

# Declarative distributed concurrency in Scala

Sergei Winitzki

Scale by the Bay 2018

November 16, 2018

# Talk summary

How I learned to forget semaphores and to love concurrency

Chymyst = an implementation of the Chemical Machine (CM) paradigm

- CM  $\approx$  Actors made purely functional and auto-parallelized
- Intuitions about why CM works better than other concurrency models
  - ▶ Comparison with related work: ING Baker, BPMN (workflow)
- New extension for distributed programming: DCM
- Code examples

Not in this talk: academic theory

- Petri nets,  $\pi$ -calculus, join calculus, joinads, mobile agent calculus...
- DCM formulated within a theory of distributed programming?

# Concurrent & parallel programming: How we cope

*Imperative* concurrency & parallelism is difficult to reason about:

- low-level API: callbacks, threads, semaphores, mutex locks
- hard to reason about mutable state and running processes
- hard to test – non-deterministic runtime behavior!
  - ▶ race conditions, deadlocks, livelocks

Known declarative approaches to avoid these problems:

Kind of concurrency	Typeclass	Scala implementation
synchronous parallelism	applicative functor	Spark, <code>.par.map()</code>
asynchronous streaming DAG	monadic functor	<code>Future</code> , <code>async/await</code> , RxJava, Akka Streams
unrestricted streaming	recursive monad+	Flink, fs2, ZIO
unrestricted concurrency	?	Akka, Chymyst

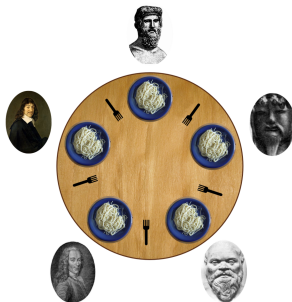
For distributed computing: challenges remain

- coordination and consensus, persistence and fault tolerance
- cluster configuration and discovery
  - ▶ distributed coordination as a service: Apache ZooKeeper, etcd

# “Dining philosophers”

The paradigmatic example of concurrency, parallelism and resource contention

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



Problem: simulate the process, avoiding deadlock and starvation

Solutions in various programming languages: see [Rosetta Code](#)

- Can this be implemented via (effectful) streams? (I think not.)
- The Chemical Machine code is purely declarative

# What is the Chemical Machine

The Chemical Machine paradigm:

- A *declarative language* for concurrent and parallel computations
  - ▶ largely unknown and unused by the software engineering community
  - ▶ Chymyst – an **open-source library & embedded DSL** for Scala
  - ▶ presented in my SBTB talks in 2016 and 2017
- Implement anything in 10 lines of code
- Derive programs quickly from specifications

# Chemical Machine vs. Amazon's AWS Lambda

How AWS $\lambda$  works:

- wait for an event that signals arrival of input data
- run a computation whenever input data becomes available
- the computation is automatically parallelized, data-driven
- writing the output data will create a new event

Modify the AWS $\lambda$  execution model by adding new requirements:

- a Lambda should be able to wait for several *unrelated* events
- several Lambdas may contend *atomically* on shared input events

With these new requirements, AWS $\lambda$  becomes “AWS $\pi$ ” – a model of “unrestricted concurrency”

- (Implementation on AWS will be tricky)

# The Chemical Machine vs. the Actor model

Modify the Actor execution model by adding new requirements:

- when messages arrive, actors are auto-created, maybe *in parallel*
- actors may wait atomically for messages in *several* different mailboxes

It follows from these requirements that...

- Auto-created actor instances are *stateless* and invisible to user
- User code defines *mailboxes* and *computations* that consume messages
- Repeated messages may be consumed in parallel
- Messages are sent to mailboxes, not to specific actor instances:

<pre>// Akka val a: ActorRef = ... receive(x) =&gt;... val b: ActorRef = ... receive(y) =&gt;... a ! 100 b ! 1;   b ! 2;   b ! 3</pre>	<pre>// Chymyst ... go { case a(x) =&gt; ... } ... go { case b(y) + c(z) =&gt; ... } a(100) b(1); b(2); b(3); c("hello");</pre>
--	---

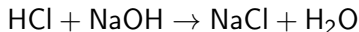
- All data resides on messages in mailboxes, is consumed automatically
- Mailboxes and computations are *values*, can be sent on messages

Any Actor program can be straightforwardly translated into CM

# Understanding the CM via the chemical metaphor

From real to abstract chemistry

Real chemistry is “asynchronous”, “concurrent”, and “distributed”

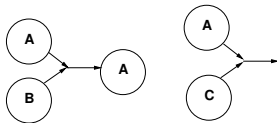


Want to run *computations* similarly to how chemical reactions run!

Begin by formulating the execution model of “abstract chemistry”:

- Abstract “molecules” float around in a “chemical reaction site”
- Certain sorts of molecules may combine to start a “reaction”:

Abstract chemical laws:



- Program code defines molecules  $a$ ,  $b$ ,  $c$ , ... and chemical laws
- At initial time, the code emits some molecules into the site
- The runtime system evolves the molecules *in parallel*
  - ▶ A simulator for this is easy to implement in any language

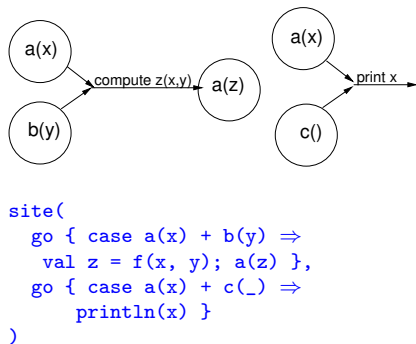


# Chemical Machine in a nutshell

From abstract chemistry to computation

Translating the chemical metaphor into a model of computation:

- Each molecule carries a **value** (“concurrent data”)
- Each reaction computes new values from its input values
- Some molecules with new values may be emitted back into the reaction site



When a reaction starts: input molecules disappear, new values are computed, output molecules are emitted

Reactions are *functions* from input values to output values

- Need to learn how to “think in molecules”

# Chemical Machine vs. Actor model

- reaction  $\approx$  function body for an (auto-started) actor
- emitted molecule with value  $\approx$  message with value, in a mailbox
- molecule emitters  $\approx$  mailbox references

Programming with actors:

- user code creates and manages explicit actor instances
- actors typically hold mutable state and/or mutate “behavior”
  - ▶ reasoning is about running processes *and* the data sent on messages

Programming with the Chemical Machine:

- processes auto-start when the needed input molecules are available
- many reactions may start at once, with automatic parallelism
  - ▶ user code does not manipulate references to processes
    - ★ no state, no supervision, no lifecycle, no “dead letters”, no routers
  - ▶ reasoning is only about the *data currently available* on molecules
    - ★ no reasoning about running processes having state

Chymyst code is typically 2x – 3x shorter than equivalent Akka code

## Example: throttling

Throttle emitting molecules `s(x)` with minimum allowed delay of `delta` ms

```
def throttle[X](s: M[X], delta: Long): M[X] = {  
  val r = m[X]  
  val enable = m[Unit]  
  // This molecule is confined to the local scope.  
  site(  
    go { case r(x) + enable(_) =>  
          s(x)  
          Thread.sleep(delta)  
          enable()  
        }  
  )  
  enable() // Enable emitting 's' initially.  
  r // Outside scope will be able to emit 'r'.  
}
```

- No threads/semaphores/locks, no mutable state
- External code may emit `r(x)` at will, and `s(x)` is then throttled

Implementations in Akka, in Monix, and ZIO: > 50 LOC each

# Example: map/reduce

## A simple map/reduce implementation:

```
val c = m[A] // Initial values have type 'A'.
val d = m[(Int, B)] // 'B' is a commutative monoid.
val res = m[B] // Final result of type 'B'.
val fetch = b[Unit, B] // Blocking emitter.
site(
  // 'map'
  go { case c(x) => d((1, long_computation(x))) },
  // 'reduce'
  go { case d((n1, b1)) + d((n2, b2)) =>
    val (newN, newB) = (n1 + n2, b1 |+| b2)
    if (newN == total) res(newB) else d((newN, newB))
  },
  go { case fetch(_, reply) + res(b) => reply(b) }
)
(1 to 100).foreach(x => c(x))
fetch() // Blocking call will return the final result.
```

Compare with the [Akka implementation here](#) (100+ LOC)

# Example: parallel merge-sort

Chymyst code: `MergeSortSpec.scala`

```
val mergesort = m[(Array[T], M[Array[T]])]
site(
  go { case mergesort((arr, sortedResult)) =>
    if (arr.length <= 1) sortedResult(arr)
    else {
      val sorted1 = m[Array[T]]
      val sorted2 = m[Array[T]]
      site(
        go { case sorted1(x) + sorted2(y) =>
          sortedResult(arrayMerge(x,y))
        }
      )
      val (part1, part2) = arr.splitAt(arr.length/2)
      // Emit lower-level mergesort molecules:
      mergesort(part1, sorted1) + mergesort(part2, sorted2)
    }
  })
```

Implementation in Akka: 30 LOC for the same functionality

# Example: Dining philosophers

## Five Dining Philosophers

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

```
// ... definitions of emitters, think(), eat() omitted for brevity
site (
  go { case t1(_) => think(1); h1() },
  go { case t2(_) => think(2); h2() },
  go { case t3(_) => think(3); h3() },
  go { case t4(_) => think(4); h4() },
  go { case t5(_) => think(5); h5() },

  go { case h1(_) + f12(_) + f51(_) => eat(1); t1() + f12() + f51() },
  go { case h2(_) + f23(_) + f12(_) => eat(2); t2() + f23() + f12() },
  go { case h3(_) + f34(_) + f23(_) => eat(3); t3() + f34() + f23() },
  go { case h4(_) + f45(_) + f34(_) => eat(4); t4() + f45() + f34() },
  go { case h5(_) + f51(_) + f45(_) => eat(5); t5() + f51() + f45() }
)
t1() + t2() + t3() + t4() + t5()
f12() + f23() + f34() + f45() + f51()
```

Source code: [DiningPhilosophers.scala](#)

For more examples, see the [code repository](#) (first-of, barriers, rendezvous, critical sections, readers/writers, Game of Life, elevators, etc.)

# Reasoning about code in the Chemical Machine paradigm

Reasoning about concurrent data:

- Emit molecule with value  $\approx$  lift data into the “concurrent world”
- Define reaction  $\approx$  lift a function into the “concurrent world”
- Reaction site  $\approx$  container for concurrent functions and data
- Reaction consumes molecules  $\approx$  function consumes input values
- Reaction emits molecules  $\approx$  function returns result values

Reasoning about code:

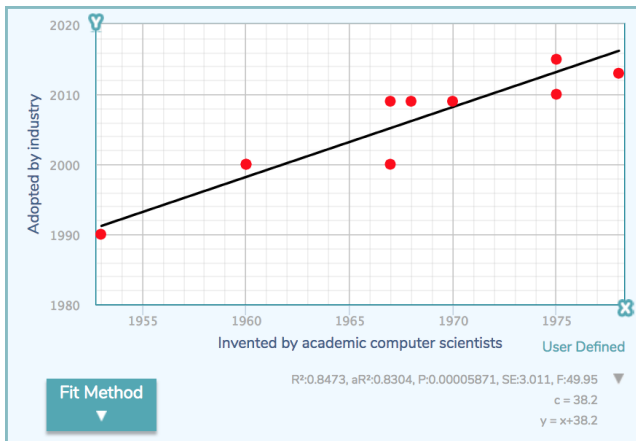
- What data do we need to handle concurrently? (Put it on molecules.)
- What computations consume this data? (Define reactions.)

Guarantees:

- Molecule emitters and reactions are immutable values in local scopes
- Reaction sites are immutable once activated; can refactor to libraries
- Multiple input molecules are consumed atomically by reactions

# Chemical Machine paradigm to become mainstream in 2033

- The Chemical Machine paradigm was **invented in 1995**
- The gap from academic invention to industry adoption is 38.2 years (declarative math, map/reduce, continuations, OOP, CSP, Actors, constraint programming, DAG dataflow,  $\lambda$ -functions, Hindley-Milner)





# Current features of Chymyst

- Blocking molecules with timeouts and back-signalling
- Automatic pipelining of molecules (ordered mailboxes)
- “Static” molecules with read-only access (similar to Akka “agents”)
- Compile-time and early run-time DSL error reporting
- Logging, debugging, unit-testing facilities
- Thread pools with thread priority control

## Related frameworks: Petri nets

Workflow management: an approach based on **Petri nets**

- **ING Baker** – a DSL for workflow management
- Process modeling and control (“elevator system” etc.)
- Business process management (BPM) systems

Chymyst implements a rich version of Petri nets:

- Transitions admit arbitrary guard conditions and error recovery
- Transitions carry values, reactions are values, can be nested
- Nondeterministic, asynchronous, parallel execution

A Petri net model is straightforwardly translated into a CM program

# Distributed Chemical Machine

Run concurrent code on a cluster with no code changes

- Some molecules are declared as “distributed”, of type `DM[T]`
- No other new language constructions are necessary!
  - ▶ early prototype in progress, as extension of Chymyst

## A simple implementation of map/reduce in DCM: 15 LOC

```
implicit val cluster = ClusterConfig(???)
val c = dm[Int] ; val d = dm[Int] // distributed
val res = m[(Int, List[Int])] // local
val fetch = b[Unit, List[Int]]
site(
  go { case c(x) => d(x * 2) }, // ‘map’ on cluster,
  // ‘reduce’ on the driver node only
  go { case res((n, list)) + d(x) => res((n-1, s::list)) },
  // fetch results
  go { case fetch(_, reply) + res((0, list)) => reply(list) }
)
if (isDriver) { // ‘true’ only on the driver node
  Seq(1, 2, 3).foreach(x => c(x))
  res((3, Nil)) ; fetch() // Returns the result.
}
```

Comparison: Akka implementation of distributed map/reduce (400+ LOC)

# Distributed cache in 10 lines of code

- Mutable `Map[String, String]` with operations: `put`, `get`, `delete`

```
implicit val cluster = ClusterConfig(???)
val data = dm[mutable.Map[String, String]]
val put = dm[(String, String)]
val get = dm[(String, M[Option[String]])]
val delete = dm[String]
site(
  go { case data(dict) + put((k, v)) => data(dict.updated(k, v)) },
  go { case data(dict) + get((k, r)) => data(dict); r(dict.get(k)) },
  go { case data(dict) + delete(k) => dict.remove(k); data(dict) }
)
if (isDriver) data(mutable.Map[String, String]())
```

- Comparison: Distributed cache in 100 lines of Akka

# Reasoning in the Distributed Chemical Machine

Distributed computing is made declarative

- Determine which data needs to be distributed and/or concurrent
- Determine which computations will need to consume that data
- Emit initial molecules and let the DCM run

Pure peer-to-peer architecture

- Distributed molecules may be consumed by *any* DCM peer
- All DCM peers operate in the same way (no master/worker)
- All DCM peers need to define the same distributed reaction sites
  - ▶ To designate a DCM peer as a “driver”, use config files

Examples (see documentation)

- Broadcast (DCM peers see it exactly once upon connecting)
- Distributed peer-to-peer chat

# Chemical Machine: implementation details

- Each reaction site has a scheduler thread and a worker thread pool
- Each molecule is “bound” to a unique reaction site
- Each emitted molecule is stored in a multi-set at its reaction site
- Each emitted molecule triggers a search for possible reactions
  - ▶ Reaction search proceeds concurrently for different reaction sites
- Reactions are scheduled on the worker thread pool
  - ▶ The thread pool can be configured per-reaction or per-site
- Scala macros are used for static analysis and optimizations
  - ▶ Automatically pipelined molecules
  - ▶ Simplify and analyze Boolean conditions
- Error analysis is also performed at early run time
  - ▶ Reaction site with errors remain inactive

# Distributed Chemical Machine: implementation details

- Each distributed molecule (DM) is bound to a unique reaction site
- Emitted DM data goes into the ZK instance
- Each DCM peer listens to ZK messages and checks for its DMs
  - ▶ Once a DM is found, its data is downloaded and deserialized
- On a DCM peer, each DM is identified with a unique local RS
  - ▶ Downloaded molecules are emitted into the local RS to run reactions
  - ▶ All DCM peers must run identical reaction code for DMs
- Each DCM peer acquires a distributed lock on its DMs
  - ▶ Lock is released once reaction scheduling is complete
- If a node goes down or network fails, molecules will be *unconsumed*
  - ▶ Another DCM peer will pick up these molecules later

# Conclusions and outlook

- Chemical Machine = declarative, purely functional concurrency
  - ▶ Similar to “Actors”, but easier to use and “more purely functional”
  - ▶ Significantly shorter code, easier to reason about
- An open-source Scala implementation: **Chymyst**
  - ▶ Static DSL code analysis (with Scala macros)
  - ▶ Industry-strength features (thread priority control, pipelining, fault tolerance, unit testing and debugging APIs)
  - ▶ Extensive documentation: **tutorial book**
- Promising applications:
  - ▶ Workflow management
  - ▶ Distributed peer-to-peer systems
  - ▶ Process modeling, GUIs, BPM
- Distributed Chemical Machine in the works