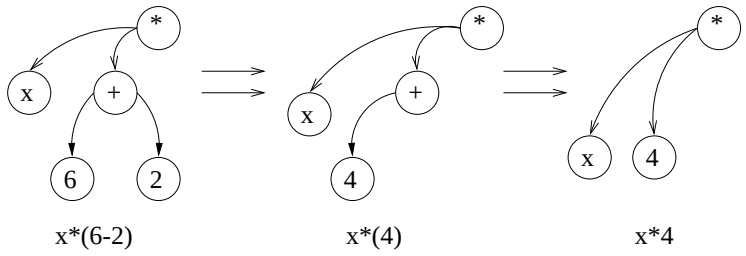

Figura 8: deleting possibly useless nodes.

sibilities, the signs of operands on multiplications are normalized, as shown in fig. 9. This is really a straightforward operation, as all operands of a multiplication node

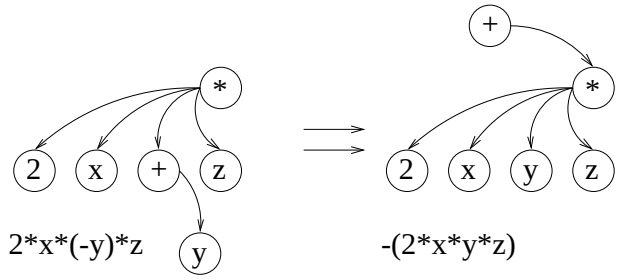

Figura 9: signs are normalized and "float up".

are analyzed and if an odd number of negative operands is found the multiplication node has its sign reversed. In any case, all its operands lose their explicit minus signs. In the previous rule (see fig. 8) it was said that if the node representing '4' had a unary minus, then the sum node representing it would have to be kept. Now, we know that this unary minus would float up to be canceled later with some other minus sign that may exist.

The previous operations are maintenance ones, but from now on effective simplification procedures can be applied with ease as the representation is in a standard form:

Check for neutral elements: some operations may insert nodes representing the numbers 0 and 1 as neutral elements after a simplification, and it may be the case that the references to these nodes may be safely deleted, or (even better) that these nodes indicate a whole expression to be safely eliminated. Some examples are given in fig. 10 below.

Dealing with numbers: the first step is to check nodes and find numbers that may be evaluated. So, multiplications and sums are visited and if there are two or more numbers as operands of a node, these numbers are operated together and reduced to a single value, as the example in fig. 11 shows. The example also gives an extra reason to restart the simplification sequence, as a simplification of two numbers (and

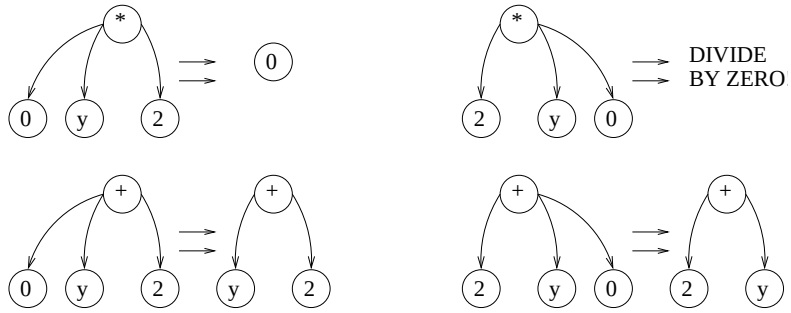

Figura 10: operations with 0 and 1.

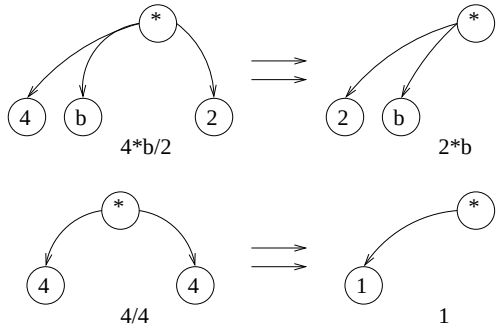

Figura 11: operating on numbers.

many other operations, in fact) may produce an operation with a single operand, and this operation will be deleted if previous rules are applied again.

It is interesting to notice that one may easily control the kinds of operations applied to the numbers at this step. For example, one may perform only operations between rational numbers and avoid rounding errors or convert the final results to floating point numbers to get some extra speed on the final code. Moreover, if the types of the operands are known in advance one may control the operations that can (or cannot) be applied to them, according to their type.

At this step one may also perform the evaluation of functions that are applied on numbers, such as sines, cosines and so on. These can be replaced by floating point numbers (for speed) or intervals (for safety), according to the predefined settings on the simplification process.

Search for cancellations: This procedure searches every node to guarantee that no operand appears on its both sides, as in this case there would be a pair of canceling arithmetic operations. If such a case is found, the repeated operands are simply deleted from both sides, as in fig. 12. This operation just checks for cancellations of variables and numbers, and it does not check for cancellation of whole subexpressions, although this would be possible. This option was left out at this point as it was considered rather costly, but it could easily be implemented at the end of the whole process,

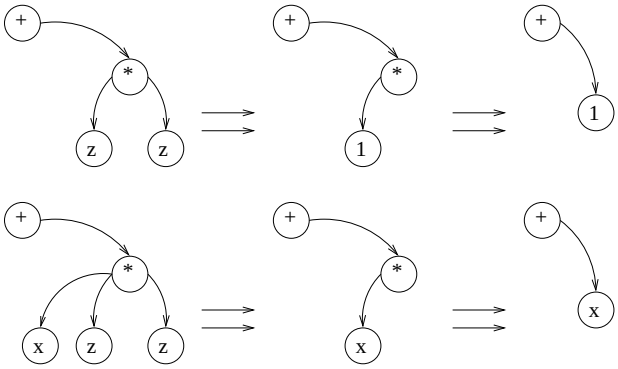

Figura 12: canceling out operations.

after common subexpressions are collected.

Once again, knowledge on the types of the operands makes a big difference: for example, knowing that a certain variable is an interval, one may forbid strict cancellations but allow a relaxed form that substitutes the repeated occurrences of the interval with its interval counterpart. That is, we perform the substitutions

$$\begin{aligned} X - X &\Rightarrow [-1, 1] * \mathrm{diam}(X) \\ X/X &\Rightarrow Y \ni 1 \end{aligned}$$

Produce better sums: trying to reduce the computations, all sum nodes are searched to check if there is a possibility of grouping some of its operands (variables and/or numbers) through multiplications. Once more, this operation may insert new nodes

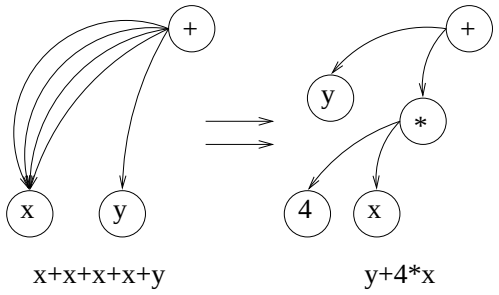

Figura 13: turning sums into multiplications.

or change already existing nodes that will be affected later by other simplification procedures. A small oexample of this operation is given in fig. 13.

Select common parts: this operation tries to find common parts in the operands of a sum, so that a common subexpression can be taken out to form a parenthesized expression: This operation is more complex than the others as there may be more than one obvious expression to be selected and there may be more than two multiplication nodes involved, so several situations are possible:

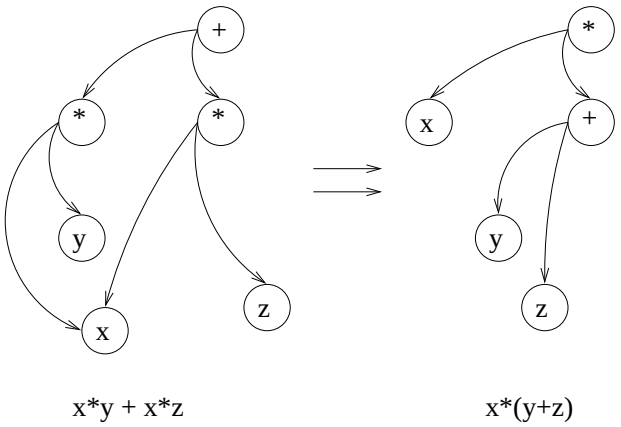

Figura 14: taking out common subexpressions.

- depending on the selected subexpression, a multiplication node may not be affected at all as it does not have the select subexpression;
- a node may have the chosen subexpression deleted from its operands, as in the example given in fig. 14;
- a node may disappear if it is identical to the selected subexpression. In this case, it leaves in its place a single node with value one, the neutral element for multiplication. In any case, a valid expression is kept.

It is interesting to notice that if types are known in advance, one may tune the selection process to follow restrictions according to these types: for example, if some variable is an interval, we would allow its powers to be collected but never split. So, for example, $X^3 + YX^3$ would be collected into $X^3 * (1 + Y)$, but $X^3 + YX^2$ would remain unchanged, as the obvious selection $X^2 * (X + Y)$ would break a power that could be exactly computed otherwise.

Futhermore, there are usually two ways of choosing the best common factor to take out:

- To choose the factor that occurs most frequently in the multiplications being examined;
- To choose the longest factor that occurs in a pair of multiplications.

This basic set of simplification procedures is applied whenever possible for all nodes, and the reduction process will stop only when there are no nodes for which any of the procedures can be applied. In our example, when the process is finished we end up with the graph presented in fig. 15. As can easily be seen, this corresponds to the expression

$$x * (x + y + z * (1 - 3 * y))$$

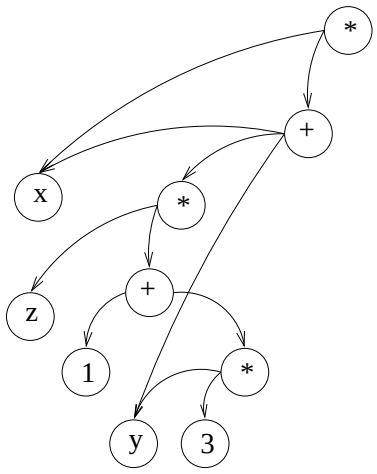

Figura 15: after the full simplification.

and can be represented by the factorization below. This factorization, in a suitable representation, may then be stored in the `Formula` object to be consulted as numerical evaluations are needed.

$$
\begin{aligned}
f_1 &= x \\
f_2 &= y \\
f_3 &= z \\
f_4 &= 1 \\
f_5 &= 3 \\
f_6 &= f_5 * f_2 \\
f_7 &= f_4 - f_6 \\
f_8 &= f_3 * f_7 \\
f_9 &= f_1 + f_2 \\
f_{10} &= f_9 + f_8 \\
f_{11} &= f_1 * f_{10}
\end{aligned}
$$

3 Collecting subexpressions

For large formulas, the gain obtained by the above procedures pays the effort of the implementation, specially when the obtained expressions will be evaluated several times, or when relatively complex methods like automatic differentiation, automatic slopes or Hessian matrices are used, because in this case for each avoided operation in the expression at least one vector operation is discarded. Similar gains are also obtained when computations are to be done with relatively large objects, like complex and matrix operations (although in the general case one must be careful to assure that all properties of the objects are respected. So, for example, if a matrix expression is to be simplified the system must assure that multiplication is taken as non-commutative).

To reduce the effort further, after simplifying we may find repeated subexpressions that are far away from each other in the graph to reuse their results (in this case, our relatively simple algorithms will not find them in the simplification step). This is done as a last step when analyzing the graph to generate the factorization to be associated with the `Formula` variable.

The algorithm may be depicted as a repeated search in the graph, analyzing every pair of multiplications and selecting the pair with the largest common subset of two or more operands. This subset is represented in a node by itself, and the two nodes are changed to refer to it. At this point the graph has one more multiplication node corresponding to the shared expression, and the algorithm continues until no product is shared by any pair of multiplications. Thereafter, the same process is repeated for addition nodes.

Although it may seem trivial, some care is necessary for a few special cases. Some of these cases are summarized below:

1. If the operands of a node i are a proper subset of the operands of a node j. In this case no new node is needed and j is simply changed to point to i.

2. In the case of two exactly equal nodes i and j, one of them is deleted (say, j) and the nodes pointing to j are changed to point to i.

3. It may be the case that the best match is *inverted*, that is, two nodes i and j match left and right sides inversely. As an example, both expressions

$$x * \frac{y * z}{x - y} \quad \text{and} \quad x * \frac{x - y}{y * z}$$

would match quite well, but with sides changed for some operands. In this case, depicted in fig. 16, operands are shared between nodes, but these nodes do not use them in the same way. Instead, an inversion occurs. To explore this, the search for shared subexpressions tries both ways (regular and inverse) when analyzing which pair of nodes has the best subexpression. In fig. 16, computations are reduced by

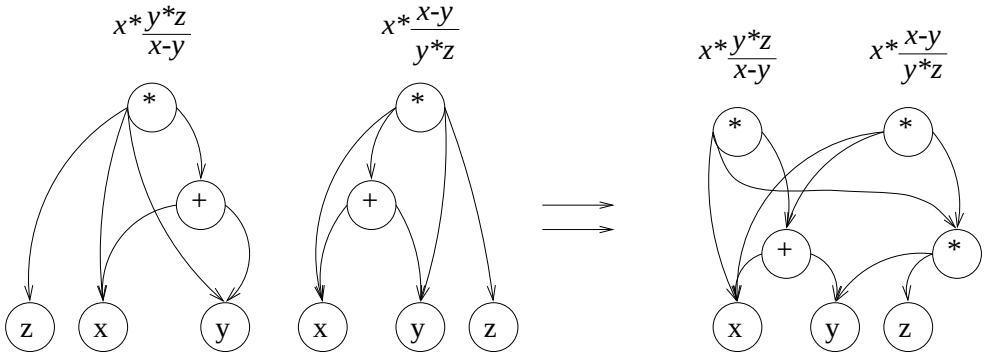

Figura 16: best match is inverted.

choosing an inverse match. This can be also presented in the following example: taking the expression

$$f(x,y,z) = x * \frac{y*z}{x-y} + \cos(x * \frac{x-y}{y*z}),$$

we notice that both the most external expression and the argument of the cosine function could share some common values. The table below shows two possible factorizations, the left one with no terms collected and the right one with full collect.

$$\begin{array}{lcl} f_1 & = & x \\ f_2 & = & y \\ f_3 & = & z \\ f_4 & = & f_1 - f_2 \\ f_5 & = & f_1 * f_2 \\ f_6 & = & f_5 * f_3 \\ f_7 & = & f_6/f_4 \\ f_8 & = & f_1 - f_2 \\ f_9 & = & f_1 * f_8 \\ f_{10} & = & f_3 * f_2 \\ f_{11} & = & f_9/f_{10} \\ f_{12} & = & \cos(f_{11}) \\ f_{13} & = & f_7 + f_{12} \end{array} \qquad \begin{array}{lcl} f_1 & = & x \\ f_2 & = & z \\ f_3 & = & y \\ f_4 & = & f_2 * f_3 \\ f_5 & = & f_1 - f_3 \\ f_6 & = & f_4/f_5 \\ f_7 & = & f_1 * f_6 \\ f_8 & = & f_1/f_6 \\ f_9 & = & \cos(f_8) \\ f_{10} & = & f_7 + f_9 \end{array}$$

The amount of operations used in each case can be summarized as below:

	$+,-$	$*,\div$
No collected terms	3	6
Collected terms	2	4

In our example for the `Formula` class this step will bring no further reduction as the sums and multiplications obtained after the simplification step contain no common terms. In the general case, however, a good reduction may be obtained if just a few common expressions are found.

4 Reducing the amount of memory

From this point no further reductions in code size will be tried, and we direct efforts to minimize the amount of memory needed, as well as assure its allocation before any calculations are done, and deallocation just when the corresponding `Formula` variable is destroyed. Doing so, we may be sure that repeated evaluations will not be hampered by memory management. This is really important when vectors are associated to the steps of a factorization, as in the case of automatic differentiation, automatic slope evaluation, and other methods. Assuring that no memory management will occur, one of the major drawbacks of object oriented programming is avoided, namely the large number of constructor and destructor routines when operations with large, complex objects are done.

The steps to assure a safe memory management will be described later in more detail, but they are essentially the following:

1. Finding the minimal amount of memory needed to evaluate the factorization.
2. Allocating that memory as an array belonging to the `Formula` variable.
3. Reindexing the steps in the factorization to use the memory of previous steps as they are not needed anymore.

First of all, we need to compute the minimal amount of memory needed to evaluate a factorization, and this clearly depends on the data types being used, both on precision (float, double, long double and others) and on their type (real, complex, interval and so on). Our example will be based upon the usual evaluation, but it is clear that we should allocate accordingly to the data type defined for the expression being evaluated.

Based on the factorization, it is easy to reduce the amount of memory needed to evaluate it. A useful step is a reordering of the factorization, moving variables and numerical constants to the beginning and keeping the arithmetical operations after them.

Using our example and reordering the steps, we associate to each step an index for a (future) memory area that will contain all intermediate results. We start from the first step, searching for the place where it is used by the last time and changing the step following this one to use the memory now available. As an example, we search for the place where f_1 is no longer needed, verifying that it is used in the very last step of the factorization, thus its memory cannot be used by other steps. Step f_2, however, is last used in the evaluation of f_7, so f_8 can use the memory previously allocated to f_2. Further, we search for the last occurrence of f_8, and reindex the step following this last occurrence (f_{10} in our example) to use the same memory location, that is, the place originally used to store f_2. Reaching the end of the factorization, we start searching for places where the area associated to f_3 can be used, and so on. Coming to a step where no index was attributed, we are forced to use another memory location (this will happen in steps f_5 and f_6).

The following table shows the original factorization and the successive columns present the index of every step, starting with step f_2 up to f_6, when all steps have their indexes defined. The last column shows the final form of the steps, using the indexes already defined on a memory area called *mem*.

Factorization			Index				Final code
f_1	=	x	1	1	1	1	$mem[1] := x;$
f_2	=	y	2	2	2	2	$mem[2] := y;$
f_3	=	z	3	3	3	3	$mem[3] := z;$
f_4	=	3	4	4	4	4	$mem[4] := 3;$
f_5	=	$f_1 + f_2$			5	5	$mem[5] := mem[1] + mem[2];$
f_6	=	$f_3 + f_5$				6	$mem[6] := mem[3] + mem[5];$
f_7	=	$f_2 * f_4$			5	5	$mem[5] := mem[2] * mem[4];$
f_8	=	$f_3 * f_7$	2	2	2	2	$mem[2] := mem[3] * mem[5];$
f_9	=	$f_6 - f_8$		3	3	3	$mem[3] := mem[6] - mem[2];$
f_{10}	=	$f_1 * f_9$	2	2	2	2	$mem[2] := mem[1] * mem[3];$

So, the above factorization may be evaluated using six memory locations, each one storing at least one intermediate result. As the process ends we also know where to place the results of all steps, using the indexes. It is important to notice that this is not the minimal number of memory locations, as this minimal number depends among other things on the ordering of the steps (specially the first four steps), and the search of the minimal amount is far too expensive to be used for problems with more than a few steps. Even if this approach is rather simple, it reduced the memory use in this example by 40%.

5 Last step: partial vectorization

This last step is quite different from the previous ones as it is just an extra facility to be used if some kind of vectorization is supported by the target machine. Experience shows that many scientific applications perform repeated iterations that usually depend on each other, but also there are many times when they do *not* strongly depend. Typical examples are methods for global optimization based on random search, when each sample is near some other sample but does not depend on it, or branch and bound methods analyzing a domain box at a time. In both cases, we could evaluate the target function for a few points (or boxes) at a time if we could pack them together and send them to a vector processor/machine. This also happens for many quadrature methods, where several independent evaluations of a function are used to compute a final value for some integral.

If our `Formula` class is to be implemented, we can add the capability of performing evaluations for several values instead of one at a time, just by associating vectors to each step and operating elementwise on each vector. As an example, let us suppose that we wish to sample 100 points of the function given in our example, using y and z fixed (say, $y = 0.1$ and $z = 0.2$) and x varying from 0.0 to 0.99. As we already know that just six memory locations are needed and each location has to be large enough to store 100 real numbers, we simply store an array of vectors *mem*, each vector corresponding to 100 real numbers and the factorization is changed accordingly:

$$
\begin{array}{lcll}
f_1 & = & x & mem[1] := (0.0, 0.01, 0.02, 0.03, \ldots, 0.99); \\
f_2 & = & y & mem[2] := (0.1, 0.1, 0.1, 0.1, \ldots, 0.1); \\
f_3 & = & z & mem[3] := (0.2, 0.2, 0.2, 0.2, \ldots, 0.2); \\
f_4 & = & 3 & mem[4] := (3, 3, 3, 3, \ldots, 3); \\
f_5 & = & f_1 + f_2 & mem[5] := mem[1] + mem[2]; \\
f_6 & = & f_3 + f_5 & mem[6] := mem[3] + mem[5]; \\
f_7 & = & f_2 * f_4 & mem[5] := mem[2] * mem[4]; \\
f_8 & = & f_3 * f_7 & mem[2] := mem[3] * mem[5]; \\
f_9 & = & f_6 - f_8 & mem[3] := mem[6] - mem[2]; \\
f_{10} & = & f_1 * f_9 & mem[2] := mem[1] * mem[3];
\end{array}
$$

At the end, $mem[2]$ contains all desired evaluations. Accordingly, we could sample several points of a function to be optimized, or evaluate interval bounds for several boxes at once in a branch and bound algorithm, as well as explore applications in computer graphics, specially rendering and surface lighting models.

It is possible to achieve a similar gain with machines that do not have vector facilities, since every time we group n points in a vector the number of function calls needed to evaluate these points is reduced by $n - 1$ times (with $n > 1$), avoiding context changes and manipulation of registers. Even better, if the processor being used has some kind of predictive scheme to prepare for the next instruction, as is the case for the Intel 586 family and others, the use of arrays will grant full advantage of this facility.

6 Conclusions

Starting from the need to reduce the amount of computations needed to evaluate expressions, we were able to develop a simple but very efficient strategy that starts with purely symbolical simplification procedures, reuses common subexpressions and can be also adapted to statically allocate all needed memory for the evaluation of expressions, using this capability to take advantage of vector facilities that may be present on the processor being used. All this can be made during compilation or as the code is executed by the first time, but we think that it should be ideally embedded into a compiler or preprocessor.

This set of algorithms, although powerful and reliable, took advantage of a very elegant representation that makes the implementation of the simplification rules much easier, and can be implemented on some 2000 lines of C++ code to generate output ideally suited to be used as input to a true compiler, that will (hopefully) perform further optimizations near the instruction level. So, we are in a certain sense just increasing the readability of arithmetical expressions, making them more easily manageable to the compiler and taking advantage of a series of facilities a compiler does not usually provide, as non-obvious simplification and repeated subexpressions.

It is interesting to notice that with simple syntactic changes on the output, we might generate straight FORTRAN, Matlab, Maple or Mathematica code using all their vector facilities and with reduced memory management on running time.

Referências

[1] Alefeld, G.; Herzberger, J.: **Introduction to Interval Computations**. Academic Press, New York, 1983.

[2] Beck, T.; Fischer, H.: **The if-problem in automatic differentiation.**, Journal of computational and applied mathematics, 50 (1994), 119-131.

[3] Breuer, M. A.: **Generation of optimal code for expressions via factorization**, Comm. ACM, 12 (1969), pp. 330-340.

[4] Fischer, H.: **Special problems in automatic differentiation.**, In *Automatic differentiation of algorithms: Theory, Implementation and Application*, A. Griewank and G. F. Corliss (eds.), SIAM, Philadelphia, Penn., 1991, pp. 43–52.

[5] Griewank, A.; Corliss, G. (editors): **Automatic differentiation of algorithms: theory, implementation and application**, SIAM Proceeding Series, Philadelphia, 1991.

[6] Kedem, G.: **Automatic differentiation of computer programs**, ACM Trans. on Mathematical Software, Vol. 6 (1980), No. 2, pp 150-165.

[7] Moore, R. E.: **Methods and applications of interval analysis**, SIAM Studies in Applied Mathematics, pp 24-31, Philadelphia, 1979.

[8] Moses, J.: **Algebraic simplification – a guide to the perplexed**, Comm. ACM 14 (1971), pp 548-560.

[9] Stroustrup, B.: **Die C++ Programmiersprache**, 2 ed., Addison–Wesley, Bonn, 1992.