

FC++: Functional Tools for Object-Oriented Tasks

Yannis Smaragdakis and Brian McNamara

College of Computing

Georgia Institute of Technology

Atlanta, GA 30332

{yannis,lorgon}@cc.gatech.edu

<http://www.cc.gatech.edu/~yannis/fc++>

Abstract

FC++ is a library for programming functionally in C++. Compared to other C++ functional programming libraries, FC++ is distinguished by its powerful type system which allows manipulating parametrically polymorphic functions (e.g., passing them as arguments to other functions and returning them as results).

In this paper, we show how FC++ can be used in common OO programming tasks. We demonstrate FC++ implementations of several common design patterns (Adapter, Builder, Command, and more). Compared to conventional C++ implementations of these patterns, our implementations are either simpler (in that fewer classes/dependencies are needed), more efficient, or more type-safe (thanks to parametric polymorphism and type-inference).

1 Introduction

Functional and object-oriented programming are the most active fields of research in programming languages and methodologies. Several pieces of work have attempted to connect the two paradigms. Among them are the Pizza language [18], extending Java, as well as libraries for programming functionally in C++ [11][12][15][21]. FC++ [16] is one such library, distinguished from all others by its powerful type system: FC++ allows the programmer to define and fully manipulate parametrically polymorphic functions. The conventional C++ way of representing polymorphic functions is via function templates (e.g., see the C++ Standard Library [20]). Nevertheless, function templates suffer severe limitations—e.g., they cannot be passed as parameters to other functions or returned as results. FC++ polymorphic functions overcome these limitations. This allows FC++ to re-implement

straightforwardly many common functional operators (a large part of the Haskell Standard Prelude [19]). The library is currently quite rich in functionality and has an efficient implementation.

In a previous paper [16], we introduced FC++ and its innovative type system, showed the components of the library, and demonstrated how its implementation is more efficient than previous similar attempts. In this paper we show how to leverage the library to create simple, efficient, and safe implementations of common OO design patterns [8].

Certainly a lot has been written about language support for implementing design patterns (e.g., [4][6]), functional techniques in OO programming, etc. Some of the approaches in the literature are even very close in philosophy to our work. For instance:

- Alexandrescu [2] demonstrates how the meta-programming capabilities of the C++ language can be used to yield elegant pattern implementations.
- Kühne’s dissertation proposes several patterns inspired by functional programming [14].
- Using functional techniques (higher-order functions) to implement the Observer and Command patterns is common—in fact, even standard practice in Java and Smalltalk.
- The benefits of polymorphic and higher-order functions have often been discussed in the functional programming literature [22].

Therefore, by necessity, part of our material is not conceptually novel. In particular, Section 3 shows that FC++ offers a rich framework for OO tasks, but similar results can be obtained with other lan-

guages or libraries (although, among C++ approaches, FC++ is arguably the most complete).

Section 4, however, shows new implementations of common patterns by concentrating on some of the more novel elements of FC++—mainly type inference and its ability to manipulate polymorphic functions. We see how parametric polymorphism can find its way into some design patterns, how FC++ can handle such tasks, and how using parametric polymorphism results into more generic code. [Note that we do not discuss improvements in design patterns’ implementations by the mere addition of parametric typing (e.g., C++ class templates) in a language. These are well understood and are even discussed in [8], as implementation suggestions.]

Although some C++ background is required for much of the paper, we believe that the principles are interesting even to non-C++ programmers. In particular, the paper offers insights for language designers and shows a platform where both sub-type polymorphism and parametric polymorphism with type inference are readily available as complementary tools for problem solving.

2 Background: Functional Programming with FC++

We begin by introducing FC++. We divide this introduction into two parts. Section 2.1 gives an overview of the FC++ library, from the perspective of a user of the library components. Section 2.2 describes a few of the important implementation techniques of the FC++ library. These will be employed in Section 4 and are necessary for the user who intends to write polymorphic functions using FC++. The ability to manipulate polymorphic functions is what sets FC++ apart from all other functional programming libraries in C++. For a more complete introduction to FC++, the reader should refer to [16].

2.1 FC++ Basics

In FC++, we express functions as instances of classes that follow certain conventions. We call such classes *functoids*. The key advantage to using functoids (rather than C++ functions or function templates) is that we can pass them as parameters

and return them as results—even if they are polymorphic. There are two kinds of functoids: *direct* and *indirect*. Direct functoids are the usual representation for functions in FC++. Direct functoids can be either monomorphic or polymorphic. Indirect functoids, on the other hand, must always be monomorphic but can express *first-class* functions. That is, with indirect functoids, we can define variables that range over all functions with the same type signature. Thus, indirect functoids can be viewed as indirect function references, much like C/C++ function pointers. In addition to direct and indirect functoids, FC++ provides a number of useful operations for creating functoids, composing them, specializing them, etc. We shall now discuss a few of the key components of FC++ in more detail.

Indirect functoids are represented as the `FunN` family of classes. `FunNs` specify function signatures via template parameters; `N` is the number of arguments. For example, `Fun2<int, char, string>` is the type of a two-argument indirect functoid which takes an `int` and a `char` and returns a `string`. Note that the first `N` template arguments comprise the argument types of the function, and the last template argument is the result type. Thus, the simplest kind of indirect functoid is a `Fun0<void>`—a function that takes no arguments and returns no result. (As we show in Section 3, this simplest functoid plays a significant role in a number of patterns.)

A common way to create indirect functoids is with `ptr_to_fun`. `ptr_to_fun` transforms a normal C++ function into a functoid. Here is a simple example, which also demonstrates how indirect functoids can range over different functions:

```
int i_times( int x, int y ) { return x*y; }
int i_plus( int x, int y ) { return x+y; }
...
Fun2<int, int, int> f;
f = ptr_to_fun(&i_times);
f(3,4); // returns 12
f = ptr_to_fun(&i_plus);
f(3,4); // returns 7
```

Note that `f`’s behavior depends on which functoid it is bound to. This may seem reminiscent of OO dynamic dispatch (where a method call depends upon the dynamic type of the object that the receiver is bound to), and rightly so! There is just

such a virtual method call buried inside the implementation of all indirect functoids.

Indirect functoids are more versatile than function pointers: they employ automatically currying, they can be bound to new function objects that are created on-the-fly, and they exhibit a form of subtype polymorphism (see [16]). What follows will demonstrate some of these features, which have important applications in Section 3 and Section 4.

Currying is a functional technique that allows us to bind a subset of a function's arguments to specific values. For example, we can use `curry` to bind the first argument of `f` to the value 1, creating a new one-argument function:

```
Fun1<int,int> inc = curry2(f,1);
inc(4); // returns 5 - i.e., i_plus(1,4)
```

(The 2 in `curry2` refers to the number of arguments that `f` expects.) In fact, FC++ also allows the currying to be implicit—when a functoid is called with fewer actual arguments than it expects, it returns a curried functoid. For instance, the previous example can be written more simply as:

```
Fun1<int,int> inc = f(1);
inc(4); // returns 5 - i.e., i_plus(1,4)
```

Although this simpler form is what a typical FC++ user would write, we will avoid it in this paper, in order to emphasize that currying is done through polymorphic functions (e.g., `curry2`) that manipulate other (possibly polymorphic) functions.

Functional composition is easily expressed with `compose`:

```
Fun1<int,int> inc2 = compose(inc,inc);
inc2(4); // returns 6 - i.e., inc(inc(4))
```

Currying and composition are among the powerful functional techniques for building new functions on-the-fly.

Unlike indirect functoids, *direct functoids* can be polymorphic. Consider the simple example of a function to create a `std::pair`. (`std::pair` is the template struct in C++ used to represent a pair of values.) The direct functoid `mk_pair` makes a `std::pair` from its two parameters. For example,

```
mk_pair(3,'c')
returns a std::pair structure whose first field is
```

the `int` 3, and whose `second` field is the `char` 'c'. Indeed, the C++ standard defines a template function for the same purpose, which goes by the name `std::make_pair`. However, compared to `mk_pair`, `std::make_pair` suffers extreme limitations, by virtue of being defined as a template function. Template functions cannot be passed as parameters, which means we cannot use the functional techniques mentioned above (i.e., currying and composition) on templates. Direct functoids avoid these limitations. For example, we can say

```
curry2( mk_pair, 3 )
to return a new direct functoid which takes one
argument of any type T, and returns a
std::pair<int,T> whose first field is 3.
```

2.2 Expressing Polymorphic FC++ Functoids

In the previous example, we demonstrated passing a polymorphic functoid to a higher-order function which returned a polymorphic result. How is this accomplished using C++? The trick in FC++ is to use a struct with nested template members for both the actual function as well as an explicit representation of the type signature of functoid, so that we can exploit the language's type inference of function arguments. Thus in FC++ we would define `mk_pair` as:

```
struct MkPair {
    template <class T, class U>
    std::pair<T,U> operator()( T x, U y ) {
        return std::pair<T,U>(x,y);
    }
    template <class T, class U>
    struct Sig
        : FunType<T,U,std::pair<T,U> > {};
} mk_pair;
```

The `operator()` member (the usual way to define a function object in C++) is defined just as we would expect. The key is the `Sig` member. FC++ functoids all have member structs named `Sig` which encode their function signatures. These `Sigs` contain typedefs named `ResultType`, `FirstArgType`, etc., according to FC++ library conventions. To ease the task of defining such `Sig` members, we inherit the generic `FunType` class which defines the typedefs; `FunType` follows the same conventions as the indirect functoid `FunN` classes (the first few template parameters are the argument types and the final template parameter is the result type).

This encoding mechanism is the key that allows FC++ to create higher-order functors that can directly manipulate polymorphic functors. Specifically, other functors can determine what the result type of a particular polymorphic functor would be, for given arguments.

To see how, consider the simple functor `apply`, which applies a binary function to its arguments. That is, `apply(f,x,y)` behaves just as `f(x,y)` does. If `f` is monomorphic, it is easy to implement such a function in C++ (using techniques from the STL [20]). However, suppose we want to use `apply` on a polymorphic function like `mk_pair`—how do we do it? In FC++, we just say:

```
struct Apply {
    template <class F, class X, class Y>
    typename F::Sig<X,Y>::ResultType
    operator()( F f, X x, Y y ) {
        return f(x,y);
    }
    template <class F, class X, class Y>
    struct Sig : public FunType<F,X,Y,
        typename F::Sig<X,Y>::ResultType> {};
} apply;
```

Note that `apply`'s result type depends on both the type of the functor and the types of arguments it receives;

```
F::Sig<X,Y>::ResultType
expresses this. Thus, for instance,
apply( mk_pair, 3, 'c' )
will return a
MakePair::Sig<int,char>::ResultType
which is just a typedef for
std::pair<int,char>.
```

Note that `apply` also has its own nested `Sig` member, which means that `apply` itself could be manipulated by other higher-order functions.

The process of inferring a function's type from its arguments is called *type inference*. Type inference is automatic in modern functional languages (e.g., Haskell and ML). Type inference in C++ is semi-automatic: the argument types can be inferred from the actual arguments, but using these types to infer the return type of a function has to be done “manually”. This is the role that the `Sig` template members play in FC++.

As a more realistic example of type inference, consider the `compose` function applied to two unary

functors `f` and `g` of types `F` and `G`, respectively. `compose(f,g)` returns a (possibly polymorphic) direct functor with the following `Sig` member:

```
template <class T>
struct Sig : public FunType<
    T,
    typename F::Sig<
        typename G::Sig<T>::ResultType>::
        ResultType>
{};
```

That is (take a big breath), the return type of `compose(f,g)` is a functor of a single argument of type `T`, whose return type is the same as that of functor `f` when `f`'s argument type is the same as the return type of functor `g` when `g`'s argument is of type `T`.

Despite FC++'s abilities, it is not a complete functional language with polymorphism and type inference. One of the main drawbacks is that variable types have to be declared explicitly. Although FC++ type inference eliminates the need for typing intermediate results, if the final result of an FC++ expression needs to be stored, the variable should be explicitly typed. This restriction will hopefully be removed with the addition of the `typeof` keyword in the next revision of the C++ standard.

Although the above examples may seem quite complicated, there are not too many useful abstract higher-order functions like `compose` and they are all already pre-defined in FC++. As a result, clients are shielded from most of the complexity. Nevertheless, generic combinators like `curry` and `compose` owe their generality to this mechanism. Thus, most of the FC++ examples we shall see in Section 4 are realizable only because of this unique feature of our library.

3 Reusability with Object-Oriented and Functional Patterns

Functional programming promotes identifying pieces of functionality as just “functions” and manipulating them using higher-order operations on functions. These higher-order functions may be specific to the domain of the application or they may be quite general (like the currying and function composition operations are). Several design patterns [8] follow a similar approach through the

use of *subtype polymorphism*. Subtype polymorphism allows code that operates on a certain class or interface to also work with specializations of the class or interface. This is analogous to higher-order functions: the holder of an object reference may express a generic algorithm which is specialized dynamically based on the value of the reference. Encapsulating functionality and data as an object is analogous to direct function manipulation. Other code can operate abstractly on the object's interface (e.g., to adapt it by creating a wrapper object).

It has long been identified that functional techniques can be used in the implementation of design patterns. For instance, the Visitor pattern is often considered a way to program functionally in OO languages. (The interested reader should see [13] and its references for a discussion of Visitor.) The Smalltalk class `MessageSend` (and its variants, see [3], p.254), the C++ Standard Library functors, Alexandrescu's framework ([2], Ch. 5), etc., are all trying to capture the generic concept of a "function" and use it in the implementation of the Command or Observer pattern. Here we will briefly review some of these well-known techniques, from the FC++ standpoint, by using indirect functoids.

Command. The Command pattern turns requests into objects, so that the requests can be passed, stored, queued, and processed by an object which knows nothing of either the action or the receiver of the action. An example application of the pattern is a menu widget. A pull-down menu, for instance, must "do something" when an option is clicked; *Command* embodies the "something". Command objects support a single method (usually called `execute`). Any state that the method operates on needs to be captured inside a command object.

The motivation for using the Command pattern is twofold. First, holders of command objects (e.g., menu widgets) are oblivious to the exact functionality of these objects. This decoupling makes the widgets reusable and configurable dynamically (e.g., to create context-sensitive graphical menus). Second, the commands themselves are decoupled from the application interface and can be reused in different situations (e.g., the same command can be executed from both a pull-down menu and a toolbar).

Here is a brief example which illustrates how Command might be employed in a word-processing application:

```
class Command {
public:
    virtual void execute()=0;
};

class CutCommand : public Command {
    Document* d;
public:
    CutCommand(Document* dd) : d(dd) {}
    void execute() { d->cut(); }
};

class PasteCommand : public Command {
    Document* d;
public:
    PasteCommand(Document* dd) : d(dd) {}
    void execute() { d->paste(); }
};

Document d;
...
Command* menu_actions[] = {
    new CutCommand(&d),
    new PasteCommand(&d),
    ...
};
...
menu_actions[choice]->execute();
```

The abstract `Command` class exists only to define the interface for executing commands. Furthermore, the `execute()` interface is just a call with no arguments or results. In other words, the whole command pattern simply represents a "function object". From a functional programmer's perspective, `Command` is just a class wrapper for a "lambda" or "thunk"—an object-oriented counterpart of a functional idiom. Indirect functoids in FC++ represent such function-objects naturally: a `Fun0<void>` can be used to obviate the need for both the abstract `Command` class and its concrete subclasses:

```
Document d;
...
Fun0<void> menu_actions[] = {
    curry(ptr_to_fun(&Document::cut), &d),
    curry(ptr_to_fun(&Document::paste), &d),
    ...
};
...
menu_actions[choice]();
```

In this last code fragment, all of the classes that comprised the original design pattern implementation have disappeared! `Fun0<void>` defines a natural interface for commands, and the concrete instances can be created on-the-fly by making indirect functors out of the appropriate functionality, currying arguments when necessary.

The previous example takes advantage of the fact that `ptr_to_fun` can be used to create functors out of all kinds of function-like C++ entities. This includes C++ functions, instance methods (which are transformed into normal functions that take a pointer to the receiver object as an extra first argument—as in the example), class (static) methods, C++ Standard Library `<functional>` objects, etc. This is an example of design inspired by the functional paradigm: multiple distinct entities are unified as functions. The advantage of the unification is that all such entities can be manipulated using the same techniques, both application-specific and generic.

Observer. The *Observer* pattern (a.k.a. *Publish-Subscribe*) is used to register related objects dynamically so that they can be notified when another object's state changes. The main participants of the pattern are a *subject* and multiple *observers*. Observers register with the subject by calling one of its methods (with the conventional name `attach`) and un-register similarly (via `detach`). The subject notifies observers of changes in its state, by calling an observer method (`update`).

The implementation of the observer pattern contains an abstract `Observer` class that all concrete observer classes inherit. This interface has only the `update` method, making it similar to just a single function, used as a callback. In fact, the implementation of the Observer pattern can be viewed as a special case of the Command pattern. Calling the `execute` method of the command object is analogous to calling the `update` method of an observer object.

The FC++ solution strategy for the Observer pattern is exactly the same as in Command. The Subject no longer cares about the type of its receivers (i.e., whether they are subtypes of an abstract `Observer` class). Instead, the interesting aspect of

the receivers—their ability to receive updates—is encapsulated as a `Fun0<void>`. The abstract `Observer` class disappears. The concrete observers simply register themselves with the subject. We will not show the complete code skeletons for the Observer pattern, as they are just specializations of the code for Command. Nevertheless, one aspect is worth emphasizing. Consider the code below for a concrete observer:

```
class ConcreteObserver {
    ConcreteSubject& subject;
public:
    ConcreteObserver( ConcreteSubject& s )
        : subject(s)
    {
        s.attach( curry( ptr_to_fun(
            &ConcreteObserver::be_notified
        ), this ) );
    }

    void be_notified() {
        cout << "new state is" <<
            subject.get_state() << endl;
    }
};
```

Note again how `ptr_to_fun` is used to create a direct functor out of an instance method. The resulting functor takes the receiver as its first parameter. `curry` is then used to bind this parameter. This approach frees observers from needing to conform to a particular interface. For instance, the above concrete observer implements `be_notified` instead of the standard `update` method, but it still works fine. Indeed, we can turn an arbitrary object into an observer simply by making a functor out of one of its method calls—the object need not even be aware that it is participating in the pattern. This decoupling is achieved by capturing the natural abstraction of the domain: the function object.

Summarizing, the reason that `Fun0<void>` can replace the abstract `Observer` and `Command` classes is because these classes serve no purpose other than to create a common interface to a function call. In `Command`, the method is named `execute()`, and in `Observer`, it is called `update()`, but the names of the methods (and classes) are really immaterial to the pattern. Indirect functors in FC++ obviate the need for these classes, methods, and names, by instead representing the core of the

interface: a function call which takes no argument and returns nothing.

C++'s parameterization mechanism lets us extend this notion to functions which take arguments and return values. For example, consider an observer-like scenario, where the notifier passes a value (for instance, a string) to the observer's `update` method, and the `update` returns a value (say, an integer). This can be solved using the same strategy as before, but using a `Fun1<string,int>` instead of a `Fun0<void>`. Again, the key is that the interface between the participants in the patterns is adequately represented by a single function signature;¹ extra classes and methods (with fixed names) are unnecessary to realize a solution.

Virtual Proxies. The Virtual Proxy pattern seeks to put off expensive operations until they are actually needed. For example, a word-processor may load a document which contains a number of images. Since many of these images will reside on pages of the document that are off-screen, it is not necessary to actually load the entire image from disk and render it unless the user of the application actually scrolls to one of those pages. In [8], an `ImageProxy` class supports the same interface as an `Image` class, but postpones the work of loading the image data until someone actually requests it.

In many functional programming languages, the Virtual Proxy pattern is unnecessary. This is because many functional languages employ *lazy evaluation*. This means that values are never computed until they are actually used. This is in contrast to *strict* languages (like all mainstream OO languages), where values are automatically computed when they are created, regardless of whether or not they are used.

Since C++ is strict, FC++ is also strict by default. Nevertheless, a value of type `T` can be made lazy by wrapping the computation of that value in a `Fun0<T>`. This is a common technique in strict functional languages. It encapsulates a computa-

tion as a function and causes the computation to occur only when the function is actually called (i.e., when the result is needed). For instance, in FC++ a call `foo(a,b)` can be delayed by writing it as `curry2(foo, a, b)`. The latter expression will return a 0-argument functoid that will perform the original computation, but only when it is called. Thus, passing this functoid around enables it to be evaluated lazily.

We should mention that FC++ defines some more tools for conveniently expressing lazy computations. First, the `LazyPtrProxy` class in FC++ works as a generic form of the `ImageProxy` mentioned earlier. A `LazyPtrProxy` has the same interface as a pointer to an object, but it does not actually create the object until it is dereferenced. That is, `LazyPtrProxy` is a way to delay *object construction* (as opposed to method calls). Second, FC++ contains an implementation of a *lazy list* data structure. This enables interesting solutions to some problems. For example, to compute the first N prime numbers, we might create an infinite (lazy) list of all the primes, and then select just the first N elements of that list. FC++ lazy lists are compatible with the data structures in the C++ Standard Library and can be processed by a multitude of predefined FC++ functions.

4 Design Patterns and Parametric Polymorphism

In the previous section, we saw how several common design patterns are related to functional programming patterns. All of our examples relied on the use of higher order functions. Another trait of modern functional languages (e.g., ML and Haskell) is support for parametric polymorphism with type inference. Type inference was discussed in Section 2.2: it is the process of deducing the return type of a function, given specific arguments. In this section, we will examine how some design patterns can be improved if they employ parametric polymorphism with type inference and how they can further benefit from the entire arsenal of FC++ techniques for manipulating these polymorphic functions. (The discussion of this section is only relevant for statically typed OO languages, like Java, Eiffel, or C++. The novelties of FC++ are

1. A tuple of indirect functoids can be used if multiple function signatures are defined in an interface; the example in [8] of `Command` used for `do/undo` could be realized in FC++ with a `std::pair<Fun0<void>,Fun0<void>>`, for instance.

in its type system—it has nothing new to offer to a dynamically typed language, like Smalltalk.)

4.1 Parametric vs. Subtype Polymorphism

Design patterns are based on subtype polymorphism—the cornerstone of OO programming. Parametric polymorphism, on the other hand, is not commonly available in OO languages, and, even when it is, its power is limited (e.g., there is no type inference capability). FC++ adds this capability to C++. It is interesting to ask when parametric polymorphism can be used in place of subtype polymorphism and what the benefits will be, especially in the context of design patterns.

Parametric polymorphism is a static concept: it occurs entirely at compile time. Thus, to use a parametrically polymorphic operation, we need to know the types of its arguments at each invocation site of the operation (although the same operation can be used with many different types of arguments). In contrast, subtype polymorphism supports dynamic dispatch: the exact version of the executed operation depends on the run-time type of the object, which can be a subtype of its statically known type.

Therefore a necessary condition for employing parametric polymorphism is to statically know the type of operands of the polymorphic operation at each invocation site. When combined with type inference, parametric polymorphism can be as convenient to use as subtype polymorphism and can be advantageous for the following reasons:

- *No common supertype is required.* The issue of having an actual common superclass or just supporting the right method signature is similar to the *named/structural subtyping* dilemma. All mainstream OO languages except Smalltalk use named subtyping: a type A needs to declare that it is a subtype of B. In contrast, in structural subtyping, a type A can be a subtype of type B if it just implements the right method signatures. The advantage of requiring a common superclass is that accidental conformance is avoided. The disadvantage is that sometimes it is not easy (or even possible) to change the source code of a class to make it declare that it is a subtype of another. For instance, it may be

impossible to modify pre-compiled code, or it may be tedious to manipulate existing inheritance hierarchies, or the commonalities cannot be isolated due to language restrictions (e.g., no multiple inheritance, no common interface signature). Even in languages like Java where a supertype of all types exists (the `Object` type), problems arise with higher-order polymorphic functions, like our `curry` operator. The problem is that an `Object` reference may be used to point to any object, but it cannot be passed to a function that expects a reference of a specific (but unknown) type. Thus, implementing a fully generic `curry` with subtype polymorphism is impossible.

- *Type checking is static.* With subtype polymorphism, errors can remain undetected until runtime. Such errors arise when an object is assumed to be of a certain dynamic type but is not. Since the compiler can only check the static types of objects, the error cannot be detected at compile-time. In fact, for many of the most powerful and general polymorphic operations, subtype polymorphism is impossible to use with any kind of type information. For instance, it would be impossible to implement a generic `compose` operator with subtype polymorphism, unless all functions composed are very weakly typed (e.g., functions from `Objects` to `Objects`). The same is true with most other higher-order polymorphic operations (i.e., functions that manipulate other functions).
- *Method dispatch is static.* Despite the many techniques developed for making dynamic dispatch more efficient, there is commonly a runtime performance cost, especially for hard-to-analyze languages like C++. Apart from the direct cost of dynamic dispatch itself, there is also an indirect cost due to lost optimization opportunities (such as inlining).

4.2 Pattern Examples

Adapter. The Adapter pattern converts the interface of one class to that of another. The pattern is often useful when two separately developed class hierarchies follow the same design, but use different names for methods. For example, one window toolkit might display objects by calling `paint()`,

while another calls `draw()`. Adapter provides a way to adapt the interface of one to meet the constraints of the other.

Adaptation is remarkably simple when a functional design is followed. Most useful kinds of method adaptation can be implemented using the currying and functoid composition operators of FC++, without needing any special adapter classes. These adaptation operators are very general and reusable.

Consider the Command or the Observer pattern. As we saw, in an FC++ implementation there is no need for abstract `Observer` or `Command` classes. More interestingly, the concrete observer or commands do not even need to support a common interface—their existing methods can be converted into functoids. Nevertheless, this requires that the existing methods have the right type signature. For instance, in our `ConcreteObserver` example, above, the `be_notified` method was used in place of a conventional `update` method, but both methods have the same signature: they take no arguments and return no results. What if an existing method has *almost* the right signature, or if methods need to be combined to produce the right signature?

For an example, consider a class, `AnObserver`, that defines a more general interface than what is expected. `AnObserver` may define a method:

```
void update(Time timestamp) { ... }
```

We would like to use this method to subscribe to some other object's service (which we will call the *publisher*) that will issue periodic updates. As shown in the Observer pattern implementation, the publisher expects a functoid object taking no arguments. This is easy to effect by adapting the observer's interface:

```
curry2(ptr_to_fun(&AnObserver::update),  
      this, current_time())
```

(recall that `curry2` is the currying function for 2-argument functoids.) In the above, we used a constant value (the current time) to specialize the `update` method so that it conforms to the required interface. That is, all update events will get the same timestamp—one that indicates the subscrip-

tion time instead of the update time. A better approach is:

```
compose(  
  curry2(ptr_to_fun(&AnObserver::update),  
        this),  
  ptr_to_fun(current_time))
```

In this example we combined currying with function composition in order to specialize the interface. The resulting function takes no arguments but uses global state (returned by the `current_time()` routine) as the value of the argument of the `update` method. In this way, each update will be timestamped with the value of the system clock at the time of the update!

Other parametric polymorphism approaches (e.g., the functional part of the C++ Standard Library [20], or Alexandrescu's framework for functions [2], Ch.5) support currying and composition for *monomorphic* functions. The previous examples demonstrate the value of type inference, which is not unique to FC++. Nevertheless, FC++ also extends type inference to *polymorphic* functions. We will see examples of currying and composition of polymorphic operations in the implementations of the next few patterns.

Decorator. The *Decorator* pattern is used to attach additional responsibilities to an object. Although this can happen dynamically, most of the common uses of the Decorator pattern can be handled statically. Consider, for instance, a generic library for the manipulation of windowing objects. This library may contain adapters, wrappers, and combinators of graphical objects. For example, one of its operations could take a window and annotate it with vertical scrollbars. The problem is that the generic library has no way of creating new objects for applications that may happen to use it. The generic code does not share an inheritance hierarchy with any particular application, so it is impossible to pass it concrete factory objects (as it cannot declare references to an abstract factory class).

This problem can be solved by making the generic operations be parametrically polymorphic and enabling type inference. For instance, we can write a generic FC++ functoid that will annotate a window with a scrollbar:

```

struct AddScrollbar {
    template <class W>
    struct Sig :
        public FunType<W, ScrollWindow<W> *>
    {};

    template <class W>
    typename Sig<W>::ResultType
    operator() (const W& window) const {
        return
            new ScrollWindow<W>(window);
    }
} add_scrollbar;

```

The above decorator functoid can be used with several different types of windows. For a window type `w`, the functoid's return type will be a pointer to a decorated window type: `ScrollWindow<W>`. (In fact, `ScrollWindow` can be a mixin, inheriting from its parameter, `w`.)

Since the functoid conforms to the FC++ conventions, it can be manipulated using the standard FC++ operators (e.g., composed with other functoids, curried, etc.). Composition is particularly useful, as it enables creating more complex generic manipulators from simple ones. For instance, a function to add both a scrollbar and a title bar to a window can be expressed as a composition:

```
compose(add_titlebar, add_scrollbar)
```

instead of adding a new function to the interface of a generic library. Similarly, if the `add_titlebar` operation accepts one more argument (the window title), the currying operation can be used (implicitly in the example below):

```
add_titlebar("Application X Window")
```

The previous examples showed how classes can be statically decorated, possibly with new abilities added to them. Nevertheless, a common kind of decoration is pure wrapping, where the interface of the class does not change, but old operations are extended with extra functionality. Using parametric polymorphism one can write special-purpose polymorphic wrappers that are quite general. These could also be written as C++ function templates, but if they are written as FC++ functoids, they can be applied to polymorphic functoids and they can themselves be manipulated by other functoids (like `curry` and `compose`). Consider, for instance, an instrumentation functoid that calls a one-argument

operation, prints the result of the invocation (regardless of its type) and returns that same result:

```

struct GenericInstrumentor {
    template <class C, class A>
    struct Sig :
        public FunType<C, A,
            C::template Sig<A>::ResultType>
    {};

    template <class C, class A>
    Sig<C,A>::ResultType
    operator() (const C& operation,
        const A& argument ) const {
        Sig<C,A>::ResultType r =
            operation(argument);
        cerr << "Result is: " << r << endl;
        return r;
    }
} generic_instrumentor;

```

`GenericInstrumentor` exemplifies a special-purpose functoid (it logs the results of calls to an error stream) that can be generally applied (it can wrap *any* one-argument function).

Builder. The *Builder* design pattern generalizes the construction process of conceptually similar composite objects so that a generic process can be used to create the composite objects by repeatedly creating their parts. More concretely, the main roles in a Builder pattern are those of a "Director" and a "Builder". The Director object holds a reference to an abstract Builder class and, thus, can be used with multiple concrete Builders. Whenever the Director needs to create a part of the composite object, it calls the Builder. The Builder is responsible for aggregating the parts to form the entire object.

A common application domain for the Builder pattern is that of data interpretation. For instance, consider an interpreter for HTML data. The main structure of such an interpreter is the same, regardless of whether it is used to display web pages, to convert the HTML data into some other markup language or word-processing format, to extract the ASCII text from the data, etc. Thus, the interpreter can be the Director in a Builder pattern. Then it can call the appropriate builders for each kind of document element it encounters in the HTML data (e.g., font change, paragraph end, text strings, etc.).

In the Builder pattern, the Director object often implements a method of the form:

```
void construct(ObjCollection objs) {
    for all objects in objs{
        if (object is_a A)
            builder->build_part_A(object);
        else if (object is_a B)
            builder->build_part_B(object);
        ...
    }
}
```

(Pseudocode shown in italics.) Note that the `build_part` method of the `builder` objects returns no result. Instead, the Builder object aggregates the results of each `build_part` operation and returns them through a method (we will call it `get_result`). This method is called by a client object (i.e., *not* the Director!).

A more natural organization would have the Director collect the products of building and return them to the client as a result of the `construct` call. In an extreme case, the `get_result` method could be unnecessary: the Director could keep all the state (i.e., the accumulated results of previous `build_part` calls) and the Builder could be stateless. Nevertheless, this is impossible in the original implementation of the pattern. The reason for keeping the state in the Builders is that Directors have no idea what the type of the result of the `build_part` method might be. Thus, Directors cannot declare any variables, containers, etc. based on the type of data returned by a Builder. Gamma et al. [8] write: “In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class.”

This scenario (no common interface) is exactly one where parametric polymorphism is appropriate instead of subtype polymorphism. Using parametric polymorphism, the Director class could infer the result types of individual Builders and define state to keep their products. Of course, this requires that the kind of Builder object used (e.g., an HTML to PDF converter, an on-screen HTML browser, etc.) be fixed for each iteration of the `construct` loop, shown earlier. This is, however, exactly how the Builder pattern is used: the interpretation

engine does not change in the middle of the interpretation. Thus, the pattern is static—another reason to prefer parametric polymorphism to subtyping. This may result in improved performance because the costs of dynamic dispatch are eliminated.

The new organization also has other benefits. First, the control flow of the pattern is simpler: the client never calls the Builder object directly. Instead of the `get_result` call, the results are returned by the `construct` call made to the Director. Second, Directors can now be more sophisticated: they can, for instance, declare temporary variables of the same type as the type of the Builder’s product. These can be useful for caching previous products, without cooperation from the Builder classes. Additionally, Directors can now decide when the data should be consumed by the client. For instance, the Observer pattern could be used: clients of an HTML interpreter could register a callback object. The Director object (i.e., the interpreter) can then invoke the callback whenever data are to be consumed. Thus, the `construct` method may only be called once for an entire document, but the client could be getting data after each paragraph has been interpreted.

Another observation is that the Director class can be replaced by a functoid so that it can be manipulated using general tools. Note that the Director class in the Builder pattern only supports a single method call. Thus, it can easily be made into a functoid. Calling the functoid will be equivalent to calling `construct` in the original pattern. The return type of the functoid depends on the type of builder passed to it as an argument (instead of being `void`). An example functoid which integrates these ideas is shown here:

```
struct DoBuild {
    template <class B, class OC>
    struct Sig: public FunType<B,OC,
        Container<B::ResultType> >
    {};

    template<class B, class OC>
    Container<B::ResultType> operator()
    (B b, OC objs){
        Container<B::ResultType> c;
```

```

    for all objects in objs {
        if(object is_a A)
            c.add(b.build_part_A(object));
        else if (object is_a B)
            c.add(b.build_part_B(object));
        ...
    }
    return c;
}
} do_build;

```

With this approach, the “director” functoid is in full control of the data production and consumption. The Director can be specialized via currying to be applied to specific objects or to use a specific Builder. Two different Directors can even be composed—the first building process can assemble a builder object for the second!

5 Related Work

We have referred to related work throughout the previous sections. Here we will selectively discuss only some particularly related work that we did not get the chance to analyze earlier.

There are several libraries that add functional programming features to C++. Some of them [11][12][21] focus on front-end support (e.g., a `lambda` keyword) for creating functions on-the-fly. Other libraries [15][20] provide reusable functionality without any special front-end support. FC++ [16] is in this latter category: it provides mechanisms for expressing higher order and polymorphic functions, but does not hide the implementation behind a more convenient front end. FC++ is distinguished from the rest by its full type system for polymorphic functions, which enables creating and manipulating polymorphic functions on-the-fly, and by its support for indirect function references.

Dami’s currying mechanism for C/C++ [7] was used to demonstrate the advantages of function specialization, but required a language extension. As we saw, the same benefits can be obtained in C++ without extending the language.

Alexandrescu [2] offers a mature C++ implementation of the Abstract Factory pattern. His approach consists of a generic (i.e., polymorphic) Abstract Factory class that gets parameterized statically by all the possible products. It is worth noting that this is the exact scenario that Baumgartner et al. [4]

studied. Their conclusion was that meta-object protocols should be added to OO languages for better pattern support. Thus, Alexandrescu’s implementation is a great demonstration of the meta-programming capabilities of C++—the language’s ability to perform template computation on static properties can often be used instead of meta-object protocols.

Géraud and Duret-Lutz [9] offer some arguments for redesigning patterns to employ parametric polymorphism. Thus, they propose that parametric polymorphism be part of the “language” used to specify patterns. In contrast, our approach is to use parametric polymorphism with type inference in the *implementation* of patterns. From an implementation standpoint, the Géraud and Duret-Lutz suggestions are not novel: they have long been used in C++ design pattern implementations. Furthermore, the examples we offer in this paper are more advanced, employing type inference and manipulation of polymorphic functions.

The Pizza language [18] integrates functional-like support to Java. This support includes higher-order functions, parametric polymorphism, datatype definition through patterns, and more. Pizza operates as a language extension and requires a pre-compiler. Support for parametric polymorphism in Java has been a very active research topic (e.g., [1][5][17][23]). Type inference is not a part of most approaches, but is used at least in GJ [5]. Nevertheless, due to the GJ translation technique (erasure) it does not seem possible to extract static type information nested inside template parameters. Thus, it is not possible to use the GJ type system to pass polymorphic functions as arguments and return them as results (in a type-safe way).

It should be noted that Java inner classes [10] are excellent for implementing higher-order functions. Inner classes can access the state of their enclosing class, and, thus, can be used to express *closures*—automatic encapsulations of a function together with the data it acts on. Java inner classes can be anonymous, allowing them to express anonymous functions—a capability that is not straightforward to emulate in C++. Many of our observations of Section 3 also apply to Java. In fact, the most common Java implementations of the Command and

Observer design patterns use inner classes for the commands/callbacks.

6 Conclusions

In this paper we examined how functional techniques in general, and FC++ in particular, can be applied to OO tasks, by illustrating the implementations of some common design patterns. Our examples from Section 3 are similar to others in the literature, but, to our knowledge, our example pattern implementations from Section 4 have not appeared before, even in different contexts. Additionally, we are not aware of another mainstream, statically-typed OO language with the capabilities of FC++ for manipulating polymorphic functions and employing type inference.

Our implementations demonstrate the value of parametric polymorphism and type inference (even in a rather primitive form) in a statically-typed object-oriented language. By selectively using parametric polymorphism with type inference and higher-order functions, we can create simple, yet general, implementations of patterns that are both efficient and type safe.

References

- [1] O. Agesen, S. Freund, and J. Mitchell, "Adding Type Parameterization to the Java Language", *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1997*, 49-65.
- [2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.
- [3] S.R. Alpert, K. Brown, B. Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998.
- [4] G. Baumgartner, K. Läufer, V.F. Russo, "On the Interaction of Object-Oriented Design Patterns and Programming Languages", Tech. Report CSD-TR-96-020, Dept. of Comp. Sci., Purdue University, February 1996.
- [5] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, "Making the future safe for the past: Adding Genericity to the Java Programming Language", *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1998*.
- [6] C. Chambers, B. Harrison, and J. Vlissides, "A Debate on Language and Tool Support for Design Patterns", *ACM Symposium on Principles of Programming Languages*, 2000 (PoPL 00).
- [7] L. Dami, "More Functional Reusability in C/C++/Objective-C with Curried Functions", *Object Composition*, Centre Universitaire d'Informatique, University of Geneva, pp. 85-98, June 1991.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [9] T. Géraud and A. Duret-Lutz, "Generic Programming Redesign of Patterns", in *Proc. European Conf. on Pattern Languages of Programs, 2000 (EuroPLoP'2000)*.
- [10] Javasoft, *Java Inner Classes Specification*, 1997. In <http://java.sun.com/products/jdk/1.1/docs/>.
- [11] J. Järvi and G. Powell, "The Lambda Library: Lambda Abstraction in C++", TUCS Tech. Report No. 378, November 2000, available from <http://www.tucs.abo.fi/publications/techreports/TR378.html>.
- [12] O. Kiselyov, "Functional Style in C++: Closures, Late Binding, and Lambda Abstractions", *poster presentation, Int. Conf. on Functional Programming*, 1998. See also: <http://www.lh.com/~oleg/ftp/>.
- [13] S. Krishnamurthi, M. Felleisen, D. P. Friedman, "Synthesizing Object-Oriented and Functional Design to Promote Re-Use", *European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.
- [14] T. Kühne, *A Functional Pattern System for Object-Oriented Design*, Verlag Dr. Kovac, Hamburg, 1999.
- [15] K. Läufer, "A Framework for Higher-Order Functions in C++", *Proc. Conf. Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995.
- [16] B. McNamara and Y. Smaragdakis, "FC++: Functional Programming in C++", *Proc.*

International Conference on Functional Programming (ICFP), Montreal, Canada, September 2000.

- [17] A. Myers, J. Bank and B. Liskov, “Parameterized Types for Java”, *ACM Symposium on Principles of Programming Languages*, 1997 (PoPL 97).
- [18] M. Odersky and P. Wadler, “Pizza into Java: Translating theory into practice”, *ACM Symposium on Principles of Programming Languages*, 1997 (PoPL 97).
- [19] S. Peyton Jones and J. Hughes (eds.), *Report on the Programming Language Haskell 98*, available from www.haskell.org, February 1999.
- [20] A. Stepanov and M. Lee, “The Standard Template Library”, 1995. Incorporated in ANSI/ISO Committee C++ Standard.
- [21] J. Striegnitz, “FACT!—The Functional Side of C++”,
<http://www.fz-juelich.de/zam/FACT>
- [22] S. Thompson, “Higher-order + Polymorphic = Reusable”, *unpublished*, May 1997. Available at:
<http://www.cs.ukc.ac.uk/pubs/1997/224>
- [23] K. Thorup, “Genericity in Java with Virtual Types”, *European Conference on Object-Oriented Programming (ECOOP) 1997*, 444-471.