ACM SIGPLAN

Functional Programming

Editor: *Philip Wadler, Bell Laboratories, Lucent Technologies; wadler@research.bell-labs.com*

Functional Programming in C++ using the FC++ Library

Brian McNamara and Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
`http://www.cc.gatech.edu/~yannis/fc++/`

> "...the determined Real Programmer can write FORTRAN programs in any language."
>
> *Real Programmers Don't Use PASCAL, Ed Post (1982)*

Some twenty years and a few major languages later, the ability to write "FORTRAN code" in any language is still easy to assert. One would hope, however, that the property of transcending languages is not limited to the unrefined "Real Programmer" and to "FORTRAN code". Can the determined Functional Programmer write Haskell programs in any language?

Okay, probably not.

We would like to suggest, however, that a determined programmer *can* write functional programs in C++. In this short article, we will introduce you to FC++, a library that supports functional programming in C++. If you are using functional languages, reading this article will probably not motivate you to abandon them in favor of C++. Nevertheless, FC++ is thought-provoking both for functional programmers and for object-oriented programmers. It shows how polymorphic, higher-order functions can be expressed in C++ and provides a good platform for combining the functional and object-oriented paradigms. We invite you to take a tour of FC++ and catch a glimpse of the possibilities.

1 What is FC++?

FC++ [5] is a library for doing functional programming in C++. The library comprises a general framework for creating FC++ functions (which we sometimes call *functoids*) as well as about 100 common/useful functions.

Using classes to represent functions and objects to represent closures is a standard technique in object-oriented languages. Among others, the Pizza language [6] uses this approach in translating functionally-flavored constructs to Java code. Along the same lines, the C++ Standard Library contains some basic functionality for expressing and manipulating functions. C++ even allows classes representing functions to be used with the usual function call notation, by overloading the function application operator, `operator()`. Nevertheless, the C++ Standard Library stops short of providing a general framework for functional programming. Other libraries have attempted to fill the gap by supplying either syntax support (e.g., a "lambda" operator for anonymous functions) [3][8] or a framework for expressing higher-order function types [4].

FC++ is distinguished from all such libraries by its powerful type system. FC++ offers complete support for manipulating polymorphic functions—passing them as arguments to other functions and returning them as results. For instance, FC++ supports higher-order polymorphic operators like `compose()`: a function that takes two (possibly polymorphic) functions as arguments and returns a (possibly polymorphic) result (the composition). Thus, FC++ can be used to embed a lot of the capabilities of modern functional programming languages (like Haskell or ML) in C++. Indeed, FC++ comes with a wealth of useful pre-defined functions—a large part of the Haskell Standard Prelude—as well as support for lazy evaluation, including a "lazy list" data structure and a number of functions that operate on these lazy lists. The library also contains a number of support functions for transforming FC++ data structures into the data structures of the C++ Standard Template Library (STL), and vice versa, as well as operators for promoting normal functions into FC++ functoids. Finally, the library supplies "indirect functoids": run-time variables that can refer to any functoid with a given monomorphic type signature.

The library is implemented in ISO Standard C++. Its implementation relies heavily on C++ templates and the C++ type system. Unlike other libraries for functional

```
#include <assert.h>
#include <string>
#include "prelude.h"

int main() {
  int x=1, y=2, z=3;
  std::string s="foo", t="bar", u="qux";

  List<int> li =
    cons(x,cons(y,cons(z,NIL)));
  List<std::string> ls =
    cons(s,cons(t,cons(u,NIL)));

  assert( head(li) == 1 );
  // list_with() makes short lists
  assert( tail(li) == list_with(2,3) );

  ls = compose(tail,tail)(ls);
  assert( head( ls ) == "qux" );
  assert( tail( ls ) == NIL );
}
```

Figure 1: Lists and compose

programming in C++, FC++ does not focus on improving the syntax using either the preprocessor (e.g., `#define`) or overloading techniques (like expression templates). Such approaches have value but are brittle. Instead, the value of FC++ is in its type system for polymorphic functions—providing a nicer syntactic front-end for defining functions is an orthogonal (and perhaps secondary) issue. The FC++ library currently comprises about 3000 lines of C++ code. We are continuing to develop the library to make it both more expressive and more convenient to use.

2 What can I do with FC++?

Now let's introduce you to the library, by walking through some examples that demonstrate the capabilities of FC++.

Many of the examples will use lists, so we begin with code that shows a little about the `List` class (Figure 1). A `List` is parameterized by the type of its elements; in the listing, we show both a list of `int`s and a list of `string`s. The usual operators `cons()`, `head()`, `tail()`, and the constant `NIL` work as you would expect.

This example also demonstrates the capabilities of FC++ for manipulating polymorphic functions. The `tail()` function takes a "list of T" and returns a "list of T" where T can be any type; in Haskell, for example, we would write its type as

```
tail :: [a] -> [a].
```

The `compose()` operator composes two unary functions, that is, `compose(f, g)` yields a new function `h` such that `h(x)` is the same as `f(g(x))`. The `compose()` operator can take polymorphic functions as parameters and return a polymorphic function as a result. In Haskell, we would describe the type of `compose` as

```
compose :: (a->b) -> (c->a) -> (c->b).
```

As a result, `compose(tail,tail)` is a polymorphic function with the same signature as `tail`.

FC++ lists are lazy. For example, we can say

```
List<int> integers = enumFrom(1);
```

to create an infinite list of all the integers 1, 2, 3, ... Elements of the list are only evaluated as they are requested. We can perform various operations lazily on such lists, such as the `filter()` function defined in the library that returns only those elements of a list which meet a certain predicate. For example,

```
List<int> evens = filter(even, integers);
```

creates a list of the even integers (2, 4, 6, ...); `even` is another function defined in the FC++ library. We can easily define our own predicates by writing normal C++ functions, for example

```
bool prime( int x ) { ... }
```

and then use, for example,

```
filter( ptr_to_fun(&prime), integers );
```

to compute a list of primes. The FC++ function `ptr_to_fun()` transforms a normal C++ function into a functoid. It is one of a number of library members which provide the interface between FC++ functoids and both C functions and C++ standard library function objects. Figure 2 shows a complete program, which also demonstrates `take()`—a function that selects the first N elements of a list and discards the rest.

FC++ functoids support currying. If we start with the list of numbers 1-3:

```
List<int> integers = list_with(1,2,3);
```

we can generate the list 2-4 with `map(inc,integers)` where `inc()` is a function that increments a number by 1, and `map()` applies a function to each element to a list. Suppose instead we want to add 2 to each element of the list. Of course, we could say

```
#include <assert.h>
#include "prelude.h"

bool prime( int x ) {
  if( x<2 ) return false;
  for( int i=2; i <= x/2; i++ )
    if( x%i == 0 ) return false;
  return true;
}

int main() {
  List<int> integers = enumFrom(1);
  assert(take(3,integers) ==
    list_with(1,2,3));

  List<int> evens = filter(even,integers);
  assert(take(3,evens)==list_with(2,4,6));

  List<int> primes =
    filter(ptr_to_fun(&prime), integers );
  assert(take(3,primes) ==
    list_with(2,3,5));
}
```

Figure 2: Lazy operations and C++ functions

`map( compose(inc,inc), integers )`

but we can also just say

`map( plus(2), integers ).`

The FC++ library defines function `plus()` such that `plus(x,y)` yields `x+y`. (Indeed, the library contains named functions for all of the common operators.) Like all functoids in the FC++ library, `plus` is curryable. That is to say, `plus(2)` yields a new function `f(x)`, where `f(x) = 2 + x`.

As you might expect, currying of polymorphic functions is fully supported and may yield other polymorphic functions. In fact, currying is implemented by FC++ operators that are themselves (higher-order polymorphic) functoids. We can use these operators explicitly, if needed. For instance, `bind1of2()` is a function that takes a binary function and binds its first argument to a particular value, resulting in a unary function. Thus, `bind1of2( plus, 2 )` is the same as `plus(2)`. We can also write `plus(2,_)` to mean the same thing; "`_`" is a special value in FC++ that serves as a placeholder for arguments to be curried. Figure 3 shows a number of examples which demonstrate currying in FC++.

The FC++ library supplies users with many useful predefined functions. More than 50 functions from the

```
#include <assert.h>
#include "prelude.h"

List<int> answer;
// holds the answer of upcoming
// computations for exposition purposes

void check( List<int> l )
{ assert( l==answer ); }

int main() {
  List<int> integers = list_with(1,2,3);

// each small group of statements
// demonstrates similar functionality with
// different syntax
  answer = list_with(2,3,4);
  check( map( inc, integers )     );
  check( map( plus(1), integers ) );

  answer = list_with(3,4,5);
  check( map(compose(inc,inc),integers) );
  check( map( plus(2), integers )       );

  answer = list_with(0,-1,-2);
  check( map(bind1of2(minus,1),integers));
  check( map( minus(1), integers )      );
  check( map( minus(1,_), integers )    );

  answer = list_with(0,1,2);
  check( map(bind2of2(minus,1),integers));
  check( map( minus(_,1), integers ) );

  // map can also be curried
  answer = list_with(3,4,5);
  check( map( plus(2) )( integers )    );
  check( map( _, integers )( plus(2) ) );
}
```

Figure 3: Currying examples

```
#include <assert.h>
#include "prelude.h"

int f( int x, int y ) { return 3*x + y; }

class Foo {
  int n;
public:
  Foo( int nn ) : n(nn) {}
  int bar( int x, int y ) const
  { return n*x + y; }
};

int main() {
  assert( ptr_to_fun(&f)(3)(1) == 10 );

  Foo foo(3);
  assert(ptr_to_fun(&Foo::bar)(&foo,3)(1)
         == 10);
}
```

Figure 4: FC++ and native C++ functions

Haskell Standard Prelude are included in the library. We have already seen a few such functions, like `map()`, `take()`, `filter()`, and `enumFrom()`. FC++ also supports `until()`, `foldr()`, `iterate()`, `cycle()`, `span()`, `zipWith()`, and many others. These predefined functions make it easy for users of the FC++ library to rapidly compose algorithms to suit their needs using functional programming techniques.

FC++ has interfaces to normal C++ functions and the C++ standard library. We have already encountered `ptr_to_fun()`, which converts a normal function into an FC++ functoid. The `ptr_to_fun()` operator works on member functions as well, creating a functoid which takes a pointer to the receiver object as an extra first parameter. Figure 4 shows `ptr_to_fun()` applied to both normal and member functions, and demonstrates that the results are functoids by using the currying ability of FC++ functoids.

To interface to the C++ standard library data structures, FC++ supports iterators. Figure 5 shows that the List class supports iterators of the STL style. This makes converting to and from STL data structures easy.

The functoids we have seen thus far are called *direct functoids* in the FC++ library, because calls to them are statically bound (they are called directly). FC++ also supports *indirect functoids* via the `FunN` classes. These functoids are dynamically bound, and thus can change their "function values" by assignment. Indirect functoids are

```
#include <assert.h>
#include <vector>
#include <algorithm>
#include "prelude.h"

int main() {
  List<int> l = take( 5, enumFrom(1) );
  std::vector<int> v(5);
  std::copy(l.begin(),l.end(),v.begin());
  std::reverse( v.begin(), v.end() );
  List<int> r( v.begin(), v.end() );
  assert( r == list_with(5,4,3,2,1) );
}
```

Figure 5: FC++ and STL

described by their monomorphic type signature, and variables of type `FunN` can be bound to any function with the right signature. For example, a `Fun1<int,bool>` describes a one-argument function that takes an `int` and returns a `bool`, whereas a `Fun2<int,int,string>` describes a two-argument function which takes two `int`s and returns a `string`. The function `makeFunN()` converts a direct functoid into an indirect one. In the case of polymorphic functions, a monomorphic instance must be selected with `monomorphizeN()`. In fact, both conversions can be performed implicitly when an indirect functoid variable is assigned a direct functoid value. Figure 6 gives some examples of indirect functoids.

3 Where is the magic?

In the previous section we saw how functoids can be used. We also saw how to convert regular C++ functions or methods into functoids, so that they can be used with the FC++ pre-defined functionality, including higher-order operators like currying and `compose`. Nevertheless, we have not shown you how the polymorphic functoids inside FC++ (`compose`, `map`, etc.) are implemented or how to define your own polymorphic functoids. To simplify the discussion, we will concentrate on the key insights instead of specifics. The examples of this section will be both more simplified and more complicated than actual code: we will eliminate some C++ syntax verbosity but will also avoid several FC++ shorthands in order to expose the implementation. More detailed instructions can be found in the FC++ tutorial and documentation [1].

To create your own polymorphic functoid, you need to create a class with two main elements: a template `operator()` and a member structure template named

```
#include <assert.h>
#include "prelude.h"

bool prime( int x ) {
  if( x<2 ) return false;
  for( int i=2; i <= x/2; i++ )
    if( x%i == 0 ) return false;
  return true;
}

bool big( int x ) { return x > 100; }

int main() {
  Fun1<int,bool> f =
    makeFun1( ptr_to_fun(&prime) );
  assert( f(11) == true );
  f = makeFun1( ptr_to_fun(&big) );
  assert( f(11) == false );

  List<int> l = list_with(1,2,3);

  // explicit conversion of "tail" to
  // an indirect functoid
  Fun1<List<int>,List<int> > g =
    makeFun1( monomorphize1<List<int>,
                List<int> >(tail));
  assert( g(l) == list_with(2,3) );

  // implicit conversion
  g = init;
  assert( g(l) == list_with(1,2) );
}
```

Figure 6: Indirect functoids examples

`Sig`. To make things concrete, consider the definition of `map` (or rather, the class `Map`, of which `map` is a unique instance):

```
struct Map {
 template <class F, class L>
 struct Sig {
  typedef
  List<F::Sig<L::ElementType>::ResultType>
    ResultType;
 };

 template <class F, class T>
 Sig<F, List<T> >::ResultType operator()
      (const F& f, const List<T>& l) const
 {
   if( null(l) )
     return NIL;
   else
     return cons(f(head(l)),
                 lazy(Map(), f, tail(l)));
 }
};
```

The `operator()` will allow instances of this class to be used with regular function call syntax. What is special in this case is that the operator is a template, which means that it can be used with arguments of multiple types. When an instance of `Map` is used with arguments `f` and `l`, unification will be attempted between the types of `f` and `l`, and the declared types of the parameters (`const F&`, and `const List<T>&`). The unification will yield the values of the type parameters `F` and `T` of the template. This will determine the return type of the functoid.

Now, let's examine the `Sig` member class of the `Map` class. By FC++ convention, the `Sig` member should be a template over the argument types of the function you want to express (in this case the function type `F` and the list type `L`). The `Sig` member template is used to answer the question “what type will your function return if I pass it these argument types?” The answer in the `Map` code is:

```
List<F::Sig<L::ElementType>::ResultType>
```

This means: “map returns a `List` of what `F` would return if passed an element like the ones in list `L`”.

In Haskell, one would express the type signature of `map` as:

```
map :: (a -> b) -> [a] -> [b]
```

The `Sig` members of FC++ functoids essentially encode the same information, but in a computational form:

`Sig`s are type-computing compile-time functions that are called by the C++ unification mechanism for function templates and implement the FC++ type system. This type system is completely independent from the native C++ type system—`map`'s type as far as C++ is concerned is just `class Map`.[1] Other FC++ functoids, however, can read the FC++ type information from the `Sig` member of `Map` and use it in their own type computations. The `map` functoid itself uses that information from whatever functoid happens to be passed as its first argument (see the `F::Sig<L::ElementType>::ResultType` expression, above).

4 Limitations

As you may expect, functional programming in C++ is not without its limitations. The most obvious one for functional programmers is the lack of a construct like "lambda". C++ has no built-in support for defining new functions inside an expression or creating automatic closures, capturing all variables from the local environment. There are a number of "lambda libraries" for C++ [3][8] which use expression-templates to simulate lambda syntax (sometimes rather convincingly). Even these libraries, however, cannot create automatic closures; to truly add "lambda" to C++ would require a language extension.

A more thorough discussion of the capabilities and limitations of FC++ can be found in [5].

5 Applications and Conclusions

The FC++ library supports functional programming in C++, by enabling users to write and manipulate polymorphic and higher-order functions. The library has a smooth interface to the rest of C++, so that functional code and OO code can blend well. In this paper we gave an overview of FC++. More information can be found in the references [1][5][7].

FC++ is useful for functional programmers because it provides an alternative, commonly available platform for implementing familiar designs. An example of this approach is the XR (*Exact Real*) library [9]. XR uses the FC++ infrastructure to provide exact (or *constructive*) real-number arithmetic, using lazy evaluation.

[1] (Actually, this is a small lie—`map` is not an instance of `Map`, but rather an instance of `Curryable2<Map>`. Curryability is expressed via the `CurryableN` combinators in FC++.)

FC++ is also an interesting platform for object-oriented programming, because it allows functional techniques to be used in conjunction with common OO styles. In another paper [7], we show how a number of OO design patterns [2] can be simplified, generalized, or made safer using functional programming techniques.

References

[1] The FC++ web page: `http://www.cc.gatech.edu/~yannis/fc++/`

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

[3] *The lambda library*. `http://lambda.cs.utu.fi/`

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

[5] B. McNamara and Y. Smaragdakis, "FC++: Functional Programming in C++", *Proc. International Conference on Functional Programming (ICFP)*, Montreal, Canada, September 2000.

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

[7] Y. Smaragdakis and B. McNamara, "Bridging Functional and Object-Oriented Programming" Georgia Tech CoC Tech. Report 00-37, also available in [1].

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

[9] *The XR Exact Real Home Page*. `http://www.btexact.com/people/briggsk2/XR.html`

Brian McNamara is a PhD student at the Georgia Institute of Technology. His research interests include programming language design and implementation, as well as program analysis and formal verification methods.

Yannis Smaragdakis is an Assistant Professor at the Georgia Institute of Technology. His research interests include programming language design and implementation but also memory hierarchies, especially from an operating systems standpoint. His work has received high praise—Charles Simonyi once called him a "master hacker" for successfully fiddling with the climate control system in his mansion. ("His" is intentionally left ambiguous.)