Functional Programming with the FC++ Library

BRIAN MCNAMARA and YANNIS SMARAGDAKIS
College of Computing
Georgia Institute of Technology

Abstract

We describe the FC++ library, a rich library supporting functional programming in C++. Prior approaches to encoding higher order functions in C++ have suffered with respect to polymorphic functions from either lack of expressiveness or high complexity. In contrast, FC++ offers full and concise support for higher-order polymorphic functions through a novel use of C++ type inference.

The FC++ library has a number of useful features, including a generalized mechanism to implement currying in C++, a “lazy list” class which enables the creation of “infinite data structures”, a subtype polymorphism facility, and an extensive library of useful functions, including a large part of the Haskell Standard Prelude.

The FC++ library has an efficient implementation. We show the results of a number of experiments which demonstrate the value of optimizations we have implemented. These optimizations have improved the run-time performance by about an order of magnitude for some benchmark programs that make heavy use of FC++ lazy lists. We also make an informal performance comparison with similar programs written in Haskell.

1 Motivation and Overview

It is a little known fact that part of the C++ Standard Library consists of code written in a functional style. Although the C++ Standard Library offers rudimentary support for higher order functions and currying, it stops short of supplying a sophisticated and reusable module for general purpose functional programming. This is the gap that our work aims to fill. The result is a full embedding of a simple pure functional language in C++, using the extensibility capabilities of the language and the existing compiler and run-time infrastructure.

At first glance it may seem that C++ is antithetical to the functional paradigm. The language not only supports direct memory manipulation but also only has primitive capabilities for handling functions. Function pointers are first-class entities, but they are of little use since new functions cannot be created on the fly (e.g. as specializations of existing functions by fixing some state information). Nevertheless, the elements required to implement a functional programming framework are already in the language. The technique of representing first-class functions using classes is well known in the object-oriented world. Among others, the Pizza language (Odersky and Wadler, 1997) uses this approach in translating functionally-flavored constructs to Java code. The same technique is used in previous implementations of higher-order functions in C++ (Kiselyov, 1998; Läufer, 1995). C++ also allows

users to define a familiar syntax for function-classes, by overloading the function application operator, "()". Additionally one can declare methods so that they are prevented from modifying their arguments; this property is enforced statically by C++ compilers. Finally, using the C++ inheritance capabilities and dynamic dispatch mechanism, one can define variables that range over all functions with the same type signature. In this way, a C++ user can "hijack" the underlying language mechanisms to provide a functional programming model.

All of the above techniques are well-known and have been used before. In fact, several researchers in the recent past (Striegnitz, 2001; Kiselyov, 1998; Järvi and Powell, 2002; Läufer, 1995; Meijer and Kettner, 2000) have (re)discovered that C++ can be used for functional programming. Nevertheless, all of the above approaches, as well as that of the C++ Standard Library, suffer from one of two drawbacks:

- *High complexity when polymorphic functions are used*: Polymorphic functions may need to be explicitly turned into monomorphic instances before they can be used. This causes the implementation to become very complex. Läufer observed in (1995): "...the type information required in more complex applications of the framework is likely to get out of hand, especially when higher numbers of arguments are involved."
- *Lack of expressiveness*: In order to represent polymorphic functions, one can use C++ function templates. This approach does not suffer from high complexity of parameterization, because the type parameters do not need to be specified explicitly whenever a polymorphic function is used. Unfortunately, function templates cannot be passed as arguments to other function templates. Thus, using C++ function templates, polymorphic functions cannot take other polymorphic functions as arguments. This is evident in the C++ Standard Library, where "higher order" polymorphic operators like `compose1`, `bind1st`, etc. are not "functions" inside the Standard Library framework and, hence, cannot be passed as arguments to themselves or other operators.

Our work addresses both of the above problems. Contrary to prior belief (see Läufer (1995), who also quotes personal communication with Dami) no modification to the language or the compiler is needed. Instead, we are relying on an innovative use of C++ type inference. Effectively, our framework maintains its own type system, in which polymorphic functions can be specified and other polymorphic functions can recognize them as such.

[Important note: Since C++ type inference is in the core of our technique, a disclaimer is in order: C++ type inference is a unification process matching the types of actual arguments of a function template to the declared polymorphic types (which may contain type variables, whose value is determined by the inference process). C++ type inference does not solve a system of type equations and does not relieve the programmer from the obligation to specify type signatures for functions. Thus, the term "C++ type inference" should not be confused with "type inference" as employed in functional languages like ML or Haskell. The overloading is unfortunate but unavoidable as use of both terms is widespread. We will always use the prefix "C++" when we refer to "C++ type inference".]

The result of our approach is a convenient and powerful parametric polymorphism scheme that is well integrated in the language: with the FC++ library, C++ offers as much support for higher-order polymorphic functions as it does for native types (e.g. integers and pointers).

Apart from the above novelty, FC++ also offers a few more new elements:

- First, we define a subtyping policy for functions of FC++, thus supporting subtype polymorphism. The default policy is what one would expect: a function A is a subtype of function B, iff A and B have the same number of arguments, all arguments of B are subtypes of the corresponding arguments of A, and the return value of A is a subtype of the return value of B. (Using OO typing terminology, we say that our policy is covariant with respect to return types and contravariant with respect to argument types.) Subtype substitutability is guaranteed; a function `Animal* -> Dog*` can be used where a function `Mammal* -> Mammal*` is expected.
- Second, FC++ provides reusable combinators to support automatic currying. Curryable functions can be called with a subset of the arguments they expect, and those values will be bound, resulting in a new function that expects the remainder of the arguments.
- Third, FC++ has a high level of technical maturity. For instance, compared to Läufer's approach, we achieve an equally safe but more efficient implementation of the basic framework for higher order functions. In a previous paper (McNamara and Smaragdakis, 2000a), we illustrated that our implementation was 4 to 8 times faster than Läufer's. In this paper, we describe a number of new optimizations we have recently applied to the library, and demonstrate that the performance has increased by almost an order of magnitude compared to our old implementation.

Additionally, FC++ builds significant functionality on top of the basic framework. We export two fairly mature reference-counting "pointer" classes to library users, so that use of C++ pointers can be completely eliminated at the user level. We define a wealth of useful functions (a large part of the Haskell Standard Prelude) to enhance the usability of FC++ and demonstrate the expressiveness of our framework. It should be noted that defining these functions in a convenient, reusable form is possible exactly because of the support for polymorphic functions offered by FC++. It is no accident that such higher-order library functions are missing from other C++ libraries: supplying explicit types would be tedious and would render the functions virtually unusable.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to the library with some short code examples. Section 3 describes *direct functoids*. (We use the term "functoid" to describe our implementation of functions; the term is borrowed from Läufer (1995).) Direct functoids enable the creation of higher-order polymorphic functions, and are one of the key innovations of FC++. Section 4 describes *indirect functoids*. Indirect functoids are monomorphic, but they are first-class and support subtype polymorphism. Section 5 demonstrates how direct functoids simplify the task of programming with polymorphic functions in

FC++. Section 6 describes two other features of FC++. Section 6.1 describes how *currying* is implemented in FC++. Reusable combinators can be wrapped around any functoid to make it curryable, and we introduce a nice syntax for binding a subset of a functions' arguments. Section 6.2 describes how *subtype polymorphism* is implemented for indirect functoids. Section 7 describes how FC++ interfaces with its host language. Section 8 describes the expressiveness of the library, as well as the fundamental limitations imposed by C++. Section 9 discusses the run-time performance of the library. Section 10 analyzes the performance and discusses a number of optimizations we have applied to the implementation. Section 11 describes some details of the lazy list implementation in FC++. Section 12 gives a summarizing overview of the library and a description of the organization of its modules. Section 13 describes a few applications of the library. Section 14 discusses related work.

2 Library Introduction

In this section we give a brief overview of how the FC++ library is used. Figure 1 will serve as a running example to illustrate the main features of the library.

FC++ lists support the usual list interface; `cons()`, `null()`, `head()`, and `tail()` are among the basic functions that work on `List`s. `List`s are parameterized by the data type they contain; `List`s and the associated functions are polymorphic. Part (A) of Figure 1 illustrates some basic list code.

FC++ has a number of higher-order functions, like `compose()`, which can take polymorphic functions as arguments. Part (B) of Figure 1 illustrates that `compose(tail,tail)` yields a new (polymorphic) function which discards the first two elements of a list. FC++ is the only C++ library that enables the user to generally combine higher-order functions with polymorphic ones; with FC++, polymorphic functions may be passed as arguments to other functions and returned as results.

FC++ `List`s are lazy. Part (C) of Figure 1 demonstrates infinite lists in FC++; the elements of the list are produced only as they are needed.

FC++ functoids support currying. For example, in Part (D) of Figure 1, `plus()` is a two-argument function, but it can be called with just one argument, yielding a new one-argument function as a result. In the example, `map()` applies this new function to each element of the list, yielding a new lists where all of the values have been incremented by 1. As seen in the example, `map()` is also curryable. To bind values to arguments other than the initial arguments, an underscore can be used as a placeholder for arguments that should be curried.

The FC++ library contains more than 50 useful functions from the Haskell Standard Prelude (Peyton-Jones and Hughes, 1999). The prior examples have already used familiar functions like `map()` and `filter()`; FC++ offers dozens of such general functions, including `take()`, which selects the first N elements of a list, and `foldl1()` which left-accumulates all of the values in a list using a given function. Part (E) of Figure 1 demonstrates such functionality.

FC++ has "indirect functoids", run-time variables which can be bound to any function with a given monomorphic signature. Part (F) of Figure 1 illustrates an

```
int x=1, y=2, z=3;
string s="foo", t="bar";

// (A) List basics
List<int> li = cons( x, cons( y, cons( z, NIL )));
List<string> ls = cons( s, cons( t, NIL ));
assert( head(ls) == "foo" );
assert( length(tail(li)) == 1 );

// (B) Higher-order polymorphic compose()
li = compose( tail, tail )(li);
assert( head(li) == 3 );

// (C) Laziness (infinite lists)
li = enumFrom(1);      // [1,2,3,...]
li = filter(even,li); // [2,4,6,...]

// (D) Currying
li = map( plus(1), li );
li = map( plus(1) )( li );
li = map( _, li )( plus(1) );

// (E) Haskell Standard Prelude
li = take( 5, enumFrom(1) );
assert( foldr(plus,3,li) == 18 );
assert( foldl1(plus,ls) == "foobar" );

// (F) Indirect functoids
Fun2<int,int,int> f = monomorphize1<int,int,int>( plus );
assert( f(3,2) == 5 );
f = minus;  // implicit conversion
assert( f(3,2) == 1 );
```

Fig. 1. Some examples of what FC++ can do

indirect functoid variable `f` of type `Fun2<int,int,int>`—a two-argument function which takes two integer arguments and returns an integer result. This variable can be bound to different functions with the right signature, for instance, `plus()` or `minus()`. Since `plus()` is a polymorphic function, a monomorphic instance must be selected to be bound to the indirect functoid variable. This monomorphizing conversion may be done either explicitly or implicitly.

3 Direct Functoids

FC++ represents polymorphic functions with "direct functoids". We will begin by describing the special case of monomorphic direct functoids, because they are simple and serve as a good introduction for readers not familiar with C++. Then we shall move on to describe polymorphic direct functoids. Later, in Section 5, we illustrate how FC++ simplifies the use of polymorphic functions in C++ as compared to other approaches.

3.1 Monomorphic Direct Functoids

C++ is a class-based object-oriented language. Classes are defined statically using the keywords `struct` or `class`. C++ provides a way to overload the function call operator (written as a matching pair of parentheses: "()") for classes. This enables the creation of objects which look and behave like functions (function objects). For instance, we show below the creation and use of function objects that double and add one to a number:

```
struct TwoTimes {
  int operator()( int x ) { return 2*x; }
} twoTimes;

struct Inc {
  int operator()( int x ) { return x+1; }
} inc;

twoTimes(5)   // returns 10
inc(5)        // returns 6
```

The problem with function objects is that their C++ types do not reflect their "function" types. For example, both `twoTimes` and `inc` represent functions from integers to integers. To distinguish from the C++ language type, we say that the *signature* of these objects is

```
int -> int
```

(the usual functional notation is used to represent signatures). As far as the C++ language is concerned, however, the *types* of these objects are `TwoTimes` and `Inc.` (Note our convention of using an upper-case first letter for class names, and a lower-case first letter for class instance names.) Knowing the signature of a function object is valuable for further manipulation (e.g. for enabling parametric polymorphism, as will be discussed in Section 3.2). Thus, we would like to encapsulate some representation of the type signature of `TwoTimes` in its definition. The details of this representation will be filled in Section 3.2, but for now it suffices to say that each direct functoid has a member called `Sig` (e.g. `TwoTimes::Sig`) that represents its type signature. `Sig` is not defined explicitly by the authors of monomorphic direct functoids—instead it is inherited from classes that hide all the details of the type representation. For instance, `TwoTimes` would be defined as:

```
struct TwoTimes : public CFunType<int, int> {
  int operator()( int x ) { return 2*x; }
} twoTimes;
```

That is, `CFunType` is a C++ class template whose only purpose is to define signatures. A class inheriting from `CFunType<A,B>` is a 1-argument monomorphic direct functoid that encodes a function from type `A` to type `B`. In general, the template `CFunType<A1,A2,...,AN,R>` is used to define signatures for monomorphic direct functoids of `N` arguments.

Note that in the above definition of `TwoTimes` we redundantly specify the type signature information (`int -> int`): once in the definition of `operator()` (for compiler use) and once in `CFunType<int,int>` (for use by FC++). There seems to be no way to avoid this duplication with standard C++, but non-standard extensions, like the GNU C++ compiler's `typeof` operator, address this issue.

Monomorphic direct functoids have a number of advantages over normal C++ functions: they can be passed as parameters and returned as results, they can capture state, etc. Native C++ functions can be converted into monomorphic direct functoids using the operator `ptr_to_fun` of FC++. It is worth noting that the C++ Standard Template Library (STL) also represents functions using classes with an `operator()`. FC++ provides conversion operations to promote STL function objects into monomorphic direct functoids.

3.2 Polymorphic Direct Functoids

Polymorphic direct functoids support parametric polymorphism. Consider the Haskell function `tail`, which discards the first element of a list. Its type would be described in Haskell as

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

Here `a` denotes any type; `tail` applied to a list of integers returns a list of integers, for example.

One way to represent a similar function in C++ is through member templates:

```
struct Tail {
  template <class T>
  List<T> operator()( const List<T>& l );
} tail;
```

Note that we still have an `operator()` but it is now a member function template. This means that there are multiple such operators—one for each type. C++ type inference is used to produce concrete instances of `operator()` for every type inferred by a use of the `Tail` functoid. Recall that C++ type inference is a unification process matching the types of actual arguments of a function template to the declared polymorphic types. In this example, the type `List<T>` contains type variable `T`, whose type value is determined as a result of the C++ type inference process. For instance, we can refer to `tail` for both lists of integers and lists of strings, instead of explicitly referring to `tail<int>` or `tail<string>`. For each use of `tail`, the language will infer the type of element stored in the list, based on `tail`'s operand.

As discussed earlier, a major problem with the above idiom is that the C++ type of the function representation does not reflect the function type signature. For instance, we will write the type signature of the tail function as:

```
List<T> -> List<T>
```

but the C++ type of variable `tail` is just `Tail`.

The solution is to define a member, which we call `Sig`, that represents the type signature of the polymorphic function. That is, `Sig` is our way of representing "arrow" types. `Sig` is a template class parameterized by the argument types of the polymorphic function. For example, the actual definition of `Tail` is:

```
struct Tail {
  template <class L>
  struct Sig : public FunType<L,L> {};

  template <class T>
  List<T> operator()( const List<T>& l ) const
  { return l.tail(); }
} tail;
```

where `FunType` is used for convenience, as a reusable mechanism for naming arguments and results.

In reality, the `Sig` member of `Tail`, above, does not have to represent the most specific type signature of function `tail`. Instead it is used as a compile-time function that computes the return type of function `tail`, given its argument type. This is easy to see: the `Sig` for `Tail` just specifies that if `L is` the argument type of `Tail`, then the return type will also be `L`. The requirement that `L` be an instance of the `List` template does not appear in the definition of `Sig` (although it could).

The above definition of `Tail` is an example of a polymorphic direct functoid. In general, a direct functoid is a class with a member `operator()` (possibly a template operator), and a template member class `Sig` that can be used to compute the return type of the functoid given its argument types. Thus the convention is that the `Sig` class template takes the types of the arguments of the `operator()` as template parameters. As described in Section 3.1, for monomorphic direct functoids, the member class `Sig` is hidden inside the `CFunType` classes, but in essence it is just a template computing a constant compile-time function (i.e., returning the same result for each instantiation).

The presence of `Sig` in direct functoids is essential for any sophisticated manipulation of function objects (e.g. most higher-order functoids need it). For example, in Haskell we can compose functions using ".":

```
(tail . tail) [1,2,3]    -- evaluates to [3]
```

In C++ we can similarly define the direct functoid `compose` to act like ".", enabling us to write expressions like

```
compose(tail,tail).
```

The definition of `compose` uses type information from `tail` as captured in its `Sig` structure. Using this information, the type of `compose(tail,tail)` is inferred and does not need to be specified explicitly. More specifically, the result of a composition of two functoids `F` and `G` is a functoid that takes an argument of type `T` and returns a value of type:

```
F::Sig< G::Sig<T>::ResultType >::ResultType
```

that is, the type that `F` would yield if its argument had the type that `G` would yield if its argument had type `T`.[1] This example is typical of the kind of type computation performed at compile-time using the `Sig` members of direct functoids.

In essence, FC++ defines its own type system which is quite independent from C++ types. The `Sig` member of a direct functoid defines a compile-time function computing the functoid's return type from given argument types. The compile-time computations defined by the `Sig` members of direct functoids allow us to perform type inference with fully polymorphic functions without special compiler support. Type errors arise when the `Sig` member of a functoid attempts to perform an illegal manipulation of the `Sig` member of another functoid. All such errors will be detected statically when the compile-time type computation takes place—that is, when the compiler tries to instantiate the polymorphic `operator()`.

Polymorphic direct functoids can be converted into monomorphic ones by specifying a concrete type signature via the operator `monomorphizeN`. For instance:

```
monomorphize1<List<int>, int>( head )
```

produces a monomorphic version of the `head` list operation for integer lists.

In Section 5 we demonstrate how using direct functoids greatly simplifies the task of programming with polymorphic functions in C++, by drawing a comparison with the alternatives. One of the alternatives involves indirect functoids, so we discuss them next.

4 Indirect Functoids

Direct functoids are not first-class entities in the C++ language. Most notably, one cannot define a (run-time) variable ranging over all direct functoids with the same signature. We can overcome this by using a C++ subtype hierarchy with a common root for all functoids with the same signature and declaring the function application operator, "()", to be `virtual` (i.e., dynamically dispatched). In this way, the appropriate code is called based on the run-time type of the functoid to which a variable refers. On the other hand, to enable dynamic dispatch, the user needs to refer to functions indirectly (through pointers). Because memory management (allocation and deallocation) becomes an issue when pointers are used, we encapsulate references to function objects using a reference counting mechanism. This mechanism is completely transparent to users of FC++: from the perspective of the user, function objects can be passed around by value. It is worth noting that our encapsulation of these pointers inside indirect functoids prevents the creation of cyclical data structures,[2] thus avoiding the usual pitfalls of reference-counting garbage collection.

[1] For syntactically correct C++, the type should be written `typename F::template Sig< typename G::template Sig<T>::ResultType >::ResultType`. Throughout the paper, we omit the `typename` and `template` keywords for simplicity of exposition, except in complete program listings.

[2] Actually, it is possible to create cyclic data structures within indirect functoids, but not without a good understanding of the indirect functoid implementation details that are necessary to circumvent the encapsulation. Users cannot leak memory "by accident".

Indirect functoids are classes that follow the above design. An indirect functoid representing a function with N arguments of types A1, ..., AN and return type R, is a subtype of class

```
FunN<A1,A2...,AN,R>.
```

For instance, one-argument indirect functoids with signature

```
A -> R
```

are subtypes of class `Fun1<A,R>`. This class is the reference-counting wrapper of class `Fun1Impl<A,R>`. Both classes are produced by instantiating the templates shown below:

```
template <class Arg1, class Result>
class Fun1 : public CFunType<Arg1,Result> {
  Ref<Fun1Impl<Arg1,Result> > ref;
  ...
public:
  typedef Fun1Impl<Arg1,Result>* Impl;
  Fun1( Impl i ) : ref(i) {}
  Result operator()( const Arg1& x ) const
    { return ref->operator()(x); }
  ...
};

template <class Arg1, class Result>
struct Fun1Impl : public CFunType<Arg1,Result> {
  virtual Result operator()( const Arg1& ) const =0;
  virtual ~Fun1Impl() {}
};
```

(Note: The ellipsis (...) symbol in the above code is used to denote that parts of the implementation have been omitted for brevity. These parts implement our subtype polymorphism policy and will be discussed in Section 6.2. The `Ref` class template implements our reference-counted "pointers" and will be discussed in Section 10.3. For this internal use, any simple reference counting mechanism would be sufficient.)

Concrete indirect functoids can be defined by subclassing a class `Fun1Impl<A,R>` and using instances of the subclass to construct instances of class `Fun1<A,R>`. Variables can be defined to range over all functions with signature

```
A -> R.
```

For instance, if `Inc` is defined as a subclass of `Fun1Impl<int,int>`, the following defines an indirect functoid variable `f` and initializes it to an instance of `Inc`:

```
Fun1<int, int> f (new Inc);
```

In practice, however, this definition would be rare because it would require that `Inc` be defined as a monomorphic function. As we have seen in Section 3.2, the most

common and convenient representation of functions is that of polymorphic direct functoids.

Monomorphic direct functoids can be explicitly converted to indirect functoids, using the operation `makeFunN` (provided by FC++). For instance, consider direct functoids `TwoTimes` and `Inc` from Section 3.1 (the definition of `Inc` was not shown). The following example is illustrative:

```
Fun1<int,int> f = makeFun1( twoTimes );
f( 5 );                        // returns 10
f = makeFun1( inc );
f( 5 );                        // returns 6
```

In fact, the calls to `makeFunN` can be elided—we show them here to help explain the transformation, however a carefully designed implicit conversion function template in the `FunN` classes makes the library functoids "smart" enough to let the transformation happen implicitly. Polymorphic direct functoids can also be assigned to indirect functoids, by first selecting a monomorphic instance. This conversion was illustrated back in Figure 1 in Section 2 as

```
Fun2<int,int,int> f = monomorphize1<int,int,int>( plus );
f = minus;  // implicit conversion
```

Note that just like `makeFunN`, the call to `monomorphize` can be elided.

It should be noted here that our indirect functoids are very similar to the functoids presented in Läufer's work (1995) and the functors presented in Chapter 5 of Alexandrescu's book (2001). Indeed, the only difference is in the wrapper classes, `FunN<A1,A2,...,AN,R>`. Whereas we use a reference counting mechanism, both Läufer's and Alexandrescu's implementations allowed no aliasing: different instances of `FunN<A1,A2...,AN,R>` had to refer to different instances of `FunNImpl<A1,A2,...,AN,R>`. To maintain this property, objects had to be copied every time they were about to be aliased. This copying results in an implementation that is significantly slower than ours—in a previous paper (McNamara and Smaragdakis, 2000a), we demonstrated that our implementation was four to eight times faster than Läufer's. (Our new implementation, which uses intrusive reference counting for indirect functoids, is even faster, as we shall see in Section 10.) Another difference from other implementations is that our indirect functoids will rarely be defined explicitly by clients of FC++. Instead, they will commonly only be produced by fixing the type signature of a direct functoid.

5 Use of Direct Functoids

In this section we will demonstrate the use of FC++ direct functoids and try to show how much they simplify programming with polymorphic functions. The comparison will be to the two alternatives: templatized indirect functoids, and C++ function templates.

Consider a polymorphic function `twoTimes` that returns twice the value of its numeric argument. Its type signature would be

```
// N: Number type
template <class N>
struct TwoTimes : public FunImpl<N, N> {
  N operator()( const N& n ) const
  { return 2*n; }
};

// E: element type in original list
// R: element type in returned list
template <class E, class R>
struct Map : public
  FunImpl<Fun1<E,R>, List<E>, List<R> > {
  List<R>
  operator()( const Fun1<E,R>& f, const List<E>& l ) const {...}
};
```

Fig. 2. Polymorphic functions as templates over indirect functoids

```
a -> a.
```

(In Haskell one would say

```
Num a => a -> a.
```

It is possible to specify this type bound in C++, albeit in a roundabout way—see the short discussion on type constraints in Section 8 for details.)

Consider also the familiar higher-order polymorphic function `map`, which applies its first argument (a unary function) to each element of its second argument (a list) and returns a new list of the results. One can specify both `twoTimes` and `map` as collections of indirect functoids. Doing so generically would mean defining a C++ template over indirect functoids. This is equivalent to the standard way of imitating polymorphism in Läufer's framework. Figure 2 shows the implementations of `map` and `twoTimes` using indirect functoids. (For brevity, the implementation of `operator()` in `Map` is omitted. The implementation is similar in all the alternatives we will examine.)

Alternatively, one can specify both `twoTimes` and `map` using direct functoids (Figure 3). Direct functoids can be converted to indirect functoids for a fixed type signature, hence there is no loss of expressiveness.

The direct functoid implementation is only a little more complex than the indirect functoid implementation. The complexity is due to the definition of `Sig`. `Sig` encodes the type signature of the direct functoid in a form that can be utilized by all other higher order functions in our framework. According to the convention of our framework, `Sig` has to be a class template over the types of the arguments of `Map`. Recall also that `FunType` is just a simple template for creating function signatures.

To express the (polymorphic) type signature of `Map`, we need to recover types from the `Sig` structures of its function argument and its list argument. The type computation `F::Sig<L::EleType>::ResultType` means "result type of function `F`, when its argument type is the element type of list `L`".

```
struct TwoTimes {
  template <class N> struct Sig : public Fun1Type<N,N> {};
  template <class N>
  N operator()( const N& n ) const { return 2*n; }
} twoTimes;

// F: function type
// L: list type
struct Map {
  template <class F, class L>
  struct Sig : public Fun2Type<F,L,List<F::Sig<L::EleType>::ResultType> >{};

  template <class F, class L>
  typename Sig<F,L>::ResultType
  operator()( const F& f, const L& l ) const {...}
} map;
```

Fig. 3. Polymorphic functions as direct functoids

In essence, using `Sig` we export type information from a functoid so that it can be used by other functoids. Recall that the `Sig` members are really compile-time functions: they are used as type computers by the FC++ type system. The computation performed at compile time using all the `Sig` members of direct functoids is essentially the same type computation that a conventional type inference mechanism in a functional language would perform. Of course, there is potential for an incorrect signature specification of a polymorphic function but the same is true in the indirect functoid solution.

To see why the direct functoid specification is beneficial, consider the uses of `map` and `twoTimes`. In Haskell, we can say

```
map twoTimes [1..]
```

to produce a list of even numbers. With direct functoids (Figure 3) we can similarly say

```
map( twoTimes, enumFrom(1) ).
```

This succinctness is a direct consequence of using C++ type inference. With the indirect functoid solution (Figure 2) the code would be much more complex, because all intermediate values would need to be explicitly typed as in

```
Map<int,int>()( Fun1<int,int>( new TwoTimes<int> ), enumFrom(1) ).
```

Clearly this alternative would have made every expression terribly burdensome, introducing much redundancy (`int` appears 5 times in the previous example, when it could be inferred everywhere from the value `1`). Note that this expression has a single function application. Using more complex expressions or higher-order functions makes matters even worse. For instance, using the `compose` functoid mentioned in Section 3.2, we can create a list of multiples of four by writing

```
map( compose( twoTimes, twoTimes ), enumFrom(1) ).
```

The same using indirect functoids would be written as

```
Fun1<int,int> twoTimes( new TwoTimes<int> );
Map<int,int>()( Compose<int,int,int>()(twoTimes, twoTimes),
                enumFrom(1) ).
```

We have found even the simplest realistic examples to be very tedious to encode using templates over indirect functoids (or, equivalently, Läufer's framework (1995)).

In short, direct functoids allow us to simplify the *use* of polymorphic functions substantially, with only little extra complexity in the functoid *definition*. The idiom of using template member functions coordinated with the nested template class `Sig` to maintain our own type system is the linchpin in our framework for supporting higher-order parametrically polymorphic functions.

Finally, note that `twoTimes` could have been implemented as a C++ function template:

```
template <class N>
N twoTimes( const N& n ) { return 2*n; }
```

This is the most widespread C++ idiom for approximating polymorphic functions (e.g. (Meijer and Kettner, 2000; Stepanov and Lee, 1995)). C++ type inference is still used in this case. Unfortunately, as noted earlier, C++ function templates cannot be passed as arguments to other functions (or function templates). That is, function templates can be used to express polymorphic functions but these cannot take other function templates as arguments. Thus, this idiom is not expressive enough. For instance, our example where `twoTimes` is passed as an argument to `map` is not realizable if `twoTimes` is implemented as a function template.

The closest approximation of such functionality before FC++ was with the use of a hybrid of class templates (Stepanov and Lee, 1995), like in Figure 2, and function templates. In the hybrid case, each function has two representations: one using a template class (so that the function can be passed to other functions) and one using a function template (so that C++ type inference can be used when arguments are passed to the function). The C++ Standard Library (Stepanov and Lee, 1995) uses this hybrid approach for some polymorphic, higher-order functions. This alternative is quite inconvenient because class templates still need to be turned into monomorphic function instances explicitly (e.g. one would write `TwoTimes<int>` instead of `twoTimes` in the examples above), and because two separate representations need to be maintained for each function. The user will have to remember which representation to use when the function is called and which to use when the function is passed as an argument.

What all of the above alternatives to direct functoids lack is the ability to express polymorphic functions that can accept other polymorphic functions as arguments. As an example, consider a function `foo(f,x,y)` whose body is just

```
return makePair( f(x), f(y) );
```

(`makePair` is the function to create a 2-tuple of values). With `twoTimes` defined as a polymorphic direct functoid, we can write the expression

```
foo( twoTimes, 2, 3.1 )
```

which will resolve to a value of type `pair<int,double>`. This is rank-2 polymorphism (Kfoury and Tiuryn, 1992); inside the call to `foo()`, `f` is used polymorphically. Neither of the other approaches enable functions like `foo()` to be defined. That is, this higher-rank polymorphism capability of FC++ direct functoids is unique among C++ libraries for functional programming.

6 Other Features

In this section we describe two other features of the FC++ library: *currying* and *subtype polymorphism.*

6.1 Currying

FC++ supports currying of functoid arguments. Currying is implemented in the `CurryableN` combinators. The fact that these are combinators make it easy to transform any functoid into a curryable version of that functoid. Indeed, all of the functoids exported by the library are curryable.[3] Here is an example:

```
struct Plus { ... } xplus;
Curryable2<Plus> plus(xplus);
...
xplus(2,3);   // xplus requires both args,
plus(2,3);    // whereas plus is curryable
plus(2);      // and can be called in any
plus(2,_);    // of these ways.
plus(_,3);
```

In the example, `xplus` is defined as a normal direct functoid, and `plus` is an object that adds the currying functionality to the underlying functoid. The underscore (_) is a special value (the unique instance of a type named `AutoCurryType`) that curryable functoids know about which serves as a placeholder meaning “this argument will be supplied later”.

The `CurryableN` classes, which implement the currying functionality, take advantage of two C++ language features: overloading and partial specialization. In C++, functions can be overloaded (ad hoc) based on the number and types of their arguments. Similarly, templates can be specialized (ad hoc) for certain argument types. So class `Curryable2` has four separate `Sig` specializations (`Sig<X,Y>`, `Sig<X>`, `Sig<X,AutoCurryType>`, `Sig<AutoCurryType,Y>`—where `X` and `Y` are template parameters (free type variables)) and four separate overloaded `operator()`

[3] While all our functoids are curry*able*, curry*ing* only happens when a subset of a function’s arguments are passed. When all of the expected arguments are passed to a curryable functoid, the `Curryable` wrapper class transparently “forwards” the call to its underlying functoid. An optimizing C++ compiler can eliminate the overhead of the forwarding function, so there is no penalty to adding the currying capability to every functoid.

implementations, which correspond to the four different ways a two-argument functoid may be called:

```
f(x,y)
f(x)
f(x,_)
f(_,y)
```

The `CurryableN` classes enable functions to be curried either implicitly (by only supplying some of the leading arguments) or semi-explicitly (using underscores as placeholders for arguments to be curried). The FC++ library also contains fully explicit functions for currying, named `bindMofN`. For example `plus(_,3)` and `bind2of2(plus,3)` are equivalent: both bind the second argument (of `plus`'s two arguments) to the value 3, and return a new function of one argument. The explicit binders have an additional capability: if one binds all of a function's arguments, a zero-argument functoid (a thunk) is returned. For example, `bind1and2of2(plus,2,3)` returns a zero-argument functoid, which yields the value 5 when called. The `curryN` functions can also be used to enact the same behavior: `curry2(plus,2,3)` yields the same result as `bind1and2of2(plus,2,3)`.

There are typically a few different ways to express the same "curried function call" expression in FC++. The redundancy is a historical accident; the explicit binders (`bindMofN`) were created first, whereas the `curryN` functions and `CurryableN` classes came later (after we discovered the template specialization tricks needed to enable such functionality). Nowadays, we typically prefer to use the implicit currying of the `Curryable`s to curry any subset of a function's arguments, and use the `curryN` functions when we want to bind all of the arguments and create a thunk. The explicit binders are retained for compatibility with legacy code, and because they are conceptually easier for novice users to understand.

Currying is a useful feature in functional programming, as it enables programmers to easily specialize and adapt general functions to fit specific needs, by fixing some subset of the functions' arguments. Whereas some functional languages have built-in support for currying, in C++, it must be supplied via a library. FC++ is the only C++ library which supports implicit currying, by exploiting the existing features of the C++ language to make it appear to clients as though currying is a built-in language feature that works automatically. Other C++ libraries have other levels of support for currying. For example, the Lambda Library (Järvi and Powell, 2002) has a mechanism similar to FC++ semi-implicit currying, where explicit placeholders can be used for curried arguments. Both the STL (Stepanov and Lee, 1995) and Alexandrescu's "functors" (2001) do support "binding" one of the arguments of a function to a specific value, however in each of these libraries, the binding must be done explicitly (with a call to a binding function like FC++'s `bindMofN`), and the binding only works on monomorphic functions—a severe limitation.

6.2 Subtype Polymorphism

Another innovation of our framework is that it implements a policy of subtype polymorphism for indirect functoids. Our policy is contravariant with respect to argument types and covariant with respect to result types. Subtype polymorphism is important because it is a familiar concept in object orientation—it ensures that indirect functoids can be used like any other C++ object reference in real C++ programs.

A contrived example: Suppose we have two type hierarchies, where Dog is a subtype of Animal and Car is a subtype of Vehicle. This means that a Dog is an Animal (i.e., a reference to Dog can be used where a reference to Animal is expected) and a Car is a Vehicle. If we define a functoid which takes an Animal as a parameter and returns a Car, then this functoid is a subtype of one that takes a Dog and returns a Vehicle. For instance:

```
Fun1<Ref<Animal>, Ref<Car> >  fa;
Fun1<Ref<Dog>, Ref<Vehicle> > fb = fa; //ok: fa is a subtype of fb
```

(Note the use of our `Ref` class template which implements references—a general-purpose replacement of C++ pointers. The example would work identically with native C++ pointers—e.g. `Car*` rather than `Ref<Car>`.)

That is, `fa` is a subtype of `fb` since the argument of `fb` is a subtype of the argument of `fa` (contravariance) and the return type of `fa` is a subtype of the return type of `fb` (covariance). We cannot go the other way, though (assign `fb` to `fa`). This means that we can substitute a "specific" functoid in the place of a "general" functoid. Since subtyping only matters for variables ranging over functions, it is implemented only for indirect functoids.

Subtype polymorphism is implemented by defining an implicit conversion operator between functoids that satisfy our subtyping policy. This affects the implementation of class templates `FunN` of Section 4. For instance, the definition of `Fun1` has the form:

```
template <class Arg1, class Result>
class Fun1 : public CFunType<Arg1,Result> {
  ... // private members same as before
public:
  ... // same as before
  template <class A1s,class Rs>
  Fun1(const Fun1<A1s,Rs>& f) : ref(convert1<Arg1,Result>(f.ref)){}
};
```

Without getting into all the details of the implementation, the key idea is to define a template implicit conversion operator from `Fun1<A1s,Rs>` to `Fun1<Arg1,Result>`, if and only if `A1s` is a supertype of `Arg1` and `Rs` is a subtype of `Result`. The latter check is the responsibility of direct functoid `convert1` (not shown). In particular, `convert1` defines code that will explicitly test (at compile time) to ensure that an `Arg1` is a subtype of `A1s` and that `Rs` is a subtype of `Result`. In this way, the implicit

conversion of functoids will fail if and only if either of the above two conversions fails. Since the operator is templatized, it can be used for any types `A1s` and `Rs`.

We should note that, although the above technique is correct and sufficient for the majority of conversions, there are some slight problems. First, C++ has inherited from C some unsafe conversions between native types (e.g. implicit conversions from floating point numbers to integers or characters are legal). There is no good way to address this problem (which was inherited from C despite the intentions of the C++ language designer; see (Stroustrup, 1996) p. 710). Second, we cannot overload (or otherwise extend) the C++ operator `dynamic_cast`. Instead, we have provided our own operation that imitates `dynamic_cast` for indirect functoids. The incompatibility is unfortunate, but should hardly matter for actual use: not only do we provide an alternative, but also down-casting functoid references does not seem to be meaningful, except in truly contrived examples. More details on our implementation of subtype polymorphism can be found in the documentation of FC++ (Smaragdakis and McNamara, 2002).

7 Embedding interface

In this section we discuss how FC++ interfaces with the rest of the C++ language and with C++ libraries, as well as how FC++ can capture "effects".

FC++ has interfaces to normal C++ functions and the C++ Standard Library. We have already encountered `ptr_to_fun()`, which converts a normal function into an FC++ functoid. The `ptr_to_fun()` operator works on member functions as well, creating a functoid which takes a pointer to the receiver object as an extra first parameter. Figure 4 shows `ptr_to_fun()` applied to both normal and member functions, and demonstrates that the results are functoids by using the currying ability of FC++ functoids. Note also that `ptr_to_fun()` may be applied to both `const` and non-`const` member functions. Creating a functoid from a non-`const` member function results in a functoid which can have an effect. This is possible since the functoid takes a *pointer* to the receiver object. Indeed, this is the usual way to capture effects inside functoids: whereas the parameters and results of the functoids are `const` as a result of the FC++ library's design, there is nothing to stop a client from passing a (`const`) *pointer* to a non-`const` object into a functoid, which may then manipulate the object via the pointer.

FC++ functoids are designed to work smoothly with the C++ Standard Template Library (STL). Monomorphic FC++ functoids conform to the requirements for what the STL calls "adaptable functions", which enables FC++ functoids to be passed to STL algorithms like `std::transform()` (the imperative analog of `map()`). (Polymorphic functoids can be suitably adapted simply by first `monomorphize()`ing them.) Indeed, when using STL algorithms, it is often easier to use FC++ functoids rather than use the STL's own support. For example, to add `3` to each element of a `std::vector<int>` named `v`, one must write

```
std::transform(v.begin(),v.end(),v.begin(),
  std::bind1st(std::plus<int>(),3) );
```

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

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

void example() {
  assert( ptr_to_fun(&f)(3)(1) == 10 );

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

  ptr_to_fun(&Foo::inc_n)(&foo,1); // effect
  assert( foo.n() == 4 ); // updated value
}
```

Fig. 4. FC++ and native C++ functions

```
  List<int> l = take( 5, enumFrom(1) );
  // Make a vector from a List
  std::vector<int> v( l.begin(), l.end() );
  std::reverse( v.begin(), v.end() );
  // Make a List from a vector
  List<int> r( v.begin(), v.end() );
  assert( r == list_with(5,4,3,2,1) );
```

Fig. 5. FC++ and STL

using STL, whereas when FC++ is brought to bear, just

```
std::transform(v.begin(),v.end(),v.begin(), fcpp::plus(3) );
```

is sufficient. (Note the explicit namespace qualification of identifiers—FC++ defines namespace `fcpp` for its symbols.) On the other side of the coin, FC++ provides combinators to promote STL "adaptable functions" into (monomorphic) FC++ functoids so that functions from STL can be used inside FC++. Finally, FC++ `List`s are designed to fit into the STL framework for data structures. Figure 5 shows that the `List` class supports iterators of the STL style. This makes converting both to and from STL data structures easy, and enables `List`s to be passed to (non-mutating) STL algorithms.

Another interface that is somewhat common in legacy C/C++ code is the use of types like `void (*)(void*)`[4] as a sort of generic interface for "callback functions". It is not possible to automatically convert an FC++ functoid into such a function

[4] That is, a pointer to a function which takes as an argument a pointer to an arbitrary data structure.

pointer, but it is straightforward to hand-code an adapter function: just write a normal C++ function with the proper signature that forwards the call to the appropriate functoid. In this way, functoids can be used with such legacy libraries. (On the other hand, if callbacks are desired but there is no need to interface with a legacy callback library, then FC++ itself can serve as a complete callback library, with indirect functoids serving as type-safe interfaces to arbitrary functions. See reference (Smaragdakis and McNamara, to appear) as well as examples on our web site (Smaragdakis and McNamara, 2002) for more about using FC++ as a callback library.)

In summary, the interfaces between FC++ and STL and also between FC++ and the object-oriented portions of C++ are smooth. FC++ makes it straightforward to utilize the extra functional support on top of the existing imperative/object-oriented programming platform that C++ provides (Smaragdakis and McNamara, to appear).

8 Expressiveness

At this point we can summarize the level of support for functional programming that FC++ offers, as well as its limitations.

- *Complexity of type signature specifications:* FC++ allows higher-order polymorphic function types to be expressed and used. Type signatures are explicitly declared in our framework, unlike in ML or Haskell, where types can be inferred. Furthermore, our language for specifying type computations (i.e., our building blocks for `Sig` template classes) is a little awkward. We used our framework to define a large number (over 50) of common functional operators and have not found our type language to be a problem—learning to use it required only minimal effort.
  The real advantage of FC++ is that, although function *definitions* need to be explicitly typed, function *uses* do not (even for polymorphic functions). In short, with our framework, C++ has as good support for higher-order and polymorphic functions as it does for any other first-class C++ type.
- *Polymorphic variables:* While FC++ has a great deal of support for polymorphic functions, we still cannot create run-time variables with polymorphic types, because these types cannot be expressed directly in C++. For example, even though `tail` and `init` (the dual of `tail`, which discards the *last* element of a list) both have the signature

  ```
  [a] -> [a]
  ```

  we cannot create a variable "f" which can be bound to both functions during the course of its lifetime

  ```
  f = tail;
  ...
  f = init;
  ```

because we have no C++ type to declare "f" to be an instance of. Similarly, we cannot create a `List` which contains both `tail` and `init`, as these two objects have different C++ types (namely `Tail` and `Init`) and therefore cannot be put into the same (homogeneously-typed) list. This limitation is fundamental, common to all approaches to functional programming in C++.

- *Limitations in the number of functoid arguments:* There is a bound in the number of arguments that our functoids can support. This bound can be made arbitrarily high (templates with more parameters can be added to the framework) but it will always be finite. This has not proven to be a significant problem in practice.

  A closely related issue is that of naming. We saw base classes like `Fun1` and `Fun1Impl` in FC++, as well as operators like `makeFun1` and `monomorphize1`. These entities encode in their names the number of arguments of the functions they manipulate. Using C++ template specialization, this can be avoided, at least in the case of class templates. Thus, we can have templates `Fun` and `FunImpl` with a variable number of arguments. If template `Fun` is used with two arguments, then it is assumed to refer to a one-argument function (the second template argument is the return type). We experimented with this idea, and elected to use it only in the `CFunType` and `FunType` classes (which help implement `Sig` type signatures in class definitions). In client code, where indirect functoid variables are declared and used, the redundant `N` in the `FunN` names seems valuable to the human reader.
- *Automatic currying:* all of the library functoids support automatic currying via the `CurryableN` combinators. This enables a functoid to be called with fewer arguments than it expects, resulting in a new functoid which expects the remainder of the arguments. It is also possible to enable functoids to accept *more* arguments than they expect. For example, imagine a one-argument function named `foo` which returns another one-argument function. We could imagine writing `foo(x,y)` to mean the same thing as `foo(x)(y)`. FC++ only supports the latter form; while it is possible to support the former as well, we have rarely come across cases where this form of "uncurrying" is desirable.
- *Compiler error messages:* C++ compilers are notoriously verbose when it comes to errors in template code. Indeed, our experience is that when a user of FC++ makes a type error, the compiler typically reports the full template instantiation stack, resulting in many lines of error messages. In some cases this information is useful, but in others it is not. We can distinguish two kinds of type errors: errors in the `Sig` definition of a new functoid and errors in the use of functoids. Both kinds of errors are usually diagnosed well and reported as "wrong number of parameters", "type mismatch in the set of parameters", etc. In the case of `Sig` errors, however, inspection of the template instantiation stack is necessary to pinpoint the location of the problem. Fortunately, the casual user of the library is likely to only encounter errors in the *use* of functoids.

  Reporting of type errors is further hindered by non-local instantiations of FC++ functoids. Polymorphic functoids can be passed around in contexts

that do not make sense, but the error will not be discovered until their subsequent invocation. In that case, it is not immediately clear whether the problem is in the final invocation site or the point where the polymorphic functoid was passed as a parameter. Fundamentally, this problem cannot be addressed without type constraints in template instantiations, something that C++ does not offer[5]. Overall, however, type error reporting in FC++ is adequate, and, with some experience, users have little difficulty with it.

- *Creating closures:* Our functoid objects correspond to the functional notion of closures: they can encapsulate state together with an operation on that state. Note, however, that, unlike in functional languages, "closing" the state is not automatic in our framework. Instead, the state values have to be explicitly passed during construction of the functoid object. Of course, this is a limitation in every approach to functional programming in C++.

  The reader may have noticed our claim in Section 4 that our (internal) reference-counted functoid pointers cannot form cycles. This implies that our closures (i.e., functoid objects) cannot be self-referential. Indeed, this is a limitation in FC++, albeit not an important one: since our closures cannot be anonymous, and since the "closing" of state is explicit, it is convenient to replace self-referential closures with closures that simply create a copy of themselves. This approach is slightly inefficient, but the efficiency gains of using a fast reference counting technique for functoid objects far outweigh this small cost. (In fact, as we shall see in Section 10.4, we can even eliminate the need to make copies by using `Reuser`s, thereby also eliminating this final small cost.)

- *Pure functional code vs. code with side-effects:* In C++, any method is allowed to make system calls (e.g. to perform I/O, access a random number generator, etc.) or to change the state of global variables. Thus, there is no way to fully prevent side-effects in user code. Nevertheless, by declaring a method to be `const`, we can prevent it from modifying the state of the enclosing object (this property is enforced by the compiler). This is the kind of "side-effect freedom" that we try to enforce in FC++. Our indirect functoids (as shown in Section 4) are explicitly side-effect free—any class inheriting from our `FunNImpl` classes has to have a `const operator()`. Nevertheless, users of the library could decide to add other methods with side-effects to a subclass of `FunNImpl`. We strongly discourage this practice but cannot prevent it. It is a good convention to always declare methods of indirect functoids to be `const`.

  For direct functoids, guarantees are even weaker. We cannot even ensure that `operator()` will be `const`, although this is, again, a good practice. While functoids with side effects can be implemented in our framework (as described in Section 7), such functoids should be used with care. Other opportunities

[5] Such type constraints are known in the C++ community as "concept checks". There are some C++ "concept checking libraries" (McNamara and Smaragdakis, 2000b; Siek and Lumsdaine, 2000) which cleverly exploit the language to check some of the constraints and coerce the compiler into emitting more readable diagnostic messages. We have not so far applied these techniques to the particular constraints which would benefit FC++ functoids, though.

for code with side effects abound in C++. Our recommendation is that most code with side effects should be implemented outside the FC++ framework. For instance, such code could be expressed through native C++ functions. The purist can even define monads (Wadler, 1990) using FC++; we have implemented a few example monads, which are available (along with dozens of other example files) on the FC++ web site (Smaragdakis and McNamara, 2002).

9 Performance

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, tail-recursion elimination is not automatically performed. Furthermore, FC++ does not even have a binary (platform-specific) runtime system—FC++ is entirely a language-level library, thus gaining in simplicity and portability: FC++ is as portable as standard C++. As a result, FC++ has no special runtime support for specialized functions or lazy evaluation. Similarly, FC++ offers a language-level 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 the following section (Section 10), we will describe the optimizations in detail.

In this section we show some simple performance measurements comparing FC++ to Hugs (a well-known Haskell interpreter (Jones and Reid, 2002)) and ghc (an optimizing Haskell compiler (Peyton-Jones, Marlow, and Seward, 2002)). 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, in that 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 the case of both g++ and ghc, we used -O2 and static linking.

The next three subsections illustrate our benchmark programs and the performance results. The final subsection in this section notes the many caveats of a cross-language performance comparison, and draws only a very basic conclusion from our data.

```
module Main where

   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)
```

Fig. 6. Primes in Haskell

9.1 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 6 shows the code for primes in Haskell. Figure 7 shows the code for primes in FC++. Note that this is the first complete C++ program presented in this paper. It includes a `main` routine, as well as all the necessary `#include` and namespace statements. Also, throughout this section we use the datatype `OddList` for FC++ lazy lists for performance reasons. We explain the slight difference between FC++ `Lists` and `OddLists` in Section 11.

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. While Haskell uses the arbitrary precision type `Integer` by default, explicitly requesting 32-bit `Ints` had no measurable effect on the ghc-compiled program's performance. On the other hand, using `Ints` did speed up the Hugs times by about 15% for each run (the numbers in the table for Hugs are without the speedup).

9.2 Tree

Tree is a program that generates a random binary search tree of integers and then (lazily) computes the "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 8 shows the Haskell code for Tree; Figure 9 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

```
#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;
}
```

Fig. 7. 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)*

```
module Main where

   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))
```

Fig. 8. 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)*

Hugs, but about three times slower than ghc. Investigating the disparity between the FC++ and ghc performance, we found that ghc performs lazy list concatenation much faster than FC++ does. We plan to search further for a generally applicable optimization that will speed up list concatenation. Note that for Tree, using `Int`s instead of `Integer`s had no measurable effect on the times for either ghc or Hugs.

9.3 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

```
#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;

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);
   }
}
```

Fig. 9. Tree in FC++

is very easy to express recursively in Haskell; it is given in Figure 10. 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 11.

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. For this program, we could not use the 32-bit `Int` in place of `Integer`, as `Int` is not wide enough—our C++ Hamming code needs the g++-specific `long long int` (64 bits) to handle the large numbers involved in this example.

```
module Main where

   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)
```

Fig. 10. Hamming in Haskell

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)*

9.4 Disclaimers and Conclusions

In this 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.

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

```
#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 {
     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;
}
```

Fig. 11. Hamming in FC++

- *Memory management.* FC++ manages memory with reference-counted pointers and uses the default allocator provided by the C++ implementation. Haskell uses garbage collection, and a sophisticated allocator designed for optimal performance for a lazy functional language.[6]

[6] For reference, we have also experimented with the Boehm-Demers-Weiser conservative garbage collector for C/C++ (Boehm and Demers, 2002) but did not perform a comprehensive test with memory-intensive programs where locality would matter. Hence, our experience mostly shows

- *Exception handling.* 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++.
- *Runtime.* Haskell has a run-time system which supports a mix of compiled and interpreted code and supports concurrent threads of execution. C++ has no comparable run-time system.
- *Optimizations.* 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 this 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, even for benchmarks which make heavy use of lists and lazy evaluation, FC++ can perform roughly comparably to an optimized functional implementation.

10 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.) At the end of the section, we also repeat an experiment from an earlier paper (McNamara and Smaragdakis, 2000a), comparing the performance of FC++ with Läufer's library.

10.1 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). In an older version of FC++ where caching was not available to lists, Hamming(300) took over 30 seconds to compute!

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

```
class Cache {
  bool value_is_valid;
```

the constant overheads of the two methods: the conservative GC was more than twice as slow as non-intrusive reference counting.

```
    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.

10.2 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 12. (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 10.1), 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 very portable "hack": creating a union of all kinds of C++ objects (primitive data types, 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

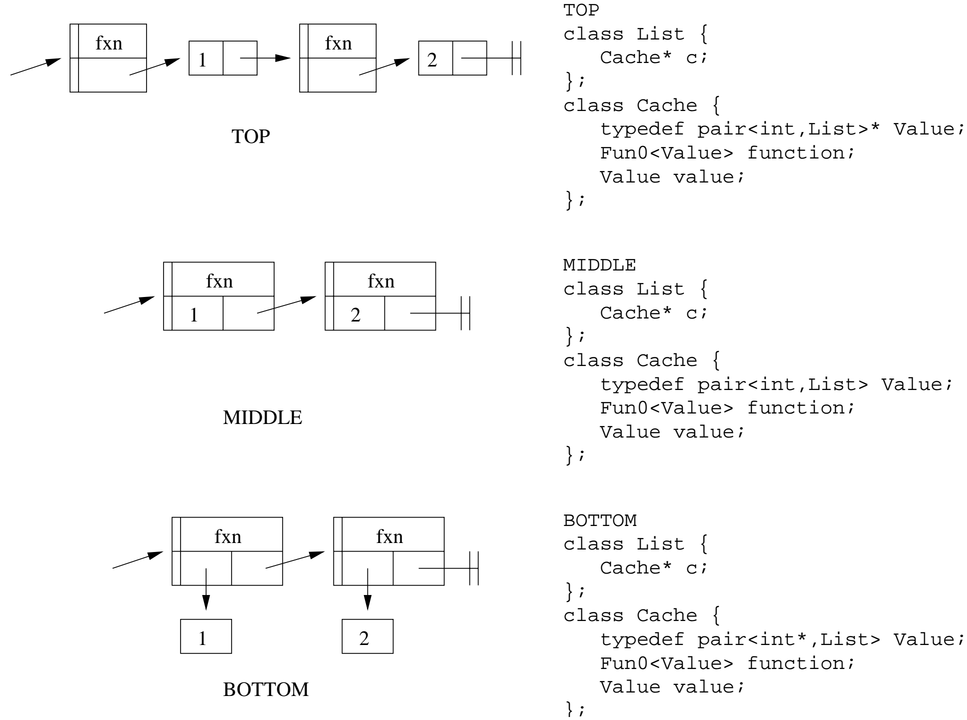

Fig. 12. Three possible list implementations

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

Table 4. *Comparison of different list structures*

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`.)

10.3 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

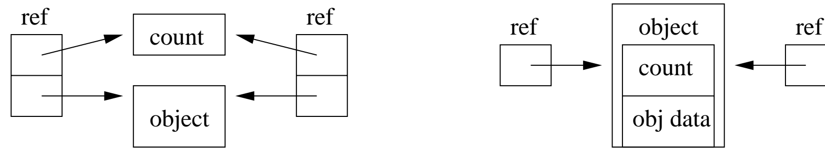

Fig. 13. Non-intrusive reference counting (left) and intrusive reference counting (right)

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*

depicted in Figure 13. 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 reference (Alexandrescu, 2001), 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.

10.4 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 14 (with `Sig` member elided). (Recall that `take` selects the first N elements of a list and 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 15 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

```
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;
```

Fig. 14. take() without functoid reuse

```
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;
```

Fig. 15. take() with functoid reuse

```
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;
```

Fig. 16. take() with reuse via a Reuser

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

Table 6. *The value of reusing functoids*

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

Comparing Figures 14 and 15, 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 16 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 channeled 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 14 and 16), and perform nearly as well as the "hand-written" code to perform the optimization (there is only a small "abstraction penalty").

10.5 Avoiding functions with static data

The `Cache` implementation (Figure 12, 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
};
```

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*

However for many compilers (e.g. g++2.95.2, used in our tests), it is far better to say

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

The reason is that in the former case, 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 case, 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 with static initializations that require constructors to be called can be perilous; there are order-of-initialization and order-of-destruction issues for global objects in C++ that 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 (ISO, 1998), prescribes the order of initialization for such objects). As for order-of-destruction issues, we circumvent the potential problems by artificially 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 may degrade the performance even more severely. This is because returning a `Ref` object by value may create (needless) work, incrementing and decrementing the reference count as the temporary object lives its short life.

We should mention that one of the proposed changes (Core Language issue #270) to the C++ language standard would invalidate the above optimization, by making the order of initialization undefined even for static variables within the same compilation unit. Nevertheless, newer compilers are more likely to optimize function-static data well, removing the need for the optimization in the first place. (g++3.0, for instance, appears to optimize function-static data references.) Thus, the measurements of this section are likely to keep reflecting future trends. The latest release of

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*

FC++ has a compile-time flag that lets users use the library both with compilers that conform to the current C++ semantics and with those that may implement the revised semantics.

10.6 Using iteration instead of tail recursion

C++ compilers do not transform tail recursion into iteration—such an optimization would be rarely safely applicable in C++ code. 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 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.

10.7 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.

10.8 A final comparison

In an earlier paper (McNamara and Smaragdakis, 2000a), we ran an experiment comparing the performance of the FC++ library with Läufer's functoid library.

Tree(100000)	Time (s)
FC++ (+IRC +REUSE +GL +TRO)	1.62
Läufer's library	23.00

Table 10. *Latest comparison with Läufer's library*

That experiment used a program similar to the "tree" program in Section 9.2. The experiment showed that the (previous) FC++ implementation was 4 to 8 times as fast as Läufer's library, thanks to the reference-counting in our implementation. We re-ran the experiment with the new FC++ implementation with all of the optimizations enabled. The results are shown in Table 10; FC++ is now more than 14 times as fast for this benchmark.

11 Lazy Lists: Odd and Even

In this section we shall discuss FC++ lazy lists in more depth. We focus on the unusual dual representation we have for lists—one that exploits C++ implicit conversions to allow lazy lists that are both efficient and easy to use.

As we saw in Section 2, FC++ lazy lists can be `cons()`ed up in the usual way:

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

However, we could also create "infinite" lists with functions like `enumFrom()`; for example, `enumFrom(1)` returns the list of integers 1, 2, 3, ... The implementation of `enumFrom()` reveals how this is done:

```
struct EnumFrom:public CFunType<int,List<int> > {
  List<int> operator()( int x ) const {
    return cons( x, curry1( EnumFrom(), x+1 ) );
  }
} enumFrom;
```

Though short, the function body nevertheless requires explanation because of the apparent inconsistency in the use of `cons()`: does `cons()` accept as its second argument a list, or a thunk (like the above `curry1` expression) returning a list?

The answer is that `cons()` is overloaded to accept either a list or a list thunk. As described in Section 10.2, a `List` is represented as a kind of variant record, whose tail portion may either be a reference to the rest of the list, or an unevaluated thunk which will produce the remainder list on demand.

Wadler, Taha, and MacQueen (1998) point out that there are different "degrees of laziness". Depending upon implementation choices, we may consider a lazy stream of values to be "even" or "odd", where even streams are completely lazy, whereas odd streams sometimes exhibit a little too much eagerness. For example, in this

code (adapted from the main running example in reference (Wadler, Taha, and MacQueen, 1998)):

```
List<double> l = cons( 1.0, cons( 0.0, cons( -1.0, NIL )));
l = take( 2, map( sqrt, l ) );
```

we would expect `l` to have the final value `[1.0,0.0]`. However this will only work for "even" lists; an "odd" list will evaluate one element too far, and end up trying to compute `sqrt(-1.0)` and fail. The details of the differences between the even and odd styles are explicated in reference (Wadler, Taha, and MacQueen, 1998). (As we shall see, FC++ code does not fail in this example; our lists are not over-eager.)

FC++'s lazy lists are neither even nor odd. We have instead chosen a hybrid approach that works well in C++. There are two kinds of lists exposed to users in FC++: `List` and `OddList`. The former is "even", whereas the latter is "odd". An important feature is that *the two kinds of lists are implicitly convertible to one another*. That is, an odd list can be used where an even list is expected (it will be automatically wrapped into a thunk) and an even list can be used where an odd list is expected (it will be automatically unwrapped).

The two list types have exactly the same interface; the only difference between the two is that `OddList`s are always eager in their first element, whereas `List`s are not. It is noteworthy that the eagerness of `OddList`s is effectively limited to the first element; taking the `tail()` of an `OddList` returns an (even) `List` as a result. Most functions other than `tail()` that produce list values (e.g. `cons()`, `enumFrom()`, and `map()`) actually return `OddList`s, and not `List`s.

This may seem an awkward state of affairs, at first. However, this peculiar implementation offers an interesting benefit: by exposing the fact that some list elements are already evaluated (`OddList`s) to the type system, we can overload certain core functions like `head()` to take advantage of this information—to take the `head()` of a `List`, we must evaluate a thunk to produce a value (*after* checking to see if the value has previously been cached, as discussed in Section 10.1), whereas to take the `head()` of an `OddList`, we can simply access a stored value directly, which is much more efficient. Hence the separation of lists into two data types can improve the run-time performance of list code.

This performance benefit comes with two potential costs. First, the overall complexity of the FC++ library is increased by having two list types. Second, since `OddList`s are not completely lazy, there is danger of over-eager computation. We address each concern in turn:

- *Complexity.* While the internal complexity of the library is undoubtedly increased due to the list duality, this extra complexity need not be exposed to library users. Clients of the FC++ library can be oblivious of the existence of `OddList`s and get along just fine. The overloading and implicit conversions with `List`s enable users to write working code that appears to deal solely in `List`s. `List`s effectively provide a facade that shields casual users from the extra complexity. Nevertheless, for users who understand the details of `OddList`s, the datatype is there, ready to be exploited by clients who want to hand-tune some of their code to improve the run-time performance.

- *Eagerness.* Allowing implicit conversions between `OddList`s and `List`s sacrifices some of the safety of even lists. Nevertheless, while the danger of over-eagerness exists, it lives only in "edge cases". The examples of (Wadler, Taha, and MacQueen, 1998), like

  ```
  List<double> l = cons( 1.0, cons( 0.0, cons( -1.0, NIL )));
  l = take( 2, map( sqrt, l ) );
  ```

  work as expected, even though `map` returns an `OddList`. The reason is that our `OddList`s have "even" tails and `tail` is the only way to deconstruct a list in FC++. Only direct calls on boundary cases like

  ```
  List<double> l = cons( -1.0, NIL );
  l = take( 0, map( sqrt, l ) );
  ```

  will cause a failure (`map()` tries to take the square root of -1 before `take()` has the opportunity to mention that it is not interested in evaluating any elements). Note that it is only when the *original call in the client* begins with the "edge case" that the problem occurs—in the first of the two previous examples, the same boundary case (which fails in the second example) is reached after two recursive calls, but since we have already moved past the first element of the list, we are safely in the "even" domain. As a result, the edge cases are unlikely to occur in practice. When necessary, the client can always resort to explicitly forcing the first element to be lazy: rather than evaluating

  ```
  map( sqrt, l )
  ```

  (the offending expression in the boundary case), which has type `OddList`, the user can evaluate

  ```
  curry2( map, sqrt, l )
  ```

  The latter expression evaluates to a thunk that returns an `OddList`, which is implicitly convertible to an (even) `List`. Indeed, this strategy seems well within the spirit of C++; C++ is an eager language, and calling a function is an eager language mechanism, which typically produces a value (or an effect). In those cases where the programmer desires extra laziness, she codes it explicitly using `curryN()`. This is, after all, how lazy functions like `enumFrom()` are implemented (with calls to `curryN()`).

To summarize, the list implementation in FC++ uses a novel "hybrid" approach to laziness. While we consider the details of this approach to be interesting and important from an implementation perspective, we emphasize that these details are almost never forced upon clients. We have dozens of example programs that use FC++ lists with no knowledge of any of the details of `OddList`s or edge cases. The goal of this section is simply to describe our implementation as an interesting alternative to the possibilities considered in reference (Wadler, Taha, and MacQueen, 1998).

12 Library organization

The FC++ library is distributed as a set of nine C++ header files. The physical organization of the code matches the conceptual organization of the library.

- `list.h` contains the implementation of FC++ lazy lists. This includes both the `List` and `OddList` classes and their associated functoids, most notably `head()`, `tail()`, `cons()`, and `null()`.
- `function.h` contains the implementation of indirect functoids (the `FunN` classes), including the code that supports implicit conversion from direct to indirect functoids and the code to support subtype polymorphism.
- `curry.h` contains the `CurryableN` combinator classes, and the `bindMofN` and `curryN` functoids for explicit currying.
- `ref_count.h` contains the code for both the `Ref` and `IRef` classes, which support non-intrusive and intrusive reference-counted pointers, respectively.
- `reuse.h` contains the code for the `ReuserN` classes, as described in Section 10.4.
- `signature.h` and `config.h` are tiny "support" classes. `signature.h` contains the definitions of the `FunType` and `CFunType` classes, which are used to help encode the signatures of functoids as described in Section 3.2. `config.h` contains some preprocessor macros that detect certain compiler versions and work around a few known compiler bugs.
- Finally, `operator.h` and `prelude.h` contain most of the useful general-purpose functoids in the library. `operator.h` includes the definitions of named functoids for all the common operators like `plus` and `minus`, as well as a number of miscellaneous functoids. `prelude.h` contains a great deal of functions from the Haskell Standard Prelude, like `map`, `zipWith`, and `take`. `prelude.h` is at the root of the dependency tree of the headers and includes all of the other library functionality—as a result, library users need only `#include "prelude.h"` to import all of the library functionality.

As mentioned earlier, the entire library is contained in `namespace fcpp`, thus shielding the names of library classes and functions from conflicting with names in client applications. The entire library comprises about 5400 lines of code, and is available on the FC++ web site (Smaragdakis and McNamara, 2002).

13 Applications

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.

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 (Briggs, 2002). 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 (Smaragdakis and McNamara, to appear), we show how a number of OO design patterns can be simplified, generalized, or made safer using functional programming techniques.

14 Related Work

Läufer's paper (1995) contains a good survey of the 1995 state of the art regarding functionally-inspired C++ constructs. Here we will only review more recent or closely related pieces of work.

Dami (1991) implements currying in C/C++/Objective-C and shows the utility in applications. His implementation requires modification of the compiler, though. The utility comes mostly in C; in C++, more sophisticated approaches (such as ours) can achieve the same goals and more.

Kiselyov (1998) implements some macros that allow for the creation of simple mock-closures in C++. These merely provide syntactic sugar for C++'s intrinsic support for basic function-objects. We chose not to incorporate such sugar in FC++, as we feel the dangers inherent in C-preprocessor macros outweigh the minor benefits of syntax. FC++ users can define their own syntactic helpers, if desired.

Two other interesting recent approaches are FACT! (Striegnitz, 2001) and the Boost Lambda Library (Järvi and Powell, 2002). Both of these libraries emphasize the "front-end", by providing lambda expressions in C++ via expression templates[7] and operator overloading. FC++, on the other hand, provides sophisticated type system support for higher-order and polymorphic functions. Hence, these approaches are complementary. Syntactic support for creating lambdas through expression templates like in FACT! or the Boost Lambda Library can be added to FC++[8]. At the same time, most of the type system innovations of FC++ can be integrated into the back-end of these libraries (enabling full support for higher-order polymorphic functions and rank-2 polymorphism).

The C++ Standard Template Library (STL) (Stepanov and Lee, 1995) includes a library called `<functional>`. It supports a very limited set of operations for creating and composing functoids that are usable (in monomorphic form) with algorithms from the `<algorithm>` library. While it serves a useful purpose for a number of C++ tasks, it is inadequate as a basis for building higher-order polymorphic functoids.

Läufer (1995) presents a framework for supporting functional programming in C++. His approach supports lazy evaluation, higher-order functions, and binding variables to different function values. His implementation does not include polymorphic functions, though, and also uses an inefficient means for representing function objects. In many ways, our work can be viewed as an extension to Läufer's; our framework improves on his by adding both parametric and subtype polymorphism,

[7] See Veldhuizen's paper (1995) for an introduction to the subject of expression templates.

[8] Indeed, we are currently working on adding just such syntactic sugar: lambdas, let-bindings, do-notation for monads, etc.

improving efficiency, and contributing a large functional library. Läufer also examines topics that we did not touch upon in this paper, like architecture-specific mechanisms for converting higher-order functions into regular C++ functions.

Alexandrescu's book (2001) contains a chapter on "generalized functors". These functors are similar to our indirect functoids, except that they do not support implicit currying or subtype polymorphism. In another chapter, Alexandrescu also describes reference-counting mechanisms, including intrusive ref-counts, like the ones we use with FC++'s internal reference-counted pointers.

15 Conclusions

FC++ is a library for doing functional programming in C++. In the examples throughout the paper, we have demonstrated how the FC++ library adds many functional programming features to C++, including

- the ability to use higher-order polymorphic functions succinctly (a major novelty among C++ libraries),
- run-time variables which can range over functions with the same monomorphic signature,
- currying, and
- lazy evaluation, using lists and a number of functions which lazily manipulate lists.

These features have been implemented efficiently, in a way that blends well with the C++ language. It is an interesting fact, and a testimony to the extensibility features of C++, that so much language functionality can be added in directions that were almost certainly not foreseen when the language was designed. Although a number of sophisticated techniques are used within the library implementation, we have provided a library that is both useful and usable; the vast majority of the complexity is hidden from clients of the library.

FC++ is a good platform for language experimentation, as it offers a combination of functional and object-oriented language features. It allows traditional C++ programmers to integrate functional techniques in their arsenal. At the same time, it gives functional programmers a distinctly different platform for experimentation. Perhaps most importantly, FC++ brings maturity to a long line of research efforts in using C++ as a functional language.

Acknowledgments

We thank the users of the FC++ library, many of whom have given us useful feedback on earlier versions of the library and helped motivate us to make improvements. We also thank the reviewers for their many helpful suggestions.

References

Alexandrescu, A. (2001) *Modern C++ Design*, Addison-Wesley.

Boehm, H., and Demers, A. (2002). The Boehm-Demers-Weiser conservative garbage collector. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

Briggs, K. (2002) *The XR Exact Real Home Page.* `http://www.btexact.com/people/briggsk2/XR.html`

Dami, L. (1991) "More Functional Reusability in C/C++/ Objective-C with Curried Functions", Object Composition, Centre Universitaire d'Informatique, University of Geneva, pp. 85-98.

Fokker, J. (1995) Functional Programming, `http://www.haskell.org/bookshelf/functional-programming.dvi`

Hamilton, J. (1997) "Montana Smart Pointers: They're Smart, and They're Pointers", *Proc. Conf. Object-Oriented Technologies and Systems (COOTS)*, Portland.

ISO (1998) *ISO/IEC 14882: Programming Languages – C++.*

Järvi, J. and Powell, G. (2002) *The Boost Lambda Library.* `http://boost.org/libs/lambda/doc/index.html`

Johnson, R. and Foote, B. (1988) "Designing Reusable Classes", *Journal of Object-Oriented Programming*, 1(2): 22-35.

Jones, M., and Reid, A. (2002) The Hugs homepage: `http://www.haskell.org/hugs/`

Kfoury, A. and Tiuryn, J. (1992) "Type reconstruction in finite rank fragments of the second-order lambda-calculus", *Information and Computation*, 98(2):228-257.

Kiselyov, O. (1998) "Functional Style in C++: Closures, Late Binding, and Lambda Abstractions", poster presentation, *Int. Conf. on Functional Programming*, Baltimore.

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

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

McNamara, B. and Smaragdakis, Y. (2000b) "Static Interfaces in C++", *Workshop on C++ Template Programming*, Erfurt, Germany. `http://www.oonumerics.org/tmpw00/`

Meijer, E. and Kettner, L. (2000) "C++ as a Functional Language", discussion in Dagstuhl Seminar 99081. See: `http://www.cs.unc.edu/~kettner/pieces/flatten.html`

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

Peyton-Jones, S. and Hughes, J. (eds.) (1999) Report on the Programming Language Haskell 98, available from `http://www.haskell.org/onlinereport/`

Peyton-Jones, S., Marlow, S., and Seward, J. (2002) The Glasgow Haskell Compiler homepage: `http://www.haskell.org/ghc/`

Siek, J. and Lumsdaine, A. (2000) "Concept Checking: Binding Parametric Polymorphism in C++", *Workshop on C++ Template Programming*, Erfurt, Germany. `http://www.oonumerics.org/tmpw00/`

Smaragdakis, Y. and McNamara, B. (2002) The FC++ web page: `http://www.cc.gatech.edu/~yannis/fc++/`

Smaragdakis, Y. and McNamara, B. (*to appear*) "FC++: Functional Tools for Object-Oriented Tasks". *Software—Practice & Experience.* Earlier version is available as Georgia Tech CoC Tech. Report 00-37 and at the FC++ web page.

Stepanov, A. and Lee, M. (1995) "The Standard Template Library". Incorporated in ANSI/ISO Committee C++ Standard.

Striegnitz, J. (2001) *FACT! The Functional Side of C++*, `http://www.fz-juelich.de/zam/FACT`

Stroustrup, B. (1996) "A History of C++: 1979-1991", in T. Bergin, and R. Gibson (eds),

Proc. 2nd ACM History of Programming Languages Conference, pp. 699-752. ACM Press, New York.

Veldhuizen, T. (1995) "Expression Templates", *C++ Report*, Vol. 7 No. 5, pp. 26-31.

Wadler, P. (1990) "Comprehending Monads", Proc. ACM Conf. on Lisp and Functional Programming, p. 61-78.

Wadler, P., Taha, W., and MacQueen, D. (1998) "How to add laziness to a strict language, without even being odd", *Workshop on Standard ML*, Baltimore.