Day 2

Functions and functors

Algebras

```haskell
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Algebras

An *instance* is the piece of code that ties a typeclass declaration to a particular type.

A typeclass declaration is maximally polymorphic, to be flexible enough to work with many types.

An instance gives the implementation for a given type.

Product

Defined in `Data.Monoid`:

```haskell
newtype Product a = Product { getProduct :: a }

instance Num a => Semigroup (Product a) where
        (<>) = (*)

-- the infix operator for `mappend` is `<>`.

instance Num a => Monoid (Product a) where
        mempty = Product 1
```

Sum

Defined in `Data.Monoid`:

```
newtype Sum a = Sum { getSum :: a }

instance Num a => Semigroup (Sum a) where
        (<>) = (+)

instance Num a => Monoid (Sum a) where
        mempty = Sum 0
```

Integer monoids

`Product`: Monoid under multiplication.

```
newtype Product a
```

```
> getProduct (Product 3 <> Product 4 <> mempty)
12
```

And `Sum`: Monoid under addition.

```
newtype Sum a
```

```
> getSum (Sum 1 <> Sum 2 <> mempty)
3
```

Algebras

As we said, typeclass declarations are maximally polymorphic.

They are not fully parametric: `<>` doesn't work with *any type in the universe*, only those with instances of `Monoid` .... er, `Semigroup`.

Aside

(since `Semigroup` now superclasses `Monoid`, `<>` really comes from `Semigroup` and `Monoid` only adds the `mempty` to the semigroup operation)

Algebras

But how do we know if a type has an instance of `Monoid`?

Algebras

By looking it up during compilation.

The instance is looked up by the type name.

Algebras

Welcome to GHC Core.

```
> 5 == 5

==================== Simplified expression ==================
let {
  it :: Bool
  it = == $fEqInteger 5 5 } in
thenIO (print $fShowBool it) ...
```

`fEqInteger` and `fShowBool` are references to the instances of `Eq` and `Show` that fit the types here.

Aside

(you can get a GHCi that shows you the Core output like that, too)

```
ghci -ddump-simpl -dsuppress-idinfo -dsuppress-coercions
-dsuppress-type-applications -dsuppress-uniques
-dsuppress-module-prefixes
```

Algebras

Typeclasses and types form an algebraic system, linked by instance declarations.

A slightly different operation, such as the difference between a monoid and a semigroup, means we give it a new typeclass name.

For different implementations of the same abstract function, though, we give the types new names.

At any rate, it is important to preserve uniqueness so the compiler knows what to do.

Aside

(yes, we're here to talk about functors, and we will! this will all be relevant to our interests)

Functions

Only two things matter in Haskell:

Types and Functions

But

Type constructors are functions that make types.

Data constructors are functions that make values of a type.

And the `(->)` type constructor makes …. functions?

The function type

Functions are a product of two values, sort of like tuples:

```
data (->) a b
infixr 0 `(->)`
```

It is often said that functions in Haskell always take one argument and return one result, and you can see that in the type!

Infix operator notation

Infix operators like + and == and -> do not have parentheses around them when they appear in their default infix position.

We need parentheses to make them prefix (or to do things like ask for their type in GHCi).

Infix operator notation

This

```
(->) a b
```

is the same as this

```
a -> b
```

Function arguments

Of course, `a` and `b` can themselves be functions!

So a function can accept a function as an argument and return a function as a result.

Function associativity

The arrow is a type-level infix operator that associates to the right.

So this

```
max  :: Ord a => a -> a -> a
```

is really this

```
max  :: Ord a => a -> (a -> a)
```

Function associativity

That's why we can write functions like this

```
a -> b -> c -> d
```

and still say we have "one argument, one result".

It associates like

```
a -> (b -> (c -> d))
```

It's still an `a -> b` function but the `b` here is more function: `(b -> (c -> d))`.

Aside

(we don't need those parentheses because Haskell is *curried by default*)

Another aside

(function *application* is left associative. so, `f x y` is `(f x) y`. the ability to apply a function to one argument and return a new function is what makes partial application so prevalent and convenient in Haskell.)

Function arguments

The default currying is why these type signatures need these parenetheses:

```
(.)  :: (b -> c) -> (a -> b) -> a -> c

flip :: (a -> b -> c) -> b -> a -> c
```

The first argument must be a function of the specified type.

Function arguments

If we took those parentheses off, we'd have

```
(.)  :: b -> (c -> (a -> (b -> (a -> c))))

flip :: a -> (b -> (c -> (b -> (a -> c))))
```

… and that is not at all what we want.

Functions

And the `(->)` type has typeclass instances:

```haskell
instance Monad ((->) r)
instance Functor ((->) r)
instance Applicative ((->) a)
instance Monoid b => Monoid (a -> b)
```

Functors

FINALLY

FUNCTORS

Functors

Typeclass definition:

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

Functors

Functors in Haskell are type constructors with a lawful definition of `fmap`.

Type constructors

Some types are “concrete” or *nullary* constructors. Many familiar types are concrete.

```
data Bool = False | True
```

Type constructors

Many “types” are parameterized.

```
data Maybe a = Nothing | Just a

data Either a b = Left a | Right b

data [] a = [] | a : [a]

data (,) a b = (,) a b

data (->) a b
```

Type constructors

Type constructors are functions that make types.

```
Maybe :: Type -> Type
```

```
Maybe String :: Type
```

Aside

(We actually call those “kinds” in Haskell. Kinds are the *types of types*.)

```
> :kind Maybe
Maybe :: * -> *

> :kind Maybe String
Maybe String :: *
```

Functors

Some typeclasses are typeclasses for *types.* Functor is a typeclass for *type constructors.*

Functors must be of type `Type -> Type`.

So they must be type constructors with *one parameter.*

Functors

And those type constructors must have a lawful implementation of `fmap`.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

But we know we can read that type signature like this

```
fmap :: (a -> b) -> (f a -> f b)
```

It takes one argument – a function – and returns a *lifted* function.

Functors

Since we also need to apply `fmap` to an `f a` value, where the `f` is a functor (type constructor), the compiler knows which `f` – and thus, which implementation of `fmap` we're using – by that `f`.

Functors

`fmap` gives us a way to apply a function to a value (or values in the case of lists and the like) that is *inside* some other type.

```
> fmap (+1) [1, 2, 3]
[2,3,4]

> fmap (+1) (Just 3)
Just 4
```

(`[]` and `Maybe` are functors; the numeric values there are not)

Functors

Lifting power

Functor instances

```
fmap @Maybe :: (a -> b) -> Maybe a -> Maybe b

fmap _ Nothing = Nothing
fmap function (Just x) = Just (function x)
```

Functor instances

Must be `Type -> Type` but `Either` and `(,)` are `Type -> Type -> Type` so we must partially apply the type constructor.

```
> :kind Either
Either :: * -> * -> *

> :kind Either _
Either _ :: * -> *
```

Functor instances

```
fmap @(Either _) :: (a -> b) -> Either t a -> Either t b
```

Notice which of `Either`'s parameters the `(a -> b)` function applies to.

Write the instance

```haskell
instance Functor (Either a) where
  -- fmap :: (a -> b) -> (Either t) a -> (Either t) b
```

Write the instance

```
instance Functor (Either a) where
  -- fmap :: (a -> b) -> (Either t) a -> (Either t) b
  fmap function (Left x)  = Left x
```

Write the instance

```haskell
instance Functor (Either a) where
  -- fmap :: (a -> b) -> (Either t) a -> (Either t) b
  fmap function (Left x)  = Left x
  fmap function (Right a) = Right (function a)
```

Applicative and Monad

There are other ways of lifting.

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

Applicative

`Applicative` typeclass is similar to `Functor`. Applicatives are functors, but with some extra stuff.

`pure` can lift an `a` into the necessary context.

```
 > :type pure
pure :: Applicative f => a -> f a

 > :type pure @Maybe
pure @Maybe :: a -> Maybe a

 > :type pure @(Either _)
pure @(Either _) :: a -> Either t a
```

Applicative

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

Applicative

```haskell
instance Applicative (Either a) where
  pure                    = Right
  Left x <*> _            = Left x
  _ <*> Left x            = Left x
  Right func <*> Right a = Right (func a)
```

Monad

`Monad` is just another functor typeclass.

```
fmap  :: Functor f => (a -> b) -> f a -> f b

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

Notice the inputs are the opposite of `fmap`.

Maybe Monad

```
> :set -XTypeApplications
> :type (>>=) @Maybe
(>>=) @Maybe :: Maybe a ->
                (a -> Maybe b) ->
                Maybe b
```

Either Monad

```
> :type (>>=) @(Either _)
(>>=) @(Either _) :: Either t a ->
                     (a -> Either t b) ->
                     Either t b
```

Monad instances

```haskell
instance Monad Maybe where
  Nothing >>= _     = Nothing
  Just a  >>= func  = func a

instance Monad (Either a) where
  Left x  >>= _    = Left x
  Right a >>= func = func a
```

Function instances

Since function `(->)` is a datatype, we can write instances for it as well.

Function instances

We only care about the `Functor` instance for our current purposes, but you should totally check out the `Applicative` and `Monad` instances because they are $_{\text{cool}}$.

The `(->)` type has two parameters.

```
data (->) a b
```

```
> :kind (->)
(->) :: * -> * -> *


> :kind ((->) _)
((->) _) :: * -> *
```

Function instances

So we'll need to partially apply it like we did with `Either`.

Compare

```
instance Functor (Either a) where

instance Functor ((->) a) where
```

Function instance

```
fmap :: (a -> b) -> ((->) t a) -> ((->) t b)

-- or

fmap :: (a -> b) -> (t -> a) -> (t -> b)
```

Function instance

Lifting power

Function instance

Compare

```
fmap @((->) _) :: (a -> b) -> (t -> a) -> t -> b

(.)            :: (b -> c) -> (a -> b) -> a -> c
```

Are these the same?

Function instance

```
instance Functor ((->) a) where
  fmap = (.)
```

Covariance

While we do not usually specify this, this is a typeclass for *covariant* functors.

Covariant functors are a way to express the relation between the outputs of a function.

```
1:(t -> a)
2:(t -> b)
```

An `(a -> b)` function takes us from the output of the first to the output of the second.

Covariance

`Either a b` is a function (that constructs a type).

The covariant `Either _` functor gives us a way to take `Either _ a` to `Either _ b`:

```
Right :: a -> Either _ a
Right :: b -> Either _ b
```

The `(a -> b)` function that `fmap` takes as its first argument takes us from the `a` to the `b`.

Aside

(the “co-” prefix here is not the one that means mathematical dual; it’s the one that means “together”, like in “cooperate”.)

Covariance

So that's *covariant* functors. We have one more typeclass to talk about before we go onto its opposite, *contravariance*.

Bifunctor!

Tired of ignoring your `Lefts`?

Try `Bifunctor`!

Bifunctor

`Bifunctor` is a typeclass that gives us the opportunity to lift *two* functions instead of one, for type constructors that have *two* parameters.

```haskell
class Bifunctor p where
  -- | Map over both arguments at the same time.
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
```

Bifunctors are *covariant* in both arguments, so we're transforming *two outputs*.

Bifunctor

```
bimap @Either :: (a -> b) -> (c -> d) ->
                 Either a c ->                 -- inputs
                 Either b d                    -- outputs

bimap @(,) :: (a -> b) -> (c -> d) ->
              (,) a c  -> (,) b d
```

Bimapping

```
greet = bimap ("hello " ++) ("goodbye " ++)

 > greet (Left "Julie")
Left "hello Julie"

 > greet (Right "to all that")
Right "goodbye to all that"

 > greet ("Poland", "Switzerland")
("hello Poland","goodbye Switzerland")
```

Tomorrow

Profunctors are

- bifunctors

that are

- contravariant in their first argument
- covariant in their second argument.

We've seen covariant functors (many) and covered bifunctors.

Tomorrow we'll tackle contravariance and combine everything we know into one sweet `dimap`.