Introduction to Join Calculus

Sergei Winitzki

Scala Study Group

November 10, 2013

Motivation

Imperative concurrency is difficult:

- callbacks, semaphores, locks, threads, shared state
- testing??

Pure functional concurrency is better:

- futures = "async monads"
- Erlang's purely functional messaging; "actors" (Akka in Scala)

Join Calculus:

- Elegant, concise model of concurrent computation
- Join Calculus = "*more* purely functionally concurrent" actors
- Working implementation: **JoCaml** (jocaml.inria.fr)

A taste of OCaml

Common features to F#, Haskell, OCaml, SML, Scala:

- Expression-oriented programming:
  ```
  let s = (if 1=1 then "Hello, world!" else "Error") in print_string s
  ```
- Algebraic data types, parametric polymorphism:
  ```
  type 'a bTree = Leaf of 'a | Node of ('a bTree * 'a bTree)
  ```
- Immutable, scoped values, with statically inferred types:
  ```
  # let x = 3 in (let x = x+1 in x/2) * x;;
  - : int = 6
  ```
- Mutually recursive definitions:
  ```
  # let rec  isEven n = if n=0 then true else isOdd (n-1)
    and   isOdd n = if n=0 then false else isEven (n-1);;
  val isEven : int -> bool = <fun>
  val isOdd : int -> bool = <fun>
  # let result = List.map (fun x -> (isEven x, isOdd x)) [1; 2];;
  val result : (bool * bool) list = [ (false, true); (true, false) ]
  ```

Join Calculus in a nutshell

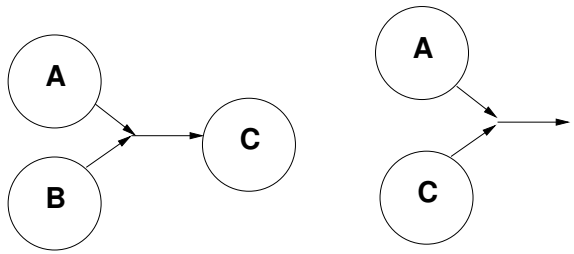
The Reflexive Chemical Abstract Machine (RChAM)

Abstract chemistry: "molecules" and "reactions"

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

```
jocaml>   def  a() & b() = c()
          and  a() & c() = 0;;
val a :  unit Join.chan = <abstr>
val b :  unit Join.chan = <abstr>

val c :  unit Join.chan = <abstr>
```



Using the "chemical machine":

- We define arbitrary "chemical laws" and "molecules": `a`, `b`, `c`, ...
- We inject some "molecules" into the soup: `spawn a() & a() & b()`
  - Note: `a() & a() & b()` is the syntax for "molecule-valued" expressions
- The runtime system evolves the soup *asynchronously*

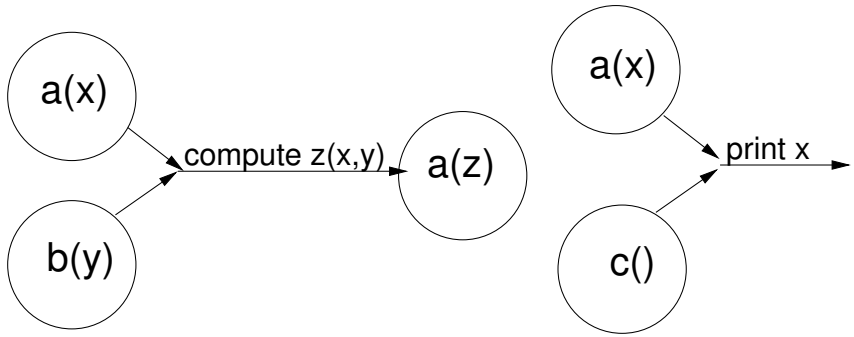
Join Calculus in a nutshell

Concurrent computations

Sequential computation = evaluating an expression
Concurrent computation = evaluating several expressions at once

Join Calculus organizes concurrent computations through "chemistry":

- Each molecule carries a **value**
- Each reaction computes a "molecule-valued" **expression**
- Results of computation are injected back into the soup
- Reactions start asynchronously after injecting initial molecules




```
def  a(x) & b(y) =
  let z = (big_function x y) in a(z)
and  a(x) & c() = (print_int x; 0);;
```

When reaction starts: input molecules disappear, expression is computed, output molecules are injected

Join Calculus in a nutshell

More features of JoCaml

- Mutual recursion:
  `def a(x) = a(x+1) & b(x+2) and b(x) & c(y) = a(x+y)`
- Pattern-matching on molecule's payload values:
  `def a(Some x) & b(y) = b(x+y) or a(None) & b(y) = b(y)`
- **Instant** molecules (same type as function calls):
  `def a(x) & f() = a(x+1) & reply x to f`
- **Local** molecules and reactions:
  `def c(n) = ( if n>0 then c(n-1) else 0 ) in  spawn c(10)`
- Injection as side-effect: `let x=3 in (spawn a(x); printf "%d\n" x)`
- **Molecule constructors** are defined as values and can be manipulated:

```
# def a(x) = Printf.printf "%d\n" x; 0;;

val a :  int Join.chan = <abstr>

# def b(m,y) = Printf.printf "injecting m(%d)\n" y; m(y);;

val b:  (int Join.chan * int) Join.chan = <abstr>
```

Join Calculus: Examples

Options, Futures, and Map/Reduce

Future with synchronous poll ("get"):

```
# def fut(f,x) = let res = f x in finished(res)
  and get() & finished(res) = reply res to get;;
val get :  unit -> '_a = <fun>
val finished :  '_a Join.chan = <abstr>
val fut :  (('a -> '_b) * 'a) Join.chan = <abstr>
```

Future with synchronous callback:

```
def fut(f,x,c) = let res = f x in ( c(res); finished(res) )
  and get() & finished(res) = reply res to get
```

Future with asynchronous callback:

```
def fut(f,x,m) = let res = f x in ( m(res) & finished(res) )
  and get() & finished(res) = reply res to get
```

- Exercise: implement a "future with cancellable callback"

Join Calculus: Examples

Options, Futures, and Map/Reduce

Asynchronous counter:

```
# def inc() & c(n) = c(n+1)
or get() & c(n) = reply n to get & c(n);;
val inc :  unit Join.chan = <abstr>
val get :  unit -> int = <fun>
val c :  int Join.chan = <abstr>
# spawn c(0) & inc() & inc() & inc();;
- :  unit = ()
# get();;
- :  int = 3
```

Map/Reduce:

```
def res(list) & c(s) = res (s::list) or get() & res(list) = reply list to get;;

spawn res([]);;

List.map (fun x-> spawn c(x*2)) [1; 2; 3];;
get();; (* this returned [4; 6; 2] in one test *)
```

- Exercise: implement a concurrent "fold" (e.g. sum of int list)

Join Calculus: Examples

Five Dining Philosophers

Philosophers `A, B, C, D, E`; forks `fAB, fBC, fCD, fDE, fEA`.

```
let report(message) = Printf.printf "%s\n" message;
                      Unix.sleep (Random.int 3600) ;;
def hA() & fEA() & fAB() = report("A is eating"); tA() & fEA() & fAB()
or  hB() & fAB() & fBC() = report("B is eating"); tB() & fAB() & fBC()
or  hC() & fBC() & fCD() = report("C is eating"); tC() & fBC() & fCD()
or  hD() & fCD() & fDE() = report("D is eating"); tD() & fCD() & fDE()
or  hE() & fDE() & fEA() = report("E is eating"); tE() & fDE() & fEA()
and tA() = report("A is thinking"); hA()
and tB() = report("B is thinking"); hB()
and tC() = report("C is thinking"); hC()
and tD() = report("D is thinking"); hD()
and tE() = report("E is thinking"); hE() ;;
spawn fAB() & fBC() & fCD() & fDE() & fEA()
     & tA() & tB() & tC() & tD() & tE() ;;
```

Limitations and restrictions of Join Calculus

Less is more!

- Reactions are defined **statically** and with **local scope**:
  - no molecules with computed names:
    `a(x) & molecule_named("b")(x) = (not allowed!)`
  - cannot dynamically add a new reaction to a previously defined molecule:

```
def a(x) & b(y) = ...  ;;
def b(y) & c(z) = ...  shadows the old definition of b()!
```

- No "guard conditions" for reactions:

```
def a(x) & b(y) & start_if (x==y) = ...  (not allowed!)
```

- No duplicated input values: `a(x) & b(x) = (not allowed!)`
- No duplicated input molecules: `a(x) & a(y) = (not allowed!)`
- No way to test dynamically for the presence/absence of a molecule!

```
def a(x) & b(y) = if have_molecules(c & d) then ...  else ...  (not allowed!)
```

Limitations and restrictions of Join Calculus

It seems they do not limit the expressive power!

What if we *need* a reaction with pairs of molecules?

```
a(x) & a(y) = a(x+y)
```

- Solution: use two "`or`"-coupled reactions with new molecules `a'` and `b`:

```
def a(x) & b() = a'(x) or a(x) & a'(y) = whatever(x,y)
```

- Make sure that one `b()` is injected together with each `a(x)`

Questions:

- Can we prevent the error of not injecting `b()`?
- Can we do a reaction with n molecules, where n is dynamic?
- Can we do "n dining philosophers"?

Local scope and recursion

Skeleton code for concurrent merge-sort

The mergesort molecule:

- receives the upper-level "`sorted_result`" molecule
- defines its own "`sorted`" molecule in *local scope*
- emits upper-level "`sorted_result`" when done

```
def mergesort(arr, sorted_result) =
  if Array.length arr <= 1 then sorted_result(arr)
  else
   let (part1, part2) = array_split arr
   in
   def sorted(x) & sorted(y) = sorted_result(array_merge x y)
   in
   mergesort(part1, sorted) & mergesort(part2, sorted)
```

Note: "`sorted(x) & sorted(y)`" is pseudocode, easy to rewrite.
See tutorial for complete working JoCaml code.

Comparison: Join Calculus vs. Actor model

Reaction $\approx$ actor; molecule $\approx$ message to actor.

Actors:

- receive messages asynchronously
- process one message at a time (one actor = one thread)
- must hold mutable state (e.g. for a thread-safe counter)
- explicitly create and configure other actors

Reactions:

- start when several molecules are available
- many reactions can start at once, automatically
- do not need mutable state
- all reactions are defined statically, but locally scoped
- simulate actors: `def message(m) & actor(state) = actor(compute_new state m)`

Implementation of Join Calculus

JC = a DSL + run-time library, or just DSL?

Implement Join Calculus using Actors (Akka)?

- Each reaction has 1 "monitor actor" and ≥ 1 "worker actors"
- Monitor receives messages for each "spawn", keeps track of molecules
- Monitor starts a worker actor when all molecules are present
- Monitors have to talk to competing monitors - "use up" molecules
  - but all competitions are statically defined!
- Monitors / workers need to be locally scoped!
- No globally shared state of the "soup" is needed!
- Discuss

Conclusions and outlook

- "Join Calculus" is concurrent programming in pure functional style
- Similar to "Actors", but more concurrent and "more pure"
- Very little known, and very little used in practice
- Existing literature is not suitable as introduction to practical programming
- My tutorial text is in progress (search Google for "tutorial on join calculus")