λ 's on the edge

Advanced Functional Programming in C++

Alfons Haffmans

June 11, 2013

Introduction

Functional programming emphasizes the use of immutable data strsuctures and pure functions. However, this by itself does not capture all the complexity programmers encounter. The need to deal with state and IO are obvious examples. The extension of functional programming to handle those cases requires the introduction of concepts like functors, applicative functors and monads (why ??). Programming relies on the existence of side-effects. In the late '80's and early 90's an approach was pionereed that sought to incorporate impure features into so-called pure functional programming languages. That approach relied heavily on category theory and the concept of monads. Clearly C++ already supports impure features. Does it therefore make sense to incorporate that framework into the language ?

In this article I investigated to what extend C++ can handle these concepts. In a previous article [..] I looked at what C++ supported out-of-box. None of the concepts I'm going to discuss are available in C++ or its standard libraries. In this article I'll be looking to extend C++ to support advanced functional concepts like functors, applicative functors and monads.

I have taken a few approaches : I'm using Haskell's notation for function signatures and type classes. I'm using std::forward_list as the standard list container. Extensions to either std::list or std::vector are trivial.

In functional programming everything is a function. The output of a function is determined solely by its input. Computations generalize functions in that they can return more than a single value as a consequence of function application [moggi]. This is done using concepts from category theory. This allows for the introduction of exceptions, side-effects and io if a functionla setting.

Programmers deal with exceptions,side-effects and io all the time.

For exqmple how would failure be handled in a functional setting ?

A fairly straight forward approach would be to have the function return a pair.

$$f :: a \rightarrow [bool\ ,\ b]$$

The protocol would be to check the first member of the pair and it it's true, the second slot would contain a valid value. We can generalize this to any type class M wich would represent the 'context' which contains the value we operate on.

We can rewrite the above to be :

$f :: a \to Mb$

In fact, there are a few ways to apply a function to a value in context M:

$$\begin{aligned} fmap &:: (a \to b) \to M\ a \to M\ b \\ apply &:: M\ (a \to b) \to M\ a \to M\ b \\ do &:: M\ a \to (a \to M\ b) \to M\ b \end{aligned}$$

Note that in all cases the value remains within the context. These are all higher-order functions. $fmap$ takes a function $a \to b$ and applies to a value of type a in context M. The result is a value of type b in the same context.
$apply$ uses a function $a \to b$ 'lifted' into the context M and similarly applies it to a value in the context. Note that $apply$ requires $fmap$ to be implemented.
Lastly do has a slightly different order in the type signature for historical reasons. It takes a value in a context, and feeds that value into the function $(a \to M\ b)$. Again do immplies the existence of $apply$.
These three functions more-or-less represent a more advanced way to look at functional programming. $fmap$ corresponds to mapping a function over a container of values. $apply$ is part of an applicative functor type class. It allows easier pipelining of functions working on values in contexts. do is part of the monad type class. It allows you to combine computations in the most flexible way.
In addition to the type interface each implemtation needs to satisfy a number

of laws. The justification for these laws is not so abvious from the intuitive introduction above. The justification for these laws and for the introduction of these functions can be found in abstract algebra and category theory.
I'll investigate the implemention of functors, applicative functors and monads in C++.

Currying

Currying turns any function into a higher order function of one variable [8]. The curry of the function returns a partially applied version of the original.
The operator $curry2$ is a higher order function which takes a binary function as input and returns a unary higher order function. That function is the curry of the binary function.

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

When the curried version of f is called with an argument of type a it returns another unary function Calling this function with an argument of type b returns the same value as f.

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

$(curry2plus)$ 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 1 shows an implementation of curry2 in C++:

```
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 1: curry for binary operators

Currying plays an important role in functional programming [8]. It simplifies the design of higher order functions because 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.

Functors

Let's start with just a little category theory. Category theory is the study of algebras of functions.

A category consists of objects and maps or arrows between the objects. The maps or arrows are functions which take a value from the domain and generate a value in the codomain. Maps can be composed to form another map. The composition operation is denoted as $(g \circ f)(x) = g(f(x))$ and is read as $g\ after\ f$. The objects A,B,C,.. and maps f,g,h.. need to satisfy the following conditions :

$$\begin{aligned} f \circ id_A &= id_B \circ f \\ (h \circ g)\circ &= h \circ (g \circ f) \end{aligned}$$

Think of a category as a combinations of concepts and relationships. Of particular interest are mappings between categories themselves. Mappings between categories are used to identify relationships between different sets of concepts. In this case we're trying to understand how to generalize the concepts of function and function combination.

A very simple category is the monoid category $\mathbb{1}$. It consists of a single element $\star$ and the idenity id. It's domain and codomain are the same. We can construct maps and label them as $1_+, 2_+, ..n_+...$ The composite of two maps is defined as $m_+ \circ n_+ = (n+m)_+$ so that $5_+ \circ 3_+ = 8_+$.

We can interpret this construction by letting $\star$ correspond to $\mathbb{Z}$. Each map n_+

in the monoid corresponds to a map f_n such that $f_n(x) = n + x$. It's easy to see that this correspondence respects function composition : $(f_m \circ f_n)(x) = f_m(f_n(x)) = m + n + x = f_{n+m}(x)$. f_0 is the identity operator. f_n is the a curried $+$ operator. Note that this interpreation of the monoid $\star$ is also a category. The single object is $\mathbb{Z}$ and the arrows are curried $+$ operators. What we have just defined is a mapping from one category (the monoid $(\star, n_+)$) to an other category : $(\mathbb{Z}, (n+))$. A mapping between categories is called a functor. A functor $\mathcal{F}$ maps objects and arrows or maps from one category to an other:

$$\begin{aligned} \mathcal{F}(f : A \to B) &= \mathcal{F}(f) : \mathcal{F}(A) \to \mathcal{F}(B) \\ \mathcal{F}(id_A) &= id_{\mathcal{F}(A)} \\ \mathcal{F}(g \circ f) &= \mathcal{F}(g) \circ \mathcal{F}(f) \end{aligned}$$

We can create an alternative interpretation using lists of integers of size n : $L_n = (a_1, ..., a_n$. A natural operation on lists is concatenation $\oplus$ which basicallt appends the elements of one list to the other. The identity element is the empty list L_0. The interpretation of the mappings 1_n is that of a curried concatenation to a list of size n L_n. Function combination is easily verified $(L_m \circ L_n)(x) = L_{m+n}(x)$. This interpretation satisfies the conditions of a functor between the monoid $(\star, n_+)$ and the monoid $(\mathcal{L}, (L_n \oplus))$.

We can now also define a functor between the category $(\mathbb{Z}, f)$ and the category $(\mathcal{L}, (L_n \oplus)$. What that functor does is apply functions defined between elements of $\mathcal{Z}$ to computations involving lists of integers. The functor preserves the

structure of the category of $(\mathbb{Z}, f)$.

Can we construct a functor mapping between $(\mathbb{Z}, f)$ and $(\mathcal{L}, (L_n \oplus))$?. The Functor $\mathcal{F}$ preserves the structure (what does that mean ??) of the category $(\mathbb{Z}, f)$ if it satifies :

$$\mathcal{F} \circ f = \mathcal{F}(f) \circ \mathcal{F}$$

A reasonable choice of $\mathcal{F}$ is $x \to [x]$ i.e. we map each element of $\mathbb{Z}$ to a single list. In that case we would have $(\mathcal{F} \circ f)(x) = \mathcal{F}(f(x)) = [f(x)]$. So that $\mathcal{F}(f) : [x] \to [f(x)]$.

In other words the functor would apply the mapping (or function) $f : \mathbb{Z} \to \mathbb{Z}$ to the element in the list. It's easy to see that this mapping $\mathcal{F}$ satifies the requirement of a functor : $\mathcal{F}(id(x)) = [id(x)] = [x] = [] \oplus [x]$.

I've obviously ignored a lot of details. That said , it's important to note that this approach relies strictly on statements about function composition.

Functors in functional programming generalize the concept of mapping a function over values in a container of typeclass M. The typeclass in Haskell is :

class Functor f where

$$fmap \;::\; (a \to b) \to f\ a \to f\ b$$

The $fmap$ function generalizes the simple functor $\mathcal{F}$. $fmap :: a \to b$ corresponds

to $\mathcal{F}(f)$. In terms of $fmap$ the functor requirements read

$$\begin{aligned} fmap\, id &= id \\ fmap(g \circ f) &= (fmap\, g) \circ (fmap\, f) \end{aligned}$$

$fmap\, id$ returns a function which takes a container as an argument. Note that the id function on the left-hand side takes an value of type a whereas the one on the right-hand side take a container of type a.

In C++ this is defined in the template

```
1  template <template<typename T1, typename... D> class F>
2  struct functor {
3    template<typename A, typename B>
4    static std::function < F<B> (F<A>)> fmap(std::function <B (A)>
         f);
5
6    template<typename A, typename B>
7    static F<B> fmap(std::function <B (A)> f, F<A> L) {
8      return fmap(f)(L);
9    }
10 };
```

The class F can have more than one template parameter, as indicated by the variadic template. This allows specialization for containers since they have more than one template parameter.

For lists like $std :: list$ and $std :: foward\_list$ fmap corresponds to map.

A shared pointer is a container which holds a value of a particular type.

An implementation of $fmap$ would return new shared pointer holding the result of the function applied to the value of the shed pointer.

```
1  template <>
2  struct functor<std::shared_ptr> {
3    template<typename A, typename B>
4    static std::function<std::shared_ptr<B> (std::shared_ptr<A>)>
         fmap (std::function<B(A)> f) {
5      return [=](std::shared_ptr<A> v) {
6        if (v) {
7    return std::make_shared<B>(f(*v));
8        }
9        return  std::shared_ptr<B>(nullptr);
10     };
11   }
12
13   template<typename A, typename B, typename F>
14   static std::function<std::shared_ptr<B> (std::shared_ptr<A>)>
         fmap (F f) {
15     return [=](std::shared_ptr<A> v) {
16       if (v) {
17   return std::make_shared<B>(f(*v));
18       }
19       return  std::shared_ptr<B>(nullptr);
20     };
21   }
22 };
```

Listing 2: shared pointer as container

```
1  typedef std::tuple<int,std::string> C;
2  std::list<std::shared_ptr<C>> L = {std::shared_ptr<C>{new C
       (10, "a")}, std::shared_ptr<C>{new C(20, "b")}, std::
       shared_ptr<C>{new C(3467, "mnhjk")}};
3  auto F = functor<std::shared_ptr>::fmap(std::function<C (C)
       >([](const C& c) { std::cerr << c << std::endl; return c;})
       );
4  functor<std::list>::fmap(F, L);
5  return 0;
```

Listing 3: mapping over a list of shared pointers

Listing 3 applies a function to a list of shared pointers using $fmap$. First I construct a list of shared pointer to a tuple of string and integer. I then create a function F by partially applying $fmap$ for a shared pointer with the a lambda which prints the tuple to stdout. The lambda is written in terms of the tuple type. I have decoupled the container type from the value type.

```
1 template <typename A, typename B> struct unary_op {
2
3 };
```

Listing 4: unary operator

The type in 4 conforms to the type required by functor.

```
1 template<>
```

```
2  struct functor<unary_op>
3  {
4    template<typename A, typename B, typename R>
5    static auto  fmap (std::function<B(A)> f) {
6      return [f](std::function<A(R)> g)  {
7        return [f,g] (R x) {
8    return f(g(x));
9        };
10     };
11   };
12
13   template<typename A, typename B, typename R>
14   static std::function<B (R)> fmap (std::function<B(A)> f, std::
       function<A(R)> g) {
15     return [f,g](R x) -> B {
16       return f(g(x));
17     };
18   };
19
20 };
```

A unary operator $\rightarrow$ constructs a unary function :$(\rightarrow)r\ a => (r \rightarrow a)$. So for a unary operator M in $fmap$ would be $(\rightarrow)r$ and fmap :

$fmap\ (a \rightarrow b) \rightarrow (r \rightarrow a) \rightarrow (r \rightarrow b)$

This corresponds to function composition. $fmap$ takes a function $(a \rightarrow b)$ and

applies it $after$ function $(r \to a)$ to yield a function $(r \to b)$.

Applicative Functors

Applicative functors have the following type class

$$\begin{aligned} &\text{class (Functor f)} \;\Rightarrow\; \text{Applicative f where} \\ &\qquad pure \;::\; a \to f\ a \\ &\qquad (<\star>) \;::\; f\ (a \to b) \to f\ a \to f\ b \end{aligned}$$

The applicative functor adds the infix operation $<\star>$ and operation $pure$ to those of the functor. In C++ we don't have the same leeway in constructing functions names and fixity as we do in Haskell. I'll call $(<\star>)$ $apply$ and use it in prefix mode.

The $apply$ or $<\star>$ operation of th applicative functor looks similar to $fmap$, except that the function $a \to b$ is inside the 'container' f. $pure$ puts a value of type a into the container.

There is a close relation ship between functors and applicative functors :

$$fmap f x = pure f <*> x$$

I.e. lifting a function f into an environment using $pure$ and applying to a value in a context is the same as using $fmap$ to apply f directly.

Applicative functors address an obvious limitation of functors: What if we wanted

to apply a function to multiple effectful results ?

```
template <template<typename T1, typename... D> class F>
struct applicative_functor : public functor <F>
{

  template <typename A>
  static F<A> pure(A val);

  template<typename A, typename B>
  static std::function < F<B> (F<A>)> apply(F <std::function<B(A
      )>> f );
};
```

Listing 5: applicative functor template in C++

?? shows the applicative functor as a template in C++ and an implementation would be provided for each individual container.

```
typedef std::tuple<int,std::string> W;
  std::forward_list<std::shared_ptr<W>> L = {std::shared_ptr<W>{
      new W(10, "a")},
                                              std::shared_ptr<W>{
                                                  new W(20, "b")},
                                              std::shared_ptr<W>{
                                                  new W(3467, "
                                                  mnhjk")}};
  std::function<W (W)> show =[](const W& w) { std::cerr << w <<
      std::endl; return w;};
  auto lifted_show = applicative_functor<std::forward_list>::
```

```
    pure(applicative_functor<std::shared_ptr>::apply(
    applicative_functor<std::shared_ptr>::pure(show)));
applicative_functor<std::forward_list>::apply(lifted_show)(L);
```

Listing 6: using an applicative functor to map over a list of shared pointers

In 5 an applicative functor is used to map the fucntion $show$ over a list of shared pointers to three tuples. The argument to show is a tuple; not a pointer to a tuple. First we 'lift' show into the $std :: shared_ptr$ context, using $pure$. This gives us a shared pointer which contains $show$. We then partially apply $applicative_functor < std :: shared_ptr >:: apply$ with this lifted version. This is a function which takes a shared pointer as input and applies $show$ to its content. This function is now lifted into the $std :: forward_list$ context, i.e. inserted into a list. This is then applied to the list of shared pointers. In fact it's the same result as 3.

Monads

Do-like Notation

Conclusions

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*