The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

A Framework For Higher-Order Functions In C++

Konstantin Laufer
Loyola University of Chicago

A Framework for Higher-Order Functions in C++

Konstantin Läufer

Loyola University of Chicago
laufer@math.luc.edu

Abstract

C and C++ allow passing functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions declared at global or file scope, these function arguments cannot capture local environments. This leads to the common misconception that C and C++ do not support function closures.

In fact, function closures can be modeled directly in C++ by enclosing a function inside an object such that the local environment is captured by data members of the object. This idiom is described in advanced C++ texts and is used, for example, to implement callbacks.

The purpose of this paper is twofold: First, we demonstrate how this idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that support composition and partial application. Second, we explore the expressiveness of the framework and compare it with that of existing functional programming languages.

We illustrate by means of various examples that object-oriented and functional idioms can coexist productively and can be used to enhance the functionality of common classes, for example, of nonlinear collections such as trees. A C++ implementation of the framework is available on request.

1 Introduction

The programming languages C [HS87] and C++ [ES90] allow passing functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions declared at global or file scope, these function arguments cannot capture local environments. This leads to the common misconception that C and C++ do not support function closures.

On the contrary, function closures can be modeled in C++ by enclosing a function inside an object such that the local environment or parts thereof are captured by data members of the object. This is possible because objects in C++ are essentially higher-order records, that is, records with fields that can contain not only values, but also functions [Red95].

The idiom of objects that enclose functions is first described by Coplien [Cop92], who calls these objects "functors", and further developed by Kühne [Küh95], who calls them "function objects". In this paper, we use the term "functoid" for brevity and to avoid confusion with established meanings of "functor" in other areas of computer science. The functoid idiom has been used, for example, in callbacks and iterators with type-safe interfaces. A related idiom that focuses on maintaining a binding between a receiver and a function is called "command" by Gamma et al. [GHJV93]. Unlike Kühne, Gamma et al. do not establish a relationship between the "command" and "iterator" idioms. The "command" idiom is also known as "action" or "transaction" and is used in various object-oriented application frameworks [JF88], including ET++ [WGM88], InterViews [LCI+92], MacApp [App89], and Unidraw [VL90].

Nevertheless, various existing class libraries provide internal (passive) iterators with weakly-typed interfaces and place the responsibility of applying suitable type casts on the user. For example, Borland C++ [Bor94] uses the following internal iterator in its container class templates.

```
void Container<Item>::forEach
    (void(* f )(Item&, void*), void* args);
```

The purpose of the second argument of `f` and the argument `args` is to allow passing specific arguments to `f`. Using the functoid idiom, the iterator could be provided in the following type-safe way.

```
void Container<Item>::forEach
    (Visitor<Item>& f);

template <class Item> class Visitor
{
public:
    virtual void operator()(const Item&)
    = 0;
};
```

By deriving from the class template `Visitor`, information can be passed to and from the iterator in a type-safe manner, where specific arguments are passed as arguments to the constructor of the visitor. For example, the following visitor adds the elements of a container of integers.

```
class Adder : public Visitor<int>
```

```
{
public:
    Adder(int& s) : sum(s) { }
    virtual void operator()
        (const int& item)
    { sum += item; }
private:
    int& sum;
};

int sum = 0;
Adder add(sum);
container.forEach(add);
cout << sum;
```

The purpose of this paper is twofold: First, we demonstrate how the functoid idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that support composition and partial application. The framework could be translated to other object-oriented languages that support inheritance and genericity. Second, we explore the expressiveness of the framework and compare it with that of existing functional programming languages.

We show informally that there is a simple compositional translation from functional programs to the framework. We illustrate by means of various examples that object-oriented and functional idioms can coexist productively and can be used to enhance the functionality of common classes, for example, of nonlinear collections such as trees. To integrate the framework with C and C++ programs, we incorporate an existing mechanism to convert member functions back to nonmember functions. A C++ implementation of the framework is available on request.

In the remainder of this paper, Section 2 describes in detail the requirements, the implementation, and the structure of the functoid framework. Section 3 conducts a case study in which a typical functional program expressed within the framework. Section 4 explains how an existing conversion mechanism from C++ member functions to ordinary nonmember functions is incorporated in the framework. Section 5 concludes with an assessment of this work and a look at related and future work.

2 Functoids: An Abstraction of Functions

This section introduces the framework of functoids. In this framework, a *functoid* is an abstraction of the familiar concept "function".

Requirements

We first establish the requirements of the functoid abstraction. Our goal is to provide a *type-safe* abstraction, that is, there should be no need for type casts or untyped pointers at the user level. The abstraction should be provided in the form of C++ classes or class templates. We require that the abstraction supports the following essential operations performed on or by functions:

- *Application*

  Functoids can be applied to arguments. When a functoid is invoked by applying it to one or more arguments of appropriate types, the functoid returns a value of the appropriate result type.

- *Creation*

  Functoids can be created statically or dynamically. Upon creation, the functoid can capture and remember parts of the current environment.

- *Composition*

  Functoids can be composed with one another. When a functoid `f` is composed with another functoid `g`, the result of the composition is a new functoid `h`. When `h` is applied to an argument, it first obtains an intermediate result by applying `g` to the argument and then returns as a final result the application of `f` to the intermediate result.

- *Partial application*

  Functoids can be applied partially to fewer arguments than they actually accept. The result of partial application is a new functoid that accepts the remaining arguments. The conversion to a functoid that takes its arguments one at a time is known as "currying" in functional programming terminology.

- *Conversion*

  Functoids are equivalent to ordinary functions. An ordinary function can be converted to a functoid, and a functoid can be converted to an ordinary nonmember function when such a function is required, for example, as a callback function for an existing library. Conversion back to nonmember functions is difficult and will be addressed in Section 4.

- *Extension*

  Functoids provide extensible functionality. We can add application-specific operations to a functoid besides the basic operations described above.

We implement our abstraction as a class template `Fun` that provides the interface to the abstraction *functoid* and is parameterized by the types of argument and result. The creation requirement will be handled by the constructors of this class, and the application requirement is captured by a function call operator of the appropriate type.

```
Out Fun<In,Out>::operator()(In arg) const;
```

The question arises how users of the functoid framework should incorporate their own functoid classes. The idea is that users derive concrete functoid classes from the class `Fun`, providing their own implementations of the function call operator. To make this approach work, the function call operator would have to be declared as virtual so that dynamic method selection is used, and functoids would always have to be passed and returned by pointer or reference [ES90]. On the other hand, memory management becomes an important issue when objects are not returned by value [Mey92]. What we want here is both call-by-value and dynamic method selection.

Fortunately, the *envelope/letter* idiom [Cop92], also known as the *bridge* pattern [GHJV93], gives us a way out of this dilemma. We apply this idiom to the framework as follows. We provide a class template called `Fun` to capture the interface of our abstraction. This is the envelope class, and functoids are passed and returned by value as instances of this class. We also provide an abstract class template called `FunImpl` for implementations of functoids. This class is an abstract letter class, and users of the framework provide their own functoid implementations by deriving from this class. The envelope class has a pointer to the letter class, and invocations of the function call operator in an envelope object are simply passed on to the letter object.

We first present the envelope class template `Fun` because it comes first conceptually, although it depends on the class template `FunImpl`. We provide two constructors, one to create a functoid from an existing functoid implementation, and a copy constructor that makes an explicit copy of the implementation of the functoid it copies. This is necessary so that no two functoid implementations are shared and the destructor can safely delete the corresponding implementation. Furthermore, we provide a function call operator that simply passes the function call on to the implementation, accessible via the pointer `impl`.

```
template <class In, class Out> class Fun
{
public:
    Fun(FunImpl<In,Out>* const f)
        : impl(f) { }
    Fun(const Fun& fun)
        : impl(fun.impl->copy()) { }
    ~Fun() { delete impl; }
    Out operator()(In arg) const
    { return (*impl)(arg); }
private:
    FunImpl<In,Out>* const impl;
};
```

We now present the abstract letter class template `FunImpl`. It has a virtual destructor so that the appropriate destructor in a concrete subclass is invoked when a functoid deletes its implementation. Furthermore, it provides a member function that makes a copy of the receiver to support the copy constructor of the envelope class.

```
template <class In, class Out>
class FunImpl
{
public:
    virtual ~FunImpl() { }
    virtual Out operator()(In arg) const
    = 0;
    virtual FunImpl<In,Out>* copy() const
    = 0;
};
```

Composition

The next issue deals with the composition of functoids. In a functional programming language such as ML [MTH90], a function that composes two functions can be expressed as follows:

```
fun compose(f,g) = fn x => f(g(x))
```

The form "`fn x => e`" creates an anonymous function with argument `x` and body `e`. Thus the composition yields a new function with argument `x` and result `f(g(x))`. The composition is permitted only if the result type of `g` is compatible with the argument type of `f`. The new function has the same argument type as `g` and the same result type as `f`.

To avoid excess parameterization of the template `Fun`, we provide this functionality as a nonmember function that returns a functoid composed from the two functoid arguments. This resulting functoid is an instance of the class template `Compose` and holds the two functoids to be composed; the composition itself is carried out in the function call operator of this class. Both the class and the function templates have three type parameters for the argument, intermediate, and result types.

```
template <class In, class Med, class Out>
```

```
class Compose : public FunImpl<In,Out>
{
public:
    Compose(const Fun<Med,Out>& f,
            const Fun<In,Med>& g)
        : ffun(f), gfun(g) { }
    virtual Out operator()(In arg) const
    { return ffun(gfun(arg)); }
    virtual FunImpl<In,Out>* copy() const
    {
        return new Compose<In,Med,Out>
                   (ffun, gfun);
    }
private:
    const Fun<Med,Out> ffun;
    const Fun<In,Med> gfun;
};

template <class In, class Med, class Out>
Fun<In,Out> compose(const Fun<Med,Out>& f,
                    const Fun<In,Med>& g)
{
    return Fun<In,Out>
        (new Compose<In,Med,Out>(f, g));
}
```

Conversion from nonmember functions

The next requirement is conversion from an ordinary nonmember function to a functoid. The reverse direction is discussed below in Section 4. It is not hard to create a functoid from a nonmember function. Such a functoid can be implemented with a data member that points to the function and a function call operator that passes its argument on to the function pointer.

```
template <class In, class Out> class Global
    : public FunImpl<In,Out>
{
public:
    typedef Out(* FunPtr )(In);
    Global(FunPtr f) : theFun(f) { }
    virtual Out operator()(In arg) const
    { return theFun(arg); }
    virtual FunImpl<In,Out>* copy() const
    { return new Global<In,Out>(theFun); }
private:
    FunPtr theFun;
};
```

To enable automatic conversion from an ordinary function to a functoid, we add the following constructor to the class template `Fun`, where `FunPtr` is defined as in the class template `Global`.

```
Fun<In,Out>::Fun(FunPtr f)
    : impl(new Global<In,Out>(f)) { }
```

Partial application

Another issue is how to deal with functoids that take more than one argument. In ML, a function that partially applies a function of two arguments to the first argument can be written as follows:

```
fun apply(f,x) = fn y => f(x,y)
```

The result of the partial application of `f` to the first argument `x` is a new function with a single argument `y` and result `f` applied to `x` and `y`. The argument type of the new function is the type of the second argument of `f`, and its result type is the result type of `f`.

In the framework, the class template for functoids with multiple arguments would have to be parameterized by all argument types and the result type. Therefore the framework has to provide an envelope and a letter class for each number of arguments that could reasonably arise. If the maximum number of arguments is exceeded, a solution is to group several arguments in a single object. However, our partial application requirement can be satisfied only if the functoid accepts its arguments one-by-one. A better approach would thus be to automate the generation of the class templates depending on the maximum number of arguments desired in the application. The structure of the framework is sufficiently systematic to make this a feasible option.

We now illustrate partial evaluation for functoids of two arguments. First comes the abstract letter class `Fun2Impl`, followed by the corresponding envelope class `Fun2`. These classes differ from `FunImpl` and `Fun` in that they have two function call operators: one that takes two arguments instead of one, and one that takes a single argument and returns a new functoid. The second function call operator provides partial evaluation by applying the functoid to the first argument only.

```
template <class In1, class In2, class Out>
class Fun2Impl
{
public:
    virtual ~Fun2Impl() { }
    virtual Out operator()
        (In1 arg1, In2 arg2) const = 0;
    virtual Fun2Impl<In1,In2,Out>* copy()
    const = 0;
};

template <class In1, class In2, class Out>
class Fun2
{
public:
    typedef Out(* Fun2Ptr )(In1, In2);
    Fun2(Fun2Impl<In1,In2,Out>* const f)
        : impl(f) { }
    Fun2(Fun2Ptr f)
        : impl(new Global2<In1,In2,Out>(f))
    { }
    Fun2(const Fun2<In1,In2,Out>& fun)
```

```
        : impl(fun.impl->copy()) { }
    ~Fun2() { delete impl; }
    Out operator()(In1 arg1, In2 arg2)
    const
    { return (*impl)(arg1, arg2); }
    inline Fun<In2,Out> operator()
        (In1 arg1) const;
private:
    Fun2Impl<In1,In2,Out>* const impl;
};
```

To complete our implementation of partial evaluation, we must implement the function call operator that takes only one argument. This operator returns an instance of the class template `Apply21`, which keeps track of the *first* argument and the original functoid. When the function call operator of an instance of `Apply21` is invoked with the *second* argument, the operator simply applies the original functoid to both arguments.

```
template <class In1, class In2, class Out>
class Apply21 : public FunImpl<In2,Out>
{
public:
    Apply21(const Fun2<In1,In2,Out>& fun,
            const In1& arg1)
        : theFun(fun), theArg(arg1) { }
    virtual Out operator()(In2 arg2) const
    { return theFun(theArg, arg2); }
    virtual FunImpl<In2,Out>* copy() const
    {
          return new Apply21<In1,In2,Out>
              (theFun, theArg);
    }
private:
    const Fun2<In1,In2,Out> theFun;
    const In1 theArg;
};

template <class In1, class In2, class Out>
inline Fun<In2,Out> Fun2<In1,In2,Out>::
operator()(In1 arg1) const
{
    return Fun<In2,Out>
        (new Apply21<In1,In2,Out>
            (*this, arg1));
}
```

For additional flexibility in combining partial evaluation and composition, we also provide partial evaluation without application to any arguments. In ML, such a function is written as follows:

```
fun curry(f) = fn x => fn y => f(x,y)
```

This function converts its argument `f` to a new function that takes its arguments one after the other instead of both at the same time.

In the framework, the additional member function `curry1` in class `Fun2` converts a functoid of two arguments to a new functoid of the auxiliary class `Curry1`. The function call operator in the new functoid takes one argument (the first one) and returns a functoid that takes one argument (the second one) by invoking the partial function call operator in the original functoid on the first argument.

```
template <class In1, class In2, class Out>
class Curry1
    : public FunImpl<In1, Fun<In2,Out> >
{
public:
    Curry1(const Fun2<In1,In2,Out>& fun)
        : theFun(fun) { }
    virtual Fun<In2,Out> operator()
        (In1 arg1) const
    { return theFun(arg1); }
    virtual FunImpl<In2, Fun<In2,Out> >*
    copy() const
    {
        return new Curry1<In1,In2,Out>
            (theFun);
    }
private:
    const Fun2<In1,In2,Out> theFun;
};

template <class In1, class In2, class Out>
Fun<In1, Fun<In2,Out> >
Fun2<In1,In2,Out>::curry1() const
{
    return new Curry1<In1,In2,Out>(*this);
}
```

Adding methods to functoids

The last requirement addresses the extensibility of functoids. We will want to add application-specific member functions to the basic functionality provided by functoids. This can be done by deriving a class `UserFun` from the envelope class `Fun` and a class `UserImpl` from the abstract letter class `FunImpl`. Similarly to the function call operator, the additional member function `f` is implemented in `UserFun` as a wrapper that invokes the real one in `UserImpl`. We are facing a minor problem: not only is the pointer `impl` to the letter object private in class `Fun`, but it also is of class `FunImpl`, which does not have the new member function. We solve this problem by making `impl` protected in `Fun` and casting it to class `UserImpl` in the member function `f`. This cast is safe, since it is hidden from the user of the class.

The following example of a class `Cont` for continuations illustrates this requirement. Besides application to a consumer object of some other class `Consumer`, a continuation supports the method `done` to check whether the continuation has finished. We therefore make the envelope class `Cont` a subclass of `Fun` and the associated letter class `ContImpl` a subclass of `FunImpl`, each instantiat-

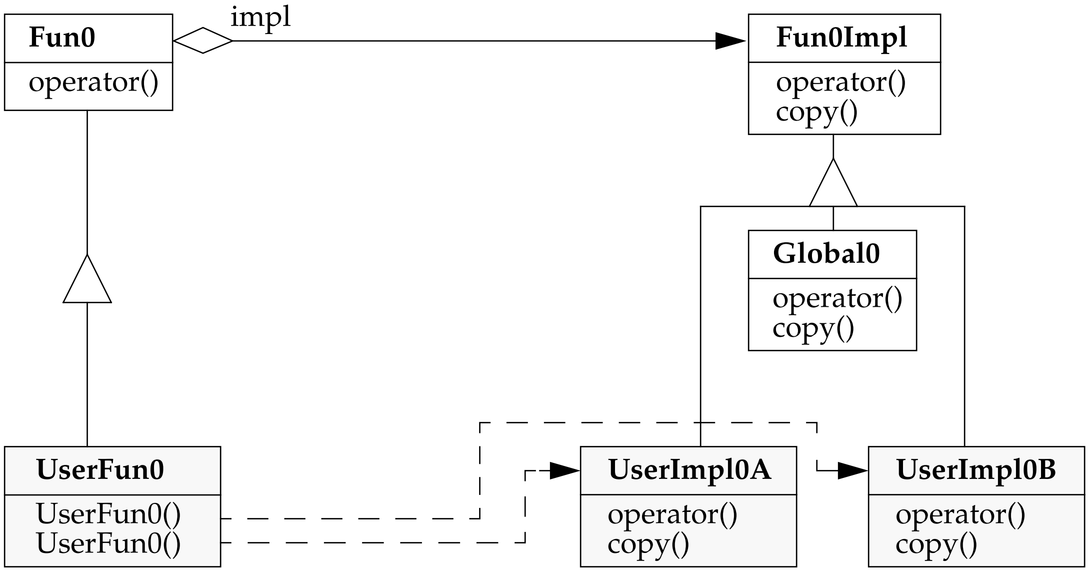

Figure 1: The functoid framework for zero arguments

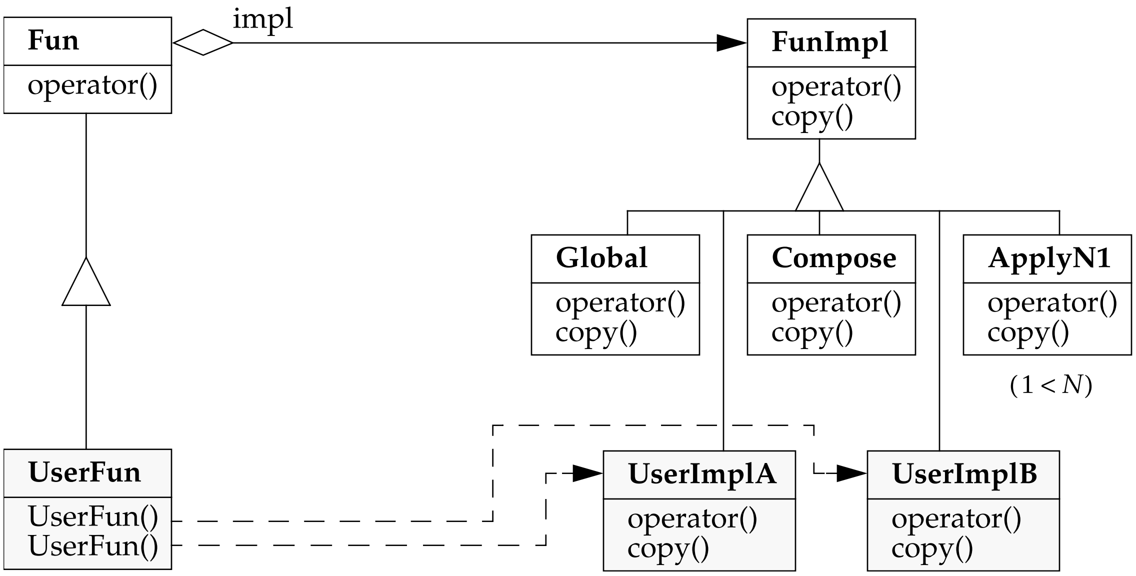

Figure 2: The functoid framework for one argument

ed with appropriate argument and result types. We extend the functionality of `Fun` and `FunImpl` by adding the member function `done` in the subclasses. The member function `done` in class `Cont` first casts the pointer `impl` to class `ContImpl` and then invokes the member function `done` in class `ContImpl`.

```
class ContImpl
    : public FunImpl<const Consumer&, bool>
{
public:
    virtual bool done() const { ... }
};
```

```
class Cont
    : public Fun<const Consumer&, bool>
{
public:
    bool done() const
    { return ((ContImpl*) impl)->done(); }
    ...
};
```

The structure of the functoid framework

Since the framework uses the envelope/letter idiom, it consists of separate abstraction and implementation class hierarchies. There are sub-frame-

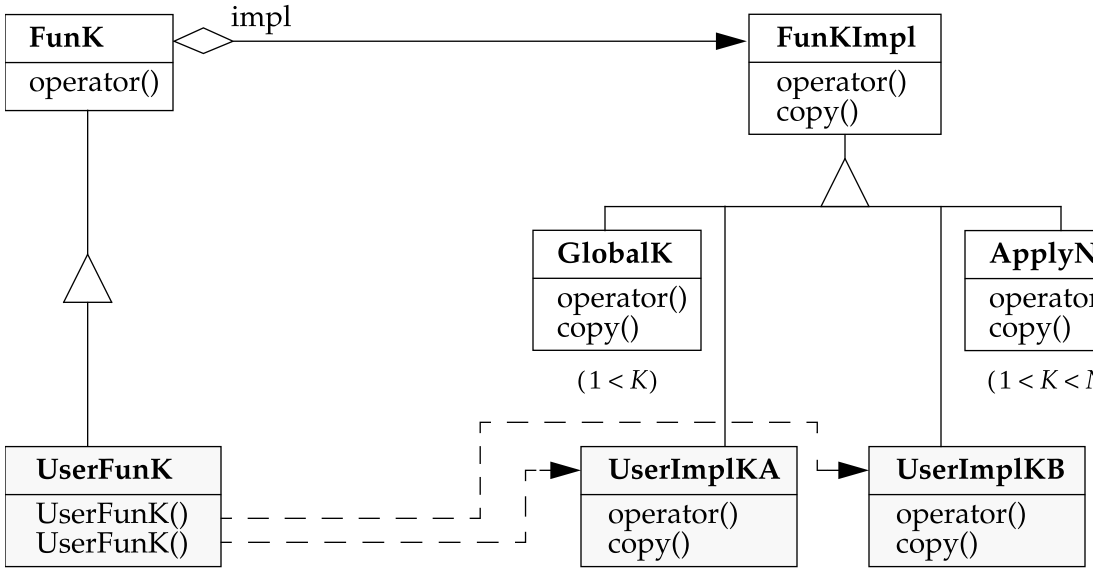

Figure 3: The functoid framework for two or more arguments

works for functoids of zero, one, and more arguments. We first describe the case of a single argument. The framework provides an abstraction class `Fun`, which is the class for functoids in user programs. The framework also provides an abstract implementation class `FunImpl`, from which users derive their own implementations of functoids by overriding the function call operator. Users of the framework derive classes `UserFun` from `Fun` to add constructors that instantiate the user-defined functoid implementation classes `UserImplA`, `UserImplB`, and so on, derived from `FunImpl`. The framework predefines several functoid implementation classes: `Global` implements wrappers around nonmember functions; `Compose` implements functoids resulting from composition; `Apply21`, `Apply31`, and so on, implement functoids resulting from partial application of functoids with more than one argument such that the resulting functoid takes the remaining single argument; finally, `Curry1` implements functoids resulting from "currying" functoids with more than one argument with respect to the first argument.

The sub-frameworks for two or more arguments have a similar structure. In the case of K arguments, there is an abstraction class `FunK` and an abstract implementation class `FunKImpl`. As in the case of a single argument, users derive from both framework classes. Again, there are various predefined functoid implementation classes: `GlobalK` implements wrappers around nonmember functions of K arguments. For $N > K$, `ApplyNK` implements functoids resulting from partial application of a functoid of N arguments to $N - K$ arguments. Since the resulting functoids take the remaining K arguments, the class `ApplyNK` is a subclass of `FunKImpl`. For $K \geq 2$ the class `CurryK` describes curried functoids that take K arguments one at a time. The actual currying is carried out by the corresponding member function `curryK` in the class `FunN`, where $N > K$. As function composition is defined only for functions of one argument, we do not consider it for $K \neq 1$.

We have not yet addressed the case of functions of zero arguments. We could treat functions without arguments as functions with one dummy argument of an enumerated type `unit` with a single value, but this approach would cause difficulties when creating wrappers for global functions with no arguments. We therefore provide a separate, simple sub-framework for this case consisting of only three class templates, `Fun0`, `Fun0Impl`, and `Global0`, whose roles are similar to the corresponding classes in the other cases. These class templates are parameterized only by the result type of the function.

We illustrate the structure of the framework in the notation used by Gamma et al. [GHJV93], which is an extension of the OMT (Object Modeling Technique) notation [R+91]. The framework for zero arguments is shown in Figure 1. The framework for one argument is shown in Figure 2. The framework for two or more arguments is shown in Figure 3, where K is the number of arguments. For simplicity, the classes `CurryK` are not shown.

A small example

Now that we have described the framework in detail, it is time to look at an example that illustrates the various features. Besides composition and partial application, the following example illustrates conversion from and to nonmember functions; we present the implementation of conversion to nonmember functions in Section 4. The functoid `Add` simply adds two numbers.

```
int add7(int x) { return x + 7; }

int callf(int(* f )(int)) { return f(9); }

class Add : public Fun2Impl<int,int,int>
{
public:
    virtual int operator()(int x, int y)
        const
    { return x + y; }
    virtual Fun2Impl<int,int,int>* copy()
        const
    { return new Add; }
};

main()
{
    // conversion from nonmember function
    const Fun<int, int> f(add7);

    const Fun2<int,int,int> g(new Add);

    cout << add7(3) << endl;
    cout << callf(add7) << endl;

    // f and add7 are now equivalent
    cout << f(3) << endl;
    // conversion to nonmember function
    cout << callf(f) << endl;

    // partial application
    cout << g(11)(3) << endl;
    // conversion to nonmember function
    cout << callf(g(11)) << endl;

    // composition and partial application
    cout << compose(f,g(11))(5) << endl;

    // currying and composition
    cout << compose(g.curry1(),f)(11)(5)
         << endl;
}
```

3 Case Study: The Same-Fringe Problem

The purpose of this section is threefold. First, it demonstrates how functional programming styles can be incorporated directly in C++ programs. Second, it serves as a case study that shows the practical usefulness of our system. Third, it disproves claims that this style of programming is not supported by C++ [Bak93].

The same-fringe problem

The *fringe* of a finite tree is the enumeration of its leaves in left-to-right order. The *same-fringe* problem is the problem of deciding whether two finite trees have the same fringe. In practice, this problem occurs when comparing for equality two trees that store data only in their leaves. A brute-force solution to this problem would involve generating the fringe of each tree as a list and then comparing the two lists for equality. This shortcoming of this solution is that it goes through considerable work to construct the entire lists although there might be a mismatch at the beginning of the lists. We could do slightly better by constructing the fringe of only one tree and iterating through the other tree.

A far better solution to the same-fringe problem is to compare the first leaf of each tree and continue only if they match. Such a solution could be expressed in terms of coroutines, which would need unbounded storage to keep track of the current path in the tree. These coroutines could be modeled by external iterators in C++. The drawback of this approach is that the tree traversal has to be made explicit instead of implicit and recursive.

A solution in a functional language

In a functional language, this problem could be solved elegantly in terms of *lazy streams* [FW76]. A lazy stream is a recursive data structure that is either an empty stream or a data item paired with a function that evaluates to another stream when invoked. This technique allows us to *delay* the generation of the entire fringe: we seemingly construct the fringe like an ordinary list, but the actual construction is performed on demand. In the functional language ML [MTH90], a data structure for lazy streams could be defined as follows. The two cases are called `Nil` and `Cons` in analogy to ordinary lists in functional languages. In ML, functions without arguments take a single argument of type `unit`.

```
datatype 'a Stream =
    Nil
  | Cons of 'a * (unit -> 'a Stream)
```

We first deal with the question of generating the fringe of a binary tree in form of a lazy stream. A binary tree is either a leaf containing an item or a node joining two subtrees together.

```
datatype 'a Tree =
    Leaf of 'a
  | Node of 'a Tree * 'a Tree
```

We now generate the fringe of a tree recursively. If the tree is a leaf, then the fringe is simply the pair of the item and a function evaluating to an empty stream. Otherwise the tree is a node, and the fringe is the concatenation of the fringes of the subtrees. We actually concatenate two functions that evaluate to fringes when invoked to delay generating the entire fringes until requested. ML uses pattern matching to examine the structure of function arguments. The form "`fn () => expr`" is used to create an anonymous function closure on the fly. The comments identify the three different cases of anonymous functions we are creating.

```
fun fringe (Leaf x) =
    Cons(x, fn () => Nil)       (* Case 1 *)
  | fringe (Node(l,r)) =
    concat (fn () => fringe l)
           (fn () => fringe r) (* Case 2 *)
```

The concatenation of the two functions follows. If the first function evaluates to an empty stream, the fringe is simply the invocation of the second function. Otherwise the fringe is the first item of the first fringe paired with the concatenation of the rest of the first fringe and the second fringe:

```
fun concat f g =
    case f () of
      Nil        => g ()
    | Cons(x, h) =>            (* Case 3 *)
          Cons(x, fn () => concat h g)
```

The next job is to compare two lazy streams for equality. If both are empty, then they are equal. Otherwise, their first items have to match and the remaining streams have to be equal. In all other cases, the two streams are not equal. The following recursive function captures this notion of equality:

```
fun eq Nil             Nil            =
    true
  | eq (Cons(v1, f1)) (Cons(v2, f2)) =
    (v1 = v2) andalso eq (f1 ()) (f2 ())
  | eq s1             s2             =
    false
```

Now we are ready to define the samefringe function for two trees:

```
fun samefringe t1 t2 =
    eq (fringe t1) (fringe t2)
```

For example, among the following three trees, `t1` and `t2` have the same fringe, although they do not have the same shape, whereas `t0` has a different fringe:

```
val t0 = Node(Node(Leaf 3, Leaf 4),
              Node(Leaf 5, Leaf 7))
val t1 = Node(Node(Leaf 3, Leaf 4),
              Node(Leaf 5, Leaf 6))
val t2 = Node(Node(Node(Leaf 3, Leaf 4),
                   Leaf 5),
              Leaf 6)
```

Translating the solution to C++

We show how to translate the ML solution directly into C++ using the functoid framework. For the sake of simplicity, we deal only with integer items, but we could also have used templates for the various classes. Assume we are given a tree class with the following public member functions:

```
class Tree
{
public:
    int label() const;
    bool isleaf() const;
    const Tree& left() const;
    const Tree& right() const;
    ...
};
```

Our first task is to express lazy streams in C++. One approach to representing a recursive data structure in C++ is as a tagged union, using an enumerated tag field to indicate which of the cases an object belongs to and providing data members for all components of the data structure. We choose a better, more object-oriented approach that models each case of the data structure as a different subclass of a class for the data structure itself. To facilitate passing streams by value, we again employ the envelope/letter idiom. The class `Stream` becomes the envelope class, and we have an abstract letter class `StreamImpl` with concrete subclasses `NilStream` and `ConsStream` for the two cases of the data structure. We first present the class `Stream`.

```
class Stream
{
private:
    StreamImpl* theStream;
```

We need constructors for both cases, a copy constructor, and a destructor. The constructors take as arguments the components of the corresponding cases of the data structure. We assume a forward declaration of the class `Delay` for functions evaluating to streams.

```
public:
    Stream();
    Stream(int hd, const Delay& tl);
    Stream(const Stream& s)
        : theStream(s.theStream->copy()){ }
    ~Stream() { delete theStream; }
```

Now we need to design an interface for the stream class that allows us to distinguish between the two alternatives and to extract the components in the second case. The function `empty` tells us whether a stream is empty; in the nonempty case, `head` extracts the item, and `tail` extracts the function.

```
    bool empty() const
```

```
    { return theStream->empty(); }
    int head() const
    { return theStream->head(); }
    const Delay& tail() const
    { return theStream->tail(); }
    bool operator==(const Stream& s) const;
    Stream& operator=(const Stream& s);
};
```

Although we cannot actually implement the constructors, the destructors, and the equality operator until the class `Delay` is fully defined, we give their definitions at this point. The equality operator is a straightforward translation of the ML function `eq` given above.

```
Stream::Stream()
    : theStream(new NilStream) { }
Stream::Stream(int hd, const Delay& tl)
    : theStream(new ConsStream(hd, tl)) { }

bool Stream::operator==(const Stream& s)
    const
{
    if (empty() && s.empty())
        return true;
    else if (empty() || s.empty())
        return false;
    else
        return head() == s.head() &&
            tail()() == s.tail()();
}
```

Next, we present the abstract letter class `StreamImpl`. Its pure virtual member functions correspond to the member function of class `Stream`.

```
class StreamImpl
{
public:
    virtual ~StreamImpl() { }

    virtual bool empty() const = 0;
    virtual int head() const = 0;
    virtual const Delay& tail() const = 0;

    virtual StreamImpl* copy() const = 0;
};
```

We now define the two subclasses corresponding to the two cases of the data structure. These subclasses implement the pure virtual member functions defined in class `StreamImpl`. For brevity, we omit the `copy` member function, which simply duplicates the receiver. A `NilStream` is always empty and does not have a defined head or tail.

```
class NilStream : public StreamImpl
{
public:
    virtual bool empty() const
    { return true; }
    virtual int head() const { abort(); }
    virtual const Delay& tail() const
    { abort(); }
};
```

A `ConsStream` is never empty. The `head` and `tail` member functions return the corresponding data members, a number and a function, respectively.

```
class ConsStream : public StreamImpl
{
public:
    ConsStream(int x, const Delay& f)
        : hd(x), tl(f) { }
    virtual bool empty() const
    { return false; }
    virtual int head() const { return hd; }
    virtual const Delay& tail() const
    { return tl; }
private:
    int hd;
    Delay tl;
};
```

Now we must define the class `Delay`, which in turn depends on the stream class. We integrate this class in the functoid framework. The class `Delay` is a subclass of an appropriate instance of the class template `Fun0`, and the implementations of `Delay` will be subclasses of instances of the class template `Fun0Impl`. The purpose of introducing the class `Delay` is to capture the mutual dependency with the class `Stream` and to introduce appropriate constructors for each implementation of this class that we want to create. In the ML solution above we identified three cases of anonymous function closures that correspond to three implementations of the class `Delay`.

```
class Delay : public Fun0<Stream>
{
public:
    Delay();
    Delay(const Delay& f, const Delay& g);
    Delay(const Tree<int>& t);
};
```

We are going to implement the three cases as subclasses of `Fun0Impl`. We again omit the `copy` member function. Case 1 is a function of the form "`fn () => Nil`" evaluating to an empty stream. It is represented by the following functoid:

```
class EmptyDelay : public Fun0Impl<Stream>
{
public:
    virtual Stream operator()() const
    { return Stream(); }
};
```

Case 2 is a function of the form "`fn () => fringe t`" evaluating to the fringe of a tree. The corresponding functoid `FringeDelay` stores the tree `t` and invokes the function `fringe`, a direct translation of the corresponding ML function.

```
Stream fringe(const Tree& t)
{
    if (t.isleaf())
        return Stream(t.label(), Delay());
    else
        return concat(Delay(t.left()),
                      Delay(t.right()));
}
```

```
class FringeDelay : public Fun0Impl<Stream>
{
public:
    FringeDelay(const Tree& t)
        : tree(t) { }
    virtual Stream operator()() const
    { return fringe(tree); }
private:
    const Tree& tree;
};
```

Case 3 is a function of the form "`fn () => concat g h`". The associated functoid `ConcatDelay` stores the two functions evaluating to the streams to be concatenated and invokes the function `concat`, again a translation of the corresponding ML function.

```
Stream concat(const Delay& f,
              const Delay& g)
{
    Stream s = f();
    if (s.empty()) return g();
    else return Stream(s.head(),
                       Delay(s.tail(), g));
}
```

```
class ConcatDelay : public Fun0Impl<Stream>
{
public:
    ConcatDelay(const Delay& f,
                const Delay& g)
        : fdelay(f), gdelay(g) { }
    virtual Stream operator()() const
    { return concat(fdelay, gdelay); }
private:
    Delay fdelay, gdelay;
};
```

Finally, we give the implementations of the three constructors for the class `Delay`. Each constructor creates an instance of the corresponding implementation class of the class `Delay`.

```
Delay::Delay()
    : Fun0<Stream>(new EmptyDelay)
{ }
Delay::Delay(const Delay& f,
             const Delay& g)
    : Fun0<Stream>(new CombineDelay(f,g))
{ }
Delay::Delay(const Tree<int>& t)
    : Fun<unit,Stream>(new FringeDelay(t))
{ }
```

We can now determine whether two trees have the same fringe by generating the corresponding streams and checking them for equality.

```
bool samefringe(const Tree& t1,
                const Tree& t2)
{ return fringe(t1) == fringe(t2); }
```

We extend the tree class in two ways. If we define equality of trees as having the same fringe, we can add the following equality operator to the class `Tree`:

```
bool Tree::operator==(const Tree& t) const
{ return fringe(*this) == fringe(t); }
```

Furthermore, we can enhance the class `Tree` with an external (active) iterator class that traverses the fringe of the tree. This class `TreeIterator` enables us to define more than one iterator on the same tree.

```
class TreeIterator
{
public:
    TreeIterator(const Tree& t)
        : theTree(t) { restart(); }
    operator bool() const
    { return ! theFringe.empty(); }
    int current() const
    { return theFringe.head(); }
    void next()
    {
        assert(! theFringe.empty());
        theFringe = theFringe.tail()();
    }
    void restart()
    { theFringe = fringe(theTree); }
private:
    const Tree& theTree;
    Stream theFringe;
};
```

The `next` function uses the assignment operator for the class `Stream`, which we define using the `copy` function.

```
Stream& Stream::operator=(const Stream& s)
{
    if (this != &s)
    {
        delete theStream;
        theStream = s.theStream->copy();
    }
    return *this;
}
```

4 Converting Functoids to Ordinary Functions

The framework presented in Section 2 falls short of the requirement that functoids be convertible to ordinary nonmember functions. This shortcoming stems from a fundamental difference between member functions and nonmember functions, which precludes us from simply using a pointer to the function call operator of a functoid as an ordinary function.

The heterogeneity problem

The fundamental difference between member functions and nonmember functions was recognized by Young [You92] and is called the *heterogeneity problem* by Dami [Dam94]. Technically, a call to a nonmember function requires a stack pointer to store the actual arguments and the address of the function to be called. A member function invocation, on the other hand, requires a stack pointer to store the arguments, the address of the member function, and the address of the receiver. This fundamental difference in the calling mechanism makes it impossible to use a member function where a nonmember function is expected, for example, as a callback from an existing class library.

The proposed solutions [Fek91, You92, CL95] require that the programmer writes a nonmember function that explicitly invokes the C++ member function from a specific receiver. This solution is generally not very good because the programmer has to write a wrapper for every combination of a member function and a receiver to be used as a callback. More seriously, this solution does not work at all for the framework because we create functoids on the fly and thus cannot anticipate what wrappers to provide.

The solution using partial binding

Rescue comes in the form of a solution proposed and implemented by Dami [Dam94], which addresses a more general partial binding problem. In this solution, when we perform a partial binding, we create a data structure that stores the address of the function, the arguments, and code to complete the bindings and invoke the function later. This approach is compiler- and machine-dependent; it currently works with the GNU CC compiler [Sta94] on NeXT and Sparc architectures, but could be ported to other languages, compilers, or architectures. A similar mechanism that maps Scheme closure objects to C functions is described by Rose and Muller [RM92]. The ObjectKit system for ParcPlace Smalltalk allows passing Smalltalk objects, including closures, to C functions [RM92, quoting P. Deutsch].

From the programmer's perspective, Dami's mechanism consists of the function `curry`, whose arguments are a pointer to the memory where the data structure should be allocated, the function to be invoked, the total number of arguments, and the number of arguments supplied here, and those arguments.

```
typedef void*(* anyFunc )();

extern anyFunc curry(void* mem, anyFunc f,
                     int nargs, int cargs, ...);
```

This mechanism extends to object-oriented languages in the sense that the receiver of a message is an (implicit) first argument to the method invoked. We can thus convert a member function to a nonmember function by partial application to the receiver `this`. While the mechanism itself is not type-safe, we safely hide it inside the framework, and the user only sees it as a type conversion operator of functoids back to nonmember functions. We describe how the mechanism is implemented for functoids with a single argument; the implementation for functoids with more arguments is analogous. The class template `FunImpl` gets an additional member function that performs the conversion of its function call operator to a nonmember function.

```
typedef Out(* FunPtr )(In);

virtual FunPtr FunImpl<In,Out>::cfun()
const
{
    return FunPtr(curry(0,
        anyFunc(this->operator()),
                3, 1, this));
}
```

The class template `Fun` is extended by a type conversion operator that invokes `cfun` on the implementation of the functoid. To make sure that the conversion is executed only once, we store the resulting function pointer in an additional data member `fun` of `Fun`, which the constructors initialize to the null pointer. The associated data must be deallocated using the `free` function when the functoid is destructed. The new members of `Fun` are as follows.

```
template <class In, class Out> class Fun
{
public:
    typedef Out(* FunPtr )(In);
    ...
    ~Fun()
    { delete impl; if (fun) free(fun); }
    operator FunPtr() const
    {
        if (fun == NULL)
            ((Fun<In,Out>*) this)->fun =
                impl->cfun(); return fun;
    }
    ...
```

```
    private:
        FunPtr fun;
    };
```

Now our conversion requirement is satisfied both ways, and functoids and nonmember functions are indistinguishable to the user. The example at the end of Section 2 illustrates this feature.

5 Conclusion

We have presented a type-safe generalized framework that supports higher-order functional programming styles within C++ programs. The framework is implemented entirely in the form of C++ class templates, except for a compiler- and machine-dependent mechanism for converting member functions to nonmember functions [Dam94]. The framework could be translated to other object-oriented languages that support both inheritance and genericity.

The main issues in the assessment of our framework are expressiveness and efficiency. To address the first issue, we compare our framework to existing functional programming languages.

It is a fundamental limitation of most class-based object-oriented languages that each distinct behavior must be given a class name [Ros95a]. Consequently, our framework does not provide a mechanism for creating anonymous function closures on the fly. This is in contrast to functional languages, in which anonymous closures are routinely passed to and returned from functions. Rose [Ros95b] describes an extension of C++ with parameterless anonymous functions called *thunks*; a thunk can be converted to a parameterized function by specifying which variables used in the body of the thunk are to be treated as parameters.

Another limitation of the functoid idiom in general, not just of the functoid framework, is that the programmer must establish and maintain an explicit correspondence between variables used in the body of the closure and instance variables of the functoid. By contrast, functional and other languages with block structure and nested functions, such as Algol or Pascal, automatically capture all local variables that are used in the closure. Breuel [Bre88] solves this shortcoming in C and C++ by allowing functions to be nested. Thunks [Ros95b] provide a solution as well.

Another drawback of the functoid framework stems from the way type information is required in instantiations of C++ class templates. While the examples presented in this paper do not require lengthy type parameters, the type information required in more complex applications of the framework is likely to get out of hand, especially when higher numbers of arguments are involved. Dami [Dam95] suggests extending the compiler to keep track of the required type parameters automatically.

There are several sources of inefficiency in the framework as compared to typical implementations of functional languages. First, we use call-by-value to facilitate memory management. This approach requires a considerable amount of copying, depending on the size of the functoid implementations involved. The problem could be addressed by improving the memory management strategy, for example, by using garbage collection for functoids. Memory management could still be hidden from the user by overloading the `new` and `delete` operators for functoids. Second, the structure of the framework requires a virtual function call operator that is overridden in the user classes to allow dynamic selection of the appropriate functoid implementation. This problem is inherent in the design of the framework and has no simple solution. Third, unlike in functional languages, function closures are controlled by the programmer instead of the compiler. This precludes the sort of optimizations a compiler of a functional language would apply.

Other approaches that combine functional languages and C++ include an interpreter accessible within C++ [Kla93] and an interpreter written in C++ [RK88]. A detailed comparison with our work would go beyond the scope of this paper. While the translation outlined informally in Section 3 is not suitable at present as an efficient implementation of functional languages, the paper demonstrates that the framework provides access to various functional idioms within object-oriented languages.

Acknowledgments

Special thanks go to Gerald Baumgartner for his careful review of the initial version of this paper, to Laurent Dami for helpful comments and for providing his Curry code, to Thomas Kühne for stimulating email discussions on this and related material, and to John Rose for valuable comments and for sharing his insights on thunks with me.

References

[App89] Apple Computer, Inc., Cupertino, CA. *Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.

[Bak93] H. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, July 1993.

[Bor94] Borland, Inc. *Borland C/C++ 4.0 Reference Manual*, 1994.

[Bre88] T. Breuel. Lexical closures for C++. In *Proc. USENIX C++ Conf.*, pages 293–304, Denver, CO, October 1988.

[CL95] M. Cline and G. Lomow. *C++ FAQs*. Addison-Wesley, 1995.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[Dam94] L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Université de Genève, April 1994.

[Dam95] L. Dami. Adding closure support to the C++ compiler. Personal communication, March 1995.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Fek91] J. Fekete. WWL, a widget wrapper library for C++, 1991. Laboratoire de Recherche en Informatique, Orsay.

[FW76] D. Friedman and D. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1993.

[HS87] S. Harbison and G. Steele. *C: A Reference Manual*. Prentice-Hall, 2nd edition, 1987.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[Kla93] H. Klagges. A functional language interpreter integrated into the C++ language system. Master's thesis, Balliol College, University of Oxford, University Computing Laboratory, September 1993.

[Küh95] T. Kühne. Inheritance versus parameterization. In Christine Mingins and Bertrand Meyer, editors, *Proc. Technology of Object-Oriented Languages and Systems (TOOLS Pacific '94)*, pages 235–245, Prentice Hall International, Inc., London, 1995. Prentice-Hall. For correct version ask author; proceedings contain corrupted version.

[LCI+92] M. Linton, P. Calder, J. Interrante, S. Tang, and J. Vlissides. *InterViews 3.1 Reference Manual*. CSL, Stanford University, 1992.

[Mey92] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Red95] U. Reddy. The design of Core C++. Unpublished draft, March 1995.

[RK88] V. Russo and S. Kaplan. A C++ interpreter for Scheme. In *Proc. USENIX C++ Conf.*, pages 95–108, Denver, CO, October 1988.

[RM92] J. Rose and H. Muller. Integrating the Scheme and C languages. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 247–259, San Francisco, 1992.

[Ros95a] J. Rose. C closures. Personal communication, April 1995.

[Ros95b] J. Rose. Functional programming and call-by-name in C++. Personal communication, April 1995.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Sta94] R. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, September 1994. Available as part of the GCC-2.6.3 distribution.

[VL90] J. Vlissides and M. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ — An object-oriented application framework in C++. In *Proc. ACM Conf. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 46–57, San Diego, CA, 1988.

[You92] D. Young. *Object-Oriented Programming with C++ and OSF/Motif*. Prentice-Hall, 1992.