Functional Idioms in C++

Alfons Haffmans

September 6, 2013

Abstract

In this paper I review the implementation of functors, applicative functors and monads in C++.

Contents

1 Introduction

1.1 Functional Programming

Functional programming emphasizes the use of immutable data structures and pure functions. A pure function's output is determined solely by its input. However, this does not capture all the complexity of realistic programs, like state, IO and exceptions.

In the late '80's and early 90's an approach was pioneered which incorporates impure features, like exceptions, side-effects and IO into functional programming languages. That approach relied heavily on concepts from category theory like functors and monads. Clearly C++ already supports impure features so does it therefore make sense to incorporate that framework into the language ? In this article I'll attempt to present an answer to that question.

1.2 Support in C++

1.3 Conventions and Notation

2 Functional Idioms

2.1 Introduction

To start off, let's try to model failure in a functional setting. A fairly straightforward approach is to have the function return a Boolean and a value as part of a pair. In Haskell's function notation the signature of that function is :

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

The function f takes a value of type a an returns a pair of a Boolean and a value of type b. If the Boolean is true, an exception has occurred, and the other member of the pair is ignored. If it's false, the second slot contains the result of the computation. We can generalize this to a type class M:

$$f :: a \rightarrow Mb$$

M represents the 'context' in which the computation occurs. Here are a few ways to apply a function to a value in context M:

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

$$bind \;::\; M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$$

These are all higher-order functions are higher order functions, which apply a function to a value in a context.

$fmap$ 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. $fmap$ corresponds to mapping a function over a container of values.

$apply$ 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. $apply$ is part of an applicative functor type class. It allows easier pipe-lining of functions working on values in contexts.

Lastly $bind$ 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 $(a \rightarrow M\ b)$. $bind$ implies the existence of $apply$. $bind$ is part of the monad type class. Note that the functions applied by $fmap$ and $apply$ are pure functions. $bind$ takes a function which returns a value in a context. It allows you to combine computations in the most flexible way.

In addition to the type interface each implementation needs to satisfy a number of laws. The justification for these laws is not so obvious from the intuitive introduction above, and can be found in abstract algebra and category theory.

2.2 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.

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

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.

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

$(curry2plus)$ 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++:

```
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.

2.3 Functor

Let's start with just a little bit of category theory.

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 to a value in the co-domain. A simple example is the function $f(n) = 2 * n + y$ where n is a integer. Here domain and co-domain are the same.

Maps can be combined to form other maps. This 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.. form a category if they satisfy :

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

Of particular interest are mappings between categories themselves.

A very simple category is the monoid category $\mathbb{1}$. It consists of a single element $\star$ and the identity id. It's domain and co-domain 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 $f_n(x) = n + x$, which is a curried $+$ operator. It's easy to see that this correspondence respects function composition : $(f_m \circ f_n)(x) = f_m(f_n(x)) = m + n + x = f_{n+m}(x)$. and that f_0 is the identity operator.

Note that this interpretation of the monoid $\star$ is also a category. 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:

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

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

We can create an alternative interpretation using lists of integers of size n : $L_n = (a_1, ..., a_n)$. The identity element is the empty list L_0. A natural operation on lists is concatenation $\oplus$ which appends the elements of one list to the other. We define the function $f_n(x) = L_n \oplus [x]$

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 satisfies :

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

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 $\mathcal{F}$ satisfies 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

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

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.

2.4 Applicative Functors and Brackets

2.4.1 Applicatives

Applicative functors address an obvious limitation of functors: What if we wanted to apply a function to multiple effectful results ? They were first discussed by

McBride and Patterson [...], although they were documented before that. In Haskell's notation applicative functors have the following type class

$$\begin{array}{rcl} \text{class (Functor f)} & \Rightarrow & \text{Applicative f where} \\ pure & :: & a \rightarrow f\ a \\ (<\star>) & :: & f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \end{array}$$

Here f represent the container. The applicative functor adds the infix operation $<\star>$ and operation $pure$ to those of the functor.
$pure$ puts a value of type a into the container.
There is a close relation ship between functors and applicative functors :

$$\text{fmap} f\ x = \text{pure} f < * > x$$

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.

2.4.2 Brackets

In their paper [....] McBride and Paterson introduce a convenient 'bracket' notation to capture the application of a pure function f to a sequence of sub computations :

$$[\![f\, L_1 \ldots L_n]\!] = f < * > L_1 < * > \ldots < * > L_n$$

$pure$ lifts the pure function f into the applicative functor, which is then applied to the containers in the remainder of the bracket. Note that the structure of the computation remains fixed. This makes brackets easier to use than using applicatives directly. The essence of the bracket notion is captured in listing 4

2.5 Monads

The monad type class has the following declaration in Haskell :

class Monad m where

$$\begin{aligned} \text{return} &:: a \rightarrow m\,a \\ (>>=) &:: m\,a \rightarrow (a \rightarrow m\,b) \rightarrow m\,b[...] \end{aligned}$$

Functors an applicative functors apply pure functions to values in containers. Pure functions have their place. They are easily combined and heir results are easily verified. However pure functions don't capture all the complexity of a program. For example the computation defined in the body of the function may fail. We may want to combine functions like that as part of a multi step computation. In the case of failure in one of the steps the remainder of the computation should be discarded. An other example is IO. The return type of these functions is the IO channel, so these functions cannot be pure. We may want to combine functions which write data to an IO channel.
Yet we do want to combine functions like that in a meaningful way.
Monads need to satisfy certain laws, the justification of which can be found in

category theory.

[TBD]...

3 The Functional Idioms in C++

3.1 General Approach

3.2 Functor

```
template <template<typename T1, typename... D> class F>
struct functor {
  template<typename A, typename B>
  static std::function < F<B> (F<A>)> fmap(std::function <B (A)>
      f);

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

Listing 2: functor in C++

Listing 2 shows a template which defines the functor $fmap$ from type A to type B for a type class F. 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.

3.3 Applicative Functor and Brackets

3.3.1 Applicative

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 the applicative functor looks similar to $fmap$, except that the function $a \to b$ is inside the 'container' f.

```
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 3: applicative functor template in C++

3 shows the applicative functor as a template in C++. Each individual container needs to provide its own implementation.

3.3.2 Bracket

```
template <template<typename Tx, typename... D> class Cont,
    typename F, typename... T>
auto bracket (F f, Cont<T>... L)
{
  auto cf = curry<decltype(f), T...  >(f);
  return bracket_helper<sizeof...(T), Cont , decltype(cf), T
      ...>::bracket(cf, L...);
}
[...]
template<template<typename Tx, typename... D> class Cont,
    typename F, typename T1, typename T2>
struct bracket_helper<2, Cont, F, T1, T2> {
  static auto bracket(F cf, Cont<T1> L1, Cont<T2> L2) {
    return bracket(cf)(L1)(L2);
  }

  static auto bracket(F cf) {
    typedef decltype(cf(T1())(T2())) ret_t;
    return [cf] (Cont<T1> L1) {
      auto C  = bracket_helper<1, Cont, F, T1>::bracket(cf)(L1);
      return [C](Cont<T2> L2) {
        applicative_functor<Cont> APF;
        auto J  = APF.template apply<T2, ret_t>(C)(L2);
        return J;
      };
    };
  };
};
[...]
```

Listing 4: applicative brackets in c++

Brackets are implemented by currying the function and applying the partially applied function toe each subsequent container.

First we convert n-ary the function f by currying it : $f :: (x, y, z) \rightarrow k \Rightarrow f_c :: x \rightarrow y \rightarrow z \rightarrow k$. f_c is lifted into the container using pure pure $f_c = Lx \rightarrow y \rightarrow z \rightarrow k$. This is then applied to the first container L_1 and results in a lifted partially applied function $Ly \rightarrow z \rightarrow k$. That is we have taken the function f_c and applied it to the value in the first container. The result is a container of a function with a cardinality one less then f_c.

We keep doing this until a result is returned. The implementation below applies the curried function recursively to a the containers. The bottom of the recursion is reached when the last container is processed.

3.4 Monad

```
template <template<typename T1, typename... D> class F>
struct monad : public applicative_functor <F>
{
  template <typename A> static F<A> mreturn (A val) {
    return applicative_functor<F>::pure(val);
  }

  template<typename A, typename B>
  static std::function < F<B> (std::function< F<B> (A) > ) >
      bind(F<A> val);

};
```

Listing 5: monad defintion in C++

Monads extend applicative functors and that's reflected in the definition shown in listing 5. I've used $mreturn$ for $return$ and $bind$ for $>>=$. $mreturn$ is identical to the $pure$ function for applicative functors. I use a variadic template to allow monad to be specialized for stl containers which have multiple template arguments. The bind method in 5 is curried : It takes a monad of type A and return a function. This function takes an argument of type A and returns a monad of type B, just like haskell's monad definition shown above.

4 A simple example: Maybe and Option

4.1 Maybe

4.2 Option

5 Lists and ZipLists

5.1 Functor

For lists like $std :: list$ and $std :: foward\_list$ fmap corresponds to map and is equivalen to iteration over the elements in the list.

5.2 Applicative Functor

5.2.1 List

For lists there a are two possible implementations for the applicative functor. In both cases a list of functions is applied to a list of values, but in one case we apply each function to every value and in te other we apply a function only if we have a value in the corresponding position in the argument list. In the latter case we 'zip' the list of the functions and values.

```
template<> struct
applicative_functor<std::forward_list> :public functor<std::
    forward_list>{

  template<typename A>
  static std::forward_list<A> pure(A v) {
    return std::forward_list<A>(1,v);
  }
[...]
  template<typename A, typename B, typename lambda>
  static std::function< std::forward_list<B> (std::forward_list<
      A>)> apply(std::forward_list<lambda> F) {
        return [F](std::forward_list<A> L) {
        std::forward_list<B> acc;
        for (auto& func : F) {
          for (auto& arg : L) {acc.push_front(func(arg));}
          }
        acc.reverse();
        return acc;
        };
      };
};
```

Listing 6: std::forward_list is an applicative functor

The applicative functor for $std :: forward\_list$ is shown in listing 6. The body of apply consists of two nested loops traversing the input lists.

5.2.2 ZipList

```
template<typename A> using zip_list = std::list<A>;
[...]
template<> struct
applicative_functor<zip_list> :public functor<zip_list>{

    template<typename A>
    static zip_list<A> pure(A v) {
    return applicative_functor<std::list>::pure<decltype(v)>(v);
    }

[...]
    template<typename A, typename B, typename lambda>
    static std::function< zip_list<B> (zip_list<A>)> apply(
        zip_list<lambda> F) {
    return [F](zip_list<A> L) {
      zip_list<B> acc;
      auto it1 = F.begin();
      auto it2 = L.begin();
      while (it1 != F.end() && it2 != L.end()) {
        auto func = *it1;
        auto arg  = *it2;
        acc.push_front(func(arg));
        it1++;
        it2++;
      }
      acc.reverse();
      return acc;
    };
  };

};
```

Listing 7: ziplist is an applicative functor

In listing 7 I show the implementation of the applicative functor for the zip list based on $std :: list$. The implementation of the $functor$ is a straight forward map and is not shown. In the body of the $apply$ method, the two lists are traversed in parallel. The traversal stops when either one of the lists has run out of elements.

```

 forward_zip_list<int> L = {2, 5, 10};
 auto f = [](const int& c) { std::cerr << c << ","; return c;};
 auto lifted_lambda_1 = applicative_functor<std::list>::pure(f)
    ;
 applicative_functor<std::list>::apply<int,int>(lifted_lambda_1
    )(L);
 std::cerr << std::endl;
   //prints 2,5,10
   [...]
 auto lifted_lambda = applicative_functor<zip_list>::pure(f);
 applicative_functor<zip_list>::apply<int,int>(lifted_lambda)(L
    );
   //prints 2,
```

Listing 8: the applicative functor for list and ziplist

Listing 8 shows a simple example of how the applicative functors for list and zip list differ. A simple lambda function is lifted into a list and applied to a list of integers. For the list case all the elements are printed. For the zip list only one element is printed because the list of functions has only one element.

5.3 Brackets

```
[...]
  typedef int         T1;
  typedef char        T2;
  typedef std::string T3;
  typedef applicative_functor<std::forward_list> apf_t;
  m_t<T3> L3 = {std::string("hello"), std::string("goodbye")};
  auto f3 = [] (T1 a, T2 b, T3 c) { return std::make_tuple(a,b,c);};
  auto R3 = bracket(f3, L1, L2, L3);
  std::cout << R3 << std::endl;
  [...]
  [(1,y,hello),(1,y,goodbye),(1,x,hello),(1,x,goodbye),(2,y,hello),(2,y,goodbye),(2,x,hello),(2,x,goodbye),(3,y,hello),(3,y,goodbye),(3,x,hello),(3,x,goodbye),]
```

Listing 9: applicative functor for the std::forward_list using brackets

Listing 9 I show how brackets cane be used to apply a function to multiple lists.

```
  template<typename a> using s_t = std::shared_ptr<a>;
  typedef applicative_functor<std::shared_ptr> apf_t;
  s_t<T1> L1 = apf_t::pure(1);
  s_t<T2> L2 = apf_t::pure('a');
  s_t<T3> L3 = apf_t::pure(std::string("hello"));
  auto f3 = [] (T1 a, T2 b, T3 c) { return std::make_tuple(a,b,c);};
  auto R3 = bracket(f3, L1, L2,L3);
  std::cout << R3 << std::endl; //std::shared_ptr<St5tupleIJicSsEE>((1,a,hello))
```

Listing 10: applicative functor for shared pointers applied using bracket notation

In listing 10 the data in three separate pointers is combined to create a pointer to a tuple. Again notice that the function is pure in that it does not reference the container the data was in.

5.4 Monad

```
template<> struct monad<std::list> : public applicative_functor<std::list> {

  template<typename A, typename B>
  static std::function < std::list<B> (std::function< std::list<B> (A) > ) > bind(std::list<A> M) {
    return [M](std::function<std::list<B> (A)> f) {
      std::list<B> R;
      std::list<std::list<B>> res = map(f, M);
      for (auto& list : res) {
        R.insert(R.end(), list.begin(), list.end());
      }
      return R;
    };
  }
      [...]
};
```

Listing 11: monad for std::list

The implementation for the stl $std :: list$ container is showm in listing 11. I've elided the implementation of $mreturn$, which is passing its argument on to the applicative functor's pure method. Function f is mapped over list M. Since the return type of f is a list the result is a list of lists. The this list is flattened, i.e. the lists are merged into a single list B which is then returned. This flattening operation is an essential part of the monad. It allows monadic operations to be combined.

6 Shared Pointers

6.1 Shared pointers

A shared pointer is another example of a container which holds a value of a particular type. The functor implementation for shared pointers is shown in list 12. $fmap$ returns a new shared pointer holding the result of the function application. We need to return a new shared pointer because the input and return types of f may be different. Also note that if the derefrence fails, an empty shared pointer is returned, rather than an exception thrown.

```
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);
    };
  }
};
```

Listing 12: shared pointer as container

6.2 Functor

6.3 Applicative Functor

6.4 Monad

```
template<> struct monad<std::shared_ptr> : public
    applicative_functor<std::shared_ptr> {

  template<typename A, typename B>
  static std::function < std::shared_ptr<B> (std::function< std
      ::shared_ptr<B> (A) > ) > bind(std::shared_ptr<A> M) {
    return [M](std::function<std::shared_ptr<B> (A)> f) {
      if (M) {
        return f(*M);
      }
      return  std::shared_ptr<B>();
    };
  }
[..]
};
```

Listing 13: monad implemention for std::shared_ptr

In 13 I show the monad implementation for $std :: shared_ptr$. The validity of the shared pointer is checked before it is dereferenced and its value passed on to the function f. If the shared pointer is empty (or invalid), an empty shared pointer of type B is returned. This implies that a chain of monadic computations can return null if any one of its individual computations fails.

7 list of shared pointers

7.1 Functor

```
typedef std::tuple<int,std::string> C;
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")}};
auto F = functor<std::shared\_ptr>::fmap(std::function<C (C)>([](const C& c) { std::cerr << c << std::endl; return c;}));
functor<std::list>::fmap(F, L);
return 0;
```

Listing 14: mapping over a list of shared pointers

A functor gives the ability to apply a pure function to a value in a container. This decouples the values and the functions that operate on them from the containers these values may be in. For example, consider a list of shared pointers.
In listing 14 I define a function $show$ which prints out the value of type C. I then combine two functors, one for each container to apply $show$ to each shared pointer in the list. Lists of pointers are relatively standard in C++ programs. But functors can also be defined for unary functions $a \rightarrow b$.

7.2 Applicative Functor

Applicative functors can be extended to combinations of containers like lists of shared pointers. The code snippet in listing 15 shows the functor implementation for a list of shared pointers. $fmap$ can be written as a straightforward combination of fmap for the shared pointer and the list container. Listing 15 shows a code snippet of the applicative functor definition for a forward list of shared pointers. In this case we can't write the applicative as a combination of two applicative functors. The fcuntion in the definition for the applicative functor for lists is not a reference. $apply$ encapsulates the access to the data stored in the shared pointer elements of the list.

```
struct functor<forward_list_of_ptr> {
[...]
  template<typename A, typename B, typename F>
  static std::function<forward_list_of_ptr<B> (
      forward_list_of_ptr<A>)> fmap (F f) {
    auto F = functor<std::shared_ptr>::fmap<A,B>(f);
    return [F](forward_list_of_ptr<A> L) {
    return functor<std::forward_list>::fmap(F, L);
    };
  }
};
```

Listing 15: fmap implementation for a list of shared pointers

```
template<typename A> using forward_list_of_ptr = std::
    forward_list<std::shared_ptr<A>>;
```

```
[...]
template<> struct
applicative_functor<forward_list_of_ptr> :public functor<
    forward_list_of_ptr> {
[...]
    template<typename A, typename B, typename lambda>
    static std::function< forward_list_of_ptr<B> (
        forward_list_of_ptr<A>)> apply(forward_list_of_ptr<lambda
        > F) {
    return [F](forward_list_of_ptr<A> L) {
      forward_list_of_ptr<B> acc;
      for (auto& func : F) {
        for (auto& arg : L) {
          auto res = applicative_functor<std::shared_ptr>::apply
              <A,B,lambda>(func)(arg);
          acc.push_front(res);
        }
      }
      acc.reverse();
      return acc;

    };
  };
}
```

Listing 16: applicative functor for a list of shared pointers

```
    forward_list_of_ptr<int> L = {std::make_shared<int>(5),std::make_shared<int>(15),std::make_shared<int>(25),std::make_shared<int>(35)};
    auto f = [](const int& c) { std::cerr << c << ","; return c;};

    functor<forward_list_of_ptr>::fmap<int,int>(f)(L);

    auto lifted_lambda = applicative_functor<forward_list_of_ptr>::pure(f);
    applicative_functor<forward_list_of_ptr>::apply<int,int>(lifted_lambda)(L);
  std::cerr << std::endl;
```

Listing 17: example for list of pointers

In listing 17 I show how a function can be mapped over a list of pointers, using its functor and appplicative. The results are not too surprising.

7.3 Monad

8 Unary Operations

8.1 Unary Operations and Curried Functions

8.2 Functor

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

};
```

Listing 18: unary operator

First I define the unary operator type in 18. A unary operator → constructs a unary function :$(\rightarrow)r\ a => (r \rightarrow a)$. Next I'll provide an implementation for unary operators in 19

```
template<>
struct functor<unary_op>
{
  template<typename A, typename B, typename R>
  static auto  fmap (std::function<B(A)> f) {
    return [f](std::function<A(R)> g)  {
      return [f,g] (R x) {
  return f(g(x));
      };
    };
  };

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

};
```

Listing 19: functor for unary operators

For a unary operator f in $fmap$ would be $(\rightarrow)r$ and fmap :

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

$fmap$ takes a function $(a \rightarrow b)$ and applies it $after$ function $(r \rightarrow a)$ to yield a function $(r \rightarrow b)$. This corresponds to function composition.

8.3 Applicative Functor

8.4 Monad

9 Stateful Computations

9.1 Capturing State

A change in state s is represented by a function $s \rightarrow (a, s)$. a is the value associated with the state change. The state s is mutable.

```
template<typename A, typename S>
struct state_tuple {
  explicit state_tuple (S s) : e(std::make_pair(A(), s)), set(
      false){}
  state_tuple (A a, S s) : e(std::make_pair(a,s)), set(true) {}
  state_tuple(const state_tuple& s) : e(s.e), set(s.set){}
  std::ostream& pp(std::ostream& strm) const {
    if (set) {
      strm << e;
    }
    else {
      strm << "((),"<< e.second << ")";
    }
    return strm;
  }

  std::pair<A,bool> value() const {
    return std::make_pair(e.first, set);
  }

  std::pair<S,bool> state() const {
    return std::make_pair(e.second, true);
  }

private:
  std::pair<A,S> e;
  bool set;
};
```

Listing 20: state tuple

The state tuple in listing 20 is a thin wrapper around $std :: pair$ which adds a few convenience methods. A state computation takes a state of type S and returns a state tuple of types A and S.

```
template<typename A, typename S>
using state_computation = std::function< state_tuple<A,S> (S)>;
```

Listing 21: state computation albel

The state class shown in 22 encapsulates the state computation and adds a few convenience methods.

```
template <typename A, typename S>
struct state
{
  explicit state(state_computation<A,S> C) : C(C){}
  state(const state& o) : C(o.C){}
  state& operator= (const state& o) {
    if (&o == this) {
      return *this;
    }
    C      = o.C;
    return *this;
  }
  std::ostream& pp(std::ostream& strm) const {
    strm << "[state < " << typeid(A).name() << "," << typeid(S).
        name() << "]";
    return strm;
  }

  state_tuple<A,S> run_state(S state) {
    return C(state);
  }

private:
  state_computation<A,S> C;
};
```

Listing 22: state

```
template<typename A, typename S>
state_tuple<A,S> runState(state<A,S> M, S state)
{
  return M.run_state(state);
}
```

Listing 23: runState

The run_state method is key to the use of the state class and the function $runState$ executes this method by passing a state to it.

```
[...]
  std::default_random_engine de;
  std::uniform_int_distribution<int> di(10, 20);
  state_computation<int,std::uniform_int_distribution<int>>
      getrand = [&de] (std::uniform_int_distribution<int> s) {
    auto val = s(de);
    return state_tuple<int, std::uniform_int_distribution<int>
        >(val, s);
  };

  state <int, std::uniform_int_distribution<int>> ST(getrand);
  int n = 10;
  auto S = runState(ST, di);
  std::cerr << "iter : " << n << " " << S << std::endl;;
  while ( n-- > 0) {
    S = runState(ST, S.state().first);
    std::cerr << "iter : " << n << " " << S << std::endl;;
  }
[..]
```

Listing 24: example of the use of the state class

The use of the state class is illustrated in listing 24 using a random number generator. A number is drawn from a uniform distribution of integers. $getrand$ is the state computation: It takes the current state of the uniform distribution and returns a state tuple containing a random value as well as the new state. In the while loop the state returned by the state computation is the used to generate the next state. A monad can be used to glue subsequent state computations together.

The state in listing 22 contains a value of type A as well as a state of type S. This makes it a little different of the list or ptr containers which have a single type constructor. We going to make the reasonable assumption that the type of the state is not going to change between subsequent computations, although the type of the value could. For example we could change the example above to have $getrand$ return a string in stead of an integer if the integer exceeds some threshold. We are less likely to want to combine results by different random number generators.

9.2 Functor

9.3 Applicative Functor

9.4 Monad

```
template<> struct monad<state> : public applicative_functor<
    state> {

  template<typename S, typename A, typename B>
  static state<B,S> bind(state<A,S>& M, std::function< state<B,S
      > (A)>& f) {
    state_computation<B,S> comp =[&f,&M](S s) {
      auto res                = runState(M, s);
      state<B,S> newval       = f (res.value().first);
      return runState(newval, res.state().first);
    };
    return state<B,S> (comp);
  };

  template <typename S, typename A> static state<A,S> mreturn (A
       val) {
    return applicative_functor<state >::pure<S,A>(val);
  }

};
```

Listing 25: state monad

The state monad in listing 25 takes a function with an argument of type A and a state with a value of type A and state of type S and returns a state of the same type but with a value of type B. This state contains a computation constructed in the body of the $bind$ method. In the body of $comp$ a new state is generated by calling $runState$ on M, which is the state passed into bind. The function f is then called on the value generated by the state computation. The result is a new state which is run with the new value. $comp$ is returned as the new state in the $state$ constructor.

```
[...]
  typedef std::uniform_int_distribution<int> idist;
  std::default_random_engine de;

  state_computation<int, std::uniform_int_distribution<int>>
      getrand = [&de] (std::uniform_int_distribution<int> s) {
    auto val = s(de);
    return state_tuple<int, std::uniform_int_distribution<int>
        >(val, s);
  };

  state <int, idist> ST(getrand);

  std::function<state<int, idist >(int)> f = [&de, &ST](int val) {
    std::cerr << val <<std::endl;
```

```
        return ST;
    };

    auto S1 = monad<state>::bind<idist, int, int>(ST, f);
    auto S2 = monad<state>::bind<idist, int, int>(S1, f);
    auto S3 = monad<state>::bind<idist, int, int>(S2, f);
    auto S4 = monad<state>::bind<idist, int, int>(S3, f);
    auto S5 = monad<state>::bind<idist, int, int>(S4, f);
    auto S6 = monad<state>::bind<idist, int, int>(S5, f);
    auto S7 = monad<state>::bind<idist, int, int>(S6, f);
    auto S8 = monad<state>::bind<idist, int, int>(S7, f);
    auto Sf = monad<state>::bind<idist, int, int>(S8, f);

      auto S = runState(Sf, idist (10, 20));
    std::cerr << S << std::endl;;
    S = runState(Sf, idist (100, 200));
    std::cerr << S << std::endl;;
```

Listing 26: example of the state monad

The state monad is used to string various stateful computations together. In listing 26 I revisit the random number generator example discussed earlier. $getrand$ is a computation which gets an integer from the random number generator s passed into it as an argument. ST is the initial state. The state monad is used to construct a state which represents 10 calls to $getrand$. The function f prints the result of the random number generator to stderr. Notice that when last state Sf is constructed no random numbers have been generated yet. That is done by the call to runState. First The resulting computation is called with a distribution engine with a range between 10 and 20, and next with one with a range from 100 to 200.

The results of the random number generation in listing 26 are not available for further processing. To collect the results we need to extend the state to include a list and update the list with the result of the number generator.

```
[...]
  typedef std::list<int>                          icont_t;
  typedef std::uniform_int_distribution<int>  idist_t;
  typedef std::pair<icont_t, idist_t>           state_t;
  typedef state_tuple<int, state_t>             state_tuple_t;

  std::default_random_engine de((unsigned int)time(0));

  state_computation<int,state_t>  getrand = [&de] (state_t s) {
    auto val = s.second(de);
    s.first.push_back(val);
    return state_tuple_t(val, s);
  };

  std::function<state_computation<int,state_t> (int, int)>
      getrand2 = [&de](int f, int t) {
    return [&de,f,t] (state_t s) {
      auto val = s.second(de);
      return state_tuple_t(val, std::make_pair(s.first, idist_t(
          f,t)));
    };
  };

  state <int, state_t> ST(getrand);

  std::function<state<int,state_t>(int)> f = [&ST,&getrand2](int val) {
    std::cerr << val <<std::endl;
    if (val % 7 == 0) {
      return state<int,state_t> (getrand2(10000,11456));
    }
    return ST;
  };

  auto S1 = monad<state>::bind<state_t,int,int>(ST,f);
  auto S2 = monad<state>::bind<state_t,int,int>(S1,f);
[...]
  auto Sf = monad<state>::bind<state_t,int,int>(S12,f);

  auto S = runState(Sf, std::make_pair(icont_t(), idist_t(10,20)));
  std::cerr << S << std::endl;;
  S = runState(Sf, std::make_pair(icont_t(), idist_t(100,200)));
  std::cerr << S << std::endl;;
[...]
```

Listing 27: extended state monad example

The valure in the state monad is not being used. In listing 27 the state is extended to include a list of value. In the state computation $getrand$ the random value is inserted into the list as well are returned as part of the state tuple. The second computation $getrand2$ is a curried function whose firts argument is a new range for the uniform distribution. It returns a state computation whihc uses this new distribution range. The monadic function f noew returns a different state, depending on whether the value passed in was a multiple of 7.

10 Futures and Future Values

10.1 Futures and Async

10.2 Capturing Future Values

10.3 Functor

10.4 Applicative Functor

10.5 Monad

11 Discussion

The implemention of...