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 and demos

Not in this talk: academic theory

- Petri nets, π-calculus, join calculus, joinads, mobile agent calculus...
- DCM formulated within some 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	**Formal structure**	**Scala implementation**
synchronous parallelism	applicative functor	Spark, `.par.map()`
asynchronous streaming DAG	monadic functor	`Future`, `async`/`await`, `RxJava`, Akka Streams
unrestricted streaming	recursive monad+	Flink, fs2, ZIO
unrestricted concurrency	?	Akka, `Chymyst`

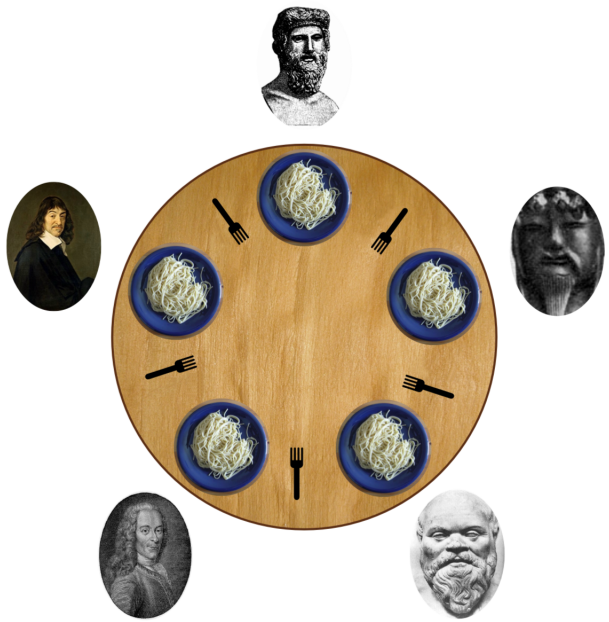
For distributed computing: challenges remain

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

"Dining philosophers"

The paradigmatic problem 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

Chemical Machine vs. AWS Lambda

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

AWS Lambda

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

Modify the AWS Lambda model by adding new requirements:

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

With these new requirements, AWS Lambda becomes a purely functional *unrestricted concurrency* model

The Chemical Machine vs. the Actor model

Modify the Actor 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:

```
// Akka
val a: ActorRef = ... receive(x) ⇒...
val b: ActorRef = ... receive(y) ⇒...
a ! 100
b ! 1;   b ! 2;   b ! 3
```

```
// Chymyst
... go { case a(x) ⇒ ... }
... go { case b(y) + c(z) ⇒ ... }
a(100)
b(1);  b(2);  b(3); c("hello");
```

- 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

The chemical metaphor

From real to abstract chemistry

Real chemistry:

$$\mathrm{HCl} + \mathrm{NaOH} \rightarrow \mathrm{NaCl} + \mathrm{H_2O}$$

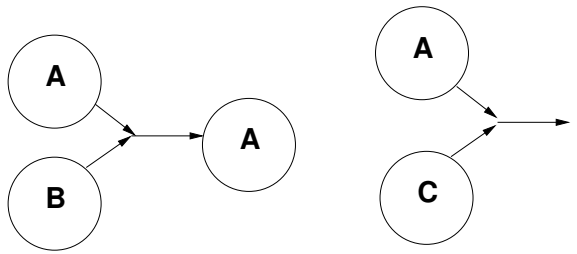
Abstract chemistry:

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

Abstract chemical laws:

`a + b` $\rightarrow$ `a`

`a + c` $\rightarrow \emptyset$



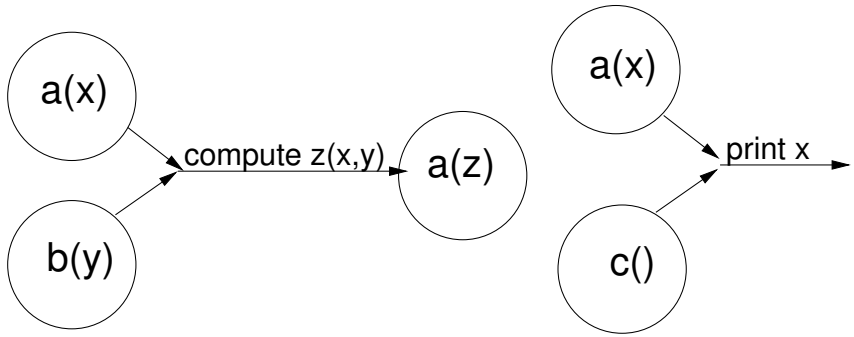
- Program code defines molecules `a`, `b`, `c`, ... and chemical laws
- At initial time, the code emits some molecules into the site
- The runtime system evolves the molecules *concurrently* and *in parallel*

Chemical Machine in a nutshell

"Better concurrency through chemistry"

Translating the chemical metaphor into practice:

- 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




```
site(
  go { case a(x) + b(y) ⇒
   val z = f(x, y); a(z) },
  go { case a(x) + c(_) ⇒
      println(x) }
)
```

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

Reactions are functions from input values to output values

Chemical Machine vs. Actor model

- reaction $\approx$ template 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, automatically parallel
  - user code does not manipulate references to processes
    - ★ no state, no supervision, no lifecycle to manage
  - reasoning is *only* about the *data* currently available on molecules

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

Example: throttling

Throttle emitting a molecule `s(x)` with min. delay of `delta` ms

```
def throttle[X](s: M[X], delta: Long): M[X] = {
 val r = m[X]
 val allow = m[Unit]
 site(
  go { case r(x) + allow(_) ⇒
         s(x)
         Thread.sleep(delta)
         allow()
     }
 )
 allow() // Beginning of time; we allow requests.
 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: about 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 returns the final result.
```

Compare with the Akka example (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: 25 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, 8 queens, etc.)

Reasoning about code in the Chemical Machine paradigm

Chemical metaphor vs. concurrent data metaphor:

- 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 items
- 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
- Molecules are consumed atomically by reactions

Current features of Chymyst

- Blocking molecules with timeouts
- Automatic pipelining of molecules
- "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 with 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

Distributed Chemical Machine

Run concurrent code on a cluster with no code changes

- Same as CM except some molecules are declared as "distributed"
- No other code changes necessary!
  - early prototype in progress

A simple implementation of map/reduce in DCM:

```
implicit val cluster = ClusterConfig(???)
val c = dm[Int] ; val d = dm[Int] // distributed
val res = m[(Int, List[Int])]
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)

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

Peer-to-peer architecture

- 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
- Distributed molecules may be consumed by *any* DCM peer

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 and is stored there
- Each emitted molecule triggers a search for possible reactions
- Reactions are scheduled on the worker thread pool
- Each reaction may emit further molecules
- Scala macros are used for static analysis and optimizations
  - Automatically pipelined molecules
  - Simplify and analyze Boolean conditions
- Some error analysis is performed at early run time

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 DMs
- On a DCM peer, each DM is identified with a unique local RS
  - In this way, downloaded molecules can be emitted locally
  - All DCM peers must run identical reaction code
- Each DCM peer acquires a distributed lock on its DMs
- 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"
  - Short, declarative code implementing barriers, rendezvous, etc.
- 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 and draft paper
- Promising applications:
  - Workflow management
  - Distributed peer-to-peer systems
  - Process modeling, GUIs, BPM
- Example code for this talk: github.com/Chymyst/jc-talk-2017-examples