Reagents:

Functional programming meets scalable concurrency

Aaron Turon
Northeastern University

Concurrency ≠ Parallelism

Concurrency is overlapped execution of processes.

Parallelism is simultaneous execution of computations.

The trouble is that *essentially all the interesting applications of concurrency involve the deliberate and controlled mutation of shared state*, such as screen real estate, the file system, or the internal data structures of the program. The right solution, therefore, is to provide mechanisms which allow (though alas they cannot enforce) the safe mutation of shared state.

-- Peyton Jones, Gordon, and Finne
in *Concurrent Haskell*

Concurrency ∩ Parallelism = *Scalable* Concurrency

Use cases:

- Concurrent programs on parallel hardware (e.g. OS kernels)
- *Implementing* parallel abstractions (e.g. work stealing for data parallelism)
- "Last mile" of parallel programming (where we must resort to concurrency)

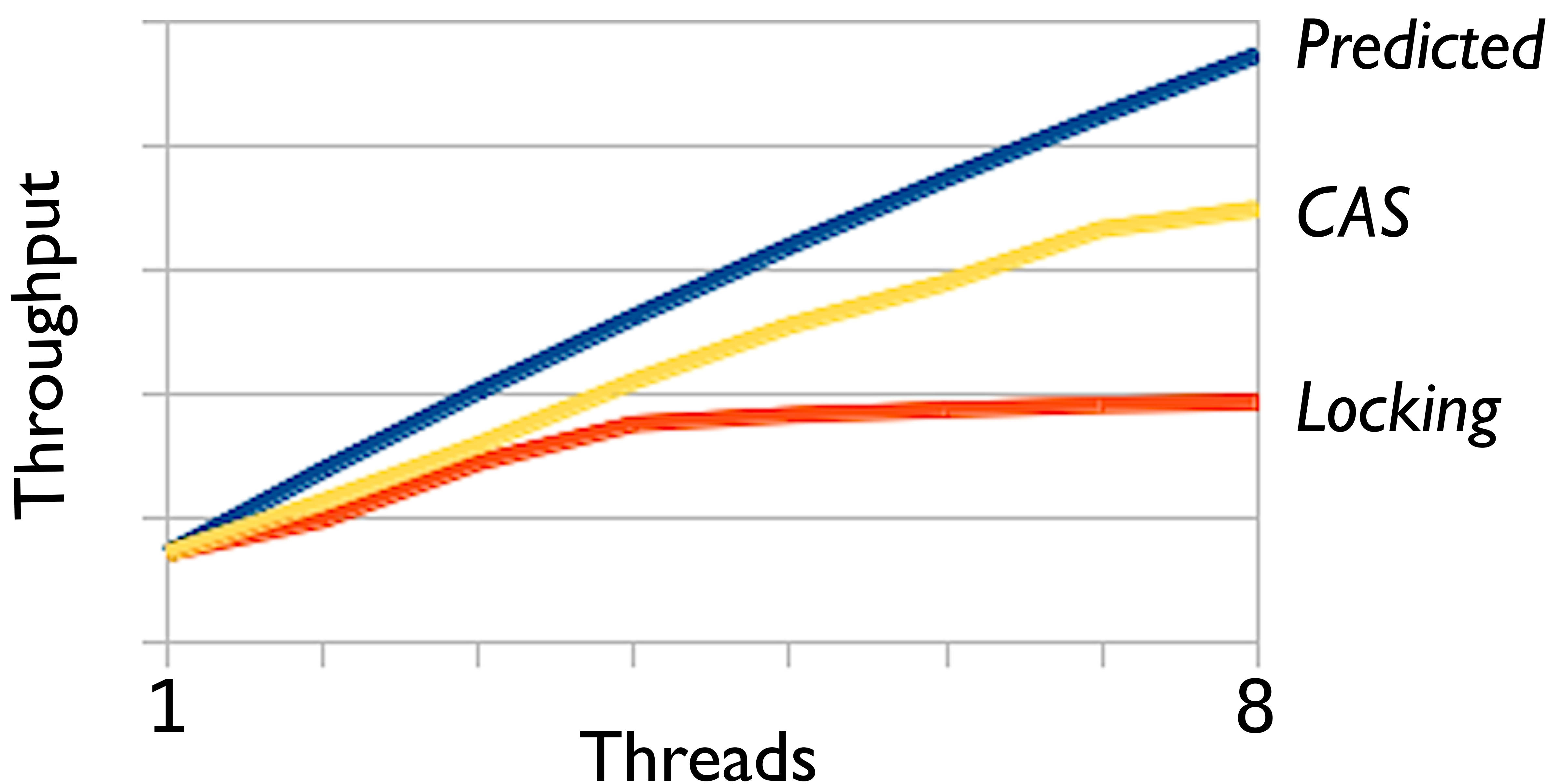
```scala
class LockCounter {
  private var c: Int = 0
  private var l = new Lock
  def inc: Int = {
    l.lock()
    val old = c
    c = old + 1
    l.unlock()
    old
  }
}
```

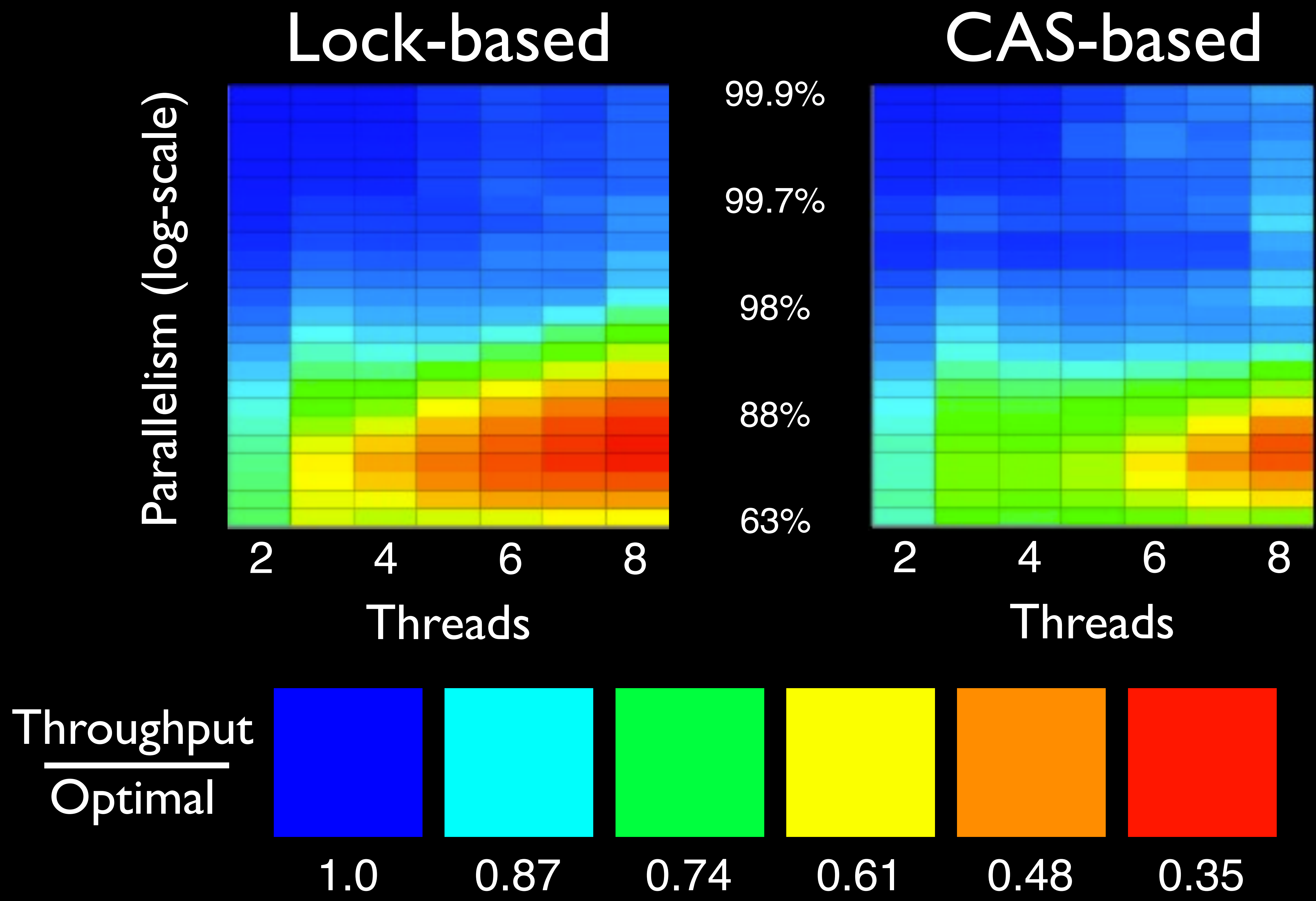
```scala
class CASCounter {
  private var c = new AtomicRef[Int](0)
  def inc: Int = {
    while (true) {
      val old = c
      if (c.cas(old, old+1)) return old
    }
  }
}
```

A simple test

- Increment counter
- Busywait for *t* cycles (no cache interaction)
- Repeat

Results for 98% parallelism



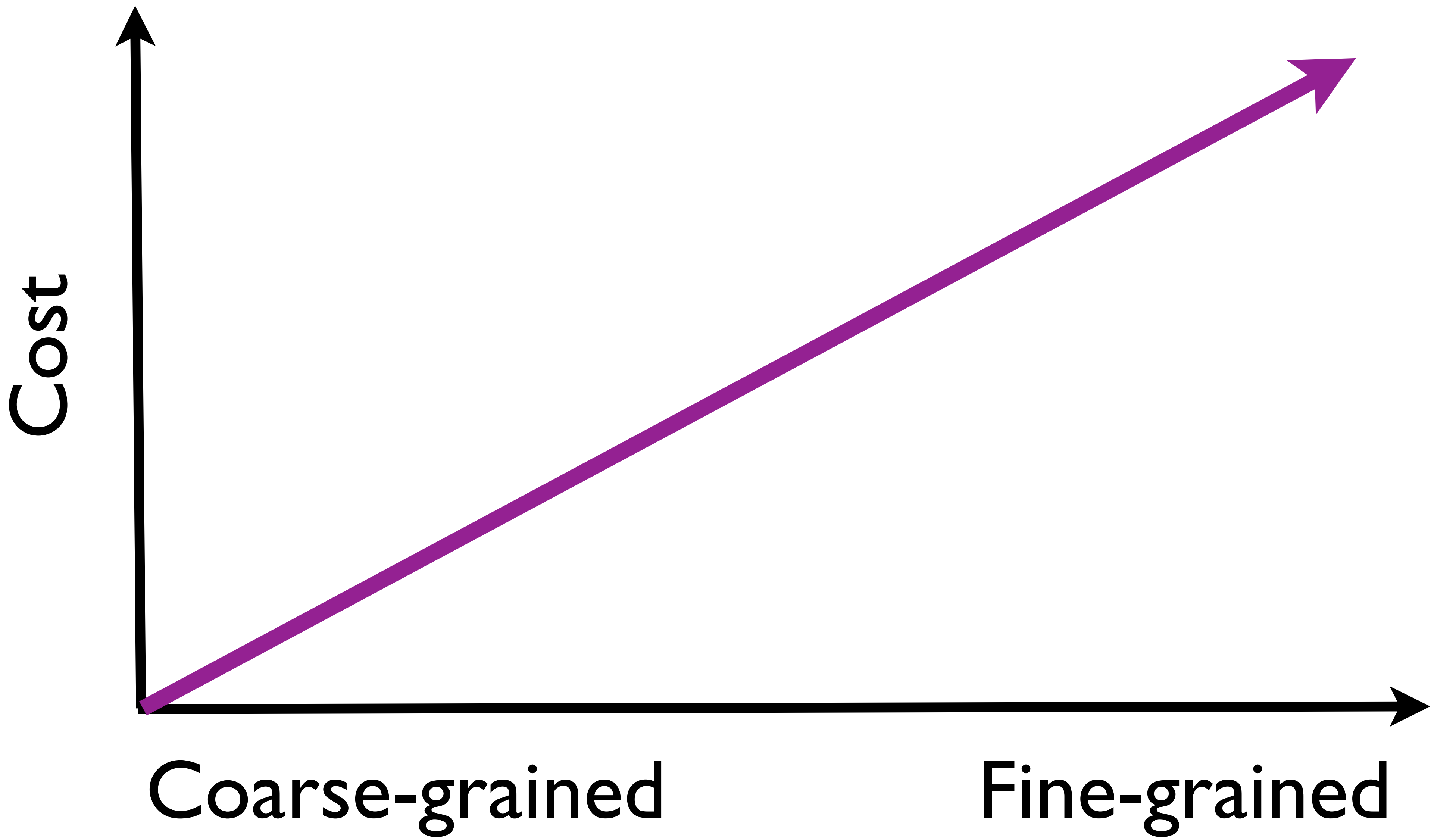


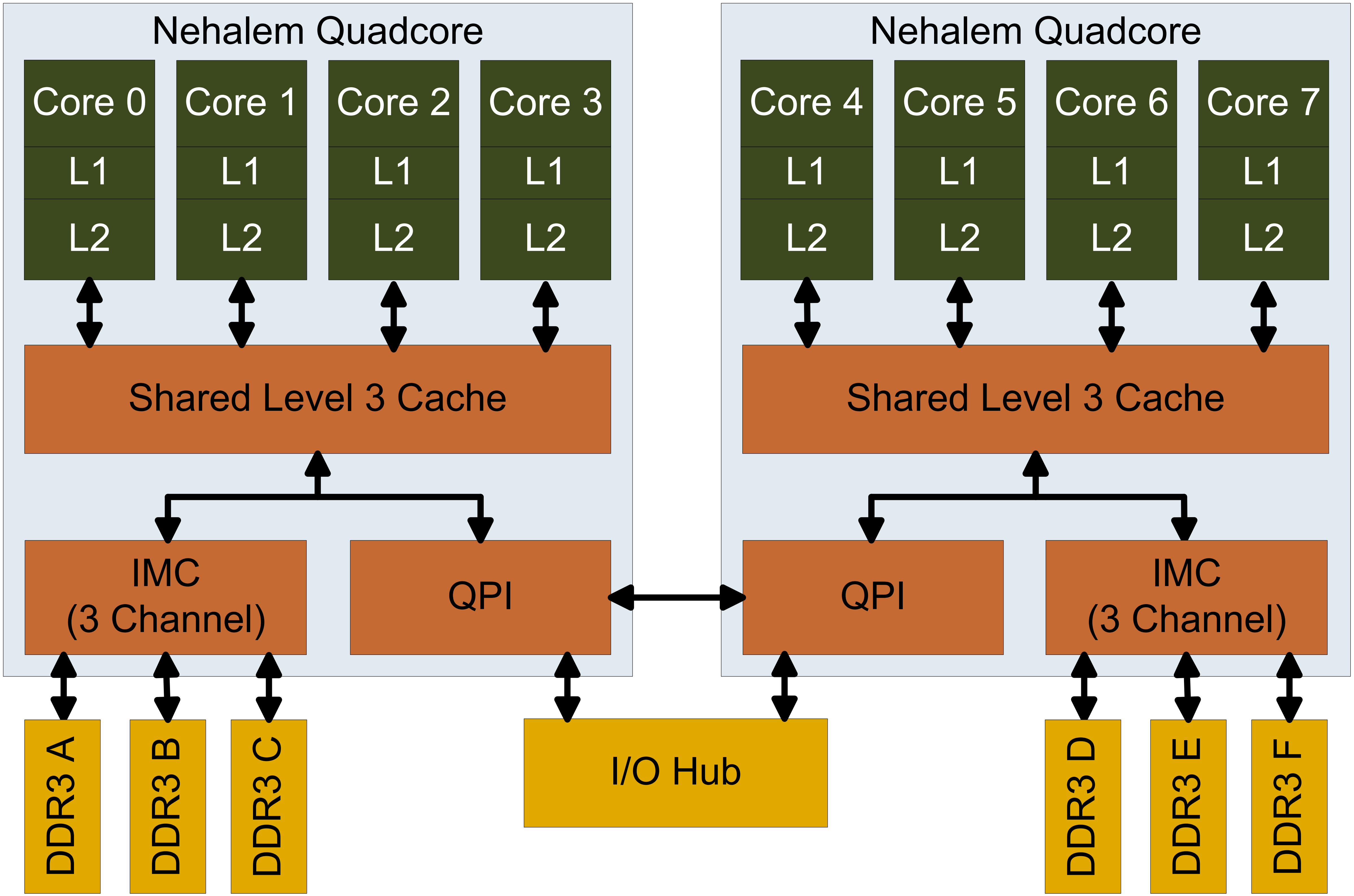
Lock-based
CAS-based
Parallelism (log-scale)
99.9%
99.7%
98%
88%
63%
2
4
6
8
Threads
2
4
6
8
Threads
Throughput
Optimal
1.0
0.87
0.74
0.61
0.48
0.35

What's going on here?

What's going on here?

Communication




Nehalem Quadcore
Core 0
Core 1
Core 2
Core 3
L1
L2
Shared Level 3 Cache
IMC
(3 Channel)
QPI
Nehalem Quadcore
Core 4
Core 5
Core 6
Core 7
L1
L2
Shared Level 3 Cache
QPI
IMC
(3 Channel)
DDR3 A
DDR3 B
DDR3 C
I/O Hub
DDR3 D
DDR3 E
DDR3 F

java.util.concurrent

Synchronization

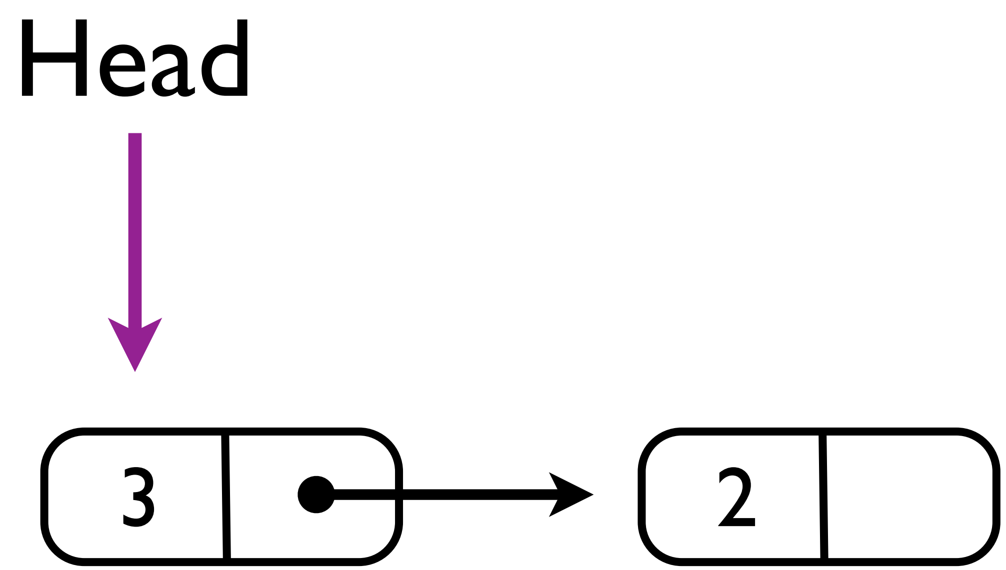
Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition variables
Countdown latches
Cyclic barriers
Phasers
Exchangers

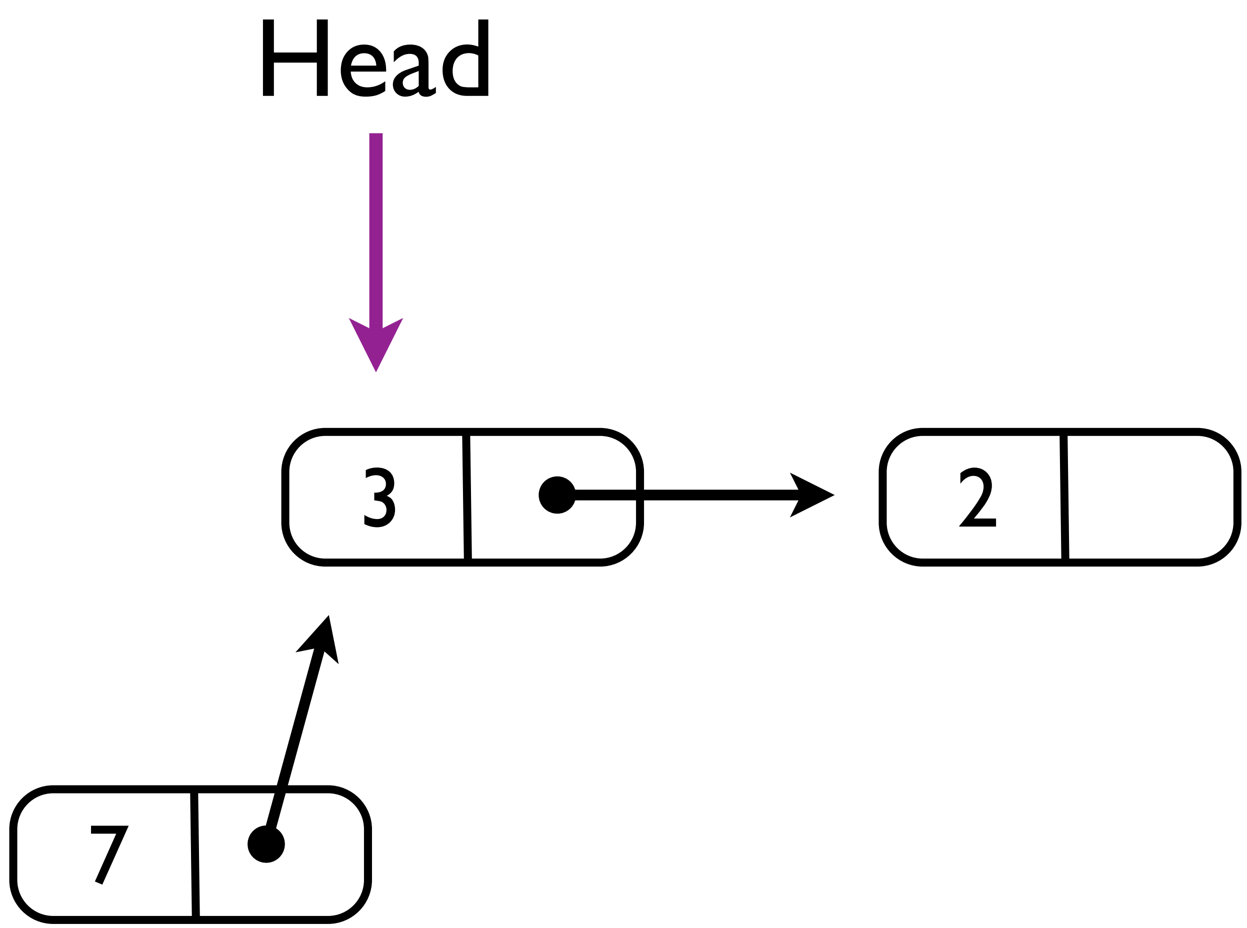
Data structures

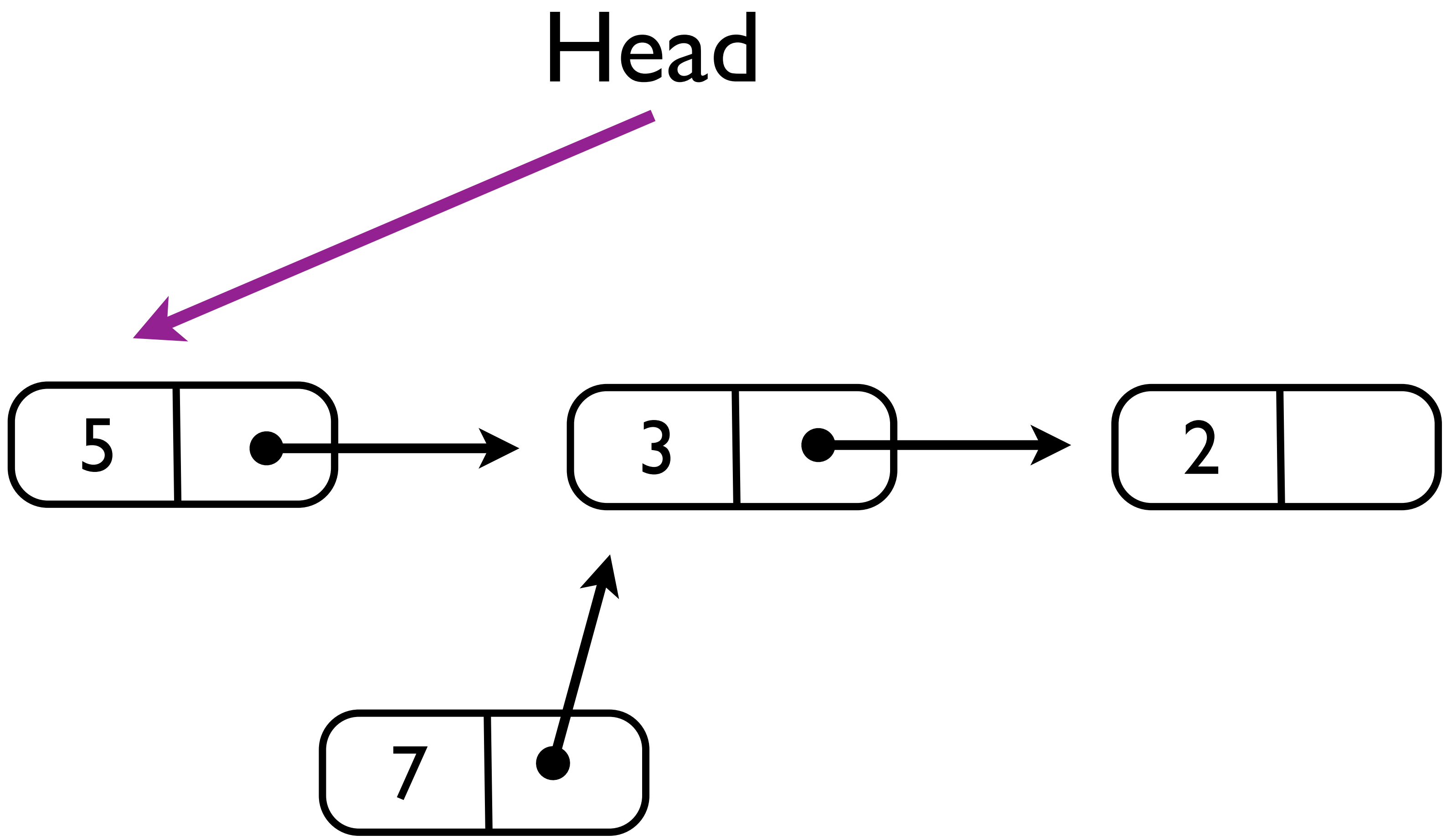
Queues
- Nonblocking
- Blocking (array & list)
- Synchronous
- Priority, nonblocking
- Priority, blocking

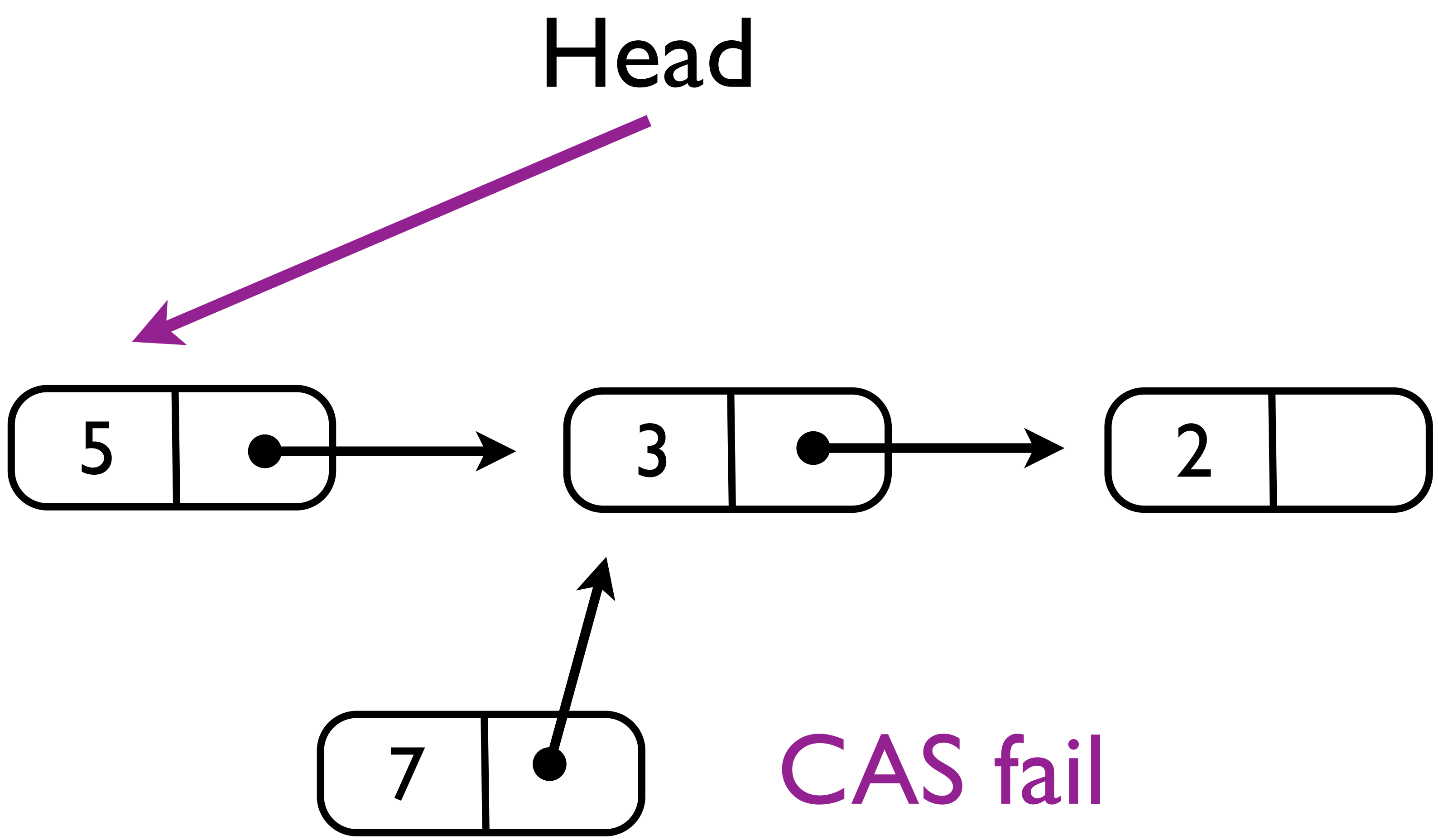
Deques
Sets
Maps (hash & skiplist)

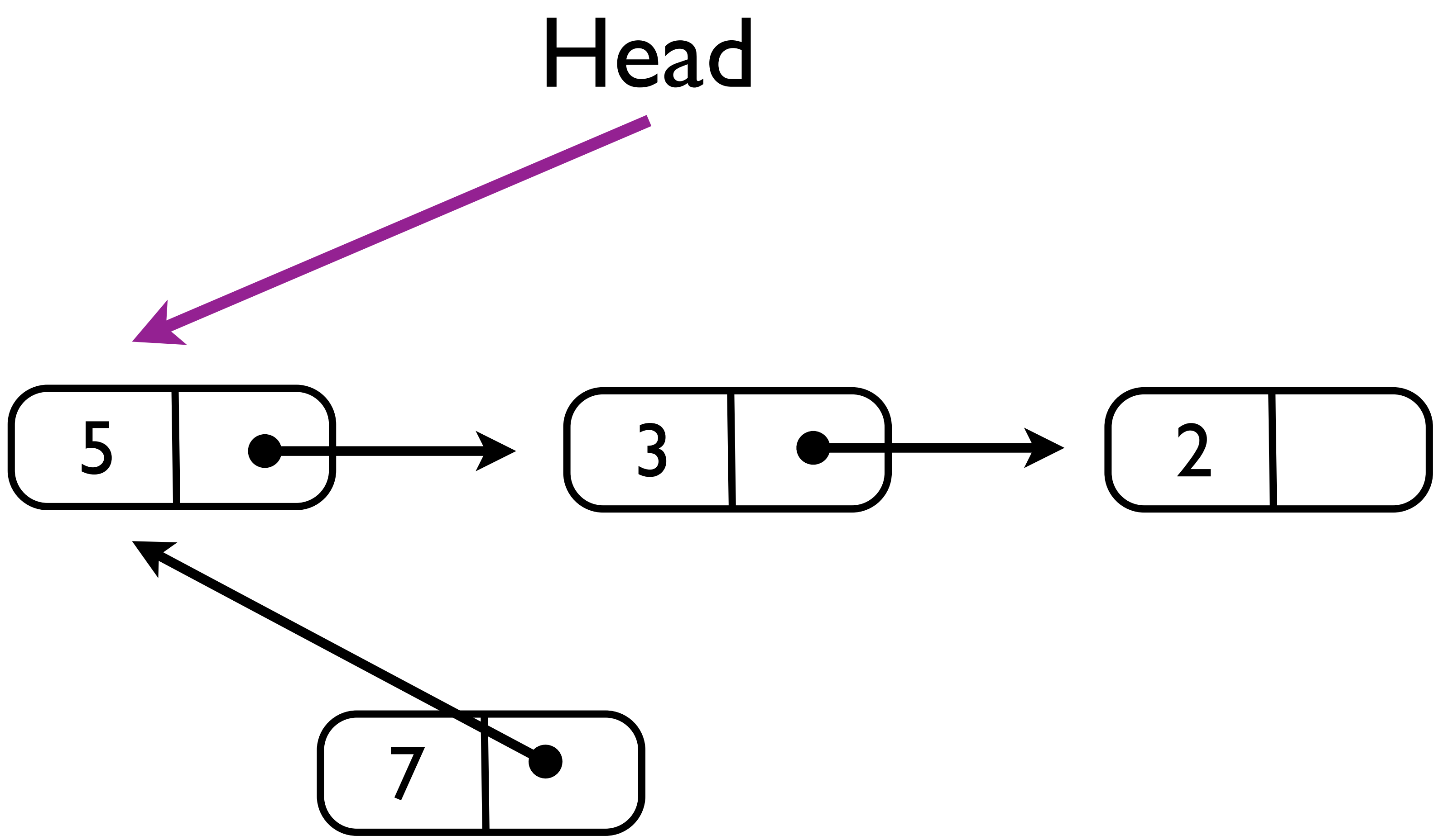
```scala
class TreiberStack[A] {
  private val head =
    new AtomicRef[List[A]](Nil)

  def push(a: A) {
    val backoff = new Backoff
    while (true) {
      val cur = head.get()
      if (head.cas(cur, a :: cur)) return
      backoff.once()
    }
  }

...
```

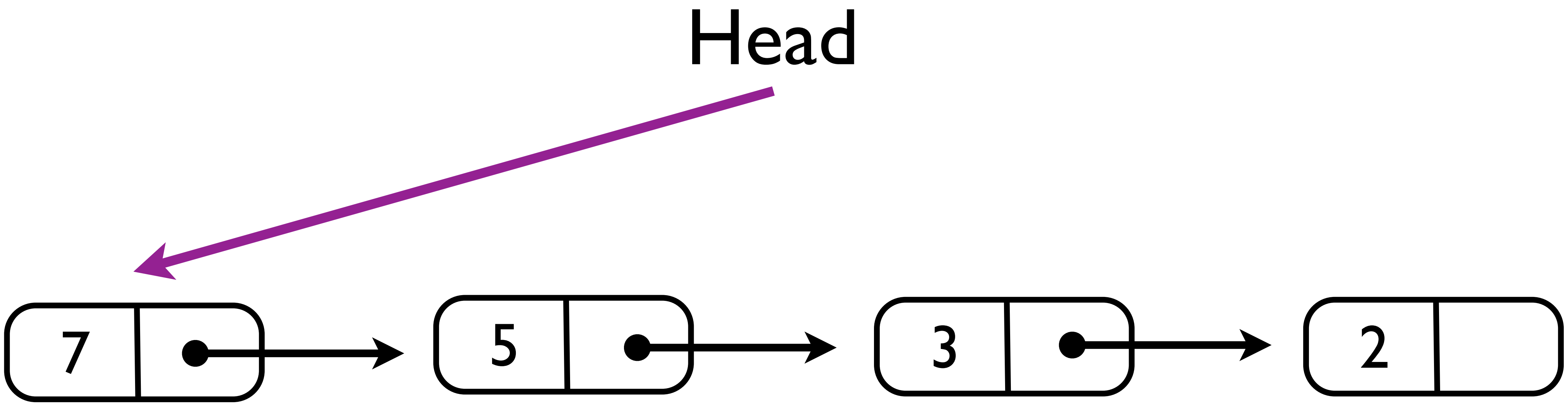

Head
3
2


Head
3
2
7


Head
5
3
2
7


Head
5
3
2
7
CAS fail


Head
5
3
2
7


Head
7
5
3
2

```scala
def tryPop(): Option[A] = {
  val backoff = new Backoff
  while (true) {
    val cur = head.get()
    cur match {
      case Nil     => return None
      case a::tail =>
        if (head.cas(cur, tail))
          return Some(a)
    }
    backoff.once()
  }
}
```

The Problem:

Concurrency libraries are indispensable, but hard to build and extend

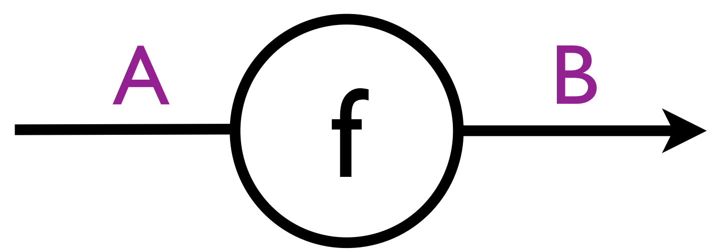
The Proposal:

Build and extend scalable concurrent algorithms using a monad with shared-state *and* message-passing operations

Design

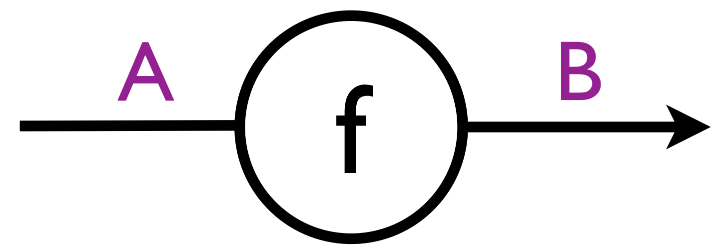
Reagents are (first) arrows

Lambda abstraction:

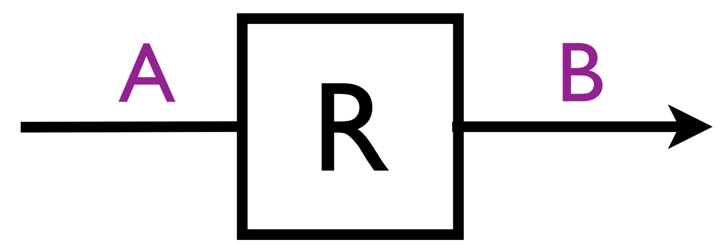


Reagents are (first) arrows

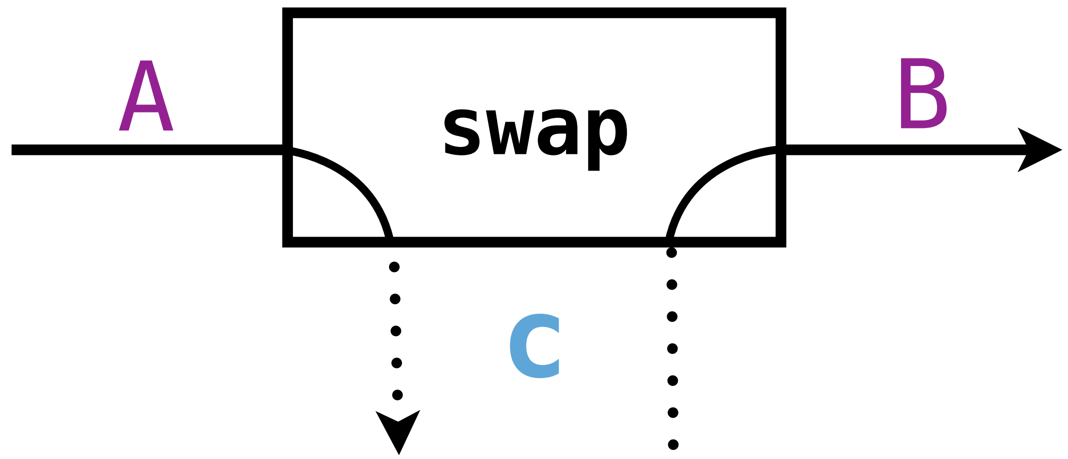
Lambda abstraction:



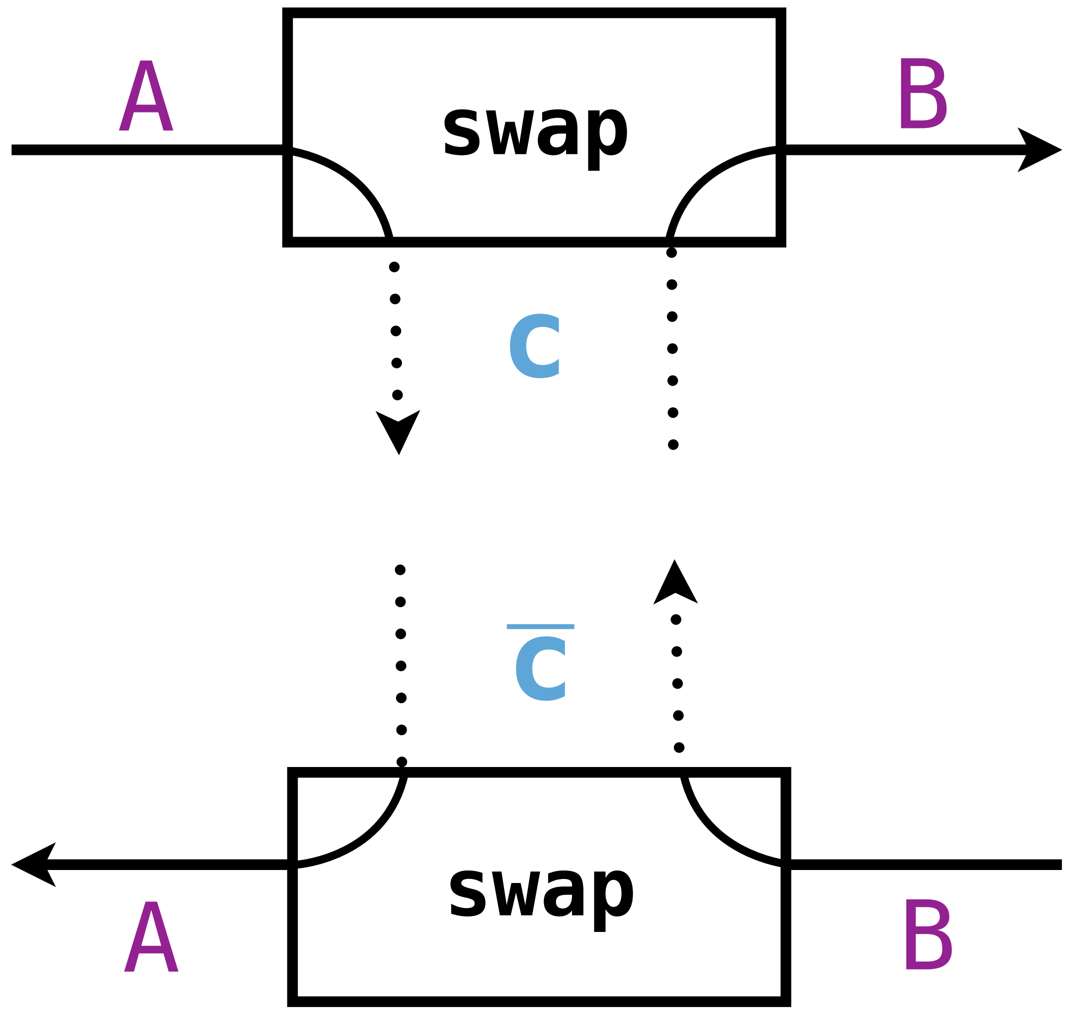
Reagent abstraction:



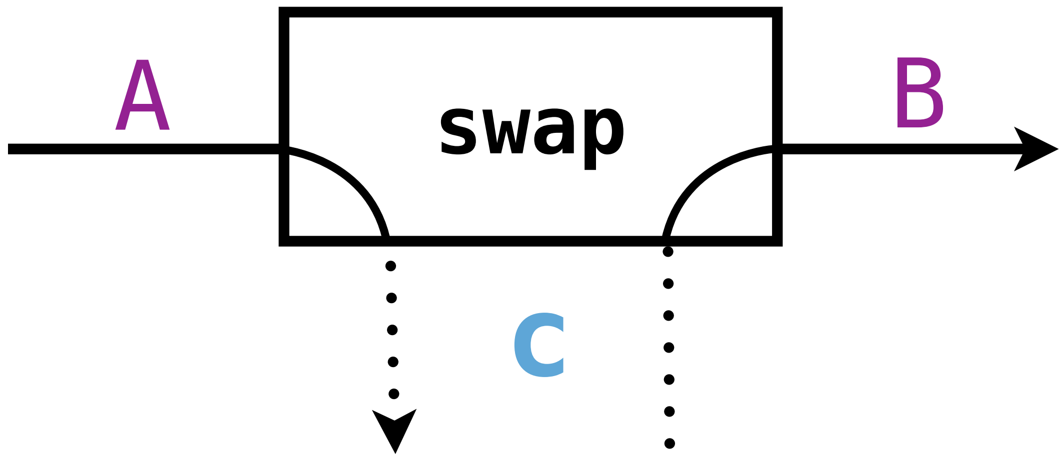
`c: Chan[A,B]`



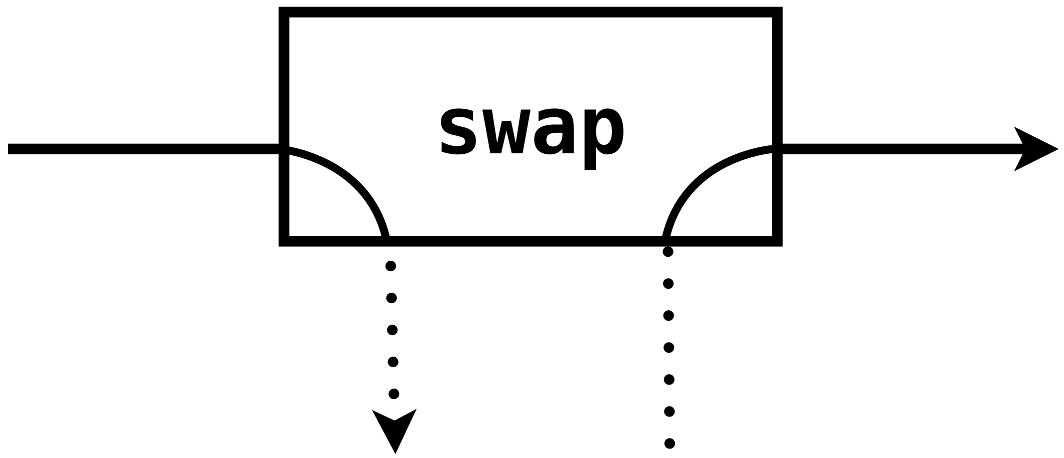
`c: Chan[A,B]`



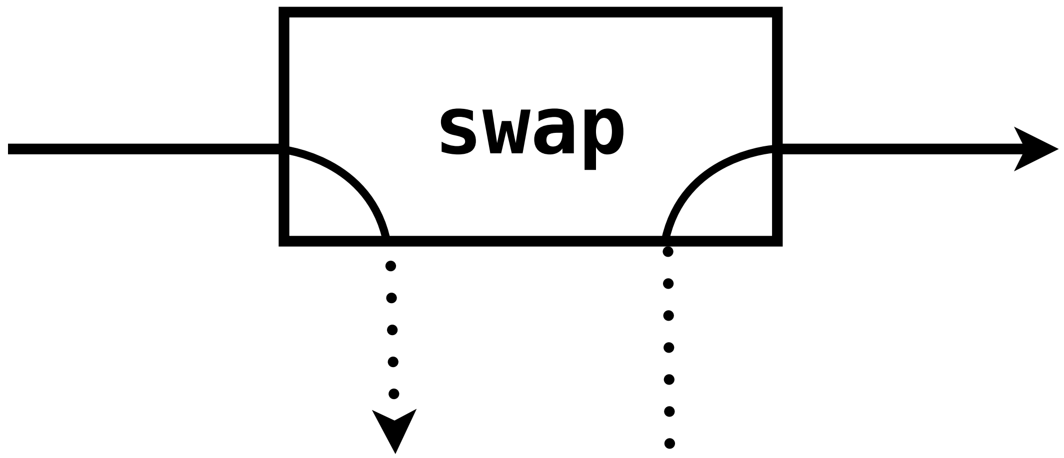
`c: Chan[A,B]`



Message passing

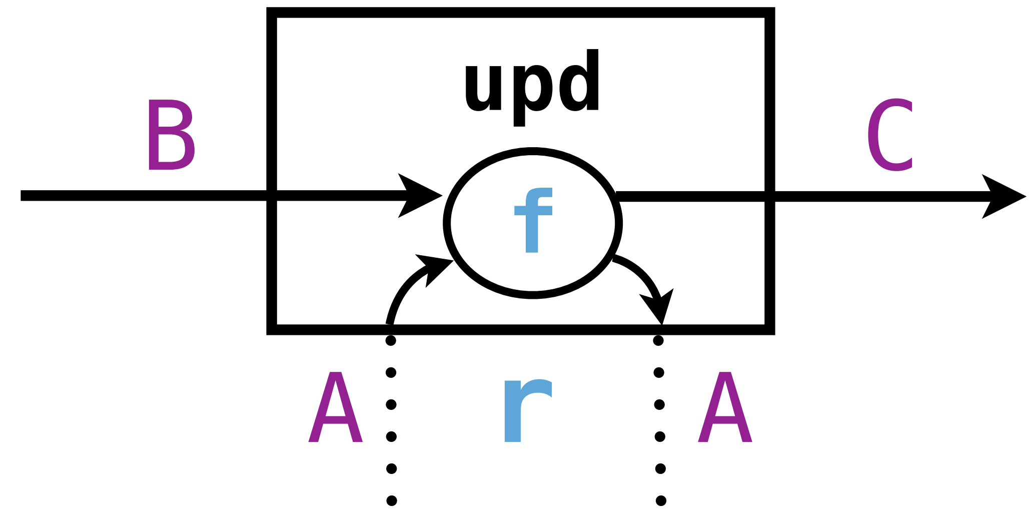


Message passing

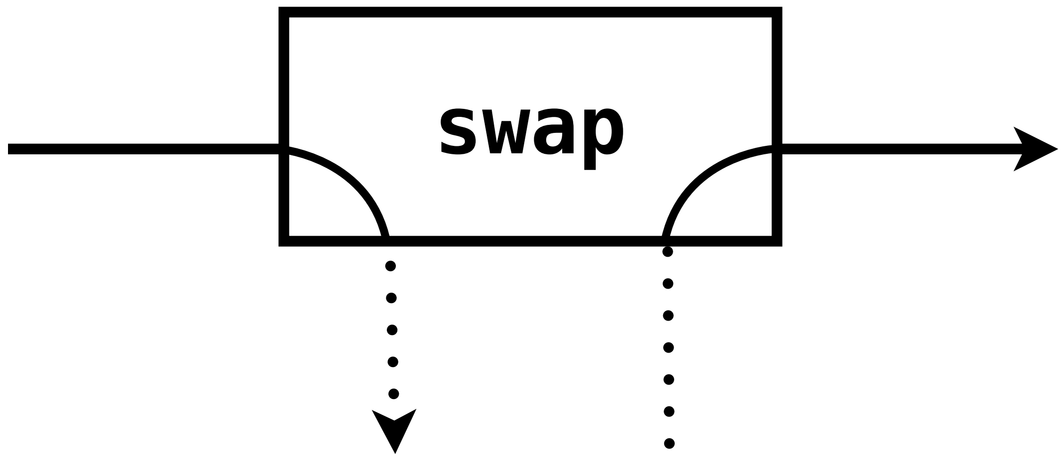


`r: Ref[A]`

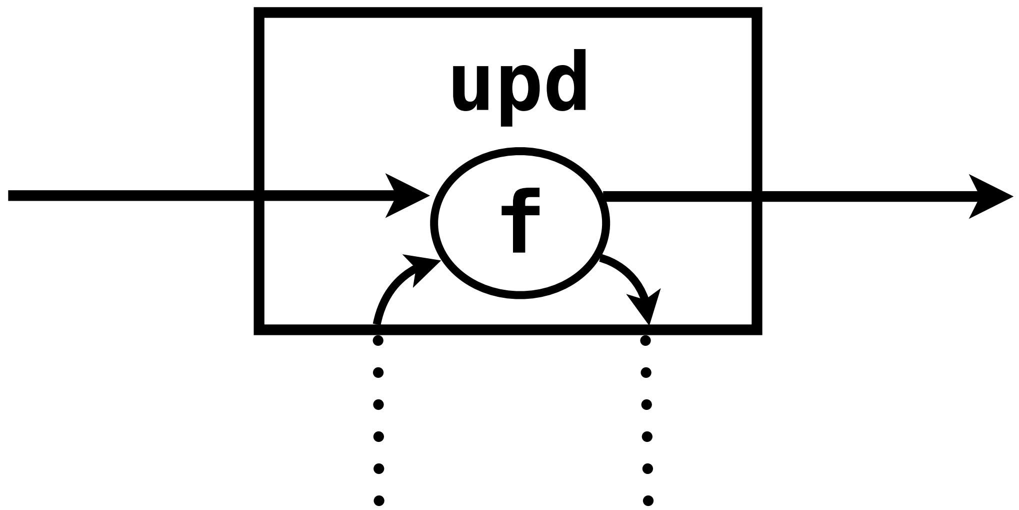
`f: (A,B)→(A,C)`



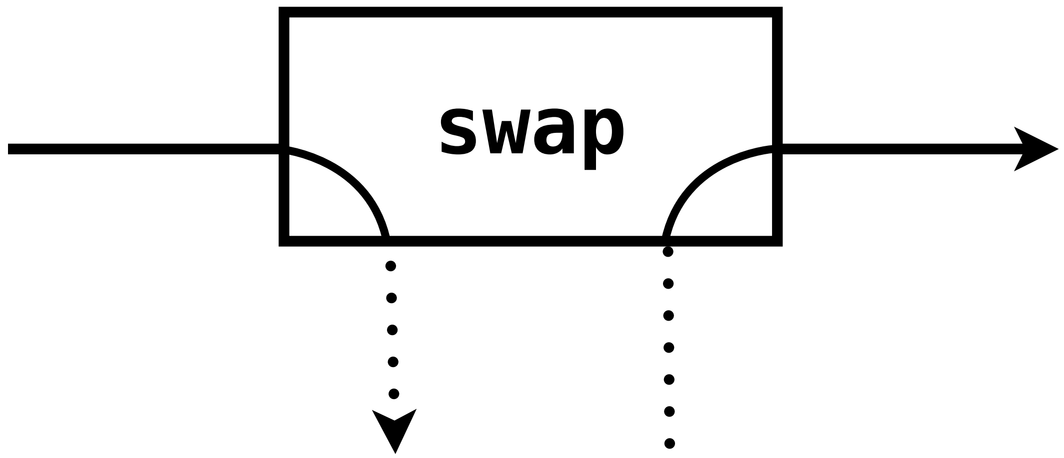
Message passing



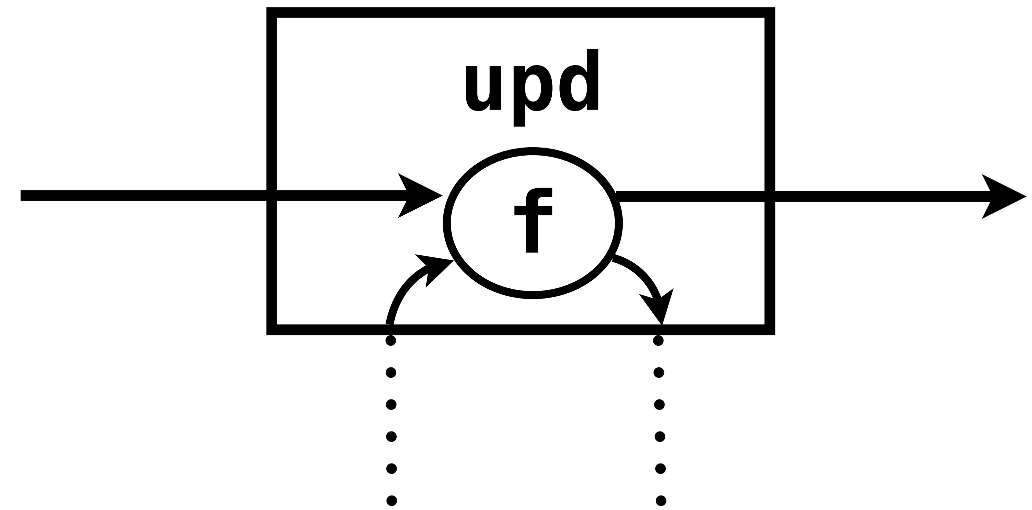
Shared state

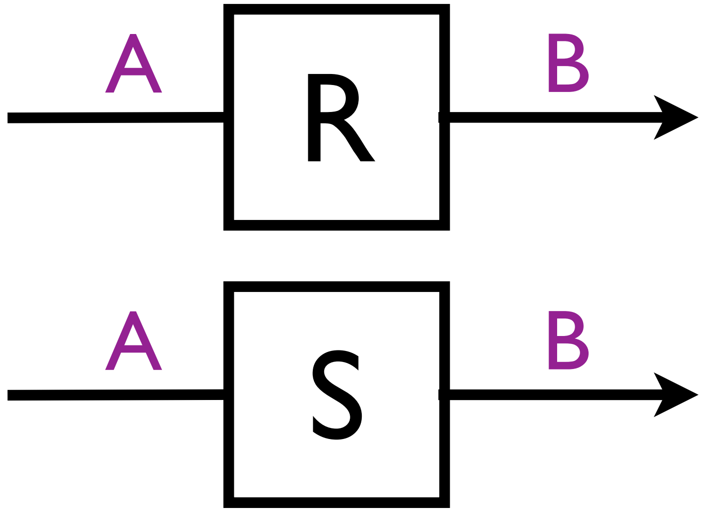


Message passing

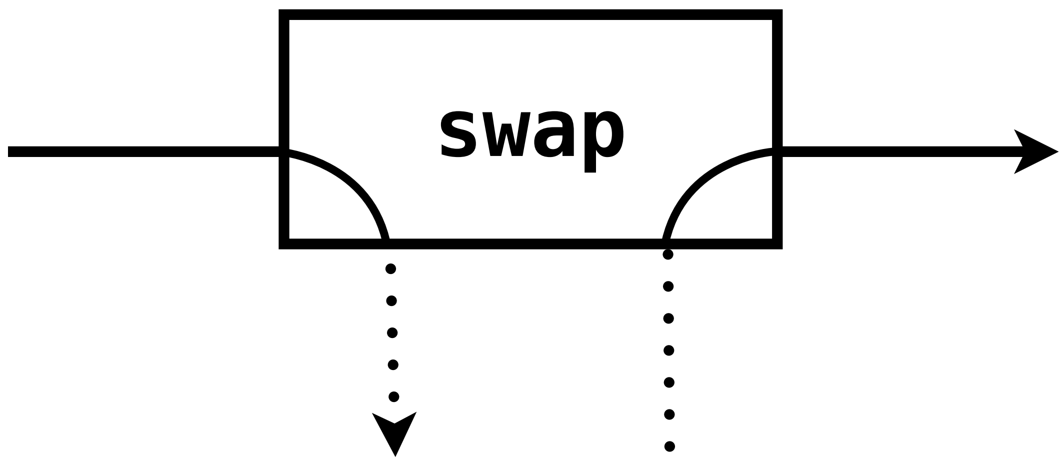


Shared state

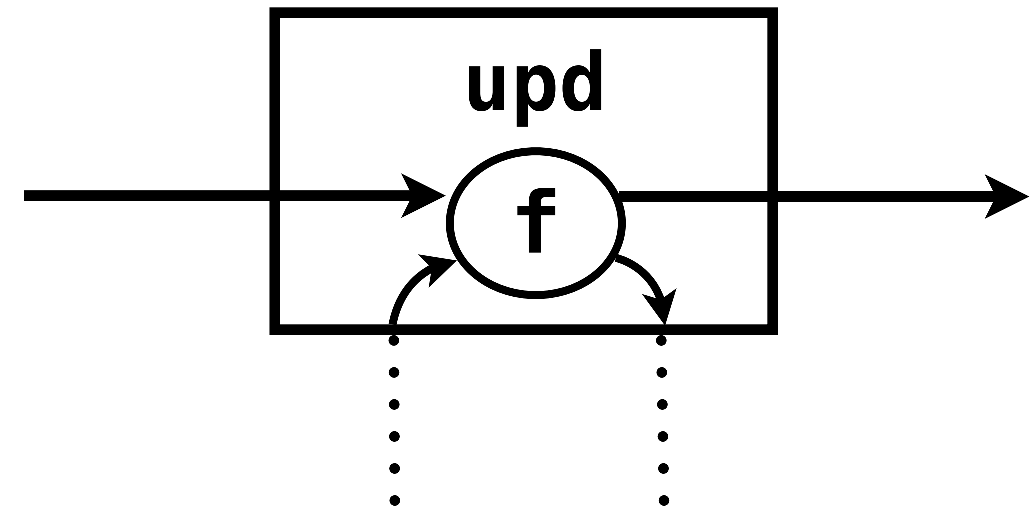


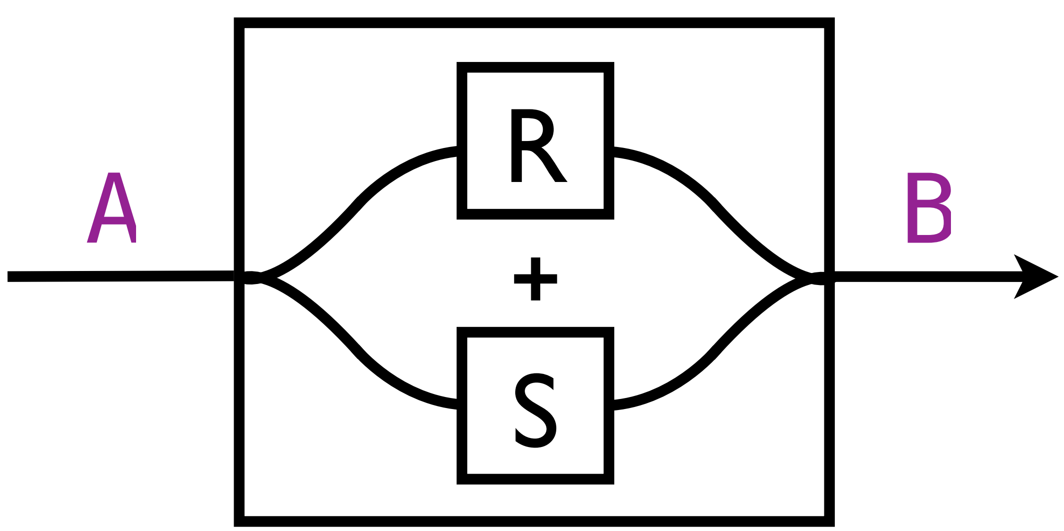


Message passing

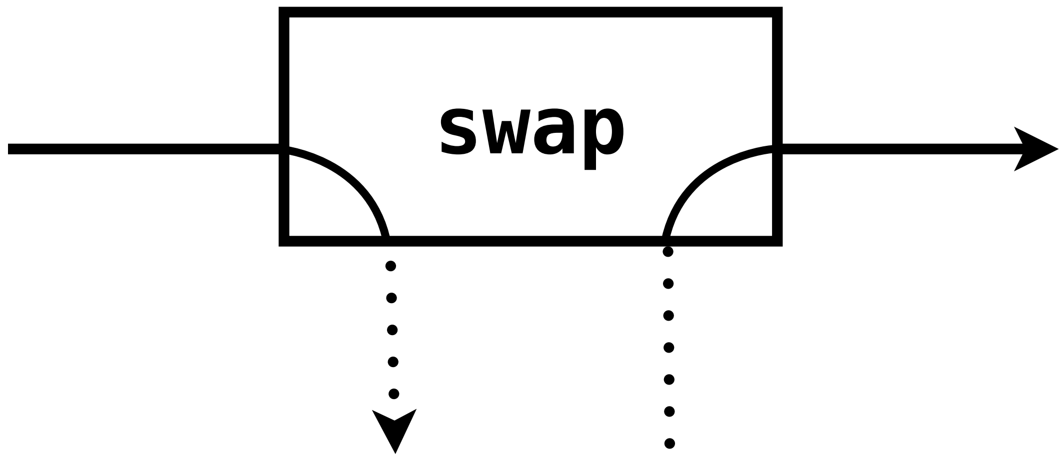


Shared state

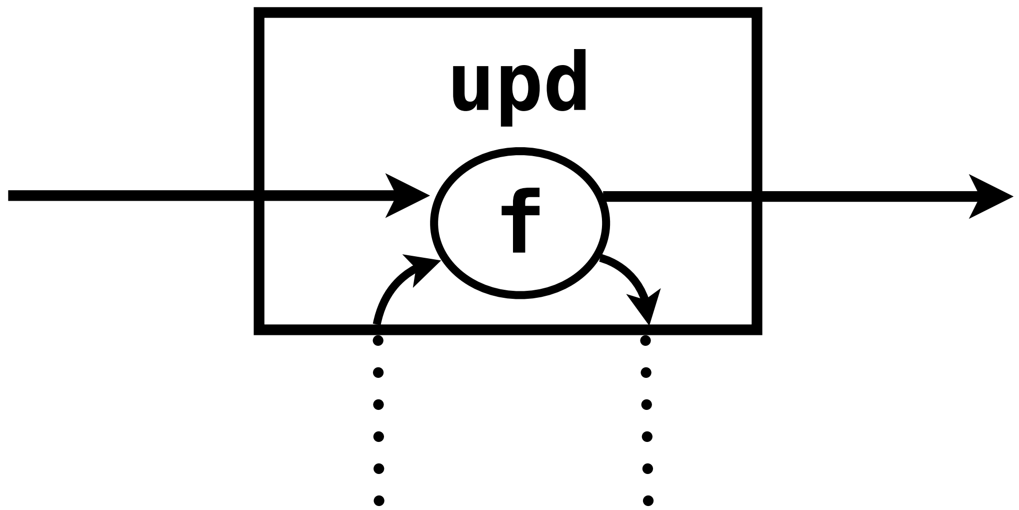




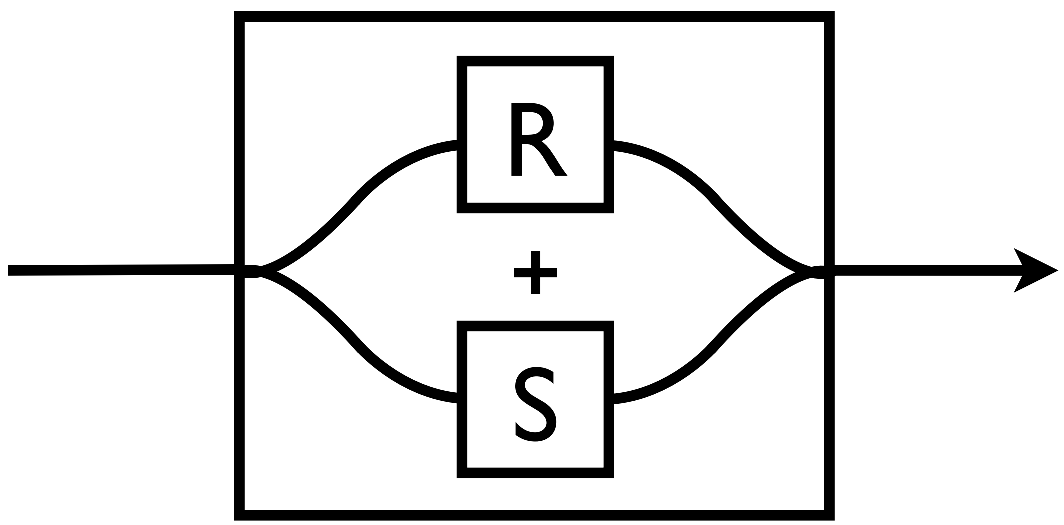
Message passing



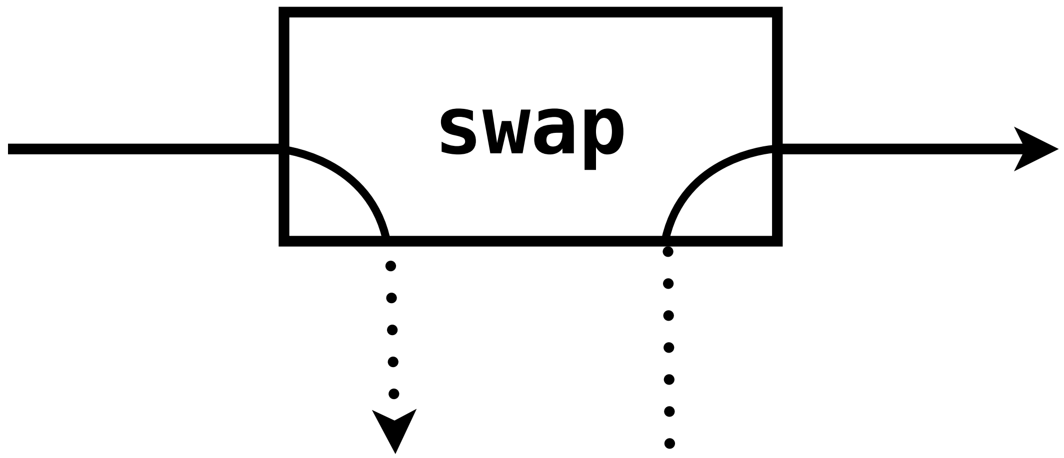
Shared state



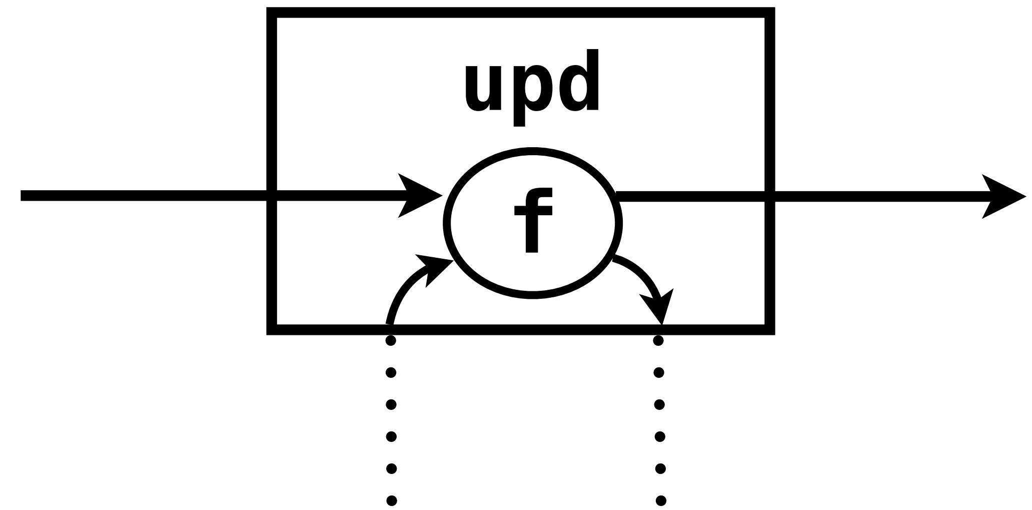
Disjunction



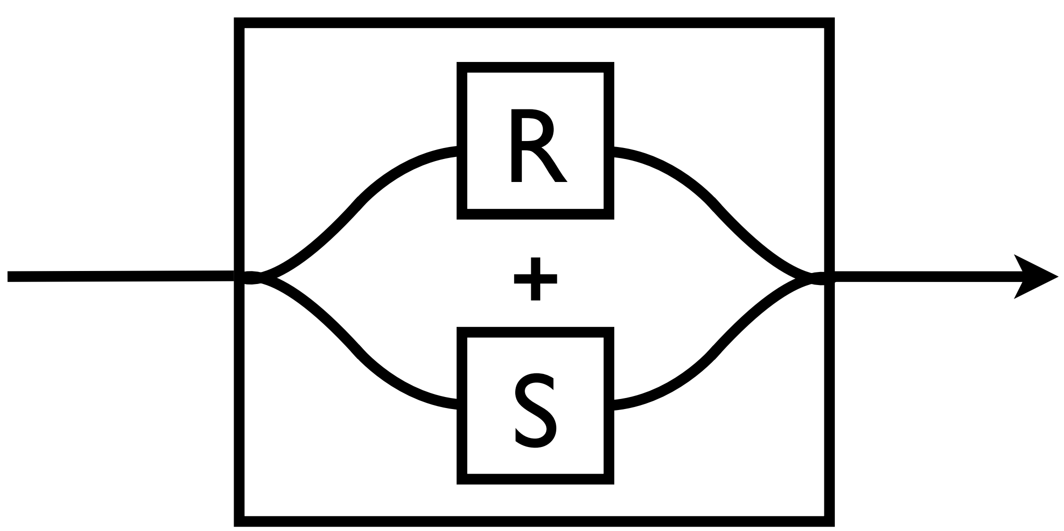
Message passing

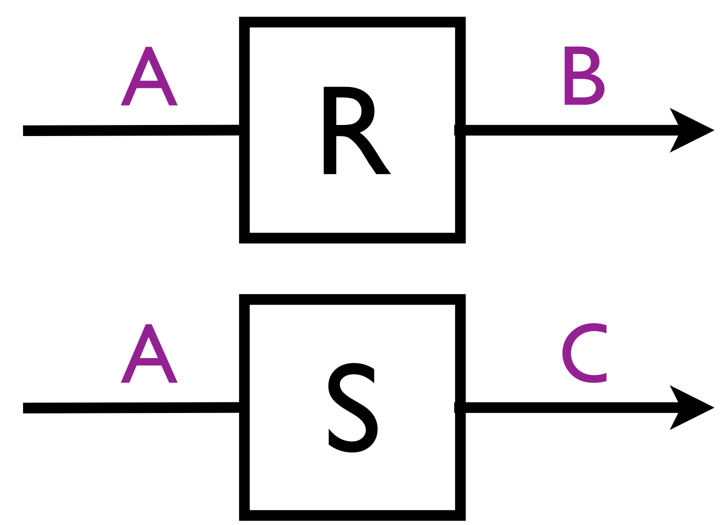


Shared state

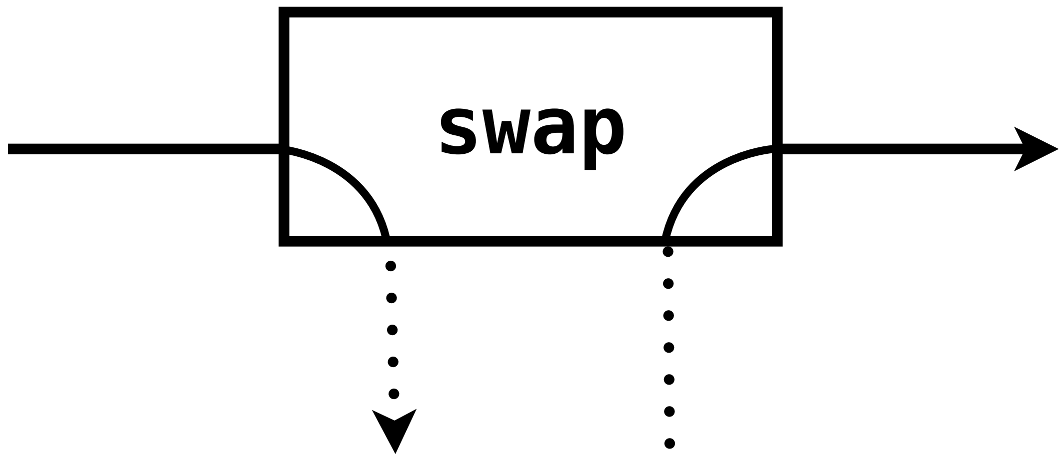


Disjunction

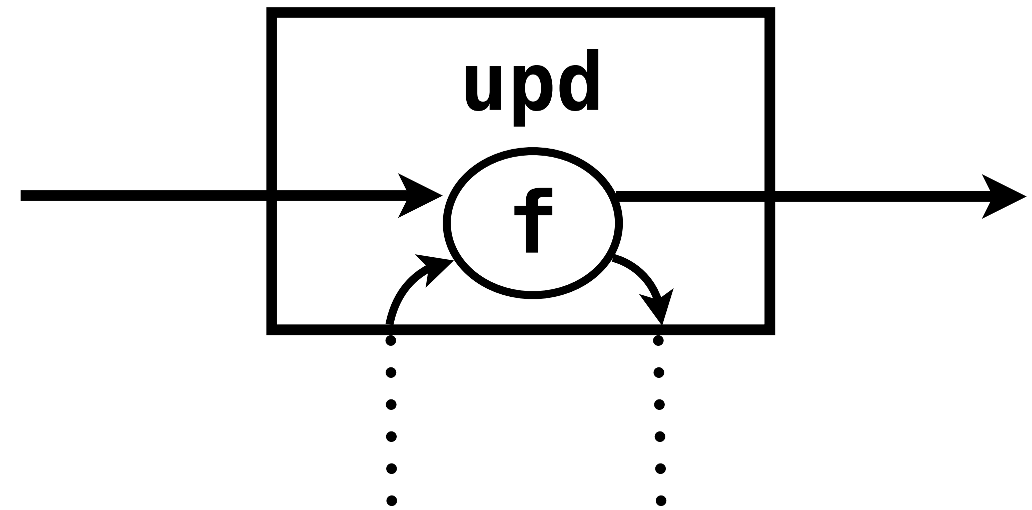




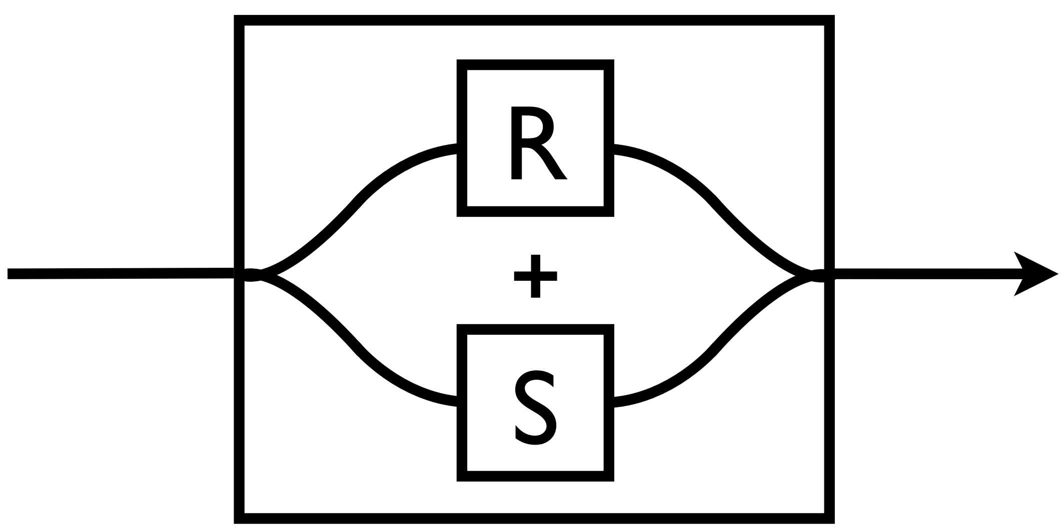
Message passing

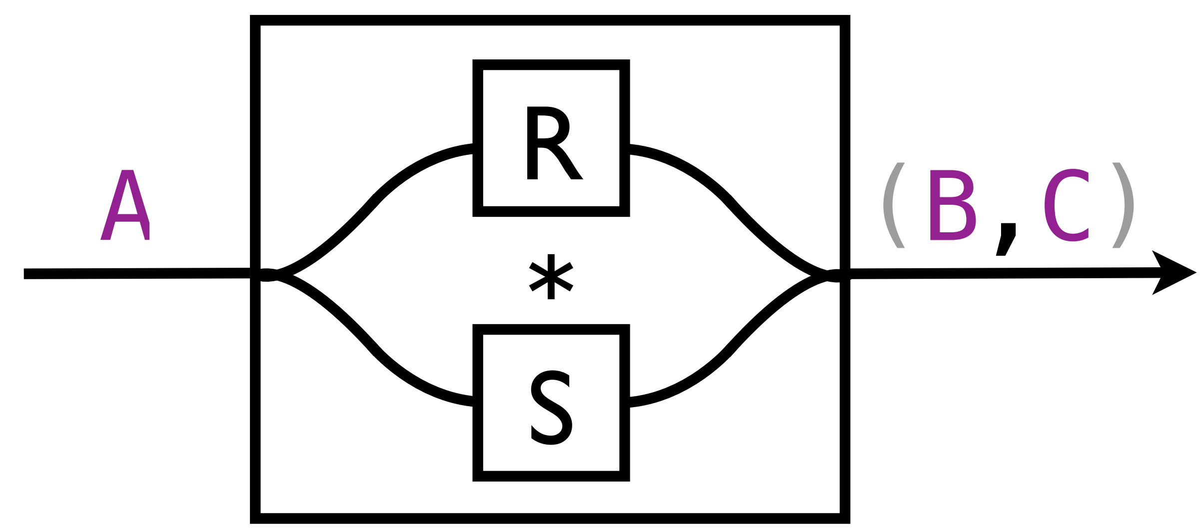


Shared state

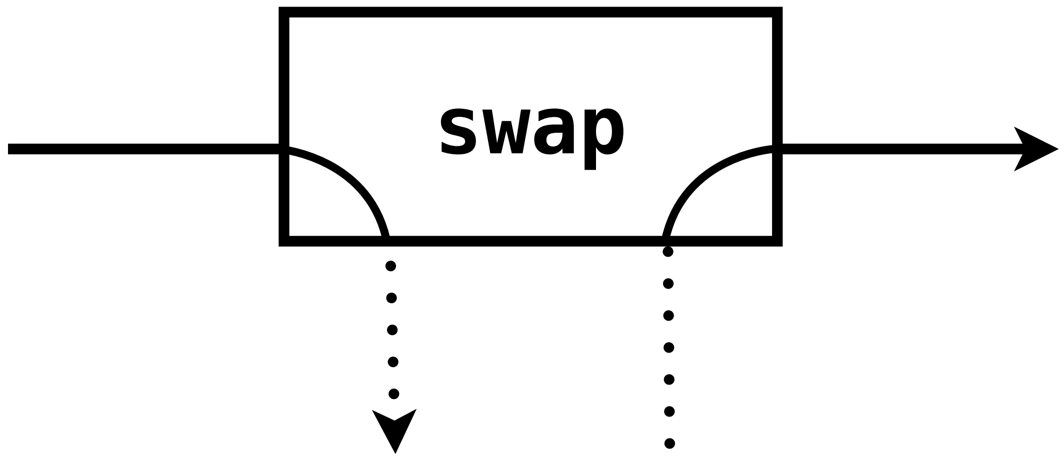


Disjunction

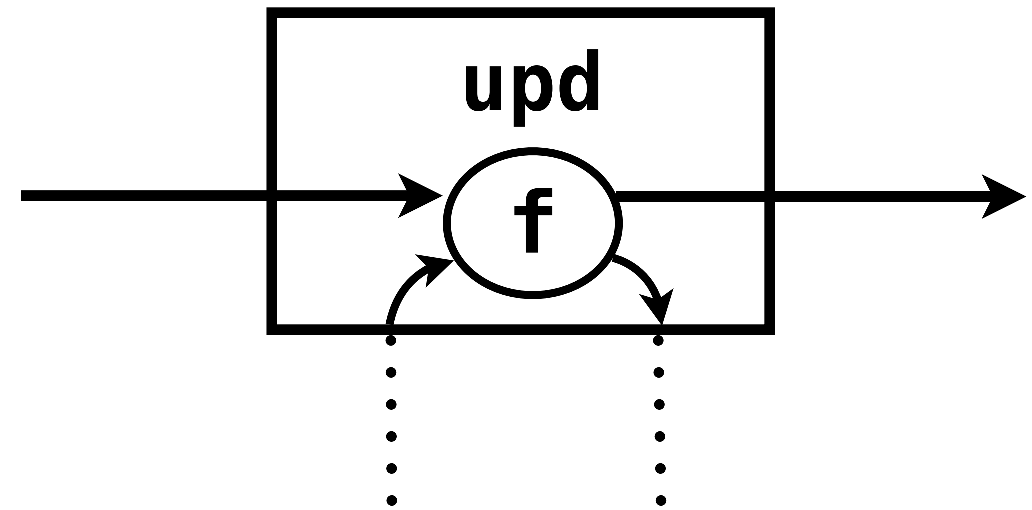




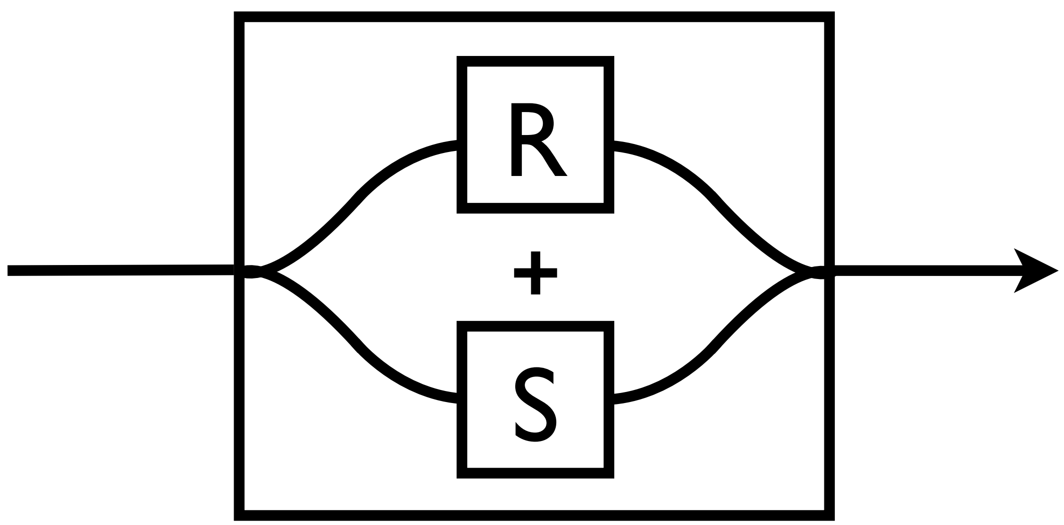
Message passing



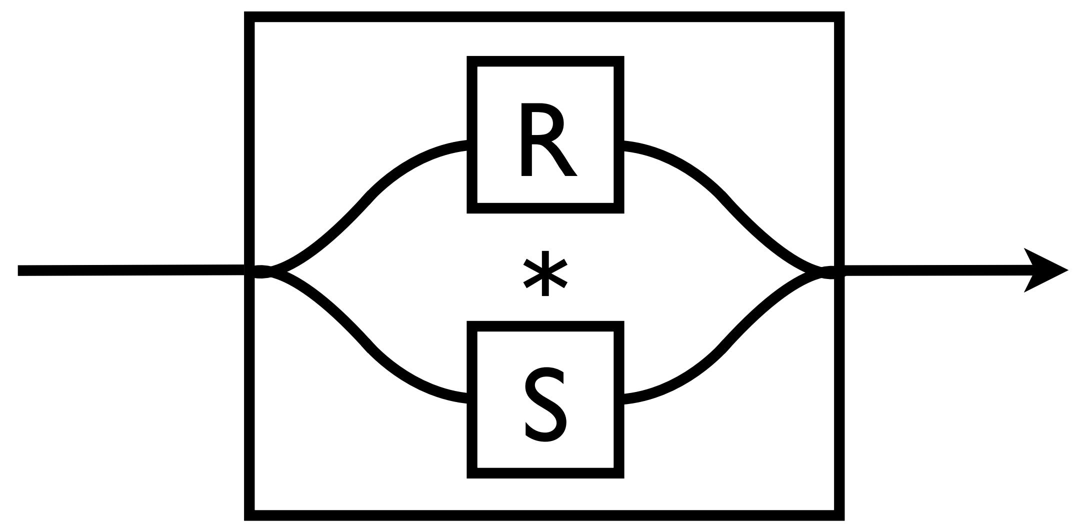
Shared state



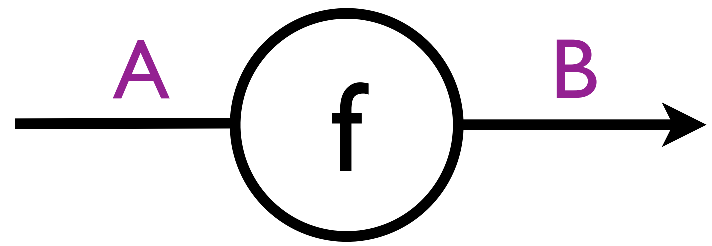
Disjunction



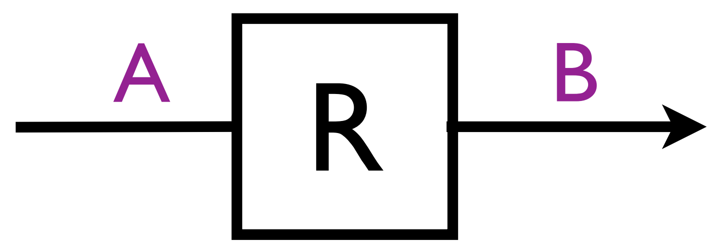
Conjunction



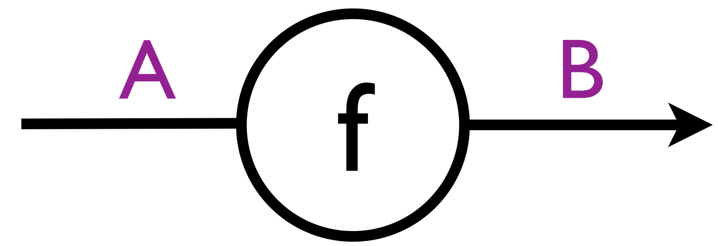
Lambda abstraction:



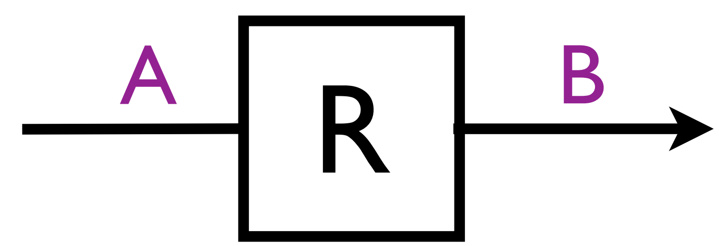
Reagent abstraction:



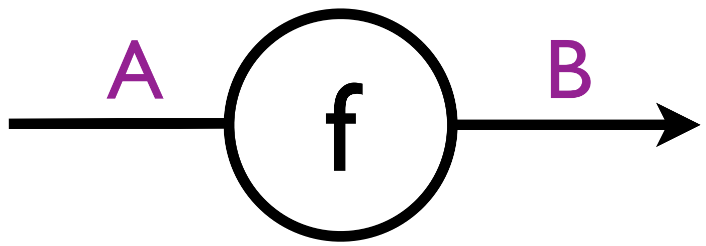
Lambda abstraction:



Reagent abstraction:



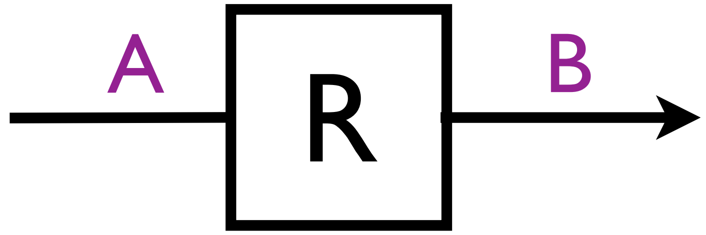
Lambda abstraction:



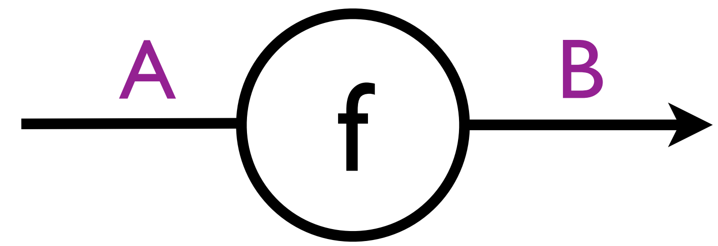
application:

f(a) = b

Reagent abstraction:

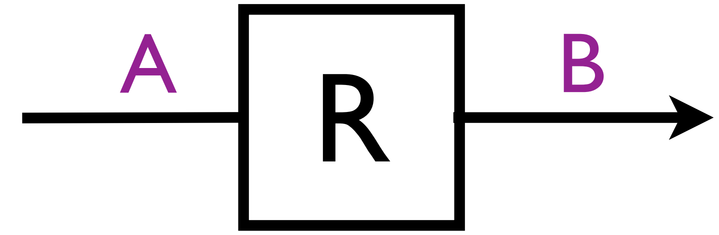


Lambda abstraction:



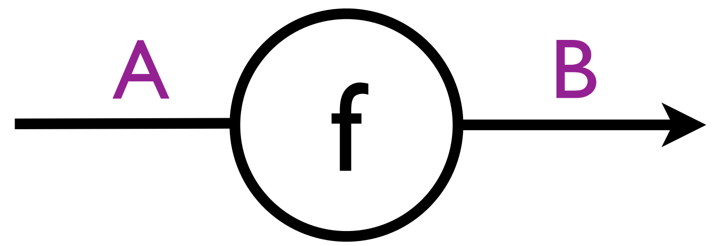
application: $f(a) = b$

Reagent abstraction:



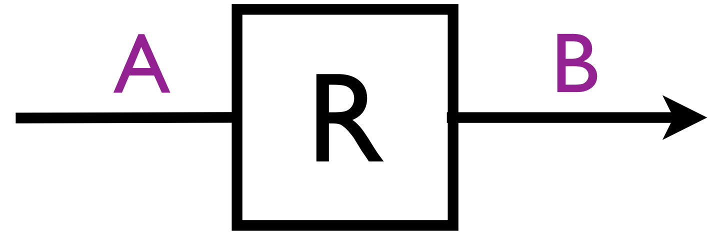
apply as reactant: R ! a = b

Lambda abstraction:



application: f(a) = b

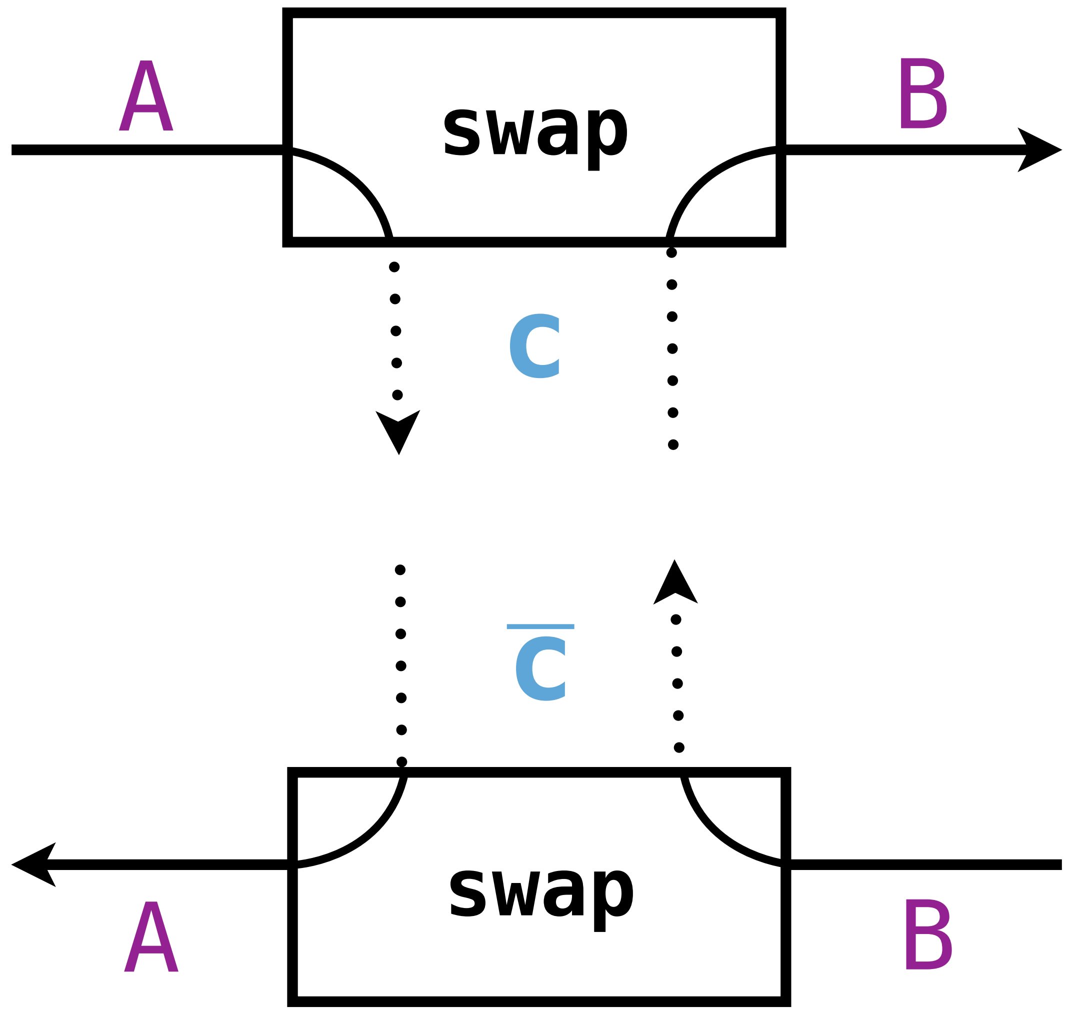
Reagent abstraction:



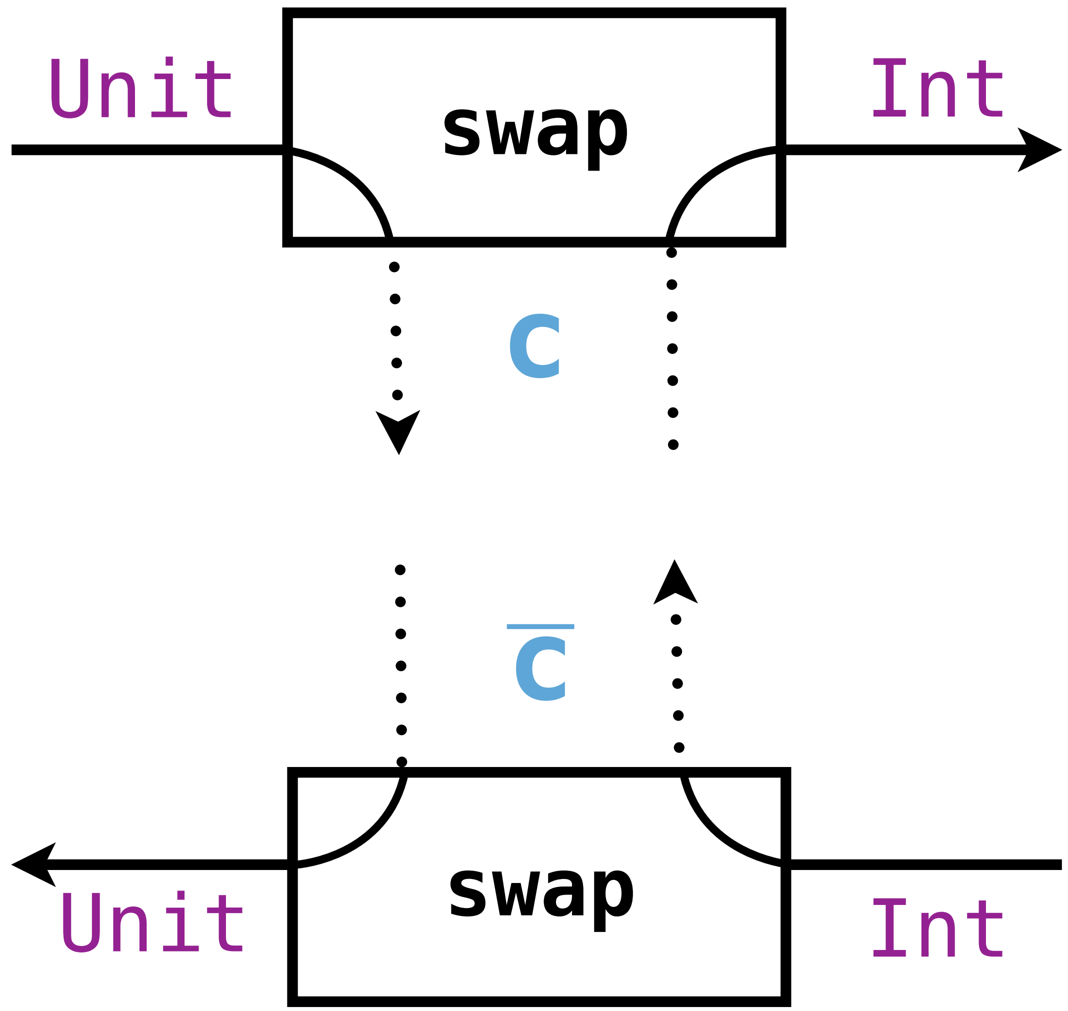
apply as reactant: R ! a = b

apply as catalyst: dissolve(R)

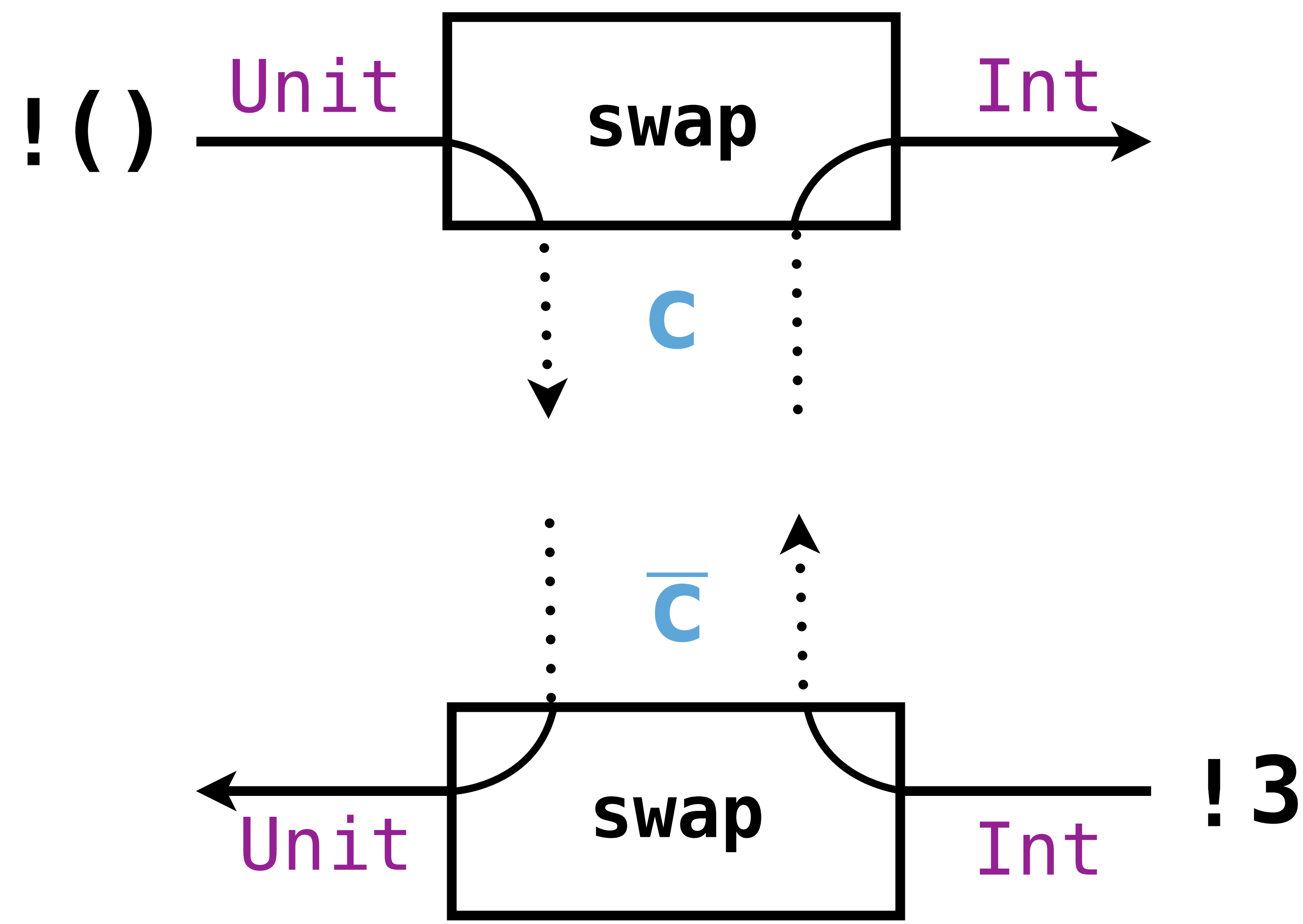
`c: Chan[A,B]`



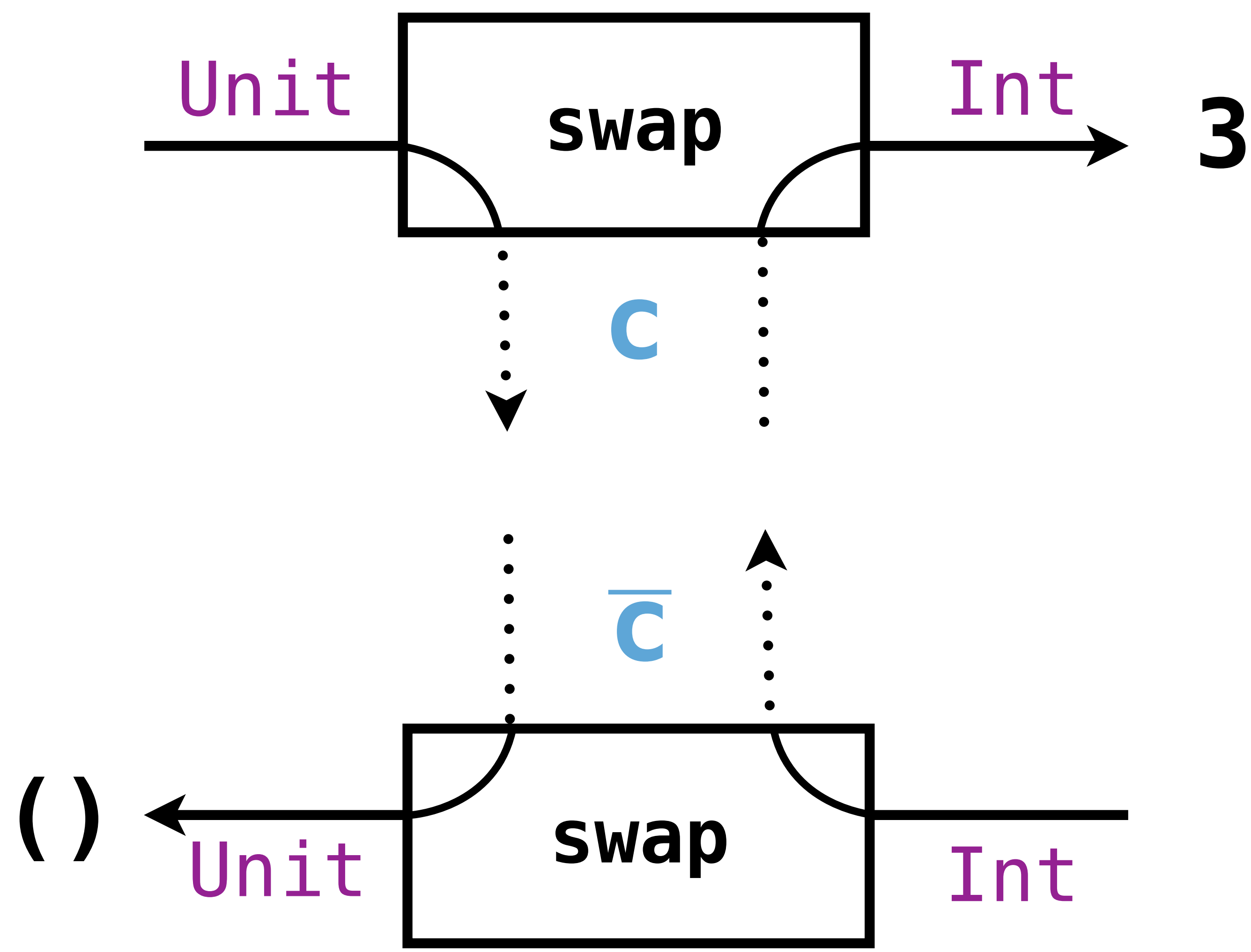
c: Chan[Unit,Int]



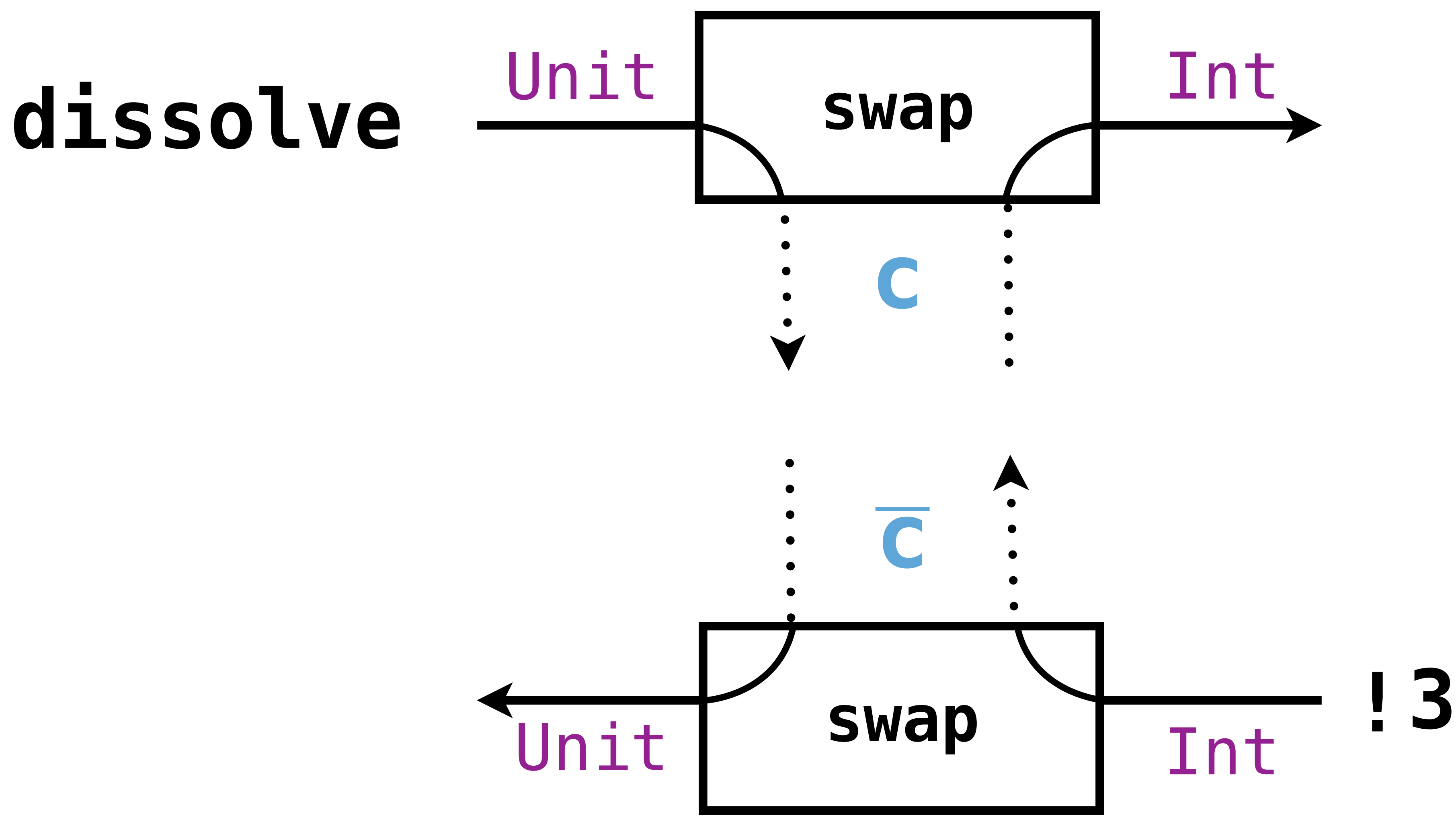
`c: Chan[Unit,Int]`



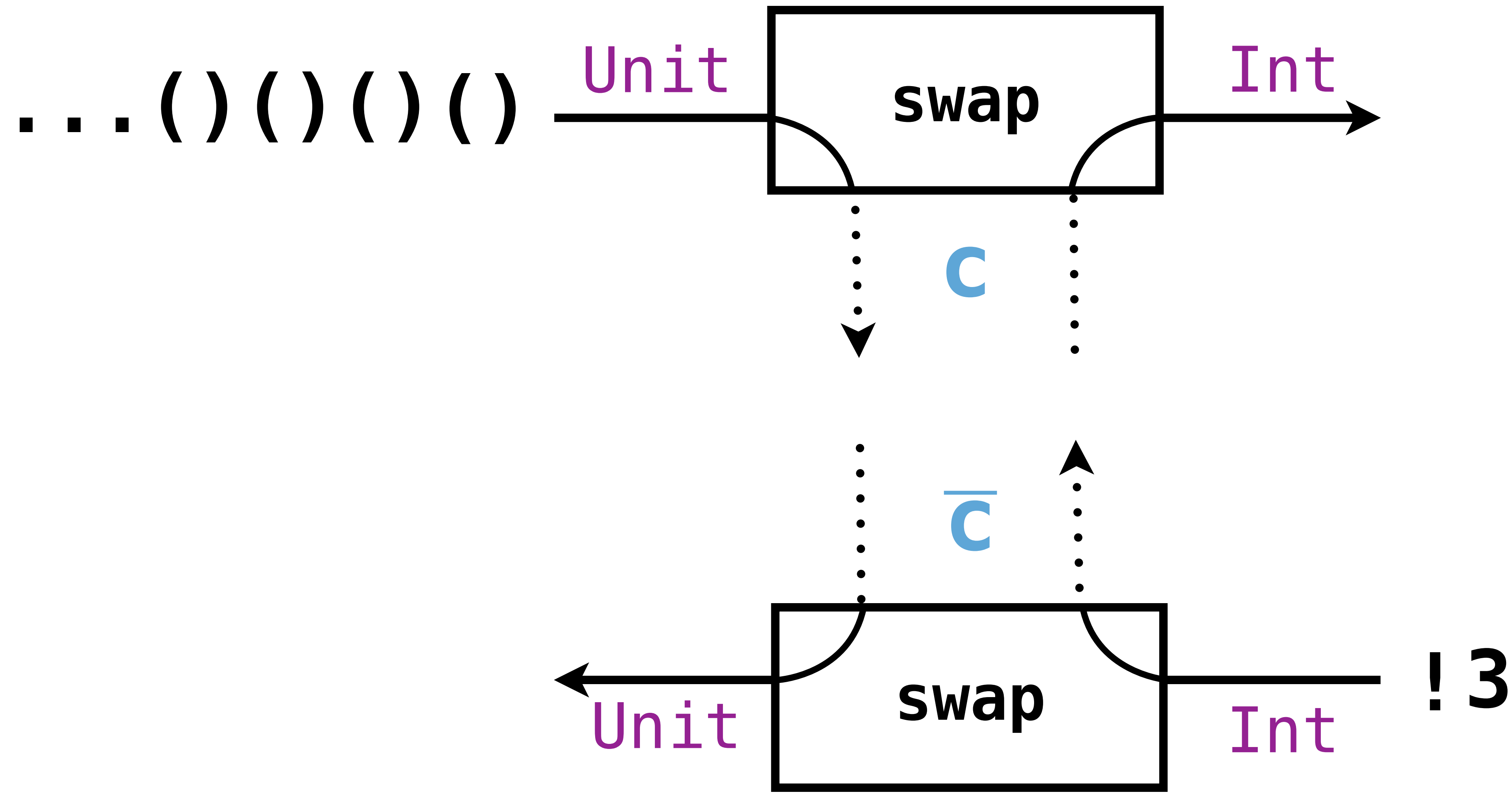
`c: Chan[Unit,Int]`



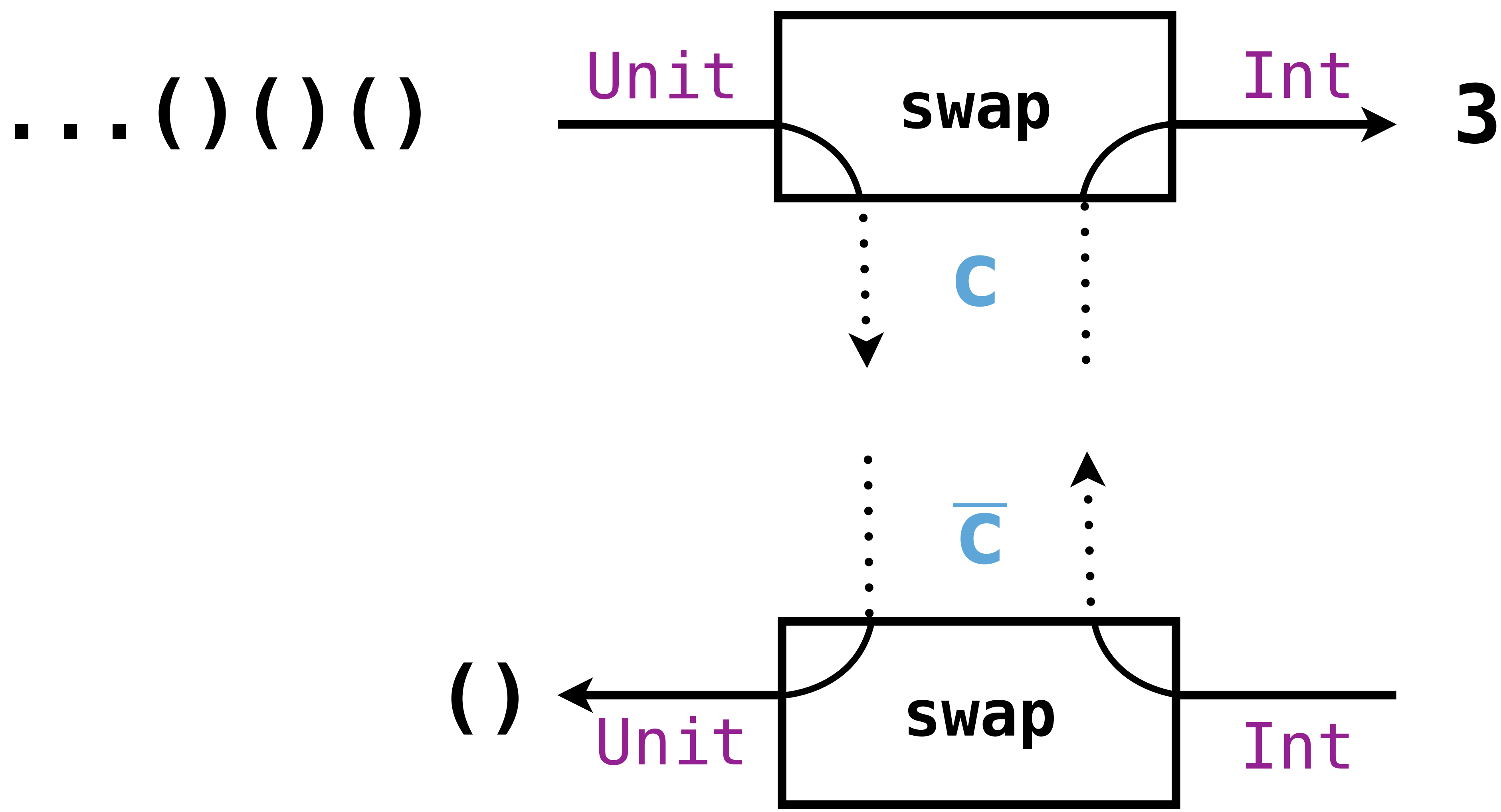
c: Chan[Unit,Int]

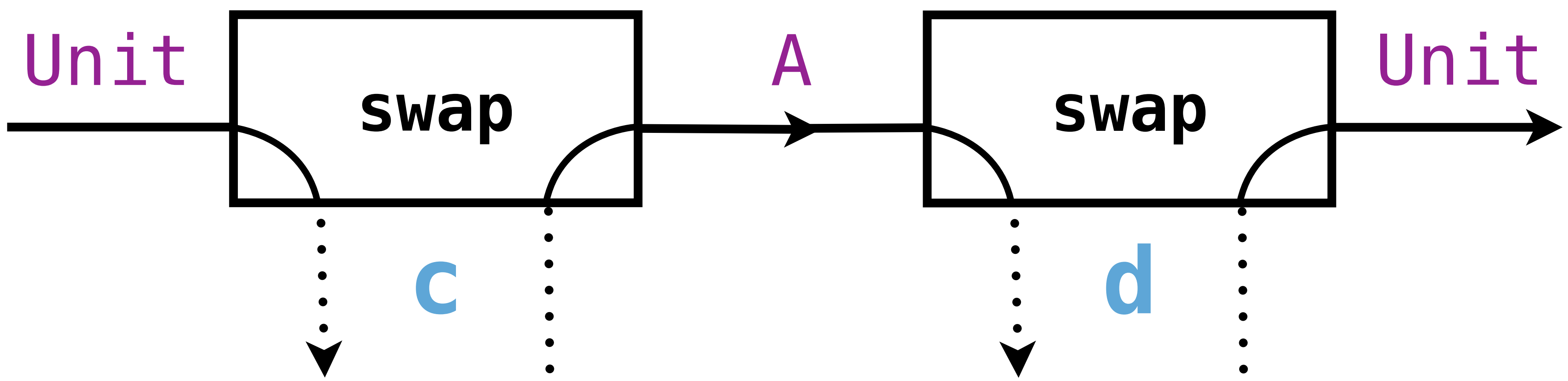


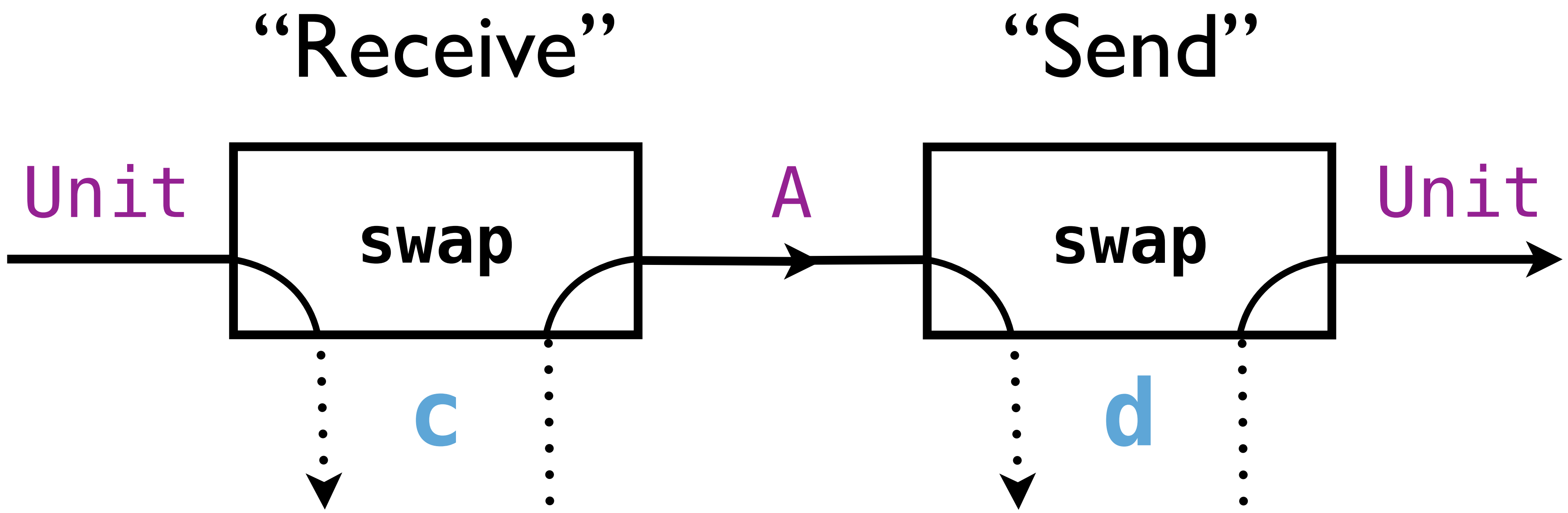
c: Chan[Unit,Int]

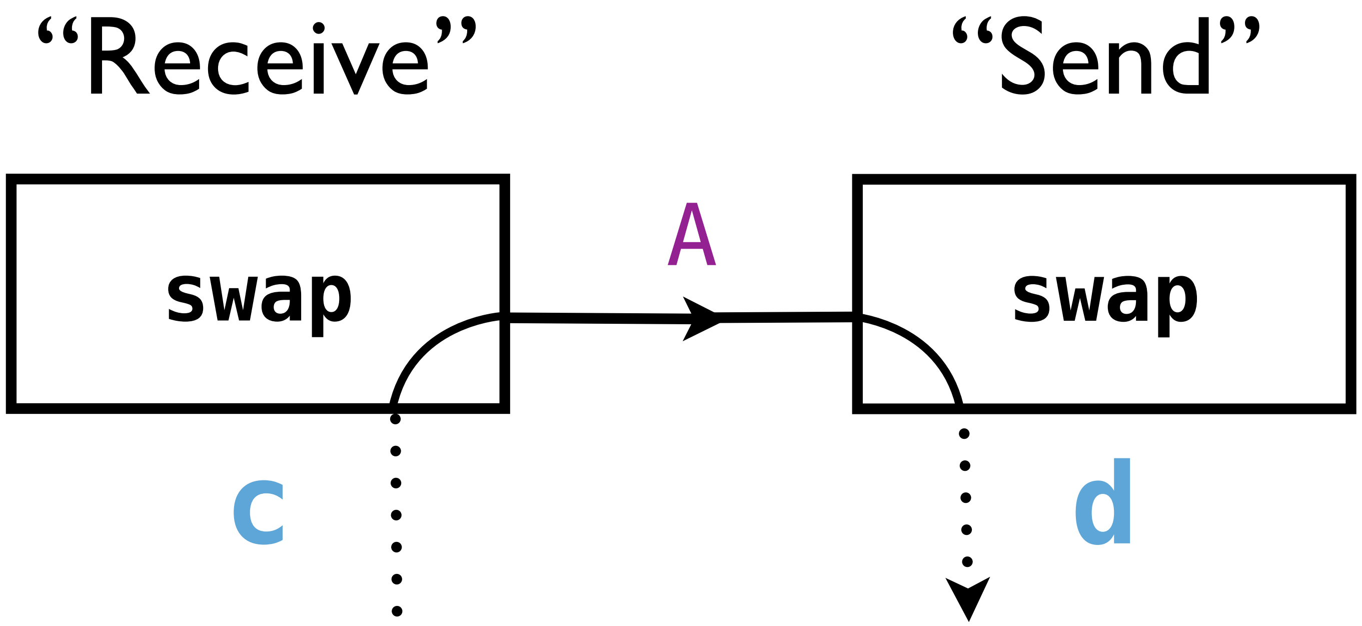


c: Chan[Unit,Int]

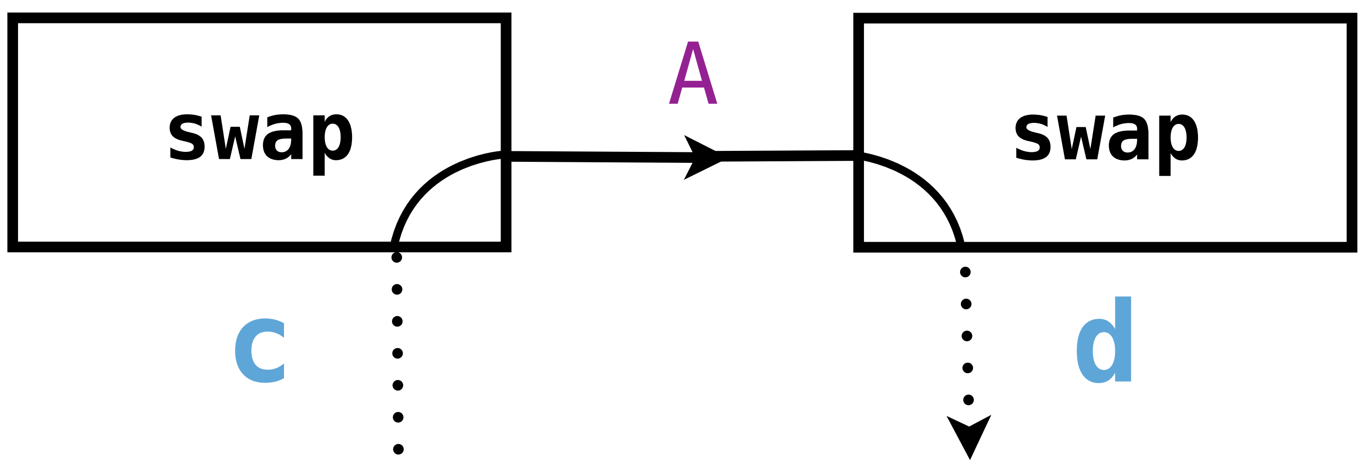



Unit
swap
A
swap
Unit
c
d

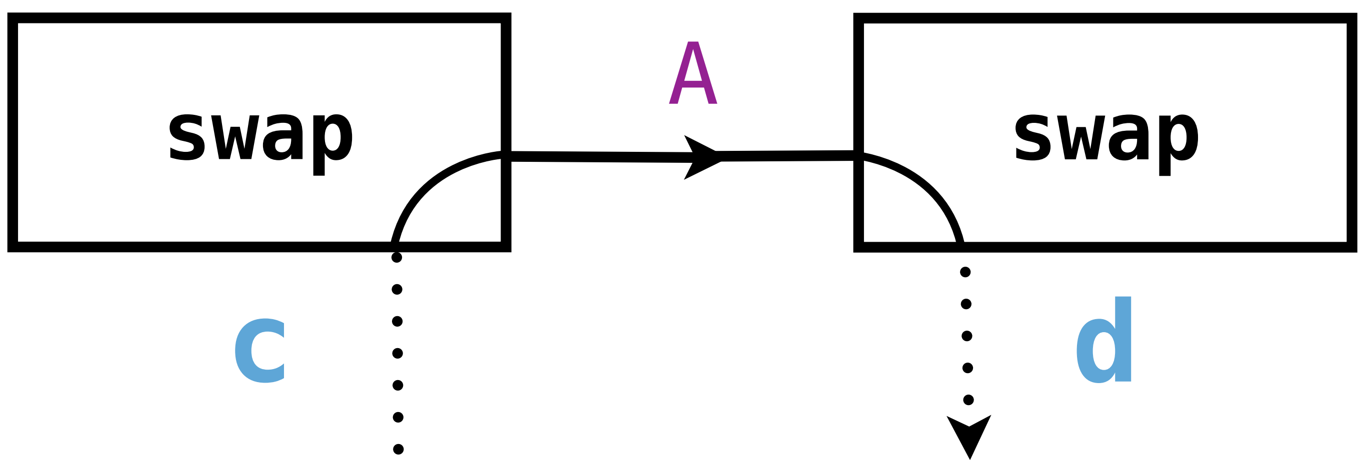

“Receive”
“Send”
Unit
swap
A
swap
Unit
c
d


“Receive”
“Send”
swap
A
swap
c
d

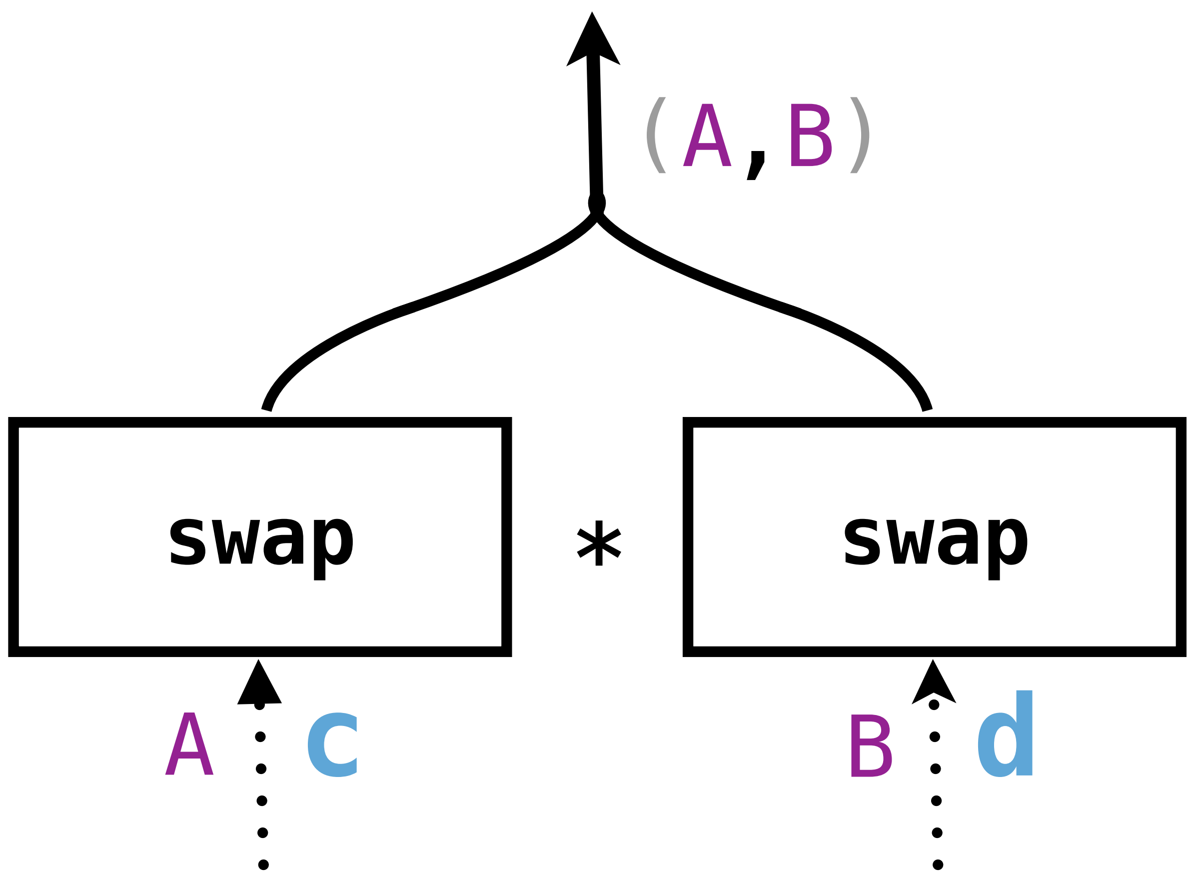
Pipeline catalyst



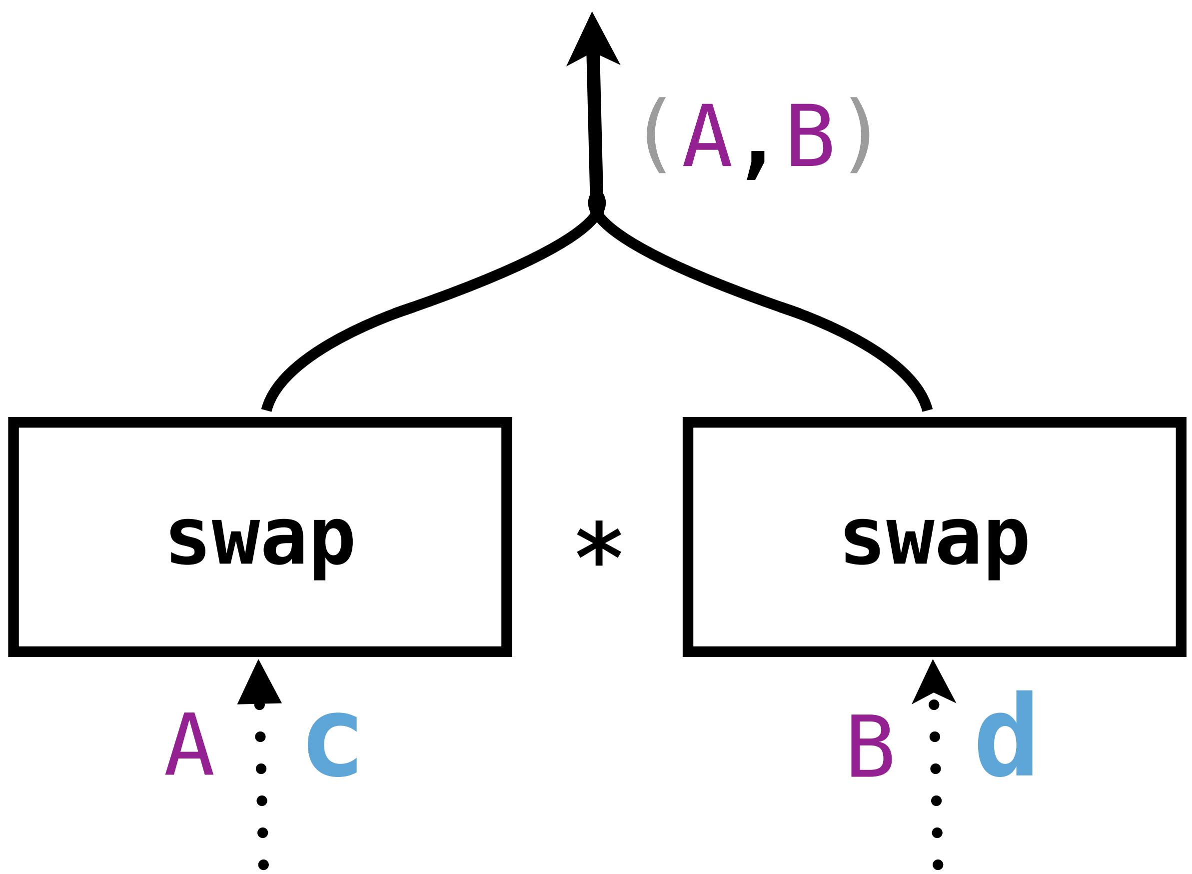
Pipeline catalyst



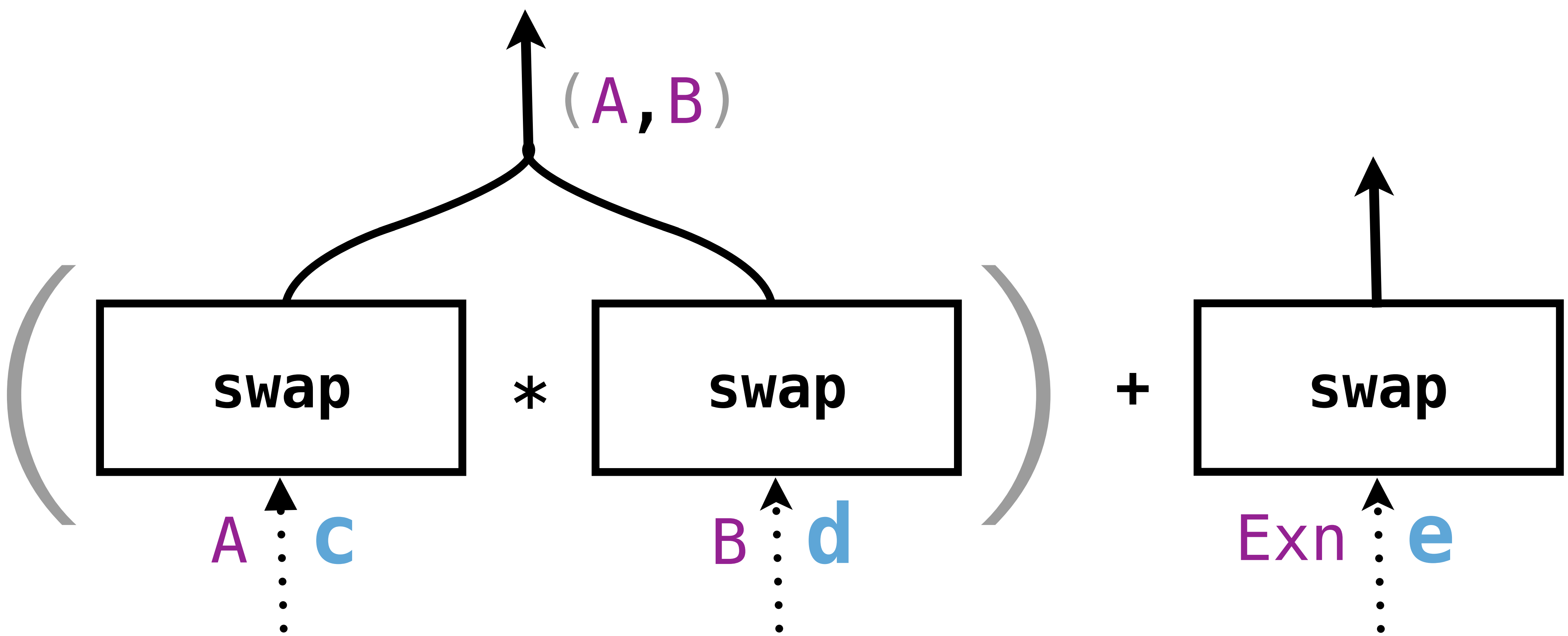
NB: transfer is atomic


(A,B)
swap
*
swap
A
c
B
d

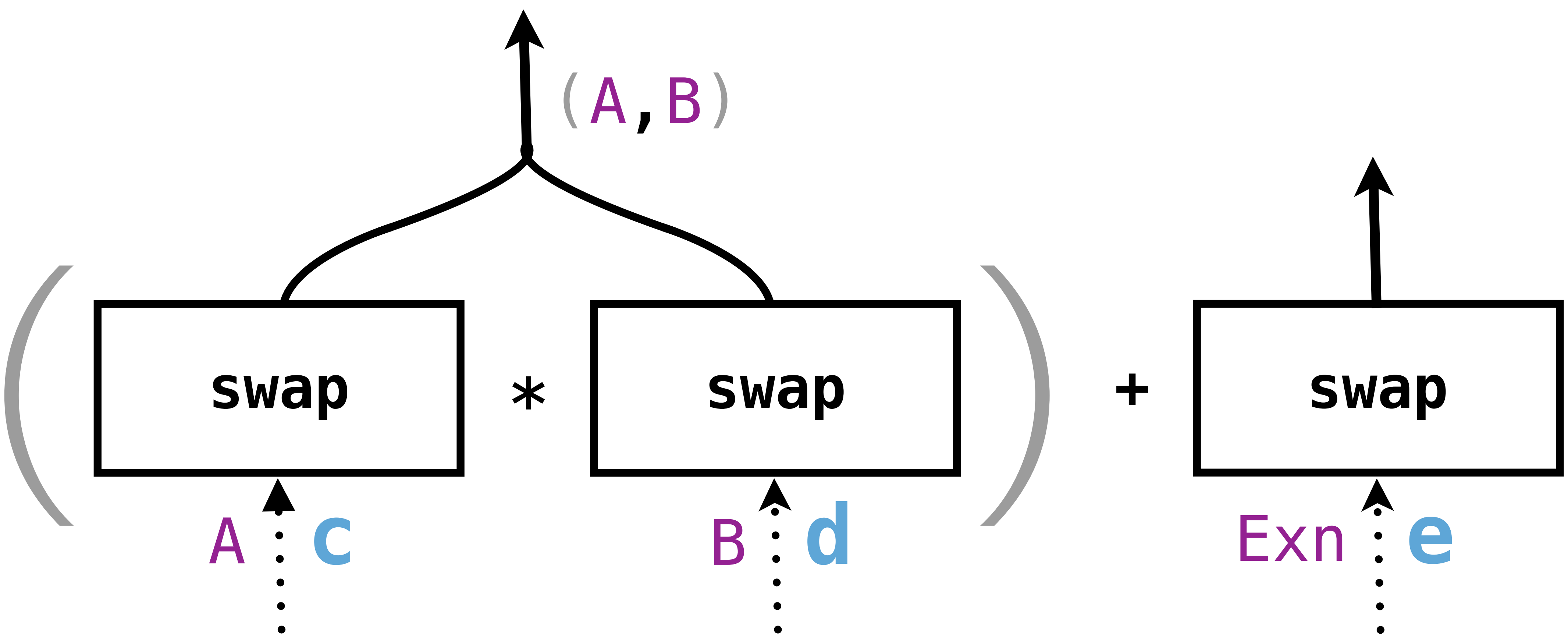
2-way join



2-way join



Abortable 2-way join



Join *Calculus*

$$c_1(x_1) \mathbin{\&} \cdots \mathbin{\&} c_n(x_n) \Rightarrow e$$

Join *Calculus*

$$c_1(x_1) \mathbin{\&} \cdots \mathbin{\&} c_n(x_n) \Rightarrow e$$

becomes

$$(\mathtt{swap}\ c_1 * \cdots * \mathtt{swap}\ c_n)$$
$$\mathtt{>>>}\ \mathtt{postCommit}\ e$$

Join *Calculus*

$$c_1(x_1) \mathbin{\&} \cdots \mathbin{\&} c_n(x_n) \Rightarrow e$$

becomes

$$\texttt{dissolve}\left(\begin{array}{c}(\texttt{swap}\ c_1 * \cdots * \texttt{swap}\ c_n) \\ \texttt{>>> postCommit e}\end{array}\right)$$

```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    : A  ↣ () = upd(head)(cons)
  val tryPop : ()  ↣ A? = upd(head) {
    case (x :: xs) => (xs,  Some(x))
    case Nil        => (Nil, None)
  }
}
```

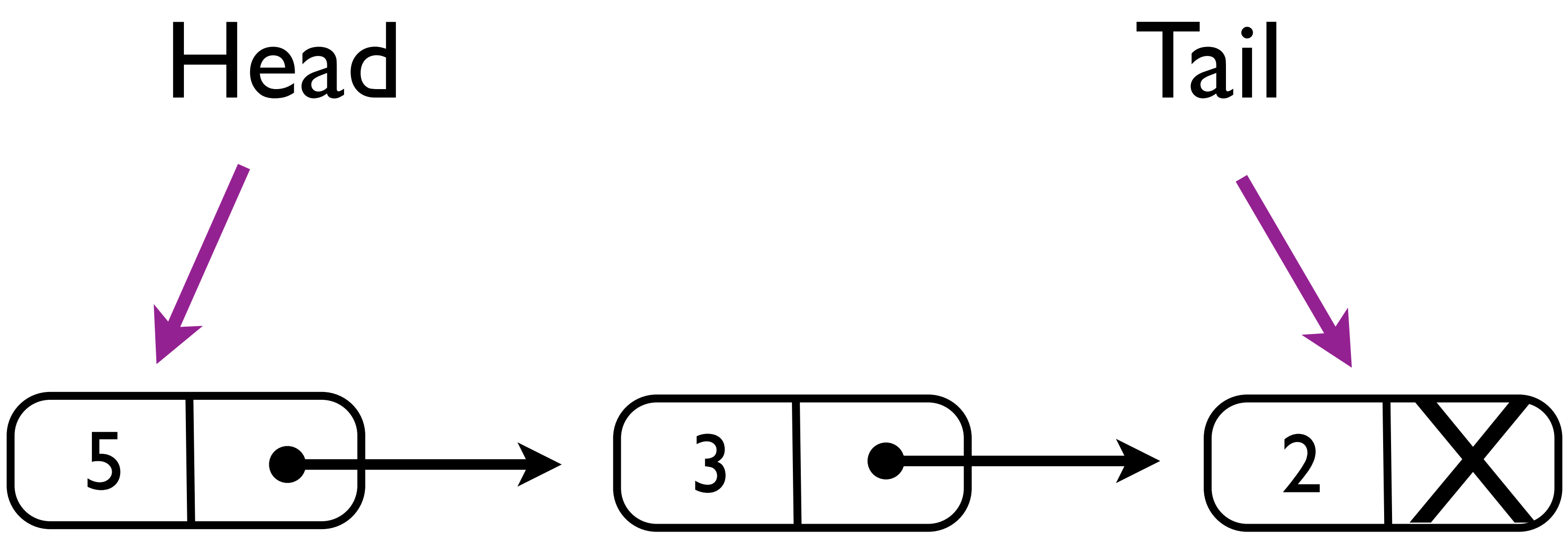
```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    : A  ↣ () = upd(head)(cons)
  val tryPop : () ↣ A? = upd(head) {
    case (x :: xs) => (xs,  Some(x))
    case Nil         => (Nil, None)
  }
  val pop     : () ↣ A  = upd(head) {
    case (x :: xs) => (xs,  x)
  }
}
```

```scala
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop     = upd(head)(split)
}
```

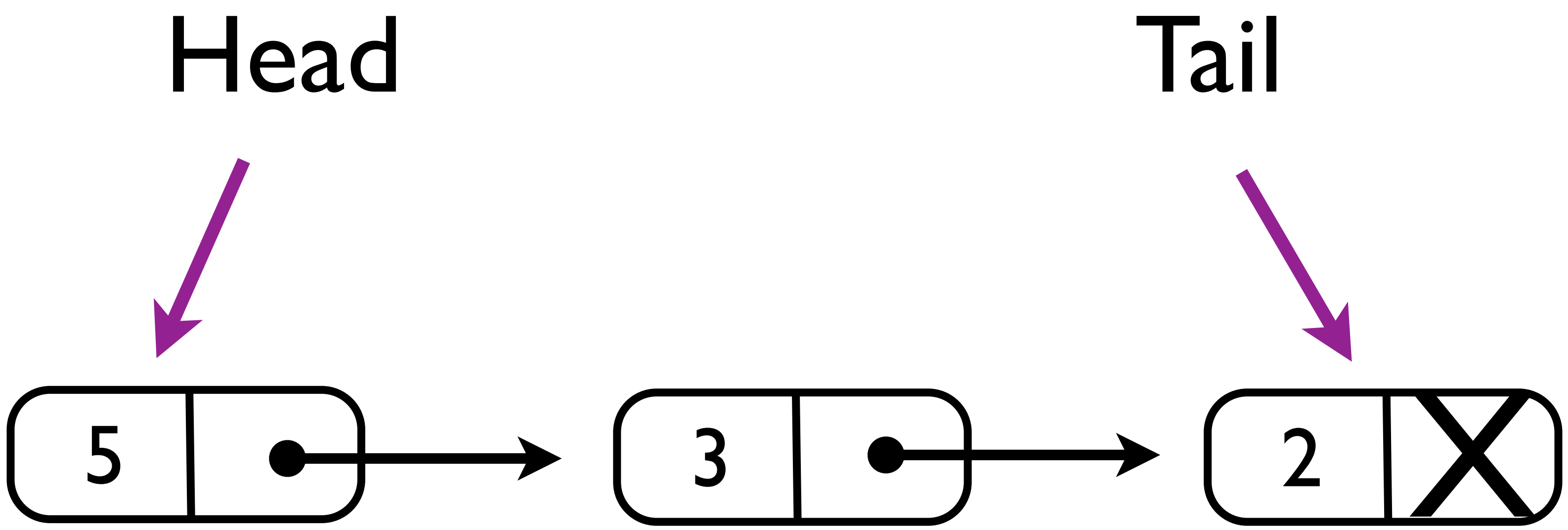
```scala
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push    = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop     = upd(head)(split)
}

class EliminationStack [A] {
  private val stack = new TreiberStack[A]
  private val (send, recv) = new Chan[A]
  val push = stack.push + swap(send)
  val pop  = stack.pop  + swap(recv)
}
```

```
stack1.pop >>> stack2.push
```

Going Monadic



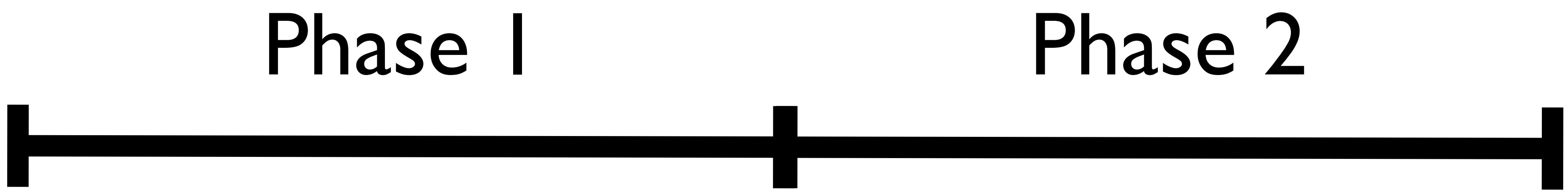
Going Monadic




```
computed: A → (() ↣ B) → (A ↣ B)
```

Use *invisible side-effects* to traverse the queue while computing the `upd` operation to perform

Implementation

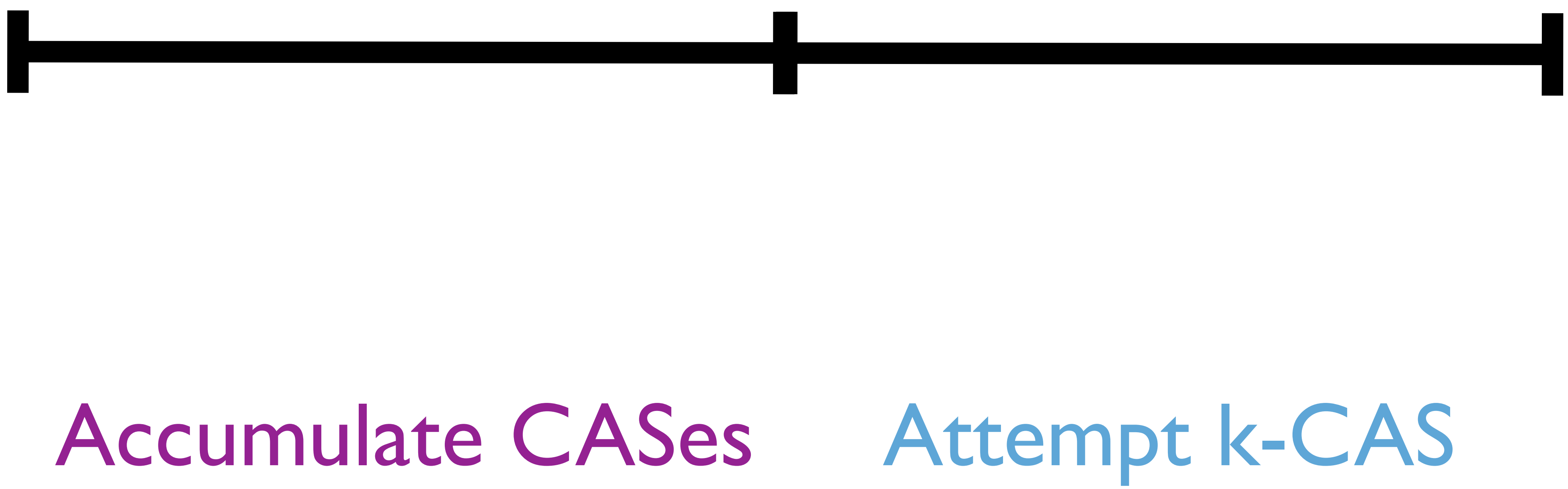

Phase 1
Phase 2

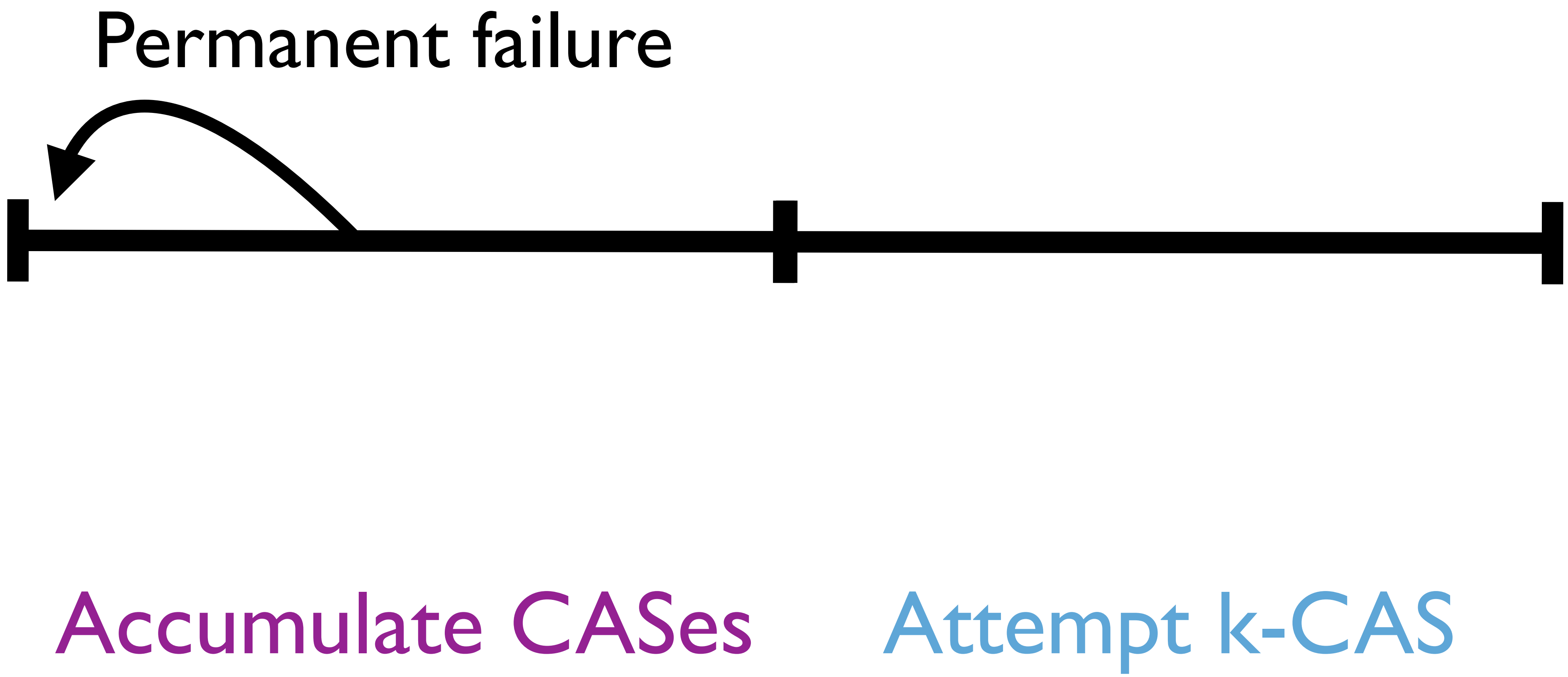
Phase 1 Phase 2

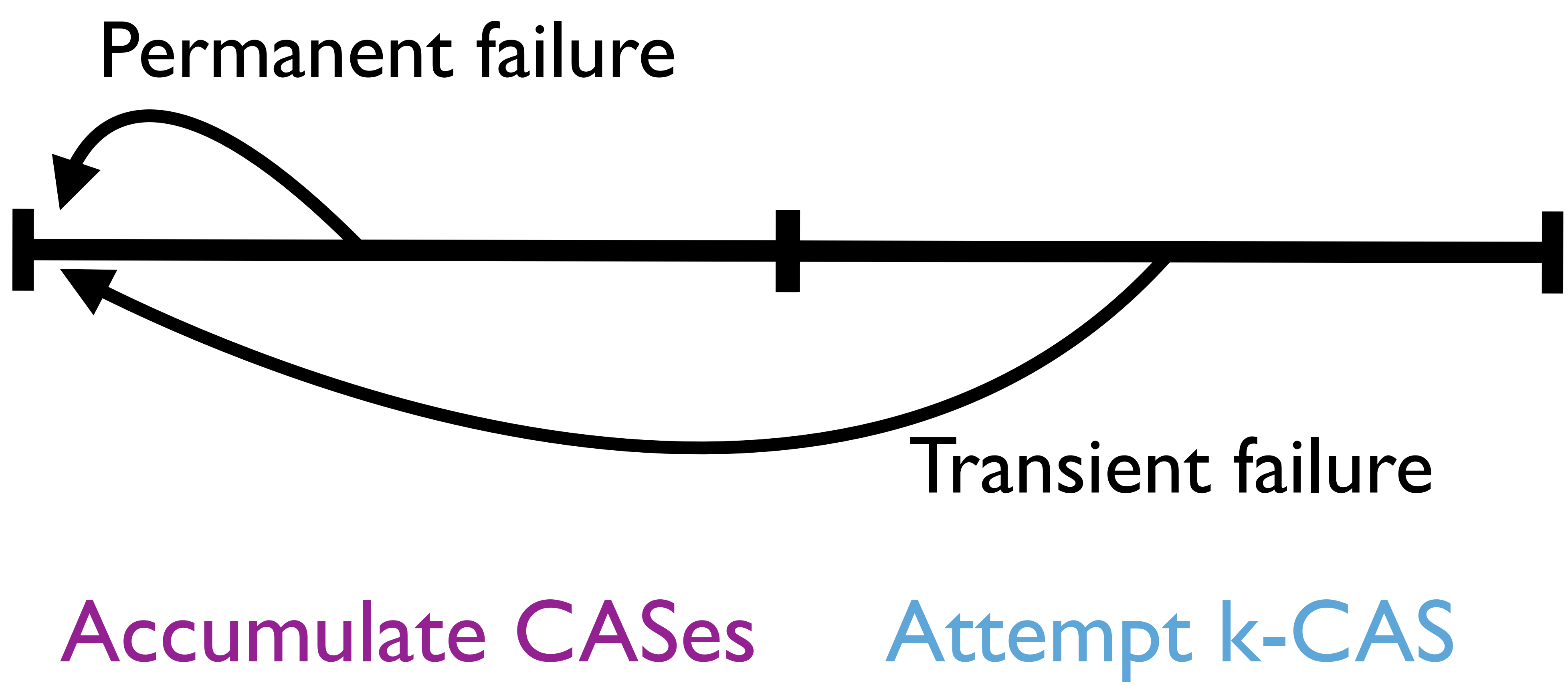
Accumulate CASes

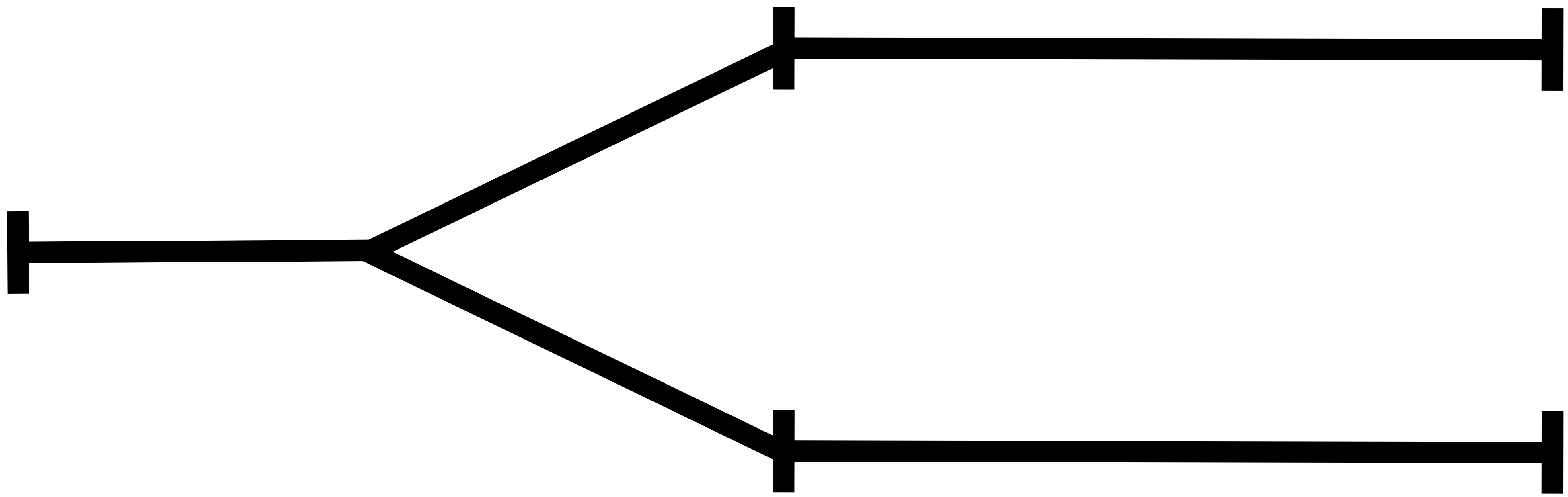
Phase 1 | Phase 2

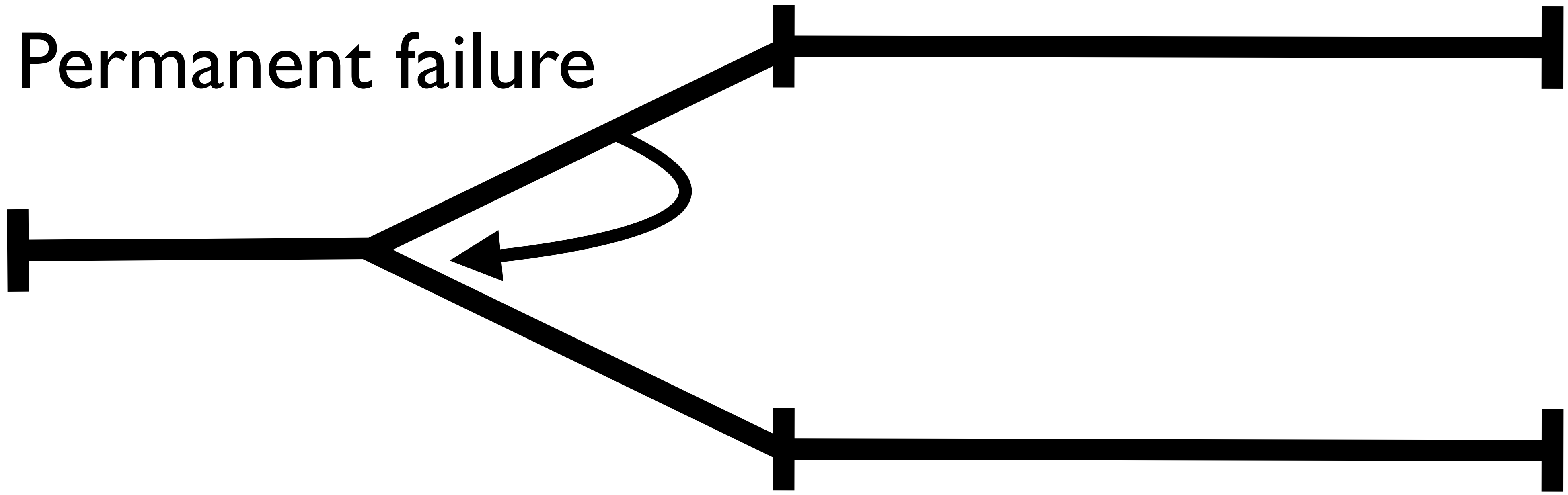
Accumulate CASes | Attempt k-CAS

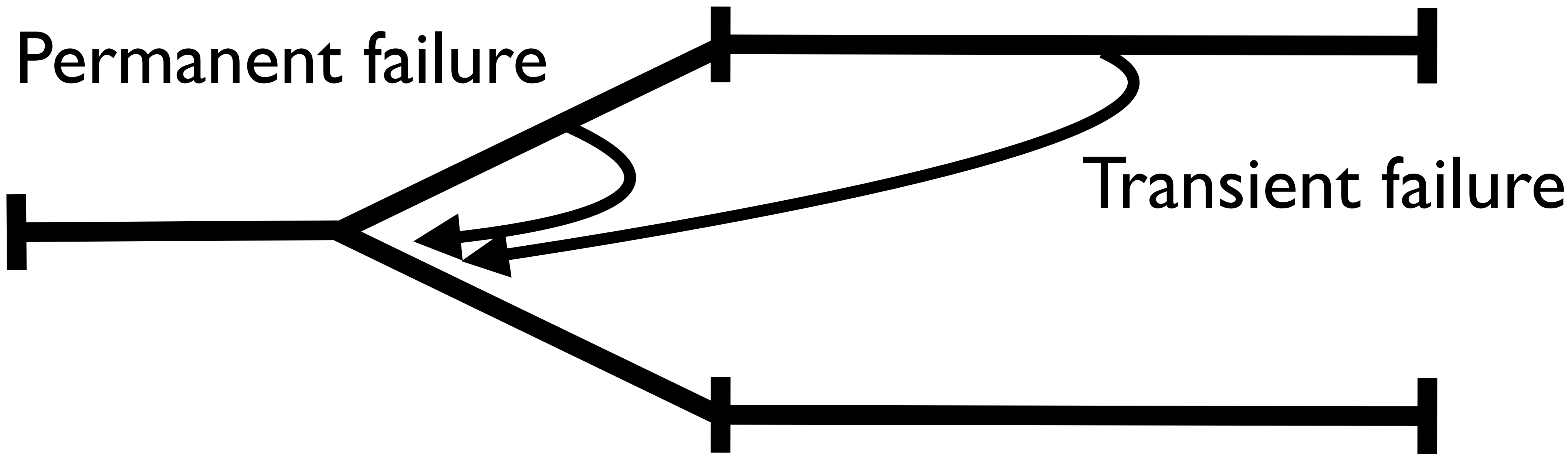

Accumulate CASes
Attempt k-CAS

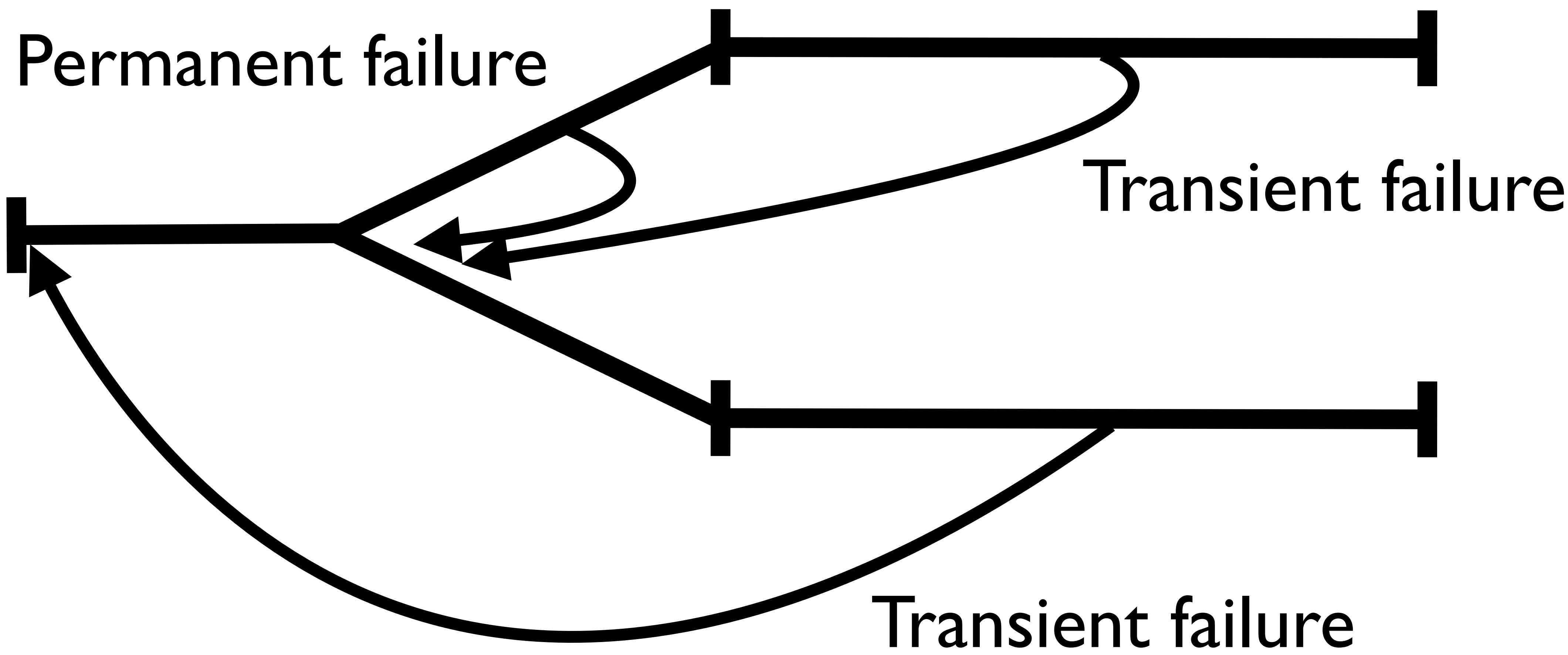

Permanent failure
Accumulate CASes
Attempt k-CAS

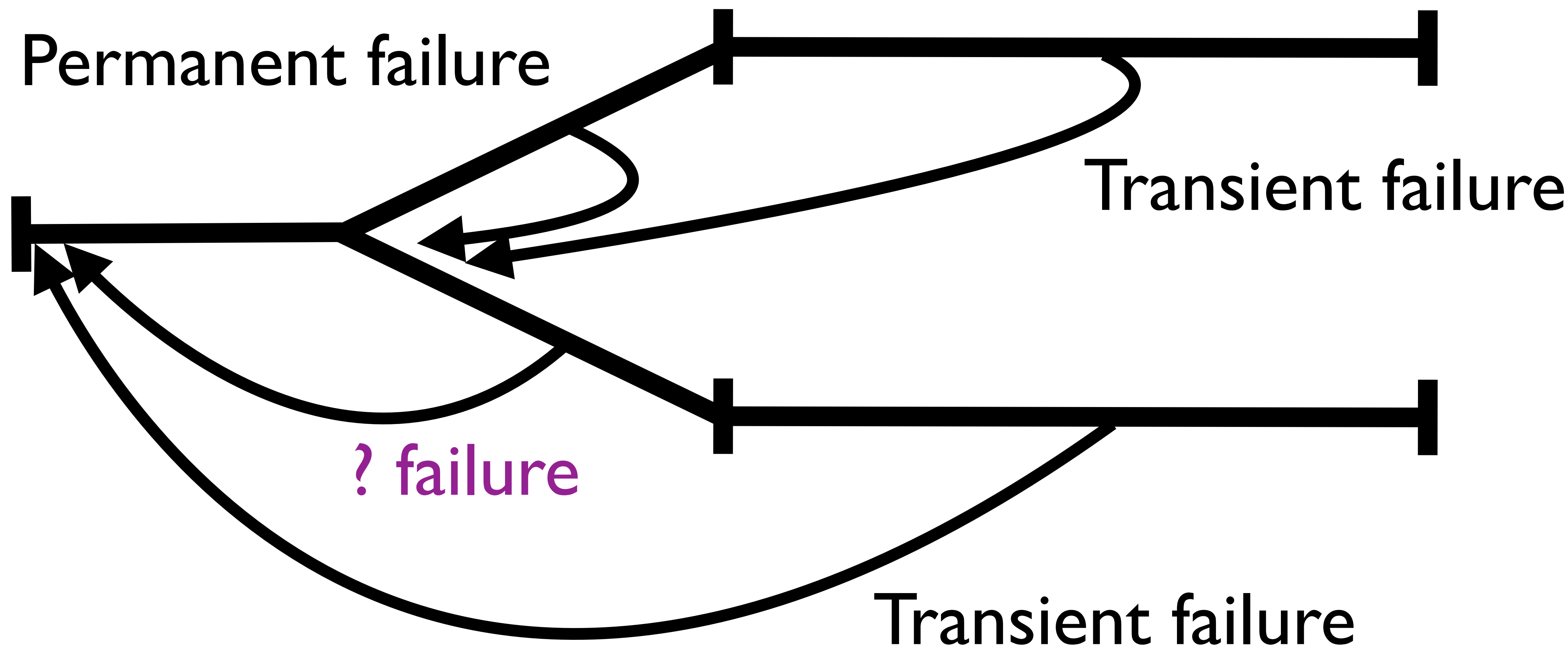

Permanent failure
Transient failure
Accumulate CASes
Attempt k-CAS

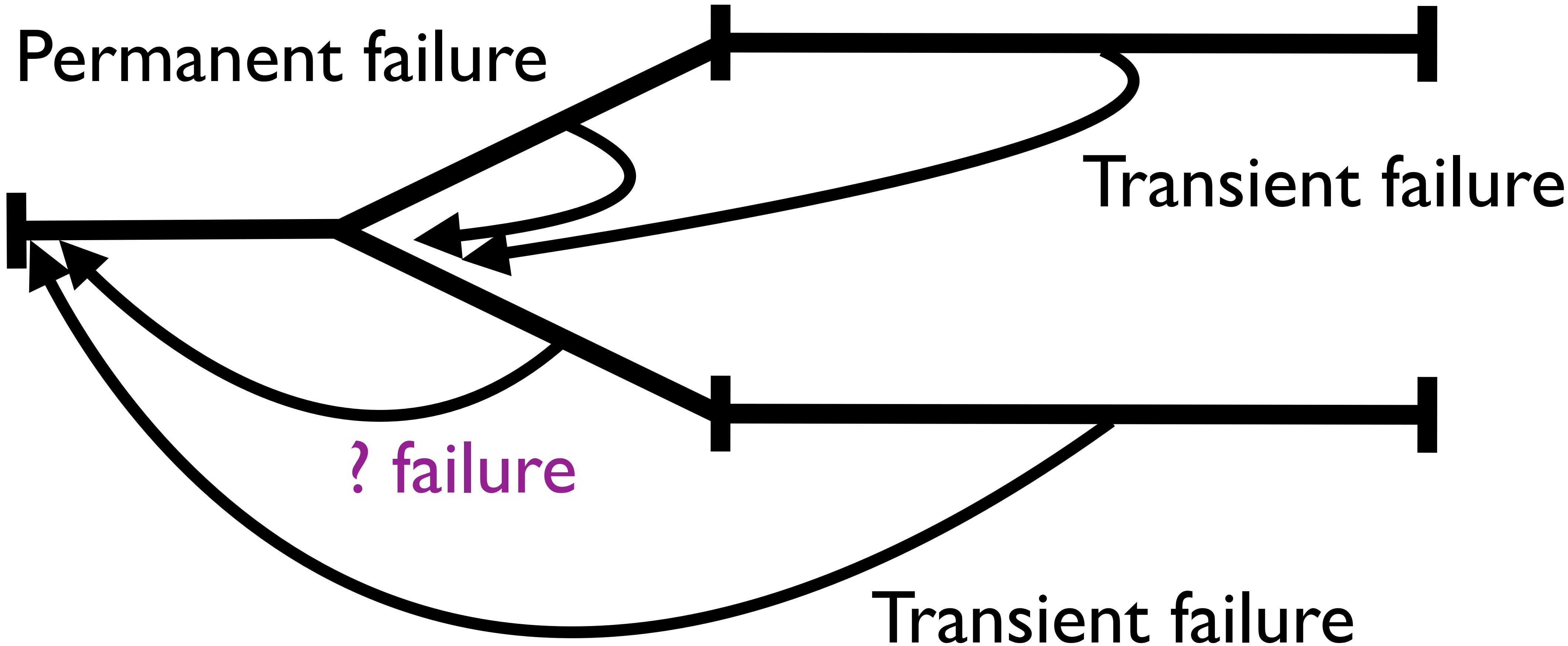



Permanent failure


Permanent failure
Transient failure


Permanent failure
Transient failure
Transient failure


Permanent failure
Transient failure
? failure
Transient failure



P & P = P   P & T = T

T & T = T   T & P = T

Is this just STM?

Is this just STM?

No:

- Single CAS collapses to single phase
- Multiple CASes to single location forbidden

  So the "redo log" is write-only for phase 1!

Therefore: **pay-as-you-go**

- Treiber stack is really a Treiber stack
- Pay for kCAS only for compositions

Is this just STM?

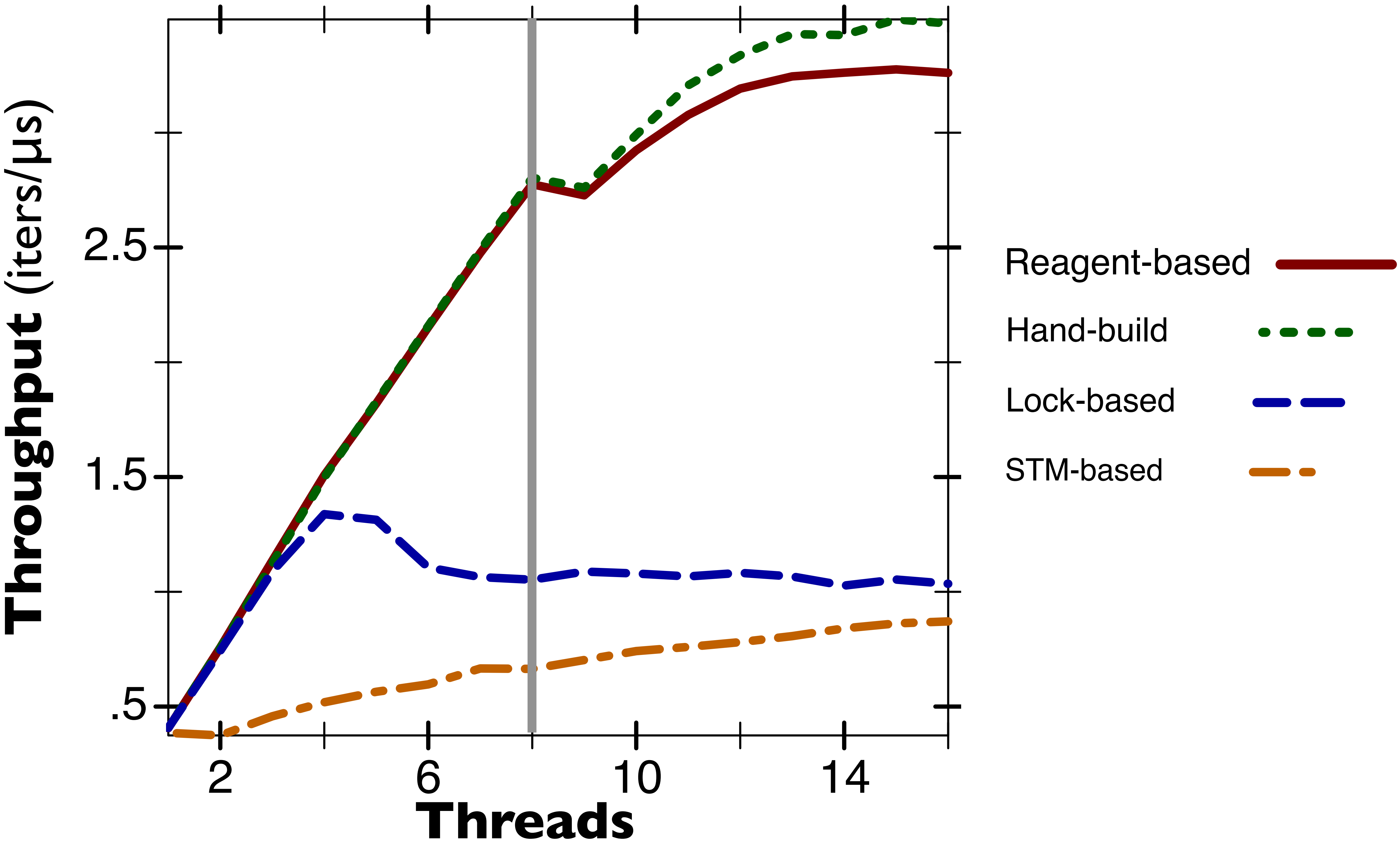
Isolation	**Interaction**
Shared state	Message passing

Is this just STM?

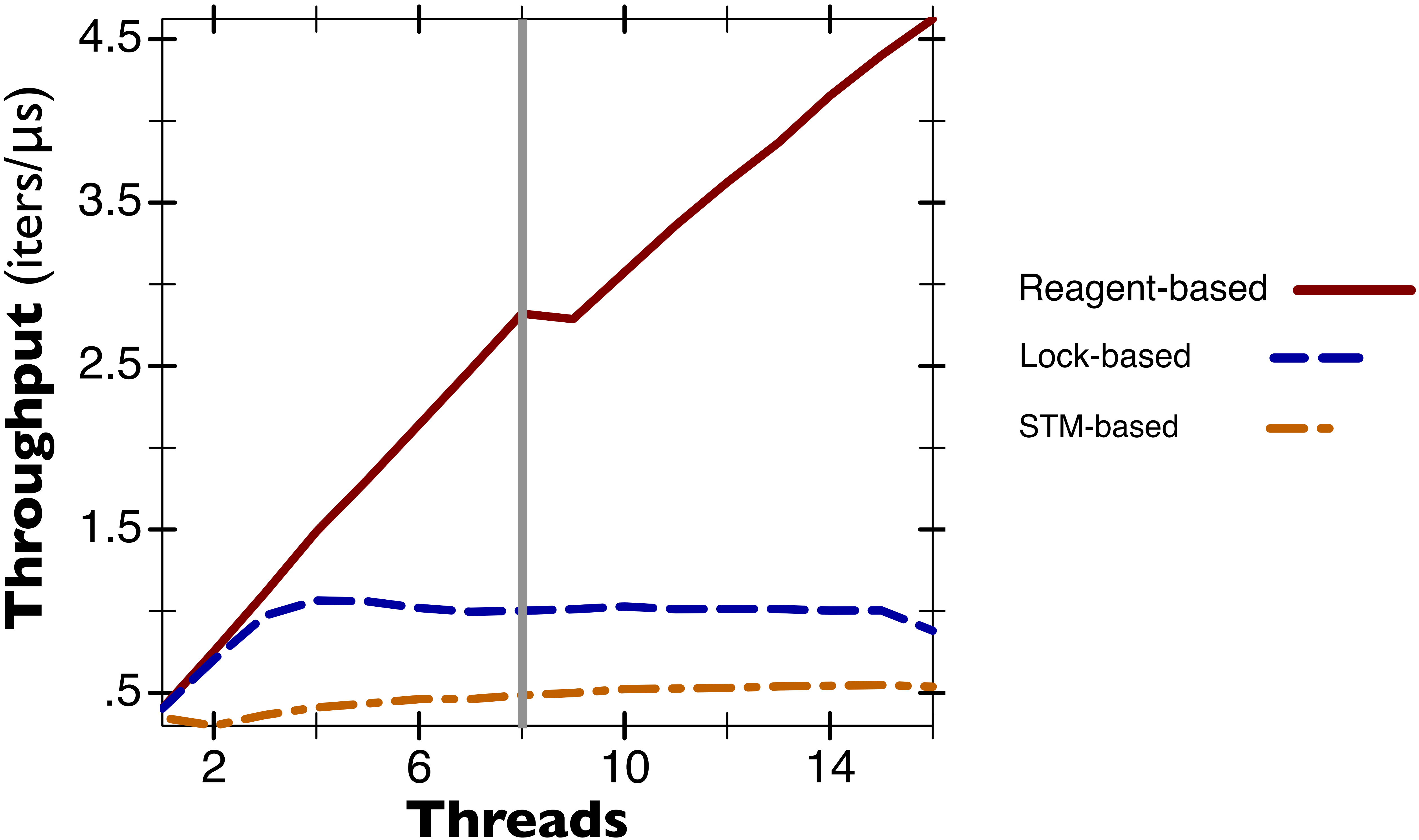
Isolation Shared state	**Interaction** Message passing

Using lock-free bags,
based on earlier work
with Russo [OOPSLA'11]

Treiber stack



Stack transfer



Open Questions

- Composition and invisible read/writes
  - Find a better rule?
  - Statically detect bad cases?
- Composition with *lock-based* algorithms?
- Conflicts between interaction and isolation?

Open Questions 2

- Guaranteed inlining
    - Read/CAS windows must be short
    - “CAPER” with Sam Tobin-Hochstadt
- Formal semantics
    - Integrate Haskell’s STM semantics with message-passing?

Related work

Joins

CML

STM

Related work

Joins

CML

STM

Transactional events

Communicating transactions