Functional Programming in C++

Brian McNamara and Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{lorgon, yannis}@cc.gatech.edu

ABSTRACT

This paper describes FC++: 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.

Another new element in FC++ is that it implements a subtype polymorphism policy for functions, in addition to the more common parametric polymorphism facilities. Subtype polymorphism is common in object oriented languages and ensures that functions in FC++ fit well within the C++ object model.

Apart from these conceptual differences, FC++ is also an improvement in technical terms over previous efforts in the literature. Our function objects are reference-counted and can be aliased without needing to be copied, resulting in an efficient implementation. The reference-counting mechanism is also exported to the user as a general-purpose replacement of native C++ pointers. Finally, we supply a number of useful functional operators (a large part of the Haskell Standard Prelude) to facilitate programming with FC++. The end result is a library that is usable and efficient, while requiring no extensions to the base C++ language.

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 [9] 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++ [5][6]. C++ also allows users to define a nice 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 (e.g., [5][6][8]) have (re-)discovered that C++ can be used for functional programming. Läufer's work [6], stands out as it represents a well-documented, *reusable* framework for higher-order functions, instead of a one-of-a-kind implementation. 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 [6]: "...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.[1]

Our work addresses both of the above problems. Contrary to prior belief (see Läufer [6], who also quotes personal communication with Dami) no modification to the language or the compiler is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP '00, Montreal, Canada.
Copyright 2000 ACM 1-58113-202-6/00/0009...$5.00.

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 our 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 two more new elements:

- First, we define a subtyping policy for functions of FC++, thus supporting *subtype polymorphism*. The default policy is hardly unexpected: 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 `double -> char` can be used where a function `int -> int` is expected.
- Second, 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. This is done by allowing function objects to be multiply referenced (aliased), albeit only through garbage collected "pointers". The difference in performance is substantial: compared to Läufer's framework,[2] and running with the same client code (the main example implemented by Läufer) we achieve a 4- to 8-fold speedup.

  Additionally, FC++ builds significant functionality on top of the basic framework. We export a fairly mature reference-counting "pointer" class 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 describes our representation of monomorphic and polymorphic functions. Section 3 shows some representative examples using our framework. Section 4 analyzes the expressiveness and limitations of FC++. Section 5 discusses aspects of our implementation. Section 6 presents related work. Section 7 contains our conclusions.

2 REPRESENTING FUNCTIONS

To clarify our goal of embedding a functional language in C++, a quick example is useful. In Haskell [10]:

```
take 5 (map odd [1..])
```

evaluates to

```
[True,False,True,False,True].
```

With FC++, we can do the same in C++:

```
take( 5, map( odd, enumFrom(1) ) )
```

Note some aspects of the above example: `enumFrom` creates an "infinite" list. `map` demonstrates the support for higher order functions. `take` is one of many standard Haskell functions supported by FC++. The differences between the C++ and Haskell code are limited to syntax details: the C++ version has extra parentheses and commas, and lacks the syntactic sugar for common operations (e.g. "`[1..]`").

Previous C++ libraries that supported functional idioms could also express something resembling the above example. The biggest difference, however, is that in FC++ all the elements of our example are fully polymorphic. FC++ is the first C++ library to support *rank-2 polymorphism*: passing polymorphic entities as parameters to other polymorphic entities. Support for rank-2 polymorphism is much more useful in C++ than in functional languages like Haskell or ML, because of the limited capabilities of C++ type inference. Since C++ type inference only unifies function arguments, without rank-2 polymorphism the above example would need to either a) use a monomorphic function `odd`, or b) explicitly instantiate a class template `Odd`, e.g.,

```
take( 5, map( Odd<int>, enumFrom(1) ) ).
```

The problem with approach (a) is that `odd` would not be general enough to be used with, say, complex numbers. The problem with approach (b) is that it is inconvenient due to the explicit type signatures and because it requires maintaining class template versions as well as function template versions for the same concepts.

Following a convention also used by Läufer, we use the term *functoid* to refer to our abstraction of the familiar concept of "function". For clarity, we will distinguish between two kinds of functoids in FC++: *indirect functoids*, and *direct functoids*.

Indirect functoids are first-class entities in FC++. That is, one can define variables ranging over all indirect functoids with the same

---

1. The Standard Library offers "Class template versions" of these operators (`unary_compose` for `compose1` and `binder1st` for `bind1st`) but these suffer from the first problem (high complexity of parameterization).
2. In the object oriented community, the term "framework" usually refers to an "applications framework" [4]: a set of reusable classes that can be refined through subclassing and definition of dynamic methods. Läufer's framework is a minimal applications framework. FC++ is based on an applications framework (slightly different from Läufer's) but also employs other mechanisms (e.g., C++ method templates). In this paper, the word "framework" will not always refer to an applications framework.

type signature. Indirect functoids can only represent monomorphic functions, however. In contrast, direct functoids can represent either monomorphic or polymorphic functions but they are not first-class entities. Direct functoids allow us to exploit C++ type inference. Conversion operators from direct functoids (for a monomorphic type signature) to indirect functoids are provided in the library.

We will first describe the special case of *monomorphic* direct functoids, because they are simpler and serve as a good introduction for readers not familiar with C++.

2.1 Monomorphic Direct Functoids

C++ is a class-based object-oriented language. Classes are defined statically using the keywords `struct` or `class`.[3] 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 to double and add one to a number:

```
struct Twice {
  int operator()( int x ) { return 2*x; }
} twice;

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

  twice(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 `twice` 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 `Twice` 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 2.3). Thus, we would like to encapsulate some representation of the type signature of `Twice` in its definition. The details of this representation will be filled in Section 2.3, but for now it suffices to say that each direct functoid has a member called `Sig` (e.g., `Twice::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, `Twice` would be defined as:

```
struct Twice : public CFun1Type<int, int> {
  int operator()( int x ) { return 2*x; }
} twice;
```

3. The difference between the two keywords is that members of a `struct` are by default accessible to non-class code (`public`), whereas members of a `class` are not (they default to `private`).

That is, `CFun1Type` is a C++ class template whose only purpose is to define signatures. A class inheriting from `CFun1Type<A,B>` is a 1-argument monomorphic direct functoid that encodes a function from type `A` to type `B`. In general, the template `CFun`*`N`*`Type` is used to define signatures for monomorphic direct functoids of *`N`* arguments.

Note that in the above definition of `Twice` we specify the type signature information (`int -> int`) twice: once in the definition of `operator()` (for compiler use) and once in `CFun1Type<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`, address this issue.

Monomorphic direct functoids have a number of advantages over normal C++ functions: they can be passed as parameters, they can have state, they can be given aliases using "`typedef`", 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 classes into monomorphic direct functoids.

2.2 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,[4] 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 prevents the creation of cyclical data structures, thus avoiding the usual pitfalls of reference-counting garbage collection.

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

`Fun`*`N`*`<`*`A1`*`,`*`A2`*`...,`*`AN`*`,`*`R`*`>`.

For instance, one-argument indirect functoids with signature

`P -> R`

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

```
template <class Arg1, class Result>
class Fun1 : public CFun1Type<Arg1,Result> {
  Ref<Fun1Impl<Arg1,Result> > ref;
  ...
public:
```

4. In C++, memory deallocation is explicit; objects created with `new` must be explicitly freed with `delete`.

```
  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 CFun1Type<Arg1,Result> {
  virtual Result operator()(const Arg1&) const=0;
  virtual ~Fun1Impl() {}
};
```

(*Notes on the code*: 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 2.5. The `Ref` class template implements our reference-counted "pointers" and will be discussed in Section 5.3. For this internal use, any simple reference counting mechanism would be sufficient.)

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

`P -> 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 will see in Section 2.3, the most convenient representation of functions is that of polymorphic direct functoids.

Monomorphic direct functoids can be converted to indirect functoids, using operations `makeFun`*N* (provided by FC++). For instance, consider direct functoids `Twice` and `Inc` from Section 2.1 (the definition of `Inc` was not shown). The following example is illustrative:

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

It should be noted here that our indirect functoids are very similar to the functoids presented in Läufer's work [6]. Indeed, the only difference is in the wrapper classes, `Fun`*N*`<`*A1*`,`*A2*`,...,`*AN*`,`*R*`>`. Whereas we use a reference counting mechanism, Läufer's implementation allowed no aliasing: different instances of `Fun`*N*`<`*A1*`,`*A2*`...,`*AN*`,`*R*`>` had to refer to different instances of `Fun`*N*`Impl<`*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, often by an order of magnitude, as we will see in Section 5.1. Also, unlike Läufer's functoids, 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.

2.3 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, called `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 Fun1Type<L,L> {};

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

where `Fun1Type` is used for convenience, as a reusable mechanism for naming arguments and results.[5]

5. Its definition is just:
```
template <class A1, class R> struct Fun1Type {
  typedef R ResultType;
  typedef A1 Arg1Type;
};
```

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 2.1, for monomorphic direct functoids, the member class `Sig` is hidden inside the `CFun`*`N`*`Type` 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 `compose1` to act like ".", enabling us to create functoids like

```
compose1(tail,tail).
```

The definition of `compose1` uses type information from `tail` as captured in its `Sig` structure. Using this information, the type of `compose1(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`. 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 `monomorphize`. For instance:

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

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

2.4 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. Section 3.2 also shows the benefits of direct functoids in a more realistic, and hence more complex, example. To avoid confusion we will show here a toy example where the difference is, hopefully, much clearer.

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

```
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—we will not concern ourselves with the issue in this paper.)

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 `twice` 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 1 shows the implementations of `map` and `twice` using indirect functoids:

```
// N: Number type
template <class N>
struct Twice : public Fun1Impl<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
  Fun2Impl<Fun1<E, R>, List<E>, List<R> >
{
  List<R>
  operator()(Fun1<E,R> f, List<E> l) const {...}
};
```

Figure 1: Polymorphic functions as templates over 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 `twice` and `map` using direct functoids (Figure 2). Direct functoids can be converted to indirect functoids for a fixed type signature, hence there is no loss of expressiveness. (In fact, the code for `Map` shown in Figure 2 is simpler than the real code, but only very slightly. In reality, the C++ keyword `typename` needs to precede the reference to `RT1` and `L::EleType`, so that the compiler knows that their nested members `EleType` and `ResultType` are types.)

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

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

// F: function type
// L: list type
struct Map {
  template <class F, class L>
  struct Sig : public
   Fun2Type<F,L,
     List<RT1<F, L::EleType>::ResultType> > {};

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

Figure 2: Polymorphic functions as direct functoids.

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 `Fun2Type` is just a simple template for creating function signatures—see footnote 5.

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 `RT1` type is just a shorthand for reading the return type of a 1-argument function. Since that type may be a function of the input type, `RT1` is parameterized both by the function and by the input type. That is, the type computation
`RT1<F, L::EleType>::ResultType`
means "result type of function `F`, when its argument type is the element type of list `L`".

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 `twice`. In Haskell, we can say
```
  map twice [1..]
```
to produce a list of even numbers. With direct functoids (Figure 2) we can similarly say
```
  map( twice, enumFrom(1) ).
```
This succinctness is a direct consequence of using C++ type inference. With the indirect functoid solution (Figure 1) 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 Twice <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 `compose1` functoid mentioned in Section 2.3, we can create a list of multiples of four by writing
```
  map(compose1(twice, twice), enumFrom(1)).
```
The same using indirect functoids would be written as
```
  Fun1<int, int> twice (new Twice <int>());
  Map <int, int>()
    (Compose1<int, int, int>()
      (twice, twice),
     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 [6]).

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 `twice` could have been implemented as a C++ function template:
```
template <class N>  N twice (const N &n)
{ return 2*n; }
```
This is the most widespread C++ idiom for approximating polymorphic functions (e.g., [8][11]). 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 `twice` is passed as an argument to `map` is not realizable if `twice` is implemented as a function template.

What both of the above alternatives to direct functoids lack is the ability to express polymorphic functions that can accept other polymorphic functions as arguments. This rank-2 polymorphism capability of FC++ direct functoids is unique among C++ libraries for functional programming. The closest approximation of this functionality before FC++ was with the use of a hybrid of class templates, like in Figure 1, 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 type inference can be used when arguments are passed to the function). The C++ Standard Library 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 `Twice<int>` instead of `twice` 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.

2.5 Subtype Polymorphism

Another innovation of our framework is that it implements a policy of subtype polymorphism for functoids. Our policy is contravariant with respect to argument types and covariant with respect to result types.

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;
  // legal: 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 `Fun`*N* of Section 2.2. For instance, the definition of `Fun1` has the form:

```
template <class Arg1, class Result>
class Fun1 : public CFun1Type<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)) {}
};
```

(The new part is italicized.) 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 [13] 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++ [7].

Subtype polymorphism is important, because it is a familiar concept in object orientation. It ensures that indirect functoids can be used like any C++ object reference in real C++ programs.

3 USAGE EXAMPLES

In this section, we show some complete functional programs written using FC++. For reference, we compare to implementations of the same examples in Haskell, with the type signatures specified explicitly for easy reference.

3.1 Primes

A simple algorithm for determining if a number x is prime is to compute all of its factors, and see if the resulting list of factors is just the list `[1,`*x*`]`. Figure 3 shows the Haskell code to compute the first n prime numbers (the code is slightly modified version of that on p. 29 of [2]) and the corresponding implementation in C++. This example illustrates that the mapping from Haskell to C++ is straightforward and exact.

3.2 Tree Fringe

As another sample program, consider computing the fringe of a binary tree. We define a Tree to be either a Node, which comprises a piece of data and references to two other Trees, or Nil. For convenience, we say that a Tree meets the predicate leaf() if it is a Node whose subtrees are both Nil. The code for Trees in Haskell is:

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

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

We can define a similar data type in C++ as:

```
template <class T>
struct Tree {
  typedef T WrappedType;
  T data;
  Ref<Tree<T> > left, right;

  Tree( T x ): data(x), left(0), right(0) {}
  bool leaf() const
  { return (left==0) && (right==0); }
};
```

Let us define the *fringe* of the tree to be a list of all the data values stored in the leaves of the tree, in left-to-right order. Figure 4 shows the Haskell and C++ code to compute the fringe. The C++ version has the usual extra "noise" to declare the type of a direct functoid, but the body is exactly analogous to the Haskell version: If we reach nil, we return an empty list; else if we find a leaf, we return a one-element list; otherwise we lazily catenate the results

```
divisible  :: Int -> Int -> Bool
divisible t n = t `rem` n == 0


factors  :: Int -> [Int]
factors x = filter (divisible x) [1..x]


prime  :: Int -> Bool
prime x = factors x == [1,x]


primes  :: Int -> [Int]
primes n = take n (filter prime [1..])
```

```
struct Divisible
: public CFun2Type<int,int,bool> {
  bool operator()( int x, int y ) const
  { return x%y==0; }
} divisible;

struct Factors
: public CFun1Type<int,List<int> > {
  List<int> operator()( int x ) const
  { return filter( bind1of2(divisible,x),
                   enumFromTo(1,x) ); }
} factors;

struct Prime : public CFun1Type<int,bool> {
  bool operator()( int x ) const
  { return factors(x)==list_with(1,x); }
} prime;

struct Primes
: public CFun1Type<int,List<int> > {
  List<int> operator()( int n ) const {
    return take(n,filter(prime,enumFrom(1)));
  }
} primes;
```

Figure 3: Haskell and C++ code for computing the first n prime numbers.

```
fringe :: Tree a -> [a]


fringe Nil        = []


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

```
struct Fringe {
  template <class RTT> struct Sig
  : public Fun1Type<RTT,List<typename
    RTT::WrappedType::WrappedType> > {};

  template <class T>
  List<T> operator()(Ref<Tree<T> > t)const{
    if( t==0 )
      return List<T>();
    else if( t->leaf() )
      return one_element(t->data);
    else
      return cat( bind1of1(Fringe(),t->left),
                  bind1of1(Fringe(),t->right));
  }
} fringe;
```

Figure 4: Haskell and C++ code for computing the fringe of a polymorphic tree.

of recursive calls on the left and right subtrees. (Note that `bind1of1` is used here to effect laziness; `bind1of1` curries the only argument of a unary function to yield a 0-arg functoid which does not actually "do work" until it is needed and explicitly invoked.)

Clients can then compare the fringes of two trees with

```
fringe tree1 == fringe tree2   -- Haskell
fringe(tree1) == fringe(tree2) // C++
```

which again demonstrates that *using* functoids is easy, even if *implementing* them is somewhat tedious due to the `Sig` member. The `fringe` functoid is both polymorphic (it works on trees of integers, strings, etc.) and lazy (if two large trees' fringes differ in the first element, the rest of the trees will not be traversed, since the result of the equality comparison is guaranteed to be false after the first mismatch).

To see the importance of our implementation of parametric polymorphism via direct functoids, Figure 5 shows what the code would look like if we instead had chosen to implement `fringe` using templatized versions of indirect functoids. The last return statement (italicized) in the function exemplifies the complexity one repeatedly encounters when trying to express polymorphic functions using the framework of [6].

In reality, the code segment of Figure 5 *masks much of the complexity*, because it uses `makeFun0` and `makeFun1`, which are function templates and employ C++ type inference. If we had also expressed `makeFun0` and `makeFun1` using exclusively indirect functoids, the example would be even longer. We believe that Figure 5 demonstrates convincingly why we consider our framework to be the first usable attempt at incorporating both higher-order functions and polymorphism in C++. It is not surprising that previous approaches using higher-order functions in C++ (e.g, [5]) have shied away from polymorphism.

```
template <class T>
struct Fringe
: public CFun1Type<Ref<Tree<T> >,List<T> > {
  List<T> operator()( Ref<Tree<T> > t )
  const {
    if( t==0 )
      return List<T>();
    else if( t->leaf() )
      return OneElement<T>()(t->data);
    else              // take a deep breath!
      return Cat<T>()(
     makeFun0(
      Bind1of1<
        Fun1<Ref<Tree<T> >,List<T> >,
        Ref<Tree<T> > >()
          (makeFun1(Fringe<T>()), t->left),
      Bind1of1<Fun1<Ref<Tree<T> >,List<T> >,
               Ref<Tree<T> > >()
         (makeFun1(Fringe<T>()), t->right));
  }
};
```

Figure 5: Tree fringe computation using templates over indirect functoids. This example demonstrates why templates over indirect functoids are undesirable.

4 DISCUSSION: SUPPORT AND LIMITATIONS

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 only required 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 (e.g., pointers and numbers, but *not* C++ native functions, which are not first-class entities).

- *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. We do not expect this to be a 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 have experimented with this idea, and it is a candidate for inclusion in the next version of FC++.

  Another desirable capability is that of implicit currying whenever a function is used with fewer actual arguments than formal arguments. The challenge is to implement this functionality in a reusable way, instead of adding it to the definition of individual functoids. This is straightforward for monomorphic functoids, but not for polymorphic direct functoids. Therefore, for uniformity reasons, we have not yet included this feature in FC++.

- *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. Overall, however, type error reporting in FC++ is quite 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 2.2 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 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.

- *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 2.2) are explicitly side-effect free—any class inheriting from our `Fun`*`N`*`Impl` 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 `Fun`*`N`*`Impl`. 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. Certainly functoids with side effects can be implemented in our framework, but this seems both unnecessary and dangerous. Other opportunities for code with side effects abound in C++. Our recommendation is that code with side effects be implemented outside the FC++ framework. For instance, such code could be expressed through native C++ functions. The purist can even define monads [14] using FC++.

5 TECHNICAL ISSUES

5.1 Library Performance

As explained in Section 2.2, our basic framework for indirect functoids is quite efficient due to its fast memory management through reference counted pointers. Läufer's framework for functional programming in C++ [6] defines functoids that are very similar to our indirect functoids. Läufer's implementation ensures that no functoid is accessible through multiple references, by copying functoids at aliasing time (i.e., when a reference is assigned to another). Since indirect functoid objects do not hold mutable state (they cannot, as they only have `const` methods), reference-counting is as safe as copying.

To measure the consequences of our optimization, we performed a simple experiment. We used Läufer's implementation of a lazy tree fringe computation, with practically no modifications to his code (to ensure that no bias is introduced). This program comprises the main example in Läufer's library and consists of a *monomorphic* tree fringe computation (Figure 5 suggests that a polymorphic implementation would be unmanageable). We attached to the main implementation a client program that requests a tree fringe computation on a randomly produced tree of a given size. In Table 1, we show timings in milliseconds on a Sun Ultra 5 (Sparc 300MHz processor), with both programs compiled using egcs-2.91.66 (the GNU C++ compiler) at optimization level 3. Different optimization levels did not change the results qualitatively. The timings are only for the fringe computation (i.e., after tree construction). We first determined that the variation across different executions was not significant, and then ran a single experiment for each problem size (with a warm cache, as the tree was created right before the fringe computation).

The first column of Table 1 shows the number of nodes in the tree. The second shows the time taken when the tree fringe client used Läufer's functoid implementation. The third column shows the execution time of the same program with our indirect functoid implementation. The fourth column shows the ratio of the two for easy reference. The fifth column shows the performance of a reference implementation: a *strict* tree fringe computation, using regular C++ functions (no functoids, no lazy evaluation) but still free of side-effects.

Table 1: Performance of FC++ in the tree fringe problem (msec)

Tree Size (nodes)	Läufer Library	Indirect Functoids	Ratio	Strict Fringe
1000	121	28	4.32	-
2000	273	63	4.33	-
4000	716	132	5.42	-
8000	1320	256	5.16	16
16000	3590	600	5.98	33
32000	7611	1182	6.44	71
64000	17816	2449	7.27	157
128000	38758	6122	6.33	327
256000	96789	12995	7.45	816
512000	195091	23195	8.41	1361

As can be seen, our indirect functoids perform 4 to 8 times faster than the implementation in [6]. Some small variation in performance can be observed, and this is expected, since the trees are generated from random data. Nevertheless, overall the trend is very clear.[6]

The reason for the superlinear running times of both lazy implementations is that for *n* nodes, a list of approximately *logn* functoid objects is kept as the state of the lazy list. Since the implementation is purely functional, this list needs to be copied when the state is updated, making the overall asymptotic complexity *O(n logn)*. Since copying is very fast with our implementation (a reference count is incremented) the performance of our code is dominated by a linear component, and thus the speedup relative to Läufer's implementation increases for larger trees.

The superlinear scaling of the strict tree fringe computation is also due to copying the fringe for subtrees at each node. That is, the strict computation is also side-effect free and has asymptotic complexity of *O(n logn)*. We chose this to closely match the lazy implementation so that a comparison is meaningful. The performance of the strict version is significantly faster than both lazy versions, but this is hardly unexpected. The main overheads are due to the creation of functoid objects and to dynamic dispatch. We suspect (but have yet to verify) that the former is much more taxing than the latter.

The conclusion from our simple experiment is that our mechanism for avoiding aliased referencing is at least 8 times faster than copying functoids, for realistic kinds of functoids (the more data a functoid object encapsulates, the higher the overhead of copying).

6. Clear enough that we did not consider a more detailed experiment (e.g., one that will quantify the variance between runs) to be needed.

Since many functional programs will involve passing a lot of functoid objects around, as this example amply demonstrates, the consequences for overall performance can be significant.

5.2 Library Members

Our implementation is not merely a framework for functional programming in C++, but an actual usable and extensible library. This section describes the library, which currently consists of a few thousand lines of code.

The entire library is free of side-effects, in that every method is a `const` method, and all parameters are passed by constant reference. This was merely a design choice based on our desire to demonstrate pure functional programming in C++.

The library code conceptually comprises three parts.

5.2.1 Functoids and the `List` *class*

This portion of the library implements the backbone for functional programming in C++. The `Fun`*N* and `Fun`*N*`Impl` classes are defined here, as well as the inheritable signature-helper classes like `Fun1Type`. `List` is defined in a way so that elements of a list are evaluated lazily, thus supporting structures like "infinite" lists.

5.2.2 Standard Prelude

Here we define about 50 functions that appear in the Haskell Standard Prelude (see [10], Section 5.6 and Appendix A), including `map`, `filter`, `take`, `foldl`, `zipWith`, `any`, and `elem`. These functions serve a dual purpose: first, they convinced us that it was possible (and easy!) to convert arbitrary functional code into C++ code using our framework; second, they provide a large and useful library to clients, just as the Haskell Standard Prelude does for Haskell programmers. We chose the exact names and behaviors of Haskell functions whenever possible.[7]

It is worth noting that this portion of the library was implemented in less than a week[8]—even though this section is nearly half of the entire library! Once our general framework was in place and we became familiar with it, implementing a large number of functoids was very easy.

5.2.3 Other utilities

FC++ has a number of utility functoids. We create functoids to represent C++ operators, such as `plus` and `equal_to`, which roughly correspond to operator sections in Haskell. We have functoids for various conversions, such as `monomorphize`, which converts a direct functoid into an indirect functoid by fixing a type signature, and `stl_to_fun1`, which promotes a unary STL-style functoid into one of our functoids.

We also provide functoids for currying function arguments. For example, rather than supplying both arguments to `map` as in

```
map(odd,int_list) // returns List<bool>
```

one could use `bind1of2` to curry the first parameter

```
bind1of2(map,odd) //List<int> -> List<bool>
```

or instead curry the second via

```
bind2of2(map,int_list) //(int->T)-> List<T>.
```

Finally, this portion of the library also implements `Ref`, our reference-counting class, which is described next.

5.3 Ref

There are many "smart pointer" implementations in C++. For FC++, we use a simple reference-counting scheme, as that is sufficient for our functoids. Each `Ref` contains the actual pointer that it wraps up, as well as a pointer to an integer count, which may be shared among `Ref`s.

```
template<class T>
class Ref {
protected:
  T* ptr;
  unsigned int* count;
  void inc()     { ++(*count); }
  ...  // continued next
```

A key for FC++ is that the `Ref`s are subtype polymorphic; that is, `Ref<U>` should behave as a subtype of `Ref<T>` if `U` is a subtype of `T`. We create an implicit conversion via a templatized constructor, which exists in addition to the normal copy constructor.

```
public:
  ...  // implementation technicalities
  Ref(const Ref<T>& other)//copy constructor
  : ptr(other.ptr), count(0) {
     if(ptr) { count = other.count; inc(); }
  }
  template <class U>  // implicit conversion
  Ref(const Ref<U>& other)
  : ptr(implicit_cast<T*>(other.ptr)),
    count(0) {
     if(ptr) { count = other.count; inc(); }
  }
};
```

We are assured that `U` is a subtype of `T` by `implicit_cast` (a common C++ idiom), which is defined as:

```
template<class T, class U>
T implicit_cast( const U& x ) {
  return x;
}
```

`Ref` is a fairly mature "smart pointer" class and can be used as a complete replacement of C++ pointers. A common criticism of smart pointer classes (e.g., [3]) is that they do not support the same conversions as native C++ pointers (e.g., a smart pointer to a derived class cannot be converted into a smart pointer to a base class). `Ref` supports such conversions if and only if they would be supported for native C++ pointers. We encourage use of the `Ref` class for all functional tasks in C++.

6 RELATED WORK

Läufer's paper [6] 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 [1] 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

7. Some Haskell functions, like `break`, have names that conflict with C++ keywords, so we were forced to rename them.
8. In fact, a portion of that week was "wasted" tracking down a template bug in the g++ compiler, and finding a legal workaround.

C++, more sophisticated approaches (such as ours) can achieve the same goals and more.

Kiselyov [5] 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.

An interesting recent approach is that of Striegnitz's FACT! library [12]. FACT! provides a functional sublanguage inside C++ by extensive use of templates and operator overloading. FACT! emphasizes the front-end, with lambda expressions and support for functions with arbitrarily many arguments. FC++, on the other hand, provides sophisticated type system support for higher-order and polymorphic functions. Hence, the two approaches are complementary. The FACT! front-end support for expression templates can be added to FC++. At the same time, most of the type system innovations of FC++ can be integrated into FACT!'s back-end (enabling full support for higher-order polymoprhic functions and rank-2 polymorphism). Lazy lists are to be included in the next version of FACT! [Striegnitz, personal communication].

The C++ Standard Template Library (STL) [11] includes a library called `<functional>`. It supports a very limited set of operations for creating and composing functoids that are usable with algorithms from the `<algorithm>` library. While it serves a useful purpose for many C++ tasks, it is inadequate as a basis for building higher-order polymorphic functoids.

Läufer [6] has the most sophisticated 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, 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.

7 IMPACT AND CONCLUSIONS

We have described our implementation of a functional programming library for C++. Our work improves upon previous work in the literature in two key ways: we add better support for polymorphism, and we improve the run-time efficiency of functoids. We believe these improvements make our framework the first that is actually usable and scalable for doing real functional programming in C++. In particular, our novel way to support parametric polymorphism using direct functoids allows our functoids to be used without having to explicitly specify intermediate types; our framework enables the compiler to infer these types and the final type of an expression is checked against its declared type, similarly to C++ expressions on native types (e.g., numbers). Thus, we demonstrated that C++-style polymorphism, coupled with classes can be used to express higher-order, polymorphic functions—a surprising, and therefore interesting, result.

Our work has the potential for impact on a number of different communities:

- For C++ programmers, our framework opens new opportunities for functional programming using C++. The reusability benefits of higher-order polymorphic functions are well-known. Additionally, there are problems (e.g., callbacks within a specific context, delayed evaluation, dynamic function specialization) to which FC++ offers more concise solutions than any of the alternatives.
- For functional programmers, our framework enables an alternative platform for implementing functional programs. C++ may be an interesting platform, exactly because it is very different.
- For language researchers, our work shows that C++ type inference is a promising mechanism. The results are surprising even to experts, indicating that the C++ type system is still not well understood.

8 REFERENCES

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

[2] J. Fokker, *Functional Programming*, `http://haskell.org/bookshelf/functional-programming.dvi`

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

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

[5] O. Kiselyov, "Functional Style in C++: Closures, Late Binding, and Lambda Abstractions", *poster presentation, Int. Conf. on Functional Programming*, 1998. See also: `http://www.lh.com/~oleg/ftp/` .

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

[7] B. McNamara and Y. Smaragdakis, "FC++: The Functional C++ Library", `http://www.cc.gatech.edu/~yannis/fc++` .

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

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

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

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

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

[13] B. Stroustrup, "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, 1996.

[14] P. Wadler, "Comprehending Monads", *Proc. ACM Conf. on Lisp and Functional Programming*, p. 61-78, 1990.