A Generic and Self-Optimizing Polynomial Library

Rüdiger Sonderfeld[*]
TU Wien,
Gußhausstraße 27-29, Vienna, Austria

René Heinzl[†]
Institute for Microelectronics, TU Wien,
Gußhausstraße 27-29, Vienna, Austria

ABSTRACT

Application of polynomial expressions is an important concept in scientific computing. The here presented implementation of such a polynomial library is designed to be generic and to be easily adoptable to new problems while providing the necessary performance. Generic programming and self optimization techniques are discussed. Examples from different areas of scientific computing conclude the presentation.

Categories and Subject Descriptors

I.1 [**Computing Methodologies**]: SYMBOLIC AND ALGEBRAIC MANIPULATION; D.2.11 [**Software Engineering**]: Architectures—*Data abstraction, domain-specific architectures*

Keywords

Polynomial, finite elements, generic programming, meta-programming, high performance

1. INTRODUCTION

Polynomials are an important and efficient tool for numerous fields of science although a generic implementation with respect to high performance is still missing. Due to the simple rules regarding differentiation and integration polynomials have found wide spread application. GSSE's [1, 2] algebraic topology library [3] introduces a common property environment protocol to specify data structures for scientific computing. The polynomial library presented here is built on top of this library providing particular extensions to describe and manipulate polynomials. This generic approach not only provides enough flexibility, it also has no run time speed costs. As run time performance is highly optimized by using meta-programming to support the compiler to produce optimal code. The language of choice is C++ due to several reasons: C++ is a well established language in the scientific computing community and is supported with mature tools on all major systems. Furthermore, C++ is also a very flexible language supporting a vast amount of paradigms and programming techniques including generic and meta-programming.

[*]Contact information: ruediger@c-plusplus.de

[†]Contact information: heinzl@iue.tuwien.ac.at

2. POLYNOMIALS

Polynomials can be defined as a weighted sum of exponential terms in at least one variable or expression, with the exponents being restricted to non-negative whole numbers. Their simple definition as well as the fact that their algebraic structure is not only closed under addition, subtraction, and multiplication, but also under differentiation and integration, result in their widespread application. The demand of additional properties such as, e.g., orthogonality with respect to an inner product results in special classes of polynomials, orthogonal polynomials, which further increases their appeal in fields such as finite elements as demonstrated in Section 6.2. A polynomial consists of coefficients (a_i) and a variable expression (x^i):

$$a_0\ x^0 + a_1\ x^1 + a_2\ x^2 + \ldots + a_n\ x^n$$

Thus a container representation to store the coefficients for polynomials was chosen so that a generic C++ variable contains the expression:

```
gsse::polynomial<Expression, Coefficients>
```

Usually the expression is represented as a Boost.Phoenix Actor [4]. But it can also be represented in different ways which allows a highly flexible deployment of the polynomial library and even mathematical terms which are not considered to be polynomials are representable using this concept (e.g. Fourier series).

When storing the coefficients in a container great care has been taken to implement the library to be generic with respect to the type of the underlying data structure. In this way it is possible to use compile time containers if the size or even the concrete coefficients are already known at compile time. This allows the compiler to inline and execute operations at compile time.

3. RUN TIME AND COMPILE TIME PROGRAMMING

C++ has a classic build system. Code is compiled by the compiler into machine language and after the compilation (and linking) the executable is run by the machine. The compilation process is referred to here as compile time, while the process of executing the code on the machine is referred to as run time. Using the preprocessor and templates it is possible to get the compiler to execute code at compile time, which is usually referred to as meta-programming.

3.1 Compile Time Container

A compile time container is a container whose size is known at compile time. Associative compile time containers use a key which must be distinguishable at compile time. A run time container is a container which is not a compile time container. C++ provides a wide arrange of run time containers derived from SGI's STL [5] and compile time containers from the Boost library collection [6]. An example for a run time container is `std::vector` which does not have a fixed size and contains homogeneous elements[1].

```
std::vector<int> v;
v.resize(10); // run time changeable size
```

An example for a compile time container is `std::tr1::tuple` (or `boost::tuple`). The elements are specified at compile time and thus is the size. A tuple can contain elements of different types and is thus an inhomogeneous compile time container:

```
boost::tuple<int, double, string> t;
get<0>(t); // is an integer
get<1>(t); // is a double
```

The `boost::fusion::map` is an example of an associative compile time container. The association is known at compile time and thus the size and the mapping of the elements. The tuple example can be translated into a map:

```
fusion::map<
  fusion::pair<mpl::int_<0>, int>,
  fusion::pair<mpl::int_<1>, double>,
  fusion::pair<mpl::int_<2>, string>,
> m;
```

3.2 Meta-Functions

Meta-functions are functions evaluated at compile time. In C++ terms a meta-function is a template-class. The function parameters are passed as template-parameters and the result is a `member-typedef` called `type`.

```
template<typename Arg0, ..., typename ArgN>
struct a_metafunction
{
  typedef ... type;
};
```

The main use of meta-functions in the polynomial library is to determine the result type of a run time function. Each function has a meta-function with the same name and template parameters in the `result_of` sub-namespace.

4. LAZY EVALUATION AND VIEWS

Operations in the polynomial library are "lazy", which means the expressions are only evaluated when the result is required. This "laziness" is achieved by using views. An operation does not change the polynomial, it just provides a new view to the polynomial. A view is a wrapper around the polynomial or container class which allows the interception of operations directed at the polynomial. The calculations can therefore be made precisely when the values are accessed.

Polynomial addition is a good example for the view concept. Two polynomials with the same variable are added by summing the coefficients of same degree

$$\begin{aligned} p &= a_0\, x^0 + a_1\, x^1 + \ldots + a_n\, x^n \\ q &= b_0\, x^0 + b_1\, x^1 + \ldots + b_n\, x^n \\ p + q &= (a_0 + b_0)\, x^0 + (a_1 + b_1)\, x^1 + \ldots + (a_n + b_n)\, x^n \end{aligned}$$

The addition of two polynomials can therefore be accomplished by providing a view which takes a reference to the coefficient containers of the input polynomials and adding the two corresponding coefficients when a coefficient is accessed. A simplified code example:

```
template<typename LhsCoef, typename RhsCoef>
struct add_view {
  LhsCoef const &lhs;
  RhsCoef const &rhs;

  type operator[](index_type n)
  {
    return lhs[n] + rhs[n];
  }
};
```

Lazy evaluation reduces the problem of unnecessary computations but brings its own set of problems. In the view presented in the listing the sum is always computed when the coefficient is accessed. If the coefficient is accessed often, it can therefore reduce performance. Adding a transparent cache to the view to save already calculated values adds its own share of management overhead and thus can even cost more than doing a large amount of additions. Another issue with lazy evaluation is that it gets hard to do run time predictions since a slight change in the input set might cause the code to access more coefficients and thus suddenly increase the amount of computations done. A solution to this problem is to copy the view into a new container. This will behave exactly as if performing the calculations in a non "lazy" fashion. Due to these problems lazy evaluation is used for compile time containers only.

5. COMPILE TIME / RUN TIME TRANSPARENCY

The most suitable container to use for the coefficients usually depends on the input and not the algorithms. It is therefore important to provide a basic set of programming utilities which are generic with regard to the used container type. Compile time and run time containers have a few incompatible requirements which make it hard to define a common set of utilities.

5.1 Accessing Coefficients

Accessing a polynomial's coefficients is an important operation. There exist two basic ways of accessing the coefficient. Compile time accessors are used when the index of the coefficient to be accessed is known at compile time, while run time accessors have to be used otherwise. The compile time version takes the index as a template-parameter, while the run time entity as a function argument.

```
namespace compiletime {
 template<class Polynomial, index_type n>
 typename result_of::coeff<Polynomial, n>::type
 coeff(Polynomial const &p);
}

namespace runtime {
 template<class Polynomial>
 typename result_of::coeff<Polynomial>::type
 coeff(index_type n, Polynomial const &p);
}
```

[1]Due to space constraints, the presented code examples lack unnecessary namespaces and headers.

Access to the coefficient is then available by:

```
polynomial<X, Coef> p;

compiletime::coeff<0>(p);
runtime::coeff(n, p);
```

Thus it is possible for the compiler to simplify the code and determine more information about the coefficient. Therefore the compile time version is more flexible than the run time version. Using inhomogeneous compile time containers in conjunction with the run time accessor is not possible since it is not possible to determine the return type in advance. This reduces the flexibility of the code using the run time accessors. A workaround to this problem can be achieved by using the visitor pattern.

```
template<class Polynomial, typename Visitor>
void coeff_visitor(
  index_type n,
  Polynomial const &p,
  Visitor v
);
```

However, this approach has the disadvantage of being more complicated to use than the `coeff` function.

The coefficient accessors are not simple wrappers around the accessors of the underlying container. They check the access and return a zero value if the container does not contain the coefficient. The zero value is determined by the `coeff_trait` template class:

```
template<class CoeffType, class Polynomial>
struct coeff_trait
{
  typedef CoeffType zero_type;

  static zero_type const
    zero_value = zero_type();
};
```

By using partial template-specialization it is possible to define the corresponding zero value for the correct type. For inhomogeneous polynomials `default_coeff` is passed as `CoeffType` and the default behavior is to return an `int`.

5.2 Setting Coefficients

Coefficients may set using the `set_coeff` function. It does not change the given polynomial but creates a new view instead. This provides the polynomial library with a functional programming style. Setting the coefficients and changing the polynomial can only be achieved by directly manipulating the coefficient container.

```
namespace compiletime
{
 template<index_type n,
          class Polynomial,
          class Coeff>
 typename result_of::set_coeff<n,
                              Polynomial,
                              Coeff>::type
 set_coeff(Polynomial const &p,
           Coeff const &c);
}

namespace runtime
{
 template<class Polynomial,
          class Coeff>
 typename result_of::set_coeff<Polynomial,
                              Coeff>::type
 set_coeff(index_type n,
           Polynomial const &p,
           Coeff const &c);
}
```

Write access is then available by:

```
polynomial<X, Coef> p;

compiletime::set_coeff<0>(p, 1);
runtime::set_coeff(n, p, 1);
```

5.3 Degree

The degree of the polynomial is defined as the maximum degree of all of its terms, where the degree of a term is given as the sum of the degree of all variables in this term. The polynomial library defines the degree as the index of the highest non-zero coefficient. To obtain the correct degree requires to use a polynomial for each variable and finally combine them:

$$\mathrm{degree}\left(3\,x^4\,y^2\right) = 4 + 2 = 6$$

```
struct X;
typedef polynomial<
 X,
 fusion::map< pair< mpl::int_<4>, double> >
> inner_poly;

typedef polynomial<
 Y,
 fusion::map< pair< mpl::int_<2>, inner_poly> >
> the_polynomial;
```

By instantiating the polynomial the calculation of its degree is possible:

```
the_polynomial p;
assert( degree(p) == 6 );
```

6. EXAMPLES

To illustrate the flexibility of the polynomial library, not only a compile time example is given, but also examples for a finite element application and arbitrary precision arithmetic.

6.1 Compile Time Programming

The application of meta-programming is presented which utilizes the compiler to execute code at compile time and then reduce the result of the expressions. As an example the derivative of a second-degree polynomial is calculated and a second polynomial is added:

$$\frac{\mathrm{d}(3 + 4.5\,x + 10\,x^2)}{\mathrm{d}x} + (1 + 2x) = 5.5 + 22\,x$$

The type list represents the type of each coefficient starting from the zero to the second degree coefficient.

```
struct X { } x;
typedef fusion::vector<double,
                       double,
                       int> coeffs;

typedef polynomial<X, coeffs> poly;
poly p(x, coeffs(3.0, 4.5, 10));

typedef result_of::diff<poly, X>::type diffed;
diffed d = diff(p, x);

poly q(x, coeffs(1.0, 2.0, 0));

std::cout << coeff<1>(q + d);
```

By compiling and evaluating the assembler code it is revealed that the calculations were performed at compile time and the binary only contains the final result of 22.

6.2 Finite Element Integration

In the theory of finite elements [7, 8], a continuous function space is projected onto a finite function space P^k, where the space P^k is the space of polynomials up to the total order of k.

For many special cases, finite element integrals can be computed manually and added into the source code of an application. This results in excellent run time performance but lacks flexibility. For more general cases, e.g., general coefficients, they must be computed by numerical integration at run-time. To prevent an ill-conditioned system matrix, orthogonal polynomials have to be chosen as numerical integration weights. One possible type of polynomial is a normalized Legendre polynomial [9]. Coefficients for such a polynomial P_k of order k can be efficiently evaluated by using the recursion procedure:

$$\begin{aligned} &P_0(x) = 1 \\ &P_1(x) = x \\ &P_k(x) = \frac{2j-1}{j}\, x\, P_{k-1}(x) - \frac{j-1}{j} P_{k-2}(x) \quad k \geq 2 \end{aligned} \tag{1}$$

To use arbitrary p-finite elements (polynomial order [10, 11]) the numerical coefficients have to be calculated either manually and inserted into the source code or determined numerically at run time.

The polynomial library presented here is then used to store manually pre-calculated integration tables at compile time (order 1-5). If the user requires higher order finite elements, numerical coefficients are calculated at run time to any order.

6.3 Arbitrary-Precision Arithmetic

The application of the polynomial library to perform arbitrary-precision arithmetic (or “bignum arithmetic”) is also presented here. It uses the fact that a number is in essence a polynomial with a fixed base.

$$1372 = 1 \cdot 10^3 + 3 \cdot 10^2 + 7 \cdot 10 + 2$$

This can easily be translated into C++ code by using the polynomial library. Note, that the first element in the array is the zero coefficient:

```
typedef unsigned char byte_t;
typedef array<byte_t, 4> coeffs_t;

coeffs_t coeffs = {{2, 7, 3, 1}};

gsse::polynomial<mpl::int_<10>,
                 array<byte_t, 4> > p(coeffs);
```

Since computer systems usually operate on binary numbers base-2 is the optimal choice. The difference between polynomial arithmetic and arbitrary-precision arithmetic is that the coefficients need to be realigned to the base after each operation.

7. OUTLOOK

A major problem with the polynomial library due to templates are hard to read compiler error messages. With the upcoming revision of the C++ Standard [12] (referred to as “C++0x”) concepts will be introduced to the language as a form of type system for template parameters. This will allow the library to define specifications about the template parameters and thus will improve the readability of compiler output. At the moment every operation provides a meta-function in the `result_of` sub-namespace to determine the result type of the operation. A large part of the source code is dedicated to these meta-functions. The introduction of `decltype` and `auto` in C++0x will allow the elimination of these meta-functions leading to a great reduction of the code size.

8. CONCLUSION

To fully harvest a polynomial library compile time and run time access and evaluation have been implemented. Special consideration is placed on the use of performance optimizing techniques such as lazy evaluation and compile-time evaluation. By developing the compile time (lazy) version first, the run time (non-lazy) layer can be integrated easily to provide not only high performance but also great flexibility.

9. ACKNOWLEDGMENTS

We want to thank Philipp Schwaha for his support and Prof. Siegfried Selberherr for the resources at the Insitute of Microelectronics, TU Wien. This work has been supported by the Austrian Science Fund FWF, project P19532-N13.

10. REFERENCES

[1] R. Heinzl, P. Schwaha, and S. Selberherr. A High Performance Generic Scientific Simulation Environment. In B. Kaagström et al., editor, *Lecture Notes in Computer Science*, volume 4699/2007, pages 996–1005. Springer, Berlin, June 2007.

[2] R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr. Parallel Library-Centric Application Design by a Generic Scientific Simulation Environment. In *Proc. of the POOSC*, Paphos, Cyprus, July 2008.

[3] R. Heinzl. Data Structure Properties for Scientific Computing. In *Proc. of the POOSC*, Genova, Italy, July 2009.

[4] Boost. *Boost Phoenix 2*, 2006. http://spirit.sourceforge.net/.

[5] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[6] Boost. *Boost C++ Libraries.* http://www.boost.org.

[7] C. Johnson. *Numerical Solutions of Partial Differential Equations by the Finite Element Method.* Cambridge University Press, Cambridge, UK, 1987.

[8] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method.* McGraw-Hill, Berkshire, England, 1987.

[9] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover, New York, 1964.

[10] I. Babuscaronka, M. Griebel, and J. Pitkäranta. The Problem of Selecting the Shape Functions for a p-Type Finite Element. *Int. Journal for Numerical Methods in Engineering*, 28(8):1891–1908, 1989.

[11] V. Korneev, J. E. Flaherty, J. T. Oden, and J. Fish. Additive Schwarz Algorithms for Solving hp-Version Finite Element Systems on Triangular Meshes. *Appl. Numer. Math.*, 43(4):399–421, 2002.

[12] Open Standards. *C++0x Standard.*