Boost FC++ Documentation

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

September 21, 2003

1 Introduction, important definitions, and notes (Read this!)

FC++ is a library for doing functional programming in C++. The library provides a general framework to support various functional programming aspects, such as higher-order[1] polymorphic[2] functions, currying, lazy evaluation, and lambda. In addition to the framework, FC++ also provides a large library of useful functions and data types.

In FC++, we program with *functoids*, which are C++ classes or structs which overload `operator()` and obey certain other conventions. We will describe functoids in more detail in Sections 5, 6, and 7. Functoids are simply our chosen representation for functions in C++; for now you may equate the terms "functoid" and "function" in your mind. Later we shall see what exactly a functoid is and why it is useful.

This document describes "Boost FC++". Prior to being "boostified", FC++ existed as a research project at Georgia Tech for a number of years [2]. This is the first document to describe the boostified version of FC++. It is important to note the changes made to the library between "FC++" and "Boost FC++", in order to understand the old documentation. The changes are mostly just a matter of naming conventions. In Figure 1, we summarize the naming changes from FC++ to Boost FC++, by giving a sense of the general name changes as well as some particular examples. To avoid confusion, the reader of this document should at least be aware of this mapping before reading any of the prior documentation on FC++.

[1] A higher-order function is a function which takes another function as an argument or returns a function as a result.

[2] We use the term "polymorphism" to mean parametric polymorphism (e.g. templates). This is the usual definition in the sphere of functional programming. In object-oriented programming, the term "polymorphism" usually refers to dynamic dispatch (virtual function call). Please note the definition we are using. In contrast, a monomorphic function is a function which only works on one set of argument types.

```
Original FC++                 Boost FC++
-------------------------     --------------------------------
SomeFunctoid someFunctoid; -> some_functoid_type some_functoid;
   EnumFrom enumFrom;      -> enum_from_type enum_from;
SomeType                   -> some_type
   Full2                   -> full2
SomeFunc_                  -> some_func_x_type
   BindM_                  -> bind_m_x_type
NestedType                 -> nested_type
   Arg1Type                -> arg1_type
Misc:
   RT,LAM,COMP,...             -> unchanged
   ElementType,LEType,Inv,Var  -> value_type,LE,INV,VAR
   Ref,IRef   -> boost::shared_ptr, boost::intrusive_ptr
   curryN -> thunkN
```

Figure 1: Naming changes from FC++ to Boost FC++

2 Pure Functional Programming

Many of the functions and data types in the FC++ library are designed to mimic those in the Haskell programming language. Haskell is a *pure* functional programming language: there are no effects, global variables, or destructive updates in the language. Programming in this pure style ensures referential transparency.

The FC++ library uses this "pure" style for the most part. All parameters to FC++ functoids are passed by `const&` by default, for example. Nevertheless, since this is C++, you can use FC++ to create side-effecting functoids in a number of ways, which we will see later (Section 8). The introductory examples will all be effect-free, however.

3 Roadmap

Here we give an overview of what each section of the documentation covers.

Section 4 provides an overview of most of the major features of the FC++ library via a number of small examples.

The next few sections describe functoids in detail:

- Section 5 describes *direct functoids*. Direct functoids are the way we represent most functions in FC++.
- Section 6 describes *indirect functoids*. Indirect functoids are data types which can be used to create function variables which can be bound to different function values at run-time.
- Section 7 describes *full functoids*. Full functoids are functoids which support all of the features of FC++, such as currying and infix syntax.

Section 8 describes how to include side effects in functoids, typically by using global variables, pointers, or "thunks".

Section 9 describes the FC++ "lazy list" data structure, as well as more general support FC++ has for lazy evaluation.

Section 10 gives a short description of all of the functoids and data types that are included in FC++ the library.

Section 11 describes the relationships and interfaces between FC++ and other libraries (STL and Boost).

Section 12 describes the FC++ lambda construct, for creating anonymous functions on-the-fly, as well as the rest of our lambda sublanguage.

Section 13 introduces monads and describes FC++'s support for them.

Section 14 provides larger example contexts which illustrate the utility of FC++.

Section 15 describes some aspects of the run-time (and compile-time) performance of the library.

4 Overview Examples

In this section, we provide an overview of many of the major features of the FC++ library, through a number of short, illustrative examples.

FC++ functoids can be simultaneously higher order (able to take functoids as arguments and return them as results) and polymorphic (template functions which work on a variety of data types). For example, consider the library function `compose()`, which takes two functoids and returns the composition:

```
// compose( f, g )(args)    means    f( g(args) )
```

We could define a polymorphic functoid `add_self()`, which adds an argument to itself:

```
// add_self( x )    means    x + x
```

We could then compose `add_self` with itself, and the result would still be a polymorphic functoid:

```
int x = 3;
std::string s = "foo";
compose( add_self, add_self )( x )    // yields 12
compose( add_self, add_self )( s )    // yields "foofoofoofoo"
```

Section 5 describes the infrastructure of these "direct functoids", which enables this feat to be implemented.

FC++ defines a lazy list data structure called `list`. `lists` are lazy in that they need not compute their elements until they are demanded. For example, the functoid `enum_from()` takes an integer and returns the infinite list of integers starting with that number:

```
list<int> l = enum_from( 1 );  // l is infinite list [1, 2, 3, ...]
```

A number of functoids manipulate such lists; for instance `map()` applies a functoid to each element of a list:

```
l = map( add_self, l );  // l is now infinite list [2, 4, 6, ...]
```

The FC++ library defines a wealth of useful functoids and data types. There are named functoids for most C++ operators, like

```
plus(3,4)       // 3+4      also minus, multiplies, etc.
```

There are many functoids which work on `lists`, including `map`. Most of the `list` functions are identical those defined in Haskell[3]. Additionally, a number of basic functions (like the identity function, `id`), combinators (like `flip`: `flip(f)(x,y)==f(y,x)`), and data types (like `list` and `maybe`; `maybe` will be discussed in Section 13) are designed to mimic exactly their Haskell counterparts. We also implement functoids for C++ constructs such as constructor calls and `new` calls:

```
construct3<T>()(x,y,z)   // yields T(x,y,z)
new2<T>()(x,y)           // yields new T(x,y)
```

and many more.

Functoids are curryable. That is, we can call a functoid with some subset of its arguments, returning a new functoid which expects the rest of the arguments. Currying of leading arguments can be done implicitly, as in

```
minus(3)      // yields a new function "f(x)=3-x"
```

Any argument can be curried explicitly using the placeholder variable `_` (defined by FC++):

```
minus(3,_)    // yields a new function "f(x)=3-x"
minus(_,3)    // yields a new function "f(x)=x-3"
```

We can even curry all N of a function's arguments with a call to `thunkN()`, returning a *thunk* (a zero-argument functoid):

```
thunk2( minus, 3, 2 )   // yields a new thunk "f()=3-2"
```

FC++ functoids can be called using a special infix syntax (implemented by overloading `operator^`):

```
x ^f^ y        // Same as f(x,y).  Example: 3 ^plus^ 2
```

This syntax was also inspired by Haskell; some function names (like `plus`) are more readable as infix than as prefix.

FC++ defines *indirect functoids*, which are function variables which can be bound to any function with the same (monomorphic) signature. Indirect functoids are implemented via the `funN` classes, which take N template arguments describing the argument types, as well as a template argument describing the result type. For example:

```
// Note: plus is polymorphic, the next line selects just "int" version
fun2<int,int,int> f = plus;
f(3,2);          // yields 5
f = minus;
f(3,2);          // yields 1
```

Indirect functoids are particularly useful in the implementation of callback libraries and some design patterns[8].

The FC++ library defines a few effect combinators. An effect combinator combines an effect (represented as a thunk) with another functoid. Here are some example effect combinators:

```
// before(thunk,f)(args)  means  { thunk(); return f(args); }
// after(g,thunk)(args)   means  { R r = g(args); thunk(); return r; }
```

An example: suppose you've defined a functoid `write_log()` which takes a string and writes it to a log file. Then

```
before( thunk1( write_log, "About to call foo()" ), foo )
```

results in a new functoid with the same behavior as `foo()`, only it writes a message to the log file before calling `foo()`.

FC++ interfaces with normal C++ code and the STL. The `list` class implements the iterator interface, so that lists can work with STL algorithms and other STL data structures can be converted into `lists`. The functoid `ptr_to_fun()` transforms normal C++ function pointers into functoids, and turns method pointers into functions which take a pointer to the receiver object as an extra first object. Here are some examples, which use currying to demonstrate that the result of `ptr_to_fun` is a functoid:

```
ptr_to_fun( &someFunc )(x)(y)          // someFunc(x,y)
ptr_to_fun( &Foo::meth )(aFooPtr)(x)   // aFooPtr->meth(x)
```

FC++ also has a *lambda* sublanguage for defining anonymous functions on-the-fly, but we hold off describing this feature until Section 12.

5 Direct Functoids

Direct functoids enable the creation of functions which are simultaneously higher-order and polymorphic. Consider the `map()` function described in the previous section: `map()` takes two arguments—a function and a list—and applies the function to every element of the list. In Haskell we would describe the type signature of map like this:

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

The letters `a` and `b` are placeholder type variables (like the `T` in `template <class T>`). The signature says that `map()` takes two arguments—a list of `a` objects and a function from `a` objects to `b` objects—and returns a list of `b` objects. Thus `map()` is an example of a higher-order (it takes a function as an argument) polymorphic (it can be instantiated for all types `a` and `b`) function.

5.1 Issues with representing higher-order polymorphic functions

Representing such a function in C++ is non-trivial. Suppose we try to implement `map` ourselves, and we call our version `mymap`. There are two key issues. The first issue is this: if we represent `mymap` as a template function:

```
template <class F, class T>
... mymap( F some_func, list<T> some_list ) { ... }
```

then the symbol `mymap` does not name a C++ object. This means we cannot pass `mymap` to another higher-order function:

```
some_other_func( mymap, ... );  // illegal if mymap is template function
```

This problem is relatively straightforward to surmount using a function object:

```
struct mymap_type {
   template <class F, class T>
   ... operator()( F some_func, list<T> some_list ) const { ... }
} mymap;
```

Now `mymap` is an instance of `struct mymap_type`, which has a template member `operator()`. As a result, we can call `mymap` using the same syntax as before, but now the symbol `mymap` does name a C++ object, and thus it can itself be passed as a parameter.

The second issue has to do with the return type. What should the return type of `mymap` be? It should be a `list<U>`, where `U` is the result type of applying an `F` function to a `T` object. Since the C++ language lacks a `typeof` operator, we need to represent the return type information ourselves. By convention, in FC++, we represent the return type of a function using a nested template member `struct` named `sig`:

```
struct mymap_type {
   template <class F, class L>
   struct sig {
      typedef list< typename F::template sig<
                       typename L::value_type >::result_type >
         result_type;
   };
   ...
} mymap;
```

More generally, the expression

```
typename F::template sig<X>::result_type
// F::sig<X>::result_type   without the "noise" words
```

represents the result type when a function of type `F` is applied to an argument of type `X`.

As a result, we could define `mymap` as

```
struct mymap_type {
   template <class F, class L>
   struct sig {
      typedef list< typename F::template sig<
                       typename L::value_type >::result_type >
         result_type;
   };
```

```
    template <class F, class T>
    typename sig< list<T> >::result_type
    operator()( F some_func, list<T> some_list ) const { ... }
} mymap;
```

This is our first example of a *functoid*. A functoid is an instance of a struct which contains a (possibly templated) `operator()` method, as well as a nested template member struct named `sig` which works as a return-type computer.

Just as the STL provides helper classes (like `binary_function`) for defining typedefs (like `first_argument_type` and `result_type`), we do the same in FC++. The class

```
funtype< A1, A2, ..., An, R >
```

defines typedefs for argument and result types. (We shall see how this affects the definition of `mymap` shortly.) FC++ functoids support functions of 0-3 arguments.

Also, to simplify naming return types, we have the `RT` helper. Rather than say

```
typename F::template sig<X,Y>::result_type
```

we just say

```
typename RT<F,X,Y>::result_type
```

That is, `RT<F,A1,...An>` computes the result type of a function of type `F` being applied to arguments with type `Ai`.

5.2 The past and the future

FC++ was the first C++ library to use this scheme for representing higher-order polymorphic functions. Since then, a number of other libraries have arisen that all use variations of the same trick to enable return-type deduction.

A relatively new proposal standardizes the return-type deduction methods. It uses conventions and syntax different from FC++. FC++ "full functoids" (Section 7) ensure that functoids are forward-compatible with the new standard. At the same time, the `RT` type computer makes FC++ code backward compatible with functoids using the extant `sig` structures. As a result, FC++ interoperates with other libraries like `boost::bind`.

5.3 Defining a direct functoid

Now, with all of the explanation out of the way, we can finally show the definition of the mymap functoid (Figure 5.3). This is an example of a polymorphic direct functoid. It has a `sig` structure, which is a template over the parameter types which computes the result type, as well as a templated `operator()` function, which uses the `sig` to name its own result type.

```
struct mymap_type {
   template <class F, class L> struct sig
   : public funtype<F,L,
      list<typename RT<F,typename L::value_type>::result_type> > {};
   template <class F, class T>
   typename sig<F, list<T> >::result_type
   operator()( const F& f, const list<T>& l ) const {
      // code to actually implement mymap() function elided
   }
} mymap;
```

Figure 2: The `mymap functoid`

The definition of `mymap` given here is what we call a *basic* direct functoid. In Section 7, we will show how to promote `mymap` into a *full* direct functoid, which adds useful capabilities.

Finally, it should be noted that *monomorphic* direct functoids can be defined without explicitly coding a `sig` structure, by inheriting from the `cfuntype` template class. For example, here is the definition of a function that increments an integer:

```
struct inc_type : public cfuntype<int,int> {
   int operator()( int x ) const {
      return x+1;
   }
} inc;
```

The `cfuntype` class defines a monomorphic `sig` structre which is inherited by the functoid directly.

6 Indirect Functoids

Indirect functoids are "function variables" which can be dynamically bound to any function with the right monomorphic type signature. Recall the example from an earlier section:

```
fun2<int,int,int> f = plus;
f(3,2);          // yields 5
f = minus;
f(3,2);          // yields 1
```

Here `f` is an indirect functoid, which can be bound to different functions (`plus` and `minus`, in the example) during its lifetime. Indirect functoids are declared as instances of the `funN` classes, where N is the arity of the function and the *N+1* template arguments name the argument types and the return type.

Indirect functoids are necessarily monomorphic. This restriction is rooted in the implementation: indirect functoids are implemented using dynamic dispatch (virtual function call), and in C++, a virtual method cannot also be a member

template. When an indirect functoid is initialized or assigned the value of a polymorphic direct functoid, the appropriate monomorphic function is selected. For example:

```
std::string foo="foo", bar="bar";
fun2<std::string,std::string,bool> sl = less;
less(foo,bar);    // yields false
fun2<int,int,bool> il = less;
less(2,3);        // yields true
```

Here we use the FC++ polymorphic functoid `less` (which has the general signature `(T,T)->bool`) to initialize two different indirect functoids. Each indirect functoid selects the appropriate monomorphic instantiation of the polymorphic functoid it is initialized with.

6.1 Subtype-polymorphism

Indirect functoids exhibit subtype-polymorphism (the dynamic polymorphism OO programmers are familiar with). This works in the expected way; indirect functoids are contravariant in their argument types and covariant in their return types. An example makes this clear: suppose there are two inheritance hierachies, one where `dog` is derived from `animal` and another where `car` is derived from `vehicle`. Then we can say

```
fun1<dog*,vehicle*> f = ...;
fun1<animal*,car*> g = f;
f = g;      // illegal (type error)
```

`f` is effectively a subtype of `g`; wherever `g` is used, `f` can be supplied as a legal implementation, but not the other way around.

6.2 Relation to `boost::function`

Indirect functoids in FC++ are similar to `boost::function` objects. There are a few notable differences. First, indirect functoids must always be initialized with a function value (there is no default constructor or the equivalent of a "null" function). Second, indirect functoids have all the FC++ full functoids features (like built-in currying and infix syntax, described in the next section), whereas `boost::function` objects do not. On the other hand, indirect functoids always pass parameters by `const&`, whereas `boost::function` objects can have parameters passed by plain reference, allowing for mutation of arguments to the function object.

7 Full Functoids

In FC++, we use the term *full functoid* to describe functoids which are blessed with all of the special features of FC++. In this section we describe how to promote basic functoids (described at the end of Section 5) into full functoids, and we enumerate the added capabilities that full functoids have.

Full functoids are implemented using the `fullN` wrapper classes (as with the `funN` classes, N describes the arity of the function). Recall in Section 5 we defined `mymap` as a basic direct functoid like this:

```
struct mymap_type {
   // a bunch of stuff
} mymap;
```

Promoting `mymap` into a full functoid is straightforward:

```
namespace impl {
   struct xmymap_type {
      // a bunch of stuff
   };
}
typedef full2<impl::xmymap_type> mymap_type;
mymap_type mymap;
```

That is, rather than have `mymap_type` refer to the direct functoid `struct` we have defined, instead we make it a `typedef` for `full2` instantiated with that type. (In FC++, we conventionally use a namespace called `impl` to store the actual definitions of basic direct functoids, and define the full functoids out in the main namespace. We also mangle the original name (prefixing an "x") in order to ensure that the basic functoid type will not be accidentally found instead of the full functoid type when various C++ name-lookup rules kick in.[3]) That's all there is to it.

Indirect functoids need no such promotion. Since the indirect functoid types (the `funN` classes) are defined by the library, they are already imbued with all of the full functoid features. We describe these features next.

7.1 Currying

As described in Section 4, all full functoids exhibit built-in currying. For example, given a 3-argument full functoid "`f`", we can call it with any subset of its arguments, either by using underscores as placeholders, or by leaving off trailing arguments. Some examples:

```
f(x,y,z)          // normal call
f(x,_,z)          // yields a new unary functoid (expecting y)
f(_,y,_)          // yields a new binary functoid (expecting x,z)
f(x,y)            // yields a new unary functoid (expecting z)
f(x)              // yields a new binary functoid (expecting y,z)
```

Additionally, all of the arguments can be curried, resulting in a "thunk" (a nullary functoid), by calling `thunkN`:

```
thunk3(f,x,y,z)   // yields a new nullary functoid
```

Thunks will be described more in Section 8.

[3]This seems ugly, but Brian can't find a better way to effectively hide the names of basic functoids. He invites email advice/discussion on the topic.

7.2 Infix Syntax

Binary and ternary full functoids can be called using a special infix syntax. This is syntactic sugar, as some functions "read" better using infix notation:

```
plus(2,3)        // normal (prefix) function syntax
2 ^plus^ 3       // FC++ infix syntax
```

For ternary functoids, note that infix syntax automatically invokes currying of the final argument:

```
f(x,y,z)         // normal (prefix) function syntax
f(x,y)           // as before: yields a new unary functoid (expecting z)
x ^f^ y          // FC++ infix syntax (new unary functoid expecting z)
```

FC++ infix syntax was inspired by a similar construct in the Haskell programming language.

7.3 Lambda Awareness

Full functoids are lambda-aware. This means that they can be called with square brackets `[]` instead of round ones `()` to create lambda expressions. Lambda is discussed in Section 12.

7.4 Smartness

Full functoids exhibit another feature which we have chosen to label "smartness". Full functoids know how many arguments they can accept; a traits class provides access to this information. If `F` is a full functoid type, then these values are available at compile-time:

```
FunctoidTraits<F>::template accepts<N>::args
// A bool which says whether F can accept N arguments

FunctoidTraits<F>::max_args
// An int which says what the most arguments F can accept is
```

and also this operation is available

```
FunctoidTraits<F>::template ensure_accepts<N>::args()
// A no-op call that compiles only if F can accept N args
```

to ensure that compilation halts with an error if the specified function does not accept the expected number of arguments.

We use a traits class to provide literate error messages (along the lines of [5, 7]) so that some common abuses (like passing an `F` that is not a full functoid) are caught within the library and a helpful identifier is injected into the compiler error message.

7.5 Return-type deduction

C++ will eventually have a `typeof` operator, but in the meantime, the standards committee has come up a stop-gap solution to return-type deduction [9]. The standard solution will be to use, e.g.,

```
result_of<F(X,Y)>::type
```

to determine the result of an object of type `F` being called with arguments of type `X` and `Y`. The `result_of` template knows how to work on function pointers, monomorphic function objects, and polymorphic function objects.

FC++ basic functoids use a different convention for defining return-type deduction within the FC++ framework (described in Section 5). However, full functoids implement the standard convention (using `result_of`) as well. As a result, FC++ full functoids interoperate with other template libraries that require return-type-deduction facilities.

8 Effects and thunks

FC++ is designed to be a library for pure functional programming. Nevertheless, this is C++, and it is reasonable to want to use effects. In this section, we describe how functoids with effects can be created.

There are three main ways to have effects inside an FC++ functoid:

- use a global variable
- use pointers
- use a "thunk"

We show examples of each in turn.

First, a global variable:

```
struct some_functoid_type : public cfuntype<int,int> {
   int operator()( int x ) const {
      std::cout << "Hello from inside some_functoid";
      return x;
   }
} some_functoid;
...  some_functoid(4) ...
```

When the functoid is called, text appears on the standard output stream due to the effect on the global variable `std::cout`.

Second, using pointers:

```
struct incr_type : public cfuntype<int*,int> {
   int operator()( int* p ) const {
      return ++*p;
   }
} incr;
```

```
...
int x;
... incr( &x ) ...
```

Here the functoid manipulates the value of variable `x` through a pointer.

Finally, thunks:

```
// building off last example
fun0<int> f = thunk1( incr, &x );   // thunk that returns a higher
f(); f(); f();                  // integer each time it is invoked
```

In pure functional programming, there is no reason to ever have a zero-argument function (it would effectively be the same as a constant value). As a result, nullary functoids (thunks) always represent functoids which have some side-effect.

9 Lists and lazy evaluation

In Section 4, we showed examples of using FC++'s lazy list data structure:

```
list<int> l = enum_from( 1 ); // l is infinite list [1, 2, 3, ...]
l = map( add_self, l );       // l is infinite list [2, 4, 6, ...]
```

In this section, we discuss the interface to the `list` class, and show how to implement lazy list functions. We also discuss the general topic of lazy evaluation.

9.1 Interface to `list`

The main interface to the `list` class is provided by just a few functoids:

```
list<int> l;      // empty list (default constructor)
l = cons(2,l);    // cons adds an element to the front of a list
l = cons(1,l);    // l is now the list [1,2]
int x = head(l);  // x is 1, the front element
l = tail(l);      // l is now the list [2], the "rest" of the list
bool b = null(l); // b is false; null() tests for the empty list
l = cat(l,l);     // l is now [2,2]; cat() concatenates two lists
l = NIL;          // l is now empty; NIL is the empty list constant
```

This is the typical interface offered for singly-linked lists common to many functional languages.

9.2 Writing lazy list functoids

In order to enable lazy evaluation, the `list` class has a special constructor which takes a thunk which returns a list. The second argument to `cons()` or `cat()` can also be a thunk-returning-a-list, rather than a list. For example, after

```
list<int> l = thunk2(cons,2,NIL);
list<int> l = thunk2(cons,1,l);
```

`l` is the list [1,2], except that none of the conses has been evaluated yet. This is not particularly interesting in itself, but now we can see how to write functions like `enum_from()`, which return infinite lists.

First, here is how we would write an eager (non-lazy) version of `enum_from()`, which goes into an infinite recursion when called. (For simplicity, we define it as a monomorphic functoid that works only on integers.)

```
struct my_enum_from_type : public cfuntype<int,list<int> > {
   list<int> operator()( int x ) const {
      return cons( x, my_enum_from_type()(x+1) );
   }
} my_enum_from;
```

Now, all we have to do to make this function lazy is to "thunk" the recursive call like so:

```
struct my_enum_from_type : public cfuntype<int,list<int> > {
   list<int> operator()( int x ) const {
      return cons( x, thunk1( my_enum_from_type(), x+1 ) );
   }
} my_enum_from;
```

This delays the recursive call so that it is stored in the "tail" portion of the cons, where it won't be evaluated until demanded. Here is an example that demonstrates the function being used:

```
list<int> l = my_enum_from(1);
for( int i=0; i<10; ++i ) {
   std::cout << head(l) << std::endl;
   l = tail(l);
}
```

This prints out the first 10 positive integers. We could print out as many as we like, as the list `l` is effectively infinite; none of the cons cells representing the list are created until they are demanded.

9.3 Other details about lazy evaluation

The discussion here provides a simple overview of lazy evaluation as implemented in the FC++ `list` class. We have elided a number of interesting details which can impact the performance of lists, most notably, the existence of the `oddlist` class and the caching ability of lists. To learn more about these details, the reader is referred to

- Section 10 of [4], which describes caching in lists, as well as some other performance optimizations,
- Section 11 of [4], which describes `lists` versus `oddlists` and the efficient list interface, and
- the FC++ web page [2], which has summary documentation on the topic.

Lists provide perhaps the most common and convenient way to utilize lazy evaluation; representing a (possibly infinite) stream of data which is computed "on demand" is an oft-used pattern. Nevertheless, any computation can be lazified. The `by_need` monad (see Section 13 for info about monads) illustrates a more general mechanism for lazifying any computation.

10 Library

In this section, we briefly describe each of the components in the library's interface.

10.1 Nuts and bolts

To use the Boost FC++ library, just add the line

```
#include "boost/fcpp/prelude.hpp"
```

to the top of your program. This file includes all of the other header files. All of the library interface is in `namespace boost::fcpp`.

Note that, by default, the "lambda" and "monad" portions of the library are not enabled. This is both because some compilers cannot handle the crazy templates there, and also because they slow down compilation a little, even if they're not being used. To enable these portions of the library, say

```
#define BOOST_FCPP_ENABLE_LAMBDA
```

before `#include`-ing the library.

The library comes with dozens of example client files (the `.cpp` files in the FC++ directory). When in doubt about how to use something, check the client examples, which exhibit coverage of most of the library's features.

10.2 Constants

FC++ defines the following constants:

```
_          // placeholder for currying
empty      // an empty struct (empty tuple)
NIL        // the empty list (zero of the list_m monad)
NOTHING    // an empty maybe (zero of the maybe_m monad)
```

10.3 Data types

FC++ defines these data types:

```
// Useful data types
list             // for lazy lists
list_iterator
maybe            // represents 1 or 0 elements of some type
odd_list         // (see section on lists for details)
```

```
// Utility (see sections on "direct functoids" and "full functoids")
RT               // Return type computer (e.g.  RT<F,X,Y>::result_type)
FunctoidTraits   // for seeing how many arguments a functoid accepts
cfuntype         // typedefs for monomorphic functoids
funtype          // typedefs for polymorphic functoids

// Miscellaneous
fullN            // full functoid wrapper classes
funN             // indirect functoid classes
fcpp_exception   // used, e.g., when taking head() of an empty list
```

Note also that every functoid has a corresponding data type. For instance, the `map()` functoid is an instance of type `map_type`.

10.4 Basic list functions

Here are the basic functions that work on lists.

```
head           // first element
tail           // all but first element
cons           // add element
null           // test for empty
cat            // concatenate
==             // compare for equality
<              // lexicographical ordering
```

as well as some other miscellaneous list functions:

```
list_with      // helper for creating small lists
force          // odd_list-ify       (see section on lists)
delay          // (even) list-ify    (see section on lists)

list_until( pred, f, x )
   // create a list of [x, f(x), f(f(x)), ...] until pred is true
   // Example:   list_until( greater(_,20), plus(3), 1 )
   // yields     [1,4,7,10,13,16,19]
```

10.5 Haskell standard prelude

A great many FC++ functions are borrowed from Haskell. See [3] for their definitions.

```
until
last
init
length    // Note: also used to force evaluation of an entire list
at
filter
concat
foldr
```

```
foldr1
foldl
foldl1
scanr
scanr1
scanl
scanl1
iterate
repeat
map
take
drop
take_while
drop_while
replicate
cycle
split_at
span
break
flip
reverse
all
any
elem
not_elem
sum
product
minimum
maximum
zip_with
zip
fst
snd
unzip
gcd
odd
even
enum_from
enum_from_to
just
// These approximate the corresponding Haskell functions
and
or
h_curry
h_uncurry
```

10.6 Operators

The following named functoids mimic C++ operators:

```
// (T,T) -> T
plus
minus
multiplies
divides
modulus

// (T) -> T
negate

// (T,T) -> bool
equal
not_equal
greater
less
greater_equal
less_equal

// (T,T) -> bool    (where T is convertible to bool)
logical_and
logical_or
logical_not

dereference    // T -> U   (where U has typeof(*T) )
address_of     // T -> T*
delete_        // T* -> void
```

The following operators require extra explanation

```
out_stream
in_stream
   // These are like operator<<() and operator>>(), but they take a
   // pointer to a stream as the left-hand side.  Examples:
   //    &cout ^out_stream^ x
   //    &cin ^in_stream^ y
   // The indirection is necessary to encapsulate the effects within
   // the library (see section on effects).

dynamic_cast_
   // This is a family of functoids, templatized by the destination
   // type.  Example use:
   //    dynamic_cast_<dog*>()( an_animal_ptr )
   // Note that the functoid
   //    dynamic_cast_<T>()
   // has type
   //    dynamic_cast_x_type<T>::type

constructN
newN
   // These call constructors.  Like dynamic_cast_ above, they define
   // a family of functoids.  Examples:
```

```
   //    construct2<std::pair<int,char> >()( 3, 'c' )
   //    new1<int>()( 3 )
   // Note that, e.g.,
   //    new3<T>()
   // has type
   //    new3_type<T>::type
```

10.7 Currying, thunks, and effect combinators

```
const_   // Turns a value into a thunk
   // Ex:   const_(3)  yields a new function "f":  f()=3

thunkN   // binds all N arguments of a function into a thunk
   // Ex:   thunk2(plus,1,2) yields a new function: f()=1+2

no_op    // do-nothing thunk

before   // before(f,g)(args) = { f(); return g(args); }
after    // after(f,g)(args)  = { r = f(args); g(); return r; }

emptify  // throws away a functions result, returning "empty" instead
   // Example use:
   //    length( map( emptify(effectFunctoid), someList ) )
   // applies effectFunctoid to each element of someList, even if
   // effectFunctoid returns void

uncurry  // uncurry(f)(x,y,z)  means  f(x)(y)(z)
   // This is rarely useful, but occasionally necessary
```

10.8 General combinators

These are some generally applicable combinators.

```
compose     // compose(f,g)(args)  means  f( g(args) )

of          // same as compose, but also works on function pointers
            // Good for infix:   f ^of^ g

thunk_func_to_func
   // thunk_func_to_func(f)(args)  means  f()(args)
   // f is a thunk that returns a functoid; this combinator hides
   // the thunk.  This can be useful to break what would otherwise
   // be infinite recursion in "letrec" expressions.

duplicate   // duplicate(f)(x)     means  f(x)(x)

ignore      // ignore(f)(x)(args)  means  f(args)
```

10.9 Functoid promotion

These functions promote non-functoids into functoids.

```
make_fullN
   // promotes an instance of a basic functoid into a full functoid

//stl_to_fun1 FIX?
//stl_to_fun2 FIX?

ptr_to_fun  // converts most anything into a functoid

funify      // Converts function pointers into functoids, but is
            // the identity function on functoids.  Use this when
            // you're not sure exactly what "f" is, but you want it
            // to be a functoid.
```

10.10 Other

There are a few other miscellaneous functoids:

```
make_pair      // creates a std::pair
min            // lesser of two args
max            // greater of two args
inc            // returns ++x
dec            // returns --x
id             // identity function

make_manip     // makeManip(aStream)(aManip) returns the manipulator
               // for that stream.  Necessary because names like
               // "std::endl" are not C++ objects.  Ugh.
```

10.11 Note about lambda/monads

We do not describe the interface to the FC++'s lambda and monad constructs here. See Sections 12 and 13 for that info.

11 Relationships with other libraries

In this section we briefly describe the relationship between FC++ and other C++ libraries.

11.1 Interfaces to STL

The main interface to the STL is via iterators in the `list` class. FC++ `lists`, like STL containers, have a constructor which takes two iterators deliniating a range, so that the contents of an STL container can be (lazily) copied into a `list`. For example:

```
std::vector<int> v = ...;
fcpp::list<int> l( v.begin(), v.end() );
// l is not yet evaulated; still holds iterators into v
fcpp::length( l );  // force evaluation of entire list
// now l has a true copy of the elements in v
```

Additionally, `lists` themselves have forward iterators:

```
for( fcpp::list<int>::iterator i = l.begin(); i != l.end(); ++i )
   cout << *i;      // print each element of list
```

The functions in the FC++ library that work on `fcpp::lists` do not work on other STL containers, as those containers do not support lazy evaluation.

Monomorphic FC++ (unary or binary) functoids are STL "adaptables". The `fcpp::ptr_to_fun()` functoid promotes any function-like entity, including STL function objects, into FC++ full functoids.

11.2 Relation to Boost

FC++ is related to a number of Boost libraries.

11.2.1 Boost `bind and lambda`

FC++'s lambda (see Section 12) and currying (Section 7) capabilities do approximately the same thing that `boost::lambda` and `boost::bind` do. These libraries were developed with different design rationales; for a description of the comparison, see [6].

Since FC++ supports the `result_of` method for return-type-deduction (see Section 7), FC++ interoperates with `boost::lambda` and `boost::bind`.

11.2.2 Boost `function`

FC++ indirect functoids (Section 6) are similar to `boost::function` objects. Indirect functoids have all of FC++'s full functoids capabilities (like currying and infix syntax; see Section 7) built in. Indirect functoids can only pass parameters by value (actually, `const&`), though (as FC++ is a library for pure functional programming).

11.2.3 Other Boost libraries

FC++ uses a number of other boost libraries in its implementation:

- `boost::intrusive_ptr` in indirect functoids and `lists`, for automatic memory management
- `boost::is_base_and_derived` and `boost::is_convertible` in a number of places
- `boost::addressof` to implement the `address_of()` functoid

```
// declaring lambda variables
lambda_var<1> X;
lambda_var<2> Y;
lambda_var<3> F;

// basic examples
lambda(X,Y)[ minus[Y,X] ]       // flip(minus)
lambda(X)[ minus[X,3] ]         // minus(_,3)

// infix syntax
lambda(X,Y)[ negate[ 3 %multiplies% X ] %plus% Y ]

// let
lambda(X)[ let[ Y == X %plus% 3,
                F == minus[2]
          ].in[ F[Y] ] ]

// if-then-else
lambda(X)[ if0[ X %less% 10, X, 10 ] ]   // also if1, if2

// letrec
lambda(X)[ letrec[ F == lambda(Y)[ if1[ Y %equal% 0,
                                        1,
                                        Y %multiplies% F[Y%minus%1] ]
          ].in[ F[X] ] ]    // factorial
```

Figure 3: Lambda in FC++

- `boost::type_with_alignment` and `boost::alignment_of` in the implementation of the `list` and `by_need` datatypes
- `boost::noncopyable` in a number of places

12 Lambda

In this section, we describe the interface to FC++'s lambda sublanguage. Those readers interested in the motivation and design rationale for FC++ lambda should read [6], which discusses those issues in detail.

12.1 Lambda in FC++

We now describe what it looks like to do lambda in FC++. Figure 3 shows some examples of lambda. There are a few points which deserve further attention.

Inside lambda, one uses square brackets instead of round ones for postfix functional call. (This works thanks to the lambda-awareness of full functoids, mentioned in Section 7.) Similarly, the percent sign is used instead of the carat for infix function call. Since `operator[]` takes only one argument in C++, we

overload the comma operator to simulate multiple arguments. Occassionally this can cause an early evaluation problem, as seen in the code here:

```
// assume f takes 3 integer arguments
lambda(X)[ f[1,2,X] ]    // oops! comma expression "1,2,X" means "2,X"
lambda(X)[ f[1][2][X] ]  // ok; use currying to avoid the issue
```

Unfortunately, C++ sees the expression "`1,2`" and evaluates it eagerly as a comma expression on integers.[4] Fortunately, there is a simple solution: since all full functoids are curryable, we can use currying to avoid comma. The issues with comma suggest another problem, though: how do we call a zero-argument function inside lambda? We found no pretty solution, and ended up inventing this syntax:

```
// assume g takes no arguments and returns an int
// lambda(X)[ X %plus% g[] ]   // illegal: g[] doesn't parse
lambda(X)[ X %plus% g[_*_] ]   // _*_ means "no argument here"
```

It's better to have an ugly solution than none at all.

The if-then-else construct deserves discussion, as we provide three versions: `if0`, `if1`, and `if2`. `if0` is the typical version, and can be used in most instances. It checks to make sure that its second and third arguments (the "then" branch and the "else" branch) will have the same type when evaluated (and issues a helpful custom error message if they won't). The other two ifs are used for difficult type-inferencing issues that come from `letrec`. In the factorial example at the end of Figure 3, for example, the "else" branch is too difficult for FC++ to predict the type of, owing to the recursive call to `F`. This results in `if0` generating an error. Thus we have `if1` and `if2` to deal with situations like these: `if1` works like `if0`, but just assumes the expression's type will be the same as the type of the "then" part, whereas `if2` assumes the type is that of the "else" part. In the factorial example, `if1` is used, and thus the "then" branch (the `int` value `1`) is used to predict that the type of the whole `if1` expression will be `int`.

12.2 Naming the C++ types of lambda expressions

Expression templates often yield objects with complex type names, and FC++ lambdas are no different. For example, the C++ type of

```
// assume:  LambdaVar<1> X;  LambdaVar<2> Y;
lambda(X,Y)[ (3 %multiplies% X) %plus% Y ]
```

is

```
fcpp::Full2<fcpp::fcpp_lambda::Lambda2<fcpp::fcpp_lambda::exp::
Call<fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Value<
fcpp::Full2<fcpp::impl::XPlus> >,fcpp::fcpp_lambda::exp::CONS<
fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Call<fcpp::
```

[4] Some C++ compilers, like g++, will provide a useful warning diagnostic ("left-hand-side of comma expression has no effect"), alerting the user to the problem.

```
fcpp_lambda::exp::Value<fcpp::Full2<fcpp::impl::XMultiplies> >,
fcpp::fcpp_lambda::exp::CONS<fcpp::fcpp_lambda::exp::Value<int>,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<1>,fcpp::fcpp_lambda::exp::NIL> >,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<2>,fcpp::fcpp_lambda::exp::NIL> >,1,2> >
```

In the vast majority of cases, the user never needs to name the type of a lambda, since usually the lambda is just being passed off to another template function. Occasionally, however, you want to store a lambda in a temporary variable or return it from a function, and in these cases, you'll need to name its type. For those cases, we have designed the `LE` type computer, which provides a way to name the type of a lambda expression. In the example above, the type of

```
lambda(X,Y)[ (3 %multiplies% X) %plus% Y ]
// desugared: lambda(X,Y)[ plus[ multiplies[3][X] ][Y] ]
```

is

```
LE< LAM< LV<1>, LV<2>,   CALL<CALL<plus_type,
   CALL<CALL<multiplies_type,int>,LV<1> > >,LV<2> > > >::type
```

The general idea is that

```
LE< Translated_LambdaExp >::type
```

names the type of `LambdaExp`. Each of our primitive constructs in lambda has a corresponding translated version understood by `LE`:

```
CALL            [] (function call)
LV              lambda_var
IF0,IF1,IF2     if0[],if1[],if2[]
LAM             lambda()[]
LET             let[].in[]
LETREC          letrec[].in[]
BIND            lambda_var == value
```

With `LE`, the task of naming the type of a lambda expression is still onerous, but `LE` at least makes it possible. Without the `LE` type computer, the type of lambda expressions could only be named by examining the library implementation, which may change from version to version. `LE` guarantees a consistent interface for naming the types of lambda expressions.

Finally, it should be noted that if the lambda only needs to be used monomorphically, it is far simpler (though potentially less efficient) to just use an indirect functoid:

```
// Can name the monomorphic "(int,int)->int" functoid type easily:
Fun2<int,int,int> f = lambda(X,Y)[ (3 %multiplies% X) %plus% Y ];
```

12.3 FC++ lambda versus `boost::lambda`

Whereas FC++'s lambda and `boost::lambda` superficially appear to do the same thing, they are actually quite different. FC++'s lambda uses explicit lambda syntax to create a minimal sublanguage with language constructs found in pure functional languages (e.g. `letrec`). On the other hand, `boost::lambda` supplies almost the entire C++ language in its lambda, overloading every possible operator in lambda expressions, which can be created implicitly just by using a placeholder variable (like `_1`) in the midst of an expression. For more discussion about the differences, see [6].

13 Monads

Monads provide a useful way to structure programs in a pure functional language. The FC++ library now implements a few monads, and provides syntax support for "do-notation" and "comprehensions" in arbitrary monads.

Monads are a particularly abstract and difficult topic to explain. Rather than try to explain monads here (and double the size of this document), we refer the reader to the discussion of monads in [6], as well as the documentation about lambda and monads on the FC++ web site [2].

14 Applications

FC++ has a number of application areas, which we describe here briefly.

- *General:* FC++ is useful for its handling of function objects in a general setting. The FC++ way of saying things is often more succinct than `boost::bind` or the standard library:

```
FC++                  STL/boost::bind
----                  ---------------
fcpp::plus            std::plus<int>()
minus(_,3)            std::bind2nd( std::plus<int>(), 3 )
minus(_,3)            boost::bind( std::plus<int>(), _1, 3 )
```

  though for a number of small cases, `boost::lambda` provides the tiniest code:

```
FC++                  boost::lambda
----                  -------------
minus(_,3)            _1 - 3
```

  FC++'s lambda syntax is explicit, which sometimes makes it easier to create complicated functions on the fly. Here is a somewhat contrived example (loosely based on a thread on the Boost mailing list)

```
void g( std::string s, int x );
void h( fcpp::fun0<void> f );
...
fcpp::fun1<int,void> f =
   lambda(X)[ ptr_to_fun(h)[
      lambda()[ ptr_to_fun(g)[ std::string("hi"), X ]
         ] ] ];
```

which is relatively straightforward to express using FC++, but difficult to express using `boost::{lambda,function,bind}`.

- *Design patterns:* FC++ is useful in the implementation of a number of OO design patterns. See [8] for details.
- *Lazy evaluation:* 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 [1]. XR uses the FC++ infrastructure to provide exact (or *constructive*) real-number arithmetic, using lazy evaluation.
- *Parsing:* In Haskell, "monadic parser combinators" are all the rage. FC++'s support for monads makes it possible to implement these libraries in C++. See [6] for more discussion, and the `parser.cc` client file for an example.
- *Other applications:* FC++ has been used as a basis on which to build other libraries as well. See the "customers" section of the FC++ web page [2] for details.

Prior to "Boost FC++", the main "audience" for FC++ was the functional programming research community. We hope that inclusion in Boost will help find more matches between the general C++ community's needs and the features provided by FC++.

15 Performance

FC++ uses a number of optimizations to improve the performance of its more "dynamic" features, like indirect functoids and lists. Sections 9 and 10 of [4] quantify the library's performance and discuss a number of the optimizations we have implemented.

For the most part, using FC++ means using direct functoids, which are templates which get instantiated (and possibily inlined) at compile-time; this use of FC++ implies no extra overhead.

FC++ is a large library, and `#include`-ing it is bound to slow down your compile-time a little. The lambda and monad facilities utilize expression templates, and expressions templates tend to slow down compile-times a lot.

References

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

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

[3] *Haskell 98 Language Report.* Available online at
`http://www.haskell.org/onlinereport/`

[4] McNamara, Brian and Smaragdakis, Yannis. "Functional Programming with the FC++ library" *Journal of Functional Programming*, to appear.

[5] McNamara, Brian and Smaragdakis, Yannis. "Static Interfaces in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at
`http://www.oonumerics.org/tmpw00/`

[6] McNamara, Brian and Smaragdakis, Yannis. "Syntax sugar for FC++: lambda, infix, monads, and more" *DPCOOL'03* Uppsala, Sweden. Available at `http://www.cc.gatech.edu/~yannis/fc++/`

[7] Siek, Jeremy and Lumsdaine, Andrew. "Concept Checking: Binding Parametric Polymorphism in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at `http://www.oonumerics.org/tmpw00/`

[8] Y. Smaragdakis and B. McNamara, "FC++: Functional Tools for Object-Oriented Tasks" *Software Practice and Experience*, August 2002.

[9] A uniform method for computing function object return types `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1454.html`