Concurrent Join Calculus in Scala

Sergei Winitzki

Scala by the Bay 2016

November 11, 2016

What is "Join Calculus"?

"Join calculus" is...

- ...a programming language for concurrent computations...
- ...largely unknown and unused by the software engineering community

Parallelism vs. Asynchrony vs. Concurrency

Parallelism

Parallelism means...

- ...to use multithreading to speed up a *sequential* computation
  - main problem: to "parallelize" a computation efficiently
- parallel collections, map/reduce, 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...

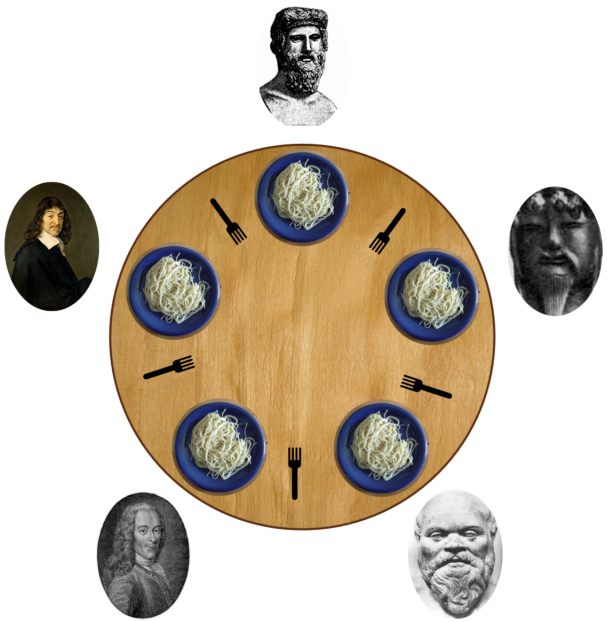
- ...mutually interacting computations, running in unknown order
  - main problem: to decide when to start a new process (or to wait)
- `Thread`, `synchronized`, `semaphore`

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 and starvation

Problems with concurrency

Imperative concurrency is difficult to reason about:

- 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!

Metaphor for join calculus:

- "chemical reactions"

Join Calculus: The genesis

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

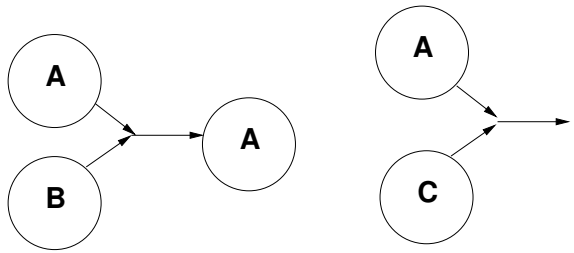
Abstract chemistry:

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

"Chemical laws":

$$\texttt{a} + \texttt{b} \to \texttt{a}$$

$$\texttt{a} + \texttt{c} \to \emptyset$$



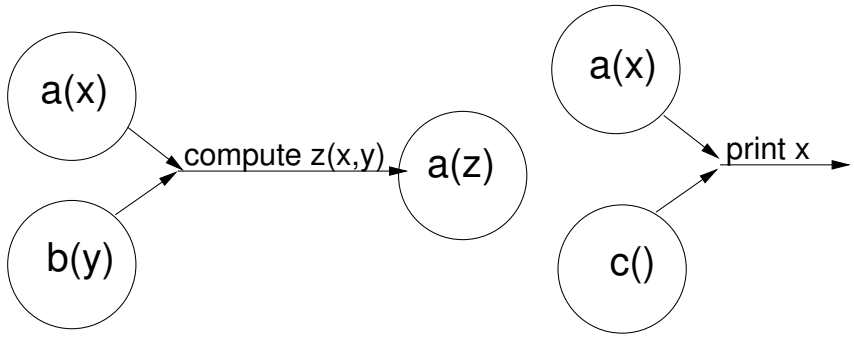
- Define molecules `a`, `b`, `c`, ... and arbitrary chemical laws
- Inject some molecules into the "soup"
- The runtime system evolves the soup *concurrently*

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

Example: concurrent counter

Chemical laws:

- counter(n) + decr() => counter(n-1)
- counter(n) + incr() => counter(n+1)

"Data stays on the molecules"

Using JoinRun: basic features

Molecule injectors, reaction definitions

Define **molecule injectors**:

```
val counter = jA[Int]

val decr = jA[Unit]
val incr = jA[Unit]
```

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

```
val r0 = run { case counter(n) + decr() => counter(n-1) }

val r1 = run { case counter(n) + incr() => counter(n+1) }
```

Activate a "**join definition**" and inject some molecules:

```
join(r0, r1)

counter(10) // non-blocking side-effect

incr() // ditto; now we have counter(11)
incr() // now we have counter(12)
```

- Calling `counter(10)` returns `Unit` and injects molecule as a side-effect

Using JoinRun: more features

Blocking vs. non-blocking molecules

Blocking molecule:

- injection will block until reply is received
- implicitly carries a pseudo-molecule injector: "reply"
- when the "reply" is injected, the value will be returned to caller
- Example:

```
f(x, replyToF) + c(y) => val z = ...; replyToF(z)
```

Using JoinRun: more features

Blocking molecules in JoinRun

Implement blocking access to the value `x` in `counter(x)`

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

JoinRun: Examples I

First benchmark: Counting to zero

Concurrent 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"):

```
fut( (f,x) ) => finished( f(x) )
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(); h1() },
  run{ case t2(_) => wait(); h2() },
  run{ case t3(_) => wait(); h3() },
  run{ case t4(_) => wait(); h4() },
  run{ case t5(_) => wait(); h5() },
  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()
```

Additional features of JoinRun

More bells and whistles

- Per-reaction thread pools:

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

Roadmap for `JoinRun`

Need still more bells and whistles

- Graceful global shutdown
- Resilience to failure in join definitions
- Runtime diagnostics, health monitoring
- Run on a cluster ("Distributed Join Calculus")

What else is needed for industry-readiness?

JoinRun: Examples IV

Concurrent merge-sort: chemistry

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

```
mergesort( (arr, sortedResult) ) =>
        val (part1, part2) = arr.splitAt(arr.length/2)
        sorted1(x) + sorted2(y) => sortedResult( arrayMerge(x,y) )
      // inject lower-level mergesort
        mergesort(part1, sorted1) + mergesort(part2, sorted2)
```

JoinRun: Examples IV

Concurrent merge-sort: JoinRun code

```
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)
    }
  })
```

Join Calculus in the wild

- Previous implementations:
  - Funnel [M. Odersky et al., 2000]
  - Join Java [von Itzstein et al., 2001-2005]
  - JOCaml (jocaml.inria.fr) [Fournet et al. 2003]
  - "Join in Scala" compiler patch [V. Cremet 2003]
  - Joins library for .NET [P. Crusso 2006]
  - ScalaJoins [P. Haller 2008]
  - Joinads (F#, Haskell) [Petricek and Syme 2011]
  - ScalaJoin [J. He 2011]
  - CocoaJoin (iOS), AndroJoin (Android) [S.W. 2013]
- JoinRun -- a new JC prototype in Scala (this talk)
  - Better syntax, more checks of code sanity
  - (Some) automatic fault tolerance
  - Fair scheduling of reactions
  - Can use thread pools or Akka event-driven actor pools

Other approaches to concurrency

- STM
- `Erlang's` message-passing $\approx$ Akka's "actors"
- CSP / `Go` language
- π-calculus, **join calculus** (academic so far)

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...

... but were afraid to ask

Most descriptions of JC use the "message/channel" metaphor...

"Chemistry"	JC terminology	JoinRun
molecule	message on channel	`a(123)` // side effect
injector	channel (port) name	`val a : JA[Int]`
blocking injector	blocking channel	`val q : JS[Int]`
reaction	process	`run { case a(x) + ... }`
injecting a molecule	sending a message	`a(123)` // side effect
join definition	join definition	`join(r1, r2, ...)`

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