CS 115
Functional Programming

Lecture 14:

The Monad Laws

Today

- The monad laws
- The `Maybe` monad
- Deriving the `Maybe` monad

Recap: The **Monad** type class

- The **Monad** type class is defined as:

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

- **Monad** is a constructor class, since **Monad** instances are type constructors (**m**)

Recap: The `Monad` type class

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

- The two fundamental `Monad` operations are `return` and `>>=`

Recap: The **Monad** type class

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

- **`>>=`** is monadic application: a monadic function (type **`a -> m b`**) is applied to a monadic value (type **`m a`**) to get a monadic value (type **`m b`**)
- **`return`** "lifts" a regular value into a monadic value
  - *i.e.* a computation "returning" that value

Recap: The Monad type class

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

- `>>` is monadic sequencing: two monadic values ("actions") are "run" one after the other in sequence
- The first monadic action normally has the type `m ()`
- The return value of `>>` is the return value of the second monadic action

Recap: The `Monad` type class

- As far as Haskell is concerned, any type constructor that implements the `Monad` methods is a valid instance of the `Monad` type class
- But for a type constructor to truly "be" a monad, more is required!

The three laws of monadics

- Many interesting natural laws come in groups of three:
  - Newton's three laws of motion
  - The three laws of thermodynamics
  - Kepler's three laws of planetary motion
  - Asimov's three laws of robotics
- Monads also have three associated laws
- Of course, the "three laws of monadics" are far more important than any of those other laws ☺

The three laws of monadics

- Recall the whole point of monads:
  - to take computations with extra effects
  - and to be able to compose them as naturally as we can compose regular functions
- It's worth looking at normal function composition to see what laws *it* obeys, then see if there are any monadic versions of those laws that monadic function composition must also obey

Function composition

- Function composition is written in Haskell using the `(.)` operator and is defined as:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

- Or if you prefer, use the `(>.>)` operator and write the arguments in a different order:

```
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
f >.> g = \x -> g (f x)
-- or: (>.>) = flip (.)
```

Identity laws

- There is an identity function `id` that takes a value and returns it unchanged, defined as:

```
id :: a -> a
id x = x
```

- What is the relationship between `id` and function composition?
- Composing an arbitrary function `f` with `id` should give...?
  - the original function `f` back!

Identity laws

- Specifically, we can define two "laws" that function composition with `id` must obey:

```
id . f = f
f . id = f
```

- In algebra, we say that `id` is a "left identity" of function composition (law 1) and a "right identity" of function composition (law 2)
- Any notion of function composition coupled with some kind of identity function should obey laws like these in order to behave in a "reasonable" way

Associativity law

- Function composition also has to be *associative*
- Consider three functions `f`, `g`, and `h`
- This must be true:

```
(f . g) . h = f . (g . h)
```

- In words: there is only one way to compose three functions `f`, `g`, and `h` together
- Which of the functions gets composed first doesn't matter; the end result is the same
- Again: this must be true for any "reasonable" notion of function composition

Laws and more laws

- Since any "reasonable" notion of function composition has to uphold three laws:
  - left identity with the identity function
  - right identity with the identity function
  - associativity
- ... we should expect that if monadic function composition is "reasonable", it should uphold three laws like this too
- In fact, this is the case

Monadic function composition

- Recall the monadic function composition operator:

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

- Defined in the `Control.Monad` module
- This is analogous to the `(>.>)` operator we defined for normal functions
- There is also another form with the arguments reversed:

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

- Also defined in `Control.Monad`

Monadic function composition

- Recall that monadic function composition can be defined in terms of monadic application:

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

- The reversed form can be defined even more simply:

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

Monadic identity function

- The `return` function has this type signature:

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

- Viewed as a function-with-effects, you could write `return`'s type signature schematically as:

```
                 [m]
return :: a ---------> a
```

- where `[m]` represents the effects embodied in a particular monad
- Thus, `return` seems to be kind of like an identity function for monads

Monad laws: the nice version

- If we consider `return` to be a monadic identity function and use the monadic composition operator `>=>`, the three monad laws have this form:

```
1)  return >=> f      ==   f
2)  f >=> return      ==   f
3)  (f >=> g) >=> h   ==   f >=> (g >=> h)
```

- These are *identical* in form to the corresponding laws for function composition, except instead of using `(.)` and `id` we use `(>=>)` and `return`!

Monad laws: the nice version

- In words, the monad laws state that:

1. monadic function composition is associative
2. `return` is a left- and right-identity for monadic function composition

- These laws, if they hold, guarantee that monadic function composition "behaves normally" *i.e.* it behaves in the way you expect function composition to behave, modulo the monadic effects

Monad laws: the ugly version

- Most Haskell literature presents the monad laws not in terms of the **>=>** operator but in terms of the **>>=** operator
- This gives rise to much less intuitive monad laws, but the translation between them is straightforward (though a bit grungy)
- Here, we simply present the ugly form and leave the derivations as an exercise

Monad laws: the ugly version

- The ugly version of the monad laws:

```
1.  return x >>= f  ==  f x
2.  mx >>= return  ==  mx   -- mx is a monadic value
3.  (mx >>= f) >>= g  ==  mx >>= (\x -> (f x >>= g))
```

- Advantages of the ugly version:
  1. Can use to simplify code: when you see patterns like `(return x >>= f)`, replace with just `(f x)`
  2. Can use to constrain definitions of `return` and `>>=` when defining new monads (very useful!)

Enforcing monad laws

- The problem with monad laws:
  - Haskell cannot enforce them!
  - (Similar to case of laws for `Functor` type class)
- Haskell is not powerful enough to use to prove theorems about whether particular instances of the type class `Monad` have definitions of `>>=` and `return` which obey the monad laws
- Haskell will even accept versions which do not obey these laws, as long as their types are correct!

Enforcing monad laws

- Therefore, it's up to the programmer who writes the **`Monad`** instance definition for a particular monad to make sure that the definitions of **`>>=`** and **`return`** obey the monad laws
- Usually, the definition of **`>>=`** follows directly from what the monad is trying to achieve
- The definition of **`return`** for a monad is often much less obvious
- We can use the monad laws to tell us what the "right" definition of **`return`** has to be

Enforcing monad laws

- We also must check that the "natural" definition of the `>>=` operator for a given monad obeys the monad laws in conjunction with the definition of `return`

Example: the `Maybe` monad

- We have already seen the **`Maybe`** type constructor:

```
data Maybe a =
    Nothing
  | Just a
```

- A **`Maybe`** type can be used as the return value of a function when that function may or may not be able to generate a value of that type
- Such functions have the general type:

```
a -> Maybe b
```

Example: the `Maybe` monad

- We have said that the purpose of monads is to represent "notions of computation" that are different from the standard notion of computation (pure functions)
- One such "notion of computation" is *"a computation that may fail"*
- Such a computation will naturally have the type `a -> Maybe b`
- Therefore, it's not unreasonable to expect that `Maybe` might be a monad

Aside: the `Maybe` monad

- There is *nothing* "non-functional" about a function with a type of `a -> Maybe b`
- You can think of this as either
  - a pure function whose return type happens to be `Maybe b`
  - a function from type `a` to type `b` which can fail
- The second interpretation gives rise to the monad
- *Both* interpretations are purely functional

Example: the `Maybe` monad

- The purpose of the **`Maybe`** monad is to enable us to easily compose functions that may fail
- We'll use a trivial contrived example:

```
f :: Integer -> Maybe Integer
f x = if x `mod` 2 == 0 then Nothing else Just (2 * x)

g :: Integer -> Maybe Integer
g x = if x `mod` 3 == 0 then Nothing else Just (3 * x)

h :: Integer -> Maybe Integer
h x = if x `mod` 5 == 0 then Nothing else Just (5 * x)
```

Example: the `Maybe` monad

- We would like to compose `f`, `g`, and `h` to get a final function `k`
- `k` will take an `Integer`, and will multiply it by `2`, then `3`, then `5` (total `30`) unless it's divisible by `2` or `3` or `5`, in which case it will return `Nothing`
- We can't use normal function composition to define `k` in terms of `f`, `g`, and `h`, because the output types of these functions (`Maybe Integer`) aren't the same as their input types (`Integer`)

Example: the `Maybe` monad

- Defining **`k`** in terms of **`f`**, **`g`**, and **`h`** is nevertheless straightforward in Haskell:

```
k :: Integer -> Maybe Integer
k x = case f x of
          Nothing -> Nothing
          Just y ->
            case g y of
              Nothing -> Nothing
              Just z -> h z
```

Example: the `Maybe` monad

- Problem: the code is repetitive and grungy
  - The `Nothing -> Nothing` line is repeated twice!
- If more functions were to be composed, the nesting would get even deeper
- We will use monads to clean this code up
- Unlike the case with the `IO` monad, monads are not "essential" in order to write code using the `Maybe` type constructor
- However, monads make working with `Maybe` much more convenient

`Maybe`: definition of **`>>=`**

- Let's start by defining the **`>>=`** (monadic application) operator for the **`Maybe`** type constructor
- It will have the (specialized) type signature:

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

- Usually, the definition of **`>>=`** can be obtained by understanding what monadic application is trying to achieve
- That will be the case here
  - (Still have to check it using the monad laws!)

`Maybe`: definition of `>>=`

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = ???
Just x  >>= f = ???
```

- Need to fill in the `???` parts

`Maybe`: definition of `>>=`

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = ???
Just x  >>= f = ???
```

- If the previous computation failed (returned `Nothing`), what would that computation composed with `f` do?

`Maybe`: definition of **`>>=`**

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = ???
Just x  >>= f = ???
```

- If the previous computation failed (returned **`Nothing`**), what would that computation composed with **`f`** do?
  - Fail!
  - *i.e.* it would also return **`Nothing`**
  - *i.e.* first equation is **`Nothing >>= f = Nothing`**

`Maybe`: definition of `>>=`

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = Nothing
Just x  >>= f = ???
```

- If the previous computation returned **`Just x`**, how would we "unpack" a value of type **`a`** to pass to **`f`**?

`Maybe`: definition of `>>=`

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = Nothing
Just x  >>= f = ???
```

- If the previous computation returned `Just x`, how would we "unpack" a value of type `a` to pass to `f`?
  - *Just* use `x`!
  - Second equation is `Just x >>= f = f x`

`Maybe`: definition of `>>=`

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = Nothing
Just x  >>= f = f x
```

- This is the "plausible" definition of `>>=` for the `Maybe` monad
- We still have to verify it using the monad laws!
- Before that, though, we have to define `return`:

```
return :: a -> Maybe a
return x = ???
```

Maybe: definition of return

```
return :: a -> Maybe a
return x = ???
```

- Since the definition has to work for *any* type `a`, there aren't many choices
- The two "obvious" choices are:

1. `return x = Nothing`
2. `return x = Just x`

`Maybe`: definition of `return`

- It should seem plausible that

```
return x = Nothing
```

- is not the best candidate
- This does not look like an identity function!
- Let's demonstrate that this won't work, using the previous definition of `>>=` and the monad laws

`Maybe`: definition of `return`

- Given

```
return x = Nothing
```

- let's check monad law 1 (ugly form):

```
return x >>= f  ==  f x
```

- Doing some substitutions:

```
return x >>= f
= Nothing >>= f
= Nothing   -- definition of >>=
```

- which cannot in general be equal to `f x` for all `f`s and `x`s

`Maybe`: definition of `return`

- So this definition

```
return x = Nothing
```

- violates monad law 1, so the correct definition is:

```
return x = Just x
```

- We still have to check this and the definition of `>>=` against the monad laws!

`Maybe`: monad law 1

- Monad law 1 (ugly form):

```
return x >>= f  ==  f x
```

- With our definition, we have:

```
return x >>= f
= Just x >>= f
= f x  -- definition of >>=
```

- So monad law 1 holds

`Maybe`: monad law 2

- Monad law 2 (ugly form):

```
mx >>= return  ==  mx
```

- `mx` can be either `Nothing` or `Just x`
- If `mx` is `Nothing`, we have:

```
Nothing >>= return
= Nothing -- definition of >>=
= mx
```

Maybe: monad law 2

- Monad law 2 (ugly form):

```
mx >>= return  == mx
```

- `mx` can be either `Nothing` or `Just x`
- If `mx` is `Just x`, we have:

```
Just x >>= return
= return x -- definition of >>=
= Just x -- definition of return
= mx
```

- So monad law 2 holds

`Maybe`: monad law 3

- Monad law 3 (ugly form):

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

- Case 1: `mx` is `Nothing`

```
(Nothing >>= f) >>= g   -- LHS
= Nothing >>= g   -- definition of >>=
= Nothing         -- definition of >>=
Nothing >>= (\x -> (f x >>= g))   -- RHS
= Nothing         -- definition of >>=
```

- OK, so case 1 checks out

`Maybe`: monad law 3

- Monad law 3 (ugly form):

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

- Case 2: `mx` is `Just v`

```
(Just v >>= f) >>= g  -- LHS
= f v >>= g  -- definition of >>=
Just v >>= (\x -> (f x >>= g)) -- RHS
= (\x -> (f x >>= g)) v  -- definition of >>=
= f v >>= g  -- function application
```

- Case 2 checks out, so monad law 3 holds

Maybe: Final form

- **Maybe** instance of **Monad** type class:

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

- We proved that this instance definition is consistent with the monad laws
- So: **Maybe** is in fact a monad!
- Monadic composition with **Maybe** monadic functions "behaves in a sensible fashion"

`Maybe`: Final form

- `Maybe` instance of `Monad` type class:

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

- `>>` definition is equivalent to the default
- The `Maybe` monad is also an instance of the `MonadFail` type class, with this definition:

```
  fail _ = Nothing
```

`Maybe`: Final form

- Let's return to our example with `f`, `g`, `h`, and `k`
- We can now define `k` monadically as:

```
k :: Integer -> Maybe Integer
k = f >=> g >=> h
```

- This is *much* simpler than the explicit definition with nested `case` statements!
- Monads have allowed us to remove all the "boilerplate" code dealing with `Nothing` values and focus on the overall structure of the computation

The point

- Aside from special cases like the `IO` monad, where monads are the *only* way to achieve the result we want...
- ...the main use of monads in Haskell is to allow us to write computations *without a lot of trivial boilerplate code* that only serves to obscure what's really going on
- We will see many examples of this in the upcoming lectures

The point

- Monad laws may seem very "theoretical", but they guide us when we are implementing new monads
  - A `Monad` instance that obeys the monad laws will behave "reasonably" with respect to monadic function composition
  - We can use the monad laws to give us a "reasonable" definition of the `return` function given a plausible definition of the `>>=` operator

The point

- From now on, we will use the monad laws with each new monad we discuss
  - to derive the correct `return` definition
  - to verify the "plausible" `>>=` definition

Coming up next

- The list monad