Function Composition and Return Type Deduction

Anthony Williams

28th February 2002

Abstract

A core feature of functional programming is the ability to build new functions from old ones, such as creating a new function $h(x)$, such that $h(x) \equiv f(g(x))$, given functions $f(x)$ and $g(x)$. This paper describes a way of providing such a facility in C++, that allows arbitrarily complex functions to be built up in such a way.

This paper also shows a mechanism for automatic return type deduction, to avoid the need for the `result_type` **`typedef`**s traditionally found in functor classes.

Part I

Introduction

1 Function Composition

The purpose of function composition is to enable the building of complex functions from basic building blocks *at the point of use*, as opposed to defining a separate function for the complex task. Consider using `std::transform` to store double the value of each element in one container in another, as in listing 1 (avoiding `std::back_inserter` for simplicity) — we can use `std::bind1st` and `std::multiplies` to create a new function object "multiply by 2", which isn't explicitly declared anywhere. However, if we now want to store double the value plus one, then we can't do it in one step anymore — we have to either explicitly define a "multiply by 2 and add 1" function, or use `std::transform` twice, once to multiply by 2, and once to add 1 (listing 2). Function composition allows us to pull the operations back together into a single step, with an implicitly declared function object (listing 3). For cases like this where the expression could be built using operators (did it not have to be a functor) we can hide the use of `compose` using operator overloading, and write `2*_1+1` instead of `compose(std::plus<double>(),compose(std::multiplies<double>(),2,_1),1)`

```
void f(const std::vector<double>& src,std::vector<double>& dest)
{
    dest.resize(src.size());
    std::transform(src.begin(),src.end(),dest.begin(),
        std::bind1st(std::multiplies<double>(),2));
}
```

Listing 1: Using `std::transform` to store double the values of a container

The `compose` function works very much like `std::bind1st`, in that it takes a function and some argument values, and produces a new function object that accepts a different number of arguments. However, if one of the arguments is itself a call to `compose`, then this second composed function is stored within the outer composed function, and is called whenever the outer function is called, possibly with arguments that depend on the supplied arguments to the outer function. Thus the functor generated by the `compose` invocation in listing 3 is directly equivalent in behaviour to the functor class of listing 4.

```
void f(const std::vector<double>& src,std::vector<double>& dest)
{
    dest.resize(src.size());
    std::transform(src.begin(),src.end(),dest.begin(),
        std::bind1st(std::multiplies<double>(),2));
    std::transform(dest.begin(),dest.end(),dest.begin(),
        std::bind1st(std::plus<double>(),1));
}

double doubleAndAddOne(double i)
{
    return 2.0*i+1.0;
}
void f2(const std::vector<double>& src,std::vector<double>& dest)
{
    dest.resize(src.size());
    std::transform(src.begin(),src.end(),dest.begin(),
        doubleAndAddOne);
}
```

Listing 2: Storing $2x + 1$ using `std::transform` twice, and with an explicit function.

```
void f(const std::vector<double>& src,std::vector<double>& dest)
{
    dest.resize(src.size());
    std::transform(src.begin(),src.end(),dest.begin(),
        compose(std::plus<double>(),compose(std::multiplies<double>(),2,_1),1));
}
```

Listing 3: Storing $2x + 1$ using `compose`.

This can be extended to any nesting level, and any number of function arguments; e.g. `compose(std::plus<double>(), compose(std::multiplies<double>(),_1,_1), compose(std::multiplies<double>(), compose(std::multiplies<double>(),_1,2),_2))` gives $x^2 + 2xy$ where $x \equiv$ `_1` and $y \equiv$ `_2`, and can be called as `f(x,y)` where `f` is the result of the call to `compose` — the generated functor can then be used as an argument of the binary form of `std::transform`, or any other algorithm that requires a binary functor.

Basically, when using `compose` to build a composed function, wherever there is a function call `f(params)` replace it with a call to `compose(f,params)`. If one of the function parameters is to be a parameter of the composed function, replace it with one of `_1`, `_2`, `_3`, etc. to indicate where in the final parameter list it should lie; if one of the parameters is a call to another function, the parameters of which are to be parameters of the final composed function, again replace that with a call to `compose`. The result is a function object which, when called, calls the composed functions in the appropriate order with the specified parameters.

```
struct __functor__
{
    std::plus<double> Plus;
    std::multiplies<double> Mult;

    double operator()(double _1) const
    {
        return Plus(Mult(2,_1),1);
    }
};
```

Listing 4: A functor equivalent to that generated by the `compose` call in listing 3

For example, to create a composed function that takes 3 arguments — x, y and z — for the complex expression f(g(x),h(x,y,i(z))), given the basic functions f, g, h, and i, we can use compose(f,compose(g,_1),compose(h,_1,_2,compose(i,_3))).

2 Return Type Deduction

One thing to notice about the above uses of std::multiplies and std::plus is that they are templates, and require the type of the arguments and return type to be specified. For overloaded functions such as std::sin and std::cos, we have to cast them to a specific type in order to select which overload by specifying the argument and return types. This clutters the code with explicit specification of types, which can be deduced from the rest of the expression. Using the return-type deduction mechanism described later, we can write classes with template members that perform the required operation, regardless of the types, as in listing 5. For binary operators, such as addition, this can extend the applicability of the functor — the Plus functor will handle pointer arithmetic as well as normal addition, something that std::plus cannot support, as it requires both arguments to be the same type as the return type.

```
struct Plus
{
    template<typename T,typename U>
    deduced_return_type operator()(const T& t,const U& u) const
    {
        return t+u;
    }
};

struct Sin
{
    template<typename T>
    deduced_return_type operator()(const T& t) const
    {
        return std::sin(t);
    }
};

void write(double); // write out a value somewhere

std::for_each(vd.begin(),vd.end(),
              compose(write,compose(Sin(),_1)));
std::for_each(vd.begin(),vd.end(),
              compose(write,compose(Plus(),_1,2)));
```

Listing 5: Functor classes with template members

As well as simplifying compose expressions by avoiding the need for casts and explicit specification of template parameters, such functors can also be used with operator overloading to write simple lambda functions in-line, as described in section 10, when making a call to an algorithm that takes a functor parameter:

```
std::for_each(vd.begin(),vd.end(),
              compose(write,_1+2*compose(Sin,_1*_1)));
```

Part II

Implementation

3 Argument Sets

When writing a functor class to hold a composed function, there are two sets of arguments to consider — one provided as part of the call to `compose`, and another given at the point where the composed function is actually called. These must be combined to give the actual arguments used to call the underlying function.

To avoid a mountain of classes, template parameters, and member variables, these argument sets are stored as instances of the `ArgSetBase` class template, or the `ArgSet0` class for an empty argument set. `ArgSetBase` is used as a recursive template, and works like a LISP list — the first template parameter is the type of the data for the current node, and the second is either `ArgSet0` to indicate the end of the list, or another instantiation of `ArgSetBase`, to indicate further elements in the list. The class then contains a **`const`** instance of the data type, and derives from the specified base type; it also provides **`typedef`**s `Arg` and `Base` for these types. See listing 6 for the implementation. To ease construction of such argument sets by the user, the class templates `ArgSet1` ...`ArgSet10` are provided (listing 7). These inherit from appropriate instances of `ArgSetBase`, and consequently behave identically, as the **`typedef`**s and member variable are **`public`**. We can also provide `ArgSetBaseRef` and `ArgSetRef`n templates, which hold references to their data, rather than containing it directly.

```
struct ArgSet0
{};

template<typename Arg_,typename Base_>
struct ArgSetBase:
    public Base_
{
    typedef Arg_ Arg;
    typedef Base_ Base;

    const Arg arg;

    ArgSetBase(const Arg& arg_,const Base& base_):
        Base(base_),arg(arg_)
    {}
};
```

Listing 6: Argument Sets

Given an argument set, we need a way of extracting the values and their types. For the first element, this is easy — just use the member **`typedef`** `Arg` and the data member `arg` — but for subsequent elements it is more tricky, as these members are hidden by those of the first element. Therefore, we use the `GetASEntry` class template to extract these for us, as in listing 8. In order to get the n-th entry, we must get the $n-1$-th entry of the `Base`; we then specialize the class for the first entry, in order to terminate the recursive expansion. From the users point of view, `GetASEntry<SomeArgSet,n>::Type` is the type of the n-th entry of `SomeArgSet`, and `GetASEntry<SomeArgSet,n>::getVal(as)` returns a reference to the n-th value of `as` (which must be of type `SomeArgSet`).

Another class template can be used to determine the number of arguments in an argument set. This is the `CountArgs` template shown in listing 9; again, it is a recursive template, this time terminated by being instantiated with `ArgSet0`, the empty argument set, which contains zero entries. The count is returned in the member `result`, so `CountArgs<SomeArgSet>::result` is the number of arguments in the argument set type `SomeArgSet`.

```
template<typename Arg1>
struct ArgSet1:
    public ArgSetBase<Arg1,ArgSet0>
{
    ArgSet1(const Arg1& a1):
        ArgSetBase<Arg1,ArgSet0>(a1,Base())
    {}
};

template<typename Arg1,typename Arg2>
struct ArgSet2:
    public ArgSetBase<Arg1,ArgSet1<Arg2> >
{
    ArgSet2(const Arg1& a1,const Arg2& a2):
        ArgSetBase<Arg1,ArgSet1<Arg2> >(a1,Base(a2))
    {}
};

template<typename Arg1,typename Arg2,typename Arg3>
struct ArgSet3:
    public ArgSetBase<Arg1,ArgSet2<Arg2,Arg3> >
{
    ArgSet3(const Arg1& a1,const Arg2& a2,const Arg3& a3):
        ArgSetBase<Arg1,ArgSet2<Arg2,Arg3> >(a1,Base(a2,a3))
    {}
};

// ...
```

Listing 7: Class templates for easier creation of argument sets

4 Unbound Parameters

Argument sets can be used for binding sets of values, but when composing functions we also need to leave some function parameters unbound, to be provided by the final caller. This is the purpose of the `ParamType` class template (listing 10) — it is a place holder for use in an argument set to indicate that the corresponding entry is to remain unbound. Not only that, but the template parameter can be used to indicate which of the arguments provided by the final caller is to be bound — `ParamType<1>` would indicate the first parameter from the final call, whereas `ParamType<7>` would indicate the seventh parameter. The objects `_1`, `_2`, etc. are instances of the `ParamType` template with the specified number as the template parameter, to ease typing, so `_4` is equivalent to `ParamType<4>()`.

Once we have argument sets containing such place holders, we need two things; we need to be able to determine the minimum number of arguments required in the final call to satisfy these place holders, and we need to be able to create a final bound argument set from an argument set with place holders, and the arguments provided by the final call, where the place holders are substituted with their actual values. The former is done through the use of the `MinArgs` and `NumArgs` class templates, as well as the `CT::Max` class template; the latter through the use of the `transformArgSet` and `transformArg` function templates, and the `TransformArgSetHelper` class template.

`NumArgs<T>::result` is 0 for normal types, and the index count `i` for place holder types `ParamType<i>`. This is used so that `MinArgs<SomeArgSet>::result` is the minimum number of arguments in the final call needed to match all the place holders — `MinArgs<SomeArgSet>::result` is defined to be the maximum of `NumArgs<SomeArgSet::Arg>` and `MinArgs<SomeArgSet>::Base`, using `CT::Max` to provide the compile-time `max` functionality. As for `CountArgs`, we specialize for `ArgSet0` to terminate the recursive template.

The function template `transformArgSet` defers to `TransformArgSetHelper< ArgSet, Call-`

```
template<typename ArgSet,unsigned index>
struct GetASEntry
{
    typedef typename GetASEntry<typename ArgSet::Base,index-1>::Type Type;
    static const Type& getVal(const ArgSet& as)
    {
        return GetASEntry<typename ArgSet::Base,index-1>::getVal(as);
    }
};

template<typename ArgSet>
struct GetASEntry<ArgSet,1>
{
    typedef typename ArgSet::Arg Type;
    static const Type& getVal(const ArgSet& as)
    {
        return as.arg;
    }
};
```

Listing 8: The `GetASEntry` class template

```
template<typename ArgSet>
struct CountArgs
{
    static const unsigned result=1+CountArgs<typename ArgSet::Base>::result;
};

template<>
struct CountArgs<ArgSet0>
{
    static const unsigned result=0;
};
```

Listing 9: The `CountArgs` class template

`ingArgSet >::doTransform()`. This function calls `transformArg` to transform the current argument, and then uses `TransformArgSet< ArgSet::Base, CallingArgSet >::doTransform()` to transform the remainder of the arguments. The result is a new argument set in which all the place holders have been replaced with the corresponding entry from the final calling argument set.

`transformArg` is a function template that takes two parameters — the argument to transform, and the final calling argument set. By default it just returns the argument to transform, as this is appropriate for bound parameters. However, it is also overloaded for the case where the argument to transform is an instance of the `ParamType` template, in which case it returns the i-th entry from the final calling argument set (where i is the index parameter of the `ParamType` instance).

`transformArg` is also where nested `compose`-d functions are handled — if the argument to transform is a composed function, then rather than just returning the function, `transformArg` invokes the function with the calling argument set and returns the result. The composed function then invokes `transformArg` for all its bound and unbound parameters, as if it was called directly, thereby supporting nesting to any level.

```
template<unsigned index>
struct ParamType
{};
```

Listing 10: The `ParamType` class template

5 Reference arguments

Reference arguments are handled by the `byRef` function templates and the `RefHolder` class template. They can either be specified at `compose` time, or at call time, or both. When specified at `compose` time, they can either be a reference to a real variable (e.g. `byRef(someVariable)`), or a reference to an unbound parameter (e.g. `byRef(_1)`). References to unbound parameters will force the actual argument supplied at call time to be an lvalue, and a reference to that object will be passed to the underlying function.

Reference arguments can also be explicitly specified at call time, again by using `byRef`, in which case a reference to the argument of `byRef` will be passed to the underlying function. If the original `compose` specified a given unbound argument was `byRef`, then repeating `byRef` at call time is OK, but unnecessary — this is achieved by overloading `byRef`, so `byRef(byRef(some_expression))` is equivalent to `byRef(some_expression)`.

Marking reference arguments at compose time is important, as it enables the use of functions that require reference parameters with algorithms such as `std::for_each` which do not know about `byRef`. It also enables the fixed parameters to be references.

Marking reference arguments at call time is also important, as it enables the caller to specify pass-by-reference when they believe it to be optimal or appropriate. It also enables the use of the functor as the first argument to `compose`, and the specification of reference arguments for this second composed functor.

Unbound parameters marked as `byRef` at `compose` time cannot bind directly to rvalues. However, temporaries can be passed to functors accepting **`const`** references, by using `byRef` at the point of call, as shown in listing 11.

```
template<typename Func>
void someAlgorithm(const Func& f,int x,int y)
{
    // use byRef to get a const reference to the temporary
    f(byRef(x+y));
    // no need for byRef with an lvalue
    f(x);
}

void someFuncTakingConstRefArg(const int&);

void someOtherFunc()
{
    // pass a compose-d functor requiring a reference
    // to some algorithm
    someAlgorithm(compose(someFuncTakingRefArg,byRef(_1)),1,5);
}
```

Listing 11: Binding temporaries with `byRef`

6 Implementation of the function call operator

Implementing the function call operator, `operator()`, is a tricky business. Firstly, we don't know in advance how many arguments the bound function is designed to accept, so we must provide multiple overloads that accept different numbers of arguments. Secondly, we wish some of the unbound parameters to be passed as references, where specified in the argument set, without restricting the use of rvalues for the remaining arguments. This requires that the signature of the function call operator function vary depending on the argument set, and which, if any, of the arguments are to be references. To simplify things, we shall specify that the task of the function call operator is merely to build an appropriate argument set for the supplied arguments, and pass that to another function which will actually perform the call. This can then be achieved by the functor class inheriting from an instance of the class template `FunctionCallOperatorImpl` which defines all the appropriate overloads. The primary definition of the `FunctionCallOperatorImpl` template

can assume that all parameters can be rvalues, and consequently only provide overloads of the function call operator on that basis. The template can then be specialized for each of the possible combinations of lvalue and rvalue arguments. For a functor to accept up to 10 arguments, this leads to an astounding 1023 distinct specializations of the template, as well as the primary template! Fortunately, they can be automatically generated by a simple program, as the building of argument sets is quite straight-forward.

The `FunctionCallOperatorImpl` class template takes two parameters. The first (`FuncCaller`) is the type of a helper struct that can be used to determine the return type via the member class template `Helper`, and what the type of the derived class implementing the `call` function is. The second (`NCRefPosSet`) is a recursive list template indicating which function call arguments must be lvalues, and which can be rvalues, that determines which specialization of `FunctionCallOperatorImpl` to use. The functor class is publicly derived from the appropriate instantiation of `FunctionCallOperatorImpl`, and must implement the `call` function. The `DerivedType` member **`typedef`** of the **`struct`** passed as the first argument of `FunctionCallOperatorImpl` must be a **`const`** reference to this derived class.

The general form of each overload of `operator()` is as shown in listing 12. This is from the specialization of `FunctionCallOperatorImpl` where the first argument must be a reference, so the `RefType` class template is used to ensure that arguments supplied as `byRef` are passed correctly. The key features are the use of the `Helper` member template of the `FuncCaller` class template parameter to deduce the return type, and the case of `*this` to the `DerivedType` member **`typedef`** of `FuncCaller` before invoking the `call` member function. Also, `ArgSetRef2` is used to avoid unnecessary copying of the supplied arguments when invoking `call`.

```
template<typename A1,typename A2>
typename FuncCaller::template Helper<
             ArgSetRef2<
                 typename RefType<A1>::Type,
                 A2
             >
         >::ReturnType
operator()(A1& a1,const A2& a2) const
{
    typedef ArgSetRef2<
                typename RefType<A1>::Type,
                A2
            > AS;
    return static_cast<typename FuncCaller::DerivedType>(*this).call(AS(a1,a2));
}
```

Listing 12: The implementation of one overload of `operator()` from `FunctionCallOperatorImpl`

With all the appropriate overloads of `operator()` implemented to forward to our single template function `call`, we now need to focus on how that is implemented — by delegating the task to yet another template; in this case, the `callFunc` **`static`** member function of the `CallerHelper` class template.

This class template has 4 parameters; the functor type, the bound argument set type, the calling argument set type, and the number of arguments in the bound argument set, which is automatically deduced by default, using the `CountArgs` class template. It is then specialized for each number of arguments in the bound argument set, to implement the actual call. Each specialization contains a single **`static`** member function `callFunc`, which extracts the appropriate arguments from the argument sets and invokes the functor; the specialization for 2 arguments is shown in listing 13. This template can then be specialized by the user for specific functor types, if they need to be called in a non-standard way.

7 Implementing return type deduction

Given that we now have an implementation of `operator()` which calls our contained functor of type `Func`, with an argument set of type `ArgSet`, we need to determine the return type. This is the job of the `RTHelper` template (listing 14), which contains the **`typedef`** `Type` that is the deduced return type. The purpose of this

```
template<typename Func,typename ArgSet,typename CAS>
struct CallerHelper<Func,ArgSet,CAS,2>
{
    typedef typename TransformArgSetHelper<ArgSet,CAS>::Type TransformedArgSet;
    typedef typename RTHelper<Func,TransformedArgSet>::Type Type;
    static Type callFunc(const Func& func,const ArgSet& as,const CAS& cas)
    {
        return func(transformArg(GetASEntry<ArgSet,1>::getVal(as),cas),
                    transformArg(GetASEntry<ArgSet,2>::getVal(as),cas));
    }
};
```

Listing 13: The `CallerHelper` specialization for 2 arguments

class template is to use all the information available from the functor class and the argument set with which it is to be called. It can be specialized (as we shall see later) for particular types of functor, in order to use information available from that functor that is not available in general. In the absence of a specialization, it is the task of the primary template to make a reasonable attempt at deducing the return type.

In the primary template, we use the compile-time **if** facility `CT::If` to check whether or not the bound function type provides a `ResultHelper` member template. If the member template is present, it is used to determine the result type, through the `ExtractResultFromHelper` class template; otherwise we check for the existence of a `result_type` member **typedef**. The `ExtractResultType` class template provides access to this **typedef**. If neither the `ResultHelper` template or the `result_type` **typedef** are present, then the return type is deduced more directly, using the `DeduceReturnType` class template. The `HasResultType` class template is written so that `result` is **true** for functors derived from the standard library base class templates — `std::unary_function` and `std::binary_function` — and false for all others. It can then be specialized for other functors by the user. The `ResultHelper` member template is an entirely new idea, so `HasResultHelper::result` defaults to **false** for all types, and is only **true** if the user specializes it.

```
template<typename Func,typename ArgSet>
struct RTHelper
{
    typedef typename CT::If<HasResultHelper<Func>::result,
                            ExtractResultFromHelper<Func,ArgSet>,
                            typename CT::If<HasResultType<Func>::result,
                                            ExtractResultType<Func>,
                                            DeduceReturnType<Func,ArgSet>
                                           >::Type
                           >::Type::Type Type;
};
```

Listing 14: The `RTHelper` primary template

If present, the `ResultHelper` member template should accept any combination of arguments supported by the functor (including no arguments) through use of default parameters where there are different numbers of valid arguments, and each valid instantation should contain a member **typedef** `Type` for the result type.

The `RTHelper` class template is then partially specialized for pointers-to-functions taking various numbers of arguments, making the return type directly available (listing 15). It is also partially specialized for functors which are themselves the result of function composition using this library, in order to use the return type of the underlying function of this nested composite, rather than doubling up on the return type deduction.

For those functor classes for which the return type is not available through a `result_type` **typedef** or through the use of a `ResultHelper` template, the class template `DeduceReturnType` is used to obtain the return type. This class is specialized for the various numbers of arguments in the `ArgSet` parameter, through the use of the third template parameter, which defaults to counting the number of arguments in the argset using the `CountArgs` class template.

```
template<typename RT,typename ArgSet>
struct RTHelper<RT(*)(),ArgSet>
{
    typedef RT Type;
};

template<typename RT,
         typename A1,
         typename ArgSet>
struct RTHelper<RT(*)(A1),ArgSet>
{
    typedef RT Type;
};

template<typename RT,
         typename A1,
         typename A2,
         typename ArgSet>
struct RTHelper<RT(*)(A1,A2),ArgSet>
{
    typedef RT Type;
};
```

Listing 15: Partial specializations of the `RTHelper` class template for pointers-to-functions

These specializations are the last resort for deducing the return type, and ideally would utilize the much-requested **`typeof`** operator. However, since such an operator is not yet standard, instead we have to rely on either non-standard extensions for particular compilers, or a restricted implementation that relies only on standard facilities. Each specialization therefore uses the `EXPR_TYPE` **`typeof`**-like facility (described in [5]) to deduce the type of a synthesized function call expression with the appropriate number and types of arguments (obtained from the `ArgSet`). This mechanism is thus subject to the shortcomings of the `EXPR_TYPE` mechanism — namely that each deducible type must be registered explicitly by the user, and **`void`** cannot be deduced. All basic types and some compound types are registered in advance. For the case of functors that do not provide a `result_type` **`typedef`**, and return **`void`**, the `voidReturn` function template is provided. This wraps the actual functor inside another functor class that ignores the return value of the contained functor, and instead returns an object of type `Void`, which is detectable by `EXPR_TYPE`.

Another restriction on deduced return types is that **`const`**-qualified reference types cannot be deduced — the library will assume they are value returns of the referred-to type. This is because **`const`**-qualified types and rvalues of the same type (whether **`const`**-qualified or not) are indistinguishable under the type deduction mechanism, since rvalues can bind to **`const`**-qualified references. Reference types that are not **`const`**-qualified are deduced without problems using the `RValueLValueType` class template extension to the `EXPR_TYPE` type deduction mechanism (listing 16) — only lvalues can be deduced to be non-**`const`** types, so the `Type` typedef in this class template is a reference for non-**`const`** types, and the original type otherwise.

8 The function call operator with no arguments

Implementing the function call operator with no arguments is marginally harder than implementing the function call operator with arguments, because it must be a normal member function rather than a member function template, and the consequence of this is that the declaration must be well-formed, even if the composed functor cannot be called without arguments. To achieve this, the `ComposedHelper` class template is partially specialized for empty calling argument sets, to only try and perform type deduction if the bound argument set doesn't include any unbound parameters, and to return **`void`** otherwise. This specialization uses the compile-time **`if`** facility `CT::If` along with the `MinArgs` template to perform this check (listing 18).

It is still an error to actually instantiate `operator()` with no arguments if there are unbound parameters in

```
template<typename T>
struct RValueLValueType
{
    typedef T& Type;
};

template<typename T>
struct RValueLValueType<const T>
{
    typedef const T Type;
};

template<typename T>
struct RValueLValueType<T&>
{
    typedef T& Type;
};
```

Listing 16: The `RValueLValueType` class template

```
template<typename Func,typename ArgSet,typename CAS>
struct ComposedHelper
{
    typedef typename TransformArgSetHelper<
        ArgSet,
        CAS
    >::Type TransformedArgSet;

    typedef typename CallerHelper<
        Func,
        TransformedArgSet
    >::Type Type;
};
```

Listing 17: The `ComposedHelper` primary template

the argument set, so setting the return type to **`void`** in such a case causes no errors that wouldn't otherwise be present, and enables the functor to compile.

9 Extending `compose` to handle member functions

`compose` can be extended to handle member functions without requiring the use of `std::mem_fun` or similar, by specializing the `CallerHelper` and `RTHelper` class templates. Specializing the `RTHelper` class template is straightforward, and is along the same lines as specializing it for normal function pointers, as shown in listing 19.

On the other hand, specialization of `CallerHelper` is a fraction more complicated for precisely the reason it is necessary — you can't call a member function the same way you call a normal function, as member functions need an "object" to apply to. There are two ways of specifying the object; either as a pointer to the object or as a reference to the object. In addition, we have to deal with **`const`** member functions, as well as non-**`const`** ones (and possibly **`volatile`** or **`const volatile`** member functions, though these are less common). We resolve this by separating concerns — we specialize `CallerHelper` for the different varieties of member function, and delegate the actual call to a separate template (`MemFunCallerHelper`) to avoid duplication of the function call code. `MemFunCallerHelper` depends in turn on the `GetObjectHelper` class template to return the object to which to apply the member function; `GetObjectHelper` is specialized to return a

```
template<typename Func,typename ArgSet>
struct ComposedHelper<Func,ArgSet,ArgSet0>
{
    typedef typename CT::If<(MinArgs<ArgSet>::result==0),
                            TransformArgSetHelper<ArgSet,ArgSet0>,
                            CT::Identity<void> >::Type::Type TransformedArgSet;
    typedef typename CT::If<(MinArgs<ArgSet>::result==0),
                            RTHelper<Func,TransformedArgSet>,
                            CT::Identity<void> >::Type::Type Type;

};
```

Listing 18: The specialization of `ComposedHelper` for empty calling argument sets

```
template<typename RT,typename Object,
         typename A1,
         typename A2,
         typename ArgSet>
struct RTHelper<RT(Object::*)(A1,A2),ArgSet>
{
    typedef RT Type;
};
```

Listing 19: Specialization of `RTHelper` for a non-**`const`** member function with 2 arguments

reference to the object, even if a pointer is provided. We can identify which is provided by examining the type of the parameter that is to be bound to the object for the member function — if it is the same as, or a class publicly derived from, the object type for the member function, then the we have a direct reference to the object. If, on the other hand, the parameter is not an object of the appropriate type, or a derived class, then it must be a pointer (or smart pointer/iterator) to the object. As an added complication, references bound through the use of `byRef` must also be handled correctly, by removing such wrappers before checking the type.

To check whether or not the supplied argument is publicly derived from the required object type, we use a helper class template `IsPubliclyDerivedFrom`, which strips any references or `byRef` uses from the supplied types before checking. The actual check is done in the way advocated by Andrei Alexandrescu in [1] when he implements the `SUPERSUBCLASS` macro — check whether a pointer to the supplied type can be implicitly converted to a pointer to the required type using **`sizeof`** and the return types of overloaded helper functions (as in listing 20).

Specializations of `CallerHelper` now look like that shown in listing 21.

10 Using operator overloading to create lambda functions

`compose` can be used with operator overloading to enable the writing of normal expressions involving operators that result in composed functions. For example, `_1+3` can be made equivalent to `compose(OpBinPlus(),_1,3)`, where `OpBinPlus` is a functor class that returns the sum of its (two) parameters. This part is relatively simple, as shown in listing 22, though it does require multiple overloads for the various possibilities:

- Unbound parameter + Value
- Value + Unbound parameter
- Unbound Parameter + Unbound Parameter
- Composed function + Value

```
template<typename T>
struct DerivedHelper
{
    typedef char Small;
    struct Big
    {
        Small dummy[2];
    };

    static Small check(const volatile T* p);
    static Big check(...);
};

template<typename T,typename U>
struct IsPubliclyDerivedFrom
{
    typedef typename RemoveRef<T>::Type TNoRef;
    typedef typename RemoveRef<U>::Type UNoRef;

    static const bool result=
        sizeof(DerivedHelper<UNoRef>::check(
                    ExprType::makeT<const volatile TNoRef*>()))
        ==sizeof(DerivedHelper<UNoRef>::Small);
};
```

Listing 20: Implementing `IsPubliclyDerivedFrom` to handle `byRef` types

- Value + Composed function
- Composed function + Composed function
- Composed function + Unbound parameter
- Unbound parameter + Composed function

There are even more possibilities if you consider that the parameters may also be passed `byRef`. For ease of implementation, the supplied code ducks the issue, and relies on argument-dependent lookup to find the overloaded operators. However, this is not 100% reliable, and may yield the wrong operator under some circumstances.

Writing the `OpBinPlus` functor is a fraction harder — the return type of **`operator()`** is unknown, in general. This is where `EXPR_TYPE` is useful — the type of the additive expression can be deduced directly, provided it has

```
template<typename Object,typename RT,
         typename A1,
         typename A2,
         typename ArgSet,typename CAS>
struct CallerHelper<RT (Object::*)(A1,A2),ArgSet,CAS,3>
{
    typedef RT (Object::*Func)(A1,A2);

    static RT callFunc(const Func& func,const ArgSet& as,const CAS& cas)
    {
        return MemFunCallerHelper<RT,Object,Func,ArgSet,CAS,2>::call(func,as,cas);
    }
};
```

Listing 21: One of the specializations of `CallerHelper` for member functions.

```
template<unsigned i,typename U>
ComposedFunc<OpBinPlus,ArgSet2<ParamType<i>,U> > operator+(const ParamType<i>& t,const U&
{
    return compose(OpBinPlus(),t,u);
}
```

Listing 22: An example **operator+** to enable expressions like _1+someValue

been registered with the EXPR_TYPE mechanism. This deduction can then be wrapped in a ResultHelper member template so it can be used with RTHelper, provided we specialize HasResultHelper (see section 7). This is shown in listing 23.

```
struct OpBinPlus
{
    template<typename T,typename U>
    struct ResultHelper
    {
        typedef typename ExprType::RValueLValueType<
            typename EXPR_TYPE((
                ExprType::makeT<const T&>() + ExprType::makeT<const U&>()
            ))>::Type Type;
    };

    template<typename T,typename U>
    typename ResultHelper<T,U>::Type
    operator()(const T& t,const U& u) const
    {
        return t + u;
    }
};

template<>
struct HasResultHelper<OpBinPlus>
{
    static const bool result=true;
};
```

Listing 23: Implementation of the OpBinPlus functor.

This technique can then be extended to cover all the overloadable operators, both unary and binary. For those cases where the return type can be deduced directly from the arguments in a simple fashion (such as operator* for pointer types), the ResultHelper template can be specialized to do so, rather than use EXPR_TYPE.

11 Operators with special restrictions

Certain operators are required to be class members rather than free functions. This makes supporting them in a lambda function library that much harder; some expressions (such as x=_1) cannot be supported at all in their simplest form, and supporting other variations requires modification of several different classes. These operators are:

- The assignment operators (=, +=, -=, etc.),
- The function call operator,
- The subscript operator, and

- The class member access operator (`->`)

Other operators cannot be overridden at all, which makes implementing a lambda function library to support them as operators impossible — the Lambda Library [2] uses a function `if_then_else_return` to implement the conditional operator `?:`, for example. The operators that cannot be overloaded are:

- The class member access operators `.` and `.*`,
- The scope operator `::`,
- The conditional operator `?:`,
- The **`sizeof`** operator,
- The **`typeid`** operator, and
- **`new`** and **`delete`** *expressions*

Operators **`new`** and **`delete`** *can* be overloaded, but the result is not what would be required to use them in a lambda expression — the overloaded operators are merely used to allocate memory, not to select the type of object to construct, or what arguments to pass to its constructors. If allocation and deallocation expressions are required in a lambda expression, then we must implement factory functions instead.

The built-in logical operators `&&` and `||` use *short-circuit evaluation*, whereas user-defined overloads do not. This means that if we use the "normal" composition mechanism for binary operators with the logical operators, then the right-hand expression may be unnecessarily evaluated for those cases where the built-in operators should be used. Therefore we need a special means of permitting, but not requiring, *short-circuit evaluation* for the functions that implement the logical operators. This can be achieved by specializing the `CallerHelper` template for these functors to use the appropriate operator directly rather than delegating to the function-call operator of the functor, as shown in listing 24 for `OpBinLogicalAnd` (the implementation of `OpBinLogicalAnd` is similar to that of `OpBinPlus` shown in listing 23) — the call to `transformArg` on the right-hand side is avoided if the transformed types imply the use of the built-in logical-AND operator and the left-hand side evaluates to **`false`**.

```
template<typename ArgSet,typename CAS>
struct CallerHelper<OpBinLogicalAnd,ArgSet,CAS,2>
{
    typedef typename TransformArgSetHelper<ArgSet,CAS>::Type TransformedArgSet;
    typedef typename RTHelper<OpBinLogicalAnd,TransformedArgSet>::Type Type;

    static Type callFunc(const OpBinLogicalAnd&,const ArgSet& as,const CAS& cas)
    {
        typedef GetASEntry<ArgSet,1> Arg1;
        typedef GetASEntry<ArgSet,2> Arg2;

        return transformArg(Arg1::getVal(as),cas) && transformArg(Arg2::getVal(as),cas);
    }
};
```

Listing 24: Specializing of `CallerHelper` for `OpBinLogicalAnd` to allow short-circuit evaluation.

12 Conclusion

I am not attempting to provide a complete lambda-expression library here, as much work towards that has been done by others — such as Jaako Järvi and Gary Powell with their Lambda Library [2] — so I have not implemented those operators that are required to be member functions. Nor am I attempting to provide a

complete functional programming framework — Brian McNamara and Yannis Smaragdakis have done much work towards such a goal with their FC++ library [3], as have others.

Rather, I have shown how the framework which I have described can be used to easily build complex functors from others at the point of use, and to build a basic lambda expression facility; I hope that my framework can be combined with existing libraries to provide a more complete feature set, or used as a basis for extensions more within the philosophy and structure of those libraries.

References

[1] Andrei Alexandrescu. Generic<Programming>: Mappings between Types and Values. *C/C++ Users Journal*, 18(10), October 2000. Available online at http://www.cuj.com/experts/1810/alexandr.htm.

[2] Jaako Järvi and Gary Powell. The Lambda Library: Lambda Abstraction in C++. Technical Report 378, Turku Centre for Computer Science (TUCS), November 2000. Available online at http://www.tucs.fi/Publications/techreports/TR378.php. See also http://lambda.cs.utu.fi/.

[3] Brian McNamara and Yannis Smaragdakis. Functional Programming with the FC++ Library. Submitted to the Second Workshop on C++ Template Programming, 2001. Available online at http://www.cc.gatech.edu/ yannis/fc++/fcpptw.pdf. See also http://www.cc.gatech.edu/ yannis/fc++/.

[4] The Boost Team. Boost C++ libraries. See http://www.boost.org.

[5] Anthony Williams. `EXPR_TYPE` — An Implementation of `typeof` Using Current Standard C++. *Overload*, (??), ?? 2002? Available online at http://cplusplus.anthonyw.cjb.net/articles.html.