Dataflow Constructs for a Language Extension based on the Algebra of Communicating Processes

André van Delft

1 July 2013

Presentation at

LaME 2013 / Scala’13

Montpellier

Overview

- Programming is Still Hard
- Algebra of Communicating Processes
- SubScript Now
  - Scala Extension
  - Implementation
  - Debugger demo
- New Dataflow Constructs
  - One-time Flow
  - Lasting Flow
  - 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
- Process Algebra: ACP

Math!

Algebra of Communicating Processes - 1

Bergstra & Klop, Amsterdam, 1982 - ...

ACP ~ Boolean Algebra

$+$	choice
$\cdot$	sequence
0	deadlock
1	empty process

atomic actions a,b,…

parallelism

communication

disruption, interruption

...

Algebra of Communicating Processes - 2

Specification & Verification

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

Strengths

- Familiar syntax
- Precise semantics for humans+machines
- Reasoning by term rewriting
- Events as actions

X

Algebra of Communicating Processes - 3

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

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

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

34

5

Algebra of Communicating Processes - 4

$$x \| y = x ⫝̸ y + y ⫝̸ x + x | y$$

$$(x+y) ⫝̸ z = \ldots$$
$$a \cdot x ⫝̸ y = \ldots$$
$$1 ⫝̸ x = \ldots$$
$$0 ⫝̸ x = \ldots$$

$$(x+y) | z = \ldots$$
$$\ldots = \ldots$$

X

ACP Language Extensions

- 1980: Jan van den Bos - Input Tool Model
- 1988-2011: AvD - Scriptic
  - Pascal, Modula-2, C, C++, Java
- 2011-...: AvD - SubScript
  - Scala (active+passive quality)
  - JavaScript, ... (?)
- Application Areas
  - GUI Controllers
  - Text Parsers
  - Discrete Event Simulation
  - Dataflow Programming (work in progress)
  - Parallel Processing (TBD)

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, ...
- Shared scripts:
  ```
  script send,receive = {}
  ```

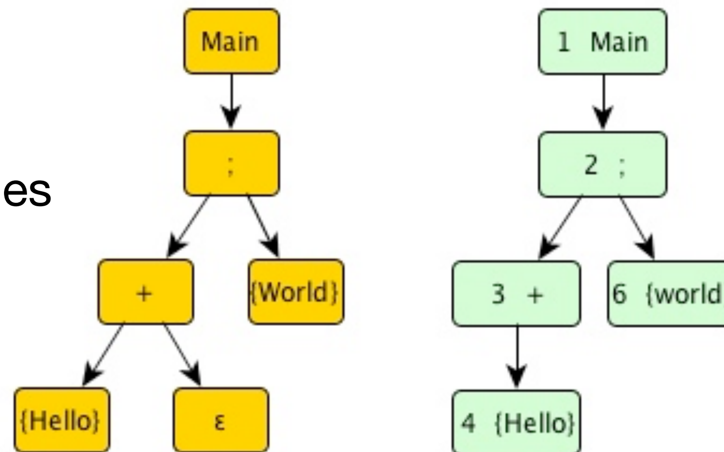
Implementation - 1

- 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 *}
              if (shouldStep)
              (  @gui: {! updateDisplay !} stepCommand
              || if (autoCheckBox.isChecked) waitForStep
              )
              { messageBeingHandled = false }
              ...

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

exitCommand = exitButton + windowClosing
```

One-time Dataflow

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

exit     = exitCommand; @gui:confirmExit ==> while(!_)
```

- `x==>y` definition `do_flowTo(<x^>,<y^>)`

```
do_flowTo[T,U](s:script[T],t:T=>script[U]): U = s ; t($s)^
```

- Script result type `script confirmExit:Boolean = ...`
- Result values `$` `$s` `t^`
- Script Lambda's `<b:Boolean => while(!b)>`

Lasting Dataflow - 1

```
def copy(in: File, out: File): Unit = {

  val  inStream = new  FileInputStream(in)
  val outStream = new FileOutputStream(out)

  val eof = false
  while (!eof) {
    val b = inStream.read()
    if (b==-1) eof=true
    else outStream.write(b)
  }

   inStream.close()
  outStream.close()
}
```

Lasting Dataflow - 2

```
fileCopier(in:File, out:File)   =   reader(in) &==> writer(out)

reader(f:File)   = val inStream = new FileInputStream(f);
                   val b = inStream.read()  <=b   while (b!=-1);
                   inStream.close

writer(f:File)   = val outStream = new FileOutputStream(f);
                   =>?i: Int   while (i!=-1)    outStream.write(i);
                   outStream.close

   <==>(i:Int)  = {}
```

Lasting Dataflow - 3

```
fileCopier  (in:File, out:File) = reader,in &==> writer,out

fileCrFilter(in:File, out:File) = reader,in &==> crFilter
                                            &==> writer,out

 crFilter  =  =>?c:Int  if(c!='\r') <=c  while(c!=-1)
```

Performance ~ 50 k actions / second

Optimization - TBD

Akka Receive: Partial scripts - 1

```
def receive = {
  case Request   (r) => sender ! calculate(r)
  case Shutdown      => context.stop(self)
  case Dangerous (r) => a.tell(Work(r),sender)
  case OtherJob  (r) => a!JobRequest(r,sender)
  case JobReply(r,s) => s ! r
}
```

```
live = .. <<
  case Request   (r) => {sender ! calculate(r)}
  case Dangerous (r) => {a.tell(Work(r),sender)}
  case OtherJob  (r) => {a!JobRequest(r,sender)}
  case JobReply(r,s) => {s!r}
  >> ;
  << Shutdown >>
```

Akka Receive: Partial scripts - 2

Plain Scala

```
var initializationReady = false
var activeActors        = 0
var sum: Double         = 0

def receive = {
  case context: Context =>
    sum = 0 //reset the instance variables
    activeActors = 0
    for(task <- context.tasks) {
      val actor = actorOf[Delegate].start
      actor ! DoTask(task)
      activeActors += 1
    }
    initializationReady = true

  case delegateResult: Double =>
    sum += delegateResult; sender.get.stop
    activeActors -= 1
    if(initializationReady && activeActors<=0) {
      clientActor ! sum
}   }
```

Akka Receive: Partial scripts - 3

```
live = ...
       << context: Context =>
            var sum: Double = 0
            ( for(task <- context.tasks)
            & {!val actor = actorOf[Delegate].start
                actor ! DoTask(task) !}
              << d:Double => {sum+=d; sender.get.stop} >>
            )
            {clientActor ! sum}
       >>
```

Conclusion

- Easy and efficient programming
- Support in Scalac branch
- Simple implementation: 5000 lines, 50%
- Still much to do and to discover
- Open Source:
  subscript-lang.org
  github.com/AndreVanDelft/scala
- Help is welcome
  Participate!