LA-UR- 00-4402

Approved for public release; distribution is unlimited.

Title: An Expression Template Aware Lambda Function

Author(s): Jörg Striegnitz
Stephen A. Smith

Submitted to: Workshop on C++ Template Programming
Efurt, Germany

Los Alamos
NATIONAL LABORATORY

DISCLAIMER

An Expression Template aware Lambda Function

Jörg Striegnitz

Central Institute for Applied Mathematics
Research Center Jülich
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. We will show how the paradigms of lambda functions and expression templates fit together in order to provide a means to increase the expressiveness of existing STL algorithms. We will demonstrate how the expression templates approach could be extended in order to work with built-in types. To be portable, our solution is based on the Portable Expression Template Engine (PETE), which is a framework that enables the development of expression template aware classes.

1. Introduction

The Standard Template Library (STL) [C++] contains many so called *Higher Order Functions* (*HOF*s). These are functions that take functional arguments and/or return functional results (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 plenty of small functions or functional objects respectively. The *point of use* and *the point of declaration* may get more and more dispersed, making code harder to read and understand.

This problem may even be worse, as it is impossible to pass function templates to STL's HOFs. In order to mimic rank-2 polymorphism (passing polymorphic arguments to polymorphic functions) , either function overloading or the definition of a class with generalized `operator()` has to be taken into account (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 function called *lambda* to define polymorphic functions.

Our C++ framework **FACT!** (Functional Additions to C++ through Templates and Classes [FACT]) offers such a lambda function and thereby helps to keep point of use and point of declaration close together. As with its 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 will discuss the implementation of our lambda functions and show how to implement expression template aware classes. After giving a short introduction into the lambda functions, we will show how to build lambda expressions by relying on 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

A 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 dimension 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.

Evaluation of a function that was created through `lambda`, 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. Next, 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` goes like this:

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

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

Functions returned by `lambda` are of polymorphic type , 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, they even may be bound to values of different type.

Lambda expressions may contain calls to other functions or just return constants:

```
lambda(a,b,c, sqrt( sqr(a) + sqr(c) + sqr(b) ) )
lambda(x, 2)
```

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

```
lambda(f,x,y, f(x,y) )
lambda(f, f) ( lambda(x,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 offers the possibility of partial application.

Functions returned by `lambda` get their arguments *by value*, thus they do not impose any side effects. This makes lambda very useful for parallel environments.

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

- functions of varying signature,
- 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 could be solved through function overloading. There is no need to take care of lambda variables that don't occur in the lambda expression, thus, it is sufficient to store them inside the expression. The remaining task is to develop mechanisms to store, manipulate, and evaluate lambda expressions.

With respect to performance, expression templates [Vh95] are considered as a possible way to handle lambda expressions. Expression templates are nested template structures, used to represent the parse tree of an expression. They are build during compile time through overloaded arithmetic operators, which - instead of immediately applying an operation - return objects that incrementally built 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 the expression template tree during compile time and in conjunction with inlining techniques the expression object could be used to produce efficient code.

Using template meta programs, substitution of lambda variables could be done during compile time, but relies on different lambda variables having different types. This is required, because template meta programs actually get types as arguments and thus, in order to support functions of arbitrary dimension, arbitrary types of lambda variables are needed:

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

`ARG` is a suitable representation, because it can be used to form at least `numeric_limits<int>::max()` different types, which we assume to be an acceptable limit. For convenience reasons, **FACT!** offers a huge amount of predefined lambda variables, all of them defined in the scope of `namespace LAMBDA` and thus, the user usually does not need to pay attention on the *real type* of a lambda variable , but just types 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 the means to make a user defined class aware of expression templates. 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 : a` cannot be overloaded.

In order to integrate user defined types, special variants of all these operators need to be added, which can easily be achieved through PETE's `MakeOperator` tool. After creating these operators, there are three tasks left to make a class aware of expression templates:

- tell how the type should be stored inside the expression tree
- add an assignment operator to the user class that takes a PETE-expression
- tell how to access data during evaluation

To see how PETE works, 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; }
  Vec3(const Vec3& o) { d[0]=o.d[0]; d[1]=o.d[1]; d[2]=o.d[2]; }
  ~Vec3() {}
      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);
  }
};
```

To save space and to avoid unnecessary calls to copy constructors, references to `Vec3` are stored at the leaves (PETE identifies references by wrapping them into the `Reference` struct).

Besides offering the type of the leaf, `CreateLeaf` provides the `apply` method to build one. In this case, it takes a reference to an instance of `Vec3` and wraps it into the `Reference` class template. If there is no specialization of `CreateLeaf`, PETE wraps leafs into the template class `Scalar`. Thus, the `Scalar` wrapper provides a convenient way to differentiate between expression-aware nodes from other objects.

In PETE an expression object has type `Expression<T>`. To traverse the expression tree during compile time, 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`. There are two default combinator tags in PETE: `OpCombine` and `TreeCombine`. `OpCombine` combines results according to the operators stored at non-leaf nodes, while `TreeCombine` is used to combine non-leaf nodes in order to build a new expression object.

There are also some predefined functor tags. One of them is the class `EvalLeaf1`, which stores a single

integer index, accessible through the method `val1()`. 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()];
  }
};
```

This `LeafFunctor` acts on leafs of type `Vec3` and performs the operation selected by `EvalLeaf1`. It offers 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 now is 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() );
}
```

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>`. The only thing to be done is to tell PETE how values of type `ARG` should be stored in the expression object. As for the `Vec3` class this is done by supplying a suitable specialization of `CreateLeaf`.

The lambda function has to return a function whose computational rule comes from the generic expression object. With C++, functions could be modeled through functors (classes that provide a parenthesis - or function call operator). The dimension of this operator depends on the number of lambda variables that the generic expression object contains. 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` (that is expected to contain exactly two lambda variables) and provides a binary function call operator. Since `lFUNC2` should represent a polymorphic function, this operator is declared as a template (the return type *`result_t`* will be discussed in a later section).

The lambda function itself just has to take some lambda variables, as well as a generic expression object, and has to return an instance of an appropriate `lFUNC` 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 choose the variant of lambda to call. Furthermore, `m,n` and `o` maybe any integer values. To make substitution easier, these values get normalized to `1,2...` by some sophisticated template meta programs.

To support functions of arbitrary dimensions, an endless number of `lFUNC` classes and `lambda` functions are needed. To cover most situations, we developed a code generating tool, that is supplied with the largest functional dimension to support, and produces a C++ header file that contains all the necessary definitions.

4. Applying the Result of a Lambda Function

The purpose of `lFUNC`'s function call operator has to be: substitute all lambda variables by the given arguments and return

- an expression object, if the result type of the expression object is aware of expression templates,
- or the result of evaluation, otherwise.

Since function overloading on return types is not possible with C++, a proxy class needs to be returned by `lFUNC`'s parenthesis operator. Before discussing this proxy in detail, we will show how to do substitution.

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 `1FUNC`'s parenthesis operator and due to normalization, association with the corresponding `ARG<i>` values of the expression object is clear.

Now, substitution simply could 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 once 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 {
  SIG() {}
  SIG(A1 a1) : a1_m(a1) {}
  ...
  SIG(A1 a1,...,AN aN) : a1_m(a1),...,a1_N(aN) {}

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


  typedef A1 ARG1_t;
  ...
  typedef AN ARGN_t;
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 - accesible through the member `signature`), substitution could be done by PETE's `forEach` function. Whenever a value of type `ARG` is reached, it is getting replaced by the suitable value of the signature:

```
template <typename SIG,int n>
struct LeafFunctor< ARG<n>, Substitute<SIG> > {
   typedef typename ARG_TYPE<SIG, n> Leaf_t;
   static inline 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 it's arguments. Thus, a call to it can be optimized away.

4.2. Evaluation

Not only expression templates have to be considered during evaluation. The result of evaluation also should be *reusable* for other expression templates. For instance,

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

Vec3 a,b;
lambda(x,y, x + y)(a,b) - Vec3(1,2,3)
```

should yield the same code as `a + b - Vec3(1,2,3)` does.

The easiest way to achieve this, is to use PETE's `Expression` class template as a wrapper for the previously mentioned proxy class, thus doing a template specialization. The class to be wrapped must memorize the signature that has been passed to `lFUNC`'s function call operator, as well as the generic expression object, since this is essential to do evaluation. It should have a member function that performs substitution and returns the monomorphic expression. In **FACT!** this class is called `FACT_PETE_ROOT` and the specialized expression type is

```
template <typename E,typename S>
struct Expression<FACT_PETE_ROOT<E,S> > { ... };
```

Usually, evaluation is triggered through a call to an assignment operator. The assignment operator has to be a member function and therefore, overloading for built-in types is not possible. Anyway, something similar is needed to allow assignment from a lambda function which returns a built-in type, like for instance `int i = lambda(x,y, x + y)(2,3)`. A possible solution is to supply a conversion operator which allows us to convert an expression object into the type of the result it represents. This requires knowledge of the result type of an expression, but fortunately, PETE offers suitable mechanisms to obtain it. Using the compose tag `OpCombine`, PETE uses several template meta programs to compute the result type. If necessary, the user can specialize from some class templates to specify the return type for computation for his own classes. This cannot even be done on an argument type level, but PETE also allows to specify different return types for different operators (e.g. Matrix plus Matrix yields a Matrix while Matrix multiplication yields a vector).

Once the return type of the expression object is known, it is quite easy to determine wether it is aware of expression templates, or not. If the result type `T` is *not* aware of expression templates, then `CreateLeaf<T>::Leaf_t` is equal to `Scalar<T>` and thus, the following meta programs can be used to select the return type.

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

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

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

// This proxy gets returned in the case where
// the result type of the expression object is a
// built-in type
template < typename E,typename R >
struct CLE2N {
  static inline R apply(const E& e) {
    return forEach(e,EvalLeaf1(0),OpCombine());
  }
};

// This proxy gets returned in the case where
// the result type of expression object is aware of
// expression templates
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;
};
```

Notice, that `CLE2E` relies on the result type having a constructor from `Expression`. To avoid implicit conversion, this constructor should be declared `explicit`.

In the end, `Expression<FACT_PETE_ROOT<E,S> >` should provide a cast operator that returns a proxy of type `RetFLA<E,R>::Type_t`, where `R` is the type obtained by `ForEach<E,GetLeafType,OpCombine>::Type_t;` (the tag `GetLeafType` returns the type/value of a leaf).

Finally, `1FUNC2`'s parenthesis operator should return an object of type `Expression<FACT_PETE_ROOT<E,SIGNATURE<A1,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 means to replace just the first k variables. In practice this means to add some more functional operators

to the `1FUNC` classes. Consider for example `1FUNC5`, then four additional parenthesis operators are needed. One that takes a single argument and returns an object of type `1FUNC4`:

```
template <typename A>
1FUNC4<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 `1FUNC3`, and so on.

Obviously, partially applying the result of a lambda function still yields a generic function and it is important to notice that type checking does not happen until full application occurs. Unfortunately this causes 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 it 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 be 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 to it before. In short, the `curry` function is somewhat similar to STL's `ptr_func` 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 `NODEX` (*X* is a placeholder for the dimension of the function) was developed. `NODEX` 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 cast operator that allows a `NODEX` 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 `NODEX` 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 `NODEX` that has been wrapped into the structure template `FUNCTION` - thus, it returns a value of type `FUNCTION<NODEX>`. The `FUNCTION` structure acts as a proxy class: it offers a conversion operator that is identical to the one of the wrapped `NODEX` class, making it possible to write e.g. `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) ) )
```

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

6. Lambda Variables as Placeholders for Functions

In order to enable lambda variables to be placeholders for functions, the only thing to do is to add a unction call operator to the `ARG` structure that returns an instance of `NODE` *X* whose operation is represented by a lambda variable (to allow this node to be used in an expression, it has to be 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`.

7. Performance

To estimate the performance of our lambda function, we used the expression template aware `Vec3` class that has been described in an earlier section. 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. Conclusion

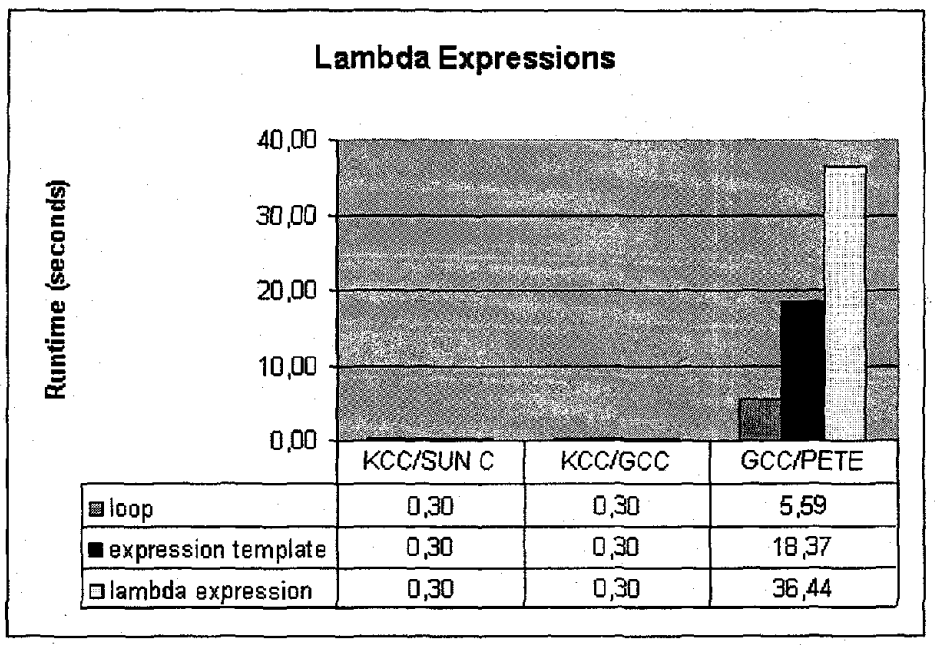
We have shown that the lambda function offers a convenient and efficient way to keep the definition and

application of functions close together. Since there are no side effects with lambda functions, they are very useful in parallel environments and thus, we considered to use them to build stencil objects for POOMA [POOMA]. We could invent a compact notation for stencils using lambda functions (such as `a = stencil(lambda(x, x(1) - 2 * x(0) + x(-1)))(b) )`). Unlike the current implementation of POOMA stencils, the definition of the stencil can appear at the point of use. 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 will not only make C++ an interesting target platform for developers of compilers for functional languages. We also think of investigating, if 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) 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
[Vh95]	*T. Veldhuizen*: Expression Templates, C++ Report, June 1995
[Vh95-2]	*T. Veldhuizen*: Using C++ Template Meta Programs, C++ Report, May 1995
[EC00]	*U. Eisenecker, C. Czarnecki*: Generative Programming, Addison Wesley, 2000
[MS00]	*B. McNamara, Y. 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
[St00]	*J. Striegnitz*: Making C++ Ready for Algorithmic Skeletons, Internal Report IB08-2000, Research Center Juelich
[FACT]	FACT! home page: http://www.fz-juelich.de/zam/FACT



	KCC/SUN C	KCC/GCC	GCC/PETE
loop	0,30	0,30	5,59
expression template	0,30	0,30	18,37
lambda expression	0,30	0,30	36,44