Clojure
core.async

@stuarthalloway
stu@cognitect.com

the problems

function chains make poor machines

direct-connect relationships

callback hell

j.u.c queues block real threads

threads are expensive and/or nonexistent

the opportunity

first class conveyance (queuelike)

indirection

multi reader/writer

library (not language) feature

Clojure brings to the JVM *and the browser*

CSP

Communicating Sequential Processes

first class processes

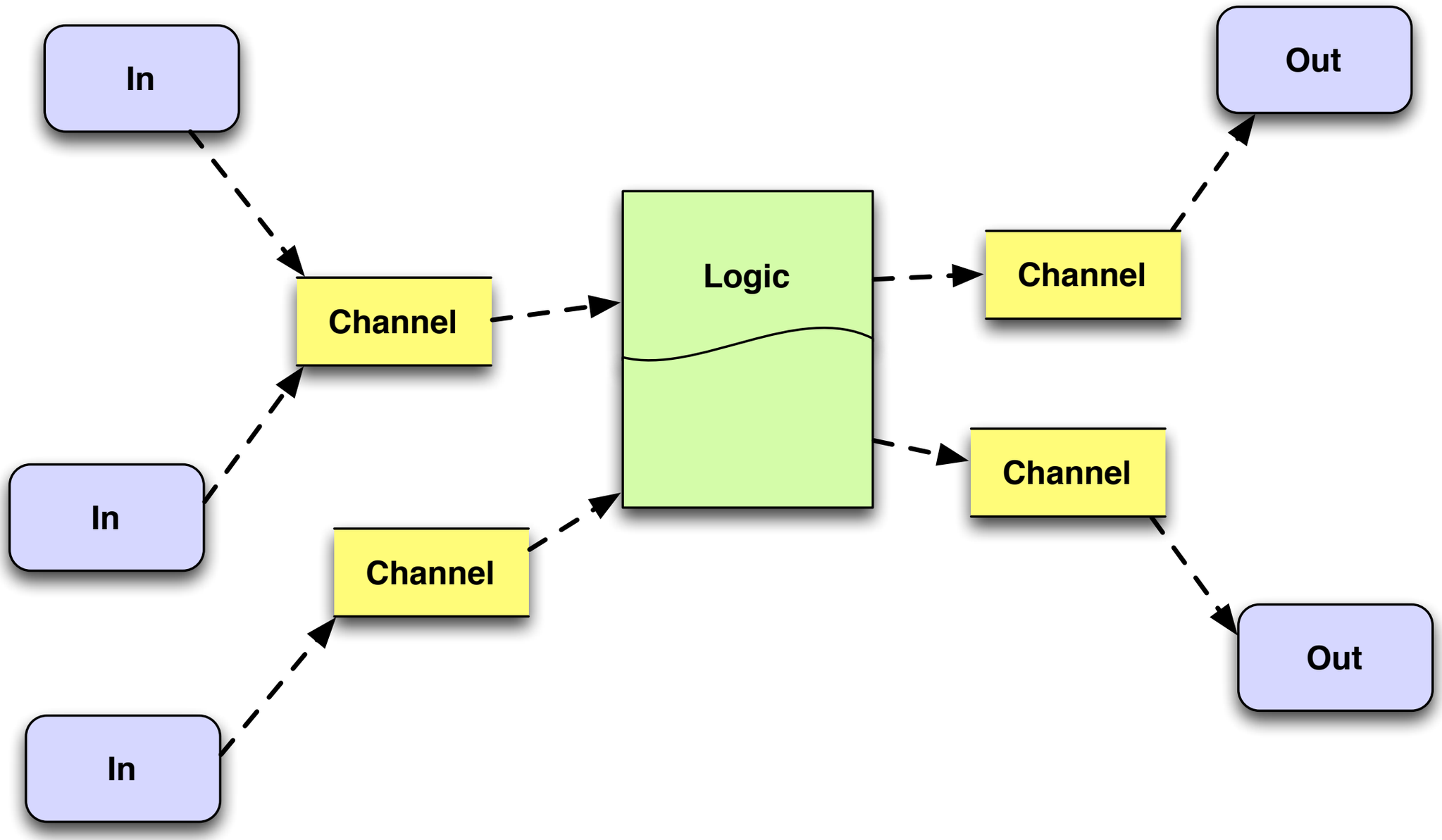
first class channels

coherent sequential logic

blocking, buffering, backpressure

select / alt

core.async



processes

(go

IOC 'thread',
state machine,
parking

)

(thread

real thread,
blocking

)

channels

op	go	thread	*(external)*
create	(chan)	(chan)	(chan)
put	(>! ch val)	(>!! ch val)	(put! ch val)
take	(<! ch)	(<!! ch)	(take! ch)
close	(close! ch)	(close! ch)	(close! ch)

buffering

strategy	semantics	example
unbuffered	rendezvous	`(chan)`
fixed	block when full	`(chan 10)`
sliding	drop oldest when full	`(chan (sliding-buffer 10))`
dropping	drop newest when full	`(chan (dropping-buffer 10))`

alt!, alt!!

wait on multiple channel operations

puts, takes, timeouts

compare unix select

works with threads *or go blocks*

alt!, alt!!

choice of takes

choice of puts

```
(alt!
  [c t]          ([val ch] (foo ch val))
  x              ([v] v)
  [[out val]] :wrote
  :default       42)
```

single "take" op

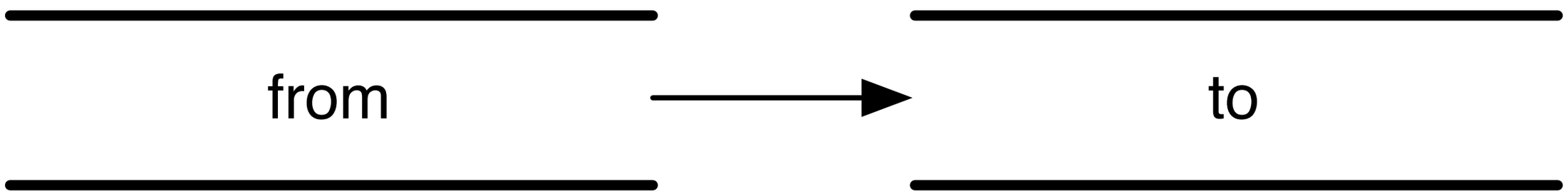
[value, channel] taken

10

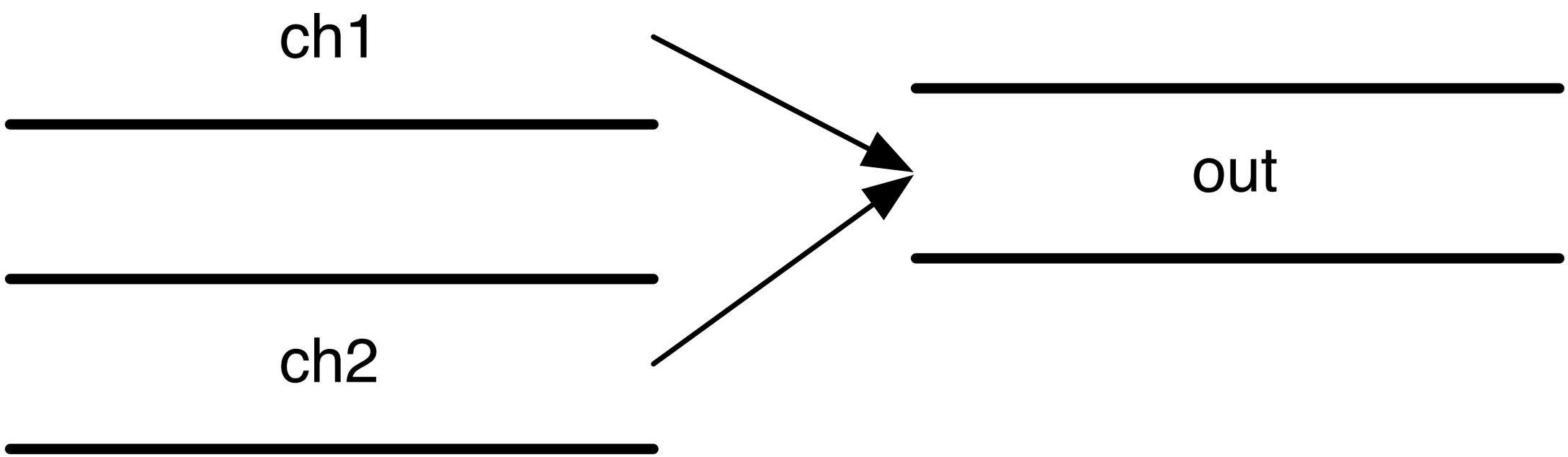
channel transform ops

fns of source channel	fns of target channel
`(map< f ch)`	`(map> f ch)`
`(filter< f ch)`	`(filter> f ch)`
`(remove< f ch)`	`(remove> f ch)`
`(mapcat< f ch)`	`(mapcat> f ch)`
`(reduce f init ch)`	

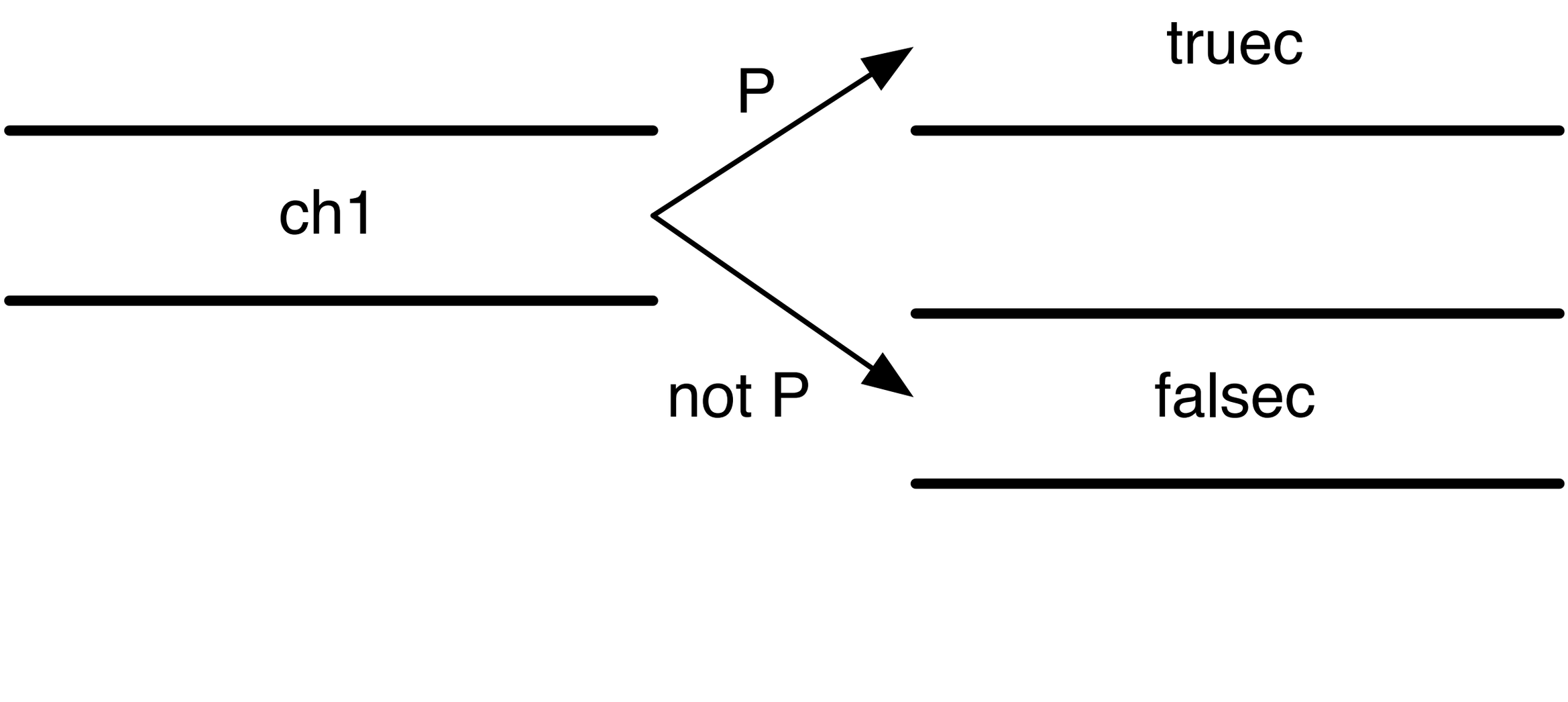
```
(pipe from to)
```



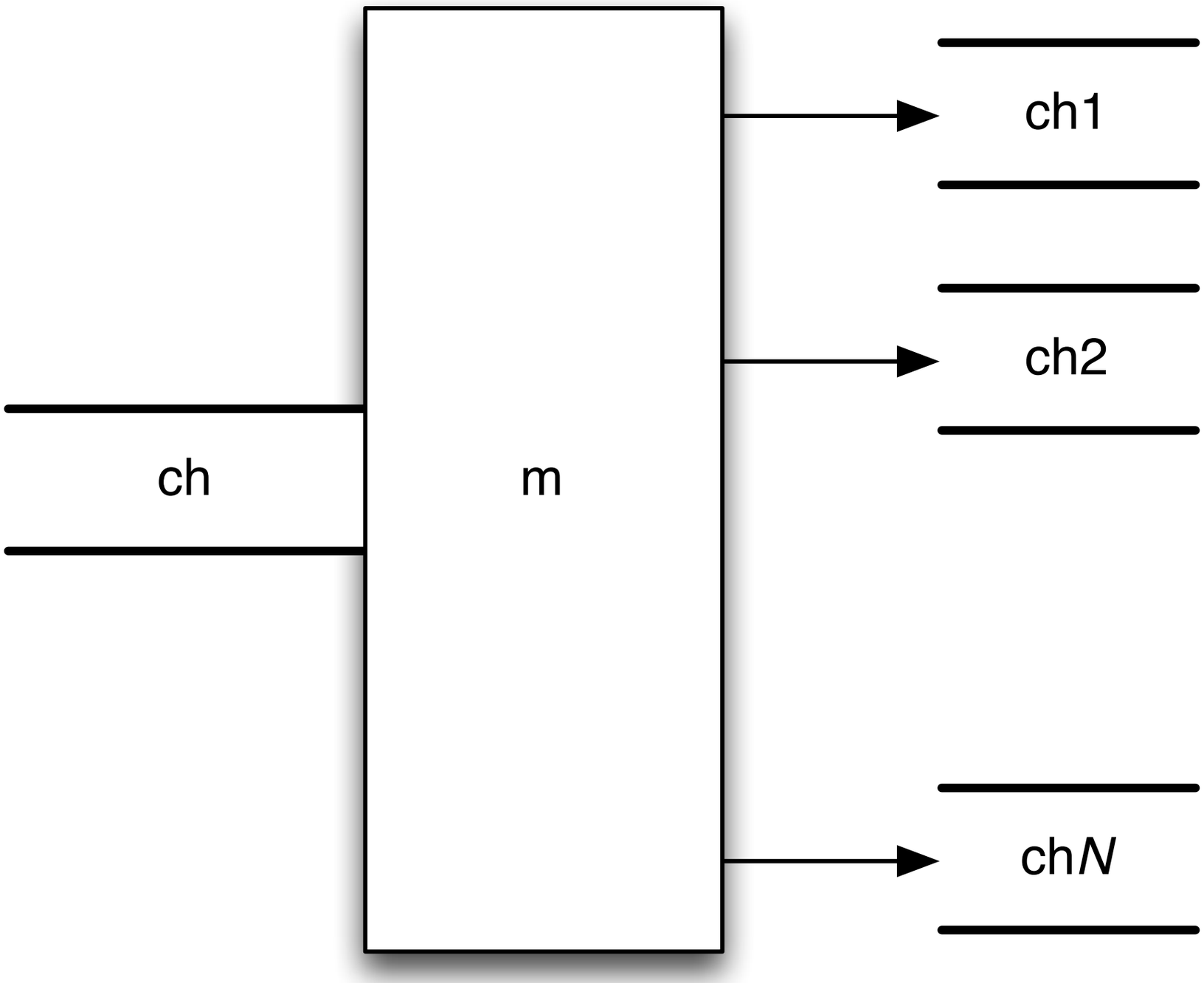
```
(merge ch1 ch2 out)
```



```
(split p ch1 truec falsec)
```



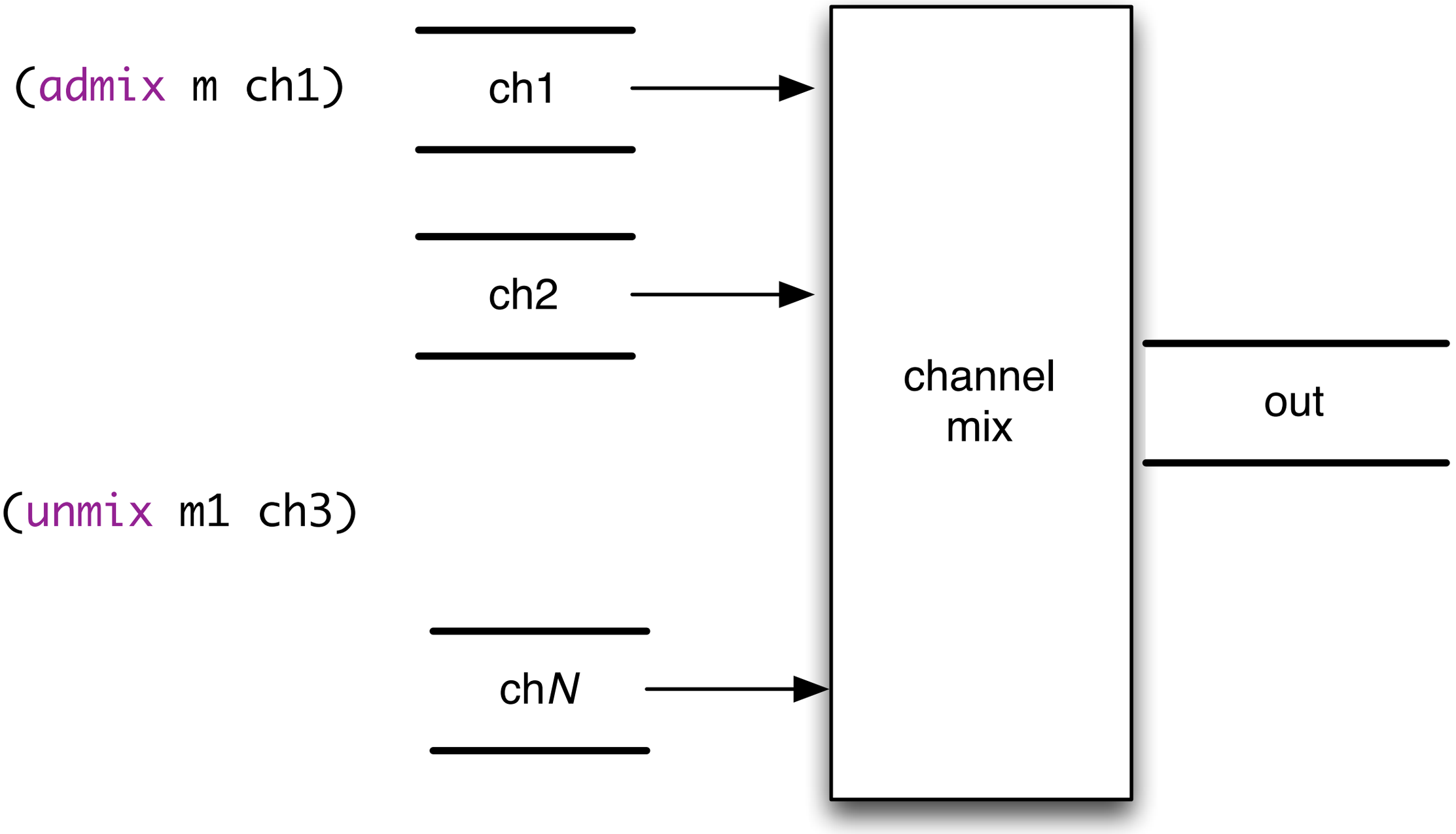
(mult ch)



(tap m1 ch1)

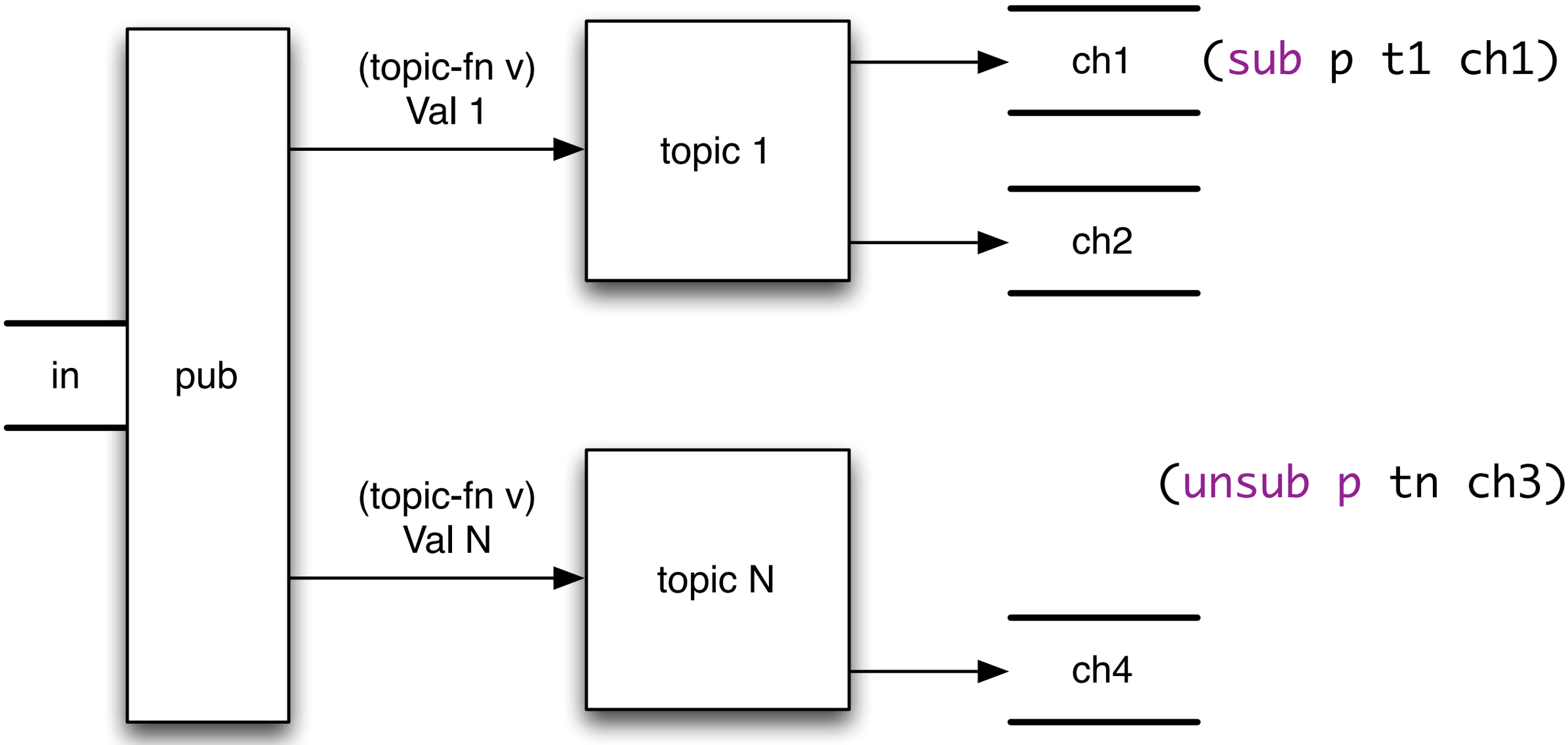
(untap m1 ch3)

(mix ch)



(also supports soloing, muting, pausing)

(pub ch topic-fn)



examples

running in the browser

```
(go (while true (<! (timeout 250)) (>! c 1)))
(go (while true (<! (timeout 1000)) (>! c 2)))
(go (while true (<! (timeout 1500)) (>! c 3)))
```

channel put

IOC 'thread"

```
(let [out (by-id "ex0-out")]
  (go (loop [results []]
        (set-html out (render results))
        (recur (-> (conj results (<! c)) (peekn 10))))))
```

channel get

search with SLA

```
(defn search [query]
  (let [c (chan)
        t (timeout 80)]
    (go (>! c (<! (fastest query web1 web2))))
    (go (>! c (<! (fastest query image1 image2))))
    (go (>! c (<! (fastest query video1 video2))))
    (go (loop [i 0
               ret []]
          (if (= i 3)
            ret
            (recur (inc i)
                   (conj ret (alt! [c t] ([v] v)))))))))
```

coordinates all searches and shared timeout

http://talks.golang.org/2012/concurrency.slide#50

20

differences from go

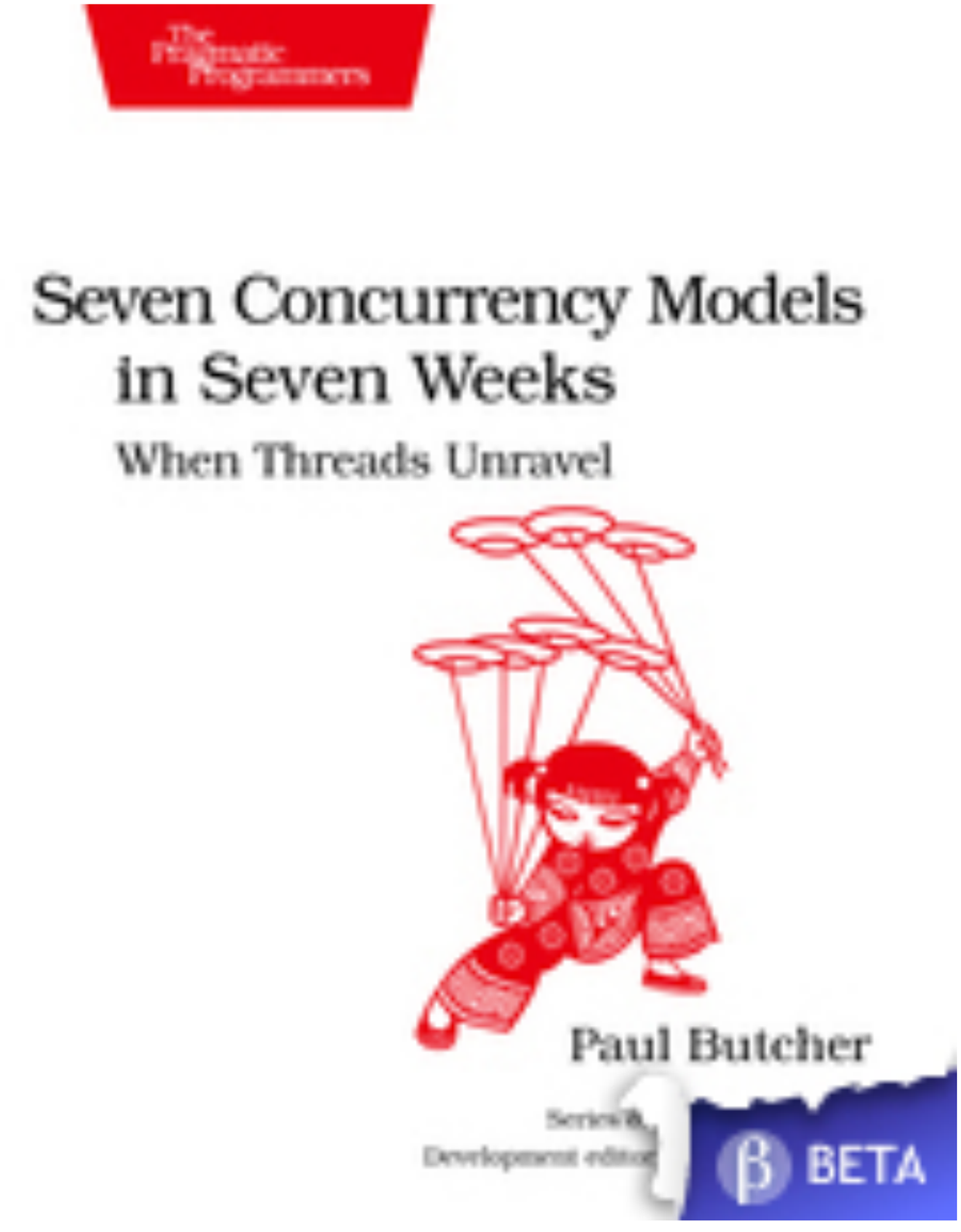
operations are expressions (not statements)

core.async is library, not language feature

alts! is a *function*

alt supports priority

what about actors?

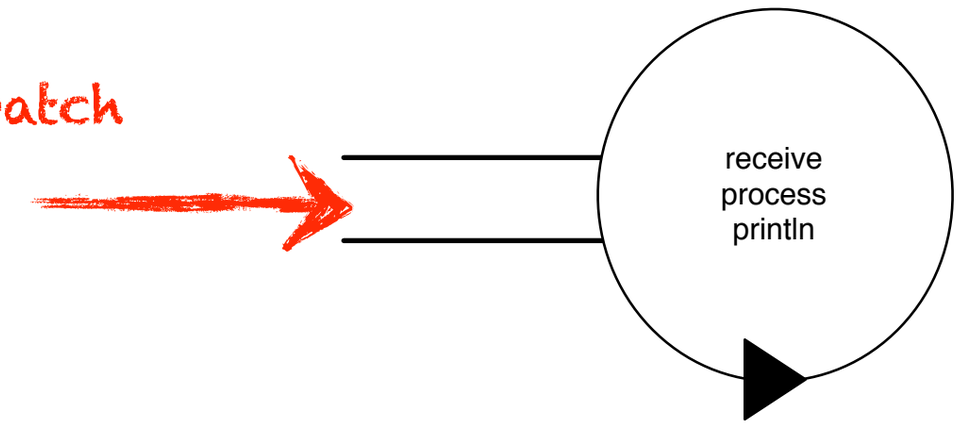


http://pragprog.com/book/pb7con/seven-concurrency-models-in-seven-weeks

hello actors

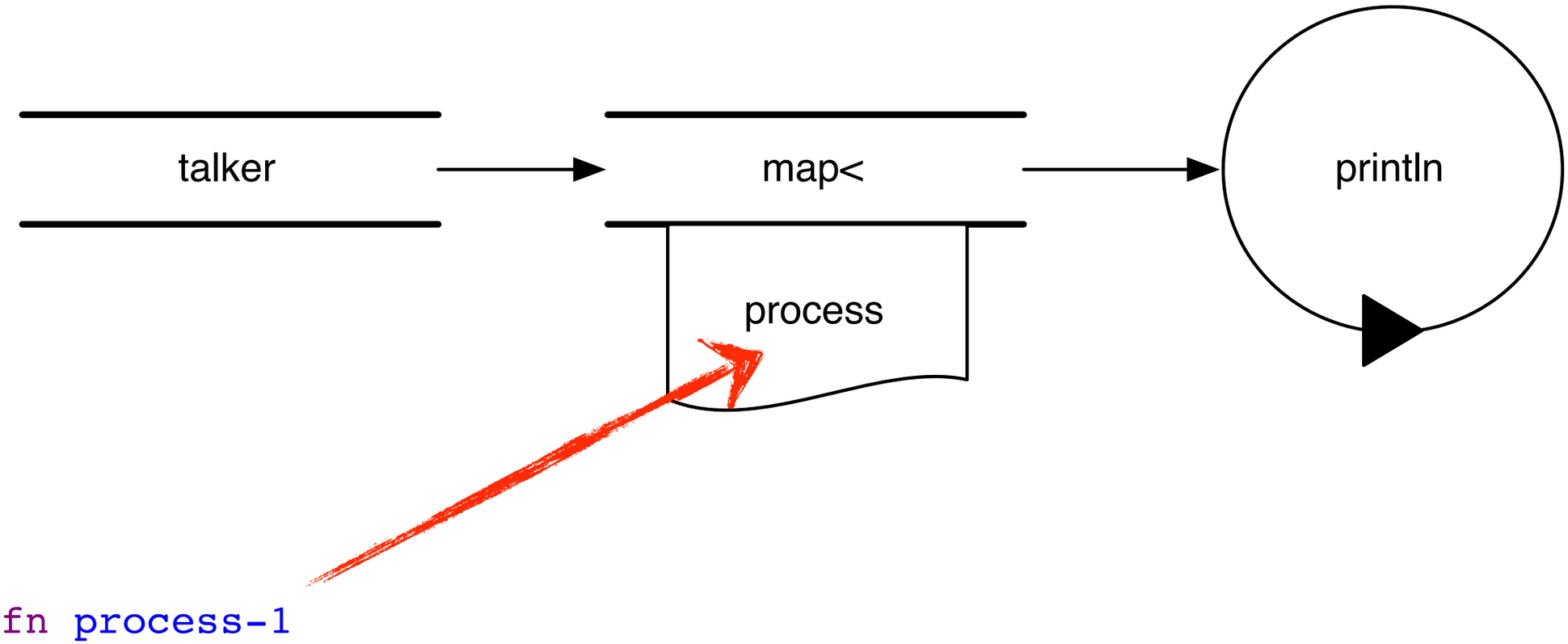
```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
    end
    loop
  end
end
```

channel, process, & dispatch fused together



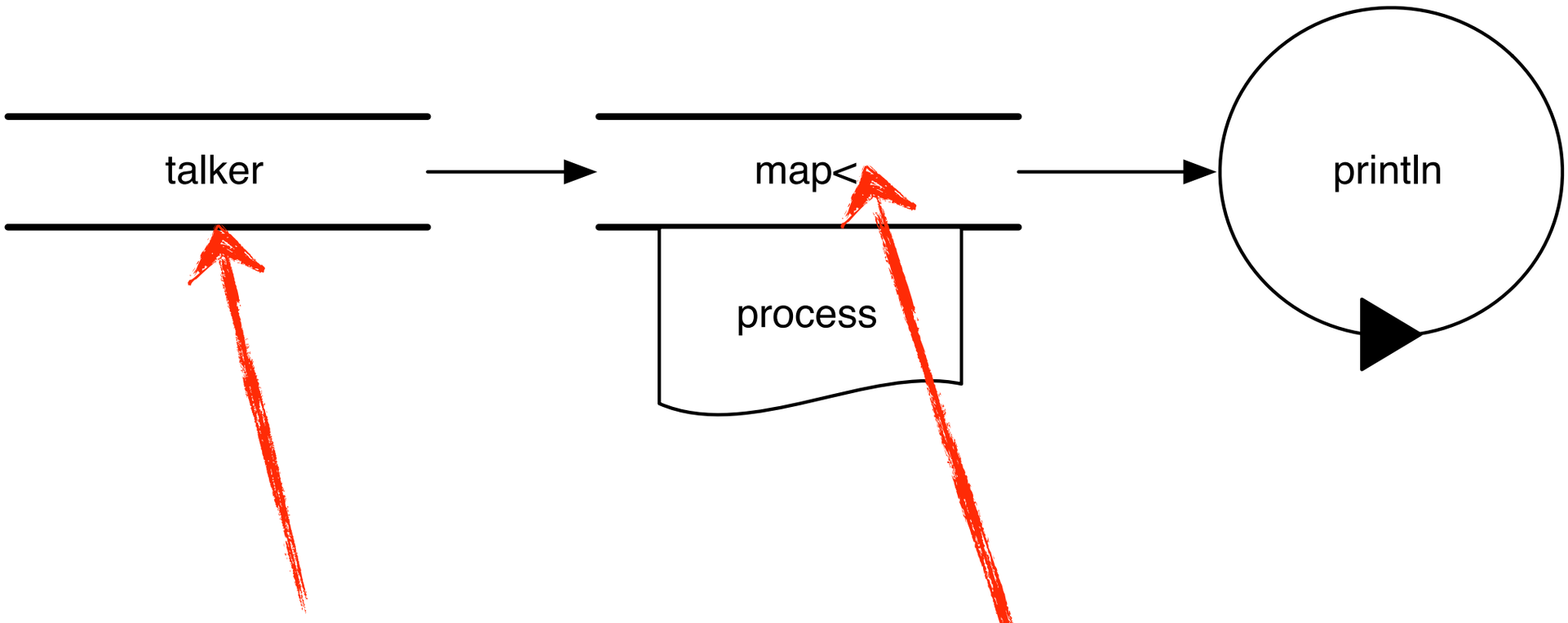

```
pid <- {:greet, "Huey"}
pid <- {:praise, "Dewey"}
pid <- {:celebrate, "Louie", 16}
```

closed (pattern) dispatch



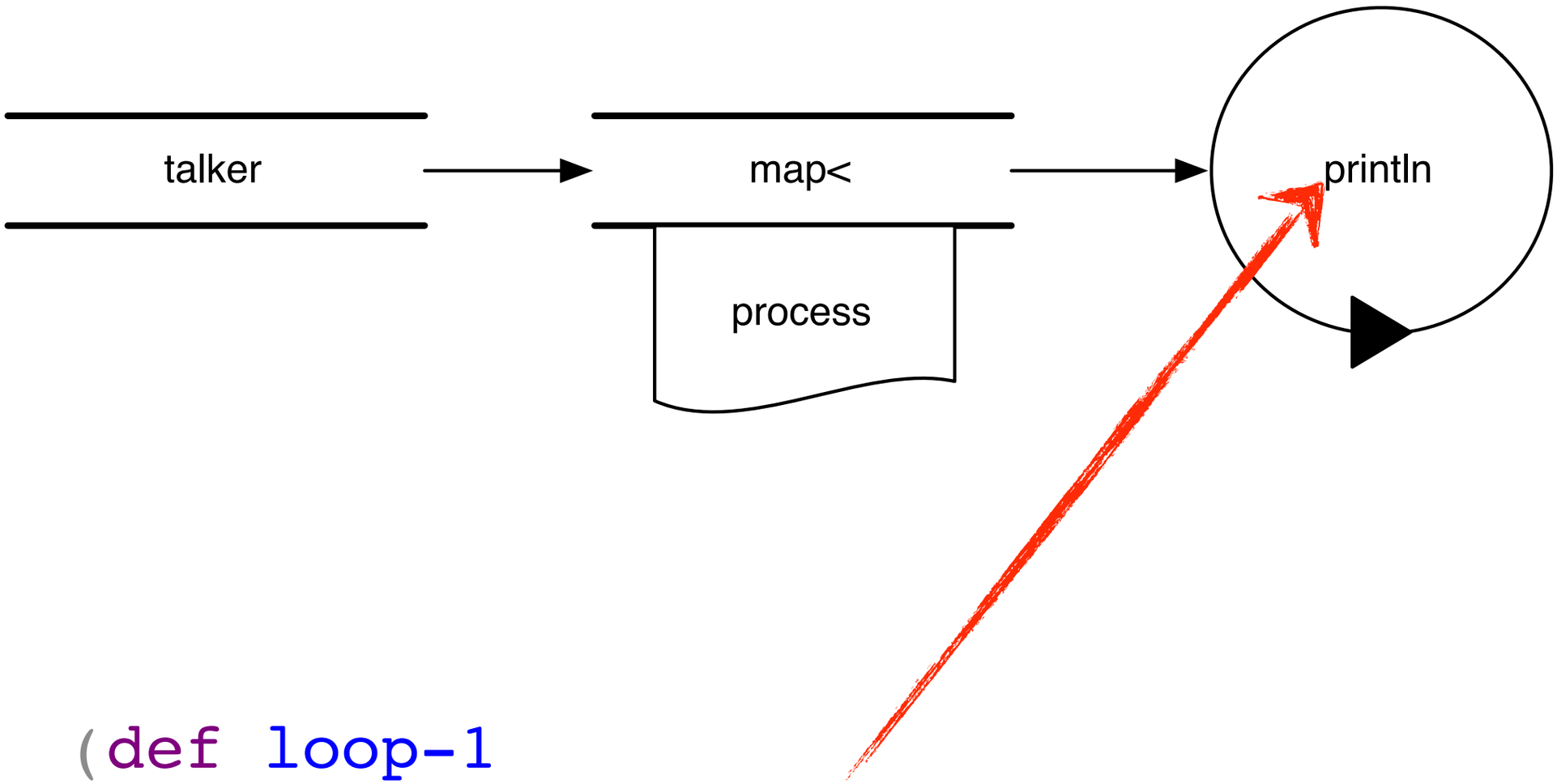

```clojure
(defn process-1
  [item]
  (match
   [item]
   [[:greet & [name]]] (str "Hello " name)
   [[:praise & [name]]] (str name ", you're amazing")
   [[:celebrate & [name age]]] (str "Here's to another " age " years, " name)))
```

+channels



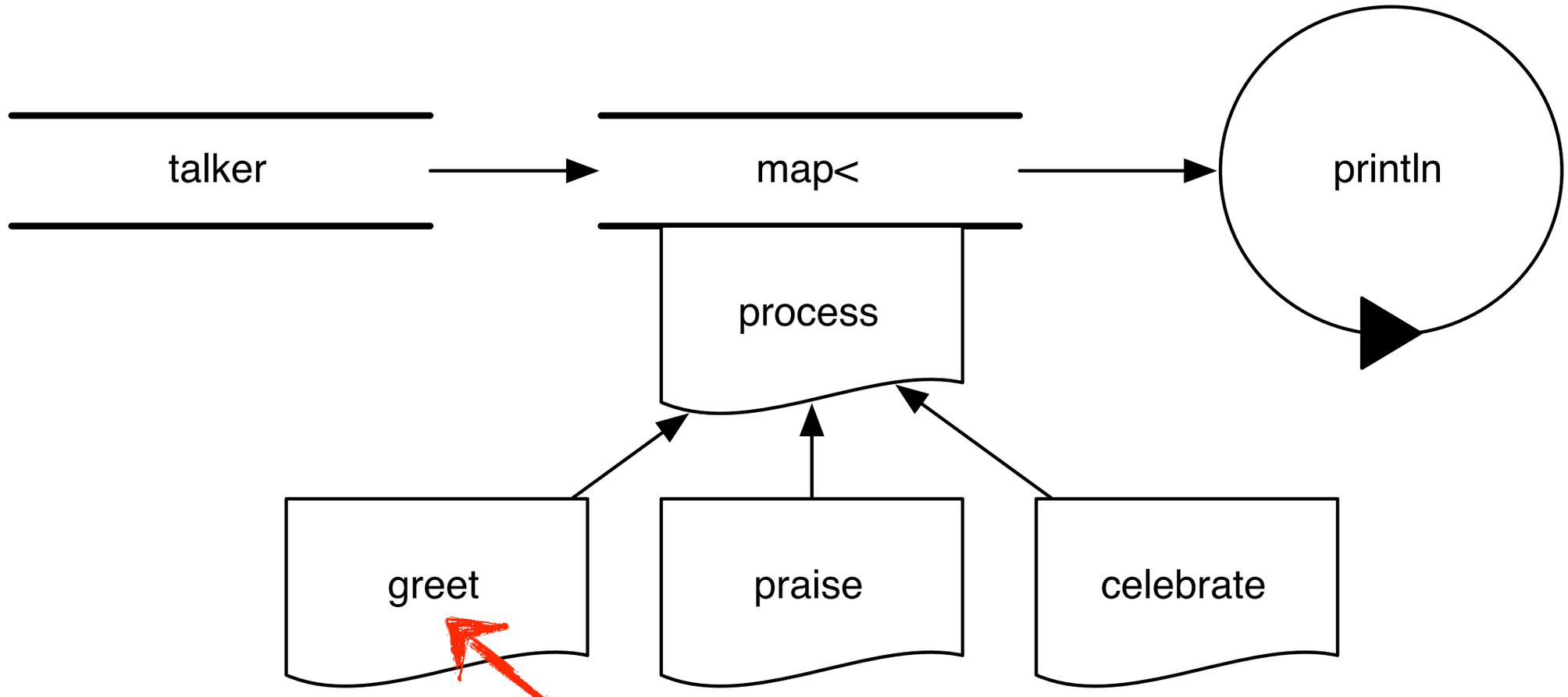

```
(def talker-ch-1 (chan))

    (def composed-ch-1
      (->> talker-ch-1 (map< process-1)))
```

+processes



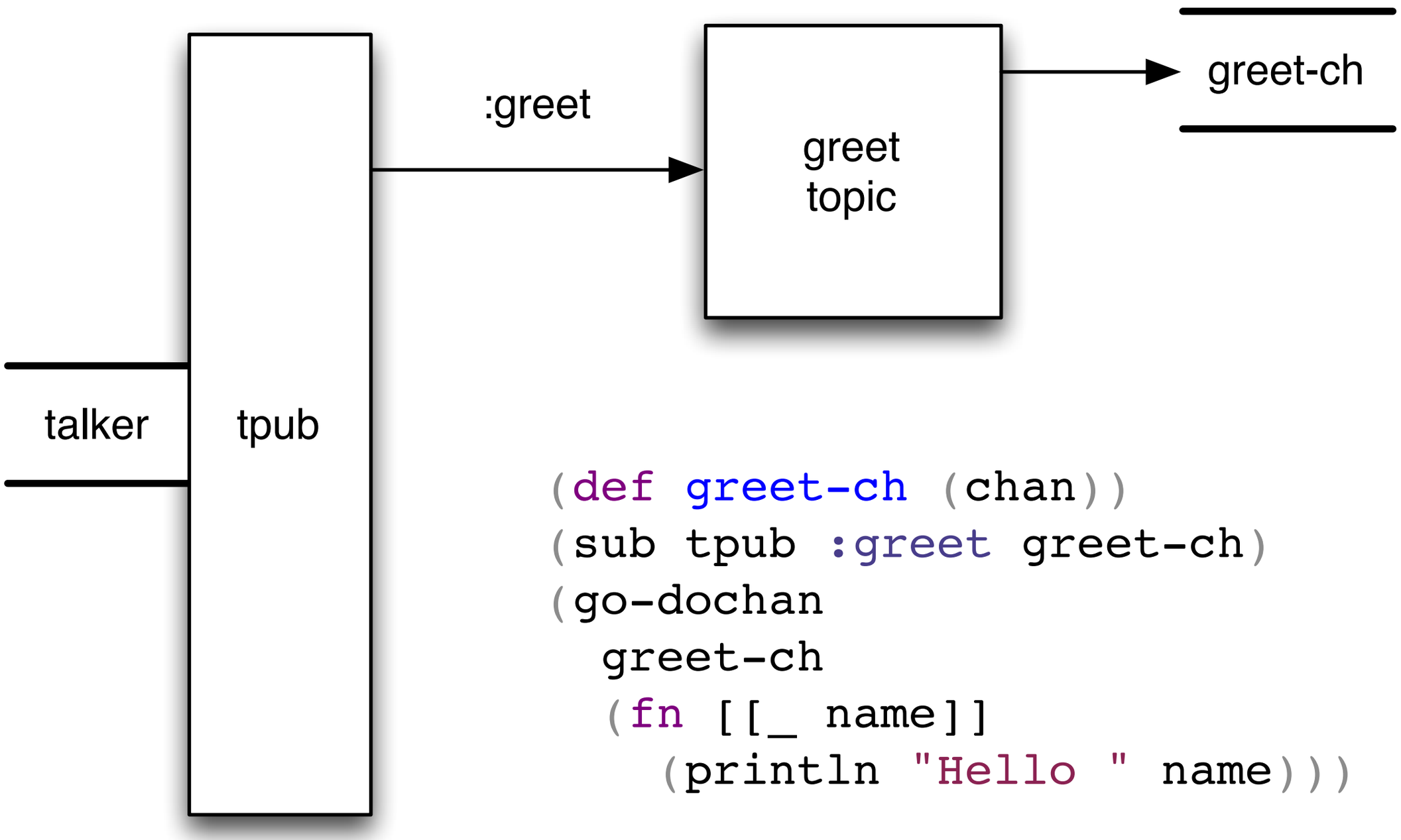

```
(def loop-1
  (go-loop [msg (<! composed-ch-1)]
           (when msg
             (println msg)
             (recur (<! composed-ch-1)))))
```

substitute open dispatch




```
(defmulti process-2 (fn [[k & more]] k))

(defmethod process-2 :greet
  [[_ & [name]]]
  (str "Hello " name))
```

substitute pub/sub




```
(def greet-ch (chan))
(sub tpub :greet greet-ch)
(go-dochan
  greet-ch
  (fn [[_ name]]
    (println "Hello " name)))
```

resources

core.async and CSP
http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html
https://github.com/clojure/core.async
http://www.cs.kent.ac.uk/projects/ofa/jcsp/
http://www.usingcsp.com/

@stuarthalloway
https://github.com/stuarthalloway/presentations/wiki. Presentations
http://www.linkedin.com/pub/stu-halloway/0/110/543/
https://twitter.com/stuarthalloway
mailto:stu@cognitect.com


cognitect


relevance


Datomic