Declarative Concurrent Programming with Join Calculus in Scala

Sergei Winitzki

Lambda-Conf 2020 Global Edition

August 11, 2020

Chemical Machine: a new hope

...and some new hype

In this talk:

- "join calculus" as a "**Chemical Machine**", without academic jargon
  - "λ-calculus" — a small (but complete) programming language
  - "join calculus" — a small (but complete) language for concurrency
    - adds 2 features ("async mailbox" and "async function") to λ-calculus
    - in the chemical metaphor: "molecule" and "reaction"
- `Chymyst` – open-source implementation of Chemical Machine (Scala)
- examples of concurrent programs in `Chymyst`
  - implement anything in 10-15 lines of code
- comparisons with Actor Model, "AWS Lambda", and Petri nets
- an extension for distributed programming: DCM

Not in this talk: ~~academic theory~~

- ~~Petri net theory, π-calculus, join calculus, joinads, formal semantics~~
- DCM defined ~~within some formalism for distributed programming~~

The chemical metaphor I. “Abstract” chemistry

Real chemistry is asynchronous, concurrent, and parallel:

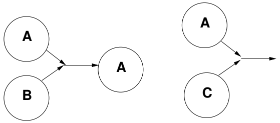
$$\mathrm{HCl} + \mathrm{NaOH} \to \mathrm{NaCl} + \mathrm{H_2O}$$

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 kinds of molecules may combine to start a “reaction”:

“abstract” chemical laws:

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



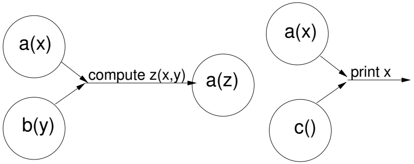
- Program code defines molecules `a`, `b`, `c`, ... and chemical reactions
- At initial time, the code emits some molecules into the site
- The runtime system runs reactions *concurrently and in parallel*
  - A chemical simulation engine is easy to implement in any language

The chemical metaphor II. 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



```
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

- Need to learn how to “think in molecules and reactions”

Molecule *emitters* are values of type `M[A]`:

```
val count = m[Int]; val inc = m[Unit]
```

Molecule *instances* are emitted by calling the emitter's `apply` method:

```
count(0); inc(); inc(); inc(); // Side-effect: emit molecules.
```

Reactions are values of type `Reaction`:

```
val r: Reaction = go { case count(x) + inc(_) ⇒ count(x + 1) }
```

- A reaction may be any partial function with a single `case` clause
- All molecules (e.g., `count`, `inc`) must be already defined

A group of reactions are "activated" within a "reaction site":

```
site(r1, r2, r3) // Side-effect: activate reactions r1, r2, r3.
```

- A molecule may be emitted only after activating a reaction site where some reactions consume that molecule as input
- A site *must* include all reactions that consume a given molecule

A Chemical Machine program declares some molecules, reactions, and reaction sites, and then can emit some initial molecules

- The Chymyst runtime will run the simulation

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] // Emitter 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 (green or not), no semaphores, no locks, no mutable state
- External code may emit `r(x)` at will, and `s(x)` is then throttled
- External code cannot emit `enable()` because of local scope

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

Example: throttling test

Assume a molecule `p` is already bound in our reaction(s)
Create a "throttled version" `tp` that has the same effect as `p` except that `tp` is always emitted at a limited rate
To test, emit `tp` several times quickly:

```
val p = m[Int]
site(
  go { case p(x) ⇒ println(s"Got $x at time ${LocalDateTime.now}") }
)
val pt = throttle(p, 1000L) // 1 second minimum delay.
pt(10); pt(20); pt(30);
scala> Got 10 at time 2020-08-10T15:33:05.921
Got 20 at time 2020-08-10T15:33:06.914
Got 30 at time 2020-08-10T15:33:07.920
```

Example: async cancellation

Prevent emission of molecule `x` after a molecule `cancel` is emitted

```
def cancellation[A](x: M[A]): (M[A], M[Unit]) = {
  val cx = m[A] // Cancellable version of ‘x‘.
  val cancel = m[Unit]; val enable = m[Unit]
  site(
    go { case cx(a) + enable(_) ⇒ x(a); enable() },
    go { case enable(_) + cancel(_) ⇒ },
  )
  enable()      // Enable emitting ‘x‘ initially.
  (cx, cancel) // Outside scope will be able to emit these.
}
```

- Emitting `cancel()` many times has no further effect
- External code cannot emit `enable()` because of local scope

Libraries such as Monix or ZIO provide a cancellation facility
The Chemical Machine code is easy to modify (e.g., add uncancel)

Example: async cancellation test

Assume a molecule `x` is already bound in our reaction(s)
Create a "cancellable version" `cx` that has the same effect as `x` but can be (eventually) canceled by emitting a molecule `cancel`:

```
val x = m[Int]
site( go { case x(a) ⇒ println(s"Got $a") } )
val (cx, cancel) = cancellation(x)
scala> cx(1); cx(2); cx(3); cancel(); cx(4); cx(5);
Got 1
Got 2
```

Asynchronous cancellation cannot guarantee order of events!

Example: async racing

For two molecules `p` and `q`, determine which one was emitted first and report its value on a given molecule (`reply`):

```
def racing[A, B](reply: M[Either[A, B]]): (M[A], M[B]) = {
  val p = m[A]; val q = m[B]
  val enable = m[Unit]
  site(
    go { case p(a) + enable(_) ⇒ reply(Left(a))  },
    go { case q(b) + enable(_) ⇒ reply(Right(b)) },
  )
  enable() // Initially, we only have one copy of ‘enable‘.
  (p, q)   // Outside scope will be able to emit these.
}
```

- External code will emit `p(...)` and `q(...)`, will get `reply(...)` emitted
- External code cannot emit `enable()` because of local scope
- Order of async molecules is not guaranteed

Libraries such as Monix or ZIO provide a racing facility
The Chemical Machine code is easy to modify (e.g., add cleanup)

Example: async racing test

Assume a molecule `result` is already bound in our reaction(s)

```
val result = m[Either[Int, String]]
site(
  { go { case result(Left(x))  ⇒ ... },
  { go { case result(Right(y)) ⇒ ... },
)
val (p, q) = racing[Int, String](result)
// Two parallel computations can now emit ‘p‘ and ‘q‘ asynchronously.
// First computation:
... p(123)    // Non-blocking.
// Second computation:
... q("abc") // Non-blocking.
// Eventually, a ‘result(...)‘ molecule will be emitted.
```

Additional feature: blocking molecules

A blocking molecule has type `B[A, R]` and receives a reply value of type `R`:

```
val fetch = b[Int, String] // Blocking molecule's emitter.
val data = m[Int] // A non-blocking molecule emitter.
site(
  go { case data(x) + fetch(y, reply) ⇒ reply(s"got $x and $y") }
) // ‘reply‘ is a ReplyEmitter[String] defined within reaction scope.
data(123)
val x = fetch(456) // Blocking call, will set x = "got 123 and 456".
```

Blocking emitters are a convenience, do not increase expressive power

Example: synchronous rendez-vous

Create two blocking molecules `p(x: X)` and `q(y: Y)` to implement synchronous rendez-vous that exchanges `x` and `y`:

```
def syncRendezvous[X, Y]: (B[X, Y], B[Y, X]) = {
  val p = b[X, Y]; val q = b[Y, X]
  site( go { case p(x, rx) + q(y, ry) ⇒ rx(y); ry(x) } )
  (p, q)
}
// To test: Create the molecules for a rendez-vous.
val (p, q) = syncRendezvous[Int, String]
// Two parallel computations can exchange data. First computation:
... val myResponse: String = p(123) // Blocking until response.
// Second computation:
... val myResponse: Int = q("abc")  // Blocking until response.
```

Example: racing with blocking molecules

Given two blocking molecules `p(x: X)` and `q(y: Y)`, create a new blocking molecule `r(...)` to determine which of `p` and `q` returns first and return the corresponding value of type `Either[X, Y]`:

```
def racing[X,Y](p: B[Unit,X], q: B[Unit,Y]): B[Unit, Either[X,Y]] = {
  val get = b[Unit, Either[X, Y]]
  val res = b[Unit, Either[X, Y]]
  val left = m[Unit]; val right = m[Unit]
  val done = m[Either[X, Y]]
  site(
    go { case res(_, reply) ⇒ left(); right(); reply(get()) },
    go { case left(_)  ⇒ done(Left(p()))  },
    go { case right(_) ⇒ done(Right(q())) },
    go { case get(_, r) + done(x) ⇒ r(x) },
  )
  res // Outside scope will be able to emit this.
}
```

- External code calls `r(...)`, which will call both `p(...)` and `q(...)`

Libraries such as Monix or ZIO provide a racing facility

Example: parallel 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” to perform a long computation:
  go { case c(x) ⇒ d((1, long_computation(x))) },
  // “reduce” to aggregate the results:
  go { case d((n1, b1)) + d((n2, b2)) ⇒
   val (newN, newB) = (n1 + n2, b1 |+| b2) // Aggregation.
   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

Reactions can be recursive

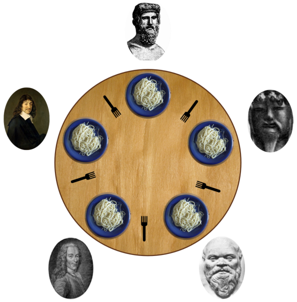
```
val mergesort = m[(Array[T], M[Array[T]])]
site(
  go { case mergesort((arr, sortedResult)) ⇒
    if (arr.length <= 1) sortedResult(arr)
      else {
        val sorted1 = m[Array[T]]
        val sorted2 = m[Array[T]]
        site(   // Define a lower-level reaction site.
          go { case sorted1(x)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

Complete `Chymyst` code: MergeSortSpec.scala
Implementation in Akka: 30 LOC for the same functionality

Dining philosophers I. Declarative vs. non-declarative code

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

- The Chemical Machine code is purely declarative

Dining philosophers II. Implementation in Chymyst

Five Dining Philosophers implemented in 15 lines of code

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 (barriers, critical sections, readers/writers, Conway's "Game of Life", elevator control, etc.)

Reasoning about code in the Chemical Machine paradigm

Reasoning about concurrent data:

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

Reasoning about code:

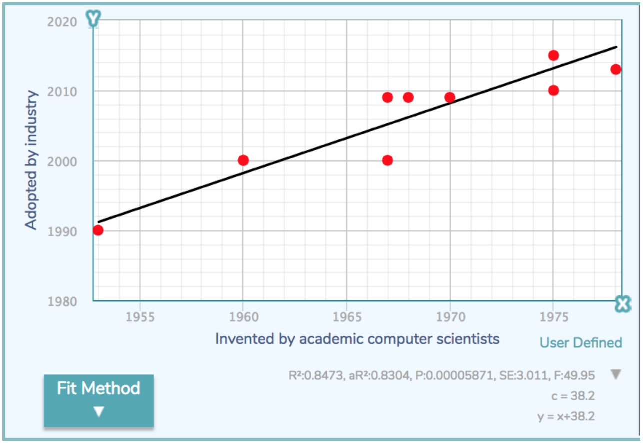
- What data do we need to handle concurrently? (Put it on molecules.)
- What computations consume this data? (Define as reactions.)

Guarantees:

- Molecule emitters and reactions are immutable values in local scopes
- Reaction sites are immutable; input molecule sets are defined statically
- Multiple input molecules are consumed atomically by reactions

Chemical Machine paradigm to become mainstream in 2033

- The gap from academic invention to industry adoption is 38.2 years (infix math, continuations, λ-functions, OOP, CSP, map/reduce, Actor Model, constraint programming, DAG dataflow, Hindley-Milner types)



R²:0.8473, aR²:0.8304, P:0.00005871, SE:3.011, F:49.95

c = 38.2

y = x+38.2

- The Chemical Machine paradigm was invented in 1995

Current features of `Chymyst`

Experience with Chemical Machine programming is limited

A tutorial book goes through many examples

Some features (already implemented) that promise to be useful:

- Blocking molecules with timeouts and timeout back-signalling
- Automatic pipelining of molecules (ordered mailboxes)
- Thread pools, thread priority control, graceful shutdown
- Facility to increase parallelism when using blocking code
- Errors in DSL are reported at compile-time or early run-time
  - `Chymyst` uses Scala macros – but only to inspect code
  - Static analysis enables optimization and error reporting
- "Static" molecules with read-only access (similar to Akka "agents")
  - Forgetting to emit a molecule is #1 programmer error
- Logging, metrics monitoring, debugging, unit-testing facilities

Possible extensions:

- Persistence?
- Automatic parallelism adjustment?

Comparison to Petri nets

Workflow management: an approach based on Petri nets

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

`Chymyst` implements a feature-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

Any Petri net model is straightforwardly translated into a CM program

Chemical Machine vs. Amazon AWS Lambda

How AWSλ 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 AWSλ 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, AWSλ becomes "AWSπ" – a model of unrestricted concurrency

- (Implementation on AWS could be tricky)

Chemical Machine vs. the Actor model. I

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:

```
// Akka                                      // Chymyst
val a: ActorRef = ... receive(x) ⇒...      ... go { case a(x) ⇒ ... }
val b: ActorRef = ... receive(y) ⇒...      ... go { case b(y) + c(z) ⇒ ... }
a ! 100                                     a(100)
b ! 1;   b ! 2;   b ! 3                     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

Chemical Machine vs. Actor model. II

- reaction ≈ function body for an (auto-started) actor
- emitted molecule with value ≈ message with value, in a mailbox
- molecule emitters ≈ 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 internal state

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

Distributed Chemical Machine

Run concurrent code on a cluster with no code changes

- Declare some molecules as "distributed", of type `DM[T]`
- No other new language constructions are necessary!
  - early prototype in progress, as extension of `Chymyst`

Distributed map/reduce in 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 LOC

- 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

Distributed peer-to-peer chat in 15 LOC

- Register user names in chat room
- Fetch list of users
- Send and receive text messages

```
implicit val cluster = ClusterConfig(???)
val users = dm[List[DM[String]]] // List of users' message emitters.
val carrier = dm[DM[String]] // Carries this node's message emitter.
val fetch = b[Unit, List[DM[String]]]
site(go { case users(es) + carrier(e) ⇒ users(e :: es) }
, go { case users(es) + fetch(_, r) ⇒ users(es); r(es) } )

val peerName = ??? // Read from config on node.
val sender = new DM[String](peerName) // Assign unique molecule name.
site(go { case sender(x) ⇒ println(s"Peer $peerName reads $x")})

carrier(sender)
if (isDriver) users(Nil)
// Fetch list of users and send a message to Sergei if present.
fetch()
  .find(_.name == "Sergei")
  .foreach(sender ⇒ sender("hello"))
```

- Comparison: Distributed chat 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
  - Non-DCM code may differ between peers
  - Code or configuration could designate DCM peer as a "driver" or have different roles

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
  - Enough power to replace threads, semaphors, atomic vars, etc.
  - Similar to Actor Model, 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
  - Distributed Chemical Machine – work in progress
- Promising applications:
  - Workflow management, BPM
  - Asynchronous GUIs
  - Distributed peer-to-peer systems