# Functional Data Validation

Dave Gurnell, @davegurnell

# How to Think Functionally

Dave Gurnell, @davegurnell

underscore

# Why?

Everyone knows validation

Lots of FP concepts

The rabbit hole goes deep…

# What?

Functional building blocks

Composition

Lifting operations

# What?

Natural conclusions:

Monads

Applicative Functors

scalaz.Validation

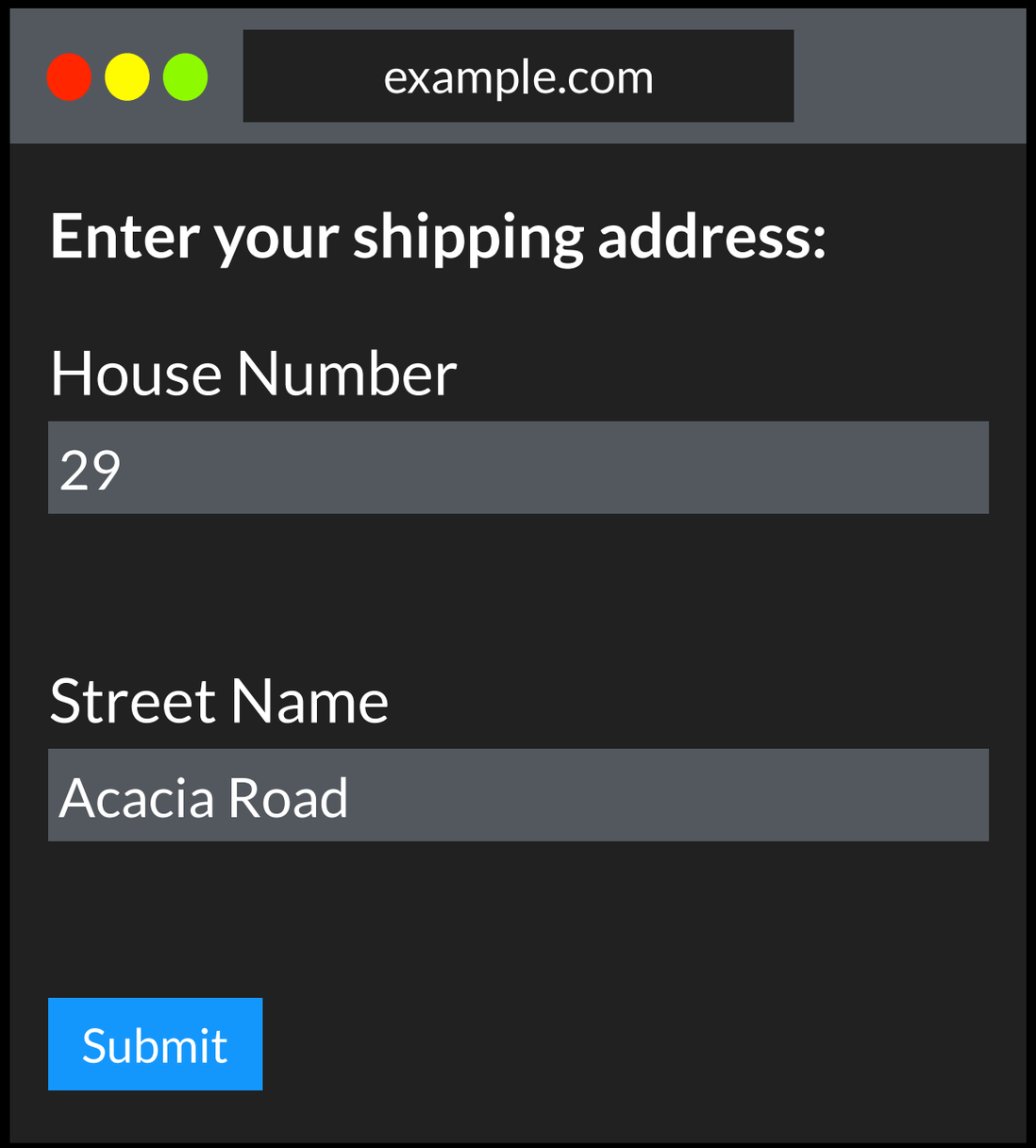
…but this is not a Scalaz talk!

Sprint #1

# Making the Rules

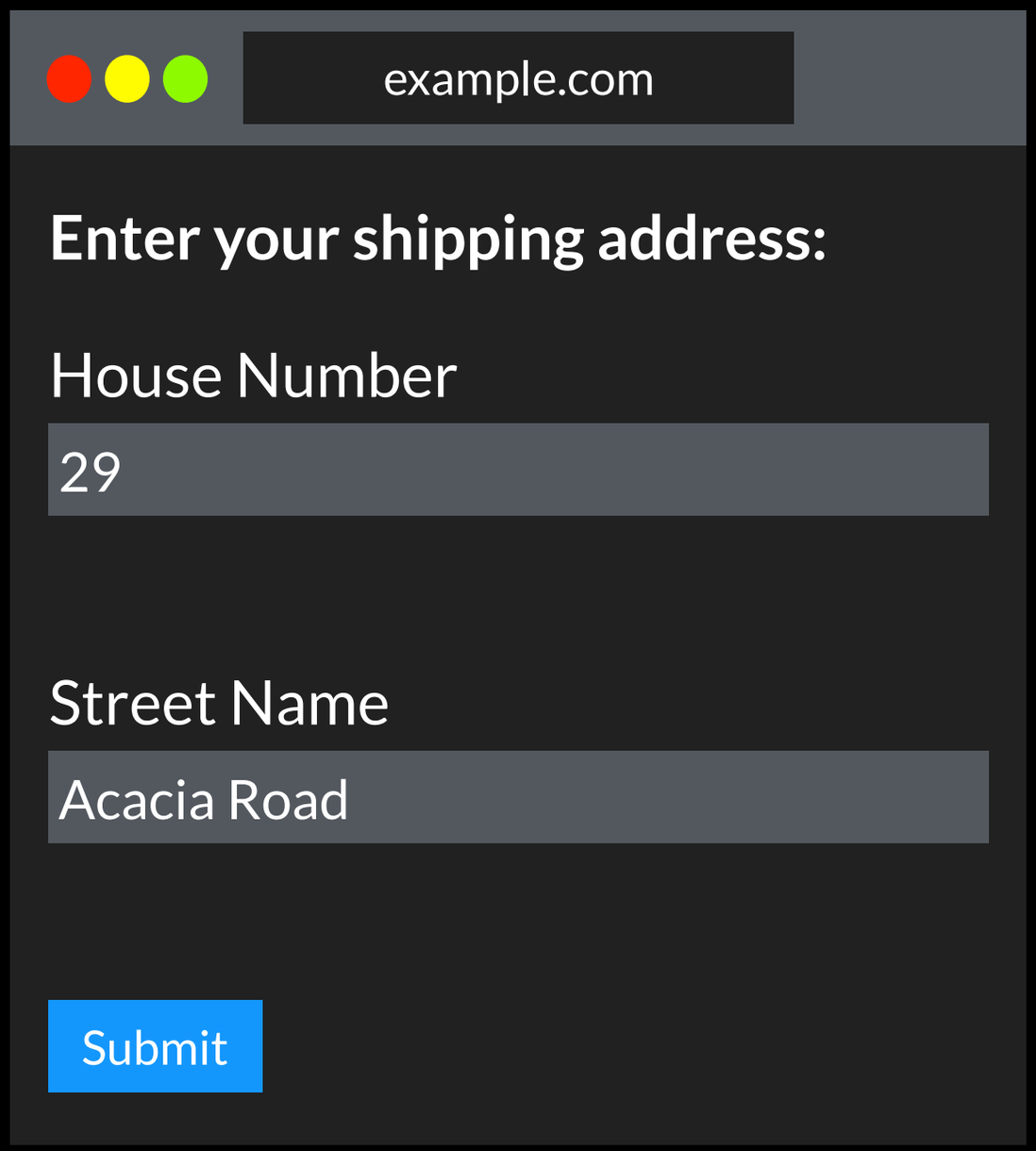
*in which we find a suitable model for validation*

# Goals




```
case class Address(
  number: Int,
  street: String)
```

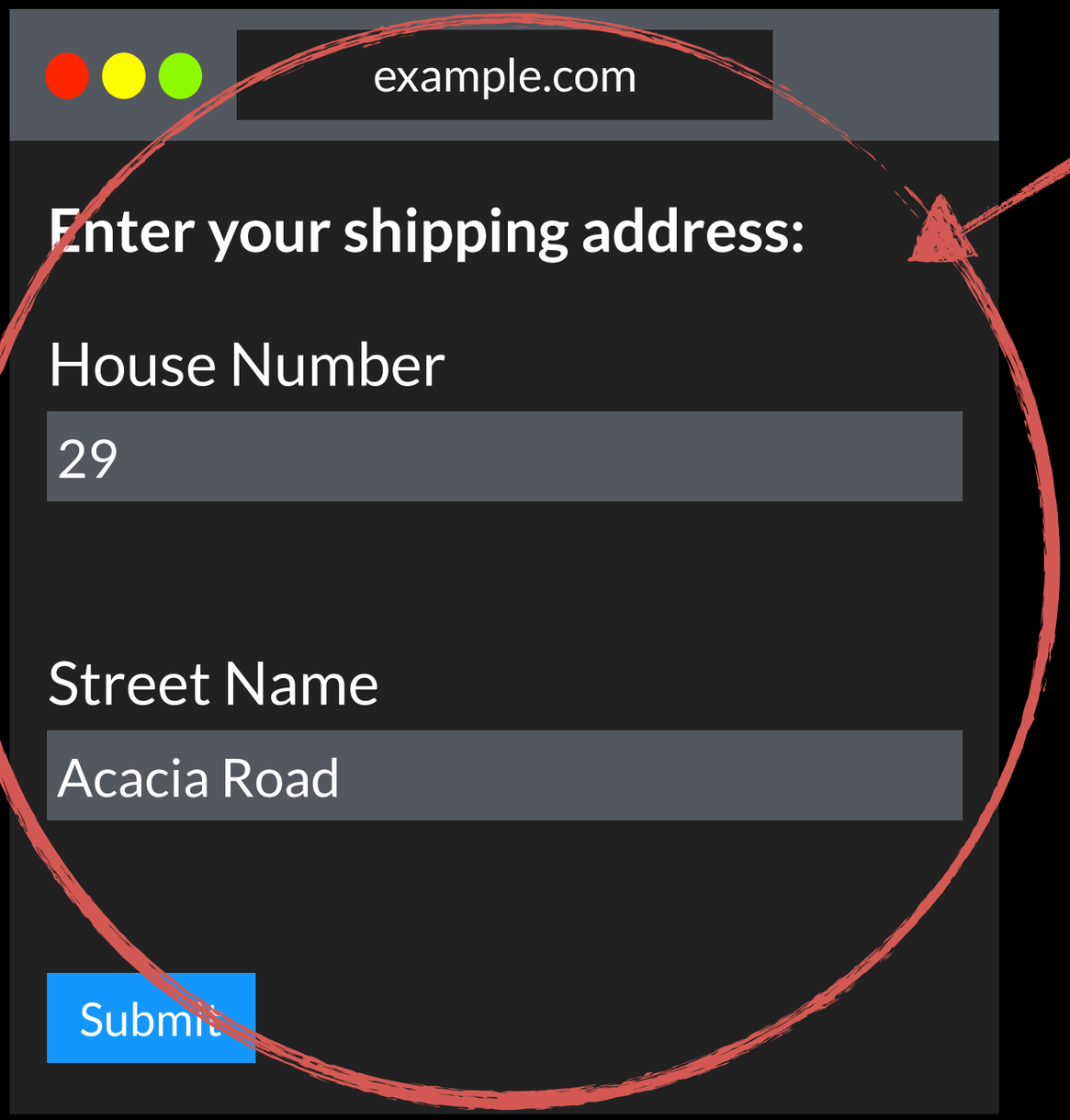
# Goals




```
case class Address(
  number: Int,
  street: String)
```

"number" is > 0
"street" is not empty

# Goals




"number" and "street" present
"number" is a valid integer

```
case class Address(
  number: Int,
  street: String)
```

"number" is > 0
"street" is not empty

# Goals

"number" and "street" present
"number" is a valid integer

example.com

**Enter your shipping address:**

House Number

abc

This field must be 1 or greater

Street Name

This field is required

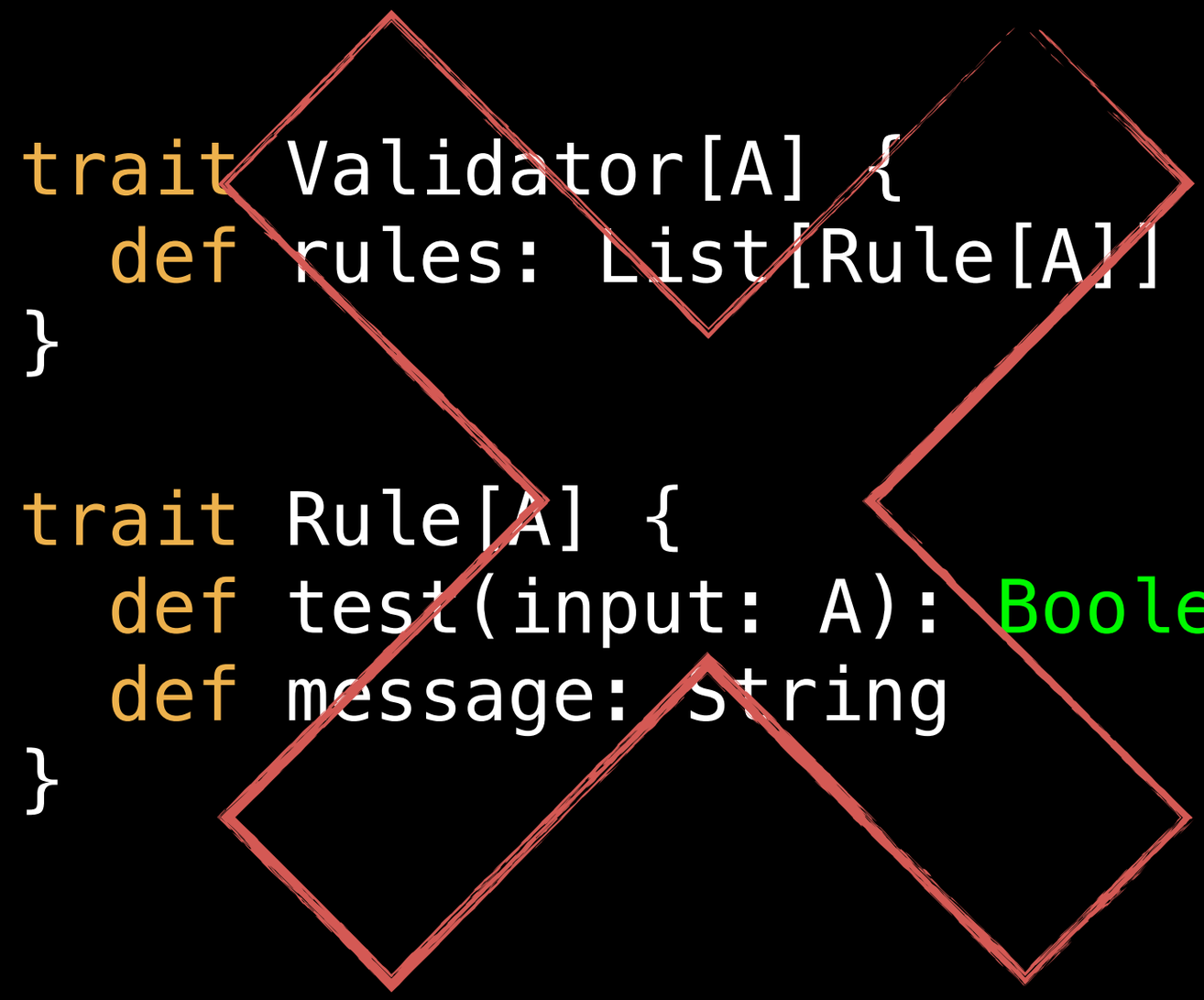
Submit

```
case class Address(
  number: Int,
  street: String)
```

"number" is > 0
"street" is not empty

```scala
trait Validator[A] {
  def rules: List[Rule[A]]
}
```

```scala
trait Rule[A] {
  def test(input: A): Boolean
  def message: String
}
```



```
trait Validator[A] {
  def rules: List[Rule[A]]
}

trait Rule[A] {
  def test(input: A): Boolean
  def message: String
}
```

```
type Rule[A] = A => Result
```

How do we combine rules?

How do we sequence code?

```
type Rule[A] = A => Result
```

How do we model binary rules, e.g. x >= y ?

What is this type?

A result is a success or a failure

Success indicates everything is OK

A failure reports any error messages

A result is a success or a failure

Success indicates everything is OK

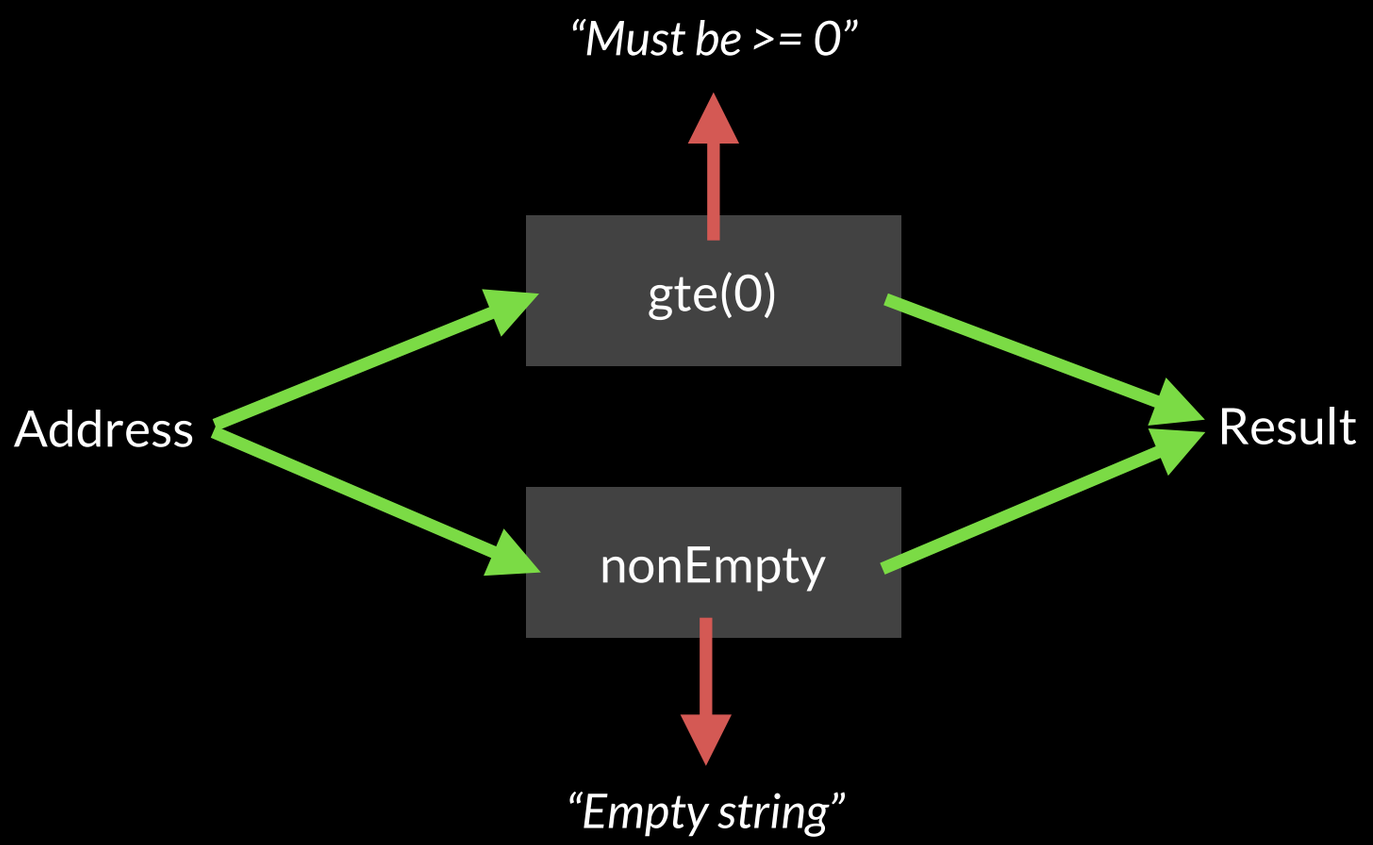
A failure reports any error messages

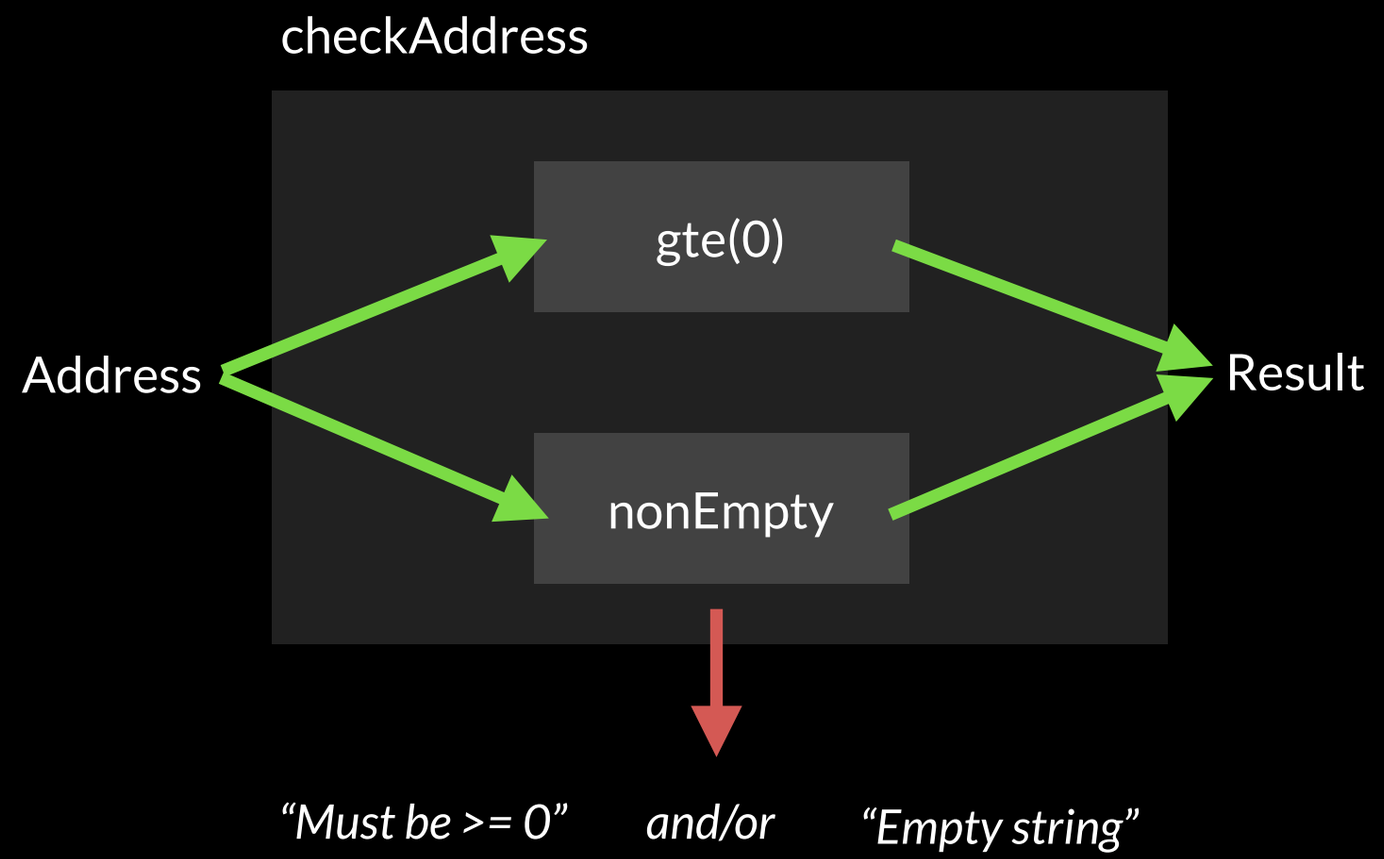
Algebraic data-type

```scala
sealed trait Result

final case object Pass
  extends Result

final case class Fail(messages: List[String])
  extends Result
```

```
val nonEmpty: Rule[String] =
  (value: String) =>
    if(value.isEmpty)
      Fail(List("Empty string"))
    else
      Pass
```

```scala
def gte(min: Int): Rule[Int] =
  (value: Int) =>
    if(value < min)
      Fail(List("Too small"))
    else
      Pass
```


"Must be >= 0"
gte(0)
Address
Result
nonEmpty
"Empty string"


checkAddress
gte(0)
Address
Result
nonEmpty
“Must be >= 0”
and/or
“Empty string”

```scala
val checkAddress: Rule[Address] =
  (address: Address) =>
  gte(1)(address.number) and
  nonEmpty(address.street)
```

```scala
val checkAddress: Rule[Address] =
  (address: Address) =>
    gte(1)(address.number) and
    nonEmpty(address.street)
```

```
result1.and(result2)
```

```
sealed trait Result {
  def and(that: Result): Result = ???
}
final case object Pass
  extends Result

final case class Fail(messages: List[String])
  extends Result
```

```
sealed trait Result {
  def and(that: Result): Result =
    ???
}
```

```
sealed trait Result {
  def and(that: Result): Result =
    this match {
      case Pass => ???
      case Fail(e) => ???


    }
}
```

```
sealed trait Result {
  def and(that: Result): Result =
    this match {
      case Pass => that match {
        case Pass    => ???
        case Fail(b) => ???
      }
      case Fail(a) => that match {
        case Pass    => ???
        case Fail(b) => ???
      }
    }
```

```scala
sealed trait Result {
  def and(that: Result): Result =
    this match {
      case Pass => that match {
        case Pass    => Pass
        case Fail(b) => Fail(b)
      }
      case Fail(a) => that match {
        case Pass    => Fail(a)
        case Fail(b) => Fail(a ++ b)
      }
    }
```

```scala
val checkAddress: Rule[Address] =
  (address: Address) =>
  gte(1)(address.number) and
  nonEmpty(address.street)
```

# End of Sprint #1

A rule is a function,
a result is an algebraic data type

Construct complex rules by:
composing simple rules (e.g. using *and*)
creating higher order functions

Write algorithms using structural recursion
(aka pattern matching)

Sprint #2

# Form and Function

*in which we build sequences of rules*

example.com

**Enter your shipping address:**

House Number

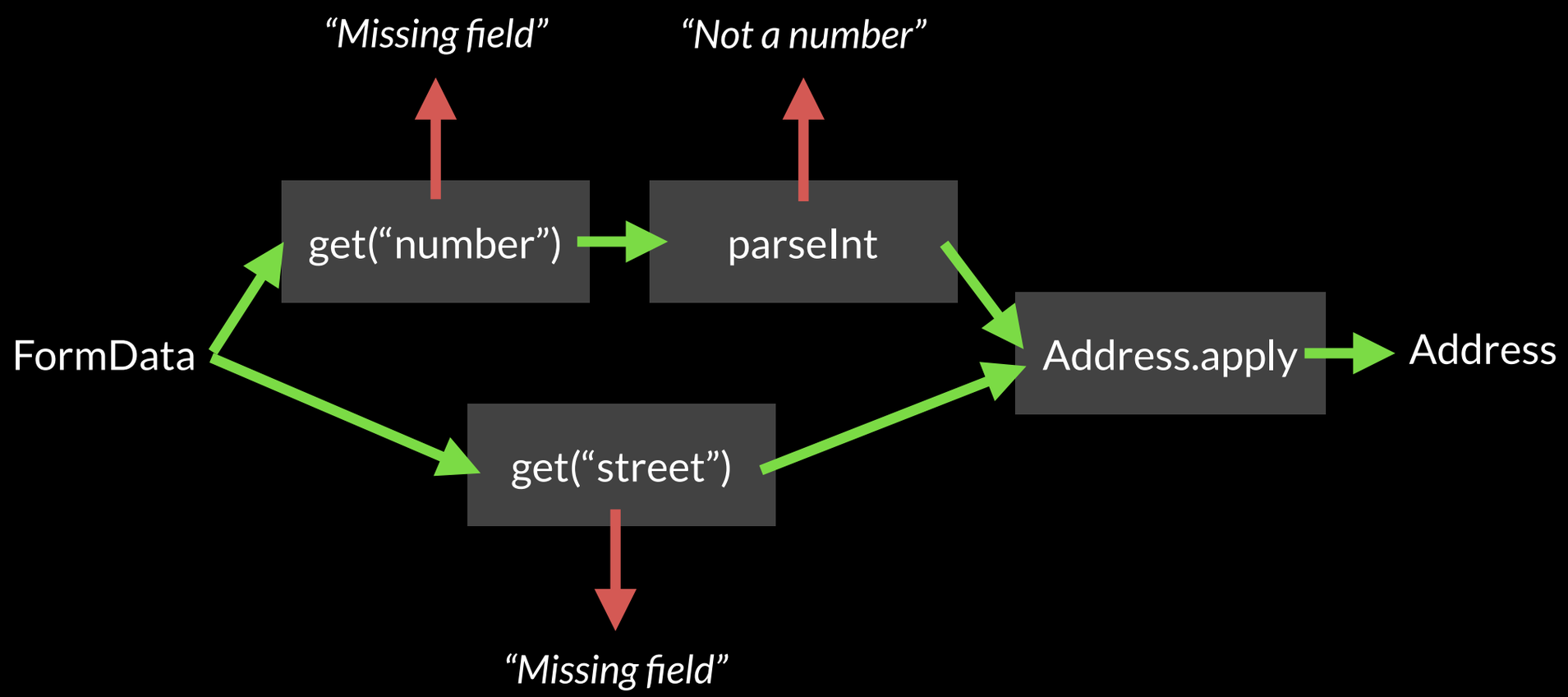
29

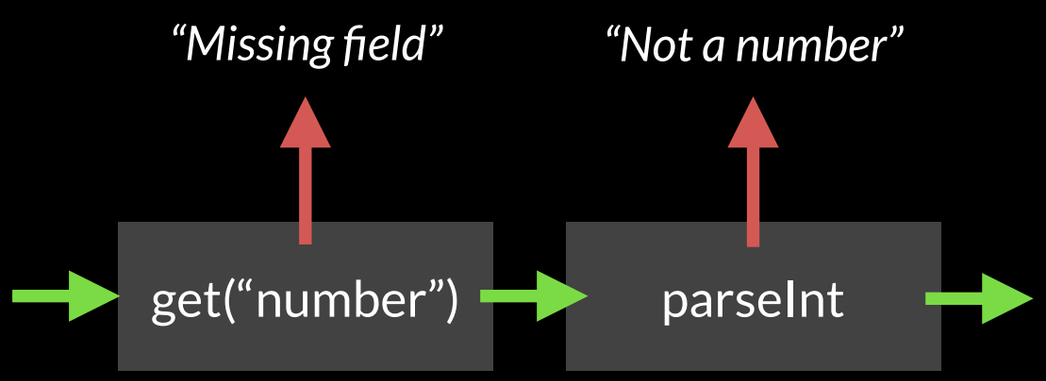
Street Name

Acacia Road

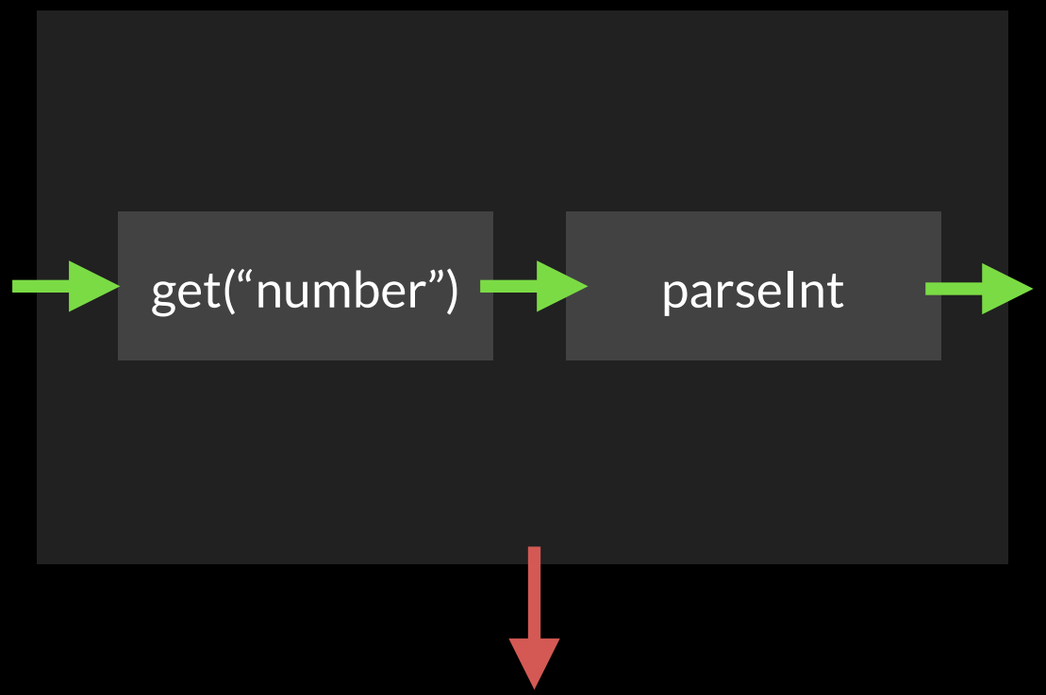
Submit

```
case class Address(
  number: Int,
  street: String)
```


“Missing field”
“Not a number”
get(“number”)
parseInt
FormData
Address.apply
Address
get(“street”)
“Missing field”


“Missing field”
“Not a number”
get(“number”)
parseInt

readNumber




*“Missing field”*

*“Not a number”*

```scala
val readNumber: Rule[FormData] =
  (form: FormData) => {
    val result1: Result =
      getField("number")(form)

    val result2: Result =
      parseInt(???)

    result2
  }
```

```
val readNumber: Rule[FormData] =
  (form: FormData) => {
    val result1: Result =
      getField("number")(form)

    val result2: Result =
      parseInt(???)

    result2
  }
```

```scala
type Rule[A] = A => Result

sealed trait Result
final case object Pass
  extends Result

final case class Fail(messages: List[String])
  extends Result
```

```
type Rule[A] = A => Result

sealed trait Result

final case class Pass[A](value: A)
  extends Result

final case class Fail(messages: List[String])
  extends Result
```

```scala
type Rule[A] = A => Result

sealed trait Result[+A]

final case class Pass[A](value: A)
  extends Result[A]

final case class Fail(messages: List[String])
  extends Result[Nothing]
```

```scala
type Rule[A, B] = A => Result[B]

sealed trait Result[+A]

final case class Pass[A](value: A)
  extends Result[A]

final case class Fail(messages: List[String])
  extends Result[Nothing]
```

```
val readNumber: Rule[FormData, Int] =
  (form: FormData) => {
    val result1: Result[String] =
      getField("number")(form)

    val result2: Result[Int] =
      parseInt(???)



    result2
  }
```

```
val readNumber: Rule[FormData, Int] =
  (form: FormData) => {
    val result1: Result[String] =
      getField("number")(form)

    val result2: Result[Int] =
      parseInt(???)

    result2
  }
```

```scala
val readNumber: Rule[FormData, Int] =
  (form: FormData) => {
    val result1: Result[String] =
      getField("number")(form)

    val result2: Result[Int] =
      result1 match {
        case Pass(a) => parseInt(a)
        case Fail(e) => Fail(e)
      }

    result2
  }
```

```
val readNumber: Rule[FormData, Int] =
  (form: FormData) => {
    val result1: Result[String] =
      getField("number")(form)

    val result2: Result[Int] =
      result1.???(parseInt)


    result2
  }
```

```
sealed trait Result[+A] {
  def flatMap[B](r: A => Result[B]): Result[B] =
    ???

}
```

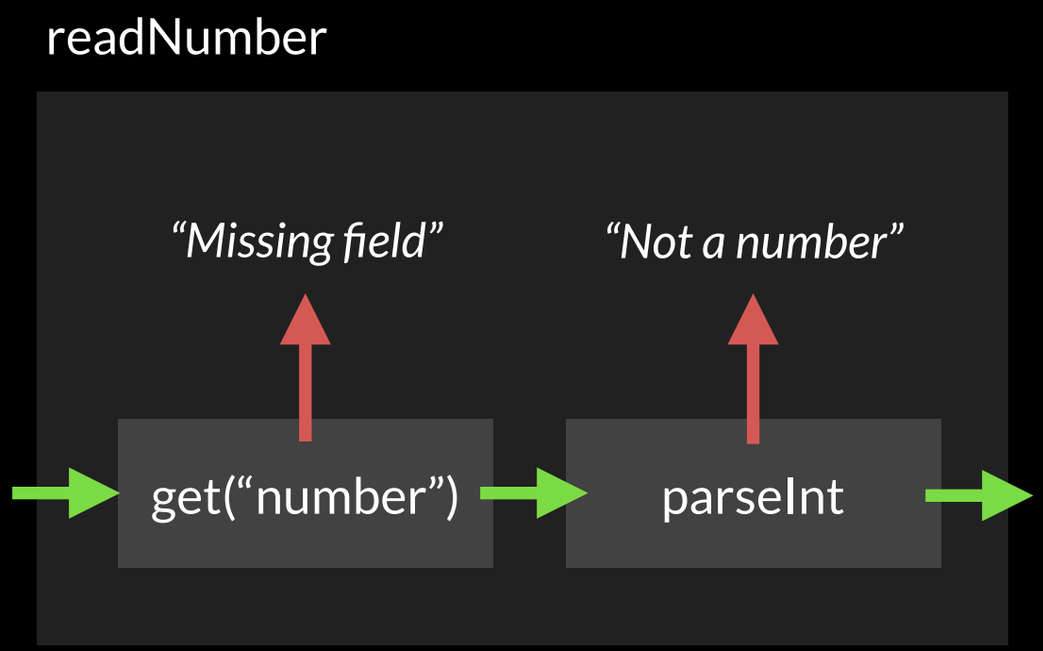
```scala
sealed trait Result[+A] {
  def flatMap[B](r: A => Result[B]): Result[B] =
    this match {
      case Pass(a) => ???
      case Fail(e) => ???
    }
}
```

```scala
sealed trait Result[+A] {
  def flatMap[B](r: A => Result[B]): Result[B] =
    this match {
      case Pass(a) => r(a)
      case Fail(e) => Fail(e)
    }
}
```

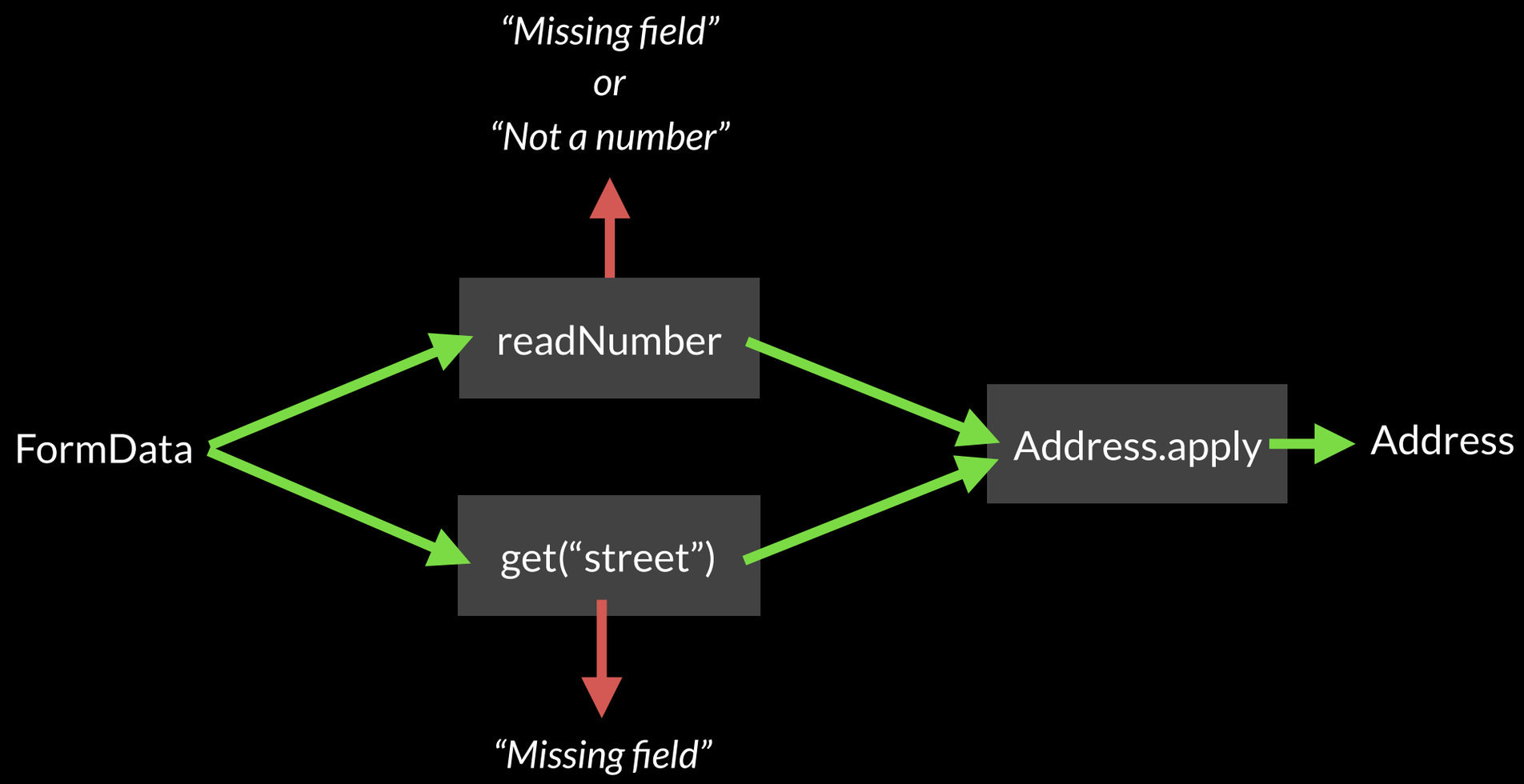
```scala
val readNumber: Rule[FormData, Int] =
  (form: FormData) => {
    val result1: Result[String] =
      getField("number")(form)

    val result2: Result[Int] =
      result1.flatMap(parseInt)

    result2
  }
```

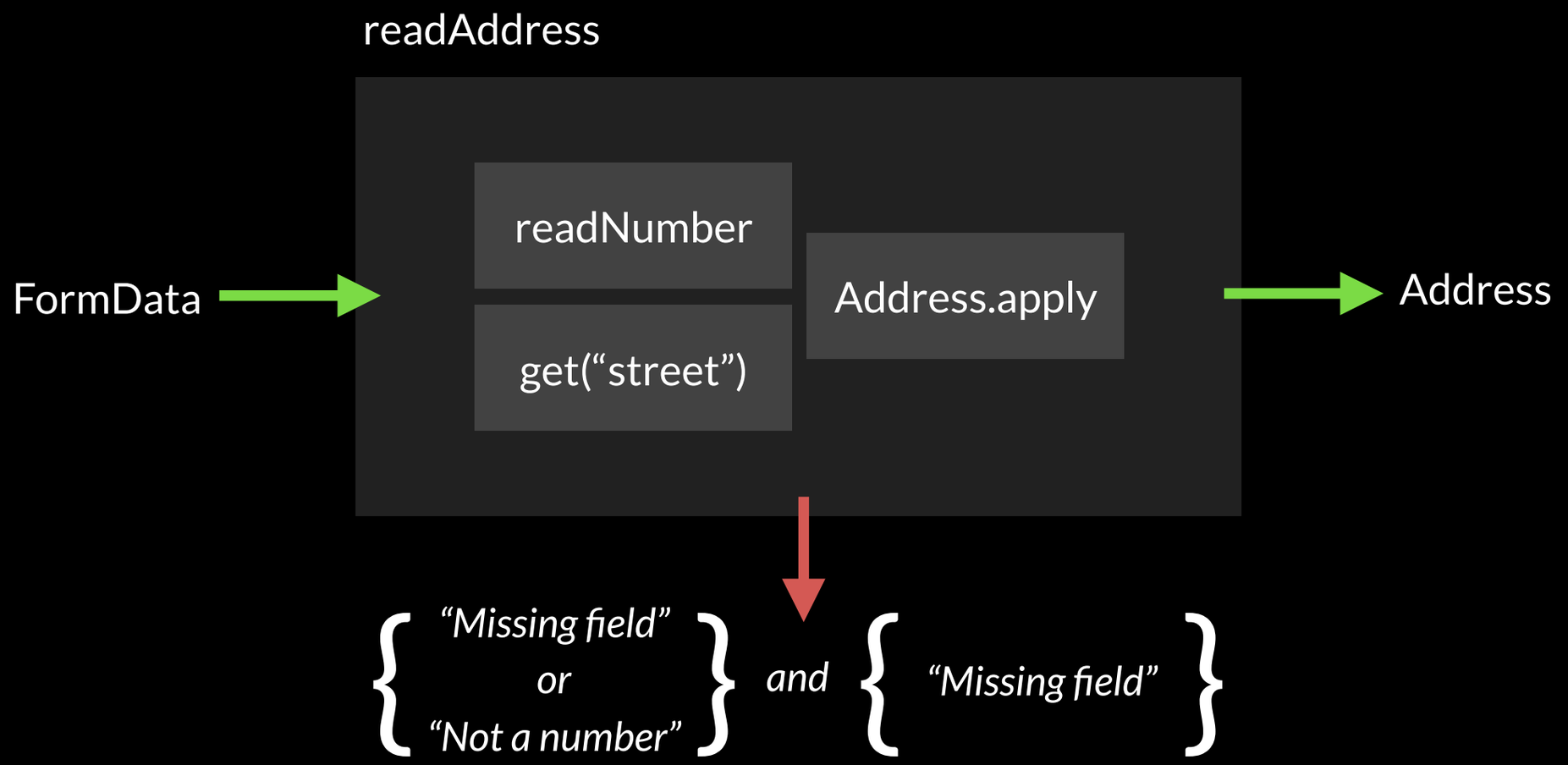
```
val readNumber: Rule[FormData, Int] =
  (form: FormData) =>
    for {
      str <- getField("number")(form)
      num <- parseInt(str)
    } yield num
```

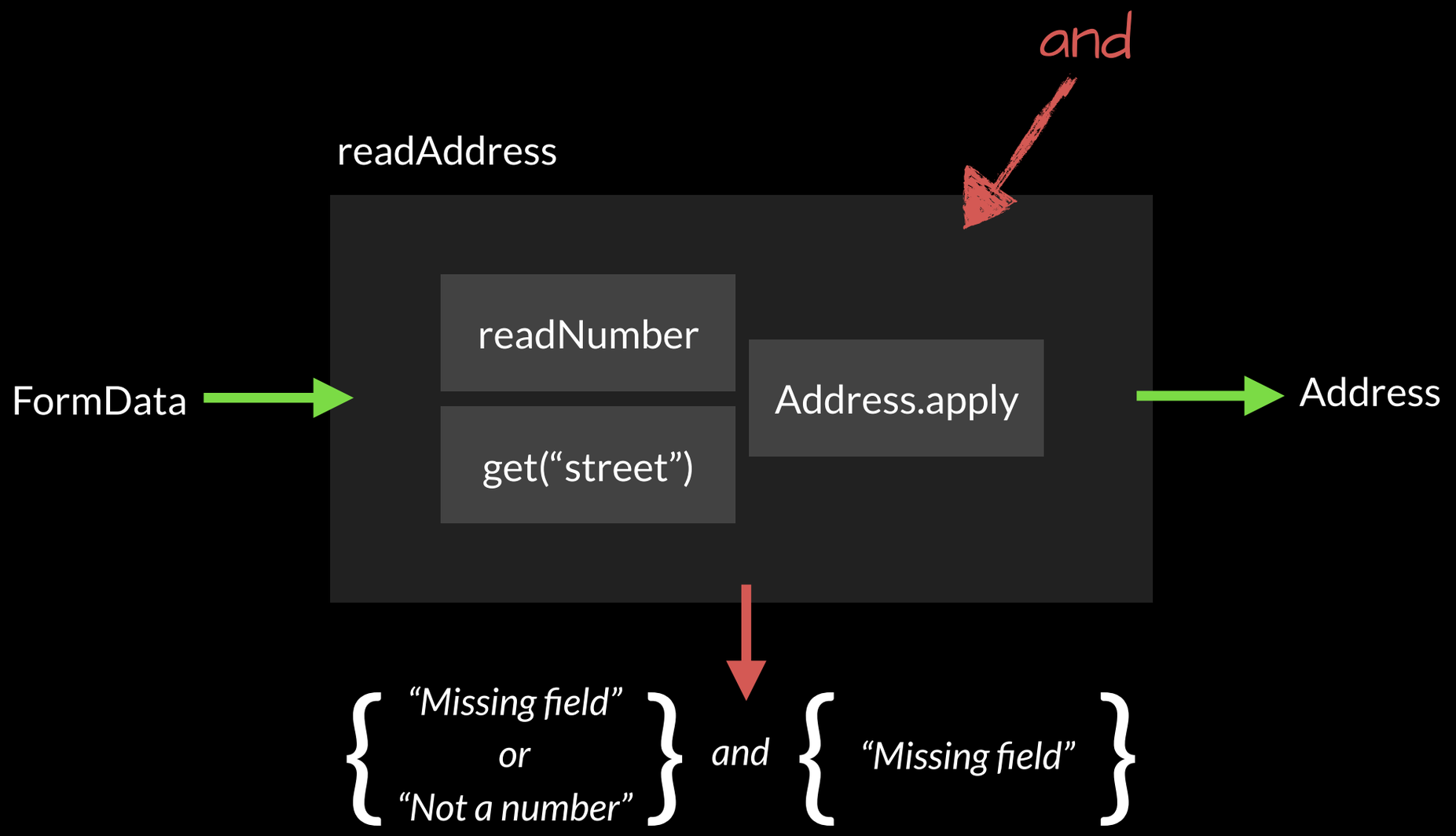
# flatMap throws away errors…




…but that’s a good thing!


“Missing field”
or
“Not a number”
readNumber
FormData
Address.apply
Address
get(“street”)
“Missing field”


readAddress
readNumber
get("street")
Address.apply
FormData
Address
"Missing field"
or
"Not a number"
and
"Missing field"


and
readAddress
readNumber
get("street")
Address.apply
FormData
Address
"Missing field"
or
"Not a number"
and
"Missing field"

```scala
sealed trait Result {
  def and(that: Result): Result =
    (this, that) match {
      case (Pass,    Pass   ) => Pass
      case (Fail(a), Pass   ) => Fail(a)
      case (Pass   , Fail(b)) => Fail(b)
      case (Fail(a), Fail(b)) => Fail(a ++ b)
    }
}
```

```scala
sealed trait Result[A] {
  def and[B, C](that: Result[B]): Result[C] =
    (this, that) match {
      case (Pass(a), Pass(b)) => Pass(???)
      case (Fail(a), Pass(b)) => Fail(a)
      case (Pass(a), Fail(b)) => Fail(b)
      case (Fail(a), Fail(b)) => Fail(a ++ b)
    }
}
```

```scala
sealed trait Result[A] {
  def and[B, C](that: Result[B]): Result[C] =
    (this, that) match {
      case (Pass(a), Pass(b)) => Pass(???)
      case (Fail(a), Pass(b)) => Fail(a)
      case (Pass(a), Fail(b)) => Fail(b)
      case (Fail(a), Fail(b)) => Fail(a ++ b)
    }
}
```

```scala
sealed trait Result[A] {
  def and[B, C](that: Result[B])
               (fn: (A, B) => C): Result[C] =
    (this, that) match {
      case (Pass(a), Pass(b)) => Pass(fn(a, b))
      case (Fail(a), Pass(b)) => Fail(a)
      case (Pass(a), Fail(b)) => Fail(b)
      case (Fail(a), Fail(b)) => Fail(a ++ b)
    }
}
```

```
val readAddress: Rule[FormData, Address] =
  (form: FormData) => {
    val numberResult = readNumber(form)
    val streetResult = getField("street")(form)
    numberResult.and(streetResult)
                (Address.apply)
}
```

# End of Sprint #2

Rules are now pipelines
(that can alter data as it passes through)

Results carry values from rule to rule

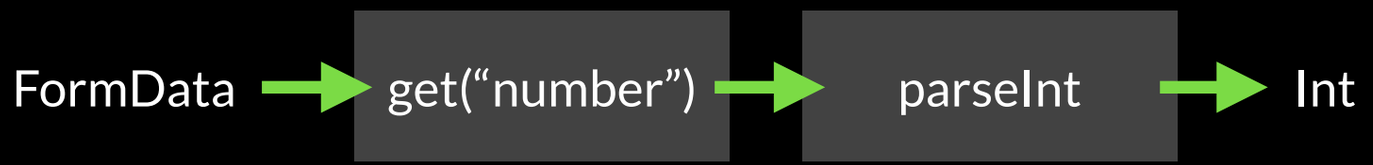
We sequence with map and flatMap
(and optionally for comprehensions)

Sequencing throws away errors
(but composition using and retains them)

Sprint #3

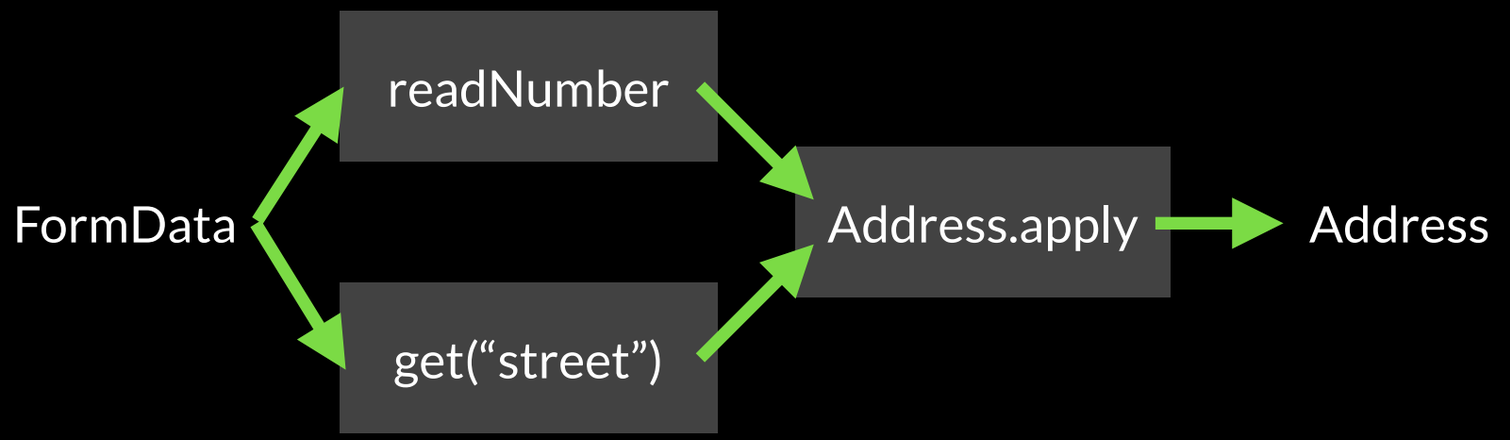
# Heavy Lifting

*in which rules need combinators too*


FormData
get(“number”)
parseInt
Int

```
val readNumber: Rule[FormData, Int] =
  (form: FormData) =>
    for {
      str <- getField("number")(form)
      num <- parseInt(str)
    } yield num
```

```
val readNumber =
  getField(“number”) ??? parseInt
```


readNumber
FormData
Address.apply
Address
get(“street”)

```
val readAddress: Rule[FormData, Address] =
  (form: FormData) => {
    val numberResult = readNumber(form)
    val streetResult = getField("street")(form)
    numberResult.and(streetResult)
                (Address.apply)
}
```

```
val readAddress =
  readNumber.???(getField(“street"),
    Address.apply)
```

```
implicit class RuleOps[A, B](r1: Rule[A, B]) {

}
```

```
implicit class RuleOps[A, B](r1: Rule[A, B]) {
  def map[C](fn: B => C): Rule[A, C]

  def flatMap[C](r2: Rule[B, C]): Rule[A, C]

  def and(r2: Rule[A, C])(fn: (B, C) => D): Rule[A, D]
}
```

```
implicit class RuleOps[A, B](r1: Rule[A, B]) {
  def map[C](fn: B => C): Rule[A, C] =
    (in: A) => r1(in).map(fn)

  def flatMap[C](r2: Rule[B, C]): Rule[A, C] =
    (in: A) => r1(in).flatMap(r2)

  def and(r2: Rule[A, C])(fn: (B, C) => D): Rule[A, D] =
    (in: A) => r1(in).and(r2)(fn)
}
```

```scala
implicit class RuleOps[A, B](r1: Rule[A, B]) {
  def map[C](fn: B => C): Rule[A, C] =
    (in: A) => r1(in).map(fn)

  def flatMap[C](r2: Rule[B, C]): Rule[A, C] =
    (in: A) => r1(in).flatMap(r2)

  def and(r2: Rule[A, C])(fn: (B, C) => D): Rule[A, D] =
    (in: A) => r1(in).and(r2)(fn)
}
```

map

Rule[A, B] + B => C → Rule[A, C]

flatMap

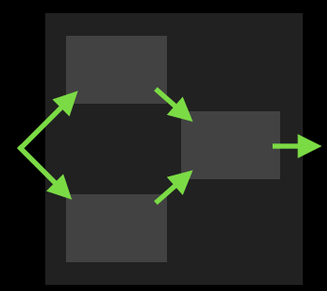
Rule[A, B] + Rule[B, C] → Rule[A, C]

and

Rule[A, B]

Rule[A, C]

+ (B, C) => D → Rule[A, D]



```
val readNumber: Rule[FormData, Int] =
  getField("number") flatMap parseInt

val readStreet: Rule[FormData, String] =
  getField("street")

val readAddress =
  readNumber.and(readStreet)(Address.apply)
```

```scala
val checkNumber: Rule[Int, Int] =
  gte(1)

val checkStreet: Rule[String, String] =
  nonEmpty

val checkAddress: Rule[Address, Address] =
  ???
```

```
val checkNumber: Rule[Int, Int] =
  gte(1)

val checkStreet: Rule[String, String] =
  nonEmpty

val checkAddress: Rule[Address, Address] =
  ???
```

```
def rule[A]: Rule[A, A] =
  (input: A) => Pass(input)
```

```scala
val checkNumber: Rule[Int, Int] =
  gte(1)

val checkStreet: Rule[String, String] =
  nonEmpty

val checkAddress: Rule[Address, Address] =
  ???
```

```scala
val checkNumber: Rule[Address, Int] =
  rule[Address] map (_.number) flatMap gte(1)

val checkStreet: Rule[Address, String] =
  rule[Address] map (_.street) flatMap nonEmpty

val checkAddress: Rule[Address, Address] =
  checkNumber.and(checkStreet)(Address.apply)
```

# End of Sprint #3

We have lifted our combinators
(from Result to Rule)

Each method is directly comparable
(isomorphic)

We can now compose rules directly
(map and flatMap in sequence, and in parallel)

The methods on Result are still useful

# Summary

# Functional Design

Model simplest parts
(Rules and Results)

Create combinators
(building blocks)

# Functional Design

In validation, combinators are:

**Sequencing**
(transform data, *map*, *flatMap*)

**Parallel composition**
(*and*, applicative functors/builders)

# Functional Design

We can sometimes lift operations

This leads to higher level abstractions

Low-level abstractions are often still useful

# Functional Design

We didn't talk about and for >2 rules

Scala hates arbitrary arity

Abstract it away with…

Applicative functors

Applicative builders

Recursive data structures (e.g. HLists)

# Thank You

Dave Gurnell, @davegurnell

http://github.com/davegurnell/functional-data-validation