Functional Programming with the FC++ Library

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

July 26, 2001

FC++ is a library for functional programming in C++. At a high-level, FC++ can be described as "STL functionals on steroids". Compared to all other libraries for functional programming in C++ (including the Standard Library) FC++ is distinguished by its powerful type system, which enables the library to reproduce a large amount of useful functionality (including a large portion of the Haskell Standard Prelude, an STL-compatible lazy list data structure, and several general operators on functions). This paper is addressed to an audience familiar with C++ and attempts to clarify what FC++ does and how.

1 What is FC++?

FC++ [6] 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.

Functional programming is a programming paradigm in which functions are treated as regular values. Readers familiar with the "functional" part of the Standard Library have already encountered some of the ideas behind FC++. 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) [2][8] or a framework for expressing higher-order function types[5].

FC++ is distinguished from all such libraries by its powerful type system. FC++ offers complete support for manipulating polymorphic[1] 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

[1] Throughout the paper, we use the term "polymorphic" to refer to *parametric polymorphism* (e.g. template functions). This is a common use for the term among functional programmers, though object-oriented programmers often use the word "polymorphism" to mean *subtype polymorphism* (e.g. dynamic dispatch). In this paper, "polymorphism" means parametric polymorphism by default.

and the C++ type system. Unlike other libraries for functional programming in C++, FC++ does not focus on improving syntax by using either the preprocessor 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 (like in the "Lambda Library" [2] or "FACT!" [8]) is an orthogonal issue. This is often a point of confusion, so we would like to emphasize it again: the only "template computation" in FC++ is for the type system. That is, template functions are used in FC++ for determining the types of complicated polymorphic operations (e.g., the polymorphic type of the result of composing two polymorphic functions). Nevertheless, the actual code contained in functions is regular C++ expressions. Any piece of C++ code can be easily wrapped as an FC++ functoid. Anything expressible in the base C++ language (including template functions) is also expressible using the FC++ conventions with only minor additions, so that it can be used with the rest of the functionality in the library.

The FC++ library currently comprises about 4000 lines of C++ code. We are continuing to develop the library to make it more expressive, more efficient (a large part of our recent work), and more convenient to use.

2 What can I do with FC++?

In this section, we will walk 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. We use the functions `cons()` (which adds an element to the front of a list), `head()` (the first element of a list), `tail()` (the rest of the list), and the constant `NIL` (empty list), which are common in functional languages.

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

```
#include <assert.h>
#include <string>
#include "prelude.h"
using namespace fcpp;

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

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, which returns only those elements of a list that 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

```
#include <assert.h>
#include "prelude.h"
using namespace fcpp;

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

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

and then use, for example,

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

to compute the (infinite) 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()` is a function that 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

```
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 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

```
#include <assert.h>
#include "prelude.h"
using namespace fcpp;

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"
using namespace fcpp;

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

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 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,char>` describes a two-argument function which takes two `int`s and returns a `char`. The function `makeFunN()` converts a direct functoid into an indirect one. In the case of poly-

```
#include <assert.h>
#include <vector>
#include <algorithm>
#include "prelude.h"
using namespace fcpp;
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

morphic 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 create your own polymorphic functoid, you need to create a class with two main elements: a template `operator()` and a member structure template named `Sig`. To make things concrete, consider the definition of `map` (or rather, the class `Map`, of which `map` is a unique instance) shown in Figure 7. This definition uses the helper template `FunType`, which is a specialized template for different numbers of arguments. For two arguments, `FunType` is essentially:

```
template <class A1, class A2, class R>
struct FunType {
  typedef R ResultType; typedef A1 Arg1Type;
  typedef A2 Arg2Type;  };
```

```
#include <assert.h>
#include "prelude.h"
using namespace fcpp;
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) );

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

Figure 6: Indirect functoids examples

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

 template <class F, class T>
 typename 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)),
                  curry2(Map(), f, tail(l))); }
};
```

Figure 7: Map in FC++

We can now analyze the implementation of `Map`. 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>
```

(we have elided the `typename` and `template` keywords for readability). 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`.[2] 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).

[2] 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++.

4 Performance Tests

FC++ is quite efficient in its implementation of functional concepts. For common programming tasks that use the FC++ conventions, the overhead is either zero or negligible (i.e., just a dynamic dispatch indirection for indirect functoids). The only case where performance is a legitimate concern is if one attempts to copy functional idioms directly to C++ using FC++. FC++ is not an optimizing compiler for a functional language, so it misses several common optimizations; for example: no special runtime support for specialized functions exists; tail-recursion elimination is not automatically performed; no runtime support for lazy evaluation exists. Additionally, FC++ offers a simple reference counting mechanism (used internally for indirect functoids and lazy lists), which is not directly comparable to an optimized garbage collector. Nevertheless, the implementation of FC++ carefully tries to avoid unnecessary overhead and a number of optimizations are employed. In a previous paper [6] we showed that the FC++ reference counting mechanism is much faster (by a factor of 6 to 8) than another technique used in the literature [5] for keeping track of functoid references.

In this section we show some simple performance measurements comparing FC++ to Hugs (a well-known Haskell interpreter) and ghc (an optimizing Haskell compiler). The benchmarks are programs that C++ programmers are unlikely to write in this form, but they show common functional programming idioms, involving heavy use of lazy (infinite) lists. Therefore, these benchmarks serve as stress tests of FC++'s lazy lists.

For each benchmark, we wrote two programs: one in Haskell, and one in C++ using the FC++ library. The programs are faithful translations of each other; they each represent the same solution to the given problem. The programs were run on a Sun Sparc Ultra-30 with 128M of RAM. We used g++2.95.2, ghc5.00.1, and the February 2001 version of Hugs. In

```
divisible t n = t `rem` n == 0

factors x = filter (divisible x) [1..x]

prime x = factors x == [1,x]

primes n = take n (filter prime [1..])

l = primes 600

main =do print (l !! 599)
```

Figure 8: Primes in Haskell

the case of both g++ and ghc, we used -O2 and static linking.

(In the C++ code for the benchmarks, you may notice that `OddLists` are used in addition to `Lists`. `OddLists` are distinguished from `Lists` to expose the fact that the first element of the list is an evaluated `cons`; this enables other optimizations in our list implementation.)

4.1 Primes

Primes is a simple program that computes a (lazy) list of the first N prime numbers and then prints the Nth prime. It does so simply by filtering all the primes from the (infinite) list of integers, and then taking the first N of them. Figure 8 shows the code for primes in Haskell. Figure 9 shows the code for primes in FC++.

Table 1 shows the performance results for primes for various values of N. FC++ is about 55 times as fast as Hugs for this program, and also consistently faster than ghc. The reason FC++ is faster than ghc appears to be due to the integer arithmetic: ghc uses 64-bit integers (`long long`s) for all integer operations. Indeed, if the C++ version of the program is changed to use `long long` instead of `int`, then the times are practically equal.

4.2 Tree

Tree is a program that generates a random binary search tree of integers and then (lazily) computes the

```
#include <iostream>
#include "prelude.h"
using namespace fcpp;
using std::cout; using std::endl;

struct Divisible : public CFunType<int,int,bool> {
   bool operator()( int x, int y ) const
   { return x%y==0; }
} divisible;

struct Factors : public CFunType<int,OddList<int> > {
   OddList<int> operator()( int x ) const {
      return filter( curry2(divisible,x),
                     enumFromTo(1,x) );
   }
} factors;

struct Prime : public CFunType<int,bool> {
   bool operator()( int x ) const {
      return factors(x)==cons(1,cons(x,NIL));
   }
} prime;

struct Primes : public CFunType<int,OddList<int> > {
   OddList<int> operator()( int n ) const {
      return take( n, filter( prime, enumFrom(1) ) );
   }
} primes;

int main() {
   OddList<int> l = primes(NUM);
   cout << at( l, NUM-1 ) << endl;
}
```

Figure 9: Primes in FC++

N	FC++	ghc	Hugs
200	0.26	0.27	13
400	1.17	1.21	60
600	2.64	3.46	146
800	4.89	5.37	271
1000	7.77	8.56	424

Table 1: Primes (all times in seconds)

```
data Tree a = Node !a !(Tree a) !(Tree a)
            | Nil

leaf (Node _ Nil Nil) = True
leaf (Node _ _ _)     = False

fringe Nil             = []
fringe n@(Node d l r)
   | leaf n            = [d]
   | otherwise         = fringe l ++ fringe r

main =do --// code to make a random tree "t"
         print (filter (== 13) (fringe t))
```

Figure 10: Tree in Haskell

N	FC++	ghc	Hugs
10000	0.08	0.03	0.24
20000	0.19	0.06	0.56
30000	0.29	0.10	0.89
40000	0.41	0.12	-
80000	0.87	0.26	-
160000	1.69	0.56	-

Table 2: Tree (all times in seconds)

"fringe" of the tree. The fringe of a tree is a list of all of the leaves of the tree, in the order they are encountered during an inorder traversal. The main program prints all of the nodes in the fringe that match an arbitrary value (13 in the listings); this is merely a convenient way to force the evaluation of the lazy list.

Figure 10 shows the Haskell code for tree; Figure 11 shows tree in FC++. For both the Haskell and C++ programs, the code that actually builds the random binary trees is elided from the listings.

Table 2 shows the performance results for tree. N is the number of nodes in the tree. No results are reported for Hugs for more than 30,000 nodes because the system memory was exhausted. For this benchmark, FC++ is consistently faster than Hugs, but about three times slower than ghc. Investigating the disparity between the FC++ and ghc performance,

```
#include <iostream>
#include "prelude.h"
using namespace fcpp;
using std::cout; using std::endl;

struct Tree {
   int data;
   Tree *left;
   Tree *right;

   Tree( int x ) : data(x), left(0), right(0) {}
   Tree( int x, Tree* l, Tree* r )
      : data(x), left(l), right(r) {}
   bool leaf() const
   { return (left==0) && (right==0); }
};

struct Fringe : public CFunType<Tree*,OddList<int> > {
   OddList<int> operator()( Tree* t ) const {
      if( t==0 )
         return NIL;
      else if( t->leaf() )
         return cons(t->data,NIL);
      else
         return cat( Fringe()(t->left),
                     curry(Fringe(),t->right) );
   }
};
Fringe fringe;

int main() {
   // code to build tree "t"
   List<int> l = fringe(t);
   l = filter( fcpp::equal(13), l );
   while( !null(l) ) {
      cout << head(l) << endl;
      l = tail(l);
   }
}
```

Figure 11: Tree in FC++

```
merge a@(x:xs) b@(y:ys) =
   if      x < y then x : (merge xs b)
   else if x > y then y : (merge a ys)
   else               x : (merge xs ys)

hamming =
   1 : (merge (merge (map (*2) hamming)
                     (map (*3) hamming))
              (map (*5) hamming) )

main =do putStr "Hamming number: "
         print 2000
         putStr "is "

         print (hamming !! 2000)
```

Figure 12: Hamming in Haskell

we found that ghc performs list concatenation much faster than FC++ does.

4.3 Hamming

The final program computes Hamming numbers. Hamming numbers are all the integers which are products of powers of 2, 3, and 5. An elegant way to compute the (infinite) list of all Hamming numbers is to say that the first number in the list is 1, and that the rest of the list is computed by merging three other lists: twice, three times, and five times the list of Hamming numbers itself. The solution is very easy to express recursively in Haskell; it is given in Figure 12. Notice how the definition of `hamming` refers to `hamming` itself. To construct the same solution in C++, we need to be a little more verbose, but the structure is exactly the same. The FC++ code is shown in Figure 13.

Table 3 shows the relative performance of the programs to print the Nth Hamming number. Again, FC++ outperforms Hugs, this time by a factor of about 10; the times for FC++ and ghc are nearly equal. It is worth noting that Hamming is an excellent test of "caching" in lists (Section 5.1). In an older version of FC++ where caching was not available to lists, the performance grew exponentially. (For example, Hamming(300) took over 30s to compute!)

```
#include <iostream>
#include "prelude.h"
using namespace fcpp;
using std::cout; using std::endl;

struct Merge {
  template <class L, class M>
  struct Sig : public FunType<L,M,
    OddList<typename L::ElementType> > {};

  template <class T>
  OddList<T> operator()( const List<T>& a,
                         const List<T>& b ) const {
    T x = head(a);
    T y = head(b);
    if( x < y )
      return cons( x, curry2(Merge(),tail(a),b));
    else if( x > y )
      return cons( y, curry2(Merge(),a,tail(b)));
    else
      return cons( x,
                curry2(Merge(),tail(a),tail(b)));
  }
} merge;

typedef long long int FOO; // g++ has "long long"

struct Hamming : public CFunType< List<FOO> > {
  List<FOO> operator () () const {
     using fcpp::multiplies;
     static List<FOO> h = Hamming();
     static List<FOO> x =
               curry2(map,multiplies((FOO)2),h);
     static List<FOO> y =
               curry2(map,multiplies((FOO)3),h);
     static List<FOO> z =
               curry2(map,multiplies((FOO)5),h);
     static List<FOO> m1= curry2( merge, x, y );
     static List<FOO> m2= curry2( merge, m1, z );
     return cons( (FOO)1, m2 );
  }
} hamming;

int main() {
   cout << "The "<<NUM<<"th hamming number is: ";
   cout << at( hamming(), NUM ) << endl;
}
```

Figure 13: Hamming in FC++

N	FC++	ghc	Hugs
1000	0.02	0.01	0.17
1500	0.03	0.02	0.24
2000	0.03	0.02	0.34
4000	0.07	0.05	0.68
8000	0.14	0.13	1.42
12000	0.21	0.19	2.21

Table 3: Hamming (all times in seconds)

5 Performance Analysis

The current FC++ implementation is more than an order or magnitude faster than the previous release of the library. In this section, we discuss six major optimizations we have applied to our implementation, quantifying the individual benefits whenever possible. For each optimization, we picked an appropriate benchmark that clearly demonstrates the difference in performance. (The difference for the other programs is typically less dramatic.)

5.1 Caching

The first optimization is caching (memoization) in lazy lists. A lazy list is represented by an unevaluated function, or "thunk". When the value of the list is requested (`head()`, `tail()`, or `null()` is called), the thunk is called in order to produce the value. Rather than re-call the thunk each time the list's value is needed, the thunk should be called only once, and its value remembered. This optimization is imperative for programs like Hamming; without caching, Hamming grows exponentially (rather than linearly).

Caching is implemented as a kind of variant record. Conceptually, a "memoized thunk" or "cache" is

```
class Cache            {
  bool value_is_valid;
  Fun0<Value> function;
  Value value;
public:
  Value val()     {
    if( !value_is_valid )
      { value=function(); value_is_valid=true; }
    return value; }    };
```

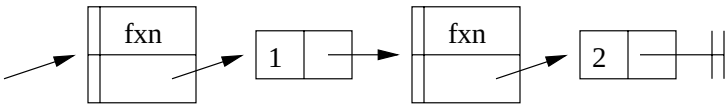
In the actual implementation, we eliminate the space overhead of the boolean variable by using a distinguished `Value` (named `XBAD`) to represent the `!value_is_valid` state.

5.2 Structure of list implementation

When we reimplemented FC++ lazy lists to use caching, we experimented with three different structures for the underlying implementation of lazy lists. We arbitrarily named the three versions TOP, MIDDLE, and BOTTOM (the names reflect the order that we wrote them on a white board). These structures are represented both as skeleton C++ code and pictorially in Figure 14. (To simplify the exposition, the code assumes that lists hold only `int`s (rather than being `template <class T>`s), and also uses raw pointers rather than reference-counted pointers.)

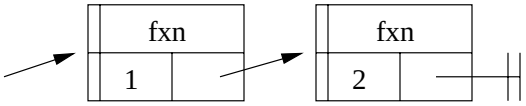
We tested all three list implementations on Primes(1000); the results are shown in Table 4. It should be no surprise that MIDDLE was the winner; MIDDLE contains fewer indirections than the other two solutions. TOP and BOTTOM are both slower due to the extra indirection and poorer locality. Additionally, BOTTOM (and MIDDLE too, actually) suffers another hit because it needs a special `value` to represent the empty list (called `XNIL`, which is like `XBAD` mentioned in Section 5.1), and every evaluation of a list requires an extra test to determine which member of the variant record is active.

The challenge is implementing MIDDLE for `List<T>`s where `T` has no default constructor. C++ requires that constructors be called for all members of an object, but in the case of MIDDLE, when the `value` in the `Cache` isn't valid, we have no constructor to call. As a result, the first field of the `pair` is actually an `unsigned char` array whose size and alignment are appropriate for `T`s. Placement `new` and explicit destructor invocations are used to explicitly manage the lifetime of the `T` created in the raw storage when the `Cache value` becomes valid. It should be noted that the C++ language standard provides no mechanism to ensure that the `unsigned char` array is properly aligned to hold data of type `T`. Nevertheless, there is a relatively portable "hack": creating a union of all kinds of C++ objects (primitive data types,



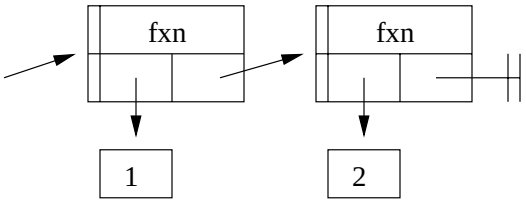
TOP

```
TOP
class List {
   Cache* c;
};
class Cache {
   typedef pair<int,List>* Value;
   Fun0<Value> function;
   Value value;
};
```



MIDDLE

```
MIDDLE
class List {
   Cache* c;
};
class Cache {
   typedef pair<int,List> Value;
   Fun0<Value> function;
   Value value;
};
```



BOTTOM

```
BOTTOM
class List {
   Cache* c;
};
class Cache {
   typedef pair<int*,List> Value;
   Fun0<Value> function;
   Value value;
};
```

Figure 14: Three possible list implementations

Primes(1000)	Time (s)
TOP	12.43
MIDDLE	7.77
BOTTOM	26.36

Table 4: Comparison of different list structures

Hamming(12000) (no functoid reuse)	Time (s)
FC++, non-intrusive (-IRC -REUSE)	0.451
FC++, intrusive (+IRC -REUSE)	0.280

Table 5: The value of intrusive reference counting

structures, pointers, pointers to functions, pointers to members, etc.) ensures that the alignment of the union is wide enough to hold any kind of object on almost any system. Life would be a lot simpler if C++ were extended to have either a mechanism to specify alignments (a system-level solution) or a way to explicitly ask to have a particular structure member's constructor *not* called when the structure is created (a language-level solution); in the meantime, the hack works well enough on most systems. (A system for which the hack does not work can always revert to an alternative implementation of lists, e.g. `TOP`.)

5.3 Intrusive reference counting

The FC++ library contains two reference-counted pointer classes: one that uses an intrusive reference count, and one that is non-intrusive. The two schemes are depicted in Figure 15. The advantage of non-intrusive reference counts is that the object being counted does not need to support any particular interface; it is ignorant of the reference counting. Intrusive reference counts, on the other hand, require that the objects they count supply the counting mechanism. The benefits of intrusive reference counts are increased locality and fewer separate calls to `new`. (For a more thorough introduction to the topic of intrusive reference counts, see [3], Chapter 7.)

We tested Hamming both with and without intrusive reference counts. Since the "reuse functoids" optimization (discussed in the following subsection) requires intrusive reference counts, we turned off that optimization for both of these runs, in order to have a fair comparison. As seen in Table 5, the lack of intrusive reference counts makes Hamming slow down by a factor of about 1.6.

```
struct Take {
  template <class T>
  OddList<T>
  operator()( size_t n, const List<T>& l ) const {
    if( n==0 || null(l) )
      return NIL;
    else
      return cons( head(l),
                   curry2(Take(),n-1,tail(l)) );
  }
} take;
```

Figure 16: take() without functoid reuse

5.4 Reusing functoids during recursive calls

The typical implementation of a functoid which operates on lazy lists contains a curried recursive call as its last line. For example, consider the `Take` functoid shown in Figure 16 (with `Sig` member elided). (Recall that `take` selects the first N elements of a list an discards the rest.) The call to `curry2()` that is passed to `cons()` in the last line of the functoid creates a new object on the heap that represents the recursive call (the "thunk" that makes functoids lazy). The only thing that differs between the newly created functoid and the current functoid itself are the values of `l` and `n`. Instead of discarding the called functoid and creating a similar new functoid, we can recode `take` so that it reuses the functoid. Figure 17 shows the code with this reuse (again, with `Sig` members elided).

We tested Primes both with and without "reuse" versions of `filter()`, `take()`, `at()`, `enumFrom()`, and `enumFromTo()`. The results are shown in Table 6. Clearly, reusing functoids is a big win. When there is no reuse, each call to `take()` has a functoid destruc-

Figure 15: Non-intrusive reference counting (left) and intrusive reference counting (right)

```
struct TakeHelp:public Fun0Impl<OddList<T> > {
  mutable size_t n;
  mutable List<T> l;
  TakeHelp( size_t nn, const List<T>& ll )
    : n(nn), l(ll) {}
  OddList<T> operator()() const {
    if( n==0 || null(l) )
      return NIL;
    else {
      T x = head(l);
      l = tail(l);
      --n;
      return cons(x,Fun0<OddList<T> >(this));
    }
  }
};
struct Take {
  template <class T>
  List<T>
  operator()(size_t n,const List<T>& l) const {
    return Fun0<OddList<T> >
              (new TakeHelp<T>(n,l));
  }
} take;
```

Figure 17: take() with functoid reuse

Primes(1000)	Time (s)
FC++, no functoid reuse (-REUSE)	26.36
FC++, reusing functoids (+REUSE)	7.77

Table 6: The value of reusing functoids

```
struct Take {
   template <class T>
   OddList<T>
   operator()( size_t n, const List<T>& l,
      Reuser2<Inv,Var,Var,Take,size_t,List<T> >
      r = REUSE_INIT ) const {

      if( n==0 || null(l) )
         return NIL;
      else
         return cons( head(l),
                      r( Take(), n-1, tail(l) ) );
   }
} take;
```

Figure 18: take() with reuse via a Reuser

ted, deallocated, and has a new functoid allocated and constructed. With reuse, there is only mutation; no heap allocation/deallocation occurs.

Comparing Figures 16 and 17, one can see that hand-coding a "reuse" version of a functoid takes a bit more code than the non-reuse version. In order to simplify the task of applying this valuable optimization, we have added `Reuser`s to the library. `Reuser`s enable us to capture the essence of functoid reuse with significantly less coding effort. Figure 18 shows `Take` written with a `Reuser`. A `ReuserN` is similar to a call to `curryN()`. The `Reuser` appears as an extra parameter to the functoid. This parameter has a default value (thus making the interface change effectively "invisible" to clients) which is used to create a new thunk on the heap. As a result, the initial call to a functoid that employs a `Reuser` allocates space for a thunk. Subsequent recursive calls are then channelled through the `Reuser` (rather than via

a call to `curry()`); the `Reuser`'s heap thunk, when invoked, explicitly passes itself along to the next call as the extra parameter. This enables reuse of the existing heap thunk. `Reuser`s take template parameters specifying the argument types of the to-be-curried call, as well as extra template parameters that specify whether those parameters are invariant (`Inv`) or variant (`Var`) between calls (knowing this information prevents needless overwriting of duplicate values). Though the internal mechanism is quite complicated, `Reuser`s are relatively easy to apply (compare the code in Figures 16 and 18), and perform nearly as well as the "hand-written" code to perform the optimization (there is only a small "abstraction penalty").

5.5 Avoiding functions with static data

The `Cache` implementation (Figure 14, MIDDLE) uses two distinguished values for its pointer field. The value `XNIL` represents an empty list, and the value `XBAD` represents an "uncached" value (the function is valid, the value is not). These were originally encoded as

```
template <class T> class Cache { ...
   static Ref<Cache<T> >& XNIL() {
      static Ref<Cache<T> > dummy( new Cache );
      return dummy;
   } // XBAD similarly
}
```

However it is far better to say

```
template <class T> class Cache { ...
   static Ref<Cache<T> > XNIL;
}
Ref<Cache<T> > Cache<T>::XNIL( new Cache );
```

In the former, each time `XNIL()` is called, a boolean flag (inserted by the compiler) must be checked (to see if initialization of the static variable has already occurred). In the latter, initialization happens at the start of the program, and `XNIL` is just a value. We tested both versions on Primes; the results are shown in Table 7.

Using global data that calls constructors can be perilous; there are order-of-initialization and order-of-destruction issues for global objects in C++ that

Primes(1000)	Time (s)
FC++, static data in functions (-GL)	11.63
FC++, global data (+GL)	7.77

Table 7: The value of using global data

are often hard to solve. Fortunately, all of these global objects (which sometimes refer to one another) are defined in the same translation unit. This greatly simplifies the issue, and enables us to ensure the correct order of initialization for these objects (section 3.6.2, paragraph 1 of the C++ standard [4], prescribes the order of initialization for such objects [3]). As for order-of-destruction issues, we circumvent the potential problems by artifically incrementing the reference counts of the global objects during initialization. Then, even when the reference-counted pointers are destructed after the end of `main()`, the ref counts do not go to zero, and so the objects to which they refer are left alive; they dangle in the heap until the system collects them when the program exits.

Note also that having `XNIL()` return a reference in the former version is quite important; return by value degrades the performance even more severely. This is because returning a `Ref` object by value does (needless) work, incrementing and decrementing the reference count as the temporary reference lives its short life.

5.6 Using iteration instead of tail recursion

g++ does not transform tail recursion into iteration. As a result, we have done the transformation by hand in library functions like `filter()` and `at()`, and call this the "tail recursion optimization". We ran Primes

[3] It should be noted that a defect report has been filed to the C++ committee, and the proposed resolution of the defect report would invalidate the paragraph cited above. The authors of this paper have suggested an alternative resolution for the issue, which both addresses the defect and preserves initialization order. The matter is still an open issue at the time of this writing. The interested reader should visit http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_active.html#270 and read comp.std.c++ for a thorough discussion of the issue.

Primes(1000)	Time (s)
FC++ with tail recursion (-TRO)	10.69
FC++ with iteration (+TRO)	7.77

Table 8: The value of transforming tail recursion into iteration

Primes(1000)	Time (s)
FC++ (-IRC -REUSE -GL -TRO)	62.05
FC++ (+IRC +REUSE +GL +TRO)	7.77

Table 9: The value of four optimizations combined

both with and without this optimization; the results are shown in Table 8. Transforming tail recursion to iteration has a significant impact on the performance.

5.7 Summary of Optimizations

The results of these optimizations accumulate. We ran Primes both in its optimal configuration, and also with all four of the previous optimizations turned off (intrusive reference counting (IRC), reusing functoids (REUSE), global data (GL), and tail recursion optimization (TRO)). The results are shown in Table 9; note that without any of these optimizations, Primes is eight times slower. Keep in mind also that the unoptimized program still includes the best caching and list implementation; our original naive implementation was even slower.

5.8 Disclaimers and Conclusions

In the previous section, we have compared the performance of C++ programs with Haskell programs. It is important to note that no direct comparison can really be made. All cross-language experiments are fraught with factors that make a direct apples-to-apples comparison impossible, and our experiments are no different. There are many confounding factors, a few of which were mentioned at the beginning of this section. Here we list a handful of obvious differences between FC++ and Haskell which we have not attempted to account for.

- Haskell is lazy (non-strict) throughout, whereas C++ is strict except in FC++ lazy lists, which are explicitly coded to be lazy.

- FC++ manages memory with reference-counted pointers and uses the default allocator provided by our implementation. Haskell uses garbage collection, and a sophisticated allocator designed for optimal performance for a lazy functional language.

- Haskell has more exception-handling by default; for example, taking the `head()` of an empty list raises an exception in Haskell, whereas it simply leads to undefined behavior in FC++ (though we do have a compiler flag that enables exceptions for this kind of misuse).

- GHC Haskell uses a 64-bit representation for all integers, whereas g++ uses a smaller representation for `int`s.

- Haskell has a run-time system which supports a mix of compiled and interpreted code, manages storage allocation, and supports concurrent threads of execution. C++ has no comparable run-time system.

- Many FC++ optimizations must be done "by hand"; the Haskell compiler performs similar optimizations automatically.

By listing these confounding factors, it is not our intention to invalidate the results of the experiments of the previous section. Rather, we simply wish to make explicit the context in which the results must be interpreted. It is meaningless to make general statements like "FC++ is faster than Haskell" or vice-versa. Our goal is merely to demonstrate that FC++ can perform comparably to Haskell on some simple example programs, and to suggest that the two platforms may have roughly comparable run-time performance for programs which make heavy use of lists and lazy evaluation.

6 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++ and analyzed the performance of its lazy lists. More information about FC++ can be found in the references [1][6][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.

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

[3] A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001.

[4] *ISO/IEC 14882: Programming Languages – C++.* ANSI, 1998.

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

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

[7] Y. Smaragdakis and B. McNamara, "FC++: Functional Tools for Object-Oriented Tasks" 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`