Generative Programming for Automatic Differentiation in C++0x

Marco Nehmeier

Institute of Computer Science, University of Würzburg

Am Hubland, D 97074 Würzburg, Germany

`nehmeier@informatik.uni-wuerzburg.de`

December 15, 2011

Abstract

The computation of the first or higher derivatives of a function is a common problem in scientific computing.

The most obvious approach to compute the derivative values is the symbolic differentiation which applies the well known rules of differentiation onto the expression to compute a formal expression of the derivative function.

Another method called numerical differentiation is the approximation of the derivative with difference quotients.

Superior to these approaches is the automatic differentiation which is a technique to compute a value of the expression and the derivative together by applying the well known rules of differentiation. The difference to symbolic differentiation is that it propagates numerical values instead of formal expressions.

Commonly used implementations of automatic differentiation may be divided into two categories, the implementations using operator overloading to compute the values of the function and the derivative together in contrast to special tools applying the technique of source transformation to mix in the expressions to compute the derivatives. Both of these techniques are somehow inflexible in the manner that the operators are not overloaded for high order differentiation or additional tools are required for the source transformation.

In this paper we use expression templates and template meta programming to mix in the code for the partial differentiation. The main concept is to apply the symbolic differentiation at compile time only onto the single operations of the expression tree to create the run time code for the automatic differentiation. This has two advantages, we can, in theory, compute derivatives of any order and secondly the differentiation of the operations is done with types. This means that the symbolic differentiation of an operation with a specified order is performed only once by the compiler.

Our generic approach provides a domain specific language for partial automatic differentiation of an arbitrary order which is easily extendable by new types. Several C++ and especially C++0x template programming concepts and techniques like variadic templates, type lists or variadic tuples are used to realize the specification of the automatic differentiation at compile time. Common template meta programming techniques are refined.

We tested the functionality, correctness and performance of our implementation in different case studies for floating point as well as interval data types and compared it against other implementations.

1 Automatic Differentiation

The computation of the first or higher derivatives of a function is a common problem in scientific computing like optimization, solving of nonlinear equations or nonlinear inverse problems.

Besides the specification of the derivative formula as a formal expression by hand using the well known rules of differentiation there are some methods to compute the derivatives of a function automatically.

The analogon to the manual differentiation is the symbolic differentiation which is commonly provided by computer algebra systems like Mathematica or Maple. For this approach the (symbolical) rules of differentiation are applied onto a formal expression to compute another formal expression defining the derivative of the function. This is a feasible approach for problems with modest size but the limitation of symbolic processors is reached quickly [8].

Another method called numerical differentiation is the approximation of the derivative with difference quotients. In practice this approach suffers from quirks of the floating point numbers like truncation errors and roundoff errors [21].

The third and most powerful method is the automatic differentiation [22] which circumvents the drawbacks of the symbolical as well as numerical differentiation. The main concept of the automatic differentiation is to compute the value of the expression and the derivative together by applying the well known rules of differentiation. The difference to symbolic differentiation is that it propagates numerical values instead of formal expressions. This has the benefit that the result of the automatic differentiation is as accurate[1] as the result of the symbolic differentiation but could be efficiently computed without the expression swell of symbolic processors [8].

The technique of automatic differentiation can be divided into two different modes, the forward mode and the backward mode. The forward mode follows the normal computation of the expression applying the chain rule from right to left propagating the values from the input variables toward the output variables [6] or in the case of an expression tree from the leaf nodes toward the root node. In contrast to the forward mode the backward mode use the chain rule from left to right or in case of an expression tree the computation is per-

[1]If the computation is done with real arithmetic the result is exact [29].

formed from the root node toward the leaf nodes [6]. Both of these modes have their advantages and disadvantages. The reverse mode is cheaper in relation to the required operations for the computation but is consuming much more memory [6].

Besides the two different modes commonly used implementations of automatic differentiation may be divided into two categories [5], the implementations like ADOL-C [12], FADBAD++ [4] or C-XSC [13] using operator overloading to compute the values of the function and the derivative together as well as special tools like ADIC [7] or TAPENADE [15] applying the technique of source transformation to mix in the expressions to compute the derivatives. Both of these techniques are somehow inflexible in the manner that the operators are not overloaded for high order differentiation or additional tools are required for the source transformation.

In this paper we only cover the forward mode which is easy to compute by replacing the return value of a node of the expression tree specifying the (sub-) expression $f : \mathbb{R} \to \mathbb{R}$ by a tuple

$$(f(x), \frac{\mathrm{d}f}{\mathrm{d}x}(x), \ldots, \frac{\mathrm{d}^n f}{\mathrm{d}x^n}(x)) \tag{1}$$

storing the result of the function f as well as the derivatives up to the desired order n. For a constant c or a variable x the values are easily replaced by the required tuples as follows:

$$c \to (c, 0, 0, \ldots, 0) \tag{2}$$
$$x \to (x, 1, 0, \ldots, 0) \tag{3}$$

The basic arithmetic operations and elementary functions are replaced as well by operations and functions working on these tuples.

$$(u\,,\,u') \cdot (v\,,\,v') := (u \cdot v\,,\,u \cdot v' + u' \cdot v) \tag{4}$$
$$\sin((u\,,\,u')) := (\sin(u)\,,\,u' \cdot \cos(u)) \tag{5}$$

The formulas (4) and (5) are two examples for the replaced basic arithmetic operations and functions which compute the first derivative by applying the basic differentiation rules.

Figure 1 shows the expression tree for the expression $\sin(x) \cdot c$ together with the assigned tuples for the automatic differentiation of the first order.

Partial differentiation as well as the differentiation of other types like intervals are handled in a similar manner.

2 Expression Templates

One of the outstanding features of C++ is the feasibility to overload several operators like the arithmetic operators, the assignment operator, the subscript

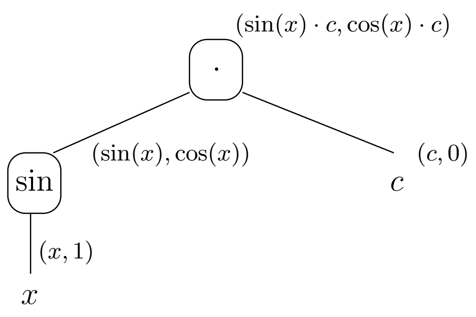


Figure 1: Expression tree of the expression $\sin(x) \cdot c$ showing the assigned tuples of the automatic differentiation of the first order.

operator or the function call operator. A developer of particularly mathematical data types may now implement the interface of this types as a domain specific language embedded into C++. But besides the possibility to define an easy and intuitionally usable data type, the operator overloading in C++ has a drawback called pairwise evaluation problem [27]. This means that operators in C++ are defined as unary or binary functions. For expressions with more than one operator like `r = a + b + c` this has the consequence that `a + b` is evaluated first and than their result, stored in a temporary variable, is used to perform the second operator to evaluate the addition with the variable `c`. Obviously this leads to at least one temporary variable for each `+` operator but if the data types are e.g. vectors it also performs several consecutive loops to perform the particular vector additions. Both, the temporary variables as well as the consecutive loops are a performance penalty compared to a hand coded function computing the result with a single loop and no temporaries [27, 24], see Listing 1.

```
for (int i = 0; i < a.size(); ++i)
    r[i] = a[i] + b[i] + c[i];
```

Listing 1: Hand coded and optimal computation of `r = a + b + c`.

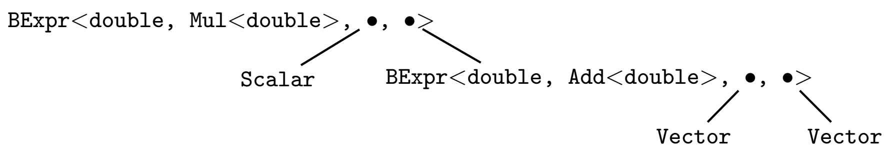
Expression templates [25] are a C++ technique to avoid these unnecessary temporaries as well as the consecutive loops by generating an expression tree composed of nested template types. Furthermore, this expression tree is explicitly visible at compile time and could be evaluated as a whole resulting in a similar computation as shown in Listing 1. This could be achieved by defining a template class `BExpr<typename T, class OP, typename A, typename B>` as an abstract representation of a binary operation[2] specified by the template parameter `OP` as an exchangeable policy class [1] working on the arguments of the types `A` and `B` which are stored as reference member variables.

[2]Types for operations with a different order are defined in a similar manner.

```
template<typename T, class OP, typename A, typename B>
class BExpr {
    A const& a_;
    B const& b_;
  public:
    BExpr(A const& a, B const& b) : a_(a), b_(b) { }

    T operator[] (size_t i) const {
        return OP.eval(a_[i], b_[i]);
    }
    ...
};
```
Listing 2: Class BExpr<typename T, class OP, typename A, typename B>.

Thereby, the template type T specifies the type of the vector elements which is important to implement the element-wise evaluation using the subscript operator. In principle, the subscript operator uses the policy class specified by the template parameter OP to compute the ith element of the result of the operation. Listing 3 shows an implementation of a policy class for a vector addition.

```
template<typename T> struct Add {
    static T eval(T a, T b) { return a + b; }
};
```
Listing 3: Policy class Add<typename T>.



BExpr<double, Mul<double>, •, •>
Scalar
BExpr<double, Add<double>, •, •>
Vector
Vector

Figure 2: Tree structure for the expression Scalar * (Vector + Vector).

With these building blocks it is quite easy to overload the arithmetic operators for vector and scalar types to implement the operations like the vector addition or the multiplication with a scalar returning an expression tree. Figure 2 shows the composed tree structure for the expression Scalar * (Vector + Vector). Therefore, expression templates are a technique to compose expression trees with nested template classes at compile time.

The evaluation of such an expression tree then could be performed as a whole by overloading the assignment operator. For the example with vector operations it is quite easy by using one loop over the size of the vectors computing the result element-wise by calling the overloaded subscript operator of the tree[3].

Obviously, there are a lot of small objects and inline functions used for the expression template approach but due to optimizations included in modern compilers, this evaluation leads almost to a similar machine code as Listing 1.

[3]Note that for this approach it is required that the Vector class as well as the Scalar class overload the subscript operator. For an Scalar it could easily return the value of the scalar.

Using expression templates for vector operations to avoid unnecessary loops and temporaries is the classical example used in [25] to introduce this powerful technique. Besides loop fusion, expression templates are used in many other areas like the adaptation to grids in finite element methods [14], to increase the accuracy in dot product expressions [16] or to reduce the necessary but time-consuming rounding mode switches of the floating point unit to speed up interval computations [20, 19].

3 Template Meta Programming

Besides the expression templates template meta programming [26] is another powerful programming technique offered by the C++ templates approach. Amazingly, this feature was not intended by C++ language designers and was primarily detected by Erwin Unruh [23] who demonstrated how to compute prime numbers at compile time. After this example by Erwin Unruh, further research in the area of template meta programming was investigated and it was shown that the C++ compiler is turing complete [28] and could act as an interpreter working on types or values of primitive data types at compile time.

This offers the opportunity to write compile time functions like the exponential function in Listing 4 which could be used to compute constant values like `Exp<2,4>::result` at compile time.

```
template<int B, int N>
struct Exp {
    enum { result = B * Exp<B, N-1>::result };
};

// terminating condition B^0
template<int B>
struct Exp<B, 0> {
    enum { result = 1 };
};
```

Listing 4: Compile time function to compute the exponential function B^N.

Listing 4 shows clearly the main concept of template meta programming. Recursive instantiations of template classes are used to construct the required “program flow” at compile time. The result of the function is then stored in a constant variable or an `enum` which can be accessed from the outside. In this example, the most important concept is the partial specialization of the template class to realize a terminating condition of the computation for the parameter `N = 0`.

Furthermore, partial specializations of template classes can be used to implement control structures like *if-then-else* statements, see Listing 5.

```
// General case ( Cond == true )
template<bool Cond, class Then, class Else>
struct IF {
    typedef Then action;
};

// Cond == false
template<class Then, class Else>
struct IF<false, Then, Else> {
    typedef Else action;
};
```

Listing 5: Compile time *if-then-else* control structure.

Other control structures like *switch* statements or loops can be implemented in a similar manner [9] which offers a way for generative programming. For example, the *if-then-else* statement of Listing 5 could be used for code selection between two different implementations [9] depending on the value of the condition, see Listing 6 .

```
IF<con, AlgoA, AlgoB>::action::execute();
```

Listing 6: Compile time selection of the method `execute()` of the classes `AlgoA` or `AlgoB` depending on the value of the constant expression `con`.

In addition to the compile time functions and the compile time control structures, there exist some other useful template based techniques like traits [18] or type lists [1] which could be used for generative programming during compile time.

4 Implementation

The inflexibility of common implementations of automatic differentiation using operator overloading or source transformation as well as the positive qualities of expression templates and template meta programming for the use of generative programming motivated us to implement automatic differentiation using template based technologies.

The process of the code generation for the automatic differentiation of our implementation is divided into two steps:

1. The creation of the expression tree using expression templates.

2. The code generation of the automatic differentiation out of the expression tree using template meta programming.

4.1 Creation of the Expression Tree

The expression tree in our approach has, excepted for the leaf nodes, only symbolic characteristics. All the behavior of the represented expression later is associated via trait classes during the code generation of step 2. All the inner nodes of the tree are almost empty classes distinguished by different template parameters to specify their behavior.

But first of all we need different types to specify the leaf nodes of the expression tree. Because there are constants and different variables required for a partial differentiation, we introduce the type `BaseType<typename T, class D>` as a base class for variables, constants, and as well for the root nodes of the expression tree.

```
template<typename T, class D>
struct BaseType {
    typedef D deri;

    deri const& derivedType() const {
        return static_cast<deri const&>(*this);
    };
};
```

Listing 7: Base class for variables, constants, and root nodes.

Use of this base class has the advantage that the operators and functions, which are necessary to construct the expression tree, only have to be overloaded for the base type. The concrete types of the several instances can be determined or obtained using the typedef `deri` or the method `derivedType()`, respectively. This is possible because the type of the subclass is specified as the template parameter `D` of the base type, see Listing 7, 8 and 15 for details.

```
template<typename T, unsigned int N>
class Var : public base::BaseType<T, Var<T, N>> {
    // implementation details ...
};
```

Listing 8: Skeleton of the class for Variables.

Listing 8 shows the skeleton of the class `Var<typename T, unsigned int N>` derived from the base type implementing the type for different independent variables. Thereby, the template parameter `T` specifies the underlying type for the automatic differentiation like `double` or `interval`. The parameter `N` is an identifier which is used to separate the different independent variables at compile time using different `int` or `char` values, e.g `Var<double, 1>`, `Var<double, 2>` or `Var<double, 'x'>`, `Var<double, 'y'>`.

A type `Con<typename T>` representing constants of the expression is defined in a similar manner as the class `Var<typename T, unsigned int N>` but without the necessity of the template parameter `N`.

The introduction of independent variables demands for a technique to keep track of all different variables used in an expression to generate the code for the partial differentiation correctly. This could be done by using a type list [1] for storing each independent variable of the expression. With the upcoming

C++ standard (C++0x) [3] this could easily be realized by using variadic templates [11] which are a feature to use an arbitrary number of types or values of a primitive data type as template parameters. Listing 9 shows an implementation which can be used to store all the different identifiers of the independent variables. Thereby the ellipsis after the keyword `unsigned int` of the template identifier `Vars` in the first line of Listing 9 labels this parameter as a template type parameter pack [11] which can be used with an arbitrary number of arguments.

```
template<unsigned int... Vars>
struct VarList { };
```

Listing 9: Class storing an arbitrary number of `unsigned int` values as template parameters.

With the feasibility of "unpacking" a fixed number of arguments of the parameter pack, it is possible to address each element using recursive template specializations. The last specialization in Listing 10 illustrates the separation of the first argument `Head` from the tail of the parameter pack. This offers the possibility to write compile time functions for the search or the insertion[4] of values into a `VarList<Vars...>`. Listing10 shows a compile time union of two `VarList<Vars...>` which is required to determine the independent variables of two subtrees of an expression tree, see Listing 15 for details.

```
template<typename L1, typename L2>
struct union;

template<typename L2>
struct union<VarList< >, L2> {
    typedef L2 list;
};

template<unsigned int Head,
    unsigned int... Vars, typename L2>
struct union<VarList<Head, Vars...>, L2> {
    typedef typename union<VarList<Vars...>,
            typename insert<Head,L2>::list>::list list;
};
```

Listing 10: Compile time union of two `VarList<Vars...>`.

Now it is possible to define the type `Expr<typename T, class OP, class List>` for the root node of our expression tree. Listing 11 shows the specialization using a `VarList<Vars...>` as the third template parameter to keep track of the independent variables. Note that this specialization is also derived from the base type of Listing 7 to disburden the operator overloading for the creation of the expression tree.

[4]Note that for our implementation each value is inserted only once into the `VarList<Vars...>`.

```
template<typename T, class OP, unsigned int... Vars>
class Expr<T, OP, varlist::VarList<Vars...>>
    : public base::BaseType<T, Expr<T, OP,
        varlist::VarList<Vars...>>> {
  private:
    OP op_;

  public:
    typedef varlist::VarList<Vars...> list;
    typedef OP type;

    Expr(OP const& op) : op_(op) {}

    OP const& rep() const {
        return op_;
    }
};
```

Listing 11: Root node of the expression tree.

More precisely, the class `Expr<typename T, class OP, class List>` is not the true root node of the expression tree. Rather it is a wrapper class around the actual root node of the type `OP` to alleviate the handling of the expression tree. The actual tree is then composed of `Var<typename T, unsigned int N>` and `Con<typename T>` types for the leaf nodes and instances of the type `ETNode<class Policy, class... Para>` for the inner nodes. This new class for the inner nodes as well uses variadic templates which offer the feasibility for operations/nodes with an arbitrary number of parameters or successors, see Listing 12.

```
template<class Policy, class... Para>
class ETNode {
  public:
    typedef Policy policy;
    typedef std::tuple<Para...> tuple;

    ETNode(Para const& ... para) : s_(para...) { }

    tuple& successor() {
        return s_;
    }

    tuple const& successor() const {
        return s_;
    }
  private:
    tuple s_;
};
```

Listing 12: Inner node type of an expression tree with an arbitrary number of successors.

As described above, the inner nodes of the expression tree have only a symbolic behavior which is specified by the policy class [1] associated with the template parameter `Policy`. Listing 13 shows the definition of such a policy class for the sinus function.

```
template<typename T, class P>
struct Sin {
  // cos(P) * P'
  typedef Mul<T, Cos<T, P>, typename P::df> df;
};
```

Listing 13: Policy class for the sinus function

A typedef of nested template specifications of this new descriptive policy classes, applying the basic rules of differentiation, is used to define the derivative `df` of these operations. Additionally the descriptive policy class `RefId<typename T, unsigned int I, unsigned int N>`, see Listing 14, is defined to handle the relation of the operation and its arguments. Thereby, the template parameters stands for the $\texttt{N}^{\text{th}}$ derivative of the $\texttt{I}^{\text{th}}$ argument.

```
template<typename T, unsigned int I, unsigned int N>
struct RefId {
    typedef RefId<T, I, N + 1> df;
};
```

Listing 14: Policy class describing the $\texttt{N}^{\text{th}}$ derivative of the $\texttt{I}^{\text{th}}$ argument.

With these descriptive policy classes it is easy to assemble the nested policy `OP` describing the operation required by an inner node. The following line defines the policy `OP` for a multiplication

$$\texttt{OP} := \texttt{Mul<RefId<0,0>,RefId<1,0>>}$$

as a multiplication of the values ($\texttt{0}^{\text{th}}$ derivative) of the first (index `0`) and second (index `1`) argument. Now the derivative of this operation

$$\begin{aligned}\texttt{OP'} &= \texttt{OP::df}\\ &= \texttt{Add<Mul<RefId<0,0>,RefId<1,1>>,}\\ &\qquad\quad \texttt{Mul<RefId<0,1>,RefId<1,0>>>}\end{aligned}$$

can be easily deduced by using the typedef `df` of the policy `OP` at compile time.

With all these building blocks it is now easy to define the functions or to overload the operators to create an expression tree similar to Figure 2. Listing 15 shows the overloaded `+` operator. Note that this single operator is sufficient to work with all different subtypes of the base type `BaseType<typename T, class D>`.

```
template<typename T, class P1, class P2>
Expr<T, ETNode<Add<T, RefId<T, 0, 0>, RefId<T, 1, 0>>,
    typename P1::type, typename p2::type>,
    typename listunion<typename P1::list,
    typename P2::list>::list>
operator+ (BaseType<T, P1> const& p1,
    BaseType<T, P2> const& p2) {

    typedef typename listunion<
        typename P1::list,
        typename P2::list>::list list;

    typedef ETNode<Add<T, RefId<T, 0, 0>,
        RefId<T, 1, 0> >,
        typename P1::type,
        typename p2::type> node;

    typedef Expr< T, node, list> expr

    return expr(node(p1.derivedType().rep(),
        p2.derivedType().rep()));
}
```

Listing 15: Overloaded + operator

4.2 Code Generation

After the creation of the expression tree, it is time for the code generation of the automatic differentiation. For this process, a new data type is required which stores the results of the partial differentiation with the requirement that the specific references for the different values are accessible at compile time. This offers the recursive type `Derivative<typename T, unsigned int N, class List>`, see Listings 16 and 17.

```
template< typename T, unsigned int N>
class Derivative<T, N, VarList< >> {
  protected:
    T val_;
  public:
    Derivative() : val_() {};

    T& val() { return val_; }
    T const& val() const { return val_; }
};
```

Listing 16: Data structure for the partial differentiation used as a terminating condition for the inheritance.

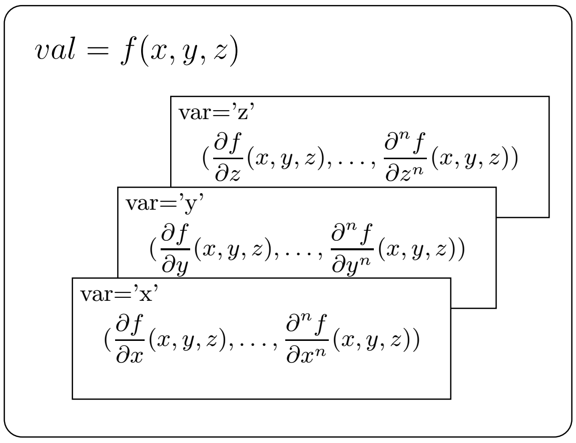
```
template<typename T, unsigned int N,
    unsigned int E, unsigned int ... Vars>
class Derivative<T, N, VarList<E, Vars... >>
    : public Derivative<T, N, VarList<Vars...>> {
  public:
    typedef typename crTuple<N, T>::type tuple;
    typedef Derivative<T, N, VarList<Vars... >> inherited;

    enum {var = E};

    Derivative() : t_() {};

    tuple& derivatives() {
        return t_;
    }

    tuple const& derivatives() const {
        return t_;
    }

    inherited& nextVar() {
        return *this;
    }

    inherited const& nextVar() const {
        return *this;
    }

  private:
    tuple t_;
};
```

Listing 17: Data structure for the partial differentiation.

This data structure can be seen as container which stores the value of the function as well as a tuple `std::tuple<T,...,T>` of the size `N` for each independent variable storing the derivatives from the first up to the required order `N`. This tuple structure could be easily generated with a recursive compile time function `crTuple<unsigned int N, typename T>`. Figure 3 shows the aspired structure of the type `Derivative<typename T, unsigned int N, class List>`.

The implementation is realized with a recursive inheritance of the same data type with a reduced `VarList<Var...>`, see Listing 17. Listing 16 shows the specialization for an empty `VarList<>` which is the terminating condition for the inheritance. Additionally, the variable for the function value is defined in this specialization.

This recursive structure using inheritance as well as the tuple type `std::tuple<T, ..., T>` offers the possibility to access references of each value at compile time. For tuple types this is achieved by the template function



$val = f(x,y,z)$
var='z'
$(\frac{\partial f}{\partial z}(x,y,z),\ldots,\frac{\partial^n f}{\partial z^n}(x,y,z))$
var='y'
$(\frac{\partial f}{\partial y}(x,y,z),\ldots,\frac{\partial^n f}{\partial y^n}(x,y,z))$
var='x'
$(\frac{\partial f}{\partial x}(x,y,z),\ldots,\frac{\partial^n f}{\partial x^n}(x,y,z))$

Figure 3: Aspired structure of the data structure for the partial differentiation.

`std::get<int I, class Tuple>(Tuple&)` accessing the reference of the $\texttt{I}^{th}$ element at compile time. For the data structure of a partial differentiation a function `get<unsigned int V, unsigned int I, class Tuple>(Tuple&)` could be defined in a similar manner accessing the reference of the $\texttt{I}^{th}$ partial derivative of the independent variable with the identifier `V`. Note that the identifier for the partial differentiation is stored in an enum `var`, see Listing 17.

With these new data structure as well as our expression tree from Section 4.1 it is now possible to generate the code for the automatic differentiation up to the $\texttt{N}^{th}$ order using template meta programming. This can be done by defining a template class `Eval<typename T, class DF, class Node>` which provides a method `static void eval(DF& df, Node const& n)` for the code generation. Thereby, the template parameter `DF` describes the data structure from Figure 3. In this case, `Node` is the type of the actual node of the expression tree. Note that the variables as well as the required order `N` for the differentiation are already specified by the type `DF`, see Listings 16 and 17. The method for the code generation then performs the following steps:

1. Mix in the run time code to create instances $\texttt{df}_1$, ..., $\texttt{df}_n$ of the type `DF` for the evaluation of the n successors of the current node. These instances are stored in a tuple `t` of the type `std::tuple<DF, ..., DF>`.

2. Use the class `Eval<typename T, class DF, class Node>` recursively to mix in the evaluation of the child nodes, writing their results (value and derivatives) into the instances $\texttt{df}_1$, ..., $\texttt{df}_n$.

3. Mix in the run time code for the evaluation of the operation and its partial derivatives of the independent variables $\texttt{V}_1$,..., $\texttt{V}_k$ by applying the tuple `t` containing the instances $\texttt{df}_1$, ..., $\texttt{df}_n$ onto the corresponding trait `EvalPol<typename T, unsigned int V, class OP>`.

- get<V_1,0>(df) = EvalPol<T,V_1,OP>::eval(t)
- get<V_1,1>(df) = EvalPol<T,V_1,OP::df>::eval(t)
- get<V_1,2>(df) = EvalPol<T,V_1,OP::df::df>::eval(t)
- ...

The whole code generation process is illustrated in Figure 4.

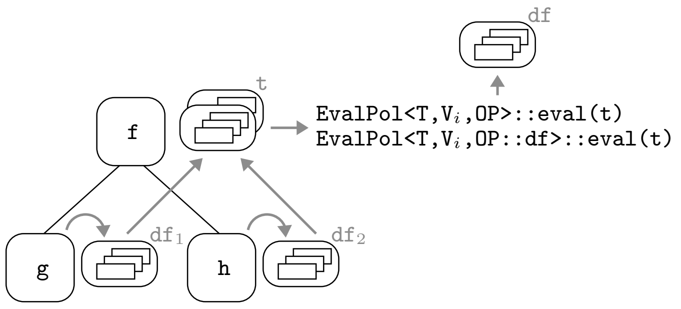

df
t
f
EvalPol<T,V_i,OP>::eval(t)
EvalPol<T,V_i,OP::df>::eval(t)
df$_1$
df$_2$
g
h

Figure 4: Performed steps of the evaluation for an inner node, computing the function `f(g(...),h(...))` together with its derivatives.

The trait class `EvalPol<typename T, unsigned int V, class OP>` is a mapping between the symbolic policy classes from Section 4.1 and the required operation. The specialization in Listing 18 realizes a mapping for the type in Listing 14 to access the Nth partial derivative for the variable `V` of the Pth child node.

```
template<typename T, unsigned int V,
    unsigned int P, unsigned int N>
struct EvalPol<T, V, RefId<T, P, N>> {
    template<class DF_Tuple>
    static T eval(DF_Tuple const& dft) {
        return get<V, N>(std::get<P>(dft));
    }
};
```
Listing 18: Mapping between a symbolic policy class and the required operation for the function sinus.

Listing 19 shows the evaluation of an addition by calling the trait recursively and applying its results onto the associated function of the template class `TypePol<typename T>` which is responsible for the correct basic operation for a data type `T`.

```
template<typename T, unsigned int V, class P1, class P2>
struct EvalPo<T, V, Add<T, P1, P2>> {
    typedef EvalPol<T, V, P1> evalP1;
    typedef EvalPol<T, V, P2> evalP2;

    template<class DF_Tuple>
    static T eval(DF_Tuple const& dft) {
        return TypePol<T>::add(
            evalP1::eval(dft),
            evalP2::eval(dft));
    }
};
```

Listing 19: Mapping between a symbolic policy class and the required operation for an addition.

The evaluation of the whole expression tree can be realized by extending the type `Expr<typename T, class OP, class List>` in Listing 11 by a template method `df<unsigned int N>()`, see Listing 20.

```
template<unsigned int N>
Derivative<T, N, list>  df() const {
    typedef df::Derivative<T, N, list> DF;
    DF res;

    df::Eval<T, DF, OP >::eval(res, op_);

    return res;
}
```

Listing 20: Method for the code generation and evaluation.

This realizes a domain specific language for partial automatic differentiation which is flexible, fast and easy to use, see Listing 21.

```
Var<double, 'x'> x(1.5);
Var<double, 'y'> y(2.3);
Con<double> c(2.0);

auto res = (x * x + c * y).df<2>();

cout << "d/dx "   <<  get<'x', 1>(res)  << endl;
cout << "d/dydy " <<  get<'y', 2>(res)  << endl;
```

Listing 21: Example for the partial automatic differentiation up to the second order for the expression $x^2 + c \cdot y$

4.3 Extensibility for other Types

As described in Section 4.2, the correct basic operations for a type `T` are associated through a template class `TypePol<typename T>`. This concept allows us to easily extend our automatic differentiation approach by new data types. Listing 22 shows the specialization for the type `double`. The integration of other types is done similarly.

```
template<>
struct TypePol<double> {
    static double const zero() {
        return 0.0;
    }

    static double const one() {
        return 1.0;
    }

    static double const add(double a, double b) {
        return a + b;
    }

    ...

    static double const sin(double a) {
        return ::sin(a);
    }

    ...
};
```

Listing 22: Implementation of the class `TypePol<double>`.

Note that the two methods `zero()` and `one()` are required to create the corresponding values for the specified data type to realize the differentiation of variables and constants. Listing 23 shows the integration of filib++ for the automatic differentiation using intervals.

```
template <typename T, filib::rounding_strategy K,
    filib::interval_mode E>
struct TypePol<filib::interval<T,K,E>> {
    static filib::interval<T,K,E> const zero() {
        return filib::interval<T,K,E>(0.0);
    }

    static filib::interval<T,K,E> const one() {
        return filib::interval<T,K,E>(1.0);
    }

    ...
};
```

Listing 23: Integration of filib++ for the automatic differentiation using intervals.

5 Experimental Results

	$x^2y^3 + y\log(x)$	$3x^2y - y^3$	$(1-x)^2 +$ $100(y-x^2)^2$
ET	7.24 ms	2.36 ms	3.20 ms
By hand	14.45 ms	1.67 ms	1.89 ms
FADBAD++	73.80 ms	60.41 ms	85.66 ms
ADOL-C	199.97 ms	180.47 ms	200.39 ms
ADOL-C reused tape	127.94 ms	115.88 ms	121.59 ms

Table 1: Performance comparison of the automatic differentiation of the first order in `double` precision.

For our test environment we used a Linux 2.6.32 64 Bit system running on an Intel(R) Core(TM)2 Quad CPU Q9550@2.83GHz with 8 GB random access memory. As a compiler we used the GNU C++ compiler g++ 4.4.3 with the options `-O2` as well as `-std=c++0x` for the required C++0x support.

As a first test case we compared our expression template approach (ET) in `double` precision against the two common automatic differentiation libraries FADBAD++ and ADOL-C 2.2.0 with the following three expressions:

$$x^2y^3 + y\log(x) \tag{6}$$
$$3x^2y - y^3 \tag{7}$$
$$(1-x)^2 + 100(y-x^2)^2 \tag{8}$$

In this case we measured the time required for the computation of the automatic differentiation of the first order for 100000 randomized input variables x and y. Thereby, the randomized values for the input variables are computed before the measurement and are stored on the heap. During the measurement, the input variables are read from the heap, used for the computation and the results (function as well as the partial derivatives) are written back to the heap. Table 1 as well as Figure 5 show the measured run time (the mean of 10 runs).

Note that ADOL-C works with an internal data structure called *tape* which records all calculations involving variables between the two function calls `trace_on()` and `trace_off()` [12]. Afterwards, the recorded *tape* can be used to perform the automatic differentiation. In our measurement, we have used two different test cases for ADOL-C. The first one records a new *tape* for each computation. For the second one the *tape* is recorded only once during the measurement and then is reused for the computation of the other values.

In addition to the two automatic differentiation libraries, we have measured the time for a differentiation by hand. Surprisingly for expression (6), our approach is faster then the hand coded differentiation. Thereby, the symbolic differentiation of the subexpression $y \cdot \log(x)$ requires a second computation of

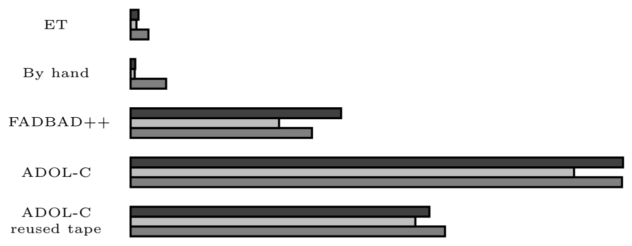


Figure 5: Performance comparison of the automatic differentiation of the first order in `double` precision for the expressions (6) [gray], (7) [light gray] and (8) [dark gray].

the function $\log(x)$. With an optimized hand coded version which reuses the result of $\log(x)$, the run time of the hand coded version could be reduced to 7.16 ms.

Additionally to the test case in `double` precision, we used our approach together with the interval library filib++ to measure the behavior of our implementation together with interval arithmetic. For the measurement we compared our implementation against the common interval library C-XSC 2.3.1.

The first test cases are similar to the test in `double` precision. We computed the derivatives of the functions

$$x^3 + x - 1 \tag{9}$$

$$e^x \sin(4x) \tag{10}$$

for the first and second order. By the reason of the slower execution time of interval arithmetic compared to floating point arithmetic we only computed the derivatives for 10000 different values. Table 2 as well as the Figures 6 and 7 show the measured run time (the mean of 10 run's) for the automatic differentiation of the first and second derivative for expressions (9) and (10). Due to the fact that C-XSC has implemented the automatic differentiation directly into their operations and functions [13], the speedup is much smaller compared to the speed up measured in `double` precision.

	$x^3 + x - 1$		$e^x \sin(4x)$	
	$\frac{\mathrm{d}}{\mathrm{d}x}$	$\frac{\mathrm{d}^2}{\mathrm{d}x^2}$	$\frac{\mathrm{d}}{\mathrm{d}x}$	$\frac{\mathrm{d}^2}{\mathrm{d}x^2}$
ET filib++	5.99 ms	15.86 ms	6.78 ms	21.61 ms
C-XSC	9.62 ms	23.25 ms	10.37 ms	23.39 ms

Table 2: Performance comparison of the evaluation of interval expressions using automatic differentiation to compute the derivatives up to the first or second order.

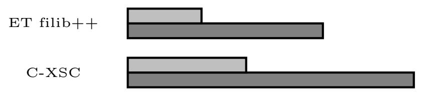


Figure 6: Performance comparison of the evaluation of the interval expression (9) using automatic differentiation to compute the derivatives of the first (light gray) and second (gray) order.

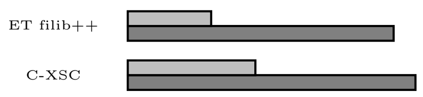


Figure 7: Performance comparison of the evaluation of the interval expression (10) using automatic differentiation to compute the derivatives of the first (light gray) and second (gray) order.

The last test case was a comparison of a root finding method working on intervals. In [13], automatic differentiation with intervals is used for the `Halley's` method to approximate a zero of a non linear function. The `Halley's` method is an iterative method approximating the zero $x^{(k)}$ with $k = 0, 1, \ldots$ as follows, see [13] for details:

$$a^{(k)} := -\frac{f(x^{(k)})}{f'(x^{(k)})} \tag{11}$$

$$b^{(k)} := a^{(k)} \cdot \frac{f''(x^{(k)})}{f'(x^{(k)})} \tag{12}$$

$$x^{(k+1)} := x^{(k)} + \frac{a^{(k)}}{1 + \frac{b^{(k)}}{2}} \tag{13}$$

In our measurement we have used the `Halley's` method to find a zero for each of the expressions (9) and (10) using C-CXSC and our approach. The start interval $x^{(0)}$ was for both cases defined as $[1.25, 1.25]$. Table 3 as well as the Figure 8 shows the measured run time (the mean of 10 run's).

	$x^3 + x - 1$	$e^x \sin(4x)$
ET filib++	1.01 ms	2.33 ms
C-XSC	1.91 ms	2.84 ms

Table 3: Performance comparison of the `Halley's` method using interval arithmetic for the approximation of a root of a expression.

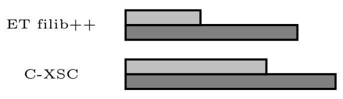


Figure 8: Performance comparison of the `Halley's` method using interval arithmetic for the approximation of a root of the expressions (9) (light gray) and (10) (gray).

6 Related Work

In [10], Gil and Gutterman used expression templates and template meta programming to perform symbolic differentiation at compile time whereas Aubert, Césaré and Pironneau used expression templates to reduce the number of loops, temporaries and copies while computing the first order partial derivatives of an expression [2].

7 Conclusion and Future Work

In this paper we have shown how generative programming using expression templates and template meta programming can be used to implement a domain specific language for partial automatic differentiation of an arbitrary order which is flexible, fast and easy to use.

A comparison of our approach against common automatic differentiation libraries showed a dominating performance for floating point and interval types. Additionally, the flexibility of our approach like the use of different data types for the differentiation or the code generation for the differentiation of any arbitrary order is a surplus.

Further investigations are planned to improve the automatic differentiation with generative programming. An analysis of the adaptability of the reverse mode of the automatic differentiation as well as the extension for the computation of results like the *Hessian* matrix are planned.

References

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] P. Aubert, N. Di Césaré, and O. Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3:197–208, 2001.

[3] P. Becker. Working Draft, Standard for Programming Language C++. Technical Report N3242=11-0012, ISO/IEC JTC1/SC22/WG21, February 2011.

[4] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, aug 1996.

[5] C. H. Bischof and H. M. Bücker. Computing derivatives of computer programs. In J. Grotendorst, editor, *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition*, volume 3 of *NIC Series*, pages 315–327. NIC-Directors, Jülich, 2000.

[6] C. H. Bischof, P. Hovland, and B. Norris. On the implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation*, 21(3):311–331, 2008.

[7] C. H. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[8] G. F. Corliss and A. Griewank. Operator Overloading as an Enabling Technology for Automatic Differentiation. Technical Report CRPC-TR93431, Center for Research on Parallel Computation, Rice University, Houston, TX, USA, May 1993.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[10] J. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, COOTS'98, pages 18–18, Berkeley, CA, USA, 1998. USENIX Association.

[11] D. Gregor, J. Järvi, and G. Powell. Variadic templates (revision 3). Technical Report N2080=06-0150, ISO/IEC JTC1/SC22/WG21, Oct. 2006.

[12] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[13] R. Hammer, D. Ratz, U. Kulisch, and M. Hocks. *C++ Toolbox for Verified Computing I: Basic Numerical Problems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[14] J. Härdtlein. *Moderne Expression Templates Programmierung*. PhD thesis, Universität Erlangen-Nürnberg, 2007. in German.

[15] L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis, 2004.

[16] M. Lerch and J. Wolff von Gudenberg. Expression templates for dot product expressions. *Reliable Computing*, 5(1):69–80, 1999.

[17] S. B. Lippman, editor. *C++ Gems*. SIGS Publications, Inc., New York, NY, USA, 1996.

[18] N. Myers. A new and useful template technique: "Traits". *C++ Report*, 7(5):32–35, June 1995. Reprinted in [17].

[19] M. Nehmeier. Interval arithmetic using expression templates, template meta programming and the upcoming C++ standard. *Computing*, pages 1–14, 2011. 10.1007/s00607-011-0176-6.

[20] M. Nehmeier and J. Wolff von Gudenberg. filib++, Expression Templates and the Coming Interval Standard. *Reliable Computing*, 15(4):312–320, July 2011.

[21] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, oct 1992.

[22] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.

[23] E. Unruh. Prime number computation. Technical Report N0462=94-0075, ISO JTC1/SC22/WG21 – C++ working group, 1994.

[24] D. Vandevoorde and N. M. Josuttis. *C++ Templates – the Complete Guide*. Addison-Wesley, 2003.

[25] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in [17].

[26] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in [17].

[27] T. Veldhuizen. Techniques for scientific C++. Technical Report 542, Indiana University Computer Science, August 2000. version 0.4.

[28] T. Veldhuizen. C++ templates are turing complete. Technical report, 2003.

[29] A. Verma. An introduction to automatic differentiation. *Current Sci.*, 78(7):804–807, 2000.