Living with λ 's
Functional Programming in C++

alfons haffmans

April 13, 2013

Introduction

In C++ is a multi-paradigm language geared towards developing hight performance code. It has always been a multi-paradigm language with an emphasis objects.
what is functional programming ?
Typically we see that functional programming emphasizes te use of functions over

Programming in a functional style

I'd like to make the distinction between a 'pure' functional prgramming languge and programming in a functional style. A pure functional language like Haskell is designed to support functional programming from the ground up. On the other hand a language like C++ or Lisp supports multiple paradigms, one of which if functional.

object oriented programming style

Object oriented programming has been a phenomenally successful programming paradigm.
Object oriented programming models the world with classes and objects. A class encapsulates a state and provides an interface to manipulate that state. Complex behaviours are achieved by combining classes through inheritance. Semantically similar behaviours are grouped through the use of interfaces. State changes are

implemented using an imperative programming style. An imperative program consists of statements, uses for loops and assignments to change data and uses if-then-else to control the flow. Recursion is avoided as a control statement.

functional programming style

functions are first class objects. functions can be used as arguments, or returned by other fucntions. We can construct references to functions and use those as we would references to any other value type. functional programming seeks to model the world as a set of computations which work on immutable data. Recursion is an important way to control the flow of execution. Expressions.. Because functions are accorded the same status as data they cnan be used as arguments or return values. Functions are which take functions as arguments or return functions are called higher order functions. Higher order functions play an important role in construction computions by combining relativey simple function.
referential integrity.

1. everything is a computation
2. data is immutable
3. referential transparancy
4. no side effects
5. decouple data from operations on the data
6. construct a computation
7. few data structures ; lots of operations
8. declarative style : combine functions into expressions

function signatures and type constructors

I use Haskell's notation to formally represent function signatures and type constructors. As an example, here's how a function *f* which takes two arguments of type *a* and *b* and returns a value of type *c* is represented

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

For function implementations the notation is changed slightly : the return type follows a double colon **::** after the argument list. Heres the type signature and two implentations of the identity function em id :

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

the id finction is polymorphic. Types that are paramatrized by other types are refered to as type constructors [scala for impatient] or generic types. I$M\ a$ represents a type constructor M which takes a single type variable a. The arrow operator $\rightarrow$ can be considered a type constructor which takes two types: the type of the argument a and the type of the reterun value b. In c++ type constructors are templates and $M\ a$ is just a much terser representation of $\mathbf{template} <$ typename **a** $>$ struct **M**..... The $a \rightarrow b$ operator corresponds to the function wrapper $std :: function < a(b) >$. A list of elements of type a is created by the list type constructor $[\ a\ ]$. This corresponds to $std :: list < a >$ or $std :: forward_list < a >$ A few cases have a special notation :

λ expressions and closures

Lambda expressions allow you to create functions on-the-fly. The lambda has a list of arguments and it's body is an expression. Variables referrence in the expression which are not speicified in the argument list 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 [fp field harrison]. This capture of the enclosing environment by the lambda expression is called a closure.
The (slightly abbreviated) C++ syntax for the lambda expression is [..refer to more complete syntax.]: $[...]\ (params) \rightarrow rettype\ body$ In this particular form the body of the lambda is unable to modify the captured variables. $[...]$ 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 declaration or $std :: function$ type variable can be used to bind the lambda expression.

```
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
```

In this example 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 capture specifier $[x, \&y]$ caused x to be captured by value and y be reference. So we would expect x to remain the same, regardless of what value it takes on later in the program. On the other hand, y is captured by reference, so if the value of y changes we should see that change reflected any subsequent execution of func. The variable inc references a lambda which captures y by reference and increments it by one. the lambda bound to inc_alt on the other hand captures y by value. The keyword $mutable$ allows the lambda expression to change y. The one bound to inc_alt_alt captures the 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 but is immutable.

```
1 [...]
2   // as opposed to [=] or [] or [1]
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
```

This snippet 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

A partial function is created when a function is supplied with less than it's full compliment of arguments. In c++ partial function application can be achieved through the use of *std::bind* and std::placeholders::... std::bind is a template function which takes a callable object like a function object or a function pointer [.ref..] as it's first argument. Subsequent arguments are either values, which will be passed on to f, or placeholders provided by std::placeholders. std::bind returns a function object which can be stored in std::function. If placeholders are used, the function object will have arguments. Each distinct place holder will correspond to an argument and each distinct placeholder can be bound to one or more arguments of the function f. Here's a simple example:

```
1 [....]
2   auto l = std::bind([=](int x, int y){return 5 * x + y; }, _1,
        10);
3   auto r = l(902);
4   cout << l(902) << endl;
5   return 0;
6 \\ print 4520 = 5 * 902 + 10
```

A lambda expression is partially applied by bunding it's first argument to a placeholder, and it's second argument to 10. The collable object l returned by $std :: bind$ takes one argument.

```
[...]
  std::function<double (double)> l1      = [](double x) { return
     2*x-0.906;};
  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);

  auto val = rpl(9, l1);
  std::cout << " result : " << val << std::endl; // print
     4145.03
```

The first argument of the lambda determines how many times the function passed in as the thrird argument is repeated. It's second argument is 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.

currying and higher order functions

Although partial function application goes a long way towards constructing functions out of othe functions there is a more powerful way to do this. *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 $f(x, y)$.

$$curry2 \ :: \ ((a,b) -> c) -> (a -> b -> c)$$

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

$$plus \ : : (int \ x, int \ y) \ :: int = x + y \ \Rightarrow \ cplus(int \ x) \ :: \ (int \to int) \to (inty) \ :: \ int \to x$$

$$plus(5,6) = 11 \ \Leftrightarrow \ (curry2 \ plus)(5)(6) = 11$$

curry2 is a higher-order function which turns a function which takes a pair of arguments into a function with returns a unary (single valued) function with an argument of type a. When we call the curried function with an argument it will return an other function which takes an argument of type b. For all intents and purposes this is a partially applied version of the orginal function with the first argument applied. The final result of type is obtained after calling the partially applied function with a parameter of type b. (curry2 plus) is the curried version of plus. I've made the return types explicit. *curry2 plus)(5)* returns a partially applied plus function, which is then called with 6 as the argument. The resut is a unsuprising 11. A simple implementation of curry2 binary functions is shown below :

```
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
```

Currying and partial function application simplify the design of higher order functions since we only have to consider unary functions. In haskell a function like $f\ x\ y\ z$ .. is a higher order function. If only only a subset of all the argument is supplied to f, a unary partiall applied function f is returned. In C++ that 's not the case, unless you specifically desing your functions to do this.

basic higher order functions

looping

A basic operation in programming is looping over data in a linear container, like a list, array or vector.
An imperative program is designed to make data changes 'in-place'. In an imperative programming this is done in a for-loop. In the body of the for-loop a reference to an element is created. Any modifications made to the element are don in place. When the for-loop is done the list we started out with has new or modified elements in it. This list can be passed on for further processing to a different for-loop, and more modifications can made to it After all this processing,

any reference to the original list will in fact be pointing to a materially different list. Such is life in an imperative world. Functional programming emphasizes referential transparency and changing data without changing the reference is avoided. In functional programming kanguage a new version of the element in the list is created if this element is mutated. This also creates a new list and you would need to rebind your reference if you want to work with the modified list. 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.
map, foldl (or reduce) and foldr are basic higher order functions used to process a container of data elements.

map

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

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

This is the dual of the visitor pattern [..COPELIEN REF]. In the visitor pattern we'd visit each element of an object graph.
The stl function std::for_each is a templated function which takes two iterators and a unary callable object as input. The callable object 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.
The stl function $std :: transform$ has two flavors [ref cppref; josuttis]: In of them $std :: transform$ takes a unary callable object as input, applies it to the elements of one range and returns an other, viz.

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

In the second it takes a binary callable function, applies it to each element of two input ranges, and returns the result in an other range :

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

A map function for the std::forward_list which uses the first flavor of std::transform function is shown below:

```
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 }
```

map is implemented as a template with to type parameters. The first parameter specifies a very generic callable. A more typesafe alternative would have the use of std::function, but that would not allow us to auto assigned or inline lambda fucntions. The type of each lamda is unique and therefore would not be convertable something more typesafe and generic like std::function. The type of the elements in the return list is determined by applying the decltype specifier to the return type of the callable *f* applied to an instance of A.

```
1 [...]
2   std::forward_list<int> L = {1,67,89,23,45,1,3,99,-90};
3   auto show = [] (int v) { std::cout << v << ","; return v;};
4
5   map(show,
6                     map ([] (int y) {return (y + 79) % 45;},
7                      map(show, L)));
8   //prints 1,67,89,23,45,1,3,99,
```

```
9  // -90,35,11,33,12,34,35,37,43,-11,
10
11   std::function< std::function <int(int)>(int)> op   = [] (int x
         ) {
12     return [=] (int y) {
13       return 4 * x + y;
14     };
15   };
```

In this example map is applied three times. The best way to look at this is from the right/bottom to the top left. In the right/bottom is the first application of show, which is followed by the one in the middle, and the last one at the start of the map expression. The first example maps the show over list the L and each element is printed to std::cout. A lambda is mapped over the list and the resuls is printed out std:cout using show.

```
1  [...]
2    map(show,
3        map([](std::function<int(int)> f){return f(2);},
4      map(op, L))); //prints : 6,270,358,94,182,6,14,398,-358,
```

The second example starts by mapping a curried function *op* over the same list L as in the previous example. The resulting list of partially applied functions is mapped over by a lambda which calls these functions with the argument 2. The result is printed to std::cout. 94 the result of 4 * 23 + 2, 182 the result of 4 * 45 + 2 etc.

```
1  [...]
2   auto l = std::bind([](std::function<int(int)> f){return f(2);},
3            std::bind(op, std::placeholders::_1));
4
5    map(show, map(l, L)); //prints 6,270,358,94,182,6,14,398,-358,
```

Here std::bind is used to combine *op* and the lamba. This eliminates one iteration and list copy and illustrates how function combination can be used to combine functions into other functions.
The second flavor of std:: transform applies a function to the elements of two lists to produce a third. This corresponds zipWith [ref haskell] :

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

foldl and reduce

foldr

conclusions