# Formal methods ... in Action !

# typestate-oriented actor programming in Scala akka


**Silvia Crafa**

Università di Padova, Italy


*Paris - May 2017*

In distributed systems the

***coordination*** **of concurrent entities**

is a key issue


# **Protocol-Oriented** Programming

***thinking*** (hence programming)
***in terms of***
communication ***protocols***

it requires:

- **high-level support to declare/express** the coordination protocol
- **support to check it!** …hopefully statically…(gradual) typing?

# **TypeState-Oriented** Programming

S. Crafa 2017

# TypeState-oriented OOP

Strom, Yemini '86
DeLine, Fahndrich '04
Aldrich et al. '09

```
class File {
  public void open(){ … }
  public int read(){ … }
  public void write(int i){ … }
  public close(){ … }
}
```

**interface**
**vs**
**protocol**

```
File f = … //initialize

f.open(); f.write(5); f.close();

f.write(5); f.open(); f.read();

f.open(); f.write(5);
```

✓

✗ ***wrong*** protocol

✗ ***incomplete*** protocol

# TypeState-oriented OOP

a `File` has two possible **states**: CLOSED, OPEN

```
public void open(){ … }  //move the state to OPEN
```

```
public int read(){ … }
public void write(int i){ … }
public close(){ … }     // move the state to CLOSED
```

a type system **statically** guarantees that **methods** are **invoked** when the object is **in the correct state**

Mechanisms
- state annotations in types
- tracking of state transitions
- **aliasing control**

# **TypeState-Oriented** Programming

## for **Concurrent** objects

concurrent objects are typically aliased
state transitions aren't always statically trackable

OOPSLA'15 - TOPLAS'17

# The Chemical Approach to Typestate-Oriented Programming

Silvia Crafa
Università di Padova, Italy

Luca Padovani
Università di Torino, Italy

a sound **behavioral type system**
for the Objective ***Join Calculus***

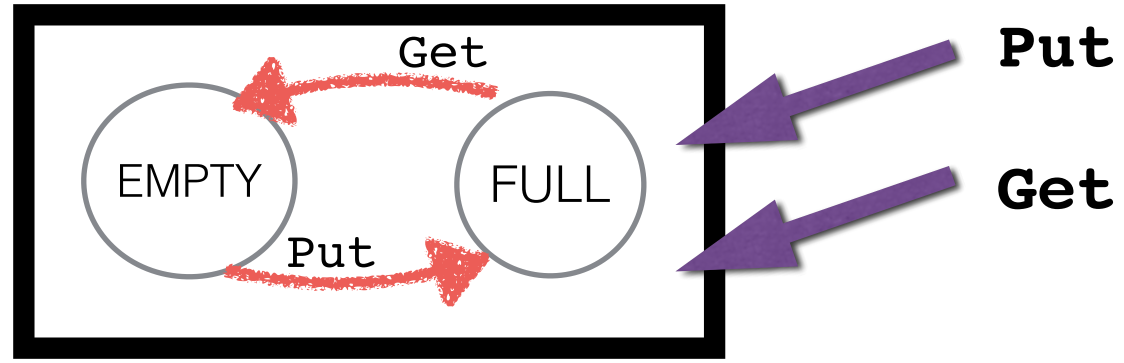
<u>**both**</u> object **state** and object **messages** as ***molecules***

## Chemical Abstarct Machine:

- object **methods** are **join reactions** that consume molecules and produce new ones:
- **programs** are object definitions and a **soup** of molecules
- **runtime semantics** resolves races and executes methods at the right time

```
def a = m ▸ b.n
    or  n ▸ a.m
in def b = m | n ▸ b.m
in a.m | a.n | b.m
```

# 1-place buffer




```
def buffer =
      EMPTY    | Put(v,r) ▶ buffer ! FULL(v) | r ! reply(buffer)
or    FULL(v)  | Get(r)   ▶ buffer ! EMPTY   | r ! reply(v,buffer)
in
   buffer ! EMPTY   | // CONSTRUCTOR
```

```
   let buffer      = buffer ! Put(10)
in let (v,buffer)  = buffer ! Get
in let buffer      = buffer ! Put(20+v)
in let (v,buffer)  = buffer ! Get
```

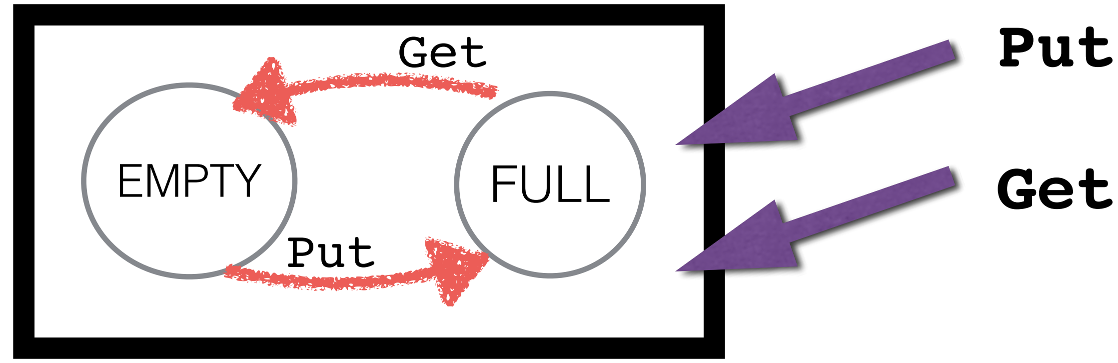
continuation-passing style

$$t_{buff} = (\mathrm{EMPTY} \otimes t_{EMPTY}) \oplus (\mathrm{FULL(int)} \otimes t_{FULL})$$

$$t_{EMPTY} = \mathrm{Put(int, reply(}t_{FULL}\mathrm{))}$$
$$t_{FULL} = \mathrm{Get(reply(int,}t_{EMPTY}\mathrm{))}$$

**behavioral type**
with *linear connectives*

# 1-place buffer




```
def buffer =
      EMPTY   | Put(v,r) ▶ buffer ! FULL(v)  | r ! reply(buffer)
or    FULL(v) | Get(r)   ▶ buffer ! EMPTY    | r ! reply(v,buffer)
in
   buffer ! EMPTY    | // CONSTRUCTOR
```

```
buffer!Put(10) & buffer!Put(20) & buffer!Put(30) | // PRODUCER
buffer!Get & buffer!Get &  buffer!Get              // CONSUMER
```

runtime semantics will **execute** methods **at the right time**

$$t_{buff} = (\mathtt{EMPTY} \oplus \mathtt{FULL(int)}) \otimes t_{prod} \otimes t_{cons}$$

$$t_{prod} = \mathtt{Put(int,reply(}t_{prod}\mathtt{))}$$
$$t_{cons} = \mathtt{Get(reply(int,}t_{cons}\mathtt{))}$$

# **1** object - **many** protocols / types

$t_{buff}$=(EMPTY⊕FULL)⊗$t_{prod}$⊗ $t_{cons}$

buffer : **t**

buffer : **t1**

$t_{prod}$ = Put(int,reply($t_{prod}$))

the **user** is not aware of internal states:
it receives a reference whose **type** just
express **how to use it**

# 1 object - **many** protocols / types

`buffer : t`

`buffer : t1`

`buffer : t2`

with `t ≤ t1,t2`

**subtyping:**

a super-type/protocol express a usage that is safe w.r.t. `t`

## different from Session Types :

- there is **no strong duality** between the communicating parties
  - no fixed number of parties,
  - no precise protocol progression,
- in TSOP there is ***asymmetry*** between
  - the **object** is **stateful**
  - the **user**(s) receives a typed handle which prescribes **how to use the object**
- linear connectives allow *AND-states* and *OR-states*

## different from other TSOP approaches (e.g., Plaid, Mungo):

- **no state is exposed**, it is a private “molecule”
- hence there is **no need to control aliasing**
- but the ***runtime synchronisation is essential*** to keep messages and consume them at the right time

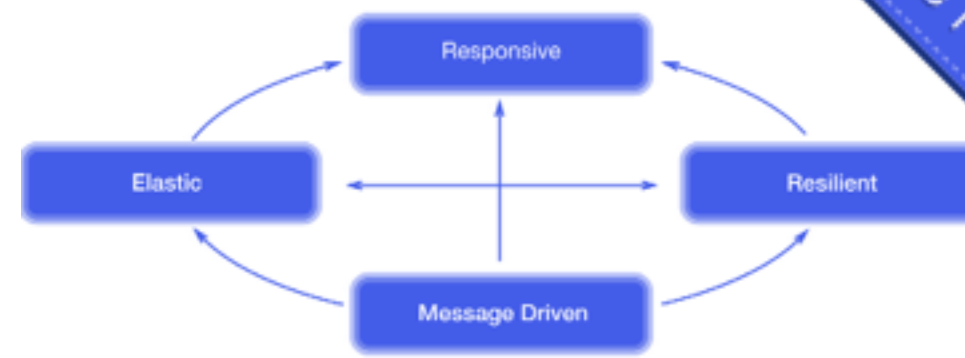
what about **mainstream distributed** systems **programming?**

**TypeState-Oriented** Programming

for **Concurrent** objects

Objective
**Join Calculus**

We Are Reactive




# **Distributed** Programming

# by means of **Actor systems**

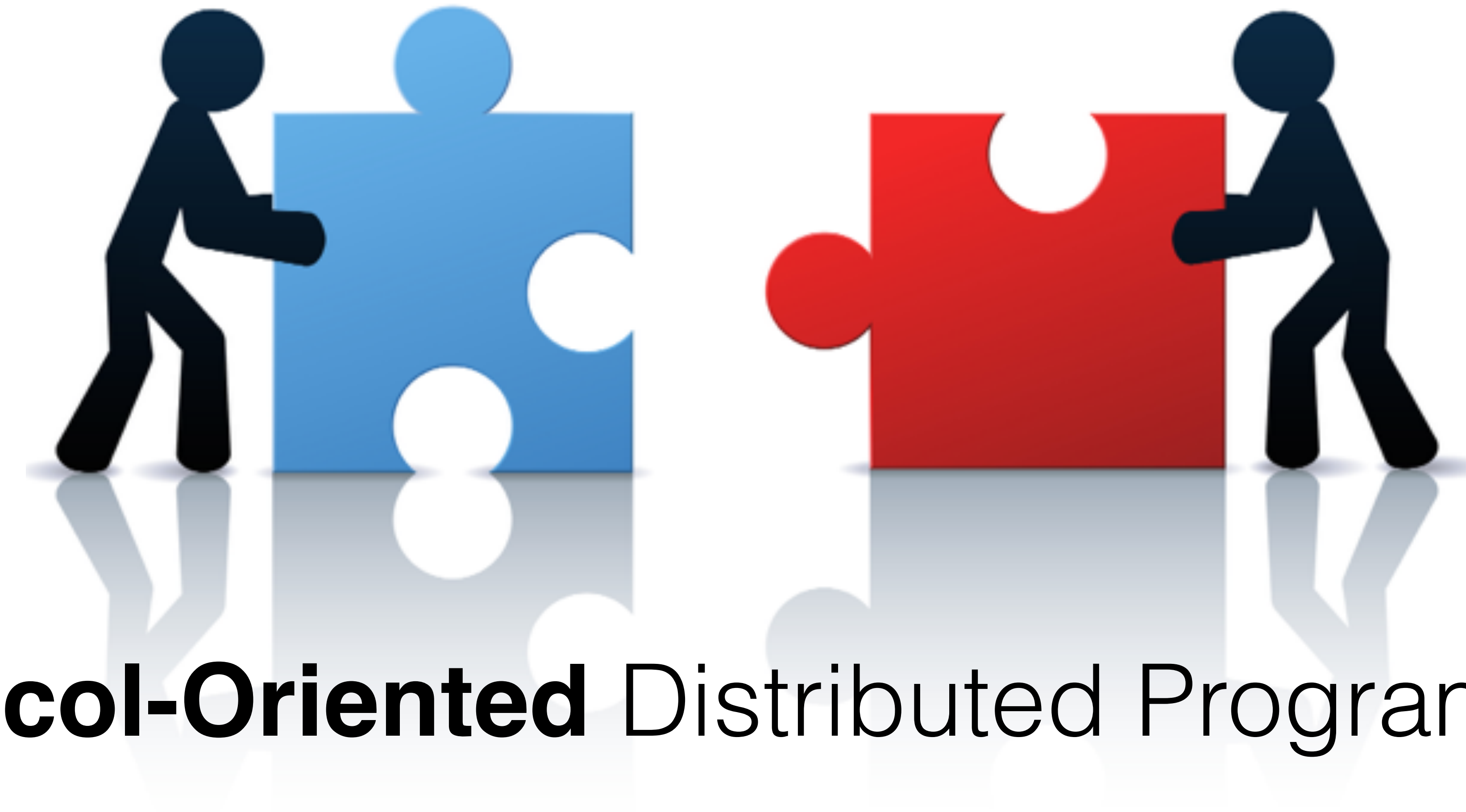
more Object-Oriented than objects

types inside

Scala

akka

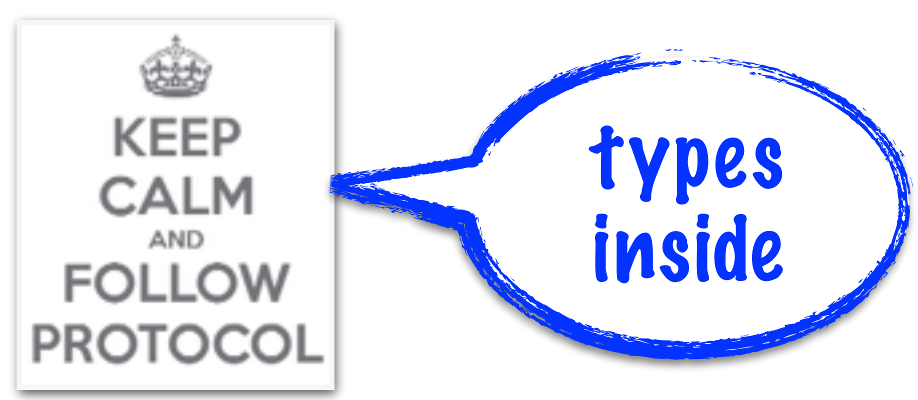
We Are Reactive



# **Protocol-Oriented** Distributed Programming

# as

# **TypeState**-based **Actor** programming



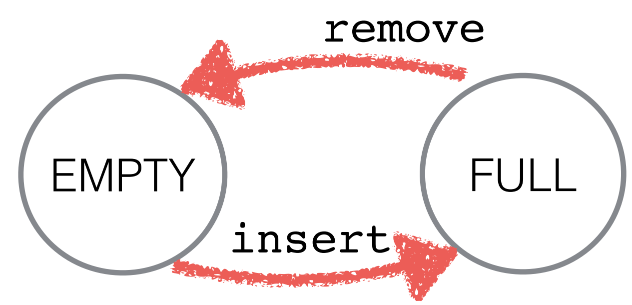

Scala

akka

S. Crafa 2017

# 1-place buffer as TSOAP

```
case class insert[T](value:T)
case class remove()
```




```
class Buffer[T] extends Actor {
   def EMPTY:Receive = {
        case insert(x:T) => context.become(FULL(x))
  }

   def FULL(x:T):Receive = {
        case remove() => context.become(EMPTY)
   }
   def receive = EMPTY
}
```

**clean logic:**

**each behaviour** /state defines **only the intended messages!**

*no defensive programming*

```
val s = ActorSystem()
val buffer = s.actorOf(Props(new Buffer[Int]))
val user = s.actorOf(Props(new Actor{
              buffer ! insert(4)
              buffer ! remove()    }
```

# 1-place buffer as TSOAP

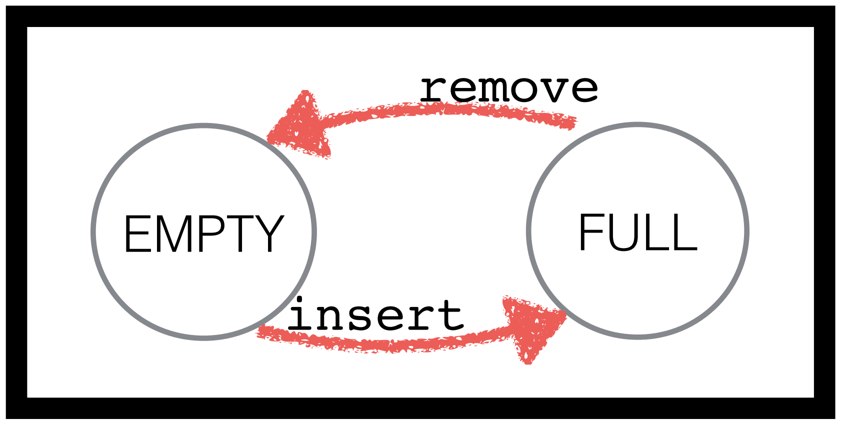
```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```

**no support from the compiler**

**bad msg!!**

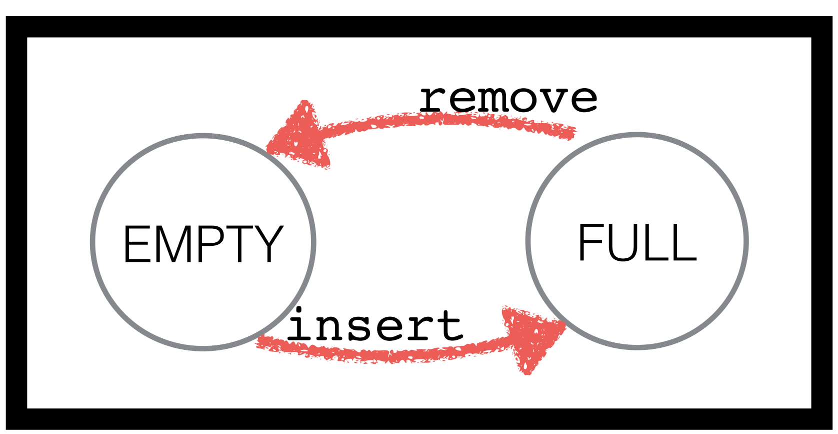
```
buffer ! "hello"
buffer ! insert(1)
buffer ! remove()
buffer ! remove()
```

**bad state!!**



# 1-place buffer as TSOAP

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```




**Producer**

```
insert(1)
insert(2)
insert(3)
…
```

```
remove()
remove()
remove()
…
```

**Consumer**

**`insert` msg can <u>arrive</u>** in the buffer's mailbox while the buffer is **in state FULL** !

***we still want to ensure that `insert` / `remove` are <u>served</u> in the correct state!***

# 1-place buffer as TSOAP

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive
      case remove() =
  }
  def receive = EMPTY
}
```

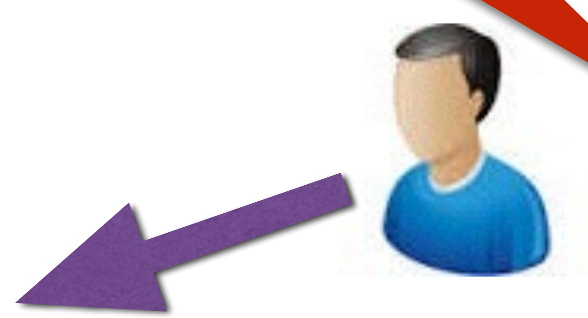
**add a layer of typing!**

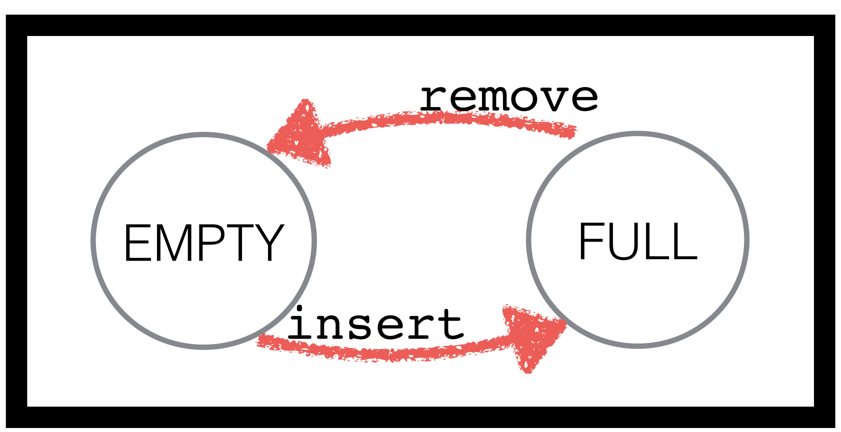
**no support from the compiler**

**bad msg!!**

```
buffer ! "hello"
buffer ! insert(1)
buffer ! remove()
buffer ! remove()
```

**bad state!!**





"simpler" than Akka Typed

# add a layer of typing

**buffer ! "hello"**

has type **ActorRef**, which declares **def !(msg:Any)** which is *always well-typed*

→

**buffer ! "hello"** ✗

has type **TypedRef[T]**, which

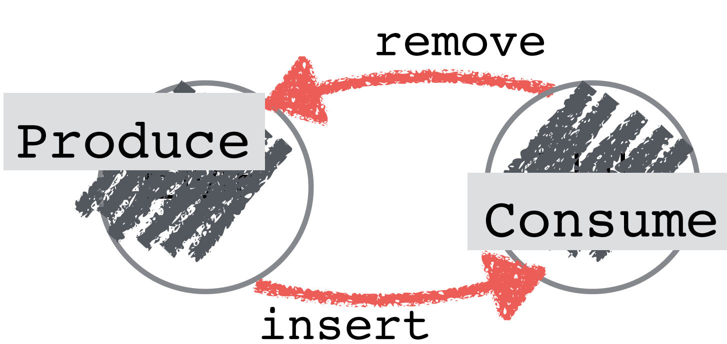
- is a wrapper for an actor **at state T**
- declares **def !(msg:T)**, hence it **statically type-checks** that only msg belonging to the interface **T** are sent

# add a bit of **encapsulation**

**Nominal typing:** a Type for each set of allowed messages, i.e. ***an Interface for each State***

```
trait BufferInterf
trait Produce extends BufferInterf
trait Consume extends BufferInterf

case class insert[T](value:T) extends Produce
case class remove()  extends Consume
```



# 1-place buffer …*with typed reference*

```
trait BufferInterf
trait Produce extends BufferInterf
trait Consume extends BufferInterf

case class insert[T](value:T) extends Produce
case class remove()  extends Consume

class Buffer[T] extends Actor {  …as before… }
```

```
  val s = ActorSystem()
  val untypedBuffer = s.actorOf(Props(new Buffer[Int]),"buff")
  val buffer = new TypedRef[BufferInterf](untypedBuffer)


  val user = s.actorOf(Props(new Actor{
                 buffer ! insert(4)    ok
                 buffer ! remove()     ok
                 buffer ! “hello”    does not compile

}))
```

1. **no other messages** but `insert` and `remove`

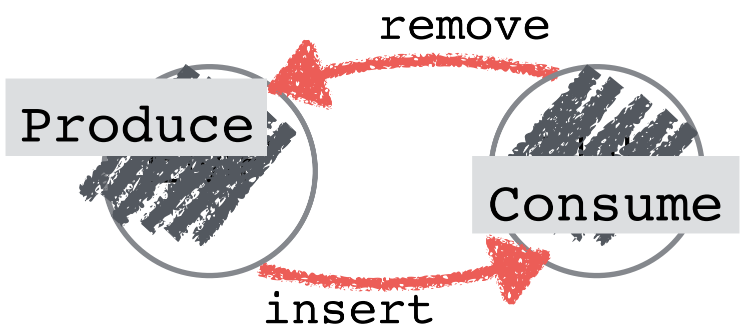
# 1-place buffer …*with typed reference*

```
val s = ActorSystem()
val untypedBuffer = s.actorOf(Props(new Buffer[Int]),"buff")
val buffer = new TypedRef[Produce](untypedBuffer)

val user = s.actorOf(Props(new Actor{
          buffer ! insert(4)    ok
          buffer ! remove()    does not compile

}))
```

the `buffer` reference
**dynamically changes its type**
between `TypedRef[Produce]`
and `TypedRef[Consume]`




thus at any change we take **a new reference, statically typed with the new type**

**Continuation-passing Style**

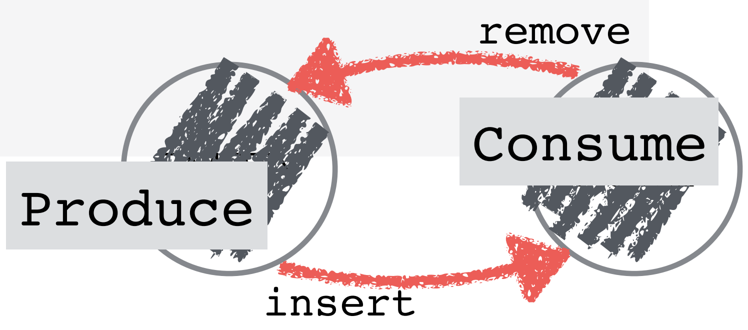
# 1-place buffer ***with <u>explicit</u> continuations***

```
case class insert[T](value:T, replyTo:ActorRef) extends Produce
case class remove(replyTo:ActorRef) extends Consume

case class insertReply(o:TypedRef[Consume])
case class removeReply[T](o:TypedRef[Produce])
```

*reply messages*
***carry the continuation reference***
*with suitable type*

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T,r) => context.become(FULL(x))
                            r ! insertReply(new TypedRef[Consume](self))
  }
  def FULL(x:T):Receive = {
      case remove(r) => context.become(EMPTY)
                            r ! removeReply(new TypedRef[Produce](self))
  }
  def receive = EMPTY
}
```



# 1-place buffer ***with explicit continuations***

```
case class insert[T](value:T, replyTo:ActorRef) extends Produce
case class remove(replyTo:ActorRef) extends Consume
case class insertReply(o:TypedRef[Consume])
case class removeReply[T](o:TypedRef[Produce])
```

*reply messages carry the continuation reference with suitable type*

```
val s = ActorSystem()
val untypedBuffer = s.actorOf(Props(new Buffer[Int]),"buff")
val buffer = new TypedRef[Produce](untypedBuffer)

val user = s.actorOf(Props(new Actor{
      buffer ! insert(1,self)
      def run(v:Int) :Receive = {
          case insertReply(o) => o ! remove(self)   // insert rises an error
          case removeReply(o) => o ! insert(v+1,self)
                                 context.become(run(v+1))
      }
      def receive = run(0)
}))
```

**à la Akka Typed**

# 1-place buffer ***with <u>implicit</u> continuations***

```
case class insert(value:Int) extends Produce
case class remove() extends Consume
```

**(Typed) Monad**
**using Futures/ Promises**

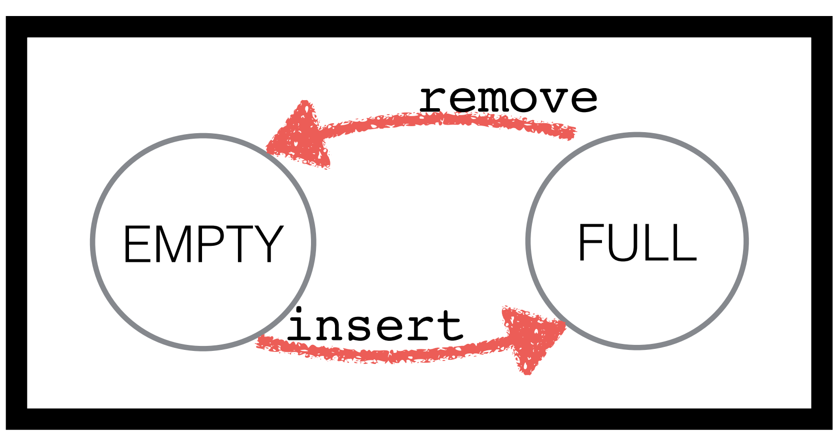
```
val s = ActorSystem()
val untypedBuffer = s.actorOf(Props(new Buffer),"buff")
val buffer = new ProtRef[Produce](untypedBuffer, Buffer.protocol)

val user = s.actorOf(Props(new Actor{
      for {
          o <- buffer ! insert(1)
          o <-  o ! remove()
          o <-  o ! insert(2)
          o <-  o ! remove()
          o <-  o ! insert(3)
             // o ! insert(4) compiler error
          o <-  o ! remove()
      } yield print("END")

     def receive= PartialFunction.empty
}))
```

# 1-place buffer as TSOAP

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```




**Producer**

insert(1)
insert(2)
insert(3)
…

remove()
remove()
remove()
…

**Consumer**

**`insert` msg can arrive** in the buffer's mailbox while the buffer is **in state FULL** !

***we still want to ensure that `insert` / `remove` are served in the correct state!***

# Chemical semantics *by mixing-in* ***typed stashing***

```
class Buffer[T] extends Actor with Chemical {
  def EMPTY:Receive = chemReact {
      case insert(x:T,r) => chemBecome(FULL(x))
                            r ! insertReply(new TypedRef[Consume](self))
  }
  def FULL(x:T):Receive = chemReact {
      case remove(r) => chemBecome(EMPTY)
                            r ! removeReply(x, new TypedRef[Produce](self))
  }
  def receive = EMPTY
}
```

**still a clean logic:**
different states with
different interfaces

```
trait ProtocolMsg

trait Chemical extends Actor with Stash {
  private def check() = { unstashAll() }
  private def keep :Receive = { case (msg:ProtocolMsg,p) => stash() }
  def chemBecome(newState:Receive)={ check(); context.become(newState) }
  def chemReact(behave:Receive):Receive = behave orElse keep
}
```

```
class Producer(buffer:ProtRef[Produce]) extends Actor {
   for {
      o <- buffer ! insert(0)
      o <-   o ! insert(10)
      o <-   o ! insert(20)
      o <-   o ! insert(30)
      o <-   o ! insert(40)
   } yield println("End Producer")
  def receive = PartialFunction.empty
}
```

```
class Consumer(buffer:ProtRef[Consume]) extends Actor{
   for {
      o <- buffer ! remove()
      o <-   o ! remove()
      o <-   o ! remove()
      o <-   o ! remove()
   } yield println("End Consumer")

  def receive = PartialFunction.empty
}
```

```
val s = ActorSystem()
val bufferUntyped = s.actorOf(Props(new Buffer[Int]),”buffer”)
val buffer = new ProtocolRef[BufferInterf](bufferUntyped,Buffer.protocol)
val producer1 = s.actorOf(Props(new Producer(buffer,1)))
val consumer1 = s.actorOf(Props(new Consumer(buffer," pippo")))
val consumer2 = s.actorOf(Props(new Consumer(buffer," pluto")))
```

**contravariance**
of ProtRef[-T]

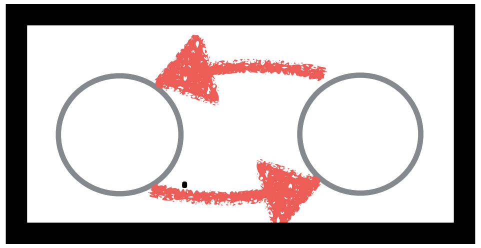
```
class Producer(buffer:ProtRef[Produce]) extends Actor {
   for {
      o <- buffer ! i
      o <-   o ! inse
      o <-   o ! inse
      o <-   o ! inse
      o <-   o ! inse
   } yield println("En
  def receive = Partia
}
```

```
val s = ActorSystem(
val bufferUntyped =
val buffer = new Pro
val producer1 = s.ac
val consumer1 = s.ac
val consumer2 = s.ac
```

```
produce 1
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 1
removed 1
produce 11
 pippo consumed 1
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 11
produce 21
removed 11
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 21
removed 21
 pippo consumed 11
 pluto consumed 21
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
produce 31
inserted 31
produce 41
removed 31
 pippo consumed 31
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 41
removed 41
End Producer n.1
 pippo consumed 41
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
```

# what is missing w.r.t. the behavioural typing of Join?

after `insert` there **must** be a `remove`



`insert(1) ; remove ; insert(3) ; remove`

use the buffer **just once**

**Producer**

`insert()`
~~`insert()`~~
~~`insert()`~~
...

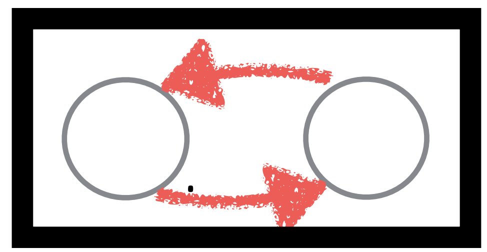
**Consumer**

`remove()`
~~`remove()`~~
~~`remove()`~~
...

**protocols expressing linear capabilities**

# what is missing w.r.t. the behavioural typing of Join?

after `insert` there **must** be a `remove`



`insert(1) ; remove ; insert(3) ; remove`

compliance with protocol **obligations** requires **linear types**

use the buffer **just once**

**Producer**

```
insert()
~~insert()~~
~~insert()~~
...
```

**Consumer**

```
remove()
~~remove()~~
~~remove()~~
...
```

Scala

# That's it !

Objective
**Join Calculus**

**Protocol-Oriented** Distributed Programming

as

**TypeState**-based **Actor** programming

types inside

Scala

akka

S. Crafa 2017

Objective
**Join Calculus**

Algorithmic Typestate Checking
for Concurrent Objects

Silvia Crafa[1] and Luca Padovani[2]

**Formal foundation**

integrating
the type inference in
**Scala Akka -compiler**

types
inside

**Scala**

akka

# merci !