Living with λ 's

Functional Programming in C++

alfons haffmans

April 18, 2013

Introduction

Functional programming and C++. The combination will strike an equal mixture of disgust and terror in some of you. Others may be intrigued and daunted by the prospect.

Yet C++ has always been a multi-paradigm language [1]. Recent additions to the standard, like lambda's, have improved the support for functional programming [11]. In fact, previous attempts to add functional programmingfeatures required ignificant effort [2]. This paper explores the support out-of-box for functional programming provided by the new standard We'll like at techniques typically found in introductory functional programming textbooks [5, 6, 7]. This article assumes familiarity with C++, but not nescesarily with basic functional programming.

The source code is available on github [12] and is compiled using gcc 4.8 installed on Mac OSX using MacPorts [13].

Object Oriented and Functional Programming Style

The heart of object-oriented programming is the encapsulation of data and methods in a coherent class or object. Each class or object represents an entity in the real world. Each object is resposible for the managment of it's state and as long as it fulfills the contract implied by it's interface the implementation is of no concern to the caller. Objects interact by sending each other messages through method calls which change the internal state of the receiving object. Classes can be combined through inheritance or composition to form more complex entities [10].

The for-loop is a typical construct used in C++ classes. The for-loop processes elemenst in a container. These elements can be object instances or pointers to object instances. Usually the forr-loop uses iterator to point to the next element to process in its body. The body of the for-loop typically has statements which affect the state of the element referenced by the iterator. When the for-loop reaches the end of the container all elements have been processed and some or all of them have been modified in some way. Any reference to the list acquired before the for-loop was executed will now reference the changed list. The same thing goes for references to elements in the list. So the execution of the for-loop may cause side-effects in other parts of the program, either by design or by

accident. The type of programming which emphasizes the use of mutable data and statements is called an imperative programming style. It is hard to prove by simple statement inspection alone if an imperative program is correct, because its state maybe affected by changes away from statement being reviewed.

By contrast functional programming stresses the construction of computations or functions acting on immutabe values. Data and operations on the data are not comingled. Immutable data acts like a value like 1. You can hold a refeence to 1, but 1 itself is immutable. You can add 2 to the reference but the reference itself still points to 1. This referential transparancy through the use of pure functions and immutable data lies at the heart of fuunctional programming.

Functions are first-class objects in a functional language. You can reference a function like you would any other data. Functions can have functions as arguments or return functions. Functions that take functions as arguments or which return functions are called higher order fucntions. You can use higher-order functions to combine simpler functions into more complex ones. They play an important role in functional programming. for-loops are replaced by recursion for list processing [6, 7].

Because functions play such an important role we need a formal way to represent them. This article uses Haskell's notation for function signatures [6]. A binary function *f* with arguments of type *a* and *b* and a return value of type *c* is represented :

$$f :: (a, b) \rightarrow c$$

The representation of function implementations use a slightly different notation: the return type follows the double colon **::** after the argument list. Here's the type signature of the identity function :

$$id :: a \;\; \rightarrow \;\; a$$

Here are two implementations if id :

$$id(int\ x) :: int \;\; = \;\; x$$
$$id(Person\ p) :: Person \;\; = \;\; p$$

In a function defintion $a \rightarrow b$ the arrow $\rightarrow$ can be looked as a type which takes two other types a and b to be fully defined. Types that are paramatrized by other types, like the arrow operator $\rightarrow$ are refered to as type constructors [5].
In general $M\ a$ represents a type constructor M which takes a single type variable a, and $M\ a$ corresponds to the C++ class template $\mathbf{template} <$ typename **a** $>$ struct **M**..... The arrow operator corresponds to the function wrapper $std :: function < a(b) >$ [11]. An other frequently used type constructor is $[\ a\ ]$ creates a list of elements of type a. $[a]$ corresponds to the stl containers $std :: list < a >$ or $std :: forward\_list < a >$ [11].

Lambda Expressions and Closures

Lambda expressions allow you to create functions on-the-fly. The expression in the body of the lambda can reference variables which are not specified in the argument list of the lambda expression. Those variables are called free variables. Free variables are assigned the value found in the environment (i.e. the scope) in which the lambda expression is defined [8]. This capture of the enclosing environment by the lambda expression is called a closure [8, 9].
The (slightly abbreviated) C++ syntax for the lambda expression is [14]:

$$[...]\ (params)\ mutable\ \rightarrow\ rettype\ \{\ body\ \}$$

The capture specifier $[...]$ specifies how the free variables are captured. If it's empty $[]$, the body of the lambda can't reference any variables outside its scope. The $[=]$ specifier captures free variables by value, wheras the $[\&]$ captures them by referrence. The $(\ params\ )$ is the parameter list and $\rightarrow\ \ rettype$ is an optional return type specifier. Lamda's can be bounf to variables ussing using $std :: function$ [15] or auto [16].

```
1 [...]
2   int x = 0;
3   int y = 42;
4   auto func = [x, &y] () { std::cout << "Hello world from lambda
        : " << x << "," << y << std::endl; };
5   auto inc = [&y] () { y++; };
6   auto inc_alt = [y]  () mutable { y++; };
7   auto inc_alt_alt = [&]  () { y++; x++; };
```

```
8
9   func(); //prints: Hello world from lambda : 0,42
10  y = 900;
11  func(); //prints:  Hello world from lambda : 0,900
12
13  inc();
14  func(); //prints : Hello world from lambda : 0,901
15
16  inc_alt();
17  func(); //prints: Hello world from lambda : 0,901
18
19  inc_alt_alt();
20  func();//prints: Hello world from lambda : 0,902
21
22  std::cout << " x :" << x << "; y :" << y << std::endl;  // x
        :1; y :902
```

Listing 1: various ways lambda's capture the environment

Listing 1 illustrates the use of the capture specifier. IThe lambda $func$ has no arguments and prints the value of the two free variables x and y to stdout. x and y are initialized to 0 and 42 respectively preceding the lambda definition. The capture specifier of $func$ is $[x, \&y]$ so x is captured by value and y be reference. Tne next three lambda's increment the free variables x and y. The lambda inc captures y by reference. inc_alt on the other hand captures y by value. The keyword $mutable$ allows the lambda expression to change y. inc_alt_alt captures the complete environment by reference, and increments both x and y. Then $func$

is called each time y is changed. The values of x and y printed by func are shown in the comment. Since y is captured by reference is can be changes through side effects. On the other hand x is captured once and remains the same.

```
1  [...]
2    // as opposed to [=] or [] or [!]
3    std::function<int (int)> factorial = [&factorial] (int x) ->
         int {
4      std::cout << x << ",";
5      if (x == 0) return 1;
6      return x * factorial(x-1);
7
8    };
9    auto res = factorial(10);
10   std::cout << std::endl;
11   std::cout << "res : " << res << std::endl;
12   //prints : 10,9,8,7,6,5,4,3,2,1,0,
13  //      res : 3628800
```

Listing 2: implementation of factorial using lambda recursion

Listing 2 shows a recursive implemention of the factorial factorial function $n!$. Each invocation of the lambda prints the value of the argument x. The return type is specified using the optional return specifier. Notice that the lambda itself needs to be captured by reference.

Partial Function Application

A partially applied function is created when a function is called with fewer arguments than it's argument list requires. In that case a lambda is returned with the remainder of the arguments [8]. In C++ partial function application is supported by *std::bind* [17] and std::placeholders::... [18]. Both are defined in the header $< functional >$. The placeholders are in the namespace $std :: placeholders$ and are named $\_1$, $\_2$ etc.

std::bind takes a callable object or a function pointer as it's first argument. Subsequent arguments are either values, or placeholders provided by std::placeholders. std::bind returns a function object. The relative position of the values and placeholders corresponds to the position of the argument in the argument list of the function f to which tey're bound. A place holder corresponds to an argument of the callable returned by std::bind. The number of arguments is equal to the number of distinct place holders.

```
1 [...]
2
3   auto repeat = [](int n, double y, std::function<double(double)
      > f) {
4     while (n--> 0) {
5       y = f(y);
6     }
7     return y;
8   };
9
```

```
10  auto rpl    = std::bind (repeat,
11          std::placeholders::_1,
12          std::placeholders::_1,
13          std::placeholders::_2);
14
15  std::function<double (double)> l1     = [](double x) { return
        2*x-0.906;};
16  auto val = rpl(9, l1);
17  std::cout << " result : " << val << std::endl; // print
        4145.03
```

Listing 3: std::bind example

Listing 3 illustrates the use id std::bind and std::placeholders. It's used to create a function which calls an other function repeatedly with the result of the previous function call. Lambda $repeat$ is a higher order function which repeatedly calls it's third argument. This function is initially called with the value of the 2nd argument. The number of repititions is given by the first argument. std::bind is used to create a function which uses te number of repitions as the initial value. The callable object *rpl* returned by $std :: bind$ uses the number of repeats as the initial value because the first and second argument of repeat are bound to the same place holder. $rpl(l1, 9)$ calls calls lambda l1 nine times, with 9 as the inital value. The result is printed to stdout, and it's value is shown in the comment.

Currying

Currying (named after the mathematician Haskell B. Curry) is a technique which turns any function into function of one variable [8]. Currying is related to but is more powerful than partial function application. The curried version of a function is a higher order function which returns a partially applied version of the orginal function.

The function signature of a function designed to curry binary functions $f(x, y)$ is:

$$curry2 \ :: \ ((a, b) \to c) \ \ \to \ \ (a \to b \to c)$$
$$f :: (a, b) \to c \ \ \Rightarrow \ \ (curry2 \ f) :: a \to b \to c$$

curry2 takes a binary function and returns a unary function. This unary function returns another unary function when it is called with an argument of type a. This function is a partially applied version of the uncurried fucntion f, where the argument a is provided When you call this function with an argument of type b you obtain the value returned by uncrried function f. Here's an example where $plus$ is being curried:

$$plus \ : : (int \ x, int \ y) \ :: int = x + y \ \ \Rightarrow \ \ cplus(int \ x) \ :: \ (int \to int)$$
$$\to (int \ y) \ :: \ int \to x + y$$
$$plus(5, 6) = 11 \ \ \Leftrightarrow \ \ (curry2 \ plus)(5)(6) = 11$$

(curry2 plus) is the curried version of plus. The return types have been shown explicitly to highlight the fact that functions are returned. *curry2 plus)(5)* returns a lambda which represents the $plus$ function partially applied to 5. This is then called with 6 as the argument with an unsuprising 11 as the result. A simple implementation of curry2 to curry binary functions in C++ is shown in listing 4:

```
1 template <typename R, typename T, typename U>
2 std::function<std::function<R (U)> (T)> curry(std::function<R (T
      ,U)> op)
3 {
4   return [=] (T x) { return [=] (U y) {return op(x, y);};};
5 }
6   auto l = curry<int,int, int> ([](int x, int y) { return (5 + x
        ) * y;});
7   std::cout << l(1)(1) << std::endl; //prints 6
```

Listing 4: curry for binary operators

Currying and partial function application simplify the design of higher order functions since we only have to consider unary functions. In fact currying plays an important role in functional programming **??**.

Neither the C++ language nor its standard library provide facilities to curry functions. In fact, in C++ functions are not written in curried form. Compare this to Haskell whare functions are curried by default [5, 6]. The programmer needs to either use a third-party library or roll her own implementation. Writing a curry operator has become a lot easier now that lambda's are supported.

Map, zipWith and Zip

map applies a function f of type $a \to b$ to each element of a list $[a]$ and returns a new list $[b]$.

$$map \;::\; (a \to b) \to [a] \to [b]$$

The std::for_each function appears to fit the the bill **??**. It takes two iterators and a unary callable object as input. The callable is called with every element in the range delimited by the iterators, and its final state is returned. std::for_each is clearly an imperative implementation of a for-loop.
A better coice is the std::transform functiob. This function has two forms **??**. The first one takes a unary callable object as input and applies it to the elements of one range and returns an other.

$$\begin{aligned}
&\textbf{template} < && \textbf{typename}\ input_container_iterator,\\
&&& \textbf{typename}\ output_container_iterator,\\
&&& \textbf{typename}\ unary_operation >\\
&output_container_iterator && \textbf{transform}\\
&&& (input_container_iterator\ begin_1,\\
&&& input_container_iterator\ end_1,\\
&&& output_container_iterator\ destination_1,\\
&&& unary_operation\ unary_{op});
\end{aligned}$$

In the second it takes a binary callable function, applies it to two input ranges,

and returns the resulting range :

$$
\begin{aligned}
\textbf{template} < \quad & \textbf{typename } input\_container\_iterator_1, \\
& \textbf{typename } input\_container\_iterator_2, \\
& \textbf{typename } output\_container\_iterator, \\
& \textbf{typename } binary\_operation > \\
output\_container\_iterator \quad & \textbf{transform} \\
& (input\_container\_iterator_1\ begin_1, \\
& input\_container\_iterator_1\ end_1, \\
& input\_container\_iterator_2\ begin_2, \\
& output\_container\_iterator\ destination_1, \\
& binary\_operation\ binary_{op});
\end{aligned}
$$

Listing 5 shows a possible implementation of the map function for a std::forward_list.

```
1 template<typename A, typename F>
2 auto map (F f, const std::forward_list<A>& L) -> std::
      forward_list<decltype(f(A()))>
3 {
4   std::forward_list<decltype(f(A()))> H;
5   std::transform(L.begin(), L.end(), std::front_inserter(H), f);
6   H.reverse();
7   return H;
8 }
```

Listing 5: map for std::forward_list

Notice that map has two type parameters. The first variable specfies the type of the element in the input container L. The second type paramater specifies very

generic callable. std::function could have been used to provide a more typesafe interface. However that would not allow us to use inline lambda functions. The type of each lambda is unique and therefore would not be converted to std::function. The type of element in the result list is determined using decltype **??** on the return type of the callable f.

```
1  [...]
2  std::function< std::function <int(int)>(int)> cplus  = [] (int
       x) {
3     return [=] (int y) {
4       return 4 * x + y;
5     };
6   };
7
8   auto l = std::bind([](std::function<int(int)> f){return f(2);},
9            std::bind(cplus, std::placeholders::_1));
10
11   map(show, map(l, L)); //prints 6,270,358,94,182,6,14,398,-358,
```

Listing 6: using std;:bind to combine functions

In listing 6 std::bind combines two functions and the result is then mapped over the list. The inner bind takes the curried plus function $cplus$ as the first argument and puts a place holder as the second argument. The lambda returned by the inner bind is then used as an input to the outer lambda. The first argument of the outer bind is a lambda which has a function as input. In the body of the lambda this funtion is called with 2. The place holder is bound to the first

argumen of $cplus$ and 2 is used as the value for the second argument. So in affect the function $f(x) = 4 * x + 2$ is mapped over the list. The result is printed to std::cout. 94 the result of 4 * 23 + 2, 182 the result of 4 * 45 + 2 etc. This shows how function combination can be used to limit the number of iterations and list copies.

The second flavor of std:: transform applies a function to the elements of two lists to produce a third. This corresponds zipWith **??**:

$$zipWith :: (a \to b \to c) \to [a] \to [b] \to [c]$$

$$zip :: [a] \to [b] \to [(a, b)]$$

The first argument of zipWith is a curried function, with input parameters of type a and b respectively and return type c. This function is applied to a list of elements of type a and b respectively. The result is a list of type c. A closely related and widely used function is zip [23] which takes two lists and returns a list of pairs.

```
1 template<typename A, typename B, typename F>[caption=zipWith and
      zip implemented with std::transform, label=zipWith]
2 auto zipWith (F f, const std::forward_list<A>& L, const std::
     forward_list<B>& R) -> std::forward_list<decltype(f(A(),B()))
     >
3 {
4   std::forward_list<decltype(f(A(),B()))> H;
5   std::transform(L.begin(), L.end(), R.begin(), std::
       front_inserter(H), f);
```

```
6   H.reverse();
7   return H;
8 }
9 template<typename A, typename B>
10 std::forward_list<std::tuple<A,B>> zip (const std::forward_list<
      A>& L, const std::forward_list<B>& M)
11 {
12   return zipWith([] (const A& a, const B& b) {return std::
         make_tuple(a,b);}, L, M);
13 }
```

Listing **??** shows the implemention of zipWith and zip for a std::forward_list using std::transform. The type of the return list is derived by calling decltype on the function f, which is called with an instance of A and B.

```
1    [...]
2   std::forward_list<int>  L = {1,67,89,23,45,1,3,99,-90};
3   std::forward_list<char> R = {'a','b','l','u','t','v','r','6','
        h'};
4
5   auto H2 = zip (L, R);
6   map([] (std::tuple<int,char> v) { std::cout << v << ",";
        return v;},H2);
7  //prints : (1,a),(67,b),(89,l),(23,u),(45,t),(1,v),(3,r),(99,6)
       ,(-90,h),
```

Listing 7: zipping two lists

Listing 7 illustrates the use of zip on two lists.

```
1  template<typename A, typename B, typename F>
2  auto zipWith (F f) {
3    return [=](const std::forward_list<A>& L) {
4      return [=](const std::forward_list<B>& R) -> std::
           forward_list<decltype(f(A(),B()))> {
5        std::forward_list<decltype(f(A(),B()))> H;
6        std::transform(L.begin(), L.end(), R.begin(), std::
             front_inserter(H), f);
7        H.reverse();
8        return H;
9      };
10   };
11 };
12  [....]
13   auto op = [] (int x, char z) {
14       return std::make_tuple(x,z);
15   };
16   auto res = zipWith<int,char>(op)(L)(R);
17   map([] (std::tuple<int,char> v) { std::cout << v << ",";
         return v;}, res);
18   //prints : (1,a),(67,b),(89,l),(23,u),(45,t),(1,v),(3,r)
         ,(99,6),(-90,h),
```

Listing 8: curried version of zipWith

Listing8 is closer to zipWith's curried version shown in the function signature above. The listing shows the same zip operation as the previous one. However,

the call to zipWith requires a complete specification of the template types. This increases the line noise somewhat. C++'s type system is not powerful enough to infer the types from type of the arguments to *op*.

Reduce and the List Monad

The type signature for reduce is:

$$reduce \ :: \ (a \to b \to a) \to a \to [b] - > \to a$$

reduce moves or folds a binary operation over a list and returns a result. [1] The type of the first argument to the binary operation is the same as the type returned by reduce. It's also the type of the first input variable encountered after operator specification. The first input variable is used initailize the first argument to the binary operation, when the first element of the list $[b]$ is being processed.

In fact map can be implemented in terms of reduce. In that case the type a would be the list type, and the initial value would be the empty list. The binary operator would then concatenate the result of a unary operation onto the list. Because map can be implemented using reduce, reduce is more powerful than map.

The stl function which closely matches the type signature for reduce is the std::accumulate function found in the $< numeric >$ header [11, 22].

[1] In fact another name for reduce is foldl. there also exists a closely related dual foldr. I refer to [7, 6] for more on their relationship.

The version we use second takes a binary operator and a couple of list iterators as input.

template $<$ typename $input\_container\_iterator$,
typename T
typename $binary\_operation >$
T **accumulate**
($input\_container\_iterator\ begin$,
$input\_container\_iterator\ end$,
$T\ initial\_value$,
$binary\_operation\ binary_{op}$);

The function signature of std::accumulate tracks that of reduce fairly closely. The main difference is the order of the arguments and the lack of curry.

```
1 std::forward_list<int> L = {1,-6,23,78,45,13};
2   auto max = [] (int x, int y) { return (x > y) ? x : y;};
3   auto res = std::accumulate(L.begin(), L.end(), std::
        numeric_limits<int>::min(), max);
4   std::cout << "maximum : " << res << std::endl; //prints 78
```

Listing 9: example of std::accumulate

Listing 9 shows how e can use std::accumulate to the maximum value in a list. $std :: numeric_limits < int >:: min$ returns the smallest possibele integer value and is used to initialize the search. The binary operation is just a lambda wrapped around the compare operator and std::accumulate returns the expected result.

```
1  auto show   = [] (int v) { std::cout << v << ","; return v;};
```

```
2  typedef std::list<int> list_t;
3  list_t L = {1,-6,23,78,45,13};
4  auto m   = [] (list_t L, int y) { L.push_back( 2*y + 1);
       return L;};
5  auto res = std::accumulate(L.begin(), L.end(), list_t(), m);
6  map(show,res); //prints 3,-11,47,157,91,27,
```

Listing 10: processing a list using reduce

In listing 10 std::accumulate is used to process a list by applying an function to each element. Notice that the body of the lambda m does in fact two things : The actual operation we would want to perform (2*y+1 in this case) as well as the concatenation of the result of this operation to the target list.

```
1  [...]
2  typedef std::forward_list<int> list_t;
3  list_t L          = {1,-6,23,78,45,13};
4  auto op       = [] (int y) {return list_t({2*y+1});};
5  auto concat = [] (list_t A, list_t B) { A.splice_after(A.
       before_begin(), B); return A;};
6  auto bind     = std::bind(concat, std::placeholders::_1, std::
       bind(op, std::placeholders::_2));
7  auto show   = [] (int v) { std::cout << v << ","; return v;};
8  auto res        = std::accumulate(L.begin(), L.end(), list_t(),
        bind);
9  map(show, res); //prints 27,91,157,47,-11,3,(i.e reverse order
       )
```

Listing 11: unary operation and reduce

Listing 11 refactors the code in listing 10 by seperating the unary operation and the list concatenation. The generalized function signature of the unary operator (the lambda bound to op) is $a \rightarrow [b]$. It first blush this looks like a clunky reimplemention of the map function but in fact it's more powerful.

```
1   template<typename A, typename F>
2 auto mapM (F f, std::forward_list<A> L) -> decltype(f(A()))
3 {
4   typedef typename decltype(f(A()))::value_type ret_t;
5   L.reverse();
6   auto concat = [] (std::forward_list<ret_t> L, std::
        forward_list<ret_t> R) {
7     L.splice_after(L.before_begin(), R);
8     return L;
9   };
10  auto op     = std::bind(concat, std::placeholders::_1, std::
        bind(f, std::placeholders::_2));
11  return std::accumulate(L.begin(), L.end(), std::forward_list<
        ret_t>(), op);;
12 }
```

Listing 12: the list monad

Listing **??** shows the implemention of a function called mapM based on the refactoring done in listing 11. It's signature resembles that of map. Just like mapM takes a unary function f, and a list and returns a list:

$mapM :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$

However note that f returns a list of elements, rather than a single value. This makes mapM a lot more powerful.

```
1  [...]
2    auto show   = [] (std::tuple<int,char> v) { std::cout << v <<
         ","; return v;};
3    static char digits[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'
         , 'i', 'j' };
4    typedef std::forward_list<std::tuple<int, char>> list_t;
5    auto op  = [=] (int y) {return list_t({std::make_tuple(y,
         digits[abs(y)%10])});};
6    map(show, mapM(op, std::forward_list<int>({1,-6,23,78,45,13}))
         );
7  //prints :(1,b),(-6,g),(23,d),(78,i),(45,f),(13,d),
8  auto res = map(op, std::forward_list<int>({1,-6,23,78,45,13}));
9    std::cout << std::endl << "----------" << std::endl;
10   for (auto& el : res) {
11     std::cout << "[";
12     map(show, el);
13     std::cout << "], ";
14   }
15 //prints :[(1,b),], [(-6,g),], [(23,d),], [(78,i),], [(45,f),],
       [(13,d),]
```

Listing 13: comapring mapM and map

Listing 13 uses mapM to redo the previous example shown in listing 7.

The main difference is that the function op which is mapped over the list returns a list rather than a single element, like it did in listing 7. In fact we could extend

this example by having op return more than one element, or no elements at all. Regardless, mapM would return a list of results.

If you try to do the same thing with map a list of lists is returned.

In a tyical scenario you'd want to apply a number of operations to a list. mapM allows each function to have the same signature : It takes a single element and returns a list of elements. The next application of on a list returned by map, would (as you can see when the results are printed in the example) require an iteration over the result list. In fact the resulting list is fundamentally different from the input list. It's the ability of mapM to join the result lists of the operation into a single list that provides a great deal of power.

In fact the type signature of mapM is that of the monad implemention for lists **??**. The use of monads and other types provide a powerful extension of the functional approach **??**. I hope to discuss the support for those in a follow up article.

Conclusions

Is functional programming possible for the mainstream programmer in C++ ? In this article I've discussed basic functional techniques, like lambda expressions and closures, partial function application, currying, map and reduce. In addition I've introduced a more powerful, monadic form of the map function. I've shown that the new additions, notably λ's and closures to the standard have made the use of these functional techniques a posibility. Sometimes using functional features

introduces a lot of 'line noise' in the form of accolades, returns or semi-colons. But C++ has never been quiet in that respect, and the standard has added features - like the $auto$ declaration, range based for loops - which reduce this noise somewhat. The use of currying in particular may introduce some added noise in that regard. Error messages generated by te compiler are an other concern. I have not shown the reader the reams of messages produced when something goes wrong. Again, this is not something entirely new to C++ but it can be a daunting task to work through.

Functional programming emphasizes referrential transparancy through the use of immutable data. Changes are made to a copy of that data item. In the implementations of map and reduce shown here new lists are created containing the changed data elements. To remain referentially transparant this requires that the copy semantics of the objects is relatively straight forward. That in turn requires the use of straightforward data types, which behave like 'values', and don't maintain state. The creation of a completely new list of data items introduces an obvious performance penalty. In languages designed for functional programming the cost of this approach is reduces because items ae in fact reused [8]. In C++ the tradeoff of referential transparancy versus performance is a real one.

The extension of the functional approach to a richer class of problems, like IO has introduced a whole new set of concepts [7, 5, 24]. To what extend those concepts are supported in C++ will be the subject of an other paper.

References

[1] Bjarne Stroustrup

The C++ Programming Language

Addison-Wesley, 1997, 3rd edition.

[2] Brian McNamara, Yannis Smaragdakis

Functional programming with the FC++ library.

J. Funct. Program. 14(4): 429-472 (2004)

[3] David Vandevoorde, Nicolai M. Josuttis

C++ Templates

Addison-Wesley, 2003.

[4] Andrei Alexandrescu

Modern C++ Design

Addison-Wesley, 2001

[5] Miran Lipovača

Learn you a Haskell for great good : a beginner's guide

no starch press, San Fransisco, 2011

[6] Graham Hutton

Programming in Haskell

Cambridge University Press, 2007

[7] Richard Bird *Introduction to Functional Programming using Haskell* Prentice Hall Europe, 1998, 2nd edition

[8] Anthony J. Field and Peter G. Harrison
Functional Programming
Addison-Wesley, 1989.

[9] Michael L. Scott
Programming Language Pragmatics
Morgan Kauffmann, 2006, 2nd edition

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns : Elements of Resusable Object-Oriented Software
Addison Wesley Longman, 1995

[11] Nocolai M. Josuttis
The C++ Standard Library
Addison-Wesley, 2nd edition.

[12] *https://github.com/fons/functional-cpp*

[13] *http://www.macports.org/*

[14] *http://en.cppreference.com/w/cpp/language/lambda*

[15] *http://en.cppreference.com/w/cpp/utility/functional/function*

[16] *http://en.cppreference.com/w/cpp/language/auto*

[17] *http://en.cppreference.com/w/cpp/utility/functional/bind*

[18] *http://en.cppreference.com/w/cpp/utility/functional/placeholders*

[19] *http://en.cppreference.com/w/cpp/algorithm/for_each*

[20] *http://en.cppreference.com/w/cpp/algorithm/transform*

[21] *http://en.cppreference.com/w/cpp/language/decltype*

[22] *http://en.cppreference.com/w/cpp/algorithm/accumulate*

[23] zip function in Python

http://docs.python.org/2/library/functions.html#zip

zip function in Ruby

http://ruby-doc.org/core-2.0/Array.html#method-i-zip

Support in Perl

http://search.cpan.org/ lbrocard/Language-Functional-0.05/Functional.pm

[24] Brent Yorgey

The Typeclassopedia The Monad.Reader, Issue 13; p17; 12 March 2009

www.haskell.org/wikiupload/8/85/TMR-Issue13.pdf