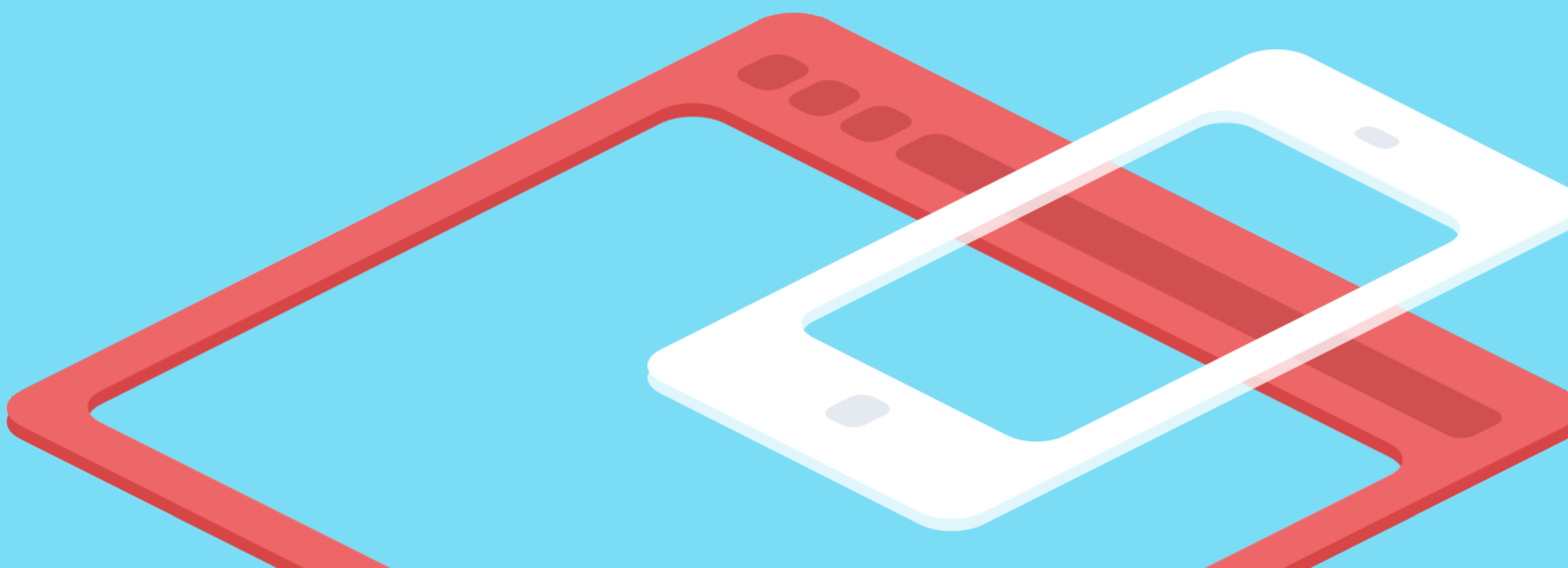
Async, Actors, and IOC

Jim Powers
Twitter: @corruptmemory
Github: corruptmemory

Typesafe



Outline

- Problem
- Actors
  - Example Problem: Scatter/Gather Search
- Async (Library)
- Better ways?

Assumptions

- Familiarity with Scala
    - Interested in learning more
- Familiarity with Actors
- Familiar with some other form of asynchronous programming. And all that implies...

Problem

- Minimize, reduce, eliminate inversion of control - more direct-style of programming
- Reduce mixing of core logic for solving a particular problem from the implementation details of working in an actor-style of programming.
- To the extent possible, maintain the scalability opportunities afforded by Actors.

Desired outcomes

- Start people thinking about this kind of problem
- Finding creative solutions for building the next wave of scalable application - with less pain.

Caveats...

- All very preliminary
- Examining toy problems and toy solutions to explore what's possible
- Nomenclature may be strange/wrong
- Some of this stuff might be re-inventing the wheel - sorry.

Actors

- Abstraction over message passing
- What can they do?
  - Respond to messages
  - Create actors
  - Send messages to actors

Actors

Key point: *message passing is king.*

- No direct access to shared state
- Communication is asynchronous

Actors: Akka (Scala)

Respond to messages

```
class FooActor extends Actor {
 def receive = {
   case "test" ⇒ log.info("received test")
   case _      ⇒ log.info("received unknown message")
 }
}
```

- Typically, Actors are stateful.

Actors: Akka (Scala)

Create actors

```scala
val system = ActorSystem.create(/* ... */)
val searchCacheActorRef =
  system.actorOf(Props(new SearchCache()), "search-cache")
```

```scala
class FooActor extends Actor {
 def receive = {
   case CreateSearchCache ⇒
     val searchCacheActorRef =
       context.actorOf(Props(new SearchCache()), "search-cache")
 }
}
```

Actors: Akka (Scala)

Send messages to actors

```
val searchCacheActorRef =
  system.actorOf(Props(new SearchCache()), "search-cache")

searchCacheActorRef ! Lookup(/* ... */)
```

Actors: Akka (Scala)

Skipping a lot

- Akka actors are supervised
  - Various (including customized) failure/recovery strategies
- Local/remote transparency
- Multiple dispatcher strategies (including customized)
- *And so much more...*

Actors: Example

Scatter/Gather search system

- Material from Josh Suereth's *Introduction to Actors* presentation.

Scatter/Gather

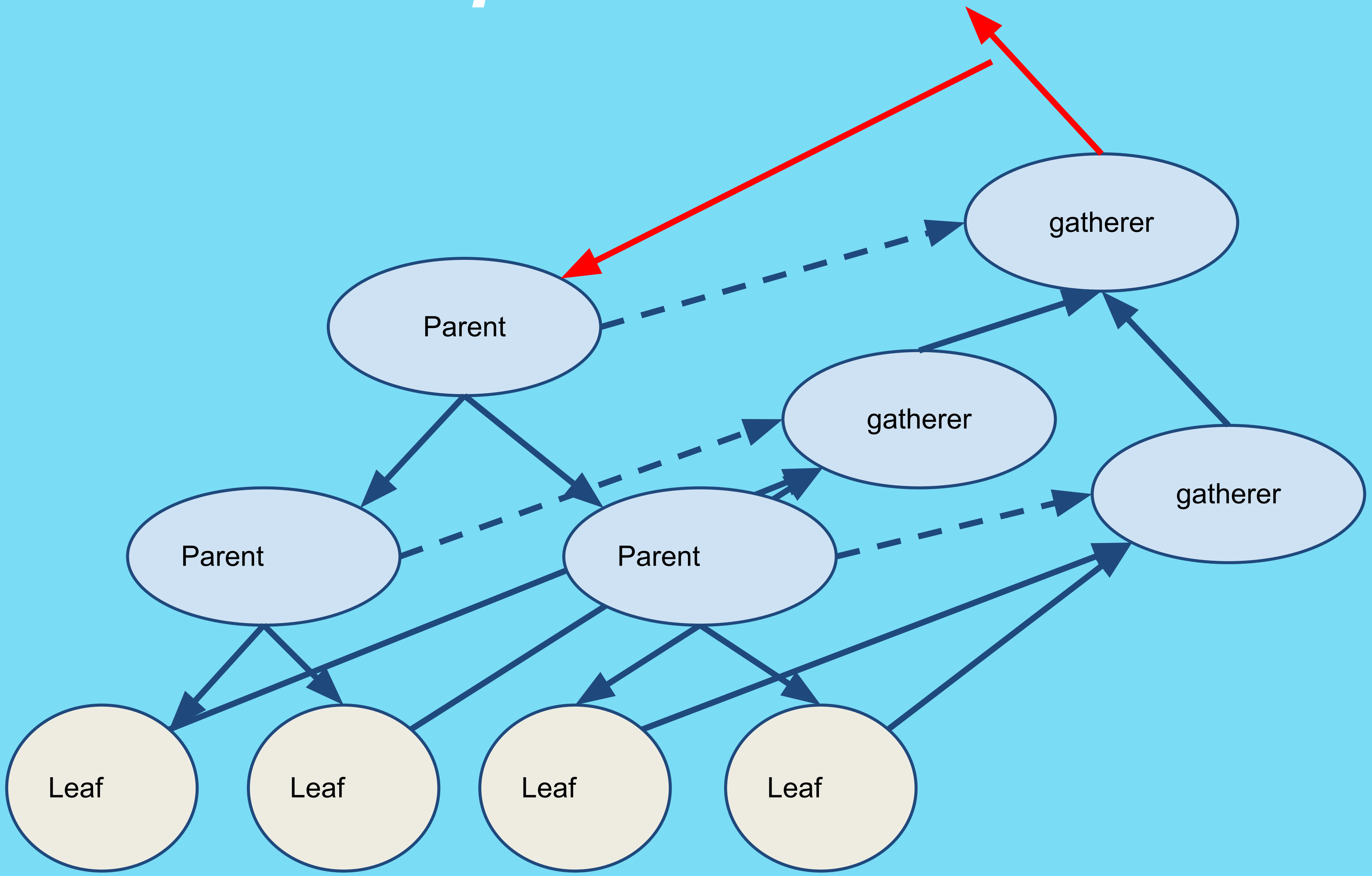
Problem: Scale a search service

Scatter/Gather

Solution:

- Sharded search index
- Send same query to each shard
- Create actor to accumulate partial results
- Send accumulated results back to original requestor

Scatter/Gather



Scatter/Gather

Look at some code

Actors: Observations

- All logic hanging off of callback functions
- Implementation details mixed with "business logic"
- Composition of Actors, er, tricky: "return type" of actors effectively *Nothing*.
- No easy way to get a "big picture"

Hrmmm...

What other asynchronous construct is callback-oriented?

Futures!

Futures

```
val fut = future {/* body run asynchronously */}

fut.onComplete(_ match {
 case Success(v) => /* ... */
 case Failure(e) => /* ... */
})

fut.onSuccess { e => /* ... */ }

fut.onFailure { e => /* ... */ }
```

Futures

Computed value can be returned to the current thread with *Await*

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.concurrent.duration._

val f1 = future(1)
val f2 = future(throw new Exception("fail"))

// Access value via onComplete, onSuccess, or onFailure
val readyValue1 = Await.ready(f1, Duration.Inf)
val readyValue2 = Await.ready(f2, Duration.Inf)

val result1 = Await.ready(f1, Duration.Inf) // => 1
val result2 = Await.ready(f2, Duration.Inf) // => Exception
```

Futures

Computed value can be returned to the current thread with *Await*

- Kinda verbose
- More importantly, *blocking!*

Async to the rescue!

Async

- Library (macro) for a more *direct style* of programming when working with Future(-like) asynchronous constructs.
- More importantly: *Non blocking!*

Async

```
def slowCalcFuture: Future[Int] = / * ... */
def combined: Future[Int] = async {
 await(slowCalcFuture) + await(slowCalcFuture)
}
val x: Int = Await.result(combined, 10.seconds)
```

Async

Compare to code without *async*

```
def slowCalcFuture: Future[Int] = / * ... */
def combined: Future[Int] = future {
 Await.result(slowCalcFuture, 5.seconds) +
        Await.result(slowCalcFuture, 5.seconds)
}
val x: Int = Await.result(combined, 10.seconds)
```

Async

Compare to code without *async*

```
def slowCalcFuture: Future[Int] = / * ... */
def combined: Future[Int] = future {
 Await.result(slowCalcFuture, 5.seconds) +
        Await.result(slowCalcFuture, 5.seconds)
}
val x: Int = Await.result(combined, 10.seconds)
```

Async

```
def slowCalcFuture: Future[Int] = / * ... */
def combined: Future[Int] = async {
 await(slowCalcFuture) + await(slowCalcFuture)
}
val x: Int = Await.result(combined, 10.seconds)
```

Async

Non-blocking version using *map, flatMap*

```
def combined: Future[Int] =
 for {
   a <- slowCalcFuture
   b <- slowCalcFuture
 } yield a + b

val x: Int = Await.result(combined, 10.seconds)
```

- Monadic-style of Future composition

Async

Thanks to some macro magic *async* blocks are rewritten in a way that parks the computation awaiting the completion of the next awaited future.

This is very much like a *continuation*.

Actors to Futures?

Ask pattern:

```
import akka.pattern.ask

val result:Future[Int] = ask(someActorRef,message)
```

You can also *pipe* Future results to another actor:

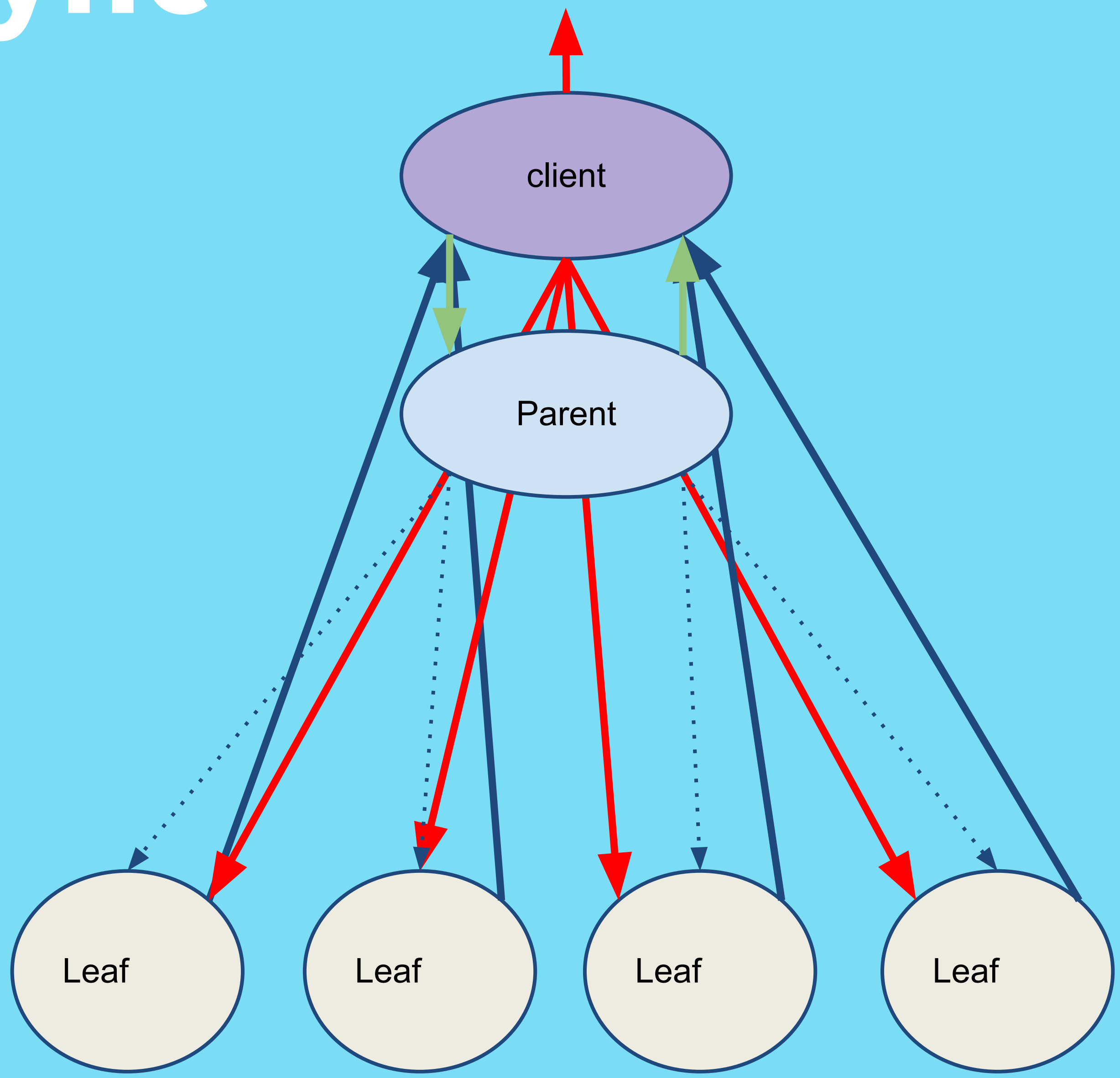
```
import akka.pattern.ask

val result:Future[Int] = ask(someActorRef,message).pipeTo(otherActorRef)
```

Async

Let's have a look at the scatter/gather search example looks like rewritten with *async*.

Async



Async: Observations

- Works fine for future-based computations (less restrictions)
- Does not really help in Actor-based applications
- Lifting actor results into Futures via ask is an anti-pattern
- Not conducive for scaling out

What to do?

Warning:
Highly Experimental

Inspiration

Rx - Reactive Extensions

- Made popular by Microsoft .NET/CLR
- Categorical "dual" of Iterator and Iterable

```
trait IObservable[T] {
 def subscribe(obs: IObserver[T]): Unit
}

trait IObserver[T] {
 def onNext(v: T): Unit
 def onError(t: Throwable): Unit
 def onComplete(): Unit
}
```

Inspiration

Rx - Reactive Extensions

- Completely abstracted implementation details
- Suitable for cross-machine/process boundary operation

Inspiration

Scala/Haskell "Machines" libraries

- Scala Machines: https://github.com/runarorama/scala-machines
- Haskell Machines: https://github.com/ekmett/machines

Inspiration

Scala/Haskell "Machines" libraries

- Composable network of stream processors
  - Compose stream flows with "monadic DSL"
- Transducer computation "effects" are *not the point* - only the consumption and production of values matters.
- Capable of producing in-memory data flows.

In CS, what do you do when confronting a difficult problem?

Add a level of indirection!

Basic Idea

- Use a monadic DSL to compose a network flow
- The output of the DSL is an *AST* representing the desired data flow
- Write one or more *interpreters* for the AST
    - Each *interpreter* maps the data flow *onto* a particular implementation
- An implementation is composed of *strategies*

Pieces: Values

Values emitted or consumed by *transformation functions*

```
sealed trait Value[+T]
case class Next[+T](value:T) extends Value[T]
case class Error(throwable:Throwable) extends Value[Nothing]
case object Empty extends Value[Nothing]
case object Complete extends Value[Nothing]
```

Pieces: Transformers

Transformation functions consume and produce values

```
trait TransFunc[+I,S,+O] { self =>
 def apply(value:Value[I],state:S):(Value[O],S)
}
```

Pieces: Transformers

```scala
case class Func[+I,S,+O](body: (I,S) => (O,S)) extends TransFunc[I,S,O] {
 def apply(value:Value[I],state:S):(Value[O],S) =
   value match {
     case Next(v) =>
       try {
         val (o,s) = body(v,state)
         (Next(o),s)
       } catch {
         case t:Throwable => (Error(t),state)
       }
     case e:Error => (e,state)
     case Empty => (Empty,state)
     case Complete => (Complete,state)
   }
}

case class TFunc[+I,S,+O](body: (Value[I],S) => (Value[O],S)) extends TransFunc[I,S,O] {
 def apply(value:Value[I],state:S):(Value[O],S) = body(value,state)
}

def func[I,S,O](body:(I,S) => (O,S)):Func[I,S,O]
def simpleFunc[I,O](body:I => O):Func[I,Unit,O]
def transFunc[I,S,O](body:(Value[I],S) => (Value[O],S)):TFunc[I,S,O]
```

Pieces: Processes

Processes represent the *abstract container* that executes the *transformer functions. Processes* can be *Linked.*

```
case class Link[+T](process:Process[_,T])

trait Process[+I,+O] {
 type STATE
 def output:Link[O] = Link(this)
}
```

Pieces: Processes

```
trait UnlinkedProcess[+I,+O] extends Process[I,O] { self =>
 type STATE
 def transformer:TransFunc[I,STATE,O]
 def apply[J >: I](input0:Link[J],input:Link[J]*):LinkedProcess[I,O]
 def flatMap[J >: I, B](f:Link[O] => LinkedProcess[J,B]):UnlinkedProcess[I,B]
 def map[B](f:Link[O] => Link[B]):UnlinkedProcess[I,B]
}

trait LinkedProcess[+I,+O] extends Process[I,O]{
 type STATE
 def transformer:TransFunc[I,STATE,O]
 def flatMap[J >: I, B](f:Link[O] => LinkedProcess[J,B]):LinkedProcess[I,B]
 def map[B](f:Link[O] => Link[B]):LinkedProcess[I,B]
}
```

Pieces: Processes

```
def producer[O,S](transformer:TransFunc[Unit,S,O]):LinkedProcess[Unit,O]
def sink[I,S](transformer:TransFunc[I,S,Unit]):UnlinkedProcess[I,Unit]
def trans[I,O,S](transformer:TransFunc[I,S,O]):UnlinkedProcess[I,O]
```

Assembly

Some sample transformer functions

```
val inc:Func[Unit,Int,Int] = func[Unit,Int,Int]((_,s) => (s+1,s+1))
val printNum:Func[Int,Unit,Unit] = simpleFunc[Int,Unit](println)
val printString:Func[String,Unit,Unit] = simpleFunc[String,Unit](println)
val intToString:Func[Int,Unit,String] = simpleFunc[Int,String](_.toString)
val stringToInt:Func[String,Unit,Int] = simpleFunc[String,Int](_.length)
```

Assembly

Some simple flows

```
for {
 n <- producer(inc)
 s <- trans(intToString)(n)
 s1 <- trans(stringToInt)(s)
 r <- sink(printNum)(s1)
} yield r
```

Assembly

Some simple flows

```
val proc1 =
 for {
   s <- trans(intToString)
 } yield s

val proc2 =
 for {
   n <- proc1
   s2 <- trans(stringToInt)(n)
 } yield s2
```

Assembly

Some simple flows

```
for {
 n1 <- producer(inc)
 n2 <- producer(inc)
 n3 <- producer(inc)
 s <- proc1(n1,n2,n3)
 s1 <- trans(stringToInt)(s)
 n <- proc2(s,s1)
} yield n
```

Pieces: Strategies

```
trait Strategy[+I,+O,S] {
 def func:TransFunc[I,S,O]
 var state:S
 def next[P >: O](value:Value[P]):Unit
 def eval(in:Value[I => Any]):Unit = {
   val (o,s) = func(in,state)
   state = s
   next(o)
 }
}
```

Pieces: Assemblies

```
trait Assembly[I,O] {
 var output:Value[O] = _
 def eval(in:Value[I]):Unit
}
```

Pieces: Compiler

```
def simpleCompiler[I,O](in:Process[I,O]):Assembly[I,O] = {
 def lookupStrategy[I1,O1](iin:Process[I1,O1]):(Value[O1] => Unit) => Strategy[I1,O1,_] =
   strategyMap(iin.transformer).asInstanceOf[(Value[O1] => Unit) => Strategy[I1,O1,_]]
 def doCompile[I1,O1](iin:Process[I1,O1],next:Value[O] => Unit):Strategy[I1,O1,_] = {
     in match {
       case x:Producer[_,_,_] =>
         lookupStrategy(x)(next)
       case x@LinkedSink(_,ls) =>
         val strat = lookupStrategy(x)(next)
         ls.foreach{ l => doCompile(l.process,simpleNext(strat)) }
         strat
       case x@LinkedTransformer(_,ls) =>
         val strat = lookupStrategy(x)(next)
         ls.foreach{ l => doCompile(l.process,simpleNext(strat)) }
         strat
       case x@UnlinkedTransformer(_) =>
         lookupStrategy(x)(next)
       case x@UnlinkedSink(_) =>
         lookupStrategy(x)(next)
       case x@FlatMappedUnlinkedProcess(_) => doCompile(x.downstream,next)
   }
 }
 new Assembly[I,O] {
   val code:Strategy[I,O,_] = {
     doCompile(in,(v:Value[O]) => output = v)
   }

   def eval(in:Value[I]):Unit = code.eval(in.asInstanceOf[Value[I => Any]])
 }
}
```

Conclusion

- We need better ways of working with the actor model
    - Disentangle "business logic" from plumbing
    - Improve reasoning
    - Composability
    - Robust, independently tested *strategies*

References

- Akka: http://akka.io/
- Scala Async: https://github.com/scala/async
- Scala Machines: https://github.com/runarorama/scala-machines
- Haskell Machines: https://github.com/ekmett/machines
- Josh Suereth's *Introduction to Actor Systems*: http://www.infoq.com/presentations/Akka-Actors
- Philipp Haller's *Scala Async: A New Way to Simplify Asynchronous Code*: http://vimeo.com/66864872
- Paul Chiusano's *Advanced Stream Processing in Scala*: http://www.youtube.com/watch?v=8fC2V9HX_m8
- Cloud Haskell: http://www.haskell.org/haskellwiki/Cloud_Haskell
- Akka Typed Channels: http://letitcrash.com/post/45188487245/the-second-step-akka-typed-channels