Typesafe Variable-length Function and Template Argument Lists

Douglas Gregor Gary Powell Jaakko Järvi

Document number: N1483=03-0066
Date: 25 April 2003
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <gregod@cs.rpi.edu>

1 Introduction

This proposal addresses three problems under a unified framework:

- The inability to instantiate class and function templates with an arbitrarily-long list of template parameters.
- The inability to pass an arbitrary number of arguments to a function in a type-safe manner.
- The argument forwarding problem.

The proposed resolution is to introduce a syntax and semantics for variable-length template argument lists (usable with function templates via explicit template argument specification and with class templates) along with a method of argument building using the same mechanism to pass an arbitrary number of function call arguments to a function in a typesafe manner.

2 Motivation

2.1 Variable-length template parameter lists

Variable-length template parameter lists (varargs) allow a class or function template to accept some number (possibly zero) of template arguments beyond the number of template parameters specified. This behavior can be simulated in C++ via a long list of defaulted template parameters, e.g., a typelist wrapper may appear as:

```
struct unused;
template<typename T1 = unused, typename T2 = unused,
         typename T3 = unused, typename T4 = unused,
         /* up to */ typename TN = unused> class list;
```

This technique is used by the Boost Tuples library [10], for the specification of class template `std::tuple<>` in the library TR [11], and in the Boost metaprogramming library [9]. Unfortunately, this method leads to very long type names in error messages (compilers tend to print the defaulted arguments) and very long mangled names. It is also not scalable to additional arguments without resorting to preprocessor magic [13]. In all of these libraries (and presumably many more), an implementation based on template varargs would be shorter and would not suffer the limitations of the aforementioned implementation. The declaration of the `list<>` template above may be:

```
template<...> class list;
```

This language feature is primarily designed for library developers, and is drawn from experience in library development where these techniques have been simulated in one way or another [9, 12, 10, 5, 1]. Novice users aren't likely to use this feature, as generically manipulating heterogeneous type container (tuples, in this case) requires some degree of metaprogramming.

2.2 Typesafe Variable-length Function Parameter Lists

Variable-length function parameter lists allow more arguments to be passed to a function than are declared by the function. This feature is rarely used in C++ code (except for compatibility with C libraries), because passing non-POD types through `...` invokes undefined behavior. However, a typesafe form of such a feature would be useful in many contexts, e.g., for implementing a typesafe C++ `printf` that works for non-POD types. The lack of such a facility has resulted in odd syntax for formatting in the Boost.Format library [14]:

```
format("writing %1%, x=%2%: %3%-th try") % "toto" % 40.23 % 50
```

2.3 Argument forwarding

The forwarding problem, as described in [4], is:

> For a given expression $E(a_1, a_2, ..., a_n)$ that depends on the (generic) parameters $a_1, a_2, ..., a_n$, it is not possible to write a function (object) f such that $f(a_1, a_2, ..., a_n)$ is equivalent to $E(a_1, a_2, ..., a_n)$.

The primary issue with the argument forwarding problem is to deduce the types of $a_1, a_2, ..., a_n$ such that they can be forwarded to the underlying function E without changing the semantics of the expression. The common implementation idiom is to accept a (non-const) reference for each parameter, and to provide overloads for some number of arguments to forward, e.g.:

```
template<typename E, typename T1>
void f(E e, T1& a1) { e(a1); }

template<typename E, typename T1, typename T2>
void f(E e, T1& a1, T2& a2) { e(a1, a2); }

template<typename E, typename T1, typename T2, typename T3>
void f(E e, T1& a1, T2& a2, T3& a3) { e(a1, a2, a3); }

// repeat up to N arguments
```

However, this idiom fails to accept literal values and also requires repetition for an arbitrary number of arguments (often done via preprocessor metaprogramming in the Boost libraries [13]). This proposal solves both issues: arguments are passed via the "Const + Non-const reference" option #3 enumerated in [4], which results in "perfect" forwarding of arguments, but does not require 2^N overloads to support N arguments: instead, a single overload suffices for any number of arguments. The solution matches all three criteria (C1-C3) of the argument forwarding paper [4] by bundling the arguments into a single argument and providing an unbundling method to pass the arguments on to another function.

3 Syntax and Semantics

3.1 Variable-length template parameter lists

The template parameter list to a function or class template can be declared to accept an arbitrary number of extra template arguments by terminating it with "..." optionally followed by an identifier that will hold a tuple of the extra arguments. Any number of template parameters may precede the "...". Thus we can define, for instance, a class `Foo` accepting a template type parameter followed by an arbitrary number of template arguments as:

```
template<typename T, ... Elements> class Foo;
```

Here, `T` is the name of the template type parameter and `Args` is the name of a `std::tuple<>` [11] containing the (possibly empty) set of template arguments given to `Foo`. Given the instantiation `Foo<int, float, double, std::string>`, `Elements` would be `std::tuple<float, double, std::string>` (note that `T` is `int`).

Nontype and template template arguments cannot be placed directly in the tuple (see [11]), so they are wrapped in `integral_constant` or `template_template_arg`, respectively.

`integral_constant` is part of the type traits library from the library TR [15], but we repeat the definition here:

```
template <typename T, T v>
struct integral_constant
{
   static  const T                value = v;
   typedef T                      value_type;
   typedef integral_constant<T,v> type;
};
```

`template_template_arg` is a class template with the following signature:

```
template<typename Designator>
struct template_template_arg {
  template<... Args> struct apply {
    typedef /* see below */ type;
  }
};
```

The `Designator` type is an unspecified type that uniquely identifies the class template (e.g., `std::vector`) but that is itself not a template[1]. The `apply` member class template accepts some number of template arguments and instantiates the template designated by the `Designator` with those parameters. Thus instantiating `Foo<int, std::vector>` results in `Args` being `std::tuple<template_template_arg<X> >` (where `X` is unspecied and unique to `std::vector`) and where `template_template_arg<X>::apply<int>::type` is equivalent to `std::vector<int>`.

Function templates may also be declared to accept variable-length template parameter lists in the same manner as class templates. In this case (and barring reuse of the vararg type as the final function parameter, as described in the next section) arguments may only by specified explicitly when calling the function template; they are not deduced, e.g.,:

```
template<... Args> void f();
// ...
f<int, double>(); // Args will be std::tuple<int, double>
```

3.2 Typesafe variable-length function parameter lists

When a function template header contains an ellipsis (followed by an identifier), that identifier may be reused as the type of the final parameter in the function's parameter list to accept the tuple of arguments passed to the function. For instance, a C++ `printf` may be declared as:

```
template<... Args>
  void printf(const char* format, Args args);
```

`Args` is deduced as an `std::tuple` containing the types of the arguments, then `args` is a value of that type containing the function call arguments passed after the `format` argument.[2] Arguments are then accessed via tuple operations [11].

[1] This definition may be expanded to allow typedef templates or template aliases as well, if they can be passed as template arguments.

[2] Note that, unlike C varargs, we do not require but do allow an argument prior to the argument bundle.

3.3 Forwarding function arguments

A tuple of values can be forwarded to a function object via a new function template `std::apply` that unbundles a tuple into separate elements to be passed to a function object. The pseudo-definitions for `std::apply` are:

```
template<typename F, typename T1, typename T2, ..., typename TN>
the-return-type apply(F& f, tuple<T1, T2, ..., TN>& t)
  { return f(get<0>(t), get<1>(t), ..., get<N-1>(t)); }

template<typename F, typename Class, typename T1, typename T2, ..., typename TN>
the-return-type apply(F Class::*pmf cv, Class& obj, tuple<T1, T2, ..., TN>& t)
  { return (obj.*pmf)(get<0>(t), get<1>(t), ..., get<N-1>(t)); }
```

The `the-return-type` placeholder denotes the exact return type of the return expression, such that the result can be returned unaltered to the caller. This placeholder will be reused through the pro[posal.

With these definitions, we can define a function `g` that forwards perfectly to its function object parameter `f`:

```
template<typename F, ... Args>
  the-return-type g(F f, Args args) {return std::apply(f, args);}
```

3.4 Argument type deduction

The type of each element in the argument tuple is deduced to achieve perfect argument forwarding. For an argument `xi`, the type `Ti` of the corresponding tuple element is the parameter type of the `forward` function determined by overload resolution using the following two candidates:

```
template<typename T> void forward(T&);
template<typename T> void forward(const T &);
```

3.5 Partial ordering of vararg class template partial specializations

The class template partial specialization partial ordering rules will need to be augmented to include partial ordering with vararg templates. Intuitively, a binding to a parameter that falls into a template's variable-length argument list is weaker than a binding to a specified template parameter. For instance, given:

```
template<...> struct foo;
template<typename T, ...> struct foo<T>; // #1
template<typename T, typename U> struct foo<T, U>; // #2
```

Partial specialization #2 is more specialized than partial specialization #1 because #2 requires that the second template argument by a type (and not a template or acceptable literal).

Formalizing this notion, we introduce a new type of template parameter we call a template *variant* parameter. Template variant parameters cannot be declared explicitly, but occur implicitly as parameters for vararg templates. However, any template argument (type, nontype, or template) can be passed to a template *variant* parameter.

Paragraph 3 of 14.5.5.2 [temp.func.order] describes the rules for transforming a template for the purpose of partial ordering. To support partial ordering with template varargs, introduce two additional bullets:

- If after removing the variable-length template parameter list designator from both templates the template parameter lists of the templates are of different length, append unique template variant parameters to the shorter template parameter list until the template parameter lists are of equal length.
- A binding of an argument to a variant parameter is weaker than a binding of an argument to a parameter of the same kind.

3.6 Overloading

The overloading rules need two minor changes to accomodate template varargs:

- Template vararg functions follow the same overloading rules as non-template vararg functions.
- If two overloads differ only in that one is a template vararg function and the other is a non-template vararg function, the template vararg function is more specialized.

Thus we prefer the typesafe template varargs to unsafe template varargs, even though these rules conflict with 13.3.3p1, which prefers non-template functions to template functions when the conversion sequences are otherwise equal.

3.7 Tuple cons operation

Tuples can be split into some number of head elements and a "tail" element using the `std::apply` function, but arguments cannot be spliced with tuples to generate larger tuples.[3] We introduce a `cons` operation for tuples, e.g., an operation that accepts a value `a1` of type `T1` and a tuple `t` of type `tuple<T2, T3, ..., TN>` and returns the tuple `tuple<T1, T2, ..., TN>(a1, get<0>(t), get<1>(t), ..., get<N-2>(t))`. The `tuple_cons` function has the following pseudo-signature:

```
template<typename T1, typename T2, ..., typename TN>
tuple<T1, T2, ..., TN> tuple_cons(T1 a1, const tuple<T2, T3, ..., TN>& t)
  { return tuple<T1, T2, ..., TN>(a1, get<0>(t), get<1>(t), ..., get<N-2>(t)); }
```

With `tuple_cons`, we have both the ability to break apart tuples and to rebuild them, allowing the construction of tuple algorithms [7] that are crucial in the use of vararg templates; see the implementation of the tuple `transform` in Section 4.1 and its use in the implementations of a function object binder (Section 4.5) and the tuples library itself (Section 4.3).

3.8 Function traits

The Type Traits library proposal [15] defines a set of class templates that report information about types. We propose to add one additional class template `function_traits` that reports information about function types. The pseudo-definition follows:

```
template<typename FunctionType> struct function_traits;

// For each 0 <= N <= implementation-defined maximum
template<typename R, typename T1, typename T2, ..., typename TN>
struct function_traits<R(T1, T2, ..., TN)>
{
  typedef R result_type;
  typedef tuple<T1, T2, ..., TN> argument_types;
};
```

This class template can be implemented for a fixed number of arguments in a library, but requires compiler support to handle an arbitrary number of arguments. It is useful for many libraries that deal with function and member function pointers as function objects, where one may need to forward a set of function arguments whose types are fixed (see Section 4.4).

[3] We are not certain that this is the case. The problem reduces to being able to name the resulting tuple type, but we have not solved it without further extension.

4 Examples

To demonstrate the use of vararg templates, this section gives skeletal implementations of four library components using vararg templates, including Tuples. All of these proposals were accepted by the library working group into the first library TR, and are implementable to some degree in C++03. However, all of them would benefit from template varargs in that the implementations can be more complete, more useful, and more readable than their C++03 counterparts. The following libraries are implemented:

- Member pointer adaptors [2]
- Tuples [11]
- Function object wrappers [6]
- Function object binder [3]

The implementations are not generally complete, but omit only features that are irrelevant to the discussion of template varargs. The implementations use several class templates (metafunctions) from the type traits proposal [15].

4.1 Building Blocks

The class template `enable_if` is used liberally to help guide the overload process by eliminating candidates from the overload set that would otherwise cause ambiguities or be selected when they shouldn't be. It is defined as:

```
template<bool Cond, typename T> struct enable_if;
template<typename T> struct enable_if<true, T> { typedef T type; };
template<typename T> struct enable_if<false, T> {};
```

We also rely on a tuple-specific version of the `transform` function found in the standard library. The tuple transform transforms a tuple of length N into another tuple of length N by applying a function object to each element in the source tuple and initializing the corresponding element in the result tuple. Note that the types in the result tuple (`U1`, `U2`, ..., `UN`) are the types returned from the transforming function object, as we assume these will be available (e.g., via a `typeof` extension or using the library-defined `result_of` [8]). The following pseudo-definition defines `transform`:

```
// Type Ui is the type of f(std::get<i-1>(t))
template<typename T1, typename T2, ..., typename TN, typename F>
tuple<U1, U2, ..., UN> transform(const tuple<T1, T2, ..., TN>& t, F f)
{
  return tuple<U1, U2, ..., UN>(f(std::get<0>(t)),
                                f(std::get<1>(t)),
                                      ...
                                f(std::get<N-1>(t)));
}
```

Transform can be implemented as follows using template varargs:

```
template<typename F>
struct transformer
{
  transformer(F& f) : f(f) {}

  // U1 is the type of f(a1)
  template<typename T1, ... Args>
  the-return-type operator()(T1& a1, Args args) const
```

```
    { return tuple_cons<U1>(f(a1), std::apply(*this, args)); }

  // U1 is the type of f(a1)
  template<typename T1, ... Args>
  the-return-type operator()(const T1& a1, Args args) const
    { return tuple_cons<U1>(f(a1), std::apply(*this, args)); }

  // basis case
  tuple<> operator()() const { return tuple<>(); }

private:
  F& f;
};

template<typename Tuple, typename F>
the-return-type transform(const Tuple& t, F f)
{
  return std::apply(transformer<F>(f), t);
}
```

4.2 Member pointer adaptor implementation

The following code implements the enhanced member pointer adaptor proposal [2]. This implementation is complete.

```
// Adaptor for member function pointers
template<typename Class, typename FunctionType>
struct mem_fn_adaptor
{
  typedef typename function_traits<FunctionType>::result_type result_type;

  mem_fn_adaptor(FunctionType Class::*pmf) : pmf(pmf) {}

  template<... Args>
  result_type operator()(Class& object, Args args) const
    { return std::apply(pmf, object, args); }

  template<... Args>
  result_type operator()(const Class& object, Args args) const
    { return std::apply(pmf, object, args); }

  template<typename T, ... Args>
  typename enable_if<(!is_base_of<Class, T>::value), result_type>::type
  operator()(T& ptr, Args args) const
    { return std::apply(pmf, *ptr, args); }

  template<typename T, ... Args>
  typename enable_if<(!is_base_of<Class, T>::value), result_type>::type
  operator()(const T& ptr, Args args) const
    { return std::apply(pmf, *ptr, args); }

private:
  FunctionType Class::*pmf;
};

// Adaptor for member data pointers (for completeness only)
template<typename Class, typename T>
```

```
struct mem_ptr_adaptor
{
  typedef const T& result_type;

  mem_ptr_adaptor(T Class::*pm) : pm(pm) {}

  T& operator()(Class& object) const { return object.*pm; }
  const T& operator()(const Class& object) const { return object.*pm; }


  template<typename Ptr>
  typename enable_if<(!is_base_of<Class, Ptr>::value), const T&>::type
    operator()(Ptr& ptr) const { return (*ptr).*pm; }

  template<typename Ptr>
  typename enable_if<(!is_base_of<Class, Ptr>::value), const T&>::type
    operator()(const Ptr& ptr) const { return (*ptr).*pm; }

private:
  T Class::*pm;
};

template<typename Class, typename FunctionType>
typename enable_if<(is_function<FunctionType>::value),
                   mem_fn_adaptor<Class, FunctionType> >::type
mem_fn(FunctionType Class::*pmf)
  { return mem_fn_adaptor<Class, FunctionType>(pmf); }

template<typename Class, typename T>
typename enable_if<(!is_function<T>::value),
                   mem_fn_adaptor<Class, T> >::type
mem_fn(T Class::*pm)
  { return mem_fn_adaptor<Class, T>(pm); }
```

4.3 Tuple implementation

The following code implements the interesting portions of the Tuple proposal [11] using template varargs as proposed here. Note that there is a mutual dependency: template varargs depend on tuples and tuples are specified as a class template accepting a variable-length template argument list. However, this mutual dependency is not problematic, because we split a tuple of length N into a head element and a tail tuple of length $N - 1$, with the base case (length 0) as an explicit specialization.

```
// Derivation from tuple_base indicates that a type is a tuple
class tuple_base {};

// Determine if type T is a tuple
template<typename T>
struct is_tuple
  { static const bool value = is_base_of<tuple_base, T>::value; };

// A tuple of arbitrary length N
template<typename T1, ... Elements>
class tuple : public tuple_base
{
  typedef T1 head_type;
  typedef typename add_reference<typename add_const<head_type>::type>::type head_param_type;
  typedef Elements tail_type;
  typedef tuple self_type;
```

```
public:
  static const size_t size = 1 + tail_type::size;

  tuple() {}

  // enable_if condition ensures that we only match if N arguments are given
  template<... Tail>
  explicit tuple(typename enable_if<(Args::size == size - 1),
                                    head_param_type>::type head,
                 Tail tail)
    : head(head), tail(tail) {}

  // enable_if condition only allows us to attempt implicit conversions
  // from tuples of the appropriate length
  template<typename Tuple>
  tuple(const Tuple& other,
        typename enable_if<(is_tuple<Tuple>::value && size == Tuple::size),
                           void>::type* = 0)
    : head(other.head), tail(other.tail) {}

  // enable_if condition only allows us to attempt assignment from tuples of
  // the appropriate length
  template<typename Tuple>
  typename enable_if<(is_tuple<Tuple>::value && size == Tuple::size),
                     tuple&>::type
  operator=(const Tuple& tuple)
  {
    head = other.head;
    tail = other.tail;
    return *this;
  }

private:
  head_type head;
  tail_type tail;
};

// Base case: a nullary tuple
template<> class tuple<> : public tuple_base
{
public:
  static const size_t size = 0;
};

// 2-element case, for pair specializations
template<typename T1, typename T2>
class tuple : public tuple_base
{
  typedef T1 head_type;
  typedef tuple<T2> tail_type;

  typedef typename add_cv<T1>::type param1_type;
  typedef typename add_cv<T2>::type param2_type;

public:
  static const size_t size = 2;
```

```
  tuple() {}

  tuple(param1_type t1, param2_type) : head(t1), tail(t2) {}

  template<typename U1, typename U2>
  tuple(const pair<U1, U2>& other) : head(other.first), tail(other.second) {}

  template<typename U1, typename U2>
  tuple(const tuple<U1, U2>& other) : head(other.head), tail(other.tail) {}

  template<typename U1, typename U2>
  tuple& operator=(const pair<U1, U2>& other)
  {
    head = other.first;
    tail = tail_type(other.second);
    return *this;
  }

  template<typename U1, typename U2>
  tuple& operator=(const tuple<U1, U2>& other)
  {
    head = other.head;
    tail = other.tail;
    return *this;
  }

private:
  head_type head;
  tail_type tail;
};

struct convert_make_tuple_arg
{
  template<typename T> T operator()(T& t) { return t; }
  template<typename T> T& operator()(reference_wrapper<T>& t) { return t.get(); }
};

template<... Args> the-return-type make_tuple(Args args)
  { return transform(args, convert_make_tuple_arg()); }

template<... Args> Args tie(Args args) { return args; }
```

4.4 Function implementation

This section implements the core interesting portions of class template `function` from the function object wrapper proposal [6].

```
struct bad_function_call : public std::exception {};

template<typename R, typename Args>
struct function_invoker_base
{
  virtual ~function_invoker_base() {}
  virtual R invoke(Args& args) = 0;
  virtual function_invoker_base* clone() = 0;
};

template<typename R, typename Args, typename F>
struct function_invoker : public function_invoker_base<R, Args>
```

```
{
  function_invoker(F f) : f(f) {}
  R invoke(Args& args) { return std::apply(f, args); }
  function_invoker_base<R, Args>* clone() { return new function_invoker(f); }

private:
   F f;
};

template<typename Args, typename F>
struct function_invoker<void, Args, F>
  : public function_invoker_base<void, Args>
{
  function_invoker(F f) : f(f) {}
  void invoke(Args & args) { std::apply(f, args); }
  function_invoker_base<void, Args>* clone() { return new function_invoker(f); }

private:
  F f;
};

template<typename Function> // Function type R(T1, T2, ..., TN)
struct function
{
  typedef typename function_traits<Function>::result_type result_type;
  typedef typename function_traits<Function>::argument_types argument_types;

private:
  typedef function_invoker_base<result_type, argument_types> invoker_type;

public:
  function() : invoker(0) {}

  function(const function& other) : invoker(0)
    { if (other.invoker) invoker = other.invoker->clone(); }

  template<typename F> function(F f) : invoker(0)
    { invoker = new function_invoker<result_type, argument_types, F>(f); }

  template<typename F> function(reference_wrapper<F> f) : invoker(0)
  {
    invoker = new function_invoker<result_type, argument_types, F&>(f.get());
  }

  template<typename T, typename C> function(T C::*pm) : invoker(0)
    { *this = std::mem_fn(pm); }

  function(int) : invoker(0) {}

  ~function() { if (invoker) { delete invoker; invoker = 0; } }

  function& operator=(const function& other)
  {
    function tmp(other);
    tmp.swap(*this);
    return *this;
  }
```

```
  template<typename F> function& operator=(F f)
  {
    function tmp(f);
    tmp.swap(*this);
    return *this;
  }

  operator bool() const { return invoker; }

  template<... Args>
  result_type operator()(Args args) const
  {
    if (!invoker) throw bad_function_call;
    return invoker->invoke(args);
  }

  void swap(function& other)
  {
    using std::swap;
    swap(invoker, other.invoker);
  }

private:
  invoker_type* invoker;
};
```

4.5 Bind implementation

The following code implements the enhanced binder proposal [3] using vararg templates, and building on the existing tuple and member pointer adaptor code. This implementation is complete, with two exceptions (both of which require uninteresting but nontrivial code, expressible in C++03):

- `result_type` is not specified in `binder`
- arity checking is not performed at bind time

```
// Placeholders
template<int N> struct placeholder {};

namespace placeholders {
  typedef placeholder<1> _1;
  typedef placeholder<2> _2;
  typedef placeholder<3> _3;
}

template<typename T> struct is_placeholder : integral_constant<size_t, 0> {};

template<int N>
struct is_placeholder<placeholder<N> > : integral_constant<size_t, N> {};

// Lambda
template<typename X> X& lambda(const reference_wrapper<X>& x)
  { return x.get(); }
template<typename X> X& lambda(X& x)
  { return x; }

// Mu
template<typename X, typename Args> X& mu(reference_wrapper<X>& x, Args&)
```

```
  { return x.get(); }

template<typename X, typename Args>
typename enable_if<(is_placeholder<X>::value),
                   typename tuple_element<X, is_placeholder<X>::value-1>::type
                   >::type
mu(X&, Args& args)
  { return std::get<is_placeholder<X>::value-1>(args); }

template<typename X, typename Args>
typename enable_if<(is_bind_expression<X>::value), the-return-type>::type
mu(X& x, Args& args)
  { return std::apply(x, args); }

template<typename X, typename Args> X& mu(X& x, Args&)
  { return x; }

// Unary function object that calls mu with the object 'x' it receives
// and the bound arguments from binder_t.
template<typename BoundArgs>
struct apply_mu_t
{
  explicit apply_mu_t(BoundArgs& bound_args) : bound_args(bound_args) {}

  template<typename X> the-return-type operator()(X& x) const
    { return mu(x, bound_args); }

private:
  BoundArgs bound_args;
};

// Return type of bind(...)
template<typename F, typename BoundArgs>
struct binder
{
  binder(const F& f, const BoundArgs& bound_args)
    : f(f), bound_args(bound_args) {}

  template<... Args> the-return-type operator()(Args args)
    { return std::apply(lambda(f), transform(args, apply_mu(bound_args))); }

  template<... Args> the-return-type operator()(Args args) const
    { return std::apply(lambda(f), transform(args, apply_mu(bound_args))); }

private:
  F f;
  BoundArgs bound_args;
};

// Helper to construct binder objects
template<typename F, typename BoundArgs>
binder<F, BoundArgs> make_binder(F f, const BoundArgs& bound_args)
  { return binder<F, BoundArgs>(f, bound_args); }

// Determine if a type T is a binder type
template<typename T> struct is_bind_expression : integral_constant<bool, false> {};

template<typename F, typename BoundArgs>
```

```
struct is_bind_expression<binder<F, BoundArgs> > : integral_constant<bool, true> {};

// Remove the reference from a value
struct remove_ref { template<typename T> T operator()(T& x) const { return x; } };

// Bind overload for function objects and function pointers
template<typename F, ... BoundArgs>
the-return-type bind(F f, BoundArgs bound_args)
  { return make_binder(f, transform(bound_args, remove_ref())); }

// Bind overload for member pointers (both function and data)
template<typename Class, typename T, ... BoundArgs>
the-return-type bind(T Class::*pm, BoundArgs bound_args)
  { return make_binder(mem_fn(f), transform(bound_args, remove_ref())); }
```

References

[1] P. Dimov. The Boost Bind library. `http://www.boost.org/libs/bind/bind.html`, August 2001.

[2] P. Dimov. A proposal to add an enhanced member pointer adaptor to the library technical report. Number N1432=03-0014 in ANSI/ISO C++ Standard Committee Pre-Oxford mailing, March 2003.

[3] P. Dimov, D. Gregor, J. Järvi, and G. Powell. A proposal to add an enhanced binder to the library technical report. Number N1455=03-0038 in ANSI/ISO C++ Standard Committee Post-Oxford mailing, April 2003.

[4] P. Dimov, H. Hinnant, and D. Abrahams. The forwarding problem: Arguments. Number N1385=02-0043 in ANSI/ISO C++ Standard Committee Pre-Santa Cruz mailing, October 2002.

[5] D. Gregor. The Boost Function library. `http://www.boost.org/doc/html/function.html`, June 2001.

[6] D. Gregor. A proposal to add a polymorphic function object wrapper to the standard library. Number N1402=02-0060 in ANSI/ISO C++ Standard Committee Post-Santa Cruz mailing, October 2002.

[7] D. Gregor. Tuple algorithms. `http://www.cs.rpi.edu/~gregod/C++0x/tuple_algo.html`, March 2003.

[8] D. Gregor. A uniform method for computing function object return types. Number N1454=03-0037 in ANSI/ISO C++ Standard Committee Post-Oxford mailing, April 2003.

[9] A. Gurtovoy. The Boost MPL library. `http://www.boost.org/libs/mpl/doc/index.html`, July 2002.

[10] J. Järvi. The Boost Tuples library. `http://www.boost.org/libs/tuple/doc/tuple_users_guide.html`, June 2001.

[11] J. Järvi. Proposal for adding tuple types to the standard library. Number N1403=02-0061 in ANSI/ISO C++ Standard Committee Post-Santa Cruz mailing, October 2002.

[12] J. Järvi and G. Powell. The Boost Lambda library. `http://www.boost.org/libs/lambda/doc/index.html`, March 2002.

[13] V. Karvonen and P. Mensonides. The Boost Preprocessor library. `http://www.boost.org/libs/preprocessor/doc/index.html`, July 2001.

[14] S. Krempp. The Boost Format library. `http://www.boost.org/libs/format/index.htm`, January 2002.

[15] J. Maddock. A proposal to add type traits to the standard library. Number N1424=03-0006 in ANSI/ISO C++ Standard Committee Pre-Oxford mailing, March 2003.