Syntax sugar for FC++: lambda, infix, monads, and more

Brian McNamara and Yannis Smaragdakis

College of Computing
Georgia Institute of Technology
http://www.cc.gatech.edu/~yannis/fc++/
lorgon,yannis@cc.gatech.edu

Abstract. We discuss the FC++ library, a library for functional programming in C++. We give an overview of the library's features, but focus on recent additions to the library. These additions include the design of our "lambda" sublanguage, which we compare to other lambda libraries for C++. Our lambda sublanguage contains special syntax for programming with monads, which we also discuss in detail. Other recent additions which we discuss are "infix function syntax" and "full functoids".

1 Introduction

FC++[7, 8] is a library for functional programming in C++. We have recently added a number of new features to the FC++ library, most notably an expression template library for creating a *lambda* sublanguage. The lambda sublanguage contains special syntax for programming with *monads* in the style of Haskell. We focus our discussion on the design of this portion of the library (Section 5 and Section 6), but begin with a run-down of the features of FC++ (Section 2 and Section 3) as well as some important implementation details (Section 4).

2 Overview

In FC++, programmers define and use *functoids*. Functoids are the FC++ representation of functions; we will discuss them in more detail in Section 4. The latest version (v1.5) of the FC++ library supports a number of useful features, including

- higher order, polymorphic functoids ("direct" functoids)
- lazy lists
- a large library of functoids, combinators, and monads (most of which duplicate a good portion of the Haskell Standard Prelude[2])
- currying
- infix functoid syntax
- dynamically-bound functoids ("indirect" functoids)

- a small library of effect combinators
- interfaces to C++ Standard Library data structures and algorithms via iterators
- ways to transform methods of classes and normal C++ functions into functoids
- reference-counted "smart" pointers for memory management (used internally by, e.g., our lazy list data structure)

We'll briefly discuss each of these features in the next section. Later on we will discuss

- special syntax to mimic functional language constructs, including *lambda*, *let*, and *letrec*, as well as *do*-notation and *comprehensions* for arbitrary monads

in detail.

The FC++ library is about 9000 lines of C++ code, and is written with strict conformance to the C++ standard[4], which makes it portable to all of the major brands of compilers.

3 Short Examples of various features

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

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

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

```
// addSelf( x )  ==  x + x
```

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

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

Section 4 describes the infrastructure of these "direct functoids", which enables this feat to be implemented.

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

```
enumFrom( 1 )     // yields infinite list [1, 2, 3, ...]
```

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

```
map( addSelf, enumFrom( 1 ) )  // yields infinite list [2, 4, 6, ...]
```

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

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

There are many functoids which work on `Lists`, including `map`. Most of the `List` functions are identical those defined in Haskell[2]. Additionally, a number of basic functions (like the identity function, `id`), combinators (like `flip`: `flip(f)(x,y)==f(y,x)`), and data types (like `List` and `Maybe`; `Maybe` will be discussed in Section 6) are designed to mimic exactly their Haskell counterparts. We also implement functoids for such C++ constructs as constructor calls and `new` calls:

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

and many more (some of which are described below).

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

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

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

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

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

```
curry2( minus, 3, 2 )   // yields a new thunk "f()=3-2"
```

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

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

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

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

```
// Note: plus is polymorphic, the next line selects just "int" version
Fun2<int,int,int> f = plus;
f(3,2);          // yields 5
f = minus;
f(3,2);          // yields 1
```

Indirect functoids are particularly useful in the implementation of callback libraries and some design patterns[11].

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

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

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

```
before( curry1( writeLog, "About to call foo()" ), foo )
```

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

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

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

FC++ comes with its own reference-counted smart pointers: `Ref` and `IRef`. `Ref<T>` works just like a `T*`, only with reference counting. `IRef<T>` implements intrusive reference counting; an efficient form of reference counting which requires supportive help from the type being used (here, `T`). Internally, the library uses `IRefs` in the implementation of `Lists` and indirect functoids.

4 Where is the magic?

In the previous section we saw how functoids can be used. Nevertheless, we have not shown you how the polymorphic functoids inside FC++ are implemented or how to define your own polymorphic functoids. In this section we show how functoids are defined, and how they gain the special functionality FC++ supports (like currying and infix syntax).

4.1 Defining polymorphic functoids

To create your own polymorphic functoid, you need to create a class with two main elements: a template `operator()` and a member structure template named `Sig`. To make things concrete, consider the definition of `map` (or rather, the class `Map`, of which `map` is a unique instance) shown in Figure 1. This definition uses the helper template `FunType`, which is a specialized template for different numbers of arguments. For two arguments, `FunType` is essentially:

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

   template <class F, class T>
   typename Sig<F, List<T> >::ResultType
   operator()( const F& f, const List<T>& l ) const {
      if( null(l) )
         return NIL;
      else
         return cons( f(head(l)), curry2(Map(), f, tail(l)) );
   }
} map;
```

Fig. 1. Defining `map` in FC++

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

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

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

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

(we have elided the `typename` and `template` keywords for readability). This means: "map returns a `List` of what `F` would return if passed an element like the ones in list `L`".

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

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

The `Sig` members of FC++ functoids essentially encode the same information, but in a computational form: `Sig`s are type-computing compile-time functions that are called by the C++ unification mechanism for function templates and implement the FC++ type system. This type system is completely independent

from the native C++ type system—`map`'s type as far as C++ is concerned is just `class Map`. Other FC++ functoids, however, can read the FC++ type information from the `Sig` member of `Map` and use it in their own type computations. The `map` functoid itself uses that information from whatever functoid happens to be passed as its first argument (see the `F::Sig<L::ElementType>::ResultType` expression, above).

4.2 Using the `FullN` wrappers to gain functionality

The definition of `map` in the previous subsection creates what we call a "basic direct functoid" in FC++. However, a number of features of functoids (such as currying and infix syntax, which we saw in Section 3, and lambda-awareness, which will shall describe in Section 5) only work on so-called "full functoids".

Transforming a normal functoid into a full functoid is easy. For example, to define `map` as a full functoid, we change the definition from Figure 1 from

```
struct Map { /* ... */ } map;
```

to

```
struct XMap { /* ... */ };
typedef Full2<XMap> Map;
Map map;
```

That is, `FullN<F>` is the type of the full functoid created out of the basic N-argument functoid `F`. The `FullN` template classes serve as a wrapper around basic functoids. They add all of the FC++ features we are accustomed to (such as currying and infix syntax) to the basic functoid.

Full functoids are a new feature of the FC++ library. Legacy code can promote its basic functoids into full functoids either by making the minor modification to the definition described above, or within an expression by calling the functoid `makeFullN()`, which takes an N-argument basic functoid as an argument and returns the corresponding full functoid as a result.

5 Lambda

Lambda is no stranger to C++. There are a number of existing C++ libraries which enable clients to create new, anonymous functions on-the-fly. Some such libraries, like the C++ STL[12] and its "binders", or previous versions of FC++, allow the creation of new functoids on-the-fly only either by binding some subset of a functions arguments to values (currying) or by using combinators (like `compose`). Other libraries, like the Boost Lambda Library[5] and FACT![13] enable the creation of arbitrary lambdas by using expression templates.

5.1 Motivation

We were motivated to implement lambda by our interest in programming with monads. Experience with previous versions of FC++ made it clear that arbitrary lambdas are a practical necessity if one wants to program with monads. Older versions of FC++ had a number of useful combinators which made it possible to express most arbitrary functions, but lambda makes it practical by making it readable. For example, while implementing a monad, in the middle of an expression you might discover that you need a function with this meaning:

```
lambda(x) { f(g(x),h(x)) }
```

It is possible to implement this function using combinators (without lambda), but the resulting code is practically unreadable:

```
duplicate(compose(flip(compose)(h),compose(f,g)))
```

Alternatively, you can define the new functoid at the top level, give it a name, and then call it:

```
struct XFoo {
   template <class X> struct Sig : public FunType<X,
      typename RT<F<typename RT<G,X>::ResultType,
      typename RT<H,X>::ResultType>::ResultType> {};
   template <class X>
   typename Sig<X>::ResultType operator()( const X& x ) const {
      return f(g(x),h(x));
   }
};
typedef Full1<XFoo> Foo;
Foo foo;
// later use "foo"
```

but clearly this is way too much work, especially when the function in question is a one-time-use ("throwaway") function. Lambda is the only reasonable solution when you need to define short, readable, arbitrary functions on-the-fly.

5.2 Problematic issues with expression-template lambda libraries

Despite the advantages to lambda, we have always maintained a degree of wariness when it comes to C++ lambda libraries (or any expression template library), owing to the intrinsic limitations and caveats of using expression templates in C++. The worrisome issues with expression template libraries in general (or lambda libraries in particular) fall into four major categories:

- **Accidental/early evaluation.** The biggest problem with expression template lambda libraries comes from accidental evaluation of C++ expressions. Consider a short example using the Boost Lambda Library:

  ```
  int a[] = { 5, 3, 8, 4 };
  for_each( a, a+4, cout << _1 << "\n" );
  ```

The third argument to `for_each()` creates an anonymous function to print each element of the array (one element per line). The output is what we would expect:

```
5
3
8
4
```

If we want to add some leading text to each line of output, it is tempting to change the code like this:

```
int a[] = { 5, 3, 8, 4 };
for_each( a, a+4, cout << "Value: " << _1 << "\n" );
```

But (surprise!), the new program prints the added text only once (rather than once per line):

```
Value: 5
3
8
4
```

This is because "`cout << "Value: "`" is a normal C++ expression that the C++ compiler evaluates immediately. Only expressions involving placeholder variables (like `_1`)[1] get "delayed" from evaluation by the expression templates. These accidents are easy to make, and hard to see at a glance.

- **Capture semantics (lambda-specific).** Since C++ is an effect-ful language, it matters whether free variables captured by lambda are captured by-value or by-reference. The library must choose one way or the other, or provide a mechanism by which users can choose explicitly.
- **Compiler error messages.** C++ compilers are notoriously verbose when it comes to reporting errors in template libraries. Things are even worse with expression template libraries, both because there tend to be more levels of depth of template instantiations, and because the expression templates typically expose clients to some new/unfamiliar syntax, which makes it more likely for clients to make accidental errors. Indecipherable error messages may make an otherwise useful library be too annoying for clients to use.
- **Performance.** Expression template libraries sometimes take orders of magnitude longer to compile than comparably-sized C++ programs without expression templates. Also, the generated binary executables are often much larger for programs with expression templates.

For the most part, these problems are intrinsic to all expression template libraries in C++. As a result, when we set out to design a lambda library for FC++, we kept in mind these issues, and tried to design so as to minimize their impact.

[1] Additionally, one can use other special constructs defined by BLL. In the example above, we could get the desired behavior by calling the BLL function `constant()` on the literal string, to delay evaluation.

5.3 Designing for the issues

Here are the design decisions we have made to try to minimize the issues described in the previous subsection.

- **Accidental/early evaluation.** Since the problem itself is intrinsic to the domain, the only way to "attack" this issue is prevention. That is, we cannot prevent users from making mistakes, but we can try to design our lambda to make these mistakes less common and/or more immediately apparent. To this end, we have designed the lambda syntax to be minimalist and visually distinct:
  - **Minimalism.** Rather than overload a large number of operators and include a large number of primitives, we have chosen a minimalist approach. Thus we have only overloaded four operators for lambda language (array brackets for postfix function application, modulus for infix function application, comma for function argument lists, and equality for "let" assignments). Similarly, apart from `lambda`, the only primitives we provide are those for `let`, `letrec`, and if-then-else expressions. These provide a minimal core of expressive power for lambda, without overburdening the user with a wide interface. A narrow interface seems more likely to be remembered and thus less error-prone.
  - **Visual distinctiveness.** Rather than trying to make lambda expressions "blend in" with normal C++ code, we have done the opposite. We have chosen operators which look big and boxy to make lambda expressions "stand out" from normal C++ code. By convention, we name lambda variables with capital letters. By making lambda expressions visually distinct from normal C++ code, we hope to remind the user which code is "lambda" and which code is "normal C++", so that the user won't accidentally mix the two in ways which create accidents of early evaluation.
- **Capture semantics (lambda-specific).** The FC++ library passes arguments by `const&` throughout the library. Effectively this is just another (perhaps efficient) way of saying "by value". As a result, FC++ lambdas capture free variables by value. As with the rest of the FC++ library, the user can explicitly choose reference semantics by capturing *pointers* to objects, rather than the capturing objects themselves.
- **Compiler error messages.** Meta-programming can be used to detect some user errors and diagnose them "within the library" by injecting *custom error messages*[9, 10] into the compiler output. Though many kinds of errors cannot be caught early by the library (lambdas and functoids can often be passed around in potentially legal contexts, but then finally used deep within some template in the wrong context), there are a number of common types of errors that can be nipped in the bud. The FC++ lambda library catches a number of these types of errors and generates custom error messages for them.
- **Performance.** There seems to be little that we (as library authors) can do here. As expression template libraries continue to become more popular, we

can only hope that compilers will become more adept at compiling them quickly. In the meantime, clients of expression template libraries must put up with longer compile times and larger executables.

Thus, given the intrinsic problems/limitations of expression template libraries, we have designed our library to try to minimize those issues whenever possible.

5.4 Lambda in FC++

We now describe what it looks like to do lambda in FC++. Figure 2 shows some examples of lambda. There are a few points which deserve further attention.

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

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

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

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

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

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

Fig. 2. Lambda in FC++

Inside lambda, one uses square brackets instead of round ones for postfix functional call. (This works thanks to the lambda-awareness of full functoids, mentioned in Section 4.) Similarly, the percent sign is used instead of the carat for infix function call. These symbols make lambda code visually distinct so that the appearance of normal-looking (and thus potentially erroneous) code inside a lambda will stand out. Since `operator[]` takes only one argument in C++,

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

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

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

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

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

The if-then-else construct deserves discussion, as we provide three versions: `if0`, `if1`, and `if2`. `if0` is the typical version, and can be used in most instances. It checks to make sure that its second and third arguments (the "then" branch and the "else" branch) will have the same type when evaluated (and issues a helpful custom error message if they won't). The other two ifs are used for difficult type-inferencing issues that come from `letrec`. In the factorial example at the end of Figure 2, for example, the "else" branch is too difficult for FC++ to predict the type of, owing to the recursive call to `F`. This results in `if0` generating an error. Thus we have `if1` and `if2` to deal with situations like these: `if1` works like `if0`, but just assumes the expression's type will be the same as the type of the "then" part, whereas `if2` assumes the type is that of the "else" part. In the factorial example, `if1` is used, and thus the "then" branch (the `int` value `1`) is used to predict that the type of the whole `if1` expression will be `int`.

Having three different ifs makes the lambda interface a little more complicated, but the alternatives seemed to be either (1) to dispose of custom error messages diagnosing if-then-elses whose branches had different types, or (2) to write meta-programs to solve the recursive type equations created by `letrec` to figure out its type within the library. Option (1) is unattractive because the compiler-generated errors from non-parallel if-then-elses are hideous, and option (2) would greatly complicate the metaprogramming in the library and slow down compile-times even more. Thus we think our design choice is justified. Of course, in the vast majority of cases, `if0` is sufficient and this whole issue is moot; only code which uses `letrec` may need `if1` or `if2`.

5.5 Naming the C++ types of lambda expressions

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

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

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

is

```
fcpp::Full2<fcpp::fcpp_lambda::Lambda2<fcpp::fcpp_lambda::exp::
Call<fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Value<
fcpp::Full2<fcpp::impl::XPlus> >,fcpp::fcpp_lambda::exp::CONS<
fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Call<fcpp::
fcpp_lambda::exp::Value<fcpp::Full2<fcpp::impl::XMultiplies> >,
fcpp::fcpp_lambda::exp::CONS<fcpp::fcpp_lambda::exp::Value<int>,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<1>,fcpp::fcpp_lambda::exp::NIL> >,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<2>,fcpp::fcpp_lambda::exp::NIL> >,1,2> >
```

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

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

is

```
LEType< LAM< LV<1>, LV<2>,
CALL<CALL<Plus,CALL<CALL<Multiplies,int>,LV<1> > >,LV<2> > > >::Type
```

The general idea is that

```
LEType< Translated_LambdaExp >::Type
```

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

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

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

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

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

5.6 Comparison to other lambda libraries

Here we briefly compare our approach to implementing lambda to that of the other major lambda libraries for C++: the Boost Lambda Library (BLL)[5] and FACT![13].[3]

Boost Lambda Library Whereas FC++ takes the minimalist approach, BLL takes the maximal approach. Practically every overloadable operator is supported within lambda expressions, and the library has special lambda-expression constructs which mimic the control constructs of C++ (like while loops, switches, exception handling, etc). The library also supports making references to variables, and side-effecting operators like increment and assignment. Lambda is implicit rather than explicit; a reference to a placeholder variables (like `_1`) turns an expression into a lambda on-the-fly.

BLL's approach makes sense given the "target audience"; the Boost libraries are designed for everyday C++ programmers. These are people who are familiar with C++ constructs, and who are hopefully C++-savvy enough to avoid most of the pitfalls of an expression-template lambda library. In contrast, FC++ is designed to support functional programming in the style of languages like Haskell. A number of our users come from other-language backgrounds, and aren't too familiar with the intricacies of C++. Thus FC++'s lambda is designed to present a simple interface with syntax and constructs familiar to functional programmers, and to shield users from C++-complexities as much as possible.

FACT! FACT!, like FC++, is designed to support pure functional programming constructs. Lambda expressions always perform capture "by value" and the resulting functions are typically effect-free. Like FC++, FACT! has an explicit lambda construct; the user can define his own names for placeholder variables, but conventionally names like `x` and `y` are used. FACT! defines few primitive control constructs in its lambda sublanguage (just `where` for if-then-else). Like BLL, however, FACT! overloads many C++ operators (like `+`) for use in lambda expressions. Thus FACT!'s interface is relatively simple and minimal, but lambda expressions are not as visually distinctive as they are in FC++.

6 Monads

Monads provide a useful way to structure programs in a pure functional language. Using monads, it is relatively straightforward to implement things like global

[3] The FACT! library, like FC++, includes features other than lambda, e.g. functions like `map()` and `foldl()` as well as data structures for lazy evaluation. BLL, on the other hand, is concerned only with lambda.

state, exceptions, I/O, and other concepts common to impure languages that are otherwise difficult to implement in pure functional languages[6, 14].

Supporting monads in FC++ is an interesting task for a number of reasons:

- Many interesting functional programs and libraries use monads; monad support in FC++ makes it easier to port these libraries to C++.
- Monads in Haskell take advantage of some of that language's most expressively powerful syntax and constructs, including *type classes*, *do-notation*, and *comprehensions*. Modelling these in C++ helps us better understand the relationship between the expressive power of these languages.
- Monads provide a way to factor out some cross-cutting concerns, so that local program changes can have global effects. (We discuss a few example applications that illustrate this.)

In the next subsection, we give a short introduction to monadic programming in Haskell. Next we discuss the relationship between *type classes* in Haskell and *concepts* in C++; understanding this relationship facilitates the discussion in the rest of this section. Then we discuss how we have implemented monads in FC++. We end with some example applications of monads.

6.1 Introduction to monads in Haskell

We briefly introduce a small portion of the Haskell programming language,[4] as its type system provides perhaps the most succinct and transparent way to understand the details of what a monad is. For the moment, know that a monad is a particular kind of data type, which supports two operations (named `unit` and `bind`) with certain signatures that obey certain properties. We shall return to the details after a short digression with Haskell.

In Haskell, the declaration `o :: T` says that object `o` has type `T`. Basic type names (like `Int`) start with capital letters. Lowercase letters are used for free type variables (parametric polymorphism – e.g. templates). The symbol `[T]` represents a list of `T` objects. The symbol `->` separates function arguments and results. The symbol `--` starts a comment. Here are a few examples.

```
x    :: Int                 -- x is an integer

add1 :: Int -> Int          -- add1 is a function from Int to Int

plus :: Int -> Int -> Int   -- plus takes two Ints and returns an Int
   -- (Or, equivalently, plus takes one Int, and returns a function
   --  which takes an Int and returns an Int.  Currying is built in.)

id   :: a -> a              -- id takes any type of object and returns
                            -- an object of the same type

map  :: (a -> b) -> [a] -> [b] -- map is a polymorphic function of two
```

[4] Haskell programmers will note that we are fudging some of the details of Haskell to simplify the discussion.

```
   -- arguments; it takes a function from type a to type b, and a
   -- list of objects of type a, and returns a list of b objects
```

Free type variables can be bounded by "type classes" (described shortly). For example, a function to sort a list requires that the type of elements in the list are comparable with the less-than operator. In Haskell we would say:

```
sort :: (Ord a) => [a] -> [a]
```

That is, `sort` is a function which takes a list of `a` objects and returns a list of `a` objects, subject to the constraint that the type `a` is a member of the `Ord` type class. Type class `Ord` in Haskell represents those types which support ordering operators like

```
class Ord a where
   ==  :: a -> a -> Bool
   <   :: a -> a -> Bool
   <=  :: a -> a -> Bool
      -- etc.
```

We say that a type `T` is an *instance* of type class `C` when the type supports the methods in the type class. For example, it is true that

```
instance Ord Int     -- Int is an instance of Ord
```

Given this overview of Haskell's types and type classes, we can now describe monads. A monad is a type class with two operations:

```
class Monad m where
   bind :: m a -> ( a -> m b ) -> m b
   unit :: a -> m a
```

In this case, instances of monads are not types, but rather they are "type constructors". These are like template classes in C++; an example is a list. In C++ `std::list` is not a type, but `std::list<int>` is. The same holds for Haskell; `[]` is not a type, but `[Int]` is. In the code describing the monad type class above, `m` is a type constructor.

It turns out that *lists* are instances of monads:

```
instance  Monad []  where
    bind m k       = concat (map k m)   -- don't worry about these
    unit x         = [x]                -- definitions yet
-- in the list monad
-- bind :: [a] -> ( a -> [b] ) -> [b]
-- unit :: a -> [a]
```

As another example, consider the `Maybe` type constructor. The type "`Maybe a`" represents a value which is either just an `a` object, or else nothing. In Haskell:

```
data  Maybe a  =  Nothing | Just a

-- Examples of variables
x  :: Maybe Int
```

```
x  =  Just 3

y  :: Maybe Int
y  =  Nothing
```

`Maybe` also forms a monad with this definition:

```
instance  Monad Maybe  where
    bind (Just x) k   =  k x            -- don't worry about
    bind Nothing  k   =  Nothing        -- these definitions
    unit x            =  Just x         -- yet
-- in the Maybe monad
-- bind :: Maybe a -> ( a -> Maybe b ) -> Maybe b
-- unit :: a -> Maybe a
```

A refinement of the `Monad` type class is `MonadWithZero`:

```
class (Monad m) => MonadWithZero m where
   zero :: m a
```

The `zero` element of a monad is a value which is in the monad regardless of what type was passed to the monad type constructor. For lists, the empty list (`[]`) is the `zero`. For `Maybe`, the `zero` is `Nothing`. Not all monads have zeroes, which is why `MonadWithZero` is a separate type class.

Monads with zeroes can be used in *comprehensions* with *guards*. Comprehensions are a special notation for expressing computations in a monad. Haskell supports comprehensions for the list monad; an example is

```
[ x+y | x <- [1,2,3], y <- [2,3], x<y ]
-- results in [3,4,5]
```

This list comprehension could be interpreted as "the list of values x plus y, for all x and y where x is selected from the list [1,2,3] and y is selected from the list [2,3], and where x is less than y". The desugared version of the Haskell code is:

```
-- (\z -> z+1) is Haskell lambda syntax:  (lambda(Z)[ Z %plus% 1 ])
-- backquotes are Haskell's infix syntax: (x `f` y == f x y)
[1,2,3] `bind` (\x ->
   [2,3]   `bind` (\y ->
      if not (x<y) then zero
                   else unit (x+y) ))
```

The translation from the comprehension notation to the desugared code is straightforward. Starting from the vertical bar and going to the right, the expressions of the form "`var <- exp`" turn into calls to `bind` and lambdas, and guards (boolean conditions) are transformed into if-then-else expressions which return the monad `zero` if the condition fails to hold. After all expressions to the right of the vertical bar have been processed, the expression to the left of the vertical bar gets `unit` called on it to lift the final computed value back into the monad.

6.2 Haskell's type classes and C++ template concepts

In the C++ literature, we sometimes speak of template *concepts*. A concept in C++ is a set of constraints which a type is required to meet in order to be used to instantiate a template. For example, in the implementation of the template function `std::find()`, there will undoubtedly be some code along the lines of

```
if( cur_element == target ) // ...
```

which compares two elements for equality using the equality operator. Thus, in order to call `std::find()` to find a value in a container, the element type must be `EqualityComparable`—that is, it must support the equality operator with the right semantics. We call `EqualityComparable` a *concept*, and we say that types (such as `int`) which meet the constraints *model* the concept. Concepts exist only implicitly in the C++ code (e.g. owing to the call to `operator==()` in the implementation), and often exist explicitly in documentation of the library. Some C++ libraries[9, 10] are devoted to "concept checking", these libraries check to see that the types used to instantiate a template do indeed model the required concepts (and issue a useful error message if not).

Haskell type classes are analogous to C++ concepts. However in Haskell they are reified; there are language constructs to define type classes and to declare which types are instances of those type classes. In C++, when a certain type models a certain concept (by meeting all of the appropriate constaints), it is merely happenstance (structural conformance); in Haskell, however, in addition to meeting the constraints of a type class interface, a type must be declared to be an instance of the concept (named conformance). "Concept checking" in Haskell is built into the language: type classes define concepts, instance declarations say which types model which concepts, and type bounds make explicit the constraints on any particular polymorphic function.

Understanding this analogy will make the FC++ implementation of monads more transparent. As we shall see, in the FC++ library, we spell out the concept requirements on monads, in order to make it easier for clients who write monads to ensure that they have provided all of the necessary functionality in the templates.

6.3 Comparing monads in FC++ to those in Haskell

Let us now illustrate monad definitions in FC++. As a first example, we shall look at `Maybe`. The `Maybe` template class and its associated entities are defined in Figure 3. `NOTHING` is the constant which represents an "empty" `Maybe`, and `just()` is a functoid which turns a value of type `T` into a "full" `Maybe<T>`. (`Maybe` is implemented using a `List` which holds either one or zero elements.)

Next we consider how to make `Maybe` a monad. Figure 4 describes the general monad concepts as specified in the FC++ documentation. A monad class must define the methods `unit` and `bind` (with the appropriate signatures); a class representing a monad with a zero must meet the above requirements as well as defining a `zero` element.

```
struct AUniqueTypeForNothing {};
AUniqueTypeForNothing NOTHING;

template <class T>
class Maybe {
   List<T> rep;
public:
   typedef T ElementType;

   Maybe( AUniqueTypeForNothing ) {}
   Maybe() {}                                    // Nothing constructor
   Maybe( const T& x ) : rep( cons(x,NIL) ) {}  // Just constructor

   bool is_nothing() const { return null(rep); }
   T value() const { return head(rep); }
};

struct XJust {
   template <class T> struct Sig : public FunType<T,Maybe<T> > {};

   template <class T>
   typename Sig<T>::ResultType
   operator()( const T& x ) const {
      return Maybe<T>( x );
   }
};
typedef Full1<XJust> Just;
Just just;
```

Fig. 3. The `Maybe` datatype in FC++

```
/*
   concept Monad {
      // full functoid with Sig   unit :: a -> m a
      typedef Unit;
      static Unit unit;
      // full functoid with Sig   bind :: m a -> ( a -> m b ) -> m b
      typedef Bind;
      static Bind bind;
   }
   concept MonadWithZero models Monad {
      // zero :: m a
      typedef Zero;           // a value type
      static Zero zero;
   }
*/
```

Fig. 4. Documentation of the monad concept requirements in FC++

```
struct MaybeM {
   typedef Just Unit;
   static Unit unit;

   struct XBind {
      template <class M, class K> struct Sig : public FunType<M,K,
         typename RT<K,typename M::ElementType>::ResultType> {};
      template <class M, class K>
      typename Sig<M,K>::ResultType
      operator()( const M& m, const K& k ) const {
         if( m.is_nothing() )
            return NOTHING;
         else
            return k( m.value() );
      }
   };
   typedef Full2<XBind> Bind;
   static Bind bind;

   typedef AUniqueTypeForNothing Zero;
   static Zero zero;
};
```

Fig. 5. Definition of the `Maybe` monad (`MaybeM`)

Figure 5 shows how we define the `Maybe` monad in FC++. Nested in `struct MaybeM` we define `unit`, `bind`, and `zero`, as well as `typedefs` for their types. This FC++ definition effectively corresponds to the definitions

```
instance Monad Maybe -- ...
instance MonadWithZero Maybe -- ...
```

in Haskell.

It should be noted here that the one major difference between monads in FC++ and monads in Haskell is that, in FC++, there is a distinction between the monad type constructor (e.g. `Maybe`) and the monad itself (e.g. `MaybeM`). We chose to make this distinction for reasons discussed next.

One advantage to separating the type constructor (`Maybe`) from the monad definition (`MaybeM`) is that, since the monad definition is itself a data type, it can be used as a type parameter to template functions. As a result, rather than supporting just list comprehensions (like Haskell does), in FC++ we support *comprehensions in an arbitrary monad*, by passing the monad as a template parameter to the comprehension. For example, the Haskell list comprehension

```
[ x+y | x <- [1,2,3], y <- [2,3], x<y ]
```

is written in FC++ as

```
compM<ListM>()[ X %plus% Y |
   X <= list_with(1,2,3), Y <= list_with(2,3), guard[ X %less% Y ] ]
```

Note how `ListM` is passed as an explicit template parameter to the `compM` function, which returns a comprehension for that monad. As a result, we can write

```
compM<MaybeM>()[ X %plus% Y | X <= just(2), Y <= just(3) ]
```

and perform a comprehension in the `Maybe` monad. Having a name apart from the data type constructor to serve as a handle for the monad definition (e.g. `ListM`, `MaybeM`) gives us a convenient way to parameterize monad operations. (The idea of generalizing comprehensions to arbitrary monads was originally discussed by Wadler[15].)

There is another advantage to separating the type constructor from the monad definition. Haskell type classes require algebraic data type constructors (not type aliases) to work. As a result, we cannot express the identity monad (a monad where `m a = a`) directly in Haskell. Instead we have to fake it by defining a new data type (which we have chosen to call `Identity`):

```
data Identity a = Ident a

instance Monad Identity where    --  m a = Identity a
   unit x   = x
   bind m k = k m
```

where values of type `a` are wrapped/unwrapped with the value constructor `Ident` to make them members of the type `Identity a`. In FC++, however, we can define the monad without also having to define a new data type to represent identities, as seen in Figure 6. The reason for the distinction is perhaps obvious. Haskell uses type inference, which means it must unambiguously be able to figure out which monad a particular data type is in. This type inference is not possible unless there is a one-to-one mapping between algebraic datatype constructors and monads. In FC++, on the other hand, the user passes the monad explicitly as a template parameter to constructs like `compM`. By requiring the user to be a little more explicit about the types, we gain a bit of expressive freedom (e.g. being able to do comprehensions in arbitrary monads).

6.4 Monads in FC++

The previous subsection introduced FC++ monads. Here we flesh out exactly what monad support FC++ provides.

FC++ provides functoids for the main monad operations. Specifically:

```
unitM<SomeMonad>()    // SomeMonad's "unit" functoid
bindM<SomeMonad>()    // SomeMonad's "bind" functoid
zeroM<SomeMonad>()    // SomeMonad's "zero" value
plusM<SomeMonad>()    // SomeMonad's "plus" functoid
bindM_<SomeMonad>()   // SomeMonad's "bind_" functoid
mapM<SomeMonad>()     // SomeMonad's "map" functoid
joinM<SomeMonad>()    // SomeMonad's "join" functoid
liftM<SomeMonad>()    // lifts a one-arg function into SomeMonad
liftM2<SomeMonad>()   // lifts a two-arg function into SomeMonad
```

```
// Nothing corresponding to Identity data type needed by Haskell
struct IdentityM {    // M a = a
   typedef Id Unit;
   static Unit unit;

   struct XBind {
      template <class M, class K> struct Sig : public FunType<M,K,
         typename RT<K,M>::ResultType> {};
      template <class M, class K>
      typename Sig<M,K>::ResultType
      operator()( const M& m, const K& k ) const {
         return k(m);
      }
   };
   typedef Full2<XBind> Bind;
   static Bind bind;
};
```

Fig. 6. Definition of the `IdentityM` monad

```
liftM3<SomeMonad>()   // lifts a three-arg function into SomeMonad
bind                  // "bind" (monad is inferred)
bind_                 // "bind_" (monad is inferred)
```

Many of these have not been previously mentioned; `plusM` is another function supported by some monads; `bindM_`, `mapM`, `joinM`, and the `liftM` functions are common monad operations which are defined in terms of `unitM` and `bindM`; `bind` and `bind_` are described more below.

FC++ supports comprehensions in arbitrary monads, using the general syntax:

```
compM<SomeMonad>()[ lambdaExp | thing, thing, ... thing ]
```

where `thing` is one of

- a gets expression of the form "`LV <= lambdaExp`" (Translates into a call to `bind`)
- a lambda expression (Translates into a call to `bind_`)
- a guard expression of the form "`guard[boolLambdaExp]`" (Translates into an if-then-else with `zero` if the test fails)

This is similar to the syntax used by Haskell's list comprehensions. FC++ also supports a construct similar to Haskell's *do-notation*:

```
doM[ thing, thing, ... thing ]
```

where each `thing` is as before, only `guard`s are no longer allowed. (The lack of a monad parameter to `doM` is discussed shortly.)

Clients can define monads by creating monad classes which model the monad concepts described in the previous subsection (`Monad` and `MonadWithZero`). There

is also a `MonadWithPlus` concept for monads which support `plus`. Additionally there is another concept called `InferrableMonad`, which may be modelled when there is a one-to-one correspondence between a datatype and a monad. In the case of `InferrableMonads`, FC++ (like Haskell) can automatically infer the monad based on the datatype in some cases; constructs like `doM` and the functoids `bind` and `bind_` do not need to have a monad passed an an explicit parameter—they infer it automatically.

The monad syntax is part of FC++'s lambda sublanguage. As with `lambda`, we strived for minimalism when implementing monads. The only new operator overloads are `operator|` and `operator<=`, and the only new syntax primitives are `compM`, `guard`, and `doM`. As with the rest of `lambda`, we provide `LEType` translations so that clients can name the result type of lambda expressions which use monads:

```
DOM             doM[]
GETS            LambdaVar <= value
GUARD           guard[]
COMP            compM<SomeMonad>()[]
```

As with the other portions of `lambda`, FC++ provides some custom error messages for common abuses of the monad constructs. We followed the same design principles discussed in Section 5 when implementing monads in FC++.

6.5 Monad examples

There are many example applications which use monads; here we discuss a small sample to give a feel for what monads are useful for.

Using MaybeM for exceptions One classic example of the utility of monads comes from the domain of exception handling. Suppose we have written some code which computes some values using some functions:

```
x = f(3);
y = g(x);
z = h(x,y);
return z;
```

(For the sake of argument, let's say that the functions `f`, `g`, and `h` take positive integers as arguments and return positive integers as results.) Now suppose that each of the functions above may fail for some reason. In a language with exceptions, we could throw exceptions in the case of failure. However in a language without an exception mechanism (like C or Haskell), we would typically be forced to represent failure using some sentinel value (`-1`, say), and then change the client code to

```
x = f(3);
if( x == -1 ) {
   return -1;
} else {
```

```
    y = g(x);
    if( y == -1 ) {
       return -1;
    } else {
       z = h(x,y);
       return z;
    }
}
```

This is painful because the "exception handling" part of the code clutters up the main line code. However, we can solve the problem much more simply by using the Maybe monad. Let the functions return values of type `Maybe<int>`, and let `NOTHING` represent failure. Now the client code can be written as just

```
compM<MaybeM>()[ Z | X <- f[3],
                     Y <- g[X],
                     Z <- h[X,Y] ]
```

The definitions of `unit` and `bind` in the `MaybeM` monad make the problem trivial; `NOTHING` values immediately propogate up through the end of the comprehension, whereas integers continue on through the computation as desired.

Using ListM for non-determinism Now imagine changing the problem above slightly; instead of the functions `f`, `g`, and `h` having the possibility of failure, suppose instead that they are non-deterministic. That is, suppose each function returns not a single integer, but rather a list of all possible integer results. Changing the original client code to deal with this change would likely be even uglier than the original change (which required all the tests for `-1`). However the change to the monadic version is trivial:

```
compM<ListM>()[ Z | X <- f[3],    -- Note ListM instead of MaybeM
                    Y <- g[X],
                    Z <- h[X,Y] ]
```

The result is a list of all the possible integer values for `Z` which satistfy the formulae.

A monadic evaluator Wadler [15] demonstrates the utility of monads in the context of writing an expression evaluator. Wadler gives an example of an interpreter for a tiny expression language, and shows how adding various kinds of functionality, such as error handling, counting the number of reduction operations performed, keeping an execution trace, etc. takes a bit of work. The evaluator is then rewritten using monads, and the various additions are revisited. In the monadic version, the changes necessary to effect each of the additions are much smaller and more local than the changes to the original (non-monadic) program. This example demonstrates the value of using monads to structure programs in order to localize the changes necessary to make a wide variety of additions throughout a program.

Monadic parser combinators Parsing is a domain which is especially well-suited to monads. In the Haskell community, "monadic parser combinators" are becoming the standard way to structure parsing libraries. As it turns out, parsers can be expressed as a monad: a typical representation type for parser monads is

```
Parser a = String -> Maybe ( a, String )   -- the monad "Parser"
```

That is, a parser is a function which takes a `String` and returns

- (if the parse succeeds) a pair containing the result of the parse and the remaining (yet unparsed) `String`, or
- (if the parse fails) `Nothing`.

Monadic parser *combinators* are functions which combine parsers to yield new parsers, typically in ways commonly found in the domain of parsing and grammars. For example, the parser combinator `many`:

```
many :: Parser a -> Parser [a]
```

implements Kleene star—for example, given a parser which parses a single digit called "`digit`", the parser "`many digit`" parses any number of digits. Monadic parser combinator libraries typically provide a number of basic parsers (e.g. `charP`, which parses any character and returns that character) and combinators (e.g. `plusP`, which takes two parsers and returns a new parser which tries to parse a string with the first parser, but if that fails, uses the second) to clients. The beauty of the monadic parser combinator approach is that it is easy for clients to define their own parsers and combinators for their specific needs. A good introductory paper on the topic of monadic parser combinators in Haskell is [3]; we implement the examples in that paper in one of the example files that comes with the FC++ library.

As we have seen in the previous examples, using monads often makes it easy to change some fundamental aspect of the behavior of the program. For example, if we have an ambiguous grammar (one for which some strings admit multiple parses), we can simply change the representation type for the parser like so:

```
Parser a = String -> [ ( a, String ) ]  -- uses List instead of Maybe
```

and redefine the monad operations (`unit`, `bind`, `zero`, and `plus`), and then parsers will return a list of every possible parse of the string. This is all possible without making any changes to existing client code.

One alternative approach to writing parsing libraries in C++ is that taken by the Boost Spirit Library[1]. Spirit uses expression templates to turn C++ into a `yacc`-like tool, where parsers can be expressed using syntax similar to the language grammar. For example, given the expression language

```
factor      ::= integer | group              // BNF
term        ::= factor (mulOp factor)*
expression  ::= term (addOp term)*
group       ::= '(' expression ')'
```

one can write a parser using Spirit as

```
factor      = integer | group;                // Spirit (C++)
term        = factor >> *(mulOp >> factor);
expression  = term >> *(addOp >> term);
group       = '(' >> expression >> ')';
```

which is almost just as readable as the grammar. Like `yacc`, Spirit has a way to associate semantic actions with each rule.

The results are similar with monadic parser combinators. Using an FC++ monadic parser combinator library, we can write

```
factor     = lambda(S)[ (integer %plusP% dereference[&group])[S] ];
term       = factor ^chainl1^ mulOp;
expression = term ^chainl1^ addOp;
group      = bracket( charP('('), expression, charP(')') );
```

to express the same parser. The above FC++ code creates parser functoids by using more primitive parsers and combining them with appropriate parser combinators like `chainl1`. (Note that, whereas Spirit's parser rules are effectively "by reference", FC++ functoids are "by value", which means we need to explicitly create indirection to break the recursion among these functoids. Hence the use of `lambda`, `dereference`, and the address-of operator.) This FC++ parser not only parses the string, but it also evaluates the arithmetic expression parsed. The semantics are built into the user-defined combinators like `addOp` and `chainl1`. For example,

```
addOp :: Parser (Int -> Int -> Int)
```

parses a symbol like `'-'` and returns the corresponding functoid (`minus`). Then,

```
chainl1 :: Parser a -> Parser (a -> a -> a ) -> Parser a
-- e.g.   p `chainl1` op
```

parses repeated applications of parser `p` , separated by applications of parser `op` (whose result is a left-assocative function, which is used to combine the results from the `p` parsers). Thus monadic parser combinator libraries allow one to express parsers at a level of abstraction comparable to tools like `yacc` or the Spirit library, but in a way in which users can define their own abstractions (like `chainl1`) for parsing and semantics, rather than just using the builtin ones (like Kleene star) supplied by the tool/library.

Lazy evaluation Previous versions of FC++ supported lazy evaluation in two main ways: first, via the lazy `List` class and the functions (like `map`) that use `Lists`, and second, via "thunks" (zero argument functoids, like `Fun0<T>`). Monads provide a new, more general mechanism to lazify computations. The datatype `ByNeed<T>` and its associated monad `ByNeedM` can be used to make a computation lazy. Additionally, the functoid `bLift` lazifies a functoid by lifting its result into the ByNeedM monad. For example, we can lazify

```
x = f(3);
y = g(x);
z = h(x,y);
```

by writing

```
compM<ByNeedM>()[ Z | X <= bLift[f] [3],
                      Y <= bLift[g] [X],
                      Z <= bLift[h] [X,Y] ]
```

The result is a `ByNeed<int>` value, which is a computation that will result in an `int` when "forced" by calling `bForce`. (Conversely, a constant can be turned into a by-need computation by calling `bDelay`.) Using values of type `ByNeed<T>` in lieu of type `T` ensures that lazy evaluation occurs: a computation is not performed until the value is demanded, and once a computation has been run to produce a value, the value is cached so that further applications of `bForce` get the cached value rather than re-running the computation.

In short, the datatype `ByNeed<T>` combines "thunks" with caching, and the `ByNeedM` monad makes syntax sugar like comprehensions available so that client code working with `ByNeed<T>` objects need not be concerned with all the "forcing" and "delaying" in the midst of the computation (the monad plumbing handles this).

Summary The examples given in this section give a sense of the kinds of applications for which monads are useful. Monads have a wide variety of utilities, which span varied domains (such as parsing and lists) and a number of cross-cutting concerns (like lazy evaluation and exception handling). Prior versions of FC++ implemented a few small monads, but they were extremely burdensome to express. The expressiveness afforded by the new FC++ syntactic sugar (like lambda and comprehensions) makes using monads in C++ a practicality for the first time.

7 Conclusions

We have given an overview of FC++ and described its new features in detail. Full functoids provide a general and reusable mechanism for adding features such as curryability, infix syntax, and lambda-awareness to every functoid. The lambda sublanguage is designed to minimize the problems common to all expression-template lambda libraries in C++. We have discussed the relationship between Haskell type classes and C++ template concepts in order to help describe how monads can be expressed in FC++. We have demonstrated a novel syntax for comprehensions which generalizes this construct to an arbitrary monad. Throughout FC++ and the lambda sublanguage, we have overloaded a select few operators to provide syntactic sugar for the library and we have used named functoids like `plus` to express the actual operations of C++ operators.

References

1. de Guzman, Joel, et al. The Boost Spirit Library. Available at `http://www.boost.org/libs/spirit/index.html`

2. *Haskell 98 Language Report.* Available online at `http://www.haskell.org/onlinereport/`
3. Hutton Graham and Meijer Erik. "Monadic parsing in Haskell" *Journal of Functional Programming*, 8(4):437-444, Cambridge University Press, July 1998.
4. *ISO/IEC 14882: Programming Languages – C++.* ANSI, 1998.
5. Järvi, Jaakko and Powell, Gary. The Boost Lambda Library. Available at `http://boost.org/libs/lambda/doc/index.html`
6. Jones, Simon Peyton and Wadler, Philip. "Imperative functional programming," *20th Symposium on Principles of Programming Languages*, ACM Press, Charlotte, North Carolina, January 1993.
7. McNamara, Brian and Smaragdakis, Yannis. "FC++: Functional Programming in C++", *Proc. International Conference on Functional Programming (ICFP)*, Montreal, Canada, September 2000.
8. McNamara, Brian and Smaragdakis, Yannis. "Functional Programming with the FC++ library" *Journal of Functional Programming*, to appear.
9. McNamara, Brian and Smaragdakis, Yannis. "Static Interfaces in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at `http://www.oonumerics.org/tmpw00/`
10. Siek, Jeremy and Lumsdaine, Andrew. "Concept Checking: Binding Parametric Polymorphism in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at `http://www.oonumerics.org/tmpw00/`
11. Y. Smaragdakis and B. McNamara, "FC++: Functional Tools for Object-Oriented Tasks" *Software Practice and Experience*, August 2002.
12. A. Stepanov and M. Lee, "The Standard Template Library", 1995. Incorporated in ANSI/ISO Committee C++ Standard.
13. Striegnitz, Jörg. "FACT! The Functional Side of C++," Available at `http://www.fz-juelich.de/zam/FACT`
14. Wadler, Philip. "Comprehending monads," *Mathematical Structures in Computer Science*, Special issue of selected papers from 6th Conference on Lisp and Functional Programming, 2:461-493, 1992.
15. Wadler, Philip. "Monads for functional programming." J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.