Living with λ 's

Functional Programming in C++

alfons haffmans

April 19, 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 have improved the support for functional programming [11]. In fact, previous attempts to add functional programming features required a significant effort [2]. This paper explores the support out-of-box for functional programming provided by the new standard We'll look at techniques typically found in introductory functional programming textbooks [5, 6, 7]. This article assumes familiarity with C++, but not necessarily 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 class. Each instance of a class represents an entity in the real world. A class is responsible for the management 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 the implemention of algorithms in C++. The for-loop processes object instances in a container. An iterator points to the element being processed in the body of the loop. The body typically has statements which affect the state of the element referenced by the iterator. When the for-loop reaches the end of the container at least some of its elements 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 style of programming which emphasizes the use of mutable data and statements is called an imperative programming style. Imperative programming makes it hard to prove by simple statement inspection if a 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 immutable values. Data and operations on the data are not commingled. Immutable data acts like a value. You can hold a reference to a value like 1, but 1 itself is immutable. You can add 2 to the reference but the reference itself still points to 1. The use of functions acting on immutable data lies at the heart of functional programming.

In a functional language functions are first-class objects You create a reference to a function like any other data type. Functions can have functions as arguments or return functions. Functions like that are called higher order functions. Higher-order functions are used to combine simpler functions into more complex ones. They play an important role in functional programming.

Recursion replaces for-loops as a control structure for list processing [6, 7]. This creates a new list containing the results of applying a function to the elements of the old list.

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]. Using that notation the function signature of a binary function *f* with arguments of type *a* and *b* and a return value of type *c* is:

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

The representation of function implementations is slightly different. The $=$ operator separates the argument list from the function definition. The double colon **::** following the argument list defines the return type. Here's the type signature and implementation of the identity function id:

$$
\begin{aligned}
id :: a &\rightarrow a \\
id(int\ x) :: int &= x
\end{aligned}
$$

In $a \rightarrow b$ the arrow $\rightarrow$ is a generic type which takes two other types a and b to be fully defined. Types that are paramatrized by other types are referred to as type constructors [5].

In general $M\ a$ represents a type constructor M which takes a single type variable a. $M\ a$ corresponds to the C++ class template $\mathbf{template} <$ typename **a** $>$ struct **M**$\{....\}$. The equivalent of the arrow operator in C++ is the function wrapper $std :: function < a(b) >$ in $< functional >$ [11, 15]. A list of values of type a is created by the type constructor $[a]$. The equivalent of $[a]$ in C++ are linear containers like $std :: list < a >$ or $std :: forward\_list < a >$ [11].

Lambda Expressions and Closures

Lambda expressions allow you to create functions on-the-fly. The body of the lambda can reference free variables which are not defined in its argument list. 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 creates a closure [8, 9].

The (slightly abbreviated) C++ syntax for the lambda expression is [14]:

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

The capture specifier [...] determines how the free variables are captured. If it's empty [], the body of the lambda can't reference any free variables. The $[=]$ specifier captures free variables by value, whereas the $[\&]$ captures them by reference. When the specifiers $=$ and $\&$ are followed by a variable only that variable will be captured. The return type specifier $\rightarrow\ rettype$ is optional. The keyword $mutable$ allows the lambda to modify the free variables captured by reference. Lambda's can be bound to variables using $std :: function$ [15] or auto [16]. The new keyword $auto$ allows you the elide a full type specification, reducing some of the line noise in the code.

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

  func(); //prints: Hello world from lambda : 0,42
  y = 900;
```

```
func(); //prints:  Hello world from lambda : 0,900

inc();
func(); //prints : Hello world from lambda : 0,901

inc_alt();
func(); //prints: Hello world from lambda : 0,901

inc_alt_alt();
func();//prints: Hello world from lambda : 0,902

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. The lambda $func$ has no arguments and prints the value of the two free variables x and y to stdout. The free variables x and y are defined earlier in the code and are initialized to 0 and 42 respectively. The capture specifier of $func$ is $[x, \&y]$. So x is captured by value and y be reference. The 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 lambda can change y because the keyword $mutable$ is in the definition inc_alt_alt captures the complete environment by reference, and increments both x and y.

Each time y is changed $func$ is called to show the current state of its free variables x and y. Their values are shown in the comment following $func()$.. Since y is captured by reference is can be changed through side effects. On the other hand x is captured once and remains the same.

```
[...]
  // as opposed to [=] or [] or [I]
  std::function<int (int)> factorial = [&factorial] (int x) -> int {
    std::cout << x << ",";
    if (x == 0) return 1;
    return x * factorial(x-1);

  };
  auto res = factorial(10);
  std::cout << std::endl;
  std::cout << "res : " << res << std::endl;
  //prints : 10,9,8,7,6,5,4,3,2,1,0,
 //       res : 3628800
```

Listing 2: Implementation of factorial using lambda recursion

Listing 2 shows a recursive implemention of the factorial $n!$. 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 value of the argument is printed each time $factorial$ is called. The output for $factorial(10)$ is shown in the comments.

Partial Function Application

A partially applied function is created when a function is called with fewer arguments than specified in its argument list. In that case a lambda is returned with the remainder of the arguments [8]. C++ supports partial function application through *std::bind* [17] and placeholders [18]. Both are defined in the header $< functional >$. 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 and returns an other function object. Subsequent arguments to std::bind are either values or placeholders.

The position of the values and placeholders corresponds to the position of the argument in the argument list of the input function. A value is bound to the argument of the input function and a placeholder creates an argument for the callable returned by std::bind. The number of arguments is equal to the number of distinct placeholders.

```
[...]

  auto repeat = [](int n, double y, std::function<double(double)
     > f) {
    while (n-- > 0) {
      y = f(y);
    }
    return y;
  };
```

```

auto rpl    = std::bind (repeat,
        std::placeholders::_1,
        std::placeholders::_1,
        std::placeholders::_2);

std::function<double (double)> l1    = [](double x) { return 2*x-0.906;};
auto val = rpl(9, l1);
std::cout << " result : " << val << std::endl; // print 4145.03
```

Listing 3: std::bind example

Listing 3 creates a function rpl which repeatedly calls function $l1$. rpl is created using std::bind and std::placeholders on the higher order function $repeat$. The function passed in as the third of $repeat$ is in the body of a while loop. This function is initially called with the value of the 2nd argument. The first argument is the number of repetitions. Through std::bind rpl sets the number of repetitions equal to the initial value because the first and second argument of $repeat$ are bound to the same placeholder $std : placeholders :: \_1$. The result of $rpl(l1, 9)$ is printed to stdout, and it's value is shown in the comment.

Currying

Currying is a technique to any function into a higher order function of one variable

[8]. The curried version of a function returns a partially applied version of the original function.

The function signature of $curry2$ designed to curry binary functions is:

$$
\begin{aligned}
curry2 \ :: \ ((a,b) \to c) \quad &\to \quad (a \to b \to c) \\
f :: (a,b) \to c \quad &\Rightarrow \quad (curry2\ f) :: a \to b \to c
\end{aligned}
$$

curry2 takes a binary function and returns a unary function which is the curried version of f. The curried version of f returns another unary function when it is called with an argument of type a. Calling this function is called with an argument of type b gives the same value as f.

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

$(curry2plus)$ is the curried version of $plus$. *curry2 plus)(5)* returns a lambda which represents the $plus$ function partially applied to 5. When this partially applied version of $plus$ is called with 6 an unsurprising 11 is the result. Listing 4 shows an implementation of curry2 in C++:

```
template <typename R, typename T, typename U>
std::function<std::function<R (U)> (T)> curry(std::function<R (T
    ,U)> op)
```

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

Listing 4: curry for binary operators

Currying plays an important role in functional programming [8]. It simplifies the design of higher order functions since we only have to consider unary functions. C++ does not provide a curry operator and functions are not written in curried form. Compare this to Haskell where functions are curried by default [5, 6]. However writing a curry operator or writing curried versions of a function has become a lot easier now that lambda's are supported.

Map, zipWith and Zip

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

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

In the stl the equivalent of the map operation would have a unary callable object and the begin and end iterators as input.

In fact $std :: for\_each$ in the header $< algorithm >$ has this interface but is an imperative implementation of a for-loop [19, 11]. Each element of the input container is changed.

A more functionally correct choice is the $std :: transform$ function which is also part of $< algorithm >$. This function has two forms [**?**, 21]. The first one takes a unary callable object as input and applies it to the elements of the input container. It puts the results in the destination range. The return value is an iterator past the last element of the destination range.

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

Listing 5 shows a possible implementation of the map function for a std::forward_list [20, 11].

```
template<typename A, typename F>
auto map (F f, const std::forward_list<A>& L) -> std::
    forward_list<decltype(f(A()))>
{
  std::forward_list<decltype(f(A()))> H;
  std::transform(L.begin(), L.end(), std::front_inserter(H), f);
  H.reverse();
```

```
    return H;
}
```

Listing 5: map for std::forward_list

The template for map takes two type parameters A and F. Parameter A specifies the type of the element in the input container L. Paramater F specifies a very generic callable. [1] The return type of map is declared as $auto$ because it depends on the types of the input arguments. The declaration is deferred until after their declaration and is determined by applying $decltype$ to the expression $f(A())$ [11, 16, 22].

The second flavor of $std :: transform$ applies a binary function to two input ranges and copies the result to the destination range. It returns an iterator past

[1] std::function could have been used to provide a more typesafe interface. However that would not allow us to use in-line lambda functions. The type of each lambda is unique and therefore would not be converted to std::function.

last element in that third range.

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

This function signature is similar to that of the $zipWith$ function found in Haskell [5, 6]:

$$
\begin{aligned}
zipWith \; &:: \; (a \to b \to c) \to [a] \to [b] \to [c] \\
zip \; &:: \; [a] \to [b] \to [(a, b)]
\end{aligned}
$$

$zipWith$ applies a curried binary function to two list with elements of type a and b. 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 [24].

```
template<typename A, typename B, typename F>
auto zipWith (F f, const std::forward_list<A>& L, const std::
    forward_list<B>& R) -> std::forward_list<decltype(f(A(),B()))
    >
```

```
{
  std::forward_list<decltype(f(A(),B()))> H;
  std::transform(L.begin(), L.end(), R.begin(), std::front_inserter(H), f);
  H.reverse();
  return H;
}
template<typename A, typename B>
std::forward_list<std::tuple<A,B>> zip (const std::forward_list<A>& L, const std::forward_list<B>& M)
{
  return zipWith([] (const A& a, const B& b) {return std::make_tuple(a,b);}, L, M);
}
```

Listing 6: zipWith and zip implemented with std::transform

Listing 6 shows the implemention of $zipWith$ and zip for a $std :: forward\_list$. Similar to the map fuunction the type of the return list is determined by calling decltype on the expression $F(A(), B())$.

```
   [...]
  std::forward_list<int>  L = {1,67,89,23,45,1,3,99,-90};
  std::forward_list<char> R = {'a','b','l','u','t','v','r','6','h'};

  auto H2 = zip (L, R);
  map([] (std::tuple<int,char> v) { std::cout << v << ","; return v;},H2);
```

```
//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.

```
template<typename A, typename B, typename F>
auto zipWith (F f) {
  return [=](const std::forward_list<A>& L) {
    return [=](const std::forward_list<B>& R) -> std::
        forward_list<decltype(f(A(),B()))> {
      std::forward_list<decltype(f(A(),B()))> H;
      std::transform(L.begin(), L.end(), R.begin(), std::
          front_inserter(H), f);
      H.reverse();
      return H;
    };
  };
};
 [....]
  auto op = [] (int x, char z) {
      return std::make_tuple(x,z);
  };
  auto res = zipWith<int,char>(op)(L)(R);
  map([] (std::tuple<int,char> v) { std::cout << v << ","; 
      return v;}, res);
  //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 8 is closer to zipWith's curried version shown in the function signature above. The listing shows the same zip operation as the one in listing 7. The line noise in this version has increased because the curried version requires explicit template instantiation. 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: [2]

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

$reduce$ moves or folds a binary operation over a list and returns a result.

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 variable encountered after operator specification. Therefore this variable is used as the first argument to the binary operation, together with first element of the list $[b]$.

map can be implemented in terms of $reduce$, which makes $reduce$ more powerful. 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

[2] 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.

operation to the list.

The $std :: accumulate$ function found in the $< numeric >$ header takes a binary operator and a couple of list iterators as input and returns the result of mapping the binary operator over the range [11, 23]:

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 main difference between function signature of $std :: accumulate$ and $reduce$ is the order of the arguments and the lack of curry.

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

Listing 9: example of std::accumulate

In listing 9 $std :: accumulate$ is used to find the maximum value in a list. The search is initialized by $std :: numeric\_limits < int >:: min$ which returns the smallest possible integer value [11]. The binary operation is a lambda wrapped around the compare operator. The result returned by std::accumulate is printed to stdout and the unsurprising result is shown in the comment.

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

Listing 10: processing a list using reduce

In listing 10 $std :: accumulate$ processes a list by applying an function to each element of the input list and returning the result in a new list. The body of the lambda m applies the unary operation $op(y) = (2 * y + 1)$ and concatenates the result to the destination list.

```
 [...]
 typedef std::forward_list<int> list_t;
 list_t L          = {1,-6,23,78,45,13};
 auto op       = [] (int y) {return list_t({2*y+1});};
 auto concat = [] (list_t A, list_t B) { A.splice_after(A.
    before_begin(), B); return A;};
 auto bind     = std::bind(concat, std::placeholders::_1, std::
    bind(op, std::placeholders::_2));
 auto show   = [] (int v) { std::cout << v << ","; return v;};
 auto res        = std::accumulate(L.begin(), L.end(), list_t(),
     bind);
```

```
  map(show, res); //prints 27,91,157,47,-11,3,(i.e reverse order)
```

Listing 11: unary operation and reduce

Listing 11 re-factors the code in listing 10 by separating the unary operation and the list concatenation.

```
  template<typename A, typename F>
auto mapM (F f, std::forward_list<A> L) -> decltype(f(A()))
{
  typedef typename decltype(f(A()))::value_type ret_t;
  L.reverse();
  auto concat = [] (std::forward_list<ret_t> L, std::forward_list<ret_t> R) {
    L.splice_after(L.before_begin(), R);
    return L;
  };
  auto op      = std::bind(concat, std::placeholders::_1, std::bind(f, std::placeholders::_2));
  return std::accumulate(L.begin(), L.end(), std::forward_list<ret_t>(), op);;
}
```

Listing 12: the list monad

Listing **??** shows the implemention of a function called mapM based on the refactoring done in listing 11. At first blush this looks like a clunky reimplementation of the map function but in fact it's more powerful. Just like map, mapM takes a unary function f, and a list and returns a list:

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

The main difference the function signature of the unary operator : $a \rightarrow [b]$. It returns a list of values rather than a value. This makes mapM more powerful than map.

```
[...]
  auto show   = [] (std::tuple<int,char> v) { std::cout << v << ",";  return v;};
  static char digits[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };
  typedef std::forward_list<std::tuple<int, char>> list_t;
  auto op  = [=] (int y) {return list_t({std::make_tuple(y, digits[abs(y)%10])});};
  map(show, mapM(op, std::forward_list<int>({1,-6,23,78,45,13})));
  //prints :(1,b),(-6,g),(23,d),(78,i),(45,f),(13,d),
```

Listing 13: example of mapM

In listing 13 $mapM$ maps unary operator op over the list $[1, -6, 78, 45, 13]$ and returns a list where each number is combined with a character in the $digits$ list. The binary operation op in listing 13 could have returned more than one element, or an empty list. Regardless of what it returns mapM returns a *list* of values.

If you map an operation $a \rightarrow [b]$ over a list $[a]$ with map a list of lists $[[b]]$ is returned. A list of a list of values is a fundamentally different data type than a list of values. In C++ its type would be $std :: forward\_list < std :: forward\_list < T >>>$ where template parameter T represents the type of the values in the inner list. The problem here is that the structure of the container has changed from a simple list $std :: forward\_list < T >$ to a nested list. $mapM$ concatenates the results of the binary operator op and the structure of the container remains the same.

In a typical 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. By contrast the next application of a function on a list returned by map, would require an iteration over the result list.

The type signature of mapM is that of the monad implemention for lists [7, 5]. The use of monads and other types provide a powerful extension of the functional approach [25]. I hope to discuss the support for those in a follow up article.

Conclusions

In this article looke at basic functional techniques, like lambda expressions and closures, partial function application, currying, map and reduce. It briefly touched on a more powerful, monadic form of the map function. The ability to use these functional techniques has become possible because of recent additions to the C++ standard. However an important technique like currying is well supported. Currying can be implemented by the programmer, but its use introduce some added line noise. The use of functional features does seem to introduce a lot of accolades, returns and sem-colons. In addition the error messages generated by the compiler can be quire daunting.

Functional programming emphasizes referential transparency through the use of immutable state. In the implementations of map and reduce shown here new lists are created containing the changed data elements. To be referentially transparent requires straight forward and simple copy semantics. That in turn implies the use of simple data types, like $std :: tuples$, which don't maintain state and behave like values [26].

The creation of a new list map and $reduce$ with copies of the data items by incurs a performance penalty. In languages designed for functional programming the cost of this approach is reduced because items that remain unchanged are in fact reused [8]. In C++ the tradeoff of referential transparency needs to be traded off against performance.

The extension of the functional approach to a richer class of data structures introduces a whole new set of concepts [7, 5, 25]. To what extend those concepts are supported by 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 Reusable 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/forward_list*

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

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

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

[24] 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

[25] Brent Yorgey

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

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

[26] *http://en.cppreference.com/w/cpp/utility/tuple*