Functional Programming In SCALA @ING Bank

Alexandru Nedelcu & Mihai Simu

November 2021

do your thing

Vesper @ ING

1. Romania's payment engine
   - With ambitions for more

2. In production
   - And heavily refactored 😛

3. We have superpowers 😎

Vesper @ ING

Distributed / remote team

Vesper @ ING

Distributed / remote team

Scala / JVM

Vesper @ ING

Distributed / remote team

Scala / JVM

Functional programming

Vesper @ ING

Distributed / remote team

Scala / JVM

Functional programming

Akka

- Streams
- Cluster
- Event Sourcing

Vesper @ ING

Distributed / remote team

Scala / JVM

Functional programming

Akka

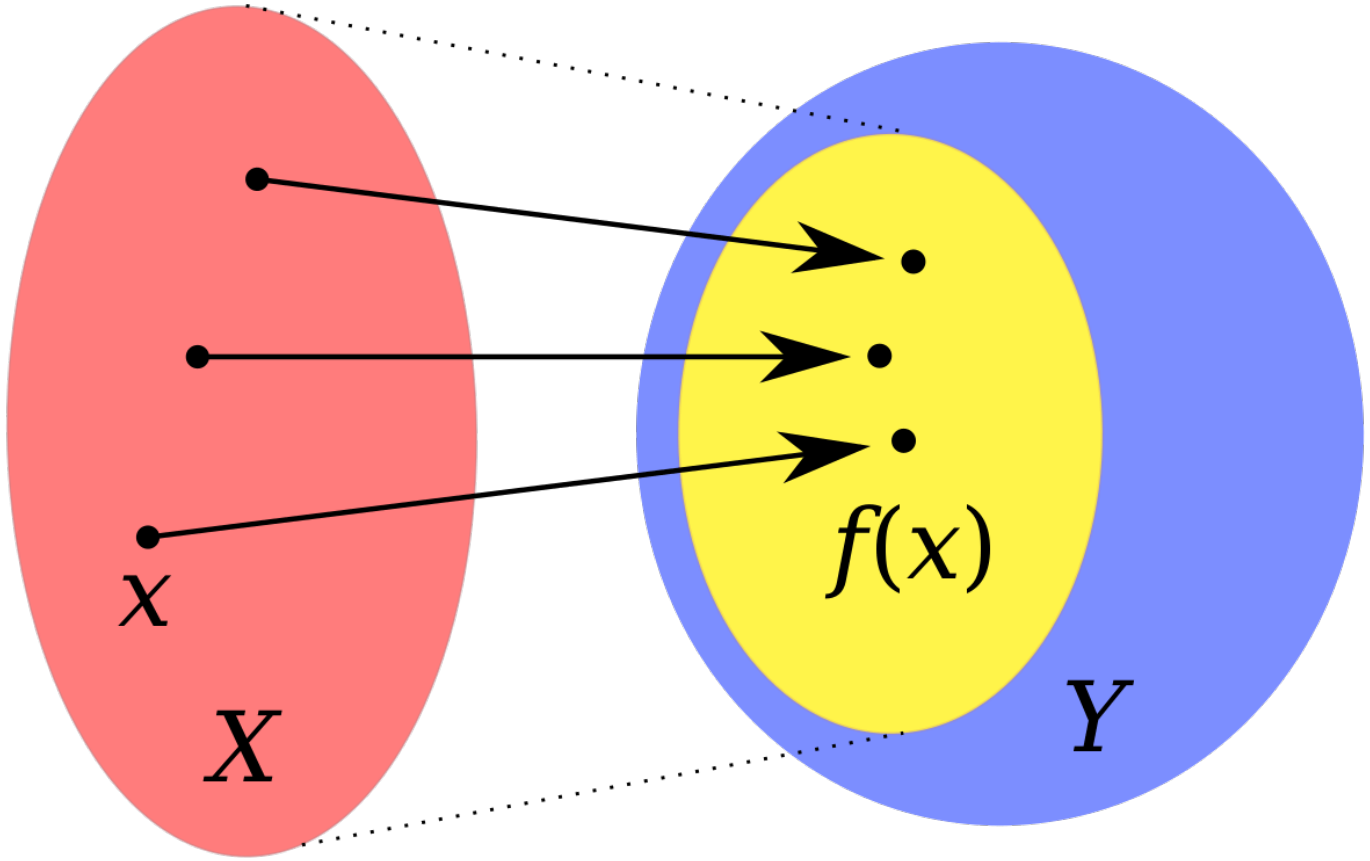
- Streams
- Cluster
- Event Sourcing

Reusable components

Resiliency, consistency, horizontal scalability

Functional Programming (FP)

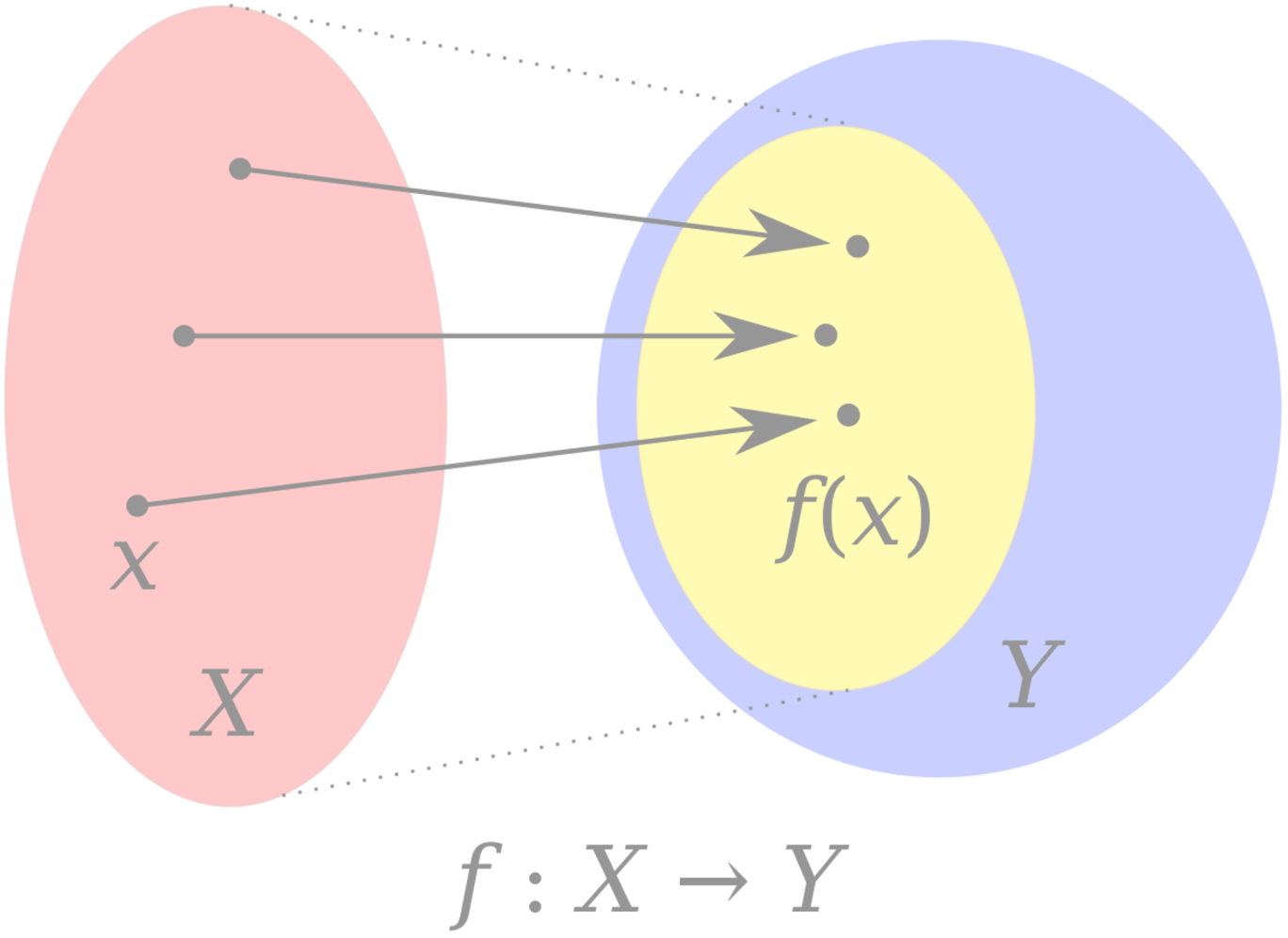
Functional Programming



$f : X \to Y$

f: X → Y, ∀ x_1, x_2 ∈ X
$f(x_1) \neq f(x_2) \Rightarrow x_1 \neq x_2$

Functional Programming



f: X → Y, ∀ x_1, x_2 ∈ X
$f(x_1) \neq f(x_2) \Rightarrow x_1 \neq x_2$

Functional Programming

- Programming with **math functions**
  - Aka "*pure functions*", or functions without side-effects
- Programming with values
  - Aka immutable data-structures

Referential Transparency

*An expression is called **referentially transparent** if it can be replaced with its corresponding value (and vice-versa) without changing the program's behavior.*

Referential Transparency

```
1
2   val r1 = foo(p)
3   val r2 = foo(p)
4
5   List(r1, r2)
6
```

—

```
1
2   val r = foo(p)
3
4   List(r, r)
5
6
```

Referential Transparency

```
1
2   import scala.math.log
3
4   def log2(x: Double): Double =
5     log(x) / log(2)
6
```

Referential Transparency

```
1
2   import scala.math.log
3
4   val lnOf2 = log(2)
5
6   def log2(x: Double): Double =
7     log(x) / lnOf2
8
```

Functional Programming :: Example

```scala

trait DelayedQueue[F[_], A] {
  def offer(key: String, payload: A, scheduleAt: OffsetDateTime): F[OfferOutcome]

  def tryPoll: F[Option[AckEnvelope[F, A]]]

  // ...
}

```

```scala
val tryPoll: IO[Option[AckEnvelope[IO, A]]] = {
  def loop: IO[Option[AckEnvelope[IO, A]]] =
    TimeUtils.currentDateTime.flatMap { now ⇒
      selectFirstAvailable(A.kind, now).flatMap {
        case None ⇒ IO.pure(None)
        case Some(row) ⇒
          acquireTableRow(row, now).flatMap {
            case false ⇒ loop // retry
            case true ⇒
              A.deserialize(row.payload) match {
                case Left(e) ⇒ IO.raiseError(e)
                case Right(payload) ⇒
                  Some(
                    AckEnvelope(
                      message = payload,
                      messageID = MessageId(row.pKey),
                      acknowledge = acknowledge(row),
                      receivedAt = now,
                      source = "delayedQueue"
                    ))
              }
          }
      }
    }
  loop // start
}
```

1

2

3

Scala

- Static type system (really static ☺)
- Culture oriented towards FP
- Optimal language for FP
  - Expression based
  - “Union types”
  - “Higher-kinded types”
  - “Type-classes”
  - Typelevel ecosystem

Scala

- Static type system (really static ☺)
- Culture oriented towards FP
- Optimal language for FP
  - Expression based
  - “Union types”
  - “Higher-kinded types”
  - “Type-classes”
  - Typelevel ecosystem

Functional Programming + Scala

- Reduced defects rate [1] [2]

1. A Large Scale Study of Programming Languages and Code Quality in Github
2. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

Functional Programming + Scala

- Reduced defects rate
- Easier maintenance (tests, refactoring)

Functional Programming + Scala

- Reduced defects rate
- Easier maintenance (tests, refactoring)
- Local reasoning
- Mathematical rigor

Functional Programming + Scala

- Reduced defects rate
- Easier maintenance (tests, refactoring)
- Local reasoning
- Mathematical rigor
- We're hiring great people
- We're still learning
- It's fun!

Payment-processing

Input Queue
- Contains transactions
- Must delete from queue when finished

Perform Processing
- Debit and credit accounts

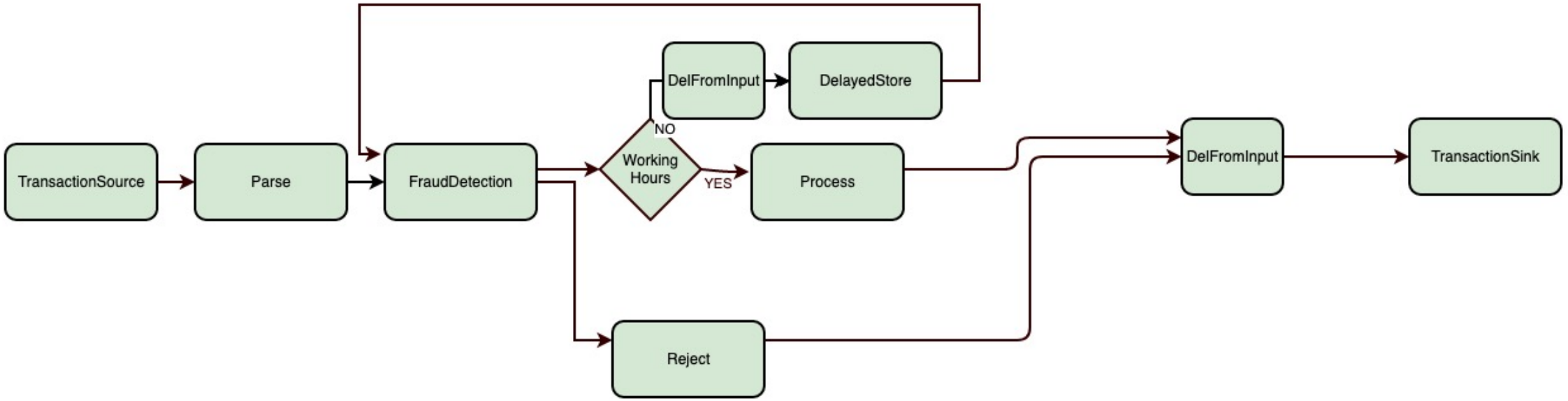
Destination Queue
- Transaction is considered finished

Payment-processing - example

- Transactions are received as String
- Processor must
  - parse
  - check fraud-detection
  - processes only during working hours
  - delete from queue when finished
  - be processed at-most-once
- Errors can occur and should be handled

Payment-processing - example

- Transactions are received as String
- Processor must
  - parse
  - check fraud-detection
  - processes only during working hours
  - delete from queue when finished
  - be processed at-most-once
- Errors can occur and should be handled



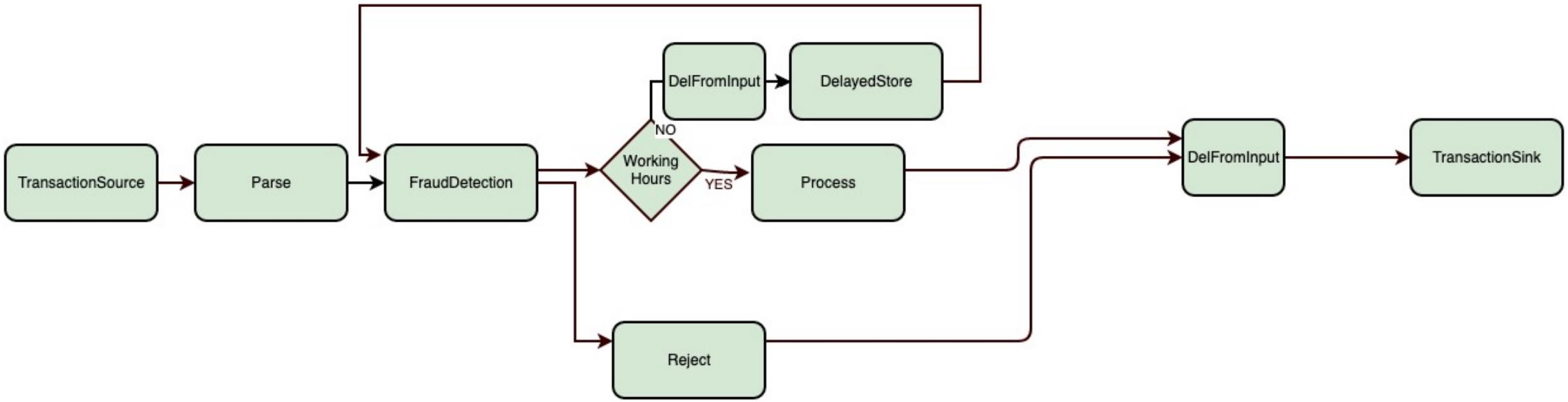
Payment-processing code (1)

```scala
def extractFromSource: IO[TString]
def parse(input: TString): IO[TParsed]
def verifyFraud(parsed: TParsed): IO[Either[TFraudInfo, TVerified]]

def duringWorkHours(): Boolean

def process(verified: TVerified): IO[TProcessed]
def reject(rejected: TFraudInfo): IO[TRejected]

def pushToDelayedStore(verified: TVerified): IO[Unit]
```



Payment-processing with Akka-Streams

- Each step is a Stream component

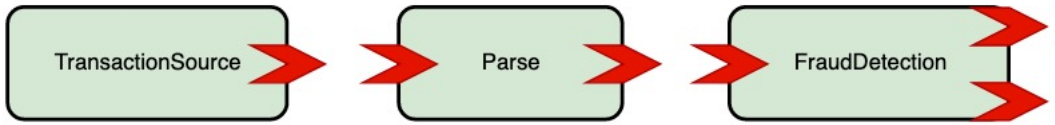
```
val source: SourceShape[TString] = ???
val parse: FlowShape[TString, TParsed] = ???
val fraudDetection: FanIn1FanOut2Shape[TParsed, TVerified, TFraudInfo] = ???
```

Payment-processing with Akka-Streams

- Each step is a Stream component

```
val source: SourceShape[TString] = ???
val parse: FlowShape[TString, TParsed] = ???
val fraudDetection: FanIn1FanOut2Shape[TParsed, TVerified, TFraudInfo] = ???
```

- Components are black-boxes with ports

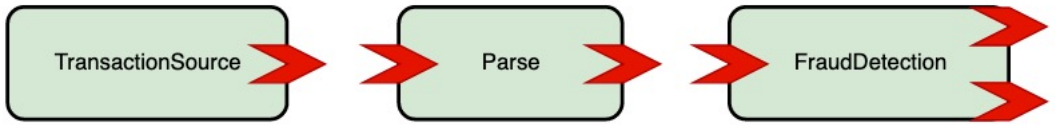


Payment-processing with Akka-Streams

- Each step is a Stream component

```
val source: SourceShape[TString] = ???
val parse: FlowShape[TString, TParsed] = ???
val fraudDetection: FanIn1FanOut2Shape[TParsed, TVerified, TFraudInfo] = ???
```

- Components are black-boxes with ports



- Compose Components with GraphDSL
  - Types of ports must match

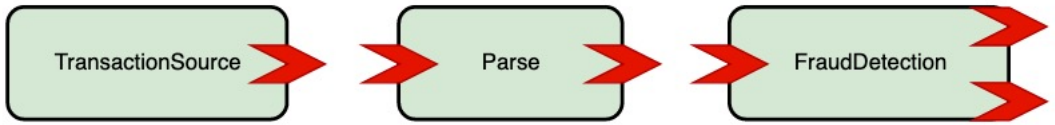
```
source ~> parse ~> fraudDetection ~> …
```

Payment-processing with Akka-Streams

- Each step is a Stream component

```
val source: SourceShape[TString] = ???
val parse: FlowShape[TString, TParsed] = ???
val fraudDetection: FanIn1FanOut2Shape[TParsed, TVerified, TFraudInfo] = ???
```

- Components are black-boxes with ports

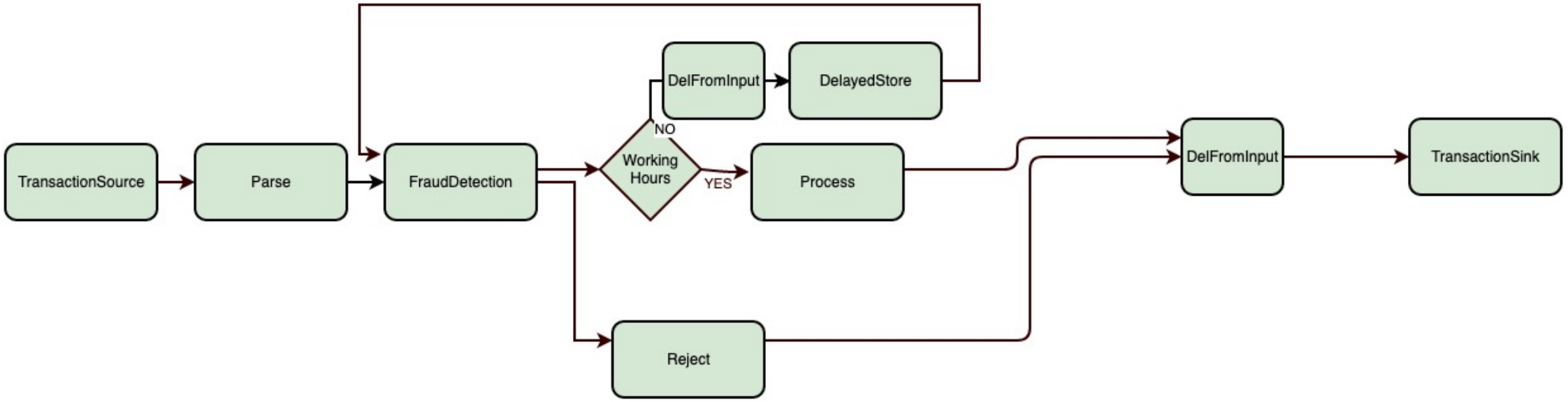


- Compose Components with GraphDSL
  - Types of ports must match

```
source ~> parse ~> fraudDetection ~> …
```

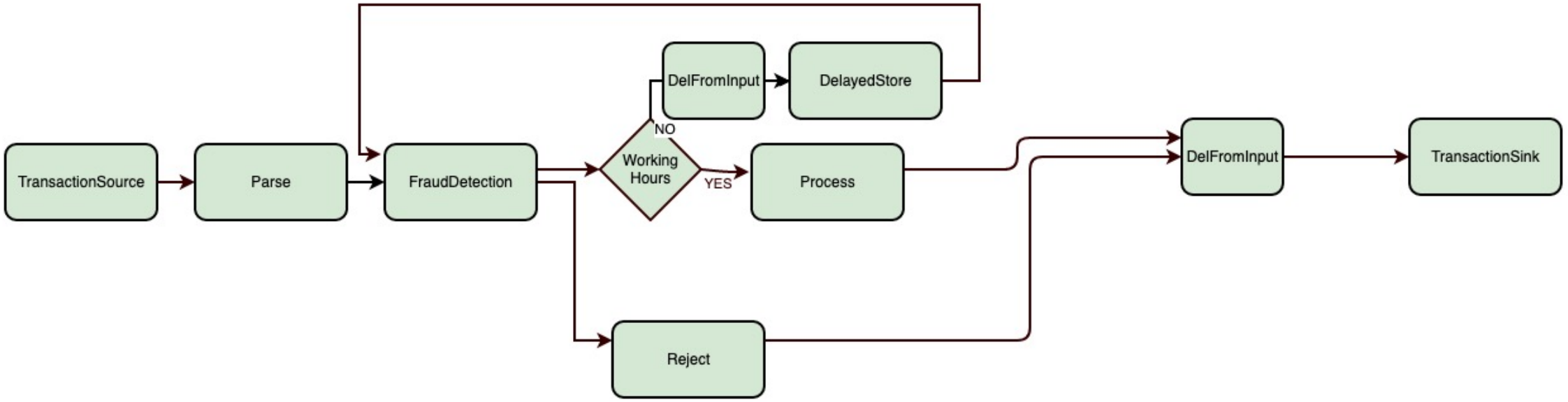
- Messages passed asynchronously between components

Payment-processing code (2)



Payment-processing code (2)

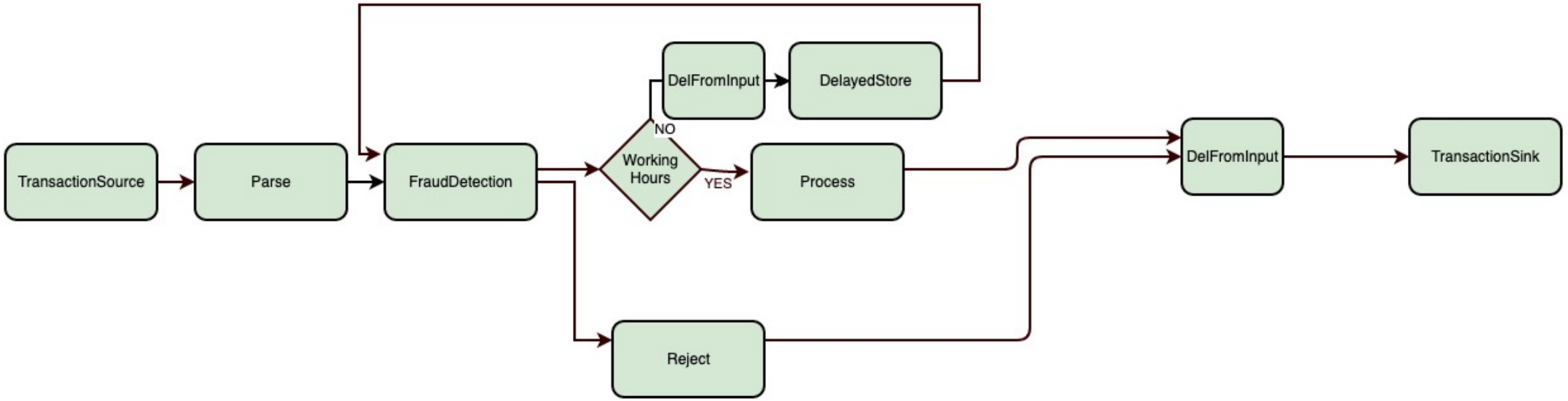
```
                    M1 <~ delayedStoreOut
                                                workHrs.o2 ~> del1 ~> delayedStoreIn
source ~> parse ~> M1 ~> fraudD; fraudD.o1 ~> workHrs; workHrs.o1 ~> process ~> MEnd ~> del2 ~> sink
                                 fraudD.o2 ~> reject ~> MEnd
```



Payment-processing error-handling

```
                    M1 <~ delayedStoreOut
                                                workHrs.o2 ~> del1 ~> delayedStoreIn
source ~> parse ~> M1 ~> fraudD; fraudD.o1 ~> workHrs; workHrs.o1 ~> process ~> MEnd ~> del2 ~> sink
                                 fraudD.o2 ~> reject ~> MEnd
```

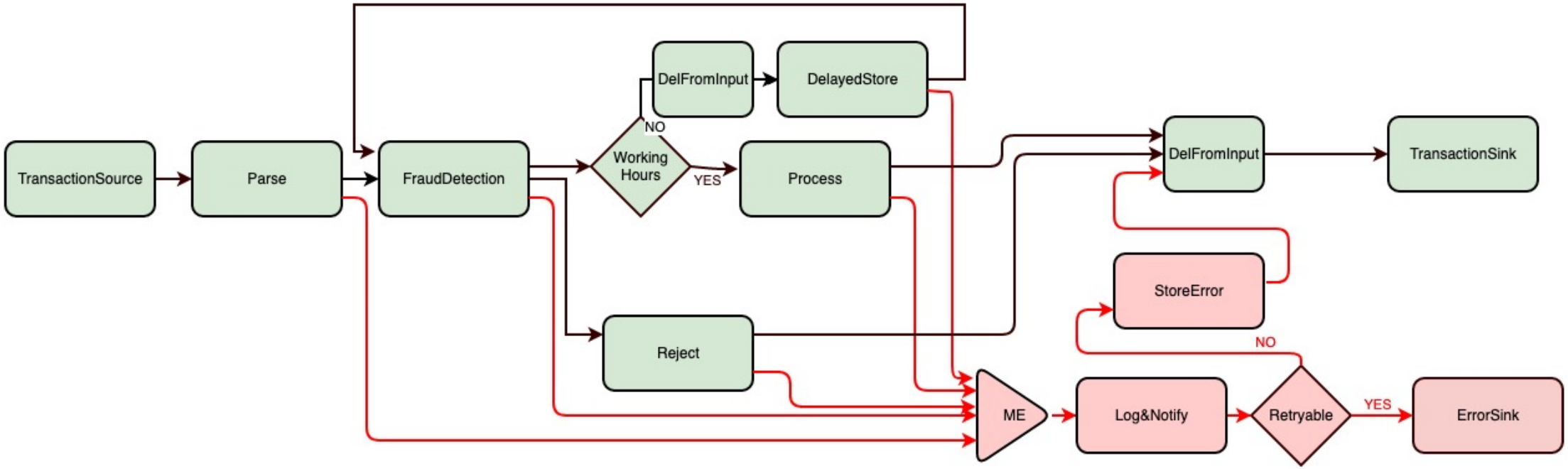
- Assume each step can fail
- Differentiate
  - Retryable errors (eg: external service unavailable)
  - NonRetryable errors (eg: parsing exception)



Payment-processing error-handling

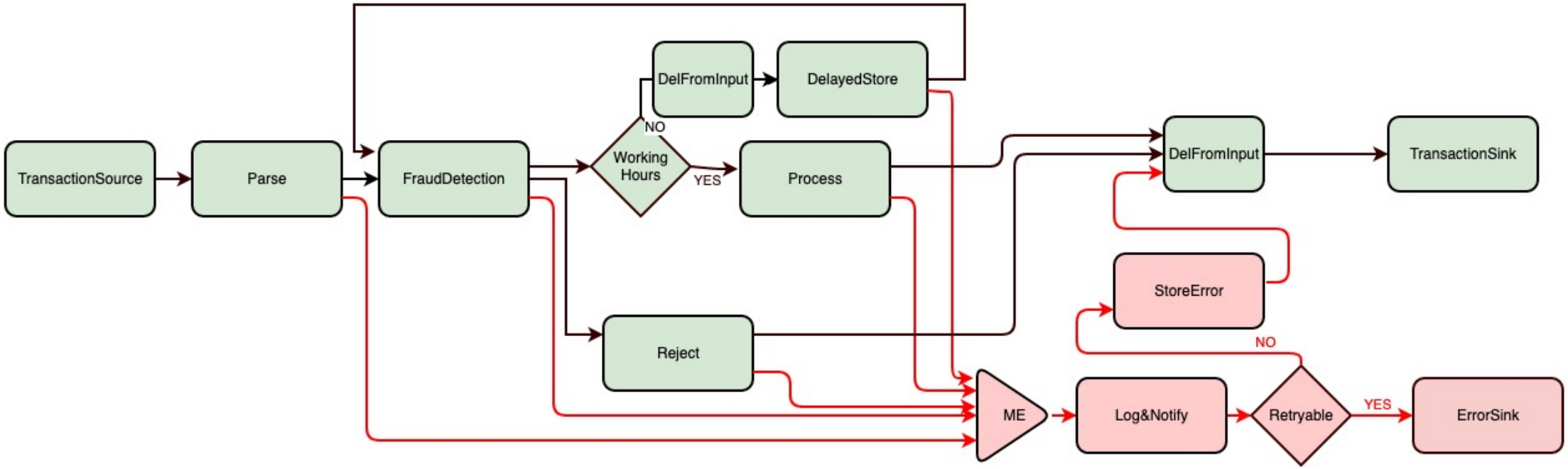
```
                    M1 <~ delayedStoreOut
                                                workHrs.o2 ~> del1 ~> delayedStoreIn
source ~> parse ~> M1 ~> fraudD; fraudD.o1 ~> workHrs; workHrs.o1 ~> process ~> MEnd ~> del2 ~> sink
                                 fraudD.o2 ~> reject ~> MEnd
```

- Assume each step can fail
- Differentiate
  - Retryable errors (eg: external service unavailable)
  - NonRetryable errors (eg: parsing exception)



Payment-processing code (3)

```
                            M1 <~ delayedStoreOut
                                                          workHrs.o2 ~> del1 ~>  delayedStoreIn
source ~> parse; parse.o1 ~> M1 ~> fraudD; fraudD.o1 ~> workHrs; workHrs.o1 ~> process; process.o1 ~> MEnd ~> del2 ~> sink
                                       fraudD.o2 ~> reject; reject.o1 ~> MEnd
          parse.oErr ~> MErr
                                       fraudD.oErr ~> MErr
                                                   reject.oErr ~> MErr
                                                                                process.oErr ~> MErr
                                                                                delayedStoreErr ~> Merr
// Error Flow
MErr ~> logNotifyErrors ~> splitErr
                           splitErr.o1 ~> storeErr ~> MEnd
                           splitErr.o2 ~> errorSink
```

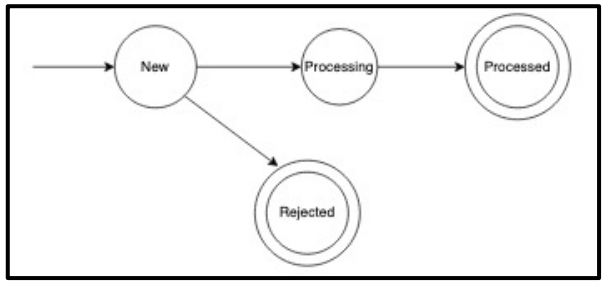


Payment-processing with Akka-Streams

- Main unit of abstraction = Flow component
  - Contain pure FP code for business logic and expose ports
  - Frequent tension: what to model with components vs FP code

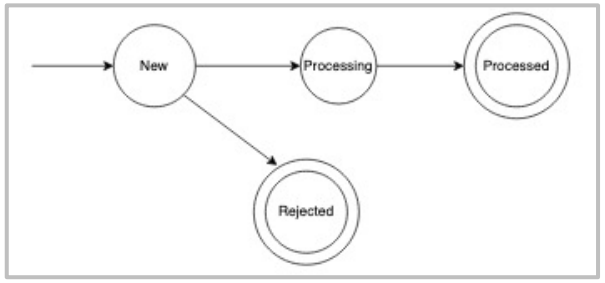
Payment-processing with Akka-Streams

- Main unit of abstraction = Flow component
  - Contain pure FP code for business logic and expose ports
  - Frequent tension: what to model with components vs FP code
- Components can be arbitrarily complex, eg:
  - Call HTTP Service/ database query
  - Maintain FSM of transaction-state



Payment-processing with Akka-Streams

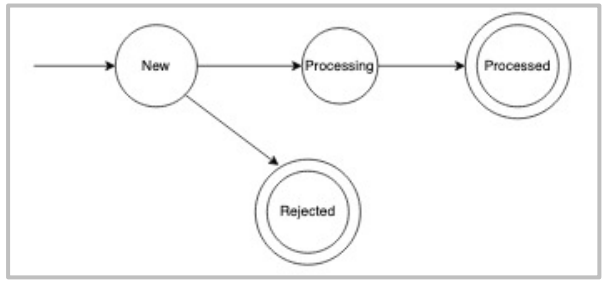
- Main unit of abstraction = Flow component
  - Contain pure FP code for business logic and expose ports
  - Frequent tension: what to model with components vs FP code
- Components can be arbitrarily complex, eg:
  - Call HTTP Service/ database query
  - Maintain FSM of transaction-state



- We allow transactions to be replayed if RetryableErrors appear
  - eg: service for verifying Fraud is temporarily unavailable
  - => components must be idempotent

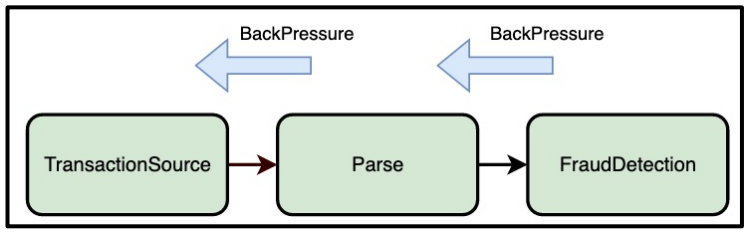
Payment-processing with Akka-Streams

- Main unit of abstraction = Flow component
  - Contain pure FP code for business logic and expose ports
  - Frequent tension: what to model with components vs FP code
- Components can be arbitrarily complex, eg:
  - Call HTTP Service/ database query
  - Maintain FSM of transaction-state



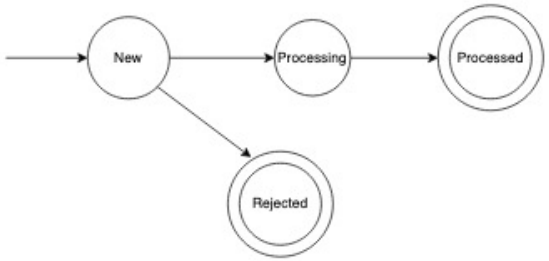
- We allow transactions to be replayed if RetryableErrors appear
  - eg: service for verifying Fraud is temporarily unavailable
  - => components must be idempotent

- Built-in backpressure



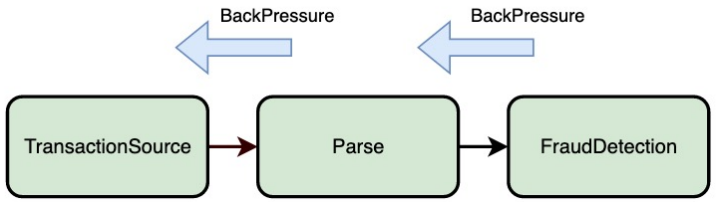
Payment-processing with Akka-Streams

- Main unit of abstraction = Flow component
  - Contain pure FP code for business logic and expose ports
  - Frequent tension: what to model with components vs FP code
- Components can be arbitrarily complex, eg:
  - Call HTTP Service/ database query
  - Maintain FSM of transaction-state



- We allow transactions to be replayed if RetryableErrors appear
  - eg: service for verifying Fraud is temporarily unavailable
  - => components must be idempotent

- Built-in backpressure



- Scale horizontally with akka-cluster + sharding

Scala and FP – closing remarks

Example: Generate the Fibonacci numbers

```scala
def basicGeneratorImpure(n: Int): Array[Int] = {
  val arr = new Array[Int](n)
  arr(0) = 1
  arr(1) = 1
  var i = 2
  while (i < n) {
    arr(i) = arr(i-1) + arr(i-2)
    i = i + 1
  }
  arr
}
```

```scala
def pureGenerator(n: Int): List[Int] = {
  @tailrec
  def loop(acc: List[Int], i: Int): List[Int] = {
    acc match {
      case h1 :: h2 :: t if i < n ⇒
        loop((h1 + h2) :: h1 :: h2 :: t, i + 1)
      case other ⇒ other
    }
  }
  loop(List(1, 1), 2).reverse
}
```

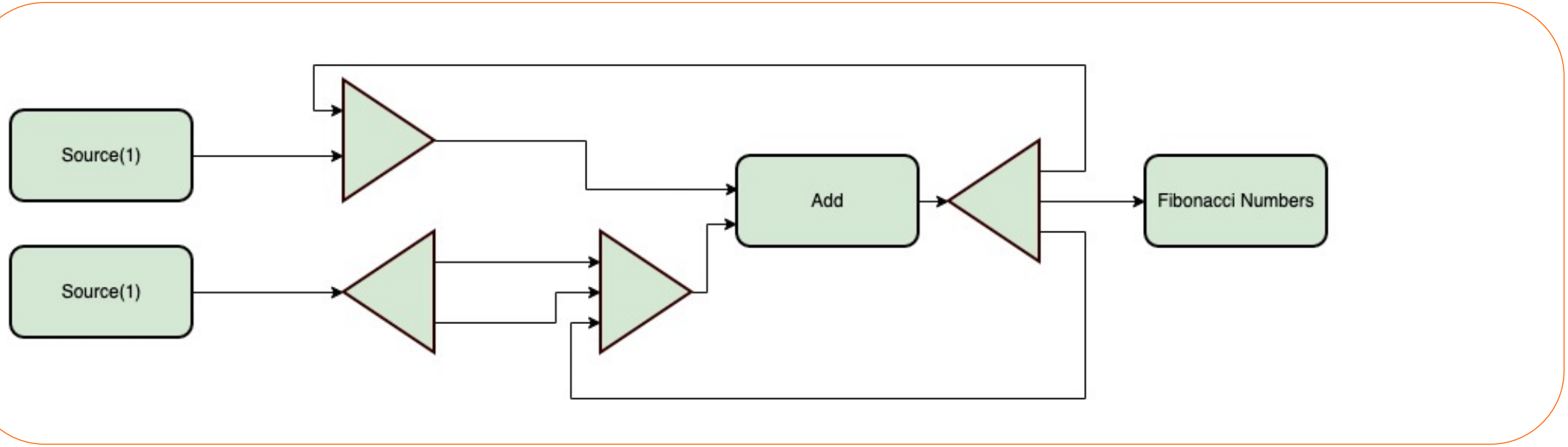
Scala and FP – closing remarks

Example: Generate the Fibonacci numbers

```
1  val fiboGraph = GraphDSL.create() { implicit builder ⇒
2    import GraphDSL.Implicits._
3    val zip = builder.add(ZipWith[Int, Int, Int]((a, b) ⇒ a + b))
4    val mergeUp = builder.add(MergePreferred[Int](1))
5    val mergeDown = builder.add(MergePreferred[Int](2))
6    val broadcastIn = builder.add(Broadcast[Int](2))
7    val broadcastOut = builder.add(Broadcast[Int](3))
8
9    mergeUp.out ⇝ zip.in0
10   broadcastOut.out(0) ⇝ mergeUp.in(0)
11   broadcastOut.out(2) ⇝ mergeDown.in(2)
12   broadcastIn.out(0) ⇝ mergeDown.in(0)
13   broadcastIn.out(1) ⇝ mergeDown.in(1)
14   mergeDown.out ⇝ zip.in1
15   zip.out ⇝ builder.add(Flow[Int].map(x ⇒ {Thread.sleep(100); x})) ⇝ broadcastOut
16   UniformFanInShape(broadcastOut.out(1), mergeUp.in(1), broadcastIn.in)
17 }
```

Scala and FP – closing remarks

Example: Generate the Fibonacci numbers

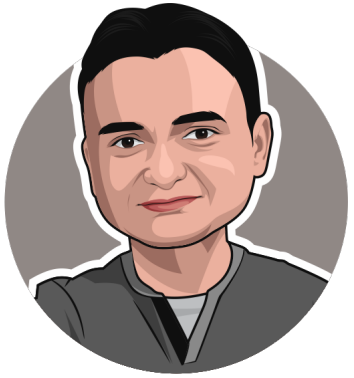


Scala and FP - choosing an ecosystem

- What you can build with it
- Library/Platform Support
- How it fits in problem-domain
- Corectness guarantees
- Fun

Questions?

We're hiring 😉



Alexandru Nedelcu
alexandru.nedelcu@ing.com

Mihai Simu
mihai-stelian.simu@ing.com

*Nu uitați că avem și un concurs pregătit pentru antreprenorii prezenți la eveniment. Am pregătit **10 pachete ING FIX pentru 12 luni consecutive, pentru 10 antreprenori**. Toți participanții eligibili la concurs vor primi gratuit, automat, un pachet ING FIX, pentru o perioadă de până la două luni, pentru a testa serviciul ING.*

*Dacă vreți să vă înscrieți, mergeți la standul ING unde, la descrierea companiei veți găsi un tab cu numele **Concurs**, unde este formularul de înscriere.*

do your thing