



CS 115

Functional Programming

Lecture 10: April 25, 2016

Monads, part 2

The **Monad** type class



Functional Programming: Spring 2015



Today

- Monadic function composition
- Monadic function application
- The `>>=` and `>=>` operators
- The `return` function
- The `Monad` type class





So far

- We've seen that "monads" are Haskell's way of representing functions with computational effects that go beyond normal "pure" functions
- We've talked about monadic functions and monadic values
- We've seen that monadic values are not particularly intuitive, but...
- ...they can be interpreted as computational "actions"





Question

- Assume we have two monadic functions with these type signatures:

$f :: a \rightarrow m\ b$

$g :: b \rightarrow m\ c$

- Or, more specifically:

$f :: a \rightarrow IO\ b$

$g :: b \rightarrow IO\ c$

- How do we compose these?





Question

- The result of composing

$f :: a \rightarrow m\ b$

$g :: b \rightarrow m\ c$

- should be $h :: a \rightarrow m\ c$

- Similarly, the result of composing

$f :: a \rightarrow IO\ b$

$g :: b \rightarrow IO\ c$

- should be $h :: a \rightarrow IO\ c$





Interpretation

$f :: a \rightarrow IO\ b$

$g :: b \rightarrow IO\ c$

- g composed with f should be

$h :: a \rightarrow IO\ c$

- This represents a function that takes an input value of type a , does some I/O (the thing f does followed by the thing g does) and then returns a value of type c
- In other words, the "effects" of f and g are combined in h





Interpretation

$f :: a \rightarrow IO\ b$

$g :: b \rightarrow IO\ c$

- Let's try to do this with regular Haskell function composition (the `.` operator)
- Either $g \cdot f$ or $f \gg g$ fails to type check!
- Reason?
- g expects value of type b , gets $IO\ b$ instead
- What can we do about this?





mcompose

f :: a -> IO b

g :: b -> IO c

- Imagine that we had a function called **mcompose** (for "monadic composition operator") with this type signature:

mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)

- Then we could compose our functions **f** and **g** like this:

h = mcompose f g

-- or: h = f `mcompose` g





`mcompose`

`mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)`

- `mcompose` will work on any monad `m`
- Specialized to the `IO` monad, the type signature will look like this:

`mcompose :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)`

- The definition of `mcompose` should not depend on `IO` or any particular monad
- How would `mcompose` work?





mcompose

$\text{mcompose} :: (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c)$

- $f \text{ `mcompose` } g$ will do the following when applied to a value x of type a :
 1. apply f to x , returning a (monadic) value of type $m\ b$ ($\text{IO } b$ if we know m is IO)
 2. somehow extract a value of type b from a value of type $m\ b$ ($\text{IO } b$)
 3. pass that value to the function g , returning a final value of type $m\ c$ ($\text{IO } c$)
- Note: steps 1 and 3 are just normal function application! (Hard part is step 2)





extract

`mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)`

- Assume we had a function called **extract** which extracted a regular (non-monadic) value from a monadic value:

`extract :: m b -> b` *-- most general form*

`extract :: IO b -> b` *-- specialized to IO*

- Then we could easily define **mcompose** as follows:

`mcompose f g x = g (extract (f x))` *-- or:*

`mcompose f g = g . extract . f` *-- or:*

`mcompose f g = f >.> extract >.> g`





extract

`extract :: m b -> b` *-- most general form*

`extract :: IO b -> b` *-- specialized to IO*

- However, we're not allowed to do this!
- Having `extract` destroys most of the advantages of monads!
- Specifically: it allows us to write functions which look like they don't have extra computational effects, when in fact they do
- Example: function that does I/O without having an `IO` in the type signature





extract

`extract :: IO b -> b` -- specialized to IO

- Consider a pure function `hh` with type signature

`hh :: a -> c`

- We know from looking at the type signature that this function does not do I/O
 - if it did, type signature would have to be `a -> IO c`
- Guarantees like this are a major strength of Haskell's type system





extract

`extract :: IO b -> b` -- specialized to IO

- What if there was a function like `extract`?
- Then `hh` could be composed from two functions:

`ff :: a -> IO b` -- does I/O

`gg :: b -> c` -- doesn't do I/O

- by using `extract`:

`hh = gg . extract . ff`

-- or: `ff >.> extract >.> gg`

- So even though `hh`'s type signature indicates that it's pure (doesn't do I/O), it isn't because of `extract`!





extract

- To enforce a clear separation between monadic and non-monadic functions, we shouldn't have **extract**
 - or if we do have it, we shouldn't use it!
- In fact, some monads do define **extract**-like functions
- But we can't have a *generic* extract that works for *any* monad, or purity guarantees of Haskell go out the window





extract

- One monad that should *not* have an **extract**-like function is the **IO** monad, since we don't want to mix up functions that do I/O with those that don't
- Unfortunately, there *is* an **extract**-like function in the **IO** monad, called **unsafePerformIO**:

unsafePerformIO :: IO a -> a

- This function should only be used by experts, as it can have undesired (non-functional) effects
- We won't need it or use it anywhere in this course!





So far...

- We want to be able to compose monadic functions
- We can't use normal function composition
- We want to define **mcompose**
- We can't define it in terms of **extract**, because we aren't allowed to have a generic **extract** that works for all monads
- So what do we do?





From `mcompose` to `mapply`

- One easy simplification: define function composition in terms of function application
- Example: pure functions
- Function composition has the type signatures:
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 $(>.>) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$
- Function application has the type signatures:
 $(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$
 $(>\$>) :: a \rightarrow (a \rightarrow b) \rightarrow b \quad \text{-- flip } (\$)$
- We will use `>.>` and `>\$>` instead of `.` and `\$`





- $$(>.)> \quad :: \quad (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$$

- $$(>\$>) :: a \rightarrow (a \rightarrow b) \rightarrow b$$

- (>.>) f g x = f x >\$> g -- g (f x)

- f :: a -> b, x :: a, f x :: b, g :: b -> c**

$$f \times g \leq c$$




From `mcompose` to `mapply`

- Similarly, we can define `mcompose`:

```
mcompose :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

- In terms of monadic function application (`mapply`):

```
mapply :: m a -> (a -> m b) -> m b
```

- Here's how:

```
mcompose f g = \x -> f x `mapply` g
```

- Types:

```
f :: a -> m b, x :: a, f x :: m b, g :: b -> m c
```

```
f x `mapply` g :: m c
```





From `mcompose` to `mapply`

- Technical point: we can describe the type signature of `mapply` as:

`mapply :: m a -> (a -> m b) -> m b`

- Or as:

`mapply :: m b -> (b -> m c) -> m c`

- As we wish (just using different names for type variables)
- I will often do this kind of substitution without explicitly saying so





From `mcompose` to `mapply`

- What have we accomplished?
- *Good news*: we don't have to define `mcompose` as a primary function
 - can define it in terms of `mapply`
- *Bad news*: how do we define `mapply`?





`mapply` in Haskell

- `mapply` is one of the two fundamental monadic operations in Haskell
- It is one of the methods of the `Monad` type class (a constructor class), and it's written as the operator `>>=`
- This operator is often referred to as the "bind" operator
- The name `mapply` won't be used anymore (just used to get us this far)





mcompose in Haskell

- **mcompose** is implemented in Haskell as a library function in the module **Control.Monad**
- It is referred to as the **>=>** operator, not as **mcompose**
- It is defined in terms of the **>>=** (bind) operator in the same way we defined **mcompose** in terms of **mapply** previously:

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





mcompose in Haskell

- We can also define reverse monadic composition/ apply operators as follows:

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(<=<) = flip (>=>)
```

```
(=<<) :: (a -> m b) -> m a -> m b
(=<<) = flip (>>=)
```

- Both are defined in the **Control.Monad** module





In practice

- We tend to use the `>>=` operator much more than the other operators
- There is also some very nice syntactic sugar that makes it possible to avoid using even `>>=` in monadic code (which we'll see later)





New problem

- So far, we've been looking at how to compose two monadic functions to get a monadic function
- Another thing we might want to do is to compose a monadic function with a non-monadic function
- What would the result of such a composition have to be?
 - a monadic function, or
 - a non-monadic function?





New problem

- In particular, consider composing **f** and **g**, where

f :: **a** -> **m b**

g :: **b** -> **c**

- Result of composition would have to have the type...?

a -> **m c**

- Result can't be of type **a** -> **c** because we don't have an **extract** function





New problem

$f :: a \rightarrow m\ b$

$g :: b \rightarrow c$

- We can't compose these using regular function composition
 - g 's input type isn't $m\ b$
- We can't compose these using monadic function composition either
 - g 's type isn't $b \rightarrow m\ c$
- So what do we do?





New problem

$f :: a \rightarrow m\ b$

$g :: b \rightarrow c$

- If we could convert g to a function gg whose type was $b \rightarrow m\ c$, then could use monadic function composition:

$f \gg= gg$

- So we want to convert

$g :: b \rightarrow c$

- To:

$gg :: b \rightarrow m\ c$





return

$g :: b \rightarrow c$

$gg :: b \rightarrow m\ c$

- Notice that we could easily convert g to gg if we could convert a value of type c to a value of type $m\ c$
- This is the second fundamental monadic operation (function), called **return**
- Note: this has *nothing* to do with returning from a function call!





return

$g :: b \rightarrow c$

$gg :: b \rightarrow m\ c$

$return :: c \rightarrow m\ c$

$gg = return \ . \ g$

- **return** makes it trivial to define **gg**
- Then we can (monadically) compose **f** with **g**:

$f :: a \rightarrow m\ b$

$g :: b \rightarrow c$

$f \gg= (return \ . \ g) \quad \text{-- type: } a \rightarrow m\ c$





return

return :: a -> m a

- What **return** does is to take a normal value and "lift" it into a monadic value
- **return** is a monadic function
- The name "return" comes from thinking of monadic values as "actions"
- **return** takes a value and creates an "action" which can do arbitrary effects and "returns" the value that was given to **return** in the first place
 - in fact, it won't do any of those effects, but it could





return

return :: a -> m a

- In fact, **return** is the monadic version of the identity function!
- We'll see this again when we cover monad laws





Core monadic operations

- So far, we've identified two core monadic operations:
 - the `>>=` operator (monadic apply operator)
 - the `return` function (monadic identity function)
- In fact, these are the *only* core monadic operations
- (There are two non-core monadic operations we'll meet later)





Core monadic operations

- The precise definitions of `>>=` and `return` are going to depend on which monad it is
 - definition for `IO` monad is going to be different from exception-handling monads, or state-handling monads, etc.
- However, the type signatures will have the same form
- In Haskell, we deal with this by using type classes
 - `Monad` is actually a constructor class in Haskell
- Let's look at the definition (core operations only)





The Monad type class

- Here is the definition of the **Monad** type class:

```
class Monad m where
```

```
    return :: a -> m a
```

```
    (>>=) :: m a -> (a -> m b) -> m b
```

```
    -- two non-core methods omitted
```

- This just consolidates what we have been describing up to this point
- We will describe specific instances of this type class in later lectures





The Monad type class

- Consider again the `>>=` operator:
$$(>>=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$
- What is this actually doing (conceptually)?
- We know we can't define an `extract` function that works for all monads
- The definition of `>>=` for a particular monad does the work of `extract` "locally" *for that monad only*
- It "unpacks" a value of type `a` from a value of type `m a`, passes that value to the function with type `a -> m b`, giving the result (of type `m b`)





The Monad type class

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

- This "unpacking" is completely monad-specific: each monad does it a different way
- We will see many different examples of how this unpacking is done in different monads





>>= vs extract

- You might wonder why we don't just have **extract** as a method in the **Monad** type class instead of **>>=**
- If we had **extract**, we could use it to define **>>=** as follows:

```
mx >>= f = f (extract mx) -- for monadic value mx
```

- However, this would allow us to extract a non-monadic value from a monadic value whenever we wanted to (not desirable, as previously shown)
- With **>>=**, we can only extract a value if we intend to immediately pass it to a monadic function
 - The (non-monadic) value cannot "escape" the monad





Next time

- Practical interlude: the **IO** monad in practice
- Monadic syntactic sugar: the **do** notation
- Writing and compiling stand-alone programs
- **ghci** and the **IO** monad

