Expression Templates for Dot Product Expressions

MICHAEL LERCH and JÜRGEN WOLFF VON GUDENBERG
Universität Würzburg, Am Hubland, 97074 Würzburg, Germany.
e-mail: {lerch, wolff}@informatik.uni-wuerzburg.de

(Received: 30 June 1998; accepted: 9 October 1998)

1. Introduction

A template is a parameterized data type. Expression templates [7] provide a means to obtain an application specific, user defined evaluation of expressions in C++. Instead of producing values, the operators are overloaded in order to return an appropriate template instantiation for each subexpression. Hence, the tree-like structure of an expression is represented by a data type for which specific evaluation strategies can be supplied. Since the structure of expressions is revealed during the instantiation of templates at compile time, it is possible to enforce several optimizations like loop unrolling or loop fusion which normally are not performed even by optimizing C++ compilers.

Dot product expressions on the other hand are mandatory in the design and development of self-verifying algorithms. In those algorithms, e.g. the verified solution of linear systems, the verification step often relies on the optimally accurate evaluation of generalized dot products, i.e. sums of products of matrices and vectors. They cannot be implemented with usual operator overloading, and therefore either require a syntax extension like in PASCAL–XSC or must be computed by accumulating the results element by element using scalar dotprecision variables like in C–XSC.

In our approach we use expression templates to obtain an easy to use, efficient C++ component for the computation of dot product expressions. Since loop fusion is done at compile time, we only need one dotprecision variable for arbitrary dot product expressions. The algorithms can easily be adapted to exploit parallelism in a distributed environment.

2. Dot Product Expressions

Crucial parts of self-validating algorithms are quite often the accurate evaluation of error terms or residuals, e.g. \( r = Ax - b \) for a matrix \( A \) and vectors \( r, x, b \). These generalized scalar products can be computed with only one rounding, if an optimal
scalar product is available. Hence they are provided in the XSC languages [3], [4].
Before commenting on the respective language embeddings and the description of
our new approach, we collect the requirements recommended for the syntax and
mandatory for the semantics.

- Each component of the resulting data structure has to be computed with maximal
  accuracy.
- Math-like notation, i.e. use of operators.
- For space and runtime efficiency do not allocate structures of exact intermediate
  results.
- Provide different rounding modes.

We explain the expression template technique and its application to dot product
expressions using the computation of defect vectors \( d = \Delta (A \tilde{x} - b) \) as an
example.

2.1. NOTATION: XSC LANGUAGES

First efforts in the exact evaluation of dot product expressions were made in
PASCAL-SC [2] where a \( \texttt{scalp} \) function was introduced that accumulates dot
products in an implicit long accumulator. In PASCAL-XSC [4] the importance of
dot product expressions had been recognized and \#-expressions were introduced.
The evaluation of a defect expression simplifies to

\[
d := \# (A \times x - b)
\]

Actually, arbitrary dot product expressions can be evaluated this way for four
rounding modes.

Although C-\textsc{Xsc} [3], a C++ class library, uses modern object oriented program-
ing language concepts, the expressiveness of \#-expressions in PASCAL-XSC
could not been retained, instead a procedural accumulation must be used.

2.2. NOTATION: EXPRESSION TEMPLATES

To provide vector/matrix arithmetic in C++ one usually defines class templates
such as \texttt{Vector<T>} and \texttt{Matrix<T>} which encapsulate the appropriate data
structures and the corresponding operators.

Clearly, these operators are only defined for partial expressions like \( A \times x \) and
some kind of delayed evaluation is needed to provide maximal accuracy for the
whole expression. If it were possible to overload the operators twice, this could be
achieved by a second meaning of operators that builds a symbolic parse tree of the
expression instead of evaluating it. Only the assignment operator then will do the
actual computations.

The trick with expression templates is to represent those parse trees as data
types, rather than as runtime objects of a suitable tree data structure. Arbitrary