Reactive Programming with Algebra

André van Delft
Anatoliy Kmetyuk

LambdaConf Boulder
22-23 May 2015

Overview

- Introduction
  - Programming is Still Hard
  - Some History
  - Algebra of Communicating Processes
- SubScript
  - Example applications
  - . Debugger demo
- Dataflow
  - Twitter Client
  - SubScript Actors
- Conclusion

Programming is Still Hard

Mainstream programming languages: imperative

- good in batch processing
- not good in parsing, concurrency, event handling
- Callback Hell

Neglected idioms

- Non-imperative choice: BNF, YACC
- Data flow: Unix pipes

Math!

Algebra can be easy and fun

Area	Objects	Operations	Rules
Numbers	0, 1, ..., x, y, ...	$+ \quad \cdot \quad - \quad /$	x+y = y+x
Logic	F, T, x, y, ...	$\vee \quad \wedge \quad \neg$	x∨y = y∨x
Processes	0, 1, a, b,..., x, y, ...	$+ \quad \cdot \quad \& \quad \mid \quad \&\& \quad \parallel \quad /$	x+y = y+x

Kleene, 1951 - 1

Summary: To what kinds of events can a McCulloch—Pitts nerve net respond by firing a certain neuron? More generally, to what kinds of events can any finite automaton respond by assuming one of certain states? This memorandum is devoted to an elementary exposition of the problems and of results obtained on it during investigations in August 1951.

REPRESENTATION OF EVENTS IN NERVE NETS AND FINITE AUTOMATA

S. C. Kleene

INTRODUCTION:

1. Stimulus and Response: An organism or robot receives certain stimuli (via its sensory receptor organs) and performs certain actions (via its effector organs). To say that certain actions are a response to certain stimuli means, in the simplest case, that the actions are performed when those stimuli occur and not when they do not occur.

Kleene, 1951 - 2

7.2 <u>An algebraic transformation</u>: We list several equivalences:

(1)	$(E \vee F) \vee G$	$E \vee (F \vee G)$.	
(2)	$(EF)G$	$E(FG)$.	Associative laws
(3)	$(E^*F)G$	$E^*(FG)$.	
(4)	$(E \vee F)G$	$EG \vee FG$.	
(5)	$E(F \vee G)$	$EF \vee EG$.	Distributive laws
(6)	$E^*(F \vee G)$	$E^*F \vee E^*G$.	
(7)	E^*F	$F \vee E^*(EF)$.	
(8)	E^*F	$F \vee E(E^*F)$.	

Some History

1955	Stephen Kleene	~~> regular expressions, *
	Noam Chomsky	~~> language grammars
1960	John Backus & Peter Naur	~~> BNF
	Tony Brooker	~~> Compiler Compiler
1971	Hans Bekič	~~> Algebra of Processes
1973	Stephen Johnson	~~> YACC
1974	Nico Habermann & Roy Campbell	~~> Path Expressions
1978	Tony Hoare	~~> Communicating Sequential Processes (CSP)
1980	Robin Milner	~~> Calculus of Communicating Systems (CCS)
1982	Jan Bergstra & Jan Willem Klop	~~> Algebra of Communicating Processes (ACP)
1989	Robin Milner	~~> Pi-Calculus
	Henk Goeman	~~> Self-applicative Processes

Goeman 1989 - 1

Towards a Theory of (Self) Applicative Communicating Processes: a Short Note

Henk Goeman
Dept. of Computer Science, Leiden University

$$P, Q, R, \ldots \quad ::=$$

$$x \mid (\lambda x.P) \mid (PQ) \mid (\lambda P) \mid (P + Q) \mid (P|Q) \mid (P;Q) \mid (P\backslash\lambda) \mid (P[s]).$$

$(\lambda x.P)$ is called abstraction or input on port λ,
(PQ) is called application,
(λP) is called output on port λ,
$(P + Q)$ is called choice,
$(P|Q)$ is called parallel composition,
$(P;Q)$ is called sequential composition,
$(P\backslash\lambda)$ is called restriction,
$(P[s])$ is called port renaming.

Goeman 1989 - 2

4 Examples of process terms

1. Let $D \equiv \mu z.\alpha x.\beta(Qx); zz$
   and $O \equiv DD = \alpha x.\beta(Qx); O$.
   The process O represents an object:
   it answers QR on port β for any request R on port α.

2. Let $D \equiv \mu z.(\beta - + \alpha x.\beta x); zz$
   and $K \equiv DD = (\beta - + \alpha x.\beta x); K$.
   The process K represents a channel with default output $-$.

3. Let $D \equiv \mu z.\lambda y.\beta y; zzy + \alpha x.zzx$
   and $R \equiv DD = \lambda y.\beta y; Ry + \alpha x.Rx$
   then $RP = \beta P; RP + \alpha x.Rx$.
   The process RP represents a register with initial content P.
   Note that $\alpha x.Rx$ represents a register without initial content.

Algebra of Communicating Processes - 1

Bergstra & Klop, Amsterdam, 1982 - ...

ACP ~ Boolean Algebra

+	choice
·	sequence
0	deadlock
1	empty process

atomic actions a,b,…
parallelism
communication
disruption, interruption
time, space, probabilities
money
...

Algebra of Communicating Processes - 2

Less known than CSP, CCS

Specification & Verification

- Communication Protocols
- Production Plants
- Railways
- Coins and Coffee Machines
- Money and Economy

Strengths

- Familiar syntax
- Precise semantics
- Reasoning by term rewriting
- Events as actions

Algebra of Communicating Processes - 3

$$x+y = y+x$$
$$(x+y)+z = x+(y+z)$$
$$x+x = x$$
$$(x+y)\cdot z = x\cdot z+y\cdot z$$
$$(x\cdot y)\cdot z = x\cdot(y\cdot z)$$

$$0+x = x$$
$$0\cdot x = 0$$
$$1\cdot x = x$$
$$x\cdot 1 = x$$

$$(x+1)\cdot y = x\cdot y + 1\cdot y$$
$$= x\cdot y + y$$

Algebra of Communicating Processes - 4

$$
\begin{array}{rcl}
x \| y & = & x \mathbin{⫽} y + y \mathbin{⫽} x + x | y \\
\\
(x+y) \mathbin{⫽} z & = & \ldots \\
a \cdot x \mathbin{⫽} y & = & \ldots \\
1 \mathbin{⫽} x & = & \ldots \\
0 \mathbin{⫽} x & = & \ldots \\
\\
(x+y) | z & = & \ldots \\
\ldots & = & \ldots
\end{array}
$$

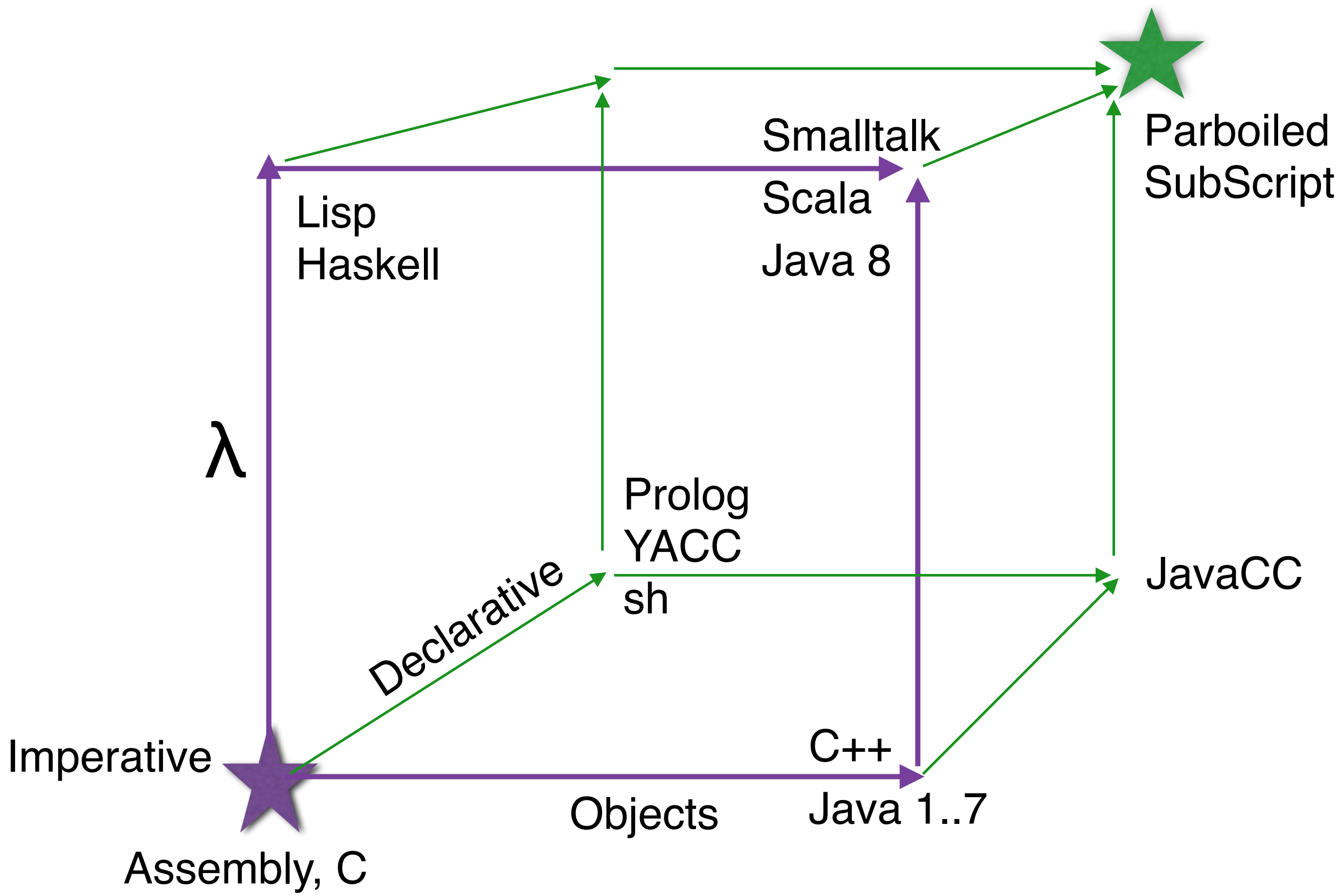
ACP Language Extensions

- 1980: Jan van den Bos - Input Tool Model [Pascal, Modula-2]
- 1988-2011: André van Delft - Scriptic [Pascal, Modula-2, C, C++, Java]
- 1994: Jan Bergstra & Paul Klint - Toolbus
- 2011-...: André van Delft - SubScript [Scala, JavaScript (?)]

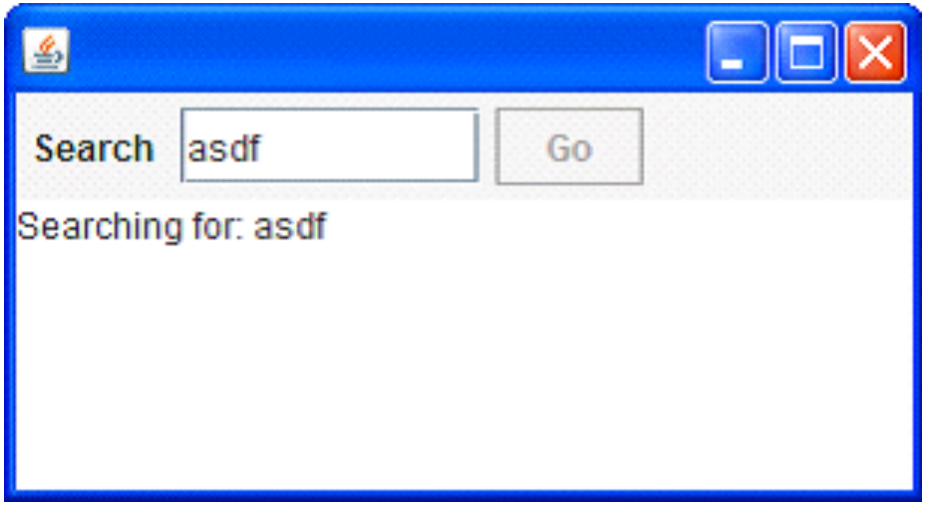
Application Areas:

- GUI Controllers
- Text Parsers
- Discrete Event Simulation
- Reactive, Actors, Dataflow

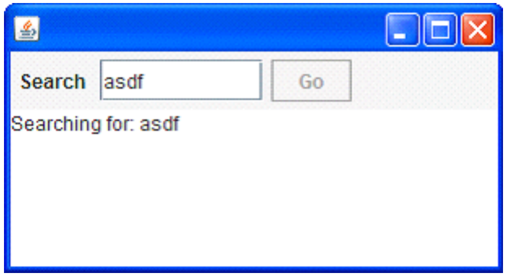
Programming Paradigms



GUI application - 1

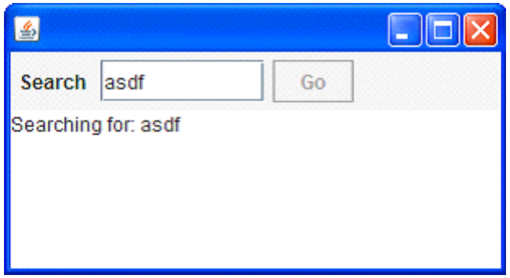


- Input Field
- Search Button
- Searching for…
- Results



GUI application - 2

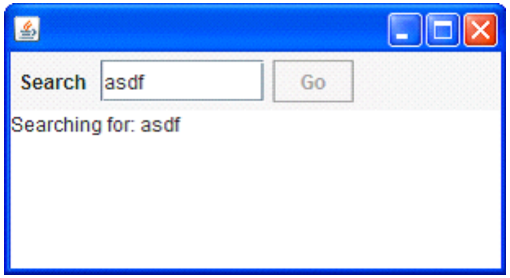
```
val searchButton = new Button("Go”) {
  reactions.+= {
    case ButtonClicked(b) =>
      enabled = false
      outputTA.text = "Starting search...”
      new Thread(new Runnable {
        def run() {
          Thread.sleep(3000)
          SwingUtilities.invokeLater(new Runnable{
            def run() {outputTA.text="Search ready”
                        enabled = true
          }})
      }}).start
  }
}
```



GUI application - 3

```
live =      searchButton
        @gui: {outputTA.text="Starting search.."}
              {* Thread.sleep(3000) *}
        @gui: {outputTA.text="Search ready"}
              ...
```

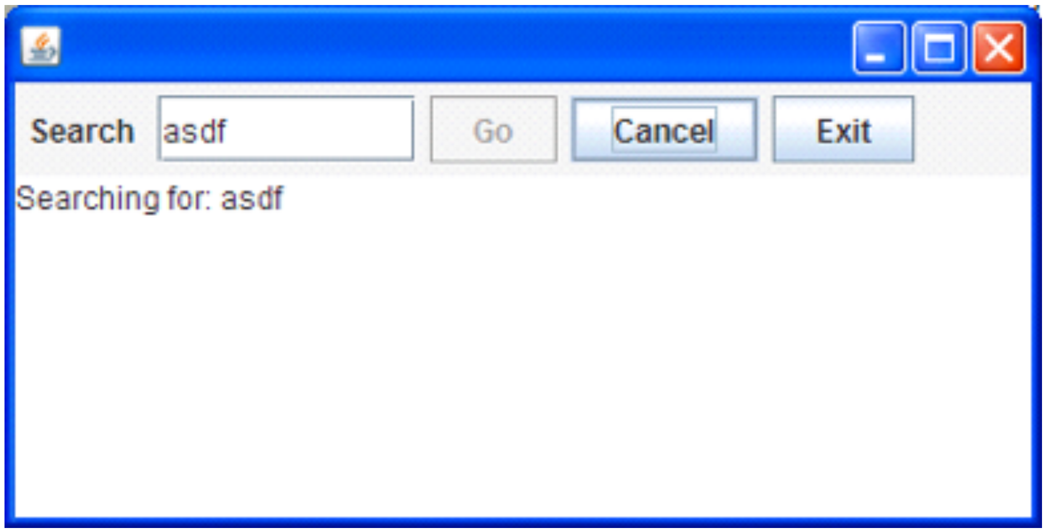
- Sequence operator: white space and `;`
- `gui`: code executor for
  - `SwingUtilities.InvokeLater`+`InvokeAndWait`
- `{* ... *}`: by executor for `new Thread`



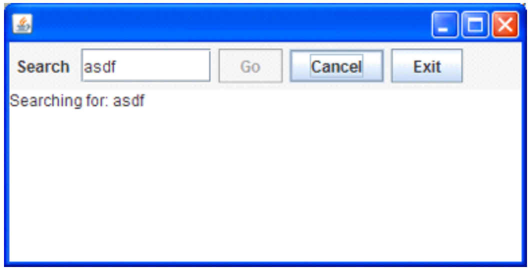
GUI application - 4

```
live              = searchSequence...

searchSequence    = searchCommand
                    showSearchingText
                    searchInDatabase
                    showSearchResults

searchCommand     = searchButton
showSearchingText = @gui: {outputTA.text = "…"}
showSearchResults = @gui: {outputTA.text = "…"}
searchInDatabase  = {* Thread.sleep(3000) *}
```

GUI application - 5



- Search: button or Enter key
- Cancel: button or Escape key
- Exit: button or ; ; "Are you sure?"…
- Search only allowed when input field **not** empty
- Progress indication



GUI application - 6

```
live              = searchSequence... || exit

searchCommand     = searchButton + Key.Enter
cancelCommand     = cancelButton + Key.Escape
exitCommand       =   exitButton + windowClosing
exit              =   exitCommand @gui:{confirmExit} ~~(b:Boolean)~~> while(!b)
cancelSearch      = cancelCommand @gui: showCanceledText

searchSequence    = searchGuard searchCommand
                    showSearchingText searchInDatabase showSearchResults
                    / cancelSearch

searchGuard       = if(!searchTF.text.isEmpty) . anyEvent(searchTF) ...

searchInDatabase  = {*Thread.sleep(3000)*} || progressMonitor
progressMonitor   = {*Thread.sleep( 250)*}
                    @gui:{searchTF.text+=here.pass} ...
```

SubScript Features

"Scripts" – process refinements as class members

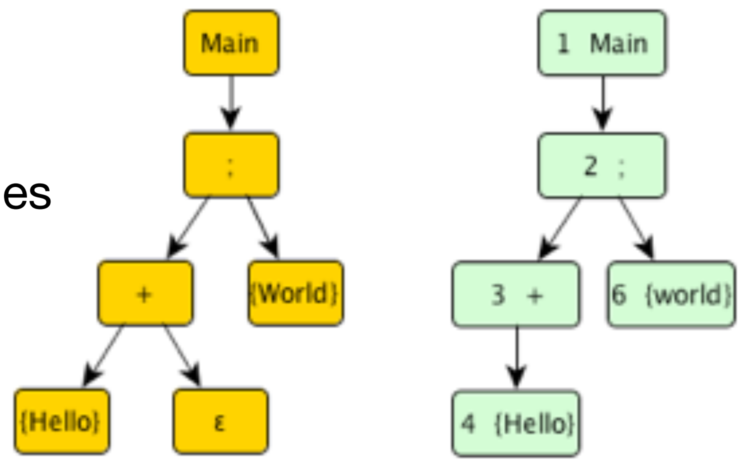
```
script a = b; {c}
```

- Much like methods: override, implicit, named args, varargs, ...
- Invoked from Scala: `_execute(a, aScriptExecutor)`
  Default executor: `_execute(a)`
- Body: process expression
  Operators: `+  ;  &  |  &&  ||  /` ...
  Operands: script call, code fragment, if, while, ...
- Output parameters: ?, ...
- Shared scripts:
  `script send,receive = {}`

Implementation - 1

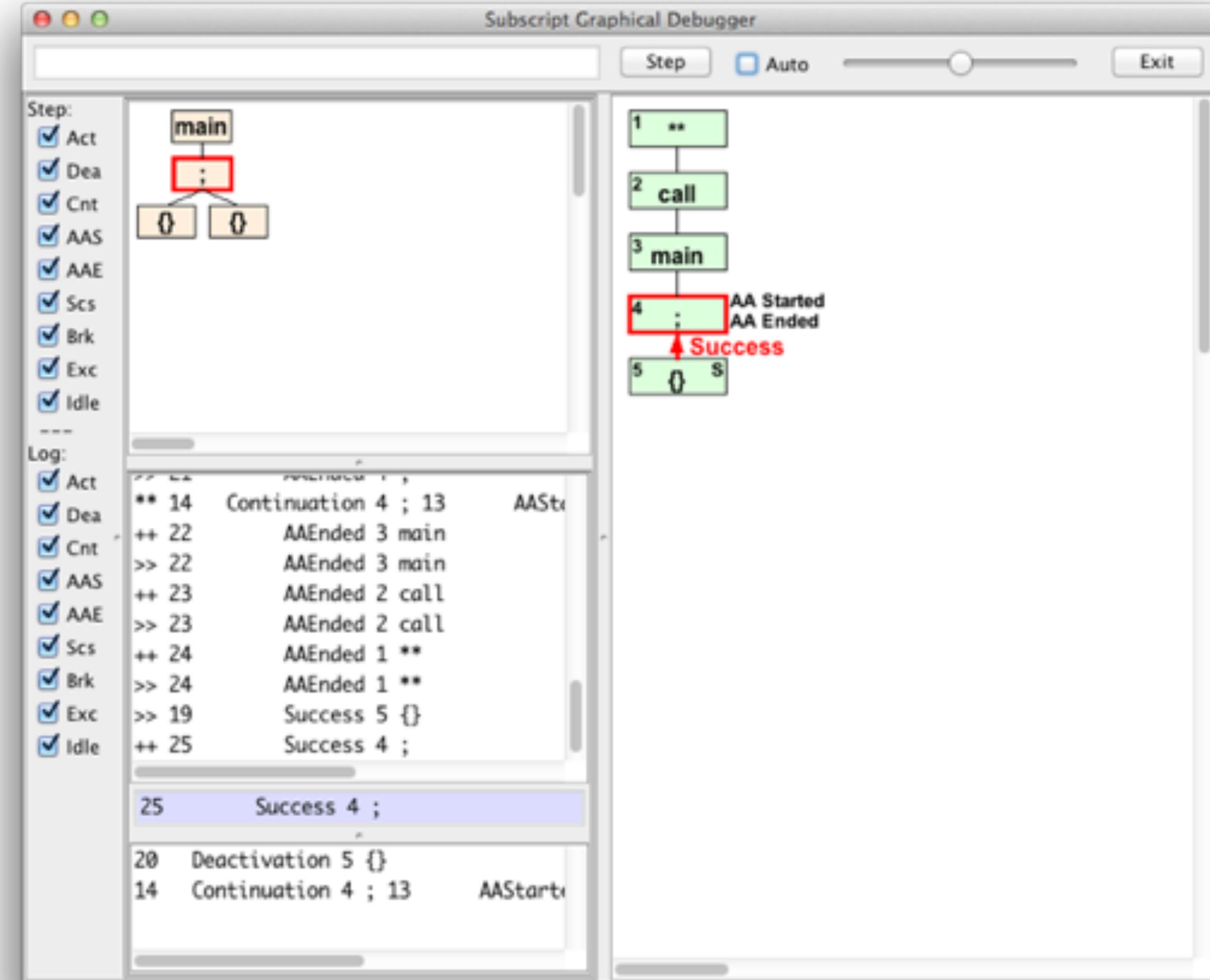
- Branch of Scalac: 1300 lines (scanner + parser + typer)

```
script Main = ({Hello} + ε); {World}

import subscript.DSL._
def Main = _script('Main) {
             _seq(_alt(_normal{here=>Hello}, _empty),
                       _normal{here=>World}          )
           }
```



- Virtual Machine: 2000 lines
  - static script trees
  - dynamic Call Graph
- Swing event handling scripts: 260 lines
- Graphical Debugger: 550 lines (10 in SubScript)

Debugger - 1



Subscript Graphical Debugger
Step
Auto
Exit
Step:
Act
Dea
Cnt
AAS
AAE
Scs
Brk
Exc
Idle
Log:
Act
Dea
Cnt
AAS
AAE
Scs
Brk
Exc
Idle
main
;
{}
{}
** 14 Continuation 4 ; 13 AASt
++ 22 AAEnded 3 main
>> 22 AAEnded 3 main
++ 23 AAEnded 2 call
>> 23 AAEnded 2 call
++ 24 AAEnded 1 **
>> 24 AAEnded 1 **
>> 19 Success 5 {}
++ 25 Success 4 ;
25 Success 4 ;
20 Deactivation 5 {}
14 Continuation 4 ; 13 AAStart
1 **
2 call
3 main
4 ;
AA Started
AA Ended
Success
5 {} S

Debugger - 2

built using SubScript

```
live          = stepping || exit

stepping      = {* awaitMessageBeingHandled(true) *}
                if shouldStep then (
                   @gui: {! updateDisplay !}
                   stepCommand || if autoCheckBox.selected then sleepStepTimeout
                )
                { messageBeingHandled(false) }
                ...

    exit          = exitCommand
                    var     isSure = false
                    @gui: { isSure = confirmExit }
                    while (!isSure)

exitCommand = exitButton + windowClosing
```

One-time Dataflow - 1

```
exit = exitCommand
       var     isSure = false
       @gui: { isSure = confirmExit }
       while (!isSure)
```

Arrows + λ's

```
exit = exitCommand  @gui:{confirmExit} ~~> (r:Boolean) => [while(!r)]

exit = exitCommand  @gui:{confirmExit} ~~> (r:Boolean) ==> while(!r)

exit = exitCommand  @gui:{confirmExit} ~~> while(!_)

exit = exitCommand  @gui:{confirmExit} ~~(r:Boolean)~~> while(!r)
```

One-time Dataflow - 2

- Script result type `script confirmExit:Boolean = ...`
- Result values `$: Try[T]`
- Result propagation `call^     {result}^`
- Data Flow `x ~~> y`
- Exception Flow `x~/~> y`
- Ternary `x ~~> y +~/~> z`
- Matching flow:

```
x ~~(b:Boolean     )~~> y1
 +~~(i:Int if i<10)~~> y2
 +~~( _            )~~> y3
+~/~(e:IOException)~~> z1
+~/~(e:  Exception)~~> z2
+~/~(e:  Throwable)~~> z3
```

Example: Slick 3

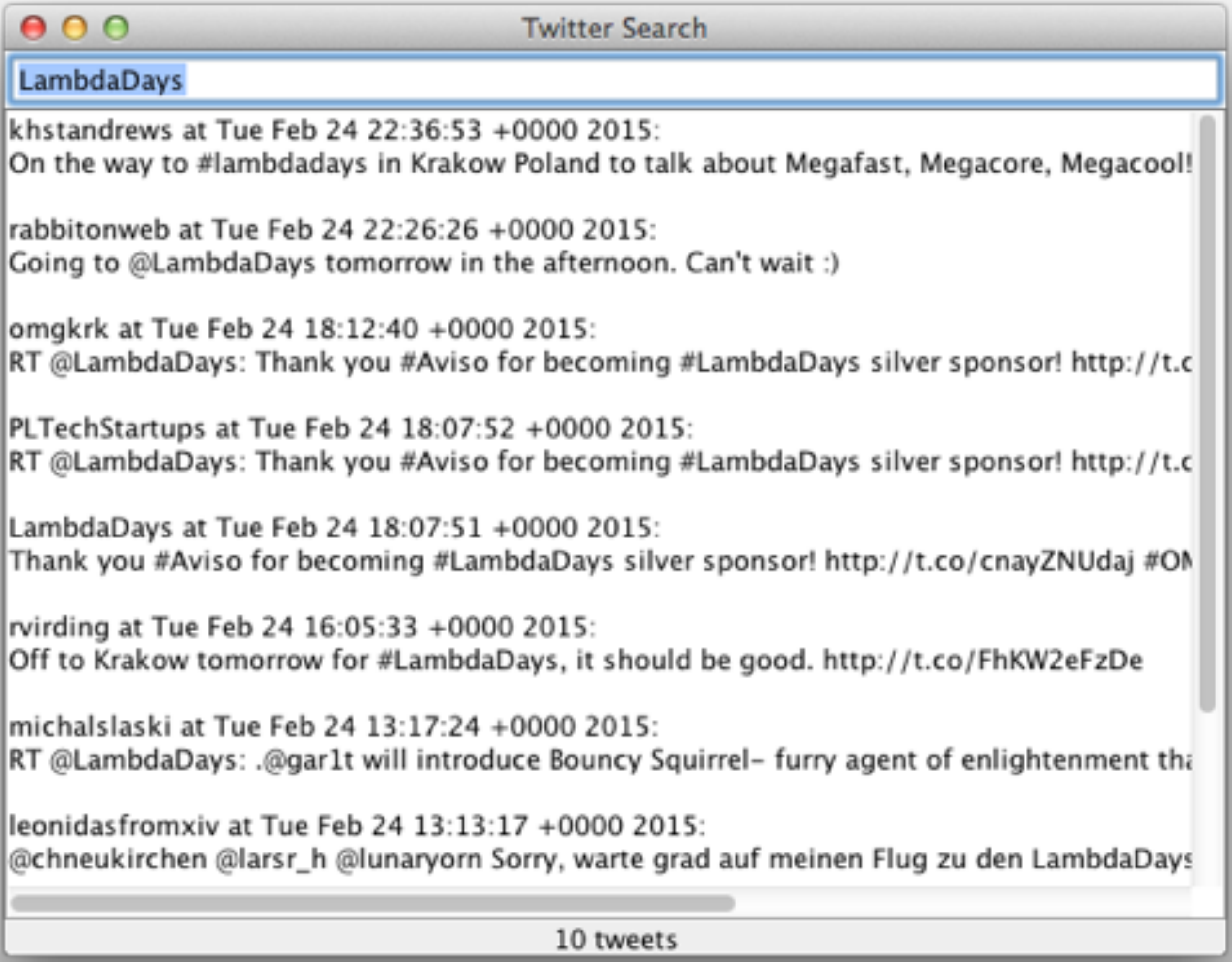
Reactive Streams for Asynchronous Database Access in Scala

http://www.infoq.com/news/2015/05/slick3

```
val q = for (c<-coffees) yield c.name
val a = q.result
val f: Future[Seq[String]] = db.run(a)

f.onSuccess { case s => println(s"Result: $s") }
```

```
val q = for (c<-coffees) yield c.name

q ~~(s)~~> println(s"Result: $s")
```

Example: Twitter Search Client - 1



Example: Twitter Search Client - 2

```
class PureController(val view: View) extends Controller with Reactor {

  def start() = {initialize; bindInputCallback}

  def bindInputCallback = {
    listenTo(view.searchField.keys)

    val fWait   = InterruptableFuture {Thread sleep keyTypeDelay}
    val fSearch = InterruptableFuture {searchTweets}

   reactions += {case _                            => fWait  .execute()
     .flatMap   {case _                            => fSearch.execute()}
     .onComplete{case Success(tweets)              => Swing.onEDT{view. ...()}
                 case Failure(e:CancelException)  => Swing.onEDT{view. ...()}
                 case Failure( e                 ) => Swing.onEDT{view. ...()}
} } } }
```

Example: Twitter Search Client - 3

```
class SubScriptController(val view: View) extends Controller {
  def start() = _execute(_live())



  script..
    live          = initialize; (mainSequence/..)...

    mainSequence = anyEvent(view.searchField)
                   waitForDelay
                   searchInBG ~~(ts:Seq[Tweet])~~> updateTweetsView(ts)
                            +~/~(t: Throwable )~~> setErrorMsg(t)

    waitForDelay = {* Thread sleep keyTypeDelay *}
    searchInBG   = {* searchTweets *}

    updateTweetsView(ts: Seq[Tweet]) = @gui: {view.set...}
    setErrorMsg     (t : Throwable ) = @gui: {view.set...}

}
```

Example: Twitter Search Client - 4

```
class SubScriptController(val view: View) extends Controller {
  def start() = _execute(_live())
  val fWait   = InterruptableFuture {Thread sleep keyTypeDelay}
  val fSearch = InterruptableFuture {searchTweets}

  script..
    live         = initialize; (mainSequence/..)...

    mainSequence = anyEvent(view.searchField)
                   fWait
                   fSearch    ~~(ts:Seq[Tweet])~~> updateTweetsView(ts)
                            +~/~(t: Throwable )~~> setErrorMsg(t)



    updateTweetsView(ts: Seq[Tweet]) = @gui: {view.set...}
    setErrorMsg     (t : Throwable ) = @gui: {view.set...}

}
```

Example: Twitter Search Client - 4

```
implicit script future2script[T](f:InterruptableFuture[T]): T
= @{f.execute()
     .onComplete {case aTry => there.executeForTry(aTry)}}: {.  .}
```

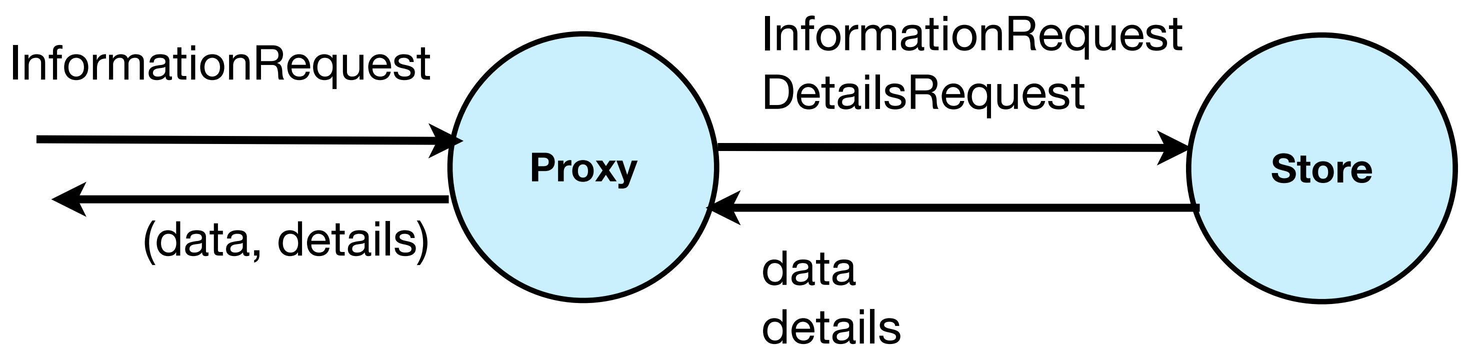
```
implicit def script2future[T](s:Script[T]): InterruptableFuture[T]
= { ... }
```

SubScript Actors: Ping Pong

```
class Ping(another: ActorRef) extends Actor {

  override def receive: PartialFunction[Any,Unit] = {case _ =>}

      another ! "Hello"
      another ! "Hello"
      another ! "Terminal"
}
```

```
class Pong extends SubScriptActor {

  implicit script str2rec(s:String) = << s >>

  script ..
    live = "Hello" ... || "Terminal" ; {println("Over")}
}
```

SubScript Actors: DataStore - 1




```
class DataStore extends Actor {

  def receive = {
    case InformationRequest(name) => sender ! getData    (name)
    case DetailsRequest     (data) => sender ! getDetails(data)
  }

}
```

SubScript Actors: DataStore - 2

```
class DataProxy(dataStore: ActorRef) extends Actor {

  def waitingForRequest = {
    case req: InformationRequest =>
      dataStore ! req
      context become waitingForData(sender)
  }

  def waitingForData(requester: ActorRef) = {
    case data: Data =>
      dataStore ! DetailsRequest(data)
      context become waitingForDetails(requester, data)
  }

  def waitingForDetails(requester: ActorRef, data: Data) = {
    case details: Details =>
      requester ! (data, details)
      context become waitingForRequest
  }
}
```

SubScript Actors: DataStore - 3

```
class DataProxy(dataStore: ActorRef) extends SubScriptActor {

  script live = << req: InformationRequest
                 => dataStore ! req
                ==>
                    var response: (Data, Details) = null
                    << data: Data
                    => dataStore ! DetailsRequest(data)
                   ==>
                       << details:Details ==> response = (data,details) >>
                    >>
                    {sender ! response}
                 >>
                 ...
}
```

SubScript Actors: DataStore - 4

```
class DataProxy(dataStore: ActorRef) extends SubScriptActor {

  script live =
    << req: InformationRequest ==> {dataStore ? req}
                   ~~(data:Data)~~> {dataStore ? DetailsRequest(data)}
           ~~(details:Details)~~> {  sender  ! (data, details)}
    >>
    ...
}
```

SubScript Actors: Shorthand Notations

```
<< case a1: T1 => b1 ==> s1
   case a2: T2 => b2 ==> s2
   ...
   case an: Tn => bn ==> sn >>
```

```
<< case a1: T1 => b1
   case a2: T2 => b2
   ...
   case an: Tn => bn >>
```

```
<< case a1: T1
   case a2: T2
   ...
   case an: Tn >>
```

```
<< case a: T => b ==> s >>

<< a: T => b ==> s >>

<< a: T => b >>

<< a: T  >>

<< 10 >>
```

SubScript Actors: Implementation - 1

```
trait SubScriptActor extends Actor {
  private val callHandlers = ListBuffer[PartialFunction[Any, Unit]]()

  def _live(): ScriptNode[Any]
  private def script terminate = Terminator.block
  private def script die       = {if (context ne null) context stop self}

  override def aroundPreStart() {
   runner.launch( [ live || terminate ; die ] )
   super.aroundPreStart()
  }

  override def aroundReceive(receive: Actor.Receive, msg: Any) {
    ...
    callHandlers.collectFirst {
      case handler if handler isDefinedAt msg => handler(msg) } match {
        case None    => super.aroundReceive( receive          , msg)
        case Some(_) => super.aroundReceive({case _: Any =>}, msg)
  } }
 ...
}
```

SubScript Actors: Implementation - 2

```
<< case a1: T1 => b1 ==> s1
   case a2: T2 => b2 ==>
   ...
   case an: Tn => bn ==> sn >>
```

```
r$(case a1: T1 => b1; [s1]
   case a2: T2 => b2; null
   ...
   case an: Tn => bn; [sn])
```

```
trait SubScriptActor extends Actor {
  ...
  script r$(handler: PartialFunction[Any, ScriptNode[Any]]) =

    var s:ScriptNode[Any]=null
    @{val handlerWithAA = handler andThen {hr => {s = hr; there.eventHappened}}
                          synchronized {callHandlers += handlerWithAA}
      there.onDeactivate {synchronized {callHandlers -= handlerWithAA}}
    }:
    {. .}
    if s != null then s
}
```

Conclusion

- Easy and efficient programming
- 10^4...10^5 actions per second
- Simple implementation: 6000 lines, 50%
  - Scalac branch ~~> Parboiled (like `ScalaTex`) + Macro's
  - VM
  - scripts for actors, swing
- Open Source:
  subscript-lang.org
  github.com/AndreVanDelft/scala
- Still much to do: JS, NodeJS, ACP style communication, ...
- and to discover: arXiv paper "New directions in ACP research"
- To join the project: andre.vandelft@gmail.com