# COMP3141

Software System Design and Implementation



## Functors, Applicatives, and Monads

Liam O'Connor
University of Edinburgh (LFCS (and UNSW))
Term 2 2020

## Motivation



We'll be looking at three very common abstractions:

- used in functional programming and,
- increasingly, in imperative programming as well.

## Motivation



We'll be looking at three very common abstractions:

- used in functional programming and,
- increasingly, in imperative programming as well.



Unlike many other languages, these abstractions are reified into bona fide type classes in Haskell, where they are often left as mere "design patterns" in other programming languages.

## Kinds



Recall that terms in the type level language of Haskell are given *kinds*.
The most basic kind is written as `*`.



- Types such as `Int` and `Bool` have kind `*`.
- Seeing as `Maybe` is parameterised by one argument, `Maybe` has kind `* -> *`: given a type (e.g. `Int`), it will return a type (`Maybe Int`).



**Question**: What's the kind of `State`?

# Functor

Recall the type class defined over type constructors called `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```



**Functor Laws**

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`



We've seen instances for lists, `Maybe`, tuples and functions.
Other instances include:



- `IO` (how?)
- `State s` (how?)

# Functor

Recall the type class defined over type constructors called `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```



**Functor Laws**

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`



We've seen instances for lists, `Maybe`, tuples and functions.
Other instances include:



- `IO` (how?)
- `State s` (how?)
- `Gen`

Demonstrate in live-coding

## QuickCheck Generators



Recall the `Arbitrary` class has a function:

```
arbitrary :: Gen a
```

The type `Gen` is an abstract type for QuickCheck generators. Suppose we have a function:



```
toString :: Int -> String
```

And we want a generator for `String` (i.e. `Gen String`) that is the result of applying `toString` to arbitrary `Int`s.



Then we use `fmap`!

# Binary Functions



Suppose we want to look up a student's zID and program code using these functions:

```
lookupID :: Name -> Maybe ZID
lookupProgram :: Name -> Maybe Program
```



And we had a function:

```
makeRecord :: ZID -> Program -> StudentRecord
```

How can we combine these functions to get a function of type `Name -> Maybe StudentRecord`?

# Binary Functions



Suppose we want to look up a student's zID and program code using these functions:

```
lookupID :: Name -> Maybe ZID
lookupProgram :: Name -> Maybe Program
```



And we had a function:

```
makeRecord :: ZID -> Program -> StudentRecord
```

How can we combine these functions to get a function of type `Name -> Maybe StudentRecord`?



```
lookupRecord :: Name -> Maybe StudentRecord
lookupRecord n = let zid     = lookupID n
                     program = lookupProgram n
                 in ?
```

## Binary Map?



We could imagine a binary version of the `maybeMap` function:

```
maybeMap2 :: (a -> b -> c)
          -> Maybe a -> Maybe b -> Maybe c
```

# Binary Map?



We could imagine a binary version of the maybeMap function:

```
maybeMap2 :: (a -> b -> c)
          -> Maybe a -> Maybe b -> Maybe c
```



But then, we might need a trinary version:

```
maybeMap3 :: (a -> b -> c -> d)
          -> Maybe a -> Maybe b -> Maybe c -> Maybe d
```



Or even a 4-ary version, 5-ary, 6-ary . . .

this would quickly become impractical!

# Using Functor

Using `fmap` gets us part of the way there:

```
lookupRecord :: Name -> Maybe (Program -> StudentRecord)
lookupRecord  n = let zid     = lookupID n
                      program = lookupProgram n
                  in fmap makeRecord zid
                     -- what about program?
```

But, now we have a function inside a `Maybe`.

# Using Functor

Using `fmap` gets us part of the way there:

```
lookupRecord :: Name -> Maybe (Program -> StudentRecord)
lookupRecord  n = let zid     = lookupID n
                      program = lookupProgram n
                  in fmap makeRecord zid
                  -- what about program?
```

But, now we have a function inside a `Maybe`.

We need a function to take:

- A `Maybe`-wrapped fn `Maybe (Program -> StudentRecord)`
- A `Maybe`-wrapped argument `Maybe Program`

And apply the function to the argument, giving us a result of type `Maybe StudentRecord`?

## Applicative

This is encapsulated by a subclass of Functor called Applicative:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

# Applicative



This is encapsulated by a subclass of Functor called Applicative:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```



Maybe is an instance, so we can use this for lookupRecord:

```
lookupRecord :: Name -> Maybe StudentRecord
lookupRecord n = let zid     = lookupID n
                     program = lookupProgram n
                 in fmap makeRecord zid <*> program
               -- or pure makeRecord <*> zid <*> program
```

## Using Applicative



In general, we can take a regular function application:

```
f a b c d
```



And apply that function to `Maybe` (or other `Applicative`) arguments using this pattern (where `<*>` is left-associative):



```
pure f <*> ma <*> mb <*> mc <*> md
```

## Relationship to Functor

All law-abiding instances of `Applicative` are also instances of `Functor`, by defining:

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

Sometimes this is written as an infix operator, <$>, which allows us to write:

```
pure f <*> ma <*> mb <*> mc <*> md
```

as:

```
f <$> ma <*> mb <*> mc <*> md
```

**Proof exercise:** From the applicative laws (next slide), prove that this implementation of `fmap` obeys the functor laws.

# Applicative laws



```
-- Identity
pure id <*> v = v

-- Homomorphism
pure f <*> pure x = pure (f x)

-- Interchange
u <*> pure y = pure ($ y) <*> u

-- Composition
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

These laws are a bit complex, and we certainly don't expect you to memorise them, but pay attention to them when defining instances!

## Applicative Lists



There are two ways to implement `Applicative` for lists.

```
(<*>) :: [a -> b] -> [a] -> [b]
```

## Applicative Lists



There are two ways to implement `Applicative` for lists.

```
(<*>) :: [a -> b] -> [a] -> [b]
```

1. Apply each of the given functions to each of the given arguments, concatenating all the results.

## Applicative Lists



There are two ways to implement `Applicative` for lists.

```
(<*>) :: [a -> b] -> [a] -> [b]
```



1. Apply each of the given functions to each of the given arguments, concatenating all the results.
2. Apply each function in the list of functions to the corresponding value in the list of arguments.

## Applicative Lists



There are two ways to implement `Applicative` for lists.

```
(<*>) :: [a -> b] -> [a] -> [b]
```



1. Apply each of the given functions to each of the given arguments, concatenating all the results.
2. Apply each function in the list of functions to the corresponding value in the list of arguments.



**Question:** How do we implement `pure`?

## Applicative Lists



There are two ways to implement `Applicative` for lists.

```
(<*>) :: [a -> b] -> [a] -> [b]
```



1. Apply each of the given functions to each of the given arguments, concatenating all the results.
2. Apply each function in the list of functions to the corresponding value in the list of arguments.



**Question:** How do we implement `pure`?

The second one is put behind a `newtype` (`ZipList`) in the Haskell standard library.

## Other instances

- QuickCheck generators: Gen

# Other instances



- QuickCheck generators: Gen

Recall from Wednesday Week 4:

```
data Concrete = C [Char] [Char]
  deriving (Show, Eq)

instance Arbitrary Concrete where
  arbitrary = C <$> arbitrary <*> arbitrary
```

# Other instances



- QuickCheck generators: Gen
  Recall from Wednesday Week 4:

```
data Concrete = C [Char] [Char]
  deriving (Show, Eq)

instance Arbitrary Concrete where
  arbitrary = C <$> arbitrary <*> arbitrary
```

- Functions: (r -> x)

# Other instances



- QuickCheck generators: `Gen`

Recall from Wednesday Week 4:

```
data Concrete = C [Char] [Char]
  deriving (Show, Eq)

instance Arbitrary Concrete where
  arbitrary = C <$> arbitrary <*> arbitrary
```

- Functions: `((->) x)`
- Tuples: `((,) x)` We can't implement `pure` without an extra constraint!
- `IO` and `State s`:

## On to Monads



- Functors are types for containers where we can `map` pure functions on their contents.

## On to Monads



- Functors are types for containers where we can `map` pure functions on their contents.
- Applicative Functors are types where we can combine $n$ containers with a $n$-ary function.

# On to Monads



- Functors are types for containers where we can `map` pure functions on their contents.
- Applicative Functors are types where we can combine $n$ containers with a $n$-ary function.



The last and most commonly-used higher-kinded abstraction in Haskell programming is the `Monad`.



**Monads**

Monads are types `m` where we can *sequentially compose* functions of the form `a -> m b`

# Monads

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```



Sometimes in old documentation the function `return` is included here, but it is just an alias for `pure`. It has nothing to do with `return` as in C/Java/Python etc.



Consider for:

- `Maybe`
- Lists

# Monads

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```



Sometimes in old documentation the function `return` is included here, but it is just an alias for `pure`. It has nothing to do with `return` as in C/Java/Python etc.



Consider for:

- `Maybe`
- Lists
- `(x ->)` (the Reader monad)

# Monads

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```



Sometimes in old documentation the function `return` is included here, but it is just an alias for `pure`. It has nothing to do with `return` as in C/Java/Python etc.



Consider for:

- `Maybe`
- Lists
- `(x ->)` (the Reader monad)
- `(x,)` (the Writer monad, assuming a `Monoid` instance for `x`)

# Monads

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```



Sometimes in old documentation the function `return` is included here, but it is just an alias for `pure`. It has nothing to do with `return` as in C/Java/Python etc.



Consider for:

- `Maybe`
- Lists
- `(x ->)` (the Reader monad)
- `(x,)` (the Writer monad, assuming a `Monoid` instance for `x`)
- `Gen`
- `IO`, `State s` etc.

## Monad Laws



We can define a composition operator with (>=>):

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) x = g x >>= f
```

# Monad Laws



We can define a composition operator with (>=>):

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) x = g x >>= f
```



**Monad Laws**

```
f <=< (g <=< x) == (f <=< g) <=< x -- associativity
pure <=< f      == f               -- left identity
f <=< pure      == f               -- right identity
```



These are similar to the monoid laws, generalised for multiple types inside the monad. This sort of structure is called a *category* in mathematics.

## Relationship to Applicative



All `Monad` instances give rise to an `Applicative` instance, because we can define `<*>` in terms of `>>=`.



```
mf <*> mx = mf >>= \f -> mx >>= \x -> pure (f x)
```

This implementation is already provided for `Monads` as the `ap` function in `Control.Monad`.

# Do notation



Working directly with the monad functions can be unpleasant.
As we've seen, Haskell has some notation to increase niceness:



```
do x <- y
   z
```

becomes

```
y >>= \x -> do z
```



```
do x
   y
```

becomes

```
x >>= \_ -> do y
```

We'll use this for most of our examples.

## Examples

**Example (Dice Rolls)**

Roll two 6-sided dice, if the difference is $< 2$, reroll the second die. Final score is the difference of the two die. What score is most common?

# Examples

**Example (Dice Rolls)**

Roll two 6-sided dice, if the difference is $< 2$, reroll the second die. Final score is the difference of the two die. What score is most common?

**Example (Partial Functions)**

We have a list of student names in a database of type `[(ZID, Name)]`. Given a list of zID's, return a `Maybe [Name]`, where `Nothing` indicates that a zID could not be found.

# Examples



**Example (Dice Rolls)**

Roll two 6-sided dice, if the difference is $< 2$, reroll the second die. Final score is the difference of the two die. What score is most common?



**Example (Partial Functions)**

We have a list of student names in a database of type `[(ZID, Name)]`. Given a list of zID's, return a `Maybe [Name]`, where `Nothing` indicates that a zID could not be found.



**Example (Arbitrary Instances)**

Define a `Tree` type and a generator for search trees:

```
searchTrees :: Int -> Int -> Generator Tree
```

## Homework



1. Next programming exercise is out now, due in Week 8.
2. This week's quiz is also up, due in Friday of Week 8.