Expression Templates in Ada

Alexandre Duret-Lutz

EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
http://www.lrde.epita.fr/
Alexandre.Duret-Lutz@lrde.epita.fr

Abstract. High-order matrix or vector expressions tend to be penalized by the use of huge temporary variables. *Expression templates* is a C++ technique which can be used to avoid these temporaries, in a way that is transparent to the user. We present an Ada adaptation of this technique which — while not transparent — addresses the same efficiency issue as the original. We make intensive use of the *signature* idiom to combine packages together, and discuss its importance in *generic programming*. Finally, we express some concerns about *generic programming* in Ada.

1 Introduction

One of the strongest requirements on source code is its *maintainability*, which essentially means that the program should be easy to understand and adjust. To avoid cluttering the actual algorithms with low-level details, current programming schemes promote *abstractions*. For instance, object oriented programming makes it possible to define and use high order operations on objects. For example, given an `Integer_10x10_Matrix` type and its accompanying operations, one could write the following expression.

```
declare
  A, B, C : Integer_10x10_Matrix;
begin
  [...]
  A := B * 5 + C;
  [...]
end;
```

While easier to read and maintain, this expression has a significant drawback over a hand-crafted loop over the matrices elements: it uses matrix-sized temporary variables to hold subexpressions. The penalty incurred by these temporary matrices can be a serious annoyance when such expression appears in speed critical subprograms.

There are various means to avoid temporary expressions, the most straightforward being a hand-crafted loop:

A. Strohmeier and D. Craeynest (Eds.): Ada-Europe 2001, LNCS 2043, pp. 191–202, 2001.

```
declare
  A, B, C : Integer_10x10_Matrix;
begin
  [...]
  for Row in Integer_10x10_Matrix'Range (1) loop
    for Col in Integer_10x10_Matrix'Range (2) loop
       A (Row, Col) := B (Row, Col) * 5 + C (Row, Col);
    end loop;
  end loop;
  [...]
end;
```

The expression now uses integer-sized temporaries (which can fit in registers) instead of matrix-sized temporaries (requiring memory usage). It is much faster but less readable, and can get cryptic when the expression become complex.

Other alternatives provide both efficiency and conciseness. *Domain Specific Languages* extend the underlying language, and "compilers" convert programs from the extended syntax to the native language; this however require an external tool. *expression templates* technique described below makes only use of the existing C++ languages features.

Since templates were introduced to C++, and more significantly since the introduction of STL [16,15], the C++ community developed a set of programming techniques globally referred to as *meta-programming* (for they abuse the compiler to perform computation or transformations at compile time) [11,12]. Most of these techniques are rather C++ specific because they rely on template specialization, a construction which is not available in Ada (see section 5.4). However this article presents an *attempt* to adapt one of these techniques to Ada.

In 1995, the C++ *generic programming* community introduced the *expression templates* [18,7,4]. *Expression templates* is a C++ programming technique which allows expressions like the first example of this section to be compiled like the second, *i.e.*, preventing the need for temporary variables. This is a powerful technique that allows to write fast and readable *user* code; the library, on the other hand, is significantly more complex (mostly because the way template functions are checked, see section 5.2).

In short, the idea is to build a type that represents the expression to assign, and an instance of that type which keeps references to the various subexpressions of the expression. The operators of the expression don't perform any computation, they simply return an object whose type represents the expression (for instance `plus<plus<vector<int>,vector<int> >,vector<int> >`). The actual computation is delayed until the assignment. This technique is transparent to the user, because the type construction is done by *implicit* template instantiations.

This paper tackles the adaptation of *expression templates* to Ada. Section 3 illustrates our solution with matrix computation, which rely heavily on the use of signatures recalled in section 2, and whose speed is compared to other approaches in section 4. Finally, we express some concern about the usage of Ada for *generic programming* [11] in section 5.

2 Expressing Concepts with Signatures

Ada 95 allows generic packages to be used as generic formal parameters. The Rationale [8] shows one use — also known as *signature* — of this possibility: characteristics of an abstraction are grouped using the formal parameters of a generic empty package.

This feature plays a significant role in *generic programming*. The STL documentation uses the term *concept* to designate a set of requirements (types or function definitions, behaviors), but these concepts have no mapping in the C++ syntax: concepts exist only in the documentation, the compiler is unaware of them and therefore it cannot help the programmer. When U. Erlingsson and A. Konstantinou adapted the STL to Ada [3,9], they found that *signatures* was the natural way to express *concepts*.

We will use *signatures* to express the concept of matrix type and matrix expression. In our matrix expression code, we want to let the user supply his own matrix type. For simplicity, we assume that matrices are always stored as double arrays[1]. Therefore we define the *matrix type* concept as a type of values, two ranges, and an array type. An instance of `Matrix_Type` will then be used everywhere matrix specifications are needed. The `Matrix_Expression` concept, for example, is defined by a matrix type and a function which can return elements from the matrix expression.

```
generic
   type Values is private;
   type Height_Range is range <>;
   type Width_Range is range <>;
   type Array_Type is array (Height_Range, Width_Range) of Values;
package Matrix_Type is end Matrix_Type;
```

```
with Matrix_Type;
generic
   with package Matrix_Spec is new Matrix_Type (<>);
   with function Get_Value (At_Row : in Matrix_Spec.Height_Range;
                            At_Col : in Matrix_Spec.Width_Range)
     return Matrix_Spec.Values;
package Matrix_Expression is
   procedure Assign (To : out Matrix_Spec.Array_Type);
end Matrix_Expression;
```

`Matrix_Expression` additionally declares the procedure `Assign` which is defined as follows. It will be used to evaluate a matrix expression while assigning the result to the target matrix.

```
package body Matrix_Expression is
   procedure Assign (To : out Matrix_Spec.Array_Type) is
   begin
      for Row in To'Range (1) loop
         for Col in To'Range (2) loop
            To (Row, Col) := Get_Value (Row, Col);
         end loop;
      end loop;
   end Assign;
end Matrix_Expression;
```

[1] Supporting other kind of storage is a matter of adding `Get_Value` and `Set_Value` in `Matrix_Type` and making `Array_Type` private.

Intuitively, `Assign` represents the loop we would have written manually, and `Get_Value` is used to evaluate the expression at the matrix cells level. Our objective is to build this function by combining generic packages.

3 Building Expressions

`Matrix_Expression` is the base building block for our matrix expressions. Our aim is to build a matrix expression from other matrix expressions. For example, given two instances of `Matrix_Expression` we want to apply a binary operator (element-wise). This is naturally done using a generic package `Matrix_Operators.Binary` parameterized by two matrix expressions and one operator function. Because this package can itself be seen as a matrix expression, it defines an instance of `Matrix_Expression`.

```
with Matrix_Expression;
generic
   with package Left_Expr is new Matrix_Expression (<>);
   with package Right_Expr is new Matrix_Expression (<>);
   with function Operator (Left : in Left_Expr.Matrix_Spec.Values;
                           Right : in Right_Expr.Matrix_Spec.Values)
     return Left_Expr.Matrix_Spec.Values;
package Matrix_Operators.Binary is

   function Get_Value (At_Row : in Left_Expr.Matrix_Spec.Height_Range;
                       At_Col : in Left_Expr.Matrix_Spec.Width_Range)
     Return Left_Expr.Matrix_Spec.Values;
   pragma Inline (Get_Value);

   -- Instances of Binary can be seen as a Matrix Expression:
   package Expr is
      new Matrix_Expression (Left_Expr.Matrix_Spec, Get_Value);

end Matrix_Operators.Binary;
```

The supplied operator is actually applied whenever `Get_Value` is called.

```
package body Matrix_Operators.Binary is
   function Get_Value (At_Row : in Left_Expr.Matrix_Spec.Height_Range;
                       At_Col : in Left_Expr.Matrix_Spec.Width_Range)
     return Left_Expr.Matrix_Spec.Values
   is
      -- we need to convert At_Row and At_Col to
      -- the range types used by the right expression.
      subtype Rh is Right_Expr.Matrix_Spec.Height_Range;
      subtype Rw is Right_Expr.Matrix_Spec.Width_Range;
   begin
      return Operator (Left_Expr.Get_Value (At_Row, At_Col),
                       Right_Expr.Get_Value (Rh (At_Row), Rw (At_Col)));
   end Get_Value;
end Matrix_Operators.Binary;
```

So far, we are able to compound matrix expressions with binary operators. Unary operators or other specialized operation (e.g. matrix multiplication) can be done likewise. The next step is to build atomic matrix expressions (i.e., matrices). Because a matrix expression is a package, we need a mean to convert

a matrix (double array) to a package. This can be done via a generic package which takes the matrix array as a generic parameter.

```
with Matrix_Type;
with Matrix_Expression;
generic
   with package Matrix_Spec is new Matrix_Type (<>);
   Object: in out Matrix_Spec.Array_Type;
   use Matrix_Spec;
package Matrix_Instance is

   -- read values from Object
   function Get_Value (At_Row : in Height_Range;
                       At_Col : in Width_Range) return Values;
   pragma Inline (Get_Value);

   package Expr is new Matrix_Expression (Matrix_Spec, Get_Value);

end Matrix_Instance;
```

We have now all the tools required to build and evaluate an expression. The following code creates the expression `Res_Expr` which can be used to compute `B*5+C`.

```
declare
   -- the user construct his own matrix type
   type Range3 is range 0 .. 2;
   type Matrix33 is array (Range3, Range3) of Integer;

   -- then he build a specification package for his matrix,
   -- this package will be used later when building high order
   -- operations on matrices.
   package Matrix33_Spec is new Matrix_Type (Values => Integer,
                                             Height_Range => Range3,
                                             Width_Range => Range3,
                                             Array_Type => Matrix33);

-- Define two dummy matrices.
B : Matrix33 := ((1, 0, 0), (0, 1, 0), (0, 0, 1));
C : Matrix33 := ((0, 1, 0), (0, 0, 1), (1, 0, 0));

-- Instanciate a package for the B matrix
-- (we map an object to a package)
package B_Inst is new Matrix_Instance (Matrix33_Spec, B);
-- the above package can the be used as a Matrix_Expression,
-- it defines a Expr subpackage for this purpose.
package B_Expr renames B_Inst.Expr;

-- idem with the second matrix
package C_Inst is new Matrix_Instance (Matrix33_Spec, C);
package C_Expr renames C_Inst.Expr;

-- Build the expression B*5
package B5_Inst is new Scalar_Binary (B_Expr, 5, "*");
package B5_Expr renames B5_Inst.Expr;

-- Build the expression B*5+C
package Res_Inst is new Binary (B5_Expr, C_Expr, "+");
package Res_Expr renames Res_Inst.Expr;
```

The resulting expression, `Res_Expr` can now be assigned to a matrix variable using the `Assign` function aforementioned. This assignment will actually evaluate the expression.

```
   A : Matrix33; -- where the result will be stored
begin
   [...]
   Res_Expr.Assign (To => A);
end;
```

It is important to note that the `B` and `C` matrices can be modified and that successive calls to `Res_Expr.Assign` will take these new values into account (this is a consequence of the `in out` mode used for the `Matrix_Instance.Object` parameter). By combining packages, we have built a function which fills a matrix according to the contents of `B` and `C`.

As this example shows, a somewhat large programming overhead is required to write an expression as simple as `B*5+C` using expression templates. This is to the point that the resulting code seems even more cryptic and less maintainable than the hand-crafted loop. However, it is interesting to see that this technique success in achieving good performance and might therefore be worth to consider in other fields than linear algebra; and as a C++ technique adaptation, it permits some comparison between the two languages for *static* component oriented programming.

4 Benchmark

Figure 1 gives the timing of the evaluation of the expression `Y = A + B + C` using different size of matrix and different expressions representations.

hand-crafted loop
The expression is computed using a double loop as shown in section 1.

package-built expression
The expression is built using packages, as described in section 3.

package-built expression (without inline)
Same as above, with the `-fno-inline` compiler option added.

standard expression
The expression is evaluated as-is, using the classical OO definition for the operator `"+"` which generate temporary variables in expressions.

abstract expression
Instead of using packages and signatures, we use tagged types and abstract tagged types (respectively) to build the expression. This correspond to the more classical object-oriented way to represent an expression (e.g. in a parser): `Expression` is an abstract tagged type which is derived into `Addition`, `Multiplication`, etc.

As it can be seen from the results, the timings of the package-built expression are equivalent to those of the hand-crafted version. Both are twice faster than expression computation using temporaries.

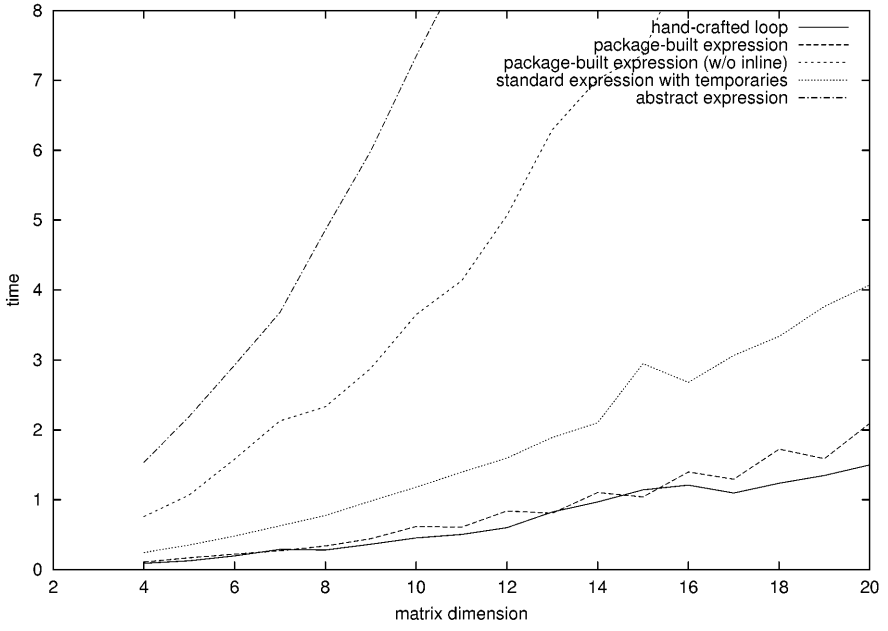

Fig. 1. Time required for the evaluation of expressions of the form `Y = A + B + C` The timing account only for the evaluation of the expression, the the building of the expression (when it is needed) is not accounted. The expression was evaluated 100000 times. The code, available from `http://www.lrde.epita.fr/download/`, was compiled with `gnat 3.13p` and run on an K6-2 processor. The compiler options used were `-gnatfvpnwuwl -O3`. `-fno-inline` option.

It is interesting to make a parallel between generic packages and tagged types. Generic packages have instances, as tagged types have, and signatures are the counterparts of abstract tagged types. Expression templates could be built using tagged types more easily, however tagged type suffer two performance loses: they hinder inlining, and add some overhead to handle dispatching calls. The comparison between the abstract expression and the non inlined package-built expression give an idea of that latter overhead. The main speedup obtained by using generics over tagged types is thus due the inlining calls allowed by these formers.

5 Critique of Ada

We have adapted a C++ technique to Ada (the converse is also possible, e.g. mixin inheritance is not well known in C++). Such conversion makes it possible to compare both languages and exhibit their weaknesses. Here, we focus on the issues we encountered in our attempts to adapt some C++ techniques to Ada.

5.1 Implicit Instantiation

Ada and C++ behave differently with respect to generic entities. C++ templates are instantiated implicitly the first time they are refered to. Ada generics need to be instantiated explicitly. This actually is a source of tediousness when using expression templates.

C++ implicit instantiation, though a source of errors, simplify the expression templates usage because the instantiations to perform are deduced from the expression itself.

5.2 Semantic Analysis

In C++, the semantic analysis (and in particular the type checking) of a function or class template is only performed after the template has been instantiated. In Ada, this analysis is done prior to any instantiation: the developer is therefore required to declare all the functions or types used (for instance one cannot use the `"+"` operator on a parameter type without declaring that this operator actually exists). This is definitely a better behavior because it allows errors to be caught earlier. C++ code can be wrong or make wrong assertions about the parameters, any error will only be revealed when the template is first instantiated, i.e., when it is first used (if it ever is).

The strong typing system of Ada is therefore a real help for the developer of a generic library, for it enforces the expression of requirements: the entities used by a generic package or subprogram must be listed in or deduced from the generic formal parameters list.

However, the difficulties arise when you start to parameterize a generic with several generic packages.

5.3 Additional Constraints on Package Parameters

The use of generic packages as formal parameters is a really powerful feature as far as *generic programming* is concerned, because it can factor requirements, allowing to express concepts (something really desired by the C++ *generic programming* community [14,10]). Still, as generic parameters get more complex, the need for constraints between parameters grows.

For instance, though the package `Matrix_Operators.Binary` allows to combine two instances of `Matrix_Expression`, it does not ensure that these two packages declare matrices of equal dimensions. Accesses to values out of the matrices ranges will throw a `Constraint_Error` at run-time, but it would be better to prevent the instantiation of `Binary` when the two operand-packages are not of equal dimensions, because this can be known at compile-time. Basically we want to ensure that `Left_Expr.Matrix_Spec.Array_Type` is the same type as `Right_Expr.Matrix_Spec.Array_Type`.

One way to constrain two non-limited types parameters to be equal is to require the availability of an equality operator for that type. For example the `Matrix_Operators.Binary` parameter list would become as follow.

```
generic
   with package Left_Expr is new Matrix_Expression (<>);
   with package Right_Expr is new Matrix_Expression (<>);
   with function Operator (Left : in Left_Expr.Matrix_Spec.Values;
                           Right : in Right_Expr.Matrix_Spec.Values)
     return Left_Expr.Matrix_Spec.Values;
   -- constraint
   with function "=" (Left : in Left_Expr.Matrix_Spec.Array_Type;
                      Right : in Right_Expr.Matrix_Spec.Array_Type)
     return Boolean is <>;
[...]
```

If `Left_Expr.Matrix_Spec.Array_Type` differs from `Right_Expr.Matrix_Spec.Array_Type`, the `"="` function *probably* does not exists and an instantiation attempt will lead to a compile-time error. Yet, this solution is not perfect: it won't work on limited types, or if the user has defined the checked `"="` function. Moreover the compiler is likely to complain about the absence of matching `"="`, which is not the best error message one would expect. Finally even if, to a certain extent, this can ensure the equality of two types, this does not make this equality explicit: the developer of the body still has *two* types to deal with and must perform conversion when needed because the compiler is unaware that the two types are equal.

A nice extension to Ada would be the addition of a whole sub-language to allow the expression of such constraints. E.g.

```
generic
   with package Left_Expr is new Matrix_Expression (<>);
   with package Right_Expr is new Matrix_Expression (<>);
   -- constraint (this is NOT Ada 95)
   where Left_Expr.Matrix_Spec.Array_Type
      is Right_Expr.Matrix_Spec.Array_Type;
```

Among the same lines, bounded genericity [1] on package parameters of generics would be useful. One doesn't always want to parameterize a generic with a single type of package, but for a whole set of package featuring a common interface: the present solution is to use a signature to express this interface and instantiate this signature in each package, however this is painful and hinder reusability. Being able to qualify formal package parameters with an interface would actually simplify expression templates implementation and usage a lot.

5.4 Template Specialization

Template specialization is among the most powerful features of C++, as far as *generic programming* is concerned. A fairly good number of *generic programming* techniques and idioms rely on template specialization. Ada does not support it (and this doesn't appear to be a trivial extension), therefore we list below some common use of template specialization in C++ and give hints about how it can be worked around in Ada.

Its primary use is to provide a better implementation of a generic entity for a given set of parameters. For example the minimum of a list can be computed more quickly when the list's elements are booleans, therefore C++ allows you to specialize your `min` function to the `list<bool>` case.

```
// generic minimum function for any (non-empty) list
template<typename T>
T min (const std::list<T>& l) {
  T m = std::numeric_limits<T>::max (); // maximum value for type T
  for (std::list<T>::const_iterator i = l.begin(); i != l.end(); ++i)
    if (*i < m)
      m = *i;
  return m;
}

// specialized version for lists of booleans
template<>
bool min (const std::list<bool>& l) {
  for (std::list<bool>::const_iterator i = l.begin(); i != l.end(); ++i)
    if (*i == false)
      return false;
  return true;
}
```

The C++ user will call `min` on a list without special care, and the compiler will implicitly instantiate the more specialized function for that particular kind of list. In Ada, since explicit instantiations are required anyhow, the Ada programmer would write two functions, say `Generic_List_Min` and `Bool_List_Min`, and left to the user the responsibility to choose the best implementation. However, this is not always practical: consider the writing of a generic package which should instantiate a `Min` function for one of it's type parameters, the appropriate implementation cannot be chosen unless `min` is actually a generic parameter of that package too.

As far expression templates are concerned, template specialization can be used to perform pattern matching on matrix expressions to call the corresponding BLAS[2] operations[20].

The second common use of template specialization is the building of traits classes [13,19]: traits classes are kind of static databases built using the type system. `numeric_limits` as used in the example above is a traits class defined in the C++ standard, the definitions of its members (e.g. `max()`) are specialized for the different type `T` available. Traits classes can be seen as an associative arrays between a type, and a signature-like class. Such associative array cannot be done in the Ada type system, therefore the associated signature has to be passed as another generic formal argument by the user.

Last, template specialization allows recurring templates, i.e. templates which instantiate themselves recursively and stop when they reach a specialized case. This is mostly used in *meta-programming*, where you force the compiler to compute some values at compile-time, or to perform loop unrolling [17]. Unfortunately, we did not found any work-around for this in Ada.

6 Conclusion

We have tackled the adaptation of expression templates to Ada. While our adaptation addresses one important issue covered by the original technique — the

[2] The *BLAS* (Basic Linear Algebra Subprograms) library provide optimized (and non generic) building block for matrix operations.

elimination of temporary variables — it is neither as powerful not practicable in Ada as it is in C++ where implicit instantiation makes it invisible to the user.

However, this paper shows one intensive use of the *signature* construction. This idiom is essential to generic programming, since it allows to work on *static abstractions* (i.e., abstractions resolved at compile-time, without any run-time cost), and is worth using when both high-order design and performance are required. Most of Gamma's design patterns [6] can be adapted to *generic programming* using *signatures* instead of abstract classes. In a previous work [2] we have shown such adaptations in C++; we also did some similar work in Ada, but it is still unpublished.

Last, we have combined packages to build a function (`Assign`). The small size of the building blocks used in matrix expression make the construction of such function a rather painful process comparatively to writing the same function manually. This technique deserves more experimentation too see how well it can serve in contexts were building blocks are larger.

References

1. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
2. Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille. Design patterns for generic programming in C++. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, Texas, USA, July 2001. To appear.
3. Ulfar Erlingsson and Alexander V. Konstantinou. Implementing the C++ Standard Template Library in Ada 95. Technical Report TR96-3, CS Dept., Rensselaer Polytechnic Institute, Troy, NY, January 1996.
4. Geoffrey Furnish. Disambiguated glommable expression templates. *Computers in Physics*, 11(3):263–269, May/June 1997. Republished in [5].
5. Geoffrey Furnish. Disambiguated glommable expression templates. *C++ report*, May 2000.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.
7. Scott W. Haney. Beating the abstraction penalty in C++ using expression templates. *Computers in Physics*, 10(6):552–557, Nov/Dec 1996.
8. Intermetrics, Inc., Cambridge, Massachusetts. *Ada 95 Rationale*, January 1995.
9. Alexander V. Konstantinou, Ulfar Erlingsson, and David R. Musser. Ada standard generic library. source code, 1998.
10. Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.
11. David R. Musser, editor. *Dagstuhl seminar on Generic Programming*, SchloßDagstuhl, Wadern, Germany, April-May 1998.
12. David R. Musser and Alexandre A. Stepanov. Generic programming projects and open problems. 1998.
13. Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5):32–35, June 1995.

14. Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.
15. Alex Stepanov. Al Stevens Interviews Alex Stepanov. *Dr. Dobb's Journal*, March 1995.
16. Alex Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, October 1995.
17. Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
18. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.
19. Todd L. Veldhuizen. Using C++ trait classes for scientific computing, March 1996.
20. Todd L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.