Bridging Functional and Object-Oriented Programming

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

Proponents of the functional programming paradigm contend that higher-order functions combined with (parametric) polymorphism result in much more reusable code. Object-oriented (OO) programming has independently followed a parallel approach to reusability. Objects can be used instead of higher-order functions and subtype polymorphism partially substitutes parametric polymorphism.

In this paper, we draw strong analogies between the object-oriented and functional programming paradigms. We show that several common OO design patterns (Visitor, Virtual Proxy, Command, Observer, and more) are closely related to functional programming patterns. Additionally, we show how better support for functional programming in OO languages can result in improvements for many design patterns (Command, Virtual Proxy, Builder, Abstract Factory, and more).

The context for demonstrating our ideas is the FC++ library. FC++ adds to C++ many of the capabilities of modern functional programming languages, without any modification to the base language. Compared to other attempts to program functionally in C++, FC++ offers much richer support for polymorphic functions (allowing functions that take polymorphic functions as arguments and/or return them as results).

1 Introduction

Most of the current research activity in programming languages concentrates either on object-oriented programming or on functional programming. Both programming paradigms have been promoted on the basis of their benefits for code reusability. Nevertheless, little work has been done to connect or relate the two paradigms (probably the most notable exceptions are the Pizza language [11], extending Java, as well as libraries for programming functionally in C++ [8][9][14]). This is surprising, since from a programmer's standpoint the two approaches to more flexible and reusable code are quite similar. The functional programming community claims that reusability benefits are a consequence of having functions that are both higher-order (i.e., can take other functions as parameters) and polymorphic [5].[1] Similarly, many common OO design patterns use object references to establish a point of indirection, instead of directly calling known functions. Additionally, subtype polymorphism is used to make code more general and supply safety guarantees about the interface that objects should support. Both of these techniques (function indirection through objects and subtype polymorphism) are in the core of the design patterns in the standard reference on the subject—the Gamma et al. "Design Patterns" book [4] (appropriately subtitled "Elements of Reusable Object-Oriented Software").

In this paper, we illustrate the connections between the two paradigms. We show that several common design patterns correspond to well-known functional programming techniques. We also show how better support for functional concepts in OO languages (in particular, *parametric polymorphism with type inference* and *first-class function objects*) can significantly simplify common design patterns,

1. Throughout the paper, we will use the term "polymorphic" to refer to *parametric* polymorphism (e.g., templates). In object-oriented programming, the term may also mean *subtype* polymorphism. In those cases where we talk about subtype polymorphism, we shall be explicit about it.

make them more general, or make them safer.[2] We shall limit our attention to the original design patterns introduced by Gamma et al. [4], although similar observations hold for more patterns.

The context of our presentation is our FC++ library [9] for programming functionally in C++. FC++ takes advantage of C++ type inference to build a type system that supports higher-order and polymorphic functions without resorting to language extensions. Compared to all other libraries for programming functionally in C++ (including the C++ Standard Library [13]) FC++ is primarily distinguished by the ability to manipulate polymorphic functions directly, without first turning them into monomorphic instances. This ability has enabled us to quickly reproduce in C++ a large functional code base (a substantial part of the Haskell Standard Prelude—the standard library of the Haskell programming language [12]).

Nevertheless, this is not a paper specific to FC++. Our goal is to illuminate the commonalities between the functional and OO paradigms and argue for using both together; FC++ serves as a convenient mechanism to articulate our ideas.

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, with the goal of presenting just enough about FC++ and functional programming to understand the examples we shall present in Section 3. Section 2.2 describes a few of the important implementation details of the FC++ library. For a more complete introduction to FC++, the reader should refer to [9].

Though the reader can skip Section 2.2 and still understand most of the examples in the rest of the paper, this material is useful so that interested readers can see how the library works and can follow the examples of Section 3.3. Specifically, Section 2.2 illustrates the key mechanism for representing polymorphic functions in the FC++ library. This mechanism is what enables FC++ to express the examples in the paper. It also sets FC++ apart from all other functional programming libraries in C++.

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, currying arguments, 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. `FunN`s 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 3.2, this simplest functoid plays a significant role in a number of patterns.)

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

2. We do not discuss improvements in design patterns by the mere addition of parametric typing (e.g., C++ class templates) in a language. These are well understood and are even discussed in [4], as implementation suggestions.

```
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 = makeFun2( ptr_to_fun(&i_times) );
f(3,4); // returns 12
f = makeFun2( 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.

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., iplus(1,4)
```

(The 2 in `curry2` refers to the number of arguments that `f` expects.) *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 FC++ implementation

In the last 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 `Sig`s 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 functoids that can directly manipulate polymorphic functoids. Specifically, other functoids can determine what the result type of a particular polymorphic functoid would be, for given arguments.

To see how, consider the simple functoid `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 [13]). 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 functoid 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 functoids `f` and `g` of types `F` and `G`, respectively. `compose(f,g)` returns a (possibly polymorphic) direct functoid with the following `Sig` member:

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

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

Although these 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 3 are realizable only because of this unique feature of our library.

3 Reusability with Object-Oriented and Functional Patterns

3.1 The Common Path to Reusability

From an OO standpoint, one can view functional programming as a very general design pattern. Indeed, 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). The use of higher-order functions promotes reusability because these functions express generic pieces of code, configured dynamically to perform appropriate tasks. The direct manipulation of functions (passing them as arguments and returning them as results) also yields reusability benefits. A function is no longer an inflexible piece of code that can only be called. Instead, functions can be "massaged" to adapt to different settings. Polymorphism is another common element contributing to reusability in functional languages. Polymorphic functions can be applied to arguments of multiple types, and, hence, are reusable across types.

It is important to note that object-oriented programming has followed a similar path to reusability. Polymorphism is common in OO languages in the form 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. In contrast to parametric polymorphism, which is a static concept, subtype polymorphism is dynamic: the resolution of what method gets called occurs at run-time. 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).

These similarities may seem obvious or superficial. Nevertheless, the interesting question is whether programmers with object-oriented training can benefit from functional tools or methodologies and vice versa. We believe that this is the case. In the next sections we outline the similarities between programming patterns in the two paradigms, we discuss the comparative advantages of both approaches, and we show how integration of functional features can improve OO idioms.

3.2 Examples: Design Patterns from a Functional Viewpoint

In this section we show how some common design patterns are related to functional programming idioms. Where appropriate, we illustrate the examples using FC++. The design patterns we consider are *Command*, *Observer*, *Adaptor*, *Visitor*, and *Virtual Proxy*. Command and Observer are similar to higher-order functions. A number of functional techniques like currying and composition are analogous to Adaptors. Visitor enables OO designs to be extended horizontally rather than vertically—a structure natural to functional programs. Virtual Proxy is akin to "lazy evaluation"—a common functional technique.

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 perspec-

tive, `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[] = {
 makeFun0(curry(
    ptr_to_fun(&Document::cut), &d)),
 makeFun0(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 have disappeared! `Fun0<void>` defines a natural interface for commands, and the concrete instances can be created on-the-fly by making indirect functoids out of the appropriate functionality, currying arguments when necessary.

The above example takes advantage of the fact that `ptr_to_fun` can be used to create functoids 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( makeFun0(curry(ptr_to_fun(
     &ConcreteObserver::get_notification
    ), this ) ) );
  }

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

Note again how `ptr_to_fun` is used to create a direct functoid out of an instance method. The resulting functoid 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 `get_notification` 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 functoid 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 functoids 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;[3] extra classes and methods (with names) are unnecessary to realize a solution.

Adaptor. The Adaptor 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()`. Adaptor 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 a special adaptor class. 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 the existing methods to have the right type signature. For instance, in our `ConcreteObserver` example, above, the `get_notification` 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, the publisher expects a functoid object taking no arguments. This is easy to effect by adapting the observer's interface:

```
makeFun0(
  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 subscription time instead of the update time. A better approach is:

3. A tuple of indirect functoids can be used if multiple function signatures are defined in an interface; the example in [4] of Command used for do/undo could be realized in FC++ with a `std::pair<Fun0<void>,Fun0<void>>`, for instance.

```
makeFun0(
 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!

We should note that although our examples only show binding the first argument(s) of a functoid, binding other arguments is just as easy. FC++ has predefined operators for most common cases.

Visitor. Object-oriented languages organize code into classes; distinct operations are grouped together with the data they act on. For instance, if an operation `display` is applicable to instances of multiple classes, each class defines its own `display` method. Functional languages typically do the opposite: Each operation is an independent entity which contains multiple cases—one for each type (i.e., class) on which the operation can be applied. The *Visitor* design pattern is the object-oriented idiom that enables exactly this operation-centric design: Multiple definitions of the same operation (applicable to objects of several different types) can all be grouped together in a visitor class, instead of being distributed in the individual classes.

This organization is often considered more in line with the functional paradigm. Thus, Visitor can be seen as a way to program functionally in OO languages. This observation is old and has been made several times independently. We will not elaborate on the topic here—the interested reader should consult [7] and its references for more information.

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 [4], 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. Laziness works especially well in languages without side-effects; languages with side-effects are strict because many function calls are computations done for their effects, not their return values. Combining laziness with side-effects necessitates an explicit mechanism to state whether or not a computation should be lazy. After all, the language cannot read the programmer's mind to know if she is using a call for its effects, its return value, or both.

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 computation 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.

3.3 Examples: 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 polymorphism with type inference. Type inference was discussed in Section 2.2, but, briefly, it is the process of deducing the return type of a function, given specific arguments. This is relevant for strongly typed OO languages like Java or C++. In this section, we will examine the Builder and Abstract Factory design patterns and see how they can be improved if they employ polymorphism.

Abstract Factory. The *Abstract Factory* pattern can be used to establish a single creation point for many related objects. The standard example is that of a user interface toolkit that supports multiple standards for look-and-feel. In this way, changing the factory object used to create all the graphical objects in the system (windows, toolbars, scrollbars, menus, etc.) will change the behavior and appearance of all these objects. This is particularly useful because client applications should not commit early to a particular kind of graphical objects by having scattered concrete object instantiations throughout their code. Instead, applications should handle all graphical object creation through an abstract factory interface and concrete factories (specializations of the abstract factory) should centralize the creation of particular graphical objects.

It is important to note that the Abstract Factory pattern is quite static by nature. Gamma et al. [4] observe: “Normally a single instance of a ConcreteFactory class is created at run-time.” Therefore, it is better to use parametric polymorphism instead of subtype polymorphism in order to express this pattern. This will improve the type safety of the code in strongly typed languages because clients will hold references to instances of concrete classes, as opposed to holding references to abstract classes. In C++, this eliminates the possibility of run-time type errors, and may also improve performance by avoiding the overhead of dynamic dispatch.

For instance, look-and-feel could be determined by choosing among different classes with member types for each of the desired graphical objects:

```
typedef LookAndFeelA CurrentLookAndFeel;
...
CurrentLookAndFeel::Window win;
...
CurrentLookAndFeel::Toolbar tb =
  new CurrentLookAndFeel::Toolbar(win);
```

Another important problem with the original Abstract Factory pattern (as well as with many design patterns based on subtype polymorphism) is that it is hard to make it span multiple applications. Consider, for instance, a generic library for the manipulation of windowing objects. This library may contain adaptors, 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 this 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 objects be parametrically polymorphic and enabling type inference. This is done by making each concrete type encapsulate information about its look-and-feel. For instance, if each graphical type has a nested type `LookAndFeel`, 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,
    W::LookAndFeel::ScrollWindow *>
  {};

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

Since the above 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.

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 markup or text it encounters in the HTML data (e.g., font change, paragraph end, text strings, etc.).

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

```
void construct() {
  for all objects {
    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. [4] 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 means that the kind of Builder object used (e.g., an HTML to PDF converter, an on-screen HTML browser,

etc.) will be fixed for each iteration of the `construct` loop, shown earlier. This is, however, exactly how the Builder pattern is used. Since the interpretation engine does not change in the middle of the interpretation, the pattern is static—another reason to prefer parametric polymorphism to subtyping. This may result in improved performance because the cost of dynamic dispatch is eliminated.

The new organization has some 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 every 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`). A functoid integrating the above ideas is shown below:

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

  template<class B>
  Container<B::ResultType> *operator()
  (B b){
    typedef Container<B::ResultType>
      Product;
    Product *c = new Product();
    for all objects {
      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.

3.4 Comparison Summary

The overwhelming characteristic of the patterns we discussed is their similarity to functional solutions. Nevertheless, we also saw some differences between the functional and object-oriented approaches. Here we summarize the main comparison points examined in earlier sections, and discuss them further.

Common supertype. A common trait in several OO design patterns is that they use inheritance to establish a common interface for several classes. Clients only use the interface and dynamically dispatch to the appropriate specialized method. In a functional approach, we can often eliminate such supertypes/interfaces and instead identify the concrete objects as just "functions" with the right type.

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 having 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.

Treating functions uniformly. In the functional approach, many different entities can be described as just "functions". We have already discussed how the FC++ operator `ptr_to_fun` can be used to create functoids out of global functions, instance methods, class (static) methods, Standard Library "functional" objects, etc. The advantage of this approach is that all these entities can be manipulated using the same general tools, like currying and composition. Currying and composition are very powerful mechanisms for adapting on-the-fly an existing interface to new requirements.

Subtype vs. parametric polymorphism.
OO design patterns often make use of subtype polymorphism. Nevertheless, subtype polymorphism is not always ideal. Subtype polymorphism requires that all possible parameters have a common supertype. Additionally, the determination of the actual type of a parameter only occurs at runtime. In contrast, parametric polymorphism is a compile-time mechanism. Combined with type inference, parametric polymorphism can be as convenient to use as subtype polymorphism. Its advantages include better type safety and more efficient method dispatch.

Clearly, parametric polymorphism is applicable only in cases where the resolution of an argument's type is possible at compile time. Some abstract designs offer strong indications that parametric polymorphism may be preferable. First, it may not be feasible to implement a common supertype for all possible parameter types. Second, a parameter setting (e.g., what abstract factory is to be used) may be fixed throughout the application's execution.

Finally, we do not claim that the patterns in Section 3.3 are the ideal justification for parametric polymorphism with type inference. Our goal was to show how the technique can be used in practice to improve design patterns. Other examples (e.g., see [9]) may be better for demonstrating the value of type inference. Additionally, the reader should keep in mind that many of the FC++ operators we have encountered (e.g., `compose`) owe their generality to parametric polymorphism with type inference.

4 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 [6][14] focus on front-end support (e.g., a `lambda` keyword) for creating functions on-the-fly. Other libraries [8][13] provide reusable functionality without any special front-end support. FC++ [9] 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 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++ [3] 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.

The Pizza language [11] 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][2][10][16]). Type inference is not a part of most approaches, but is used at least in GJ [2]. 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 [5] 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++. Some of our observations of Section 3.2 also apply to Java (the biggest exception is the currying and function composition capabilities). In fact, the most common Java implementations of the Command and Observer design patterns use inner classes for the commands/callbacks.

5 Implications and Conclusions

We believe that the OO and functional communities have more in common with each other than they may realize. Indeed, we think that there is evidence that the two paradigms may be converging. Our own background is firmly in the OO world, and in this paper we attempted to express functional techniques in object-oriented terms, but also to view object-oriented techniques with a functional eye. Our goal is to promote a better understanding of the functional paradigm among OO programmers. We claim that using a combination of both paradigms is better than using either alone.

In this paper we supported our claim by demonstrating a mapping between a number of OO design patterns and functional programming techniques. In a number of cases, we applied functional techniques to yield concrete benefits:

- In Section 3.2, we eliminated a number of needless classes from pattern implementations, and created a more natural interface among pattern participants.
- In Section 3.3, we demonstrated how patterns can be made more general and can be implemented more safely thanks to parametric polymorphism with type inference.
- In both of those sections, we demonstrated that general-purpose higher-order functions (like `curry` and `compose`) can reduce the need for special-purpose classes and simplify code—generic combinators allow us to easily create new functions on-the-fly.

There is potentially much to be gained from incorporating ideas of the functional programming paradigm into the object-oriented arsenal. Functional programming can yield a different perspective on various design problems. This new perspective can, in turn, inspire new solutions.

We hope that readers will consider adding functional techniques to their "mental tool-belt". We offer FC++ as a way to help express these ideas, and also as one concrete implementation which can realize such designs.

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] 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*.

[3] 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.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] Javasoft, *Java Inner Classes Specification*, 1997. In `http://java.sun.com/products/jdk/1.1/docs/`.

[6] 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/` .

[7] 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.

[8] K. Läufer, "A Framework for Higher-Order Functions in C++", *Proc. Conf. Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995.

[9] B. McNamara and Y. Smaragdakis, "FC++: Functional Programming in C++", *Proc. International Conference on Functional*

Programming (ICFP), Montreal, Canada, September 2000.

[10] A. Myers, J. Bank and B. Liskov, “Parameterized Types for Java”, ACM *Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

[11] M. Odersky and P. Wadler, “Pizza into Java: Translating theory into practice”, *ACM Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

[12] S. Peyton Jones and J. Hughes (eds.), *Report on the Programming Language Haskell 98*, available from `www.haskell.org`, February 1999.

[13] A. Stepanov and M. Lee, “The Standard Template Library”, 1995. Incorporated in ANSI/ISO Committee C++ Standard.

[14] J. Striegnitz, “FACT!—The Functional Side of C++”, `http://www.fz-juelich.de/zam/FACT`

[15] S. Thompson, “Higher-order + Polymorphic = Reusable”, *unpublished*, May 1997. Available at: `http://www.cs.ukc.ac.uk/pubs/1997/224`

[16] K. Thorup, “Genericity in Java with Virtual Types”, *European Conference on Object-Oriented Programming (ECOOP) 1997*, 444-471.