Arrows and Reagents

“KC” Sivaramakrishnan

Advanced Functional Programming

March 3rd, 2016

Arrows

```
module type Arrow =
sig
  type ('a,'b) t
  val arr   : ('a -> ‘b) -> ('a,'b) t
  val (>>>) : ('a,'b) t -> ('b,'c) t -> ('a,'c) t
  val first : ('a,'b) t -> ('a * 'c, 'b * 'c) t
end
```

Laws

```
arr f >>> arr g ≡ arr (compose g f)
(f >>> g) >>> h ≡ f >>> (g >>> h)
    arr id >>> f ≡ f
             ... ...
```

Functions
as
Arrows

"If we think of a library as defining a domain specific '**language**', whose constructions are represented as **combinators**, then the idea is to implement the language via a combination of a **static analysis** and **an optimised dynamic semantics**."

John Huges, "Generalising Monads to Arrows"

```
val (>>=) : 'a Monad.t -> ('a -> 'b Monad.t) -> 'b Monad.t
val (>>>) : ('a, 'b) Arrow.t -> ('b,'c) Arrow.t -> ('a,'c) Arrow.t
```

Functions with *cost* as Arrows

Reagents

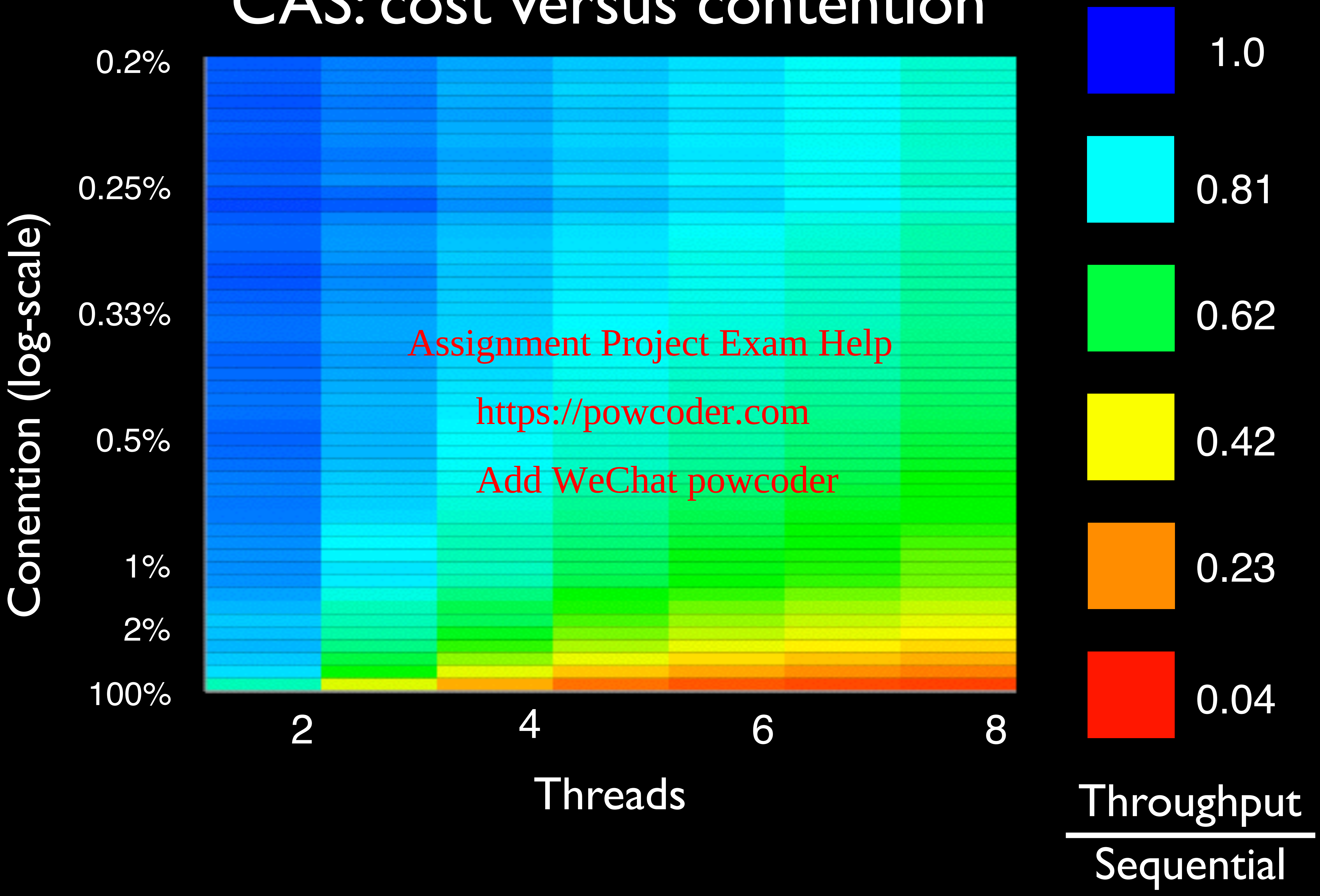
- DSL for e*xpressing* and *composing* fine-grained concurrency libraries
- Aaron Turon, "Reagents: expressing and composing fine-grained concurrency", PLDI 2012
- Based on Arrows
  - Enable dynamic optimisations
- Built on k-compare-and-swap abstraction

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



java.util.concurrent

Synchronization

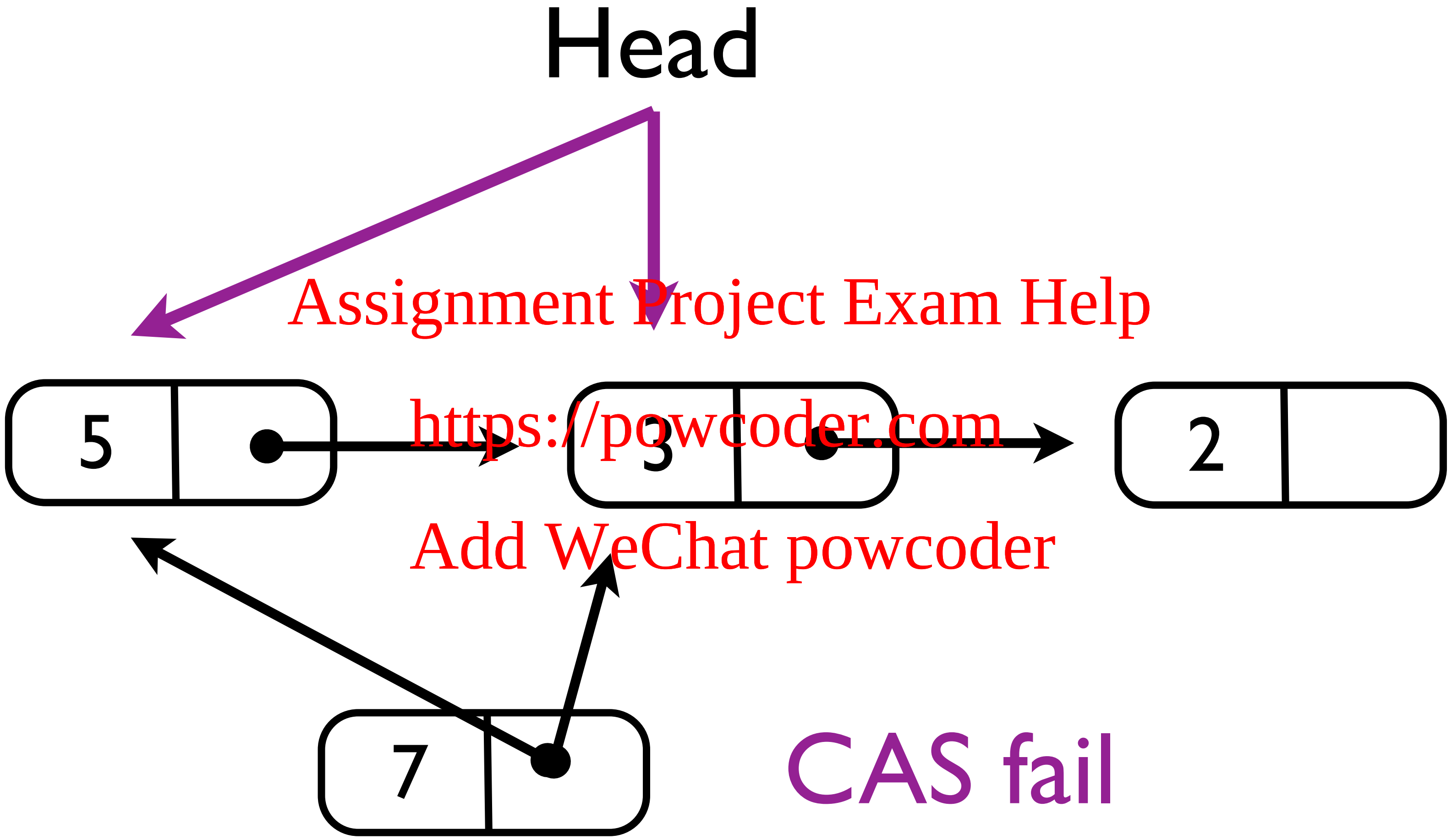
- Reentrant locks
- Semaphores
- R/W locks
- Reentrant R/W lo
- Condition va
- Countdo
- Cycl
- Pl
- Exc ngers

Data s

- Q
- cking
- cking (array & list)
  - Synchronous
  - Priority, nonblocking
  - Priority, blocking
- Deques
- Sets
- Maps (hash & skiplist)

Not Composable

```
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
```



Head

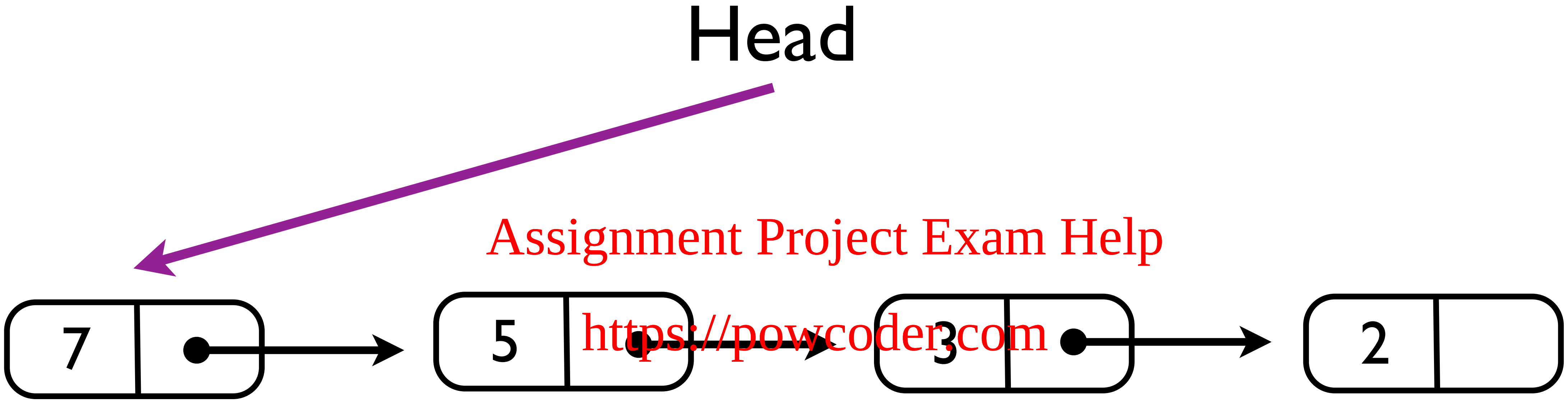
5

3

2

7

CAS fail


Head
7
5
3
2

```
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
```

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***

The Proposal:

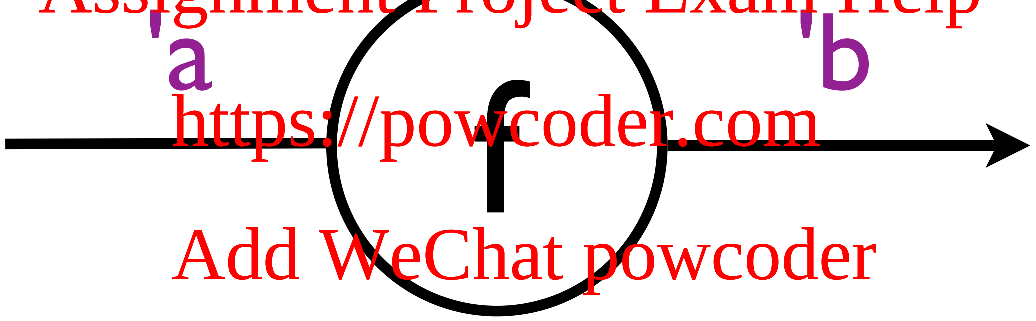
Scalable concurrent algorithms can be built and extended using abstraction and composition

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

is ***atomic***

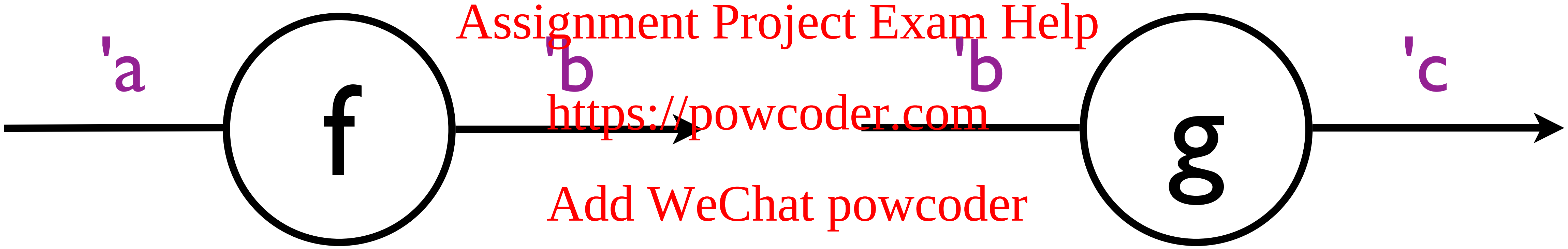
Design

Lambda: the ultimate abstraction



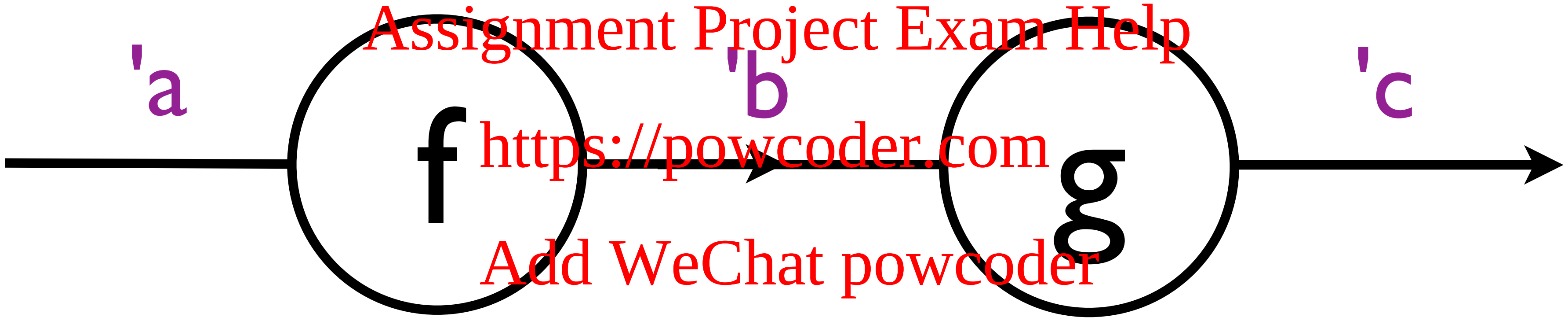
```
val f : 'a -> 'b
```

Lambda: the ultimate abstraction



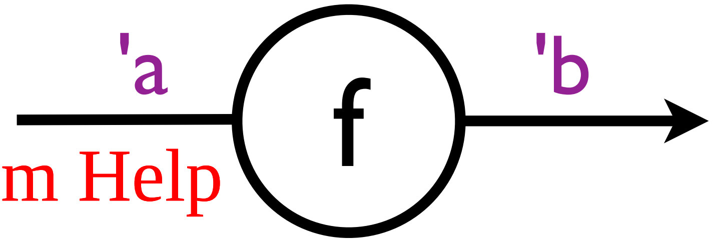
```
val f : 'a -> 'b          val g : 'b -> 'c
```

Lambda: the ultimate abstraction

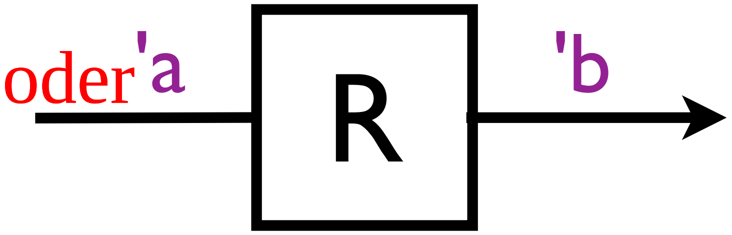


```
(compose g f): 'a -> 'c
```

Lambda abstraction:



Reagent abstraction:



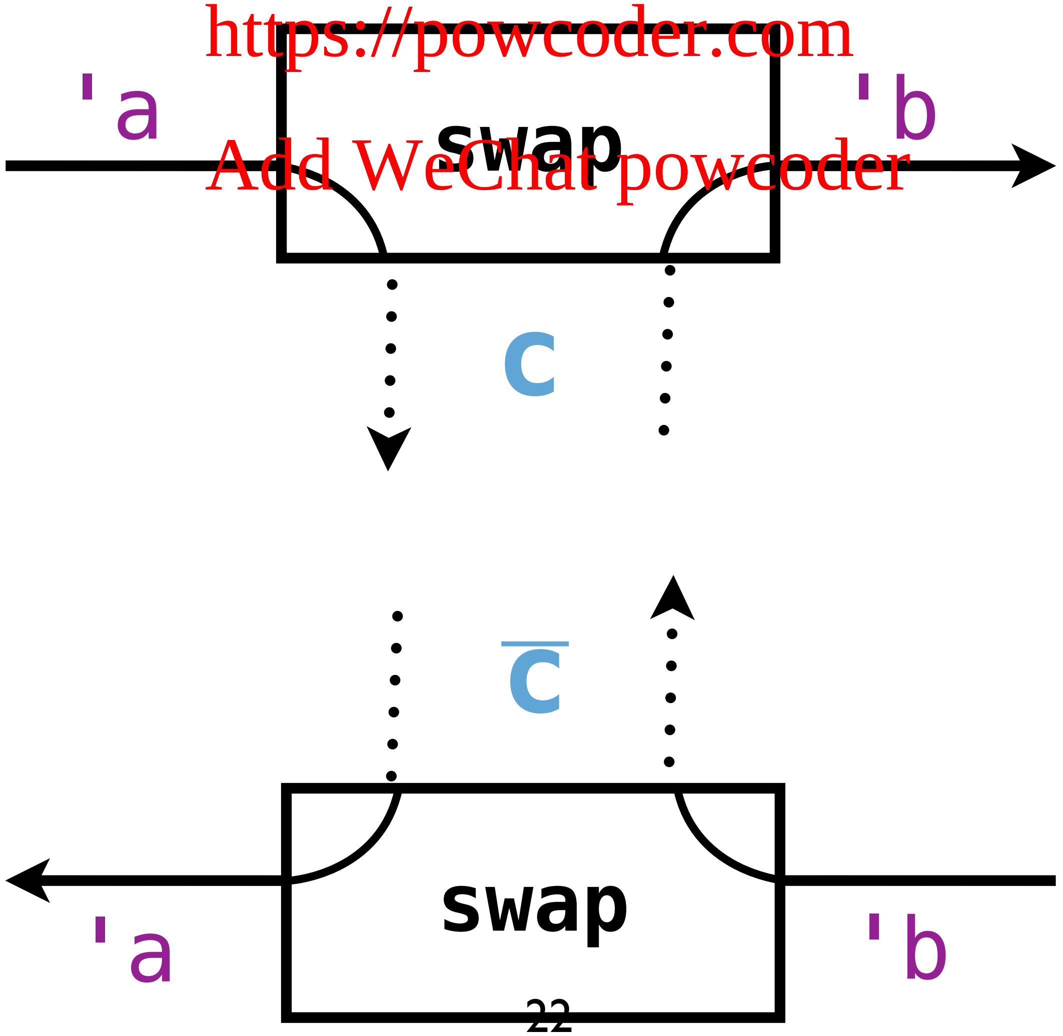
```
('a,'b) Reagent.t
```

Reagent combinators

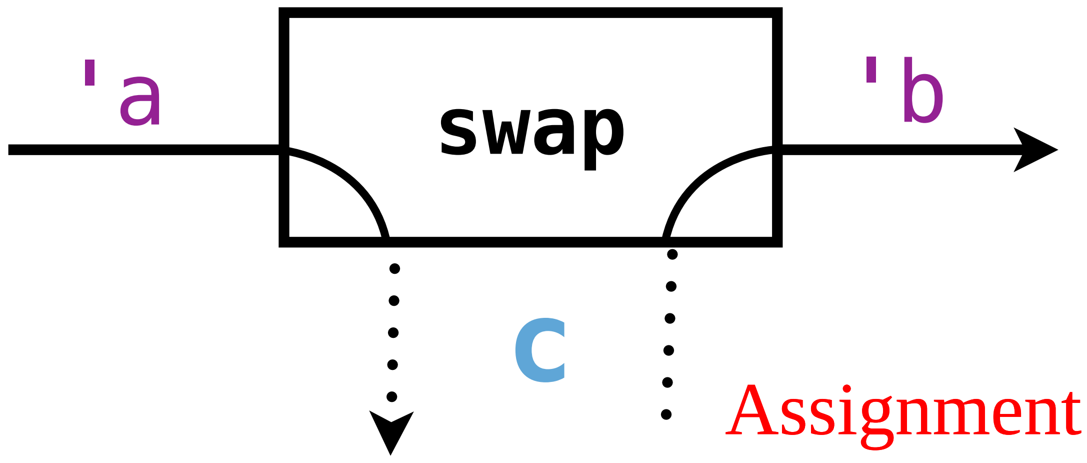
```
module type Reagents = sig
  type ('a,'b) t
  val never       : ('a,'b) t
  val constant    : 'a -> ('b,'a) t
  val (>>>)       : ('a,'b) t -> ('b,'c) t -> ('a,'c) t

  module Ref : Ref.S with type ('a,'b) reagent = ('a,'b) t
  module Channel : Channel.S with type ('a,'b) reagent = ('a,'b) t

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

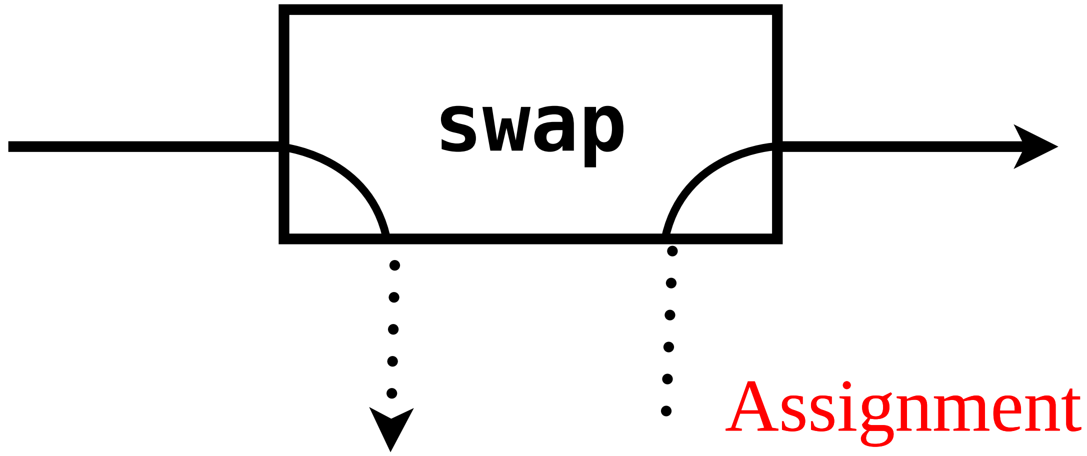
c: ('a,'b) endpoint



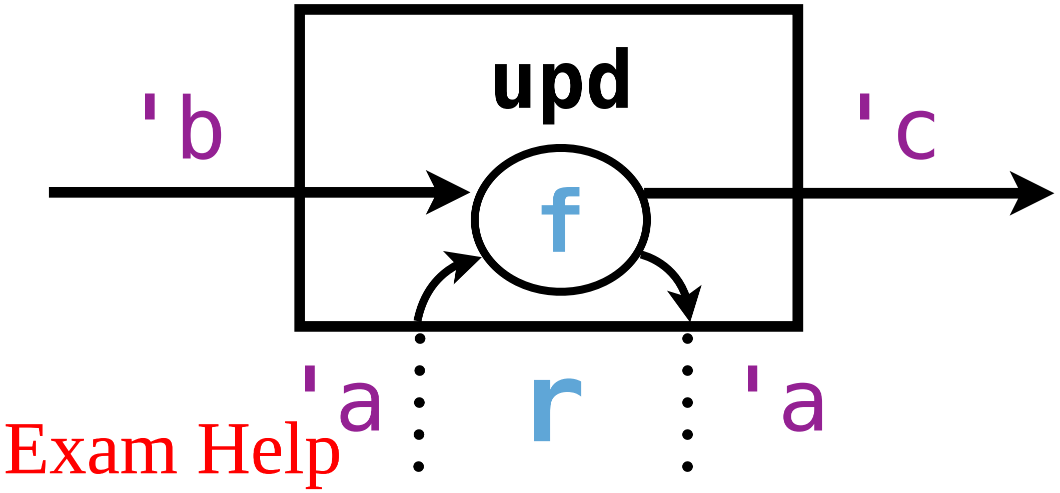
`c: ('a,'b) endpoint`



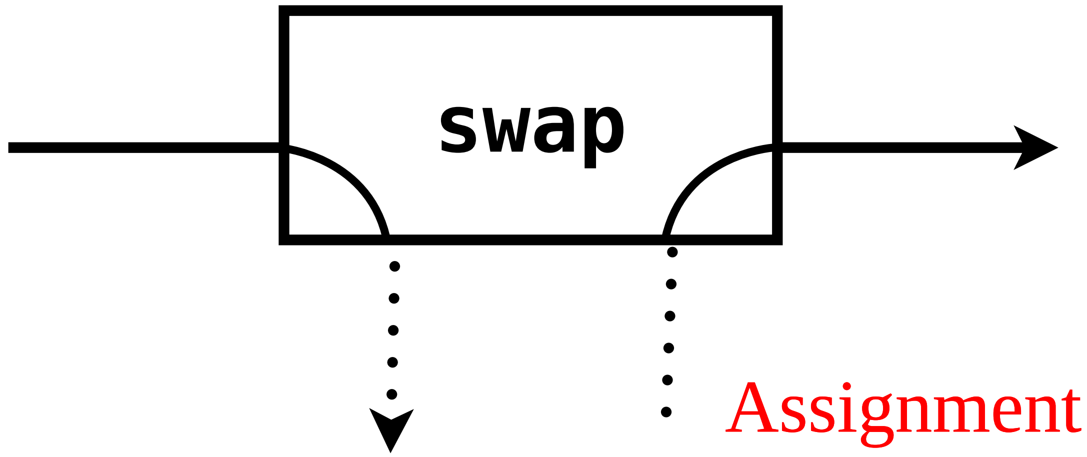
Message passing



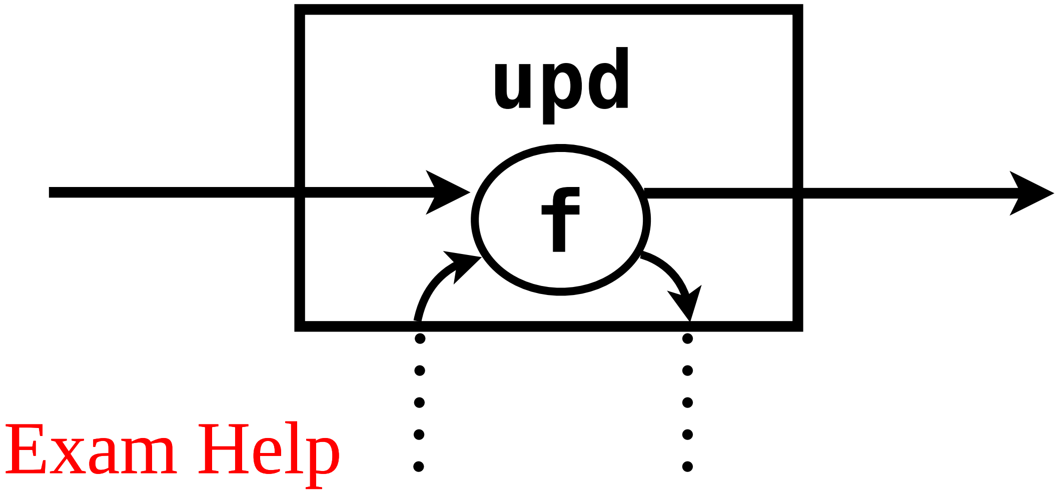
```
type 'a ref
val upd : 'a ref
  -> f:(‘a -> 'b -> ('a * ‘c) option)
  -> ('b, 'c) Reagent.t
```

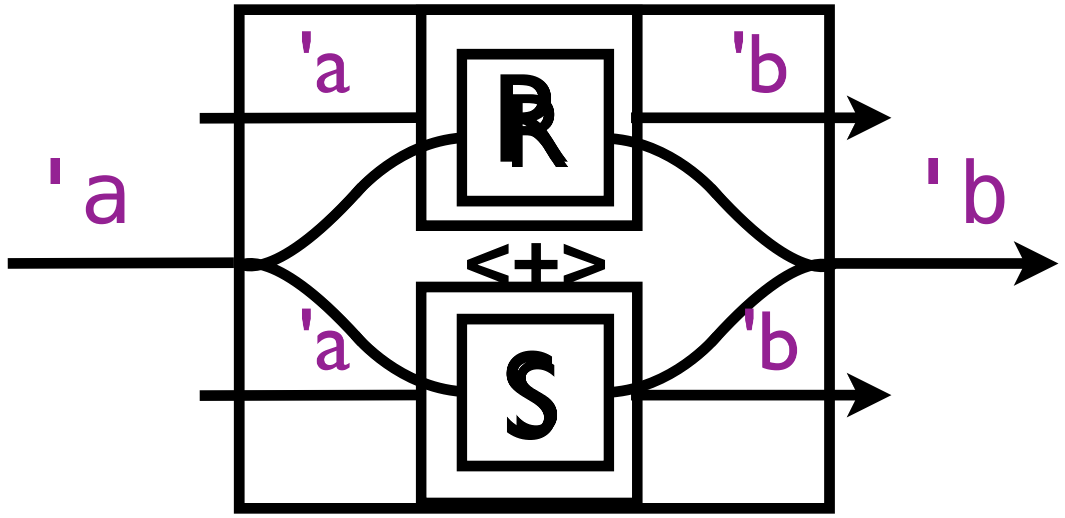


Message passing

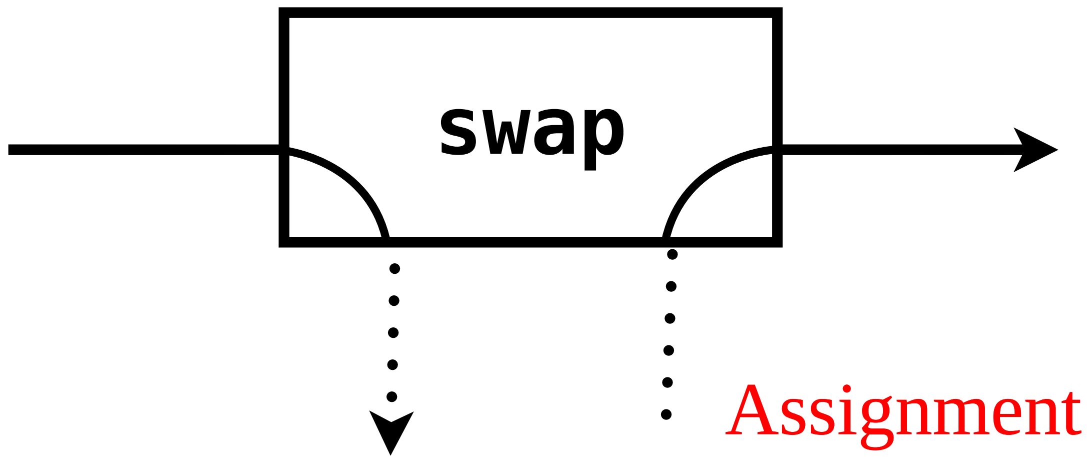


Shared state

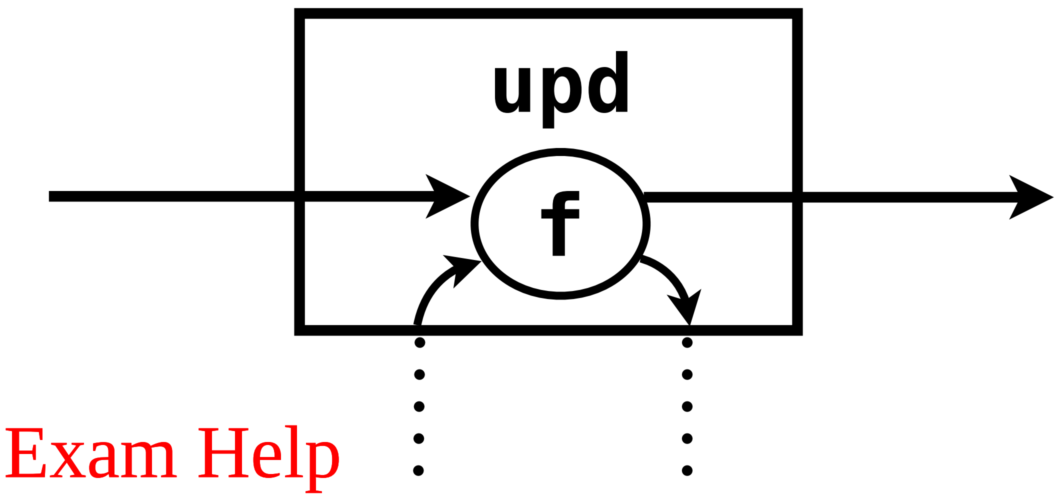




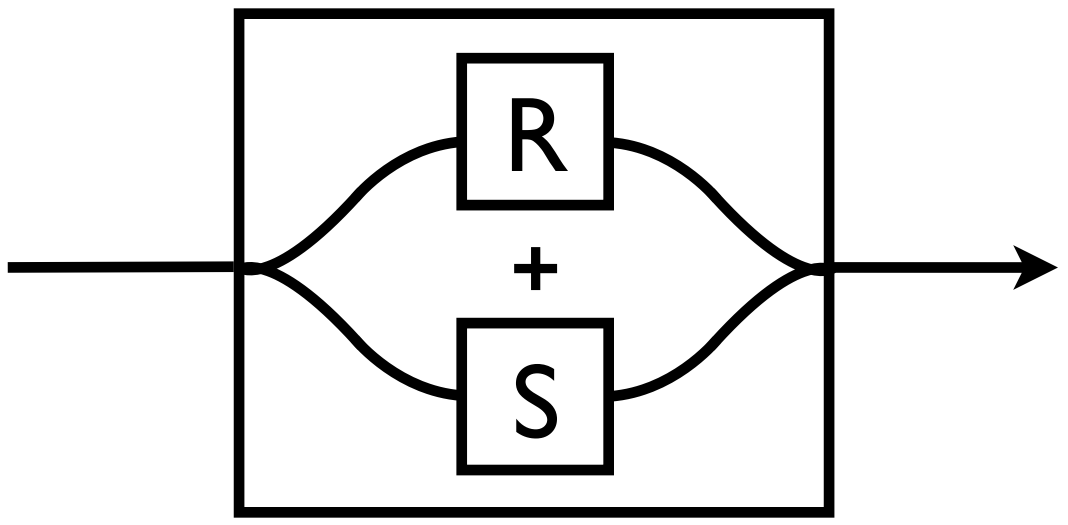
Message passing

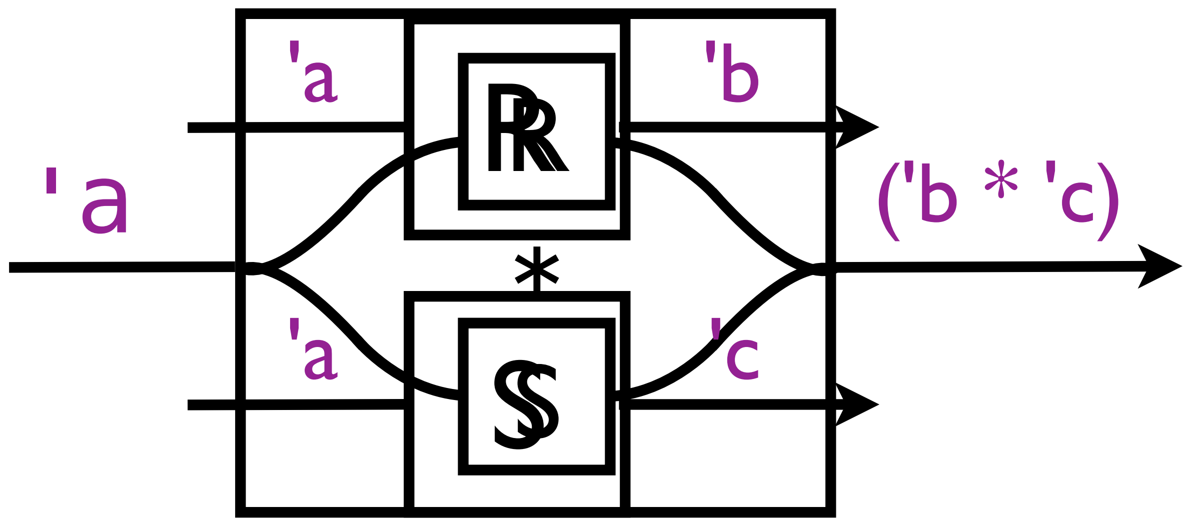


Shared state

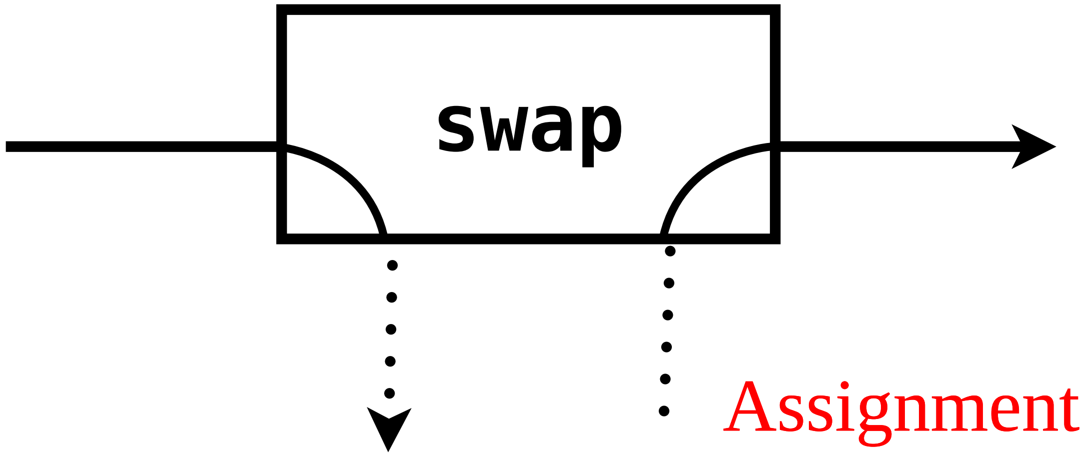


Disjunction

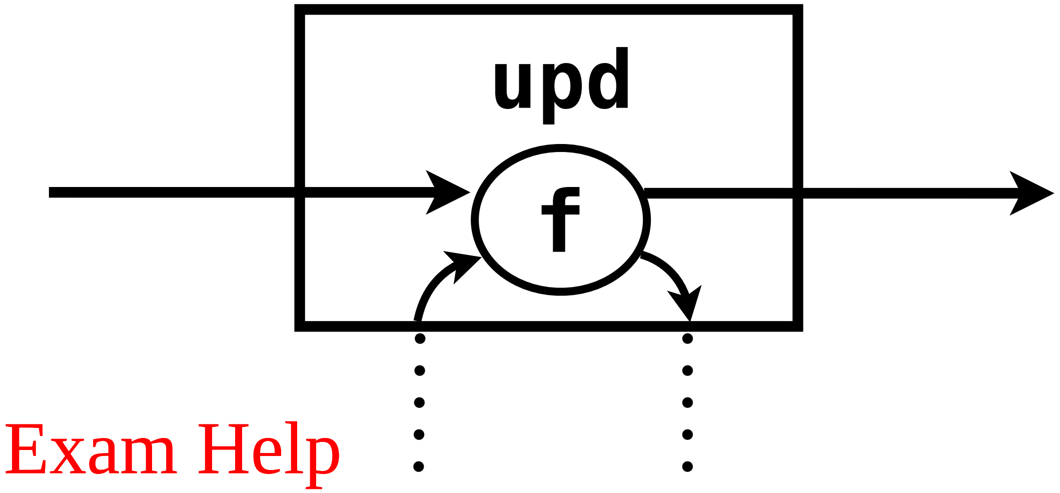




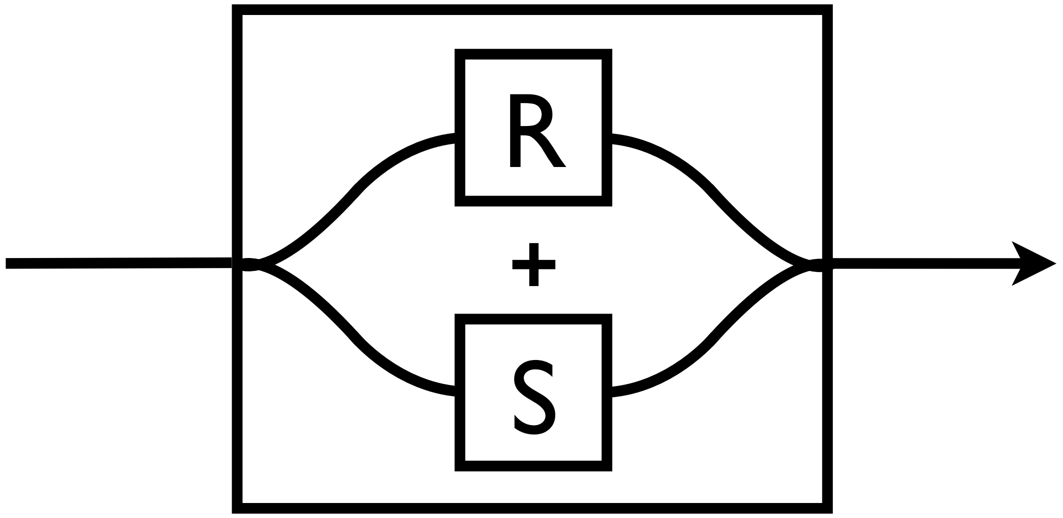
Message passing



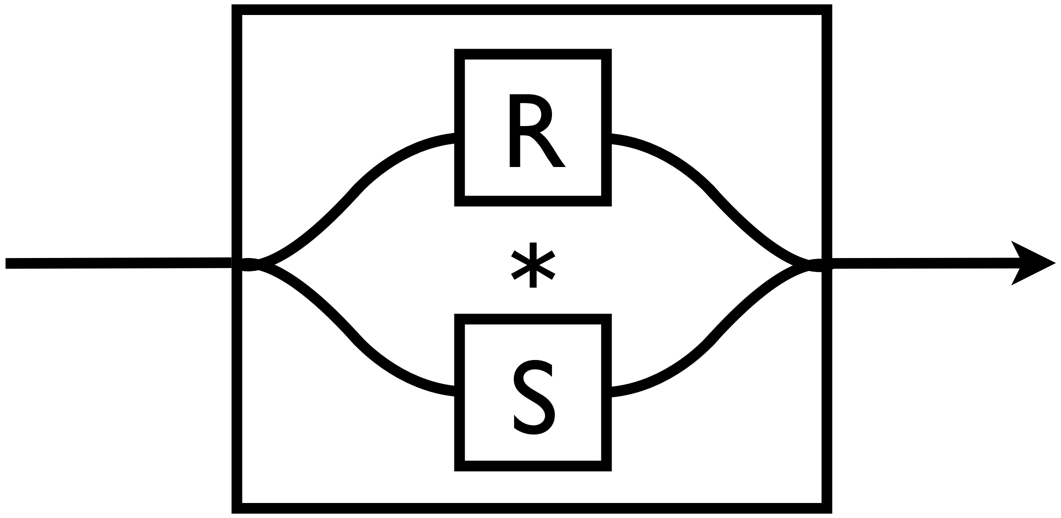
Shared state



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
  val try_pop : 'a t -> (unit, 'a option) Reagent.t
end

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

  let create () = Ref.mk_ref []

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

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

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

Composability

Transfer elements atomically

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

Consume elements atomically

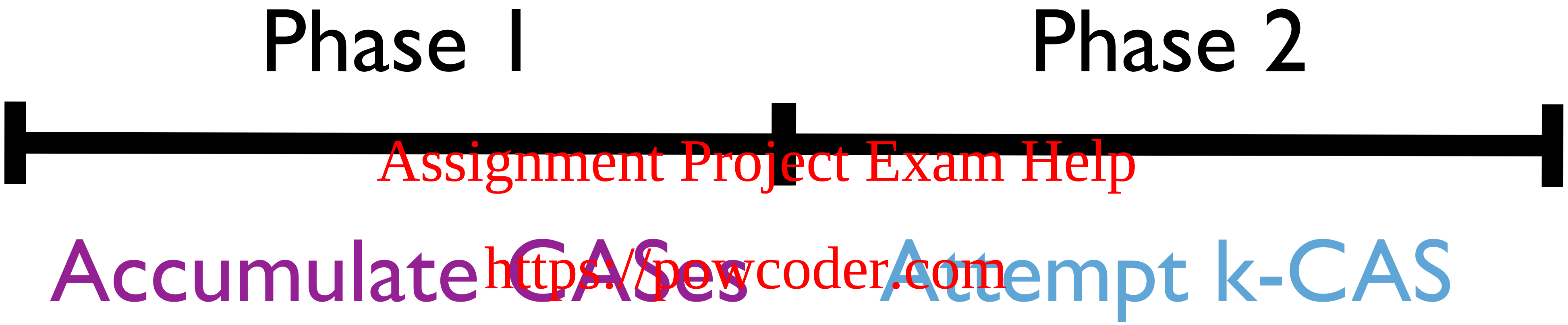
```
Treiber_stack.pop s1 <*> Treiber_stack.pop s2
```

Consume elements from either

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

```
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 init forks =
  List.iter (fun fork ->
    Thread.spawn @@ run (drop fork)) forks

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

Implementation

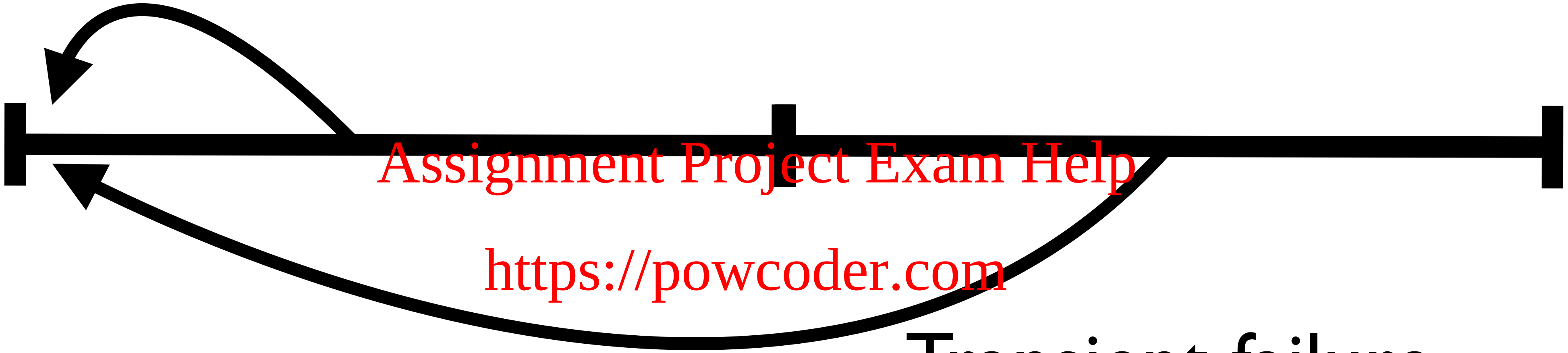


Phase 1

Phase 2

Accumulate CASes

Attempt k-CAS

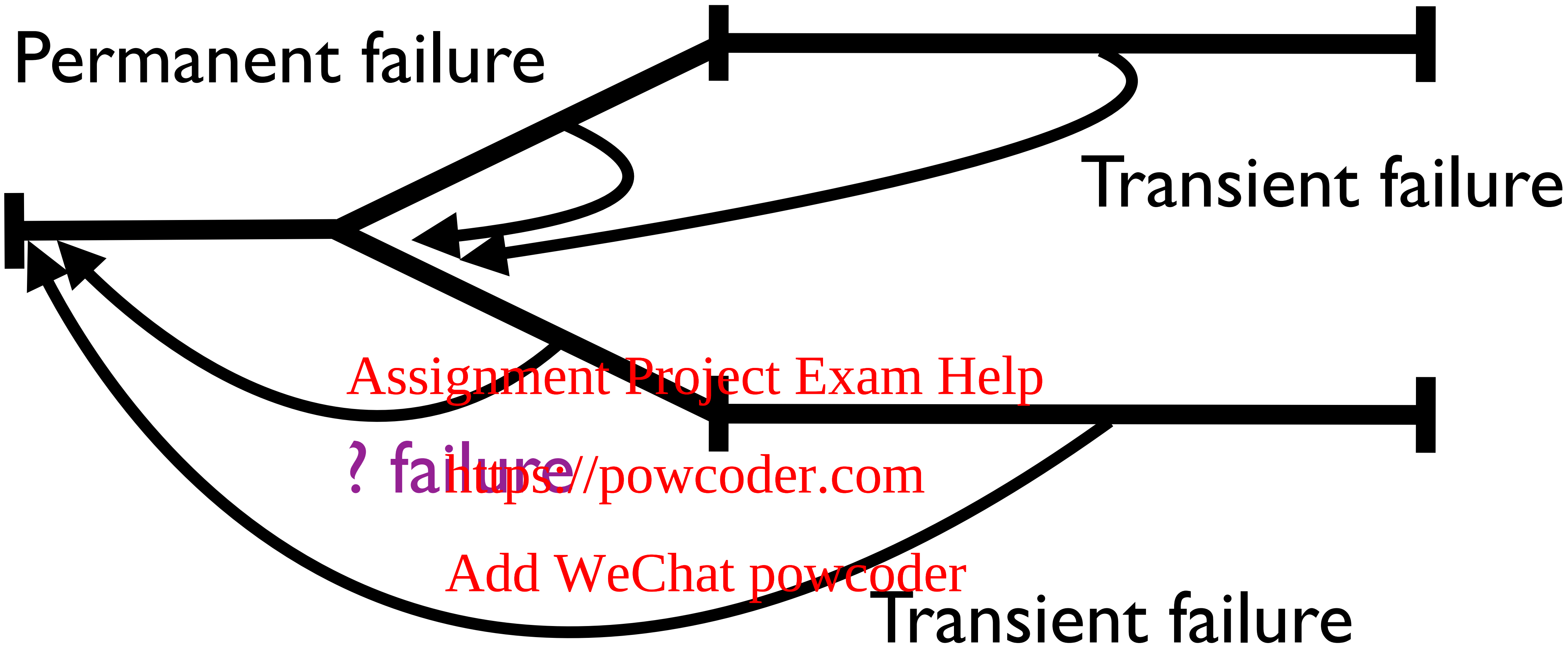


Permanent failure

Transient failure

Accumulate CASes

Attempt k-CAS



Permanent failure

Transient failure

? failure

Transient failure

P & P = P

T & T = T

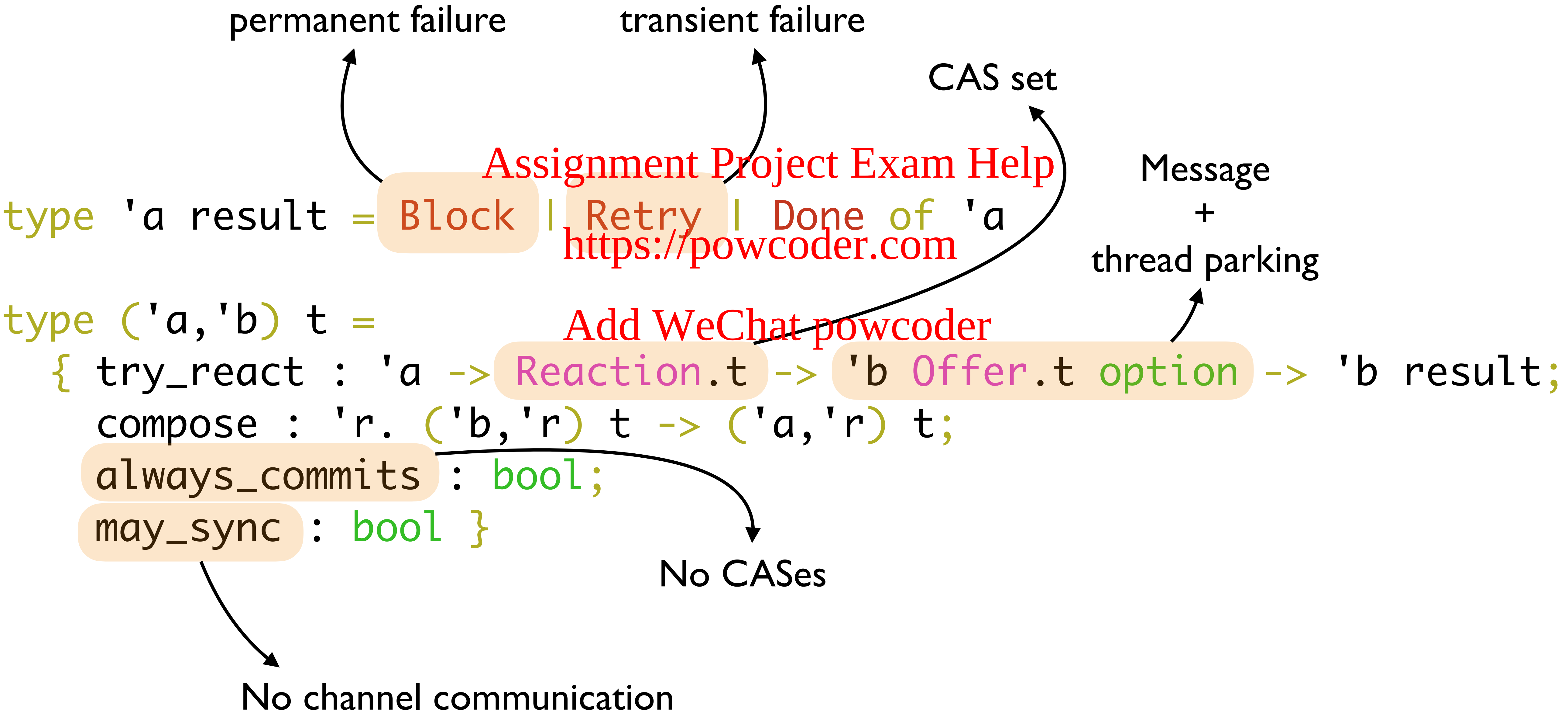
P & T = T

T & P = T

Trouble with k-CAS

- Most processors do not support k-CAS
- Implemented as a multi-phase protocol
  1. Sort refs
  2. Lock refs in order (CAS); rollback if conflicts.
  3. Commit refs
- Additional book-keeping required
  - CAS list, messages to be consumed, post-commit actions, etc.
- Common case is *just a single CAS*
  - *Identify and optimise with Arrows*

Reagent type



permanent failure

transient failure

CAS set

Message
+
thread parking

```
type 'a result = Block | Retry | Done of 'a

type ('a,'b) t =
  { try_react : 'a -> Reaction.t -> 'b Offer.t option -> 'b result;
    compose : 'r. ('b,'r) t -> ('a,'r) t;
    always_commits : bool;
    may_sync : bool }
```

No CASes

No channel communication

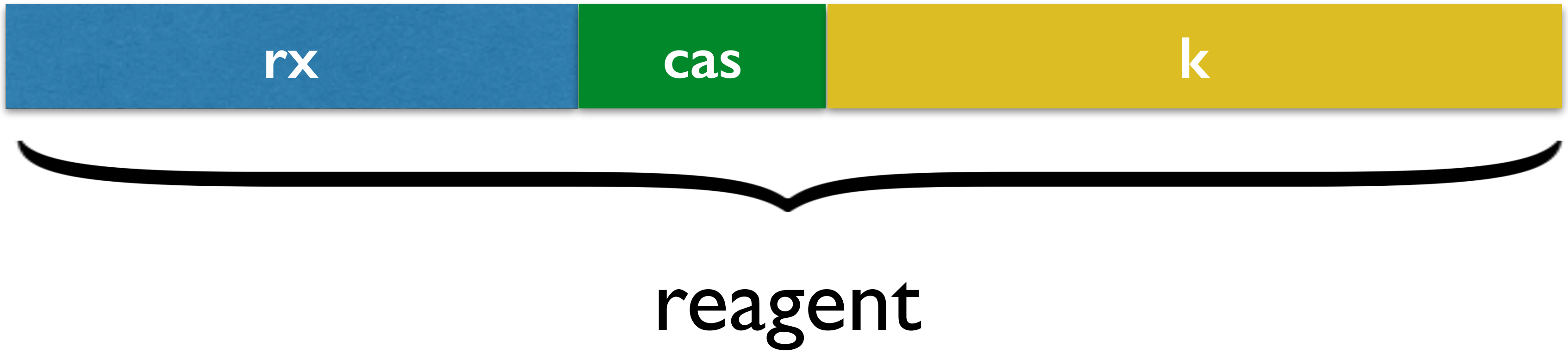
```
let rec never : 'a 'b. ('a,'b) t =
  { try_react = (fun _ _ _ -> Block);
    may_sync = false;
    always_commits = false;
    compose = fun _ -> never }
```

```
let rec constant : 'a 'b 'r. 'a -> ('a,'r) t -> ('b, 'r) t =
  fun x k (* continuation *) ->
    { may_sync = k.may_sync;
      always_commits = k.always_commits;
      try_react = (fun _ rx o -> k.try_react x rx o);
      compose = (fun next -> constant x (k.compose next)) }
```

```
let rec <+> : 'a 'b 'r. ('a,'b) t -> ('a,'b) t -> ('a,'b) t =
  fun r1 r2 ->
    { always_commits = r1.always_commits && r1.always_commits;
      may_sync = r1.may_sync || r2.may_sync;
      ...
```

Specialising k-CAS

```
let rec cas r ~expect ~update k =
  let try_react () rx o =
    if Reaction.has_no_cas rx &&
       k.always_commits then
      if CAS.cas r.data expect update then
        ( k.try_react () rx o ) (* Will succeed! *)
      else Retry
    else
      (* slow path with bookkeeping *)
  in
  ...
```



Optimising Transient Failures

```
let rec without_offer pause r v =
  match r.try_react v Reaction.empty None with
  | Done res -> res
  | Retry ->
        ( pause ();
          if r.may_sync
          then with_offer pause r v
          else without_offer pause r v)
  | Block -> with_offer pause r v

let run r v =
  let b = Backoff.create () in
  let pause () = Backoff.once b in
  without_offer pause r v
```