

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

Interner Bericht

**Higher Order Functions and Partial
Application in C++**

Jörg Striegnitz

FZJ-ZAM-IB-9913

September 1999

(letzte Änderung: 18.10.99)

Higher Order Functions and Partial Application in C++

Jörg Striegnitz
Forschungszentrum Jülich GmbH
Zentralinstitut für Angewandte Mathematik
52425 Jülich, Germany
J.Striegnitz@fz-juelich.de

Abstract

Polymorphic types, higher order functions, and partial application are common features in functional programming languages. They allow a compact formulation of algorithms and their integration into imperative languages will increase expressiveness. Current approaches to integrate functional features into imperative languages are either based on the definition of a new language, or implemented as a runtime mechanism. While the first variant needs a special compiler, the second is less efficient and may not support polymorphic types. In this paper we will present a method of how to integrate functional features into C++ without the need of runtime mechanisms, special preprocessors, or compilers. As a practical application of these features we will demonstrate how parallel environments can benefit from these features.

1 Introduction

Higher order functions are defined as functions with functional arguments and/or results. In common imperative languages (such as C) they may be represented as functions taking pointers to functions as arguments and/or returning a pointer to a function as result. Although this concept is useful to simulate higher order functions it still lacks polymorphic types as supported by functional programming languages [1]. C++ supports polymorphic types in the form of templates. Defining a template version of the previously discussed C-function gives usable polymorphic higher order functions, but the power of partial application is still missing.

To introduce partial application we first need to define **currying**. Usually an n dimensional function is of type $f : A_1 \times \dots \times A_n \rightarrow R$. Therefore, it takes an n dimensional tuple and returns a value of type R . We denote this by writing $f(a_1, \dots, a_n) = r$. It was first shown by Frege in 1893 and later independently rediscovered by Schoenfinkel that it is sufficient to restrict attention to unary functions. Every function $f : A_1 \times \dots \times A_n \rightarrow R$ can be turned into a function $g : A_1 \rightarrow \dots \rightarrow A_n \rightarrow R$ (with “ \rightarrow ” being associative to the right). g is called the **curried form** of f . From a functional point of view, currying can be described

as a function of type

$$\text{curry} : ((A_1 \times \dots \times A_n) \rightarrow R) \rightarrow (A_1 \rightarrow \dots \rightarrow A_n \rightarrow R)$$

Note, that `curry` returns a unary function that returns a unary function ... that returns a unary function that returns a value of type R . This offers the possibility to pass less than n arguments to g but still obtaining a result — a unary function. The application of a curried function $g : A_1 \rightarrow \dots \rightarrow A_n \rightarrow R$ to its first m ($m < n$) arguments (called **partial application**) yields a new function $h : A_{m+1} \rightarrow \dots \rightarrow A_n \rightarrow R$. With a set of arguments $Arg = \{x_i, \dots, x_m | x_i \in A_i\}$, h is defined by $h(a_{m+1}) \dots (a_n) = g(x_1) \dots (x_m)(a_{m+1}) \dots (a_n)$

Partial application offers more generality. Consider for example the function `MAP`. `MAP` takes a unary function f and a list l as arguments, applies f to each item in the list and stores the result into a new list, which is returned as result. Example 1 shows the polymorphic formulation of `MAP` in C++.

Example 1 Definition of `map` in C++.

```
template <class T, class F>
vector<T> MAP(vector<T> & v, F f) {
    typedef typename vector<T>::iterator iterator;
    vector<T>      n_v(v.size());
    iterator n_start = n_v.begin();
    iterator start  = v.begin();
    while (start != v.end())
        *n_start++ = f(*start++);
    return n_v;
}
```

To increment each element in a list someone might implement a function `T inc(T arg1)` (which returns the value of its argument plus one) and then call `l=MAP(l, inc)`. This looks quite comfortable, but for every value to be added, a special function is needed. A possible solution would be to write a new version of `MAP`, which takes the value to be added as third argument. Thinking about this new version of `map`, you may discover that it will be quite similar to the existing one and you may look for a way to save work. Having partial application at hand, you can use the function `T add(T arg1, T arg2)` and its partial application `add(a)` to call

$$\text{list} = \text{MAP}(\text{list}, \text{add}(\text{a})) \tag{1}$$

and easily have achieved the desired goal. The reuse of existing code is very important, e.g. consider `MAP` to be implemented data-parallel (the elements of the list get distributed among a set of processors and each processor applies the given function to its subset of data) than the user will expect a specialized version of `MAP` to be data-parallel as well.

To use partial application in C++ we need a curry function. To be flexible, this function should work with existing C++ functions and functors. In this paper we will show how to offer this operator to C++ users, by the use of standard conformant C++ code. Before discussing our approach we will show existing ones and their limits. After discussing our solution in detail, we will give some hints on how to benefit in parallel environments.

2 Related Work

The combination of polymorphic types, higher order functions, and partial application has been reserved to functional programming languages for a long time. One of the first approaches to integrate these concepts into imperative languages was started with the SKIL-Project [2]. SKIL is a programming language based on C that supports all these features. Although SKIL leads to good performance and is easy to learn, its syntax conflicts with C++ and the combination of SKIL and C++ seems to be a difficult (maybe impossible) task. Moreover, SKIL needs a special compiler and, therefore, is hard to extend.

In order to integrate functional features into C++ it would be desirable to depend on C++ only. A first way to do so can be found in the Standard Template Library (STL) [3, 4, 5], which is part of the C++ ISO/ANSI standard. Consider the function `for_each`, whose definition is shown in example 2.

Example 2 Definition of `for_each` in the Standard Template Library (STL).

```
template<class InIterator, class Function>
Function for_each(InIterator first, InIterator last, Function f) {
    while (first != last)
        f(*first++);
    return f;
}
```

`for_each` is a generic function as well as a higher order function (it takes a function as argument and it returns a function). Notice that `Function` maybe bound to a class that provides an `operator()` to simulate the behavior of a function. Such a class is often called a **functor** or **functional object** (note: using functors instead of functions is not a restriction, because you may pass every unary C++ function to `for_each` by wrapping it into a class type).

Unfortunately there is only limited support for partial application in the STL. Only binary functions may be partially applied. This is done by applying the adaptor `bind1st` to STL conformant functors:

```
for_each(vec.begin(), vec.end(), bind1st(plus<int>(), a))      (2)
```

Having partial application at hand it would be possible to write

```
for_each(vec.begin(), vec.end(), plus<int>()(a)) .          (3)
```

Both (2) and (3) are adding a value to each element of a vector, but (3) is much easier to read and understand.

Besides `bind1st` the STL gives the opportunity to bind the second argument of a function by using the adaptor `bind2nd`. The application of `bind2nd` does not lead to partial application as defined above. Moreover, the use of these adaptors may result in confusing code, especially for non commutative functions:

```
find_if(vec.begin(),vec.end(),bind2nd(less<int>(),1000))    (4)
```

```
find_if(vec.begin(),vec.end(),bind1st(less<int>(),1000))    (5)
```

While the first application of `find_if` finds the first value less than 1000, the second application returns the first number that is not less than 1000. This means that the semantic of `less` gets changed by the application of `bind1st` or `bind2nd`. This makes code hard to read and maintain.

3 Our Approach

Our goal is the integration of functional programming features into C++ without introducing any overhead and depending on standard conformant C++ code only. In the following sections we will demonstrate how to achieve these goals by the use of templates and classes. Since partial application is based on currying, we will first show how an appropriate operator could be represented by a special class type.

3.1 f-functors

We distinguish three ways of representing functions:

- C++ functions,
- functors,
- and our new concept: **f-functors**.

C++ functions are exactly what they are expected to be — global functions or static class member functions. As already described above, functors are classes which provide at least a single `operator()`. This operator may make a call to a C++ function, a call to an `operator()` of a different functor, or may calculate the return value by itself (see example 3). In all cases it is compatible with the signature of the function it should represent, and therefore, we call the functor to have the same signature as well (keeping in mind that it is of class type).

f-functors are special functors. They also support an `operator()`, but in contrast to functors this operator always has the signature of a unary function. Its return value may be either an f-functor or a value of the return type specified by the functors signature. To avoid side-effects, functional languages do not allow functions to change their arguments (there is no call by reference). To

Example 3 Examples for different functor implementations.

```
// A global function is a C++ function
inline int add(int a,int b) { return a+b; }
// A static class member is a C++ function
struct CADD {
    static inline int madd(int a,int b) { return add(a,b); }
};
// Functor calculating result by itself
struct CADD1 {
    inline int operator()(int a,int b) const { return a+b; }
};
// Functor calling a C++ function
struct CADD2 {
    inline int operator()(int a,int b) const { return add(a,b); }
};
// Functor using another functor
struct CADD3 {
    CADD1 add1;
    inline int operator()(int a,int b) const { return add1(a,b); }
};
```

simulate this behavior, f-functors should not do it either. An f-functor always **has a** functor, while a functor **may have** a functor or a C++ function. Thus, an f-functor does not know how to compute a result, but needs a functor to tell it how.

From a programmers point of view the encapsulation of a functor into an f-functor can be thought of as the application of the curry function to that functor. But as we will see, a single f-functor is not sufficient at all.

The curry function should not depend on a function's signature. This implies generality in two dimensions: types (argument types and return type) and the number of arguments. Since C++ does not support this kind of generality directly, its representation must be split into two parts. While different argument and return types could be represented by the use of templates, functions of different dimension must be represented by different classes.

f-functors are implemented as class templates and for every dimension, a function that should be curried may have, a suitable f-functor is needed. An f-functor for currying unary function has three template arguments (see example 4). A1 denotes the argument type, R is the result type, and F denotes the functor type. FUNC_01 has a functor f and hence holds a constant reference to it. As unary functions and their curried versions are identical, nothing special has to be done in this case.

To demonstrate partial application and currying we need at least a binary function. It's representation as f-functor will be constructed step by step in the following section. Example 5 shows a basic implementation without the power

Example 4 Class template for a polymorphic f-functor, representing unary functions

```
template<class A1,class R,class F>
struct FUNC_01 {
    typedef FUNC_01<A1,R,F> funcfunc_t;
    const F & f;
    FUNC_01(const F & _f) : f(_f) {}
    inline R operator()(A1 _a1) const { return f(_a1); }
};
```

of partial application. In contrast to our previous statement, that f-functors only support a unary `operator()`, the binary `operator()(const A1 _a1,A2 _a2)` has been defined. This is not necessary and only done for convenience, because it offers the freedom to mix curried and uncurried versions of a functor (thus, it does not matter if you call $f(1,2)$ or $f(1)(2)$).

Example 5 Class template for f-functors for binary functions

```
template <class A1,class A2,class R,class F>
struct FUNC_02 {
    typedef FUNC_02<A1,A2,R,F> funcfunc_t;
    const F & f;
    FUNC_02( const F & _f ) : f(_f) {}
    inline R operator()(const A1 _a1,A2 _a2) const {
        return f(_a1,_a2);
    }
};
```

Remember: currying a binary function $f : A1 \times A2 \rightarrow R$ yields a function $g : A1 \rightarrow A2 \rightarrow R$ and partial application of g to its first argument leads to $h : A2 \rightarrow R$ with $g(a1) = h$ and $f(a1, a2) = g(a1)(a2) = h(a2)$:

$$f \xrightarrow{\text{curry}} g \xrightarrow{\text{partial application}} h$$
$$\text{curry}(f)(a) = g(a) = h$$

To allow partial application we need an `operator()(const A1 _a1)` (representing g) that has to return an f-functor (representing h). This f-functor's `operator()` should take an argument of type `A2`, return a value of type `R`, and should get its calculation rule from a functor, which fulfills the following needs:

- store the f-functor $f : A1 \times A2 \rightarrow R$ that has been partially applicated,
- store a value $a1$ of type `A1`,
- provide an `operator()(A2 _a2)` that should return $f(a1, _a2)$.

We will call such a functor **pa-functor**. pa-functors are used to pass calculation rules to f-functors. Since this is necessary during partial application only, they are defined in the local scope of the f-functor. Example 6 shows the complete implementation of an f-functor representing binary functions.

Example 6 Class template for f-functors for binary functions

```
template <class A1,class A2,class R,class F>
struct FUNC_02 {
    typedef FUNC_02<A1,A2,R,F> funcfunc_t;
    const F & f;
    FUNC_02( const F & _f ) : f(_f) {}
    inline R operator()(const A1 _a1,A2 _a2) const {
        return f(_a1,_a2);
    }
    struct PAF {
        const A1          a1;
        const funcfunc_t & f;
        PAF(const funcfunc_t &func,const A1 _a1) : f(func),a1(_a1) {}
        inline R operator()(A2 _a2) const { return f(a1,_a2); }
    };
    typedef FUNC_01<A2,R,PAF> f1_type;
    inline const f1_type operator()(const A1 _a1) const {
        return f1_type( PAF(*this,_a1) );
    }
};
```

Now have a look at an example that demonstrates how partial application is done. Consider for example the following function and its corresponding functor:

```
inline int my_mult(const int a,int b) { return a*b; }
struct Cmult {
    inline int operator()(const int a,int b) const {
        return my_mult(a,b);
    }
};
```

To get a curried version of `my_mult` we need to define the functor `Cmult` and have to declare a variable `fmult` of type `FUNC_02<const int,int,int,Cmult>`. This f-functor has the signature `const int → int → int` and gets his calculation rule from class `Cmult`'s `operator()`. If partial application occurs, we will retrieve an object of type

`FUNC_01<int,int,FUNC_02<const int,int,int,Cmult>::PAF >`.

This is an f-functor that has the signature `int → int` and gets his calculation rule from the pa-functor `FUNC_02<const int,int,int,Cmult>::PAF`. The pa-

Object	Calculation rule derived from	Signature
C function	—	$A_1 \times A_2 \times A_3 \rightarrow R$
Functor	C function	$A_1 \times A_2 \times A_3 \rightarrow R$
FUNC_O3	Functor	$A_1 \rightarrow (A_2 \rightarrow A_3 \rightarrow R)$
FUNC_O2	FUNC_O3::PAF	$A_2 \rightarrow (A_3 \rightarrow R)$
FUNC_O1	FUNC_O2::PAF	$(A_3 \rightarrow R)$

Table 1: All types and functions involved during partial application of a three dimensional function.

functor’s `operator()` will return a value of type `int` and thus we can call `fmult(3)(5)` instead of `my_mult(3,5)`.

Table 1 shows calculation rules and signatures involved during partial application of a three dimensional function. It is easy to build up f-functors for functions of a dimension greater than two. Let $A_1 \times \dots \times A_n \rightarrow R$ be the signature of a function f . Then the corresponding f-functor has to define `operator()` n times. At least one unary `operator()` is needed to represent the curried version of f . The other $n - 1$ operators are provided for convenience. One `operator()` is to simulate f itself, thus should take n arguments and return a result of type `R` and the last $n - 2$ operators are used to simplify notation (e.g. you may write $f(a_1)(a_2)(a_3)$, $f(a_1, a_2, a_3)$, $f(a_1, a_2)(a_3)$, or $f(a_1)(a_2, a_3)$). To avoid defining all these f-functors by hand, we have written a special tool that generates an appropriate header file.

So far we are able to derive curried versions of C++ functions. As mentioned above, functional programming languages do not allow arguments to be passed by reference. In order to simulate this behavior, we either have to avoid call by reference (references and pointers as arguments), or we must force all arguments of a function to curry to be `const`. Since call by references does not only give the possibility to return changed arguments, but also avoids copying large data structures onto the stack, we have chosen the second variant. To make things more clear consider the following example:

```
struct unsecure_mult {
    inline int operator()(int & a,int b) { a=b ; return a*b; }
};
...
FUNC_O2<int &,int,int,unsecure_mult> fmult;
int a=5;
MAP(vec, fmult(a));
...
```

The call to `MAP` leads to unexpected results, because `a` gets changed by every application of `fmult(a)` to a vector element. Thus, the application of `fmult` causes side effects.

In contrast to functional programming languages we may allow the last argument of a function to be of non constant type. This is because the last

parameter does not play a role when partial application occurs. Moreover, we are not dealing with lazy evaluation and therefore, side effects are very easy to detect when a function gets fully applied. Unfortunately, declaring the involved parameters to be `const` (as shown above) is not sufficient. For the moment we will not bother and discuss further details in the next section.

3.2 Shifting functions from C-space to functional space

In the previous section we have shown how to implement currying and partial application in C++. Our approach is based on functors and if you like to curry an existing C++ function, you first need to define a suitable functor and then have to declare a variable of f-functor type. This section will focus on how to make this process more convenient.

First we define the set of all C++ functions and functors as the **C-space** and the set of all f-functors as **functional space**. Then shifting a C++ function f from C-space to functional space consists of two steps:

1. definition of a functor,
2. declaration of an f-functor variable.

To eliminate one of these steps we propose a generator class, offering an appropriate f-functor type as well as a static member variable of this type.

Example 7 Generator class for a unary f-functor

```
template <class A1,class R,R (*cfunc)(A1)>
struct MakeF1 {
    typedef MakeF1<A1,R,cfunc>      func_t;
    typedef FUNC_01<A1,R,func_t>    funcfunc_t;
    MakeF1() { }
    static const funcfunc_t f;
    inline R operator()(A1 _a1) const { return cfunc(_a1); }
};
template<class A1,class R,R (*cfunc)(A1)> const
MakeF1<A1,R,cfunc>::funcfunc_t MakeF1<A1,R,cfunc>::f;
```

For the same reasons as for f-functors, the generator class has to be general and therefore, was implemented as a template class. The count of template parameters depends on the dimension of the function to be represented. For a function of dimension n the first n template parameters define the argument types, the $(n+1)$ th template parameter denotes the return type and the last one is of type "pointer to function", where the function must match the signature defined by the previous template parameters. Example 7 shows a generator class for f-functors used for unary functions. Notice that `MakeF1` is a functor and the f-functor type `funcfunc_t` is based on it. For convenience `MakeF1` also provides an instance of `funcfunc_t`. Generator classes for f-functors representing n-ary

functions can be defined accordingly and they allow to shift a C++ function to functional space with only one line of code — there is no need for a temporary variable (see example 9 — first line of main).

This works quite well, but the user has to distinguish between generators and f-functors. It would be more convenient, if the generator itself is an f-functor. To achieve this, we introduce a new struct that derives from MakeFX's `funcfunc_t`

```
template <class A1,class A2,class R,R (*cfunc)(A1,A2)>
struct mkFF2 : public MakeF2<A1,A2,R,cfunc>::funcfunc_t { };
```

and does nothing else, but being an f-functor.

Generator classes automate the process of generating f-functor types and provide an instance of such an f-functor. If you do not want to bother about the type of an f-functor, you may prefer a second method to get a curried form of a function. The basic idea is to provide a function, which takes a C++ function as argument and returns a suitable f-functor. Such a function may have the following form

```
template <class A1,class A2,class R>
FUNC_02<A1,A2,R,MakeF2<A1,A2,R,cfunc> > fcxx1(R (*cfunc)(A1,A2)){
    return MakeF2<A1,A2,R,cfunc>::f;
}
```

Although `fcxx1` looks genius, it is no valid C++ code. This is because the compiler needs the value of `cfunc` to compute the return type, but since this value is a parameter to `fcxx1`, it is accessible during runtime only and no template instantiation is possible. To simulate the behavior of `fcxx1` we first need a template for a functor class that does not depend on a pointer to a C++ function (see example 8).

Example 8 Functor template, which does not have a "pointer to function" as template parameter.

```
template <class A1,class A2,class R>
struct MkOp2 {
    typedef R (*func_ptr_t)(A1,A2);
    const func_ptr_t & func_ptr;
    MkOp2(const func_ptr_t & fptr) : func_ptr(fptr) {}
    inline R operator()(A1 _a1,A2 _a2) const {
        return func_ptr(_a1,_a2);
    }
};
```

Now it is easy to give a function, which takes a pointer to a C++ function and returns a functor:

```

template <class A1,class A2,class R>
inline const MkOp2<A1,A2,R> mkop2( R (* const func)(A1,A2) ){
    return MkOp2<A1,A2,R>(func);
}

```

Having this function at hand, it is not complicated to give a function that fulfills the desired needs:

```

template <class A1,class A2,class R>
inline const FUNC_02<A1,A2,R,MkOp2<A1,A2,R> >
    curry(R (* const func)(A1,A2) ){
    return FUNC_02<A1,A2,R,MkOp2<A1,A2,R> >(mkop2(func));
}

```

By supplying MkOp functors and mkop functions for functions of different dimension and providing suitable overloaded versions of the `curry` function, it becomes very easy to get a curried form of an existing C++ function. But consider the difference between both methods. `MakeFX` is primary used to generate types (the static member variable was provided for convenience only), while `curry` is not useful for type computations, but is used to retrieve an object. Therefore, you usually will need both methods.

Example 9 Examples showing the use of the generator classes and the `curry` function.

```

template<class A2,class R,class OP>
inline R g(const FUNC_01<A2,R,OP> & f,A2 a2) { return f(a2); }

inline int add(const int a,int b) { return a + b; }

inline int sum(const int a,const int b,int c) { return a+b+c; }

int main(int argc,char **argv) {
    cout << MakeF2<const int,int,int,add>::f(1)(2)<<endl;
    cout << mkFF2<const int,int,int,add>()(1)(2)<<endl;
    mkFF3<const int,const int,int,int,sum> fsum;
    cout << fsum(1)(2)(3) << endl;
    cout << fsum(1,2,3) << endl;
    cout << fsum(1,2)(3) << endl;
    cout << fsum(1)(2,3) << endl;
    cout << curry(sum)(1)(2)(3) << endl;
    mkFF2<const int,int,int,add> fadd;
    cout << fadd(1)(2) << endl;
    cout << g(fadd(3),5) << endl;
}

```

Example 9 shows examples of how to use the generator classes and the `curry` function. Notice that the C++ function `g` takes an argument of type

FUNC_01<A1,R,OP>. Hence, a detailed type checking can be performed. Also notice that there is a partial application of `fadd` in the call to `g` at the end of `main` and that `curry(sum)` and `fsum` are of different type.

Up to this point everything looks fine, but still will not work. As mentioned above we want to force functions to be curried to have parameters of constant type. Now consider template instantiation of `MakeF2<const int,int,int,add>::f`. This will instantiate `FUNC_02<A1,A2,R,OP>` with `A1=const int`, `A2=int`, `R=int`. This instantiation is not possible (e.g. will produce compiler errors), because instantiation of `FUNC_02::PAF` will lead to `const const int a1`, what is not allowed in C++. To avoid these errors we introduce the following traits class to compute appropriate argument types:

```
template <class T>
struct FAT {
    typedef T      user_t;
    typedef const T const_t;
};
```

and some specialization for all type modifiers:

```
template <class T>
struct FAT<T&> {
    typedef T&      user_t;
    typedef const T& const_t;
};
```

```
template <class T>
struct FAT<const T&> {
    typedef const T& user_t;
    typedef const T& const_t;
};
...
```

FAT can be thought of as a function taking a type and offering a normal and a const version of this type. By the use of the FAT-classes within our generator classes, we can prevent the user from passing functions with non-constant arguments to it. Consider the modified version of `MakeF2`:

```
template <class A1,class A2,class R,R (*cfunc)(A1,A2)>
struct MakeF2 {
    ...
    inline R operator()(typename FAT<A1>::const_t _a1,
                       typename FAT<A2>::user_t _a2) const {
        return cfunc(_a1,_a2);
    }
    typedef FUNC_02<typename FAT<A1>::const_t,A2,R,func_t> funcfunc_t;
    ...
};
```

If a user tries to instantiate `MakeF2<int &,int,int,unsecure_mult>`, `MakeF2`'s `operator operator()` becomes

```
int operator()(typename FAT<int &>::const_t _a1,
               typename FAT<int>::user_t _a2),
```

that is

```
int operator()(const int & _a1,int _a2).
```

Since `unsecure_mult` takes a non-constant type as first argument, the `const` qualifier introduced by `operator()` gets dropped when calling `unsecure_mult` and a compiler should produce an error message.

To ensure that this mechanism gets invoked every time a user wants to get a curried function, we also have to change all f-functors, `MkOp` functors, `mkop` functions, and curry functions. Example 10 shows the final f-functor class for the representation of binary function.

Example 10 Class template for an f-functor - final version

```
template <class A1,class A2,class R,class F>
struct FUNC_02 {
    typedef FUNC_02<A1,A2,R,F> funcfunc_t;
    const F & f;
    FUNC_02( const F & _f ) : f(_f) {}
    inline R operator()(typename FAT<A1>::const_t _a1,
                       typename FAT<A2>::user_t _a2) const {
        return f(_a1,_a2);
    }
    struct PAF {
        typename FAT<A1>::const_t a1;
        const funcfunc_t & f;
        PAF(const funcfunc_t & func,
            typename FAT<A1>::const_t _a1) : f(func),a1(_a1) { }
        inline R operator()(typename FAT<A2>::user_t _a2) const {
            return f(a1,_a2);
        }
    };
    typedef FUNC_01<A2,R,PAF> f1_type;
    inline const f1_type operator()(
        typename FAT<A1>::const_t _a1) const {
        return f1_type( PAF(*this,_a1) );
    }
};
```

3.3 Performance

With a highly optimizing C++ compiler (such as Kuck and Associates KAI C++), currying and partial application do not result in a runtime penalty and extra storage is only needed under certain circumstances. Every version of `operator()` is declared `inline` and consists of only one line of code. This way even a "simple" compiler should not generate a call to this operator, but should produce real inlined code. Moreover, inlining does not only prevent a call but also prevents possibly arising calls to copy constructors of the arguments of `operator()`. Therefore, we call our method a compile-time mechanism.

As mentioned earlier, each f-functor has a functor and therefore holds a constant reference to it. By declaring the reference `const` we enforce evaluation during compile-time and no storage has to be wasted, if all functions, a call to `operator()` depends on, have been declared `inline`. If `operator()` depends on inlined code only, there is even no need to waste storage for the arguments in a pa-functor. In any other cases the compiler generates instances of the appropriate class types during compile-time. Since every involved class stores its member variables as `const` values, evaluation during compile-time is enforced and no extra constructor call is needed.

4 Application to Parallel Programming

A typical application of partial application and higher order functions to parallel programming is the design of algorithmic skeletons. Algorithmic skeletons are algorithmic abstractions common to a series of applications, which can be implemented in parallel [7]. They support both types of parallelism: data-parallelism (e.g. `map`, `divide_and_conquer`) and task-parallelism (e.g. `farm`) [8]. Most skeletons are higher order functions and use polymorphic types, hence most languages with skeletons build upon a functional host. As shown in [9], programs written in these languages are still 5 to 10 times slower than their low-level counterparts, e.g. parallel C. The SKIL project [6] has shown that there is only a small performance penalty in the use of algorithmic skeletons, if they are integrated into an imperative host language.

Our approach offers an easy integration of algorithmic skeletons into C++ without the need of a special compiler. The achievable results should be comparable to those obtained by the SKIL project, since all functional abstractions are resolved during compile-time.

More possible applications to parallel programming are presented in the next section.

5 Conclusion and Future Work

We have presented a method on how to curry an existing C++ function. Our implementation is based on C++, hence no special preprocessor or compiler is

needed. Since currying and partial application have been realized as compile-time mechanisms, there is no overhead to be expected.

Future work is possible in several directions. In the field of algorithmic skeletons there is the need of distributed data structures, which cover any necessary communication and synchronization [6, 10]. To be portable, such a data structure should be applicable in different environments (SMP, cluster of SMP's, Message Passing environments) and it may be interesting to investigate how the functional concepts may help to build them up.

It would be nice to have variables of f-functor type, which get bound to a specific functor during runtime. Hence, only the signature of such a **dynamic f-functor** should be given during compile-time and the functor it is based on may then be changed during runtime. C++ supports such runtime mechanisms by virtual functions. Having appropriate "combiners" at hand dynamic f-functors offer the possibility to yield new functions during runtime. Moreover, if dynamic f-functors depend on f-functors (which are generated during compile-time) only, they depend on a known set of functions. Their implementation may acknowledge this by storing an appropriate building rule. Now consider heterogeneous parallel environments. Dynamic f-functors may offer the possibility to transfer and change data as well as functions. This may not only increase expressiveness, but also offer a new point of view in parallel computing.

Using f-functors with lazy evaluation may shift C++ from a strict to a combination of strict / non-strict language. Moreover, lazy evaluation is in the discussion to increase the performance in several fields of HPC. In combination with dynamic f-functors lazy evaluation offers an MIMD (**M**ultiple **I**nstructions **M**ultiple **D**ata) version of map and there is no need for a **farm**.

Having in mind that f-functors are classes (not only functions), several additional properties may be associated with them. Consider a function which can tell its computational complexity (depending on the data size) in a parallel environments with load balancing, or a function capable of providing its own derivative.

References

- [1] P. Thiemann. Grundlagen der funktionalen Programmierung. Teubner 1994
- [2] G.H. Botorog and H.Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming in Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5), IEEE Computer Society Press, 1996, pp. 243-252.
- [3] B. Stroustrup. The C++ Language, Third Edition. Addison Wesley 1997

- [4] U. Breymann. Designing Components With the C++ Stl: A New Approach to Programming. Addison Wesley 1997
- [5] M.H. Austern. Generic Programming and the STL. Addison Wesley 1998
- [6] G. H. Botorog, H. Kuchen. Efficient Parallel Programming with Algorithmic Skeletons in Proceedings of EuroPar '96, Vol. 1, LNCS 1123, Springer, 1996, pp. 718-731.
- [7] M.I. Cole. Algorithmic Skeletons: Structured Management of parallel computation . MIT Press 1989
- [8] J. Darlington, A. J. Field, P. G. Harrison et al. Parallel Processing Using Skeletons. Proceedings of PARLE '93, LNCS 694, Springer 1993
- [9] H. Kuchen, R. Plasmeijer, H. Stoltze. Distributed Implementation of a Data Parallel Functional Language. Proceedings of the 6th International Conference on Parallel Architectures and Languages Europe (PARLE'94), LNCS 817, Springer , 1994.
- [10] J. Nieplocha, RJ Harrison, and RJ Littlefield: Global Arrays: A portable 'shared-memory' programming model for distributed memory computers. Proceedings of the Supercomputing'94, 1994
- [11] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti and M. Vanneschi.P3L: A structured high level programming language and its structured support in Concurrency: Practice and Experience vol.7 n.3, May 1995