Actors and Functional Reactive Programming

Diego E. Alonso

Typelevel Summit Lausanne

I will talk about Functional Reactive Programming

- **Message:** FRP constructs and idioms are more or less like Actors
- **Goal:** Help you to *understand* FRP using a loose analogy between Actors and a simple Scala implementation
- **Non goal**: promote concurrency or design approaches in Scala or present any Scala library you should be starting to use

47

Object design patterns and `cats` type-classes

47

Object design patterns and `cats` type-classes

- Problem: use an existing implementation that is not compatible

```
trait Printer[A] {
  def print(i: Int): Unit                    }
```

- Our Problem: we need a `Printer[Int]`, we only have one for `String`

```
class StringPrinter() extends Printer[String] {
  def print(s: String): Unit = /***/         }
```

47

Object design patterns and `cats` type-classes

- Problem: use an existing implementation that is not compatible

```
trait Printer[A] {
  def print(i: Int): Unit                    }
```

- Our Problem: we need a `Printer[Int]`, we only have one for `String`

```
class StringPrinter() extends Printer[String] {
  def print(s: String): Unit = /***/          }
```

- **Adapter**: object that implements uses another object to implement interface

```
class InputConverterIntPrinter(
    base: Printer[String])  extends Printer[Int]      {
  def print(b: Int): Unit = base.print(b.toString)    }

class InputConverterPrinterAdapter(
    base: Printer[A], fun: B => A) extends Printer[B]{
  def print(b: B): Unit = base.print(f(b))            }
```

47

Object patterns and cats

- We can combine it with a pattern of generic factory

```
// We could also use a factory object trait
trait InputConverterPrinterAdapterFactory {
  def buildInputAdapter[A, B](
    base: Printer[A], f: B => A): Printer[B]          }
```

47

Object patterns and cats

- We can combine it with a pattern of generic factory

```
// We could also use a factory object trait
trait InputConverterPrinterAdapterFactory {
  def buildInputAdapter[A, B](
    base: Printer[A], f: B => A): Printer[B]          }
```

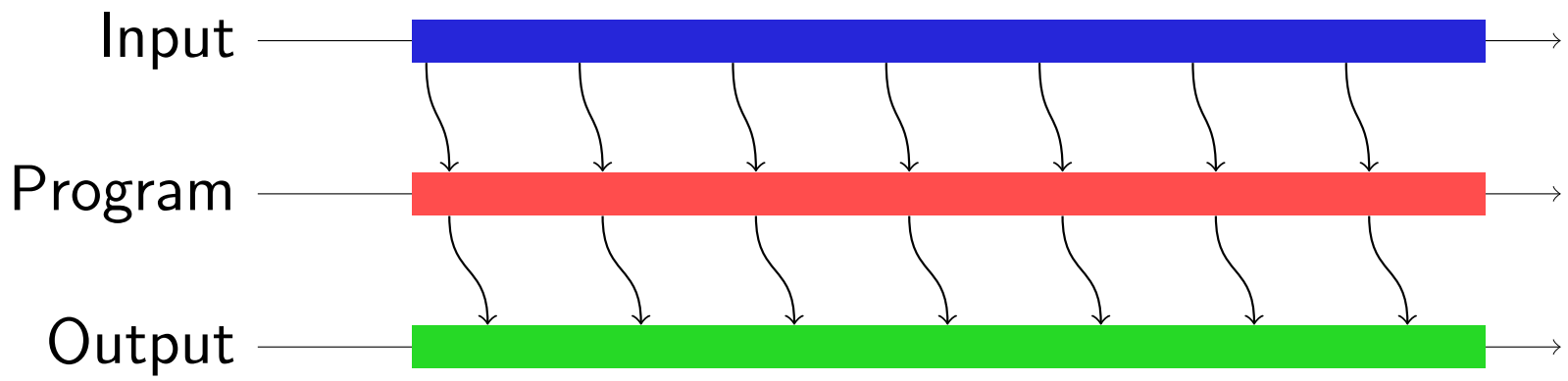
- In cats, we do the same. We just call it a Contravariant Functor.

```
trait InputConverterGenericFactory[F[_]]{
  def buildInputAdapter[A, B](
    base: F[A], f: B => A): F[B]
}
```

```
trait Contravariant[F[_]] {
  def contraMap[A, B](base: F[A], f: B => A): F[B]
}
```

47

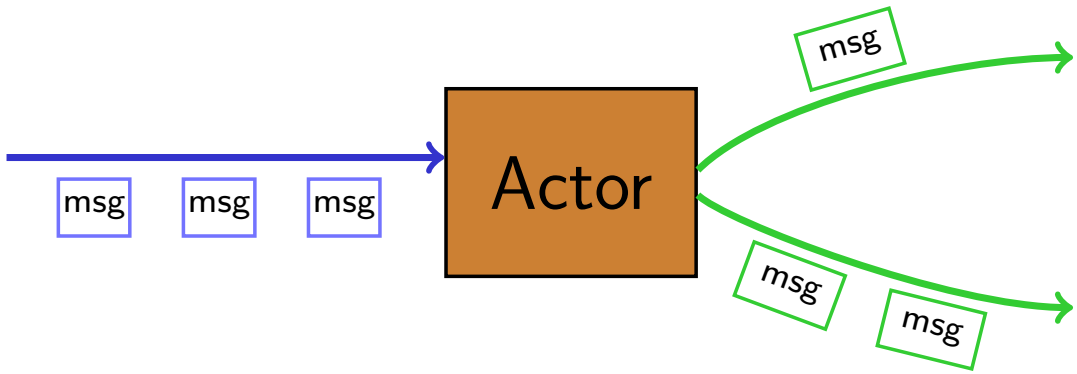
Reactive Programming



- A reactive system is a runs indefinitely (no termination)
- Every time it may take a new input (react to events)
- Every time it may emit a new outputs (actions changes)

47

Reactive Programming with Actors



- An Actor is a runtime entity that is activated when it receives a **message,** which is also referred to as "reacting to an event".
- Processes each message, one at a time, in order they arrive
  - It may send messages to other actors
  - It changes behaviour to process next message
- Actors are **operational** approach: what programs *does* and react

Functional? Reactive Programming

- Functional Programming: *programs as functions* of input to output
- But what is the input and output *value* of a reactive program?

47

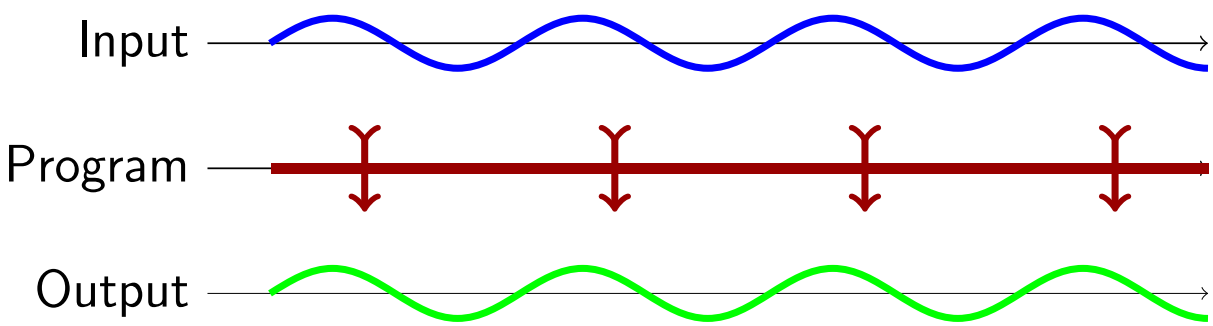
Functional? Reactive Programming

- Functional Programming: *programs as functions* of input to output
- But what is the input and output *value* of a reactive program?

- A Signal: value at each instant in time of a variable:
  air pressure (audio), mouse position, price of stock
- In practice, signals are approximated with streams

47

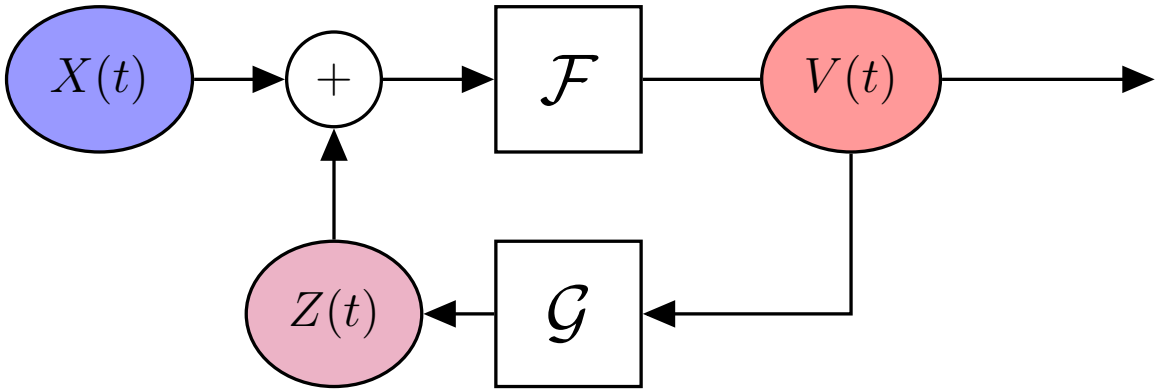
Functional Reactive Programming



- FRP: reactive programs as functions of input to output signals
- Declarative approach: what programs *are*, not what they do

47

Functional Reactive Programming



- An FRP program describes a set of signals and signal functions
- Signal functions show how signals depend on each other time
- FRP language/library: how to define simple signal functions...
- ...and how to combine signal functions into bigger programs

47

Monadic Stream Functions

47

Monadic Stream Functions

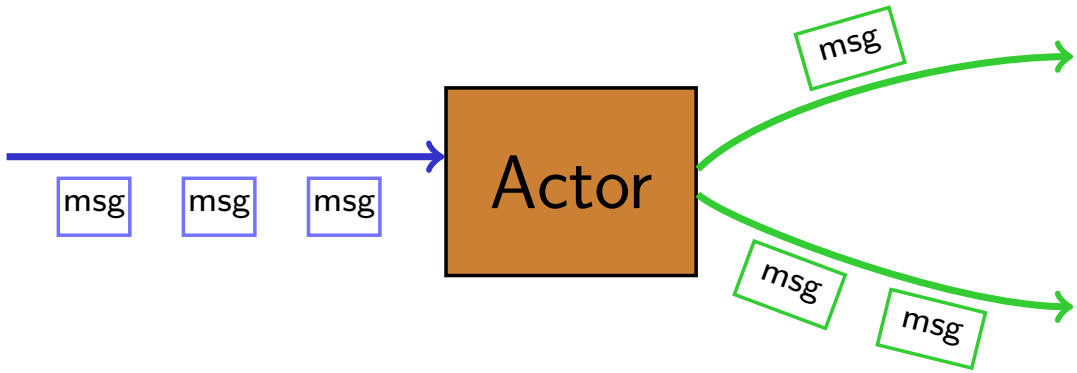
```
trait MoSF[M[_], I, O] {
  def apply(i: I): M[(O, MoSF[M, I, O])]
  //                      continuation
}
```

- Represent single step in stream transformation
- **Polymorphic:** the effect type `M[_]` is polymorphic, and we use type-classin operations.
- **Purely-Functional**: it does not assume, in the definition and operations, any kind of destructive update of the world or global variable, not even time.

47

Monadic Stream Function like actor

```
trait MoSF[M[_], I, O] {
  def apply(i: I): M[(O, MoSF[M, I, O])]
  //                      continuation
}
```



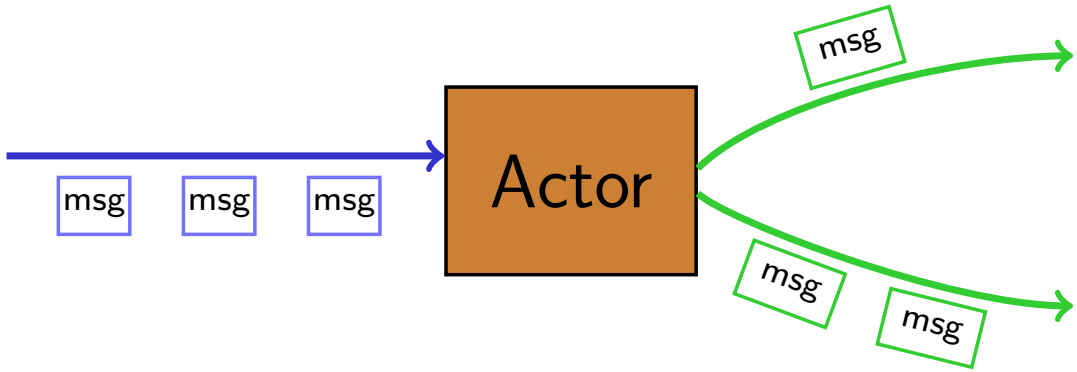
- Inputs: Received messages corresponds to the *In* input type
- Sent messages correspond to the *Out* output type
- **Continuation**: modified actor behaviour, for next messages

Running a Monadic Stream Function

```
trait MoSF[M[_], I, O] { self =>

  def apply(i: I): M[(O, MoSF[M, I, O])]

  def run(ins: List[I])(implicit M: Monad[M]): M[List[O]] =
    ins match {
      case Nil          => M.pure(Nil)
      case (in :: ins) => for {
        (out, cont) <- self(in)
        outs        <- cont.run(ins)  //use continuation on tail
      } yield out :: outs
    }
```

The meaning of executing an MSF is given by applying it to the first sample of an input stream, obtaining the first output, and using the continuation to transform the tail of the stream.(BttF)

Simple Pattern Stateless Actors



- Stateless actors have no inner mutable state or data.
- They always process each message in the same way (locally)
- Processing message never modifies behaviour
- May use or ignore message
- May involve any effectful action (reading DB)

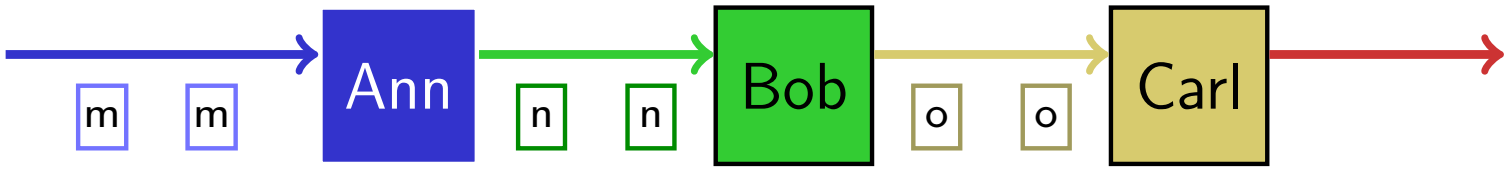
47

Stateless Monadic Stream Functions

```
import cats.{Functor, Applicative => App}

// outputs pure value o
def pure[M[_], I, O](o: O)(implicit M: App[M]): MoSF[M, I, O] =
  (i: I) => M.pure(o -> pure(o))

// outputs result of pure function
def arr[M[_],I,O](f: I => O)(implicit M: App[M]): MoSF[M, I, O]=
  (i: I) => M.pure(f(i) -> arr(f))

// gets output value from effect M
def liftM[M[_]: Functor, I, O](fo: M[O]): MoSF[M, I, O] =
  (_: I) => fo.map(o => o -> liftM(fo))

// applies effect function to input
def liftK[M[_]: Functor, I, O](fk: I => M[O]): MoSF[M, I, O] =
  (i: I) => fk(i).map(o => o -> liftK(fk))
```

- Self-recurring MSF correspond to Stateless actors
- The continuation part of their `apply` method is itself

A pipeline of Actors



- Big processing made of several steps, each step is an actor
- Each run of pipeline may alter each actor.

> *In a Pipes and Filters architecture, you compose a process by chaining together a number of processing steps. Each of the steps is decoupled from the others. Thus, each of the steps could be replaced or rearranged as the need arises. (VV)*

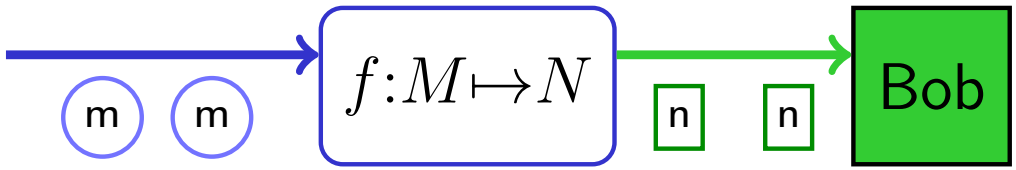
A pipeline of MSF


```
def >>>[M[_]: Monad, I, O, P](
  pref: MoSF[M, I, O],
  post: MoSF[M, O, P]
):      MoSF[M, I, P] =
  (i: I) => for {
    (o, prefCont) <- pref(i)
    (p, postCont) <- post(o)
  } yield p -> (prefCont >>> postCont)
```

- A `MoSF` built from two other ones: applies the first, applies the result to the second one, and ouputs the result
- Continuation is composition of continuations

47

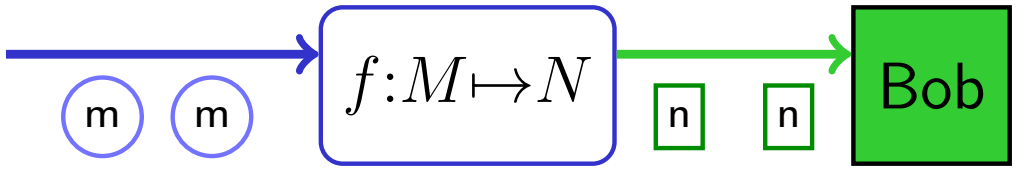
Actor pattern: Message translator



- **Message Translator:** transform the data from an incoming message to data that is compatible with the local receiving application.

```
def messageTranslator[M[_]: Monad, O](
  base: MoSF[M, String, O],
): MoSF[M, Int, O] =
  MoSF.arr(_.toString) >>> base
```

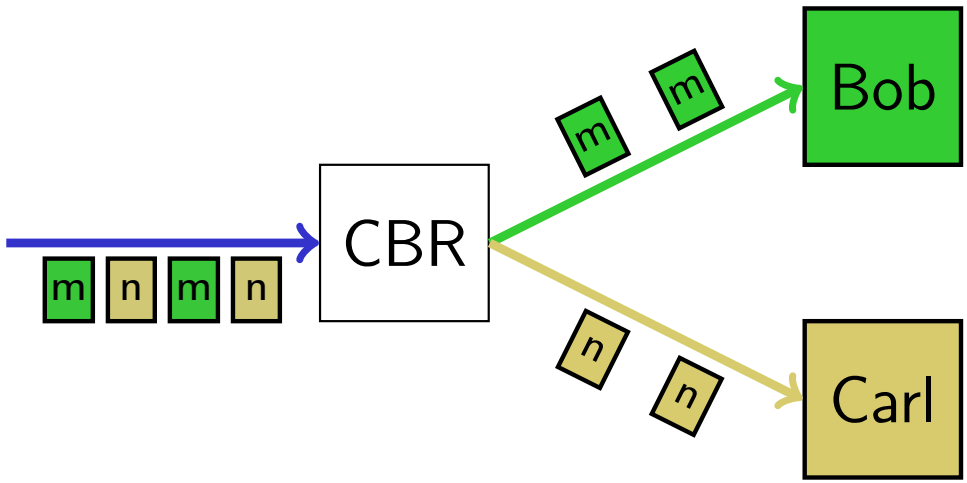
Adapting input/output of MSF




```
import cats.{Functor, Applicative => App}

trait MoSF[M[_], In, Out] { self =>
  def apply(i: In): M[(Out, MoSF[M, I, P]) ]

  def map[P](fun: O => P): MoSF[M, I, P] =
    (i: I) => self(i) map {
      case (o, cont) => fun(o) -> cont.map(fun)      }

  def contraMap[P](fun: H => I): MoSF[M, H, O] =
    (h: H) => self(fun(h)) map {
      case (o, cont) => o -> cont.contraMap(fun)     }
```

47

Actor Pattern: Content-Based router



- A **router** is an actor that re-sends any message it receives to other actors, which may be more specialised, or to balance their workload
- A **content-based router** analyses the content of the message to decide to which actor it sends each message it receives,

Content-Based Routing with MSF

- We use `Either` to parallel-combine two MSF on different types

```
def +++[M[_]: Functor, I, J, O, P](
  onLeft:  MoSF[M, I, O],
  onRight: MoSF[M, J, P]
): MoSF[M, Either[I, J], Either[O, P] =  {
  case Left(i) => onLeft(i).map {
    case (o, lcont) =>
      Left(o) -> (lcont +++ onRight)      }

  case Right(j) => onRight(j).map {
    case (p, rcont) =>
      Right(p) -> (onLeft +++ rcont)      }
}

val cbr = (bob +++ carl)
```

Summary

- Much like some FP type-classes can be seen as restricted cases of object-oriented patterns,
- we can see FRP patterns/idioms using special restricted cases of some actor patterns.
- Unlike actor patterns, which are only on paper and design, but are difficult to see on the code...
- FRP constructs are *operators*, which are in the code.

47

Bibliography / Videography

- *Functional Reactive Programming, Refactored*. Ivan Perez, Manuel Barenz, Henrik Nilsson. Haskell Symposium, 2016.
- *Reactive Messaging Patterns with the Actor Model. Applications and Integration in Scala and Akka*. Vaughn Vernon. 2016
- *The Essence and Origins of Functional Reactive Programming*. Conal Elliott. Lambda Jam 2015.

47