# Multiparadigm programming: Novel devices for implementing functional and logic programming constructs in C++

A Thesis
Presented to
The Academic Faculty

by

**Brian McNamara**

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
July 12, 2004

# TABLE OF CONTENTS

# SUMMARY

Constructs for functional and logic programming can be smoothly integrated into an existing object-oriented language. We demonstrate this in the context of C++ (a statically-typed object-oriented language with effects and parametric polymorphism) via two libraries: FC++ and LC++. FC++ is a library for functional programming in C++; FC++ supports higher-order polymorphic functions, lazy lists, and a small lambda language; it also contains a large library of useful functions, datatypes, combinators, and monads. LC++ is a library for logic programming in C++; LC++ provides the same general functionality as Prolog, including the ability to return query results lazily (one at a time). Both libraries are embedded in C++ so that they share C++'s static type system, and the library interfaces provide straightforward ways for code from within one paradigm to "call out" to another.

Our work describes the techniques used to implement these libraries in C++ and shows that the resulting multiparadigm language has useful applications in real-world domains. We also describe how many of the implementation techniques can be generalized from C++ and applied to other programming languages to yield similar results.

# CHAPTER I

# INTRODUCTION

In this chapter we provide an overview of our research work. We motivate the topic of multi-paradigm programming, state our thesis, and provide a high-level description of our contributions to the field—in terms of our research artifacts (the FC++ and LC++ libraries) and the "conceptual contributions" of our work. This chapter closes with a short summary describing each of the remaining chapters of the dissertation.

## *1.1 Motivation*

Multi-paradigm programming languages have been a topic of research for decades. The basic appeal is clear: offering the programmer a choice of paradigms enables her to choose the one best suited to the problem domain.

Recently there have been a number of good examples of languages and systems which combine functional and logic programming (e.g., [15, 20, 27, 53, 64]), as well as mature implementations which add functional features to object-oriented languages (e.g., [35, 48, 56, 60]), and few examples that extend object-oriented languages with logic programming (e.g. [11, 10, 19]). But it is rare to see programming systems which combine all three paradigms.

There appear to be two main reasons for the paucity of multiparadigm languages. The first stems from the range of "expression" that must be covered. At one extreme, logic programming provides a purely declarative specification for computing a result; at the other extreme, imperative OO code describes an algorithm for implementing a computation, filled with details about which variables get side-effected when. (Consider, e.g., how different `sort()` looks when coded in C versus Prolog.) As a result, the mere task of creating a language syntax which is capable of representing these different expressive modes (and everything in between) is a challenge.

The second reason, which seems to be more difficult to overcome than the first, is that even if a language can cover the range of expression, so that parts of programs can be written *within* each paradigm, there is still the issue of communicating *between* paradigms. There is an interface mismatch at the borders between paradigms—different paradigms treat fundamental issues, such as effects, calling conventions, data representation, and control flow differently. There is a great challenge in providing smooth interfaces which enable code from one paradigm to "call out" to code in another, so that the best-match paradigm can be used by the programmer to solve each individual portion of the problem at hand.

We show that a smooth integration of the paradigms can be achieved in C++. We do so with two libraries: FC++ for functional programming, and LC++ for logic programming.

## *1.2 Thesis*

Constructs for functional and logic programming can be smoothly integrated into an existing object-oriented language. We demonstrate this in the context of C++, and show that the resulting multiparadigm language has useful applications in real-world domains.

## *1.3 Contributions*

The contributions of our work can be divided into two main categories. The first category compares our work in C++ to other work supporting multiparadigm programming. The second category describes the "reusable lessons" of our work—that is, the ideas which can be reapplied in the context of other OO programming languages to extend them with multiparadigm features.

### 1.3.1 Comparing FC++ and LC++ to other multiparadigm work

Our libraries compare favorably with prior work in multiparadigm systems. This is true in terms of both practicality (e.g. run-time performance and proven usefulness) and language design (e.g. concision). We highlight the important points here, which are explored more fully in later sections.

- FC++ has an efficient implementation thanks to a number of optimizations. The optimization techniques are described and performance is quantified in Sections 2.5.1 & 2.5.2.

FC++ is significantly faster than the prior state of the art (Läufer's C++ framework for functions[45]).

- FC++ has been shown to be a valuable tool for implementing object-oriented designs, as described in Section 2.4. Using FC++, the implementations of some design patterns[23] are simpler, more efficient, or more typesafe than their conventional counterparts.

- FC++ provides a useful infrastructure for building other libraries, as evidenced by our "customers" [12, 46, 16]. FC++ has also influenced the development of a number of Boost libraries—a group of widely-distributed, peer-reviewed, portable C++ source libraries which are likely to be incorporated into future versions of the C++ standard library.

- FC++ provides a natural syntax for doing functional programming. Indeed, we have used C++'s extensibility features to provide various kinds of syntactic sugar for features like infix function call, lambda, and monad comprehensions. This makes programming in FC++ look and feel more like programming in a functional language (e.g. Haskell). When compared to other functional C++ libraries (e.g. [35, 60]), FC++ is often more concise and expressive.

- Similarly, LC++ provides an interface that looks very similar to that of a logic language (e.g. Prolog). The declarative specification of logic programming code is often cited as one of the key assets of the paradigm, thus preserving this declarative style is important to providing a smooth integration. In this respect, LC++ compares favorably to other attempts to add logic programming to object-oriented languages (e.g. [10, 19]).

- Both libraries provide a convenient interface between paradigms. FC++, LC++, and C++ all share the same type system and object representation, and the interfaces provide a good way for control to flow across paradigms.

- Finally, the libraries include a number of domain-specific static analyses. This enables

certain types of errors to be detected at compile-time, and the C++ compiler can emit useful diagnostics.

To reiterate the last four points above, we have implemented the libraries in a way which preserves the syntax of the functional and logic programming domains, integrates their control flow and type systems with the base language (C++), and provides static analyses specific to the domains. Thus, although we have implemented FC++ and LC++ as C++ *libraries*, it is reasonable to think of them as *domain-specific embedded languages* for functional and logic programming.

### 1.3.2 Reusable lessons of our work

In addition to the concrete contributions of our actual library implementation, many of the ideas in our work also generalize. That is, we describe "novel devices for implementing functional or logic programming constructs using C++-like mechanisms". This information is potentially valuable both to researchers using C++, who want to be able to replicate aspects of our work, as well as those working in other OO languages who want to extend them with functional or logic programming features. Put another way, we answer the question "Where is the magic?" and show how we have succeeded in adding certain features to C++ where others have previously tried and failed. Some of the most important of these ideas are briefly listed here:

- We show how to implement a type system for higher-order polymorphic functions, using C++-style type inference and template computation.
- We show a reusable mechanism to implement currying, which is based on operator overloading and template partial specialization.
- We show how to implement a general mechanism for converting prefix functions into infix ones, using operator overloading.
- We demonstrate how to make function objects exhibit subtype polymorphism, so they can participate in OO type hierarchies.

- We show that lazy lists provide a smooth way to interface logic code with OO code. Notably, returning logic query results as a lazy list encapsulates the fact that logic programming is implemented in continuation passing style.

- We show how to do static analyses specific to the domains of the added programming paradigms, utilizing the Turing-complete meta-programming capabilities of C++.

The whole of Chapter 4 is devoted to the topic of reusable lessons.

## *1.4 Roadmap*

The rest of the dissertation is organized as follows.

Chapter 2 describes the FC++ library in great detail. The library's interface is explained and the major implementation issues are discussed. There is also a discussion of applications of the library and comparisons to directly related work. FC++ is the largest and most interesting piece of our work, and this chapter is by far the largest in the dissertation.

Chapter 3 describes the LC++ library. As with FC++, the LC++ chapter describes the library interface, key implementation issues, applications, and relations to closely related work.

Chapter 4 describes the reusable lessons of our work. Some of these lessons are only useful in C++, but many of them generalize to other programming languages, and we give a number of illustrations of how our techniques can be implemented using features found in other languages.

Chapter 5 discusses related work. Whereas both Chapters 2 & 3 make direct comparisons to closely related work, Chapter 5 shows the "bigger picture" of how our work is related to a number of multiparadigm languages and systems.

Chapter 6 summarizes the contributions of our work and suggests possible avenues of further exploration.

# CHAPTER II

# FC++

This chapter describes FC++, a library for functional programming in C++. We describe the multitude of functional programming features supported by the library, both in terms of their C++ implementations and their interfaces with the rest of the language. We also demonstrate a number of applications and application domains of the library. Finally we discuss pragmatic issues, including considerations of run-time efficiency and the overall expressiveness of the library.

## *2.1 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 on FC++ fills. 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 [56] 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++ [42, 45]. C++ also allows users to define a familiar syntax for function-classes, by overloading the function application operator, "()". Additionally one can declare methods so that they are prevented from modifying their arguments; this property is enforced statically by C++ compilers. Finally, using the C++ inheritance capabilities and dynamic dispatch mechanism, one can define variables that range over all functions with the same type signature. In this way, a C++ user can "hijack" the underlying language mechanisms to provide a functional programming model.

All of the above techniques are well-known and have been used before. In fact, several researchers in the recent past (e.g., [60, 42, 35, 45, 52]) have (re)discovered that C++ can be used for functional programming. Nevertheless, all of the above approaches, as well as that of the C++ Standard Library, suffer from one of two drawbacks:

- High complexity when polymorphic functions are used: Polymorphic functions may need to be explicitly turned into monomorphic instances before they can be used. This causes the implementation to become very complex. Läufer observed in [45]: "...the type information required in more complex applications of the framework is likely to get out of hand, especially when higher numbers of arguments are involved."

- Lack of expressiveness: In order to represent polymorphic functions, one can use C++ function templates. This approach does not suffer from high complexity of parameterization, because the type parameters do not need to be specified explicitly whenever a polymorphic function is used. Unfortunately, function templates cannot be passed as arguments to other function templates. Thus, using C++ function templates, polymorphic functions cannot take other polymorphic functions as arguments. This is evident in the C++ Standard Library, where "higher order" polymorphic operators like `compose1`, `bind1st`, etc. are not "functions" inside the Standard Library framework and, hence, cannot be passed as arguments to themselves or other operators.

Our work addresses both of the above problems. Contrary to prior belief (see Läufer [45], who also quotes personal communication with Dami) no modification to the language or

the compiler is needed. Instead, we are relying on an innovative use of C++ type inference. Effectively, our framework maintains its own type system, in which polymorphic functions can be specified and other polymorphic functions can recognize them as such.

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

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

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

- First, we define a subtyping policy for functions of FC++, thus supporting subtype polymorphism. The default policy is what one would expect: a function A is a subtype of function B, iff A and B have the same number of arguments, all arguments of B are subtypes of the corresponding arguments of A, and the return value of A is a subtype of the return value of B. (Using OO typing terminology, we say that our policy is covariant with respect to return types and contravariant with respect to argument types.) Subtype substitutability is guaranteed; a function `Animal* -> Car*` can be used where a function `Dog* -> Vehicle*` is expected.

- Second, FC++ provides a number of useful syntax sugars. A set of reusable combinators support automatic currying; curryable functions can be called with a subset of the arguments they expect, binding those values and resulting in a new function

that expects the remainder of the arguments. Functions can be called using infix syntax, similar to the `` `function` `` syntax in Haskell. Anonymous functions can be created with a library that simulates *lambda*, and monads can be manipulated using *comprehensions*.

- Third, FC++ has a high level of technical maturity. For instance, compared to Läufer's approach, we achieve an equally safe but more efficient implementation of the basic framework for higher order functions. Our original implementation was already 4 to 8 times faster than Läufer's; we describe a number of optimizations we have since applied to the library. Experiments demonstrate that the performance has increased by almost an order of magnitude compared to our old implementation.

Additionally, FC++ builds significant functionality on top of the basic framework. We export two fairly mature reference-counting "pointer" classes to library users, so that use of C++ pointers can be completely eliminated at the user level. We provide facilities for lazy evaluation, via both a lazy list data type and a "by need" monad. 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.

## *2.2 Description of basic library features*

In this section we discuss the majority of the features of the library, in terms of both interface and implementation. In Section 2.3, we delve into some of the more complicated and subtle features of the library.

We briefly define one term here at the outset. In FC++, a *functoid* is our chosen representation for function objects. The different kinds of functoids will be described in detail shortly (Sections 2.2.2, 2.2.3, and 2.2.5); until then, it is fine to simply equate the terms "functoid" and "function".

### 2.2.1 Library introduction

We begin with a brief overview of how the FC++ library is used. Figure 1 will serve as a running example to illustrate many of the main features of the library.

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

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

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

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

The FC++ library contains more than 50 useful functions from the Haskell Standard Prelude [40]. The prior examples have already used familiar functions like `map()` and `filter()`; FC++ include dozens of such general functions, including `take()`, which selects the first N elements of a list, and `foldl1()` which left-accumulates all of the values in a list using a given function.

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

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

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

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

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

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

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

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

// (G) Infix syntax
assert( (3 `plus` 2) == 5 );
assert( (plus `foldr` 3)(li)==18 );
```

**Figure 1:** Some examples of what FC++ can do

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

FC++ functoids of two or more arguments can be called using a special "infix syntax". Part (G) of Figure 1 shows how functoids can be used like infix operators by surrounding their names in carets. When functoids of more than two arguments are used infix, the remaining arguments are curried.

The example in this subsection gives an overview of the main features of the library. Two notable features which are not demonstrated in this example are FC++'s "lambda" (a sub-library for creating anonymous functions on-the-fly), and the FC++ facilities for monads (including syntax sugars like monadic comprehensions). These features will be described later, in Section 2.3.

### 2.2.2 Direct functoids

FC++ represents polymorphic functions with "direct functoids". We will begin by describing the special case of monomorphic direct functoids, because they are simple and serve as a good introduction for readers not familiar with C++. Then we shall move on to describe polymorphic direct functoids. Later, in Section 2.2.4, we illustrate how FC++ simplifies the use of polymorphic functions in C++ as compared to other approaches.

#### *2.2.2.1 Monomorphic direct functoids*

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

```
struct TwoTimes {
```

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

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

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

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

```
int -> int
```

(the usual functional notation is used to represent signatures). As far as the C++ language is concerned, however, the *types* of these objects are `TwoTimes` and `Inc.` (Note our convention of using an upper-case first letter for class names, and a lower-case first letter for object 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 2.2.2.2). Thus, we would like to encapsulate some representation of the type signature of `TwoTimes` in its definition. The details of this representation will be filled in Section 2.2.2.2, but for now it suffices to say that each direct functoid has a member called `Sig` (e.g., `TwoTimes::Sig`) that represents its type signature. `Sig` is not defined explicitly by the authors of monomorphic direct functoids—instead it is inherited from classes that hide all the details of the type representation. For instance, `TwoTimes` would be defined as:

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

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

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

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

#### *2.2.2.2 Polymorphic direct functoids*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The above definition of `Tail` is an example of a polymorphic direct functoid. In general, a direct functoid is a class with a member `operator()` (possibly a template operator), and a template member class `Sig` that can be used to compute the return type of the functoid given its argument types. Thus the convention is that the `Sig` class template takes the types of the arguments of the `operator()` as template parameters. As described in Section 2.2.2.1, for monomorphic direct functoids, the member class `Sig` is hidden inside the `CFunType` classes, but in essence it is just a template computing a constant compile-time function (i.e., returning the same result for each instantiation).

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

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

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

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

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

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

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

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

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

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

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

In Section 2.2.4 we demonstrate how using direct functoids greatly simplifies the task of programming with polymorphic functions in C++, by drawing a comparison with the alternatives. One of the alternatives involves indirect functoids, so we discuss them next.

### 2.2.3 Indirect functoids

Direct functoids are not first class entities in the C++ language. Most notably, one cannot define a (run-time) variable ranging over all direct functoids with the same signature. We can overcome this by using a C++ subtype hierarchy with a common root for all functoids with the same signature and declaring the function application operator, "()", to be `virtual` (i.e., dynamically dispatched). In this way, the appropriate code is called based on the run-time type of the functoid to which a variable refers. On the other hand, to enable dynamic dispatch, the user needs to refer to functions indirectly (through pointers). Because memory management (allocation and deallocation) becomes an issue when pointers are used, we

encapsulate references to function objects using a reference counting mechanism. This mechanism is completely transparent to users of FC++: from the perspective of the user, function objects can be passed around by value. It is worth noting that our encapsulation of these pointers inside indirect functoids prevents the creation of cyclical data structures,[1] 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

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

For instance, one-argument indirect functoids with signature

```
A -> R
```

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

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

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

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

(Note: The ellipsis (...) symbol in the above code is used to denote that parts of the implementation have been omitted for brevity. These parts implement our subtype polymorphism policy and will be discussed in Section 2.2.8. The `Ref` class template implements our reference-counted "pointers" and will be discussed in Section 2.5.2.3. For this internal use, any simple reference counting mechanism would be sufficient.)

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

```
A -> R.
```

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

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

In practice, however, this definition would be rare because it would require that `Inc` be defined as a monomorphic function. As we have seen in Section 2.2.2.2, the most common and convenient representation of functions is that of polymorphic direct functoids.

Monomorphic direct functoids can be explicitly converted to indirect functoids, using the operation `makeFunN` (provided by FC++). For instance, consider direct functoids `TwoTimes` and `Inc` from Section 2.2.2.1 (the definition of `Inc` was not shown). The following example is illustrative:

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

In fact, the calls to `makeFunN` can be elided—we show them here to help explain the transformation, however a carefully designed implicit conversion function template in the `FunN` classes makes the library functoids "smart" enough to let the transformation happen implicitly. Polymorphic direct functoids can also be assigned to indirect functoids, by first selecting a monomorphic instance. This conversion was illustrated back in Figure 1 in Section 2.2.1 as

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

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

It should be noted here that our indirect functoids are very similar to the functoids presented in Läufer's work [45] and the functors presented in Chapter 5 of Alexandrescu's book [2]. Indeed, the only difference is in the wrapper classes, `FunN<A1,A2,...,AN,R>`. Whereas we use a reference counting mechanism, both Läufer's and Alexandrescu's implementations allowed no aliasing: different instances of `FunN<A1,A2...,AN,R>` had to refer to different instances of `FunNImpl<A1,A2,...,AN,R>`. To maintain this property, objects had to be copied every time they were about to be aliased. This copying results in an implementation that is significantly slower than ours—in [48], we demonstrated that our original implementation was four to eight times faster than Läufer's. (Our newer implementation, which uses intrusive reference counting for indirect functoids, is even faster, as we shall see in Section 2.5.2.) Another difference from other implementations is that our indirect functoids will rarely be defined explicitly by clients of FC++. Instead, they will commonly only be produced by fixing the type signature of a direct functoid.

### 2.2.4 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.

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

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

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

**Figure 2:** Polymorphic functions as templates over indirect functoids

```
a -> a.
```

(In Haskell one would say

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

It is possible to specify this type bound in C++, albeit in a roundabout way—see the short discussion on type constraints in Section 2.5.3 for details.)

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

(For brevity, the implementation of `operator()` in `Map` is omitted. The implementation is similar in all the alternatives we will examine.)

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

The direct functoid implementation is only a little more complex than the indirect functoid implementation. The complexity is due to the definition of `Sig`. `Sig` encodes the

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

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

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

**Figure 3:** Polymorphic functions as direct functoids

type signature of the direct functoid in a form that can be utilized by all other higher order functions in our framework. According to the convention of our framework, `Sig` has to be a class template over the types of the arguments of `Map`. Recall also that `FunType` is just a simple template for creating function signatures.

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

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

To see why the direct functoid specification is beneficial, consider the uses of `map` and

`twoTimes`. In Haskell, we can say

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

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

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

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

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

Clearly this alternative would have made every expression terribly burdensome, introducing much redundancy (`int` appears 5 times in the previous example, when it could be inferred everywhere from the value `1`). Note that this expression has a single function application. Using more complex expressions or higher-order functions makes matters even worse. For instance, using the `compose` functoid mentioned in Section 2.2.2.2, we can create a list of multiples of four by writing

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

The same using indirect functoids would be written as

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

We have found even the simplest realistic examples to be very tedious to encode using templates over indirect functoids (or, equivalently, Läufer's framework [45]).

In short, direct functoids allow us to simplify the *use* of polymorphic functions substantially, with only little extra complexity in the functoid *definition*. The idiom of using template member functions coordinated with the nested template class `Sig` to maintain our

own type system is the linchpin in our framework for supporting higher-order parametrically polymorphic functions.

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

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

This is the most widespread C++ idiom for approximating polymorphic functions (e.g., [52][59]). C++ type inference is still used in this case. Unfortunately, as noted earlier, C++ function templates cannot be passed as arguments to other functions (or function templates). That is, function templates can be used to express polymorphic functions but these cannot take other function templates as arguments. Thus, this idiom is not expressive enough. For instance, our example where `twoTimes` is passed as an argument to map is not realizable if `twoTimes` is implemented as a function template.

The closest approximation of such functionality before FC++ [59] was with the use of a hybrid of class templates, like in Figure 2, and function templates. In the hybrid case, each function has two representations: one using a template class (so that the function can be passed to other functions) and one using a function template (so that C++ type inference can be used when arguments are passed to the function). The C++ Standard Library [59] uses this hybrid approach for some polymorphic, higher-order functions. This alternative is quite inconvenient because class templates still need to be turned into monomorphic function instances explicitly (e.g., one would write `TwoTimes<int>` instead of `twoTimes` in the examples above), and because two separate representations need to be maintained for each function. The user will have to remember which representation to use when the function is called and which to use when the function is passed as an argument.

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

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

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

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

which will resolve to a value of type `pair<int,double>`. This is rank-2 polymorphism [41]; inside the call to `foo()`, `f` is used polymorphically. Neither of the other approaches enable functions like `foo()` to be defined. That is, FC++ was the first C++ library with this rank-2 polymorphism capability.

### 2.2.5 Full functoids

The definitions of direct functoids given in the previous subsections are actually what we call "basic direct functoids" in FC++. However, a number of features of functoids (such as currying and infix syntax, which we saw in Section 2.2.1, and lambda-awareness, which will shall describe in Section 2.3.1) 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

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

(as we saw in Figure 3) 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.

Any basic functoid can be promoted into a full functoid 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. Expressing "fullness" as a general combinator makes it trivial to add a large set of common useful features to every functoid. We describe two of those specific features (infix syntax and currying) next. Later, in Section 2.3.1, we describe how full functoids interact with FC++'s lambda.

### 2.2.6 Infix syntax

The first feature provided by full functoids is infix syntax. Any full functoid of at least two arguments can be effectively used as an infix operator:

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

Full functoids enable this infix syntax by overloading `operator^`. The details of the implementation of this feature are relatively straightforward, and are described in Section 4.1.3. This syntax was inspired by a similar feature in Haskell. Many function names (like `plus`) are more readable as infix than as prefix.

There is one notable limitation to this feature. Since FC++ infix is implemented by overloading `operator^`, we cannot change the precedence or associativity of functoids used as infix operators. These parsing aspects are fixed by the C++ language, and thus an expression like

```
3 ^plus^ 4 ^multiplies^ 5
```

means

```
multiplies( plus(3,4), 5 )
```

(because `operator^` is left-associative and all infix functoids have equal precedence). Of course, the user can always override the defaults by using explicit parentheses, as in

```
3 ^plus^ (4 ^multiplies^ 5)
```

### 2.2.7 Currying

FC++ supports currying of functoid arguments. Currying is another feature implemented in the `FullN` combinators. All of the functoids exported by the library are full functoids,

and thus all are curryable[2]. Here is an example:

```
struct XPlus { ... } xplus;

Full2<XPlus> plus;

...

xplus(2,3);   // xplus requires both args,

plus(2,3);    // whereas plus is curryable

plus(2);      // and can be called in any

plus(2,_);    // of these ways.

plus(_,3);
```

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

The `FullN` classes, which implement the currying functionality, take advantage of two C++ language features: overloading and partial specialization. In C++, functions can be overloaded (ad hoc) based on the number and types of their arguments. Similarly, templates can be specialized (ad hoc) for certain argument types. So class `Full2` has four separate `Sig` specializations (`Sig<X,Y>`, `Sig<X>`, `Sig<X,AutoCurryType>`, `Sig<AutoCurryType,Y>`—where `X` and `Y` are template parameters (free type variables)) and four separate overloaded implementations, which correspond to the four different ways a two-argument functoid may be called:

```
f(x,y)

f(x)

f(x,_)

f(_,y)
```

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

The `FullN` classes enable functions to be curried either implicitly (by only supplying some of the leading arguments) or explicitly (using underscores as placeholders for arguments to be curried). The FC++ library also contains more verbose functions for currying, named `bindMofN`. For example `plus(_,3)` and `bind2of2(plus,3)` are equivalent: both bind the second argument (of `plus`'s two arguments) to the value 3, and return a new function of one argument. These explicit binders have an additional capabilitity: if one binds all of a function's arguments, a zero-argument functoid (a thunk) is returned. For example, `bind1and2of2(plus,2,3)` returns a zero-argument functoid, which yields the value 5 when called. The `thunkN` functions can also be used to enact the same behavior: `thunk2(plus,2,3)` yields the same result as `bind1and2of2(plus,2,3)`.

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

Currying is a useful feature in functional programming, as it enables programmers to easily specialize and adapt general functions to fit specific needs, by fixing some subset of the functions' arguments. Whereas some functional languages have built-in support for currying, in C++, it must be supplied via a library. FC++ is the only C++ library which supports implicit currying, by exploiting the existing features of the C++ language to make it appear to clients as though currying is a built-in language feature that works automatically. Other C++ libraries have other levels of support for currying. For example, the Boost Lambda Library [35] has a mechanism similar to FC++'s placeholder currying, where explicit placeholders can be used for curried arguments. Both the STL [59] and Alexandrescu's "functors" [2] do support "binding" one of the arguments of a function to a specific value, however in each of these libraries, the binding must be done explicitly

(with a call to a binding function like FC++'s `bindMofN`), and the binding only works on monomorphic functions—a severe limitation.

### 2.2.8 Subtype polymorphism

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

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

```
Fun1<Ref<Animal>, Ref<Car> >  fa;
Fun1<Ref<Dog>, Ref<Vehicle> > fb = fa;
  // 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 `FunN` of Section 2.2.3. For instance, the definition of `Fun1` has the form:

```
template <class Arg1, class Result>
```

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

Without getting into all the details of the implementation, the key idea is to define a template implicit conversion operator from `Fun1<A1s,Rs>` to `Fun1<Arg1,Result>`, if and only if `A1s` is a supertype of `Arg1` and `Rs` is a subtype of `Result`. The latter check is the responsibility of direct functoid `convert1` (not shown). In particular, `convert1` defines code that will explicitly test (at compile time) to ensure that an `Arg1` is a subtype of `A1s` and that `Rs` is a subtype of `Result`. In this way, the implicit conversion of functoids will fail if and only if either of the above two conversions fails. Since the operator is templatized, it can be used for any types `A1s` and `Rs`.

We should note that, although the above technique is correct and sufficient for the majority of conversions, there are some slight problems. First, C++ has inherited from C some unsafe conversions between native types (e.g., implicit conversions from floating point numbers to integers or characters are legal). There is no good way to address this problem (which was inherited from C despite the intentions of the C++ language designer; see [61] 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.

### 2.2.9 C++ interface

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

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

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

```
// before(thunk,f)(args)  ==  { 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( thunk1( 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++ functoids are designed to work smoothly with the C++ Standard Template Library (STL). Monomorphic FC++ functoids conform to the requirements for what the STL

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

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

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

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

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

**Figure 4:** FC++ and native C++ functions

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

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

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

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

is sufficient. On the other side of the coin, FC++ provides combinators to promote STL "adaptable functions" into (monomorphic) FC++ functoids so that functions from from STL can be used inside FC++. Finally, FC++ `Lists` are designed to fit into the STL framework for data structures. Figure 5 shows that the `List` class supports iterators of the

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

**Figure 5:** FC++ and STL

STL style. This makes converting both to and from STL data structures easy, and enables `List`s to be passed to (non-mutating) STL algorithms.

Another interface that is somewhat common in legacy C/C++ code is the use of types like `void (*)(void*)` [3] as a sort of generic interface for "callback functions". It is not possible to automatically convert an FC++ functoid into such a function pointer, but it is straightforward to hand-code an adapter function: just write a normal C++ function with the proper signature that forwards the call to the appropriate functoid. In this way, functoids can be used with such legacy libraries. (On the other hand, if callbacks are desired but there is no need to interface with a legacy callback library, then FC++ itself can serve as a complete callback library, with indirect functoids serving as type-safe interfaces to arbitarary functions. See Section 2.4.1 as well as examples on our web site ([21]) for more about using FC++ as a callback library.)

The smoothness of the interfaces between FC++ and STL and also between FC++ and the object-oriented portions of C++ demonstrate an important point. FC++ does not merely inject into C++ a purely functional sublanguage that should be used exclusive from the rest of the language. Rather, FC++ embeds this functionality inside the language in a way that makes it straightforward to utilize the extra functional support on top of the existing imperative/object-oriented programming platform that C++ provides. Section 2.4 illustrates the value of combining the paradigms within programs by showing how the implementations of common OO design patterns are improved by adding FC++.

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

## 2.3 Description of advanced library features

In this section we discuss FC++ features that require more advanced implementation strategies in C++: lambda, monads, and static analyses. We also discuss the subtleties of our implementation of lists.

### 2.3.1 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[59] and its "binders", 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[35] and FACT![60] enable the creation of arbitrary lambdas by using expression templates.

#### *2.3.1.1 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.

#### *2.3.1.2 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" );
  ```

  Note the third argument to `for_each()`, the expression

```
cout << _1 << "\n"
```

This expression creates an anonymous function to print a value. The placeholder `_1` serves as a lambda variable; thus the expression effectively means

```
// pseudo-C++
lambda(x) { return cout << x << "\n"; }
```

As a result, the `for_each()` call prints 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`)[4] 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.

#### *2.3.1.3 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. While it is impossible to prevent

[4] 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.

users from making any mistakes, we have designed 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 the 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 capturing the 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*[50, 57] 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. Section 2.3.3 discusses this in more detail.

- **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.

#### *2.3.1.4 Lambda in FC++*

We now describe what it looks like to do lambda in FC++. Figure 6 shows some examples of lambda. There are a few points which deserve further attention.

Inside lambda, one uses square brackets instead of round ones for postfix functional call. (This works thanks to the lambda-awareness of full functoids, mentioned in Section 2.2.5.) Similarly, the percent sign is used instead of the caret 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
```

```
// 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 (see text)

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

**Figure 6:** Lambda in FC++

Unfortunately, C++ sees the expression "`1,2`" and evaluates it eagerly as a comma expression on integers.[5] 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 "if"s are used for difficult type-inferencing issues that come from `letrec`. In the factorial example at the end of Figure 6, 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 "if"s 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

[5] 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.

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`.

#### *2.3.1.5 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

```
// 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 ];
```

### 2.3.2 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 state, exceptions,

I/O, and other concepts common to impure languages that are otherwise difficult to implement in pure functional languages[39, 69].

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++. Then we show some example applications of monads in C++. We finish with a short summary of monadic programming in FC++.

#### *2.3.2.1 Introduction to monads in Haskell*

Monads have been popularized in part by being a part of the Haskell standard library. The Haskell language has a feature called "type classes" which provide bounded parametric polymorphism. This feature is essential to the definition of monads in Haskell, so we review it briefly to begin our discussion.

Free (unbound) type variables can be bounded by "type classes". 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
```

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 :: [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 we would say:

```
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
    bind Nothing  k   =  Nothing
    unit x            =  Just x
-- in the Maybe monad
-- bind :: Maybe a -> ( a -> Maybe b ) -> Maybe b
-- unit :: a -> Maybe a
```

(We show the definitions for the functions `bind` and `unit` so that they may be compared with the corresponding FC++ code in Section 2.3.2.3.)

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:

```
[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.

#### *2.3.2.2 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 that 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 an `EqualityComparable` type—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[50, 57] 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 constraints), 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. (For more on the relationship between C++ concepts and Haskell type classes, see [24].)

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.

#### *2.3.2.3 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 7. `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 8 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;
```

**Figure 7:** 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;
   }
*/
```

**Figure 8:** 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;
   };
```

**Figure 9:** Definition of the `Maybe` monad (`MaybeM`)

Figure 9 shows how we define the `Maybe` monad in FC++. Nested in `struct MaybeM` we define `unit`, `bind`, and `zero`, as well as `typedef`s 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 comprehension syntax to arbitrary monads was originally discussed by Wadler[70].)

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           = Ident x
   bind (Ident 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 10. 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).

#### *2.3.2.4 Monads in FC++*

Whereas the previous subsection introduced FC++ monads, here the details are fleshed out. FC++ provides functoids for the main monad operations. Specifically, FC++ supports:

```
unitM<SomeMonad>()      bindM<SomeMonad>()
zeroM<SomeMonad>()      plusM<SomeMonad>()
bindM_<SomeMonad>()     mapM<SomeMonad>()
joinM<SomeMonad>()      liftM<SomeMonad>()
liftM2<SomeMonad>()     liftM3<SomeMonad>()
```

```
// Nothing corresponding to Identity data type needed by Haskell
struct IdentityM {    // M a = a
   typedef Id Unit;   // "Id" is the identity functoid in FC++
   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;
};
```

**Figure 10:** Definition of the `IdentityM` monad

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

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 `bindM`)
- a lambda expression (Translates into a call to `bindM_`)
- a guard expression of the form "`guard[boolLambdaExp]`" (Translates into an if-then-else with `zeroM` 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 in a moment.)

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 `InferrableMonad`s, FC++ (like Haskell) can automatically infer the monad based on the datatype; thus the `doM` construct does not need to have a monad passed as an explicit parameter.

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 2.3.1 when implementing monads in FC++.

#### *2.3.2.5 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 propagate 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 [70] 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 [33]; we implement the examples in that paper in one of the example files that come 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
```

thus redefining 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[29]. 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 in the definition of `factor`.) 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 `List`s, 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).

#### *2.3.2.6 Monad summary*

Monads in FC++ are similar to monads in Haskell. Both rely on bounded parametric polymorphism; this is done explicitly via type classes in Haskell, and implicitly via template concepts in C++. In both Haskell and FC++, there is syntactic sugar for do-notation and comprehensions. The major difference is that in FC++, monads do not rely on datatypes with unique algebraic datatype constructors. As a result, FC++ comprehension syntax generalizes to arbitrary monads via a trade-off: the user must provide an explicit annotation on the comprehension to denote "which monad".

Monads are often seen merely as a way to provide "impure" features (like state and I/O) to pure functional languages. FC++ demonstrates that monads have useful applications in impure languages as well. The parser example shows how monads provide a way to structure programs so that some cross-cutting concerns can be easily factored out, and the FC++ `ByNeed` monad is a convenient reusable way to incorporate lazy evaluation into a C++ program.

### 2.3.3 Static analysis and error checking

In FC++, lambda and monads effectively "embed a sublanguage" into C++. Our use of expression templates, `LambdaVars`, and operator overloading provides special-purpose syntactic constructs which look almost as though they were a part of the language. Since this is all done as a library, though, there are challenges with regards to static analysis and error checking.

To begin, consider this example of an FC++ lambda expression:

```
lambda(X,X)[ X ](1,2)
```

If the lambda expression here were written in a functional language, we would expect the language compiler/interpreter to detect the repeated variable `X` in the list of lambda arguments and issue an error message. Of course, the C++ compiler is not automatically so kind—as far as it is concerned, `lambda()` is just another function call, and thus no special analysis is merited. As a result, in the FC++ library, we write small meta-programs within the C++ type system (which is Turing-complete) to perform various static analyses and issue various kinds of error messages that are related to the new "sublanguages" provided by the libraries.

Consider again this expression:

```
lambda(X,X)[ X ](1,2)
```

To detect duplicated lambda variables in the argument list to lambda, we write a meta-program which is invoked when trying to compute the result type of the call to `lambda()`. Recall that lambda variables in FC++ are declared like so,

```
LambdaVar<1> X;   LambdaVar<2> Y;
```

with variables declared as instances of the `LambdaVar` template class, each with a different integer template parameter. This enables the C++ type system to distinguinsh among lambda variables, as each lambda variable has a different type. Thus we write the meta-program (encoded in the type of `lambda()`) to walk the list of arguments passed to `lambda()` and inspect the types for duplicates. If any duplicates are found, we issue a "custom error

message" describing the error. The C++ language/compiler provide no special support for customing error messages, but we can simulate them reasonably well by referring to non-existent variables or functions with long identifiers that describe the problem. For instance, in the case of

```
lambda(X,X)[ X ](1,2)  // on line 94 of example.cpp
```

the g++ compiler emits a (somewhat long) error message which ends with

```
example.cpp:94:   instantiated from here
./FC++/lambda.hpp:1137: no method '
   YouCannotPassTheSameLambdaVarTo_lambda_MoreThanOnce<true>::go'
```

This message is not ideal, but it does contain both of the essential elements: the location where the error occurred (`example.cpp`, line 94), and the nature of the error (described in the long identifier `YouCannotPass`...).

We can use this same strategy to detect a wide variety of errors. Here are a couple more examples of common errors that clients of the library might make, and the identifiers displayed in the error messages issued by the library's internal meta-programs:

```
lambda(X)[ Y ](1)
// issues an error containing
//    YouCannotInvokeALambdaContainingFreeVars


lambda(X,F)[ if0[F[X,0],0,plus] ](1,equal)
// issues an error containing
//    TrueAndFalseBranchOfIfMustHaveSameType
```

Unfortunately, these useful identifiers can only be injected into the compiler's error messages as part of the compiler's normal templates for displaying errors (e.g. "no method named `Foo`" or "no match for call to `Foo(Bar)`"), which often hinders their visibility.

Furthermore, many compilers will dump the entire "template instantiation stack" (which effectively represents the entire state of the metaprogram) as part of the error message. As

a result, a single error message may be literally *millions* of characters long. Fortunately, the "useful" portion of the message (the injected identifier) always appears right near the beginning or right at the end of the error message (depending on the particular type of error and on which compiler is used). As a result, huge compiler diagnostic messages can be managed with only minor sleuthing work by the user.

Overall, the special static analyses and error reporting done by the library is not ideal, but it is adequate. Many types of errors are automatically detected and reported. Though it may take users a little extra effort to find and interpret these messages within the context of a super-verbose compiler diagnostic, this is far preferable to the alternative: a non-working program with no information about either the location or the nature of the error.

### 2.3.4 Lazy lists: even and odd

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

As we saw in Section 2.2.1, FC++ lazy lists can be `cons()`ed up in the usual way:

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

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

```
struct EnumFrom:public CFunType<int,List<int> > {
   List<int> operator()( int x ) const {
      return cons(x, thunk1( EnumFrom(), x+1 ));
   }
} enumFrom;
```

Though short, the function body nevertheless requires explanation because of the apparent inconsistency in the use of `cons()`: does `cons()` accept as its second argument a list, or a thunk (like the above `thunk1` expression) returning a list?

The answer is that `cons()` is overloaded to accept either a list or a list thunk. As will be described in Section 2.5.2.2, a `List` is represented as a kind of variant record, whose tail portion may either be a reference to the rest of the list, or an unevaluated thunk which will produce the remainder list on demand.

In reference [68], it is pointed out that there are different "degrees of laziness". Depending upon implementation choices, we may consider a lazy stream of values to be "even" or "odd", where even streams are completely lazy, whereas odd streams sometimes exhibit a little too much eagerness. For example, in this code (adapted from the main running example in [68]):

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

we would expect `l` to have the final value `[1.0,0.0]`. However this will only work for "even" lists; an "odd" list will evaluate one element too far, and end up trying to compute `sqrt(-1.0)` and fail. The details of the differences between the even and odd styles are explicated in reference [68]. (As we shall see, FC++ code does not fail in this example; our lists are not over-eager.)

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

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

This may seem an awkward state of affairs, at first. However, this peculiar implementation offers an interesting benefit: by exposing the fact that some list elements are already evaluated (`OddLists`) to the type system, we can overload certain core functions like `head()` to take advantage of this information—to take the `head()` of a `List`, we must evaluate a thunk to produce a value (*after* checking to see if the value has previously been cached, as discussed in Section 2.5.2.1), whereas to take the `head()` of an `OddList`, we can simply access a stored value directly, which is much more efficient. Hence the separation of lists into two data types can improve the run-time performance of list code.

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

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

- *Eagerness.* Allowing implicit conversions between `OddLists` and `Lists` sacrifices some of the safety of even lists. Nevertheless, while the danger of over-eagerness exists, it lives only in "edge cases". The examples from [68], like

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

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

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

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

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

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

```
thunk2( map, sqrt, l )
```

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

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

our implementation as an interesting alternative to the possibilities considered in reference [68].

### 2.3.5 Strict lists and a generalized list interface

In addition to supporting lazy lists, FC++ also supports strict (eager) lists via the `StrictList` data structure. Both strict lists and lazy lists have a common interface, which means that it is possible to write FC++ functoids which are polymorphic with respect to the eagerness of the lists they manipulate.

As an example, consider the functoid `Map`. In FC++, when `map()` is applied to a lazy list (`List`), the result is a lazy list, whereas when map() is applied to a strict list (`StrictList`), the result is a strict list. Here is the code for the body of the `Map` functoid:

```
template <class F, class L>
typename Sig<F,L>::result_type
operator()( const F& f, const L& l ) const {
   if( null(l) )
      return NIL;
   else
      return cons( f(head(l)), thunk2( Map(), f, tail(l) ) );
}
```

Note that the second parameter to `operator()` is just a template argument (`L`). As a result, any data type which supports the functions used in the body of `Map` (`null()`, `head()`, etc.) can be passed as the second argument to `Map`.

All list data types need to support a common set of operations; we call this set the `ListLike` interface. Any datatype which supports the `ListLike` interface (that is, a type that *models* the `ListLike` *concept*, using the terminology of Section 2.3.2.2) can be used with FC++ list functoids like `Map`, which are written so as to be polymorphic with respect to any `ListLike` data type. FC++ provides three data types which support the `ListLike` interface: `List` and `OddList` (the even and odd versions of lazy lists), and `StrictList`

(eager lists). (Section 4.1.4 describes some of the details of the `ListLike` interface in more detail.)

Though these three data types have the same interface, they have slightly different behaviors for each operation. For example, taking the `head()` of an `OddList` or a `StrictList` just fetches a field from an already-evaluated data structure, whereas taking the `head()` of a `List` requires first evaluating a thunk. As a more interesting example of differing behavior, consider the last line of the `Map` functoid:

```
return cons( f(head(l)), thunk2( Map(), f, tail(l) ) );
```

When the two arguments to `cons()` are a value and a thunk, and `cons()` is used in a lazy list context (that is, `l` is a `List` or and `OddList`), the thunk gets stored as the yet-unevaluated "tail" portion of the list (as described in Section 2.3.4). But when this same expression is evaluated in a strict list context (`l` is a `StrictList`), `cons()` behaves differently, evaluating the thunk immediately. As a result, the code

```
map( some_func, some_strict_list )
```

results in a new `StrictList` where `some_func` has been (eagerly) applied to each element of `some_strict_list`.

To sum up: FC++ contains data types for both lazy lists and strict lists. These types support a common `ListLike` interface, so that list-manipulating functions (like `map()`) can be written so they are polymorphic with respect to the eagerness of the list. Thus, FC++ list functoids can behave either lazily of eagerly, depending on the data type arguments they are passed. As a result, clients of the FC++ library may choose to use either lazy or eager evaluation, depending on their particular needs.

## *2.4 Applications*

In this section we describe a number of useful applications of FC++. First we describe how higher-order functions can be applied to a number of object-oriented design patterns. Next we discuss how parametric polymorphism affects a number of other design patterns. Finally we mention examples of the utility of the library both "in the small" (where FC++

simplifies expressions of just one or two lines of code) and "in the large" (other entire libraries built atop the FC++ infrastructure).

### 2.4.1 Higher-order functions and design patterns

Functional programming promotes identifying pieces of functionality as just "functions" and manipulating them using higher-order operations on functions. These higher-order functions may be specific to the domain of the application or they may be quite general, like the currying and function composition operations are. Several design patterns [23] follow a similar approach through the use of *subtype polymorphism*. Subtype polymorphism allows code that operates on a certain class or interface to also work with specializations of the class or interface. This is analogous to higher-order functions: the holder of an object reference may express a generic algorithm which is specialized dynamically based on the value of the reference. Encapsulating functionality and data as an object is analogous to direct function manipulation. Other code can operate abstractly on the object's interface (e.g., to adapt it by creating a wrapper object).

It has long been identified that functional techniques can be used in the implementation of design patterns. For instance, the Visitor pattern is often considered a way to program functionally in OO languages. (The interested reader should see Reference [43] and its references for a discussion of Visitor.) The Smalltalk class `MessageSend` (and its variants, see Reference [3], p.254), the C++ Standard Library functors, Alexandrescu's framework (Reference [2], Ch. 5), etc., are all trying to capture the generic concept of a "function" and use it in the implementation of the Command or Observer pattern. In this section we will briefly review some of these well-known techniques, from the FC++ standpoint, by using indirect functoids. In Section 2.4.2 we will consider how the unique features of FC++ enable some novel implementations of other patterns.

**Command.** The Command pattern turns requests into objects, so that the requests can be passed, stored, queued, and processed by an object which knows nothing of either the action or the receiver of the action. An example application of the pattern is a menu widget. A pull-down menu, for instance, must "do something" when an option is clicked;

*Command* embodies the "something". Command objects support a single method, usually called `execute`. Any state on which the method operates needs to be captured inside a command object.

The motivation for using the Command pattern is twofold. First, holders of command objects (e.g., menu widgets) are oblivious to the exact functionality of these objects. This decoupling makes the widgets reusable and configurable dynamically (e.g., to create context-sensitive graphical menus). Second, the commands themselves are decoupled from the application interface and can be reused in different situations (e.g., the same command can be executed from both a pull-down menu and a toolbar).

Here is a brief example which illustrates how Command might be employed in a word-processing application:

```
class Command {
public:
   virtual void execute()=0;
};

class CutCommand : public Command {
   Document* d;
public:
   CutCommand(Document* dd) : d(dd) {}
   void execute() { d->cut(); }
};

class PasteCommand : public Command {
   Document* d;
public:
   PasteCommand(Document* dd) : d(dd) {}
   void execute() { d->paste(); }
};
```

```
Document d;
...
Command* menu_actions[] = {
   new CutCommand(&d),
   new PasteCommand(&d),
   ...
};
...
menu_actions[choice]->execute();
```

The abstract `Command` class exists only to define the interface for executing commands. Furthermore, the `execute()` interface is just a call with no arguments or results. In other words, the whole command pattern simply represents a "function object". From a functional programmer's perspective, `Command` is just a class wrapper for a "lambda" or "thunk"—an object-oriented counterpart of a functional idiom. Indirect functoids in FC++ represent such function-objects naturally: a `Fun0<void>` can be used to obviate the need for both the abstract `Command` class and its concrete subclasses:

```
Document d;
...
Fun0<void> menu_actions[] = {
   thunk1(ptr_to_fun(&Document::cut), &d),
   thunk1(ptr_to_fun(&Document::paste), &d),
   ...
};
...
menu_actions[choice]();
```

In this last code fragment, all of the classes that comprised the original design pattern implementation have disappeared! `Fun0<void>` defines a natural interface for commands,

and the concrete instances can be created on-the-fly by making indirect functoids out of the appropriate functionality, currying arguments when necessary.

The previous example takes advantage of the fact that `ptr_to_fun` can be used to create functoids out of all kinds of function-like C++ entities. This includes C++ functions, instance methods (which are transformed into normal functions that take a pointer to the receiver object[6] as an extra first argument—as in the example), class (static) methods, C++ Standard Library `<functional>` objects, etc. This is an example of design inspired by the functional paradigm: multiple distinct entities are unified as functions. The advantage of the unification is that all such entities can be manipulated using the same techniques, both application-specific and generic.

**Observer.** The Observer pattern is used to register related objects dynamically so that they can be notified when another object's state changes. The main participants of the pattern are a *Subject* and multiple *Observers*. Observers register with the subject by calling one of its methods (with the conventional name `attach`) and un-register similarly (via `detach`). The subject notifies observers of changes in its state, by calling an observer method (`update`).

The implementation of the observer pattern contains an abstract `Observer` class that all concrete observer classes inherit. This interface has only the `update` method, making it similar to just a single function, used as a callback. In fact, the implementation of the Observer pattern can be viewed as a special case of the Command pattern. Calling the `execute` method of the command object is analogous to calling the `update` method of an observer object.

The FC++ solution strategy for the Observer pattern is exactly the same as in Command. The Subject no longer cares about the type of its receivers (i.e., whether they are subtypes of an abstract `Observer` class). Instead, the interesting aspect of the receivers—their ability to receive updates—is encapsulated as a `Fun0<void>`. The abstract `Observer` class disappears. The concrete observers simply register themselves with the subject. We

[6]Or a pointer to a `const` receiver object, if the method itself was `const`. The FC++ library strives to be `const`-correct.

will not show the complete code skeletons for the Observer pattern, as they are just specializations of the code for Command. Nevertheless, one aspect is worth emphasizing. Consider the code below for a concrete observer:

```
class ConcreteObserver {
   ConcreteSubject& subject;
public:
   ConcreteObserver( ConcreteSubject& s ) : subject(s) {
      s.attach( thunk1(ptr_to_fun(&ConcreteObserver::be_notified),this) );
   }
   void be_notified() {
      cout << "new state is" << subject.get_state() << endl;
   }
};
```

Note again how `ptr_to_fun` is used to create a direct functoid out of an instance method. The resulting functoid takes the receiver as its first parameter. `curry` is then used to bind this parameter. This approach frees observers from needing to conform to a particular interface. For instance, the above concrete observer implements `be_notified` instead of the standard `update` method, but it still works. Indeed, we can turn an arbitrary object into an observer simply by making a functoid out of one of its method calls—the object need not even be aware that it is participating in the pattern. This decoupling is achieved by capturing the natural abstraction of the domain: the function object.

Summarizing, the reason that `Fun0<void>` can replace the abstract `Observer` and `Command` classes is because these classes serve no purpose other than to create a common inteface to a function call. In `Command`, the method is named `execute()`, and in `Observer`, it is called `update()`, but the names of the methods and classes are really immaterial to the pattern. Indirect functoids in FC++ remove the need for these classes, methods, and names, by instead representing the core of the interface: a function call which takes no argument and returns nothing.

C++'s parameterization mechanism lets us extend this notion to functions which take arguments and return values. For example, consider an observer-like scenario, where the notifier passes a value (for instance, a string) to the observer's `update` method, and the `update` returns a value (say, an integer). This can be solved using the same strategy as before, but using a `Fun1<string,int>` instead of a `Fun0<void>`. Again, the key is that the interface between the participants in the patterns is adequately represented by a single function signature[7]; extra classes and methods (with fixed names) are unnecessary to realize a solution.

**Virtual Proxy.** The Virtual Proxy pattern seeks to put off expensive operations until they are actually needed. For example, a word-processor may load a document which contains a number of images. Since many of these images will reside on pages of the document that are off-screen, it is not necessary to actually load the entire image from disk and render it unless the user of the application actually scrolls to one of those pages. In [23], an `ImageProxy` class supports the same interface as an `Image` class, but postpones the work of loading the image data until someone actually requests it.

In many functional programming languages, the Virtual Proxy pattern is unnecessary. This is because many functional languages employ *lazy evaluation*. This means that values are never computed until they are actually used. This is in contrast to *strict* languages (like all mainstream OO languages), where values are automatically computed when they are created, regardless of whether or not they are used.

Since C++ is strict, FC++ is also strict by default. Nevertheless, a value of type `T` can be made lazy by wrapping the computation of that value in a `Fun0<T>`. This is a common technique in strict functional languages. It encapsulates a computation as a function and causes the computation to occur only when the function is actually called (i.e., when the result is needed). For instance, in FC++ a call `foo(a,b)` can be delayed by writing it as `thunk2(foo,a,b)`. The latter expression will return a 0-argument functoid that will perform the original computation, but only when it is called. Thus, passing this functoid

[7] A tuple of indirect functoids can be used if multiple function signatures are defined in an interface; the example in [23] of Command used for do/undo could be realized in FC++ with a `std::pair<Fun0<void>,Fun0<void>>`, for instance.

around enables the composition to be evaluated lazily.

We should reiterate that FC++ defines specific tools for conveniently expressing lazy computations. First, the `ByNeed` monad described in Section 2.3.2.5 can be used to provide a simple implementation of the `ImageProxy` mentioned earlier. Second, FC++'s *lazy list* data structure enables interesting solutions to some problems. For example, to compute the first $N$ prime numbers, we might create an infinite (lazy) list of all the primes, and then select just the first $N$ elements of that list.

### 2.4.2 Parametric polymorphism and design patterns

In the previous section, we saw how several common design patterns are related to functional programming patterns. All of our examples relied on the use of higher order functions. Another trait of modern functional languages (e.g., ML and Haskell) is support for parametric polymorphism with type inference. Type inference was discussed in Section 2.2.2: it is the process of deducing the return type of a function, given specific arguments. In this section, we will examine how some design pattern implementations can be improved if they employ parametric polymorphism with type inference and how they can further benefit from the entire arsenal of FC++ techniques for manipulating these polymorphic functions.(The discussion of this section is only relevant for statically typed OO languages, like Java, Eiffel, or C++. The novelties of FC++ are in its type system—it has nothing new to offer to a dynamically typed language, like Smalltalk.)

**Parametric vs. Subtype Polymorphism.** Design patterns are based on subtype polymorphism—the cornerstone of OO programming. Parametric polymorphism, on the other hand, is not yet commonly available in OO languages, and even when it is, its power is typically limited—e.g., there is no type inference capability. FC++ adds this capability to C++. It is interesting to ask when parametric polymorphism can be used in place of subtype polymorphism and what the benefits will be, especially in the context of design pattern implementations.

Parametric polymorphism is a static concept: it occurs entirely at compile time. Thus, to use a parametrically polymorphic operation, we need to know the types of its arguments

at each invocation site of the operation (although the same operation can be used with many different types of arguments). In contrast, subtype polymorphism supports dynamic dispatch: the exact version of the executed operation depends on the run-time type of the object, which can be a subtype of its statically known type.

Therefore a necessary condition for employing parametric polymorphism is to statically know the type of operands of the polymorphic operation at each invocation site. When combined with type inference, parametric polymorphism can be as convenient to use as subtype polymorphism and can be advantageous for the following reasons:

- *No common supertype is required.* The issue of having an actual common superclass or just supporting the right method signature is similar to the named/structural subtyping dilemma. All mainstream OO languages except Smalltalk use named subtyping: a type A needs to declare that it is a subtype of B. In contrast, in structural subtyping, a type A can be a subtype of type B if it just implements the right method signatures. The advantage of requiring a common superclass is that accidental conformance is avoided. The disadvantage is that sometimes it is not easy (or even possible) to change the source code of a class to make it declare that it is a subtype of another. For instance, it may be impossible to modify pre-compiled code, or it may be tedious to manipulate existing inheritance hierarchies, or the commonalities cannot be isolated due to language restrictions (e.g., no multiple inheritance, no common interface signature). Even in languages like Java where a supertype of all types exists (the `Object` type), problems arise with higher-order polymorphic functions, like our `curry` operator. The problem is that an `Object` reference may be used to point to any object, but it cannot be passed to a function that expects a reference of a specific (but unknown) type. Thus, implementing a fully generic `curry` with subtype polymorphism is impossible.

- *Type checking is static.* With subtype polymorphism, errors can remain undetected until run-time. Such errors arise when an object is assumed to be of a certain dynamic type but is not. Since the compiler can only check the static types of objects, the

error cannot be detected at compile-time. In fact, for many of the most powerful and general polymorphic operations, subtype polymorphism is impossible to use with any kind of type information. For instance, it would be impossible to implement a generic `compose` operator with subtype polymorphism, unless all functions composed are very weakly typed (e.g., functions from `Objects` to `Objects`). The same is true with most other higher-order polymorphic operations (i.e., functions that manipulate other functions).

- *Method dispatch is static.* Despite the many techniques developed for making dynamic dispatch more efficient, there is commonly a run-time performance cost, especially for hard-to-analyze languages like C++. Apart from the direct cost of dynamic dispatch itself, there is also an indirect cost due to lost optimization opportunities (such as inlining). Therefore, when parametric polymorphism can be used in place of subtype polymorphism, the implementation typically becomes more efficient.

The examples that follow illustrate the advantages of using parametric polymorphism in the implementations of some design patterns.

**Adapter.** The Adapter pattern converts the interface of one class to that of another. The pattern is often useful when two separately developed class hierarchies follow the same design, but use different names for methods. For example, one window toolkit might display objects by calling `paint()`, while another calls `draw()`. Adapter provides a way to adapt the interface of one to meet the constraints of the other.

Adaptation is remarkably simple when a functional design is followed. Most useful kinds of method adaptation can be implemented using the currying and functoid composition operators of FC++, without needing any special adapter classes. These adaptation operators are very general and reusable.

Consider the Command or Observer pattern. As we saw, in an FC++ implementation there is no need for abstract `Observer` or `Command` classes. More interestingly, the concrete observer or commands do not even need to support a common interface—their existing methods can be converted into functoids. Nevertheless, this requires that the existing

methods have the right type signature. For instance, in our `ConcreteObserver` example, above, the `be_notified` method was used in place of a conventional `update` method, but both methods have the same signature: they take no arguments and return no results. What if an existing method has *almost* the right signature, or if methods need to be combined to produce the right signature?

For an example, consider a class, `AnObserver`, that defines a more general interface than what is expected. `AnObserver` may define a method:

```
void update(Time timestamp) { ... }
```

We would like to use this method to subscribe to some other object's service that will issue periodic updates. As shown in the Observer pattern implementation, the publisher expects a functoid object that takes no arguments. This is easy to effect by adapting the observer's interface:

```
thunk2( ptr_to_fun(&AnObserver::update), this, current_time() )
```

In the above, we used a constant value (the current time) to specialize the update method so that it conforms to the required interface. That is, all update events will get the same timestamp—one that indicates the subscription time instead of the update time. A better approach is:

```
compose( ptr_to_fun(&AnObserver::update)(this),
         ptr_to_fun(current_time) )
```

In this example we combined currying with function composition in order to specialize the interface. The resulting function takes no arguments but uses global state (returned by the `current_time()` routine) as the value of the argument of the `update` method. In this way, each update will be correctly timestamped with the value of the system clock at the time of the update!

Other parametric polymorphism approaches (e.g., the functional part of the C++ Standard Library [59], or Alexandrescu's framework for functions [2], Ch.5) support currying and composition for *monomorphic* functions. The previous examples demonstrate the value

of type inference, which is not unique to FC++. Nevertheless, FC++ also extends type inference to *polymorphic* functions. We will see examples of currying and composition of polymorphic operations in the implementations of the next few patterns.

**Decorator.** The Decorator pattern is used to attach additional responsibilities to an object. Although this can happen dynamically, most of the common uses of the Decorator pattern can be handled statically. Consider, for instance, a generic library for the manipulation of windowing objects. This library may contain adapters, wrappers, and combinators of graphical objects. For example, one of its operations could take a window and annotate it with vertical scrollbars. The problem is that the generic library has no way of creating new objects for applications that may happen to use it. The generic code does not share an inheritance hierarchy with any particular application, so it is impossible to pass it concrete factory objects (as it cannot declare references to an abstract factory class).

This problem can be solved by making the generic operations be parametrically polymorphic and enabling type inference. For instance, we can write a generic FC++ functoid that will annotate a window with a scrollbar:

```
struct AddScrollbar {
   template <class W>
   struct Sig : public FunType<W,ScrollWindow<W> *> {};

   template <class W>
   typename Sig<W>::ResultType operator() (const W& window) const {
      return new ScrollWindow<W>(window);
   }
} add_scrollbar;
```

The above decorator functoid can be used with several different types of windows. For a window type `W`, the functoid's return type will be a pointer to a decorated window type: `ScrollWindow<W>`. (In fact, `ScrollWindow` can be a mixin, inheriting from its parameter, `W`.)

Since the functoid conforms to the FC++ conventions, it can be manipulated using the standard FC++ operators (e.g., composed with other functoids, curried, etc.). Composition is particularly useful, as it enables creating more complex generic manipulators from simple ones. For instance, a function to add both a scrollbar and a title bar to a window can be expressed as a composition:

```
compose(add_titlebar, add_scrollbar)
```

instead of adding a new function to the interface of a generic library. Similarly, if the `add_titlebar` operation accepts one more argument (the window title), the currying operation can be used (implicitly in the example below):

```
add_titlebar("Window Title")
```

The previous examples showed how classes can be statically decorated, possibly with new abilities added to them. Nevertheless, a common kind of decoration is pure wrapping, where the interface of the class does not change, but old operations are extended with extra functionality. Using parametric polymorphism one can write special-purpose polymorphic wrappers that are quite general. These could also be written as C++ function templates, but if they are written as FC++ functoids, they can be applied to polymorphic functoids and they can themselves be manipulated by other functoids (like `compose`). Consider, for instance, an instrumentation functoid that calls a one-argument operation, prints the result of the invocation (regardless of its type) and returns that same result:

```
struct GenericInstrumentor {
   template <class C, class A> struct Sig
   : public FunType<C, A, typename C::template Sig<A>::ResultType> {};

   template <class C, class A>
   typename C::template Sig<A>::ResultType
   operator() ( const C& operation, const A& argument ) const {
      typename C::template Sig<A>::ResultType r = operation(argument);
```

```
        std::cerr << "Result is: " << r << std::endl;
        return r;
    }
} generic_instrumentor;
```

`GenericInstrumentor` exemplifies a special-purpose functoid (it logs the results of calls to an error stream) that can be generally applied (it can wrap any one-argument function). Note that the `before()` and `after()` combinators (described in Section 2.2.9) are other examples of generally useful method decorators.

**Builder.** The Builder design pattern generalizes the construction process of conceptually similar composite objects so that a generic process can be used to create the composite objects by repeatedly creating their parts. More concretely, the main roles in a Builder pattern are those of a *Director* and a *Builder*. The Director object holds a reference to an abstract Builder class and, thus, can be used with multiple concrete Builders. Whenever the Director needs to create a part of the composite object, it calls the Builder. The Builder is responsible for aggregating the parts to form the entire object.

A common application domain for the Builder pattern is that of data interpretation. For instance, consider an interpreter for HTML data. The main structure of such an interpreter is the same, regardless of whether it is used to display web pages, to convert the HTML data into some other markup language or word-processing format, to extract the ASCII text from the data, etc. Thus, the interpreter can be the Director in a Builder pattern. Then it can call the appropriate builders for each kind of document element it encounters in the HTML data (e.g., font change, paragraph end, text strings, etc.).

In the Builder pattern, the Director object often implements a method of the form:

```
void construct(ObjCollection objs) {
    for all objects in objs {  // "for all" is pseudocode
        if (object is_a A)      // "is_a" is pseudocode
            builder->build_part_A(object);
        else if (object is_a B)
```

```
            builder->build_part_B(object);
        ...
    }
}
```

Note that the `build_part` method of the `builder` objects returns no result. Instead, the Builder object aggregates the results of each `build_part` operation and returns them through a method (we will call it `get_result`). This method is called by a client object (i.e., *not* the Director!).

A more natural organization would have the Director collect the products of building and return them to the client as a result of the `construct` call. In an extreme case, the `get_result` method could be unnecessary: the Director could keep all the state (i.e., the accumulated results of previous `build_part` calls) and the Builder could be stateless. Nevertheless, this is impossible in the original implementation of the pattern. The reason for keeping the state in the Builders is that Directors have no idea what the type of the result of the `build_part` method might be. Thus, Directors cannot declare any variables, containers, etc. based on the type of data returned by a Builder. Gamma et al. [23] write: "In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class."

This scenario (no common interface) is exactly one where parametric polymorphism is appropriate instead of subtype polymorphism. Using parametric polymorphism, the Director class could infer the result types of individual Builders and define state to keep their products. Of course, this requires that the kind of Builder object used (e.g., an HTML to PDF converter, an on-screen HTML browser, etc.) be fixed for each iteration of the `construct` loop, shown earlier. This is, however, exactly how the Builder pattern is used: the interpretation engine does not change in the middle of the interpretation. Thus, the pattern is *static*—another reason to prefer parametric polymorphism to subtyping. This may result in improved performance because the costs of dynamic dispatch are eliminated.

The new organization also has other benefits. First, the control flow of the pattern is

simpler: the client never calls the Builder object directly. Instead of the `get_result` call, the results are returned by the `construct` call made to the Director. Second, Directors can now be more sophisticated: they can, for instance, declare temporary variables of the same type as the type of the Builder's product. These can be useful for caching previous products, without cooperation from the Builder classes. Additionally, Directors can now decide when the data should be consumed by the client. For instance, the Observer pattern could be used: clients of an HTML interpreter could register a callback object. The Director object (i.e., the interpreter) can then invoke the callback whenever data are to be consumed. Thus, the `construct` method may only be called once for an entire document, but the client could be getting data after each paragraph has been interpreted.

Another observation is that the Director class can be replaced by a functoid so that it can be manipulated using general tools. Note that the Director class in the Builder pattern only supports a single method call. Thus, it can easily be made into a functoid. Calling the functoid will be equivalent to calling `construct` in the original pattern. The return type of the functoid depends on the type of builder passed to it as an argument (instead of being `void`). An example functoid which integrates these ideas is shown here:

```
struct DoBuild {
  template <class B, class OC>
  struct Sig: public FunType<B,OC,Container<B::ResultType> > {};
  template<class B, class OC>
  Container<B::ResultType> operator()( B b, OC objs ) const {
     Container<B::ResultType> c;
     for all objects in objs {  // "for all" is pseudocode
        if (object is_a A)      // "is_a" is pseudocode
           c.add(b.build_part_A(object));
        else if (object is_a B)
           c.add(b.build_part_B(object));
        ...
     }
```

```
        return c;
    }
} do_build;
```

With this approach, the "director" functoid is in full control of the data production and consumption. The Director can be specialized via currying to be applied to specific objects or to use a specific Builder. Two different Directors can even be composed—the first building process can assemble a builder object for the second!

### 2.4.3 Other applications

FC++ has applications outside the area of OO design patterns, as well. Here we describe other application domains for FC++, and other entire libraries which have been built atop the FC++ infrastructure.

As described in Section 2.3.2.5, FC++'s support for functional programming and monads makes it possible to implement monadic parser combinators in FC++. We have implemented a small parser combinator library, using the same elegant design found in reference [33] (which describes such a library written in Haskell).

FC++ is also useful on a very small scale for creating new anonymous functions for use with standard algorithms. For example, to add `2` to each element of a `std::vector` (named `v`) using just the C++ standard library, you must write

```
std::transform( v.begin(), v.end(), v.begin(),
                std::bind1st( std::plus<int>(), 2 ) ) );
```

but when FC++ is brought to bear, you can just write

```
std::transform( v.begin(), v.end(), v.begin(), fcpp::plus(2) );
```

(note the difference in the final argument to `std::transform()`). The currying and lambda facilities of FC++ make it much easier to write tiny anonymous functions.

On a much larger scale, FC++ is useful for functional programmers because it provides an alternative, commonly available platform for implementing familiar designs. An example

of this approach is the XR (*Exact Real*) library [12]. XR uses the FC++ infrastructure to provide exact (or *constructive*) real-number arithmetic, using lazy evaluation.

Another third-party library built atop FC++ is BSFC++ [16]. The BSFC++ Library is a library for Functional Bulk Synchronous Parallel Programming which utilizes a functional design. And finally, the LC++ library for logic programming (described in the next chapter) is another entire library built on top of the FC++ infrastructure.

To sum up, the FC++ library supports functional programming in C++, by enabling users to write and manipulate polymorphic and higher-order functions. The library has a smooth interface to the rest of C++, so that functional code and OO code can blend well. Overall, FC++ has many useful applications, involving a number of different domains (such as parsing and parallel programming), and spanning the gamut of size (FC++ has utility for tiny one-liners, for design pattern implementations, and as an entire functional infrastructure on which to build other libraries).

## *2.5 Practical considerations*

In this section we focus on pragmatics. The first two subsections discuss efficiency: we describe experiments we have done to measure the performance of FC++ and the optimizations we have applied to achieve it. The final subsection deal with expressiveness: we discuss the capabilities and limitations of the library. For a detailed comparison to other libraries that similarly "sugarize" C++, see Section 5.1.2.

### 2.5.1 Performance

FC++ is quite efficient in its implementation of functional concepts. For common programming tasks that use the FC++ conventions, the overhead is either zero or negligible (i.e., just a dynamic dispatch indirection for indirect functoids). The only case where performance is a legitimate concern is if one attempts to copy functional idioms directly to C++ using FC++. FC++ is not an optimizing compiler for a functional language, so it misses several common optimizations; for example: no special runtime support for specialized functions exists; tail-recursion elimination is not automatically performed; no runtime support for lazy evaluation exists. Additionally, FC++ offers a simple reference counting mechanism

(used internally for indirect functoids and lazy lists), which is not directly comparable to an optimized garbage collector. Nevertheless, the implementation of FC++ carefully tries to avoid unnecessary overhead and a number of optimizations are employed. In the following section (Section 2.5.2), we will describe the optimizations in detail.

In this section we show some simple performance measurements comparing FC++ to Hugs (a well-known Haskell interpreter [32]) and ghc (an optimizing Haskell compiler [26]). The benchmarks are programs that C++ programmers are unlikely to write in this form, but they show common functional programming idioms, involving heavy use of lazy (infinite) lists. Therefore, these benchmarks serve as stress tests of FC++'s lazy lists.

For each benchmark, we wrote two programs: one in Haskell, and one in C++ using the FC++ library. The programs are faithful translations of each other, in that they each represent the same solution to the given problem. The programs were run on a Sun Sparc Ultra-30 with 128M of RAM. We used g++2.95.2, ghc5.00.1, and the February 2001 version of Hugs. In the case of both g++ and ghc, we used -O2 and static linking.

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

#### *2.5.1.1 Primes*

Primes is a simple program that computes a (lazy) list of the first $N$ prime numbers and then prints the $N$th prime. It does so simply by filtering all the primes from the (infinite) list of integers, and then taking the first $N$ of them. Figure 11 shows the code for primes in Haskell. Figure 12 shows the code for primes in FC++.

Table 1 shows the performance results for primes for various values of $N$. FC++ is about 55 times as fast as Hugs for this program, and also consistently faster than ghc. While Haskell uses the arbitrary precision type `Integer` by default, explicitly requesting 32-bit `Int`s had no measurable effect on the ghc-compiled program's performance. On the other hand, using `Int`s did speed up the Hugs times by about 15% for each run (the numbers in the table for Hugs are without the speedup).

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

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

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

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

l = primes 600

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

**Figure 11:** Primes in Haskell

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

**Table 1:** Primes (all times in seconds)

*2.5.1.2 Tree*

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

Figure 13 shows the Haskell code for tree; Figure 14 shows tree in FC++. For both the Haskell and C++ programs, the code that actually builds the random binary trees is elided from the listings.

Table 2 shows the performance results for tree. $N$ is the number of nodes in the tree. No results are reported for Hugs for more than 30,000 nodes because the system memory was exhausted. For this benchmark, FC++ is consistently faster than Hugs, but about three times slower than ghc. Investigating the disparity between the FC++ and ghc performance, we found that ghc performs lazy list concatenation much faster than FC++ does. We plan to

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

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

struct Factors : public CFunType<int,OddList<int> > {
   OddList<int> operator()( int x ) const {
      return filter( thunk2(divisible,x),
                     enumFromTo(1,x) );
   }
} factors;

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

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

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

**Figure 12:** Primes in FC++

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

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

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

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

**Figure 13:** Tree in Haskell

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

**Table 2:** Tree (all times in seconds)

search further for a generally applicable optimization that will speed up list concatenation. Note that for Tree, using `Int`s instead of `Integer`s had no measurable effect on the times for either ghc or Hugs.

#### *2.5.1.3 Hamming*

The final program computes Hamming numbers. Hamming numbers are all the integers which are products of powers of 2, 3, and 5. An elegant way to compute the (infinite) list of all Hamming numbers is to say that the first number in the list is 1, and that the rest of the list is computed by merging three other lists: twice, three times, and five times the list of Hamming numbers itself. The solution is very easy to express recursively in Haskell; it is given in Figure 15. Notice how the definition of `hamming` refers to `hamming` itself. To construct the same solution in C++, we need to be a little more verbose, but the structure is exactly the same. The FC++ code is shown in Figure 16.

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

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

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

struct Fringe : public CFunType<Tree*,OddList<int> > {
   OddList<int> operator()( Tree* t ) const {
      if( t==0 )
         return NIL;
      else if( t->leaf() )
         return cons(t->data,NIL);
      else
         return cat( Fringe()(t->left), thunk1(Fringe(),t->right) );
   }
};
Fringe fringe;

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

**Figure 14:** Tree in FC++

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

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

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

         print (hamming !! 2000)
```

**Figure 15:** Hamming in Haskell

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

**Table 3:** Hamming (all times in seconds)

Table 3 shows the relative performance of the programs to print the $N$th Hamming number. Again, FC++ outperforms Hugs, this time by a factor of about 10; the times for FC++ and ghc are nearly equal. For this program, we could not use the 32-bit `Int` in place of `Integer`, as `Int` is not wide enough—our C++ Hamming code needs the g++-specific `long long int` (64 bits) to handle the large numbers involved in this example.

#### *2.5.1.4 Disclaimers and conclusions*

In this section, we have compared the performance of C++ programs with Haskell programs. It is important to note that no direct comparison can really be made. All cross-language experiments are fraught with factors that make a direct apples-to-apples comparison impossible, and our experiments are no different. There are many confounding factors, a few

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

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

   template <class T>
   OddList<T> operator()( const List<T>& a, const List<T>& b ) const {
      T x = head(a);
      T y = head(b);
      if( x < y )
         return cons( x, thunk2( Merge(), tail(a), b ));
      else if( x > y )
         return cons( y, thunk2( Merge(), a, tail(b) ));
      else
         return cons( x, thunk2( Merge(), tail(a), tail(b) ));
   }
} merge;

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

struct Hamming : public CFunType< List<FOO> > {
   List<FOO> operator () () const {
      using fcpp::multiplies;
      static List<FOO> h = Hamming();
      static List<FOO> x = thunk2( map, multiplies( (FOO)2), h );
      static List<FOO> y = thunk2( map, multiplies( (FOO)3), h );
      static List<FOO> z = thunk2( map, multiplies( (FOO)5), h );
      static List<FOO> m1= thunk2( merge, x, y );
      static List<FOO> m2= thunk2( merge, m1, z );
      return cons( (FOO)1, m2 );
   }
} hamming;

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

**Figure 16:** Hamming in FC++

of which were mentioned at the beginning of this section. Here we list a handful of obvious differences between FC++ and Haskell which we have not attempted to account for.

- *Strictness.* Haskell is lazy (non-strict) throughout, whereas C++ is strict except in FC++ lazy lists, which are explicitly coded to be lazy.
- *Memory management.* FC++ manages memory with reference-counted pointers and uses the default allocator provided by our implementation. Haskell uses garbage collection, and a sophisticated allocator designed for optimal performance for a lazy functional language.
- *Exception handling.* Haskell has more exception-handling by default; for example, taking the `head()` of an empty list raises an exception in Haskell, whereas it simply leads to undefined behavior in FC++ (though we do have a compiler flag that enables exceptions for this kind of misuse).
- *Runtime.* Haskell has a run-time system which supports a mix of compiled and interpreted code, manages storage allocation, and supports concurrent threads of execution. C++ has no comparable run-time system.
- *Optimizations.* Many FC++ optimizations must be done "by hand"; the Haskell compiler performs similar optimizations automatically.

By listing these confounding factors, it is not our intention to invalidate the results of the experiments of this section. Rather, we simply wish to make explicit the context in which the results must be interpreted. It is meaningless to make general statements like "FC++ is faster than Haskell" or vice-versa. Our goal is merely to demonstrate that, even for benchmarks which make heavy use of lists and lazy evaluation, FC++ can perform roughly comparably to an optimized functional implementation.

### 2.5.2 Performance analysis and optimizations

The current FC++ implementation is more than an order or magnitude faster than the previous release of the library. In this section, we discuss six major optimizations we have

applied to our implementation, quantifying the individual benefits whenever possible. For each optimization, we picked an appropriate benchmark that clearly demonstrates the difference in performance. (The difference for the other programs is typically less dramatic.) At the end of the section, we also repeat an experiment from [48], comparing the performance of FC++ with Läufer's library.

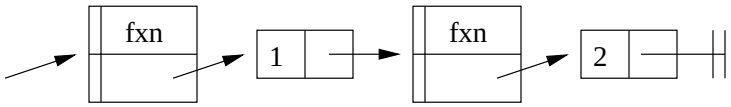
#### *2.5.2.1 Caching*

The first optimization is caching (memoization) in lazy lists. A lazy list is represented by an unevaluated function, or "thunk". When the value of the list is requested (`head()`, `tail()`, or `null()` is called), the thunk is called in order to produce the value. Rather than re-call the thunk each time the list's value is needed, the thunk should be called only once, and its value remembered. This optimization is imperative for programs like Hamming; without caching, Hamming grows exponentially (rather than linearly). In an older version of FC++ where caching was not available to lists, Hamming(300) took over 30 seconds to compute!

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

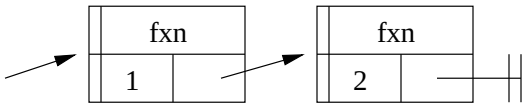
```
class Cache {
   bool value_is_valid;
   Fun0<Value> function;
   Value value;
public:
   Value val() {
      if( !value_is_valid )
         { value=function(); value_is_valid=true; }
      return value;
   }
};
```

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



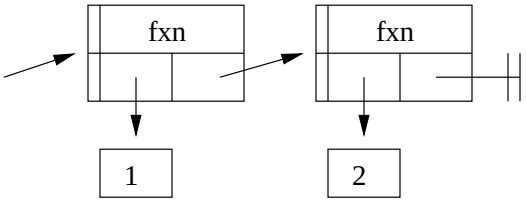

TOP

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




MIDDLE

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




BOTTOM

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

**Figure 17:** Three possible list implementations

#### *2.5.2.2 Structure of list implementation*

When we reimplemented FC++ lazy lists to use caching, we experimented with three different structures for the underlying implementation of lazy lists. We arbitrarily named the three versions TOP, MIDDLE, and BOTTOM (the names reflect the order that we wrote them on a white board). These structures are represented both as skeleton C++ code and pictorially in Figure 17. (To simplify the exposition, the code assumes that lists hold only `int`s (rather than being `template <class T>`s), and also uses raw pointers rather than reference-counted pointers.)

We tested all three list implementations on Primes(1000); the results are shown in Table 4. It should be no surprise that MIDDLE was the winner; MIDDLE contains fewer

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

**Table 4:** Comparison of different list structures

indirections than the other two solutions. TOP and BOTTOM are both slower due to the extra indirection and poorer locality. Additionally, BOTTOM (and MIDDLE too, actually) suffers another hit because it needs a special `value` to represent the empty list (called `XNIL`, which is like `XBAD` mentioned in Section 2.5.2.1), and every evaluation of a list requires an extra test to determine which member of the variant record is active.

The challenge is implementing MIDDLE for `List<T>`s where `T` has no default constructor. C++ requires that constructors be called for all members of an object, but in the case of MIDDLE, when the `value` in the `Cache` isn't valid, we have no constructor to call. As a result, the first field of the `pair` is actually an `unsigned char` array whose size and alignment are appropriate for `T`s. Placement `new` and explicit destructor invocations are used to explicitly manage the lifetime of the `T` created in the raw storage when the `Cache value` becomes valid. It should be noted that the C++ language standard provides no mechanism to ensure that the `unsigned char` array is properly aligned to hold data of type `T`. Nevertheless, there is a relatively portable "hack": creating a union of all kinds of C++ objects (primitive data types, structures, pointers, pointers to functions, pointers to members, etc.) ensures that the alignment of the union is wide enough to hold any kind of object on almost any system. Life would be a lot simpler if C++ were extended to have either a mechanism to specify alignments (a system-level solution) or a way to explicitly ask to have a particular structure member's constructor *not* called when the structure is created (a language-level solution); in the meantime, the hack works well enough on most systems. (A system for which the hack does not work can always revert to an alternative implementation of lists, e.g. `TOP`.)


**Figure 18:** Non-intrusive reference counting (left) and intrusive reference counting (right)

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

**Table 5:** The value of intrusive reference counting

#### *2.5.2.3 Intrusive reference counting*

The FC++ library contains two reference-counted pointer classes: one that uses an intrusive reference count, and one that is non-intrusive. The two schemes are depicted in Figure 18. The advantage of non-intrusive reference counts is that the object being counted does not need to support any particular interface; it is ignorant of the reference counting. Intrusive reference counts, on the other hand, require that the objects they count supply the counting mechanism. The benefits of intrusive reference counts are increased locality and fewer separate calls to `new`. (For a more thorough introduction to the topic of intrusive reference counts, see reference [2], Chapter 7.)

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

#### *2.5.2.4 Reusing functoids during recursive calls*

The typical implementation of a functoid which operates on lazy lists contains a curried recursive call as its last line. For example, consider the `Take` functoid shown in Figure 19 (with `Sig` member elided). (Recall that `take` selects the first $N$ elements of a list and discards the rest.) The call to `thunk2()` that is passed to `cons()` in the last line of the

```
struct Take {
   template <class T>
   OddList<T> operator()( size_t n, const List<T>& l ) const {
      if( n==0 || null(l) )
         return NIL;
      else
        return cons( head(l), thunk2( Take(), n-1, tail(l) ) );
   }
} take;
```

**Figure 19:** take() without functoid reuse

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

**Table 6:** The value of reusing functoids

functoid creates a new object on the heap that represents the recursive call (the "thunk" that makes functoids lazy). The only thing that differs between the newly created functoid and the current functoid itself are the values of `l` and `n`. Instead of discarding the called functoid and creating a similar new functoid, we can recode `take` so that it reuses the functoid. Figure 20 shows the code with this reuse (again, with `Sig` members elided).

We tested Primes both with and without "reuse" versions of `filter()`, `take()`, `at()`, `enumFrom()`, and `enumFromTo()`. The results are shown in Table 6. Clearly, reusing functoids is a big win. When there is no reuse, each call to `take()` has a functoid destructed, deallocated, and has a new functoid allocated and constructed. With reuse, there is only mutation; no heap allocation/deallocation occurs.

Comparing Figures 19 and 20, one can see that hand-coding a "reuse" version of a functoid takes a bit more code than the non-reuse version. In order to simplify the task of applying this valuable optimization, we have added `Reuser`s to the library. `Reuser`s enable us to capture the essence of functoid reuse with significantly less coding effort. Figure 21 shows `Take` written with a `Reuser`. A `ReuserN` is similar to a call to `thunkN()`. The `Reuser` appears as an extra parameter to the functoid. This parameter has a default value (thus making the interface change effectively "invisible" to clients) which is used to create

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

**Figure 20:** take() with functoid reuse

```
struct Take {
   template <class T>
   OddList<T> operator()( size_t n, const List<T>& l,
                          Reuser2<Inv,Var,Var,Take,size_t,List<T> >
                          r = REUSE_INIT ) const {
      if( n==0 || null(l) )
         return NIL;
      else
         return cons( head(l), r( Take(), n-1, tail(l) ) );
   }
} take;
```

**Figure 21:** take() with reuse via a Reuser

a new thunk on the heap. As a result, the initial call to a functoid that employs a `Reuser` allocates space for a thunk. Subsequent recursive calls are then channelled through the `Reuser` (rather than via a call to `thunkN()`); the `Reuser`'s heap thunk, when invoked, explicitly passes itself along to the next call as the extra parameter. This enables reuse of the existing heap thunk. `Reuser`s take template parameters specifying the argument types of the to-be-curried call, as well as extra template parameters that specify whether those parameters are invariant (`Inv`) or variant (`Var`) between calls (knowing this information prevents needless overwriting of duplicate values). Though the internal mechanism is quite complicated, `Reuser`s are relatively easy to apply (compare the code in Figures 19 and 21), and perform nearly as well as the "hand-written" code to perform the optimization (there is only a small "abstraction penalty").

#### *2.5.2.5 Avoiding functions with static data*

The `Cache` implementation (Figure 17, MIDDLE) uses two distinguished values for its pointer field. The value `XNIL` represents an empty list, and the value `XBAD` represents an "uncached" value (the function is valid, the value is not). These were originally encoded as

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

However for many compilers, it is far better to say

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

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

**Table 7:** The value of using global data

In the former, each time `XNIL()` is called, a boolean flag (inserted by the compiler) must be checked (to see if initialization of the static variable has already occurred[8]). In the latter, initialization happens at the start of the program, and `XNIL` is just a value. We tested both versions on Primes; the results are shown in Table 7.

Using global data with static initializations that require constructors to be called can be perilous; there are order-of-initialization and order-of-destruction issues for global objects in C++ that are often hard to solve. Fortunately, all of these global objects (which sometimes refer to one another) are defined in the same translation unit. This greatly simplifies the issue, and enables us to ensure the correct order of initialization for these objects (section 3.6.2, paragraph 1 of the C++ standard [34], prescribes the order of initialization for such objects). As for order-of-destruction issues, we circumvent the potential problems by artifically incrementing the reference counts of the global objects during initialization. Then, even when the reference-counted pointers are destructed after the end of `main()`, the ref counts do not go to zero, and so the objects to which they refer are left alive; they dangle in the heap until the system collects them when the program exits.

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

#### *2.5.2.6 Using iteration instead of tail recursion*

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

[8]Some compilers may employ clever optimizations to avoid the boolean check each time the function is called, however g++2.95.2 does not appear to be one of them.

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

**Table 8:** The value of transforming tail recursion into iteration

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

**Table 9:** The value of four optimizations combined

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

#### *2.5.2.7 Summary of optimizations*

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

#### *2.5.2.8 A final comparison*

In [48], we ran an experiment comparing the performance of the FC++ library with Läufer's functoid library. That experiment used a program similar to the "tree" program in Section 2.5.1.2. The experiment showed that the (previous) FC++ implementation was 4 to 8 times as fast as Läufer's library, thanks to the reference-counting in our implementation. We re-ran the experiment with the new FC++ implementation with all of the optimizations enabled. The results are shown in Table 10; FC++ is now more than 14 times as fast for this benchmark.

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

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

### 2.5.3 Expressiveness and limitations

We now 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. Nevertheless we have used our framework to define a large number (over 100) functoids and have not found our type language to be a problem—learning to use it required only minimal effort.

  The real advantage of FC++ is that, although function *definitions* need to be explicitly typed, function *uses* do not (even for polymorphic functions). In short, with our framework, C++ has as good support for higher-order and polymorphic functions as it does for any other first-class C++ type.

- *Polymorphic variables:* While FC++ has a great deal of support for polymorphic functions, we still cannot create run-time variables with polymorphic types, because these types cannot be expressed directly in C++. For example, even though `tail` and `init` (the dual of `tail`, which discards the *last* element of a list) both have the signature

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

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

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

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

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

  A closely related issue is that of naming. We saw base classes like `Fun1` and `Fun1Impl` in FC++, as well as operators like `makeFun1` and `monomorphize1`. These entities encode in their names the number of arguments of the functions they manipulate. Using C++ template specialization, this can be avoided, at least in the case of class templates. Thus, we can have templates `Fun` and `FunImpl` with a variable number of arguments. If template `Fun` is used with two arguments, then it is assumed to refer to a one-argument function (the second template argument is the return type). We experimented with this idea, and elected to use it only in the `CFunType` and `FunType` classes (which help implement `Sig` type signatures in class definitions). In client code, where indirect functoid variables are declared and used, the redundant `N` in the `FunN` names seems valuable to the human reader.

- *Automatic currying:* all of the library functoids support automatic currying via the `FullN` combinators. This enables a functoid to be called with fewer arguments than it expects, resulting in a new functoid which expects the remainder of the arguments. It is also possible to enable functoids to accept *more* arguments than they expect. For

example, imagine a one-argument function named `foo` which returns another one-argument function. We could imagine writing `foo(x,y)` to mean the same thing as `foo(x)(y)`. FC++ only supports the latter form by default. In the rare cases where this form of "uncurrying" is desirable, the `split_args()` functoid can be applied; its general behavior is described here:

```
split_args(f)(x,y,z)   means   f(x)(y)(z)
```

- *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.

  As described in Section 2.3.3, when using expression-template techniques, it is possible to apply C++ metaprogramming to analyze code for certain classes of errors. The kinds of errors detected using this mechanism are limited to those the library implementor (meta-)programmed *a priori*. Nevertheless, experience with, e.g. FC++'s

`lambda`, has shown us the most common types of errors, and we have written meta-programs to detect these errors and issue a suitable diagnostic.

Despite these issues, overall type error reporting in FC++ is adequate, and, with some experience, users have little difficulty with it.

- *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 2.2.3) are explicitly side-effect free—any class inheriting from our `FunNImpl` classes has to have a `const operator()`. Nevertheless, users of the library could decide to add other methods with side-effects to a subclass of `FunNImpl`. We strongly discourage this practice but cannot prevent it. It is a good convention to always declare methods of indirect functoids to be `const`.

  For direct functoids, guarantees are even weaker. We cannot even ensure that `operator()` will be `const`, although this is, again, a good practice. While functoids with side effects can be implemented in our framework (as described in Section 2.2.9), such functoids should be used with care. Other opportunities for code with side effects abound in C++. Our recommendation is that most code with side effects should be implemented outside the FC++ framework. For instance, such code could be expressed through native C++ functions. The purist can even use a state monad (which is one of the example monads supplied with the library) as an alternative to side-effects.

## *2.6 Discussion*

FC++ demonstrates that functional programming can be smoothly integrated into C++. The library supports major features found in modern functional programming languages,

including first-class functions, laziness, infix syntax and currying, lambda, and syntax support for monadic programming. These features have been added to the host language in a way that is more complete and seamless than prior attempts to add functional programming to existing object-oriented languages.

FC++ adds these functional abstractions to C++ in an efficient and useful manner. By utilizing features of C++ (especially templates), most of the features have been added with little or no abstraction penalty; even those features with the most potential runtime expenses (lists and laziness) are efficient when compared with similar code produced by modern functional language compilers. Furthermore, we have demonstrated a number of applications of the library, including design pattern implementations and third-party libraries built atop FC++.

In short, FC++ effectively behaves as a domain-specific embedded language for functional programming in C++. But functional programming is only part of the story—our thesis also states that logic programming can be similarly integrated into C++. Logic programming, by virtue of its more unusual way of representing control flow and data, creates a number of new challenges for a seamless integration. Thus the next chapter describes how we met those challenges with LC++, our logic programming library.

# CHAPTER III

# LC++

This chapter describes LC++, a library for logic programming in C++. We start by describing the interface to the library and how it supports the major features of logic programming, including queries, unification, backtracking, and functors. Next we discuss the LC++ implementation, which uses many of the same general techniques used in the FC++ implementation. The main challenges are different from FC++, however; whereas with functional programming, representing functions was paramount, with logic programming the control and data representations require the most ingenuity. We continue by describing applications of the library, and conclude with a comparison of the most directly related work.

## *3.1 Description of the library*

The LC++ library makes it possible to do logic programming in C++. The overall structure and syntax used are very similar to logic languages such as Prolog (the main difference being the addition of a static type system). Facts and rules are asserted into the database, and queries are run to unify free logic variables with values that solve the queries. The solutions can be accessed programmatically by C++ code, and they are only computed one-by-one (on demand), just as with Prolog.

### 3.1.1 Introductory example and syntax overview

Figure 22 shows an actual part of an LC++ program describing the Simpsons family relationships. The `lassert()` function is used to add facts and implications to the database, and the `query()` function is used to run queries. The syntax for functors, values, and logic variables is very similar to that of Prolog; Table 11 summarizes the differences between LC++ syntax and Prolog syntax.

```
FUN2(parent,string,string);        FUN1(female,string);
FUN2(father,string,string);        FUN2(sibling,string,string);
FUN2(mother,string,string);        FUN2(brother,string,string);
FUN2(child,string,string);         FUN2(sister,string,string);
FUN1(male,string);                 FUN3(ancestor,string,string,int);

DECLARE(Mom, string,0);            DECLARE(Sib, string,6);
DECLARE(Dad, string,1);            DECLARE(Sib2,string,7);
DECLARE(Kid, string,2);            DECLARE(Anc, string,8);
DECLARE(Par, string,3);            DECLARE(Tmp, string,9);
DECLARE(Bro, string,4);            DECLARE(X   ,int,  10);
DECLARE(Sis, string,5);            DECLARE(Y   ,int,  11);

string bart="bart", lisa="lisa", maggie="maggie",
   marge="marge", homer="homer", abraham="abraham";
lassert( male(bart) );         lassert( female(lisa) );
lassert( male(homer) );        lassert( female(maggie) );
lassert( male(abraham) );      lassert( female(marge) );

lassert( parent(marge,bart) );
lassert( parent(marge,lisa) );
lassert( parent(marge,maggie) );
lassert( parent(homer,bart) );
lassert( parent(homer,lisa) );
lassert( parent(homer,maggie) );
lassert( parent(abraham,homer) );

lassert( mother(Mom,Kid) -= parent(Mom,Kid) && female(Mom) );
lassert( father(Dad,Kid) -= parent(Dad,Kid) && male(Dad) );
lassert( child(Kid,Par) -= parent(Par,Kid) );
lassert( sibling(Sib,Sib2) -= father(Dad,Sib) && father(Dad,Sib2)
            && mother(Mom,Sib) && mother(Mom,Sib2)
            && not_provable( Sib==Sib2 ) );
lassert( brother(Bro,Sib) -= sibling(Bro,Sib) && male(Bro) );
lassert( sister(Sis,Sib) -= sibling(Sis,Sib) && female(Sis) );
lassert( ancestor(Par,Kid,1) -= parent(Par,Kid) );
lassert( ancestor(Anc,Kid,X) -= parent(Anc,Tmp) &&
            ancestor(Tmp,Kid,Y) && X.is(plus,Y,1) );

query( father(Dad,Kid) );
query( sibling(maggie,Sib2) );
query( ancestor(Anc,bart,X) );
```

**Figure 22:** Simpsons family relationships in LC++

| Description | Prolog | LC++ |
|---|---|---|
| unification | `=` | `==` |
| conjunction | `,` | `&&` |
| disjunction | `\|` | `\|\|` |
| implication | `:-` | `-=` |
| not provable | `not()` | `not_provable()` |
| evaluation | `is` | `is()` |
| dummy logic var | `_` | `_` |

**Table 11:** Syntax mapping between Prolog and LC++

### 3.1.2 Declaring relations and logic variables

LC++ uses the C++ type system and templates to enforce static typing of logic programming code. To declare a functor, we use the form `FUN`$n$ ($n$ denoting the arity of the functor) and specify the functor's name as well as the types of its arguments. For example, `parent` in the preceding example is declared like this:

```
FUN2(parent,string,string);
```

In fact, the declared functor is a singleton instance of a class with an overloaded `operator()` method; the example above declares `parent` as an instance of class `parent_TYPE` (which is the name of the class created by the `FUN2` macro).

Similarly, logic variables must be declared in order to statically typecheck them. In our example, to declare logic variable `X` of type `int`, we say:

```
DECLARE( X, int, 10 );
```

The surprising integer constant parameter will be explained in Section 3.2.2. As in Prolog, we use the convention of having a logic variable's name begin with a capital letter. Just as with the macro for functors, the `DECLARE` macro also declares a new typename; the C++ type of logic variable `X` is called `X_TYPE`.

### 3.1.3 Calling out to C++ functions

Like Prolog, LC++ uses "`is`" to bind logic variables to results of a computation. In the Simpsons example, the `ancestor` relation computes the number of generations between direct relatives like so:

```
lassert( ancestor(Par,Kid,1) -= parent(Par,Kid) );
lassert( ancestor(Anc,Kid,X) -= parent(Anc,Tmp) &&
             ancestor(Tmp,Kid,Y) && X.is(plus,Y,1) );
```

The code `X.is(plus,Y,1)` adds 1 to the current value of `Y` and binds the resulting value to `X`. Note that `plus` is a functoid from FC++. The general mechanism to call out to arbitrary C++ code in LC++ is

```
SomeLogicVar.is( some_functoid, arg1, ..., argN )
```

Thus, via `is()`, LC++ has a general mechanism to call functions or to have effects; just create a functoid that describes the desired computation, and pass it as the first argument to `is()`.

### 3.1.4 Asserting facts and rules, running basic queries

In the example, `lassert()` was used to populate the database with facts and implications, and `query()` was used to perform queries. But what does `query()` do with its results? In fact there are three different query functions. We postpone until Section 3.2.1 discussion of the main `query()` function. The simplest of the other two is `iquery()`: it prints out its results. So if the final query from the Simpsons example were

```
iquery( ancestor(Anc,bart,X) );
```

then the results

```
Result #1
 - Anc  = marge
 - X    = 1
Result #2
 - Anc  = homer
 - X    = 1
Result #3
 - Anc  = abraham
 - X    = 2
```

would be printed to the screen.

### 3.1.5 More on queries, environments, and result lists

Whereas `iquery()` prints its results, the `lquery()` function returns an FC++ `List` of results. `List` is a lazy list data type—the elements are not computed until actually requested. Each query result is represented by an environment object (actually, a smart pointer (`IRef`) to an environment object; more details to come in Section 3.2.1), which contains all the information about the logic variable bindings. The type of the environment object is a function of which (types of) logic variables appear in a particular query. Naming `Environment` types can be difficult; the result type of the call

```
lquery( ancestor(Anc,bart,X) );
```

is a `List` of

```
IRef<Environment<TL::CONS<X_TYPE,TL::CONS<Anc_TYPE,TL::NIL> > > >
```

objects. That is, it is a lazy list of references to a binding environment holding values for logic variables `X` and `Anc`. (More about the `TL` namespace and representing compile-time type-lists in Section 3.2.2.2.)

Such long and ugly type names are a common occurrence in C++ template libraries, especially those using expression templates. Fortunately we can shield the client from these names by providing a "type computer" which provides a managable alias for the type: the type `QRT<Foo_TYPE,Bar_TYPE>::IE` describes the type of results of a query involving the logic variables `Foo` and `Bar`. (`QRT` stands for "Query Return Type" and `IE` stands for `IRef<Environment>`.) As a result, we can just say

```
typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
List<IE> l = lquery( ancestor(Anc,bart,X) );
```

to get our list of results. We can then use the FC++ functions `null()`, `head()`, and `tail()` to traverse the list of references to environment objects.

The environment object itself responds to the method `at(`*SomeLogicVar*`)`, returning an object representing the binding for that logic variable. Assuming the variable is bound, the

"*" operator returns the value it is bound to. This way the client can print the results using its own choice of formatting. For example

```
while( !null(l) ) {
   IE env = head(l);
   std::cout << "X is " << *env->at(X) << " and Anc is "
       << *env->at(Anc) << std::endl;
   l = tail(l);
}
```

will print to the screen:

```
X is 1 and Anc is marge
X is 1 and Anc is homer
X is 2 and Anc is abraham
```

FC++ Lists are lazy, and the implementation of the LC++ query functions exploits this laziness in an important way. For instance, a query computation may not terminate. The query may yield a few results and then get "stuck" in an infinite loop, or the query may return an infinite number of results, such as in this simple example involving the natural numbers:

```
FUN1( nat, int );
DECLARE( X, int, 10 );
DECLARE( Y, int, 11 );
lassert( nat(0) );
lassert( nat(X) -= nat(Y) && X.is(plus,Y,1) );
```

Running the query `nat(X)` produces an infinite list of results. This doesn't present a problem, though; if we are only interested in the first 3 results, we just ask for those. That is, we can say

```
typedef QRT<X_TYPE>::IE IE;
List<IE> l = lquery( nat(X) );
```

```
for( int i=0; i<3; ++i ) {
   IE env = head(l);
   std::cout << "X is " << *env->at(X) << std::endl;
   l = tail(l);
}
```

and that code prints

```
X is 0
X is 1
X is 2
```

The implementation of this feature (laziness of query results) is described at length in Section 3.2.1.

### 3.1.6 Functors and data structures

The examples of the previous subsections illustrate some of the most important features of the library. In this subsection we discuss some of the deeper details of the library interface that are not covered by the simple example.

#### *3.1.6.1 User-defined C++ types as LC++ atoms*

User-defined types can be used as "atoms" for LC++, provided these types provide both a copy constructor and an equality operator. The copy constructor is required so that the object can be copied into an LC++ data structure, and the equality operator is required for unification. Here is an example of a user-defined type which can be used as an LC++ atom:

```
struct Point {
   int x;
   int y;
   bool operator==( const Point& p ) const
   { return this->x == p.x  &&  this->y == p.y; }
};
```

Depending upon how your application is structured, it might be better to represent `Points` using functors, instead (discussed next).

#### *3.1.6.2 LC++ functors as data structures*

Just as in Prolog, LC++ functors can be used as data structures as well as for predicates. For example, we could define

```
FUN2(point,int,int);
DECLARE( P, point_TYPE, 0 );
DECLARE( X, int,        1 );
```

and then use `point()` to create data structures where we can perform unification on various subparts:

```
iquery( P == point(X,3) && X == 4 );
// yields result:
//  - P    = point(4,3)
//  - X    = 4
```

Indeed, a logic variable with "functor shape" can play both the role of a term and a data structure within one query. For example,

```
iquery( A==ancestor(Anc,bart,X) && A );
```

results in

```
Result #1
 - Anc  = marge
 - X    = 1
 - A    = ancestor(marge,bart,1)
Result #2
 - Anc  = homer
 - X    = 1
 - A    = ancestor(homer,bart,1)
```

```
Result #3
 - Anc  = abraham
 - X    = 2
 - A    = ancestor(abraham,bart,2)
```

where the variable `A` (with type `ancestor_TYPE`) is both unified with an `ancestor` data structure and used as a query term.

### 3.1.7 Limitations

The current implementation of LC++ has a few important limitations and omissions, some of which we may address in the future.

*Omissions.* The current LC++ implementation omits a couple notable Prolog entities, namely the `cut` (`!`) operator and a `retract()` function.

*Parametric polymorphism.* One major restriction with LC++ is that functors are *monomorphic*. For example, we can define `append()` to work on lists of integers, but cannot use the same `append()` for lists of other types (e.g. `string`, `double`, etc.). This restriction is rooted in C++—in C++, a `virtual` function may not be a template, and our implementation makes essential use of `virtual` methods. An alternative implementation stategy might enable this restriction to be lifted. (Regardless, the applications we have encountered thus far do not require parametric polymorphism—see Section 3.3. Note also that this restriction and the monomorphic restrictions of FC++ indirect functoids (Section 2.2.3) and variables (Section 2.5.3) all have the same root cause.)

*Performance.* LC++ allows clients to update the database of facts and implications with calls to `lassert()` at any time—even during the execution of a query. This freedom allows for much dynamicity (e.g. the facts and rules can be read in at run-time from a file), but it effectively limits LC++ to executing queries like an interpreter, with all the associated performance limitations of that approach.

In addition, our current implementation is somewhat naive. Each time a predicate is invoked, the work to convert the `Rep` objects to the corresponding `Impl` objects (Section 3.2

describes the `Rep` and `Impl` objects in the implementation) is re-done; with a slightly different architecture, the `Impl` objects for each rule could be cached and reused. Also, all predicates and functors are internally represented as ternary functors (perhaps with some slots filled with "dummy" objects) even if the actual predicate or functor has a lower arity; this wastes some memory and causes some extra work to be done during unification of functors with low arities. These weaknesses make our current implementation run a few times slower than a Prolog interpreter on comparable code.

*Static analyses and transforms.* At the end of Section 3.2.2, we gave an example of one type of specialized semantic analysis that LC++ can perform at compile-time. It would be interesting to investigate if we can use static type information to enable some compile-time optimizations, such as re-structuring query trees, but we have not done this.

*Parameter modes.* Some logic programming languages, such as Mercury[53] and HAL[18], enable the programmer to annotate functors with mode and determinism declarations. These declarations enable more static checking and can help provide better run-time performance. It would be interesting to extend LC++'s static type system so that parameter modes could be expressed.

## *3.2 Beneath the Surface*

In this section we describe two of the most interesting aspects of the LC++ implementation. First, we discuss how LC++ implements its control flow, using FC++ lazy lists as a natural way to return query results one-at-a-time on demand. Second, we discuss the use of C++ "expression templates" to perform compile-time computation, enabling LC++ clients to use Prolog-like syntax but have the C++ compiler parse, type-check, and semantically analyze this code.

### 3.2.1 Query execution and C++ interfacing

LC++ represents query results as an `Environment` object, which maps each logic variable to its binding information. Since this is C++, we use effects to obtain a more efficient implementation. That is, we use destructive update to modify the environment: unification causes bindings to be added; backtracking causes bindings to be removed.

One common way to implement Prolog queries is to mimic continuation passing style (CPS). A continuation is a function which embodies "what to do next" (after the current action is completed). Using CPS makes it easy to implement Prolog's unusual control flow; any particular term can either continue forward through the query (by calling the next continuation when this portion of the query succeeds) or backtrack (by returning to its caller when this portion of the query fails). The activation stack holds the "undo" information used for backtracking, and the continuation parameter holds the "future" (the rest of the query to be evaluated). Other logic programming approaches, like MPC++[19] and J/MP[10], use explicit CPS to express the logical control flow in an imperative language.

The problem with this approach is that the action describing "what to do with the query results" must itself be passed into the query as the final continuation. In specific cases, this is not a problem: for example, if all you want to do is print out all the results, then it would be easy to create a continuation function which just prints the contents of the `Environment`, and to pass this continuation into `query()`. In the general case, however, a client of `query()` may want to use the results in some arbitrary way using arbitrary C++ code, and there is no general mechanism for creating a continuation out of "the rest of your C++ program".

Put another way, the problem is the impedence mismatch between normal C++ control flow (which uses an activiation stack) and LC++ control flow (which is effectively CPS). In simple cases where we are prepared to process all of the results at once, we can treat the `query()` function as a "stop the world" process, which uses CPS to run the query and process all the results (`query()` would not return until all of the results are processed). However in the general case, a client may not be prepared to process all the results at once; the client needs `query()` to return the results (lazily) in a data structure which can be processed later. FC++ lazy lists solve the problem; representing the results as a lazy list effectively enables arbitrary C++ code to call `query()` and then continue on its way, co-routining with the CPS query functionality whenever the next result is demanded by the client. FC++ plays a role with dual significance here: FC++ lazy lists provide a smooth way to create the *interface* to LC++ queries, and the FC++ library makes the

*implementation* much easier.

We now describe the actual implementation. At the lowest level, the main `query()` function returns a `std::pair` (standard C++ 2-tuple) whose first element is a reference to an `Environment` object and whose second element is a `List<Empty>`. `Empty` is a "nothing" data type—a struct with no members. `List<Empty>` signifies that the lazy list does not contain real values—it is only useful because traversing it produces side-effects on the `Environment`.

The purpose of the `List` is to give clients control over query evaluation. As each element of the `List` is demanded, the LC++ query runs to produce the next result by side-effecting the `Environment` object. When the `List` finally becomes `NIL` this means there are no more query results. Thus a client calls `query()` like this:

```
typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
std::pair<IE,List<Empty> > p = query( ancestor(Anc,bart,X) );
IE env = p.first;
List<Empty> l = p.second;
while( !null(l) ) {
   std::cout << "X is " << *env->at(X) << " and Anc is "
             << *env->at(Anc) << std::endl;
   l = tail(l);
}
```

The `lquery()` and `iquery()` functions (presented back in Section 3.1.5) are built atop this interface.

We now illustrate how `query()` is implemented with some code. A query is represented as a `Term` object. `Term`s can be conjunction terms, disjunction terms, unification terms, etc. All of them support a `run` method with the following interface:

```
struct Term {
   virtual List<Empty> run( IRef<Term> future ) = 0;
};
```

CPS is evident; in order to run a portion of a query, we must pass the remainder of the query (the continuation) as the `future` parameter. (Recall that `IRef<Term>` is like a `Term` pointer; FC++ provides the `IRef` class as a reference-counted smart pointer.) The `run()` method in `Term` returns a `List<Empty>`.

The body of the `query()` function ends with this code:

```
// "t" is a reference to the current Term
// "env" is a reference to the current Environment
List<Empty> l = thunk2( ptr_to_fun(&Term::run), t, end_of_query );
return std::make_pair( env, l );
```

The `end_of_query` object is just an instance of a `Term` whose `run()` body says

```
return cons( Empty(), NIL );
```

In other words, when we reach the end of the query, we should indicate one result by returning a one-element `List`.

Further examples help illuminate what is going on. Consider `DisjunctImpl`s (the "or" terms created with `||`). Here is the implementation, which just uses the FC++ `cat()` function to concatenate two lists:

```
struct DisjunctImpl : public Term {
   IRef<Term> lhs, rhs;
   List<Empty> run( IRef<Term> future ) {
      return cat( lhs->run(future),
                  thunk2( ptr_to_fun(&Term::run), rhs, future ) );
   }
};
```

and here is the code for conjuncts (`&&`):

```
struct ConjunctImpl : public Term {
   IRef<Term> lhs, rhs;
   List<Empty> run( IRef<Term> future ) {
```

```
        IRef<Term> newfuture = new ConjunctImpl( rhs, future );
        return lhs->run( newfuture );
    }
};
```

That is, given `term1&&term2` and a `future`, we run `term1` with `term2&&future` as its future.

Finally, consider unification. LC++ values can be unified using the `unify()` function, which returns a result of type `UnRes`. This is a two-element structure:

```
struct UnRes {      // UnRes means "Unification Result"
    bool ok;
    Fun0<void> undo;
};
```

If a unification fails, the `ok` field is set to `false` and the `undo` field is unused. If a unification succeeds, the environment is side-effected with the new binding, the `ok` field is set to `true`, and the `undo` field is set to a thunk which, when executed, will remove the newly-created binding from the environment. This is important to the `run()` method in `UnificationImpl`, which looks like this:

```
UnRes ur = lhs->unify(rhs);
if( !ur.ok )
    return NIL;
else
    return cat( future->run( dummy_term ),
                before( ur.undo, lambda()[ NIL ] ) );
```

The logic of `UnificationImpl::run()` reads as follows. First, try to unify the left-hand-side with the right-hand-side. If this fails, return the empty list (there are no results). Otherwise, the result is the catentation of (1) the results from running the future (the rest of the query)[1] and (2) an empty list with the undo thunk prepended. This results in the

[1]The `dummy_term` passed to `future`'s `run()` method is a just a meaningless placeholder; `query()` ensures that all `Terms` end with an `end_of_query` object, which never uses its `future` parameter.

effects happening at the right time. Since unification succeeded, we have added a binding to the environment. We run the rest of the query with that binding intact. After all of those results have been processed (that is, when the client demands the next result *after* those results created downstream from this portion of the query), we undo the binding created by this unification (to effect backtracking).

### 3.2.2 Parsing and semantic analysis

In this subsection, we discuss the C++-specific implementation techniques that LC++ uses to enable clients to express logic programs in C++ using the simple declarative syntax of the library interface. Expression templates[66] are used to parse LC++ rules and queries as C++ expressions, and template meta-programming[14] techniques are used to do basic analysis of LC++ expressions so that LC++ code works within C++'s static type system.

#### *3.2.2.1 Parsing and Representation*

The syntax of LC++ is implemented by overloading the C++ language operators that appear in Table 11 (Section 3.1.1). These overloaded operators return values of types that reflect the syntax tree of the expression. For instance, C++ operators like `-=` and `&&` are overloaded to create values of type `ImplicationRep` and `ConjunctRep`, respectively. All the different "`Rep`" types serve to represent different entities of the syntax tree. Logic variables correspond to C++ values of type `LogicVariable<`$T$`>`, where $T$ is the type of the logic variable (e.g., integer, string, etc.). For example, code like

```
X == 3  &&  Y == 4
```

creates a value whose type is a tree with a `ConjunctRep` at the root, with two `UnificationRep`s below it, each of which has a `LogicVariable<int>` child and an `int` child.

#### *3.2.2.2 Type-Checking and Semantic Checking*

The C++ type system is Turing-complete, and C++ templates can be used for meta-programming in the type system. (The C++ template system is an untyped, pure functional programming language, where the atomic values are C++ types.) Using this feature of C++ we can perform arbitrary (but very cumbersome) computations at compile-time.

There are three main high-level results of the compile-time computation performed by LC++, listed here with an example of each:

- Type checking: ensuring that in the expression `X==1`, `X` is a logic variable of type `int` (and not, say, one of type `string`).
- General semantic checking: ensuring that a client cannot ask for `env->at(X)` from the result of a query not involving `X`.
- Specialized semantic checking: ensure that the named logic variables appearing in an `lassert()` statement always appear in more than one location.

We achieve these results by using metaprogramming on "type lists" in the `Rep` classes; this is explained next.

Recall that client code to run a query looks like

```
typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
List<IE> l = lquery( ancestor(Anc,bart,X) );
```

The `QRT` type computer computes the type of an environment that has bindings for each of the logic variables[2] named by its template parameters.

The complication is that `lquery()` must compute a value whose type is compatible with the type computed by `QRT`. To do this, the implementation of `lquery()` must (at compile-time) traverse the parse tree of the logic expression passed to it and compute the set of all logic variables that appear in the term. The type representing this set should be the same as that computed by `QRT`. Discovering all of the logic variables used in a logic expression is done using the `Rep` classes. For each `Rep` type (representing an LC++ program term), we keep a list of all the logic variables that appear in the term. This compile-time list is maintained as a field of each `Rep` class called "`LVs`". Rather than discussing here the details of manipulating compile-time lists of types in C++, we refer the interested reader to [14].

[2]More precisely, for each of the logic variable *types*. The library interface is specifically designed to try to ensure that logic variables are declared in such a way that there is a one-to-one correspondence between logic variables and logic variable types. Thus the results of compile-time computations (types) can be meaningfully mapped back into the program (variables).

It suffices to accept as given the list primitives: `TL::NIL`, `TL::CONS`, and `TL::AppendList`. (The namespace `TL` stands for "type list".)

As an example, consider the definition of the `ConjunctRep` class (instances of which are created by the overloaded `&&` operator):

```
template <class LHS, class RHS> struct ConjunctRep : public HasLV {
   typedef typename TL::AppendList<typename LV<LHS>::LVs,
                                   typename LV<RHS>::LVs>::Result LVs;
   LHS lhs;
   RHS rhs;
   ConjunctRep( const LHS& l, const RHS& r ) : lhs(l), rhs(r) {}
};
```

Each `ConjunctRep` is just an expression tree node with a left-hand side and a right-hand side; a `ConjunctRep` computes its list of logic variables as the result of appending the logic variable lists of its two children.

The type expression `LV<Something>::LVs` is a compile-time function used to compute the list of logic variables appearing in `Something`. If `Something` is a `Rep` (which is determined by seeing if it is a subtype of `HasLV`) then the expression just means `Something::LVs`. Otherwise, if `Something` is a non-`Rep`—like `int` (which might appear in a `Rep` tree as the right hand side of the `UnificationRep` created by the LC++ expression `X==1`)—the expression `LV<Something>::LVs` reduces to `TL::NIL` which represents the empty list of logic variable types.

The type lists of logic variables which comprise an `Environment` for a particular query require a canonical representation. To see why, consider again this example client code:

```
   typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
   List<IE> l = lquery( ancestor(Anc,bart,X) );
```

It would be a shame if the client were required to list the logic variable types passed to `QRT` in the same order that they appear in the query—we would like

```
typedef QRT<X_TYPE,Anc_TYPE>::IE IE;   // Note reversal of arguments
List<IE> l = lquery( ancestor(Anc,bart,X) );
```

to also compile. In order to enable this, `QRT` and `lquery()` need to agree on a canonical representation for type lists. It would do us no good if `QRT` created an environment with C++ type

```
Environment<TL::CONS<X_TYPE,TL::CONS<Anc_TYPE,TL::NIL> > > >
```

whereas `lquery()` had

```
Environment<TL::CONS<Anc_TYPE,TL::CONS<X_TYPE,TL::NIL> > > >
```

as its resulting environment type. These two types are conceptually compatible,[3] but the C++ type system sees them as two distinct types which are not interconvertible. With this issue in mind, we can now appreciate one reason[4] for the "unique integer" associated with each logic variable. Recall that logic variables are declared using code like

```
DECLARE( X, int, 10 );
```

The unique integer constant that appears in the type (`10` in the example above) provides a way to *order* the logic variable types. This enables us to create a canonical representation of a set of logic variables as a list: the canonical list always has the types appear in increasing order of their unique integer constants.

The canonicalization process also filters out duplicates, so that queries like

```
lquery( ancestor(Anc,bart,X) && X==1 );
```

do not go mistakenly creating environments with type

```
Environment<TL::CONS<Anc_TYPE,TL::CONS<X_TYPE, // X_TYPE mistakenly
            TL::CONS<X_TYPE,TL::NIL> > > >     // duplicated
```

[3]That is, though we are using type *lists* as a representation type in our meta-program, we actually only care about type *sets* in this case.

[4]The other reason for the "unique integers" is to create the one-to-one type-to-variable mapping mentioned in a previous footnote.

The end result of the above computation is the general semantic checking performed by LC++. The type computers inside `QRT`, the `query()` functions, and the `Rep` classes all work to make the C++ type system ensure that LC++ code is statically checked. The type computers ensure that the environment types match, so that a client cannot ask for, e.g., `env->at(X)` from the result of a query not involving `X`. The type information maintained by LC++ lets the normal C++ type rules check more basic requirements, such as ensuring that in the expression `X==1`, `X` is a `LogicVariable<int>` (and not, say, a `LogicVariable<string>`).

In addition to basic typechecking that `QRT` facilitates, LC++ supports more sophisticated analyses which are specific to the domain of logic programming. One example of such a specialized semantic analysis is detecting one-time-use variables in calls to `lassert()`. Suppose that when the client wrote the code for the family relationship example, instead of writing

```
lassert( child(Kid,Par) -= parent(Par,Kid) );   // correct
```

she accidentally wrote

```
lassert( child(Kid,Par) -= parent(Mom,Kid) );   // oops, used Mom
```

where `Mom` had inadvertantly been used in place of `Par` on the right-hand side. The resulting code is legal and typechecks, but it does not describe the intended `child()` relation.

This type of error is automatically statically detectable because it violates the rule that logic variables appearing in an `lassert()` statement should always appear in more than one location. A logic variable that is used only once can be unified with anything; if the client *does* actually intend to use a "don't care" logic variable, they should do so explicitly using the special variable "`_`". We use meta-programming to write an algorithm which analyzes `lassert()` calls and forces the compiler to emit a warning when one-time-use variables are detected—the same general technique described in Section 2.3.3. (Using meta-programs to statically analyze code and emit domain-specific compiler diagnostics is a technique that has been used by other recent C++ libraries[6, 50, 57].) Domain-specific static analyses like the one described here sets LC++ apart from all other OO libraries for logic programming.

## 3.3 Applications

FIX to be written

## 3.4 Detailed comparison to related work

Whereas there are a number of multiparadigm languages which have logic components, there are relatively few projects which extend existing object-oriented languages with support for logic programming. LC++ is unique compared to those because LC++ cleanly integrates the control flow of the imperative and the logic programming language constructs. These alternative approaches include SOUL [11] (which extends Smalltalk), J/MP [10] (which extends Java), and MPC++ [19] (which extends C++). We first describe some common aspects of those three, and then discuss details of each in turn. (Note that Section 5.2 describes other related work.)

All three approaches (SOUL, J/MP, MPC++) suffer the same key drawback with respect to query results: they do not leave the client in control. In SOUL, the results of a query are returned as a Smalltalk `OrderedCollection` object; this means that examples that involve infinite objects, like `nat` from Section 3.1.5, cannot be realized. The problem is similar in J/MP and MPC++: the client passes in a block of code to be executed for each result produced by the query, and the query executes the client code on each and every result. In contrast, LC++ gives the client control of the query by returning the results as a lazy list; the client can demand a few results, continue on with some other computation, and demand more results later as needed. A second difference between these approaches and LC++ is that none of the other three approaches can duplicate the specialized semantic analyses that LC++ can do (as described in Section 3.2.2).

SOUL, the Smalltalk Open Unification Language adds logic programming features to Smalltalk. The original SOUL system was just an interpreter; clients would specify assertions and queries as strings, using code like

```
rep := SOULRepository new.
rep assert: 'father(homer,bart). father(homer,lisa).'
results := SOULEvaluator eval: 'if father(?dad,?kid)' in: rep.
```

However new work[11] integrates SOUL into Smalltalk so that predicates work like ordinary message-sends and Smalltalk objects can participate in unification, creating a truer embedding. Like LC++, SOUL preserves the declarative syntax that languages like Prolog provide. SOUL provides no static guarantees, however, since Smalltalk is dynamically typed and the SOUL implementation works with the reflection facilities of Smalltalk.

The J/MP language[10] is a Java extension supporting multi-paradigm programming, including logic programming. Logic programming in J/MP is enabled by using the `Relation` class and pass-by-name parameters. The `&&` and `||` operators are overloaded (as in LC++), and the method `unify()` performs unification. J/MP has a weak logic programming model: unification can only be performed on a variable with a value—two unbound logic variables cannot be unified. Also, J/MP defines relations using notation that is more operational than declarative; for example in J/MP one might write

```
public static Relation father( +String Dad, +String Kid ) {
   return parent(Dad,Kid) && male(Dad);
}
```

to define the `father()` relation described in our original LC++ example.

The MPC++ library[19] adds support for logic programming in C++. Like LC++, MPC++ is build atop FC++[48, 49]. MPC++ has been used in a graduate programming languages course to teach students about different programming paradigms. Perhaps as a result of this use-context, MPC++ exposes more implementation details to clients, resulting in verbose code. When declaring an MPC++ predicate, all known facts about the predicate need to be expressed in a closed definition. (This is also true of J/MP.) For example, `male()` would be defined like this in MPC++:

```
class Male : public Logic_Rule {
   Logic_Variable<string> person;
public:
   Male( const Logic_Variable<string>& p ) : person(p) {}
   Logic_Relation Rule_Definition() {
```

```
        return (person |= "bart") || (person |="homer")
            || (person |= "abraham");
    }
};
```

Thus, MPC++ has a static point of definition of all facts pertaining to a predicate, while LC++ allows more facts to be added with `lassert()` dynamically, based on the control flow of the program. A static approach is more amenable to optimizations, but MPC++ does not attempt to optimize queries in any way (nor does J/MP).

Finally, we note that the `Logic_Rule` and `Logic_Relation` classes of MPC++ serve a similar purpose as the `Relation` class in J/MP, and operators are overloaded (MPC++ uses `|=` for unification). Indeed, the implementation strategies of J/MP and MPC++ are quite similar.

## *3.5 Discussion*

LC++ demonstrates that logic programming can be smoothly integrated into C++. The library provides the main features of the logic paradigm and preserves the declarative syntax found in logic languages. The C++ type system provides "atomic" logic data types, and functors can be used to represent unifiable composite structures, just as in logic languages. The library shares C++'s static type checking and supplies its own domain-specific checks via template metaprogramming. FC++ lazy lists are used to mediate the control mismatch between backtracking query logic and normal C++ function calls, enabling paradigms with different views of control flow to peacefully coexist.

We utilized a number of advanced features of C++ to implement both LC++ and FC++ in a way that enables a smooth integration of multiple paradigms. Whereas this chapter and the previous one described a number of the nitty-gritty details of our implementation in C++, in the next chapter we shall take a step back, generalizing our main strategies and looking at which programming language features are most useful for multi-paradigm extensibility.

# CHAPTER IV

# GENERALIZING FROM C++

The previous two chapters describe how we added functional and logic programming features to C++. Whereas C++ is a rich language with a variety of extensibility mechanisms, it is nonetheless not an obvious ideal candidate as a base language to support these kinds of extensions. Nevertheless, by "hijacking" a number of the language's more interesting features, we have added functional and logic programming support to C++, with varying degrees of seamlessness.

Whereas our work in the previous chapters may be described as "novel devices for implementing functional and logic programming constructs in C++", in this chapter we try to remove the qualification "in C++". We discuss our key implementation ideas in more general terms, making explicit what language features (e.g. "overloading") each idea depends on. We also describe some strategies in terms of language features that C++ lacks (such as algebraic datatypes, typeclasses, or `call/cc`). We conclude with a summary of the main limitations that C++ imposes and what could be done differently if those barriers were lifted.

## *4.1 Reusable lessons*

Here we describe our key techniques from a more language-independent point of view, paying special attention to particular language features which make some of these techniques feasible.

Throughout this section we will assume a language that supports parametric polymorphism, as practically every one of our devices depends on this language feature. We will also assume a language that has some kind of way to represent first-class functions. Regardless of whether first-class functions are built-in to a particular programming language or merely representable using other mechanisms (e.g. as classes with overloaded `operator()`

in C++), we shall refer to them as "functoids" throughout this section, so as to simplify the exposition.

### 4.1.1 Type system for higher-order polymorphic functions (using C++-style type inference and template computation)

This is easily the most C++-specific aspect of our work. Given that C++ supports polymorphism only indirectly (via templates, which are not themselves first-class entities), we represent polymorphic functions via functoids. Functoids are first-class objects with templates as members, and our `Sig`s are type-computers which enable the description of entities such as "the result type when function `F` is called with an argument of type `X`". The description of how this is implemented has already been described in Section 2.2.2.

We mention this aspect here only because so many of the other techniques rely on the ability to describe the types of composing polymorphic functoids. Whereas C++ makes this complicated (by virtue of not being able to represent polymorphic functions directly in the type system, and the lack of a `typeof` operator or other general type-inferencing mechanism), most other languages with parametric polymorphism either support polymorphic types directly in the type system or provide type inference (or both).

FIX say more about resultof, typeof, C++, Boost

### 4.1.2 Currying

Currying can be simulated in any language which allows the function call syntax to be overloaded. "Implicit currying", where a functoid can be called with fewer than the expected number of arguments, only requires that function call be overloaded for different numbers of arguments. "Explicit currying", where a functoid can be called with "dummy" placeholders in the place of real arguments requires ad-hoc overloading based on the types of arguments.

Implicit currying is best expressed as a reusable combinator which can be applied to a normal (uncurryable) function. Given an N-argument function `f`, applying the implicit currying combinator (`icc`) to `f` results in a new functoid which contains a reference to `f` and has N different function call overloads, each of which dispatches to a different, specialized function for binding that many arguments. For example, if `f` takes `3` arguments, then the

implicitly curryable version of f (`icf = icc(f)`) defines overloads for multiple numbers of arguments, so that

```
icf( x )
icf( x, y )
icf( x, y, z )
```

are all legal calls, resulting in

```
bind1of3( f, x )
bind1and2of3( f, x, y )
f( x, y, z )
```

respectively. The various `bindMofN` functions must be defined "by hand", but their implementations are straighforward. (Indeed, in a language with `lambda`, the implementations are trivial.)

Explicit currying is also best expressed as a reusable combinator. For example, given a two-argument function `f`, the result of applying the explicit currying combinator (`ecc`) is a functoid `ecf` which has these overloads:

```
ecf( PlaceholderType x, Any2 y )   // ecf(_,y)
ecf( Any1 x, PlaceholderType y )   // ecf(x,_)
ecf( Any1 x, Any2 y )              // ecf(x,y) (normal call)
```

The types `Any1` and `Any2` represent the types of the actual arguments to `f` (which can be generalized into universal types for simplicity), whereas `PlaceholderType` is a distinguished type which just has one instance named "_". Just as with implicit currying, each of these overloads dispatches to the appropriate implementation; in the cases above, they would be

```
bind2of2( f, y )
bind1of2( f, x )
f( x, y )
```

respectively.

In a language like C++, where return-type-deduction for user-defined polymorphic functoids must be specified "by hand", we must also "overload" the return type computation mechanism. In FC++, both "implicit currying" and "explicit currying" are defined in the same combinator class (`FullN`). The `Sig` template in this class has default parameters:

```
template <class F> struct Full3 { ...
   template <class X,
             class Y = PlaceholderType,
             class Z = PlaceholderType>
   struct Sig ...
```

so that we can use it in type expressions with differing arities (e.g. `Sig<int,int,int>`, `Sig<int,int>`, `Sig<int>`). The `Sig` is partially specialized for all different combinations of `PlaceholderType` arguments, so for example

```
template <class X, class Z>
struct Sig<X,PlaceholderType,Z> ...
```

has a `result_type` that is representative of the call

```
f(x,_,z)
```

—that is, the type of the expression

```
bind1and3of3( f, x, z )
```

Note that partial specialization of a class template in C++ is directly analogous to ad-hoc overloading of a function based on argument types.

### 4.1.3 Infix function syntax

Infix function syntax can be simulated in any language which has an ad-hoc overloadable infix binary operator. Infix can also be simulated with *two* overloaded infix operators, without having to resort to ad-hoc-ery.

Recall that in FC++, the expression

```
x ^f^ y
```

means the same as

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

when `f` is a full functoid. The `^` operator has been overloaded in two different ways. The expression

```
x ^f^ y
```

actually parses as

```
(x ^ f) ^ y
```

in C++. The first `operator^` overload accepts any type as a left-hand argument and a functoid as a right-hand argument, and returns a new object of some temporary type `Tmp` which stores these two objects in a temporary data structure. The second `operator^` overload accepts a `Tmp` as a left-hand argument and anything as a right-hand argument, fetches the functoid and first argument stored inside the `Tmp` object, and applies the functoid to both of its arguments. (Note that in C++, by defining the `operator^` overloads as `inline` functions and having the `Tmp` type store references rather than copies, this syntax sugar does not imply any extra cost at runtime.)

The reader may have noticed a potential ambiguity between the two overloads in the previous paragraph. What if the right-hand argument is a functoid? For example, consider this example:

```
negate ^compose^ inc    // f(x) = -(x+1)
```

The expression `negate^compose` matches the first overload, resulting in some temporary object. But then `tmp^inc` matches both overloads: the right-hand argument is a functoid (first overload), and the left-hand argument is a `Tmp` object (second overload). Indeed, this ambiguity must be dealt with; in our original implementation, this expression would cause a compile-time ambiguity error. The desired behavior is to have the second overload take precedence; if the left-hand argument is a `Tmp` object, then we should prefer the second

overload, even when the right-hand argument is a functoid. In C++ the ambiguity between the two overloads can be broken in a number of ways; we use `boost::enable_if` to disable the first overload if the left-hand argument is a `Tmp` object.

It is worth noting that, if *two* suitable overloadable infix operators are available, the ad-hoc-ed-ness and potential ambiguity can be avoided entirely. For example, we can imagine using two different single quotes or two different slashes as the infix operators:

```
x `f` y        or       x \f/ y
```

This also yields the desired facade of infix functions without any appeal to exotic overloading mechanisms; to enable the `x \f/ y` syntax, we can just define the `/` and `\` operators with the appropriate signatures. This is summarized here using Haskell (since Haskell's notation describes things most succinctly):

```
-- using Haskell notation
data Tmp x f = Tmp x f


(\) :: x -> (x->y->r) -> Tmp x (x->y->r)
x \ f = Tmp x f


(/) :: Tmp x (x->y->r) -> y -> r
(Tmp x f) / y = f x y
```

### 4.1.4 Overloaded list interface

As described in Sections 2.3.4 & 2.3.5, FC++ supports three different list datatypes: `List`, `OddList`, and `StrictList`. All three support the same kind of interface, which means that list functions (like `map` and `filter`) can be applied to any kind of list.

In FC++, the common interface to all of the list types is supported via templates and ad-hoc overloading. However the general list interface can actually be expressed using non-ad-hoc methods. Here is a summary of what the interface to `ListLike` entities looks like when expressed via Haskell typeclasses:

```
-- list, even, odd  type constructors; v=value type
class ListLike l e o | l -> e o where
   nil    :: l v
   make1  :: ( ()->e v ) -> l v
   make2  :: ( ()->o v ) -> l v
   cons1  :: v -> e v -> o v
   cons2  :: v -> o v -> o v
   cons3  :: v -> ( ()->e v ) -> o v
   cons4  :: v -> ( ()->o v ) -> o v
   head   :: l v -> v
   tail   :: l v -> e v
   null   :: l v -> Bool
   force  :: l v -> o v
   delay  :: l v -> e v
instance ListLike List    List OddList List where ...
instance ListLike OddList List OddList List where ...
instance ListLike StrictList StrictList StrictList StrictList where ...
```

Each `ListLike` type constructor (`l`) has two associated type constructors which represent the "even" and "odd" versions of the list (`e` and `o`). Note that both `List`s and `OddList`s have "even" tails. `StrictList`s play both the "even" and "odd" roles; since `StrictList`s do not do lazy evaluation, there is no need for a distinction. The `make` functions correspond to C++ constructors. In the cases of both the `make` and `cons` functions, ad-hoc overloading in C++ allows all of the related functions to have the same name, despite having slightly different signatures. (That is, in C++ we don't have four functoids named `cons1` through `cons4`—we just have one functoid named `cons` which is overloaded with all four behaviors.) Note that we are using the notation "`()->a`" to represent the type "thunk returning a value of type `a`", a notion that is only necessary in eager languages (like C++). In the case of `List` and `OddList`, the implementations of `{make1,make2,cons3,cons4}` store the thunk argument to be later evaluated by-need, whereas the `StrictList` implementation evaluates

the thunk immediately.

Given such a definition, it is easy to write functions like `map` that work on any `ListLike` type:

```
-- using Haskell notation, but imagine a strict evaluation semantics
map :: (ListLike l e o)=>  (a -> b) -> l a -> o b
map f l = if null l
          then nil
          else cons4 (f (head l))
                     (thunk2 map f (tail l))
```

If `map` is passed a `List` or an `OddList`, the result is an `OddList` (where only the first element has been evaluated). If `map` is passed a `StrictList`, the result is a `StrictList` where the entire list is evaluated.

### 4.1.5 List optimizations

Section 2.5.2 discusses a number of optimizations we have applied to the implementation of lists and the functions that manipulate them. Here we generalize how a number of these optimizations can be applied in other languages.

#### *4.1.5.1 Caching*

We use "caching" to store lazy list tails so that they may be evaluated "by need". This technique is easy to use in any language with mutable variables. For example, in ML, we would describe the type of `Caches` as

```
datatype 'a Cache = Value of 'a | Thunk of unit->'a
```

and the main operation available on references to this data is

```
fun get cr =
   case !cr of
      Value x => x
      | Thunk f => let val y = f () in
                       cr := Value y; y end
```

That is, if we have a reference to a value, just return the value. If we have a reference to a thunk, call the thunk, update the reference so that it stores the value returned by the thunk, and return the value.

This technique (caching values so they are computed by-need) is useful outside the context of lists, so was have encapsulated it in its own `ByNeed` datatype in FC++, as described in Section 2.3.2.5.

#### *4.1.5.2 Reusing functoids' heap thunks*

For languages which provide update access to the representation of those thunks which are created by binding all of the arguments of a functoid, the "reuser" optimization described in Section 2.5.2.4 is likely to be valuable. Our implementation in C++ uses an extra parameter with a default value as syntactic sugar, but for languages without a default parameter mechanism, it is easy to rewrite the recursive function as two functions: the main function which creates and initializes the mutable thunk and then calls the helper, and the helper function which takes the extra "reuser" parameter and does the work (making recursive calls to itself).

For languages with builtin function types that do not provide access to the representation of thunks, it might be worthwhile to create a (mutable) user-defined datatype for representing thunks. Then the reuser strategy can be used to mutate the user-defined thunks, thus avoiding creating new (builtin) function objects during every recursive call.

Since this optimization avoids repeatedly allocating and freeing small objects, is especially important for C++. The improvement is likely to be less impressive in languages with sophisticated runtimes that have garbage collection and fast small-object allocators. But this optimization will still probably be a win on most systems, as it takes very sophisticated lifetime analysis for compilers to automatically deduce that it is safe to recycle the thunks in the manner done by the reusers.

#### *4.1.5.3 Miscellaneous*

For systems without automatic memory management, using intrusive reference-counting (rather than non-intrusive) will almost certainly be a win.

For languages which are not good at recursion (e.g. do not optimize tail calls), rewriting recursive functions to use iteration instead is likely to be a win.

### 4.1.6 Monad specializers

FIX to be written

summary: whereas languages like Haskell have a type system where the type of an expression can be deduced from outer context, languages like C++ demand that the type of an expression be independent of its context. We use clever template tricks to simulate context-dependent types in C++, and apply them to the specific case of monads.

### 4.1.7 Design pattern implementations

FIX to be written

### 4.1.8 Subtyping for functoids

A number of languages with primitive support for both first-class functions and subtyping have function-subtyping built in. For languages with functoids that don't have built-in support for subtyping, this support can often be simulated via coercion functions/operators. Thus `coerce` can serve as an "upcast" operation, which enables more specific objects to be used in place of more general ones.

For example, if one can overload a function named `coerce()` like so:

```
coerce<Base>( a_derived_obj )  // legal iff a_derived_obj <: Base
```

then, for example, given some representation of a function from `Animal`s to `Car`s, e.g. `Fun1<Animal,Car>`, we can provide a general definition for `coerce` on functoids, so that the original functoid may be coerced into a `Fun1<Dog,Vehicle>` (assuming that `Dog`s can be coerced into `Animal`s and `Car`s into `Vehicle`s).

This can be achieved in any language with just a typeclass-like overloading mechanism. For instance, in Haskell, we might write

```
data Car     = ...
data Vehicle = ...
```

```
data Animal  = ...
data Dog     = ...

class Coercable from to where
   coerce :: from -> to

-- coercion
instance (Coercable d a, Coercable c v)=> Coercable (a->c) (d->v) where
   coerce f = \x -> coerce (f (coerce x) )

-- explicitly define the subtype hierarchies of interest
instance Coercable Car Vehicle where ...
instance Coercable Dog Animal where ...

ac :: Animal -> Car
ac = ...

dv :: Dog -> Vehicle
dv = coerce ac       -- demonstration of "upcasting" a function

v :: Vehicle
v = dv Dog           -- calls "ac"
```

The instance declaration `Coercable Foo Bar` can be used to explicitly specify that `Foo` is a subtype of `Bar`; the instance declaration involving function types generalizes this notion to functions.

### 4.1.9 Lazy lists as an interface to logic query results

As described in Section 3.2.1, lazy lists are useful as the interface to results of a logic query. The lazy list interface leaves the client algorithm in control, both of the *evaluation* of the

query (e.g. the client may just ask for two results, and then stop) and of *how the results are used* (e.g. the client may print them, store them in a data structure, whatever).

From an implementation point-of-view, lazy lists nicely encapsulate the fact that a natural implementation of logic queries uses continuation-passing style (CPS). Logic query terms all take a `future` parameter describing "the rest of the query" to run, but lazy lists enable these computations to actually `return`. When a query succeeds in finding a result, a one-element lazy list is returned; when a query fails, an empty lazy list is returned; at the "choice points" (disjunct terms), the results of all possible futures for the query are lazily concatenated—thus each future will only be run as the next solution is demanded by the client. The implementation strategy described in Section 3.2.1 can be used in any language which supports an implementation of lazy lists.

It should be noted that for languages which provide a `call/cc` primitive, an alternative implementation is straightforward to provide. A query result can just be represented as a

```
Maybe< pair< Answer, Continuation > >
```

When `query()` is called by some client, the client's current continuation is passed as a the original `future` parameter to the query object. Each time the query finds a result, it invokes the client continuation with a `pair` containing the result and the query's own current continuation. This passes control back to the client, who uses the answer however it sees fit. When the client wants the next result, the client passes its own current continuation to the second element of the `pair`; this transfers control back to the query, which proceeds where it left off. When the query has no results left to produce, it calls the client's continuation with `Nothing`, to signify that it is done.

The analogy between the lazy list implementation and the `call/cc` implementation is striking. Both implementations provide a way for the client and the query to coroutine with one another. When the two datatype implementations:

```
Maybe< pair< Answer, Continuation > > // call/cc implementation
List<Answer>                         // lazy list implementation
```

are expanded one level into their algebraic equivalents:

```
Just pair< Answer, Continuation >   | Nothing
Cons       Answer  Thunk            | Nil
```

the similarity is more obvious. The main difference is that in the `call/cc` implementation, the query "calls" back to the client, whereas in the lazy list implementation, the query "returns" to the client.

#### 4.1.10 Functoids as mechanism for logic code to "call out"

Any language which supports functoids can provide a natural mechanism for logic programming code to make function calls back out to the host language. In LC++,

```
SomeLogVar.is( SomeFunctoid, Arg1, ..., ArgN )
```

creates a logic term which, when run, calls

```
SomeFunctoid( Arg1, ..., ArgN )
```

(using the values of the current query environment for any arguments involving logic variables) and unifies the result with `SomeLogVar`.

#### 4.1.11 Domain-specific static analyses

Both FC++'s "lambda" and LC++ use the C++ technique known as "expression templates" to create a domain-specific embedded language. Supporting domain-specific static analyses helps fortify both FC++ and LC++ as "embedded languages" rather than mere "libraries".

Creating domain-specific static analyses typically requires the ability to do arbitrary computation at compile-time. As a result, mechanisms like C++ templates or Scheme macros are probably a necessity to do this well. For example, warning about one-time-use variables in `lassert()`s (Section 3.2.2) requires the ability to walk an expression tree at compile-time and count the number of occurances of each logic variable. We do this in C++ by representing the entire structure of the expression within the type system, and inspecting and doing computations on this type at compile-time.

In addition to a Turing-complete macro system of some sort, a language supporting domain-specific static analyses would ideally have a nice mechanism for reporting errors or warnings when they are detected. C++ has no such mechanism—to report information at compile-time, one must hijack the existing compiler diagnostic mechanisms. Often this takes the form of instantiating a "broken" template with a type named by a really long identifier which describes the error: the compiler then reports an error in the template, and hopefully includes the long identifier in part of its diagnostic message as the only clue to the user as to what really went wrong. This mechanism is adequate, and variations of the basic theme can probably be used in any language with a Turing-complete macro system. The wise language designer would do well to provide a special mechanism by which macros can create well-formatted warning/error messages that can refer to type information and file/line numbers.

## *4.2 Capabilities and limitations of C++*

FIX to be written

Have a table in CH 4 - constructs needed, feature, C++ features used - maybe break into features that do/don't use TMP (turing tarpit) and implications for lang designers

# CHAPTER V

# RELATED WORK

Much has been written about multi-paradigm programming within a single language. Here we summarize some of the most relevant related work, broken into subsections by topic.

## *5.1 Work adding functional components to object-oriented languages*

Many researchers have worked on adding functional paradigm components to object-oriented languages, especially C++. Their work can be roughly divided into three categories: representing functions, lambda, and applications of functional techniques.

### 5.1.1 Representing functions in C++

Representing functions as first-class objects in C++ has been a popular research topic. Läufer's paper [45] 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 [17] implements currying in C/C++/Objective-C and shows the utility in applications. His implementation requires modification of the compiler, though. The utility comes mostly in C; in C++, more sophisticated approaches (such as ours) can achieve the same goals and more.

Kiselyov [42] 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. FC++'s lambda uses expression templates to provide such sugar without resorting to macros.

The C++ Standard Template Library (STL) [59] includes a library called `<functional>`. It supports a very limited set of operations for creating and composing functoids that are usable (in monomorphic form) with algorithms from the `<algorithm>` library. While it

serves a useful purpose for a number of C++ tasks, it is inadequate as a basis for building higher-order polymorphic functoids.

Läufer [45] presents a framework for supporting functional programming in C++. His approach supports lazy evaluation, higher-order functions, and binding variables to different function values. His implementation does not include polymorphic functions, though, and also uses an inefficient means for representing function objects. In many ways, our work on FC++ 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.

Alexandrescu's book [2] contains a chapter on "generalized functors". These functors are similar to our indirect functoids, except that they do not support implicit currying or subtype polymorphism. In another chapter, Alexandrescu also describes reference-counting mechanisms, including intrusive ref-counts, like the ones we use with FC++'s internal reference-counted pointers.

The Boost function library [28] provides function objects similar to FC++ indirect functoids. These function objects support subtype polymorphism and reference parameters, and are the basis for a proposed extension to the C++ standard library.

### 5.1.2 Lambda

Here we briefly compare our approach to implementing lambda to that of the other major lambda libraries for C++: the Boost Lambda Library (BLL)[35] and FACT![60].[1] (Note that Section 5.1.3 mentions how Java's anonymous inner classes effectively provide "lambda" for Java.)

[1] 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.

#### *5.1.2.1 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). Lambda is implicit rather than explicit; a reference to a placeholder variables (like `_1`) turns an expression into a lambda on-the-fly. This makes it impossible to represent some anonymous functions (involving nested lambda) using BLL. Custom error message which diagnose common statically-detectable errors are also absent from BLL.

Apart from special support for function composition and explicit currying (binding) of function arguments, BLL lacks syntactic support for other useful functional constructs like `letrec` or monadic comprehensions. There is also no support for naming the type of lambda expressions (like our `LEType`). On the other hand, when effects are desired, BLL can be used to describe functions with side-effects more gracefully than FC++. Whereas FC++ lambda expressions can only have effects by dereferencing pointers, BLL lambda expressions can directly manipulate object references and create lambdas which take mutable reference parameters.

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 effect-ful 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.

#### *5.1.2.2 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 just one primitive control construct in its lambda sublanguage ("`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++.

Like BLL, FACT! contains no facilities for manipulating monads, naming the types of lambdas, or doing static analyses and issuing custom error messages.

### 5.1.3 Applications

Certainly a lot has been written about language support for implementing design patterns (e.g., [4, 13]), functional techniques in OO programming, etc. Some of the approaches in the literature are even very close in philosophy to our work. For instance:

- Alexandrescu [2] demonstrates how the meta-programming capabilities of the C++ language can be used to yield elegant pattern implementations.
- Kühne's dissertation proposes several patterns inspired by functional programming [44].
- Using functional techniques (higher-order functions) to implement the Observer and Command patterns is common—in fact, even standard practice in Java and Smalltalk.
- The benefits of polymorphic and higher-order functions have often been discussed in the functional programming literature [62].

Alexandrescu [2] offers a mature C++ implementation of the Abstract Factory pattern. His approach consists of a generic (i.e., polymorphic) Abstract Factory class that gets parameterized statically by all the possible products. It is worth noting that this is the exact scenario that Baumgartner et al. [4] studied. Their conclusion was that meta-object protocols should be added to OO languages for better pattern support. Thus, Alexandrescu's implementation is a great demonstration of the meta-programming capabilities of C++—the language's ability to perform template computation on static properties can often be

used instead of meta-object protocols.

Géraud and Duret-Lutz [25] offer some arguments for redesigning patterns to employ parametric polymorphism. Thus, they propose that parametric polymorphism be part of the "language" used to specify patterns. In contrast, our approach is to use parametric polymorphism with type inference in the *implementation* of patterns. From an implementation standpoint, the Géraud and Duret-Lutz suggestions are not novel: they have long been used in C++ design pattern implementations. Furthermore, the examples we offer in Section 2.4 are more advanced, employing type inference and manipulation of polymorphic functions.

The Pizza language [56] integrates functional-like support to Java. This support includes higher-order functions, parametric polymorphism, datatype definition through patterns, and more. Pizza operates as a language extension and requires a pre-compiler. Support for parametric polymorphism in Java has been a very active research topic (e.g., [1, 8, 55, 63]), and a solution based on GJ [8] has been recently adopted [7]. Type inference is used in GJ. Nevertheless, due to the GJ translation technique (erasure) it is not possible to extract static type information nested inside template parameters.

It should be noted that Java inner classes [36] are excellent for implementing higher-order functions. Inner classes can access the state of their enclosing class, and, thus, can be used to express *closures*—automatic encapsulations of a function together with the data it acts on. Java inner classes can be anonymous, allowing them to express anonymous functions—a capability that is not straightforward to emulate in C++. Many of our observations from Sections 2.4.1 and 2.4.2 also apply to Java. In fact, the most common Java implementations of the Command and Observer design patterns use inner classes for the commands/callbacks.

## 5.2 Work on multiparadigm languages with logic components

There has also been a bit of work on multiparadigm languages containing a logic programming component (that is, languages which support both object-oriented and logic programming ("OO-Log"), or both functional and logic programming ("Fun-Log"), or all

three paradigms). Here we compare our work with other work that extends OO languages with a logic component, as well as survey other multiparadigm work with logic components.

### 5.2.1 Logic programming extensions to OO languages

Section 3.4 provides a detailed description of how LC++ compares to other projects which add support for logic programming to existing object-oriented languages. Here we summarize that comparison.

The three most-related approaches are SOUL (Smalltalk), J/MP (Java), and MPC++ (C++). LC++ is unique compared to such work because it integrates cleanly the control flow of the imperative and the logic programming language constructs. All three of the other approaches do not leave the client in control after a query: either all the results are returned as a collection, or a block of code is executed once on each result. In contrast, LC++ returns the query results as a lazy list, leaving the client in control of consuming and processing the results.

There are two other notable differences between our work and these other three. The first is that the other approaches have a more "operational" syntax for defining facts and rules and performing queries, whereas LC++ preserves the declarative syntax found in logic languages like Prolog. A second difference between these approaches and LC++ is that none of the other three approaches can duplicate the specialized semantic analyses that LC++ can do.

### 5.2.2 Other multiparadigm programming which includes a logic-programming component

There are quite a few recent examples combining functional and logic programming. Gödel[27], Escher[20], Curry[15], and Toy[64] are a few examples of languages in this area. Each of these languages features static typing, polymorphism, and a pure (effect-free) style of programming. Mercury[53] fits into this group, but it also supports parameter mode and determinism declarations.

These Fun-Log languages tend to differ from LC++ with regards to operational semantics: whereas LC++ requires that functions be applied only to bound logic variables, these

languages are lazy, and simply suspend computations until the variables get bound. Some of these Fun-Log languages, like Curry, also encapsulate the search strategy, so that in addition to the depth-first (Prolog-style) search that LC++ supports, other strategies (such as breadth-first) can be used as drop-in replacements for the logic search engine.

Unlike LC++, which combines logic and object-oriented programming by adding logic programming features to an existing OO language, some approaches start with a logic programming language and extend it with object-oriented features. One example in this area is Jinni[37]. Jinni is an interpreter for an extended version of Prolog (which includes features like classes and inheritance) that is written in Java. Jinni uses Java's reflection capabilities to provide a mechanism for the logic code to "call out" to Java, but the interface is heavy and there is a clear deliniation between logic code and object-oriented code.

A few languages are designed to support all three (logic, functional, and OO) paradigms. The language Oz[54] combines all three paradigms in a dynamically typed, concurrent programming language. Oz does have a strong object model, but logic is the dominant paradigm and programming with Oz looks and feels more like logic programming than OO programming. The Leda[9] language was specifically designed as a language for teaching the three paradigms; Leda is statically typed. Both J/MP[10] and Leda were created by the same man, and they share many of the same strengths and weaknesses.

# CHAPTER VI

# CONCLUSIONS

FIX to be written

Sum up interesting points of the work

Talk about interesting future directions

# REFERENCES


[1] O. Agesen, S. Freund, and J. Mitchell, "Adding type parameterization to the java language", *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1997*, 49-65.

[2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.

[3] S.R. Alpert, K. Brown, B. Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998.

[4] G. Baumgartner, K. Läufer, V.F. Russo, "On the interaction of object-oriented design patterns and programming languages", Tech. Report CSD-TR-96-020, Dept. of Comp. Sci., Purdue University, February 1996.

[5] Boehm-Demers-Weiser conservative garbage collector `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

[6] The Boost website: `http://boost.org/`

[7] G. Bracha, "Add generic types to the Java programming language", Java Specification Request (JSR) 14, Sun Microsystems, 1999.

[8] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, "Making the future safe for the past: Adding genericity to the Java programming language", *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1998.*

[9] T. Budd. *Multiparadigm programming in Leda.* Addison-Wesley, Reading, Massachusetts, 1995.

[10] T. Budd. "The Return of Jensen's Device", *Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Malaga, Spain, June 2002.

[11] J. Brichau, K. Gybels, and R. Wuyt. "Towards Linguistic Symbiosis of an Object-Oriented and a Logic Programming Language", *Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Malaga, Spain, June 2002.

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

[13] C. Chambers, B. Harrison, and J. Vlissides, "A debate on language and tool support for design patterns", *ACM Symposium on Principles of Programming Languages*, 2000 (PoPL 00).

[14] K. Czarnecki, and U. Eisenecker, *Generative Programming.* Addison-Wesley, 2000.

[15] Curry: A Truly Integrated Functional Logic Language
`http://www.informatik.uni-kiel.de/~mh/curry/`

[16] F. Dabrowksi, and F. Loulergue. "Functional Bulk Synchronous Parallel Programming in C++." *Applied Informatics 2003, Symposium on Parallel and Distributed Computing and Networks (PDCN 2003).*

[17] 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.

[18] B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. "An overview of HAL." *Proceedings of Principles and Practice of Constraint Programming*, pages 174–188, October 1999.
`http://www.csse.monash.edu.au/~mbanda/hal/index.html`

[19] S. Edwards. "MPC++ Resources", *class web page*, available at
`http://courses.cs.vt.edu/~cs5314/Spring02/mpcpp.php`

[20] The Escher programming language
`http://www.cs.bris.ac.uk/~jwl/escher.html`

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

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

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[24] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'03)*, pp.115-134. ACM Press, Oct. 2003.

[25] T. Géraud and A. Duret-Lutz, "Generic programming redesign of patterns", in Proc. *European Conf. on Pattern Languages of Programs, 2000 (EuroPLoP'2000)*.

[26] The Glasgow Haskell Compiler homepage:
`http://www.haskell.org/ghc/`

[27] The Gödel Programming Language
`http://www.cs.bris.ac.uk/~bowers/goedel.html`

[28] D. Gregor. The Boost function library.
`http://www.boost.org/doc/html/function.html`

[29] J. de Guzman, et al. The Boost Spirit Library. Available at
`http://www.boost.org/libs/spirit/index.html`

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

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

[32] The Hugs homepage:
`http://www.haskell.org/hugs/`

[33] G. Hutton, and E. Meijer. "Monadic parsing in Haskell" *Journal of Functional Programming*, 8(4):437-444, Cambridge University Press, July 1998.

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

[35] J. Järvi and G. Powell. The Boost Lambda Library. Available at
`http://boost.org/libs/lambda/doc/index.html`

[36] Javasoft, *Java Inner Classes Specification*, 1997. In
`http://java.sun.com/products/jdk/1.1/docs/` .

[37] Jinni: Java INference Engine and Networked Interactor.
`http://www.binnetcorp.com/Jinni/`

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

[39] S. P. Jones and P. Wadler. "Imperative functional programming," *20th Symposium on Principles of Programming Languages*, ACM Press, Charlotte, North Carolina, January 1993.

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

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

[42] 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/` .

[43] S. Krishnamurthi, M. Felleisen, D. P. Friedman, "Synthesizing object-oriented and functional design to promote re-use", *European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

[44] T. Kühne, *A Functional Pattern System for Object-Oriented Design*, Verlag Dr. Kovac, Hamburg, 1999.

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

[46] The LC++ web page:
`http://www.cc.gatech.edu/~yannis/lc++/`

[47] J. Maddock. Boost library: static assertions. Available at
`http://boost.org/libs/static_assert/static_assert.htm`

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

[49] B. McNamara, and Y. Smaragdakis. "Functional Programming with the FC++ library" *Journal of Functional Programming*, to appear.

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

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

[52] 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` .

[53] The Mercury Project. http://www.cs.mu.oz.au/research/mercury/

[54] The Mozart Programming System. http://www.mozart-oz.org/

[55] A. Myers, J. Bank and B. Liskov, "Parameterized types for Java", *ACM Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

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

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

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

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

[60] J. Striegnitz, *FACT! The Functional Side of C++*, http://www.fz-juelich.de/zam/FACT.

[61] 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.

[62] S. Thompson, "Higher-order + polymorphic = reusable", *unpublished*, May 1997. Available at: http://www.cs.ukc.ac.uk/pubs/1997/224 .

[63] K. Thorup, "Genericity in Java with virtual types", *European Conference on Object-Oriented Programming (ECOOP) 1997*, 444-471.

[64] The Toy System. http://titan.sip.ucm.es/toy/

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

[66] T. Veldhuizen. "Expression Templates," *C++ Report*, Vol. 7 No. 5 June 1995, pp. 26-31. See also
`http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html`

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

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

[69] P. Wadler. "Comprehending monads," *Mathematical Structures in Computer Science*, Special issue of selected papers from 6th Conference on Lisp and Functional Programming, 2:461-493, 1992.

[70] P. Wadler. "Monads for functional programming." J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.