Compile Time Recursive Objects in C++

Jaakko Järvi
Turku Centre for Computer Science, Finland
jaakko.jarvi@cs.utu.fi

Abstract

This article explores the possibilities of generic programming offered by the template features of C++. We define compile time recursive objects *as instances of class templates which contain other instances of the same template as member variables. With such templates we can define containers that contain objects of arbitrary types, but where the type of each element is known at compile time. The structure of the container is therefore fixed. The technique mimics the polymorphism achieved with dynamic binding and inheritance using static binding and template specialisation. It is obviously less flexible but offers considerable performance gains at runtime.*

We give the template definitions for compile time recursive lists and explain how to perform operations on these lists. As an example application, we use compile time lists in the definition of templates for special sparse vectors and matrices. In these vectors and matrices, the sparseness pattern can be arbitrary but must be known at compile time. The tracking of zero and nonzero elements is performed at compile time. This allows the programmer to use abstract vector and matrix expressions and still reach performance equal to hand-coded operations. This is possible since the compiler can locate the elementary expressions between zero entries and discard them entirely from the resulting code.

Keywords: generic programming, C++ templates, template metaprogramming, sparse matrices

1: Introduction

In C++, generic classes and functions can be written using templates. The template formalism is very rich in the new C++ standard [1] and allows complicated type structures. For instance, compile time bounded loops and branching statements can be written using recursive template definitions and partial specialisation of class and function templates. Template usage of this type is called *template metaprogramming*. This novel technique has been previously used to speed up algorithms by *program specialisation* [2, 3], calculate binary constants at compile time [4] and define portable integral types [5].

In this article the idea of template metaprogramming is developed further and applied to *compile time recursive objects*. By compile time recursive objects we mean template classes which have other instances of the same template class as data members. Such objects are not so rare, e.g., a two-dimensional array can be implemented as a generic array class, where the elements themselves are arrays. Recursive tree-like objects are encountered in *expression*

This work was supported by the Academy of Finland, grant 37178.

templates [6], which are templates representing expression trees. This technique aims at avoiding the creation of temporary vector objects in the evaluation of vector expressions.

This article examines the nature of compile time recursive objects more profoundly. We define templates which can be used as containers where the type of each element can be arbitrary but statically bound, being thus fixed at compile time. We show how to perform operations with them.

The most basic construct is a recursive list class, which is discussed in detail. We show the template definitions and describe how different operations for these types can be written. From a general approach we move to a specific application. We show how efficient operations for sparse vectors and matrices can be written using compile time recursive lists and template metaprogramming.

The foremost benefits achieved with the techniques presented are evidently computational speed and, to some extent, type safety. The concept of compile time recursive objects as such is intriguing and needs to be explored more.

At the moment, some of the template features used in this article are not implemented in all of the currently available C++ compilers. All features are, however, part of the new standard for C++ and will therefore most likely be implemented by major compiler vendors soon.

1.1: Template metaprogramming

As an introduction to a reader not familiar with the newest C++ template features, we briefly describe the concept of template specialisation and give an example of a simple template metaprogram. Using template specialisation, partial or total, alternative definitions for a *primary* template class can be given. During the instantiation of a template, the types of the template parameters determine whether the instance is generated using the primary template or one of the specialisations. The template arguments are matched with the template argument lists of each specialisation and the most specialised definition is selected. This is done according to a set of partial order rules. Note that the selection must be unambiguous. Basically the same mechanism applies both to class and function templates. In function templates the template arguments can be deduced from the function arguments.

Template metaprograms utilise template specialisation. The primary template is defined as recursive and specialisations are used to end the recursion. As an example of a simple template metaprogram, consider the following code calculating the factorial at compile time:

```
template<int N> struct F { enum { val = N * F<N-1>::val }; };
template<> struct F<0> { enum { val = 1 }; };
```

In the primary template, the enumeration constant `val` is intended to contain the factorial of the template parameter `N`. In the definition of the constant `val` of the class `F<N>`, the constant `val` of the class `F<N-1>` is used. Hence, the instantiation of `F<N>` triggers recursively the instantiation of `F<N-1>`. A specialisation for `F<0>` is given to end the recursion. All this occurs at compile time and `val` gets the value `N!` at compile time.

The above is an example of compile time recursion with respect to integral template parameters. In the remainder of this article, the recursion takes place with respect to normal type parameters.

2: Compile time lists

Consider the traditional LISP view of a list as a record consisting of a head and tail, which is itself a list. Such a list can be written with the following template definitions:

```
template <typename HT, typename TT>
class list {
public:
  typedef HT head_type;
  typedef TT tail_type;

  head_type head;
  tail_type tail;
};

class nil() {};
```

The list contains only two member variables, head and tail. The typedefs are provided to allow reference to the types of the variables in a uniform way. The definitions allow considerable freedom in instantiation, as HT and TT can be of arbitrary types. The template is, however, not intended to be instantiated arbitrarily. HT can be an arbitrary type but TT should be another instance of the list template or the nil class, which is the end mark of the recursive list. It would be easy to constrain the template to allow only such instantiations where the tail is of type list or nil. This would, however, complicate the syntax and for simplicity is not shown.

Note that the elements of the list may be of arbitrary types, so the class is a *heterogeneous statically bound container*. As an example consider the following:

```
class A; class B; class C;
list<A, list<B, list< A, list< C, nil >>>> x;
```

According to this definition, x is an object which has objects of types A, B, A, C and nil as nested member variables. These variables can be accessed using the common notation (e.g. x.tail.tail.head) but also using generic functions and classes. E.g., the nth item of the list can be accessed at compile time, a certain operation can be performed on each element in the list and generic operations between these structures can be defined. To clarify how compile time lists can be manipulated, we give two examples. The first shows how to perform a certain operation for each object in the list and the second describes the concatenation of two compile time lists.

2.1: Operations of compile time lists

Let us assume that each element class X has a function void apply(X&) defined, which performs some action on the element. Each element is passed to the correct apply function with a call to the following function:

```
template<typename T> inline void traverse(T& a) {
  apply(a.head);
  traverse(a.tail);
};

void traverse(nil&) {};
```

Note that this seemingly recursive function is not really recursive. Although it calls a traverse function, it is another instance of the same function template and thus a different function.

This is the basic scheme for processing objects in the list. Operations can be more complicated and they may potentially produce new instances of the list template, i.e., new types. These types must be expressed using the template parameters of the generic function. This is achieved by using a C++ idiom known as *traits* [7]. Note that these trait templates can also be recursive. The concatenation illustrates this point:

```
template<typename L1, typename L2>
inline type_concat<L1, L2>::t concat(const L1& list1, const L2& list2) {
  type_concat<L1, L2>::t result;
  tr_concat(list1, list2, result);
  return result;
};
```

This is the interface function, which creates an object to be returned, passes it along the input parameters to a compile time recursive traversing function tr_concat and finally returns the result. The resulting type is given by type_concat<L1, L2>::t, which is a type definition in a generic class type_concat. The sole purpose of this trait class is to define the type of the concatenation given the types of the lists to be concatenated. The generic class is defined as

```
template<typename L1, typename L2>
struct type_concat {
  typedef list<L1::head_type, type_concat<L1::tail_type, L2>::t> t;
};

template<typename L2> struct type_concat<nil, L2> { typedef L2 t;}
```

The type of the concatenation of two lists is thus a list, where the head is of the same type than the head of the first list. The type of the tail is given by concatenating the tail of the first list with the second list recursively. The partial specialisation is for ending the compile time recursion in the case where the first list is empty.

To traverse the lists and perform the actual concatenation, a template function and two partial specialisations are needed:

```
template<typename L1, typename L2, typename L3>
inline void tr_concat(const L1& a, const L2& b, L3& result) {
  result.head = a.head;
  tr_concat(a.tail, b, result.tail);
};

template<typename L2, typename L3>
inline void tr_concat(const nil& a, const L2& b, L3& result) {
  result.head = b.head;
  tr_concat(a, b.tail, result.tail);
};

template<typename L3> inline void tr_concat(const nil&, const nil&, L3&) {};
```

The given templates for lists and list manipulation illustrate the basic mechanisms of class definitions and operations for compile time recursive objects. To be feasible, the usage of these templates must be reasonably convenient. The programmer should not have to write such declarations as `list<x, list<y, ... >>` or access elements using such notation as `aList.tail.tail.tail.head`. For better syntax, special templates for element access and initialisation of objects are needed, even macro definitions may prove useful. Since these needs and possibilities may be application dependent, they will be discussed more with the matrix example below.

2.2: About compilation

In order to understand the potential benefits offered by the above described techniques, some issues concerning compilation and optimisation must be discussed. One can notice that traversing the list structures produces many functions calls. These functions are, however, very small. They commonly perform some operation on the list heads and pass the tails to another function. This means that the functions can be effectively inlined and thus their overhead eliminated. With this in mind, it can be seen that traversing these compile time structures and accessing their elements include no runtime cost. Inlined traversal functions produce unrolled loops when compiled and only the operations performed on the individual elements of the lists yield any code. Another important aspect is the static binding of the elements. Though the elements in the container can be of arbitrary types, the functions called for each object in the container are bound statically. This makes the inlining possible here as well. To illustrate the use of the recursive compile time objects, we proceed with a concrete example concerning sparse vectors and matrices.

3: Sparse vectors and matrices as compile time recursive objects

A vector or matrix is called sparse, if a considerable degree of its elements are zero. The sparseness of vectors and matrices can be exploited when performing computations with them. Vector and matrix operations can be defined to perform only those elementary operations which yield nonzero results. To exploit the sparseness, some indexing scheme must be used to find the right operands for the elementary operations. This bookkeeping causes extra overhead. If the vectors and matrices are small or contain relatively many nonzero elements, sparse techniques may become costly due to this overhead. As an example, consider the sparse vectors $a = (1, 0, 1, 0, 0)$ and $b = (0, 0, 2, 0, 2)$ and let $c = a + b$. To calculate the resulting vector $c = (1, 0, 3, 0, 2)$, only one addition and two value copy operations are needed. However, due to the indexing overhead, it would probably be much faster to perform all five additions than to apply some sparse scheme.

If the sparseness patterns of the vectors and matrices are known beforehand, the programmer may eliminate the indexing cost by writing specialised code which performs only the necessary operations. Hence the programmer in effect performs the indexing himself. This is of course very tedious and error-prone. Using compile time recursive objects the sparseness pattern can be expressed in the type of a vector or matrix. It is then possible to make the compiler generate this specialised code from abstract vector and matrix operations and thus avoid the bookkeeping cost entirely. To accomplish the task, we first define template classes for *compile time sparse vectors*.

3.1: Compile time sparse vectors

Compile time sparse vectors (CTSV) are basically compile time lists and can be derived from the list template classes given above:

```
template <typename HT, typename TT>
class ctsv : public list<HT, TT> {};

class zero { public: zero(const zero&) {} }
```

For simplicity, no members are given for the ctsv class here. It would, however, be worthwhile to define various assignment operations and constructors. Objects of the class zero are used as the zero elements in the ctsv. Note that the zero class has a copy constructor defined. Even though the class zero is totally empty, the objects of this class reserve some memory. This is to guarantee that each object has an identity. By defining explicitly the copy constructor to be empty, we prevent the copying of this extra memory, which might be performed by the compiler-generated default copy constructor. With this class definition the creation and copying of the zero objects bears no runtime cost.

With the above definitions, the type of a vector $(1.1, 0, 0, 2.2)$ would be

```
ctsv<double, ctsv<zero, ctsv<zero, ctsv<double,nil>>>>.
```

The type clearly contains the information about the sparseness pattern of the vector.

The intention is that the programmer can write abstract vector expressions with CTSVs and the types are deduced automatically leading to more complex types as the computation advances. To define operations between these vectors, we need to define the resulting type of the operation, how to find the matching elements within each vector, and how to apply the operation between matching elements. We give the definition of the addition operation, since it is illustrative. Other operations can be defined using the same principles.

3.1.1: Addition of two vectors

To define the result type of addition, we deal with the type changes between the element types. Again, this is done with trait classes:

```
template<typename T1, typename T2> struct type_sum {};
template<> struct type_sum<double, double> { typedef double t; };
template<> struct type_sum<double, zero> { typedef double t; };
template<> struct type_sum<zero, double> { typedef double t; };
template<> struct type_sum<zero, zero> { typedef zero t; };
```

These only state that whenever a double is involved in the addition, the result is of type double. The templates could also be defined in a more general fashion using partial specialisation.

The type of the whole addition is defined as a recursive trait class, analogously to the type deduction of the list concatenation:

```
template<typename H1, typename T1, typename H2, typename T2>
struct type_sum<ctsv<H1, T1>, ctsv<H2, T2> > {
  typedef ctsv<type_sum<H1, H2>::t, type_sum<T1, T2>::t> t;
};

template<> struct type_sum<nil, nil> { typedef nil t;};
```

The addition operator is itself just an interface function. It creates the object to be returned, lets the actual work be done by the traversing function add and finally returns the result:

```
template<typename HT1, typename TT1, typename HT2, typename TT2>
inline type_sum<ctsv<HT1, TT1>, ctsv<HT2, TT2> >::t
operator+(const ctsv<HT1, TT1>& a, const ctsv<HT2, TT2>& b) {
  type_sum<ctsv<HT1, TT1>, ctsv<HT2, TT2> >::t result;
  add(a, b, result);
  return result;
};

template<typename V1, typename V2, typename V3>
inline void add(const V1& a, const V2& b, V3& result) {
  add(a.head, b.head, result.head);
  add(a.tail, b.tail, result.tail);
};

template<> void add(const nil&, const nil&, nil&) {};
```

The add functions call the elementary add functions to add the heads and then the tails recursively. The elementary add functions are defined as

```
inline void add (const double& a, const double& b, double& c) { c = a + b; };
inline void add (const double& a, const zero& b, double& c) { c = a; };
inline void add (const zero& a, const double& b, double& c) { c = b; };
inline void add (const zero& a, const zero& b, zero& c) {};
```

These functions obey the type promotions defined with the type_sum traits. It is easy to see that when compiled, these functions yield the minimal instructions for summation. To add two zero objects, no code is produced, the addition of a double and a zero results in a single value copy operation and the addition of two doubles in a single addition. These are the functions which perform the additions optimally, the rest of the definitions are actually only for determining the correct function to be called at compile time.

As an example, consider the previous vectors a, b and c. The sparseness patterns of the vectors are $(1, 0, 1, 0, 0)$, $(0, 0, 1, 0, 1)$ and $(1, 0, 1, 0, 1)$, respectively. The following assembly language code is the result of the compilation of the statement c = a + b using KAI C++ 3.2.d compiler under Linux for Intel Pentium processor.

```
movl    -16(%ebp),%eax    // move 1st element of a
movl    %eax,-64(%ebp)
flds    -8(%ebp)          // add the 3rd elements
fadds   -32(%ebp)
fstps   -56(%ebp)
movl    -24(%ebp),%eax    // move the 5th element of b
movl    %eax,-48(%ebp)
```

As can be seen, the code is minimal, performing only the elementary operations needed without any control instructions.

3.2: Compile time sparse matrices

A matrix is a vector of vectors, hence we can use the ctsv template for matrices as well. E.g., the matrix

$$\begin{pmatrix} a & 0 & b & 0 \\ 0 & c & 0 & 0 \\ d & 0 & e & 0 \\ 0 & 0 & 0 & f \end{pmatrix}$$

is of type

```
ctsv< ctsv< double, ctsv< zero,   ctsv< double, ctsv< zero,   nil >>>>,
ctsv< ctsv< zero,   ctsv< double, ctsv< zero,   ctsv< zero,   nil >>>>,
ctsv< ctsv< double, ctsv< zero,   ctsv< double, ctsv< zero,   nil >>>>,
ctsv< ctsv< zero,   ctsv< zero,   ctsv< zero,   ctsv< double, nil >>>>,
nil >>>>
```

The previously defined vector operations can be used directly. For instance, the addition was defined to first add the heads and then the tails recursively. Now the addition of heads is a vector operation instead of a scalar operation and it is performed by adding first the heads and then the tails recursively. New operations, such as matrix transpose and multiplication, must of course be defined. The type deduction classes tend to become rather complicated and they are not shown here.

3.3: Declaring types and initialising variables

The vector and matrix types are rather tedious to write as such. Various helper templates can be defined to ease this task. We can achieve such notation as `V<1,0,1,0,0>::t` for defining vector types and

```
typedef M< V<1, 0, 1, 0>::t,
           V<0, 1, 0, 0>::t,
           V<1, 0, 1, 0>::t,
           V<0, 0, 0, 1>::t>::t matrix_type_1;
```

for matrix types. The syntax is fairly reasonable. The assignment of values to vectors and matrices can also be simplified by using the operator overloading capabilities of C++. The comma operator can be overloaded to provide a means to initialise the vectors and matrices with values.

```
matrix_type_1 m; double a, b, c, d, e, f; zero z;
m =  a,  z,  b,  z,
     z,  c,  z,  z,
     d,  z,  e,  z,
     z,  z,  z,  f;
```

This is quite convenient, particularly since an attempt to set a zero to a nonzero position or vice versa results in an error at compile time. For vectors the initialisations are as effective as normal assignments to named variables. For matrices there may be some overhead. It is, however, easy to initialise matrices rowwise.

Usually the sparseness patterns of the initial vectors or matrices are simple and more complicated types result from expressions between them. These complicated types need not be specified, since they are automatically deduced. A good example of this is *automatic differentiation* [8]. It is a technique for computing derivatives without the need to symbolically differentiate the function, but avoiding the use of divided difference approximations as well. CTSVs have proved useful in this application as shown in [10, 11], where they are used to represent the vector of partial derivatives in automatically differentiable numbers. These vectors are initially canonical unit vectors, consequently their types are easy to define. These vectors are used as variables in mathematical expressions. During the evaluation several intermediate temporary vectors and finally a dense vector are produced. The intermediate vectors may have arbitrarily complex sparseness patterns, but as pointed out, they are automatically deduced and need not be specified.

3.3.1: Transformation matrix example

As an example of the usage of CTSVs, consider the type matrix_type_1 defined above. It defines the sparseness pattern of a transformation matrix in the homogeneous coordinate system. When a vector is multiplied with the matrix

$$\begin{pmatrix} cos(\theta) & 0 & -sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

it is rotated around the y-axis [9]. Using CTSVs, a subroutine to perform this transformation can be written as follows:

```
void rotate1(double theta, const dense_vec<4>::t& v, dense_vec<4>::t& result) {
  matrix_type_1 m; zero z;
  double cost = cos(theta), sint = sin(theta);

  row<0>::get(m) = cost,   z,   -sint,   z;
  row<1>::get(m) = z,      1,   z,       z;
  row<2>::get(m) = sint,   z,   cost,    z;
  row<3>::get(m) = z,      z,   z,       1;

  result = v*m;
}
```

dense_vec<4>::t gives the type of a dense four-element vector, and the odd-looking code row<N>::get(m) is just an invocation of a template metaprogram to get the Nth row of the matrix m for rowwise initialisation.

Despite the syntactical differences, the code has apparent correspondence with the mathematical notation of the transformation. This is not the case for a low-level implementation of the same transformation, where the matrix multiplication is decomposed into elementary operations by hand:

```
void rotate2(double theta, double v[ ], double result[ ]) {
  double cost = cos(theta), sint = sin(theta);

  result[0] = v[0] * cost + v[2] * sint;
  result[1] = v[1];
  result[2] = v[2] * cost - v[0] * sint;
  result[3] = v[3];
}
```

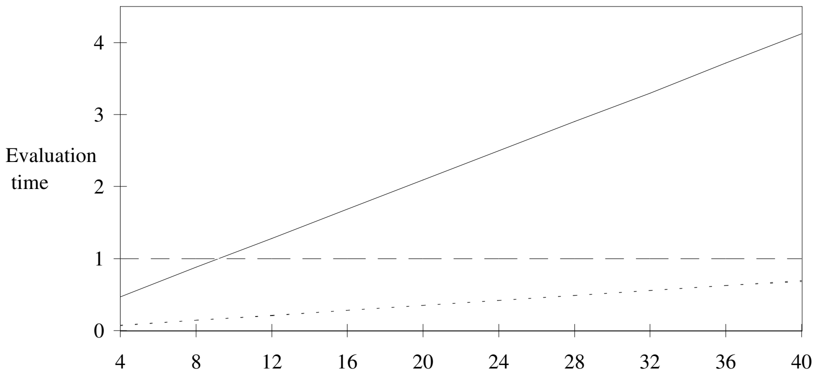

Figure 1. The relative evaluation time of the dot product of two vectors of length 40; *dotted lines:* **CTSVs and hand-coded C (they overlap entirely);** *dashed line:* **dense vectors;** *solid line:* **runtime sparse vectors. The number of nonzero elements is shown on the x-axis.**

One might suspect that the low-level implementation would be considerably faster, but when compiled (again with KAI C++, optimisation flags +K3 -O2), the CTSV implementation produced 45 sequential machine instructions compared with the 44 instructions of the low-level code. Further, if the traversing function was called directly (replacing the last line `result = v*m;` of `rotate1` with the line `multiply(v,m,result);`) the assembly listings were exactly identical.

3.4: Speed comparisons

To demonstrate the attainable performance gains of CTSVs, we measured the computing time of a dot product of two double vectors (Fig. 1). The length of the vectors was 40 elements, and the number of nonzero elements was increased from 4 to 40 with step size 4. The nonzero elements were in the same positions in both vectors, so the number of multiplications needed was the same as the number of nonzero elements. We used four different vector representations: ordinary dense vectors, ordinary sparse vectors, CTSVs and hand-coded C. In the ordinary dense vectors the dot product is a for loop from 1 to 40, so the cost is not dependent on the number of zeroes. The ordinary (runtime) sparse vectors were implemented as an array of values and a corresponding array of indices. The CTSVs were as described in this article, and finally in the hand-coded C version, each elementary operation was written directly, so the loop was totally unrolled by hand and only the nonzero elementary operations were computed. To our knowledge, the hand-coded C was the fastest way to perform the dot product without resorting to any processor-dependent tricks or inline assembly code.

The test was compiled with the KAI C++ 3.2.d compiler using optimisation flags +K2 and -O2 and it was run under Linux operating system on a Pentium PC. The test shows that the normal sparse techniques become unattractive quite soon as the number of nonzero entries increases. The execution speed of the CTSV dot product and the hand coded C version was almost identical in all cases. This was evident since the assembly code they produced was essentially the same.

It can also be seen that the unrolled versions were somewhat faster than the for loop implementation even when the vectors had no zeroes. This lead vanishes if the vector sizes are increased to the point where the code no longer fits in the processor cache.

In [10, 11], more execution speed comparisons between CTSVs, runtime sparse vectors and dense vectors can be found. The results were very favourable for CTSVs. Also, the speed of automatic differentiation of various expressions was compared using different implementations of the partial derivative vector. CTSVs outperformed clearly other alternatives and came very close to the speed of symbolically differentiated codes. The implementation of CTSVs was slightly different but the assembly code produced was similar, so the results are applicable here as well.

4: Restrictions in compilation

Since the template definitions of compile time recursive objects can be deeply recursive, there will of course be some practical limitations to the depth of the recursion set by the compiler. Some compilers may have strict limits for the number of nested recursive template instantiations, but several compilers provide a parameter for controlling this, allowing thus arbitrarily many nested instantiations. At some point, available resources, such as the amount of memory or the compilation time needed, set limits to this technique.

However, our experiments show that the technique can be applied to reasonably sized objects and compiling is fairly rapid. In the case of CTSVs, the compilation of a 100-element dot product succeeded with no difficulties. The compilation of matrix operations is somewhat harder since the type deductions are more complicated. However, as CTSV operations produce a block of sequential (unrolled and inlined) code, it is hardly desirable to apply the technique for very long vectors or large matrices.

5: Conclusions

This article presents compile time recursive objects as a new technique for generic programming in C++. The technique relies heavily on partial template specialisation and recursive template definitions. It is shown how to define a compile time list structure, and how to perform operations on it. With this template the programmer can define special heterogeneous container types which may contain objects of arbitrary types and yet the types are statically bound. Despite this, the elements of these containers can be accessed and operated on in a uniform manner, and due to the static binding, also very rapidly. The application of a certain template function to each element of such container has no runtime cost of traversing through the collection. The cost originates only from the operations performed on each element.

This article shows how template classes for special sparse vectors and matrices can be defined as compile time recursive objects. In these vector objects, the type includes the information about the positions of the nonzero elements. This makes it possible for the compiler to decide which elementary operations must be performed and which yield a zero element and thus can be discarded. It is shown that with compile time recursive objects, the programmer can write abstract vector and matrix operations and does not lose anything on performance compared with coding all elementary operations by hand. In our examples, the compiler generated assembly code essentially equivalent to the hand-coded alternative.

The concept of compile time recursive objects is attractive. The technique can be seen as an imitation of the polymorphism achieved with dynamic binding and inheritance, by static binding and template specialisation. Since the main advantage offered by this is speed, scientific computing is the potential application area for compile time recursive objects. This article demonstrates the usefulness of the technique in the implementation of special sparse vectors and matrices.

References

[1] ISO/IEC 14882 Standard for the C++ Programming Language.

[2] Veldhuizen, T.: Using C++ template metaprograms, C++ Report **7** no. 4 (1995) 36-43.

[3] Veldhuizen, T.: Linear algebra with C++ template metaprograms, Dr. Dobb's Journal **21** no. 8 (1996) 38-44.

[4] Pescio, C.: Binary Constants using Template Metaprogramming, C++ Users Journal, February (1997).

[5] Pescio, C.: Template Metaprogramming, C++ Report **9** no. 7 (1997) 22-27.

[6] Veldhuizen, T.: Expression Templates, C++ Report **7** no. 5 (1995) 26-31.

[7] Myers, N. C.: A new and useful template technique: "traits", C++ Report **7** no. 5 (1995) 32-35.

[8] Rall, L. B.: Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science **120** (1981) Springer-Verlag, Berlin.

[9] Mortenson M. E.: Geometric Modeling, (1985) Wiley, New York.

[10] Järvi J.: Processing Sparse Vectors during Compile Time in C++, Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science **1343** (1997) 41-48 Springer, Berlin.

[11] Järvi J.: Compile Time Sparse Vectors in C++, Turku Centre for Computer Science, Technical Report No. 107 (1997) (www.tucs.fi/publications).