Declarative Concurrent Programming with Chymyst

Sergei Winitzki

Scale by the Bay 2017

November 18, 2017

What is "Chymyst"?

Chymyst = a Scala implementation of the "chemical machine" paradigm

- Reflexive Chemical Abstract Machine [Fournet & Gonthier 1996]
  - known as "join calculus" in the academic world
- it is a *declarative language* for concurrent & parallel computations
  - largely unknown and unused by the software engineering community
  - available as an open-source library & DSL

Concurrent & parallel programming is hard

Imperative concurrency is difficult to reason about:

- callbacks, threads, semaphores, mutex locks
- errors with shared mutable state
- testing is hard – non-deterministic runtime behavior!
  - race conditions, deadlocks, livelocks

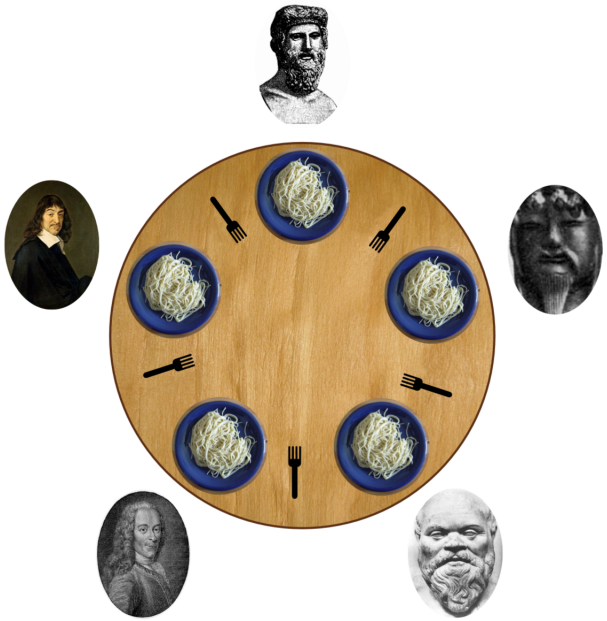
We can avoid this if we use declarative approaches:

- synchronous parallelism (parallel collections, Spark)
- asynchronous parallelism (`Future`, `async`/`await`)
- asynchronous streaming (Akka Streaming)
- Actors (Akka)
- chemical machine (Chymyst)

"Dining philosophers"

The paradigmatic example of concurrency

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: Rosetta Code

Talk overview

How I learned to forget semaphores and to love concurrency

In this talk:

- Introduction to join calculus and "chemical" style of concurrency
- Chymyst -- a new Scala EDSL for join calculus
- Join calculus as an evolution of the Actor model
- Examples and demos

Not in this talk: Other approaches to declarative concurrency

- π-calculus, PICT language (academic so far)
- `Erlang's` message-passing $\approx$ Akka's "Actors"
- CSP / `Go` language
- STM (Haskell, Scala)

Join Calculus: A New Hope

...and some new hype

Join calculus is ...

- ...a declarative language for general-purpose concurrency
- "What if Actors were auto-started, type-safe, and multiple-dispatch"
- No threads/semaphores/locks, no shared mutable state
- Concurrency is automatic and *data-driven*, not command-driven
  - Easier to reason about than any other concurrency formalism!

Metaphors for join calculus:

- "concurrent functions compute with concurrent data"
- "chemical soup with molecules and reactions"

Concurrent data and concurrent functions

Intuitions leading to join calculus

Message to auto-started actor $\approx$ concurrent function call on data item
What would it mean to make ordinary functions concurrent?

- Several functions should be able to run at once
- No shared state: Concurrent processes work on different data

This will be implemented if:

- Data items and functions are stored in a special "**site**"
- Each data item is labeled for specific concurrent function(s)
- Concurrent functions consume data from the site
- Computed results are emitted back to the site

Operational semantics:

- Concurrent functions auto-start whenever input data is available
- Different instances of a conc. function consume separate data items

Join Calculus: The Genesis

From real to abstract chemistry

Real chemistry:

$$HCl + NaOH \rightarrow NaCl + H_2O$$

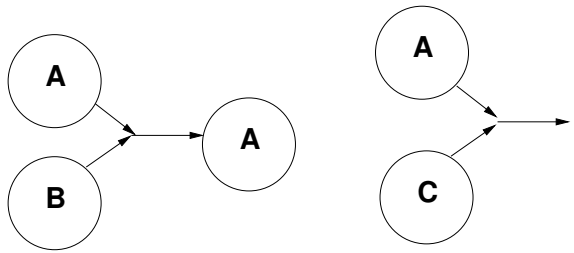
Abstract chemistry:

- Chemical "soup" contains instances of abstract "molecules"
- Combine certain sorts of molecules to start a "reaction":

Abstract chemical laws:

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

`a + c` $\rightarrow \emptyset$



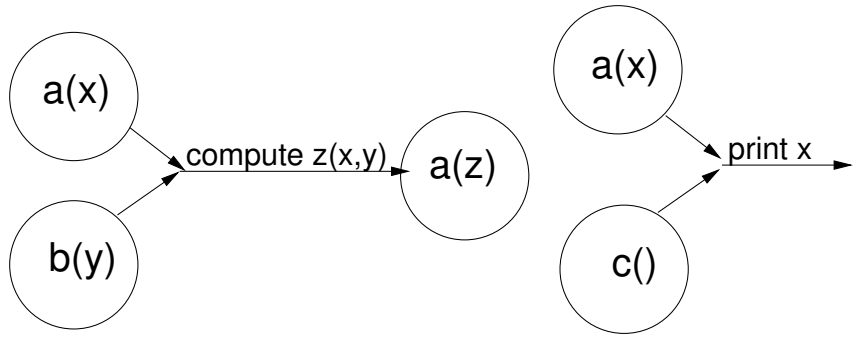
- Program code defines molecules `a`, `b`, `c`, ... and chemical laws
- At initial time, the code emits some molecules into the "soup"
- The runtime system evolves the soup *concurrently* and *in parallel*

Join Calculus in a Nutshell

"Better concurrency through chemistry"

Translating the chemical metaphor into practice:

- Each molecule carries a **value** ("concurrent data")
- Each reaction computes a "molecule-set-valued" expression from input values
- Resulting molecules are emitted back into the soup
- Whenever input molecules are available, reactions start concurrently and in parallel




```
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, expression is computed, output molecules are emitted

Programs = chemical laws + initial molecules

First example: concurrent counter

We would like to decrement and increment concurrently

Chemical laws:

- counter(n) + decr() $\Rightarrow$ counter(n − 1)
- counter(n) + incr() $\Rightarrow$ counter(n + 1)

Initial molecule instances:

- counter(0)

"Data stays on the molecules"

We may emit decr() and incr() concurrently

Chymyst: basic features

Molecule emitters and reaction definitions in the Scala DSL

Define **molecule emitters**:

```
val counter = m[Int]

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

Declare some **reactions** by pattern match on the molecule values:

```
val r0 = go { case counter(n) + decr(_) ⇒ counter(n-1) }

val r1 = go { case counter(n) + incr(_) ⇒ counter(n+1) }
```

Activate a **reaction site**:

```
site(r0, r1)
```

- For brevity, define reactions inline within reaction sites

Chymyst: basic usage

Operational semantics

Emit some molecules:

```
counter(10) // non-blocking side-effect
incr() // ditto; we will have counter(11) later
incr() // we will have counter(12) later
```

- Calling `counter(10)` returns `Unit` and emits a molecule as a side-effect
- This could be the state of the chemical soup at some point in time:
  - counter(10) + incr() + incr()

Concurrent data and concurrent functions

Chemical metaphor vs. concurrent terms metaphor

- Reaction consumes molecules $\approx$ function consumes input values
- Reaction emits molecules $\approx$ function returns result values
- 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

Chymyst: more features

Blocking vs. non-blocking molecules

Non-blocking molecules:

- emitter *does not wait* until a reaction starts with the new molecule

Blocking molecules:

- emitter will block until a reaction starts and emits a "reply value"
- molecule implicitly carries a pseudo-emitter for "reply"
- when the "reply" is emitted, its value will be returned to caller
- Example:

```
val f = b[Int, String] // create blocking emitter
go { f(x, reply) + c(y) ⇒ reply(s"${x + y}") }
c(100) // non-blocking
val result: String = f(200) // blocking call, will get "300"
```

Chymyst: examples I

Counter with blocking access

Blocking molecule `getN` reads the value `x` in `counter(x)`:

```
val getN = b[Unit, Int]
// revise the join definition, appending this reaction:
...  val r2 = go { case counter(x) + getN(_, reply) ⇒ reply(x) }
site(r0, r1, r2)
// Emit non-blocking molecules as before...
// Now emit the blocking molecule:
val x = getN() // blocking fetch, returns Int
```

Source code: CounterSpec.scala

Definitions in local scopes

`Chymyst` = functional programming + join calculus

New molecules, reactions, and sites can be defined in *local scopes*

Emitters (`read: M[Int]`) can be molecule values too!

```
def makeCounter(init: Int): (M[Unit], M[M[Int]]) = {
  val c = m[Int]
  val decr = m[Unit]
  val get = m[M[Int]]
  site(
   go { case c(x) + get(read) ⇒ c(x); read(x) },
   go { case c(x) + decr(_) ⇒ c(x - 1) }
  )
  c(init) // emit initial molecule
  (decr, get) // return emitters to the outside scope
}
// usage:
val (decr, get) = makeCounter(100)
val result = m[Int]
get(result) // non-blocking fetch
```

Chymyst: examples II

Options, Futures, and Map/Reduce

Implement `Future` with blocking "`get`":

```
go { case get(_, reply) ⇒ val x = f(); reply(x) }
```

Implement Map/Reduce:

```
go { case c(x) ⇒ d(x * 2) } // "map"
go { case res(list) + d(s) ⇒ res(s :: list) } // "reduce"
go { case get(_, reply) + res(list) ⇒ reply(list) }
res(Nil)
Seq(1,2,3).foreach(x ⇒ c(x))
get() // this returned Seq(4,6,2) in one test
```

Source code: FutureSpec.scala

Chymyst: examples III

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

From Actors to Join Calculus

"Chemical actors" are actors with new requirements:

1. chemical actors are auto-started when messages arrive
2. chemical actors may wait atomically for a *set* of different messages
3. messages carry statically typed values ("typed Akka")

It follows from these requirements that...

- User code declares *computations* and not actor instances
- Auto-created actor instances must be stateless
- Message emitters are *specific to data*, not to actor instances:

```
// Akka
val a: ActorRef = ...
val b: ActorRef = ...
a ! 100
b ! 1;   b ! 2;   b ! 3
```

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

- Multiple messages are automatically parallelized
- Blocking molecules ≈ blocking-send: `actorRef ? 1`

Join Calculus vs. Actor model

David vs. Goliath?

- reaction $\approx$ actor
- emitted molecule $\approx$ message sent to actor

Programming with actors:

- user code creates and manages explicit actor instances
- actors will consume one message at a time
- actors typically hold mutable state or mutate "behavior"

Programming with reactions:

- processes auto-start when the needed input molecules are available
- many reactions can start at once, automatically concurrent
- many molecules can be consumed at once, atomically
- immutable, stateless, and type-safe
- all reactions are defined statically, but locally scoped

Chymyst: examples IV

Concurrent merge-sort: chemistry pseudocode

The `mergesort` molecule starts a "chain reaction":

- 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) )

      // Emit lower-level mergesort molecules:
        mergesort(part1, sorted1) + mergesort(part2, sorted2)
```

Chymyst: examples IV

Concurrent merge-sort: Chymyst code

```
    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)
        }

})
```

Source code: MergeSortSpec.scala

Pipelined molecules: An automatic optimization

In join calculus: channels with *ordered* mailboxes

Reaction scheduler in *Chymyst*:

- Examines all present molecule instances and runs the next reaction
- Is it sufficient to examine *only one* molecule instance?
  - This depends on the specific chemical program
- If so, molecule instances can be held in an ordered queue ("pipelined")
- `Chymyst` *automatically* makes some molecules pipelined

In a given chemical program, can we pipeline the molecule `a(x)`?

- Consider the predicate `f(x, y, z,...)` for starting a reaction, e.g.:
  ```
  f(x, y, b) = HAVE(b(y)) && x == 0 && y > x
  ```
- The predicate `f(...)` must be *factorizable* into a conjuction:
  ```
  f(x, y, z, ...)  = p(x) && q(y, z,...)
  ```
  - I have a proof that this optimization preserves semantics

Everything you need to know about join calculus...

... but the Wikipedia page confused you, so you were afraid to ask

Academic descriptions of JC use the "message/channel" terminology

"Chemistry"	JC terminology	Chymyst code
molecule	message on channel	`a(123)` // side effect
emitter	channel name	`val a: M[Int]`
blocking emitter	blocking channel	`val q: B[Unit, Int]`
reaction	process	`go { case a(x) + ... }`
emitting a molecule	sending a message	`a(123)` // side effect
reaction site	join definition	`site(r1, r2, ...)`

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]
  - JEScala [G. Salvaneschi 2014]
- Chymyst -- a new JC implementation in Scala (this talk)
  - Better syntax, more checks of code sanity
  - (Some) automatic fault tolerance
  - Thread pool and thread priority control
  - Event monitoring and unit testing APIs

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.
- A new open-source Scala implementation: Chymyst
  - Full-featured implementation of join calculus
  - Industry-strength features (thread priority control, pipelining, fault tolerance, unit testing and debugging APIs)
  - Extensive documentation: tutorial book and draft paper
- On the future roadmap:
  - Thread fusion for better performance
  - Full continuation-passing transformation to nonblocking code
  - Automatic backpressure ("reaction temperature")
  - Automatic distributed runtime ("distributed soup")
- Example code for this talk: github.com/Chymyst/jc-talk-2017-examples