# Monoids, Monads, and Applicative Functors

## Repeated Software Patterns

Stellar Science

David Sankel - 5/10/2016 - C++Now 2016

# Why Functional Software Patterns?

# In a nutshell...

```
class Foo {
public:
  // What goes here?
private:
  // etc.
};
```

# Answer Design Questions

- What are the "fundamental" operations?
- What can the user do with the provided flexibility?
- How does this relate to other classes?

# Functional Patterns are Not for the User

- User's don't recognize them.
  - "What's a monadic bind operation?"
- Code becomes less clear.
  - Generic code is fun, but readable code is better.

# A Bit of History

# Haskell

- An academic programming language modeled after math.
- Purity is a really big deal.
- Emphasis on purity left Haskell 1.0 a toy language. Streams used as the IO model.

# Then monads hit the stage...

- 1989, Eugenio Moggi uses monads to *describe* model states and exceptions.
- 1992, Wadler uses monads to *express* IO in Haskell.
- Category Theory becomes popularized in Functional Programming.

---

# What is Category Theory?

- Investigation started in 1942 by Eilenberg and Mac Lane.
- Attempt to abstract over various mathematical domains and map between them.
- Generality made it applicable to many domains, even music theory.
- Category, Functor, Duality, Monad, Isomorphisms, ...
- All about widely repeated patterns.

# Monoid

A monoid is a type `T` with a binary operation $\oplus$ which combines its values.

- $\oplus$ must be associative. That is, `(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)` for any `a`, `b`, `c` which have type T.
- `T` has a special value `e` such that `e ⊕ x = x ⊕ e = x` for every value `x`.

# Numeric types with + are monoids.

- `(a + b) + c = a + (b + c)` for all numeric types.
- `0` is the special value. We have `0 + x = x + 0 = x` for every value `x` which is numeric.

# Numeric types with * are monoids.

- `(a * b) * c = a * (b * c)` for all numeric types.
- `1` is the special value. We have `1 * x = x * 1 = x` for every value `x` which is numeric.

## `unsigned` **with** `std::max` **forms a monoid.**

- `max(max(a, b), c) = max(a, max(b,c))`.
- `0` is the special value. We have
  `max(0, x) = max(x, 0) = x`
  for every `unsigned` `x`.

# Monoids are all over the place.

**What is the `e` value for the `float`, `std::min` monoid?**

## Does `std::vector` form a monoid with something?

What can you do with a list of monoids?

# Fancy Monoids

# std::optional template

A `std::optional<T>` is either "null" or has a value of type `T`.

```
std::optional<double> o = std::nullopt;
// o is null, viz. 'bool(o) == false'
o = 3.0
// o has value 3.0, viz. 'bool(o) == true' and '*o == 3.0'
```

# Is `std::optional` a monoid?

# A `std::optional` monoid is a monoid

```
std::optional<SomeMonoid> append(std::optional<SomeMonoid> lhs,
                                 std::optional<SomeMonoid> rhs) {
  if( !lhs )
    return rhs;
  else {
    if( rhs )
      return std::optional<SomeMonoid>( *lhs ⊕ *rhs );
    else
      return lhs;
  }
}
```

# Another `std::optional` monoid

```
template<typename T>
std::optional<T> append(std::optional<T> lhs,
                        std::optional<T> rhs) {
  if( !lhs )
    return rhs;
  else {
    if( rhs )
      return rhs;
    else
      return lhs;
  }
}
```

# Do functions returning `R` form a monoid?

# Functions of the same type that return monoids are a monoid

## Search for n best occurances of a word in a million documents

- Key insight: An n-heap is a monoid.
- Split documents between cluster nodes.
- Send word to each cluster node.
- Each cluster node generates a heap using parallelization.
- Each cluster node sends its heap to a collection node.
- The collection node joins the heaps.

# Monoids

- Monoids scale very well.
- Monoids compose via. functions, optional, and other things.
- Monoids are common.

# Functor

A functor is a class template (`Functor<T>`) with a single template parameter and a callable (`map`) which have the following properties.

- `map(f, a)` is a legal expression when:
  - `a` is a value of type `Functor<T>` for some type `T`.
  - `f` is a callable that accepts a single argument of type `T`.
  - Let `U` be the result type of of `f(t)` where `t` has type `T`. The result of `map(f, a)` is type `Functor<U>`.
- If `f(t) == t` for all values `t` of type `T`, then `map(f, a) == a` for all values `a` of type `Functor<T>`.
- `map(g, map(f, a)) = map(gf, a)` where

```
auto gf = [f,g](auto t){ return g(f(t)); }
```

# Functor: Intuition

- Functors are like containers.
- `map` applies a function to the *thing in the container* resulting in a new container.
- The laws provide reasonable rules that allow `map` composition.

# std::vector is a functor

```
template<typename T, typename U>
std::vector<U> map(const std::function<U (T)> &f,
                   const std::vector<T> &a ) {
  std::vector<U> result;
  for(const T & t : a)
    result.push_back(f(t));
  return result;
}
```

# A more efficient map

```
template<typename T, typename F>
auto map(F f, const std::vector<T> &a ) {
  std::vector<std::result_of_t<F(T)>> result;
  for(const T & t : a)
    result.push_back(f(t));
  return result;
}
```

# Is `std::optional` is a functor?

# Is `std::set` is a functor?

# Is `std::pair` is a functor?

# Is `std::function` is a functor?

# std::function **functor**

```
template<typename P, typename T, typename U>
std::function<U (P)> map(std::function<U(T)> f,
                         std::function<T(P)> a) {
  return [f,a](P p){ return f(a(p)); };
}
```

**functors allow for transformation's within**

Each map strips away one layer of your datatype.

Say you have a `std::vector<std::optional<int>>` and want to get strings for each int.

```
std::vector<std::optional<std::string>> getStrings(
    const std::vector<std::optional<int>>& ints) {
  auto f1 = [](auto i){ return std::to_string(i) };
  auto f2 = [](auto o){ return map(f1, o) };
  auto f3 = [](auto v){ return map(f2, v) };
  return f3(ints);
}
```

# Applicative Functor

An applicative functor (`Applicative<T>`) is a functor with two extra operations, `pure` and `apply`, which obey the following rules.

- `pure(t)` where `t` is of type `T` results in a value of type `Applicative<T>`.
- `apply(aff, afv)` is a legal expression when:
  - `aff` has type `Applicative<F>` for some callable `F` where `f(t)` is well defined if `f` is of type `F` and `t` is of type `T`.
  - `afv` has type `Applicative<T>` for some type `T`.
  - `apply(aff, afv)` has type `Applicative<U>` iff the result type of `f(t)` is `U`.

# Applicative Functor Laws

- If `f(t) == t` for all `t`, then `apply(pure(f), a) == a` for all `a`.
- `apply(pure(f), pure(t)) == pure(f(t))` for all `f` and `t`.
- `apply(a, pure(t)) == apply(pure(f), a)` when `f(g) == g(t)` for all `g`.
- `apply(a, apply(b, c)) == apply(apply(apply(pure(f), a), b), c)` when `f(g, h)(t) == g(h(t))` for all `g`, `h`, and `t`.
- `map(f, a)` == `apply(pure(f), a)`

# Applicative Functor Intuition

- `pure` wraps a value into the container.
- `apply` applies a contained function to a contained value to get a contained result.

Note that `apply` can be extended to `n` argument functions.

# `std::optional` applicative functor

Pure:

# `std::optional` applicative functor

Apply:

# `std::optional` applicative functor properties

Consider

```
const std::optional<double> a = /* etc. */;
const std::optional<double> b = /* etc. */;
const std::optional<double> c = apply(std::plus<>(), a, b);
const std::optional<double> d = apply(std::negate<>(), c);
```

# Or

```
const double myValue = /* etc. */;

const std::optional<Command> cmd = /* fetch command */;

const std::optional<std::function<double(double)>> func
    = map(funcFromCommand, cmd);

const std::optional<double> result = apply(func, pure(myValue));
```

## Is `std::vector` an applicative functor?

- Is it a functor? What is `map`?
- What is `pure`?

## `apply` **for** `std::vector`

You've got a vector of functions, `[f1, f2, ...]` and a vector of values `[t1, t2, ...]`. What can you do?

# `std::vector` **nondeterminism**

```
const std::vector<double> a {1.0, 2.0, 3.0};
const std::vector<double> b {10.0, 20.0, 30.0};
const std::vector<double> c = apply(pure(std::plus<>()), a, b);
```

## Many applicative functors

- `std::future`
- continuations
- exception-style errors
- behaviors in functional reactive programming
- parsers
- etc.

# Parser applicative functors

What are the fundamental operations for a parser?

Let `Parser<T>` be a stdin parser that parses into type T.

If `p` has type `Parser<T>`, `p.read()` tries to parse stdin. If it succeeds, it returns type T, otherwise it throws an exception.

```
template<typename T>
class Parser<T> {
public:
  T read();

  // Friend functions go here.
private:
  Parser<T>(){}
  std::function<T ()> m_reader;
  bool m_consumesNothing;
  std::vector<char> m_startChar;
};
```

# Some friend functions:

```
// Read any single char from stdin.
Parser<char> charP();

// Read particular char from stdin.
Parser<char> charP(char);

// Parse with 'lhs' if stream begins with the correct character,
// otherwise 'rhs'.
template<typename A>
Parser<A> either(Parser<A> lhs, Parser<A> rhs)

// Parse with 'a' as many times as possible.
template<typename A>
Parser<std::vector<A>> zeroOrMore(Parser<A> a);
```

# either

```
template<typename A>
Parser<A> either(Parser<A> lhs, Parser<A> rhs) {
  Parser<A> result;
  result.m_consumesNothing = lhs.consumesNothing;
  result.m_startChar = append(lhs.m_startChar, rhs.m_startChar);
  result.m_reader = [lhs, rhs](){
    const char c = std::cin.peek();
    if(std::find(lhs.m_startChar.begin(), lhs.m_startChar.end(), c)
        != lhs.m_startChar.end()) {
      return lhs.read();
    else
      return rhs.read();
  };
  return result;
}
```

# **What is** pure**?**

# What is `apply`?

# And we're done.

Everything else can be built on these pieces.

```
Parser<int> digitP = apply(
  successP( [](char c){ return c - '0'; } ),
  either( charP('0'), either( charP('1'), … )));

Parser<int> intP = apply(
  successP( /* convert digits to int */ ),
  zeroOrMore( digitP ) );

template<typename A, typename B>
Parser<std::pair<A,B>> operator>>(Parser<A> aParser,
                                  Parser<B> bParser) {
  return apply( /* etc. */ );
}

auto twoIntsP = intP >> charP(' ') >> intP;
```

# Review

- Monoids → Highly parallel patterns ($\oplus$).
- Functors → Do things to the stuff inside (`map`).
- Applicative Functors → Put stuff inside (`pure`). The stuff inside can do things to the stuff inside (`apply`).

# Monad

A monad (`Monad<T>`) is an applicative functor with an extra operation `join` which obeys the following rule.

- `join(a)` where `a` is of type `Monad<Monad<T>>` results in a value of type `Monad<T>`.

# Monad Laws

A bit more complex to express in C++.

- Joining outside-in vs. inside-out shouldn't make a difference.
- Similarly for `pure` and `join`.
- See https://en.wikibooks.org/wiki/Haskell/Category_theory

# Join for `std::optional`

```
template<typename T>
std::optional<T> join(std::optional<std::optional<T>> a) {
  if( a )
    return *a;
  else
    return std::nullopt;
}
```

# Other monads

- `std::vector`
- parser
- functions with a single parameter of type `A`

# So, what's the big deal?

# The monad `bind` operation

The monadic bind operation is defined in terms of the other operations. For a given `Monad<T>`:

```
template<typename T, typename U>
Monad<U> bind(Monad<T> m, std::function<Monad<U>(T)> f ) {
  return join(apply(pure(f), m));
}
```

Usually, you'd like `bind` to be an operator overload. Lets use `>>`.

```
// given
std::optional<int> getInt();

std::optional<int> result =
  getInt() >> [=](auto a) {
    return getInt() >> [=](auto b) {
      return getInt() >> [=](auto c) {
        return pure(a+b+c);
      }
    }
  };
```

# Indent a bit differently...

```
std::optional<int> result =
         getInt() >> [=](auto a) {
  return getInt() >> [=](auto b) {
  return getInt() >> [=](auto c) {
  return pure(a+b+c);
      }
    }
  };
```

# Now squint and you'll see something like this...

```
std::optional<int> result =
        getInt() >> a
        getInt() >> b
        getInt() >> c
        pure(a+b+c);
  ;
```

Now cross your eyes and you'll see...

```
auto a = getInt();
auto b = getInt();
auto c = getInt();
return a+b+c;
```

Which looks a lot like imperative computation.

# And that was a really big deal for Haskell

# But, what do monads do for us?

- Express different models of computation within C++.
  - `std::vector` gives a language with nondeterminism.
  - `std::optional` provides a language with error fallthrough.
  - Continuation language, etc.
- Provide more control over computation.
  - Serialize and de-serialize computations.
  - Command pattern embedded language.
- Imperative template metaprogramming.

# Lets wrap it up...

# This is just the beginning...

More interesting patterns:

- Semigroup
- Category
- Arrow
- Comonad

# Repeated Software Patterns

Monoid, Functor, Applicative Functor, and Monad.

Any Questions?

# Further information:

- https://wiki.haskell.org/Typeclassopedia
- Category Theory for Computing Science by Michael Barr and Charles Wells