

# Concurrent Join Calculus in Scala

Sergei Winitzki

Scala by the Bay 2016

November 11, 2016

# Concurrency vs. Parallelism vs. Asynchrony

## Parallelism

Parallelism means...

- ...to use multithreading to speed up a *sequential* computation
  - ▶ main problem: to “parallelize” a computation efficiently
- parallel collections, dataflow, Spark, ...

Typical task: count words in 10,000 text files

# Concurrency vs. Parallelism vs. Asynchrony

## Asynchrony

Asynchrony means...

- ...to optimize *sequential* computations that may have long wait times
  - ▶ main problem: to interleave wait times on a single-thread runloop
- futures/promises, async/await, streams, FRP, coroutines, ...

Typical task: implement interactive Excel tables with auto-updating cells

# Concurrency vs. Parallelism vs. Asynchrony

## Concurrency

Concurrency means...

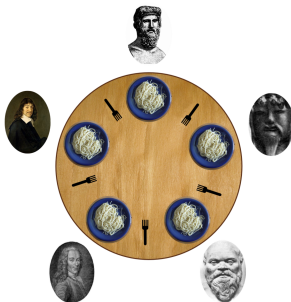
- ...mutually interacting computations, running in unknown order
  - ▶ main problem: to decide when to start a new process (or to wait)
- Thread, synchronized, semaphore, ...

Typical task: simulate “dining philosophers”

# Dining philosophers

The exemplary problem of concurrency

Five philosophers sit at a round table, taking turns eating and thinking for random time intervals



Problem: run the process, avoiding deadlock

# Problems with concurrency

Imperative concurrency is difficult:

- callbacks, threads, semaphores, mutexes, shared mutable state...
- testing is hard – non-deterministic runtime behavior!
  - ▶ race conditions, deadlocks, livelocks

We try to *avoid* concurrency whenever possible!

# How I learned to forget deadlocks and to love concurrency

In this talk:

- Introduction to the “**join calculus**” style of concurrency
- **JoinRun** -- a new Scala implementation
- Examples and demos

# Join Calculus: The new hope

...and some new hype

Join Calculus is ...

- ...a declarative language for general-purpose concurrency
- “What if actors were stateless, auto-started, and type-safe”
- No threads/semaphores/locks/mutexes/forks, no shared mutable state
- Concurrency is data-driven, not scheduled
- Easier to use than anything I’ve seen so far!

Metaphors for join calculus:

- “chemical reactions”
- “concurrent pure functions” applied to “concurrent data”



# Join Calculus: The genesis

a.k.a. the “Reflexive Chemical Abstract Machine” [Fournet & Gonthier 1996]

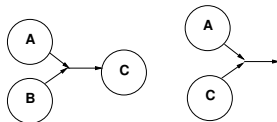
Abstract chemistry: “molecules” and “reactions”

- Chemical “soup” contains many “molecules”
- A combination of certain molecules starts a “chemical reaction”

“Chemical laws”:

$a + b \rightarrow c$

$a + c \rightarrow \emptyset$



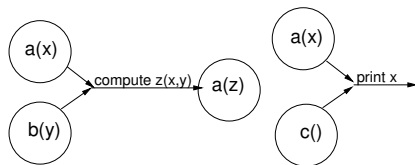
- Define molecules  $a$ ,  $b$ ,  $c$ , ... and arbitrary chemical laws
- Inject some molecules into the “soup”
- The runtime system evolves the soup *concurrently*

# Join Calculus in a nutshell

“Better concurrency through chemistry”

Translating the “chemical metaphor” into practice:

- Each molecule carries a **value**
- Each reaction computes a “molecule-valued” **expression** that depends on input values
- Resulting molecules are injected back into the soup
- Reactions start concurrently if input molecules are available



```
join(  
  run { case a(x) + b(y) =>  
    val z = compute_z(x,y); a(z) },  
  run { case a(x) + c() =>  
    println(x) } )
```

When reaction starts: input molecules disappear, expression is computed, output molecules are injected

# Using JoinRun: basic features

Molecule injectors, reaction definitions

Define some **molecule injectors**:

```
val a = new JA[Int]
val b = new JA[Option[Int]]
```

Declare some **reactions** using the known molecules:

```
val r0 = run { case a(0) => println("finished") }
val r1 = run { case a(x) + b(None) => a(x-1) }
val r2 = run { case a(x) + b(Some(y)) => a(x+y) }
```

Activate a “**join definition**” and inject some molecules:

```
join(r0, r1, r2)
a(10) // non-blocking
b(Some(5)) // non-blocking
b(None) // non-blocking; will have a(14)
```

# Using JoinRun: more features

## Blocking vs. non-blocking molecules

### Blocking molecule:

- implicitly carries a “reply” injector
- the “reply” value will be returned to caller

```
val getN = new JS[Unit, Int]
// revise the join definition, appending this reaction:
... val r3 = run { case r(x) + getN(_, reply) => reply(x) }
join(r0, r1, r2, r3)
// inject non-blocking molecules...
// now inject the blocking molecule:
val x = getN() // blocking call, returns Int
```

# JoinRun: Examples I

First benchmark: Counting to zero

Non-blocking counter:

```
val c = ja[Int]("counter")
val g = js[Unit,Int]("getValue")
val d = ja[Unit]("decr")
val f = js[LocalDateTime, Long]("finished")

join(
  run { case c(0) + f(t, reply) =>
    val elapsed = t.until(LocalDateTime.now, ChronoUnit.MILLIS)
    reply(elapsed) },
  run { case g(_,reply) + c(n) => c(n) + reply(n) },
  run { case c(n) + d(_) if n > 0 => c(n-1) }
)
val initialTime = LocalDateTime.now
c(1000)
(1 to 1000).foreach{ _ => d() }
val result = f(initialTime)
```

# JoinRun: Examples II

## Options, Futures, and Map/Reduce

Future with blocking poll (“`get`”):

```
{ case fut( (f,x) ) => finished( f(x) ) }  
{ case get(_, r) + finished(fx) => r(fx) }
```

Map/Reduce:

```
{ case res(list) + c(s) => res(s::list) }  
{ case get(_, reply) + res(list) => reply(list) }  
res( Nil )
```

```
Seq(1,2,3).foreach(x => c(x*2))  
get() // this returned Seq(4,6,2) in one test
```

# JoinRun: Examples III

## Five Dining Philosophers

Philosophers 1, 2, 3, 4, 5; forks f12, f23, f34, f45, f51.

```
// ... some declarations omitted for brevity
join (
  run{ case t1(_) => wait(); hA() },
  run{ case t2(_) => wait(); hB() },
  run{ case t3(_) => wait(); hC() },
  run{ case t4(_) => wait(); hD() },
  run{ case t5(_) => wait(); hE() },
  run{ case h1(_) + f12(_) + f51(_) => wait(); t1() + f12() + f51() },
  run{ case h2(_) + f23(_) + f12(_) => wait(); t2() + f23() + f12() },
  run{ case h3(_) + f34(_) + f23(_) => wait(); t3() + f34() + f23() },
  run{ case h4(_) + f45(_) + f34(_) => wait(); t4() + f45() + f34() },
  run{ case h5(_) + f51(_) + f45(_) => wait(); t5() + f51() + f45() }
)
t1() + t2() + t3() + t4() + t5()
f12() + f23() + f34() + f45() + f51()
```

# JoinRun: Examples IV

## Concurrent merge-sort

The `mergesort` molecule is “recursive”:

- receives the upper-level “`sortedResult`” molecule
- defines its own “`sorted`” molecules in *local scope*
- emits upper-level “`sortedResult`” when done

```
val mergesort = new JA[(Array[T], JA[Array[T]])]  
join(  
  run { case mergesort((arr, sortedResult)) =>  
    if (arr.length <= 1) sortedResult(arr)  
    else {  
      val sorted1 = new JA[Array[T]]  
      val sorted2 = new JA[Array[T]]  
      join(  
        run { case sorted1(x) + sorted2(y) => sortedResult(arrayMerge(x,y))  
      }  
      val (part1, part2) = arr.splitAt(arr.length/2)  
      // inject lower-level mergesort  
      mergesort(part1, sorted1) + mergesort(part2, sorted2)  
    }  
  })
```



# Additional features of JoinRun

Need more bells and whistles

- Per-reaction thread pools:

```
val tp1 = new JProcessPool(threads = 1)
val tp8 = new JProcessPool(threads = 8)
join( run { case a(x) => ... } onThreads tp1,
      run { case b(x) => ... } onThreads tp8
    )
```

- Auto-resume failed reactions:

```
run { case a(x) + b(y) => if (bad) throw new Exception(); ... }
```

JoinRun will reinject input molecules if exception is thrown

# Other approaches to concurrency

- STM
- Erlang's message-passing  $\approx$  Akka's "actors"
- CSP,  $\pi$ -calculus, **join calculus** (academic so far)

# Join Calculus in the wild

- Previous implementations:
  - ▶ Funnel [M. Odersky et al., 2000]
  - ▶ JOCaml ([jocaml.inria.fr](http://jocaml.inria.fr)) [Fournet et al. 2003]
  - ▶ “Join in Scala” compiler patch [V. Cremet 2003]
  - ▶ Joinads (F#, Haskell) [Petricek and Syme 2011]
  - ▶ ScalaJoins [P. Haller 2008]
  - ▶ Scala Join [J. He 2011]
  - ▶ CocoaJoin (iOS), AndroJoin (Android) [S.W. 2013]
- **JoinRun** -- a new JC prototype in Scala (this talk)
  - ▶ Can use thread pools or Akka actor pools
  - ▶ Better syntax, more checks of code sanity
  - ▶ Automatic fault tolerance

# Comparison: Join Calculus vs. Actor model

Reaction  $\approx$  actor; injected molecule  $\approx$  message to actor.

Actors:

- need to be created and managed explicitly
- will process one message at a time
- typically hold mutable state

Reactions:

- autostart when required input molecules are available
- many reactions can start at once, automatically concurrent
- immutable, stateless, and type-safe
- all reactions are defined statically, but locally scoped

# And I thought “actors” were easy...

Akka's **documentation for the Actor class**:

Actor, ActorSystem, Props (but note the 4 edge cases and 2 warnings)

Actor's companion object; ActorRef

inbox, self, sender, context, supervisorStrategy, watch

Actor lifecycle, Actor selection

send, receive, receive timeout, forward

Future, pipeTo

exceptions, exceptions vs. Future callbacks, andThen

stop, gracefulStop, PoisonPill, Kill

become, unbecome, upgrade, stash

This was item 1 in the Actors documentation.

There are 14 further items...

# Everything you need to know about JC...

Most descriptions of JC use the “message/channel” metaphor...

“Chemistry”	JC terminology	JoinRun
molecule	message on channel	<code>a(123) // side effect</code>
injector	channel (port) name	<code>val a : JA[Int]</code>
blocking injector	blocking channel	<code>val q : JS[Int]</code>
reaction	process	<code>run { case a(x) + ... }</code>
injecting a molecule	sending a message	<code>a(123) // side effect</code>
join definition	join definition	<code>&lt; □ join(r1, r2, ...) &gt;</code>

# Conclusions and outlook

- Join Calculus = declarative, purely functional concurrency
- Similar to “Actors”, but far easier and “more purely functional”
- Very little known, and very little used in practice
- Existing literature is not suitable as introduction to practical use
- A new Scala implementation, JoinRun, is in the works