Railway Oriented Programming

Une approche fonctionnelle pour la gestion d'erreurs

Ramiro Calle

Motivation

Transition vers de la programmation fonctionnelle

La terminologie peut faire peur

- Théorie des catégories

Nouvelles fonctions et d'opérateurs à connaître

- « map », « flatMap », « fold », …
- « <+> », « >>= », ... (cats, scalaz, ...)

L’ approche optimiste

"En tant qu'utilisateur, je souhaite mettre à jour mon nom et mon adresse électronique"

```
case class Request(userId: Int, name: String, email: String)

def executeUseCase: String = {
  val request = receiveRequest
  validateRequest(request)
  canonicalizeEmail(request)
  db.updateDbFromRequest(request)
  smtpServer.sendEmail(request.email)
  "Success"
}
```

L’approche réaliste

```
def executeUseCase: String = {
  val request = receiveRequest
  val isValidated =  validateRequest(request)
  if (!isValidated) {
    return "Request is not valid"
  }
  canonicalizeEmail(request)
  try {
    val result = db.updateDbFromRequest(request)
    if (!result) {
      return "Customer record not found"
    }
  }catch {
    case _: DatabaseException =>
     return "DB error: Customer record not updated"
  }
  if(!smtpServer.sendEmail(request.email)) {
    logger.error("Customer email not sent")
  }
  "Success"
}
```

+ Plus de code
+ Plus d’erreurs possible
- Moins lisible

Testabilité ?

Approche Fonctionnelle (1/2)

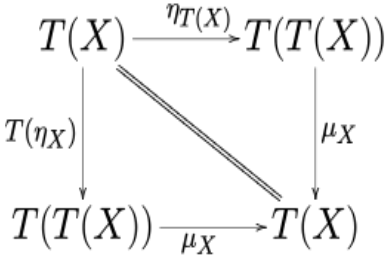
Fonction « pure »

- Pas d’effet de bord (Erreur dans le type de retour)
- La valeur de retour est la même pour les mêmes paramètres

```
sealed trait Result[T]
case class Success[T](value: T) extends Result[T]
case class Failure[T](message: String) extends Result[T]
```

Approche Fonctionnelle (2/2)

*« Une monade est une construction catégorique qui mime formellement le comportement que les monoïdes ont en algèbre. » **



* https://fr.wikipedia.org/wiki/Monade_(théorie_des_catégories)

Railway Oriented Programming

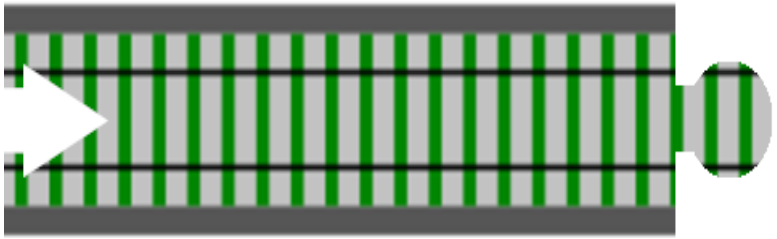
Fonctions

- Aiguillage
- Voie simple
- Impasse
- Exceptions

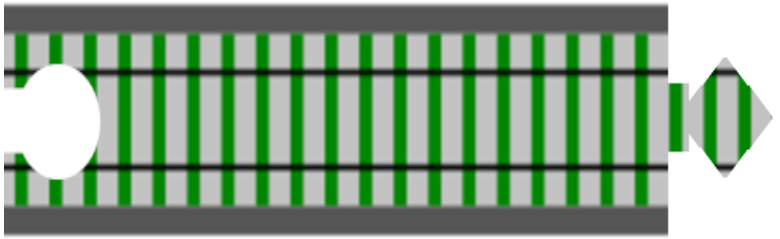
Thanks Scott Wlaschin !
https://fsharpforfunandprofit.com/rop/

Composition

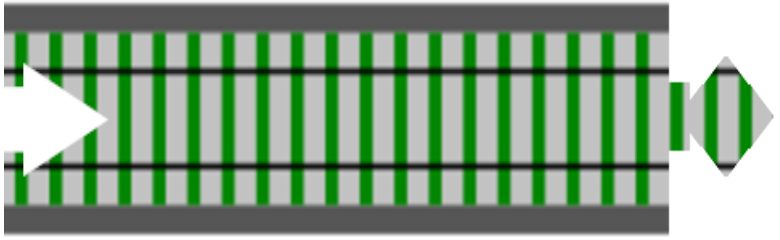
```
def triangleToCircle(t: Triangle): Circle
```



```
def circleToSquare(c: Circle): Square
```



```
def triangleToSquare(t: Triangle): Square
```

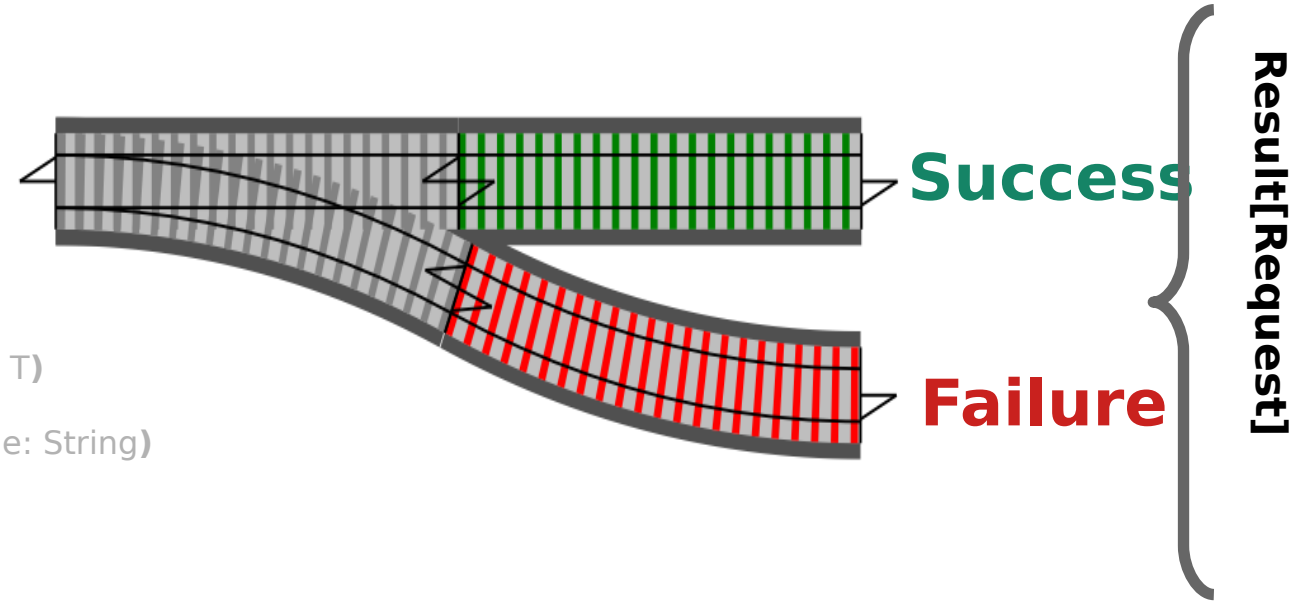


Railway Oriented Programming

Fonctions

- Aiguillage
- Voie simple
- Impasse
- Exceptions

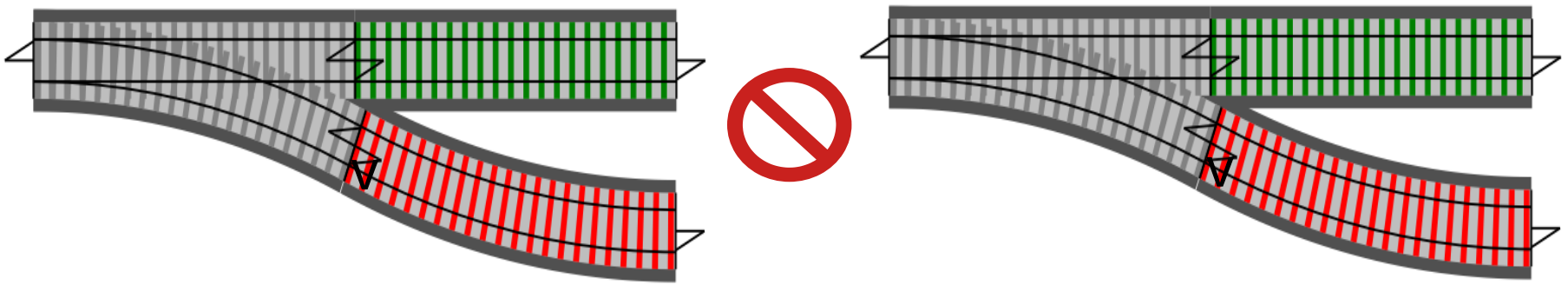
Fonctions Aiguillage



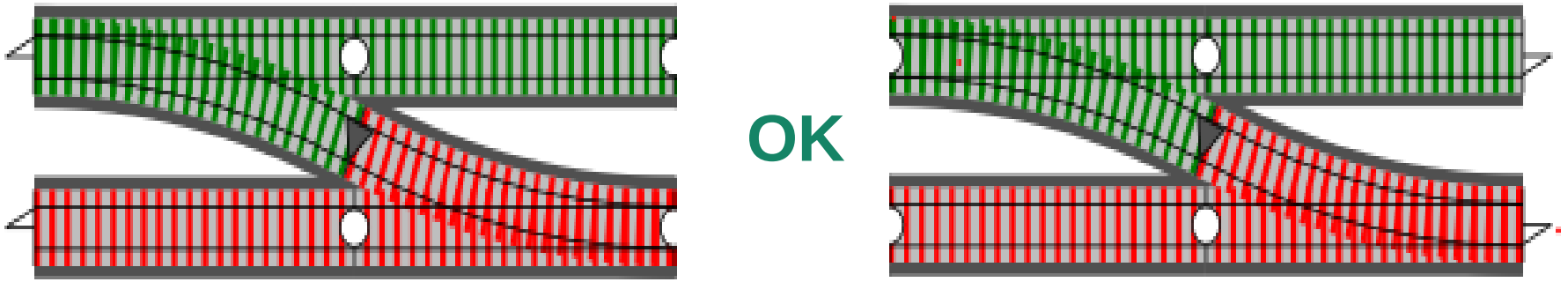

```
sealed trait Result[T]
case class Success[T](value: T)
  extends Result[T]
case class Failure[T](message: String)
  extends Result[T]
```

```
def validateInput(input: Request): Result[Request] =
  if (input.name == "")
    Failure("Name must not be blank")
  else if (input.email == "")
    Failure("Email must not be blank")
  else
    Success(input)
```

Composition ? (1/2)

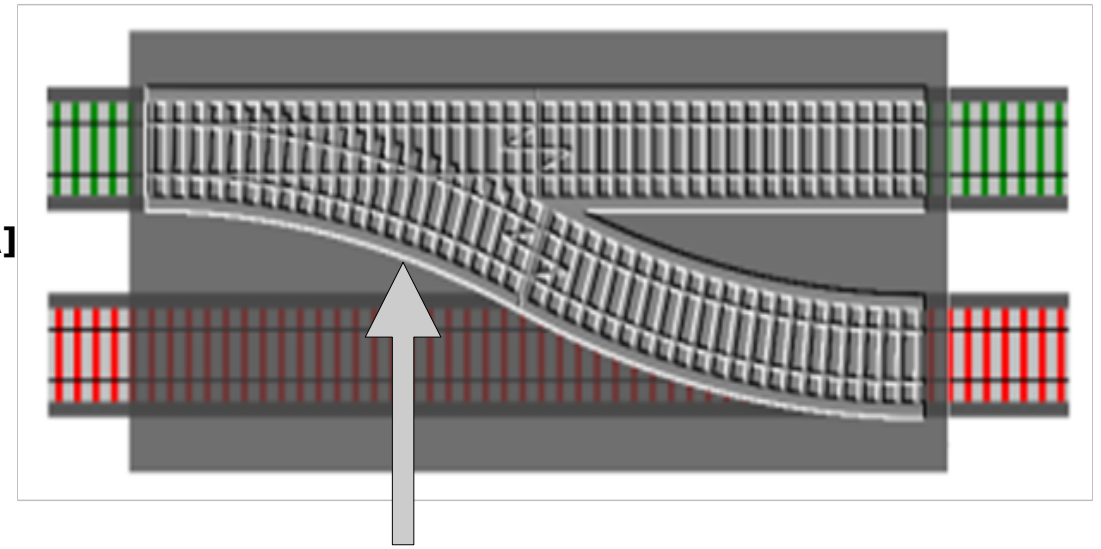


Composition ? (2/2)



FlatMap (1/2)

towTrackInput : **Result[A]**



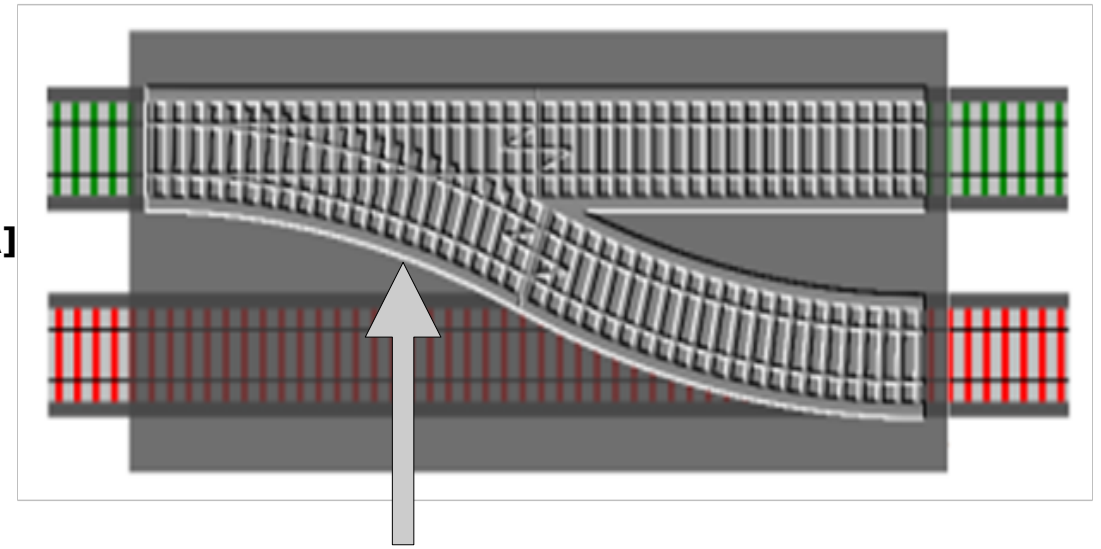
Result[B]

switchFunction: **A => Result[B]**

```
def flatMap[A, B](towTrackInput: Result[A])(switchFunction: A => Result[B]): Result[B] =
  towTrackInput match {
    case Success(value) => switchFunction(value)
    case Failure(message) => Failure(message)
  }
```

FlatMap (2/2)

towTrackInput : **Result[A]**



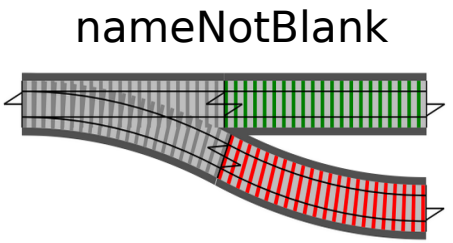
Result[B]

switchFunction: **A => Result[B]**

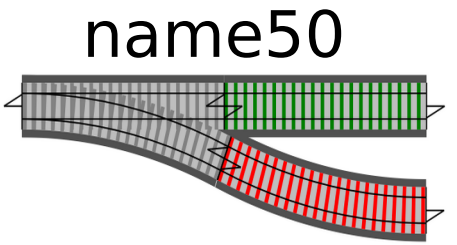
```
sealed trait Result[A] {
  def flatMap[B](switchFunction: A => Result[B]): Result[B] = this match {
    case Success(value) => switchFunction(value)
    case Failure(message) => Failure(message)
  }
}
```

FlatMap Exemple (1/2)

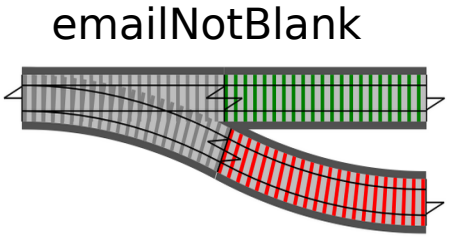
```
def nameNotBlank(input: Request): Result[Request] =
  if (input.name == "")
    Failure("Name must not be blank")
  else Success(input)
```



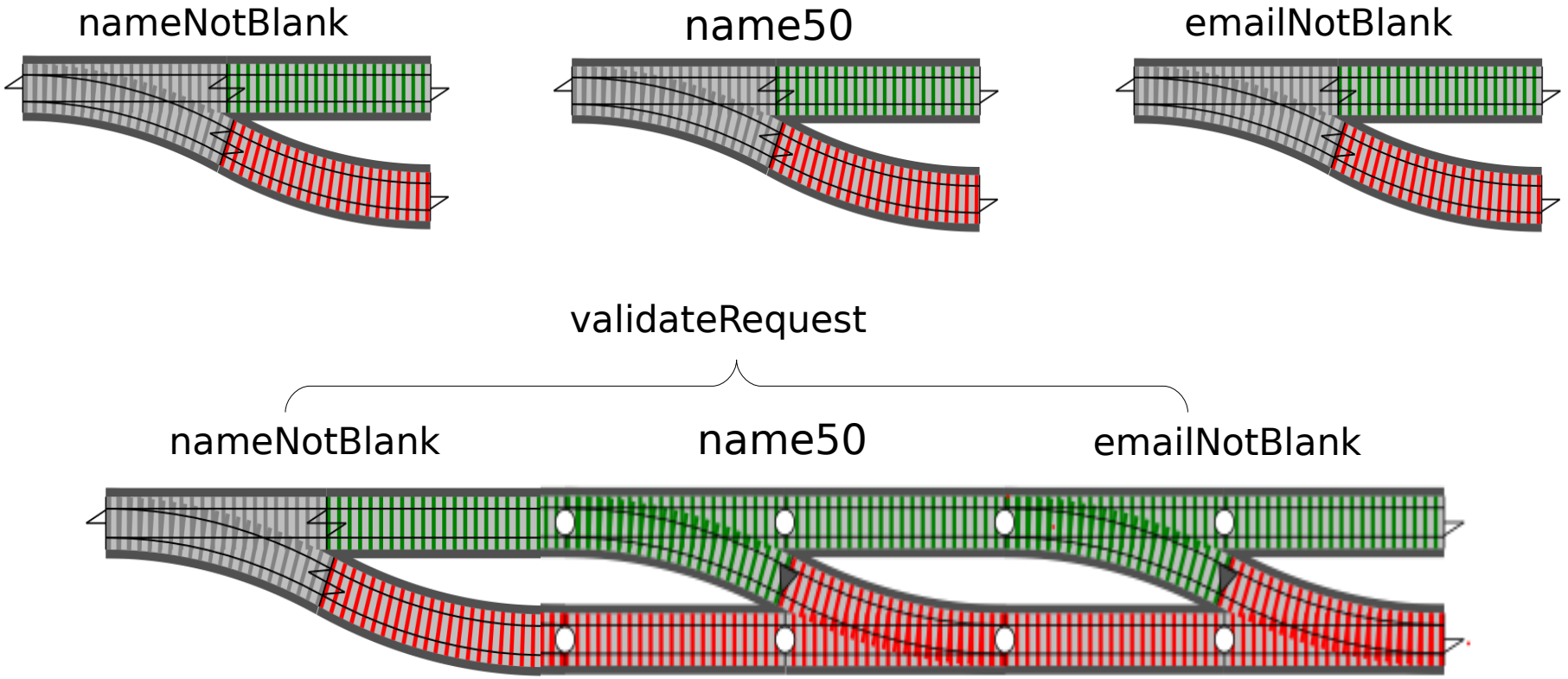

```
def name50(input: Request): Result[Request] =
  if(input.name.length > 50)
    Failure("Name must not be longer than 50 chars")
  else Success(input)
```




```
def emailNotBlank(input: Request): Result[Request] =
  if(input.email == "")
    Failure( "Email must not be blank")
  else Success(input)
```



FlatMap Exemple (2/2)




```
def validateRequest(oneTrackInput: Request): Result[Request] =
  nameNotBlank(oneTrackInput) flatMap name50 flatMap emailNotBlank
```

Railway Oriented Programming

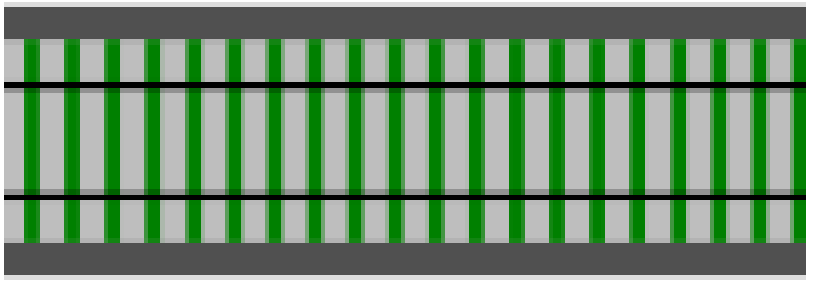
Fonctions

- Aiguillage
- Voie simple
- Impasse
- Exceptions

Voie Simple

canonicalizeEmail

input: **Request**

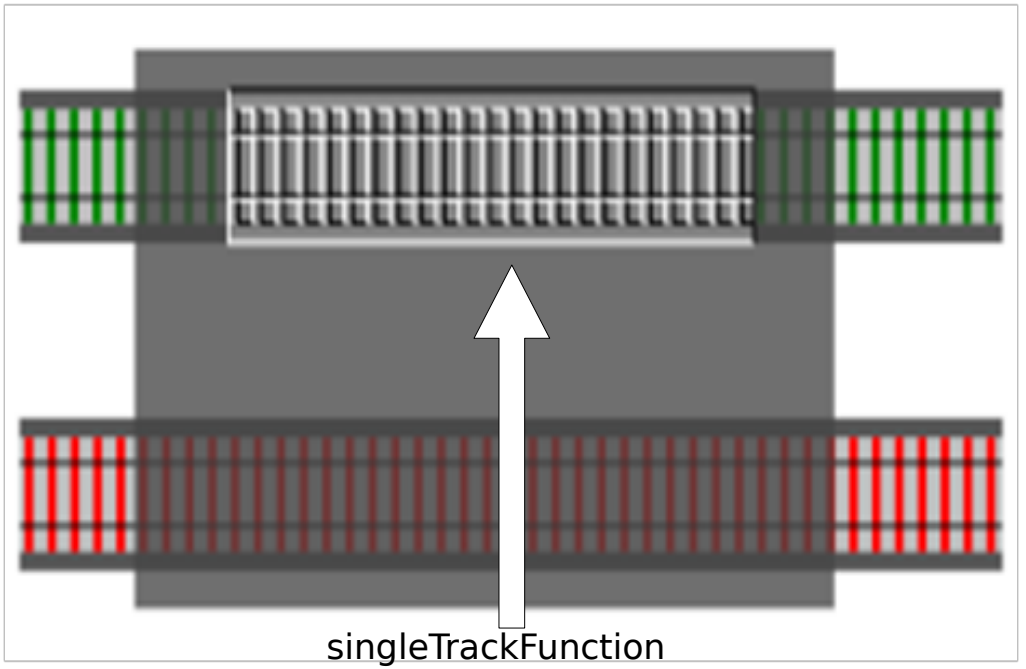


Request

```
// trim spaces and lowercase
def canonicalizeEmail(input: Request): Request =
  input.copy(email = input.email.trim().toLowerCase())
```

Map (1/2)

towTrackInput : **Result[A]**

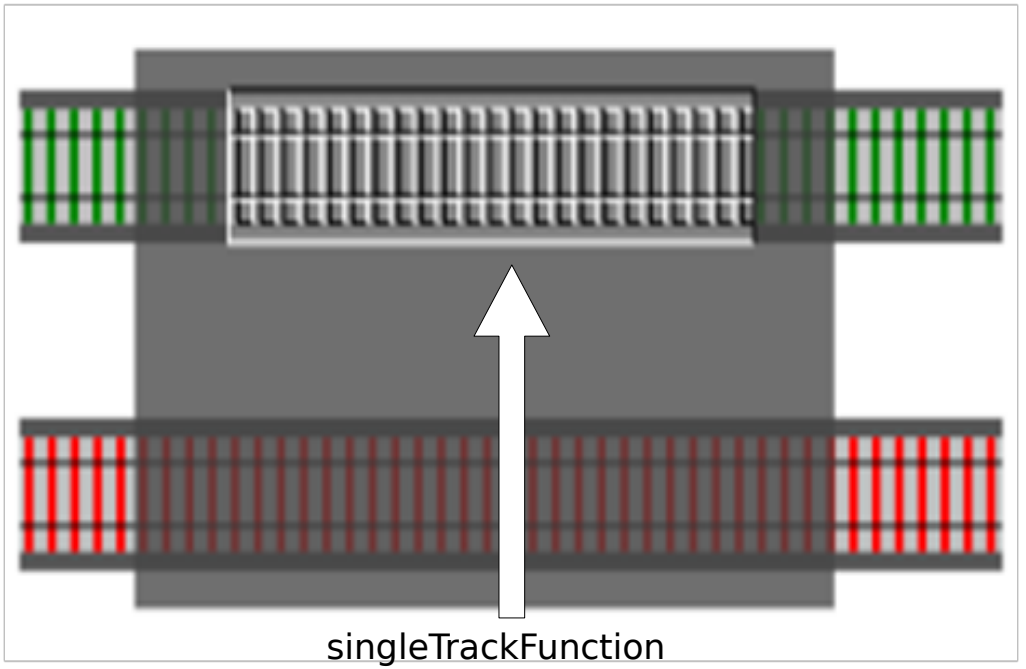


Result[B]

```
def map[A, B](towTrackInput: Result[A])(singleTrackFunction: A => B): Result[B] =
  towTrackInput match {
    case Success(value) => Success(singleTrackFunction(value))
    case Failure(message) => Failure(message)
  }
```

Map (2/2)

towTrackInput : **Result[A]**



Result[B]

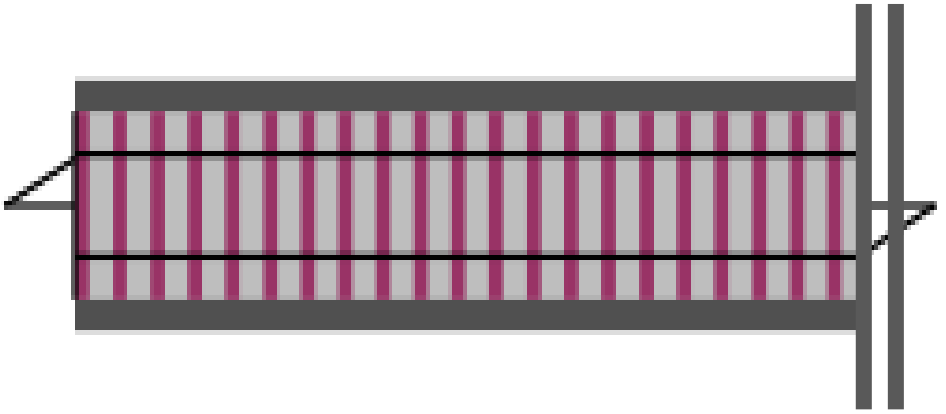
```
sealed trait Result[A] { // ...
  def map[B](singleTrackFunction: A => B): Result[B] = this match {
    case Success(value) => Success(f(value))
    case Failure(message) => Failure(message)
  }
}
```

Railway Oriented Programming

Fonctions

- Aiguillage
- Voie simple
- Impasse
- Exceptions

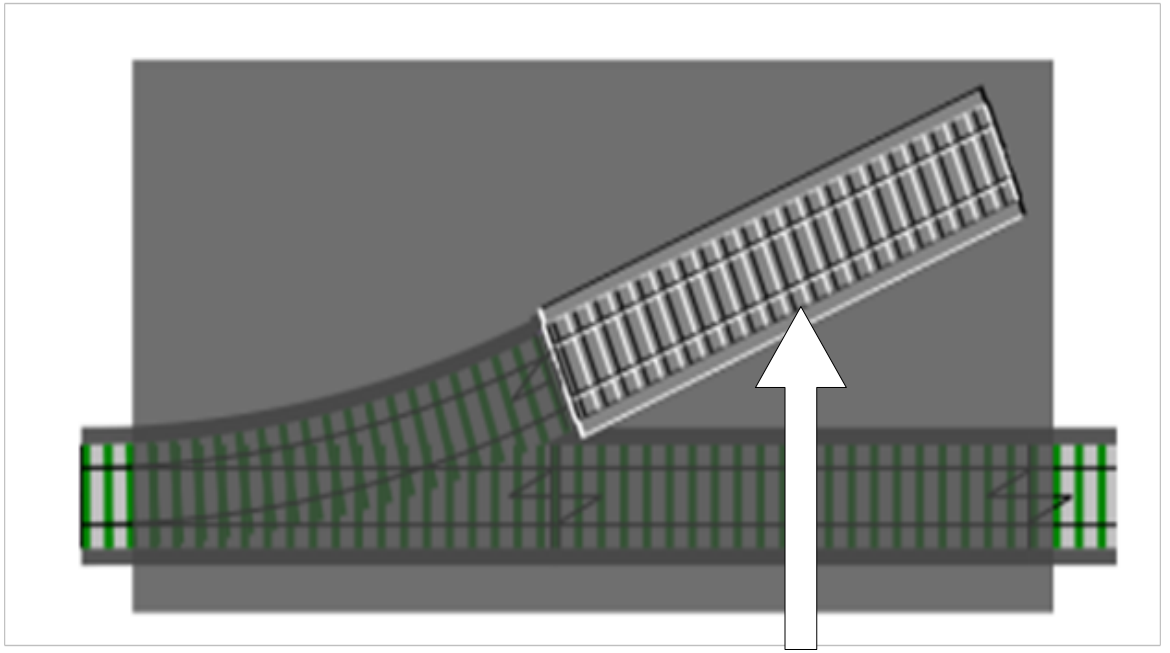
Impasse



```
def updateDb(request: Request): Unit = {
  // do something
  // return nothing at all
}
```

Tee

oneTrackInput: **T**



T

deadEndFunction: **T => Unit**

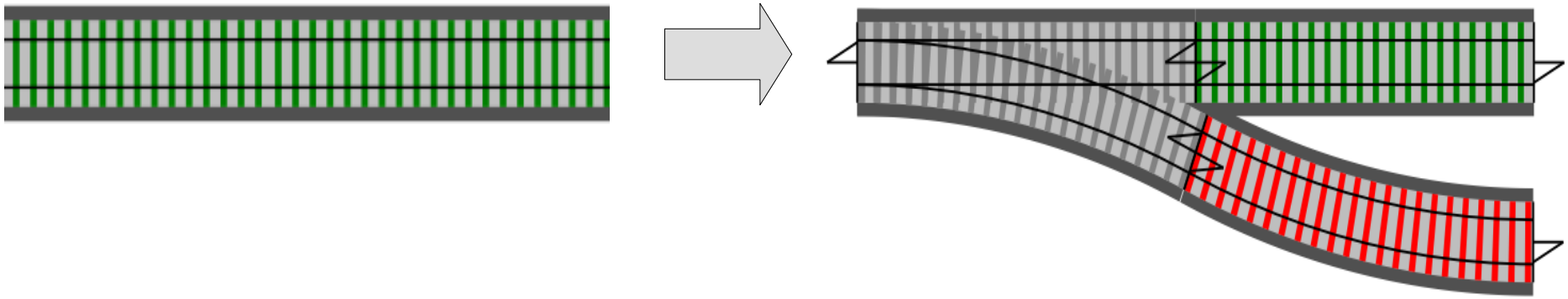
```
def tee[T](deadEndFunction: T => Unit)(oneTrackInput: T): T = {
  deadEndFunction(oneTrackInput)
  oneTrackInput
}
```

Railway Oriented Programming

Fonctions

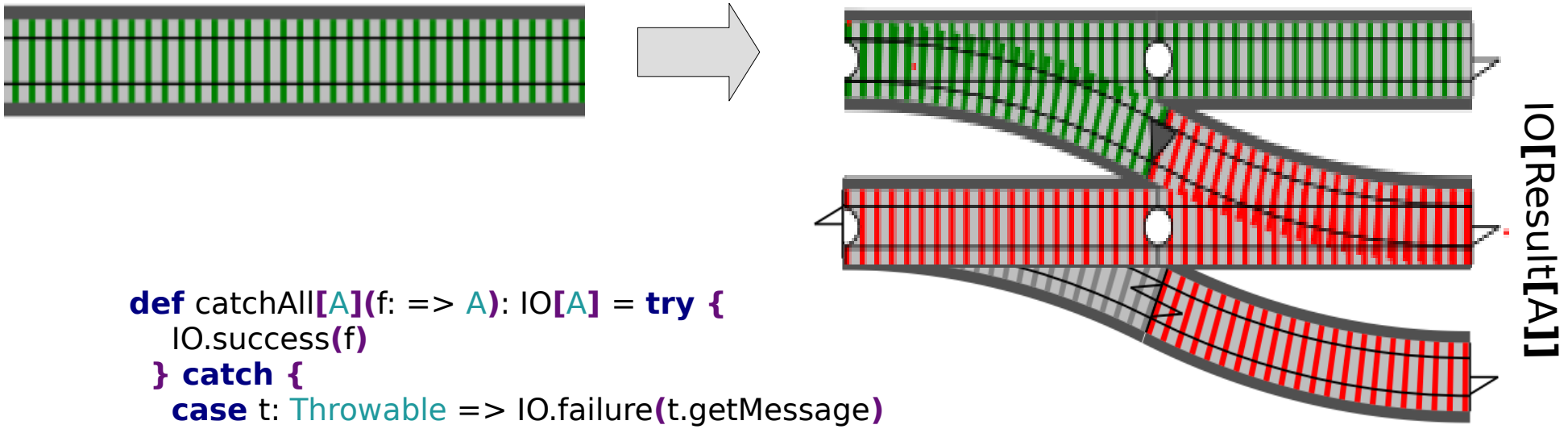
- Aiguillage
- Voie simple
- Impasse
- Exceptions

Exceptions (1/2)



```
def sendEmail(request: Request): Result[Request] =
  try {
    if (smtpServer.sendEmail(request.email)) {
      Success(request)
    } else Failure("Customer email not sent")
  } catch {
    case e: SMTPException =>
      Failure(s"SMTP error: ${e.getMessage}")
  }
```

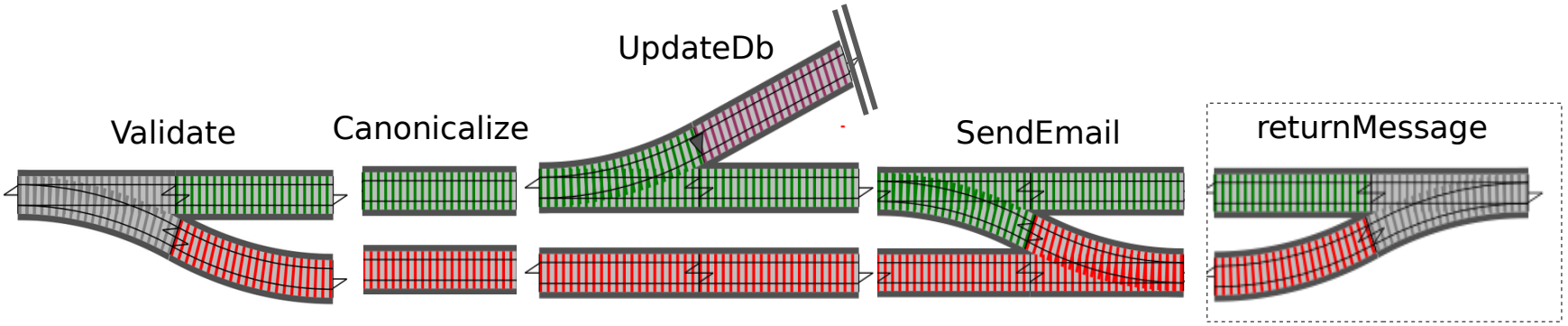
Exceptions (2/2)




```
def catchAll[A](f: => A): IO[A] = try {
    IO.success(f)
  } catch {
    case t: Throwable => IO.failure(t.getMessage)
  }
```

```
def catchOnly[T <: Throwable: ClassTag, A](f: => A): IO[A] = try {
    IO.success(f)
  } catch {
    case t: T => IO.Failure(t)
  }
```

On met tout ça bout à bout (1/3)



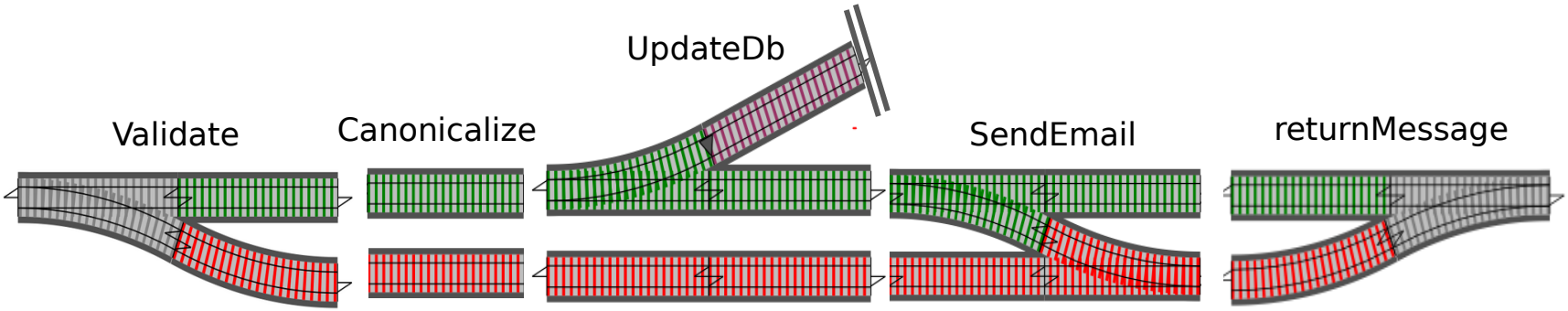

```
def returnMessage(result: Result[String]): String = result match {
  case Success(value) => value
  case Failure(message) => message
}

def executeUseCase(input: Request): String = returnMessage {
  validateRequest(input) map canonicalizeEmail map tee(updateDb) flatMap
     sendEmail map(_ => "Success")
}
```

On met tout ça bout à bout (2/3)

```scala
sealed trait Result[A] {
  def flatMap[B](switchFunction: A => Result[B]): Result[B] = this match {
    case Success(value) => switchFunction(value)
    case Failure(message) => Failure(message)
  }
  def map[B](singleTrackFunction: A => B): Result[B] = this match {
    case Success(value) => Success(singleTrackFunction(value))
    case Failure(message) => Failure(message)
  }
}
object Result {
  def pure[A](x: A): Result[A] = Success(x)
}
case class Success[T](value: T) extends Result[T]
case class Failure[T](message: String) extends Result[T]
```

« Result » a un comportement monadique !

On met tout ça bout à bout (3/3)




```
def executeUseCase(input: Request): String = returnMessage {
  for {
    validated       <- validateRequest(input)
    canonicalized <- Result.pure(canonicalizeEmail(validated))
    persisted       <- Result.pure(tee(updateDb)(canonicalized))
    sent            <- sendEmail(persisted)
  } yield "Success"
}
```

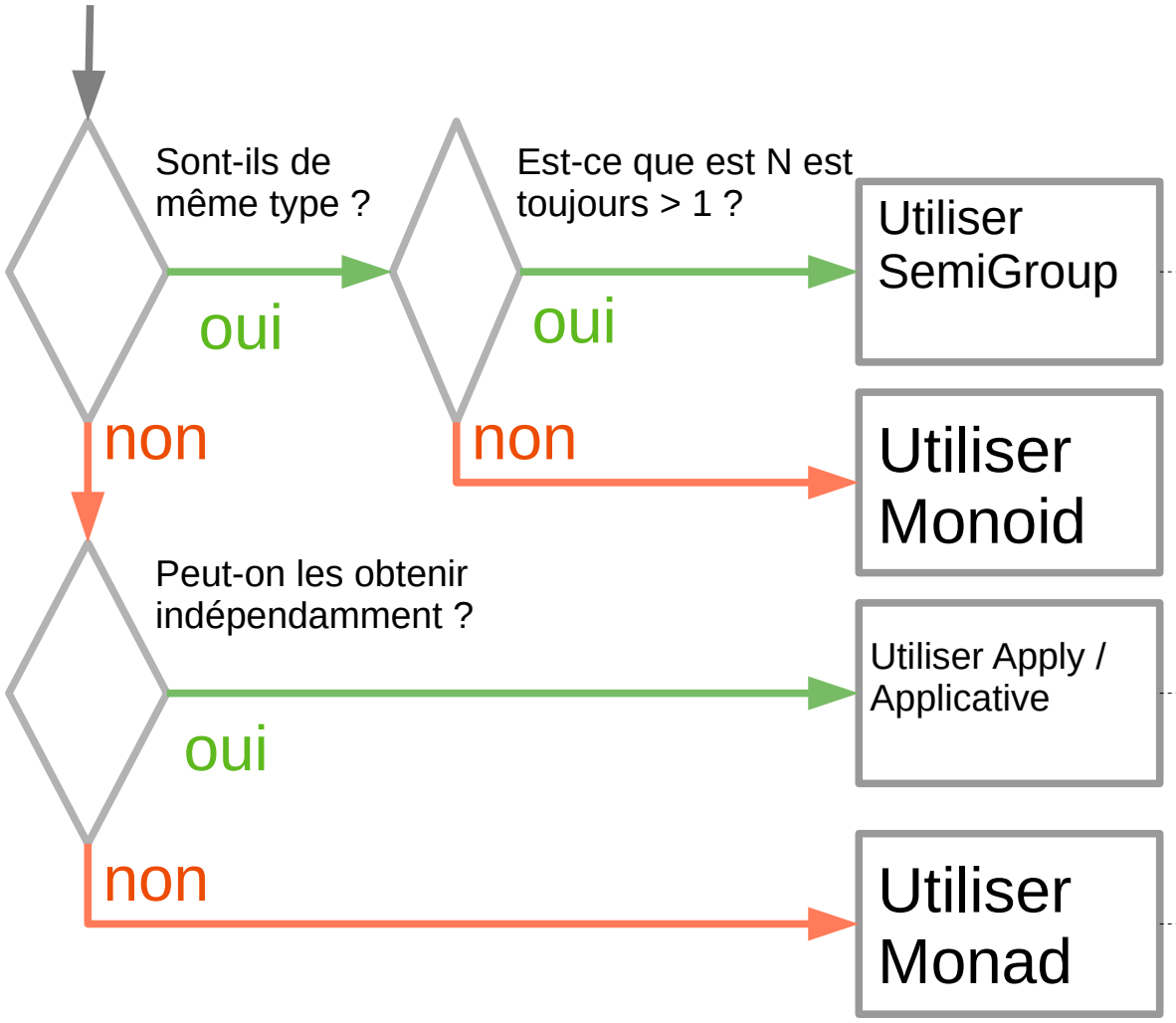
Aller plus loin

Utiliser des librairies comme « cats » ou « scalaz »

- Monad Transformers
- Writer Monad
- Effects
- ...

Ne pas utiliser des « for comprehension » partout

J’ai besoin de combiner N objets




```scala
import  cats.syntax.semigroup._

implicit val catsInstancesForF: Semigroup[F] = ???
val foo: F = ???; val bar: F = ???
def doStuff(f: F): Unit = ???

doStuff(foo |+| bar)
```

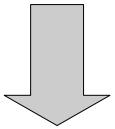
```scala
import cats.syntax.apply._

implicit val catsInstancesForF: Functor[F]
  with Semigroupal[F] = ???
val foo: A = ??? ; val bar: B = ???
def doStuff(a: A, b: B): C = ???

(F(foo), F(bar)).mapN(doStuff): F[C]
```

```scala
import cats.syntax.flatMap._, cats.syntax.functor._

implicit val catsInstancesForF: Monad[F] = ???
def obtainFoo: F[A] = ???
def obtainBar(a: A): F[B] = ???
def doStuff(a: A, b: B): C = ???

(for {
    foo <- obtainFoo
    bar <- obtainBar(foo)
  } yield doStuff(foo, bar)): F[C]
```

* @ValentinKasas

Utiliser Cats Apply

```
def validateRequest(oneTrackInput: Request): Result[Request] =
  nameNotBlank(oneTrackInput) flatMap name50 flatMap emailNotBlank
```



```
import cats.syntax.apply._

implicit val catsInstancesForResult: Functor[Result]
  with Semigroupal[Result] = ???

def newValidateRequest(oneTrackInput: Request): Result[Request] = (
  nameNotBlank(oneTrackInput),
  name50(oneTrackInput),
  emailNotBlank(oneTrackInput)
).mapN( (_, _, _) => oneTrackInput)
```

Monad transformers (1/2)

```
val fooFut : Future[Result[A]] = ???
val barFut : Future[Result[B]] = ???
def doStuff(a: A, b: B): C = ???

//Sans transformer
for {
  foo <- fooFut
  bar <- barFut
} yield {
  for {
    f <- foo
    b <- bar
  } yield doStuff(f, b)
}
```

```
//Avec transformer
{
  for {
    f <- ResultT(fooFut)
    b <- ResultT(barFut)
  } yield
    doStuff(f, b)
}.value
```

Monad transformers (2/2)

```
import scala.concurrent.ExecutionContext.Implicits.global
import cats.instances.future._

case class ResultT[F[_], A](value: F[Result[A]])(implicit T: Monad[F]){
  def map[B](f: A=>B) = ResultT(T.map(value)(_.map(f)))
  def flatMap[B](f: A=> Result[B]) = ResultT(T.map(value)(_.flatMap(f)))
}

object ResultT{
  def pure[F[_], A](result: Result[A])(implicit F: Monad[F]): ResultT[F, A] = ResultT[F, A](F.pure(result))
  def success[F[_], A](value: A)(implicit F: Monad[F]): ResultT[F, A] = pure[F, A](Success[A](value))
  def failure[F[_], A](message: String)(implicit F: Monad[F]): ResultT[F, A] = pure[F, A](Failure[A]
(message))
}

def sendEmail(request: Request): Future[Result[Request]] = ???

def executeUseCase(input: Request): Future[String] = {
  for {
    validated       <- ResultT.pure(validateRequest(input))
    canonicalized <- ResultT.success(canonicalizeEmail(validated))
    persisted       <- ResultT.success(tee(updateDb)(canonicalized))
    sent            <- ResultT(sendEmail(persisted))
  } yield "Success"
}.value.map(returnMessage)
```

Merci à tous !
Thanks Scott Wlaschin !


rrramirooo


rrramiro

http://github.com/rrramiro/rop/

https://fsharpforfunandprofit.com/rop/