COMP3141

Liam O'Conn
University of Edinburgh, LFCS (an
Term 2, 2020

Motivation

We'll be looking at t

- used in funct
- increasing

Motivation

We'll be looking at t

- used in funct
- increasing

Unlike many other languages, these abstractions are reified i
in Haskell, where they are often left as mere "design patterns" in o
languages.

Kinds

Recall that terms in the type level language of Haskell are given *kinds*.

The most basic kin

- Types such a
- Seeing as `Maybe` `Maybe` nd `* -> *`: given a type (e.g. `Int`), it will return a type (

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 fun.

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 fun.

Other instances include:

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

Demonstrate in live-coding

QuickCheck Generators

Recall the `Arbitrary` class has a function:

```
arbitrary :: Ge
```

The type `Gen` function:


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

And we want a generator for `String` (i.e. `Gen St ng`

`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
lookupProgr
```

And we had a functi

```
makeRecord :: Z
```


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
lookupProgr
```

And we had a functi

```
makeRecord :: Z
```


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 -
             _
```

Binary Map?

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

```
maybeMap2 :: (a -
             -
```


But then, we might

```
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
```


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
```


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

We need a function to take:

- A `Maybe`-wrapped fn `Maybe (Program -> Stude`
- 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) ->
```

Applicative

This is encapsulated by a subclass of Functor called Applicative:

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


Maybe is an insta

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

Using Applicative

In general, we can take a regular function application:

And apply that fun sing 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:

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

-- Homomorphi
pure f <*> pure x = pur

-- 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
```

Applicative Lists

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

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

1. Apply each o
   all the result

Applicative Lists

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

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


1. Apply each o
   all the result
2. Apply each function in the list of functions to the correspo
   of arguments.

Applicative Lists

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

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

1. Apply each o
   all the result
2. Apply each function in the list of functions to the correspo
   of arguments.

Question: How do we implement `pure`?

Applicative Lists

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

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

1. Apply each o
   all the result
2. Apply each function in the list of functions to the correspo
   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 Concr
  deriving
```



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

Other instances

- QuickCheck generators: Gen

Recall from Wednesday Week 4:

```
data Concr
  deriving
```



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


- Functions: (i -> x)

Other instances

- QuickCheck generators: `Gen`

Recall from Wednesday Week 4:

```
data Concr
  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.
- Applicativ ers with a *n*-ary function.

On to Monads

- Functors are types for containers where we can `map` pure functions on their contents.
- Applicativ ers with a *n*-ary function.

The last and most c ing is the `Monad`.

Monads

Monads are types `m` where we can *sequentially co* `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 c.

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 c.

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 c.

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 c.

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               -- l
f <=< pure      == f               -- r
```


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` inst n define `<*>`
in terms of `>>`


```
mf <*> mx = mf >>= \f -> mx
```

This implementation is already provided for `Mon`
`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 ->
z
```


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 tw

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 tw

Example (Par

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` i nd.

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 tw

Example (Par

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` i nd.

Example (Arbitrary Instances)

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

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

Homework

1. Next prog
2. This week's