

# Declarative Concurrent Programming with Join Calculus

Sergei Winitzki

Scala Bay

October 16, 2017

# What is “Join Calculus”?

“Join calculus” is...

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

# Concurrent & parallel programming is hard

Imperative concurrency is difficult to reason about:

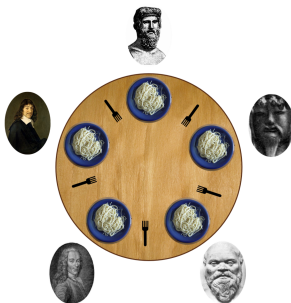
- callbacks, threads, semaphores, mutexes, shared mutable state...
- testing is hard – non-deterministic runtime behavior!
  - ▶ race conditions, deadlocks, livelocks

We try to *avoid* concurrency whenever possible

# Dining philosophers

The paradigmatic example 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

Solutions: [Rosetta Code](#)

# How I learned to forget deadlocks and to love concurrency

In this talk:

- Introduction to the “**join calculus**” style of concurrency
- **Chymyst** -- a new Scala implementation
- Join calculus as an evolution of the Actor model
- Examples and demos

Not in this talk: other approaches to declarative concurrency

- $\pi$ -calculus, PICT language (academic so far)
- Erlang’s message-passing  $\approx$  Akka’s “Actors”
- CSP / Go language
- STM (Haskell)

# Join Calculus: A 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 automatic and *data-driven* (not command-driven)
- Easier to use than anything I’ve seen so far!

Metaphors for join calculus:

- “concurrent functions computing with concurrent data”
- “chemical soup with molecules and reactions”

# Join Calculus: The Genesis

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

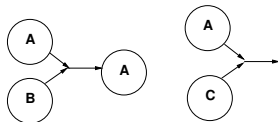
Abstract chemistry:

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

“Chemical laws”:

$a + b \rightarrow a$

$a + c \rightarrow \emptyset$



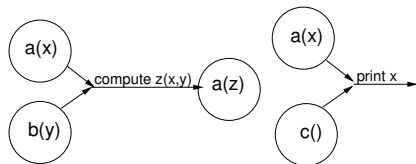
- Define molecules  $a$ ,  $b$ ,  $c$ , ... and arbitrary chemical laws
- Emit some molecules into the “soup”
- The runtime system evolves the soup *concurrently* and *in parallel*

# 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 = compute_z(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



# First example: concurrent counter

Chemical laws:

- $\text{counter}(n) + \text{decr}() \Rightarrow \text{counter}(n - 1)$
- $\text{counter}(n) + \text{incr}() \Rightarrow \text{counter}(n + 1)$

Initial molecules:

- $\text{counter}(0)$

“Data stays on the molecules”

We may emit  $\text{decr}()$  and  $\text{incr}()$  concurrently

# Concurrent data and concurrent functions

Towards a more declarative view

Molecule with value  $\approx$  data lifted into the concurrent world

Reaction  $\approx$  function lifted into the concurrent world

- $\text{counter}(n) + \text{add}(x) \Rightarrow \text{counter}(n + x)$
- $\text{counter}(n) + \text{incr}() \Rightarrow \text{counter}(n + 1)$
- $\text{counter}(n) + \text{get}(\text{result}) \Rightarrow \text{counter}(n) + \text{result}(n)$

State of the chemical soup:

- $\text{counter}(0) + \text{incr}() + \text{incr}() + \text{add}(10)$

Reaction consumes molecules  $\approx$  function consumes input values

Reaction emits molecules  $\approx$  function returns result values

# Chymyst: basic features

## Molecule emitters, reaction definitions

Define **molecule emitters**:

```
val counter = m[Int]
```

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

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

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

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

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

Activate a “**reaction site**” and emit some molecules:

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

```
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

# 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 sends a “reply value”
- molecule implicitly carries a pseudo-emitter for “reply”
- when the “reply” is emitted, the value will be returned to caller
- Example:

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

# 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 call, returns Int
```

Source code: [CounterSpec.scala](#)

# Chymyst: examples II

## Options, Futures, and Map/Reduce

Implement Future with blocking poll (“`get`”):

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

Implement Map/Reduce:

```
go { case c(x) ⇒ d(x * 2) }
```

```
go { case res(list) + d(s) ⇒ res(s :: list) }
```

```
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

```
// ... some definitions 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 and stopped when messages arrive
- 2 chemical actors may wait atomically for a *set* of different messages
- 3 messages carry statically typed values

It follows from these requirements that...

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

```
// Actors                                site( // Chymyst
val a: ActorRef = ...                    go { case a(x) => ... },
val b: ActorRef = ...                    go { case b(x) => ... })
a ! 100                                  a(100)
b ! 1;  b ! 2;  b ! 3                    b(1); b(2); b(3)
```

- Multiple messages are automatically parallelized
- Blocking molecules  $\approx$  blocking-send: `actorRef ? 1`



# Join Calculus vs. Actor model

David vs. Goliath?

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

Actors:

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

Reactions:

- autostart when the 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

# Chymyst: examples IV

## Concurrent merge-sort: chemistry pseudocode

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) )
```

```
// 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](#)

# Everything you need to know about JC...

... but were afraid to ask

Most descriptions of JC use the “message/channel” metaphor...

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

# 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](http://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

- Join Calculus = declarative, purely functional concurrency
- Similar to “Actors”, but easier and “more purely functional”
- Very little known, and very little used in practice
- A new Scala implementation, **Chymyst**
- Documentation: **tutorial book** and **draft paper**
- Example code for this talk:  
[github.com/Chymyst/jc-talk-2017-examples](https://github.com/Chymyst/jc-talk-2017-examples)