FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

An Expression Template aware Lambda Function

*Jörg Striegnitz, Stephen A. Smith**

FZJ-ZAM-IB-2001-01

Dezember 2000

(letzte Änderung: 23.01.2001)

(*) Advanced Computing Laboratory Los Alamos National Laboratory New Mexico, USA

Preprint: Proceedings of the 2000 Workshop on C++ Template Programming, 10.10.2000, Erfurt Germany

An Expression Template aware Lambda Function

Jörg Striegnitz

Research Centre Jülich
Central Institute for Applied Mathematics
Germany
J.Striegnitz@fz-juelich.de

Stephen A. Smith

Advanced Computing Laboratory
Los Alamos National Laboratory
New Mexico, USA
sa_smith@acl.lanl.gov

Abstract. Template libraries such as the STL contain several generic algorithms that expect functions as arguments and thereby cause a frequent use of function objects. User-defined function objects are awkward because they must be declared as a class in namespace scope before they may be used. In this paper, we describe a *lambda function* for C++, which allows users to define function objects on the fly, without writing class declarations. We show that, by using expression templates, the lambda function can be implemented without hurting the runtime performance of a program. Expression templates can also help to overcome the performance penalties that may arise when using expressions over user-defined types. Thus, we based our approach on PETE which is a framework that simplifies the addition of expression template functionality to user-defined classes.

1. Introduction

The Standard Template Library (STL) [C++] contains many function objects that mimic *Higher Order Functions* (*HOF*s). These are functions that take function arguments and/or return functions (e.g. `for_each`, `transform`, or `find_if`). Operations passed to HOFs are often very short in code and primarily used in a local context. Nevertheless, they have to be defined in namespace scope, possibly yielding numerous small functions or function objects respectively. The *point of use* and *the point of definition* may get more and more dispersed, making code harder to read and understand.

This problem even becomes worse, as it is impossible to pass function templates to STL's HOFs. In order to mimic rank-2 polymorphism (passing polymorphic function arguments to polymorphic functions), either function overloading or the definition of a class with an `operator()` template is required (like e.g. in [FC++] [SM00]). Especially the first approach will increase namespace pollution, while the latter also depends on a class representative, thus, the existence of an object, which has to be created manually.

A better solution would be to define functions on the fly. This feature is common in functional programming languages, which offer a special syntax called `lambda` to define and use functions in one go.

Our C++ framework **FACT!** (Functional Additions to C++ through Templates and Classes [FACT]) offers a similar functionality through a function called `lambda`, which could be used to create function objects on the fly and thereby helps to keep the point of use and the point of definition close together. As with its pure functional counterpart, functions obtained by lambda are

free of side effects and therefore may be used in parallel environments as well.

In this article we discuss the implementation of our lambda functions and show how to add expression template functionality to user-defined classes. After giving a short introduction into the lambda functions, we will show how to build lambda expressions by using the Portable Expression Template Engine (PETE). We will then concentrate on how evaluation is done and conclude with a discussion of performance and possible future work.

2. The Lambda Function

The lambda function takes a list of variables (called the *lambda list*), an expression that may contain any of this list's variables (called the *lambda expression*) and returns a function which usually has the same number of arguments as there are elements in the lambda list. Consider the following example:

```
lambda(x,y, x + y)
```

`x` and `y` form the *lambda list*, `x + y` is the *lambda expression*. Since the lambda list has two members, a binary function is returned.

Applying a function returned by `lambda` to some arguments is done as follows: first, arguments passed to the function get associated with the variables of the lambda list - this is done from left to right. Second, all occurrences of lambda variables in the lambda expression get substituted by their associated values. Finally, the expression gets evaluated and the result is returned. For instance, applying `lambda(x,y, x + y)` to `3` and `4` results in:

1. `x` is bound to 3 and `y` is bound to `4`
2. substitution yields `3 + 4`
3. evaluation leads to `7`

Thus, `lambda(x,y, x + y)` represents a function that calculates the sum of its arguments.

Functions returned by `lambda` are polymorphic, thus, `x` and `y` may be bound to values of type `int`, `float`, `complex`, `string`, or any other type that is compatible with `operator+`. As long as an appropriate `operator+` exists, `x` and `y` even may be bound to values of different type.

Lambda expressions may contain calls to other functions, e.g.:

```
lambda(a,b,c, sqrt( sqr(a) + sqr(c) + sqr(b) ) )
lambda(a,b, sin(a) / cos(b) )
```

Additionally, lambda variables may be bound to functions and lambda may return a function that returns a function as well:

```
lambda(f,x,y, f(x,y) )       // f is a placeholder for a function
lambda(x, lambda(y, x + y) )
```

Moreover, functions returned by lambda are presented in a curried form, which makes them capable of taking their arguments one at a time and thereby offers the opportunity of partial application.

```
lambda(x, pow(x) ) // partially applying pow - return unary function that
                   // returns unary function
```

At least four things are needed to develop the lambda function:

- functions of varying signature (e.g. `lambda(x1, expression)`, `lambda(x1,x2 expression)`, etc.),
- mechanisms to build and store a lambda list,
- mechanisms to store and manipulate the expression; along with
- methods to do the evaluation

Multiple variants of lambda functions are needed, each one taking a different number of lambda variables - this can be solved through function overloading. Building and storing the lambda list can be avoided. Provided that we can rediscover the ordering information of the lambda list, it is sufficient to store lambda variables directly in the expression. Thus, the most important thing that remains is to build, store, manipulate, and evaluate expression trees.

With respect to performance, expression templates [Vh95] are a way handle lambda expressions. Expression templates are nested template structures, used to represent the parse tree of an expression. They are built during compile time through overloaded arithmetic operators, which - instead of immediately applying an operation - return objects that incrementally build up the parse tree. The parse tree is represented in two fashions: as a type tree (the *expression template* tree) and as a tree of objects (the *expression object* - which indeed is an instance of the expression template tree). Template meta programs [Vh95-2] [EC00] allow one to traverse such expression template trees during compile time and in conjunction with inlining techniques the expression object can be used to produce efficient code.

Using the expression template technique, lambda variables become part of the the expression template tree. Since the expression template tree emphasizes types, different lambda variables need to be of different type, thereby enabling template meta programs to do the substitution during compile time. In order to support functions of arbitrary dimension, an unlimited number of types to represent lambda variables is needed:

```
template <int n>
struct ARG {};
```

`ARG` is a suitable representation, because it can be used to form `numeric_limits<int >::max()` different types, which we assume to be an acceptable limit. For convenience reasons, **FACT!** offers a large number of predefined lambda variables, all of them are defined in the scope of `namespace LAMBDA`. Thus, the user usually does not need to pay attention to the *real type* of a lambda variable, but just writes something like `using LAMBDA::x` to make the lambda variable `x` visible in the current scope.

In the next section we will show how to form expression templates out of expressions containing instances of `ARG` by using PETE.

3. Building Lambda Expressions with PETE

3.1. How PETE works

The Portable Expression Template Engine (PETE) [Ha99 , PETE] provides tools to simplify the addition of expression template functionality to a set of classes. PETE uses external polymorphism [CL98] , so expression templates may be implemented for existing classes, such as the Standard Template Library vector class. The PETE library is fairly lightweight, containing fewer than 3000

lines of code. As the example in this section illustrates, integration of PETE with a user-defined class requires a very small amount of code, typically provided through specializations of some PETE classes. PETE is used to implement expression objects in **FACT!**, but users of **FACT!** do not require any knowledge of PETE.

PETE supports 45 built-in operators to build expression objects out of expressions. Besides all C++ mathematical operators and a collection of common mathematical functions like `sin()`, it also provides a `where(a,b,c)` function since the conditional expression `a ? b : c` cannot be overloaded.

To integrate user-defined classes, variants of these operators have to be created, each one being capable to act on any combination of user-defined classes and PETE-specific classes. Fortunately, this has not to be done by the user, but PETE provides a tool (written in C++) called `MakeOperators` that reads a file with a simple description of the user's class and generates header files containing the hundreds of operator functions that are necessary. Once these operators are available, only three tasks are left to implement expression template functionality for the users classes:

- define how the objects are stored in the expression tree
- add assignment operators that take PETE expressions
- define how data is accessed during evaluation

To illustrate how PETE works, we will consider the following class:

```
class Vec3 {
  Vec3(double i=0.0) { d[0]=i; d[1]=i; d[2]=i; }
  Vec3(double a,double b,double c) { d[0]=a; d[1]=b; d[2]=c; }
  double &operator[](int i) { return d[i]; }
  double operator[](int i) const { return d[i]; }
private:
  double d[3];
};
```

PETE's operators need to know what to stick in the leaves of the expression tree. To offer this information, the user has to supply a specialization of the `CreateLeaf` struct:

```
template <>
struct CreateLeaf< Vec3 > {
  typedef Reference<Vec3> Leaf_t;
  static inline Leaf_t apply(const Vec3& a) {
    return Leaf_t(a);
  }
};
```

The typedef `Leaf_t` is the type of the object stored in the expression template tree. To save space and avoid unnecessary calls to copy constructors PETE provides a `Reference` object that stores a reference to the original object in the expression tree rather than a copy. Besides defining the type of the leaf, the specialization of `CreateLeaf` also provides an `apply` method that builds the object in the expression tree (in this case `Reference<Vec3>`) from the object in the expression (in this case `Vec3`). When there is no specialization of `CreateLeaf`, PETE wraps the object in the template class `Scalar`.

In PETE an expression object has type `Expression<T>`. To traverse the expression tree, PETE offers the function `forEach`, which has the following general form:

```
forEach(Expression, LeafTag, CombineTag);
```

This function traverses the nodes of the `Expression` object, applies an operation selected by `LeafTag` at the leaves, and combines the results from non-leaf nodes' children according to `CombineTag`. This is implemented by a meta program so the tree traversal is done at compile time. The return value of the `forEach` function is provided by the class template `ForEach`, so that the type produced can be used as input to other template meta programs.

There are two default combinator tags in PETE: `OpCombine` and `TreeCombine`. `OpCombine` combines results from the leaf nodes according to the operators stored at the non-leaf nodes, so that `forEach` returns a value computed for the expression. `TreeCombine` is used to combine the results from the leaf nodes back into an expression object, so that forEach returns a transformed version of the expression.

For user-defined classes, evaluation can take many forms. Some typical examples are calls to `operator[]` as in `a[i]`, or `operator()` as in `a(i,j)`, but evaluation could require calls to arbitary functions. To tell PETE how to perform a given form of evaluation, users specialize a class called `LeafFunctor`, which is templated on the user-defined class and a functor tag. One of the predefined functor tags is the class `EvalLeaf1`, which stores a single integer index, accessible through the method `val1()`. Such a functor tag primary serves as a selector while the *real* application is done by a specialization of `LeafFunctor`:

```
template <>
struct LeafFunctor<Vec3, EvalLeaf1> {
  typedef int Type_t;
  static inline Type_t apply(const Vec3& a,const EvalLeaf1& f) {
    return a[f.val1()];
  }
};
```

By defining the evaluation through specialization of an external functor, PETE is not restricted to evaluating classes that support a specific interface (such as `operator[]` in this case). Users with classes that require different evaluation mechanisms do not need to rewrite the entire expression template machinery, but just need to provide this one class specialization. In this example, the `LeafFunctor` acts on leafs of type `Vec3` and performs the operation selected by `EvalLeaf1`. It provides the function `apply` which takes a leaf (of type `Vec3`) as well as an instance of the functor tag and returns the component of the vector that is identified by the index that is stored in the functor tag.

Componentwise evaluation of vector expressions is now possible by applying `forEach` to an expression object. With PETE, this usually is done within the assignment operator of the user's class:

```
template <typename E>
Vec3 operator=(const Expression<E>& expression) {
  d[0] = forEach( expression, EvalLeaf1(0), OpCombine() );
  d[1] = forEach( expression, EvalLeaf1(1), OpCombine() );
  d[2] = forEach( expression, EvalLeaf1(2), OpCombine() );
}
```

It also makes sense to supply a constructor from an `Expression` object which offers the same functionality. To avoid implicit conversions it should be declared `explicit`.

Evaluating expressions with PETE's forEach function allows for more generic operations than simply computing the value of an expression. For example, in expressions involving arrays, one

could pull out domain information from the arrays and check for conformance. By selecting different leaf functors and combiners, very general transformations can be performed on expressions. This general capability will be used to perform substitutions in lambda expressions.

3.2. The Lambda Function

Using PETE, building lambda expressions is quite simple, since PETE's `MakeOperator` tool automatically produces code for all operators that are necessary to build expression objects out of expressions that contain instances of `ARG<i>` (we call such expression objects *generic expression objects*). To tell PETE how to handle values of type `ARG<i>`, several specializations of the `CreateLeaf` structure are needed (one for each type of lambda variable).

The lambda function has to take some lambda variables as well as an expression object and return a polymorphic function implementing the generic expression. Using C++ such a polymorphic function can be implemented by a function object whose function call operator (`operator()`) is a template. The number of arguments this operator has to take depends on the number of lambda variables that have been passed to the lambda function. Thus, for every dimension a function returned by lambda may have, a special class is needed. For binary functions it has the following form:

```
template <typename E>
struct lFUNC2 {
  lFUNC2(const E& e) : e_m(e) {}
  lFUNC2(const lFUNC2& rhs) : e_m(rhs.e_m) {}
  const E& expression() const {
    return e_m;
  }
  template <typename A1,typename A2>
  result_t operator()(A1 a1,A2 a2) const {
  ...
  }
private:
  E e_m;
};
```

This class stores a generic expression object of type `E` and provides a template for a binary function call operator. How to determine the return type `result_t` will be discussed in a later section.

The lambda function just has to create an appropriate instance of such a class. Here are examples for lambda functions to produce binary / ternary functions:

```
template <int m,int n,typename E>
lFUNC2<E> lambda(const ARG<m>& a,const ARG<n>& b,const E& e) {
  return lFUNC2<E>( e );
}

template <int m,int n,int o,typename E>
lFUNC3<E> lambda(const ARG<m>& a,const ARG<n>& b,const ARG<o>& c,const E& e) {
  return lFUNC3<E>( e );
}
```

Notice, that the `ARG` arguments are ignored and only used to select a specific variant of lambda. The indices of the lambda variables (namely `m`, `n` and `o`) may be of arbitrary value. They do not necessarily need to reflect the order in which they have been passed to the lambda function. As mentioned earlier, this ordering information is essential in order to do substitution. Therefore, the indices of all lambda variables inside the expression object get *normalized* to represent the correct

ordering (not shown in the above code). This normalization is a compile time process, handled by some sophisticated template meta programs which are quite similar to those being used during substitution (see section 4.1.).

To support functions of arbitrary dimensions an endless number of specializations of the `CreateLeaf` class as well as an endless amount of `lFUNCX` classes and overloaded `lambda` functions will be needed. To cover as many situations as possible and to keep the user away from the underlying details, we developed a code generating tool that is supplied with the largest function dimension to support, and produces a C++ header file that contains all the necessary definitions. Including support for currying of C++ functions, function composition and a few other features, this header file consists of approximately 4000 lines of code, if functions up to order five are supported.

4. Applying the Result of a Lambda Function

4.1. Substitution

An expression is represented in two different fashions: as an expression template tree (emphasizing types) and as an expression object (emphasizing values). Substitution has to be done for both and thus, for a lambda expression that contains *N* lambda variables, *N* type/value tuples are needed for substitution. These tuples are given by the parameters of `lFUNC`'s parenthesis operator and due to normalization, association of variables in the tuple with the corresponding `ARG<i>` values of the expression object is clear. For example to evaluate `lambda(x,y, x + y)( f,c )`, we need to substitute two arguments `f` and `c` of arbitrary types for `ARG<1>` and `ARG<2>` in the expression object.

Now, substitution simply can be done by template meta programs, but for every argument we intend to substitute, the full expression tree needs to be traversed. To save compilation time, it is reasonable to store all type/value tuples in an array, use the integer index that is carried by lambda variables as an index into it and traverse the expression tree just once. Such an array has to be accessible during compile- and runtime. Compile time mechanisms are based on types and thus, for every dimension an array may have, a different type is needed. Fortunately, the greatest possible dimension of the array is known, because the user has passed it to the generator tool. Using a type `mNIL` to indicate that a specific position of an array is not in use, a single structure is sufficient to implement the array:

```
template <typename A1=mNIL, ..., AN=mNIL>
struct SIGNATURE {
  typedef A1 ARG1_t;
  ...
  typedef AN ARGN_t;

  SIGNATURE() {}
  SIGNATURE(A1 a1) : a1_m(a1) {}
  ...
  SIGNATURE(A1 a1,...,AN aN) : a1_m(a1),...,aN_m(aN) {}

  const ARG1_t& operator[](const ARG<1>& ) const { return a1_m; }
  ...
  const ARGN_t& operator[](const ARG<N>& ) const { return aN_m; }

private:
  ARG1_t a1_m;
  ...
  ARGN_t aN_m;
};
```

```
template <typename SIG,int n>  struct ARG_TYPE { };
template <typename SIG> struct ARG_TYPE<SIG,1> {
   typedef typename SIG::ARG1_t Type_t;
};
...
template <typename SIG>  struct ARG_TYPE<SIG,N> {
  typedef typename SIG::ARGN_t Type_t;
};
```

Through `operator[]` the `SIGNATURE` structure offers access to the values. The `ARG_TYPE` structure allows access to the argument types. It has not been declared as a member of `SIGNATURE`, because specializing a member template without specializing the enclosing template is not allowed with C++. By introducing the functor tag `Substitute` (that holds an instance of a `SIGNATURE` struct - accessible through the member `signature`), substitution can be done by PETE's `forEach` function. Whenever a value of type `ARG` is reached, it is replaced by the suitable value of the signature:

```
// SIG is assumed to be of type SIGNATURE<>, n is an index
// that comes from an ARG<> value that's stored at the leaf we are currently
// visiting.
template <typename SIG,int n>
struct LeafFunctor< ARG<n>, Substitute<SIG> > {
   typedef typename ARG_TYPE<SIG, n>::Type_t Leaf_t;
   static inline Leaf_t apply(const ARG<n>& a,const Substitute<SIG>& s) {
     return s.signature[ a ];
   }
};
```

Any other types remain untouched.

Substitution indeed can be done during compile time: `apply` is a static inline function that does not change its arguments. Thus, a call to it can be optimized away.

4.2. Evaluation

After substitution the generic expression usually becomes an expression for which we can compute a result. Depending on the type of this result, different evaluation strategies have to be chosen. For the case that it is a user-defined class that supports the expression template functionality, we have to allow for the possibility of some existing sophisticated evaluation strategies that only the user's class is aware of. Thus, evaluation should remain the user's class' responsibility. For all other cases we can do evaluation on our own.

It is not only the result type that has to be taken into accout. The program context plays an important role, because an expression either needs to be evaluated, or has to become part of another expression:

```
using LAMBDA::x;
using LAMBDA::y;

Vec3 a,b,c,d;
cout << lambda(x,y, x + y)(a,b);          // evaluation
cout << lambda(x,y, x + y)(a,b) - c + d;  // become part of new expression
```

Of course `lambda(x,y, x + y)(a,b) - c + d` should yield the same code as `a + b - c + d` does. Immediately evaluating `lambda(x,y, x + y)(a,b)` - thus, returning a `Vec3` object - is not a good idea at this point, because some benefits of the expression template technique may get lost. Notice, that if directly evaluating the lambda term, the two examples will lead to different expression objects. In the first case (`lambda(x,y, x + y)(a,b) - c + d`), the lambda term is

evaluated first and the result of adding `a` and `b` as well as `c` and `d` will become part of the expression object. In the second case (`a + b - c + d`) all four `Vec3` variables will occur in the expression object. Thus, possibe optimization steps cannot include the `a + b` part of the expression. A better solution is to make `lambda(x,y, x + y)(a,b)` part of a new expression template tree.

The easiest way to make this possible is to wrap the result obtained by applying a function returned by lambda into PETE's `Expression` class template. The wrapped class template (called `FACT_PETE_ROOT`) stores a `SIGNATURE` object (according to the types/values that have been passed to `lFUNC`'s function call operator) and the generic expression object that originally has been passed to `lambda`.

As already mentioned above, the return type of an expression has influence on the evaluation strategy. To determine it, we first perform substitution and then traverse the expression template tree with PETE's meta program `ForEach`. The result type is computed bottom up: at each node a template meta program computes the return type according to the type of the node's childrens and the type of operation stored at the the node. This operation already has been discussed in the PETE's section and is selected by the `OpCombine` tag. To do compuatation at the leafs, **FACT!** provides the `GetLeafType` tag along with the following specialization of the `LeafFunctor` struct:

```
template<typename T>
struct LeafFunctor<T,GetLeafType> {
  typedef T Leaf_t;
  static inline Leaf_t apply(const Leaf_t& l,const GetLeafType& t) {
    return l;
  }
};
```

Computing the result type `ResultType` for an expression `E` finally looks like this:

```
typedef ForEach<E,GetLeafType,OpCombine>::Type_t ReturnType;
```

Once the return type is known, we have to check whether it is a class that offers expression template functionality. As mentioned in earlier, this functionality depends on the existence of a specialization of `CreateLeaf`. If no such specialization exists, PETE wraps values into the `Scalar` template before storing them in the expression tree. Thus, we just have to check whether `CreateLeaf<ReturnType>::Leaf_t` is equal to `Scalar<ReturnType>`. If not, we safely can assume `ReturnType` to be aware of expression templates.

For the case that `ReturnType` offers expression template functionality we suppose it to provide a constructor template that constructs a user object from an `Expression<>` object and return an appropriate temporary (see `CLE2E` below). Otherwise, we use PETE's `forEach` function to traverse the expression tree and perform computations according to the operators stored at the nodes. At the leafs we use the leaf-functor tag `EvalLeaf1` to access the values.

The following code section shows the complete code for the meta program `RetFLA` which selects the correct evaluation strategy for an expression type:

```
struct mTRUE {};
struct mFALSE {};

template < typename COND,typename THEN,typename ELSE>
struct mIF { typedef THEN Type_t; };
template < typename THEN,typename ELSE >
struct mIF<mFALSE,THEN,ELSE> { typedef ELSE Type_t; }
```

```
template < typename T1,typename T2 >
struct mEQUAL { typedef mFALSE Type_t; };
template < typename T >
struct mEQUAL<T,T> { typedef mTRUE Type_t; };

template < typename E,typename R >
struct CLE2N {
  static inline R apply(const E& e) {
    return forEach(e,EvalLeaf1(0),OpCombine());
  }
};

template < typename E,typename R >
struct CLE2E {
  static inline R apply(const E& e) {
    return R( e.expression() );
  }
};

template <typename E,typename R>
struct RetFLA {
   typedef typename mIF< typename mEQUAL<typename CreateLeaf<R>::Leaf_t,
                                         Scalar<R>
                                        >::Type_t,
                         CLE2N<E,R>,
                         CLE2E<E,R>
                       >::Type_t Type_t;
};
```

Evaluating an expression `e` of type `E` now simply means to call `RetFLA<E,ReturnType>::apply(e);`.

The remaining question is where to initiate the evaluation process. Usually, evaluation is triggered through a call to an assignment operator, which only can be overloaded through the definition of a class member function. Overloading the assignment operator for built-in types is not supported by C++. Also, a similar operation is needed to allow assignment from a built-in type that is obtained through the application of a function that was returned by lambda, like for instance in `int i = lambda(x,y, x + y)(2,3)`.

A possible solution is to equip `Expression<FACT_PETE_ROOT>` with a conversion operator that allows objects of this type to be converted into the `ResultType` that is related to the expression:

```
template <typename E,typename S>
struct Expression< FACT_PETE_ROOT<E,S> > {
...
  typedef ForEach<E,GetLeafType,OpCombine>::Type_t ResultType;
  operator ReturnType() const {
    return RetFLA< E,ResultType>::apply(*this);
  }
...
};
```

Finally, we can give the return type of `lFUNC2`'s function call operator:

```
template < typename A1,typename A2>
Expression< FACT_PETE_ROOT< E, SIGNATURE<A1,A2> > > operator()(A1 a1,A2 a2) {
 ...
}
```

4.3. Partial Application

Partial application means to bind the first *k* parameters of an *n*ary function to some specific values by yielding an *n-k* dimensional function. Thus, instead of replacing all lambda variables, partial application replaces just the first *k* variables. To implement partial application we must add some more function call operators to the `lFUNC` classes. Consider for example `lFUNC5`, then four additional parenthesis operators are needed. One that takes a single argument and returns an object of type `lFUNC4`:

```
template <typename A>
lFUNC4<typename ForEach<E,Substitute<SIGNATURE<A> >,TreeCombine>::Type_t >
  operator()(A a) {
  return forEach(e, Substitute<SIGNATURE<A> >( SIGNATURE<A>(a) ),TreeCombine()
}
```

another one that takes two values and returns an object of type `lFUNC3`, and so on.

Obviously, partially applying the result of a lambda function still yields a generic function. It is important to notice that type checking does not happen until full application occurs. Unfortunately, this behavior may cause hard to read error messages (e.g. if a suitable operator does not exist).

5. Using C++ Functions within a Lambda Expression

Using a C++ function inside a lambda expression - as we have shown above - is not possible, because applying a function usually forces a C++ compiler to produce code to execute that function. As with the overloaded mathematical operators, C++ functions should appear in the expression object rather than being executed. Furthermore, it is desirable to enable the user to pass lambda variables to a C++ function, which usually won't fit a C++ function's signature. Thus, a different representation for C++ functions is needed.

We already mentioned in [St00] that our curry function helps to shift the representation of a function into a form that we have control of. Utilizing this, it is not difficult to allow C++ functions to be used inside a lambda expression, if the user applies the curry function beforehand. In short, the `curry` function is somewhat similar to STL's `ptr_fun` function: it takes a pointer to a C++ function and returns a functional object.

Since it is necessary to store functions and their arguments inside the expression tree, a new structure template called `NODE`*`X`* (*`X`* is a placeholder for the dimension of the function) was developed. `NODE`*`X`* is a more general counterpart to PETE's `UnaryNode`, `BinaryNode` and `TernaryNode` structure templates. It offers a comparable functionality (storing an operation as well as some arguments, providing several access members), but also offers a conversion operator that allows a `NODE`*`X`* object to be converted into the type that would results from applying the stored operation to the stored operands.

Depending on the dimension the user has passed to the generator tool, *`X`* different `NODE`*`X`* structures are needed. Any of these may occur as argument to any of PETE's mathematical operators - yielding thousands of overloaded operators. To avoid this, the function call operator of the functor returned by `curry`, returns a value of type `NODE`*`X`* that has been wrapped into the structure template `FUNCTION` - thus, it returns a value of type `FUNCTION<NODE`*`X`*`>`. The `FUNCTION` structure acts as a proxy class: it offers a conversion operator that is identical to the one of the wrapped `NODE`*`X`* class thereby, making it possible to write for instance `cout << curry(sin)(3.0)`.

Finally, PETE's `MakeOperator` tool can be used to produce operators for the class template `FUNCTION` and it is possible to do

```
#define sqr curry(sqr)
#define sqrt curry(sqrt)
lambda(a,b,c, sqrt( sqr(a) + sqr(c) + sqr(b) ) )
```

and use function objects in lambda expression.

Since we have shown in [St00] that `curry` comes at no extra cost, we used a preprocessor directive to avoid typing `curry(sqr)` or `curry(sqrt)` all the time.

As long as the C++ function that is used within a lambda expression is free of side effects, the lambda expression will be as well. While it is impossible to recognize whether a function changes a global variable, side effects caused by arguments that get passed by value could be avoided by allowing curry to be applied to appropriate functions only.

6. Lambda Variables as Placeholders for Functions

In order to enable lambda variables to be placeholders for functions, several function call operators need to be added to the `ARG` structure. These operators return an instance of `NODE`*`X`* where the operation is represented by a lambda variable (to allow this node to be used in an expression, they get wrapped into the `FUNCTION` template as well). Now, the previously shown lambda expression could be rewritten like this:

```
#define sqr curry(sqr)
#define sqrt curry(sqrt)
lambda(f,a,b,c, sqrt( f(a) + f(c) + f(b) ) )(sqr)
```

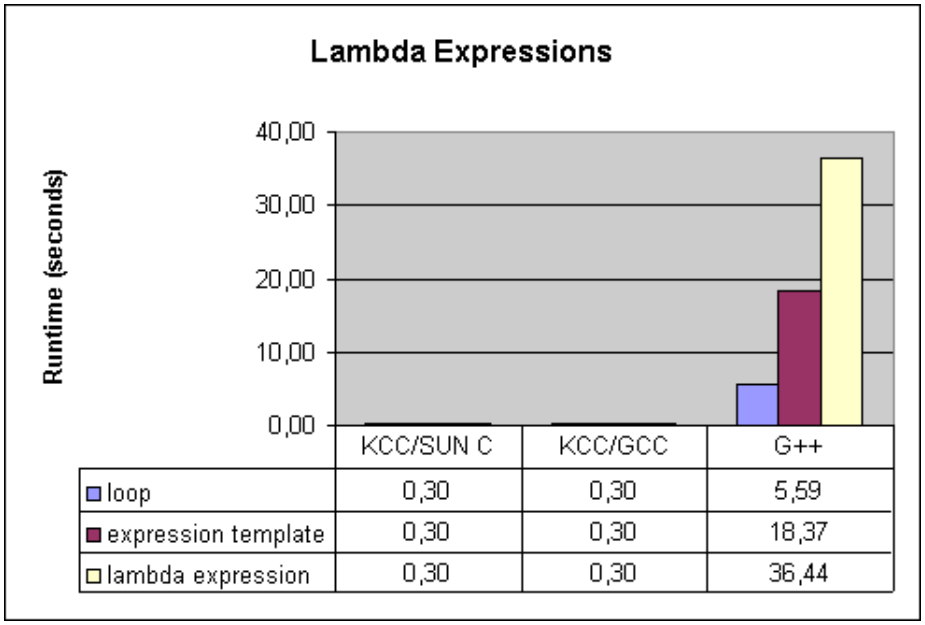
Note that there is a partial application - `f` is a placeholder for a unary function and is bound to `sqr` - the result is a ternary function.

7. Performance

To estimate the performance of our lambda function, we used the expression template aware `Vec3` class that has been described in section 3.1. We measured the time to add four instances of `Vec3` by using these methods:

- **loop**: we manually coded a loop that iterates through the vector components and performs the addition,
- **expression templates**: we simply wrote `e = a + b + c + d`, were `a` - `e` are all of type `Vec3` and let PETE do necessary optimizations,
- **lambda function**: we used `lambda(w,x,y,z, w + x + y + z)(a,b,c,d)`.

All those expression were evaluated fifty million times on a SunUltra 10 with a 333MHz UltraSparcIIi processor. We used Kuck and Associates' KCC version 4.0 with either SUN's C 5.0 or Gnu's C 2.95.2 as possible backend C compiler. Furthermore, we investigated GNU's C++ compiler 2.95.2.



As you can see from the above image, there indeed is no performance penalty if using our lambda function with KCC. Applying a lambda function to built-in types we obtained similar results: using KCC there was no difference in runtime between applying a lambda function and "directly" adding some built-in types.

8. Related Work

The lambda library [LL] also allows one to define generic function objects on the fly. Despite the name, this library does not focus on functional programming style. Rather, this library emphasizes imperative programming and allows multiple assignments, while loops, and several other imperative constructs within an expression that defines a function object. The lambda library has support for the generation of nullary, unary, binary, and ternary function objects. Support for functions of arbitrary arity is not planned by the authors as the lambda library primarily is meant to be used with STL algorithms, and none of those even accept ternary functions [Jaakko Järvi, personal communication]. In comparison to **FACT!** , the lambda library does not handle user-defined classes that offer expression template functionality. Thus, using such classes with lambda generated function objects may possibly result in a loss of runtime performance. However, the lambda library provides a simple way to define even very complex function objects through expressions.

9. Conclusion and Future Work

We have shown that the lambda function offers a convenient and efficient way to keep the definition and application of functions close together. Since there are no side effects with lambda functions, they are very useful in parallel environments and thus, we are considering using them to build stencil objects for POOMA [POOMA] . Stencil objects are used to define data-parallel operations on arrays where the computation involves neighboring array values. For example, users could write the following function:

```
double deriv2(Array &x, int i) {
  return x(i + 1) - 2 * x(i) + x(i-1);
}
```

Later in their code they can write data-parallel statements of the form `a = stencil(deriv2)(b)` to apply the computation `a(i)=b(i+1)-2*b(i)+b(i-1)` for all values of `i`. Note that the definition of the function and its use need to occur at separate places in the code. We could achieve the same result with a more compact notation using lambda functions (for example `a = stencil(lambda(x, x(1)-2*x(0)+x(-1)))(b))`. With the lambda function description, it would be easy to manipulate stencils, for example to compose them, or to form multi-dimensional products of one-dimensional stencils.

In a future project we will try to extend our lambda approach in order to become a Turing complete sub-language for C++. This project would make C++ an interesting target platform for developers of compilers for functional programming languages, as one could integrate the functional and object oriented programming paradigm. We also plan to investigate whether template meta programs will allow us to use our lambda technique to build a real compiler (e.g. use it to produce SSE or MMX code on an Intel CPU). Moreover, extending the lambda language such that a lambda expression may contain function definitions (e.g. let/letrec expressions like in ML) may yield the possibility to do context sensitive optimizations through template meta programs.

References

[C++]	International Standard, Programming Languages - C++, ISO/IEC: 14882, 1998
[Ha99]	*Scott Haney, James Crotinger, Steve Karmesin, and Stephen Smith*: PETE, the Portable Expression Templates Engine, Dr. Dobbs Journal, October 1999
[PETE]	PETE home page: http://www.acl.lanl.gov/pete
[CL98]	*Chris Cleeland, Douglas C. Schmidt and Timothy H. Harrison*: External Polymorphism, Proccedings of the 3rd Pattern Languages of Programmng Conference
[Vh95]	*Todd Veldhuizen*: Expression Templates, C++ Report, June 1995
[Vh95-2]	*Todd Veldhuizen*: Using C++ Template Meta Programs, C++ Report, May 1995
[EC00]	*Ulrich W. Eisenecker, Krzysztof Czarnecki*: Generative Programming, Addison Wesley, 2000
[SM00]	*Brian McNamara, Yannis Smaragdakis*: Functional Programming in C++
[FC++]	FC++ home page: http://www.cc.gatech.edu/~yannis/fc++"
[POOMA]	POOMA home page: http://www.acl.lanl.gov/pooma
[LL]	*Jaakko Järvi, Gary Powell*: The Lambda Library http://lambda.cs.utu.fi
[St00]	*Jörg Striegnitz*: Making C++ Ready for Algorithmic Skeletons, Internal Report IB08-2000, Research Center Jülich
[FACT]	FACT! home page: http://www.fz-juelich.de/zam/FACT