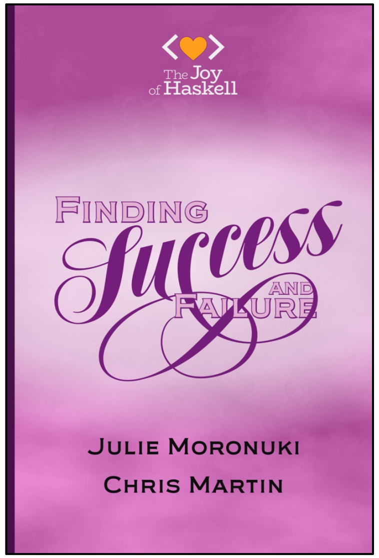
Applicative Functor

Part 2

Learn more about the canonical definition of the Applicative typeclass by looking at a great Haskell validation example by Chris Martin and Julie Moronuki
Then see it translated to Scala

@chris_martin @argumatronic



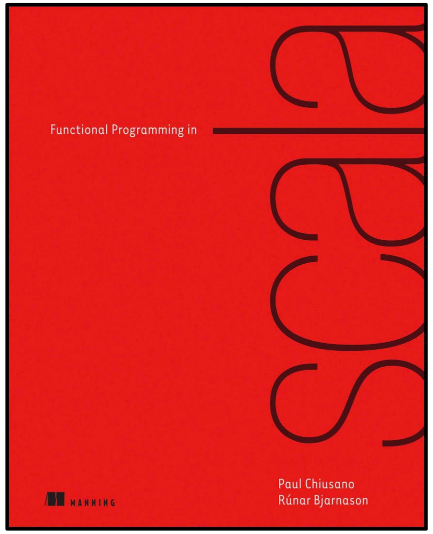
slides by @philip_schwarz

In **Part 1** we saw that the **Applicative typeclass** can be defined either in terms of **unit** and **map2**, or in terms of **unit** and **apply** (also known as **ap**).

@philip_schwarz

The name applicative comes from the fact that we can formulate the **`Applicative`** interface using an alternate set of primitives, **`unit`** and the function **`apply`**, rather than **`unit`** and **`map2`**. …this formulation is equivalent in expressiveness since … **`map2`** and **`map`** [can be defined] in terms of **`unit`** and **`apply`** … [and] **`apply`** can be implemented in terms of **`map2`** and **`unit`**.

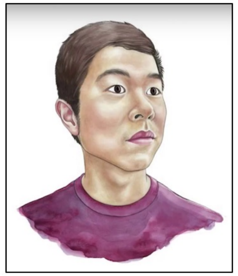
```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}

trait Applicative[F[_]] extends Functor[F] {
  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] = map2(fa, unit(()))((a, _) => f(a))
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C]
}
```



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Part 1 concluded with **Adelbert Chang** explaining that "**apply has a weird signature, at least in Scala**, where you have a function inside of an **F** and then you have an **effectful value**, and you want to **apply the function** to that value, all while remaining in **F**, and **this has a nicer theoretical story in Haskell, but in Scala it sort of makes for an awkward API**"

Applicative defined in terms of **zip** + **pure** or in terms of **ap** + **pure**



Adelbert Chang
@adelbertchang

Applicative

The Functor, Applicative, Monad talk YouTube

```scala
trait Applicative[F[_]] extends Functor[F] {
  def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)]
  def pure[A](a: A): F[A]

  def map[A, B](fa: F[A])(f: A => B): F[B]

  def ap[A, B, C](ff: F[A => B])(fa: F[A]): F[B] =
    map(zip(ff, fa)) { case (f, a) => f(a) }
}
```

As a quick note, if you go to say **Cats** or **Scalaz** today, or **Haskell** even, and you look at **Applicative**, what you'll see is this **ap** formulation instead, so what I presented as **zip**, **map** and **pure**, we will typically see as **ap**, and **ap** sort of has a **weird type signature**, at least in **Scala**, **where you have a function inside of an F, and then you have an effectful value, and you want to apply the function to that value, all while remaining in F**, and **this has a nice theoretical story**, and sort of has **a nicer story in Haskell**, but in **Scala**, this sort of makes for an **awkward API**, and so **I like to introduce applicative in terms of zip and map for that reason, I think it makes for a better story, and I think zip is conceptually simpler, because you can sort of see that zip is about composing two values, in the easiest way possible, whereas ap sort of has a weird signature**.

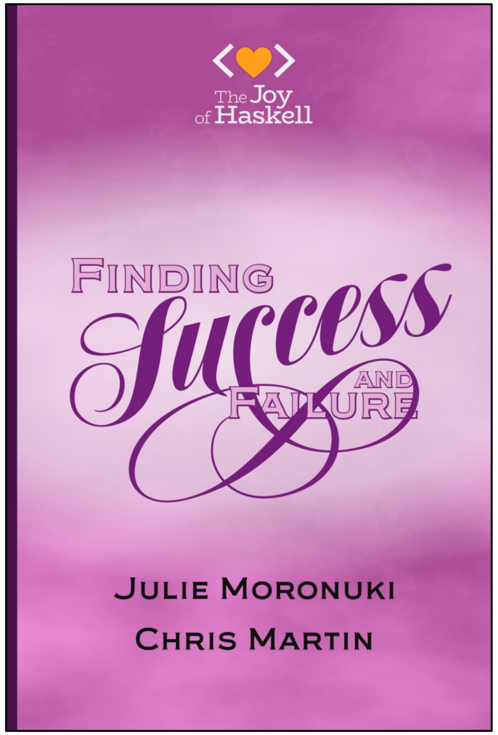
That thing said, **ap is, for historical reasons, like the canonical representation of Applicative**, so if after this talk you go and look what **Applicative** is, you'll probably see **ap**. Just as a quick note, you can implement **ap** in terms of **map** and **zip**, like I have here. You can also go the other way, you can implement **zip** and **map** in terms of **ap**, and so, exercise left to the reader.

@philip_schwarz

Recently I came across a great book called **Finding Success (and Failure) in Haskell**. In addition to generally being very interesting and useful, it contains **a great example that shows how to do validation in Haskell progressively better, and that culminates in using Haskell's Validation Applicative**.

I am grateful to **Julie Moronuki** and **Chris Martin** for writing such a great book and I believe **Scala** developers will also benefit from reading it.

In this slide deck I am going to look in detail at just two sections of their example, the one where they switch from the **Either Monad** to the **Either Applicative** and the one where they switch from the **Either Applicative** to the **Validation Applicative**. In doing so, I will translate the code in the example from **Haskell** to **Scala**, because I found it a good way of reinforcing the ideas behind **Haskell**'s canonical representation of the **Applicative typeclass** and the ideas behind its **Validation** instance.



@chris_martin @argumatronic

The validation example that **Julie Moronuki** and **Chris Martin** chose for their book is about **validating the username and password of a user**. I am assuming that if you are going through this slide deck then **you are a developer with a strong interest in functional programming and you are happy to learn by reading both Haskell and Scala code**. If you are mainly a **Scala** developer, don't worry if your **Haskell** knoweldge is minimal, you'll be fine. If you are mainly a **Haskell** developer, you will get an opportunity to see **one way in which some Haskell concepts/abstractions can be reproduced in Scala**.

The good thing about the example being simple, is that I don't have to spend any time verbally explaining how it works, I can just show you the code and make some observations at key points. **See the book for a great, detailed explanation of how the authors got to various points on their journey and what they learned in the process**, especially if you are new to **Haskell** or new to **Monads** and **Applicatives**.

In the next slide you'll see the code as it is before the authors switch from the **Either Monad** to the **Either Applicative**.

```haskell
newtype Password = Password String
  deriving Show
```

```haskell
newtype Error = Error String
  deriving Show
```

```haskell
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                           \longer than 20 characters.")
      False -> Right (Password password)
```

```haskell
requireAlphaNum :: String -> Either Error String
requireAlphaNum password =
   case (all isAlphaNum password) of
      False -> Left "Your password cannot contain \
                    \white space or special characters."
      True  -> Right password
```

```haskell
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Your password cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)
```

```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a password\n> "
    password <- Password <$> getLine
    print (validatePassword password)
```

the **bind** function of the **Either monad**

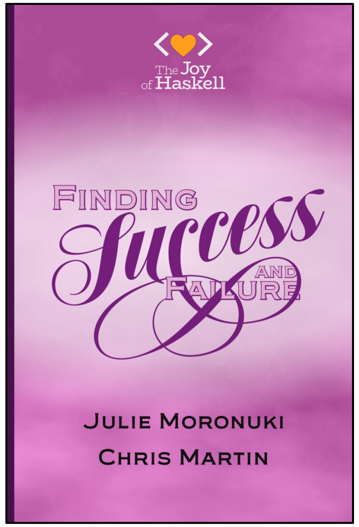
The **fmap** (**map**) function of the **IO monad**

Here is a **Haskell** program that asks the user for a password, validates it, prints out to console either the password or a validation error message.

The program is using the **IO Monad** and the **Either Monad**.

See the next slide for sample executions of the program.

@philip_schwarz



@chris_martin
@argumatronic

In **Haskell**, the entry point for an executable program must be named **main** and have an **IO type** (nearly always **IO ()**). We do not need to understand this fully right now. In a very general sense, it means that it does some **I/O** and performs some **effects**.

```haskell
newtype Password = Password String
  deriving Show
```

```haskell
newtype Error = Error String
  deriving Show
```

```haskell
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                           \longer than 20 characters.")
      False -> Right (Password password)
```

```haskell
requireAlphaNum :: String -> Either Error String
requireAlphaNum password =
   case (all isAlphaNum password) of
      False -> Left "Your password cannot contain \
                    \white space or special characters."
      True  -> Right password
```

```haskell
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Your password cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)
```

```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a password\n> "
    password <- Password <$> getLine
    print (validatePassword password)
```

@chris_martin
@argumatronic

See below for a few sample executions I did for both sunny day and rainy day scenarios.

```
*Main> main
Please enter a password
> excessivelylongpassword
Left (Error "Your password cannot be longer than 20 characters.")

*Main> main
Please enter a password
> has.special*chars
Left (Error "Cannot contain white space or special characters.")

*Main> main
Please enter a password
> has space
Left (Error "Cannot contain white space or special characters.")

*Main> main
Please enter a password
>
Left (Error "Your password cannot be empty.")

*Main> main
Please enter a password
> pa22w0rd
Right (Password "pa22w0rd")

*Main> main
Please enter a password
>    leadingspacesareok
Right (Password "leadingspacesareok")
*Main>
```

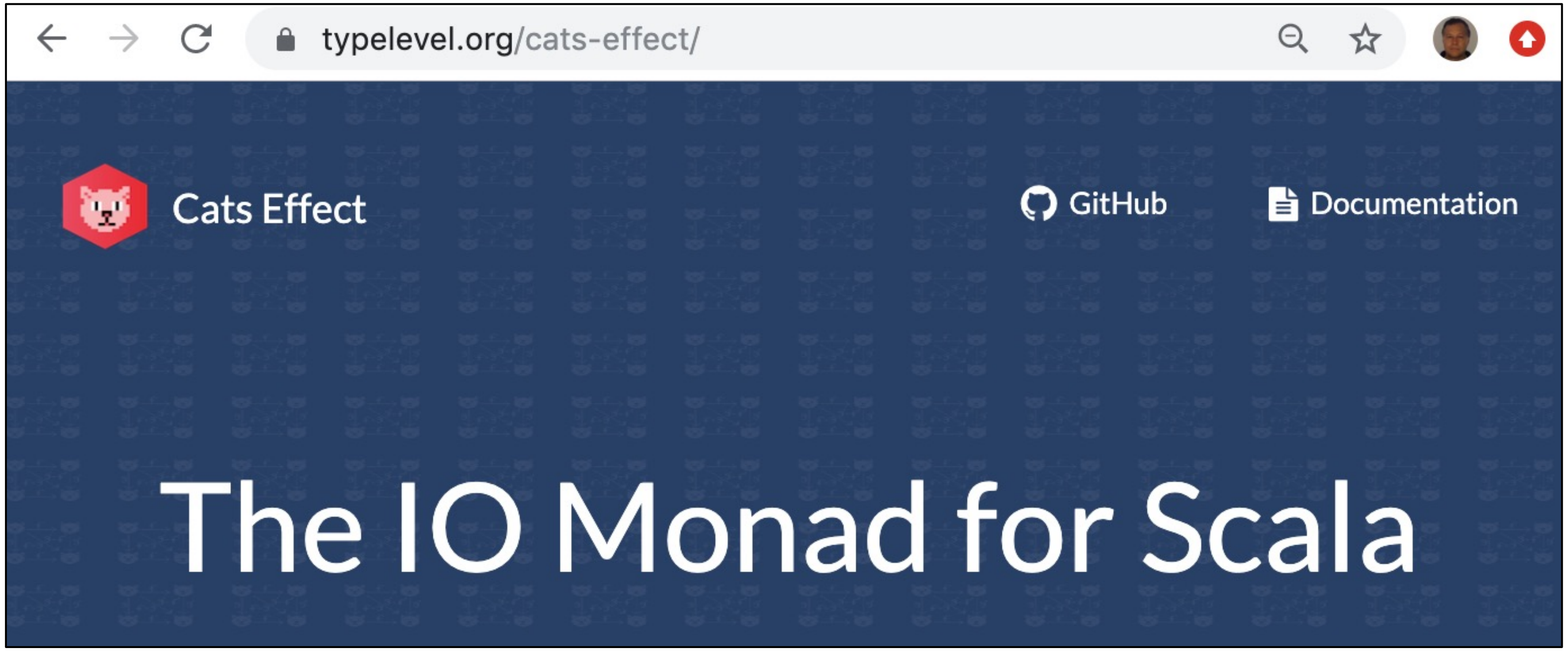
@philip_schwarz

I am going to translate the Haskell password program into Scala, but what about the `IO` type for performing side effects (e.g. I/O)?

Let's use the `IO` data type provided by Cats Effect.

If you are going through this slide for the first time, you don't need to fully absorb all of the text below before moving on.

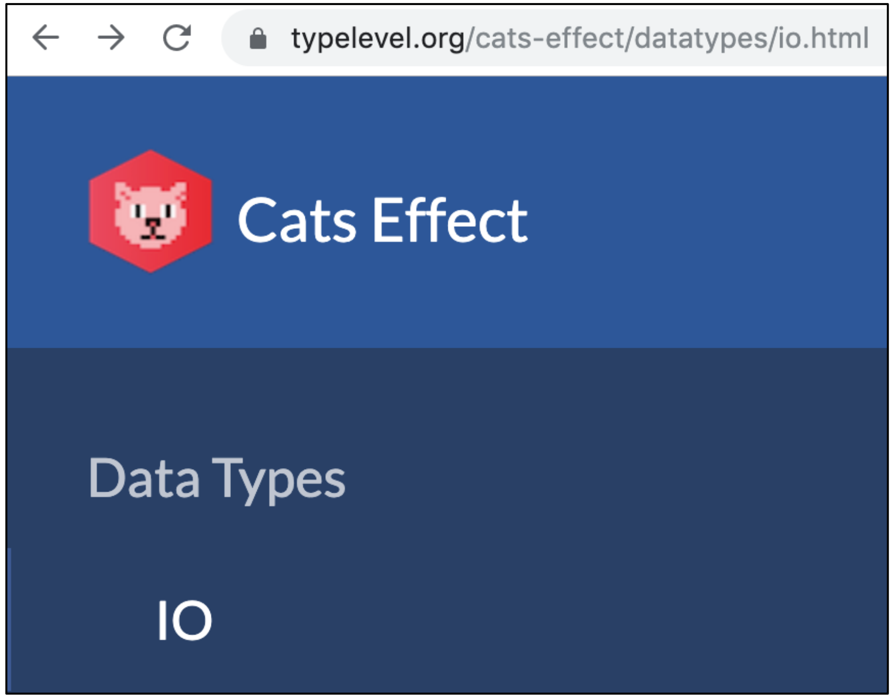
Haskell ⇒ Scala



This project aims to provide a standard `IO` type for the Cats ecosystem, as well as a set of typeclasses (and associated laws!) which characterize general effect types. This project was *explicitly* designed with the constraints of the JVM and of JavaScript in mind. Critically, this means two things:

- Manages both synchronous *and* asynchronous (callback-driven) effects
- Compatible with a single-threaded runtime

In this way, IO is more similar to common Task implementations than it is to the classic scalaz.effect.IO or even Haskell's IO, both of which are purely synchronous in nature. As Haskell's runtime uses green threading, a synchronous `IO` (and the requisite thread blocking) makes a lot of sense. With Scala though, we're either on a runtime with native threads (the JVM) or only a single thread (JavaScript), meaning that asynchronous effects are every bit as important as synchronous ones.




```scala
import cats.effect.IO

val ioa = IO { println("hey!") }

val program: IO[Unit] =
  for {
    _ <- ioa
    _ <- ioa
  } yield ()

program.unsafeRunSync()
//=> hey!
//=> hey!
()
```

IO

A **data type** for encoding **side effects** as **pure values**, capable of expressing both **synchronous** and **asynchronous** computations.

A value of type `IO[A]` is a computation which, when evaluated, can **perform effects** before returning a value of type `A`.

`IO` values are **pure**, immutable values and thus preserves **referential transparency**, being usable in functional programming. **An `IO` is a data structure that represents just a description of a side effectful computation**.

`IO` can describe **synchronous** or **asynchronous** computations that:

1. on evaluation yield exactly one result
2. can end in either success or failure and in case of failure **flatMap** chains get short-circuited (`IO` implementing the **algebra** of `MonadError`)
3. can be canceled, but note this capability relies on the user to provide cancellation logic

Effects described via this abstraction are not evaluated until the "end of the world", which is to say, when one of the "unsafe" methods are used. Effectful results are not memoized, meaning that memory overhead is minimal (and no leaks), and also that **a single effect may be run multiple times in a referentially-transparent manner**. For example:

The above example prints "hey!" twice, as the **effect** re-runs each time it is sequenced in the monadic chain.

In the next slide we'll see the same **Haskell** program we saw earlier but with an equivalent **Scala** version next to it.

Here on the right is the **Scala** equivalent of the **Haskell** program on the left.

```haskell
newtype Password = Password String
  deriving Show

newtype Error = Error String
  deriving Show

checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                           \longer than 20 characters.")
      False -> Right (Password password)

requireAlphaNum :: String -> Either Error String
requireAlphaNum password =
   case (all isAlphaNum password) of
      False -> Left "Your password cannot contain \
                    \white space or special characters."
      True  -> Right password

cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Your password cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)

validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength

main :: IO ()
main =
  do
    putStr "Please enter a password\n> "
    password <- Password <$> getLine
    print (validatePassword password)
```

Haskell

@chris_martin
@argumatronic

```scala
case class Password(password:String)
case class Error(error:String)

def checkPasswordLength(password: String): Either[Error, Password] =
   password.length > 20 match {
      case true  => Left(Error("Your password cannot be longer" +
                               "than 20 characters."))
      case false => Right(Password(password))
   }

def requireAlphaNum(password: String): Either[Error, String] =
   password.forall(_.isLetterOrDigit) match {
      case false  => Left(Error("Your password cannot contain white " +
                                "space or special characters."))
      case true   => Right(password)
   }

def cleanWhitespace(password:String): Either[Error, String] =
   password.dropWhile(_.isWhitespace) match {
      case pwd if pwd.isEmpty => Left(Error("Your password cannot be empty."))
      case pwd                => Right(pwd)
}

def validatePassword(password: Password): Either[Error,Password] = password match {
    case Password(pwd) =>
      cleanWhitespace(pwd)
        .flatMap(requireAlphaNum)
        .flatMap(checkPasswordLength)
}

val main: IO[Unit] =
   for {
         _ <- print("Please enter a password.\n")
       pwd <- getLine map Password
         _ <- print(validatePassword(pwd).toString)
   } yield ()

main.unsafeRunSync
```

```scala
import cats.effect.IO

def getLine =
  IO { scala.io.StdIn.readLine }
def print(s: String): IO[Unit] =
  IO { scala.Predef.print(s) }
```

Scala

In chapter seven of **Finding Success (and Failure) in Haskell**, the authors introduce the **Applicative typeclass**.

This chapter picks up where the previous one ended and adds a `validateUsername` function. Then, since we'd like to keep a username and a password together as a single value, we write a **product type** called `User` and a `makeUser` function that constructs a `User` from the conjunction of a valid `Username` and a valid `Password`. We will introduce the **Applicative typeclass** to help us write that function.

@chris_martin
@argumatronic

In the next slide we look at the corresponding code changes

```haskell
newtype Password = Password String
  deriving Show
```

```haskell
newtype Error = Error String
  deriving Show
```

```haskell
newtype Username = Username String
  deriving Show
```

```haskell
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                           \longer than 20 characters.")
      False -> Right (Password password)
```

The 1st change is the introduction of a **Username** and associated **checkUsernameLength** and **validateUsername** functions, almost identical to those for **Password**.

```haskell
checkUsernameLength :: String -> Either Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Left (Error "Your username cannot be \
                           \longer than 15 characters.")
      False -> Right (Username username)
```

```haskell
requireAlphaNum :: String -> Either Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Left "~~Your password c~~Cannot contain \
                    \white space or special characters."
      True  -> Right input
```

The 2nd change is that **requireAlphaNum** and **cleanWhitespace** are now used for both username and password.

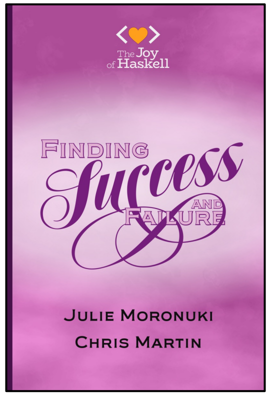
```haskell
validateUsername :: Username -> Either Error Username
validateUsername (Username username) =
  cleanWhitespace username
    >>= requireAlphaNum
    >>= checkUsernameLength
```

```haskell
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "~~Your password c~~Cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)
```

The 3rd change is the introduction of a **User**, consisting of a **Username** and a **Password**, together with a function **makeUser** that creates a **User** from a username and a password, provided they are both valid. We'll look at the function in detail in the slide after next.

```haskell
data User = User Username Password
  deriving Show
```

```haskell
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```




```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a password\n> "
    password <- Password <$> getLine
    print (validatePassword password)
```

The 4td change is that instead of just asking for a password and printing it, the main function now also asks for a username, creates a **User** with the username and password, and prints the **User** rather than just the password.

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

@chris_martin
@argumatronic

```haskell
newtype Username = Username String
  deriving Show
```

```haskell
newtype Password = Password String
  deriving Show
```

```haskell
data User = User Username Password
  deriving Show
```

```haskell
newtype Error = Error String
  deriving Show
```

```haskell
checkUsernameLength :: String -> Either Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Left (Error "Your username cannot be \
                          \longer than 15 characters.")
      False -> Right (Username username)
```

```haskell
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                          \longer than 20 characters.")
      False -> Right (Password password)
```

```haskell
requireAlphaNum :: String -> Either Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Left "Cannot contain \
                    \white space or special characters."
      True  -> Right input
```

```haskell
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)
```

Here is how the code looks after the changes

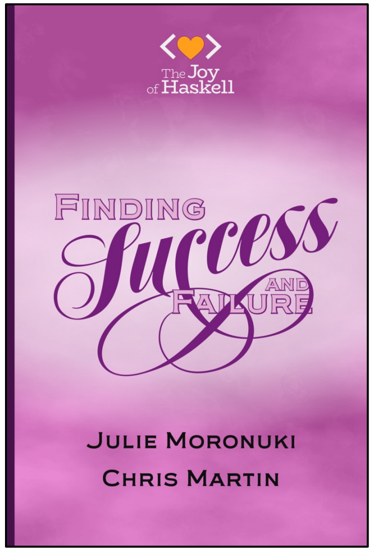
```haskell
validateUsername :: Username -> Either Error Username
validateUsername (Username username) =
  cleanWhitespace username
   >>= requireAlphaNum
   >>= checkUsernameLength
```

```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```haskell
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

@chris_martin
@argumatronic



Just like **validatePassword**, the new **validateUsername** function uses **>>=**, the **bind** function of the **Either Monad**. Every **Monad** is also an **Applicative Functor** and the new **makeUser** function uses both **<$>** and **<*>**, which are the **map** function and the **apply** function of the **Either Applicative Functor**. See below and next slide for how the **makeUser** function works.

```haskell
validateUsername :: Username -> Either Error Username
validateUsername (Username username) =
  cleanWhitespace username
    >>= requireAlphaNum
    >>= checkUsernameLength
```

```haskell
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

@chris_martin
@argumatronic

```haskell
cleanWhitespace :: String -> Either Error String
```

```haskell
requireAlphaNum :: String -> Either Error String
```

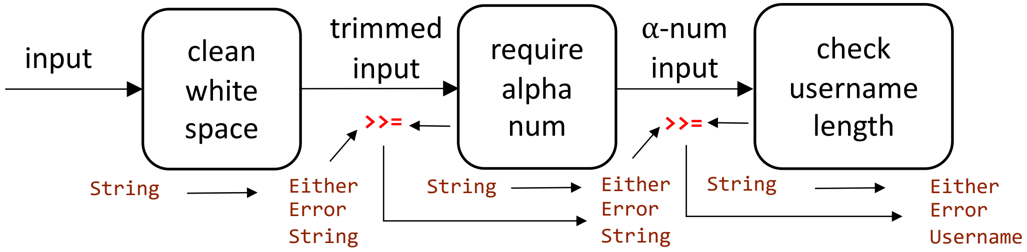
```haskell
checkUsernameLength :: String -> Either Error Username
```

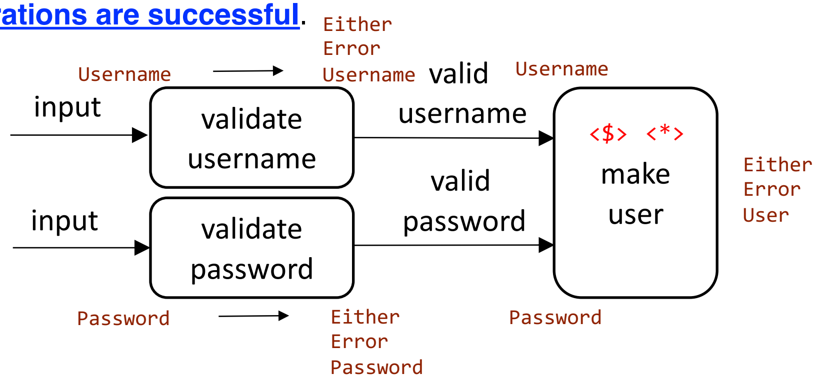
```haskell
(<$>) :: Functor m => m a -> (a -> b) -> m b
```

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

```haskell
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

When we wrote the **validateUsername** and **validatePassword** functions, we noted **the importance of using the monadic (>>=) operator when the input of a function must depend on the output of the previous function**. We wanted the inputs to our character and length checks to depend on the output of **cleanWhitespace** because it might have transformed the data as it flowed through our pipeline of validators. **However, in this case, we have a different situation**. **We want to validate the name and password inputs independently – the validity of the password does not depend on the validity of the username, nor vice versa – and then bring them together in the User type only if both operations are successful**.





I have annotated the diagrams a bit to aid my comprehension further.

In the next slide we look a little bit closer at how the **makeUser** function uses **<$>** and **<*>** to turn **Either Error Username** and **Either Error Password** into **Either Error User**.

For that, then, we will use the primary operator of a different typeclass: Applicative. We often call that operator "tie-fighter" or sometimes "apply" or "ap". Applicative occupies a space between Functor (from which we get (<$>)) and Monad, and (<*>) is doing something very similar to (<$>) and (>>=), which is allowing for function application in the presence of some outer type structure.

In our case, the "outer type structure" is the Either a functor. As we've seen before with fmap and (>>=), (<*>) must effectively ignore the a parameter of Either, which is the Error in our case, and only apply the function to the Right b values. It still returns a Left error value if either side evaluates to the error case, but unlike (>>=), there's nothing inherent in the type of (<*>) that would force us to "short-circuit" on an error value. We don't see evidence of this in the Either applicative, which behaves coherently with its monad, but we will see the difference once we're using the Validation type.

To see how <$> and <*> are working together in the **makeUser** function, let's take the four different combinations of values that **validateUsername** and **validatePassword** can return, and see how <$> and <*> process them.

@philip_schwarz

fmap (map)
```
(<$>) :: Functor m => m a -> (a -> b) -> m b
```

ap (apply)
```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

```
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

```
User <$> Right(Username "fredsmith") <*> Right(Password "pa22w0rd")

   Right(User(Username "fredsmith")) <*> Right(Password "pa22w0rd")

            Right( User (Username "fredsmith") (Password "pa22w0rd"))


User <$> Left(Error "Cannot be blank.") <*> Right(Password "pa22w0rd")

         Left(Error "Cannot be blank.") <*> Right(Password "pa22w0rd")

                   Left(Error "Cannot be blank.")


User <$> Right(Username "fredsmith") <*> Left(Error "Cannot be blank.")

   Right(User(Username "fredsmith")) <*> Left(Error "Cannot be blank.")

                   Left(Error "Cannot be blank.")


User <$> Left(Error "Cannot be blank.") <*> Left(Error "Cannot be blank.")

         Left(Error "Cannot be blank.") <*> Left(Error "Cannot be blank.")

                      Left(Error "Cannot be blank.")
```

One way to visualize the result of this

```
User <$> Right(Username "fredsmith")
```

is to think of **User** applied to its first argument as a **lambda** function that takes **User**'s second parameter

```
Right(\pwd -> User (Username "fredsmith") pwd)
```

That way maybe we can better visualise this

```
Right(User(Username "fredsmith")) <*> Right(Password "pa22w0rd")
```

as the application of the **partially applied User** function to its second parameter

```
Right(\pwd->User(Username "fredsmith")pwd) <*> Right(Password "pa22w0rd")
```

```haskell
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

```haskell
(<$>) :: Functor m => m a -> (a -> b) -> m b
```

```haskell
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

Let's see the four cases again but this time from the point of view of calling the **makeUser** function.

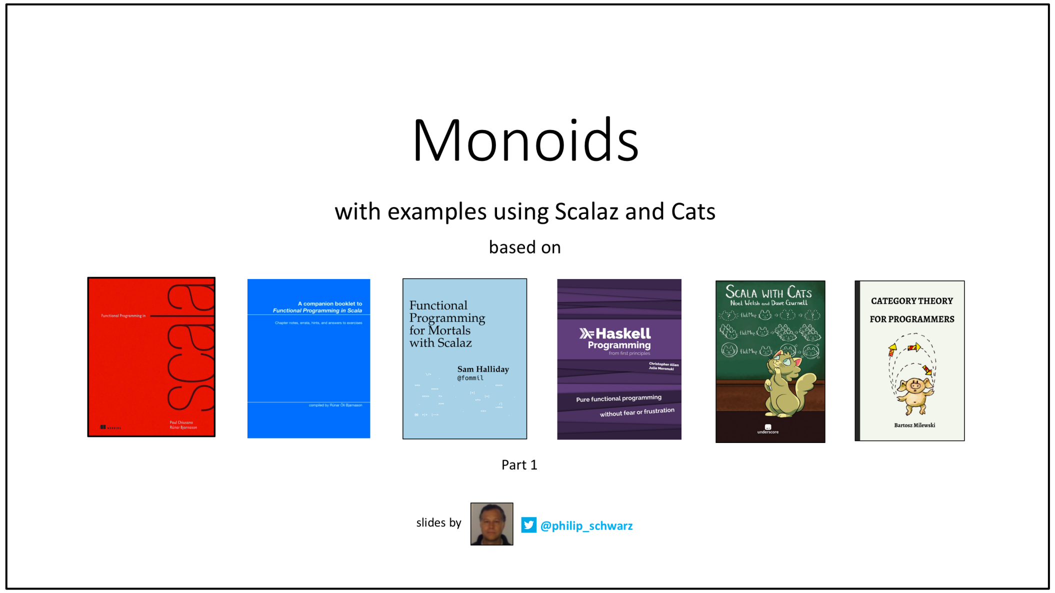
```
*Main> makeUser (Username "fredsmith") (Password "pa22w0rd")
Right (User (Username "fredsmith") (Password "pa22w0rd"))

*Main> makeUser (Username "extremelylongusername") (Password "pa22w0rd")
Left (Error "Your username cannot be longer than 15 characters.")

*Main> makeUser (Username "fredsmith") (Password "password with spaces")
Left (Error "Cannot contain white space or special characters.")

*Main> makeUser (Username "extremelylongusername") (Password "password with spaces")
Left (Error "Your username cannot be longer than 15 characters.")

*Main>
```

The bottom case is a problem: **when both username and password are invalid then the makeUser function only reports the problem with the username**.

@philip_schwarz

In upcoming slides we are going to see how the authors of **Finding Success (and Failure) in Haskell** improve the program so that it does not suffer from the problem just described. Because they will be be referring to the **Semigroup typeclass**, the next three slides are a quick reminder of the **Semigroup** and **Monoid typeclasses** (defining the latter helps defining the former).

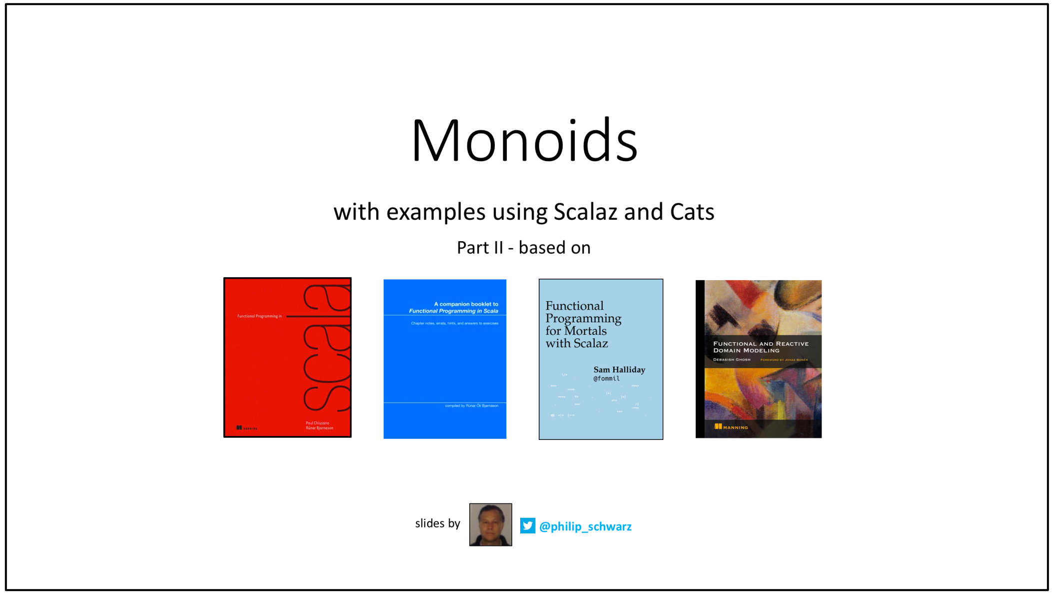
If you already know what a **Semigroup** is then feel free to skip the next three slides. Also, if you want to know more about **Monoids**, see the two slide decks on the right.



https://www.slideshare.net/pjschwarz/monoids-with-examples-using-scalaz-and-cats-part-1

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/monoids-with-examples-using-scalaz-and-cats-part-2



Monoid is an embarrassingly simple but amazingly powerful concept. It's the concept behind basic arithmetics: Both addition and multiplication form a monoid. **Monoids are ubiquitous in programming**. They show up as strings, lists, foldable data structures, futures in concurrent programming, events in functional reactive programming, and so on.
…

In **Haskell** we can define a type class for **monoids** — a type for which there is a **neutral element** called **mempty** and a **binary operation** called **mappend**:

```
class Monoid m where
mempty :: m
mappend :: m -> m -> m
```
…
As an example, let's declare **String** to be a **monoid** by providing the implementation of `mempty` and `mappend` (this is, in fact, done for you in the standard Prelude):

```
instance Monoid String where
mempty = ""
mappend = (++)
```

Here, we have reused the **list concatenation operator** (++), because a **String** is just a list of characters.

A word about **Haskell** syntax: Any infix operator can be turned into a two-argument function by surrounding it with parentheses. Given two strings, you can **concatenate** them by inserting ++ between them:

```
"Hello " ++ "world!”
```

or by passing them as two arguments to the parenthesized (++):

```
(++) "Hello " "world!"
```

@BartoszMilewski

Bartosz Milewski

Monoid

A monoid is a **binary associative operation** with an **identity**.

…

For **lists**, we have a **binary operator**, (++), that joins two lists together. We can also use a function, **mappend**, from the `Monoid` type class to do the same thing:

```
Prelude> mappend [1, 2, 3] [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

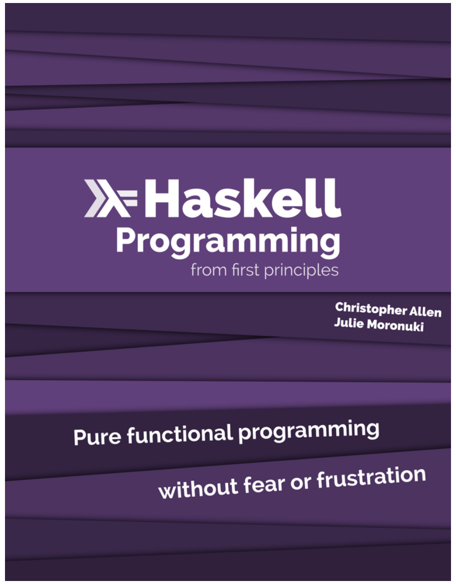
For **lists**, the empty list, [], is the **identity** value:

```
mappend [1..5] [] = [1..5]
mappend [] [1..5] = [1..5]
```

We can rewrite this as a more general rule, using **mempty** from the `Monoid` type class as a **generic identity value** (more on this later):

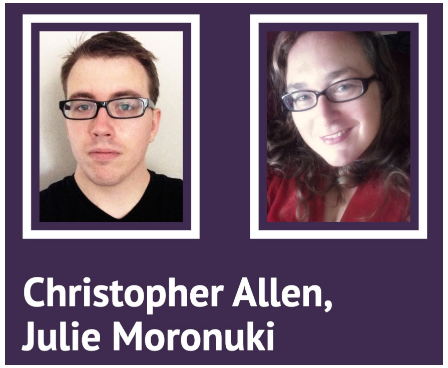
```
mappend x mempty = x
mappend mempty x = x
```

In plain English, **a monoid is a function that takes two arguments and follows two laws**: associativity and **identity**. **Associativity** means the arguments can be regrouped (or reparenthesized, or reassociated) in different orders and give the same result, as in addition. **Identity** means there exists some value such that when we pass it as input to our function, the operation is rendered moot and the other value is returned, such as when we add zero or multiply by one. `Monoid` **is the type class that generalizes these laws across types**.



By Christopher Allen
and Julie Moronuki

@bitemyapp @argumatronic



Semigroup

Mathematicians play with **algebras** like that creepy kid you knew in grade school who would pull legs off of insects. Sometimes, they glue legs onto insects too, but in the case where we're going from **Monoid** to **Semigroup**, we're pulling a leg off.

In this case, the leg is our **identity**. To get from a **monoid** to a **semigroup**, we simply no longer furnish nor require an **identity**. The **core operation** remains **binary** and **associative**. With this, our definition of **Semigroup** is:

```
class Semigroup a where
(<>) :: a -> a -> a
```

And we're left with one law:

```
(a <> b) <> c = a <> (b <> c)
```

Semigroup still provides a **binary associative operation**, one that typically **joins two things together** (as in **concatenation** or **summation**), but doesn't have an **identity** value. In that sense, it's a weaker **algebra**.

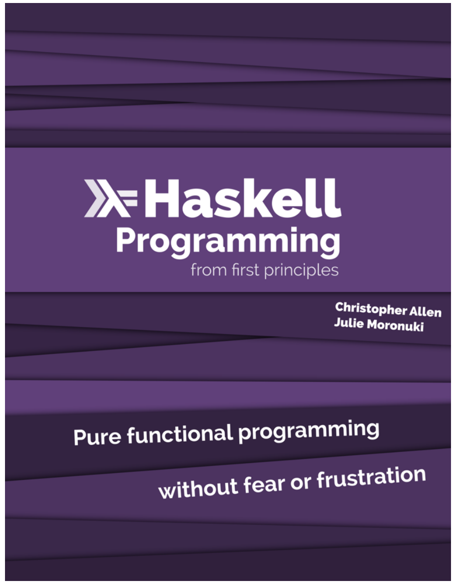
…

NonEmpty, a useful datatype

One useful datatype that can't have a **Monoid** instance but does have a **Semigroup** instance is the **NonEmpty** list type. It is a list datatype that can never be an empty list…

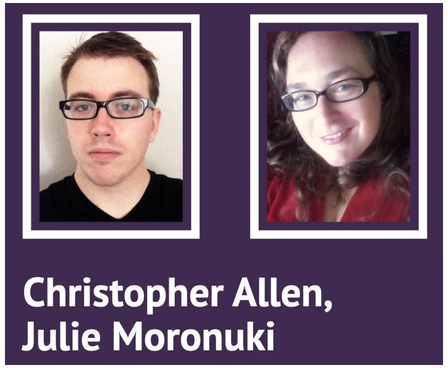
We can't write a **Monoid** for **NonEmpty** because it has no **identity** value by design! There is no empty list to serve as an **identity** for any operation over a **NonEmpty** list, yet there is still a **binary associative operation**: two **NonEmpty** lists can still be **concatenated**.

A type with a canonical **binary associative operation** but no **identity** value is a natural fit for **Semigroup**.



By Christopher Allen
and Julie Moronuki

@bitemyapp @argumatronic



After that refresher on **Semigroup** and **Monoid**, let’s turn to chapter eight of **Finding Success (and Failure) in Haskell**, in which the authors address the problem in the current program by switching from **Either** to **Validation**.

Refactoring with Validation

In this chapter we do a thorough refactoring to switch from **Either** to **Validation**, which comes from the package called validation available on Hackage.

These two types are essentially the same. **More precisely, these two types are isomorphic**, by which we mean that **you can convert values back and forth between Either and Validation without discarding any information in the conversion.**

But their Applicative instances are quite different and switching to Validation allows us to accumulate errors on the left. **In order to do this, we'll need to learn about a typeclass called Semigroup to handle the accumulation of Error values**.

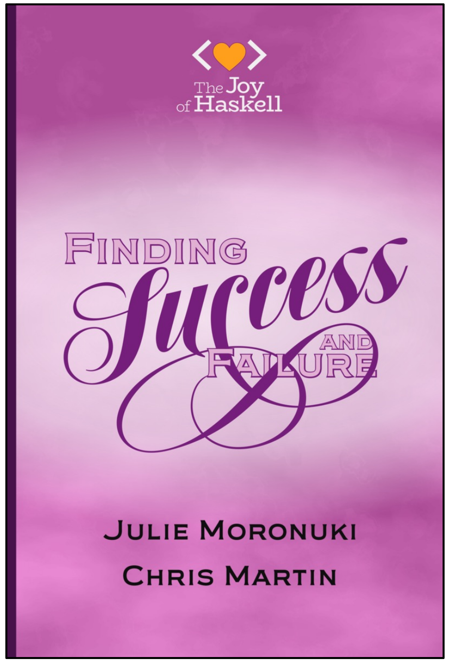
Introducing validation

Although the Validation type is isomorphic to Either, they are different types, so they can have different instances of the Applicative class. Since instance declarations define how functions work, this means overloaded operators from the Applicative typeclass can work differently for Either and Validation.

We used the **Applicative** for **Either** in the last chapter and we noted **we used Applicative instead of Monad when we didn't need the input of one function to depend on the output of the other. We also noted that although we weren't technically getting the "short-circuiting" behavior of Monad**, **we could still only return one error string**. **The "accumulating Applicative" of Validation will allow us to return more than one**.

The way the Applicative for Validation works is that it appends values on the left/error side using a Semigroup. We will talk more about **semigroups** later, but for now we can say that **our program will be relying on the semigroup for lists, which is concatenation**.

@chris_martin
@argumatronic



If you type import Data.Validation and then :info **Validation**, you can see the type definition

```
data Validation err a = Failure err | Success a
```

The type has two parameters, one called **err** and the other called **a**, and two constructors, **Failure err** and **Success a**. The output of :info **Validation** also includes a list of instances.

Validation is not a Monad

The instance list does not include **Monad**. Because of the accumulation on the left, the **Validation** type is not a **monad**. If it were a monad, it would have to **"short circuit"** and lose the accumulation of values on the left side. Remember, **monadic binds**, since they are a sort of shorthand for nested case expressions, must evaluate sequentially, following a conditional, branching pattern. **When the branch that it's evaluating reaches an end, it must stop. So, it would never have the opportunity to evaluate further and find out if there are more errors. However,** since functions chained together with **applicative** operators instead of **monadic** ones can be evaluated independently, we can accumulate the errors from several function applications, concatenate them using the underlying **semigroup**, and return as many errors as there are.

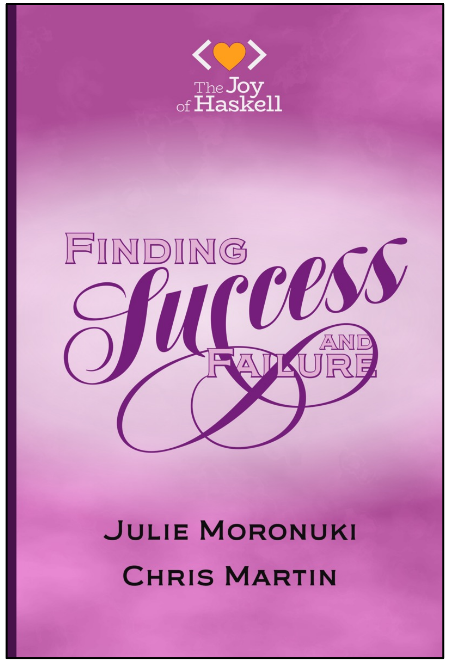
err needs a Semigroup

Notice that **Applicative** instance has a **Semigroup** constraint on the left type parameter.

```
instance Semigroup err => Applicative (Validation err)
```

That's telling us that **the err parameter that appears in Failure err must be a semigroup, or else we don't have an Applicative for Validation err**. You can read the => symbol like **implication**: If err is **Semigroupal** then **Validation err** is **applicative**. **Our return types all have our Error type as the first argument to Either, so as we convert this to use Validation, the err parameter of Validation will be Error**.

@chris_martin
@argumatronic



@philip_schwarz

In the next slide we are going to see again what the code looks like just before the refactoring, and in the slide after that we start looking at the detail of the refactoring.

```haskell
newtype Username = Username String
  deriving Show
```

```haskell
newtype Password = Password String
  deriving Show
```

```haskell
data User = User Username Password
  deriving Show
```

```haskell
newtype Error = Error String
  deriving Show
```

```haskell
checkUsernameLength :: String -> Either Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Left (Error "Your username cannot be \
                            \longer than 15 characters.")
      False -> Right (Username username)
```

```haskell
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                            \longer than 20 characters.")
      False -> Right (Password password)
```

```haskell
requireAlphaNum :: String -> Either Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Left "Cannot contain \
                     \white space or special characters."
      True  -> Right input
```

```haskell
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)
```

```haskell
validateUsername :: Username -> Either Error Username
validateUsername (Username username) =
  cleanWhitespace username
    >>= requireAlphaNum
    >>= checkUsernameLength
```

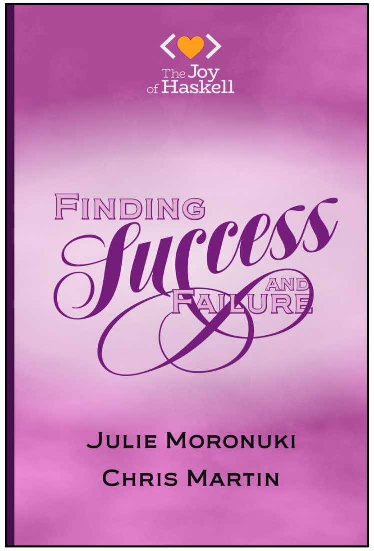
```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```haskell
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

Just a reminder of what the code looks like at this stage

@chris_martin
@argumatronic



Before we start refactoring, just a reminder that being a **Monad**, **Either** is also an **Applicative**, i.e. it has a **<*>** operator (the **tie-fighter** operator - aka **ap** or **apply**).

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

Ok, let's start: we are not happy with the way our `makeUser` function currently works.

```
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

This is because errors are modeled using **Either** and processed using its **<*>** operator, which means that **when both `validateUsername` and `validatePassword` return an error, we only get the error returned by `validateUsername`.**

In what follows below, the sample a -> b function that I am going to use is (+) 1, i.e. the binary plus function applied to one (i.e. a partially applied plus function).

The problem with **Either**'s **<*>** is that **it doesn't accumulate the errors in its Left err arguments**.

When passed a Right(a -> b), e.g. Right((+) 1), and a Right a, e.g. Right(2), **<*>** applies the function a -> b to the a, producing a Right b, i.e. Right(3). That's fine.

If the first argument of **<*>** is a **Left err** then the operator just returns that argument.

If the first argument of **<*>** is a Right(a -> b) then the operator maps function a->b onto its second argument, so if the second argument happens to be a **Left err**, then the operator ends up returning that **Left err**.

So we see that **when either or both of the arguments of <*> is a Left err then the operator returns a Left err, either the only one it has been passed or the first one it has been passed. In the latter case, <u>there is no notion of combining two Left err arguments into a result Left err that somehow accumulates the values in both Left err arguments</u>**.

@philip_schwarz

```
*Main> Right((+) 1) <*> Right(2)
Right 3

*Main> Right((+) 1) <*> Left("bang")
Left "bang"

*Main> Left("boom") <*> Right(2)
Left "boom"

*Main> Left("boom") <*> Left("bang")
Left "boom"

*Main>
```

```
instance Applicative (Either e) where
    pure          = Right
    Left  e <*> _ = Left e
    Right f <*> r = fmap f r
```

We want to replace Either with Validation, which is an Applicative whose <*> operator _does_ accumulate the errors in its arguments. Validation is defined as follows:

```
data Validation err a = Failure err | Success a
```

So the first thing we have to do is replace this

```
Either Error Username // Left Error | Right Username
Either Error Password // Left Error | Right Password
Either Error User     // Left Error | Right User
```

with this

```
Validation Error Username // Failure Error | Success Username
Validation Error Password // Failure Error | Success Password
Validation Error User     // Failure Error | Success User
```

Next, **what do we mean when we say that Validation's <*> accumulates errors in its arguments? We mean that unlike Either's <*>, when both of the arguments of Validation's <*> are failures, then <*> combines the errors in those failures. e.g if we pass <*> a Failure("boom") and a Failure("bang") then it returns Failure("boombang") !!!**

But how does Validation know how to combine "boom" and "bang" into "boombang"? Because Validation is an Applicative that requires a Semigroup to exist for the errors in its failures:

```
instance Semigroup err => Applicative (Validation err)
```

In the above example, the errors are strings, which are lists of characters, and there is a **semigroup** for lists, whose **combine operator** is defined as string **concatenation**.

```
class Semigroup a where
(<>) :: a -> a -> a
```

```
instance Semigroup [a] where
  (<>) = (++)
```

So "boom" and "bang" can be **combined** into "boombang" using the list **Semigroup**'s **<> operator** (mappend).

```
*Main> Right((+) 1) <*> Right(2)
Right 3

*Main> Right((+) 1) <*> Left("bang")
Left "bang"

*Main> Left("boom") <*> Right(2)
Left "boom"

*Main> Left("boom") <*> Left("bang")
Left "boom"
```

```
*Main> Success((+) 1) <*> Success(2)
Success 3

*Main> Success((+) 1) <*> Failure("bang")
Failure "bang"

*Main> Failure("boom") <*> Success(2)
Failure "boom"

*Main> Failure("boom") <*> Failure("bang")
Failure "boombang"
```

```
*Main> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]

*Main> "boom" ++ "bang"
"boombang"

*Main> "boom" <> "bang"
"boombang"
```

In our case, the value in the **Validation** failures is not a plain string, but rather, an **Error** wrapping a string:

```
newtype Error = Error String
  deriving Show
```

We need to define a **semigroup** for **Error**, so that **Validation Error a** can combine **Error** values.

But we don’t want accumulation of errors to mean concatenation of error messages. E.g. if we have two error messages “foo” and ”bar”, we don’t want their combination to be “foobar”.

So the authors refactor **Error** to wrap a list of error messages rather than a single error message:

```
newtype Error = Error [String]
deriving Show
```

They then define a **Semigroup** for **Error** whose **combine operator <>** (mappend) **concatenates** the error lists they wrap:

```
instance Semigroup Error where
Error xs <> Error ys = Error (xs ++ ys)
```

So now **combining two errors results in an error whose error message list is a combination of the error message lists of the two errors**.

```
*Main> Error(["snap"]) <> Error(["crackle","pop"])
Error ["snap","crackle","pop"]
```

and **passing two failures to Validation’s <*> operator results in a failure whose error is the combination of the errors of the two failures**:

```
*Main> Failure(Error(["snap"])) <*> Failure(Error(["crackle","pop"]))
Failure (Error ["snap","crackle","pop"])
```

Next, we are unhappy with the way our `validatePassword` function works.

Because it uses **>>=** (**bind**, i.e. **flatMap** in **Scala**), which **short-circuits** when its first argument is a **Left err,** it doesn't accumulate the errors in its **Left err** arguments.

```
validatePassword :: Password -> Either Error
Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```
*Main> Right(2) >>= \x -> Right(x + 3) >>= \y -> Right(x * y)
Right 10

*Main> Left("boom") >>= \x -> Right(x + 3) >>= \y -> Right(y * 2)
Left "boom"

*Main> Right(2) >>= \x -> Left("bang") >>= \y -> Right(y * 2)
Left "bang"

*Main> Left("boom") >>= \x -> Left("bang") >>= \y -> Right(y * 2)
Left "boom"
```

The authors of **Finding Success (and Failure) in Haskell** address this by **introducing another Applicative operator** (see below).

The **Applicative typeclass** **has a pair of operators that we like to call** **left- and right-facing bird**, but some people call them **left and right shark**. Either way, **the point is they eat one of your values**.

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

These effectively let you sequence function applications, discarding either the first return value or the second one.

The thing that's pertinent for us now is **they do not eat any effects that are part of the f**. Remember, when we talk about the **Applicative** instance for **Validation**, it's really the **Applicative** instance for **Validation err** because **Validation** must be applied to its first argument, so **our f is Validation Error, and that instance lets us accumulate Error values via a Semigroup instance (concatenation)**.

@chris_martin
@argumatronic

Let’s see the **Applicative** ***>** operator in action. While the ***>** operator of the **Either Applicative** does not combine the contents of two **Left** values, the ***>** operator of the **Validation Applicative** does:

```
*Main> Right(2) *> Right(3)
Right 3

*Main> Left("boom") *> Right(3)
Left "boom"

*Main> Right(2) *> Left("bang")
Left "bang"

*Main> Left("boom") *> Left("bang")
Left "boom"
```

```
*Main> Success(2) *> Success(3)
Success 3

*Main> Failure("boom") *> Success(3)
Failure "boom"

*Main> Success(2) *> Failure("bang")
Failure "bang"

*Main> Failure("boom") *> Failure("bang")
Failure "boombang"
```

And because **Error** has been redefined to be a **Semigroup** and wrap a list of error messages, the ***>** of the **Validation Applicative** combines the contents of two **Error** values:

```
*Main> Success(2) *> Success(3)
Success 3

*Main> Failure(Error ["boom"]) *> Success(3)
Failure (Error ["boom"])

*Main> Success(2) *> Failure(Error ["bang"])
Failure (Error ["bang"])

*Main> Failure(Error ["boom"]) *> Failure(Error ["bang"])
Failure (Error ["boom","bang"])
```

@philip_schwarz

The authors of **Finding Success (and Failure) in Haskell** rewrite the `validatePassword` function as follows:

```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
      >>= requireAlphaNum
      >>= checkPasswordLength
```

```haskell
validatePassword :: Password -> Validation Error Password
validatePassword (Password password) =
  case (cleanWhitespace password) of
    Failure err       -> Failure err
    Success password2 -> requireAlphaNum password2
                          *> checkPasswordLength password2
```

similarly for the `validateUsername` function.

Here is how `requireAlphaNum password2 *> checkPasswordLength password2` works:

```
*Main> Success(Password("fredsmith")) *> Success(Password("fredsmith"))
Success (Password "fredsmith")

*Main> Failure(Error ["boom"]) *> Success(Password("fredsmith"))
Failure (Error ["boom"])

*Main> Success(Password("fredsmith")) *> Failure(Error ["bang"])
Failure (Error ["bang"])

*Main> Failure(Error ["boom"]) *> Failure(Error ["bang"])
Failure (Error ["boom","bang"])
```

Similarly for the equivalent section of the `validateUsername` function.

In the next slide we look at how the behaviour of the `makeUser` function changes with the switch from **Either** to **Validation**.

Here is how **makeUser** works following the switch from the **Either Applicative** to the to **Validation Applicative**

```
*Main> makeUser (Username "fredsmith") (Password "pa22w0rd")
Success (User (Username "fredsmith") (Password "pa22w0rd"))

*Main> makeUser (Username "extremelylongusername") (Password "pa22w0rd")
Failure (Error ["Your username cannot be longer than 15 characters."])

*Main> makeUser (Username "fredsmith") (Password "password with spaces")
Failure (Error ["Cannot contain white space or special characters."])

*Main> makeUser (Username "extremelylongusername") (Password "password with spaces")
Failure (Error ["Your username cannot be longer than 15 characters.","Cannot contain white space or special characters."])

*Main>
```

The problem with the original program is solved: **when both username and password are invalid then makeUser reports all the validation errors it has encountered**

```
*Main> makeUser (Username "fredsmith") (Password "pa22w0rd")
Right (User (Username "fredsmith") (Password "pa22w0rd"))

*Main> makeUser (Username "extremelylongusername") (Password "pa22w0rd")
Left (Error "Your username cannot be longer than 15 characters.")

*Main> makeUser (Username "fredsmith") (Password "password with spaces")
Left (Error "Cannot contain white space or special characters.")

*Main> makeUser (Username "extremelylongusername") (Password "password with spaces")
Left (Error "Your username cannot be longer than 15 characters.")

*Main>
```

And here is how **makeUser** used to work before switching from **Either** to **Validation**.

```haskell
newtype Username = Username String
  deriving Show
```

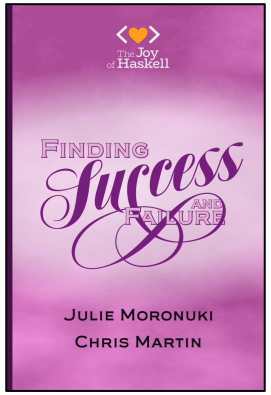
```haskell
newtype Password = Password String
  deriving Show
```

```haskell
newtype Error = Error [String]
  deriving Show
```

```haskell
instance Semigroup Error where
  Error xs <> Error ys = Error (xs ++ ys)
```

```haskell
checkUsernameLength :: String -> Validation Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Failure (Error "Your username cannot be \
                              \longer than 15 characters.")
      False -> Success (Username username)
```

```haskell
checkPasswordLength :: String -> Validation Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Failure (Error "Your password cannot be \
                              \longer than 20 characters.")
      False -> Success (Password password)
```




```haskell
requireAlphaNum :: String -> Validation Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Failure "Your password cannot contain \
                       \white space or special characters."
      True  -> Success input
```

```haskell
cleanWhitespace :: String -> Validation Error String
cleanWhitespace "" = Failure (Error "Your password cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Success (x : xs)
```

```haskell
validateUsername :: Username -> Validation Error Username
validatePassword (Username username) =
  case (cleanWhitespace username) of
    Failure err       -> Failure err
    Success username2 -> requireAlphaNum username2
                          *> checkUsernameLength username2
```

```haskell
validatePassword :: Password -> Validation Error Password
validatePassword (Password password) =
  case (cleanWhitespace password) of
    Failure err       -> Failure err
    Success password2 -> requireAlphaNum password2
                          *> checkPasswordLength password2
```

```haskell
makeUser :: Username -> Password -> Validation Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

```haskell
data User = User Username Password
  deriving Show
```

@chris_martin
@argumatronic

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

Here is how the code looks like after the switch from the **Either Applicative** to the **Validation Applicative**

```haskell
class Semigroup a where         instance Semigroup [a] where
 (<>) :: a -> a -> a              (<>) = (++)

instance Semigroup err => Applicative (Validation err)

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

Now let's look at the **Scala** equivalent of the refactoring.

In the next three slides we'll look at the **Scala** equivalent of the code as it was before the refactoring.

```haskell
newtype Username = Username String
  deriving Show

newtype Password = Password String
  deriving Show

data User = User Username Password
  deriving Show

newtype Error = Error String
  deriving Show
```

```haskell
checkUsernameLength :: String -> Either Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Left (Error "Your username cannot be \
                     \longer than 15 characters.")
      False -> Right (Username username)
```

```haskell
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Left (Error "Your password cannot be \
                     \longer than 20 characters.")
      False -> Right (Password password)
```

```haskell
requireAlphaNum :: String -> Either Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Left "Cannot contain \
                 \white space or special characters."
      True  -> Right input
```

```haskell
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Right (x : xs)
```

```haskell
validateUsername :: Username -> Either Error Username
validateUsername (Username username) =
  cleanWhitespace username
    >>= requireAlphaNum
    >>= checkUsernameLength
```

Haskell

```haskell
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
   cleanWhitespace password
     >>= requireAlphaNum
     >>= checkPasswordLength
```

Nothing noteworthy here. The next slide is more interesting

```scala
case class Username(username: String)      case class Password(password:String)

case class User(username: Username, password: Password)

case class Error(error:String)
```

```scala
def checkUsernameLength(username: String): Either[Error, Username] =
  username.length > 15 match {
    case true  => Left(Error("Your username cannot be " +
      "longer than 15 characters."))
    case false => Right(Username(username))
  }
```

```scala
def checkPasswordLength(password: String): Either[Error, Password] =
  password.length > 20 match {
    case true  => Left(Error("Your password cannot be " +
      "longer than 20 characters."))
    case false => Right(Password(password))
  }
```

```scala
def requireAlphaNum(password: String): Either[Error, String] =
  password.forall(_.isLetterOrDigit) match {
    case false  => Left(Error("Cannot contain white " +
      "space or special characters."))
    case true   => Right(password)
  }
```

```scala
def cleanWhitespace(password:String): Either[Error, String] =
  password.dropWhile(_.isWhitespace) match {
    case pwd if pwd.isEmpty => Left(Error("Cannot be empty."))
    case pwd                => Right(pwd)
  }
```

```scala
def validateUsername(username: Username): Either[Error,Username] = username match {
  case Username(username) =>
    cleanWhitespace(username)
      .flatMap(requireAlphaNum)
      .flatMap(checkUsernameLength)
}
```

Scala

```scala
def validatePassword(password: Password): Either[Error,Password] = password match {
  case Password(pwd) =>
    cleanWhitespace(pwd)
      .flatMap(requireAlphaNum)
      .flatMap(checkPasswordLength)
}
```

Scala doesn’t have an **Applicative typeclass**, so we define it ourselves in terms of **unit** and **<*>**. We then define an **Applicative** instance for **Either**.

We deliberately implement **Either’s <*>** so it behaves the same way as in **Haskell**, i.e. so that when **<*>** is passed one or more **Left** values it just returns the first or only value it is passed. i.e. when it is passed two **Left** values, it does not attempt to combine the contents of the two values.

```
scala>  main.unsafeRunSync
Please enter a username.
extremelylongusername
Please enter a password.
extremelylongpassword
Left(Error(Your username cannot be longer than 15 characters.))
scala>
```

@philip_schwarz

```scala
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}

trait Applicative[F[_]] extends Functor[F] {
  def <*>[A,B](fab: F[A => B],fa: F[A]): F[B]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] = <*>(unit(f),fa)
}

type Validation[A] = Either[Error, A]

val eitherApplicative = new Applicative[Validation] {

  def <*>[A,B](fab: Validation[A => B],fa: Validation[A]): Validation[B] =
    (fab, fa) match {
      case (Right(ab), Right(a)) => Right(ab(a))
      case (Left(err), _) => Left(err)
      case (_, Left(err)) => Left(err)
    }

  def unit[A](a: => A): Validation[A] = Right(a)
}
```

In **Haskell** every function takes only one parameter. In **Scala**, we have to **curry** User so it takes a username and returns a function that takes a password.

```haskell
makeUser :: Username -> Password -> Either Error User
makeUser name password =
  User <$> validateUsername name
       <*> validatePassword password
```

Haskell

```scala
import eitherApplicative._

def makeUser(name: Username, password: Password): Either[Error, User] =
  <*>( map(validateUsername(name))(User.curried),
       validatePassword(password) )
```

Scala

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

```scala
val main: IO[Unit] =
  for {
    _   <- print("Please enter a username.\n")
    usr <- getLine map Username
    _   <- print("Please enter a password.\n")
    pwd <- getLine map Password
    _   <- print(makeUser(usr,pwd).toString)
  } yield ()
```

```scala
import cats.effect.IO

def getLine =
  IO { scala.io.StdIn.readLine }
def print(s: String): IO[Unit] =
  IO { scala.Predef.print(s) }
```

Because it is us who are implementing <*>, instead of implementing it like this

```
def <*>[A,B](fab: Validation[A => B],fa: Validation[A]): Validation[B] =
  (fab, fa) match {
    case (Right(ab), Right(a)) => Right(ab(a))
    case (Left(err), _) => Left(err)
    case (_, Left(err)) => Left(err)
  }
```

for reference:

```
case class Error(error:String)

type Validation[A] = Either[Error, A]
```

we could, if we wanted to, implement it like this, which would be **one way to get it to combine Left values**:

```
def <*>[A,B](fab: Validation[A => B],fa: Validation[A]): Validation[B] =
  (fab, fa) match {
    case (Right(ab), Right(a)) => Right(ab(a))
    case (Left(Error(err1)), Left(Error(err2))) => Left(Error(err1 + err2))
    case (Left(err), _) => Left(err)
    case (_, Left(err)) => Left(err)
  }
```

string concatention

```
scala> main.unsafeRunSync
Please enter a username.
extremelylongusername
Please enter a password.
extremelylongpassword
Left(Error(Your username cannot be longer than 15 characters.Your password cannot be longer than 20 characters.))
scala>
```

two combined (concatented) error message strings

Now back to the refactoring. In the next thre slides we'll see what the **Scala** code looks like at the end of the refactoring.

```haskell
newtype Username = Username String
  deriving Show

newtype Password = Password String
  deriving Show

data User = User Username Password
  deriving Show

newtype Error = Error [String]
  deriving Show
```

```scala
case class Username(username: String)   case class Password(password:String)

case class User(username: Username, password: Password)

case class Error(error:List[String])
```

Error now contains a list of messages

```haskell
checkUsernameLength :: String -> Validation Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Failure (Error "Your username cannot be \
                               \longer than 15 characters.")
      False -> Success (Username username)
```

Nothing noteworthy other than the switch from **Either** to **Validation**.

The next slide is more interesting

```scala
def checkUsernameLength(username: String): Validation[Error, Username] =
  username.length > 15 match {
    case true  => Failure(Error(List("Your username cannot be " +
                                     "longer than 15 characters.")))
    case false => Success(Username(username))
  }
```

```haskell
checkPasswordLength :: String -> Validation Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Failure (Error "Your password cannot be \
                               \longer than 20 characters.")
      False -> Success (Password password)
```

```scala
def checkPasswordLength(password: String): Validation[Error, Password] =
  password.length > 20 match {
    case true  => Failure(Error(List("Your password cannot be " +
                                     "longer than 20 characters.")))
    case false => Success(Password(password))
  }
```

```haskell
requireAlphaNum :: String -> Validation Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Failure "Your password cannot contain \
                       \white space or special characters."
      True  -> Success input
```

```scala
def requireAlphaNum(password: String): Validation[Error, String] =
  password.forall(_.isLetterOrDigit) match {
    case false  => Failure(Error(List("Cannot contain white " +
                                      "space or special characters.")))
    case true   => Success(password)
  }
```

```haskell
cleanWhitespace :: String -> Validation Error String
cleanWhitespace "" = Failure (Error "Your password cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Success (x : xs)
```

Haskell

```scala
def cleanWhitespace(password:String): Validation[Error, String] =
  password.dropWhile(_.isWhitespace) match {
    case pwd if pwd.isEmpty => Failure(Error(List("Cannot be empty.")))
    case pwd                => Success(pwd)
  }
```

Scala

Haskell

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

instance Semigroup [a] where
  (<>) = (++)

instance Semigroup Error where
  Error xs <> Error ys = Error (xs ++ ys)

instance Semigroup err => Applicative (Validation err)

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

While before the refactoring, **Validation** was just an alias

```scala
type Validation[A] = Either[Error, A]
```

now it is a sum type **Validation**[+E, +A] with a **Failure** and a **Success**, and with **Failure** containing an error of type E (see bottom left of slide).

We define an **Applicative** instance for **Validation**[+E, +A]. The instance has an implicit **Semigroup** for the **Validation**'s error type E so that the instance's **<*>** function can combine the contents of two failures.

The **Applicative** typeclass now also has a **right-shark** function and the **Validation** instance of the **Applicative** implements this so that it also combines the contents of two failures.

Scala

```scala
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}

trait Semigroup[A] {
  def <>(lhs: A, rhs: A): A
}

implicit val errorSemigroup: Semigroup[Error] =
  new Semigroup[Error] {
    def <>(lhs: Error, rhs: Error): Error =
      Error(lhs.error ++ rhs.error)
  }

trait Applicative[F[_]] extends Functor[F] {
  def <*>[A,B](fab: F[A => B],fa: F[A]): F[B]
  def *>[A,B](fa: F[A],fb: F[B]): F[B]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] =
    <*>(unit(f),fa)
}

sealed trait Validation[+E, +A]
case class Failure[E](error: E) extends Validation[E, Nothing]
case class Success[A](a: A) extends Validation[Nothing, A]
```

We defined **Semigroup** and declared an implicit instance of it for **Error**, which gets used by the **Validation Applicative**.

Applicative now has a **right-shark** function

Validation is now a sum type whose **Failure** contains an error

Scala

```scala
def validationApplicative[E](implicit sg:Semigroup[E]):
  Applicative[λ[α => Validation[E,α]]] =

  new Applicative[λ[α => Validation[E,α]]] {

    def unit[A](a: => A) = Success(a)

    def <*>[A,B](fab: Validation[E,A => B], fa: Validation[E,A]): Validation[E,B] =
      (fab, fa) match {
        case (Success(ab), Success(a)) => Success(ab(a))
        case (Failure(err1), Failure(err2)) => Failure(sg.<>(err1,err2))
        case (Failure(err), _) => Failure(err)
        case (_, Failure(err)) => Failure(err)
      }

    def *>[A,B](fa: Validation[E,A], fb: Validation[E,B]): Validation[E,B] =
      (fa, fb) match {
        case (Failure(err1), Failure(err2)) => Failure(sg.<>(err1,err2))
        case _ => fb
      }
  }

val errorValidationApplicative = validationApplicative[Error]
import errorValidationApplicative._
```

After declaring the instance (the implicit Semigroup[Error] is being passed in here) we import its operators, e.g. **<*>** and ***>**, so that functions on the next slide can use them.

```haskell
validateUsername :: Username -> Validation Error Username
validateUsername (Username username) =
  case (cleanWhitespace username) of
    Failure err       -> Failure err
    Success username2 -> requireAlphaNum username2
                         *> checkUsernameLength username2
```

```scala
def validateUsername(username: Username): Validation[Error, Username] = username match {
  case Username(username) =>
     cleanWhitespace(username) match {
      case Failure(err)       => Failure(err)
      case Success(username2) => *>(requireAlphaNum(username2),
                                    checkUsernameLength(username2))
    }
}
```

```haskell
validatePassword :: Password -> Validation Error Password
validatePassword (Password password) =
  case (cleanWhitespace password) of
    Failure err       -> Failure err
    Success password2 -> requireAlphaNum password2
                         *> checkPasswordLength password2
```

```scala
def validatePassword(password: Password): Validation[Error, Password] = password match {
  case Password(pwd) =>
    cleanWhitespace(pwd) match {
      case Failure(err)  => Failure(err)
      case Success(pwd2) => *>(requireAlphaNum(pwd2),
                               checkPasswordLength(pwd2))
    }
}
```

The **right-shark** function in action

```haskell
makeUser :: Username -> Password -> Validation Error User
makeUser name password =
User <$> validateUsername name
     <*> validatePassword password
```

Haskell

```scala
def makeUser(name: Username, password: Password): Validation[Error, User] =
  <*>(map(validateUsername(name))(User.curried),
      validatePassword(password) )
```

Scala

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

```scala
val main: IO[Unit] =
  for {
    _   <- print("Please enter a username.\n")
    usr <- getLine map Username
    _   <- print("Please enter a password.\n")
    pwd <- getLine map Password
    _   <- print(makeUser(usr,pwd).toString)
  } yield ()
```

```scala
import cats.effect.IO

def getLine =
  IO { scala.io.StdIn.readLine }
def print(s: String): IO[Unit] =
  IO { scala.Predef.print(s) }
```

By the way, if you look back at the signatures of **<*>** and ***>** you'll see that rather than taking one argument at a time, they both take two arguments in one go. I did this purely because it makes for a tidier call site (e.g. by avoiding the need for a cast in one case), but it is not strictly necessary: I could have left the signatures alone.

Let’s have a go at running the **Scala** version of the refactored program.

See how if we feed it a username and a password that each violate two validation constraints then the program returns a **Failure** whose **Error** containts a list of four error messages.

The two singleton error-message lists for username get combined by ***>** into a single two error-message list. Similarly for password. This pair of two error-message lists then gets combined by **<*>** into a single four error-message list.

```
scala> main.unsafeRunSync
Please enter a username.
very long username with spaces
Please enter a password.
very long password with spaces
Failure(
  Error(
    List(Cannot contain white space or special characters.,
         Your username cannot be longer than 15 characters.,
         Cannot contain white space or special characters.,
         Your password cannot be longer than 20 characters.)))
scala>
```

*> combined by (username error messages)

*> combined by (password error messages)

combined by <*>

You’ll have noticed that the error messages for white space or special characters are not great in that they don’t say whether they apply to a username or to password. While that is easily fixed I have not yet bothered doing that in this slide deck.

@philip_schwarz

to be continued in Part III