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:

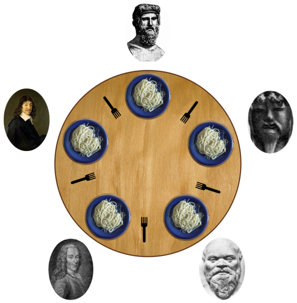
- callbacks, threads, semaphores, mutexes, shared mutable state...
- testing is hard – non-deterministic runtime behavior!
    - race conditions, deadlocks, livelocks

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

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

Join Calculus: The 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]

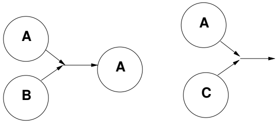
Abstract chemistry:

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

"Chemical laws":

`a + b` $\to$ `a`

`a + c` $\to$ $\emptyset$



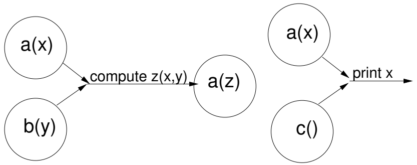
- Define molecules `a`, `b`, `c`, ... and arbitrary chemical laws
- Emit some molecules into the "soup"
- The runtime system evolves the soup *concurrently* and *in parallel*

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:

- counter(n) + decr() ⇒ counter(n - 1)
- counter(n) + incr() ⇒ counter(n + 1)

Initial molecules:

- counter(0)

"Data stays on the molecules"

We may emit decr() and incr() concurrently

Concurrent data and concurrent functions

Towards a more declarative view

Molecule with value ≈ data lifted into the concurrent world

Reaction ≈ function lifted into the concurrent world

- counter(n) + add(x) ⇒ counter(n + x)
- counter(n) + incr() ⇒ counter(n + 1)
- counter(n) + get(result) ⇒ counter(n) + result(n)

State of the chemical soup:

- counter(0) + incr() + incr() + add(10)

Reaction consumes molecules ≈ function consumes input values

Reaction emits molecules ≈ function returns result values

Using `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

Using 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)
  ```

Using 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
```

Using Chymyst: examples II

Options, Futures, and Map/Reduce

Future with blocking poll (“get”):

```
fut( (f,x) ) => finished( f(x) )
get(_, r) + finished(fx) => r(fx)
```

Map/Reduce:

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

Using Chymyst: examples III

Five Dining Philosophers

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

```
// ... some declarations omitted for brevity
site (
  go{ case t1(_) => wait(); h1() },
  go{ case t2(_) => wait(); h2() },
  go{ case t3(_) => wait(); h3() },
  go{ case t4(_) => wait(); h4() },
  go{ case t5(_) => wait(); h5() },
  go{ case h1(_) + f12(_) + f51(_) => wait(); t1() + f12() + f51() },
  go{ case h2(_) + f23(_) + f12(_) => wait(); t2() + f23() + f12() },
  go{ case h3(_) + f34(_) + f23(_) => wait(); t3() + f34() + f23() },
  go{ case h4(_) + f45(_) + f34(_) => wait(); t4() + f45() + f34() },
  go{ case h5(_) + f51(_) + f45(_) => wait(); t5() + f51() + f45() }
)
t1() + t2() + t3() + t4() + t5()
f12() + f23() + f34() + f45() + f51()
```

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 ≈ blocking-send: `actorRef ?  1`

Join Calculus vs. Actor model
David vs. Goliath?

- reaction ≈ actor
- emitted molecule ≈ 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

Using 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)
```

Using 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)
    }
  })
```

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	`a(123)` // side effect
emitter	channel (port) 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

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