Effective Parallelism with Reagents

"KC" Sivaramakrishnan

University of Cambridge

OCaml Labs

Multicore OCaml

Concurrency *Parallelism*

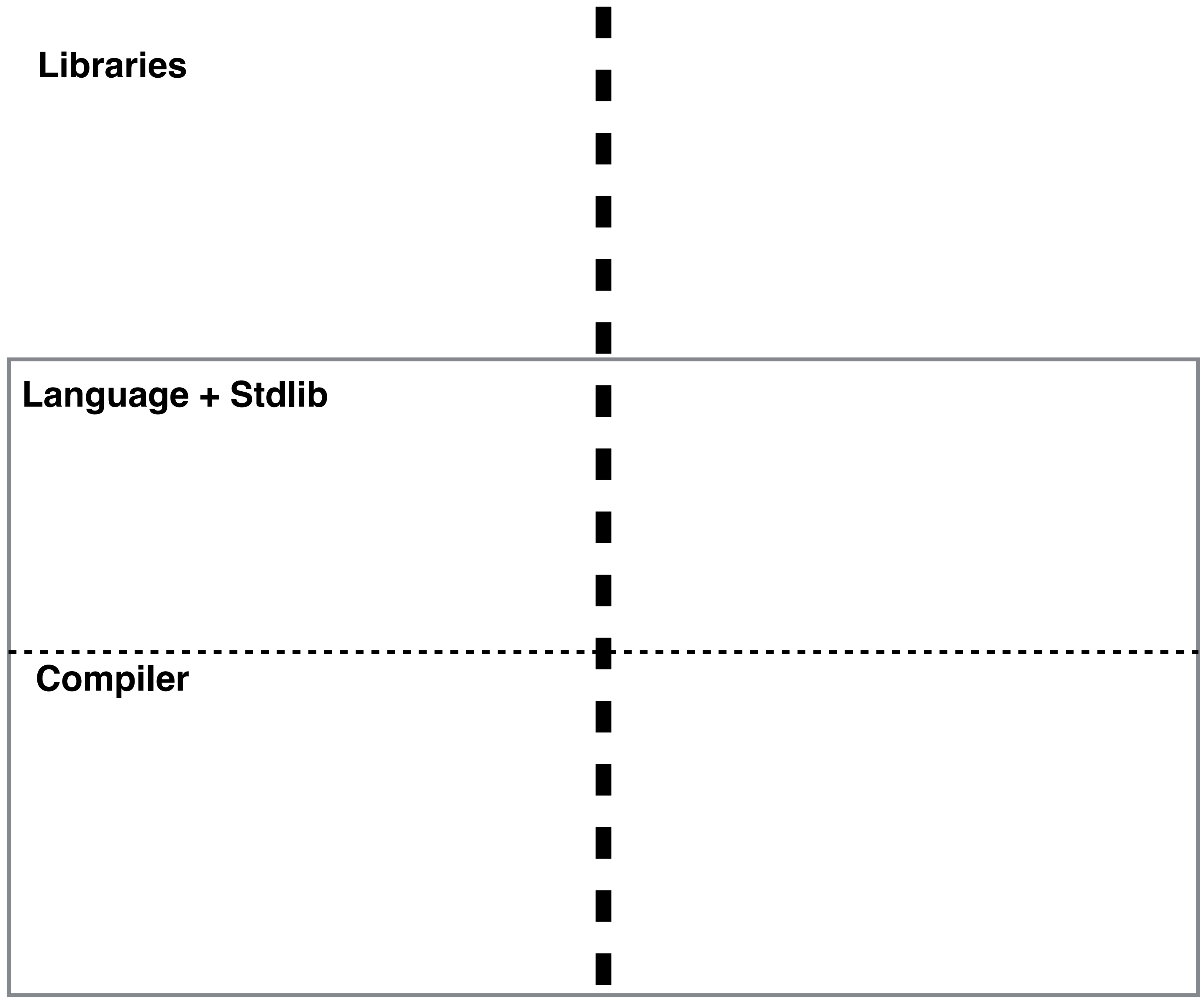
Libraries

Language + Stdlib

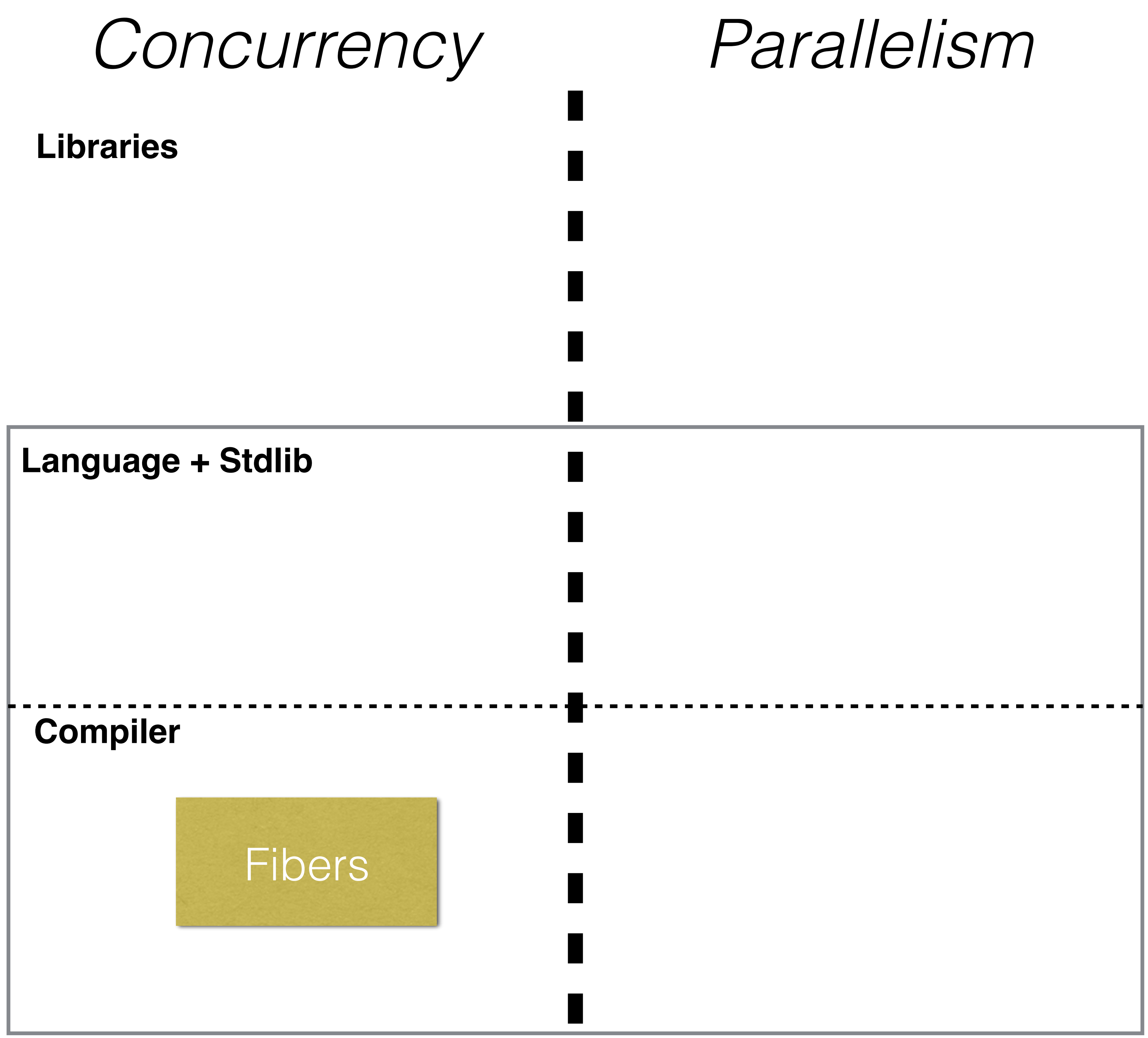
Compiler

Multicore OCaml

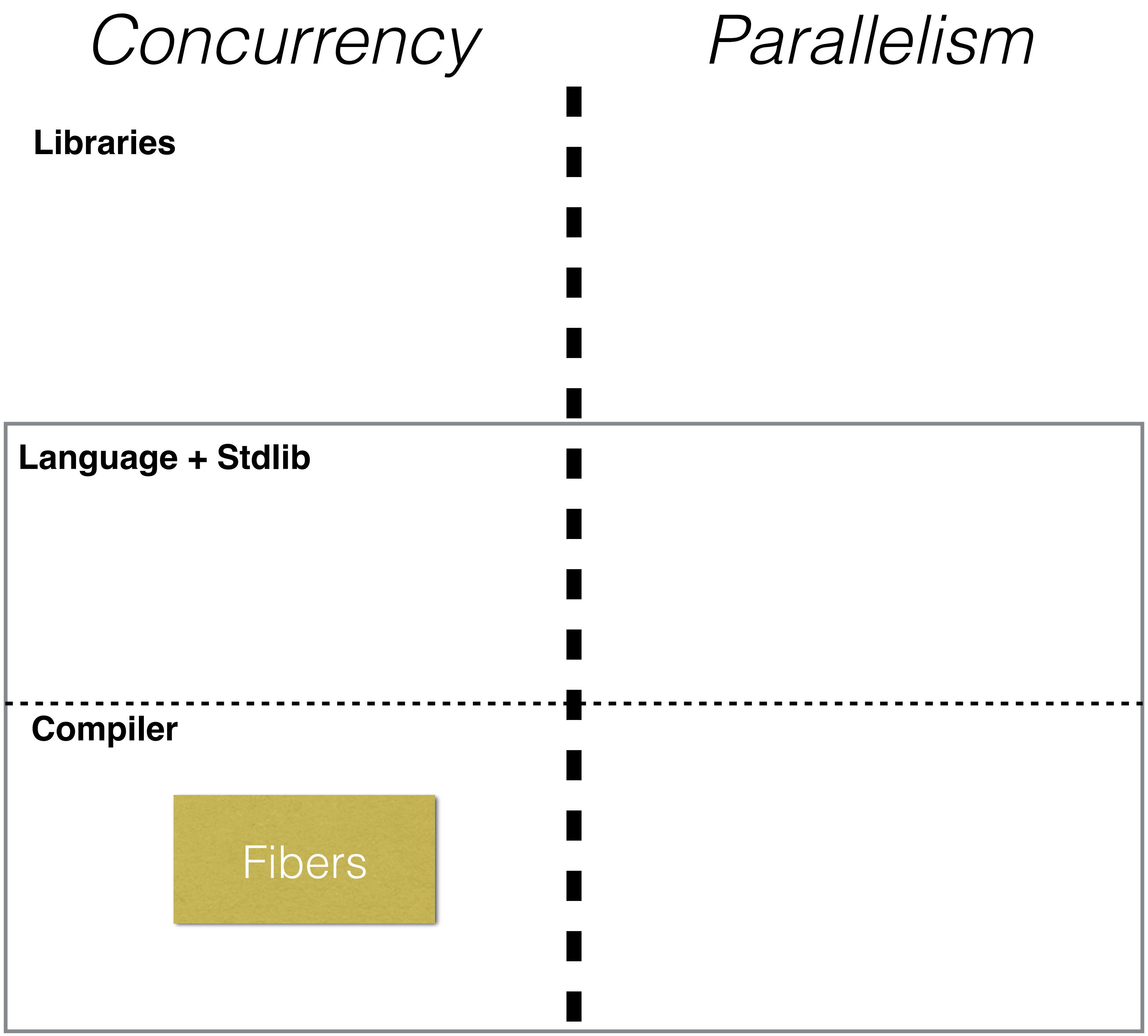
Concurrency *Parallelism*



Multicore OCaml



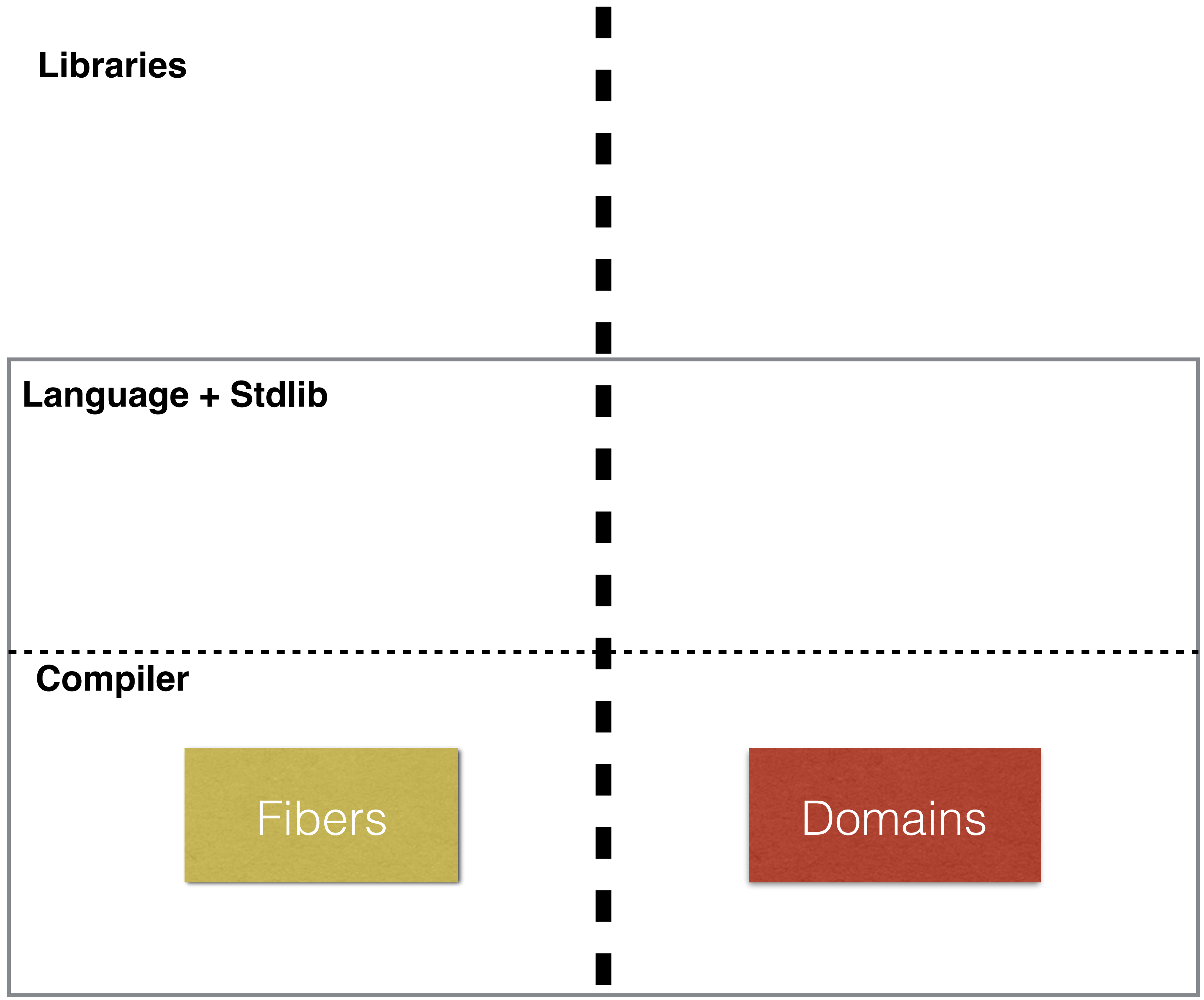
Multicore OCaml



- **12M** fibers/s
  on 1 core
- **30M** fibers/s
  on 4 cores

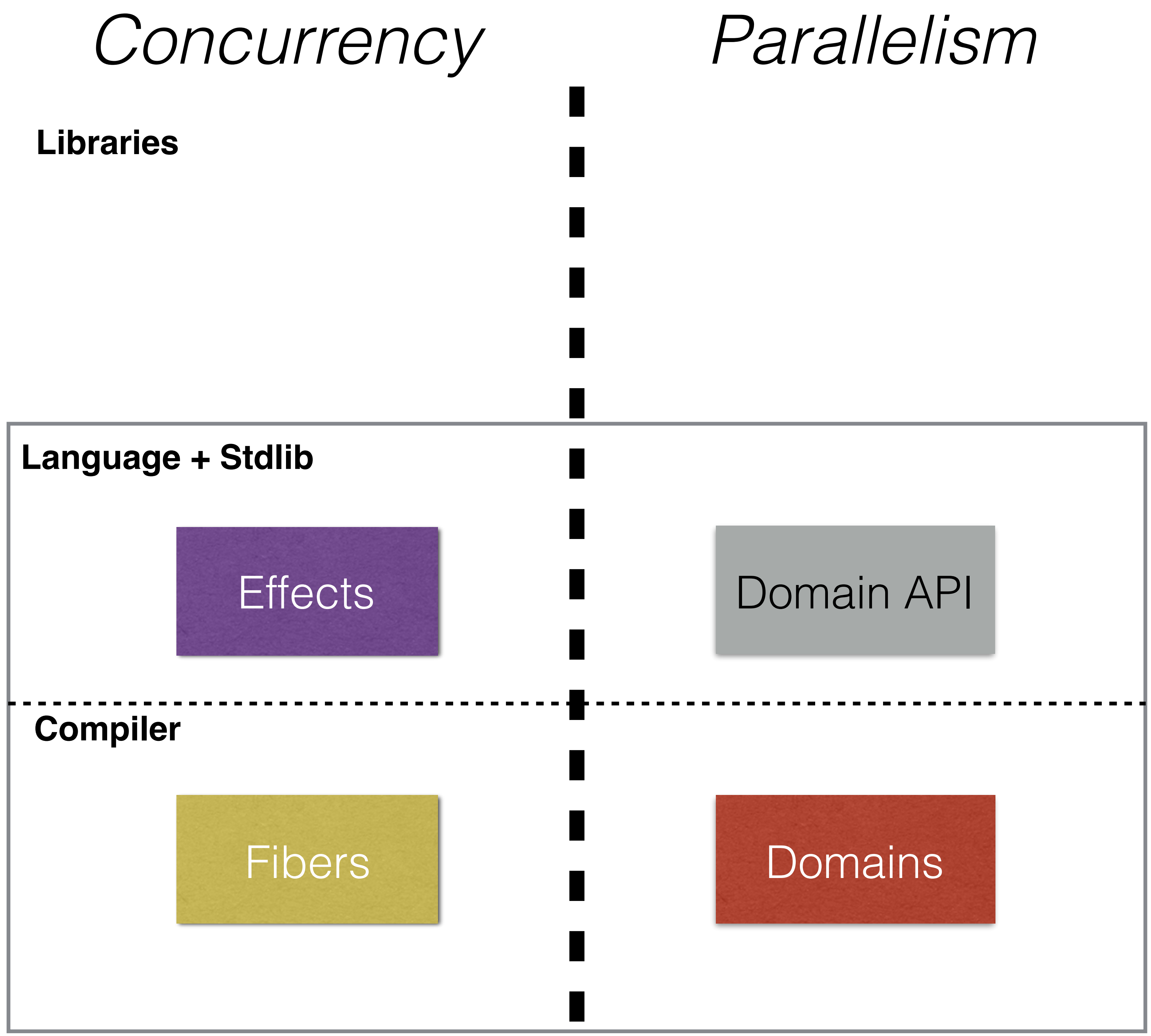
Multicore OCaml

Concurrency *Parallelism*



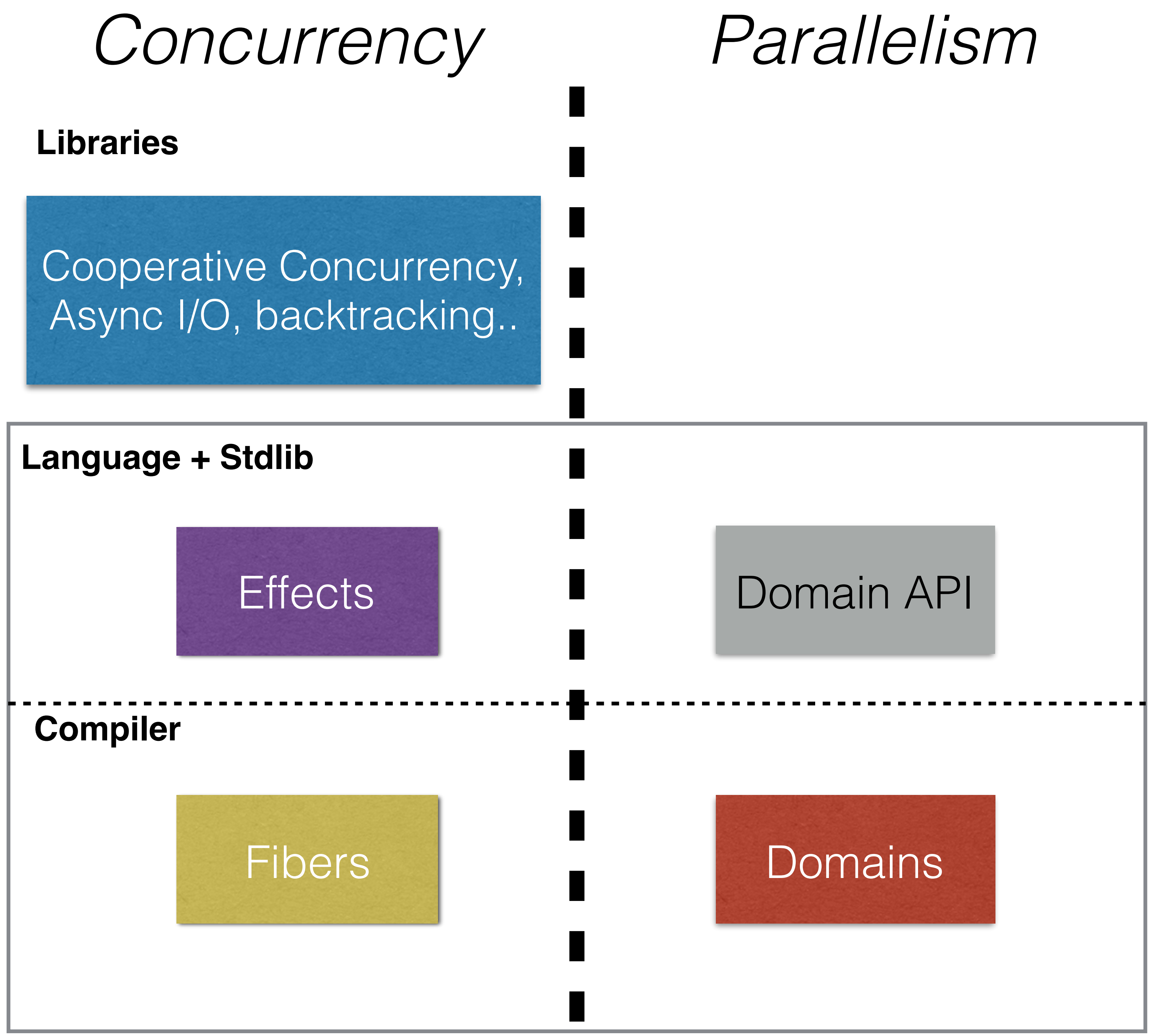
- **12M** fibers/s on 1 core
- **30M** fibers/s on 4 cores

Multicore OCaml



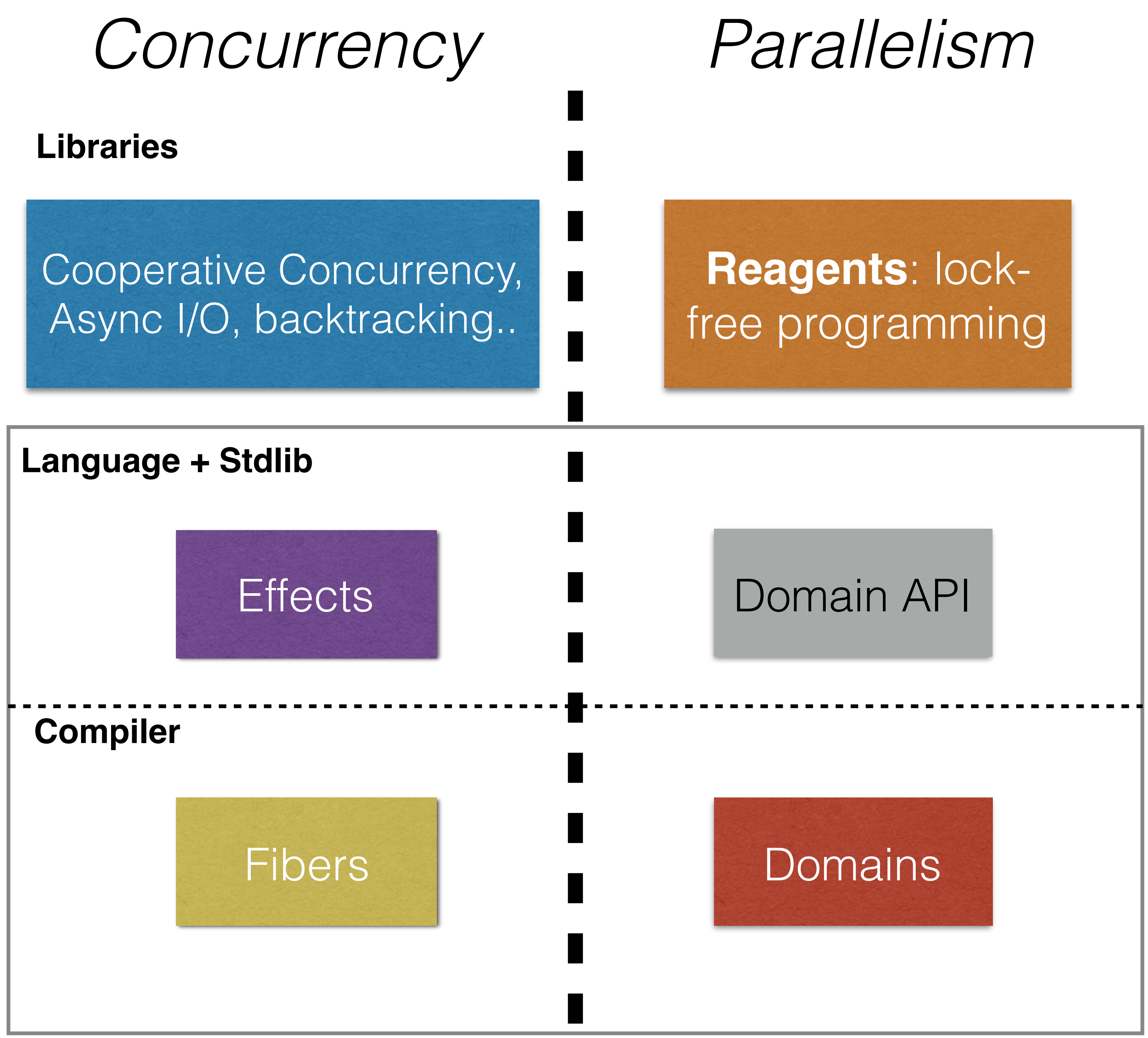
- **12M** fibers/s on 1 core
- **30M** fibers/s on 4 cores

Multicore OCaml



- **12M** fibers/s on 1 core
- **30M** fibers/s on 4 cores

Multicore OCaml



- **12M** fibers/s on 1 core
- **30M** fibers/s on 4 cores

Reagents: lock-free programming

Effects

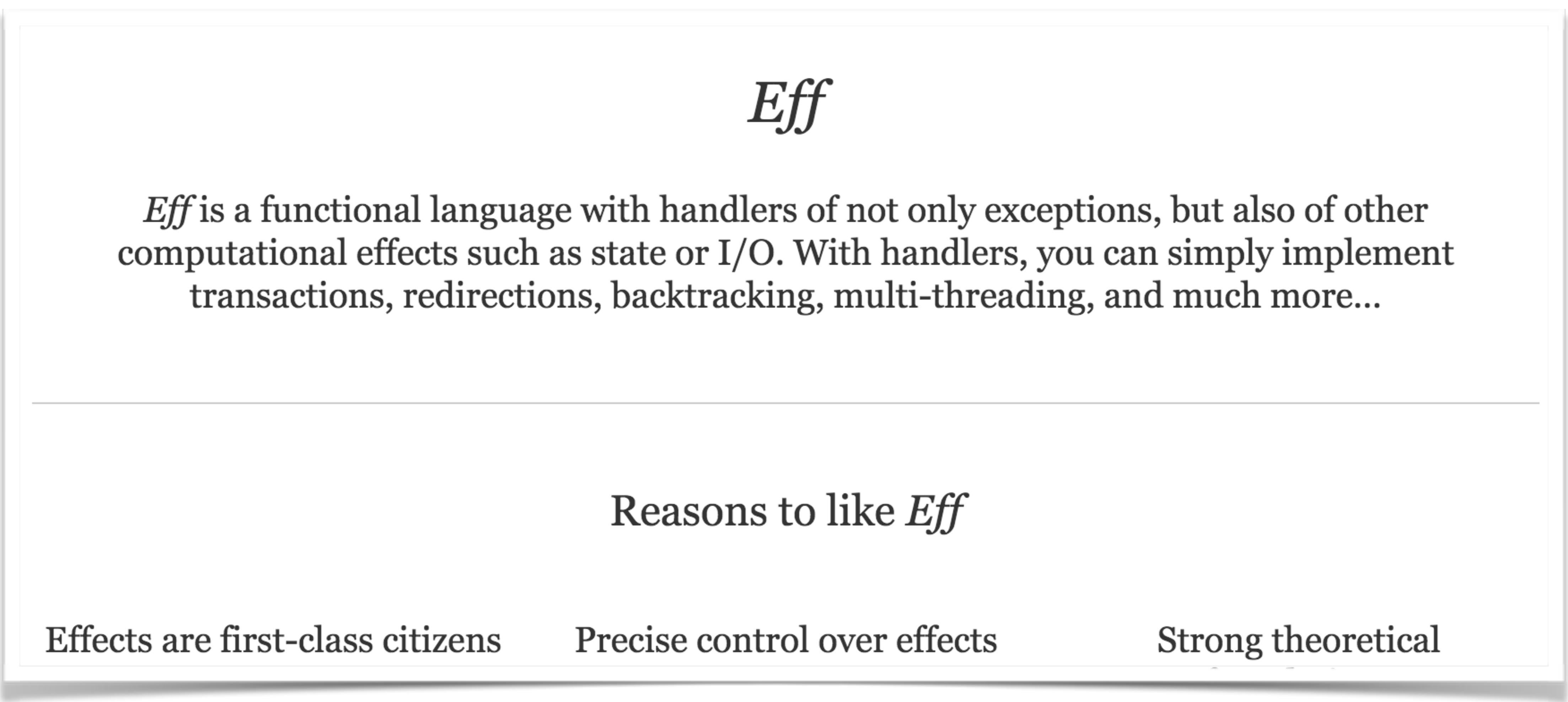
Algebraic effects & handlers

Algebraic effects & handlers

- Programming and reasoning about computational effects in a pure setting.
  - Cf. Monads

Algebraic effects & handlers

- Programming and reasoning about computational effects in a pure setting.
  - Cf. Monads
- *Eff* — **http://www.eff-lang.org/**



Eff

Eff is a functional language with handlers of not only exceptions, but also of other computational effects such as state or I/O. With handlers, you can simply implement transactions, redirections, backtracking, multi-threading, and much more…

Reasons to like *Eff*

Effects are first-class citizens

Precise control over effects

Strong theoretical

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f ()
  with effect (Foo i) k ->
              continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f ()
  with effect (Foo i) k ->
         continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f ()
  with effect (Foo i) k ->
           continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int
```

`let f () = 1 +` ~~`(perform (Foo 3))`~~ `4`

```
let r =
  try
    f ()
  with effect (Foo i) k ->
           continue k (i + 1)
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let r =
  try
    f ()
  with effect (Foo i) k ->
              continue k (i + 1)
```

val r : int = 5

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

`effect Foo : int -> int`

`let f () = 1 +` ~~`(perform (Foo 3))`~~ `4`

```
let r =
  try
    f ()
  with effect (Foo i) k ->
           continue k (i + 1)
```

val r : int = 5

fiber — lightweight stack

- Heap-allocated
- Dynamically resized
- One-shot (affine), explicit cloning

Cooperative Concurrency

```
(* Control operations on threads *)
val fork  : (unit -> unit) -> unit
val yield : unit -> unit
(* Runs the scheduler. *)
val run   : (unit -> unit) -> unit
```

Cooperative Concurrency

```
(* Control operations on threads *)
val fork  : (unit -> unit) -> unit
val yield : unit -> unit
(* Runs the scheduler. *)
val run   : (unit -> unit) -> unit
```

```
effect Fork  : (unit -> unit) -> unit
let fork f = perform (Fork f)

effect Yield : unit
let yield () = perform Yield
```

Cooperative Concurrency

```
(* A concurrent round-robin scheduler *)
let run main =
  let run_q = Queue.create () in
  let enqueue k = Queue.push k run_q in
  let rec dequeue () =
    if Queue.is_empty run_q then ()
    else continue (Queue.pop run_q) ()
  in
  let rec spawn f =
    (* Effect handler => instantiates fiber *)
    match f () with
    | () -> dequeue ()
    | exception e ->
        print_string (Printexc.to_string e);
        dequeue ()
    | effect Yield k -> enqueue k; dequeue ()
    | effect (Fork f) k -> enqueue k; spawn f
  in
  spawn main
```

Generator from Iterator

```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t
```

Generator from Iterator

```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t

let rec iter f = function
  | Leaf -> ()
  | Node (l, x, r) -> iter f l; f x; iter f r
```

Generator from Iterator

```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t

let rec iter f = function
  | Leaf -> ()
  | Node (l, x, r) -> iter f l; f x; iter f r

(* val to_gen : 'a t -> (unit -> 'a option) *)
let to_gen (type a) (t : a t) =
  let module M = struct effect Next : a -> unit end in
  let open M in
  let step = ref (fun () -> assert false) in
  let first_step () =
    try
      iter (fun x -> perform (Next x)) t; None
    with effect (Next v) k ->
      step := continue k; Some v
  in
    step := first_step;
    fun () -> !step ()
```

Concurrency

Algebraic effects & handlers

- Cooperative concurrency
- Backtracking computations
- Selection functionals
- Inversion of control
- Event-based Async I/O in direct-style

Concurrency

Algebraic effects & handlers

- Cooperative concurrency
- Backtracking computations
- Selection functionals
- Inversion of control
- Event-based Async I/O in direct-style

Parallelism

Domain API

Spawn & Join domains

Concurrency

Algebraic effects & handlers

- Cooperative concurrency
- Backtracking computations
- Selection functionals
- Inversion of control
- Event-based Async I/O in direct-style

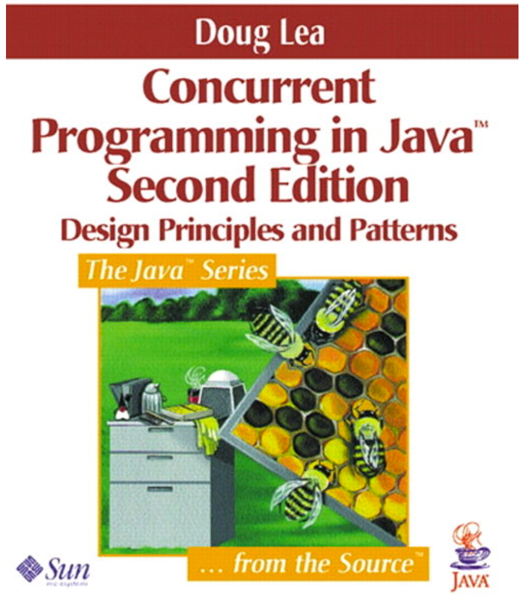
Parallelism

Reagents

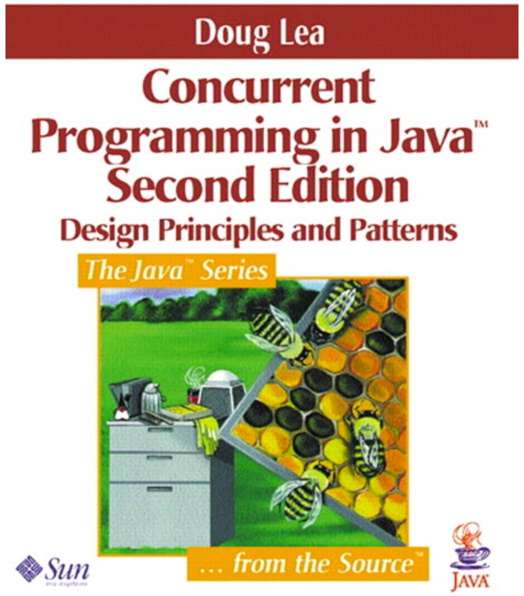
Lock-free synchronisation & data structures

Domain API

Spawn & Join domains



JVM: `java.util.concurrent` **.Net:** `System.Concurrent.Collections`



JVM: `java.util.concurrent`

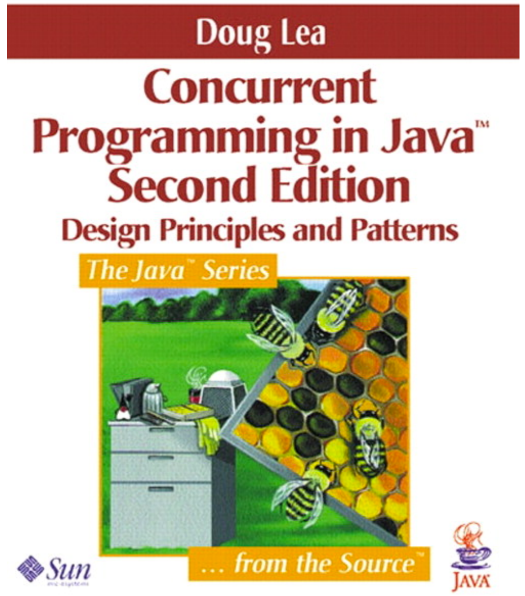
.Net: `System.Concurrent.Collections`

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)



JVM: `java.util.concurrent` **.Net:** `System.Concurrent.Collections`

Synchronization

Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition vari
Coun

Data st

ay & list)
nronous
Priority, nonblocking
Priority, blocking
Deques
Sets
Maps (hash & skiplist)

Not Composable

How to build *composable* lock-free programs?

lock-free

lock-free

Under contention, **at least 1** thread makes progress lock-free

Under contention, **at least 1** thread makes progress

obstruction-free

Single thread **in isolation** makes progress

wait-free	Under contention, **each** thread makes progress
lock-free	Under contention, **at least 1** thread makes progress
obstruction-free	Single thread **in isolation** makes progress

Compare-and-swap (CAS)

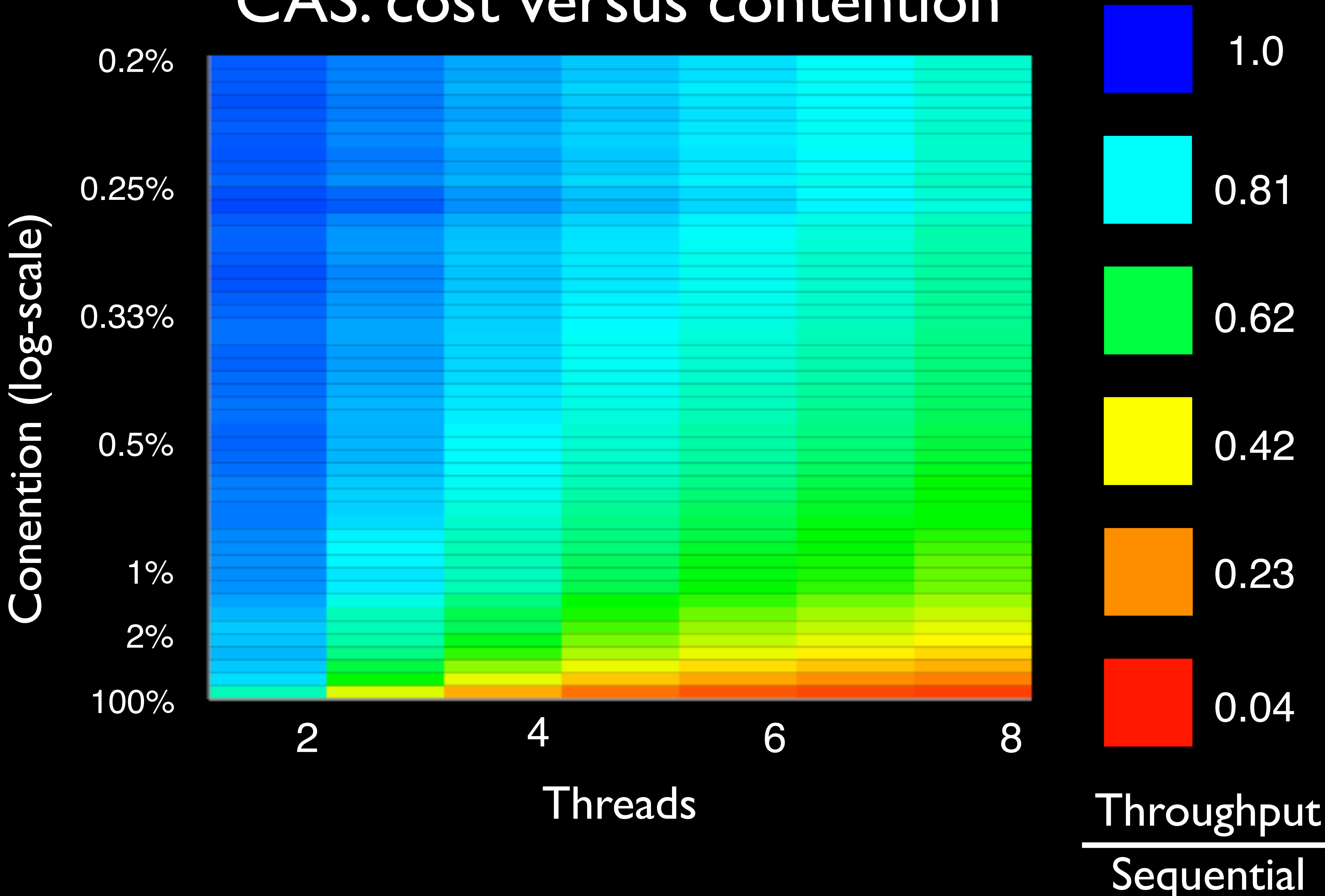
```
module CAS : sig
  val cas : 'a ref -> expect:'a -> update:'a -> bool
end = struct
 (* atomically... *)
 let cas r ~expect ~update =
    if !r = expect then
      (r:= update; true)
    else false
end
```

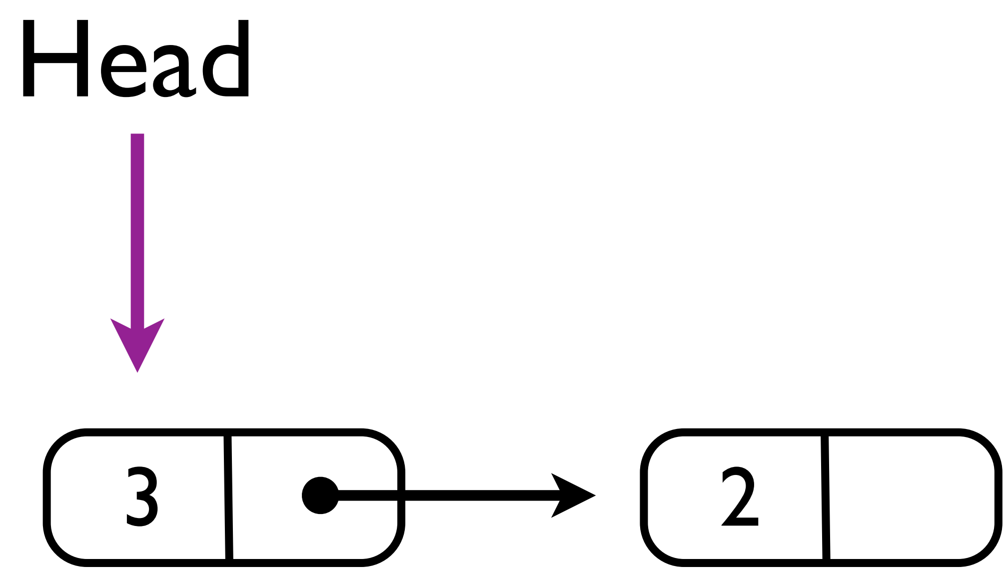
Compare-and-swap (CAS)

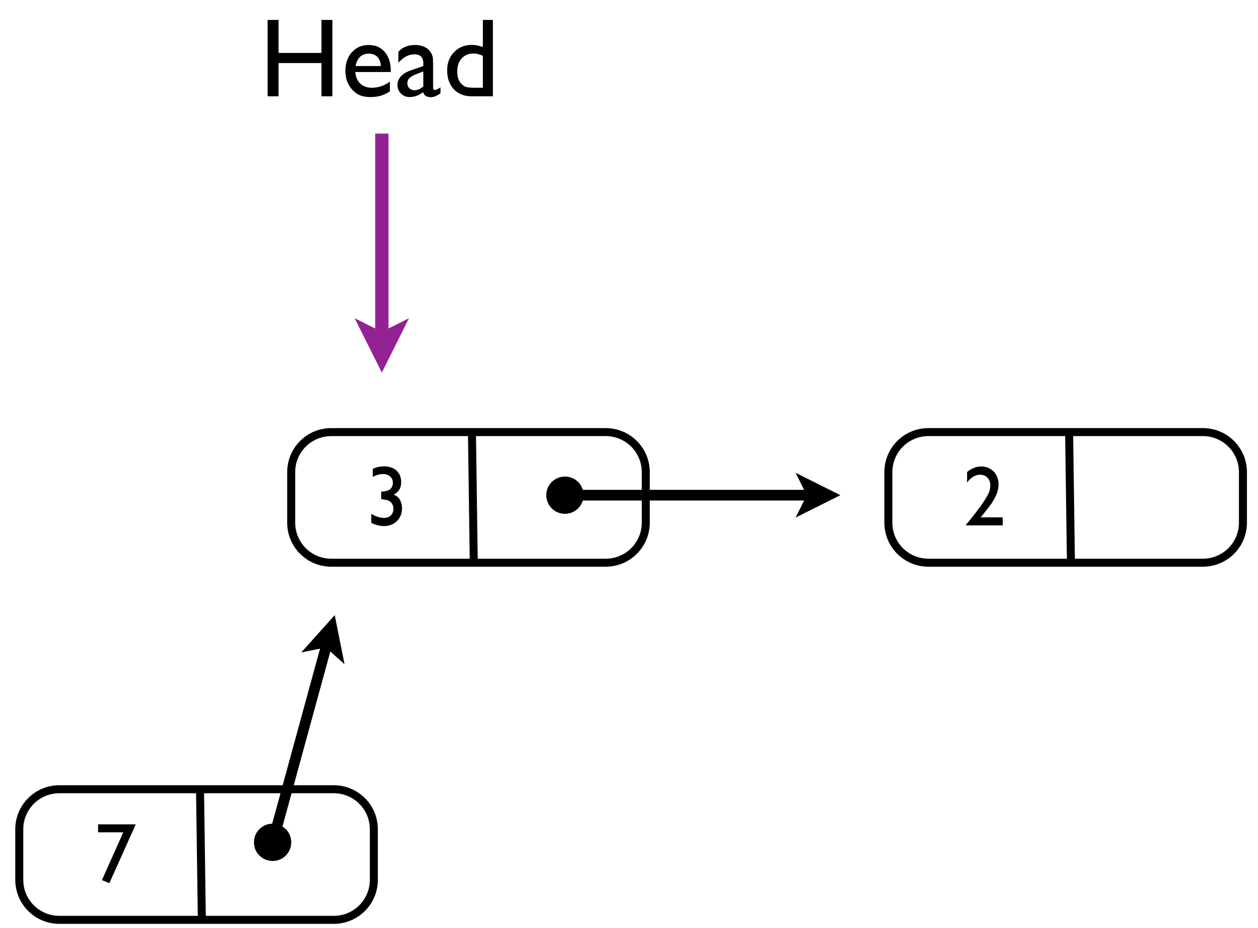
```
module CAS : sig
  val cas : 'a ref -> expect:'a -> update:'a -> bool
end = struct
 (* atomically... *)
 let cas r ~expect ~update =
    if !r = expect then
      (r:= update; true)
    else false
end
```

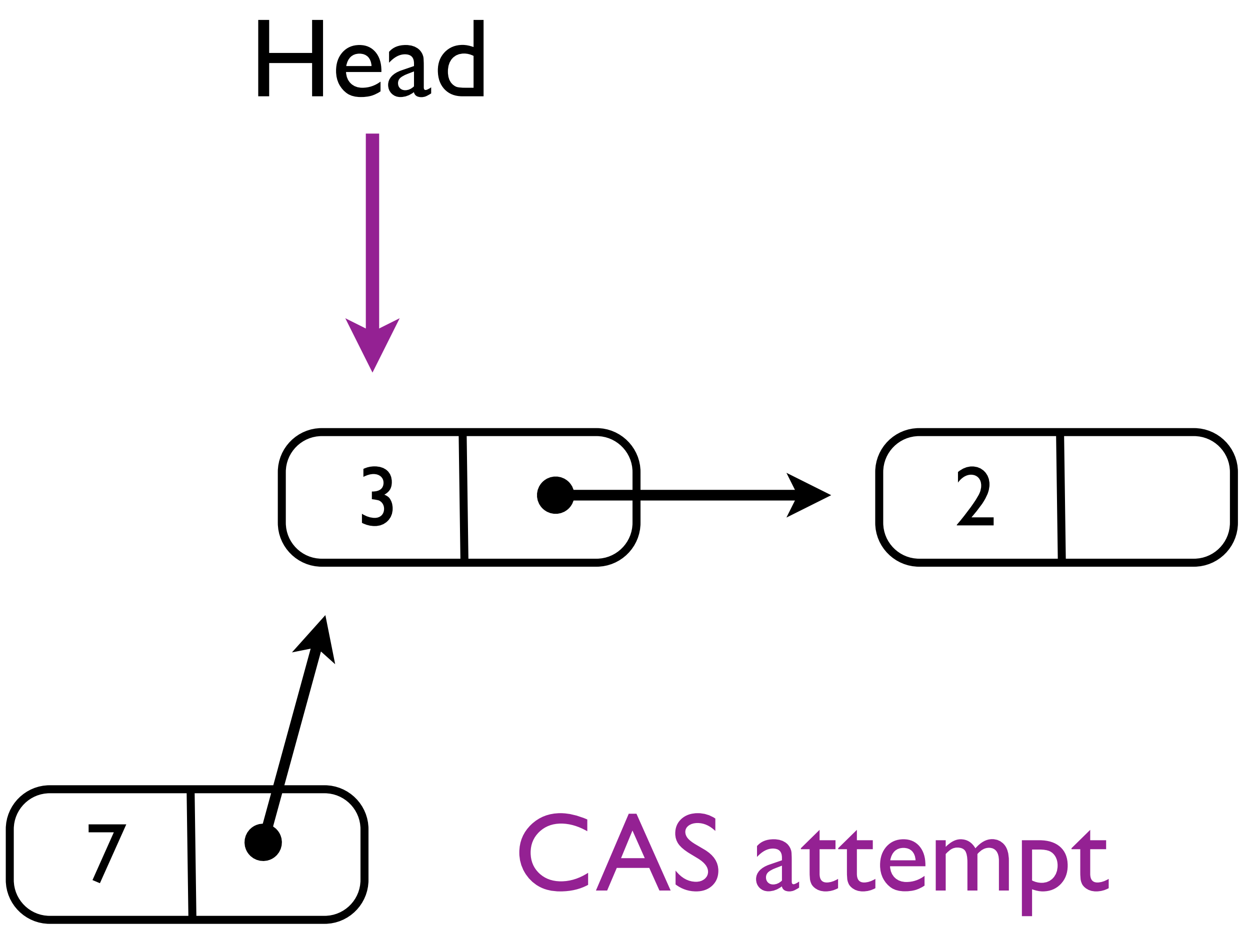
- Implemented *atomically* by processors
  - x86: CMPXCHG and friends
  - arm: LDREX, STREX, etc.
  - ppc: lwarx, stwcx, etc.

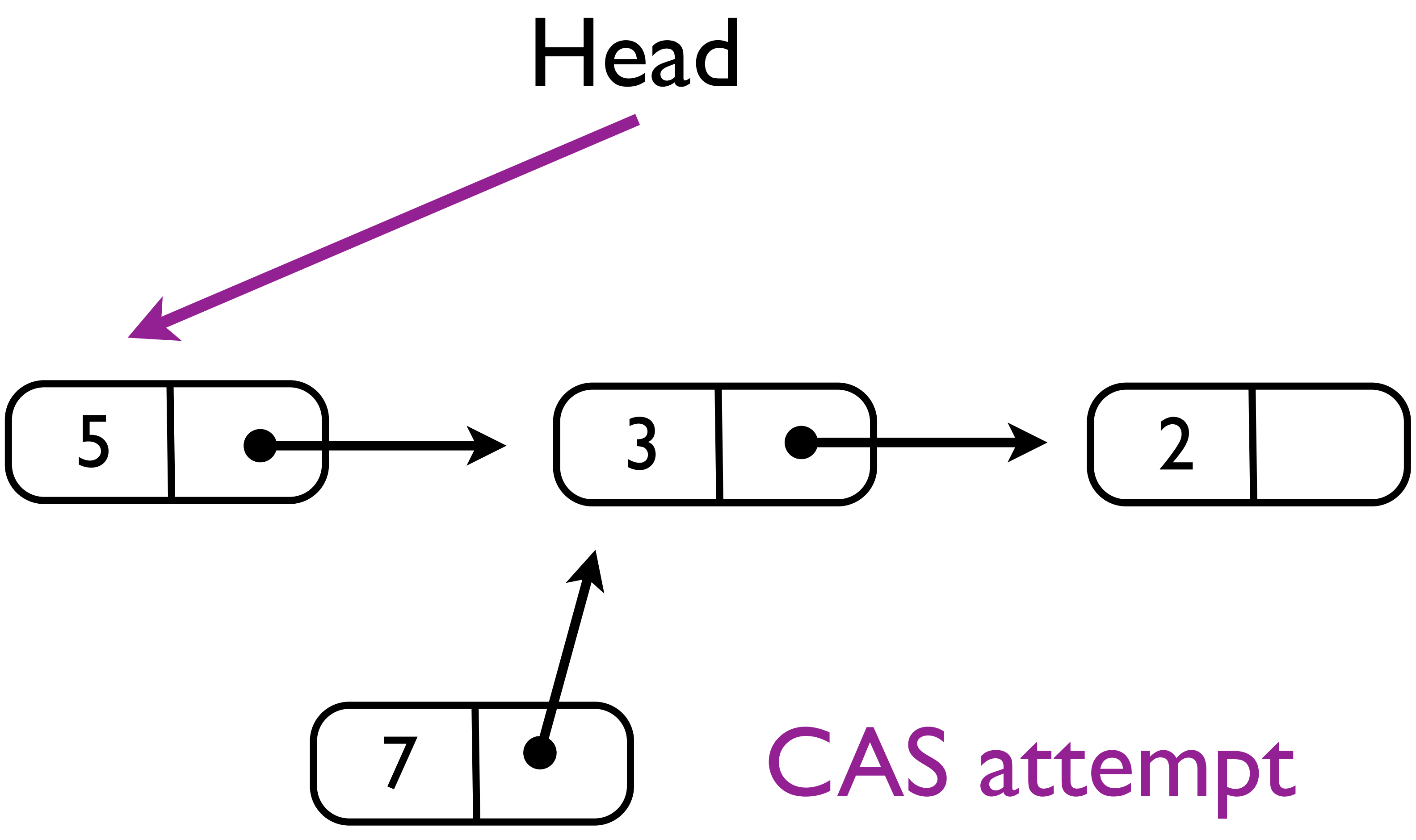
CAS: cost versus contention

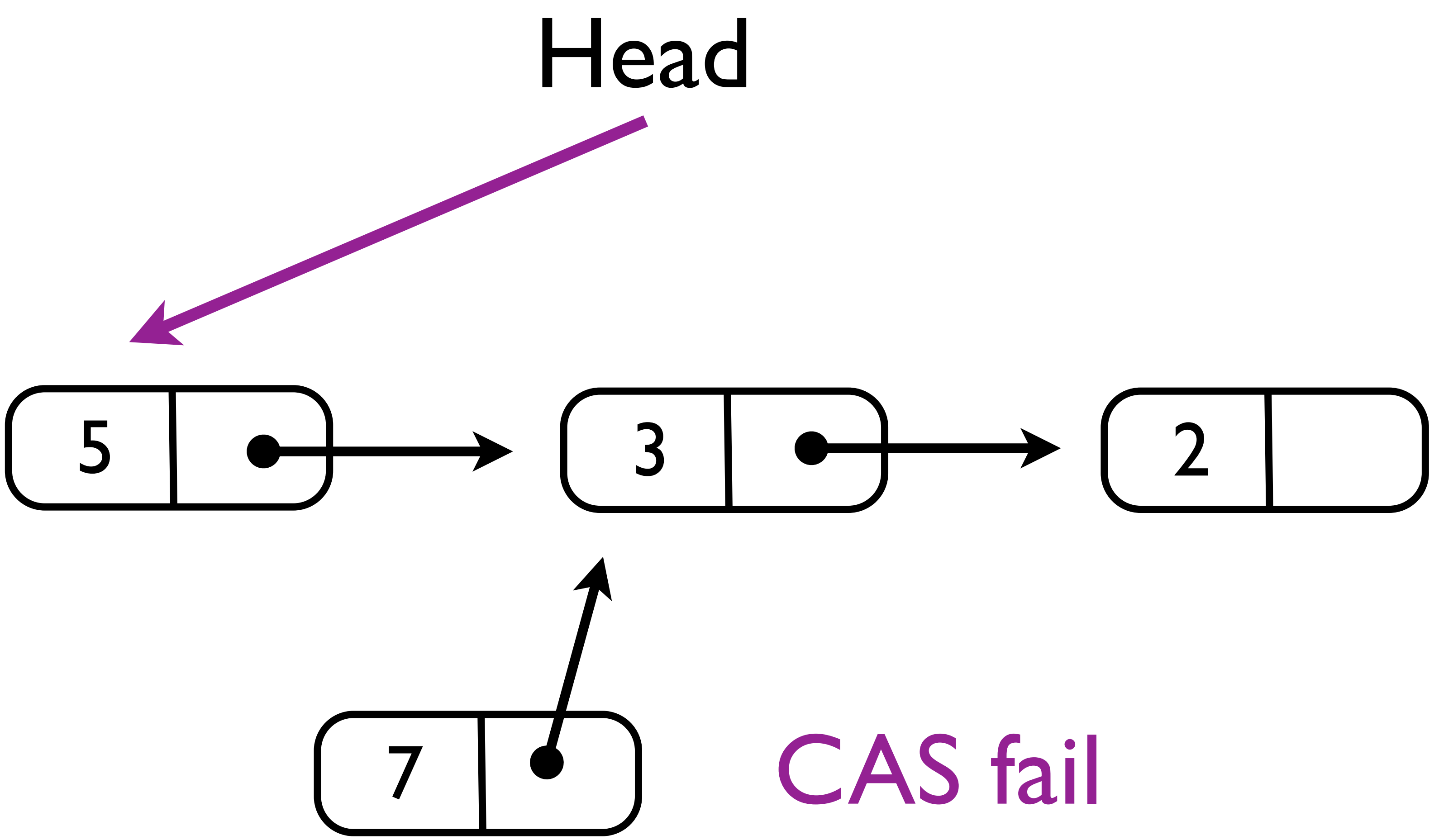


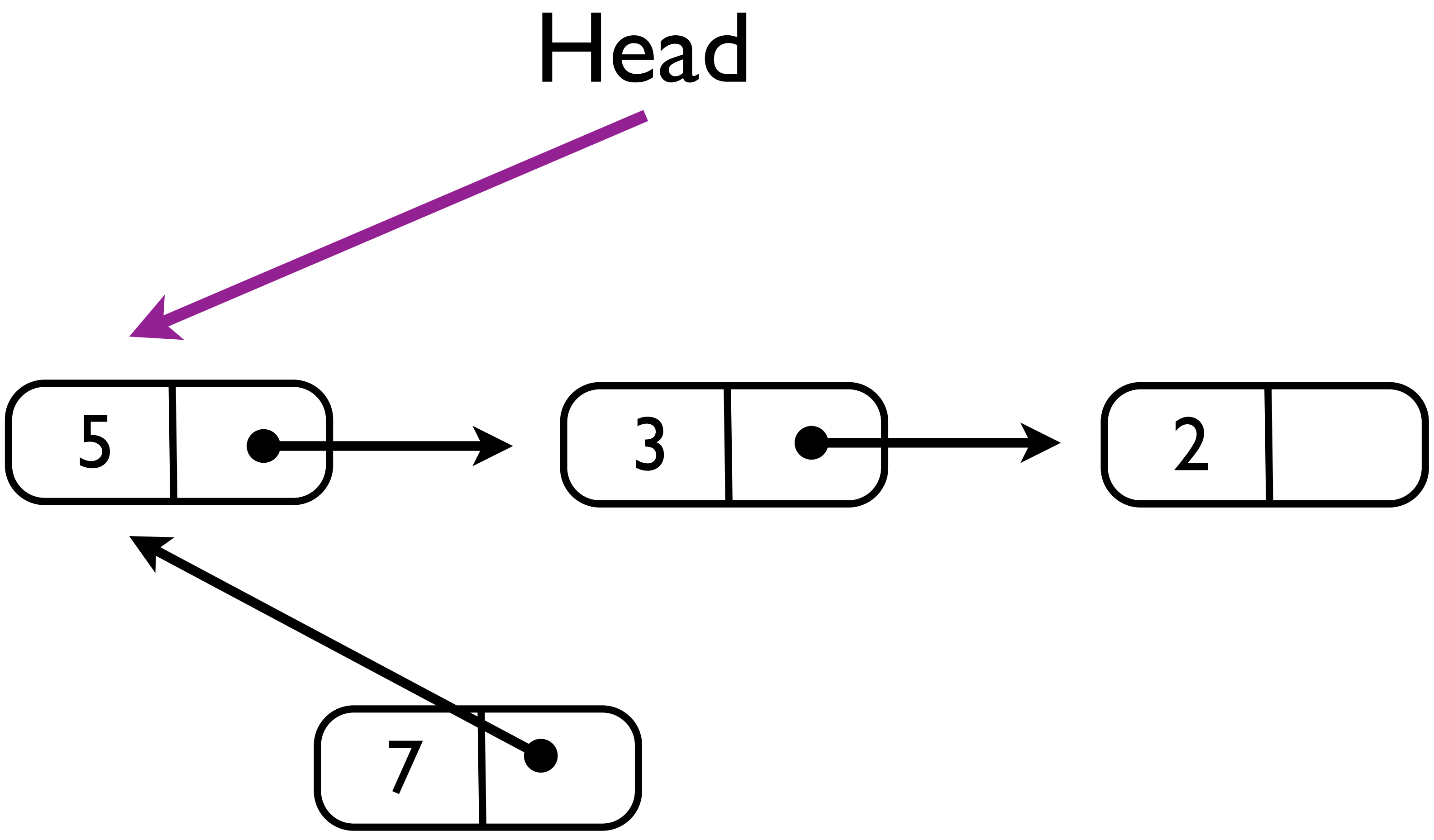

Head
3
2

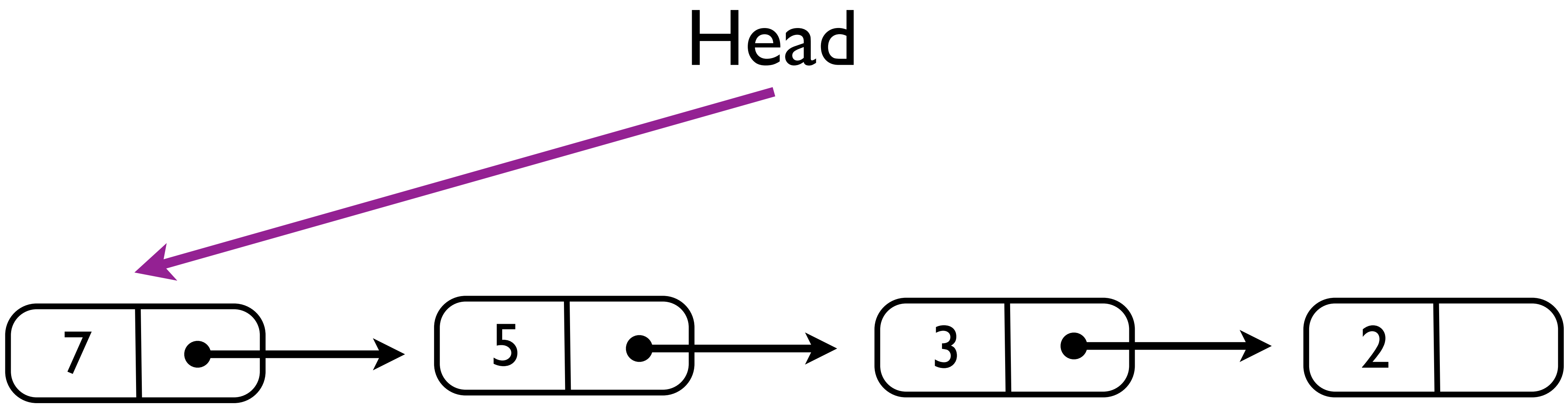

Head
3
2
7


Head
3
2
7
CAS attempt


Head
5
3
2
7
CAS attempt


Head
5
3
2
7
CAS fail


Head
5
3
2
7


Head
7
5
3
2

```
module type TREIBER_STACK = sig
  type 'a t
  val push : 'a t -> 'a -> unit
  ...
end

module Treiber_stack : TREIBER_STACK =
struct
  type 'a t = 'a list ref

  let rec push s t =
    let cur = !s in
    if CAS.cas s cur (t::cur) then ()
    else (backoff (); push s t)
end
```

```
module type TREIBER_STACK = sig
  type 'a t
  val push    : 'a t -> 'a -> unit
  val try_pop : 'a t -> 'a option
end

module Treiber_stack : TREIBER_STACK =
struct
  type 'a t = 'a list ref

  let rec push s t = ...

  let rec try_pop s =
    match !s with
    | [] -> None
    | (x::xs) as cur ->
        if CAS.cas s cur xs then Some x
        else (backoff (); try_pop s)
end
```

```
let v = Treiber_stack.pop s1 in
Treiber_stack.push s2 v
```

is not ***atomic***

The Problem:

Concurrency libraries are indispensable, but hard to build and extend

```
let v = Treiber_stack.pop s1 in
Treiber_stack.push s2 v
```

is not ***atomic***

Reagents

Scalable concurrent algorithms can be built and extended using abstraction and composition

```
Treiber_stack.pop s1 >>> Treiber_stack.push s2
```

is ***atomic***

PLDI 2012

Reagents: Expressing and Composing Fine-grained Concurrency

Aaron Turon

Northeastern University

turon@ccs.neu.edu

Abstract

Efficient communication and synchronization is crucial for fine-grained parallelism. Libraries providing such features, while indispensable, are difficult to write, and often cannot be tailored or composed to meet the needs of specific users. We introduce *reagents*, a set of combinators for concisely expressing concurrency algorithms. Reagents scale as well as their hand-coded counterparts, while providing the composability existing libraries lack.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming; D.3.3 [*Language constructs and features*]: Concurrent programming structures

General Terms Design, Algorithms, Languages, Performance

Such libraries are an enormous undertaking—and one that must be repeated for new platforms. They tend to be conservative, implementing only those data structures and primitives likely to fulfill common needs, and it is generally not possible to safely combine the facilities of the library. For example, JUC provides queues, sets and maps, but not stacks or bags. Its queues come in both blocking and nonblocking forms, while its sets and maps are nonblocking only. Although the queues provide atomic (thread-safe) dequeuing and sets provide atomic insertion, it is not possible to combine these into a single atomic operation that moves an element from a queue into a set.

In short, libraries for fine-grained concurrency are indispensable, but hard to write, hard to extend by composition, and hard to

PLDI 2012

Reagents: Expressing and Composing Fine-grained Concurrency

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Abstract

Efficient communication and synchronization is crucial for fine-grained parallelism. Libraries providing such features, while indispensable, are difficult to write, and often cannot be tailored or composed to meet the needs of specific users. We introduce *reagents*, a set of combinators for concisely expressing concurrency algorithms. Reagents scale as well as their hand-coded counterparts, while providing the composability existing libraries lack.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming; D.3.3 [*Language constructs and features*]: Concurrent programming structures

General Terms Design, Algorithms, Languages, Performance

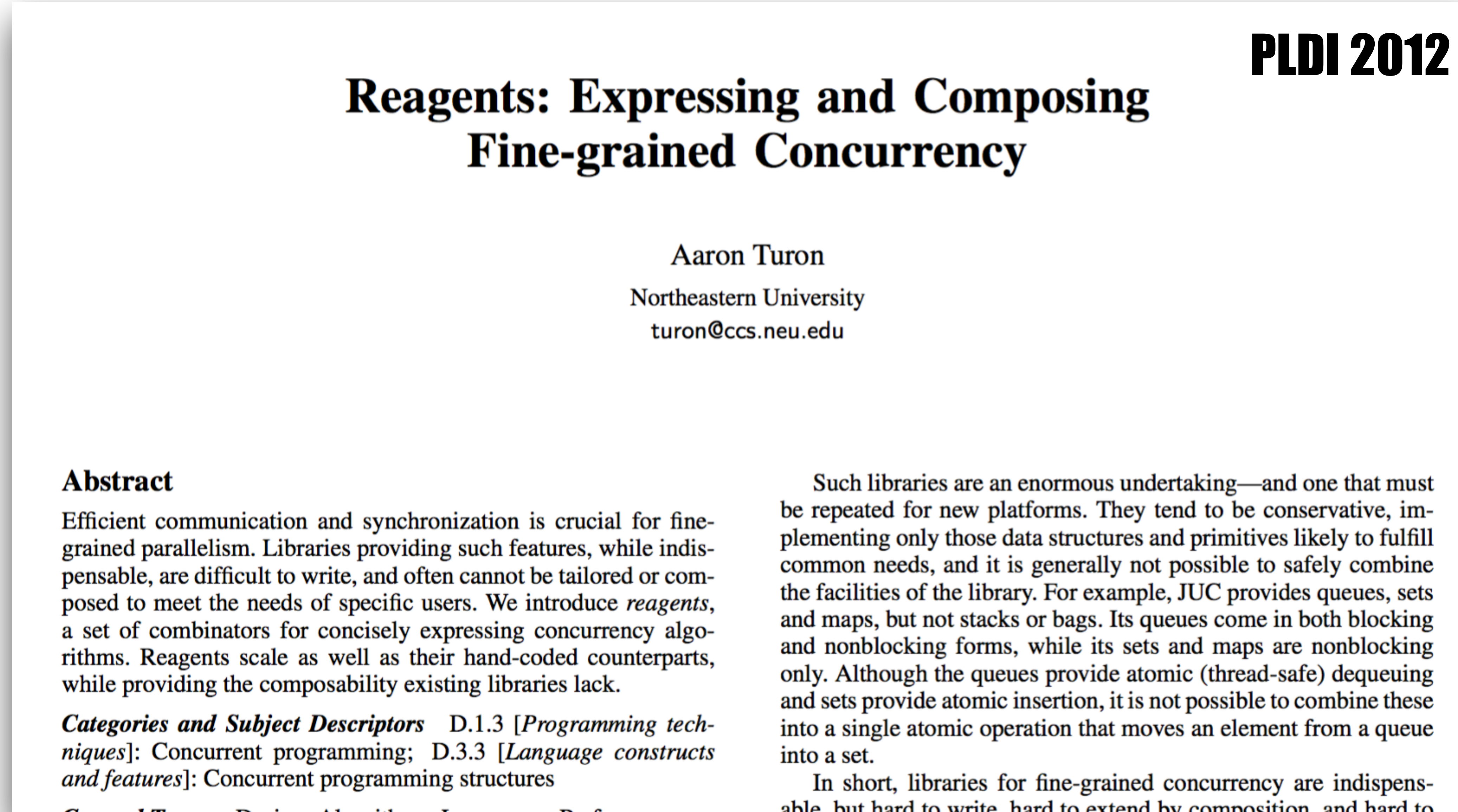
Such libraries are an enormous undertaking—and one that must be repeated for new platforms. They tend to be conservative, implementing only those data structures and primitives likely to fulfill common needs, and it is generally not possible to safely combine the facilities of the library. For example, JUC provides queues, sets and maps, but not stacks or bags. Its queues come in both blocking and nonblocking forms, while its sets and maps are nonblocking only. Although the queues provide atomic (thread-safe) dequeuing and sets provide atomic insertion, it is not possible to combine these into a single atomic operation that moves an element from a queue into a set.

In short, libraries for fine-grained concurrency are indispensable, but hard to write, hard to extend by composition, and hard to

Sequential >>> — Software transactional memory

Parallel <*> — Join Calculus

Selective <+> — Concurrent ML



PLDI 2012

Reagents: Expressing and Composing Fine-grained Concurrency

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Abstract

Efficient communication and synchronization is crucial for fine-grained parallelism. Libraries providing such features, while indispensable, are difficult to write, and often cannot be tailored or composed to meet the needs of specific users. We introduce *reagents*, a set of combinators for concisely expressing concurrency algorithms. Reagents scale as well as their hand-coded counterparts, while providing the composability existing libraries lack.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming; D.3.3 [*Language constructs and features*]: Concurrent programming structures

Such libraries are an enormous undertaking—and one that must be repeated for new platforms. They tend to be conservative, implementing only those data structures and primitives likely to fulfill common needs, and it is generally not possible to safely combine the facilities of the library. For example, JUC provides queues, sets and maps, but not stacks or bags. Its queues come in both blocking and nonblocking forms, while its sets and maps are nonblocking only. Although the queues provide atomic (thread-safe) dequeuing and sets provide atomic insertion, it is not possible to combine these into a single atomic operation that moves an element from a queue into a set.

In short, libraries for fine-grained concurrency are indispensable, but hard to write, hard to extend by composition, and hard to

Sequential >>> — Software transactional memory

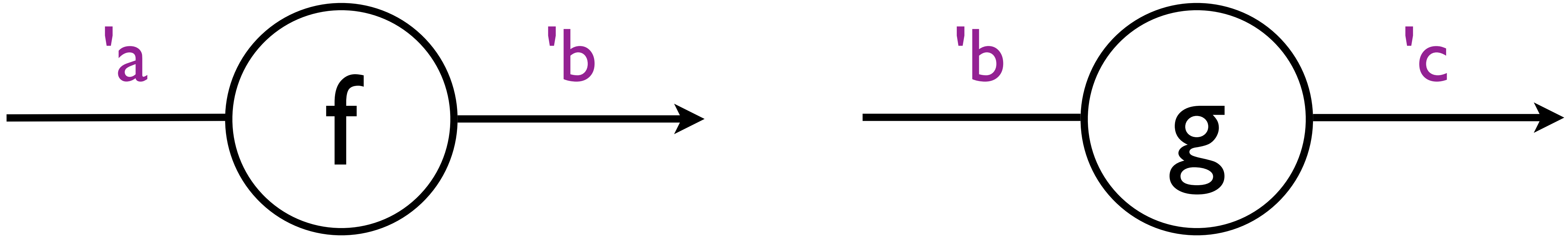
Parallel <*> — Join Calculus

Selective <+> — Concurrent ML

still lock-free!

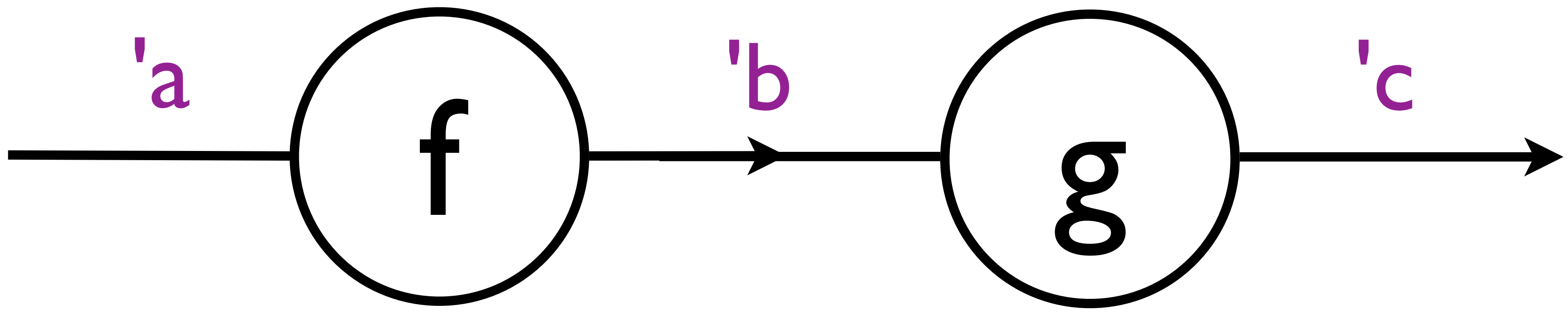
Design

Lambda: the ultimate abstraction



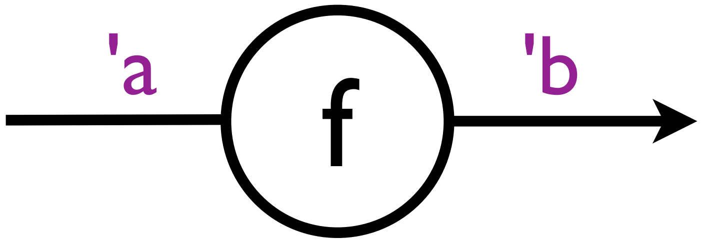

```
val f : 'a -> 'b          val g : 'b -> 'c
```

Lambda: the ultimate abstraction

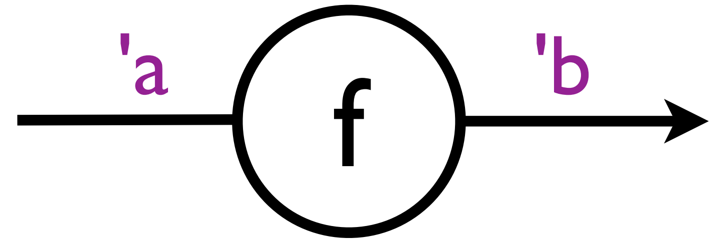



```
(compose g f): 'a -> 'c
```

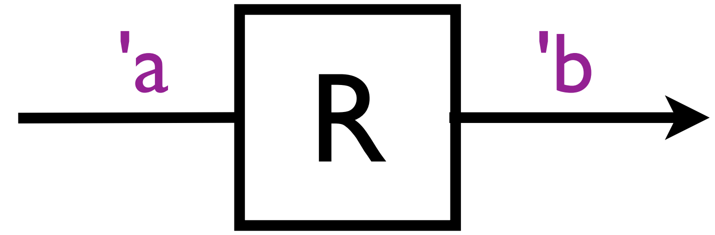
Lambda abstraction:



Lambda abstraction:

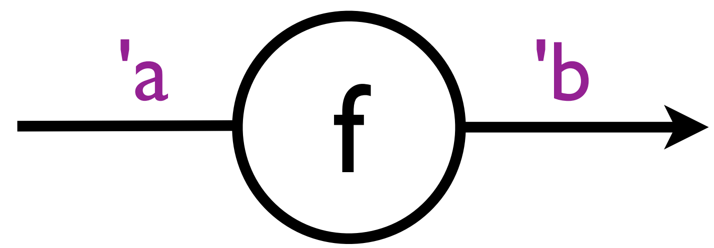


Reagent abstraction:

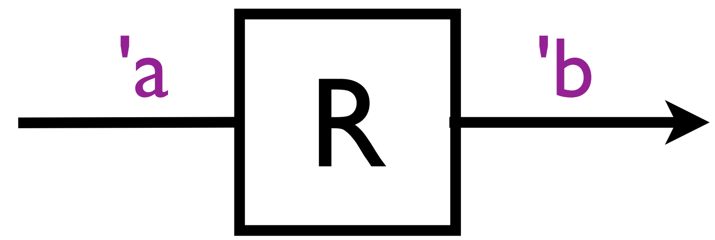



```
('a,'b) Reagent.t
```

Lambda abstraction:



Reagent abstraction:




```
('a,'b) Reagent.t

val run : ('a,'b) Reagent.t -> 'a -> ‘b
```

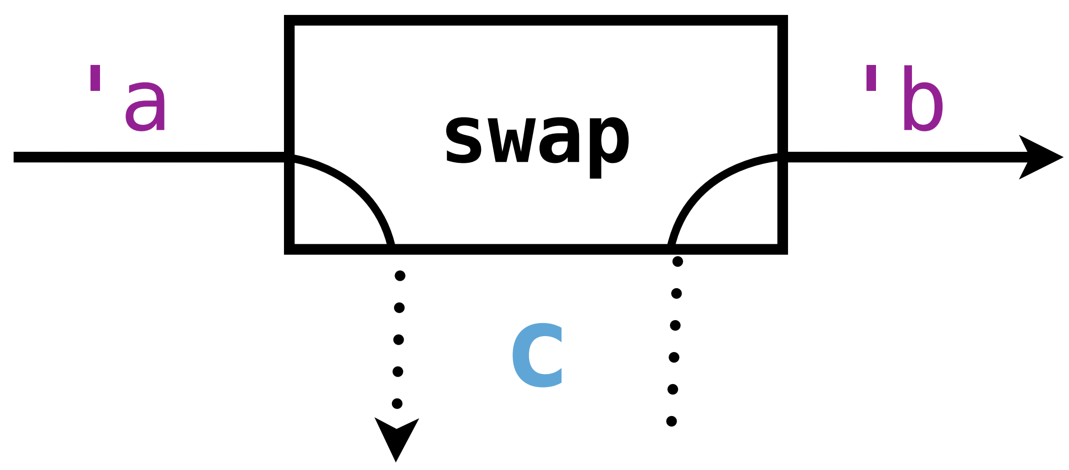
Thread Interaction

```
module type Reagents = sig
  type ('a,'b) t

  (* shared memory *)
  module Ref : Ref.S with type ('a,'b) reagent = ('a,'b) t
  (* communication channels *)
  module Channel : Channel.S with type ('a,'b) reagent = ('a,'b) t
  ...
end
```

```
module type Channel = sig
  type ('a,'b) endpoint
  type ('a,'b) reagent

  val mk_chan : unit -> ('a,'b) endpoint * ('b,'a) endpoint
  val swap    : ('a,'b) endpoint -> ('a,'b) reagent
end
```

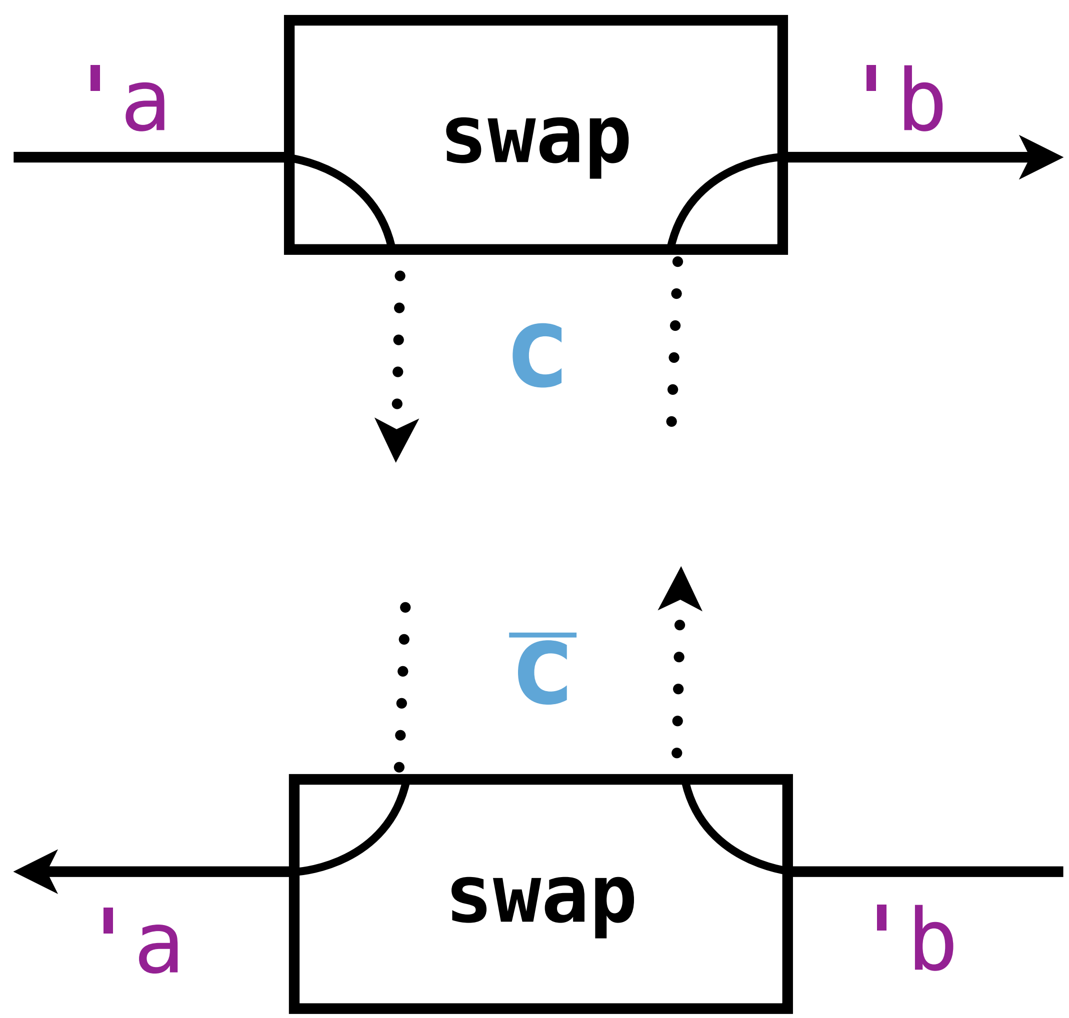
```
module type Channel = sig
  type ('a,'b) endpoint
  type ('a,'b) reagent

  val mk_chan : unit -> ('a,'b) endpoint * ('b,'a) endpoint
  val swap    : ('a,'b) endpoint -> ('a,'b) reagent
end
```

c: ('a,'b) endpoint

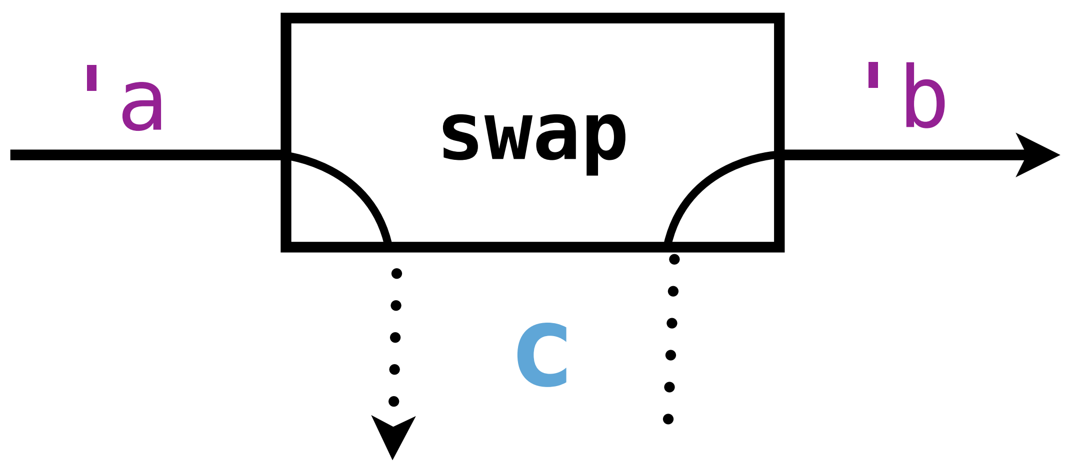


```
module type Channel = sig
  type ('a,'b) endpoint
  type ('a,'b) reagent

  val mk_chan : unit -> ('a,'b) endpoint * ('b,'a) endpoint
  val swap    : ('a,'b) endpoint -> ('a,'b) reagent
end
```

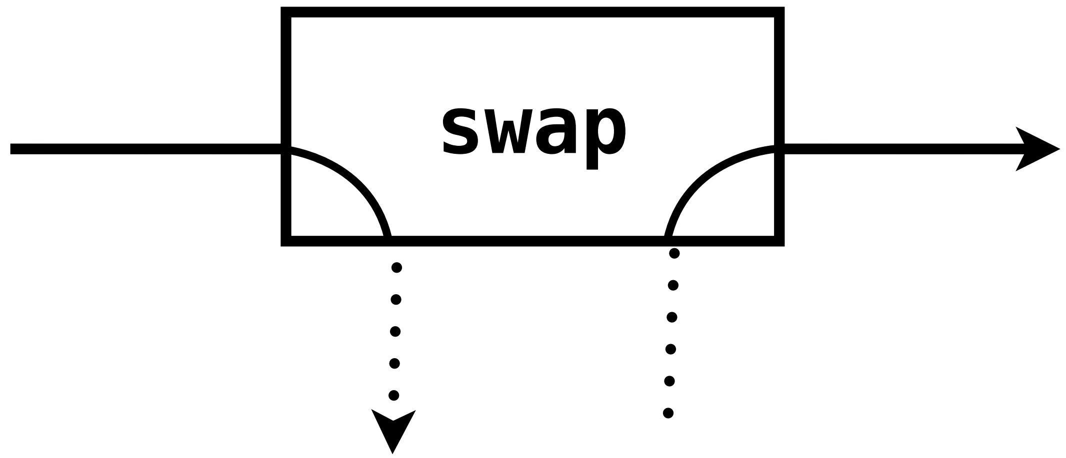
c: ('a,'b) endpoint



`c: ('a,'b) endpoint`

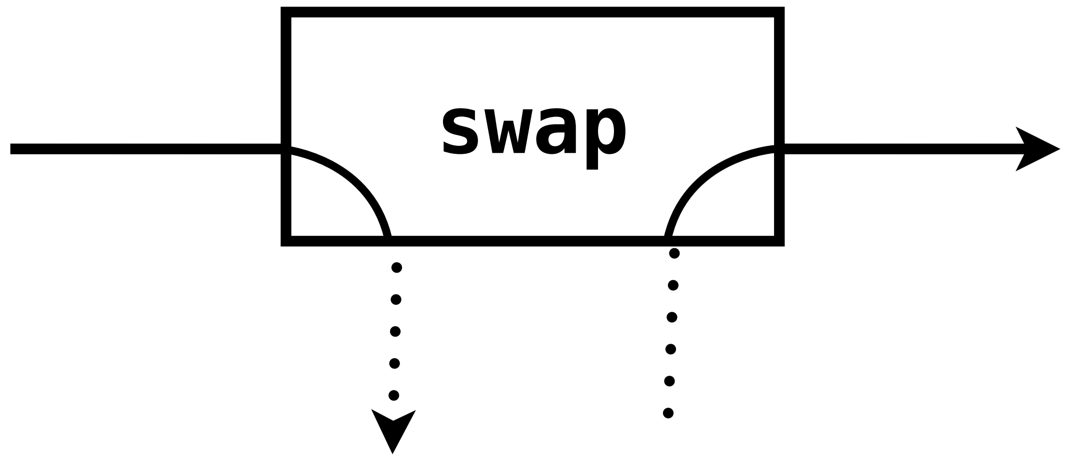


Message passing

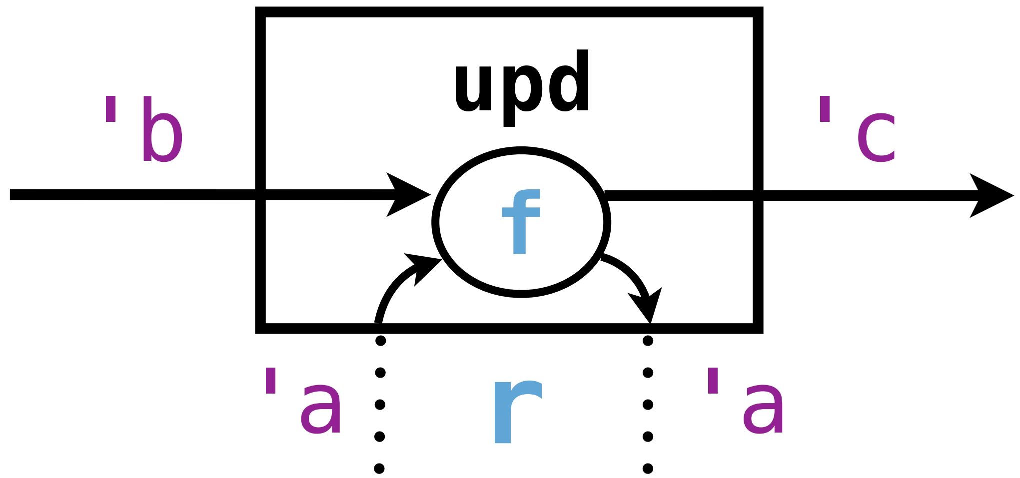



```
type 'a ref
val upd : 'a ref
  -> f:(‘a -> 'b -> ('a * ‘c) option)
  -> ('b, 'c) Reagent.t
```

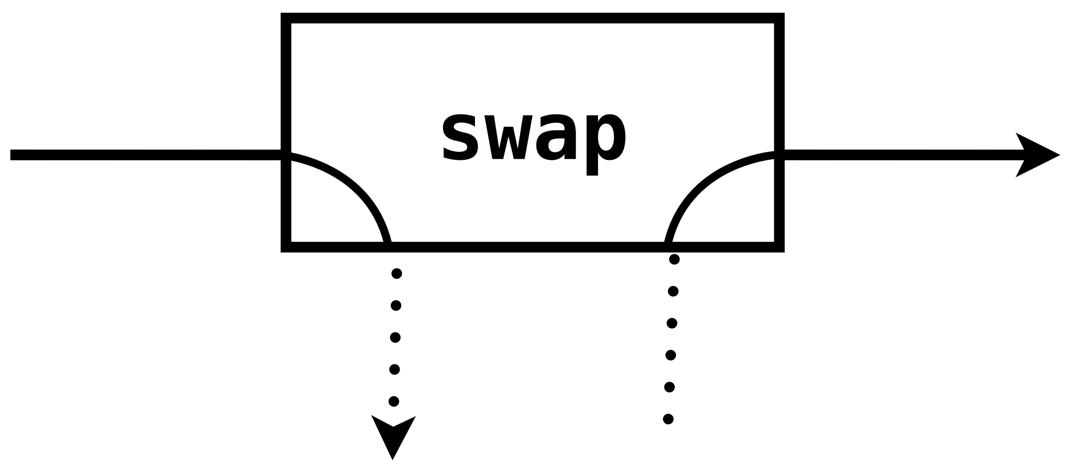
Message passing



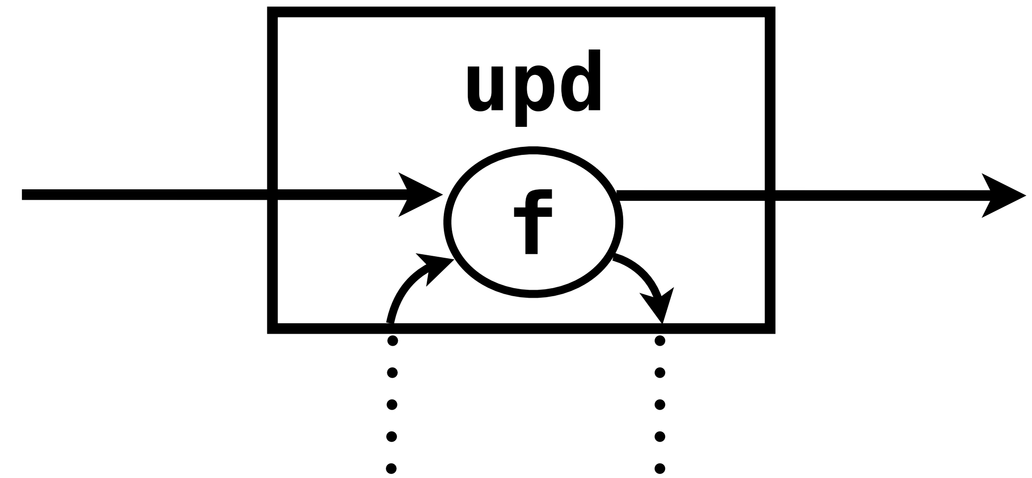

```
type 'a ref
val upd : 'a ref
  -> f:(‘a -> 'b -> ('a * ‘c) option)
  -> ('b, 'c) Reagent.t
```



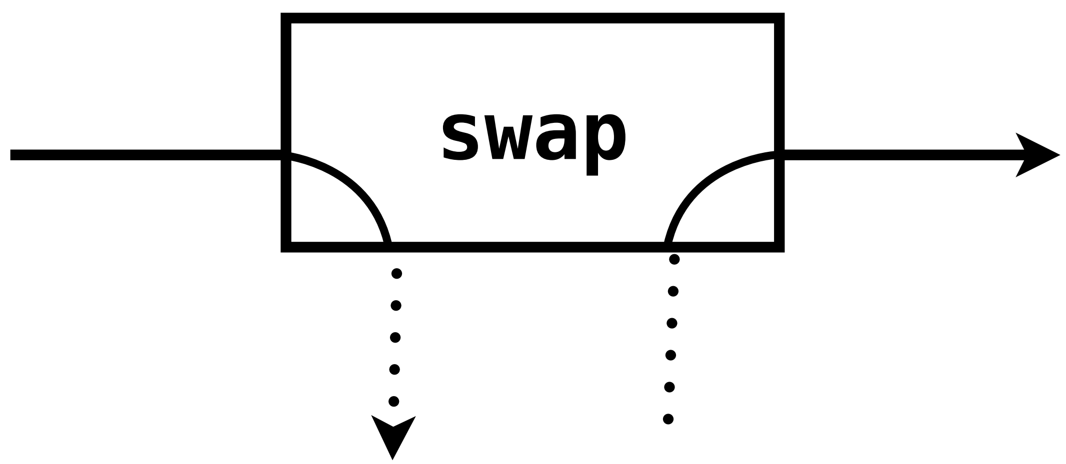
Message passing

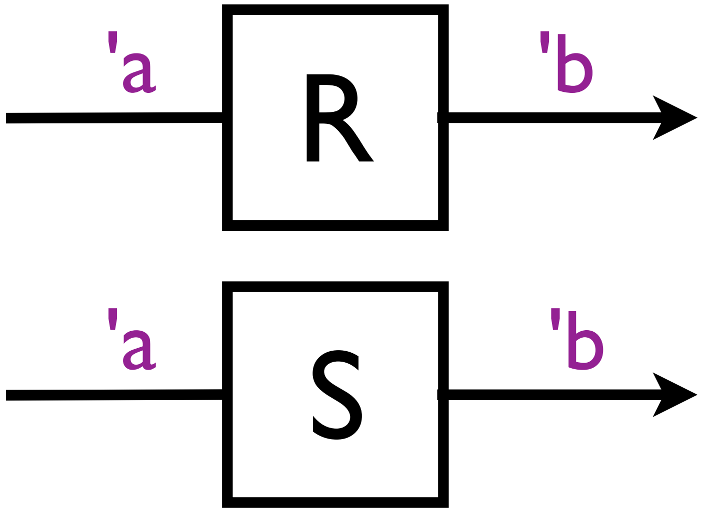


Shared state

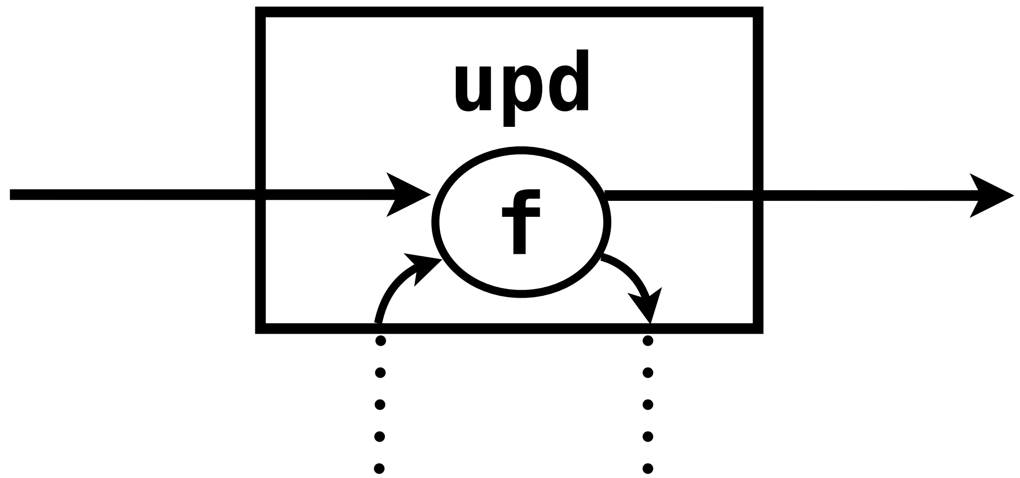


Message passing

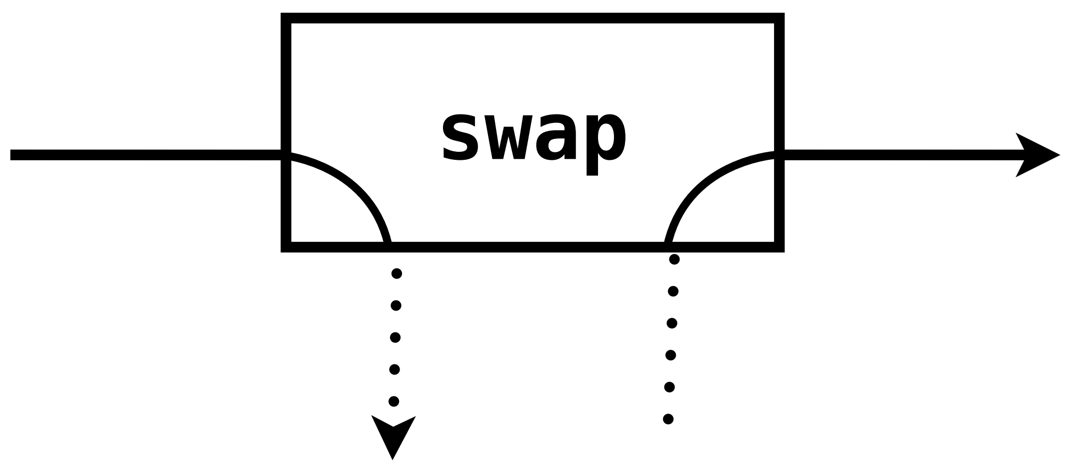


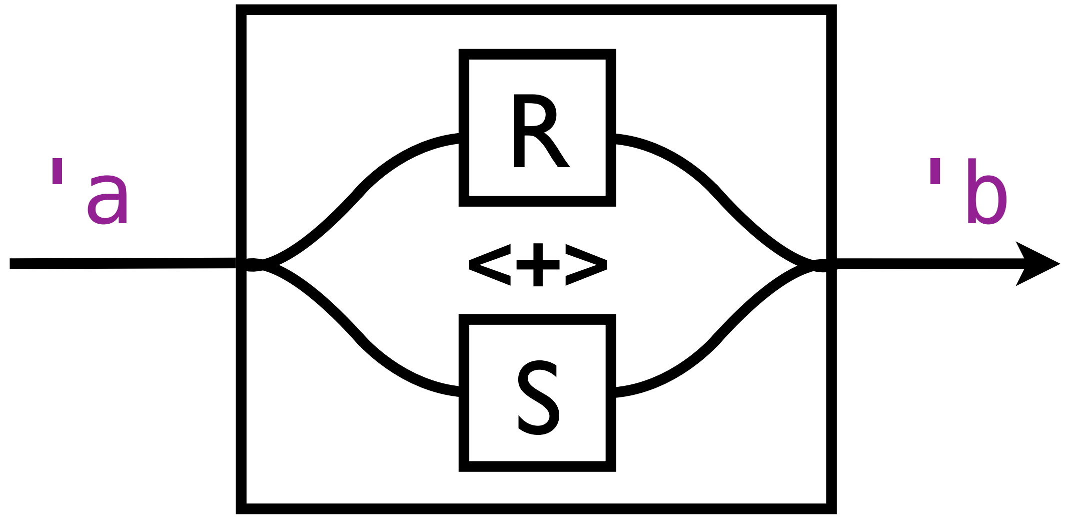


Shared state

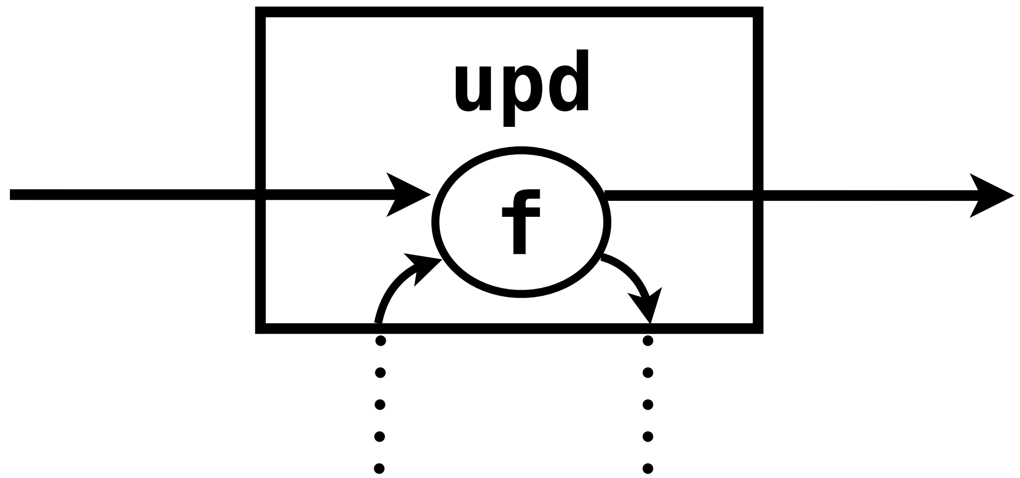


Message passing

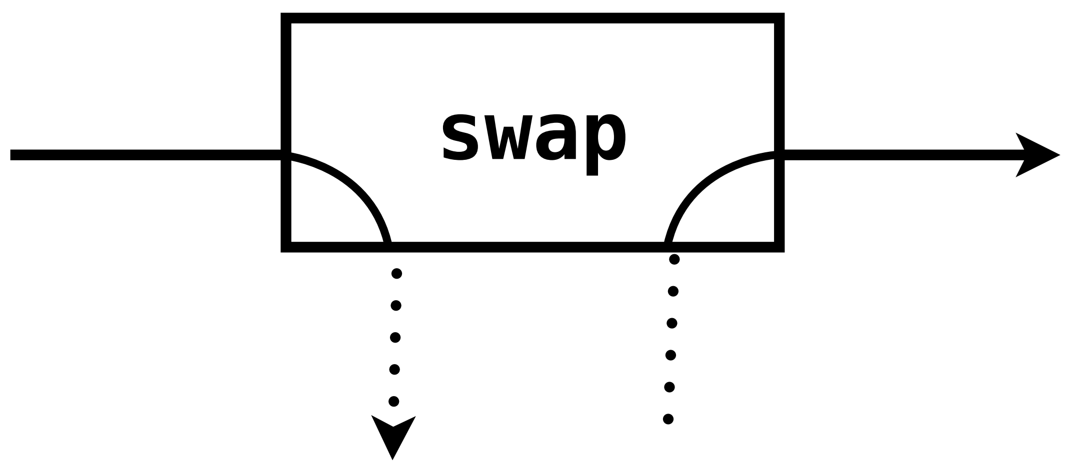




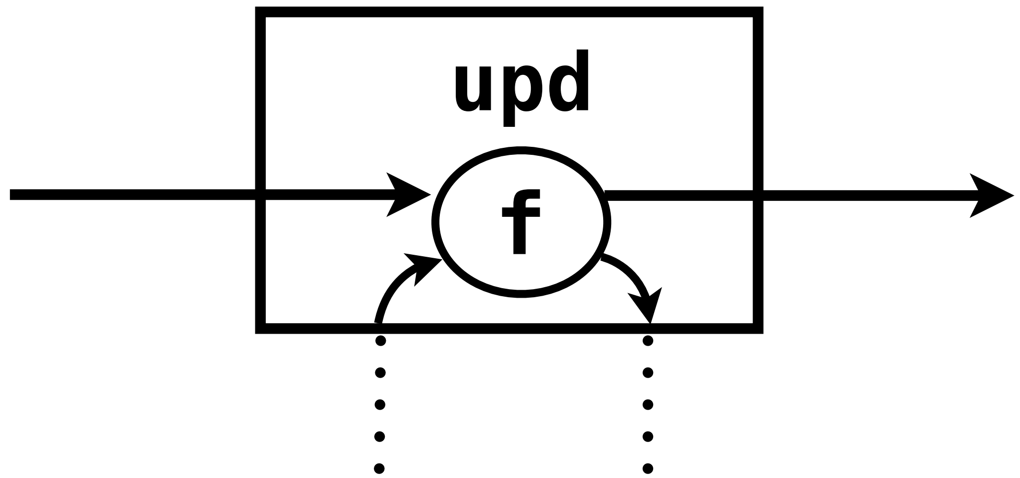
Shared state



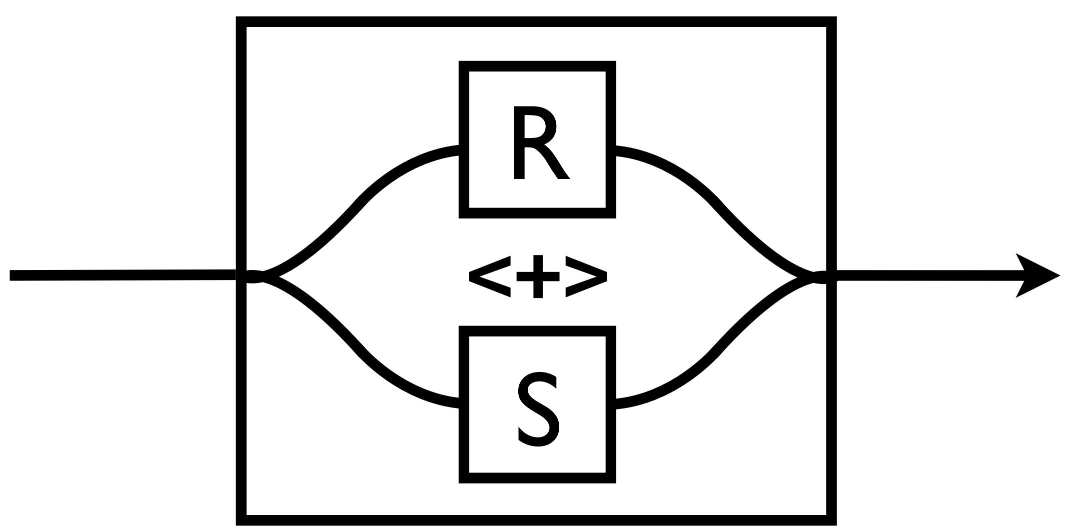
Message passing



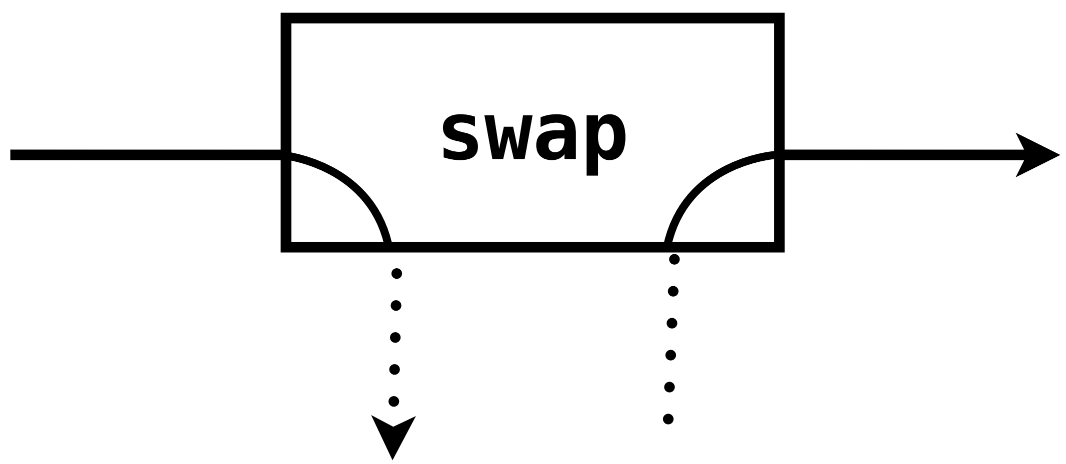
Shared state



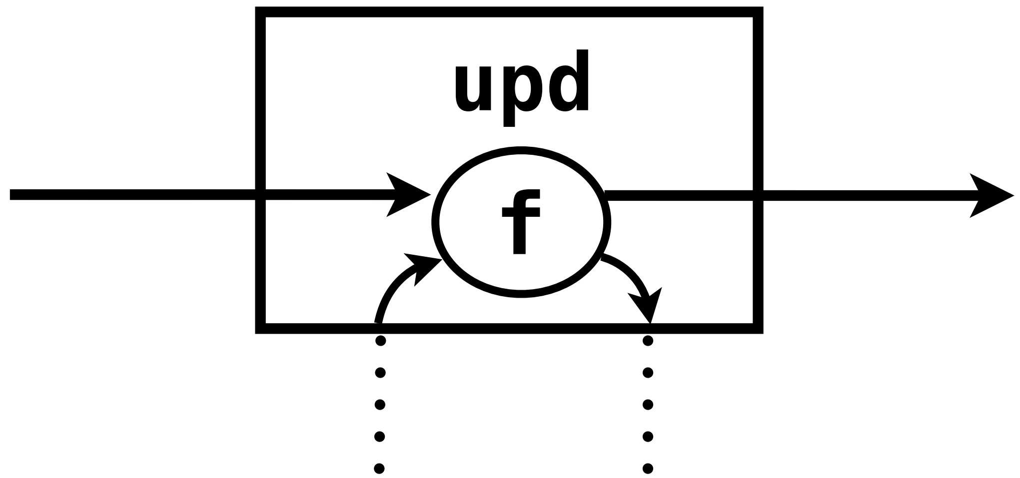
Disjunction



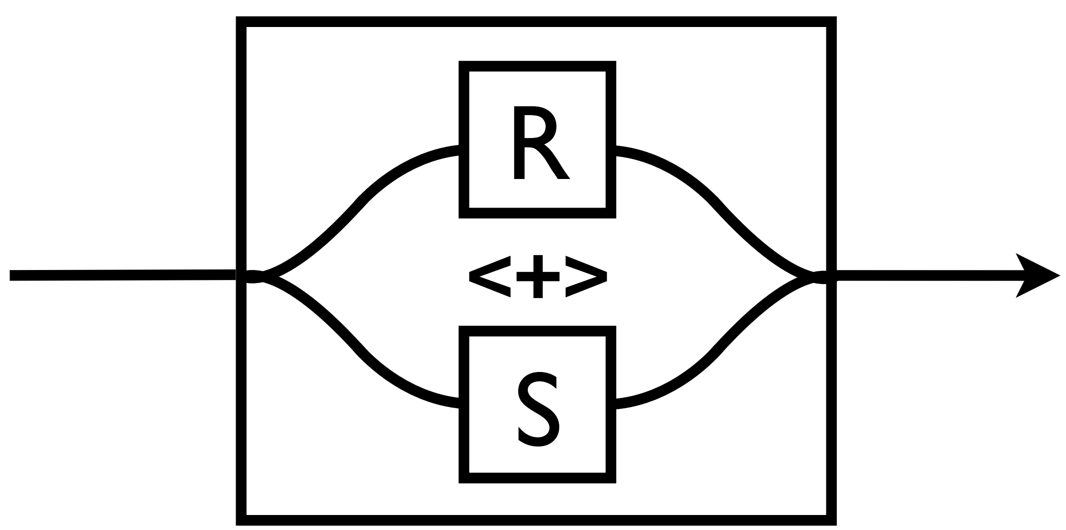
Message passing

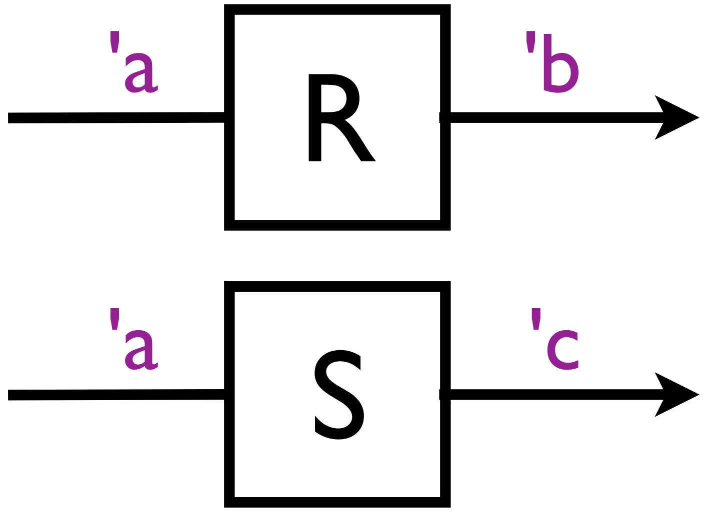


Shared state

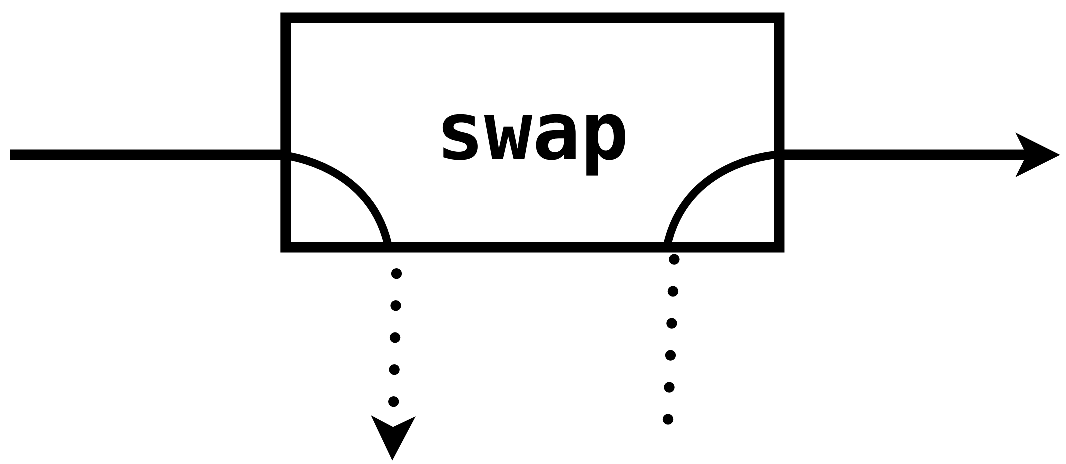


Disjunction

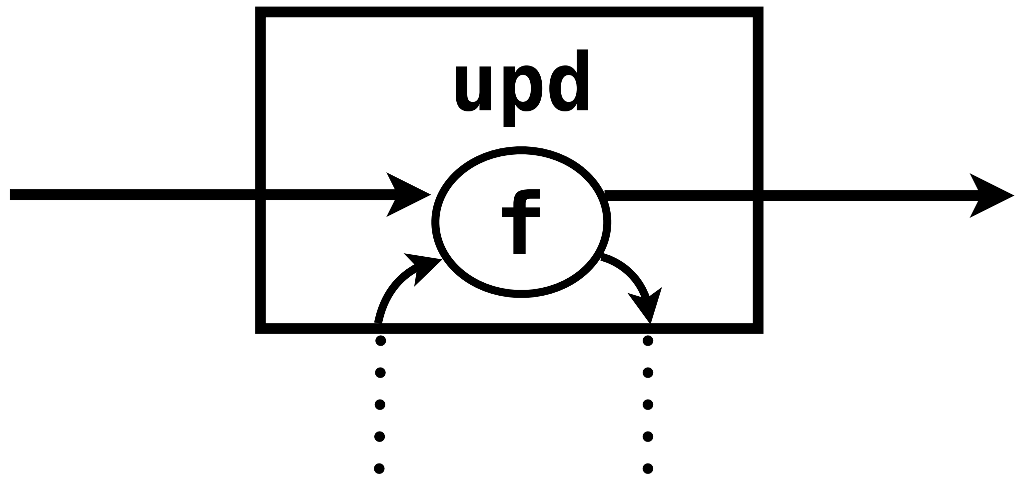




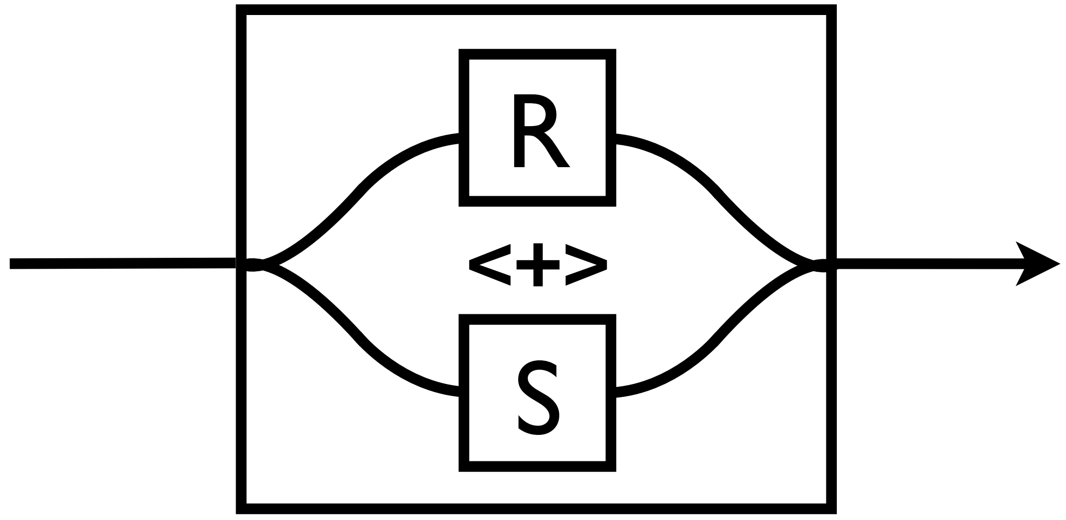
Message passing

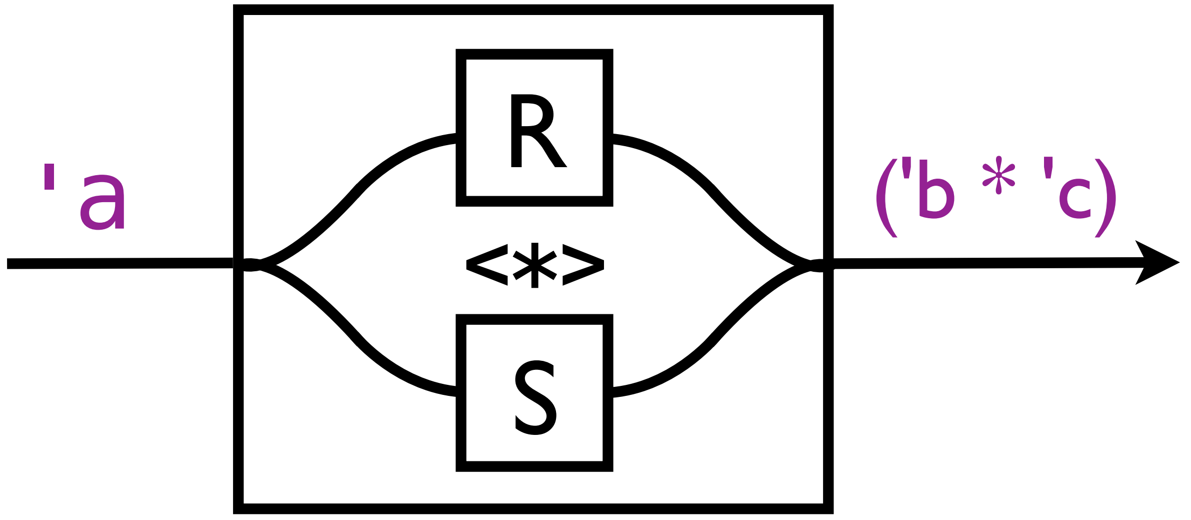


Shared state

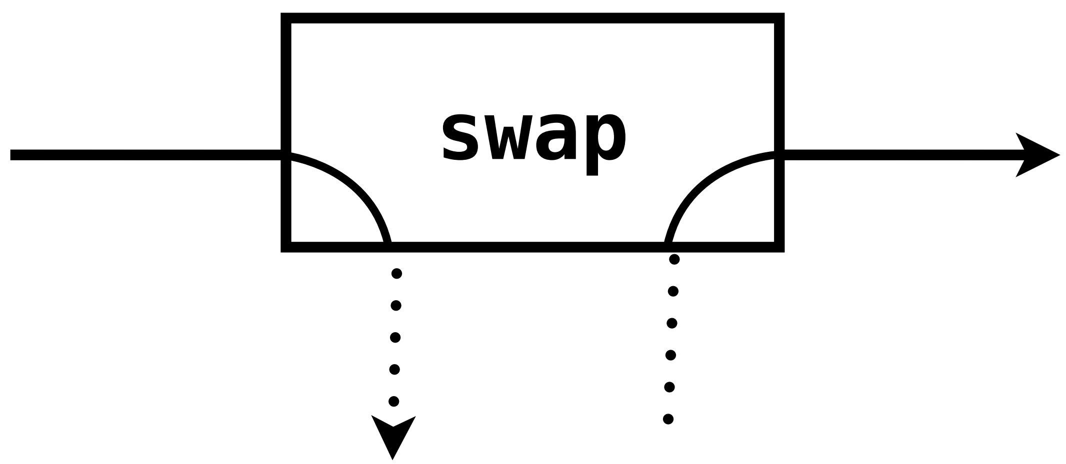


Disjunction

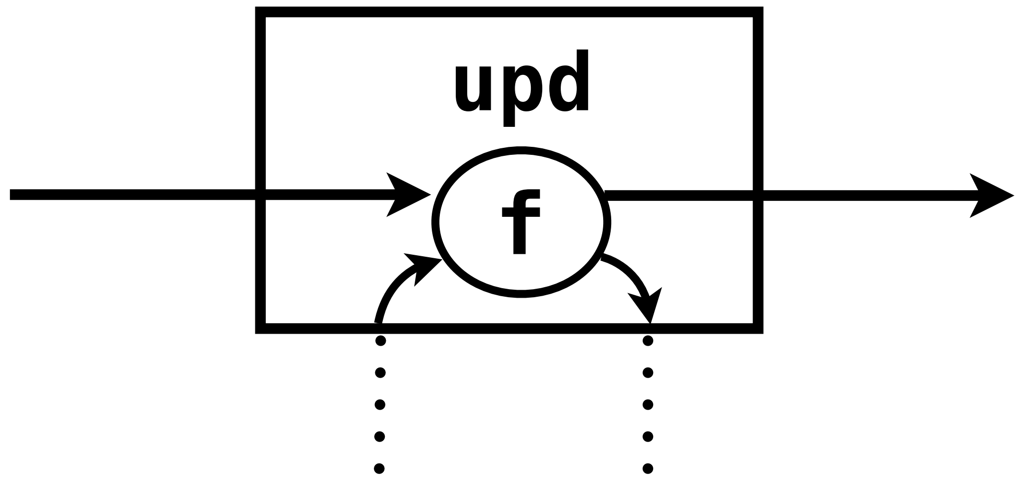




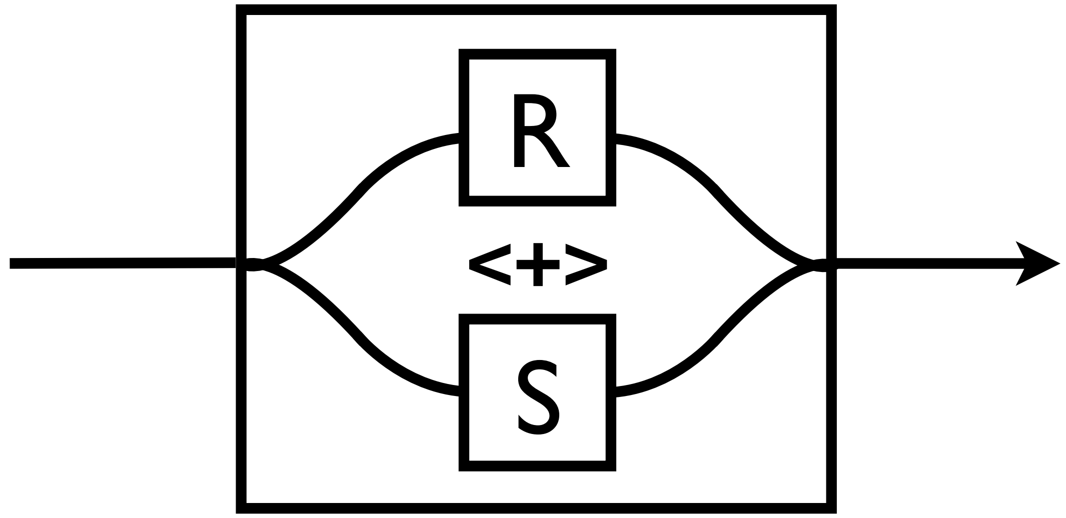
Message passing



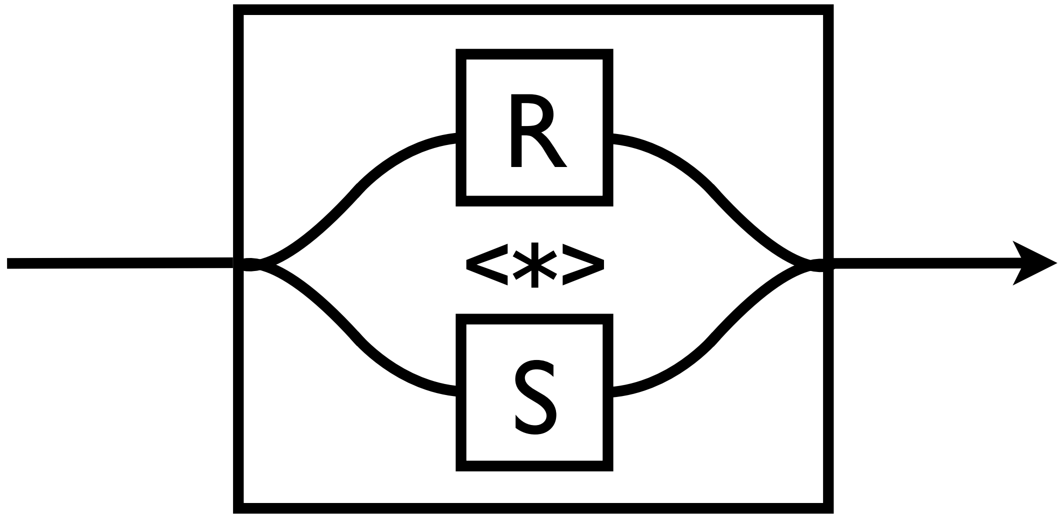
Shared state



Disjunction



Conjunction



```
module type TREIBER_STACK = sig
  type 'a t
  val create  : unit -> 'a t
  val push    : 'a t -> ('a, unit) Reagent.t
  val pop     : 'a t -> (unit, 'a) Reagent.t
  ...
end

module Treiber_stack : TREIBER_STACK = struct
  type 'a t = 'a list Ref.ref

  let create () = Ref.ref []

  let push r x = Ref.upd r (fun xs x -> Some (x::xs,()))

  let pop r = Ref.upd r (fun l () ->
    match l with
    | [] -> None (* block *)
    | x::xs -> Some (xs,x))
  ...

end
```

Composability

Transfer elements atomically

```
Treiber_stack.pop s1 >>> Treiber_stack.push s2
```

Composability

Transfer elements atomically

```
Treiber_stack.pop s1 >>> Treiber_stack.push s2
```

Consume elements atomically

```
Treiber_stack.pop s1 <*> Treiber_stack.pop s2
```

Composability

Transfer elements atomically

```
Treiber_stack.pop s1 >>> Treiber_stack.push s2
```

Consume elements atomically

```
Treiber_stack.pop s1 <*> Treiber_stack.pop s2
```

Consume elements from either

```
Treiber_stack.pop s1 <+> Treiber_stack.pop s2
```

Composability

Transform arbitrary **blocking** reagent to a **non-blocking** reagent

Composability

Transform arbitrary **blocking** reagent to a **non-blocking** reagent

```
val lift     : ('a -> 'b option) -> ('a,'b) t
val constant : 'a -> ('b,'a) t
```

Composability

Transform arbitrary **blocking** reagent to a **non-blocking** reagent

```
val lift     : ('a -> 'b option) -> ('a,'b) t
val constant : 'a -> ('b,'a) t
```

```
let attempt (r : ('a,'b) t) : ('a,'b option) t =
 (r >>> lift (fun x -> Some (Some x)))
 <+> (constant None)
```

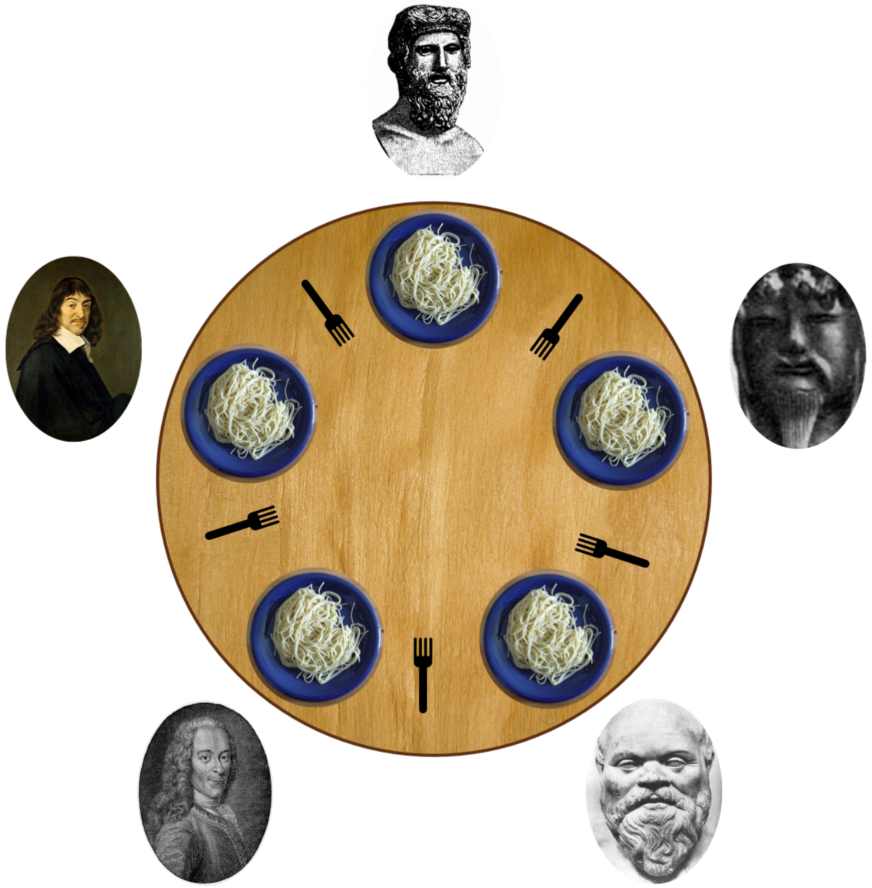
Composability

Transform arbitrary **blocking** reagent to a **non-blocking** reagent

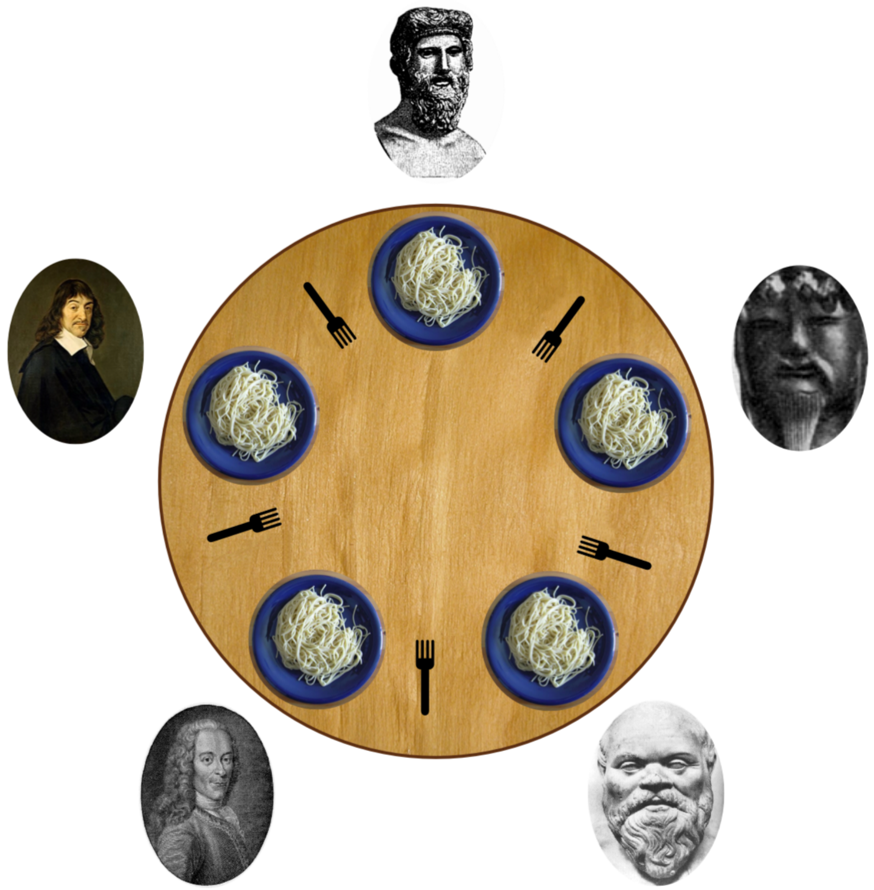
```
val lift     : ('a -> 'b option) -> ('a,'b) t
val constant : 'a -> ('b,'a) t
```

```
let attempt (r : ('a,'b) t) : ('a,'b option) t =
 (r >>> lift (fun x -> Some (Some x)))
 <+> (constant None)
```

```
let try_pop stack = attempt (pop stack)
```

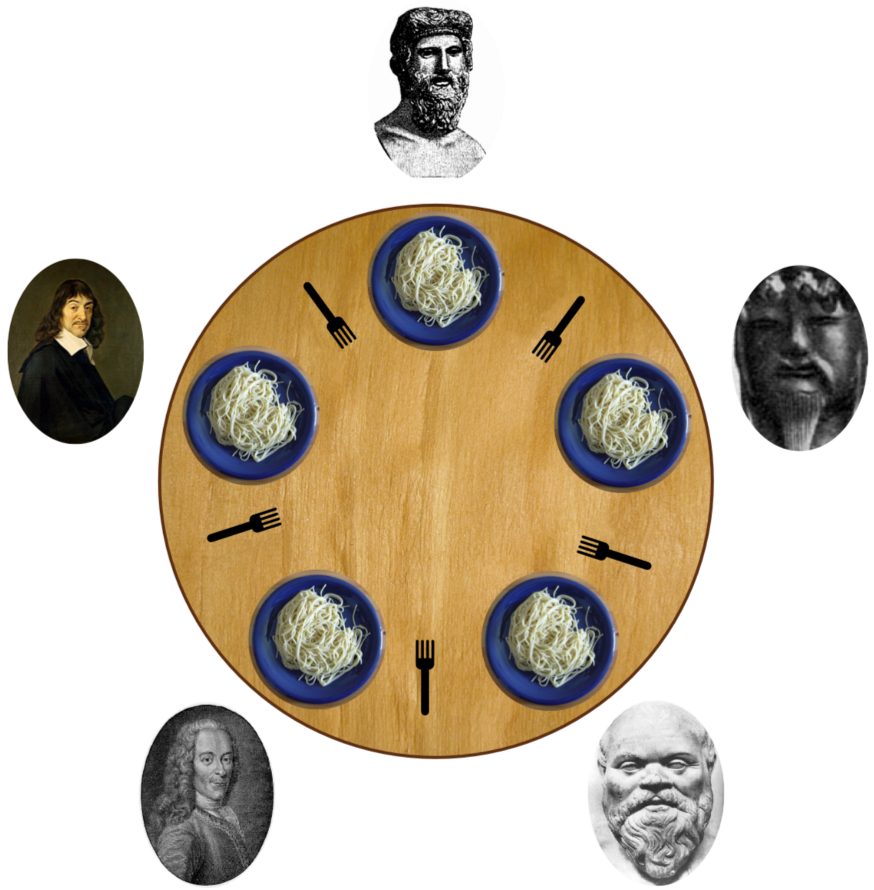


- Philosopher’s alternate between thinking and eating
- Philosopher can only eat after obtaining both forks
- No philosopher starves



- Philosopher's alternate between thinking and eating
- Philosopher can only eat after obtaining both forks
- No philosopher starves

```
type fork =
  {drop : (unit,unit) endpoint;
   take : (unit,unit) endpoint}

let mk_fork () =
  let drop, take = mk_chan () in
  {drop; take}

let drop f = swap f.drop
let take f = swap f.take
```

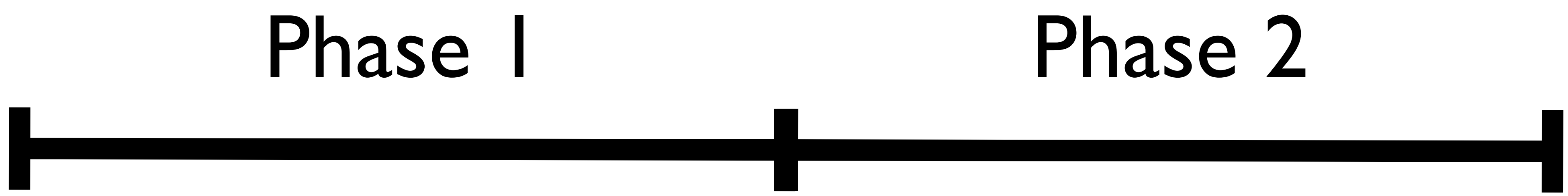


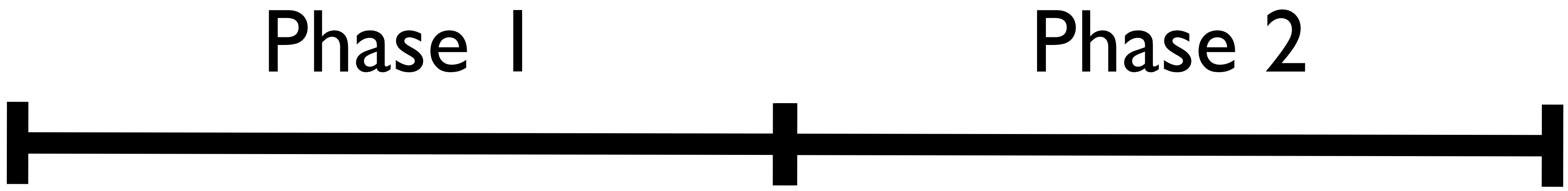
- Philosopher's alternate between thinking and eating
- Philosopher can only eat after obtaining both forks
- No philosopher starves

```
type fork =
  {drop : (unit,unit) endpoint;
   take : (unit,unit) endpoint}

let mk_fork () =
  let drop, take = mk_chan () in
  {drop; take}

let drop f = swap f.drop
let take f = swap f.take
```

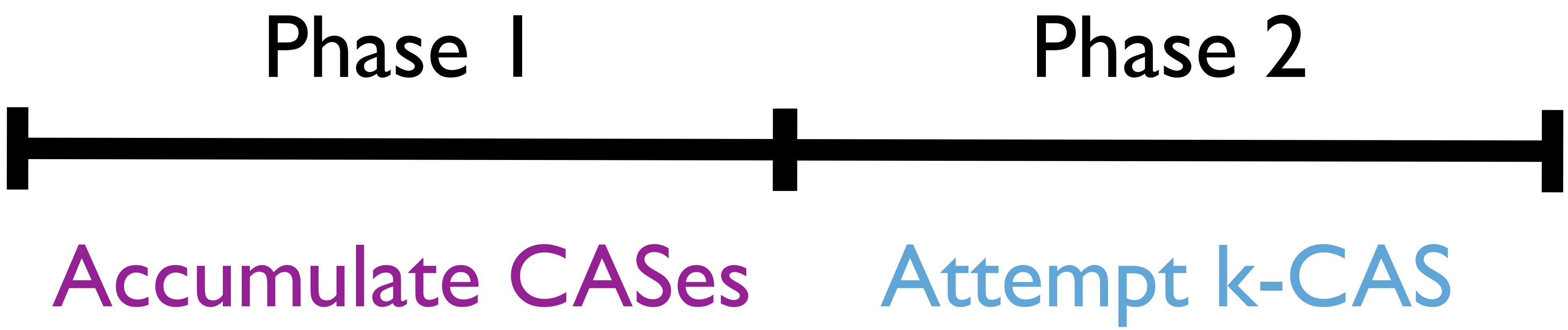
```
let eat l_fork r_fork =
  run (take l_fork <*>
       take r_fork) ();
  (* ...
   * eat
   * ... *)
  spawn @@ run (drop l_fork);
  spawn @@ run (drop r_fork)
```

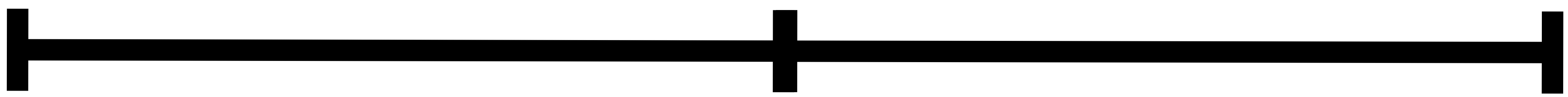
Implementation


Phase 1
Phase 2

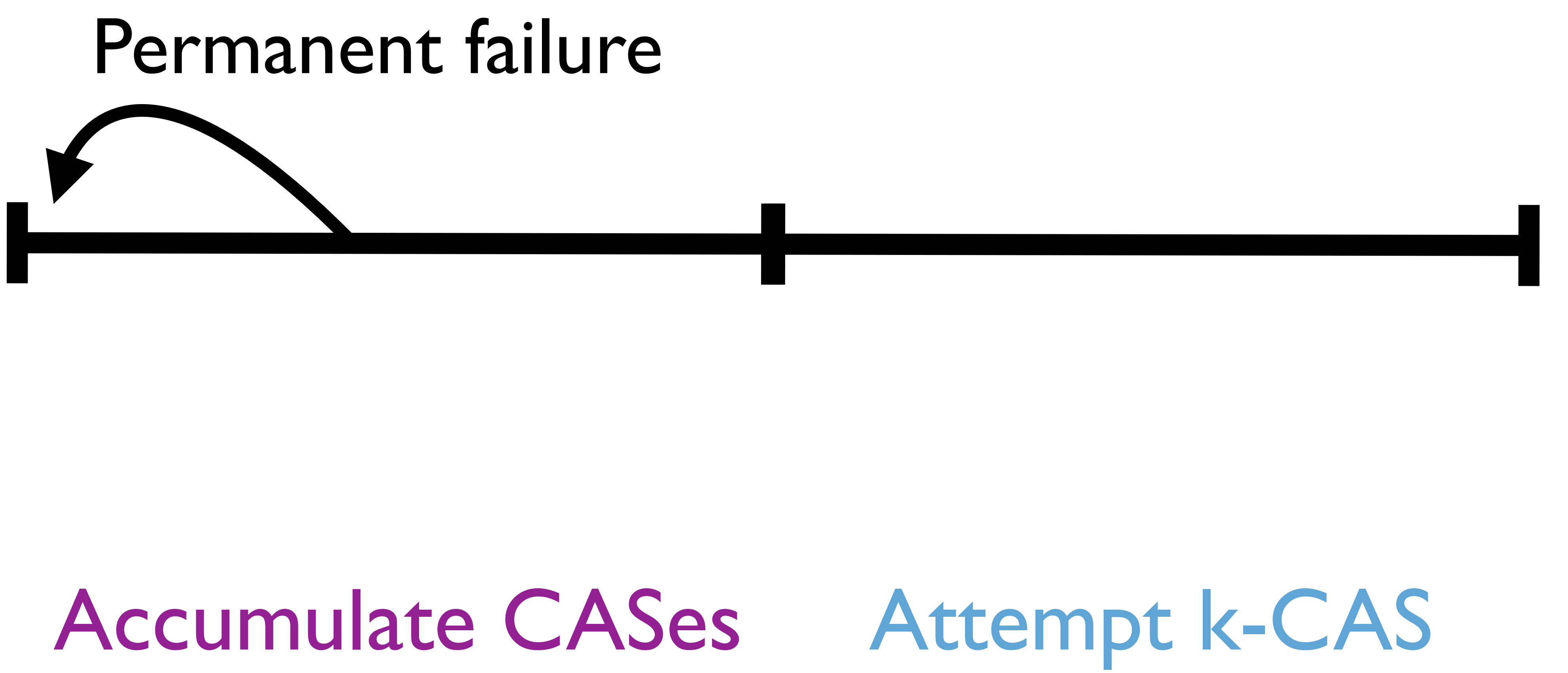


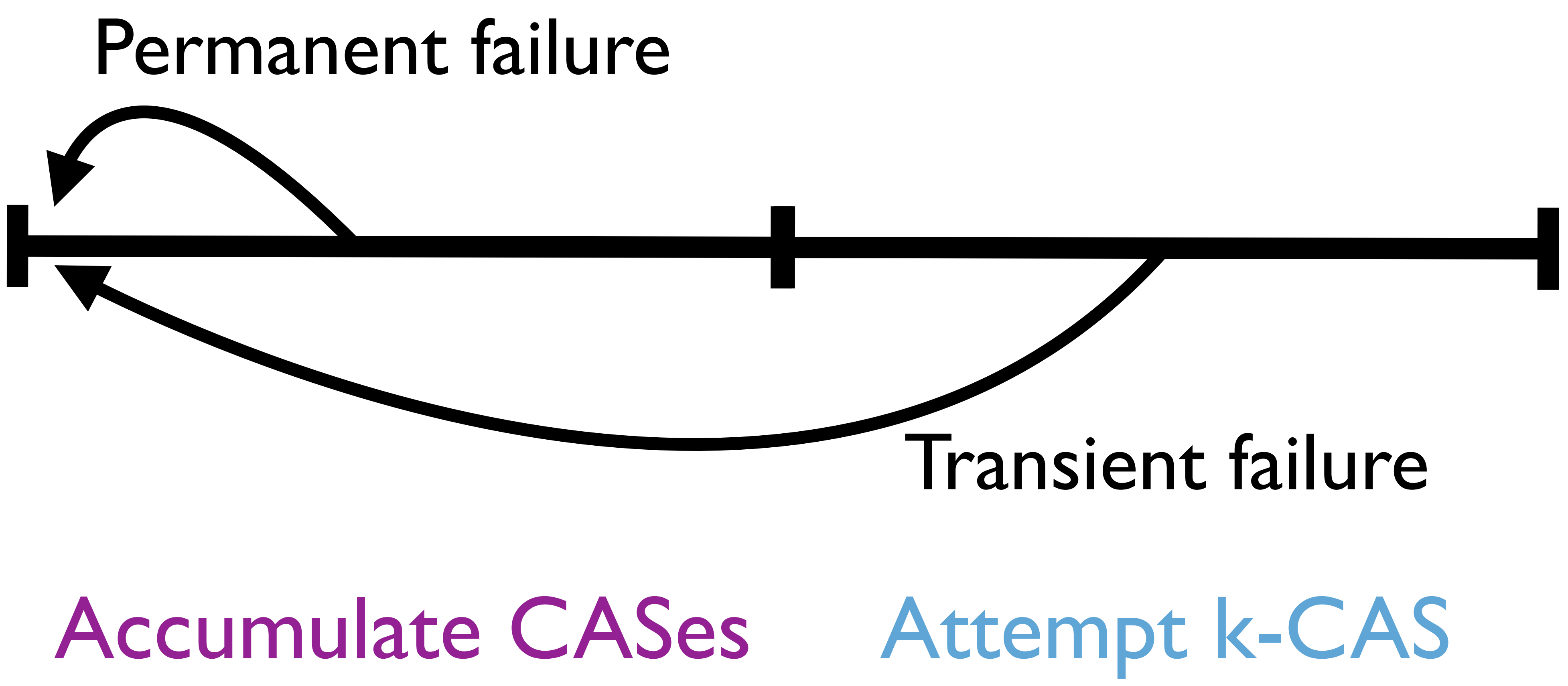
Accumulate CASes


Phase 1
Phase 2
Accumulate CASes
Attempt k-CAS

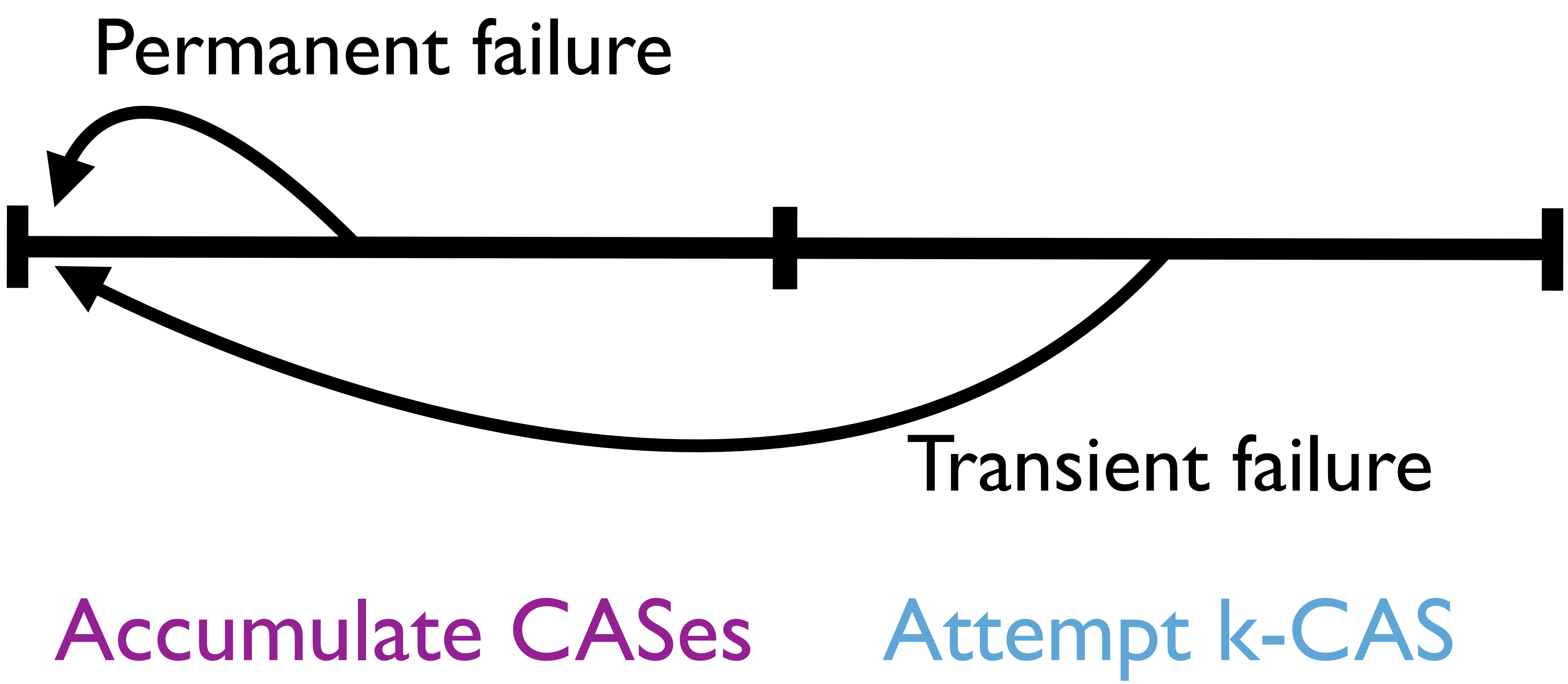


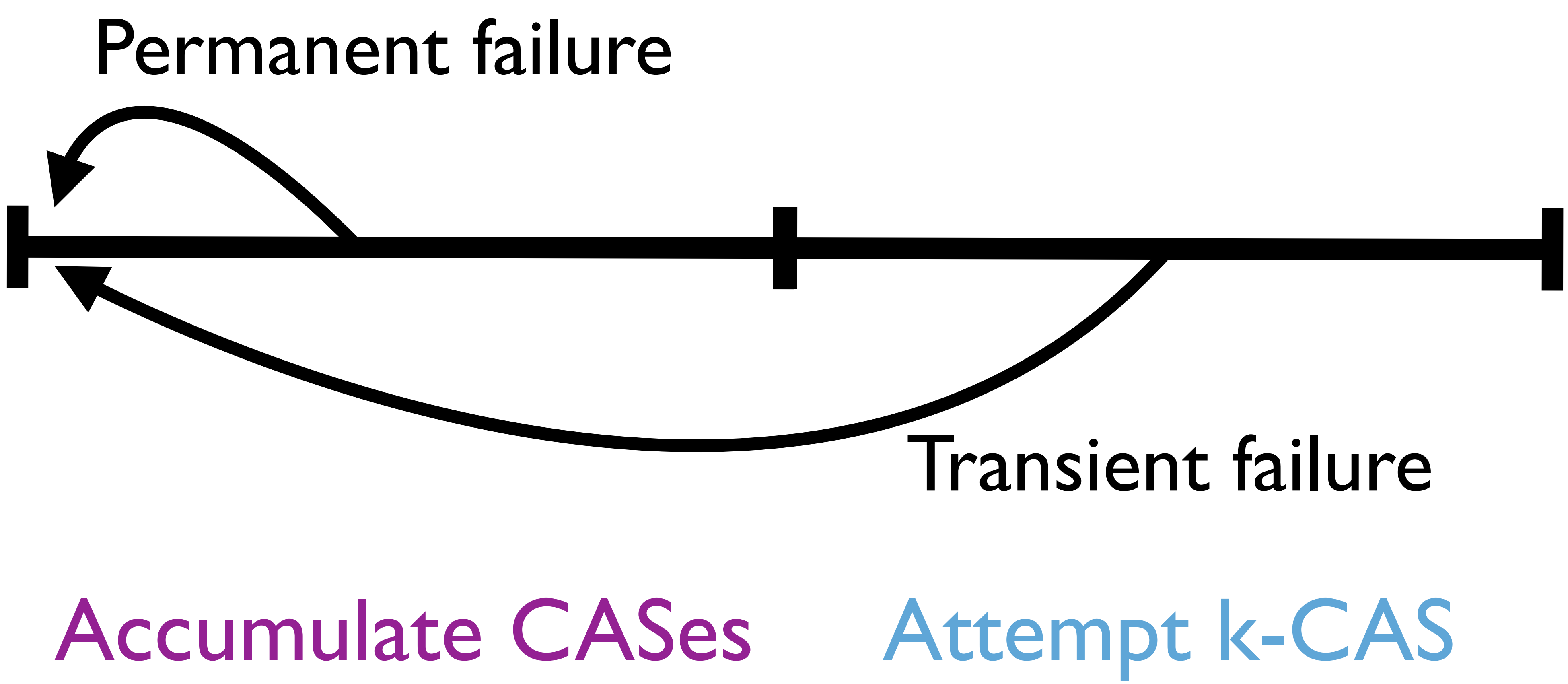
Accumulate CASes Attempt k-CAS


Permanent failure
Accumulate CASes
Attempt k-CAS

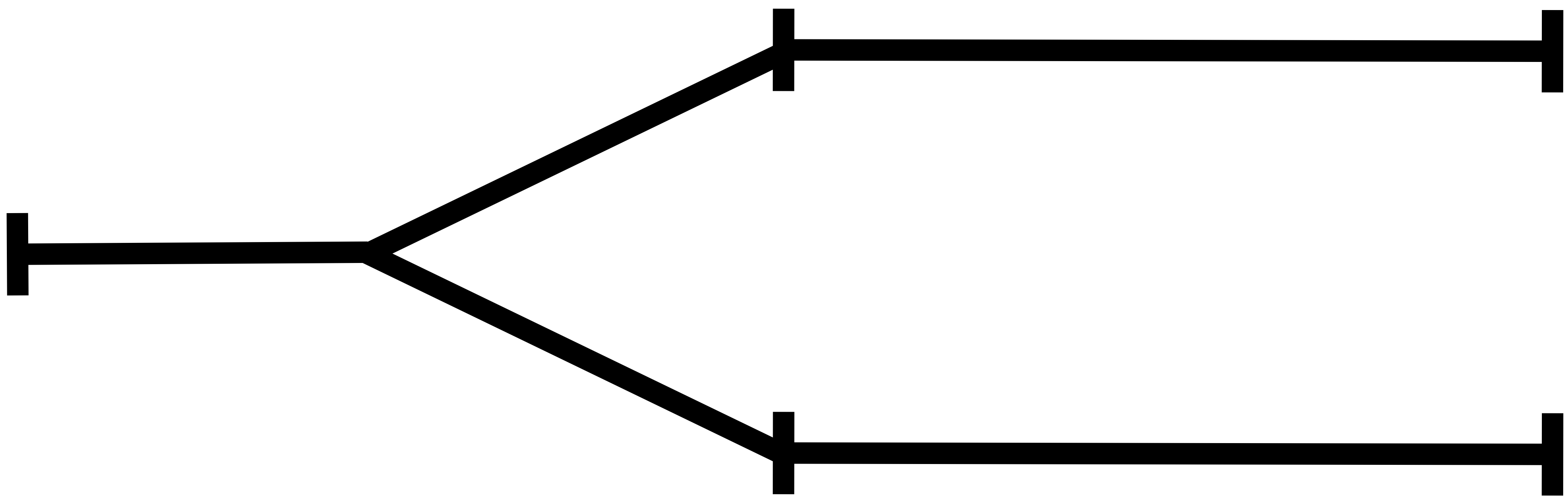

Permanent failure
Transient failure
Accumulate CASes
Attempt k-CAS

HTM Ready

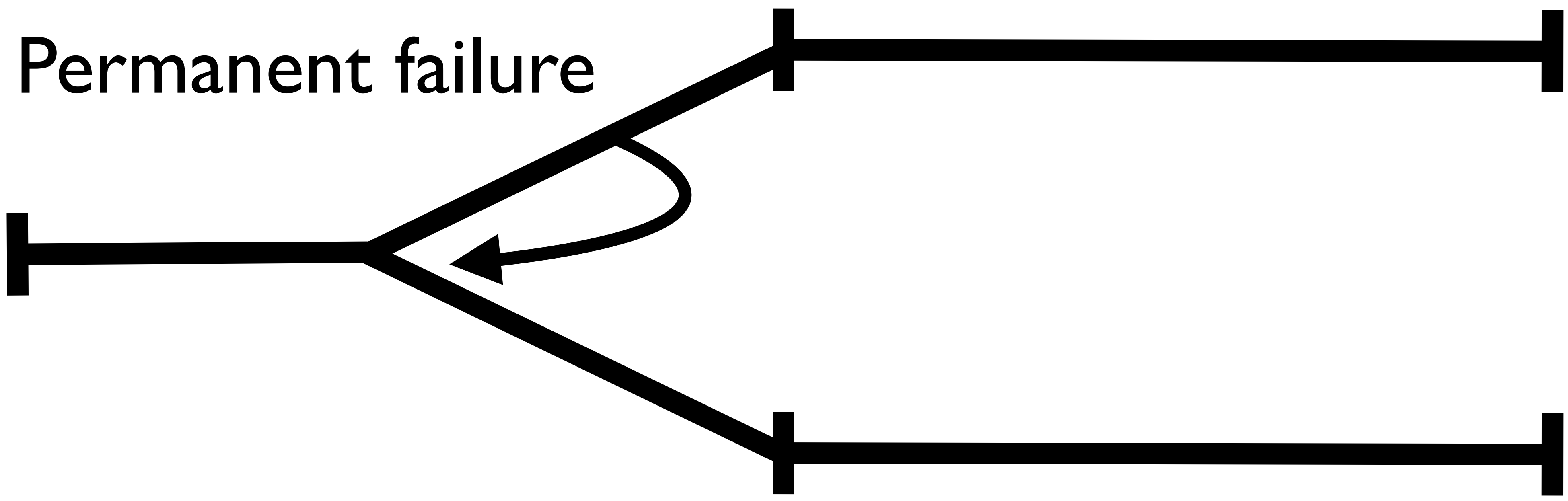

Permanent failure
Transient failure
Accumulate CASes
Attempt k-CAS



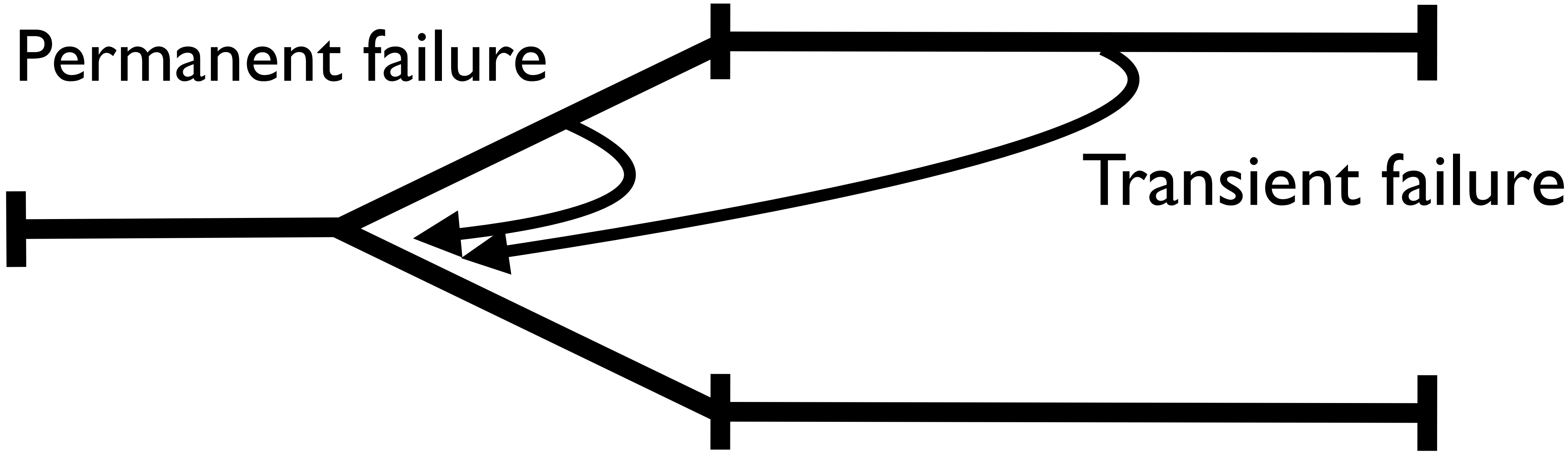
Promising early results with Intel TSX!

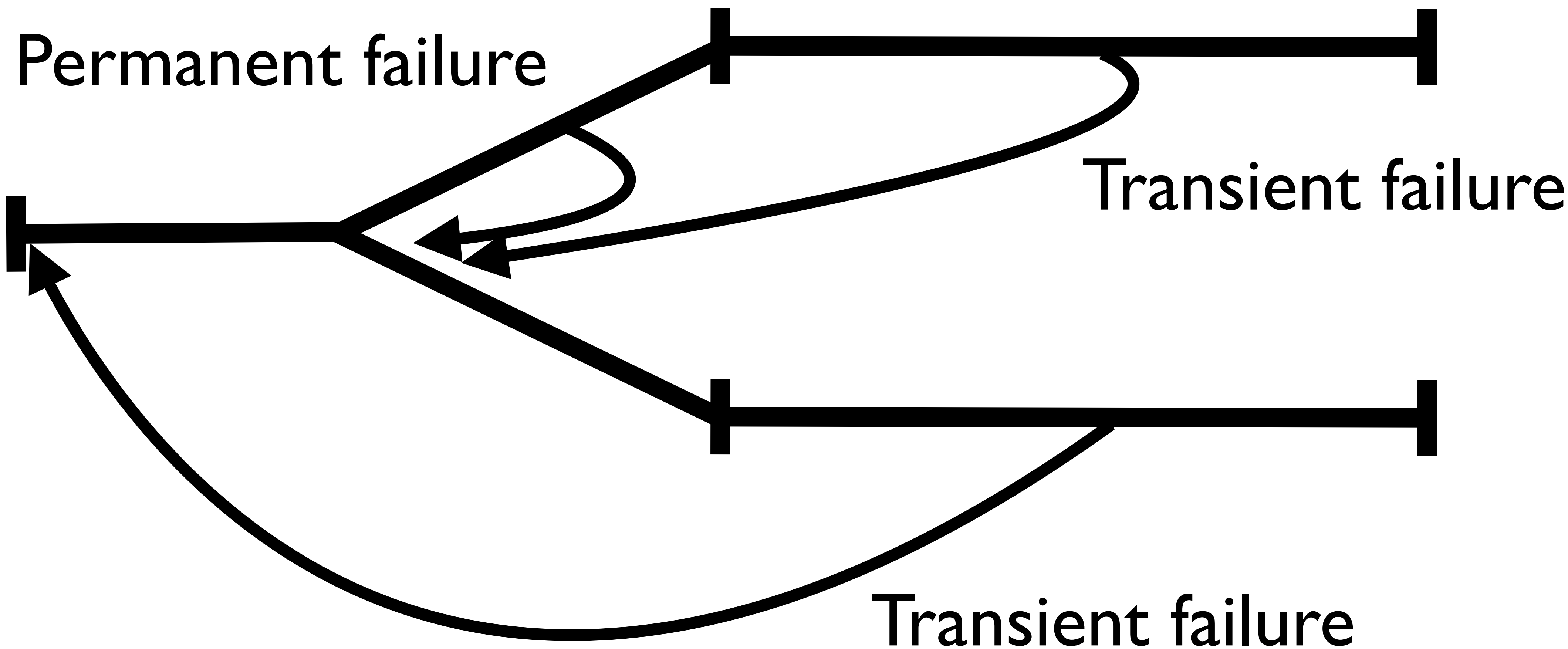


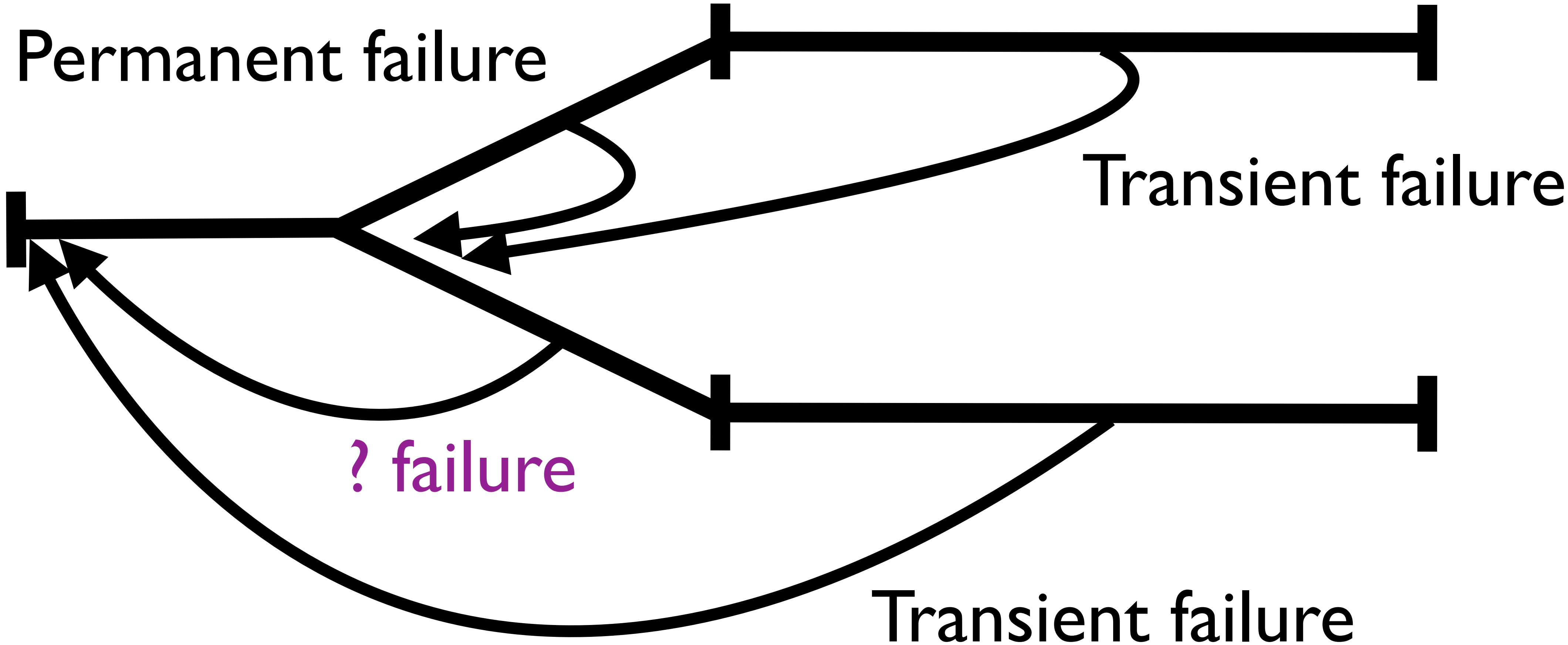
x



x

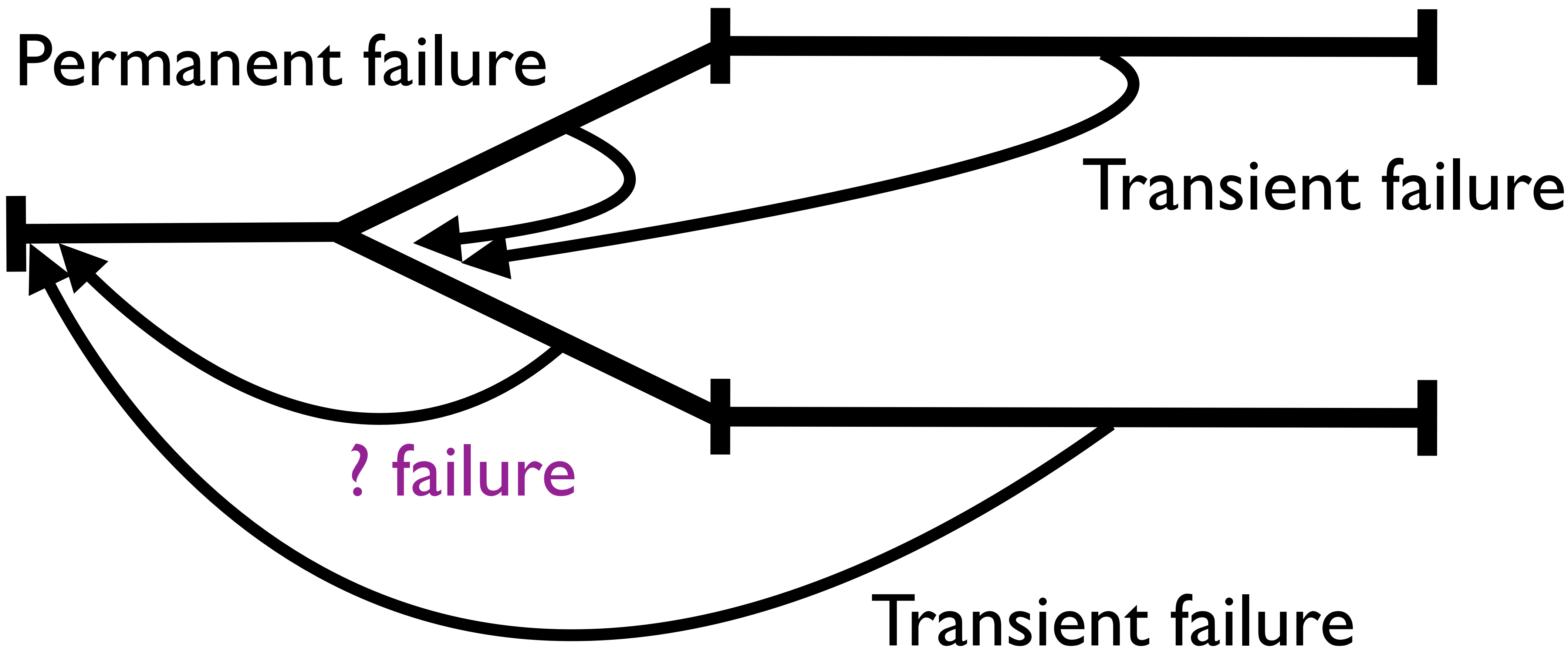

Permanent failure
Transient failure


Permanent failure
Transient failure
Transient failure



Permanent failure
Transient failure
? failure
Transient failure

x



P & P = P   P & T = T

T & T = T   T & P = T

x

Status

Synchronization

- Locks
- 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
- Stacks
  - Treiber
  - Elimination backoff
- Counters
- Deques
- Sets
- Maps (hash & skiplist)

https://github.com/ocamllabs/ocaml-multicore

https://github.com/ocamllabs/reagents

Questions?

STM vs Reagents

- STM is more ambitious — atomic { … }. Reagents are conservative.
- Reagents = STM + Communication
- Reagents don't allow multiple writes to the same memory location.
- Reagents are lock-free. STMs are typically obstruction-free.