Outline of Lecture 14

- The `Applicative` type class
- Examples of applicative functors
- The `Monad` type class
- Examples of monads

Monoids and Functors (reminder)

- In the previous lecture, we have seen two common algebras:
    - `Monoid` gives us a means of combining (folding) two values of the same type together;
    - `Functor`, on the other hand, is for function application (mapping) over some structure we do not want to have to think about
- `Monoid`'s core operation, `mappend`, smashes two structures together (e.g., two lists become one)
- The core operation of `Functor`, `fmap`, applies a function to a value that is within some structure while leaving that structure unaltered

Haskell type classes: Applicative

- Applicatives are monoidal functors. What does it mean?
- The `Applicative` type class allows for function application lifted over structure (like `Functor`)
- However, with `Applicative`, the function we are applying and the value being applied are both embedded in some structure
- As a result, we have to apply the function and merge these structures together
- So, `Applicative` involves both monoids and functors

Haskell type classes: Applicative (cont.)

- The definition:

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

- Every type that can have an `Applicative` instance must also have a `Functor` instance
- The `pure` function does a very simple thing: it lifts something into functorial structure (minimal context)
- The infix operator `(<*>)` (called apply) combines two structural values (one of them containing an embedded function) into one

Examples of `pure`

- Embedding a value of any type in the structure we are working with:

```
Prelude> pure 1 ::  [Int]
[1]
Prelude> pure 1 ::  Maybe Int
Just 1
Prelude> pure 1 ::  Either a Int
Right 1
Prelude> pure 1 ::  ([a], Int)
([],1)
```

- The left type is handled differently from the right in the final two examples since the left type is part of the structure, and the structure is not transformed by the function application.
  For the same reason as `fmap (+1) (4,5) == (4,6)`

Applicative functors are monoidal functors

- Let us check the types:

```
($) ::   (a -> b) -> a -> b
(<$>) ::  (a -> b) -> f a -> f b
(<*>) ::  f (a -> b) -> f a -> f b
```

  where `(<$>) = fmap` (i.e., infix operator alias for `fmap`)

- While `(<$>)` looks like a generalisation of `($)` over a given structure, `(<*>)` additionally "wraps" a given function into the structure `f`

- Moreover, one of Applicative laws requires:

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

- "Wrapping" of a given function (`pure f`) looks like additional (redundant?) step. What are advantages of this?

Applicative functors are monoidal functors (cont.)

- When we were dealing with fmap, we had only one bit of structure, so it was left unchanged
- With `(<*>)`, we have two bits of structure of type `f` that we need to deal with before returning a value of type `f b`. We cannot simply leave them unchanged; we must unite them somehow
- A typical solution – rely on `Monoid` for our structure and function application for our values!
- What are we gaining from that?

Examples of (<*>)

- Applying (<*>) between lists: the values of [a -> b] and [a]:

```
Prelude> [(*3)] <*> [4, 5]
[12,15]
Prelude> [(*2), (*3)] <*> [4, 5]
[8,10,12,15]
Prelude> [] <*> [Just 2]
[]
Prelude> [(*2), (*3)] <*> []
[]
```

- What are we gaining from embedding (a-> b) into [a -> b]? List-ness
- The first case could have been encoded as simply fmap (*3) [4,5] (no need for Applicative). In the other cases, we have now a list of functions that give us more expressive power

Examples of `(<*>)` `(cont.)`

- Applying `(<*>)` between the values of `Maybe (a -> b)` and `Maybe a`:

```
Prelude> Just (*2) <*> Just 2
Just 4
Prelude> Just (*2) <*> Nothing
Nothing
Prelude> Nothing <*> Just 2
Nothing
```

- What are we gaining from embedding `(a-> b)` into `Maybe (a -> b)`? Maybe-ness
- We can now express how uncertainty of different `Maybe` values can be combined

Examples of `(<*>)` (cont.)

- Applying `(<*>)` between pair values of `(a, b -> c)` and `(a, b)`:

```
Prelude> ("Woo", (+1)) <*> (" Hoo!", 0)
("Woo Hoo!", 1)
Prelude> (Sum 2, (+1)) <*> (Sum 0, 0)
(Sum {getSum = 2},1)
(Product 3, (+9)) <*> (Product 2, 8)
(Product {getProduct = 6},17)
```

- The same principle: function application between the right pair elements, monoidal folding (mappending) between the remaining structure (the left pair elements)
- The reason: `(,) a` is an instance of `Applicative`

Applicative instances

- The Applicative instance for lists:

```
instance Applicative [] where
  pure a = [a]
  [] <*> _ = []
  (f:fs) <*> xs =
    (fmap f xs) `mappend` (fs <*> xs)
```

- For lists, `mappend = (++)`
- One of several ways to combine such list values (one alternative – `ZipList`)

Applicative instances (cont.)

- The Applicative instance for Maybe:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  Just f <*> Just a = Just (f a)
```

- Nothing values propagate through the structure!

`Applicative` instances (cont.)

- The `Applicative` instance for `(,) a`:

```
instance Monoid a => Applicative ((,) a) where
  pure x = (mempty, x)
  Nothing <*> _ = Nothing
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

- Explicit dependency (constraint) `Monoid a`
- Monoidal merging and function application on different pair elements!

One more example: lookups

- The lookup function searches inside a list of tuples for a value that matches the input and returns the paired value wrapped inside a Maybe context:

```
:t lookup
lookup ::  Eq a => a -> [(a, b)] -> Maybe b
```

- Example:

```
import Control.Applicative


f x = lookup x [ (3, "hello"), (4, "julie"), (5, "kbai")]
g y = lookup y [ (7, "sup?"), (8, "chris"), (9, "aloha")]
h z = lookup z [(2, 3), (5, 6), (7, 8)]
m x = lookup x [(4, 10), (8, 13), (1, 9001)]
```

One more example: lookups (cont.)

- Combining Maybe values:

```
Prelude> f 3
Just "hello"
Prelude> g 8
Just "chris"
Prelude> (++) <$> f 3 <*> g 7
Just "hellosup?"
Prelude> (+) <$> h 5 <*> m 1
Just 9007
Prelude> (+) <$> h 5 <*> m 6
Nothing
```

- `f <$> (x :: Maybe a)` embedds a function into the Maybe structure by partial application

Haskell type classes: Monads

- A functor maps a function over some structure; an applicative maps a function that is contained in some structure over some other structure and then combines the two layers of structure like `mappend`
- Monads are applicative functors, but they have something special about them that makes them different from and more powerful than either `<*>` or `fmap` alone
- Definition:

```
class Applicative m => Monad m where
  (>>=) ::  m a -> (a -> m b) -> m b
  (>>) ::  m a -> m b -> m b
  return ::  a -> m a
  x >> y = x >>= (\_ -> y)
```

Haskell type classes: Monads (cont.)

- Dependencies between type classes:

  ```
  Functor => Applicative => Monad
  ```

  Whenever you have implemented an instance of Monad for a type you necessarily have an Applicative and a Functor as well

- `return` is just the same as `pure` of *Applicative*. Essentially, a simple constructor for a monadic value

- The operator `(>>)` is called the sequencing operator. It is a restricted version of the main `Monad` function `(>>=)`, called *bind*

Haskell type classes: Monads (cont.)

- Let us check the types:

```
fmap ::  Functor f => (a -> b) -> f a -> f b
(<*>) ::  Applicative => f (a -> b) -> f a -> f b
(>>=) ::  Monad => f a -> (a -> f b) -> f b
```

- Bind looks quite similar to both `fmap` and `(<*>)`, but with the first two arguments flipped. Yet another version of mapping a function over a value while bypassing its surrounding structure
- Can we express `(>>=)` via `fmap` by simply instantiating `b` to `f b`?

Haskell type classes: Monads (cont.)

- Example:

```
Prelude> andOne x = [x, 1]
Prelude> andOne 10
[10,1]
Prelude> :t fmap andOne [4, 5, 6]
fmap andOne [4, 5, 6] :: Num t => [[t]]
Prelude> fmap andOne [4, 5, 6]
[[4,1],[5,1],[6,1]]
```

- We ended up with an extra layer of structure, and now we have a result of nested lists
- Our mapped function has itself generated more structure! How to discard one unnecesary layer of that structure?

Haskell type classes: Monads (cont.)

- We know how to do it with lists, using the function `concat`:

```
concat ::  [[a]] -> [a]
```

- `Monad`, in a sense, is a generalisation of `concat`! The unique part of `Monad` is the following function:

```
import Control.Monad (join)

join ::  Monad m => m (m a) -> m a
```

- The ability to flatten those two layers of structure into one is what makes `Monad` special

- Using `join`, `bind` can be simply defined as
  `(>>=) f x = join (fmap f x)`

Haskell type classes: Monads (cont.)

- General intuition: a monad allows to sequence operations on structure, while feeding the result of one action as the input value to the next
- Each operation may return a structured value, which is internally flattened and combined with others
- A structured value may indicate non-determinism (e.g., several possible results as a list of values), uncertainty (a Maybe value), ...
- Monad allows to sequence such operations and their structured results in a convenient way

Monads and the do syntax

- We used the do notation while discussing of sequencing IO actions in Haskell. The IO datatype is an instance of *Monad*
- Actually, the do notation is defined and works with any monad
- The do syntax is just a syntactic sugaring and can be directly expressed using monad operations and the lambda notation

Monads and the do syntax (cont.)

- The next three pieces of Haskell code are equivalent:

```haskell
import Control.Applicative ((*>))

sequencing ::  IO ()
sequencing = do
  putStrLn "blah"
  putStrLn "another thing"

sequencing' ::  IO ()
sequencing' =
  putStrLn "blah" >> putStrLn "another thing"

sequencing'' ::  IO ()
sequencing'' =
  putStrLn "blah" *> putStrLn "another thing"
```

Monads and the do syntax (cont.)

- Sequencing with passing values (two equivalent definitions):

```
bindingAndSequencing ::  IO ()
bindingAndSequencing = do
  putStrLn "name pls:"
  name <- getLine
  putStrLn ("Why hello there:  " ++ name)

bindingAndSequencing' ::  IO ()
bindingAndSequencing' =
  putStrLn "name pls:" >>
  getLine >>=
  \name ->
    putStrLn ("Why hello there:  " ++ name)
```

With each value binding, the nesting intensifies, making the do notation much cleaner and easier to read

Monad examples

- `Maybe` is a monad:

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

- Playing with `Maybe` as a monad:

```
ghci> return "Hurra!" ::  Maybe String
Just "Hurra!"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
```

Monad examples (cont.)

- Sequencing (gluing together) monadic values, without the do syntax:

```
ghci> Nothing >>= (\x -> Just "!" >>=
    (\y -> Just (show x ++ y)))
Nothing

ghci> Just 3 >>= (\x -> Just "!" >>=
    (\y -> return (show x ++ y)))
Just "3!"

ghci> Just 3 >>= (\x -> Just "!" >>=
    (\y -> Nothing))
Nothing
```

- Failures (`Nothing` values) are propagated through the sequence

Monad examples (cont.)

- Advantage: the bound values (intermediate results) accumulated, relying on the lambda notation, and can be used later
- Advantage: failures are automatically propagated by using >>=
- (Big) disadvantage: very annoying and error-prone lambdas
- With the do notation

```
Just 3 >>= (\x -> Just "!" >>=
    (\y -> return (show x ++ y)))
Just "3!"
```

then becomes simply

```
do
  x <- Just 3
  y <- Just "!"
  return (show x ++ y)
```

IO as a monad

- IO type constructor is an instance of Monad in Haskell
- Example: the program

```
addOneInt :: IO ()
addOneInt = do
  line <- getLine
  putStrLn (show (1 + read line :: (Int))
```

is actually the "syntactic sugar" for

```
addOneInt =
  getLine >>= \line ->
    putStrLn (show (1 + read line :: (Int))
```

The list monad

- Lists can be also treated as monads
- Definition of the List instance:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

- How possibly multiple list values are handled in a sequence of monad operations?
- We can treat multiple list values as a form of non-determinism with several possible results/outcomes

The list monad (cont.)

- When binding, each list element is treated separately as a possible result to be handled
- Example:

```
lm = do
  x <- [3,4,5]
  y <- [x,-x]
  return y

ghci> lm
[3,-3,4,-4,5,-5]
```

Gives us all possible results

The list monad (cont.)

- Another example:

```
twiceWhenEven ::  [Integer] -> [Integer]
twiceWhenEven xs = do
  x <- xs
  if even x
    then [x*x, x*x]
    else [x*x]

Prelude> twiceWhenEven [1..3]
[1,4,4,9]
```

The list monad (cont.)

- Just as operation sequencing with the `Maybe` monad allows us to propagate and handle failures, chaining several monadic lists with `>>=` propagates non-determinism, e.g.,

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n, ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

- Or, using the `do` notation:

```
listOfTuples ::  [(Int,Char)]
listOfTuples = do
  n <- [1,2]
  ch <- ['a','b']
  return (n, ch)
```

The do notation and list comprehensions

- The latter piece of Haskell code looks suspiciously familiar to list comprehension

```
ghci> [(n, ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

  Even the results are the same :)

- In fact, list comprehensions are just syntactic sugar for using lists as monads $\Rightarrow$ they translate into a sequence of >>= computations that involve non-determinism