COMP3141

Software System Design and Implementation

Lecture 6: Monads, Applicatives

Zoltan A. Kocsis
University of New South Wales
Term 2 2022

Kinds

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

- Types such as `Int` and `Bool` have kind `*`.
- Since `Maybe` takes a type argument, it has kind `* -> *`; e.g. given a type `Int`, it will return a type `Maybe Int`.
- As we have seen, `State` has kind `* -> * -> *`.

Functor

Last time we looked at the `Functor` type class, where `f` has kind `* -> *`.

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

Functor Laws

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

We've seen instances for lists, `Maybe`, functions.

Monads

Last time we also defined our own `State` type using

```
type State s a = s -> (s,a)
```

and explored the following functions:

State

```
bindS :: State s a -> (a -> State s b) -> State s b
yield :: a -> State s a
```

Maybe

```
bindM :: Maybe a -> (a -> Maybe b) -> Maybe b
Just :: a -> Maybe a
```

These proved to be useful abstractions, reducing repetition in code, eliminating classes of bugs. Today we'll look at the *Monad* type class, which abstracts the similarities between these two solutions.

Monads

The most commonly-used abstraction for kinds `* -> *` in Haskell programming is the `Monad`.

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

The `(>>=)` operator is pronounced *bind*. Examples seen so far:

- `Maybe`
- `State s` for any type `s`

NB The standard library defines monads a bit differently: for the actual definition see the section on Applicatives.

Monad Laws I

Usually, type classes come with laws. `Monad` is no exception.

Monad Law 1: η-associativity

Given `f :: m a` and `g :: a -> m b`, and `h :: b -> m c`,

```
(f >>= \x -> g x) >>= \y -> h y      ==
f >>= (\x -> g x  >>= \y -> h y)
```

Allows us to write unambiguously e.g.

```
use :: State Integer Integer
use =
  get           >>= \x ->
  put (x + 1) >>= \_ ->
  return x
```

Monad Law II

As η-associativity governs `>>=`, so we have two laws governing `return`.

Monad Law 2: `return` right identity

Given `f :: m a`,

```
f >>= \x -> return x    ==
f
```

The other side is a bit more complicated.

Monad Law 3: `return` left identity

Given `x :: a` and `f :: a -> m b`,

```
return x >>= \y -> f y    ==
f x
```

You'll have to learn the three monad laws!

Kleisli Category

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

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

Monad Laws Restated

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

These look like the monoid laws. The difference is that in a monoid, any two elements can be combined using the monoid operation; here, two elements can be combined only if their types check out (if they are *composable*). This sort of structure is called a *category* in mathematics.

The category above is the *Kleisli category* of the monad. The monad laws state that the Kleisli category is a category.

Do notation

In older versions of the Haskell language, working directly with the monad functions wasquite unpleasant: it required a whole lot of extra parentheses.
This is why Haskell has `do` notation.

```
do x <- f
   rest
```

becomes

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

```
do f
   rest
```

becomes

```
f >>= \_ -> do rest
```

I'll try to use it as little as possible in this course, but you'll see it used in real-world Haskell very frequently.

Do notation example

Recall that we wrote

```
use :: State Integer Integer
use =
  get          >>= \x ->
  put (x + 1) >>= \_ ->
  return x
```

Using do notation, we could instead write

```
use :: State Integer Integer
use = do
  x <- get
  put (x + 1)
  return x
```

An unusual monad

We've worked out two examples of monads last time, `Maybe` and `State s`. This time we study the standard monad structure on list types, `[]`.

Unusally, I'll define the operations first, and explain how they work.

Then I'll provide some motivating problems.

We'll have to define two functions,

```
returnL :: a -> [a]
bindL :: [a] -> (a -> [b]) -> [b]
```

Demo: list monad operations

Motivation: enumeration

If you play pen-and-paper RPGs, you might see instructions like:

Dungeons, Dragons

On your character sheet, a damage roll is written like this: `2d6+3`. This means roll two six-sided dice, add their results, then add another 3.

You might ask questions like: *what's the probability that I deal more than 8 damage?*
Recall that this probability is:

$$\frac{\text{num. cases where I deal more than 8 dmg}}{\text{num. all possible outcomes}}$$

The list monad allows you to get exact answers to questions like these, by enumerating all the relevant cases and outcomes.
Demo: 2d6 list monad

Motivation: backtracking search

The list monad is also a powerful way of implementing backtracking search. Examples where backtracking can be used to solve puzzles or problems include:

- Programming puzzles: eight queens
- AI and generation in puzzle games: crosswords, sudoku, peg solitaire.
- Combinatorial optimization: knapsack problem, etc.

Common Divisors

Simple example: can the numbers `6, 21, 15, 3, 10` be arranged in such a way that any two consecutive numbers have a common divisor?

Demo: Backtracking

Unary Map

Consider the `fmap` function for `Maybe`:

```
maybeMap :: (a -> b)  -> Maybe a -> Maybe b
maybeMap f Nothing = Nothing
maybeMap f (Just x) = Just (f x)
instance Functor Maybe where
  fmap = maybeMap
```

This allows us to write e.g.

```
ghci> fmap (\x -> x + 2) (Just 3)
Just 5
```

but not

```
ghci> fmap (+) (Just 3) (Just 2)
error: The function 'fmap' is applied to three arguments,
       but its type has only two.
```

Binary Map?

It would be useful to have maybeMap2 function:

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

so that

```
*> maybeMap2 (+) (Just 3) (Just 2)
Just 5
*> maybeMap2 (+) Nothing (Just 2)
Nothing
```

But then, we might need a ternary 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:

```
ghci> :t fmap (+) (Just 3)
fmap (+) (Just 3) :: Maybe (Int -> Int)
```

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

We need a function to take:

- A `Maybe`-wrapped fn `Maybe (Int -> Int)`
- A `Maybe`-wrapped argument `Maybe Int`

And apply the function to the argument, giving us a result of type `Maybe Int`.

Applicative

This is encapsulated by the `Applicative` type class:

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

This is a subclass of `Functor`: every `Applicative` has to be a functor. `Maybe` is an instance, so we can use this:

```
ghci> fmap (+) (Just 3) <*> Just 2
Just 5

ghci> pure (+) <*> Just 3 <*> Just 2
Just 5

ghci> pure (+) <*> Nothing <*> Just 2
Nothing
```

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 (see laws later) instances of `Applicative` are also instances of `Functor`, by defining:

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

Usually `fmap` is written infix operator, <$>, which allows us to write

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

as

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

Relationship to Monad

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

```
pure = return
f <*> x =
  f >>= \f' ->
  x >>= \x' ->
  return (f' x')
```

But many law-abiding instances of `Applicative` are *not* instances of `Monad`!

Monads from Applicative

Since every `Monad` is an `Applicative` (but not vice versa!), the Haskell standard library defines monads using

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

I.e. if you declare a `Monad` instance, you have to declare an `Applicative` instance as well!
NB You can implement the function `return` too, but it is just an alias for `pure`.

Applicative laws

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

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

-- Interchange
f <*> pure y = pure (\g -> g y) <*> f

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

These laws are not as convenient as the `Functor` and `Monad` laws; pay attention when defining instances!

FIN

1. **Thanks!**
2. The last quiz is due 23:59 Thursday, 14 July 2022.
3. The last exercise is due 09:10 Thursday, 14 June 2022.