λ 's on the edge

Advanced Functional Programming in C++

Alfons Haffmans

May 20, 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 :

$f :: a \rightarrow Mb$

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

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

Note that in all cases the value remains within the context. These are all higher-order functions. $fmap$ takes a function $a \rightarrow b$ 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 $a \rightarrow b$ '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 $(a \rightarrow M\ b)$.
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 [**?**]. 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.

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

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.

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

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

```cpp
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 1: curry for binary operators

Currying plays an important role in functional programming [**?**]. 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 [**?**, **?**]. However writing a curry operator or writing curried versions of a function has become a lot easier now that lambda's are supported.

Functors

Functors generalize the concept of mapping a function over values in a container. Their typeclass is :

class Functor f where

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

In C++ this is defined in the template

```cpp
template <template<typename T1, typename... D> class F>
struct functor {

  //curried version
  template<typename A, typename B>
  static std::function < F<B> (F<A>)> fmap(std::function <B (A)>
      f);

  // uncurried, for functions..
  template<typename A, typename B>
  static F<B> fmap(std::function <B (A)> f, F<A> L) {
    return fmap(f)(L);
  }
};
```

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.

Useful functors satisfy the following laws

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

. is the function composition operator, read as $after$. It applies the function g after the fucntion f, viz.

$(g\ .\ f)(x) = g(f(x))$

The expressions in eq... are curried. $fmap\ id$ returns a function which takes a container as an argument. The first law essentially says that that is the same thing as applying the id operator to the container itself. Note that the id fu-unction is polymorphic. The id functions 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.
The second equation implies that the order ...... doesn't matter....

Lists

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

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

  //curried version
  template<typename A, typename B>
  static std::function < F<B> (F<A>)> fmap(std::function <B (A)>
      f);

  // uncurried, for functions..
  template<typename A, typename B>
  static F<B> fmap(std::function <B (A)> f, F<A> L) {
    return fmap(f)(L);
  }
};
```

Listing 2: fmap

pointers

In the case of pointer $fmap$ corresponds to derefencing the pointer.

```
template <>
struct functor<std::shared_ptr> {
  template<typename A, typename B>
  static std::function<std::shared_ptr<B> (std::shared_ptr<A>)>
      fmap (std::function<B(A)> f) {
    return [=](std::shared_ptr<A> v) {
      if (v) {
  return std::make_shared<B>(f(*v));
      }
      return  std::shared_ptr<B>(nullptr);
    };
  }

  template<typename A, typename B, typename F>
  static std::function<std::shared_ptr<B> (std::shared_ptr<A>)>
      fmap (F f) {
    return [=](std::shared_ptr<A> v) {
      if (v) {
  return std::make_shared<B>(f(*v));
      }
      return  std::shared_ptr<B>(nullptr);
    };
  }
};
```

binary operators

```
template <typename A, typename B> struct binary_op {

};
```

Listing 3: binary operator

The type in **??** coforms to the type required by functor.

```
template<>
struct functor<binary_op>
{

  // This does not work
  template<typename A, typename B, typename R>
  static std::function<std::function<B(R)> (std::function<A(R)>)
      > fmap (std::function<B(A)> f) {
    return [&](std::function<A(R)> g) -> std::function<B(R)> {
      return [&] (R x)->B {
  return f(g(x));
      };
    };
  };

// this does
  template<typename A, typename B, typename R>
  static std::function<B (R)> fmap (std::function<B(A)> f, std::
      function<A(R)> g) {
    return [=](R x) -> B {
      return f(g(x));
    };
  };

};
```

Applicative Functors

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*