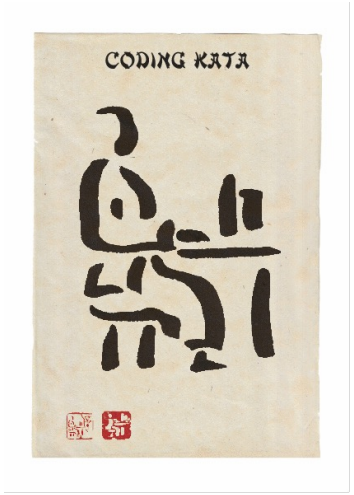
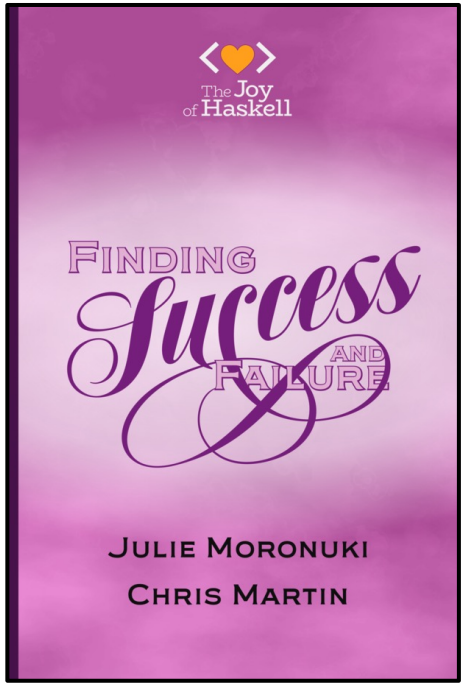


Hand Rolled Applicative User Validation Code Kata



Haskell

User Validation Program

```
<*> AKA tie-fighter, apply, ap
 *> AKA right-facing bird, right shark
```

Scala

slides by @philip_schwarz FP Illuminated http://fpilluminated.com/

@philip_schwarz

Could you use a simple piece of **Scala** validation code (granted, a very simplistic one too!) that you can rewrite, now and again, to refresh your basic understanding of **Applicative** operators **<*>**, **<***, ***>**?

The goal is not to write perfect code showcasing validation, but rather, to provide a small, rough-and ready exercise to reinforce your muscle-memory.

Despite its grandiose-sounding title, this deck consists of just three slides showing the **Scala 3** code to be rewritten whenever the details of the operators begin to fade away.

The code is my rough and ready translation of a **Haskell** user-validation program found in a book called **Finding Success (and Failure) in Haskell - Fall in love with applicative functors**.

```scala
trait Semigroup[A]:
  extension (lhs: A)
    def <>(rhs: A): A

trait Functor[F[_]]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B]

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

```scala
case class Username(username: String)
case class Password(password: String)
case class User(username: Username, password: Password)
case class Error(errors: List[String])

enum Validation[+E, +A]:
  case Failure(errors: E)
  case Success(a: A)
```

```scala
given Semigroup[Error] = new Semigroup[Error]:
  extension (lhs: Error)
    def <>(rhs: Error): Error =
      Error(lhs.errors ++ rhs.errors)

given (using Semigroup[Error]): Applicative[[X] =>> Validation[Error, X]] =
  new Applicative[[X] =>> Validation[Error, X]]:

    override def unit[A](a: => A): Validation[Error, A] = Success(a)

    extension [A, B](fab: Validation[Error, A => B])
      override def <*>(fa: Validation[Error, A]): Validation[Error, B] = (fab, fa) match
        case (Success(ab), Success(a)) => Success(ab(a))
        case (Failure(e1), Failure(e2)) => Failure(e1 <> e2)
        case (Failure(e), _) => Failure(e)
        case (_, Failure(e)) => Failure(e)


    extension [A, B](fa: Validation[Error, A])
      override def *>(fb: Validation[Error, B]): Validation[Error, B] = (fa, fb) match
        case (Failure(e1), Failure(e2)) => Failure(e1 <> e2)
        case (Failure(e), _) => Failure(e)
        case _ => fb

      override def <*(fb: Validation[Error, B]): Validation[Error, A] = (fa, fb) match
        case (Failure(e1), Failure(e2)) => Failure(e1 <> e2)
        case (_, Failure(e)) => Failure(e)
        case _ => fa
```

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

```scala
def validateUsername(username: Username): Validation[Error, Username] = username match
  case Username(usr) =>
    cleanWhiteSpace(usr) match
      case Failure(error) => Failure(error)
      case Success(cleanedUsername) => requireAlphaNumeric(cleanedUsername) *> checkUsernameLength(cleanedUsername)

def validatePassword(password: Password): Validation[Error, Password] = password match
  case Password(pwd) =>
    cleanWhiteSpace(pwd) match
      case Failure(error) => Failure(error)
      case Success(cleanedPassword) => requireAlphaNumeric(cleanedPassword) *> checkPasswordLength(cleanedPassword)
```

```scala
def checkUsernameLength(username: String): Validation[Error, Username] =
  if username.length <= 15
  then Success(Username(username))
  else Failure(Error(List("Your username cannot be longer than 15 characters.")))

def checkPasswordLength(password: String): Validation[Error, Password] =
  if password.length <= 20
  then Success(Password(password))
  else Failure(Error(List("Your password cannot be longer than 20 characters.")))

def requireAlphaNumeric(input: String): Validation[Error, String] =
  if input.forall(_.isLetterOrDigit)
  then Success(input)
  else Failure(Error(List("Cannot contain whitespace or special characters.")))

def cleanWhiteSpace(input: String): Validation[Error, String] =
  input.dropWhile(_.isWhitespace) match
    case ""      => Failure(Error(List("Cannot be empty.")))
    case cleaned => Success(cleaned)
```

```
@main
def main(): Unit =

  assert(makeUser(Username("FredSmith"), Password("pa22w0rd")) == Success(User(Username("FredSmith"), Password("pa22w0rd"))))

  assert(makeUser(Username("Fred$Smith"), Password("pa22w0rd")) == Failure(Error(List("Cannot contain whitespace or special characters."))))

  assert(makeUser(Username("FredSmith"), Password("pa22w*rd")) == Failure(Error(List("Cannot contain whitespace or special characters."))))

  assert(makeUser(Username("overlongusername"), Password("pa22w0rd")) == Failure(Error(List("Your username cannot be longer than 15 characters."))))

  assert(makeUser(Username("FredSmith"), Password("averyverylongpassword")) == Failure(Error(List("Your password cannot be longer than 20 characters."))))

  assert(

    makeUser(Username("overlong#username"), Password("averyverylongpassword"))
    ==
    Failure(Error(List(
      "Cannot contain whitespace or special characters.",
      "Your username cannot be longer than 15 characters.",
      "Your password cannot be longer than 20 characters.")))
  )
```

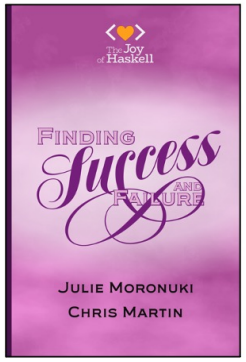
If you want to know more, check out the following deck

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

FP Iλλuminated http://fpilluminated.com/