SCALA-GOPHER

CSP WITH IDIOMATIC SCALA

https://github.com/rssh/scala-gopher

goo.gl/dbT3P7

Ruslan Shevchenko

KharkivPY Group: Scala & Fun, 2016

2. Outline

- Theory & History
  - CSP = Communication Sequence Processes
  - π-calculus (CCS = Calculus of Communicating System)
  - Languages: (Occam,Limbo, Go[Clojure, Scala, …])
- Main constructions / idioms, how they looks
  - Channels, Selectors, Transputers
- Implementation Techniques

3. Theory & History

- If you familiar with basic concepts, skip to slide 8

- Just show me scala API: skip to slide 14

4. Theory & History

- CPS = Communication Sequence Processes.
- CSS = Calculus of Communicating System.
  - 1978 First CSP Paper by Tony Hoar.
  - 1980. CSS by Robert Milner. (Algebraic Processes)
  - 1985 CSP formalism (influenced by CSS) in CSP Book
    - http://www.usingcsp.com/
  - 1992 π-calculus [CSS + Channels] (Robert Milner, Joachim Parrow, David Walker)
  - …. (large family of Process Algebras, including Actor Model, ACP,)

5. CSP Notation(basic)

- (A, B, C …) — processes,
- (x, y, z …) — events
  - atomic: x , STOP, BEEP, or c.v (channel/value)
    - c!v - send v to channel c (after this c.v happens)
    - c?v - receive v from channel c (wait until c.v)
- trace(…) — sequence of events.

6. CSP Notation(basic)

Operations on processes:

$a \rightarrow P$ // after event

$P|||Q$ // interleave concurrently

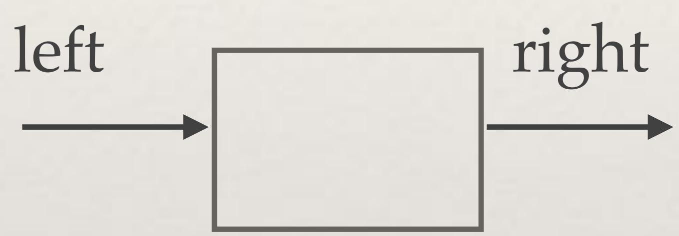
$P; Q$ // seq

$(a \rightarrow P) \Box (b \rightarrow Q)$ // choice, if a then P if b then Q

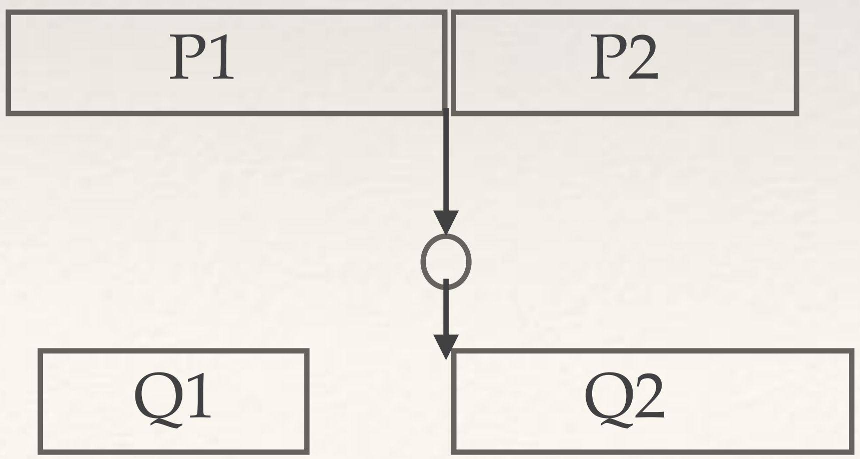
$\mu X.F(X) \equiv F(\mu X.F(X))$ // fix point (loops, recursion

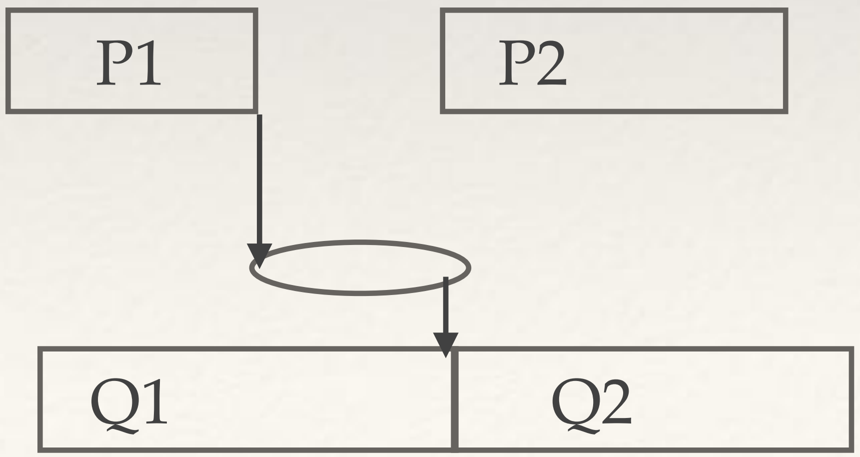
7. CSP Notation(examples)

COPY(left,right)= $\mu F.left?x \rightarrow right!x \rightarrow F$



$$(P_1 \rightarrow c!x \rightarrow P_2)|||(Q_1 \rightarrow c?x \rightarrow Q_2(x))$$





8. Occam language

William Occam, 1287-1347
'Occam razor' principle

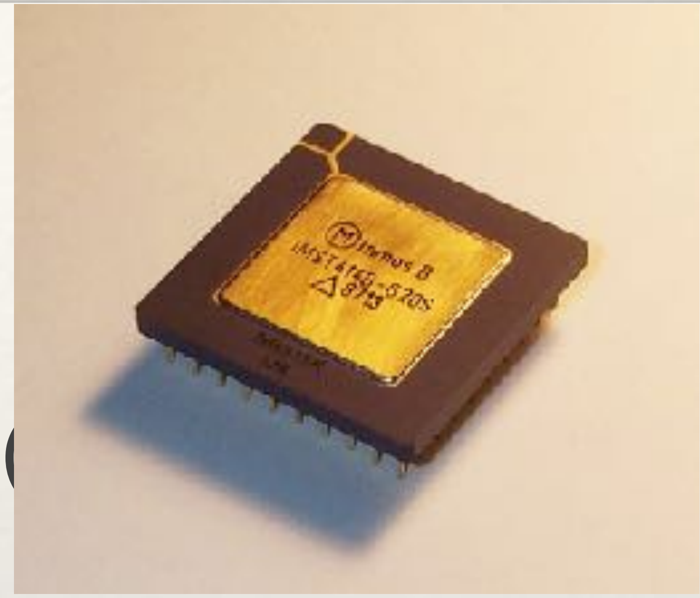
Occam language. (1983)

```
INT x, y:
  SEQ
    x := 4
    y := (x + 1)
    CHAN INT c:
    PAR
      some.procedure (x, y, c!)
      another.procedure (c?)
    y := 5
```

minimal language. (processor constructors)

braces via indentation. (before python)

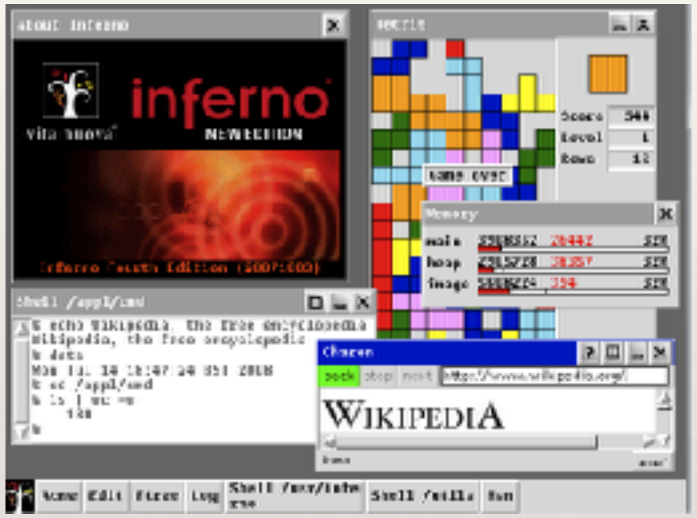
9. Occam language



- created by INMOS
- targeted Transputers CHIP architecture (
- latest dialect: occam-π [1996]
  - extensions from π-calculus
  - (more dynamic, channels can be send via channels)
- http://concurrency.cc — occam for Aurdino.

10. Inferno, Limbo, Go

- Unix, Plan9, Inferno: [At&t; Bell labs; vita nuova]



Sean Forward
David Leo Presotto
Rob Pike
Dennis M. Ritchie
Ken Thompson

- http://www.vitanuova.com/inferno/
- Limbo … C-like language + channels + buffered channels.
- Go … channels as in Limbo (Go roots is from hell is not a metaphor)

11. CPS in Limbo/Go/Scala: main constructions

- processes: operator for spawning new lightweight process.
- channels: can be unbuffered (synchronised) or buffered
  - unbuffered — writer wait until reader start to work
  - buffered — if channel buffer is not full, writer not blocked
- selector: wait for few events (channel), eval first.

12. Go: simple code example

Go

```
func fibonacci(c chan int, quit chan bool) {
  go {
      x, y := 0, 1
      for (){
        select {
            case c <- x :
                    x, y = y, x+y
            case q<-quit:
                    break;
        }
      }
      close(c)
  }
}
```

```
c = make(chan int);
quit= make(chan bool);
fibonacci(c,quit)
for i := range c {
            fmt.Println(i)
            if (i > 2000) {
                quit <- true
            }
}
```

13. Go: simple code example

Go

```
func fibonacci(c chan int, quit chan bool) {
  go {
      x, y := 0, 1
      for (){
        select {
           case c <- x :
                   x, y = y, x+y
           case q<- quit:
                   break;
        }
      }
      close(c)
  }
}
```

```
c = make(chan int);
quit= make(chan bool);
fibonacci(c,quit)
for i := range c {
          fmt.Println(i)
          if (i > 2000) {
             quit <- true
          }
}
```

scala-gopher

- Akka extension + Macros on top of SIP22-async
- Integrate CSP Algebra and scala concurrency primitives
- Provides:
  - asynchronous API inside general control-flow
  - pseudo-synchronous API inside go{ .. } or async{ ..} blocks
- Techreport: goo.gl/dbT3P7

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
      for(i <- 1 to Int.MaxValue) in.write(i)
    }
    go {
      select.fold(in){ (ch,s) =>
        s match {
          case p:ch.read => out.write(p)
                            ch.filter(_ % p != 0)
        }
      }
    }
    go {
      for(i <- 1 to n) yield out.read
    }
 }
```

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
       for(i <- 1 to Int.MaxValue)
                                 in.write(i)
    }
       s match {
         case p:ch.read => out.write(p)
                           ch.filter(_ % p != 0)
       }
     }
    }
    go {
      for(i <- 1 to n) yield out.read
    }
 }
```

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
      for(i <- 1 to Int.MaxValue) in.write(i)
    go {
      select.fold(in){ (ch,s) =>
       s match {
        case p:ch.read => out.write(p)
                          ch.filter(_ % p != 0)
       }
      }
    }
      for(i <- 1 to n) yield out.read
    }
 }
```

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
      for(i <- 1 to Int.MaxValue) in.write(i)
    }
    go {
      select.fold(in){ (ch,s) =>
        s match {
          case p:ch.read => out.write(p)
                            ch.filter(_ % p != 0)
        }
      }
    go {
           for(i <- 1 to n) yield out.read
    }

 }
```

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
      for(i <- 1 to Int.MaxValue) in.write(i)
    }
    go {
      select.fold(in){ (ch,s) =>
        s match {
          case p:ch.read => out.write(p)
                            ch.filter(_ % p != 0)
        }
      }
    }
    go {
      for(i <- 1 to n) yield out.read
    }
 }
```

Goroutines

- go[X](body: X):Future[X]
  - Wrapper around async +
  - translation of high-order functions into async form
  - handling of defer statement

Goroutines

- translation of hight-order functions into async form
  - **f**(**g**): **f**: (A=>B)=>C in **g**: A=>B,
  - **g** is invocation-only in **f** iff
    - **g** called in **f** or in some **h** inside **f** : **g** invocation-only in **h**
    - **g** is
      - not stored in memory behind **f**
      - not returned from **f** as return value
  - Collection API high-order methods are invocation-only

Translation of invocation-only functions

- f: ((A=>B)=>C),   g: (A=>B),   g invocation-only in f
- f’: ((A=>Future[B])=>Future[C])   g’: (A=>Future[B])
    - await(g’) == g   =>   await(f’) == f
        - f’   => await[translate(f)]
        - g(x) => await(g’(x))
        - h(g) => await(h’(g’)) iff g is invocation-only in h
    - That’s all
        - (implemented for specific shapes and parts of scala collection API)

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
      for(i <- 1 to n*n) in.write(i)
    }
    go {
      select.fold(in){ (ch,s) =>
        s match {
          case p:ch.read => out.write(p)
                            ch.filter(_ % p != 0)
        }
      }
    go {
          for(i <- 1 to n) yield out.read
    }

 }
```

```
go {
      (1 to n).map(i => out.read)
}
```

```
async{
     await(t[(1 to n).map(i => out.read)])
}
```

```
async{
  await((1 to n).mapAsync(t[i => async(out.read)]))
}
```

```
async{
 await((1 to n).mapAsync(i => async(await(out.aread)))
}
```

```
 mapAsync(i => out.aread)
```

Channels

- Channel[A] <: Input[A] + Output[A]
  - Unbuffered
  - Buffered

    CSP

  - Dynamically growing buffers [a-la actor mailbox]
  - One-time channels [Underlaying promise/Future]
  - Custom

Input[A] - internal API

```
trait Input[A]
{
   type read = A

   def cbread(f: ContRead[A,B]=>Option[
                              ContRead.In[A] => Future[Continuated[B]])
```

ContRead[A,B].F

```
case class ContRead[A,B](
    function: F,
    channel:  Channel[A],
    flowTermination: FlowTermination[B]
)
// in ConRead companion object
sealed trait In[+A]
case class Value(a:A)  extends In[A]
case class Failure(ex: Throwable] extends In[Nothing]
case object Skip extends In[Nothing]
case  object ChannelClosed extends In[Nothing]
```

Continuated[B]

- ContRead
- ContWrite
- Skip
- Done
- Never

Input[A] - external API

```
trait Input[A]
{
   ……

   def  aread: Future[A]  =  <implementation…>

   def  read: A = macro <implementation … >
                                                await(aread)
   ….

   def  map[B](f: A=>B):  Input[B] = ….

   //  or,   zip,  filter, …  etc
```

+ usual   operations on streams in functional language

Output[A] - API

```
trait Output[A]
{
   type write = A

   def cbwrite(f: ContWrite[A,B]=>Option[
                              (A,Future[Continuated[B]])],
                   ft: FlowTermination[B])
   …..

   def  awrite(a:A): Future[A] = ….

   def  write(a:A): A  = ….  <<  await(awrite)

   ….
```

ContWrite[A,B].F

```
case class ContWrite[A,B](
    function: F,
    channel:  Channel[A],
    flowTermination: FlowTermination[B]
)
```

Selector

$$(a \rightarrow P)\Box(b \rightarrow Q)$$

Go language:

```
go {
  for{
    select{
       case c1 -> x :   …  / /  P
       case c2 <- y :   …   / / Q
    }
  }
}
```

$$*[(c_1?x \rightarrow P)\Box(c_2!y \rightarrow Q)]$$

Scala:

```
go {
    select.forever {
        case x : c1.read =>   …  / /  P
        case y : c2.write =>  …   / / Q
    }
}
```

```
select.aforever {
        case x : c1.read =>   …  / /  P
        case y : c2.write =>  …   / / Q
}
```

Provide set of flow combinators:
forever, once, fold

select: fold API

```
def fibonacci(c: Output[Long], quit: Input[Boolean]): Future[(Long,Long)] =
    select.afold((0L,1L)) { case ((x,y),s) =>
     s match {
       case x: c.write => (y, x+y)
       case q: quit.read =>
                 select.exit((x,y))
     }
  }
```

fold/afold:

- special syntax for tuple support
- 's': selector pseudoobject
- s match must be the first statement
- select.exit((..)) to return value from flow

```
def nPrimes(n:Int):Future[List[Int]]= {
    val in = makeChannel[Int]()
    val out = makeChannel[Int]()
    go {
      for(i <- 1 to n*n) in.write(i)
    }
    go {
      select.fold(in){ (ch,s) =>
        s match {
          case p:ch.read => out.write(p)
                            ch.filter(_ % p != 0)
        }
      }
    go {
          for(i <- 1 to n) yield out.read
    }

 }
```

```
go {
      (1 to n).map(i => out.read)
}
```

```
async{
     await(t[(1 to n).map(i => out.read)])
}
```

```
async{
  await((1 to n).mapAsync(t[i => async(out.read)]))
}
```

```
async{
 await((1 to n).mapAsync(i => async(await(out.aread)))
}
```

```
mapAsync(i => out.aread)
```

Channels

- Channel[A] <: Input[A] + Output[A]
  - Unbuffered
  - Buffered

    CSP
  - Dynamically growing buffers [a-la actor mailbox]
  - One-time channels [Underlaying promise/Future]
  - Custom

Input[A] - internal API

```
trait Input[A]
{
   type read = A

   def cbread(f: ContRead[A,B]=>Option[
                        ContRead.In[A] => Future[Continuated[B]])
```

ContRead[A,B].F

```
case class ContRead[A,B](
    function: F,
    channel:  Channel[A],
    flowTermination: FlowTermination[B]
)
// in ConRead companion object
sealed trait In[+A]
case class Value(a:A)  extends In[A]
case class Failure(ex: Throwable] extends In[Nothing]
case object Skip extends In[Nothing]
case  object ChannelClosed extends In[Nothing]
```

Continuated[B]

- ContRead
- ContWrite
- Skip
- Done
- Never

Input[A] - external API

```
trait Input[A]
{
   ……

   def  aread: Future[A]  =  <implementation…>

   def  read: A = macro <implementation … >
                                             await(aread)
   ….

   def  map[B](f: A=>B):  Input[B] = ….

   //  or,   zip,  filter, …  etc
```

+ usual   operations on streams in functional language

Output[A] - API

```
trait Output[A]
{
   type write = A

   def cbwrite(f: ContWrite[A,B]=>Option[
                            (A,Future[Continuated[B]])],
                ft: FlowTermination[B])
   …..

   def  awrite(a:A): Future[A] = ….

   def  write(a:A): A  = ….  <<  await(awrite)

   ….
```

ContWrite[A,B].F

```
case class ContWrite[A,B](
    function: F,
    channel:  Channel[A],
    flowTermination: FlowTermination[B]
)
```

Selector

$$(a \to P)\Box(b \to Q)$$

Go language:

```
go {
  for{
    select{
       case c1 -> x :   …  //  P
       case c2 <- y :   …   // Q
    }
 }
```

$$*[(c_1?x \to P)\Box(c_2!y \to Q)]$$

Scala:

```
go {
    select.forever {
        case x : c1.read =>   …  //  P
        case y : c2.write =>  …   // Q
    }
}
```

```
select.aforever {
        case x : c1.read =>   …  //  P
        case y : c2.write =>  …   // Q
}
```

Provide set of flow combinators:
forever, once, fold

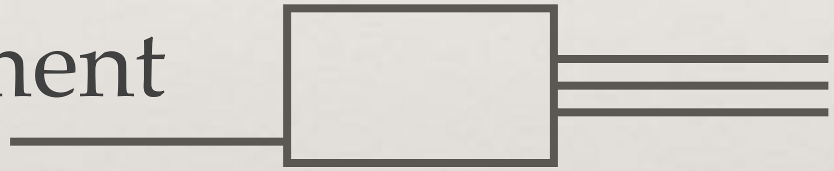
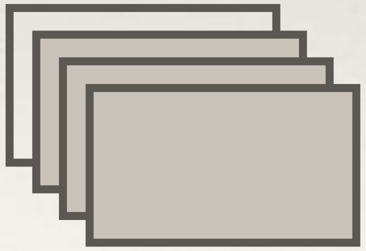
select: fold API

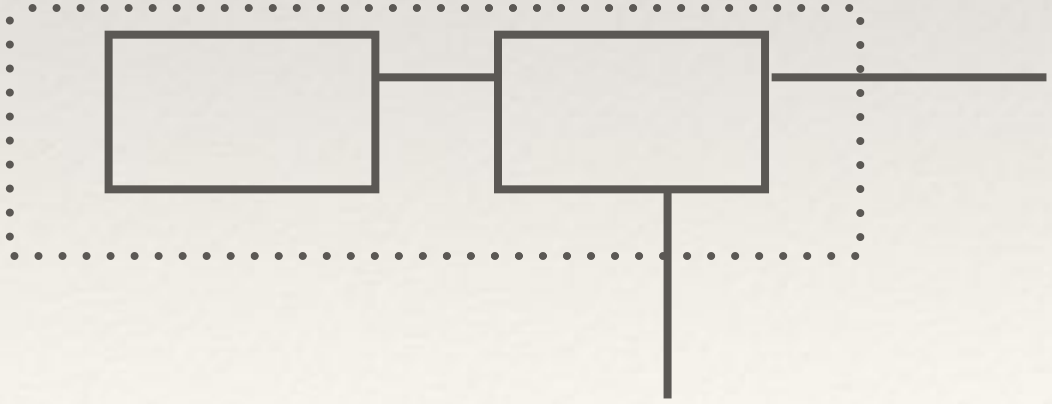
```
def fibonacci(c: Output[Long], quit: Input[Boolean]): Future[(Long,Long)] =
    select.afold((0L,1L)) { case ((x,y),s) =>
     s match {
       case x: c.write => (y, x+y)
       case q: quit.read =>
               select.exit((x,y))
     }
  }
```

fold/afold:

- special syntax for tuple support
- 's': selector pseudoobject
- s match must be the first statement
- select.exit((..)) to return value from flow

Transputer

- Actor-like object with set of input/output ports, which can be connected by channels
- Participate in actor systems supervisors hierarchy
  - SelectStatement 
  - A+B (parallel execution, common restart)
  - replicate 



Channel-based generic API

```
val listeners: Channel[Channel[T]]
val messages: Channel[T] = makeChannel[]
```

```
// private part
  case class Message(next:Channel[Message],value:T)
```

```
select.afold(makeChannel[Message]) { (bus, s) =>
   s match {
     case v: message.read => val newBus = makeChannel[Message]
                                            current.write(Message(newBus,v))
                                            newBus
     case ch: listener.read =>  select.afold(bus) { (current,s) =>
                                   s match {
                                      case msg:current.read =>  ch.awrite(msg.value)
                                                                current.write(msg)
                                                                msg.next
                                   }
              }
```

- state - channel [bus], for which all listeners are subscribed
  - on new message - send one to bus with pointer to the next bus state
    - listener on new message in bus - handle, change current and send again
  - on new listener - propagate

Channel-based generic API

```
val listener: Channel[Channel[T]]
val message: Channel[T] = makeChannel[]
```

```
// private part
  case class Message(next:Channel[Message],value:T)
```

```
select.afold(makeChannel[Message]) { (bus, s) =>
   s match {
     case v: message.read => val newBus = makeChannel[Message]
                             current.write(Message(newBus,v))
                             newBus
     case ch: listener.read =>  select.afold(bus) { (current,s) =>
                             s match {
                                  case msg:current.read =>  ch.awrite(msg.value)
                                                            current.write(msg)
                                                            msg.next
                             }
           }
          current
```

- state - channel [bus], for which all listeners are subscribed
  - on new message - send one to bus with pointer to the next bus state
    - listener on new message in bus - handle, change current and send again
  - on new listener - propagate

Channel-based generic API

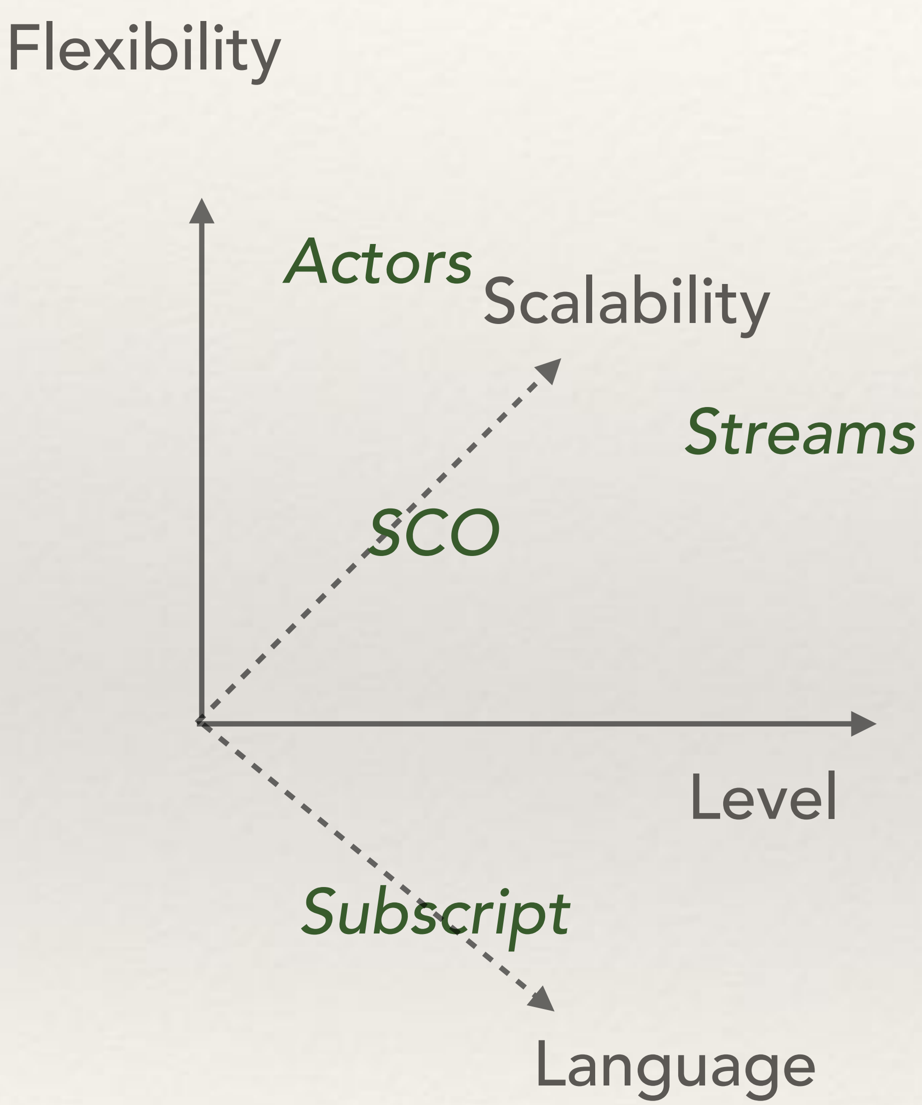
```
val listener: Channel[Channel[T]]
val message: Channel[T] = makeChannel[]
```

```
// private part
  case class Message(next:Channel[Message],value:T)
```

```
select.afold(makeChannel[Message]) { (bus, s) =>
   s match {
     case v: message.read =>  val newBus = makeChannel[Message]
                               current.write(Message(newBus,v))
                               newBus
     case ch: listener.read =>  select.afold(bus) { (current,s) =>
                              s match {
                                 case msg:current.read =>  ch.awrite(msg.value)
                                                           current.write(msg)
                                                           msg.next
                              }
                }
                current
```

- state - channel [bus], for which all listeners are subscribed
  - on new message - send one to bus with pointer to the next bus state
    - listener on new message in bus - handle, change current and send again
  - on new listener - propagate

Scala concurrency libraries



- *Actors*
  - *low level,*
  - *great flexibility and scalability*
- *Akka-Streams*
  - *low flexibility*
  - *hight-level, scalable*
- *SCO*
  - *low scalability*
  - *hight-level, flexible*
- *Reactive Isolated*
  - *hight-level, scalable,*
  - *allows delegation*
- *Gopher*
  - *can emulate each style*

Gopher vs Reactive Isolates

Gopher

- Transputer/fold
- Input
- Output

Isolates

- Isolate
- Events
- Channel

Gopher	Isolates
Many writers	One writer Channel must have owner
CSP + growing buffer	Loosely coupled (growing buffer)
Local	Distributed

Scala-gopher: early experience reports

- Not 1.0 yet
- Helper functionality in industrial software projects. (utilities, small team)
- Generally: positive
  - transformation of invocation-only hight-order methods into async form
  - recursive dynamic data flows
- Error handling needs some boilerplate

Error handling: language level issue

```
val  future = go {
   .........
   throw  some  exception
}

go {
   ..........
   throw some exception
}

Go {
   ............
   throw some exception
}
```

Core scala library:

Future.apply

(same issue)

← go { : Error is ignored

← Go { : Developers miss-up Go/go

Errors in ignored value: possible language changes.

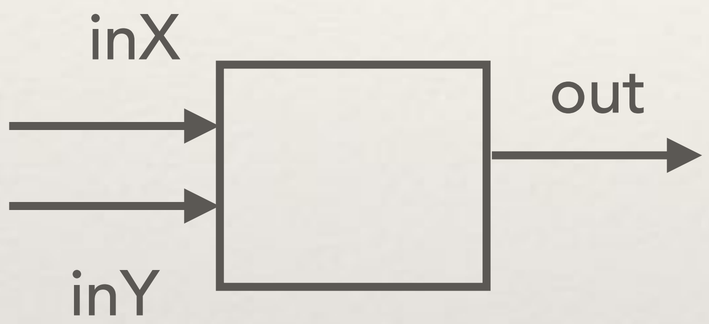
- Possible solutions:
  - Optional implicit conversion for ignored value
  - Special optional method name for calling with ignored value
  - Special return type

```
def  go[X](f: X):  Future[X]

def  go_ignored[X](f:X): Unit
```

```
trait  Ignored[F]


object Future
{
  implicit def  toIgnored(f:Future):Ignored[Future] =
….
```

```
def  go(f:X):  Ignored[Future[X]] =
```

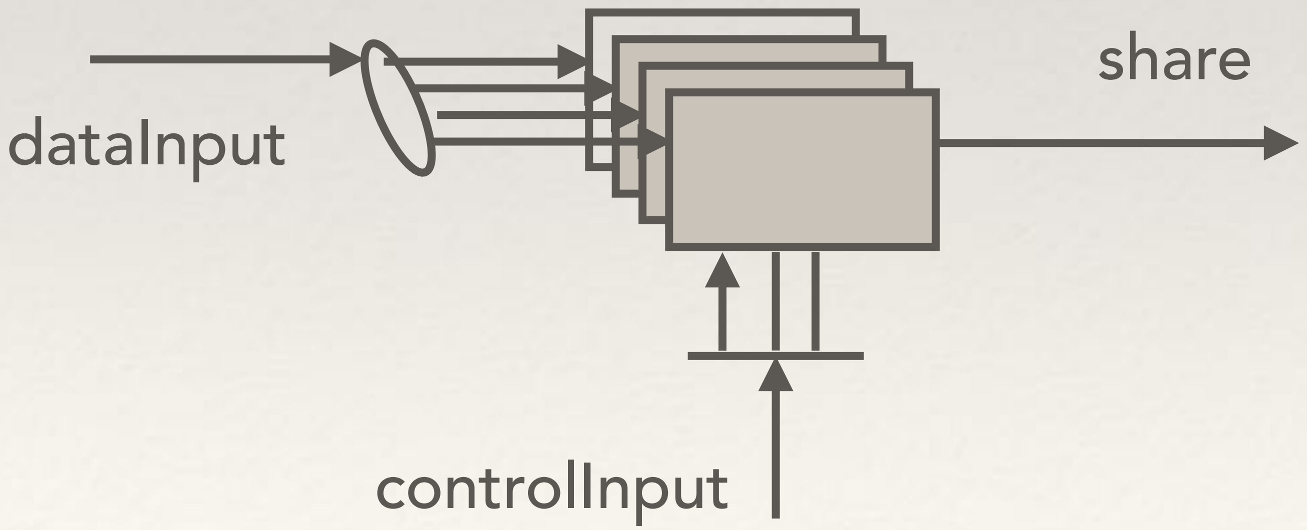
Transputer: select




```
class Zipper[T] extends SelectTranspuler
{
  val inX: InPort[T]
  val inY: InPort[T]
  val out: OutPort[(T,T)]

  loop {
    case x: inX.read =>   val y = inY.read
                          out write (x,y)
    case y: inY.read =>   val x = inX.read
                          out.write((x,y))
  }

}
```

Transputer: replicate

```
val r = gopherApi.replicate[SMTTransputer](10)
   ( r.dataInput.distribute( (_.hashCode % 10 ) ).
     .controlInput.duplicate().
      out.share()
   )
```



Programming techniques

- Dynamic recursive dataflow schemas
  - configuration in state
- Channel-based two-wave generic API
  - expect channels for reply

Dynamic recursive dataflow

```
select.fold(output){ (out, s) => s match {
  case x:input.read =>  select.once {
                          case x:out.write =>
                          case select.timeout =>  control.distributeBandwidth match {
                                   case Some(newOut) => newOut.write(x)
                                                        out | newOut
                                   case None => control.report("Can't increase bandwidth")
                                                out
                           }
                         }
  case select.timeout =>  out match {
     case OrOutput(frs,snd) => snd.close
                   frs
     case _             => out
  }
}
```

dynamically increase and decrease bandwidth in dependency from load

Dynamic recursive dataflow

```
select.fold(output){ (out, s) => s match {
  case x:input.read =>  select.once {
                          case x:out.write =>

  case select.tin
     case OrOut

     case _              => out
  }
}
```

```
case select.timeout =>
       control.distributeBandwidth match {
          case Some(newOut) => newOut.write(x)
          out | newOut
       case None =>
          control.report("Can't increase bandwidth")
          out
```

dynamically increase and decrease bandwidth in dependency from load

Dynamic recursive dataflow

```
select.fold(output){ (out, s) => s match {
  case x:input.read =>  select.once {
                          case x:out.write =>
                          case select.timeout =>  control.distributeBandwidth match {
                                   case Some(newOut) => newOut.write(x)
                                                        out | newOut
                                   case None => control.report("Can't increase bandwidth")
                                                out
                           }
                        }
```

```
case select.timeout =>  out match {
       case OrOutput(frs,snd) => snd.close
                                 frs
       case _                 => out
}    }
```

dynamically increase and decrease bandwidth in dependency from load

Channel-based generic API

- Endpoint instead function call
  - f: A=>B
  - endpoint: Channel[A,Channel[B]]
- Recursive
  - case class M(A,Channel[M])
  - f: (A,M) => M   (dataflow configured by input)

Channel-based generic API

```
trait Broadcast[T]
{
  val listeners: Output[Channel[T]]
  val messages: Output[T]

  def send(v:T):Unit = { messages.write(v)  }


  ….
```

- message will received by all listeners

Channel-based generic API

```
class BroadcastImpl[T]
{
  val listeners: Channel[Channel[T]]
  val messages: Channel[T] = makeChannel[Channel[T]]

  def send(v:T):Unit = { messages.write(v)  }

  ….
}
```

```
// private part
  case class Message(next:Channel[Message],value:T)
```

```
select.afold(makeChannel[Message]) { (bus, s) =>
    s match {
      case v: messages.read => val newBus = makeChannel[Message]
                              current.write(Message(newBus,v))
                               newBus
      case ch: listeners.read =>  select.afold(bus) { (current,s) =>
                          s match {
                              case msg:current.read =>  ch.awrite(msg.value)
                                                        current.write(msg)
                                                        msg.next
                          }
            }
          current
```

Scala-Gopher: Future directions

- More experience reports (try to use)
- Extended set of notifications
  - channel.close, overflow
- Distributed case
  - new channel types with explicit distributed semantics

Scala-Gopher: Conclusion

- Native integration of CSP into Scala is possible
  - have a place in a Scala concurrency model zoo
- Bring well-established techniques to Scala world
  - (recursive dataflow schemas; channel API)
- Translation of invocation-only high-order functions into async form can be generally recommended.
  - (with TASTY transformation inside libraries can be done automatically)

Thanks for attention

- Questions ?

- https://github.com/rssh/scala-gopher
- ruslan shevchenko: ruslan@shevchenko.kiev.ua