Living with λ 's

Functional Programming in C++

alfons haffmans

April 16, 2013

Introduction

Functional programming and C++. The combination will may strike ane equal mixture of disgust and terror in some of you. Others may be intrigued and perhaps even daunted by the prospect. C++ has always been a multi-paradigm language [**?**]. Compile time template meta-programming has always provided a pure lazy functional programming experience [**?**, **?**]. But Previous attempts to implement non-trivial functional programming features to the run-tim required significant 3rd library support [**?**]. However features , like support for lambda fucntions, added to the language by the recent C++ standard upgrade have made that a much easier [**?**]. This paper explores the support out-of-box for functional programming by exploring its application to problems typically found in introductory fuunctional programming textbooks [**?**, **?**, **?**]. This is not to say

that these techniques cannot be applied to more difficult problems. But I imagine a reader, although familiar with C++, is not nescesarily familiar with some of of the basic functional programming approaches.

A quick note on the source code. The code is available on github [**?**] and was compiled using gcc 4.8 installed on Mac OSX using MacPorts [**?**]. The code itself is relatively straightforward and I make no claim regard its production level quality ! You'll be able to find all the examples in the text somewhere in the repository.

Object Oriented and Functional Programming Style

At the heart of object-oriented programming (oop) is te encapsulation of data, state and methods in a coherant whole called a class or object. Classes can be combined through inheritance or composition to form more complex entities. Objects communicate by sending each other messages which in the case of C++ corresponds to making methods calls to change the internal state of the object. Each 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. State changes are implemented using an imperative programming style. An imperative programing style emphasizes the use of statements and mutable data A for-loop is typical imperative construct. A for loop iterates through a list of objects stored in a container. Thus is done using an iterator. The iterator points to an element

in the container. This element is then changed by calling it's methods. When the for-loop is done the elemenst in the lsit have been modified. This list can be passed on for further processing to another part of the program. After all this processing, any reference to the original list will in fact be pointing to a materially different list. This lack of referential integrity is a well know feature of this style of programming. C++ provide the keyword *const* to indicate to the caller which methods won't modify the internal state of the object [**?**].

At the heart of fucntional programming lies the construction of computations which then act on immutabe values. Data and operations on data are not comingled.

Data is immutable and acts like a value. A value, like 1 , can't be modified. You can bind a reference to 1, but 1 itself is immutable. You can add 1 to the reference but the reference will not be modified. This provides referential transparancy.

In addition functions are first-class objects. You can have a reference to a function and pass it as an argument to an other function just like you would the reference to other data. Functions are also able to return references to functions. Functions that take functions as arguments or which return functions are called higher order fucntions. They play an important role in functional programming, because they allow you to construct functions from other functions.

Iteration through a list is implemented using recursion. Typically a main function takes a list and a unary operator as an argument. It also has a third argument which is used to collect the new values. The body of this fucntion would unary

function on the head of the list. The return value would be stored in a different list, called an acumulator [**?**]. The tail of the list and the accumulator would both be arguments the next call of the main function.

There is an obvious trade off between compactness and speed versus referential transparency. In functional programming languages this tradeoff is small because the new list will reference the old, unchanged elements in its old version. In C++ that is not going to be the case. Here the pursuit of referential will come with the cost of creating copies.

I use Haskell's notation to formally represent function signatures and type constructors [**?**]. Here's how a function *f* is represented

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

em f takes two arguments of type *a* and *b* and returns a value of type *c*.

For function implementations the notation the return type follows the double colon **::** after the argument list. Here's the type signature of the identity function :

$$id :: a \quad \rightarrow \quad a$$

Here are two of it's implentations :

$$id(int\ x) :: int \quad = \quad x$$

$$id(Person\ p) :: Person \quad = \quad p$$

Here *Person* is a data type representing a person.

Types that are paramatrized by other types, like the arrow operator $\rightarrow$ are refered to as type constructors [**?**]. The arrow operator takes two types: the argument type a and the return type b. Formally, $M\ a$ represents a type constructor M which takes a single type variable a. $M\ a$ is a much terser representation of a c++ template : $\mathbf{template} <$ typename **a** $>$ struct **M**.....

The arrow operator corresponds to the function wrapper $std :: function < a(b) >$ [**?**]. The type constructor [a] creates a list of elements of type a. $[a]$ corresponds to the stl containers $std :: list < a >$ or $std :: forward\_list < a >$ [**?**].

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 [**?**]. This capture of the enclosing environment by the lambda expression is called a closure [**?**, **?**].

The (slightly abbreviated) C++ syntax for the lambda expression is [**?**]:

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

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 $(\ params\ )$ are the parameters, and the optional $\rightarrow\ rettype$ sprecifies the return type. An auto [**?**] declared variable or or a variable declared using $std :: function$ [**?**] can be used to bind the lambda expression to a variable.

```
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

In listing **??** the the lambda expression is bound to the variable called func. The lambda has no arguments and the variables x and y referenced in body of the expression are therefore free. x and y are defined earlier on and are set to 0 and 42 respectively. The specifier $[x, \&y]$ captures x by value and y be reference. So x remains the same, regardless of what value it takes on later in the program. Since y is captured by reference, changes made to it later on are reflected any subsequent execution of func.

The lambda inc captures y by reference and increments it by one. 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. The changes to x and y in the body of $func$ after a call to the 'inc's' is shown in the comment following the statement. As you can see, the value of y in the body of the lambda expression bound to func reflects the changes made to it after it's definition. 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 **??** 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. The generalized function wrapper $std :: function < int(int) >$ is used to define $factorial$ to which the lambda is bound.

Partial Function Application

Supplying a function with less than it's full compliment of arguments creates partially applied function. In C++ partial function application can be achieved through the use of *std::bind* [**?**] and std::placeholders::.. [**?**]. Both are defined in the header $< functional >$. The placeholders are in the namespace $std :: placeholders$ and are named $_1$, $_2$ etc.

std::bind is a template function which takes a callable like a function 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 values will be used as arguments to the function. The place holders correspond to the arguments of the callable returned by std::bind. Each distinct place holder will correspond to an argument and can be bound to one or more arguments of the function.

```
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

In listing **??** lambda $repeat$ is a higher order function which repeated 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. 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. Finally the lambda l1 is initially called with the value 9 and l1 is called nine times with its output value as the input on the next iteration. The result is shown in the comment.

Currying

There is a more powerful way to construct functions out of other functions than partial function application. *Currying* (named after the mathematician Haskell B. Curry) is a technique to turn a function with arity n into a function of one variable [**?**]. The curried version of a function is a higher order function which returns a partially applied version of the orginal function.
Below is the definition of the curry2 function designed to curry binary functions $f(x, y)$.

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

curry2 takes a fucntion wich takes a pair of arguments and returns a function of one argument. When this function is called with an argument of type a, it returns an other unary function which takes an argument of type b. This function is a partially applied version of the uncurried fucntion f, where the argument a is provided. When you call this function it returns the final value of type c.

$$
\begin{aligned}
plus\ ::(int\ x, int\ y)\ ::int = x + y \ \Rightarrow\ & cplus(int\ x)\ ::\ (int \to int) \\
& \to (int\ y)\ ::\ int \to x + y \\
plus(5,6) = 11 \ \Leftrightarrow\ & (curry2\ plus)(5)(6) = 11
\end{aligned}
$$

(curry2 plus) is the curried version of plus. The return types have been made explicit. *curry2 plus)(5)* returns a partially applied plus function, which is then called with 6 as the argument with an unsuprising 11 as the result. A simple implementation of curry2 binary functions is shown in listing **??**.

```
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 support for currying. In fact, C++ functions are not written in curried form. Compare this to functions in Haskell which are curried by default [**?**, **?**]. Support for currying has become a lot easier now that lambda's are supported, but the programmer needs to either use a third-party library or roll her own implementation. There is an additional problem with the use of curried functions in C++. Consider the curried version of zipWith **??** shown below. Using the curried version requires a full specification of all the template types. At the very least this increases the line noise when the curried version is used.

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.

$$
\begin{array}{rl}
\textbf{template} < & \textbf{typename } input\_container\_iterator, \\
 & \textbf{typename } output\_container\_iterator, \\
 & \textbf{typename } unary\_operation > \\
output\_container\_iterator & \textbf{transform} \\
 & (input\_container\_iterator\ begin_1, \\
 & input\_container\_iterator\ end_1, \\
 & output\_container\_iterator\ destination_1, \\
 & unary\_operation\ unary_{op});
\end{array}
$$

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 **??** 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 **??** 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 [**?**] 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 **??** 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

Listing**??** is closer to zipWith's curried version shown in the function signature above. The listing shows the same zip operation as the previous one. How-

ever, the call to zipWith requires a complete specification of the template types. 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 [**?**, **?**].

[1] In fact another name for reduce is foldl. there also exists a closely related dual foldr. I refer to [**?**, **?**] 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 **??** 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 **??** 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 **??** refactors the code in listing **??** 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 **??**. It's signature resembles that of map. Just like

mapM takes a unary function f, and a list and returns a list:

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

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 **??** uses mapM to redo the previous example shown in listing **??**.

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 **??**. 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 [**?**]. 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 [**?**, **?**, **?**]. 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*