Concurrent ML: The One That Got Away

Michael Sperber

@sperbsen

active group

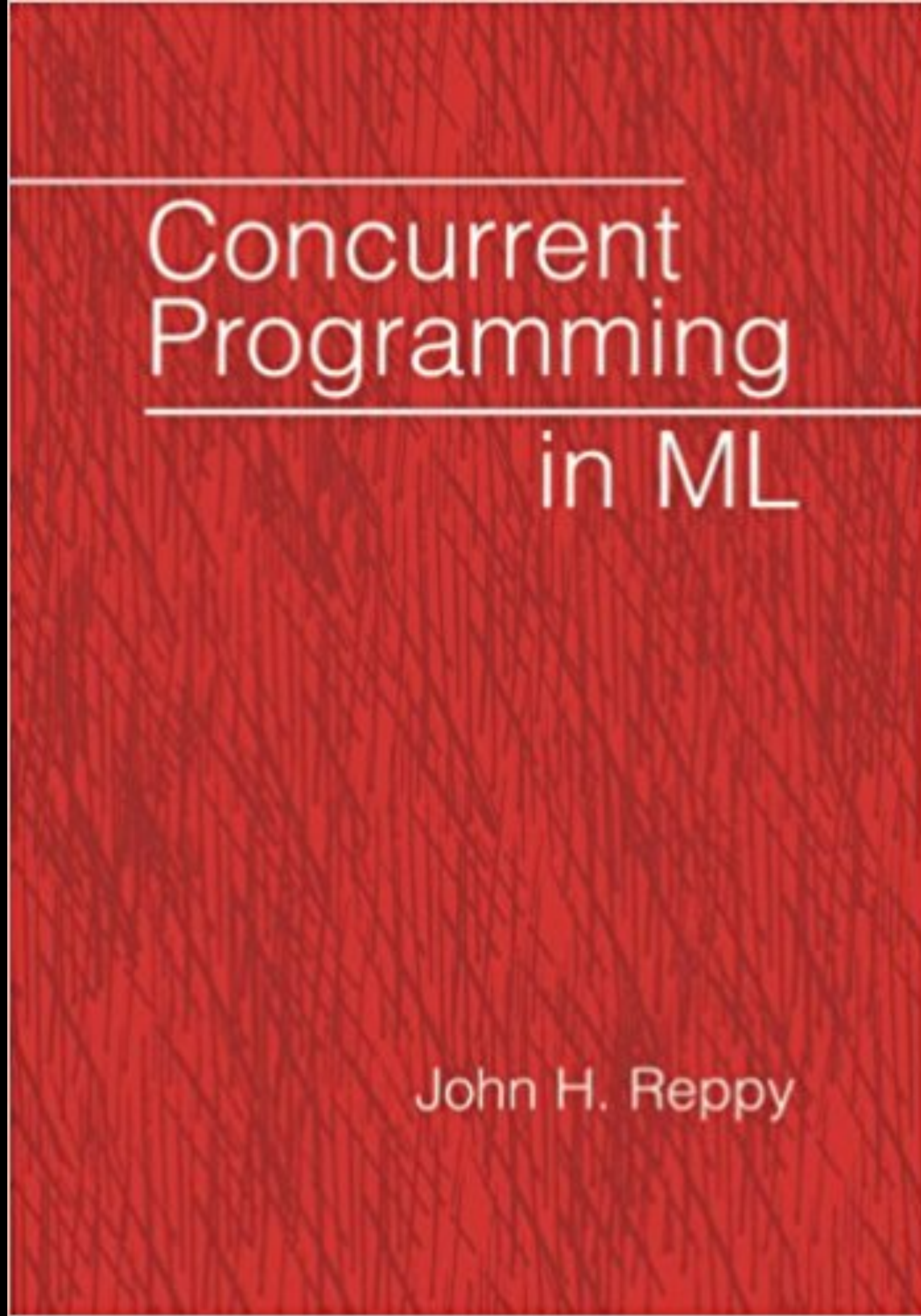
- software project development
- in many fields
- Scala, Clojure, Erlang, Haskell, F#, OCaml
- training, coaching
- co-organize BOB conference

www.active-group.de

funktionale-programmierung.de

Myself

- taught Concurrent Programming at U Tübingen in 2002
- implemented Concurrent ML for Scheme 48
- designed Concurrent ML for Star


Concurrent
Programming
in ML
John H. Reppy

F# Tutorial, CUFP 2011

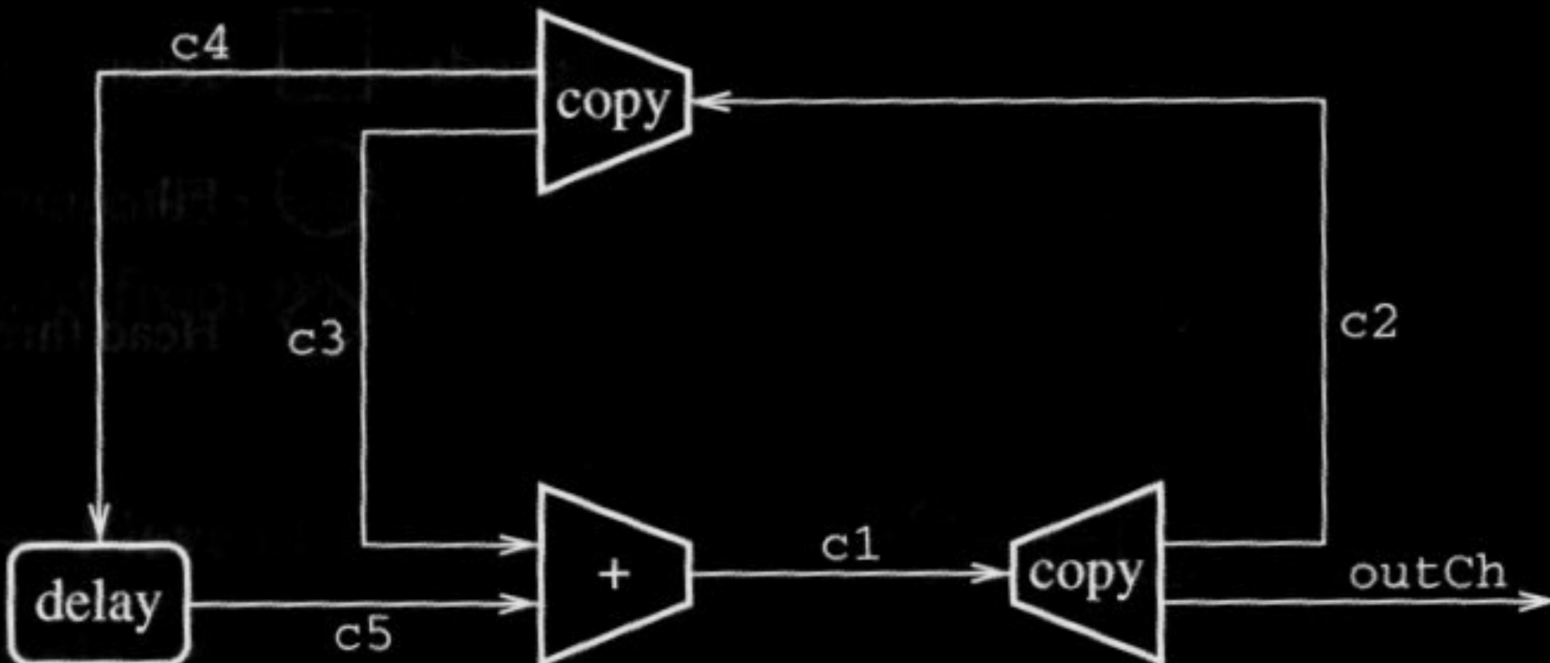
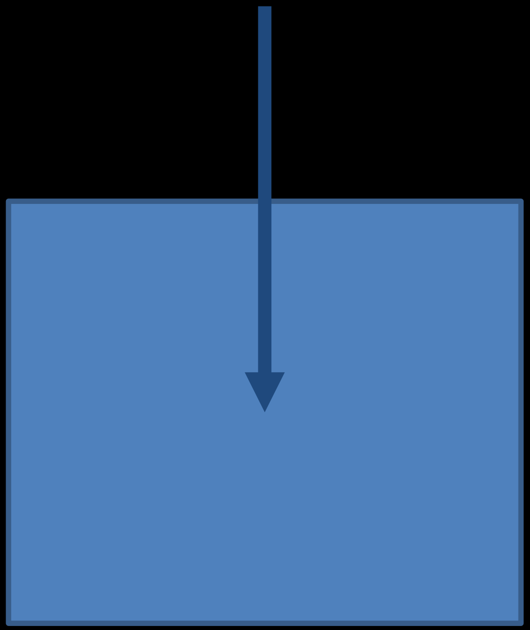


Figure 3.3: The Fibonacci stream network

John Reppy. Concurrent ML. Cambridge.

Actor Model



F#

```
/// split elements coming into inbox among the two sinks
 let split (inbox: Agent<SinkMessage<'T>>)
   (sink1: Agent<SinkMessage<'T>>) (sink2: Agent<SinkMessage<'T>>)
 : Async<unit> =
   let rec loop (sink1: Agent<SinkMessage<'T>>)
                (sink2: Agent<SinkMessage<'T>>) =
     async {
   let! msg = inbox.Receive ()
       match msg with
   | EndOfInput ->
         do sink1.Post EndOfInput
         do sink2.Post EndOfInput
    return ()
   | Value x ->
    do sink1.Post msg
         return! loop sink2 sink1
        | GetNext _ -> failwith "premature GetNext"
  }
   loop sink1 sink2
```

Abstractions Matter!

(Duh!)

Composition Matters!

Concurrent ML

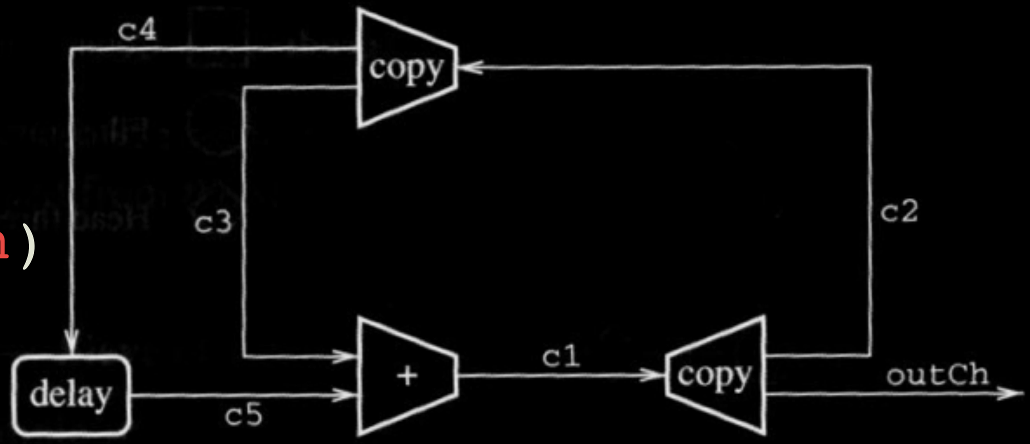
- SML/NJ
- Manticore
- Multi-MLton
- Scheme 48
- Racket
- Guile
- Star
- *Haskell

Things that are not CML

- Erlang
- Go
- Clojure core.async
- Akka streams

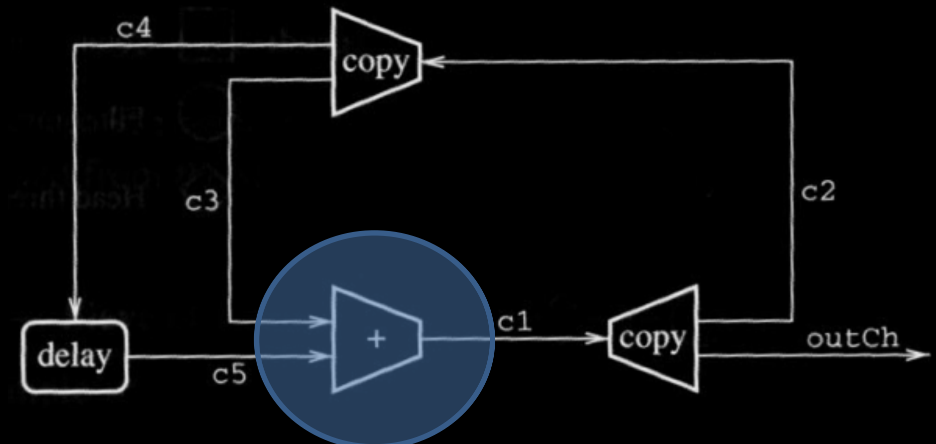
Fibonacci Network

```
(define (make-fibonacci-network)
  (let ((outch (make-channel))
        (c1 (make-channel))
        (c2 (make-channel))
        (c3 (make-channel))
        (c4 (make-channel))
        (c5 (make-channel)))
    (delay1 0 c4 c5)
    (copy c2 c3 c4)
    (add c3 c5 c1)
    (copy c1 c2 outch)
    (send c1 1)
    outch))
```



Process Networks: Add

```
(define (add inch1 inch2 outch)
  (forever
    #f
    (lambda (_)
      (send outch
        (+ (receive inch1)
           (receive inch2))))))
```

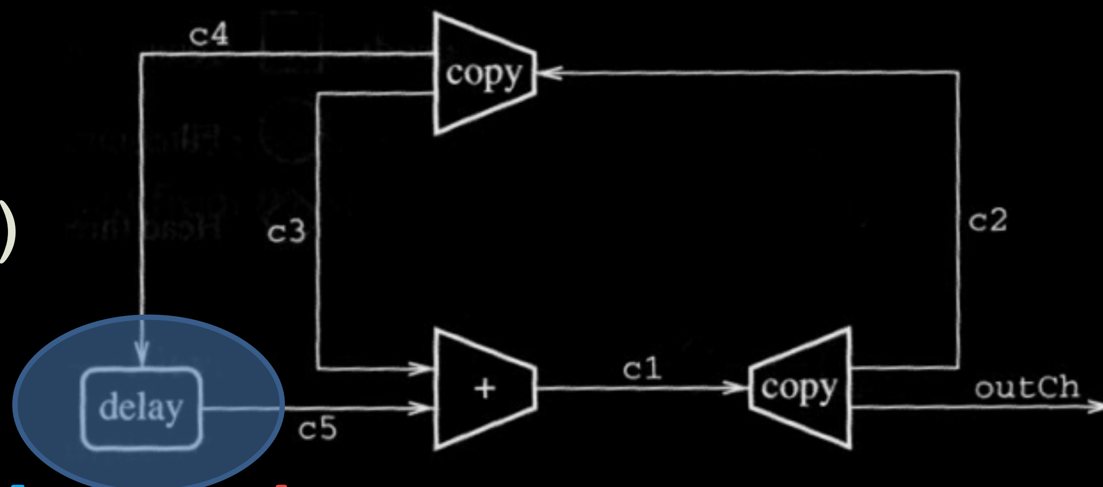


Engine

```scheme
(define (forever init f)
  (spawn
   (lambda ()
     (let loop ((s init))
       (loop (f s))))))
```

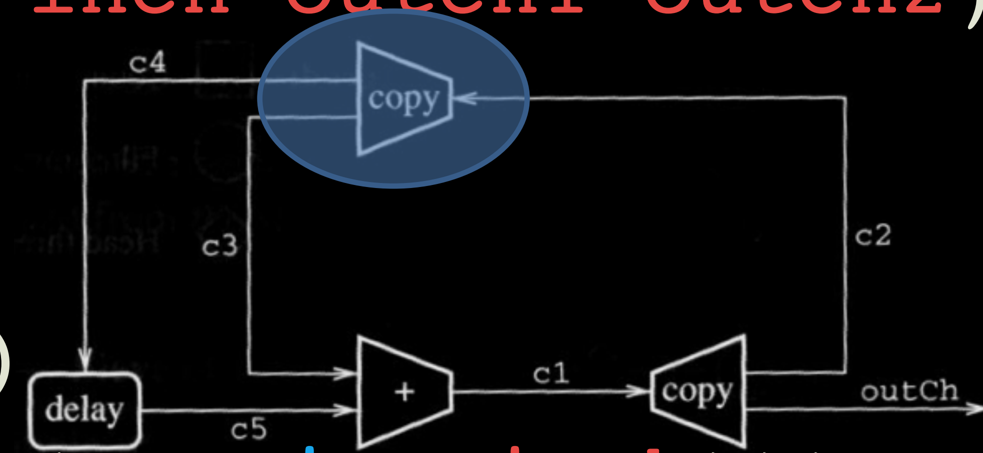
Delay

```
(define (delay1 init inch outch)
  (forever
   init
   (lambda (v)
     (if v
       (begin
         (send outch v)
         #f)
       (receive inch)))))
```



Copy

```
(define (copy inch outch1 outch2)
  (forever
   #f
   (lambda (_)
     (let ((v (receive inch)))
       (send outch1 v)
       (send outch2 v))))))
```



Straight-Up Channel Ops

```
(: send ((channel %a) %a -> void))

(: receive ((channel %a) -> %a))
```

Copy

```
(define (copy inch outch1 outch2)
  (forever
   #f
   (lambda (_)
     (let ((v (receive inch)))
1      (send outch1 v)
2      (send outch2 v))))))
```

Selective Communication

```
(: select (??????? ...) -> %a)
```

Clojure

```
(defn copy
  [inch outch1 outch2]
  (forever
   nil
   (fn [_]
     (go
       (let [v (<! inch)]
         (alt!
           [[outch1 v]] (>! outch2 v)
           [[outch2 v]] (>! outch1 v))))))))
```

Go

```
func copy(inch,
          outch1, outch2 chan interface{}) {
  for {
  select {
    v := <-inch
    select {
    case outch1 <- v:
      outch2 <- v
    case outch2 <- v:
      outch1 <- v
    }
  }
}
```

Erlang

```
copy1(Outp1, Outp2) ->
    receive
        V -> Outp1 ! V,
             Outp2 ! V
    end.

copy(Outp1, Outp2) ->
    forever([],
      fun(_) ->
        copy1(Outp1, Outp2)
      end).
```

F#

```
let copy(out1: Agent<'a>, out2: Agent<'a>)
      : Agent<'a> =
    Agent.Start (fun inbox ->
       forever((), fun () ->
        async {
             let! v = inbox.Receive ()
             out1.Post v
             out2.Post v
        }))
```

Selective Communication

```
(: select ((rv %a) ... -> %a))

(: send-rv ((channel %a) %a ->
                (rv void)))
(: receive-rv ((channel %a) ->
                   (rv %a)))
```

“Map”

```
(: wrap ((rv %a) (%a -> %b) ->
           (rv %b)))
```

Copy / Select

```
(define (copy inch outch1 outch2)
  (forever
   #f
   (lambda (_)
     (let ((v (receive inch)))
       (select (wrap (send-rv outch1 v)
                     (lambda (_)
                       (send outch2 v)))
                    (wrap (send-rv outch2 v)
                     (lambda (_)
                       (send outch1 v)))))))))
```

Add / Select

```
(define (add inch1 inch2 outch)
  (forever
   #f
   (lambda (_)
     (let ((p (select
               (wrap (receive-rv inch1)
                (lambda (a)
                  (cons a
                        (receive inch2))))
               (wrap (receive-rv inch2)
                (lambda (b)
                  (cons (receive inch1)
                        b))))))
       (send outch (+ (car p) (cdr p)))))))
```

@rtive group

Rendezvous Composition

```
(: choose ((rv %a) ... -> (rv %a)))

(: sync ((rv %a) -> %a))

(: select ((rv %a) ... -> %a))
```

Other Kinds of Rendezvous

- timeout
- async channels
- shared memory
- I/O
- ...

Roll yer own rendezvous

```
(define-record-type
    (swap really-make-swap swap?)
  (fields
   ; (channel (pair a (channel a)))
   (immutable channel
              swap-channel)))

(define (make-swap)
  (really-make-swap (make-channel)))
```

Swap Meet

```
(define (swap-rv swap message-out)
  (let ((channel (swap-channel swap)))
    (guard
     (lambda ()
      (let ((in-channel (make-channel)))
       (choose
         (wrap (receive-rv channel)
               (lambda (pair)
                 (let ((message-in (car pair))
                       (out-channel (cdr pair)))
                 (send out-channel message-out)
                message-in)))
         (wrap (send-rv channel
                        (cons message-out in-channel))
               (lambda (_)
                 (receive in-channel)))))))))
```

Guard

```
(: guard ((-> (rv %a)) -> (rv %a)))
```

Negative Acknowledgements

```
(: with-nack
   (((rv void) -> (rv %a)) ->
     (rv %a)))
```

Losing Interest

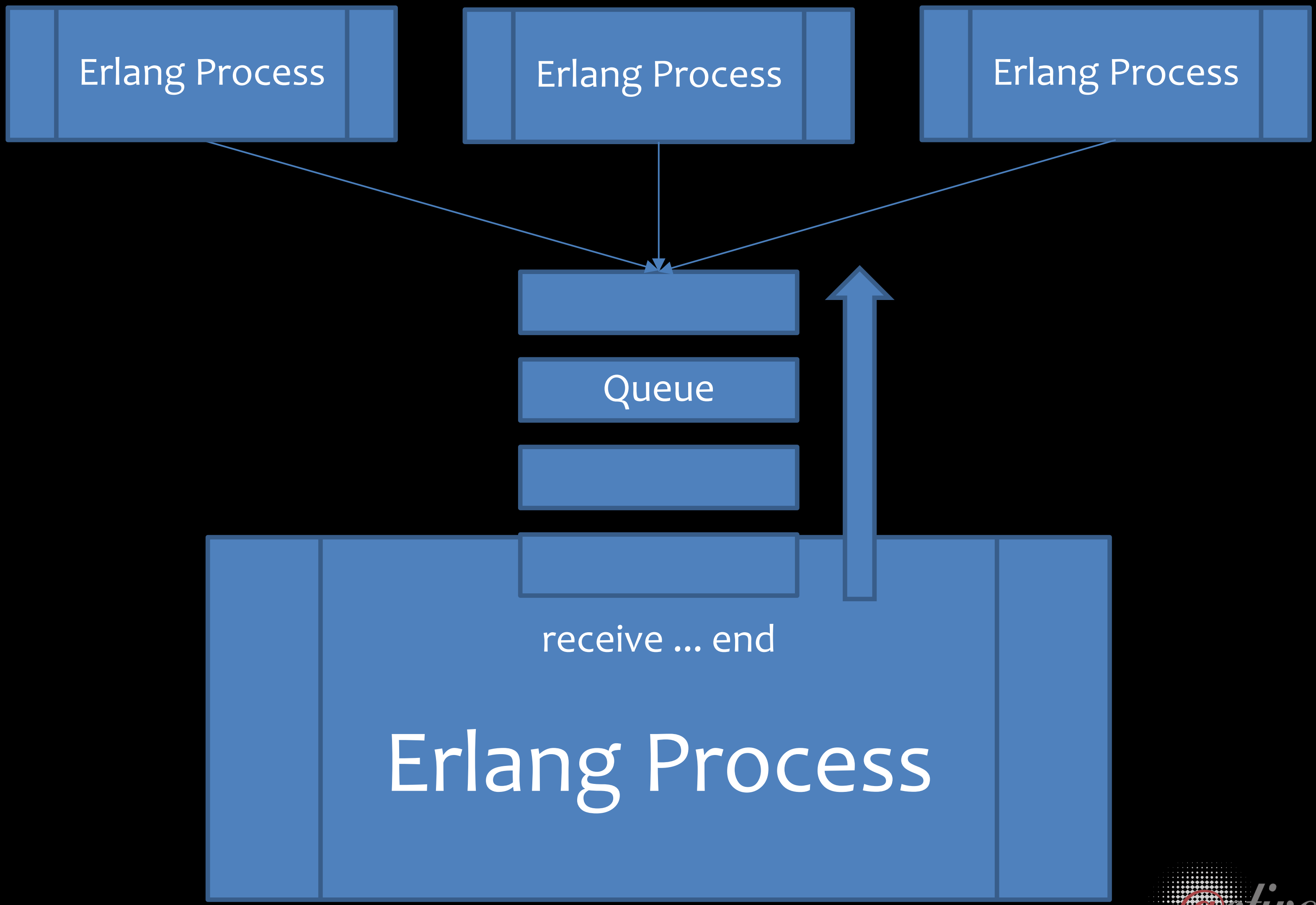
```
(with-nack
 (lambda (nack)
   (let ((replych (make-channel)))
    (spawn
     (lambda ()
       (send reqch
         (make-request replych nack))))
     (receive-rv replych))))
```

Notice You´ve Been Dumped

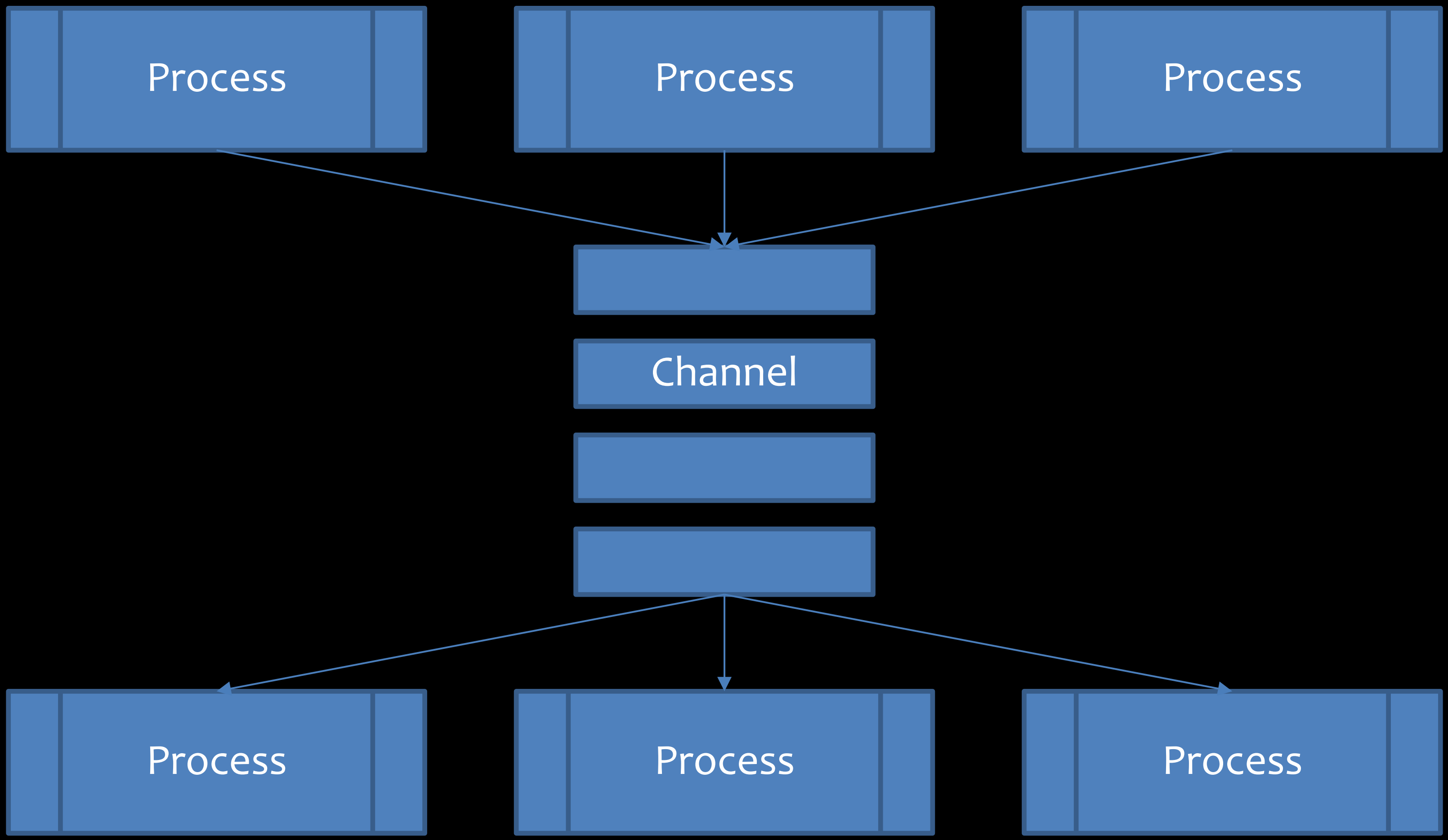
```
(let* ((rq (receive reqch))
       (replych (request-reply-channel rq))
       (nack (request-nack rq)))

  (select
   (wrap (send-rv replych)
     (lambda (_) <commit>))
   (wrap nack
     (lambda (_) <abort>))))
```

Erlang Processes



Channels



core.async vs. CML

merge
function

```
Usage: (merge chs)
       (merge chs buf-or-n)
```

Takes a collection of source channels and returns a channel which contains all values taken from them. The returned channel will be unbuffered by default, or a buf-or-n can be supplied. The channel will close after all the source channels have closed.

Combinator!

User-Defined Combinators in core.async

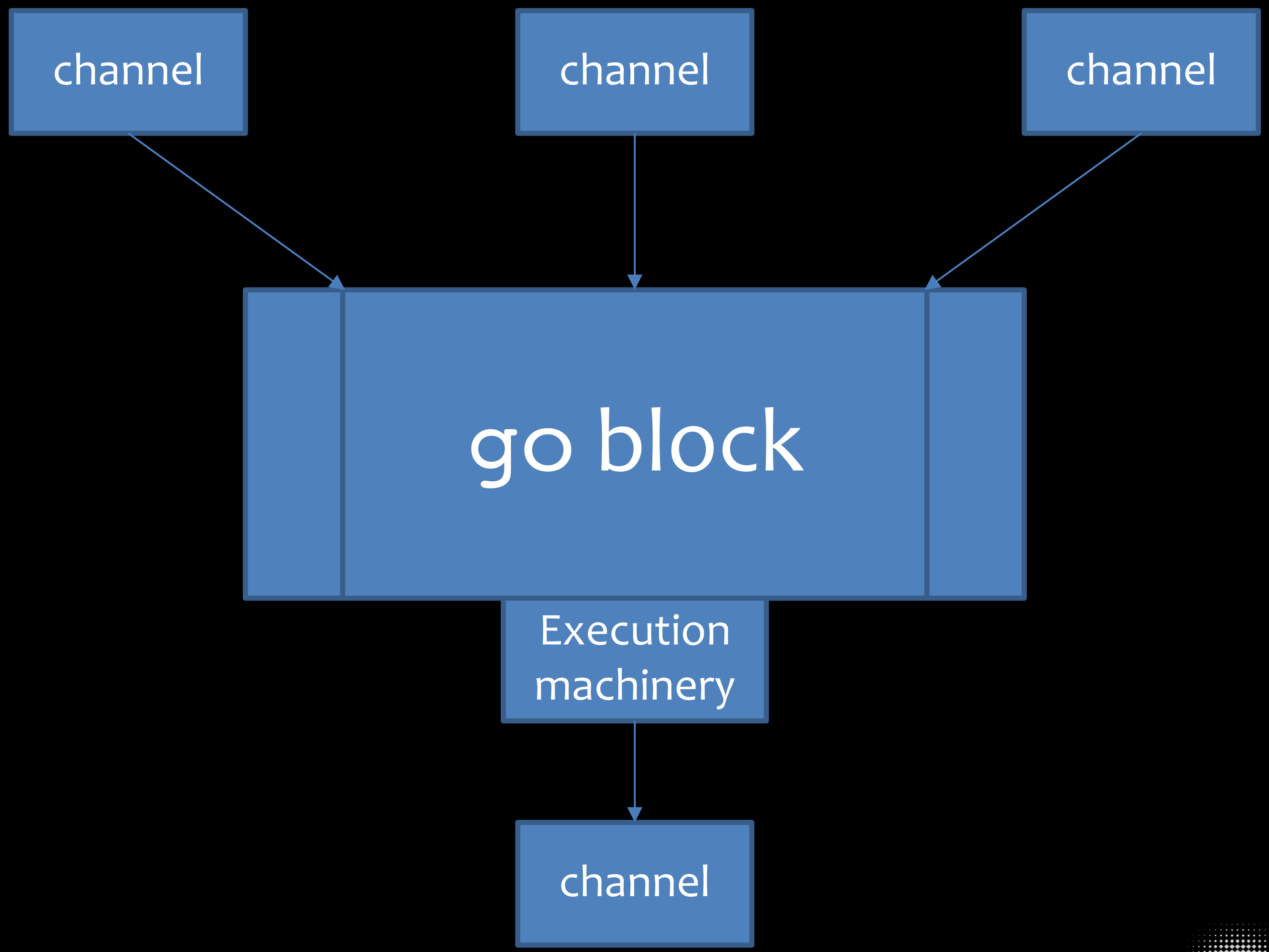
go
macro

Usage: (go & body)

Asynchronously executes the body, returning immediately to the calling thread. Additionally, any visible calls to <!, >! and alt!/alts! channel operations within the body will block (if necessary) by 'parking' the calling thread rather than tying up an OS thread (or the only JS thread when in ClojureScript). Upon completion of the operation, the body will be resumed.

Returns a channel which will receive the result of the body when completed

User-Defined Combinators in core.async



Concurrency on the JVM

Why is creating a Thread said to be expensive?

73

The Java tutorials say that creating a Thread is expensive. But why exactly is it expensive? What exactly is happening when a Java Thread is created that makes its creation expensive? I'm taking the statement as true, but I'm just interested in mechanics of Thread creation in JVM.

72

Java thread creation is expensive because there is a fair bit of work involved:

- A large block of memory has to be allocated and initialized for the thread stack.
- System calls need to be made to create / register the native thread with the host OS.
- Descriptors needs to be created, initialized and added to JVM internal data structures.

It is also expensive in the sense that the thread ties down resources as long as it is alive; e.g. the thread stack, any objects reachable from the stack, the JVM thread descriptors, the OS native thread descriptors.

All these things are platform specific, but they are not cheap on any Java platform I've ever come across.

http://stackoverflow.com/questions/5483047/why-is-creating-a-thread-said-to-be-expensive

Threads vs. Actors

stackoverflow

Questions Tags Users Badges Unanswered Ask Questio

Why can you have millions of actors in an application, but just 10,000 threads is too many?

2

Why can you have millions of actors in an application, but just 10,000 threads is too many? How is it that creating millions of actors is practical, but more than a couple threads is not? What can threads do that actors can't (or else we would use actors all the time!)?

multithreading actor

asked 4 years ago

viewed 442 times

active 4 years ago

core.async

```
(defn fake-search [kind]
  (fn [c query]
    (go
      (<! (timeout (rand-int 100)))
      (>! c [kind query]))))
```

http://hueypetersen.com/posts/2013/08/02/the-state-machines-of-core-async/

core.async

```
(let
    [old-frame__2202__auto__ (clojure.lang.Var/getThreadBindingFrame)]
    (try
      (clojure.lang.Var/resetThreadBindingFrame
        (ioc-macros/aget-object state_3730 3))
      (loop []
        (let
          [result (case (int (ioc-macros/aget-object state_3730 1))
                    3 (let [inst_3728 (ioc-macros/aget-object state_3730 2)
                            state_3730 state_3730]
                        (ioc-macros/return-chan state_3730 inst_3728))
                    2 (let [inst_3725 (ioc-macros/aget-object state_3730 2)
                            inst_3726 (vector kind query)
                            state_3730 (ioc-macros/aset-all! state_3730 5 inst_3725)]
                        (ioc-macros/put! state_3730 3 c inst_3726))
                    1 (let [inst_3722 (rand-int 100)
                            inst_3723 (timeout inst_3722)
                            state_3730 state_3730]
                        (ioc-macros/take! state_3730 2 inst_3723)))]
          (if (identical? result :recur)
            (recur)
            result)))
      (finally
        (clojure.lang.Var/resetThreadBindingFrame
          old-frame__2202__auto__))))
```

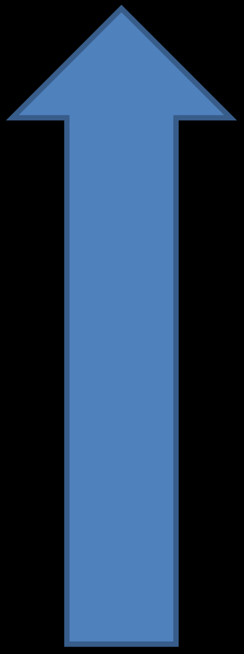
Buffering / Backpressure?

Channel Queues

The limits of my language are the limits of my mind. All I know is what I have words for.

Expressiveness

- Reagents
- Concurrent ML
- Clojure core.async
- Hoare's CSP
- Erlang



Design Choices?

- Reagents
- Concurrent ML
- Clojure core.async
- Hoare’s CSP
- Erlang

@rlive group

Concurrent ML

Compose all the things!

Open Questions

- backpressure
- distribution
- … while preserving composability