Relational Reasoning for Mergeable Replicated Data Types

KC Sivaramakrishnan
joint work with
Gowtham Kaki, Swarn Priya, Suresh Jagannathan

PURDUE UNIVERSITY PURPL

IIT MADRAS

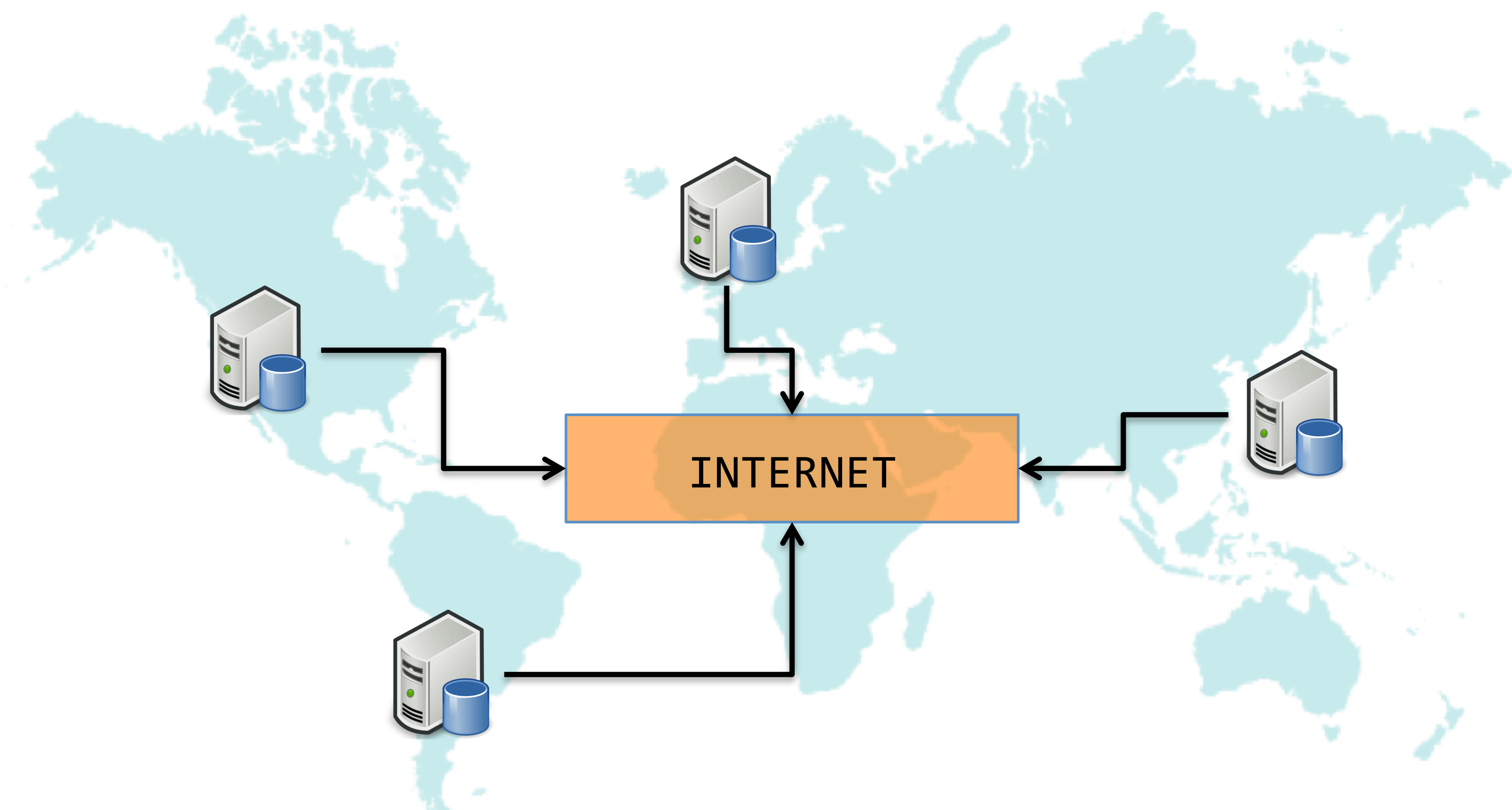
VISA

ebay

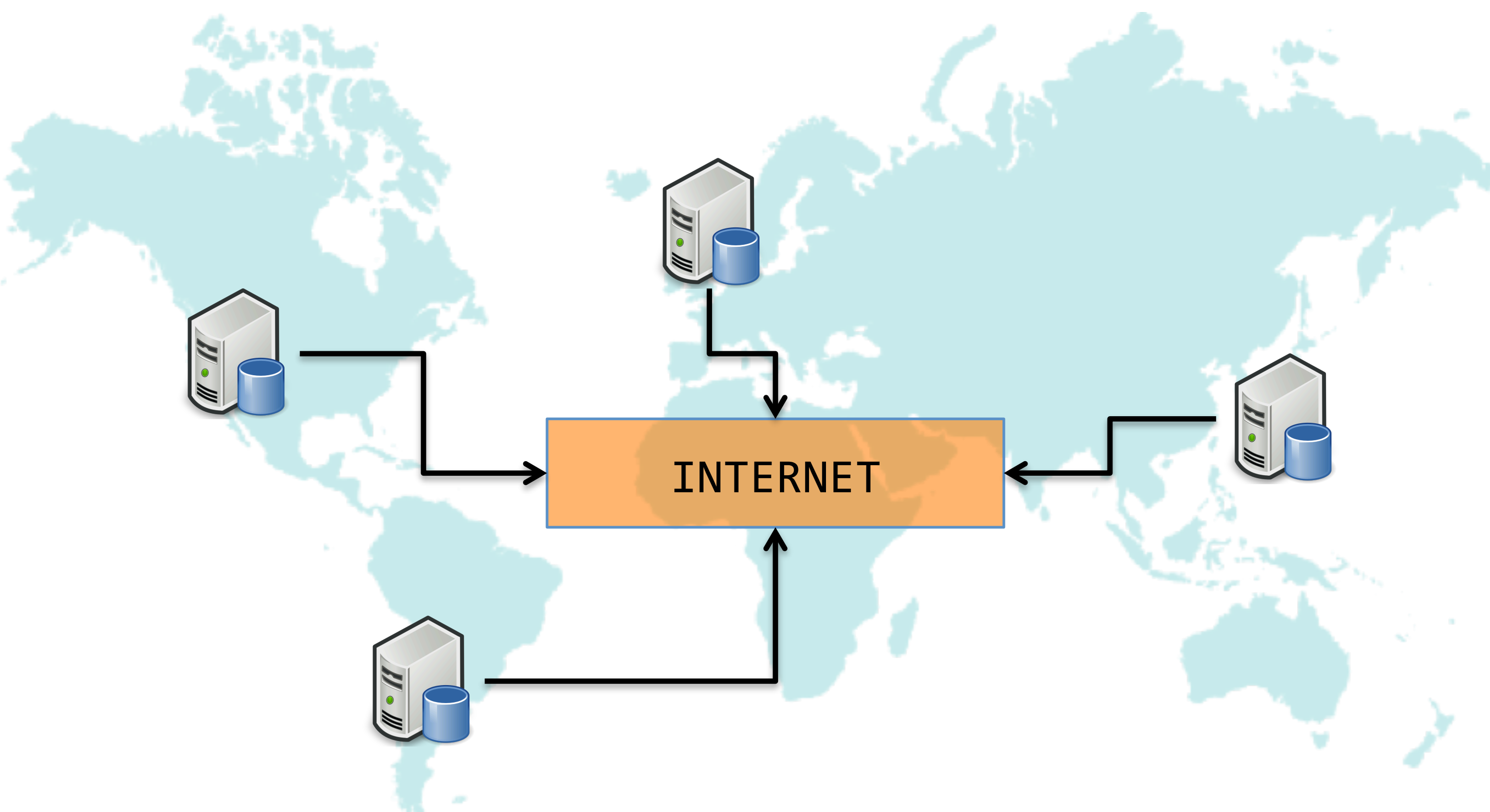
YouTube

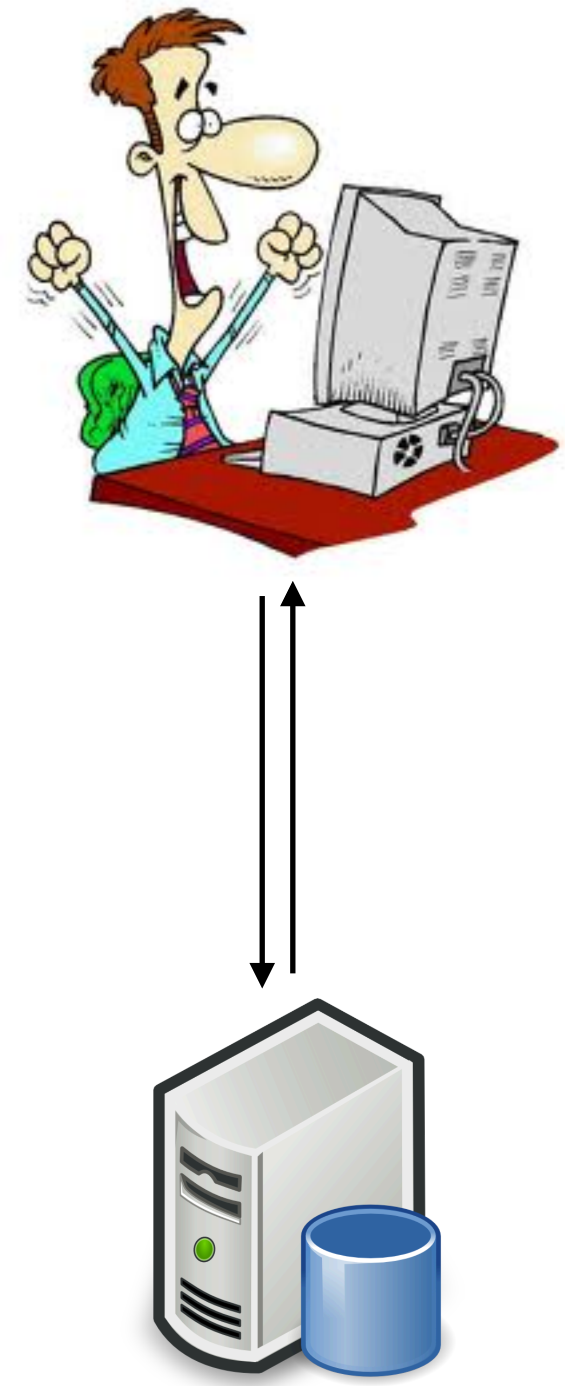
NETFLIX

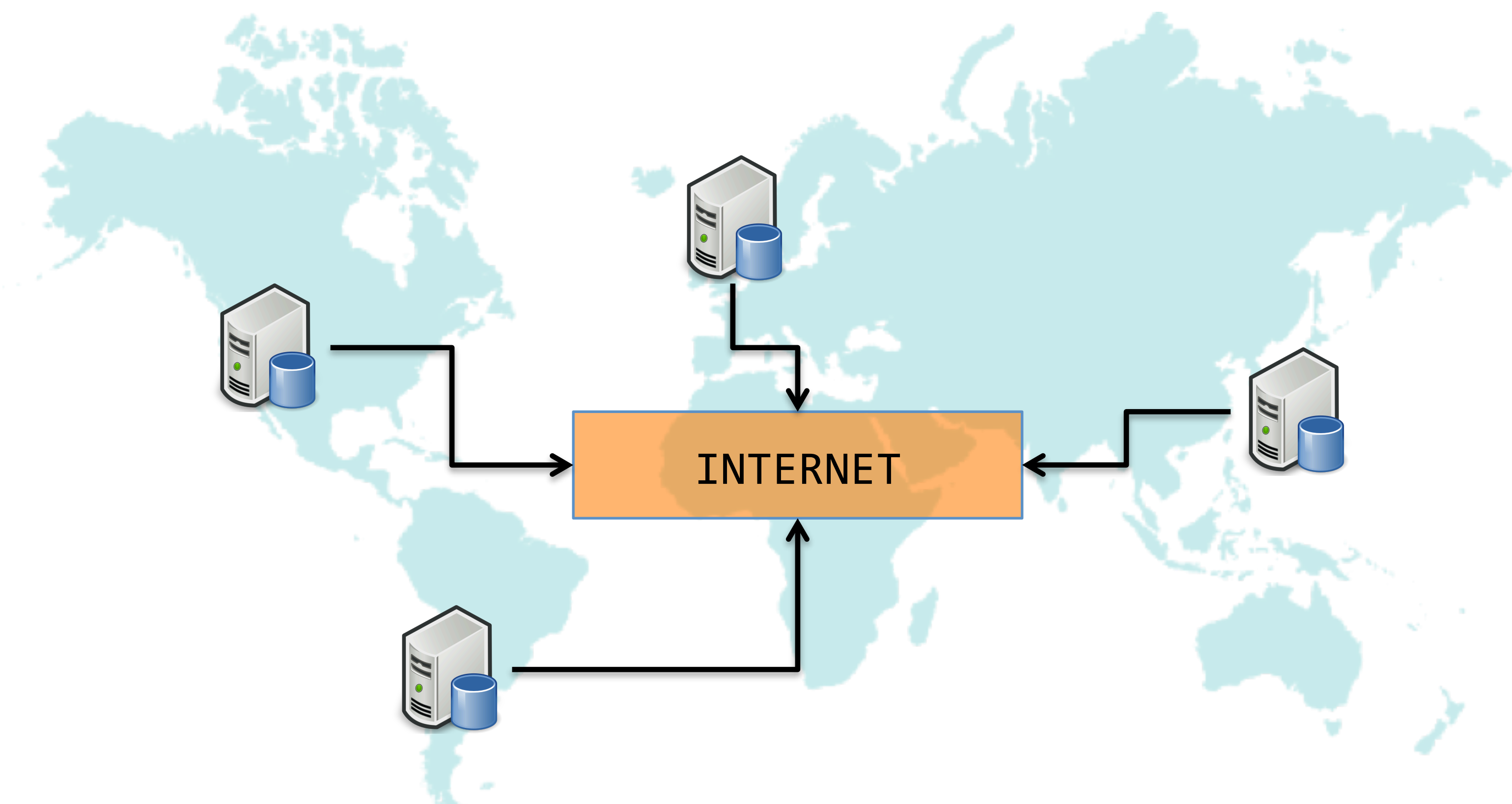
HSBC


INTERNET

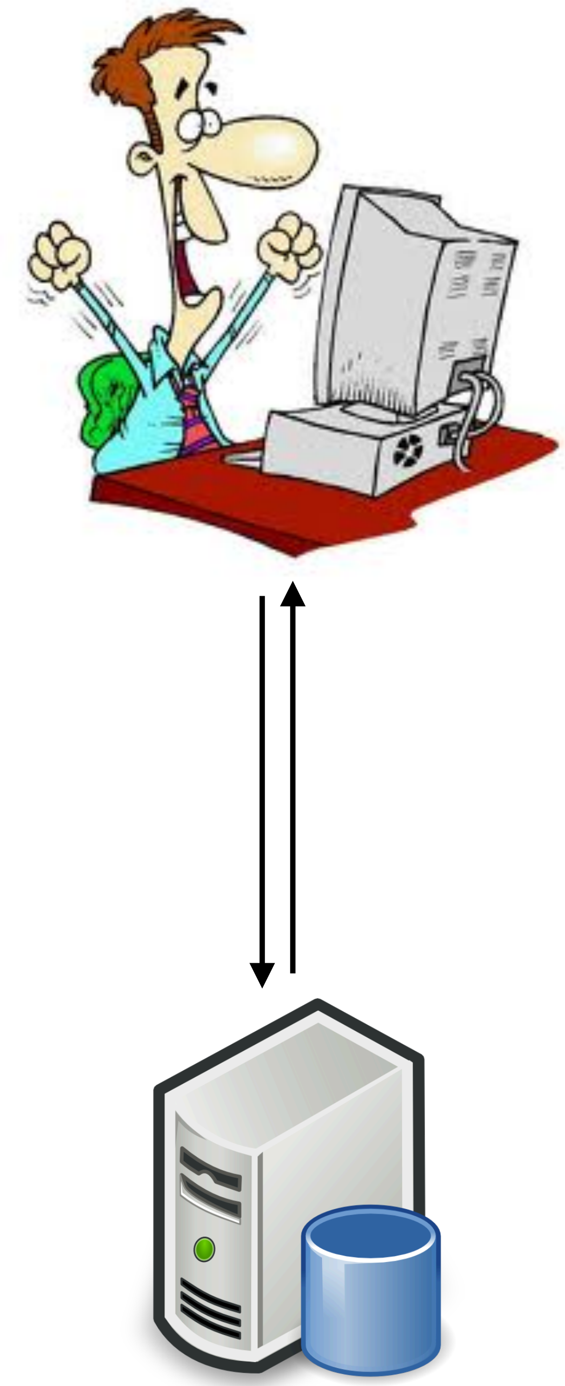
VISA
ebay
You Tube
NETFLIX
HSBC

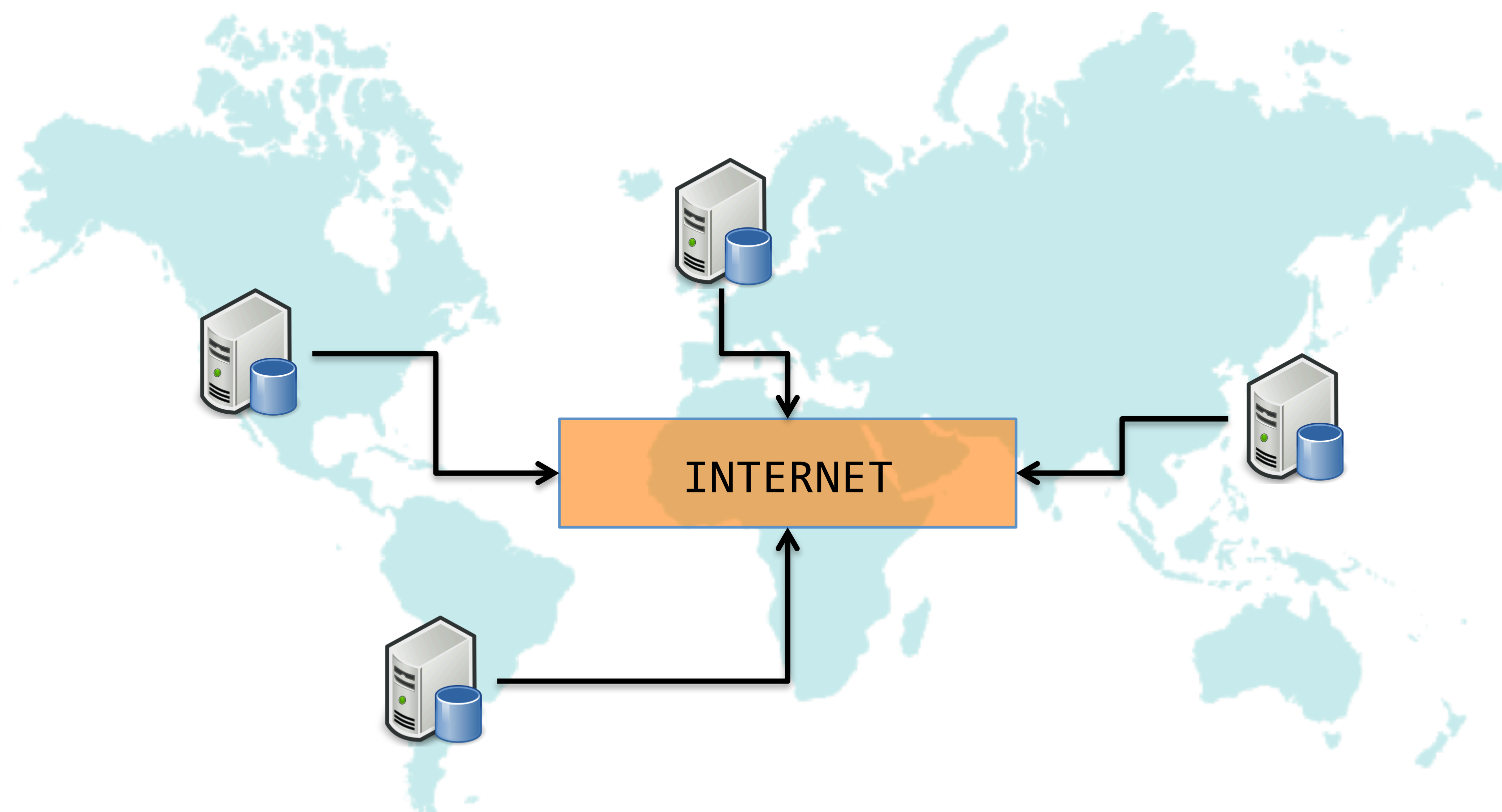




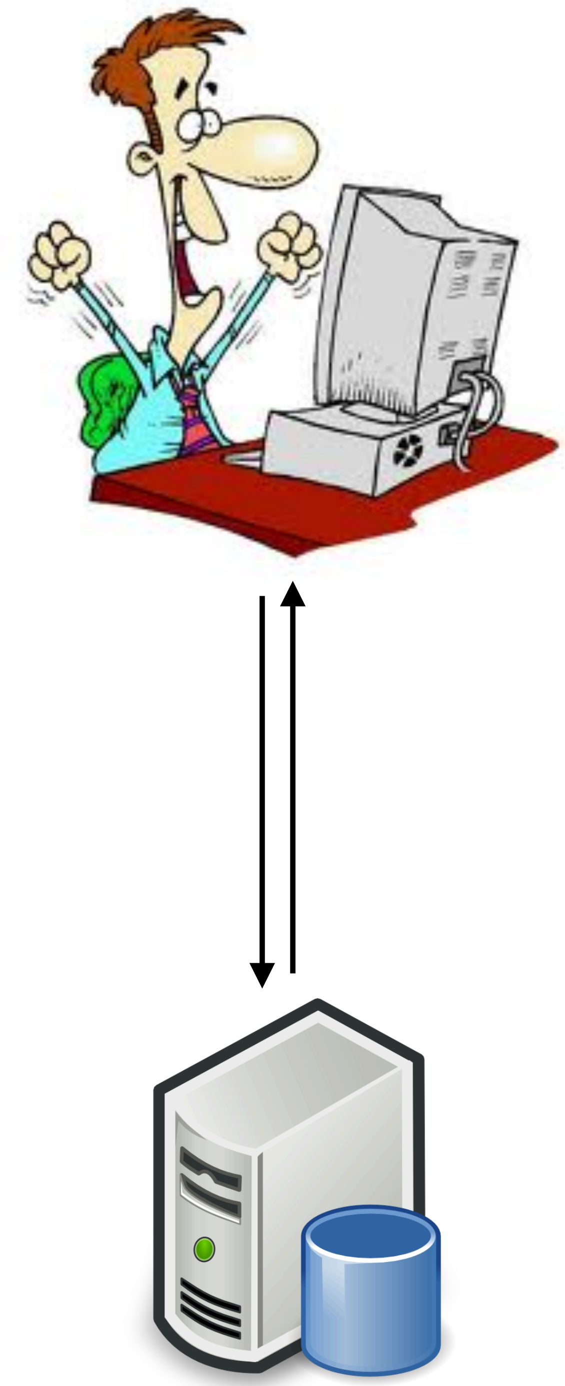


$\neq$





≠



- Serializability
- Linearizability

- Weak Consistency & Isolation

When *system-level concerns* like replication & availability affect *application-level design* decisions, programming becomes *complicated*.

natefoster
@natefoster

Seems like Twitter wants me to follow this guy.

Who to follow

Doug Woos
@dougwoos
Follow

PhD student, joining @BrownCSDept as a lecturer in Fall 2019. Into programming languages, distributed systems, baseball, and other stuff. he/him

Doug Woos
@dougwoos
Follow

PhD student, joining @BrownCSDept as a lecturer in Fall 2019. Into programming languages, distributed systems, baseball, and other stuff. he/him

Doug Woos
@dougwoos
Follow

PhD student, joining @BrownCSDept as a lecturer in Fall 2019. Into programming languages, distributed systems, baseball, and other stuff. he/him

5:43 AM · May 14, 2019 · Twitter for iPhone

Sequential Counter

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
end
```

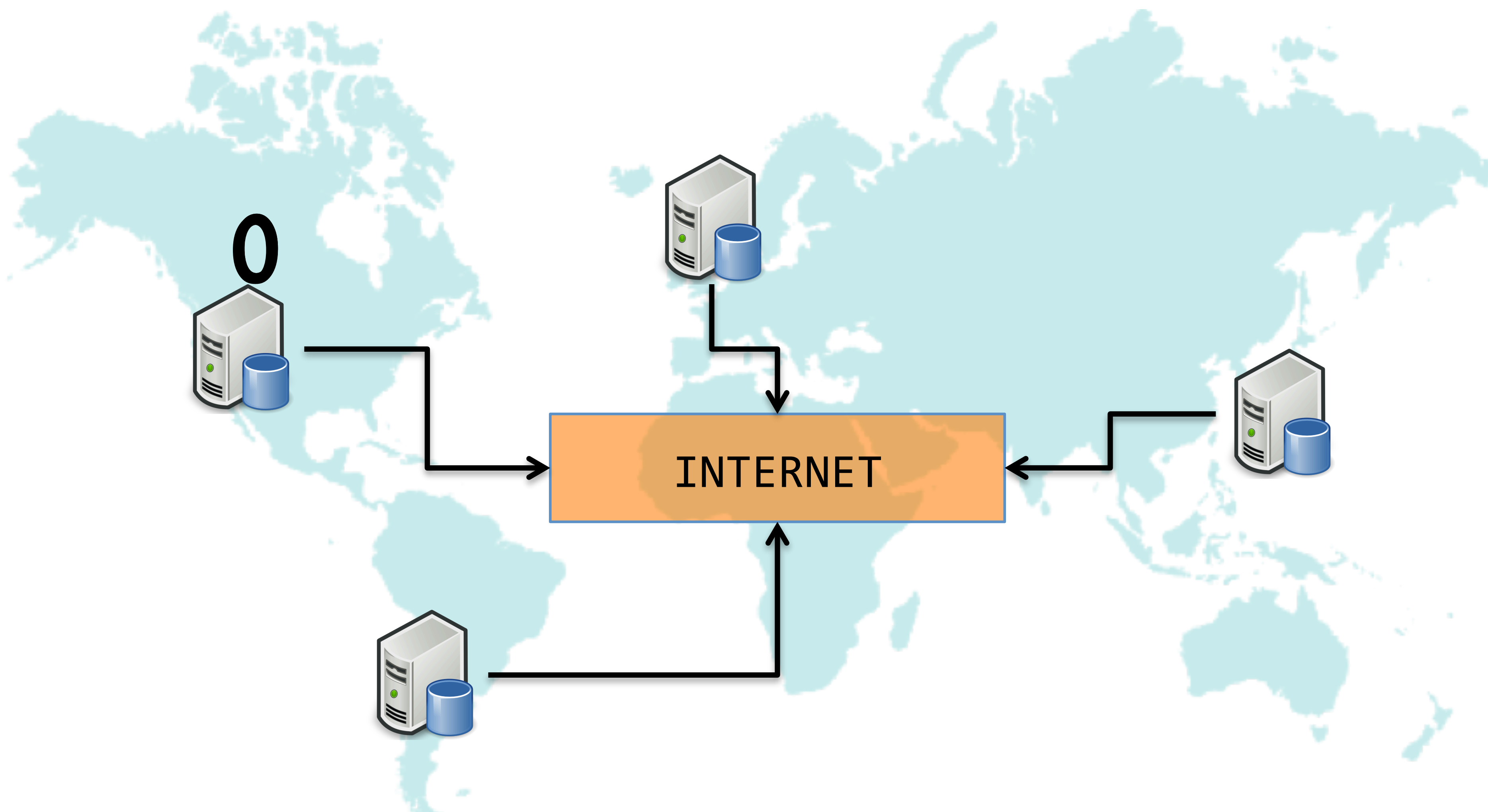
Sequential Counter

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
end
```

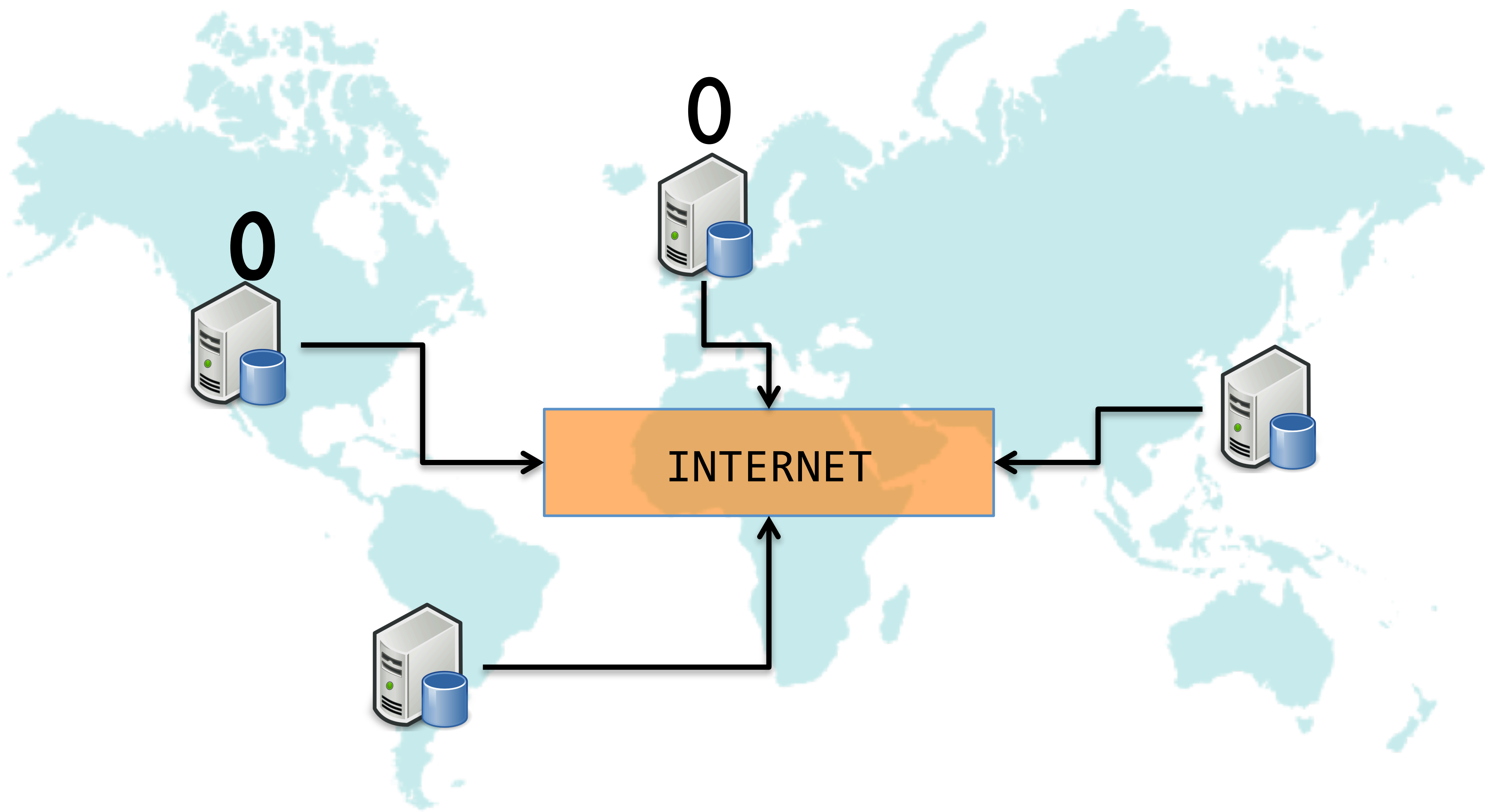
- Written in idiomatic style
- Composable

```
type counter_list = Counter.t list
```

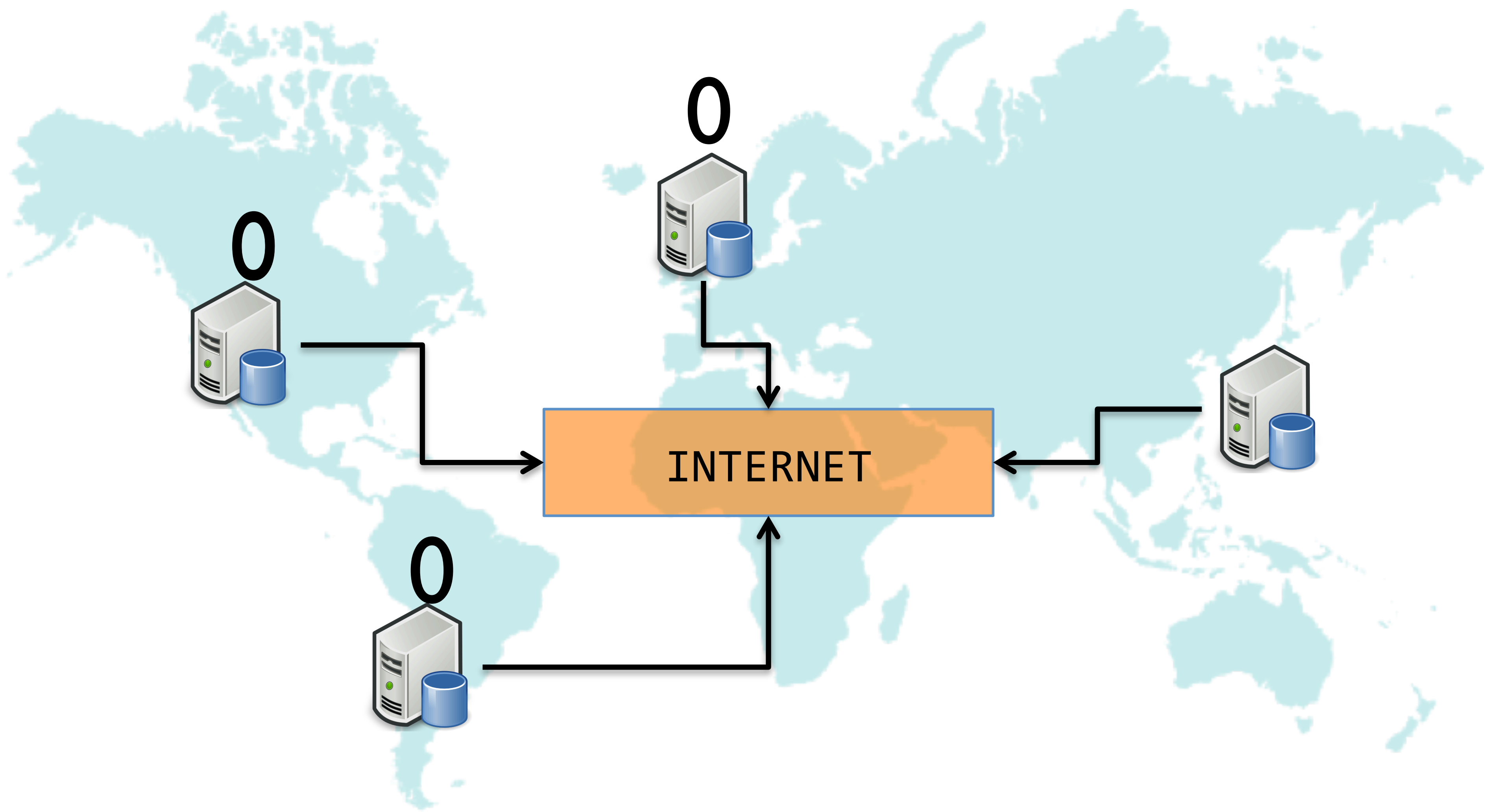
Replicated Counter



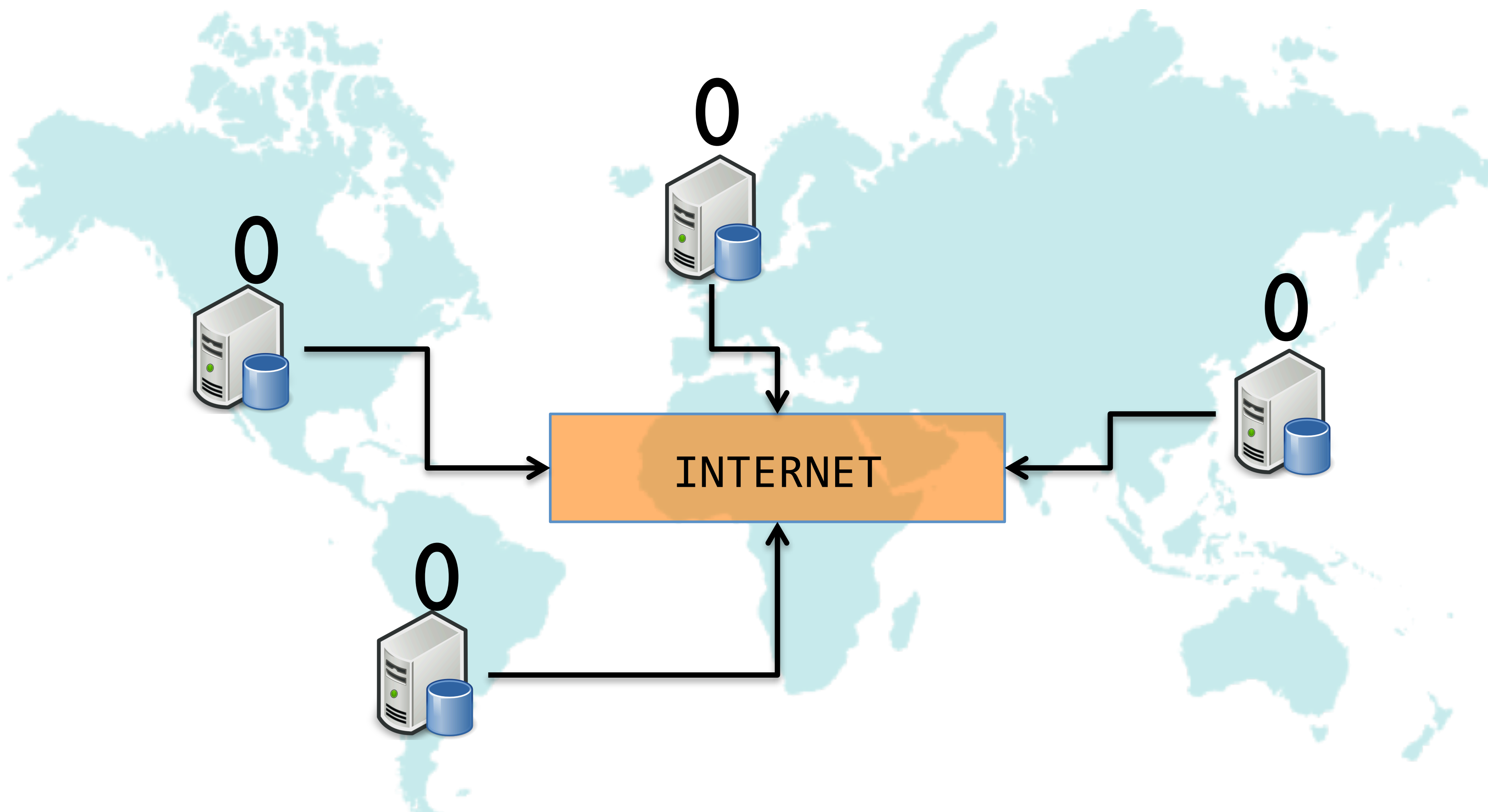
Replicated Counter



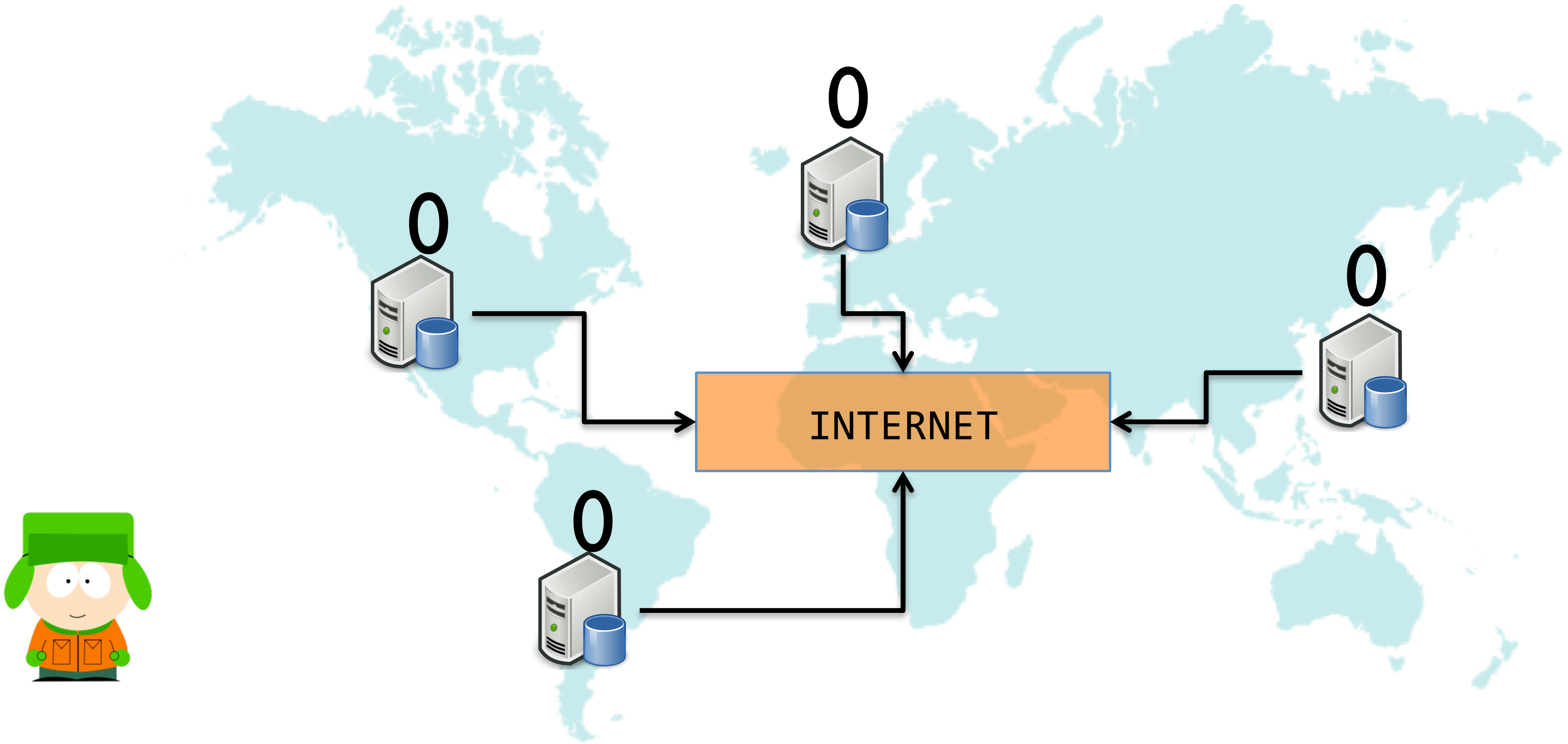
Replicated Counter



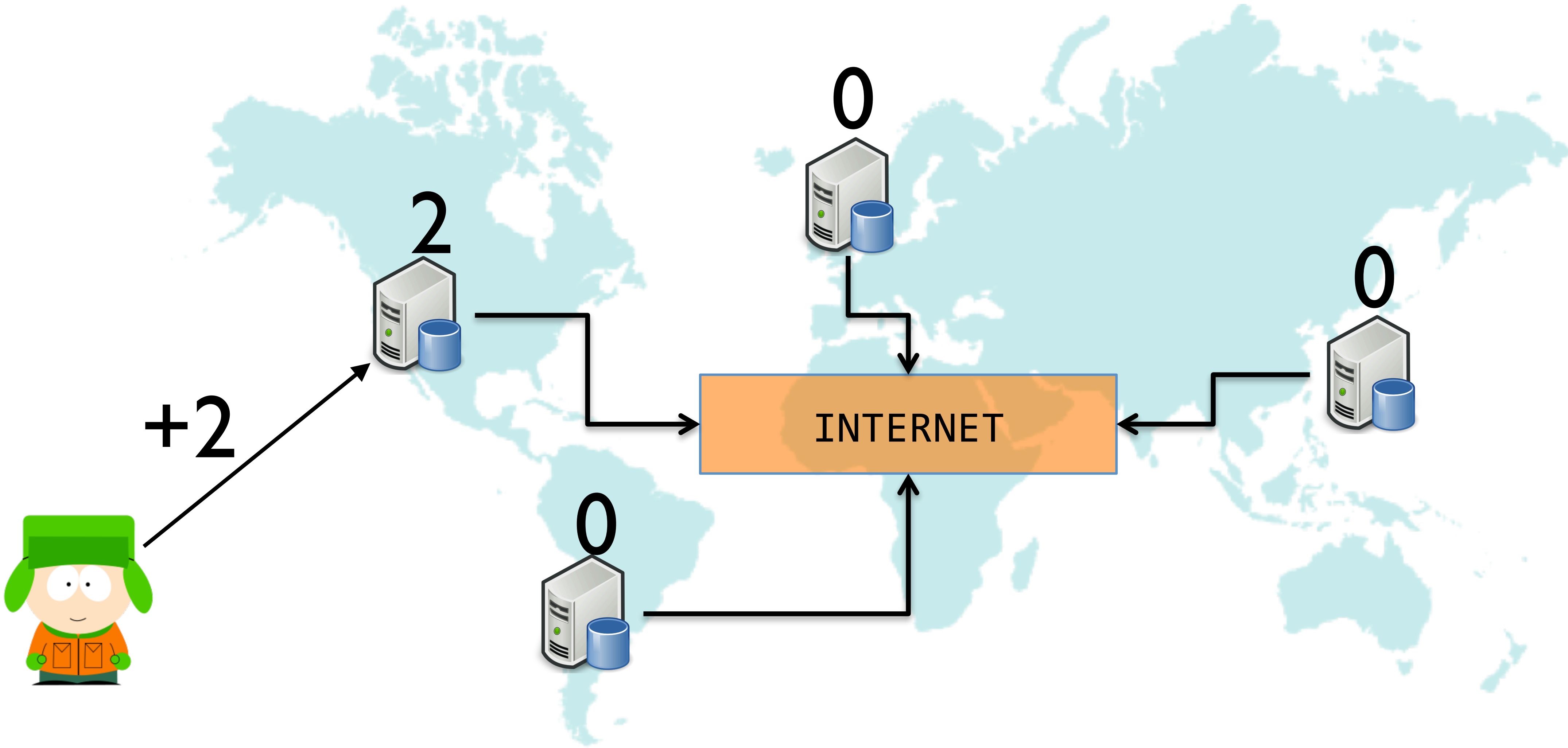
Replicated Counter



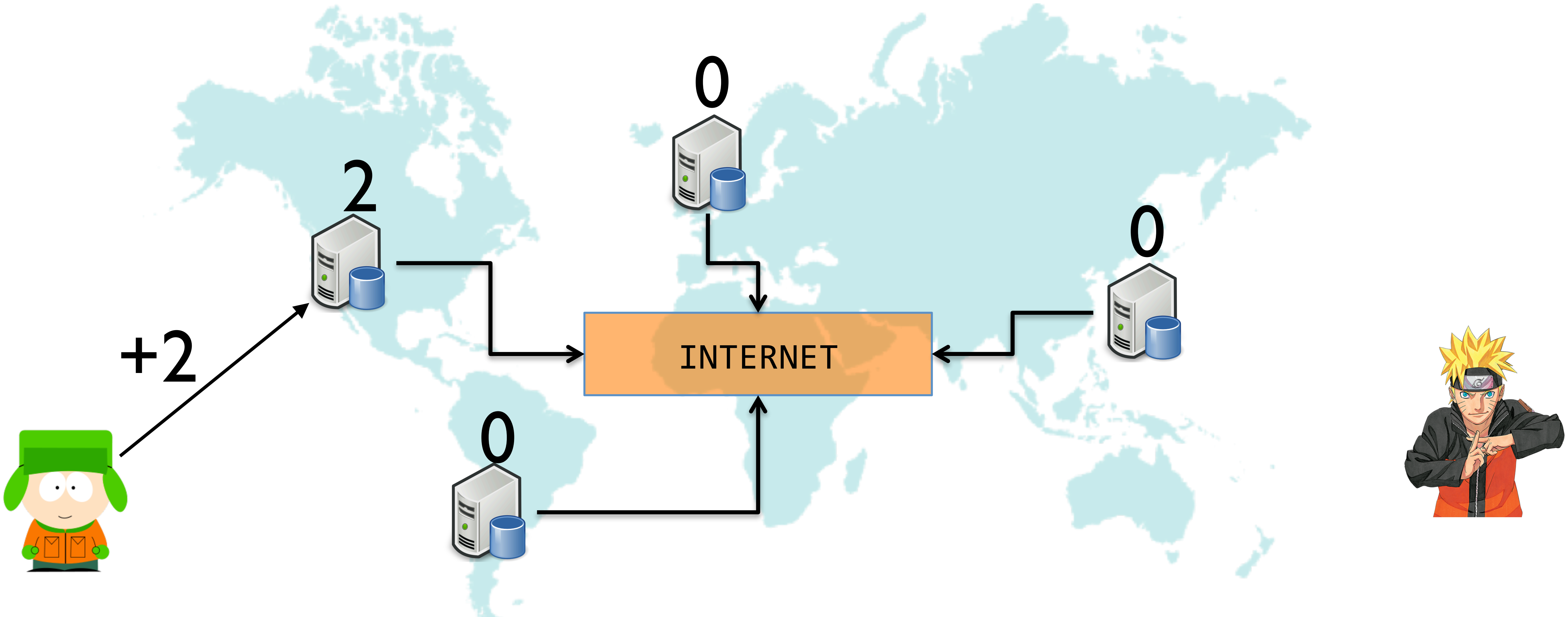
Replicated Counter



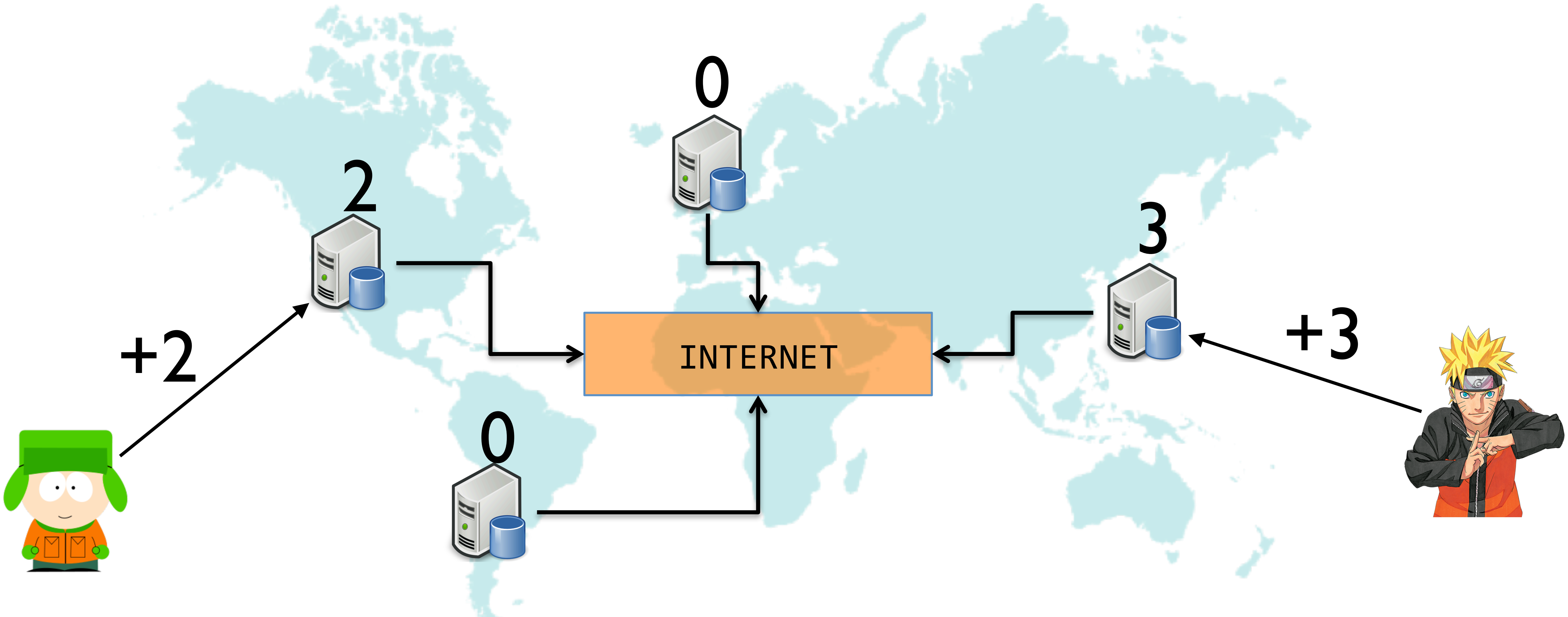
Replicated Counter



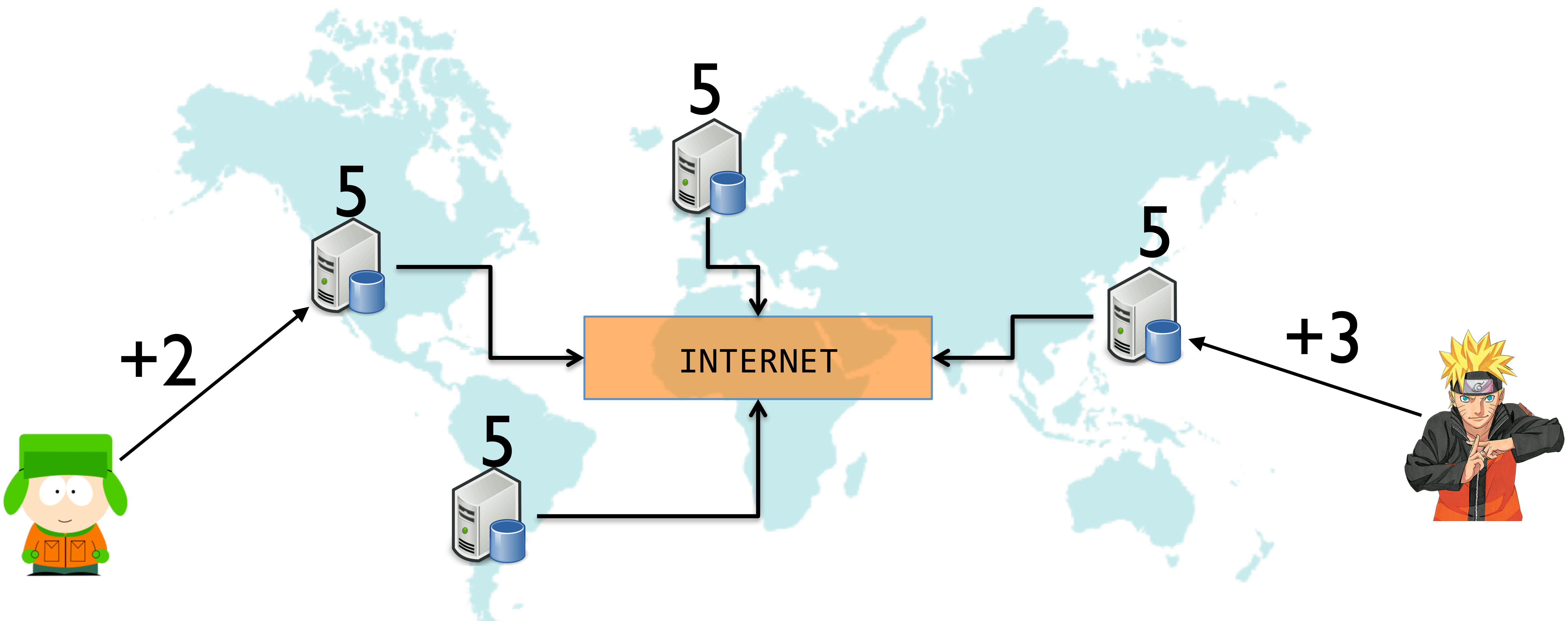
Replicated Counter



Replicated Counter



Replicated Counter

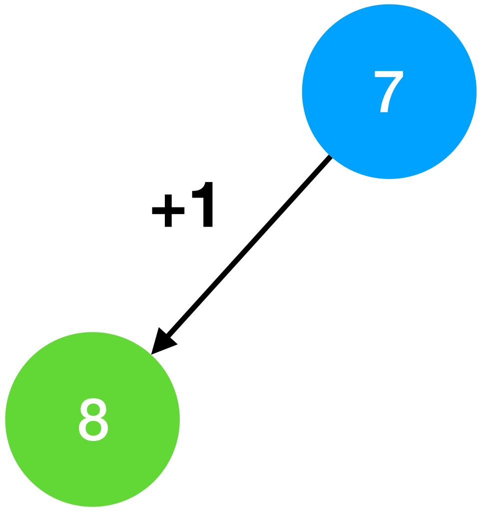


```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

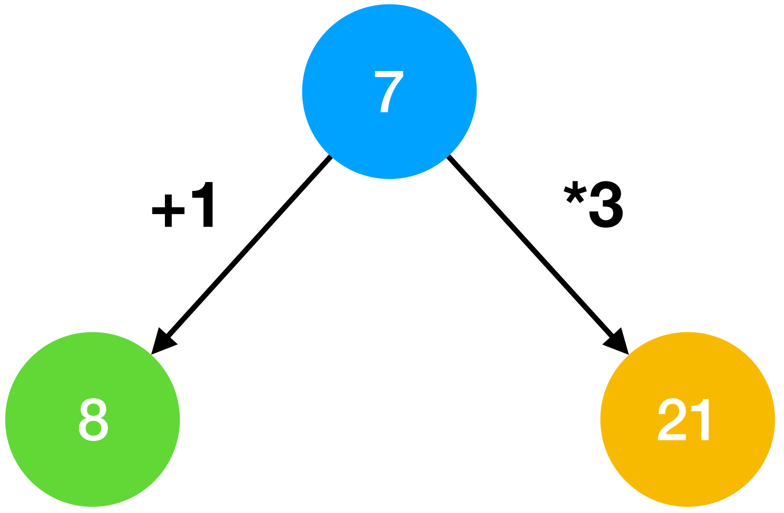
7

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

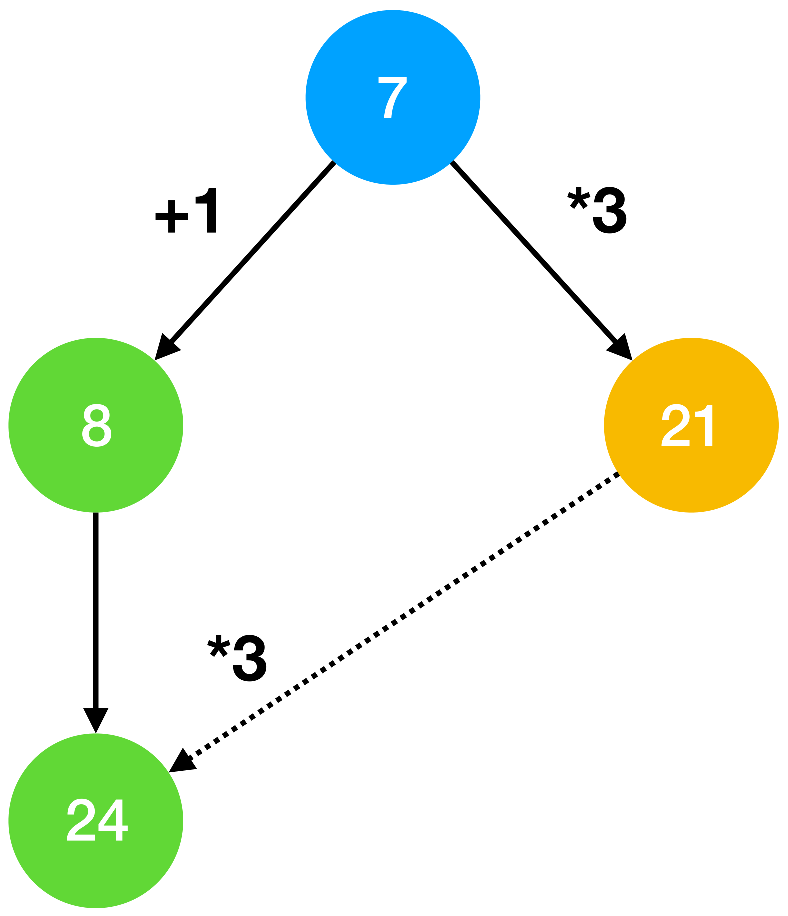
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



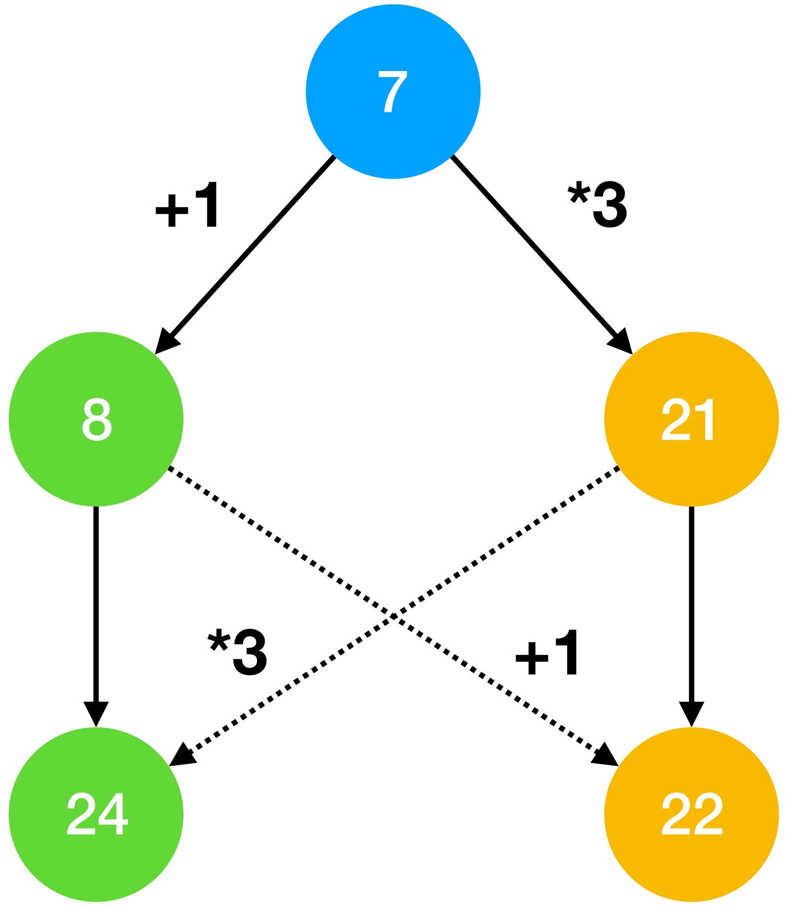
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



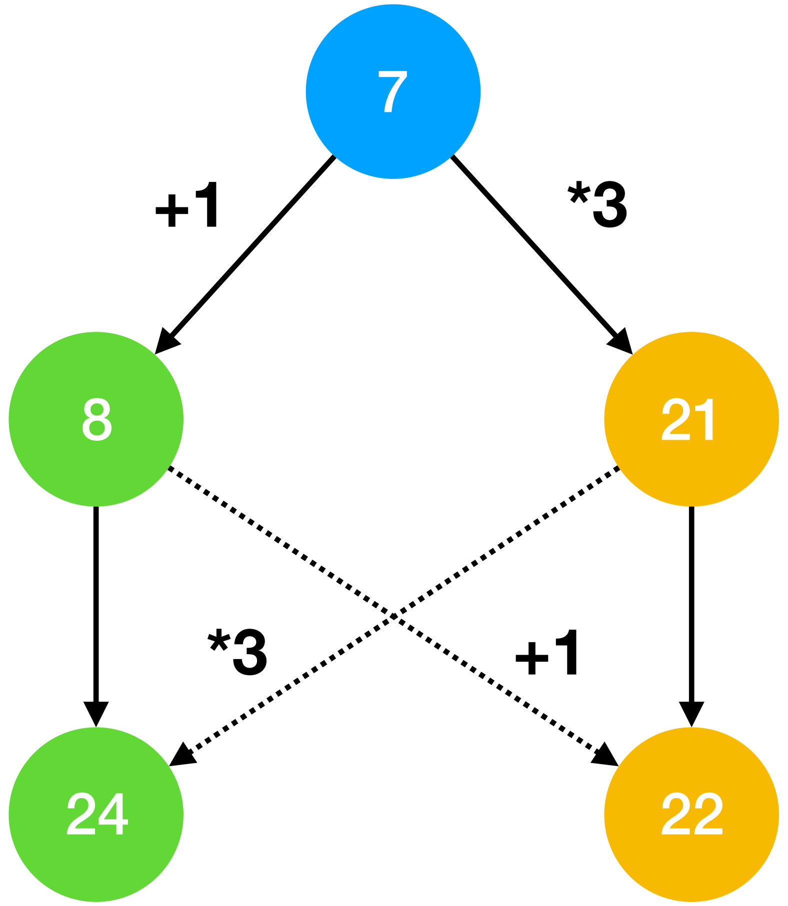
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

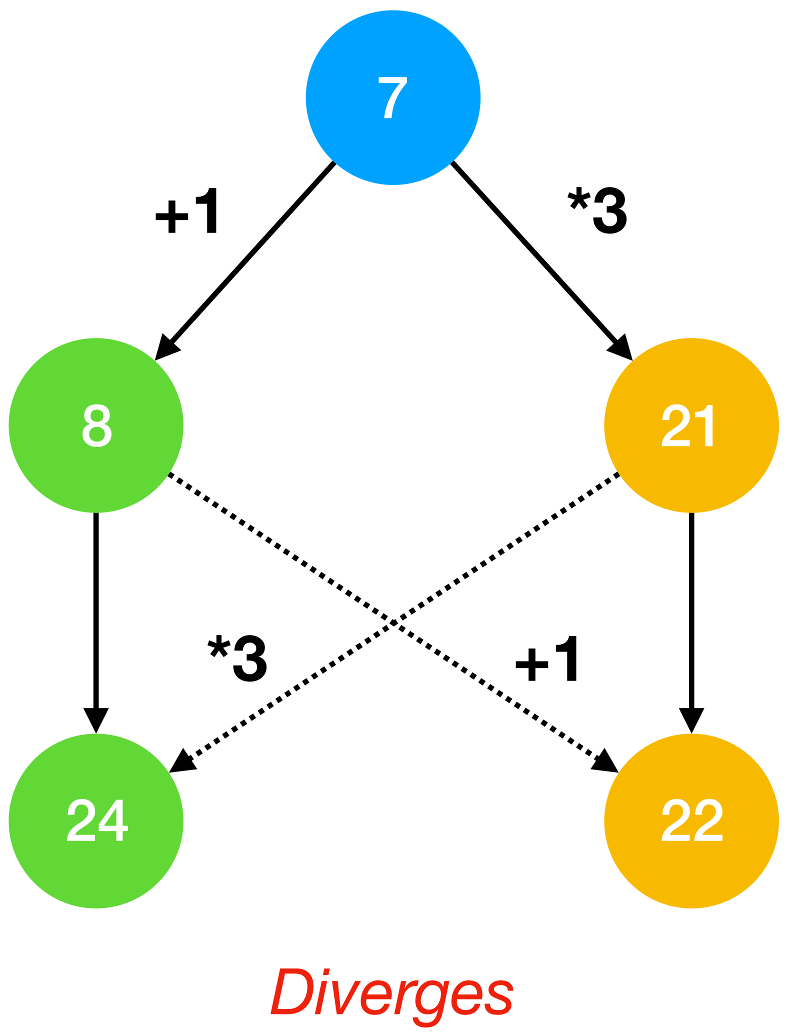


```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



Diverges

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



Addition and multiplication do not commute

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

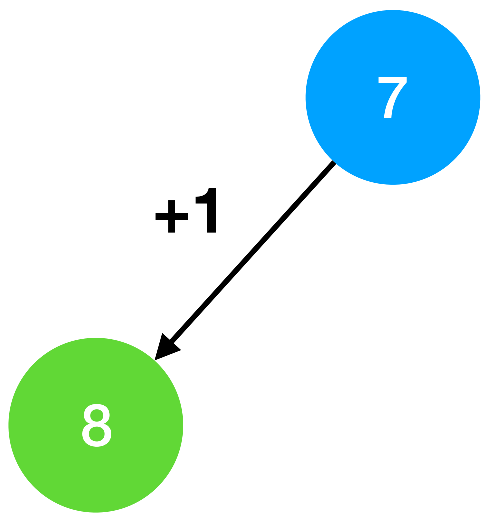
- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

7

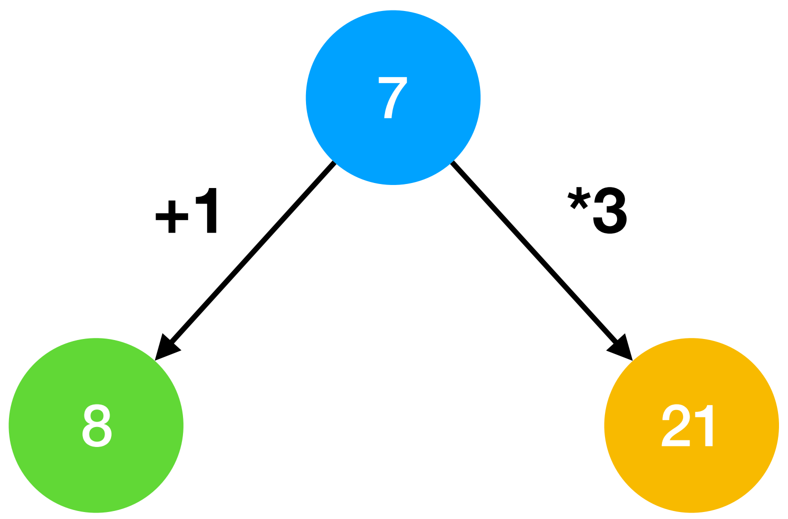
- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



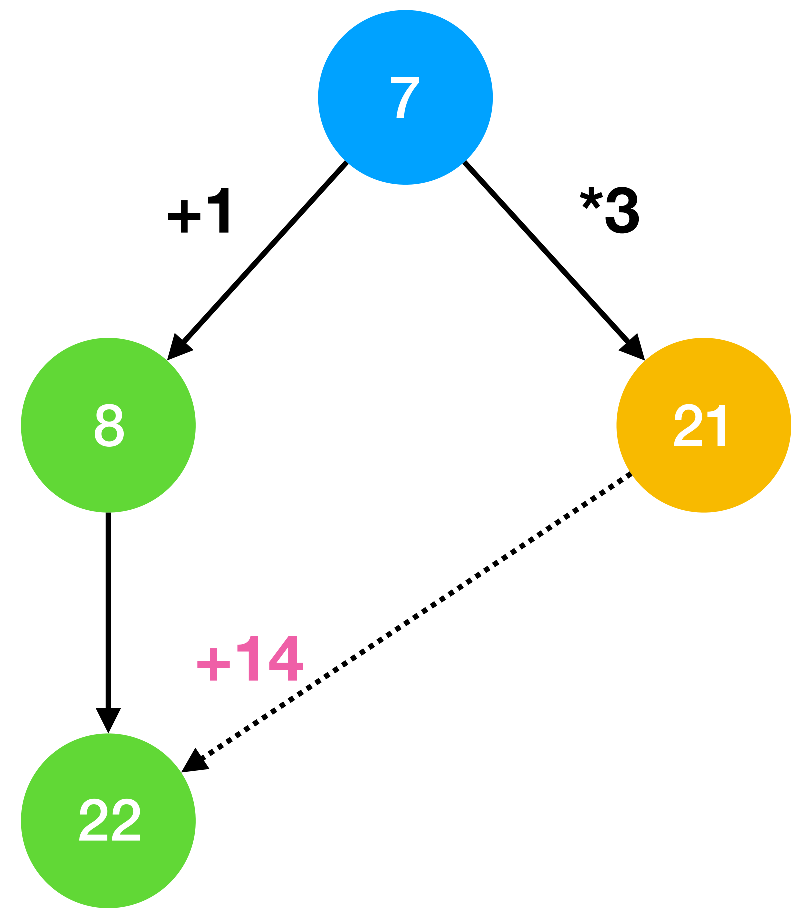
- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



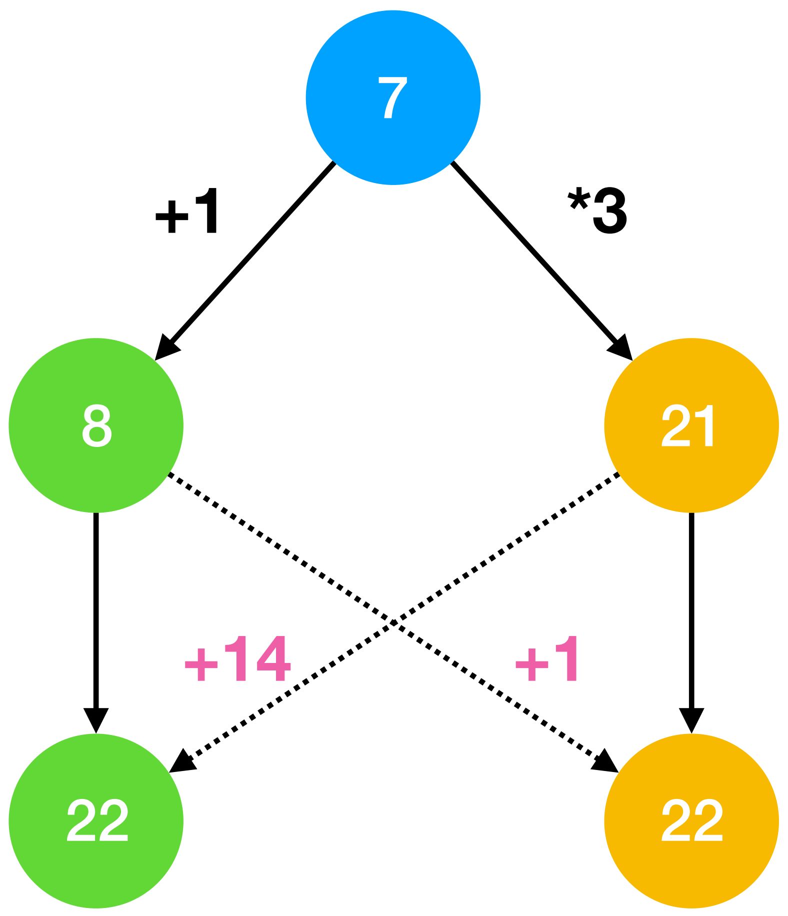
- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



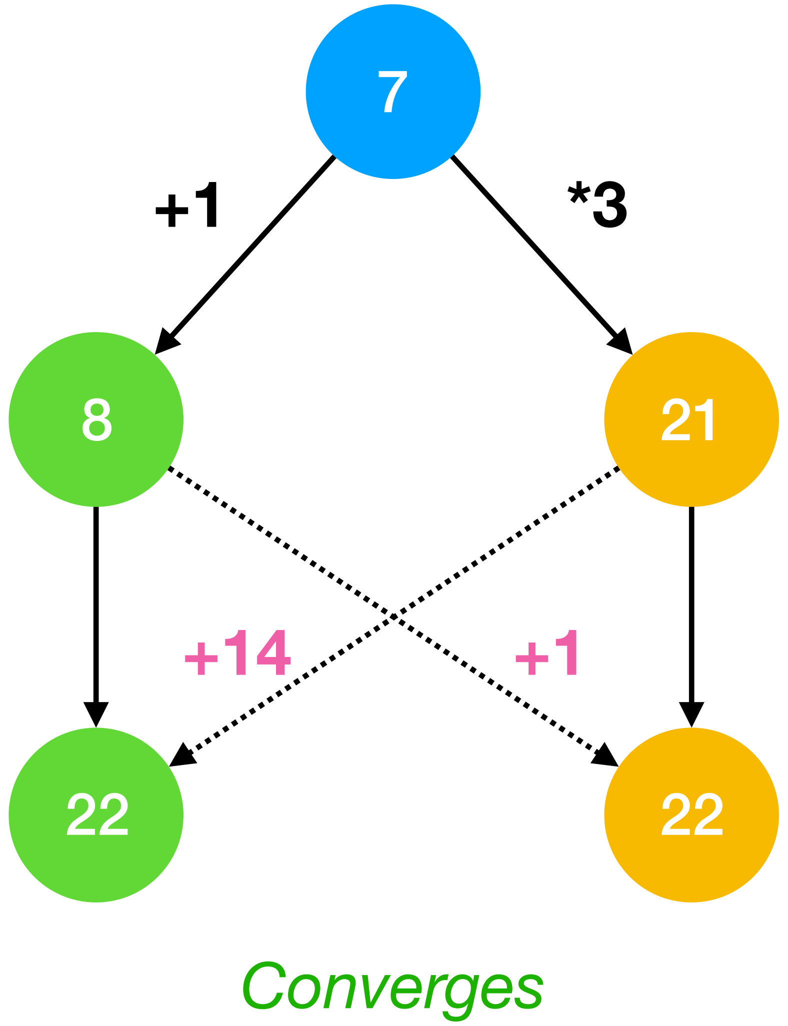
- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



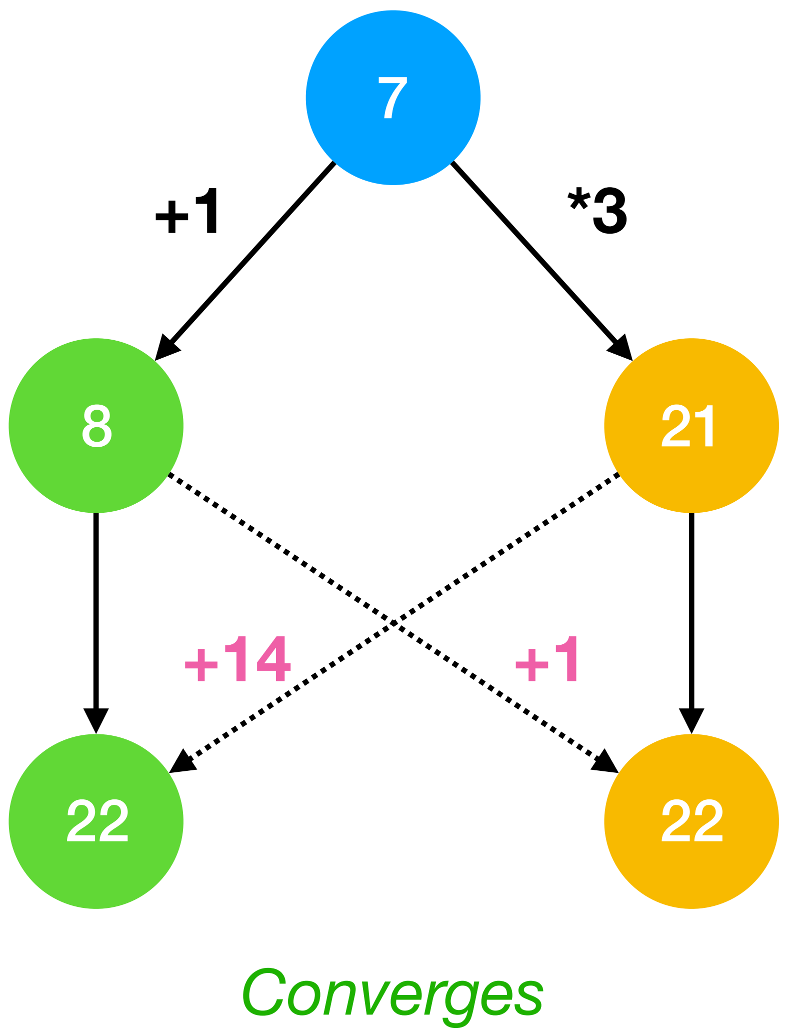
- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



- Capture the effect of multiplication through the commutative addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



- Capture the effect of multiplication through the commutative addition operation
- *CRDTs*

Conflict-free Replicated Data Types (CRDT)

Conflict-free Replicated Data Types (CRDT)

- CRDT is guaranteed to ensure *strong eventual consistency (SEC)*
  - ★ G-counters, PN-counters, OR-Sets, Graphs, Ropes, docs, sheets
  - ★ Simple interface for the clients of CRDTs

Conflict-free Replicated Data Types (CRDT)

- CRDT is guaranteed to ensure *strong eventual consistency (SEC)*
  - ★ G-counters, PN-counters, OR-Sets, Graphs, Ropes, docs, sheets
  - ★ Simple interface for the clients of CRDTs
- Need to reengineer every datatype to ensure SEC (commutativity)
  - ★ Do not mirror sequential counter parts => implementation & proof burden
  - ★ Do not compose!
    - ✦ `counter set` is not a composition of `counter` and `set` CRDTs

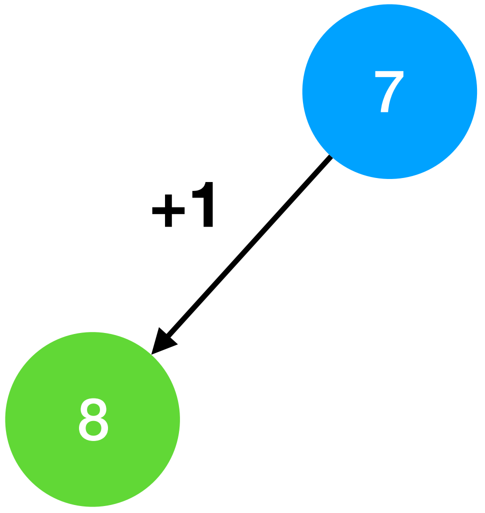
Can we *program & reason about* replicated data types as an extension of their sequential counterparts?

```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```

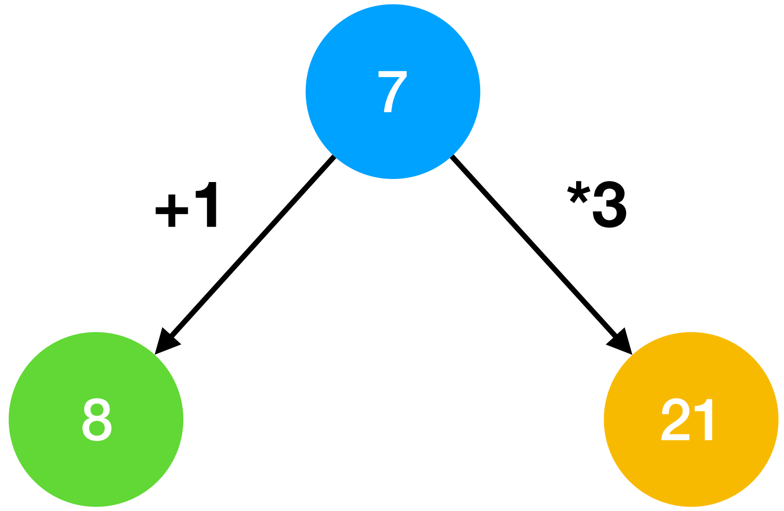
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```

7

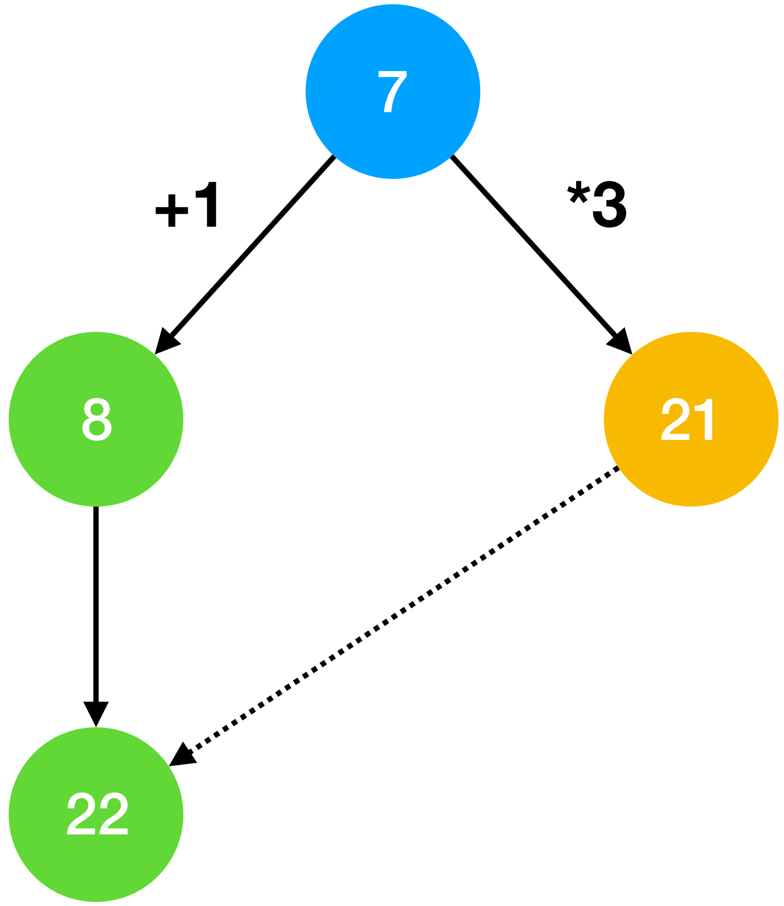
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



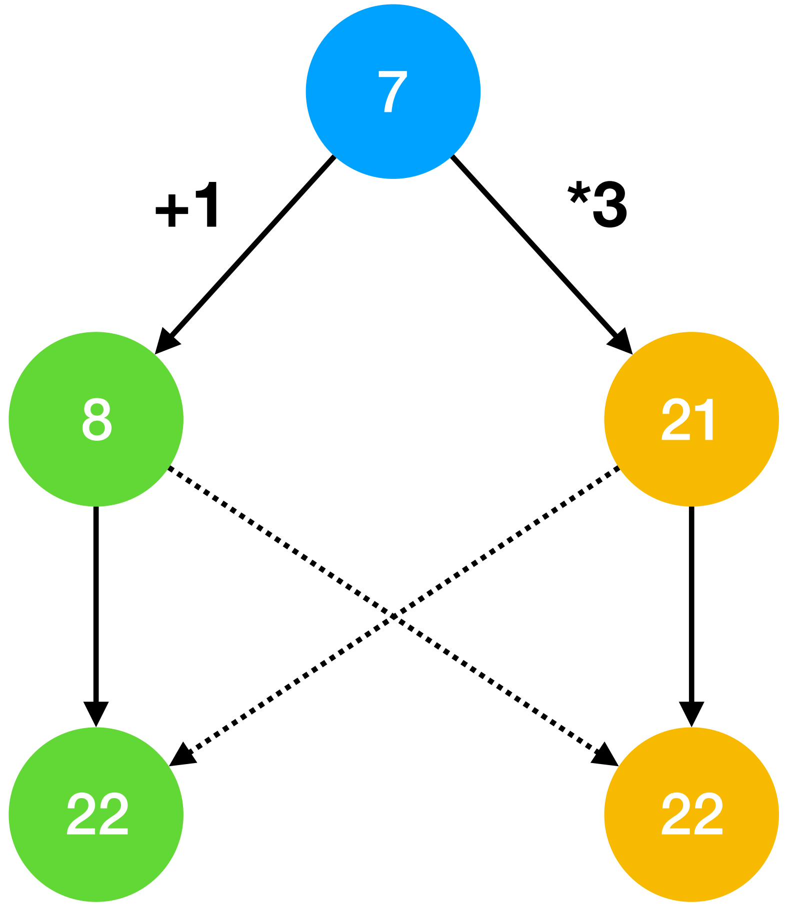
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



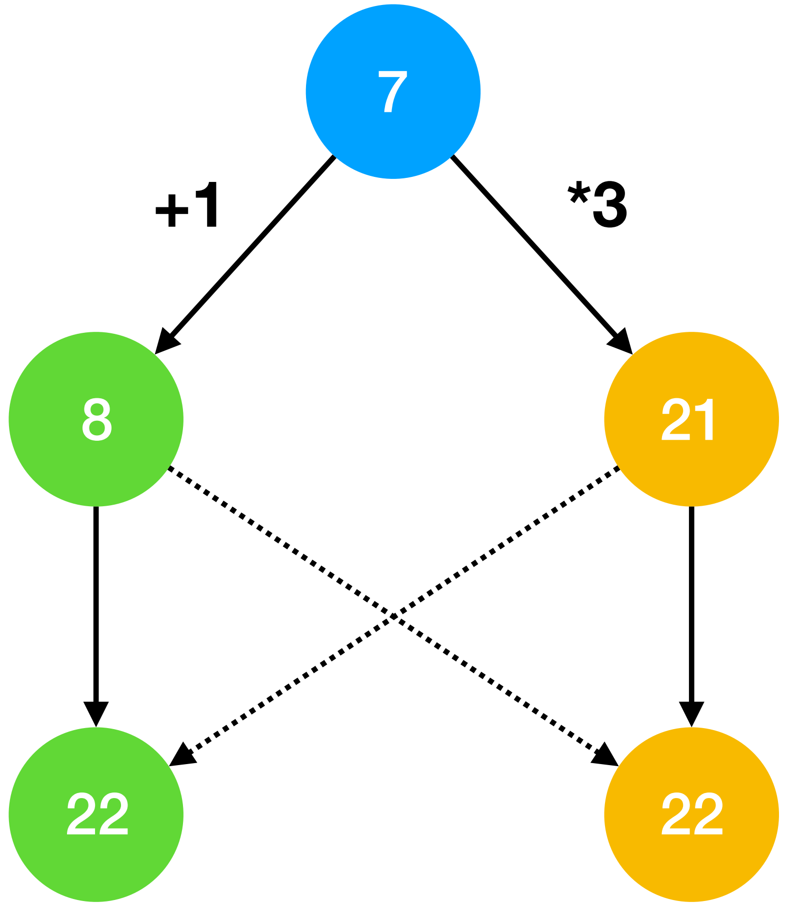
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```

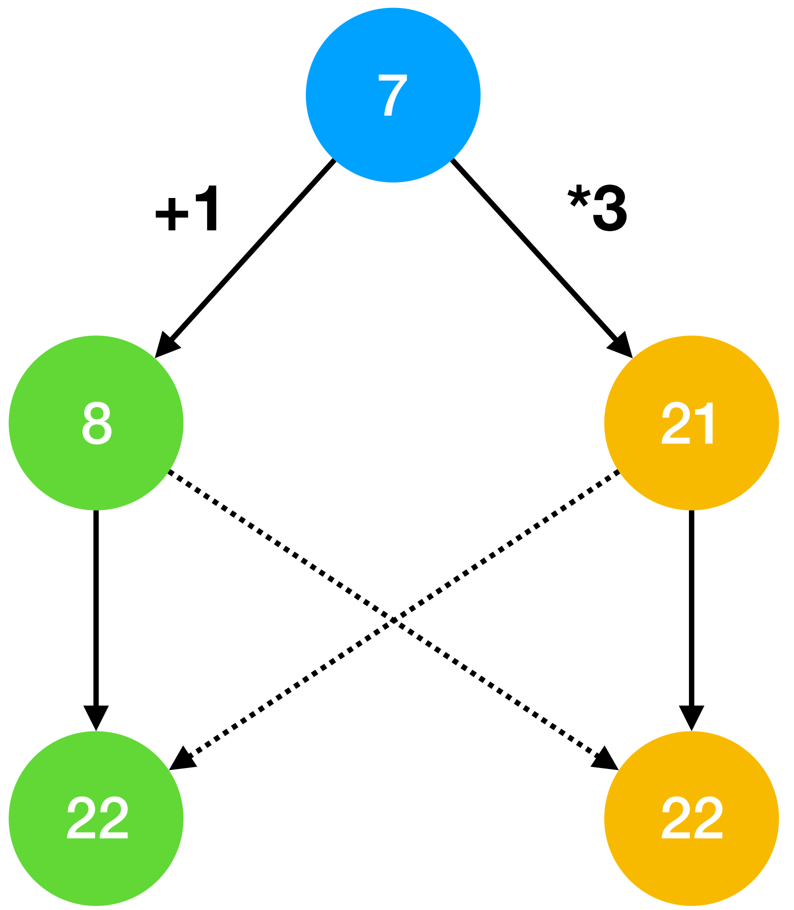


```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



22 = 7 + (8-1) + (21 -7)

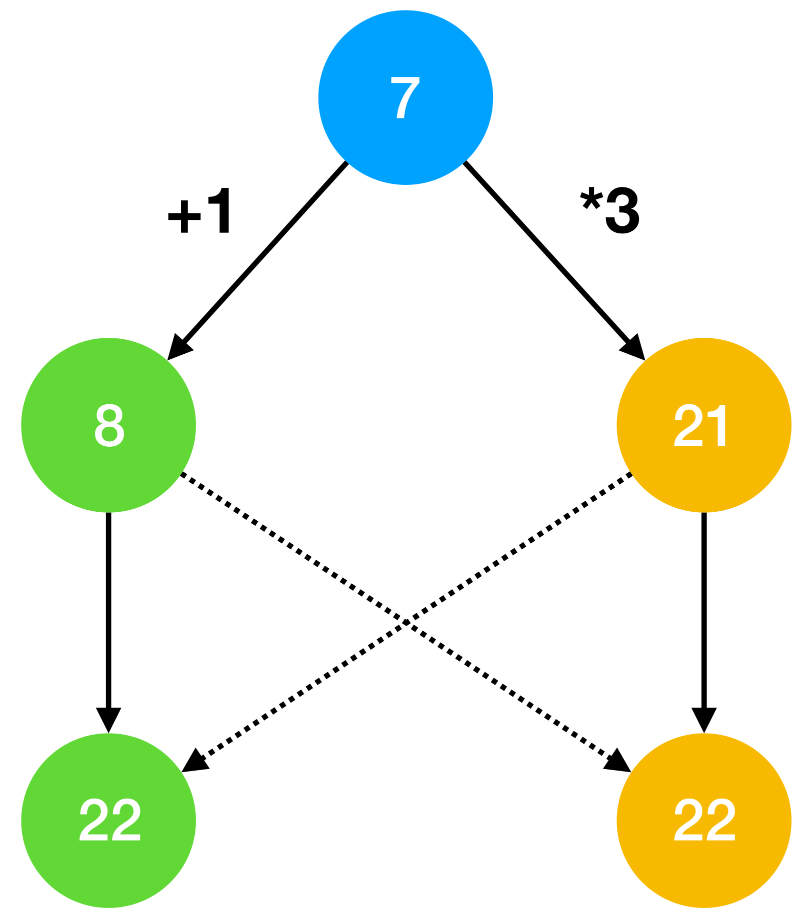
```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



22 = 7 + (8-1) + (21 -7)

- 3-way merge function makes the counter suitable for distribution

```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



22 = 7 + (8-1) + (21 -7)

- 3-way merge function makes the counter suitable for distribution
- Does not appeal to individual operations => independently extend data-type

Systems → PL

Systems → PL

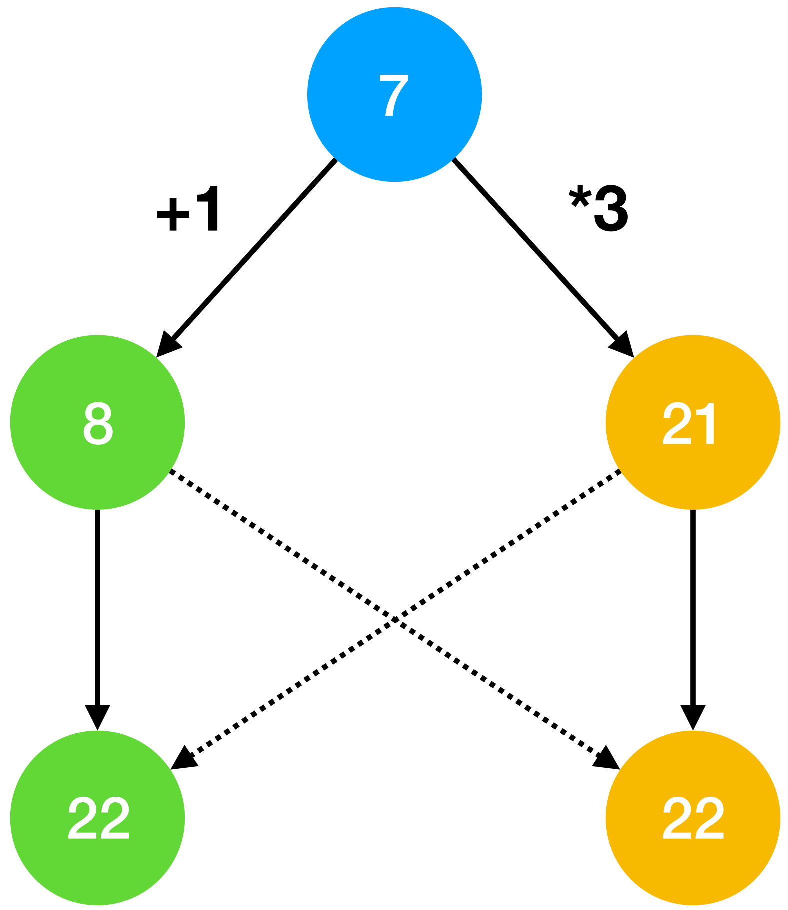
- CRDTs need to take care of systems level concerns such as exactly once delivery

Systems → PL

- CRDTs need to take care of systems level concerns such as exactly once delivery
- 3-way merge handles it automatically

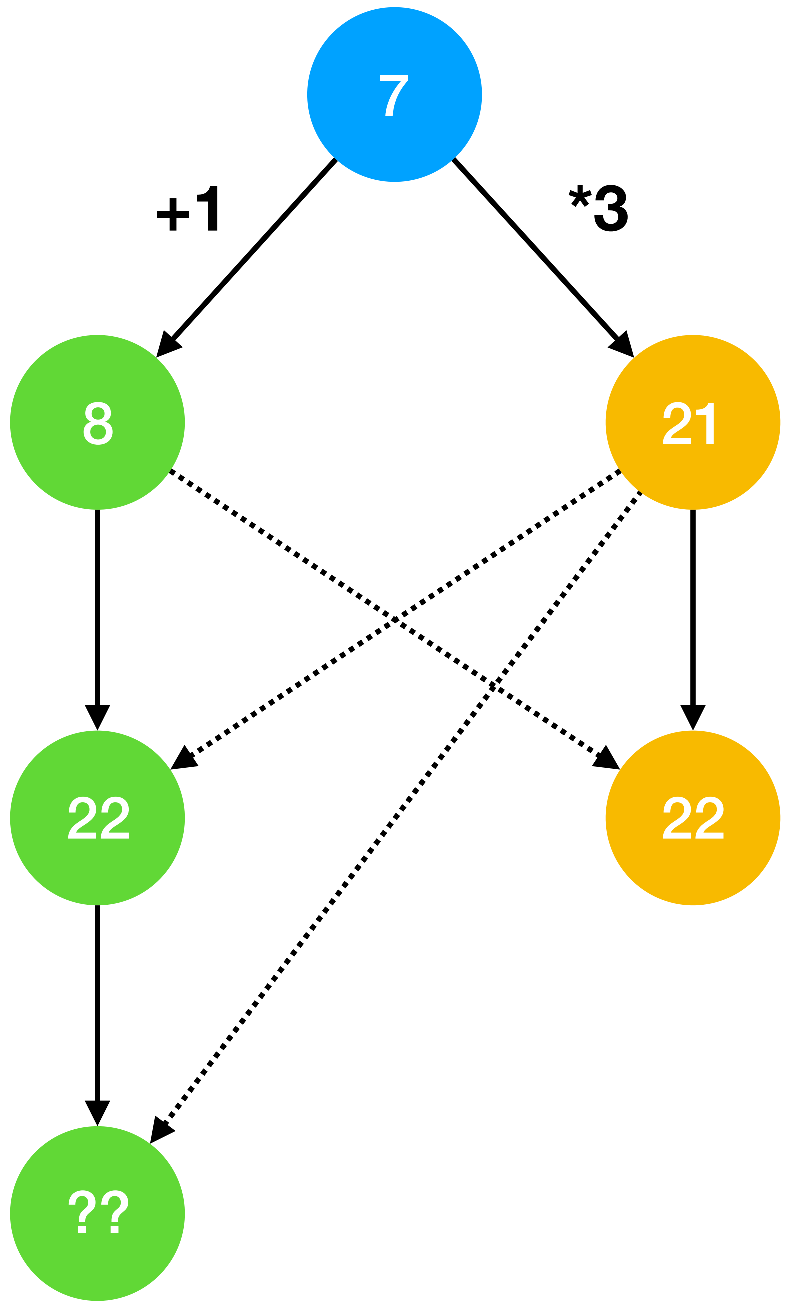
Systems → PL

- CRDTs need to take care of systems level concerns such as exactly once delivery
- 3-way merge handles it automatically



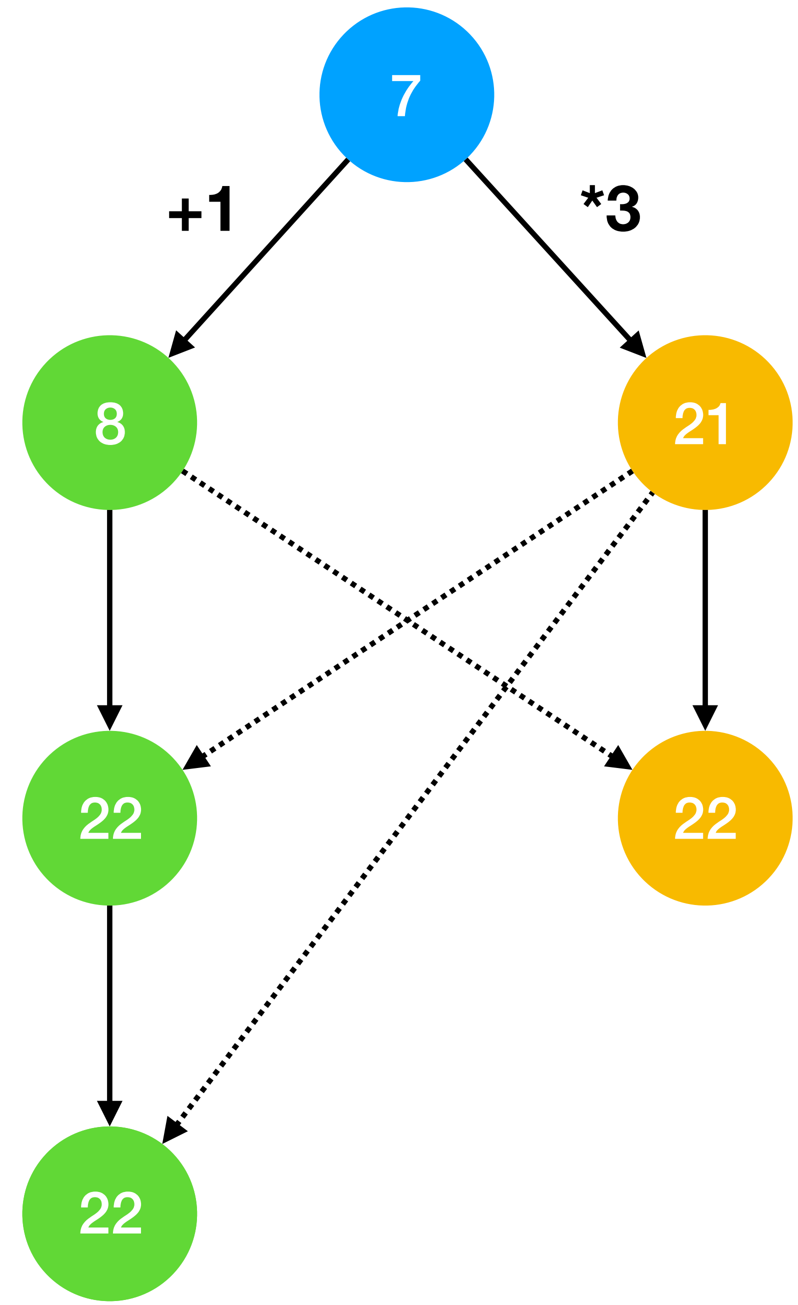
Systems → PL

- CRDTs need to take care of systems level concerns such as exactly once delivery
- 3-way merge handles it automatically



Systems → PL

- CRDTs need to take care of systems level concerns such as exactly once delivery
- 3-way merge handles it automatically



22 = 21 + (21-21) + (22 -21)

Does the 3-way merge idea generalise?

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations

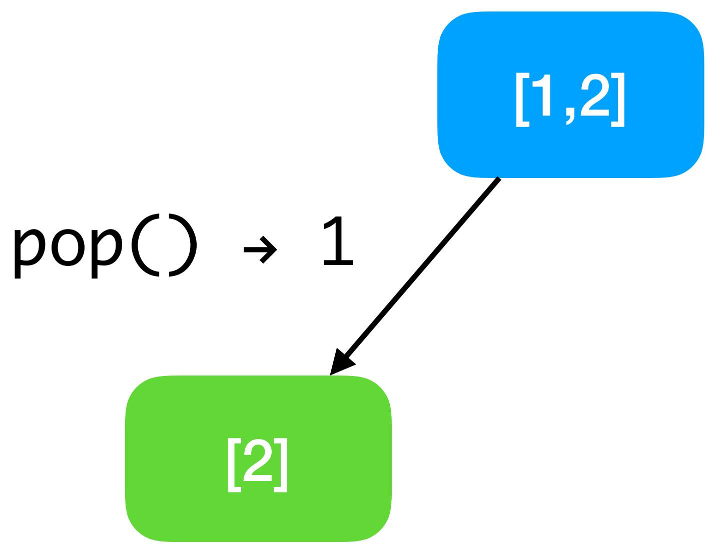
```ocaml
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations

[1,2]

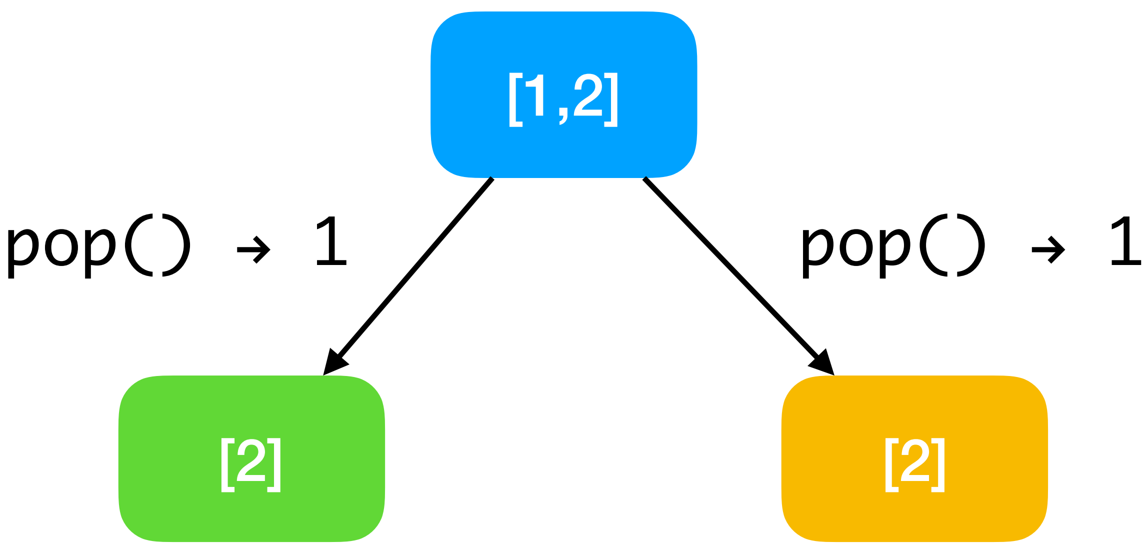
```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations



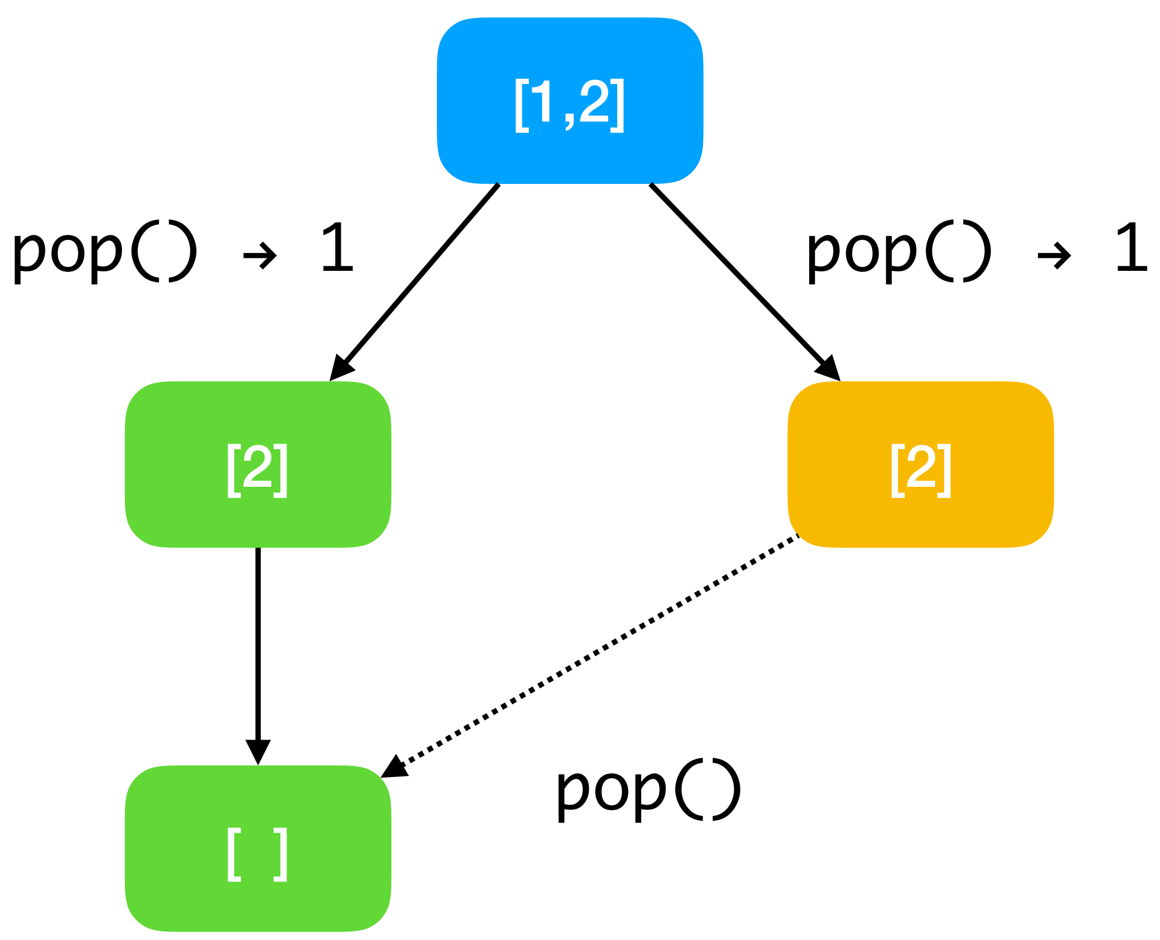
```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations



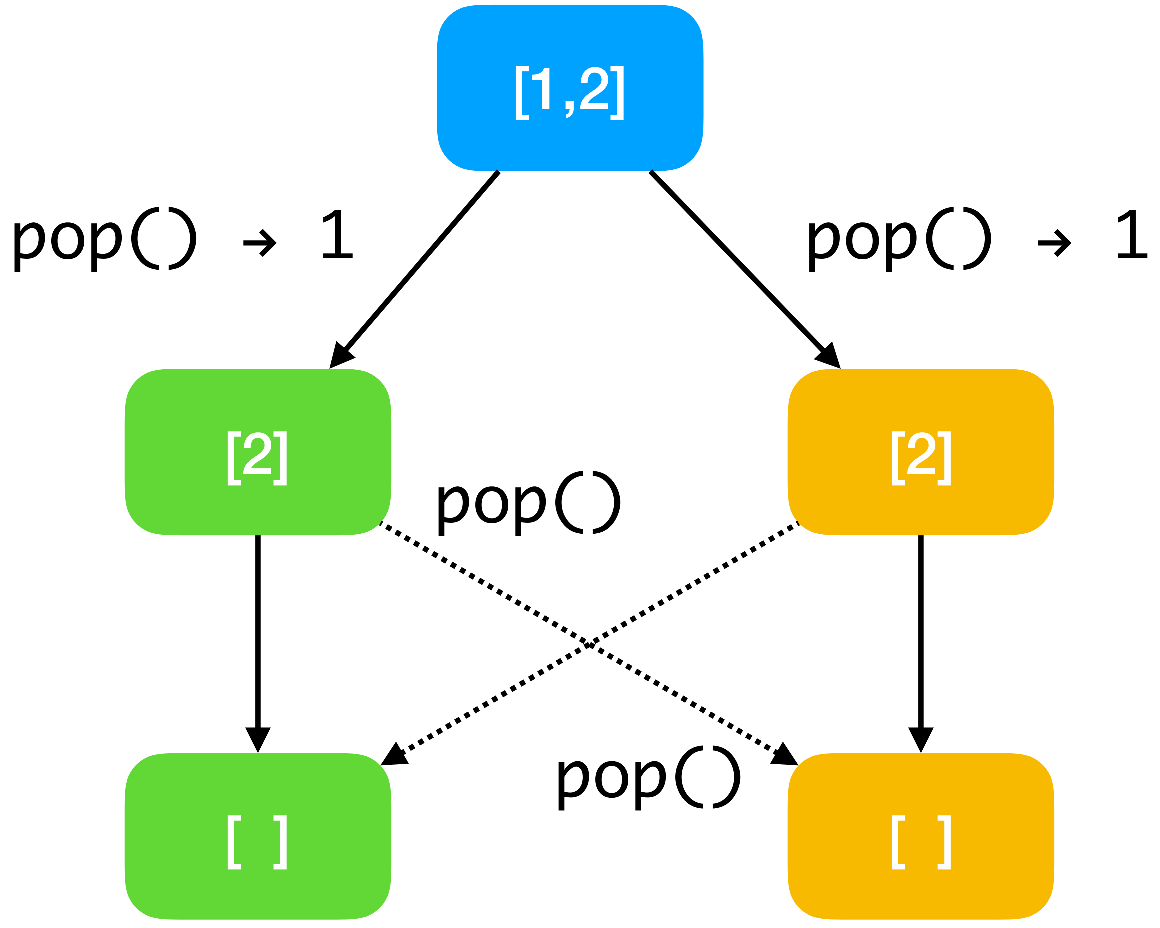
```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations



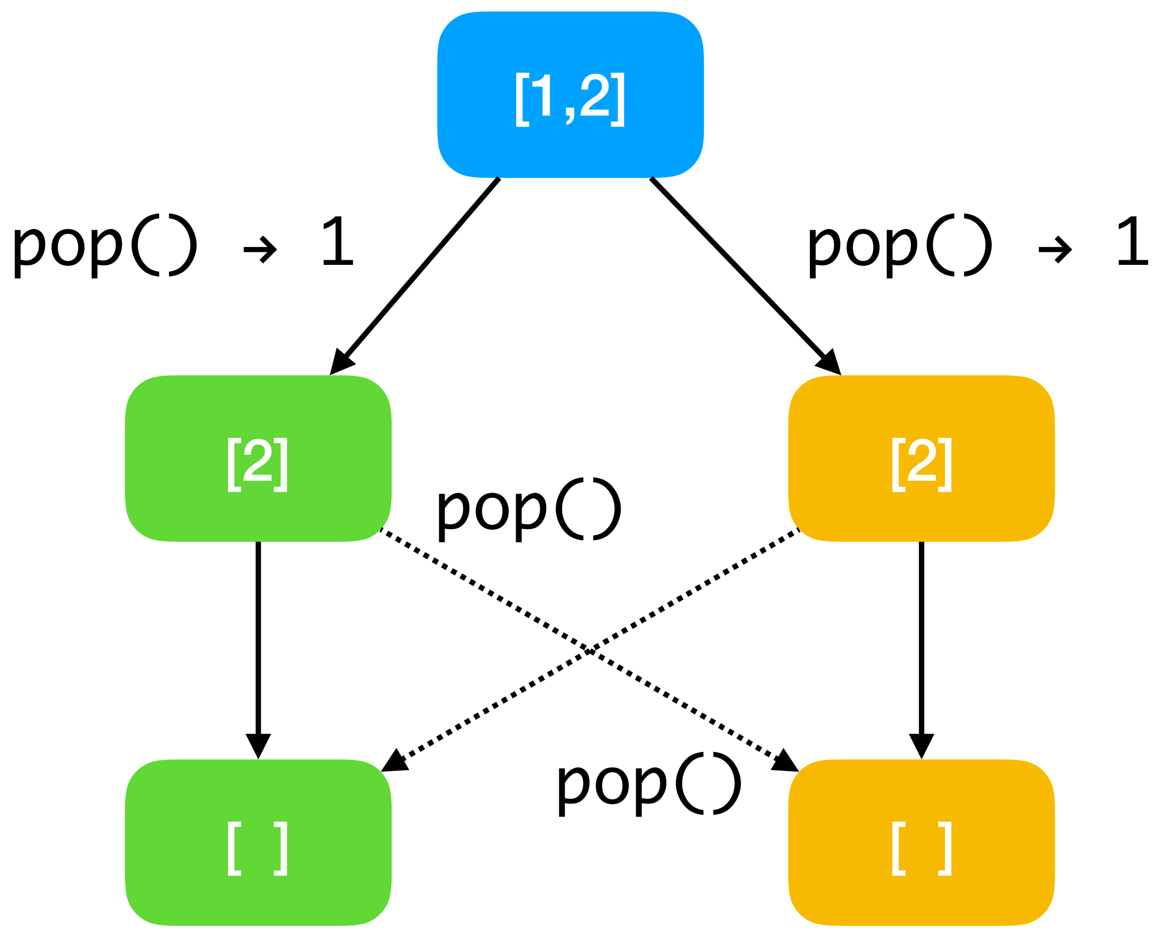
```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations



```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

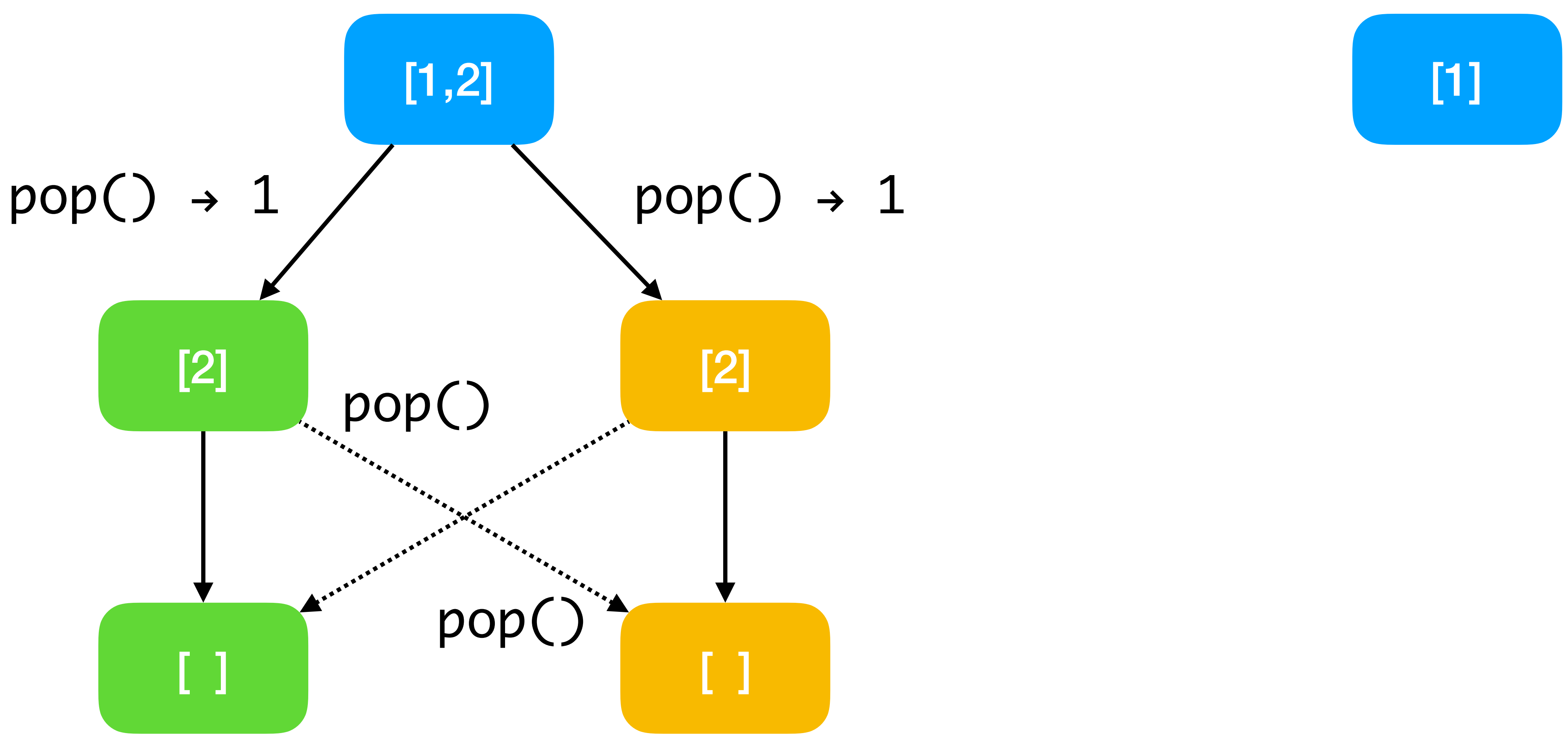
- Try replicating queues by asynchronously transmitting operations



- Convergence is not sufficient; *Intent* is not preserved

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

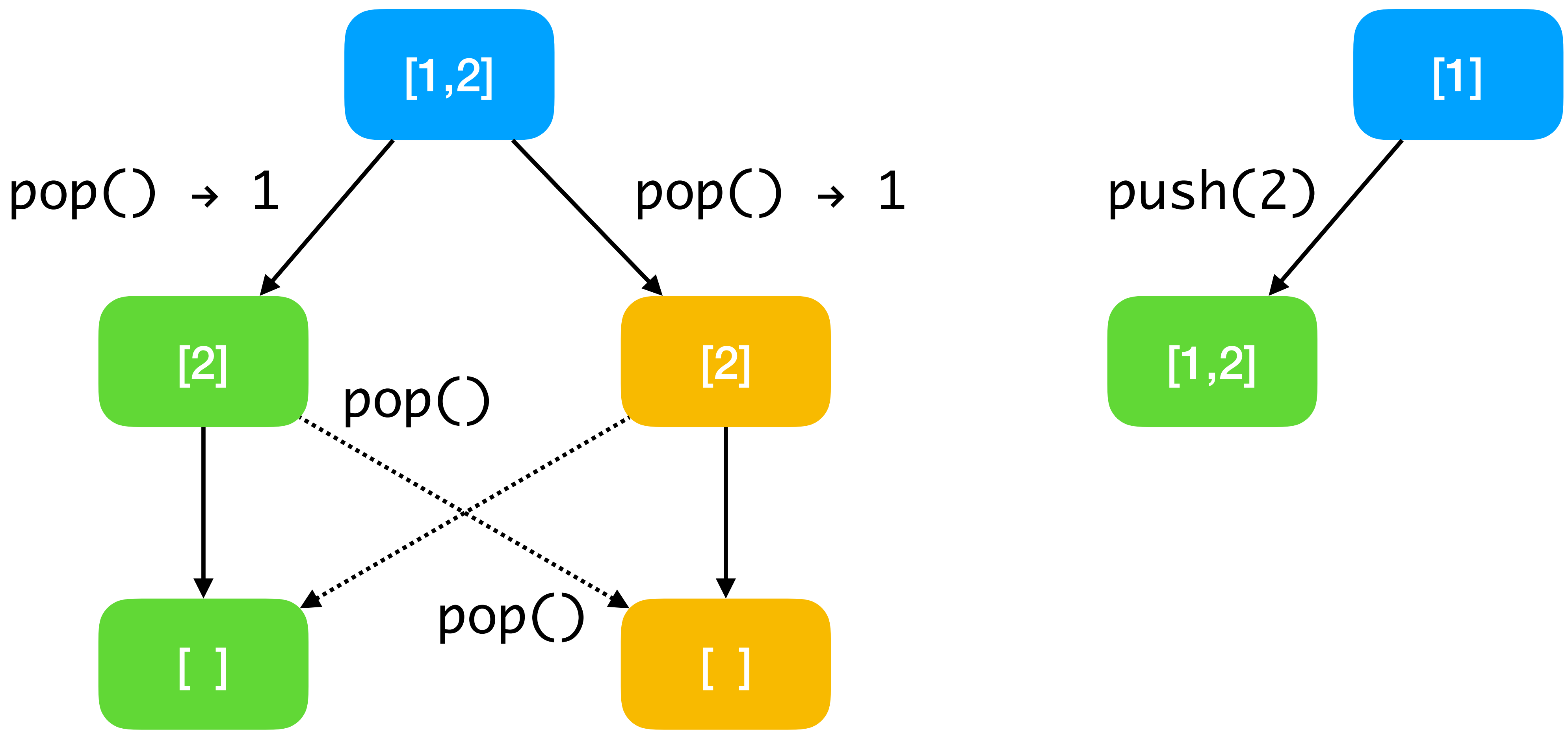
- Try replicating queues by asynchronously transmitting operations



- Convergence is not sufficient; *Intent* is not preserved

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

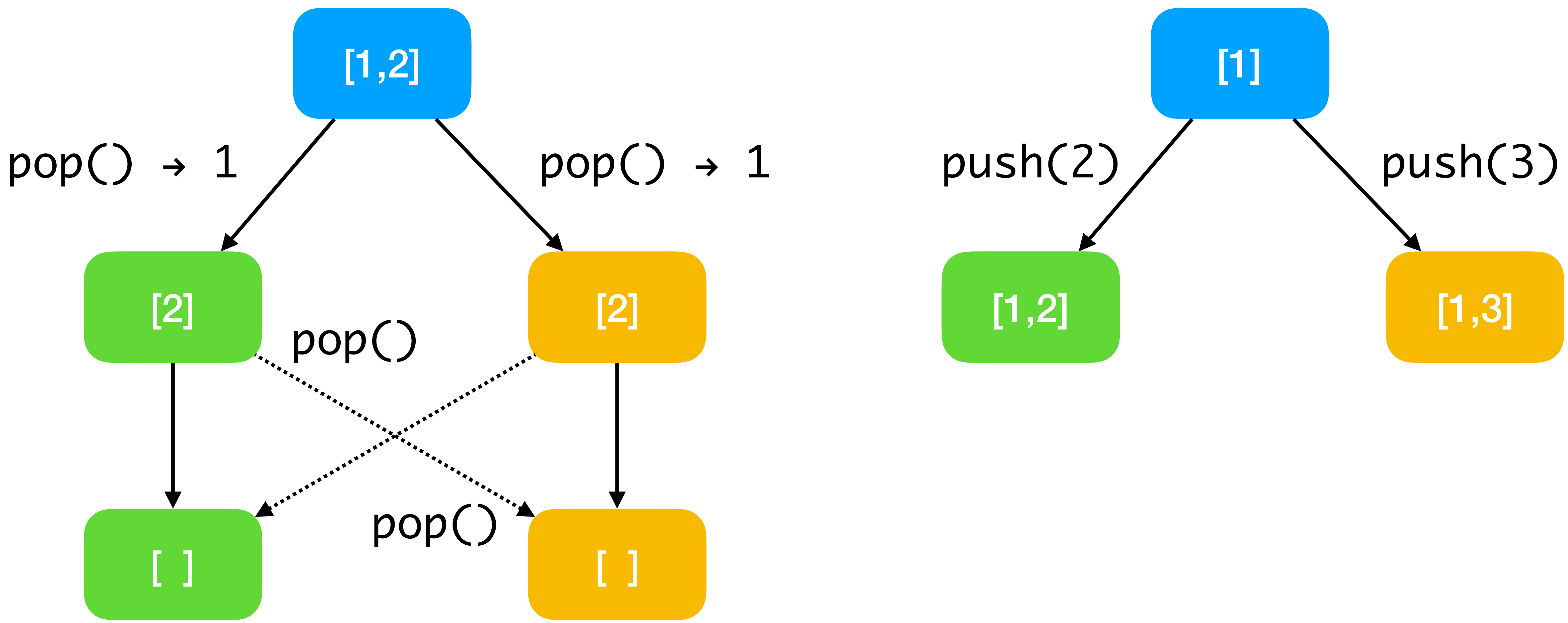
- Try replicating queues by asynchronously transmitting operations



- Convergence is not sufficient; *Intent* is not preserved

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

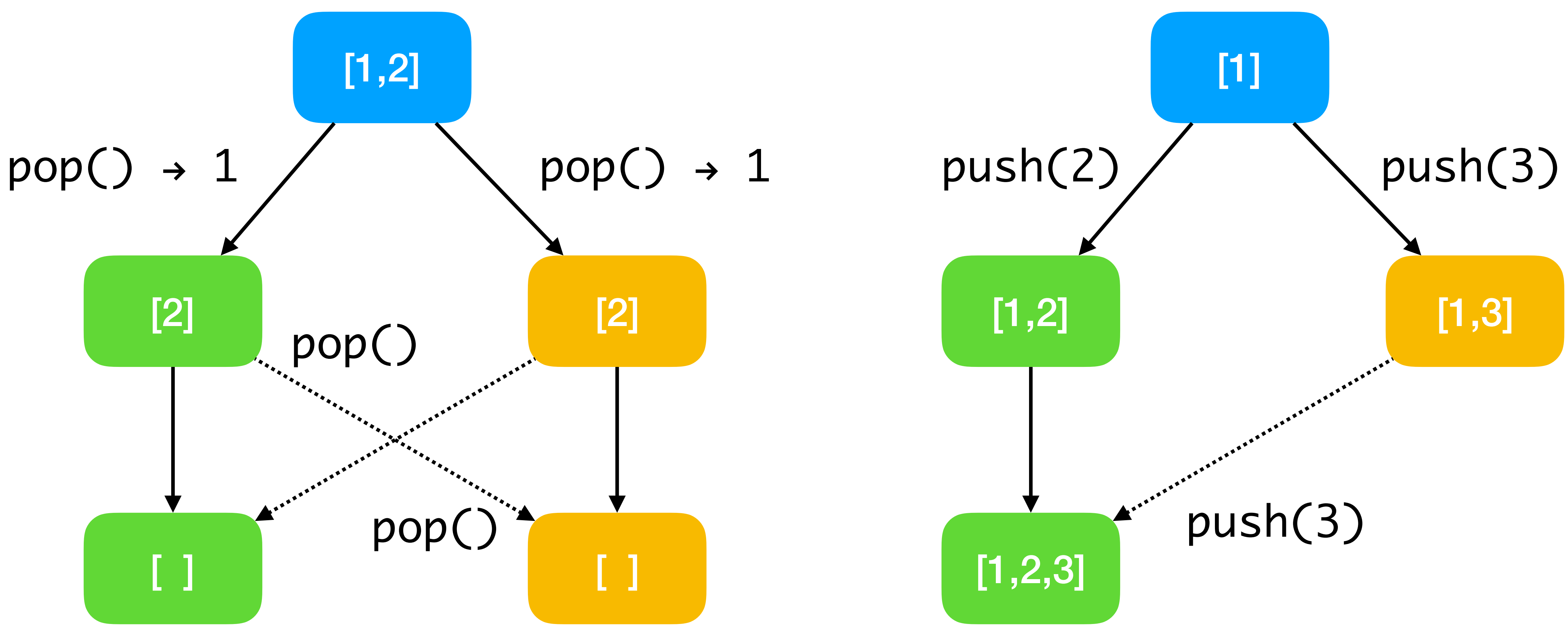
- Try replicating queues by asynchronously transmitting operations



- Convergence is not sufficient; *Intent* is not preserved

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

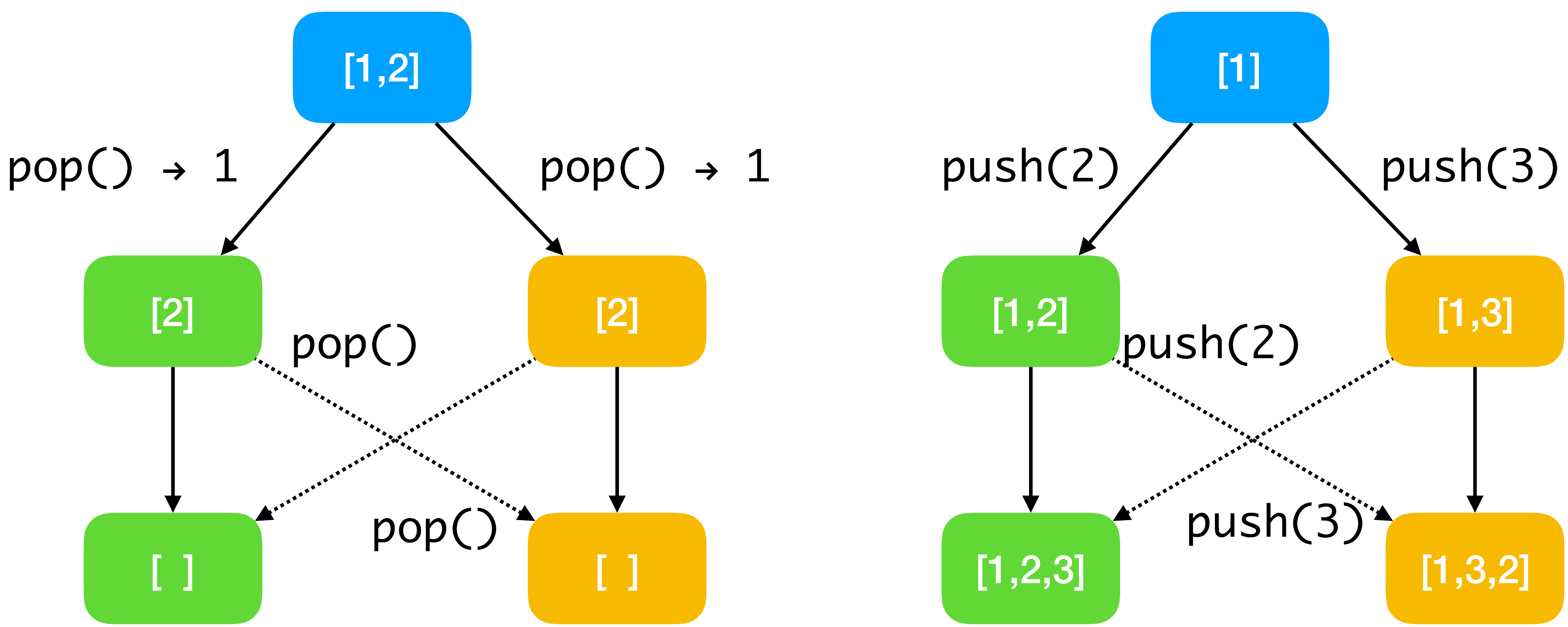
- Try replicating queues by asynchronously transmitting operations



- Convergence is not sufficient; *Intent* is not preserved

```
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end
```

- Try replicating queues by asynchronously transmitting operations



- Convergence is not sufficient; *Intent* is not preserved

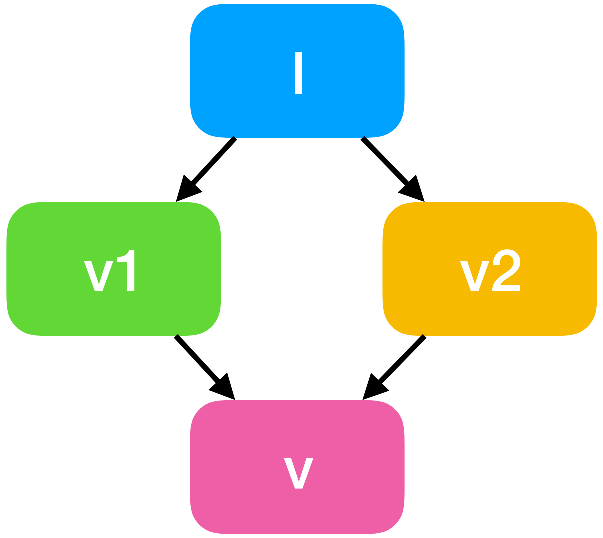
Concretising Intent

Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*

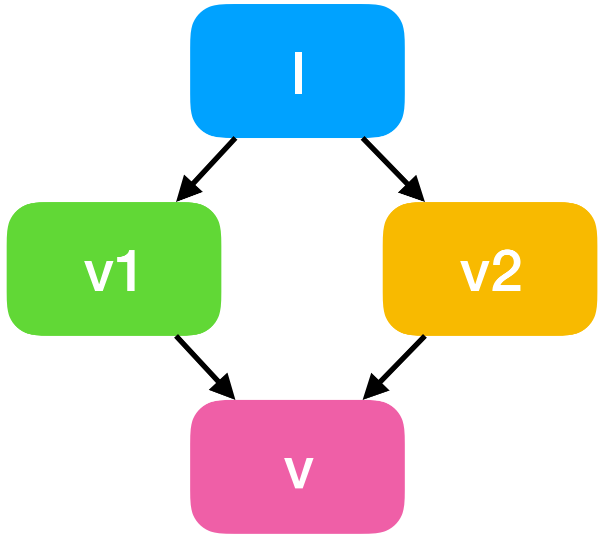
Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*



Concretising Intent

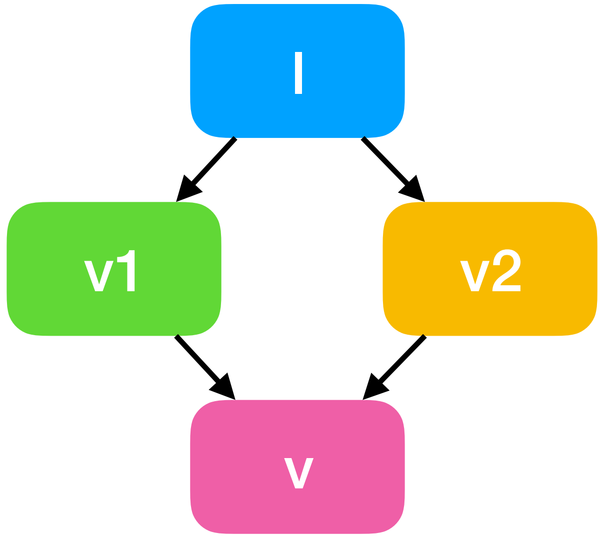
- Intent is a woolly term
    - ★ *How can we formalise the intent of operations on a data structure?*
- For a replicated queue,



Concretising Intent

- Intent is a woolly term
    - ★ *How can we formalise the intent of operations on a data structure?*
- For a replicated queue,

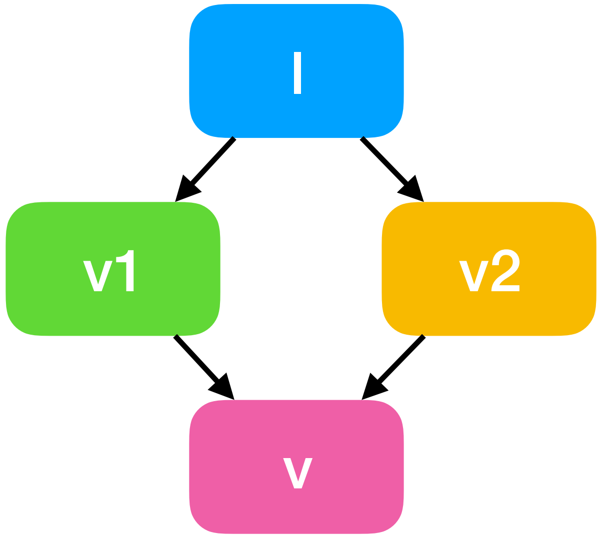
1. Any element popped in either v1 or v2 does not remain in v



Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*
- For a replicated queue,

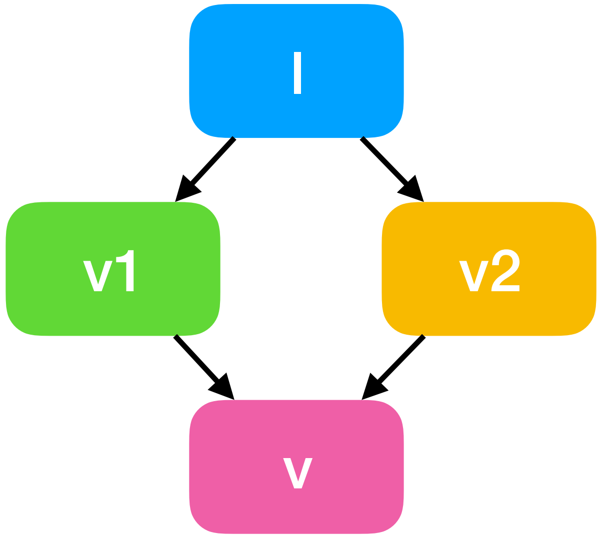
1. Any element popped in either v1 or v2 does not remain in v
2. Any element pushed into either v1 or v2 appears in v



Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*
- For a replicated queue,

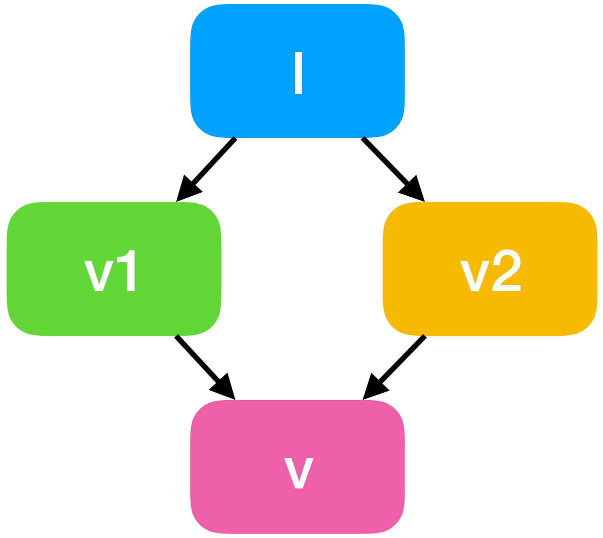
1. Any element popped in either v1 or v2 does not remain in v
2. Any element pushed into either v1 or v2 appears in v
3. An element that remains untouched in l, v1, v2 remains in v



Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*
- For a replicated queue,

1. Any element popped in either v1 or v2 does not remain in v
2. Any element pushed into either v1 or v2 appears in v
3. An element that remains untouched in l, v1, v2 remains in v
4. Order of pairs of elements in l, v1, v2 must be preserved in m, if those elements are present in v.



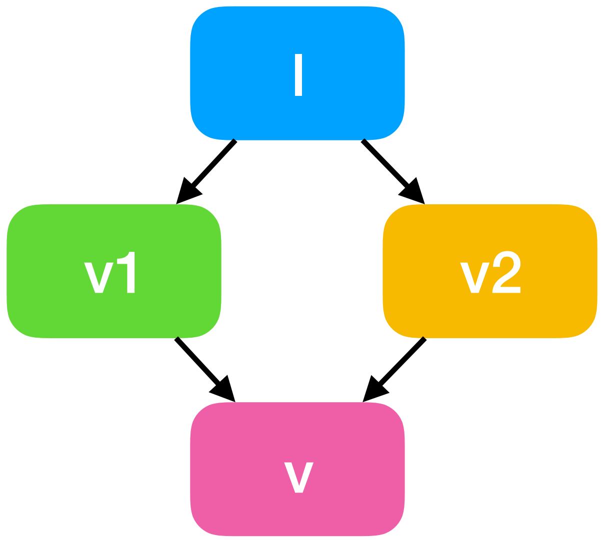
Relational Specification

Relational Specification

- Let's define relations R_{mem} and R_{ob} to capture membership and ordering
  - ★ R_{mem} [1,2,3] = {1,2,3}
  - ★ R_{ob} [1,2,3] = { (1,2), (1,3), (2,3) }

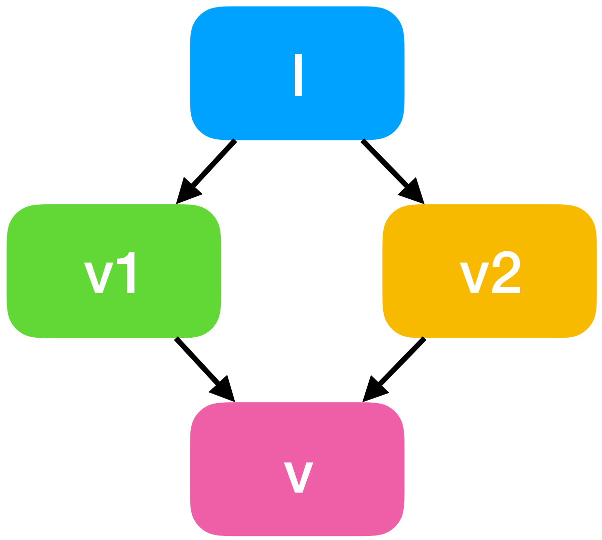
Relational Specification

- Let's define relations R_{mem} and R_{ob} to capture membership and ordering
  - ★ R_{mem} [1,2,3] = {1,2,3}
  - ★ R_{ob} [1,2,3] = { (1,2), (1,3), (2,3) }



Relational Specification

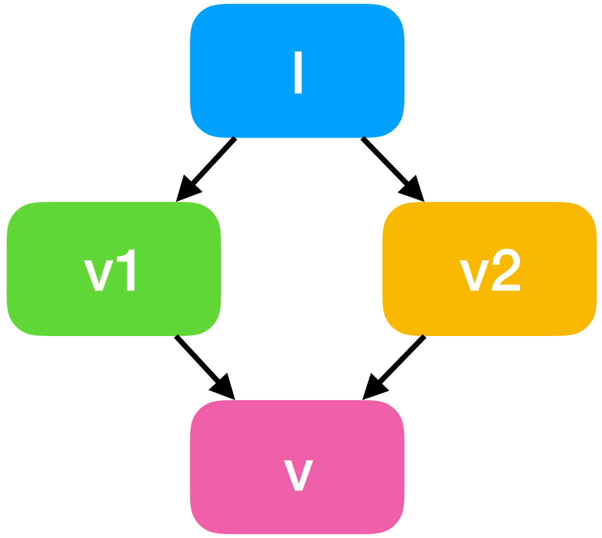
- Let's define relations R_{mem} and R_{ob} to capture membership and ordering
  - ★ R_{mem} [1,2,3] = {1,2,3}
  - ★ R_{ob} [1,2,3] = { (1,2), (1,3), (2,3) }



$$\begin{aligned} R_{mem}(v) \quad &= \quad R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \\ &\qquad \cup \quad R_{mem}(v_1) - R_{mem}(l) \quad \cup \quad R_{mem}(v_2) - R_{mem}(l) \end{aligned}$$

Relational Specification

- Let's define relations R_{mem} and R_{ob} to capture membership and ordering
  - ★ R_{mem} [1,2,3] = {1,2,3}
  - ★ R_{ob} [1,2,3] = { (1,2), (1,3), (2,3) }

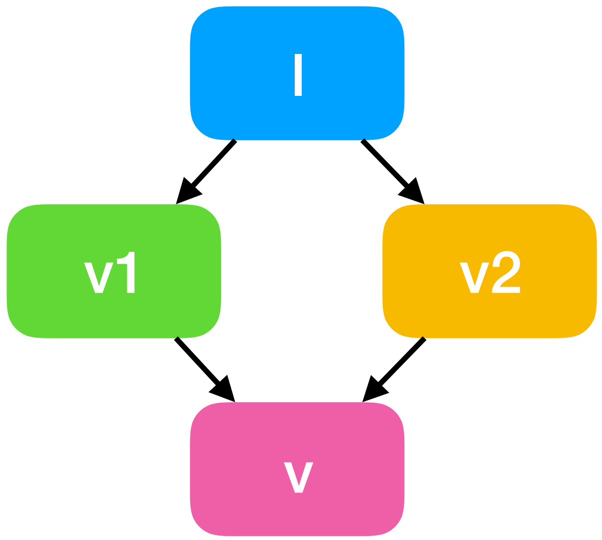


$$R_{mem}(v) = \boxed{R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2)} \cup R_{mem}(v_1) - R_{mem}(l) \cup R_{mem}(v_2) - R_{mem}(l)$$

1. Any element popped in either v1 or v2 does not remain in v

Relational Specification

- Let's define relations R_{mem} and R_{ob} to capture membership and ordering
  - ★ R_{mem} [1,2,3] = {1,2,3}
  - ★ R_{ob} [1,2,3] = { (1,2), (1,3), (2,3) }



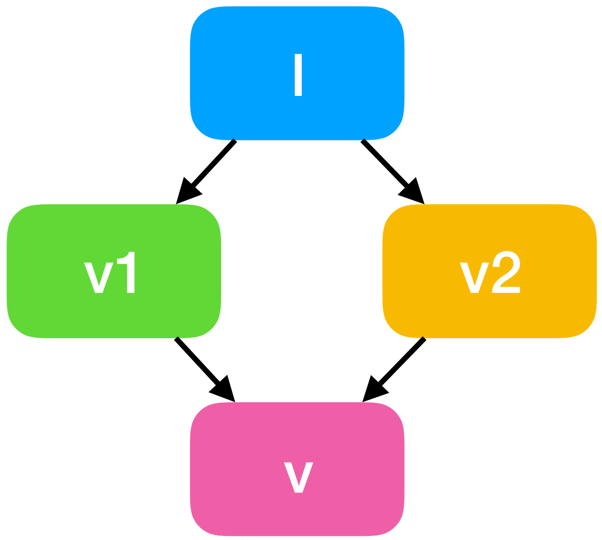
$$R_{mem}(v) = R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \cup R_{mem}(v_1) - R_{mem}(l) \cup R_{mem}(v_2) - R_{mem}(l)$$

1. Any element popped in either v1 or v2 does not remain in v

2. Any element pushed into either v1 or v2 appears in v

Relational Specification

- Let's define relations R_{mem} and R_{ob} to capture membership and ordering
  - ★ R_{mem} [1,2,3] = {1,2,3}
  - ★ R_{ob} [1,2,3] = { (1,2), (1,3), (2,3) }

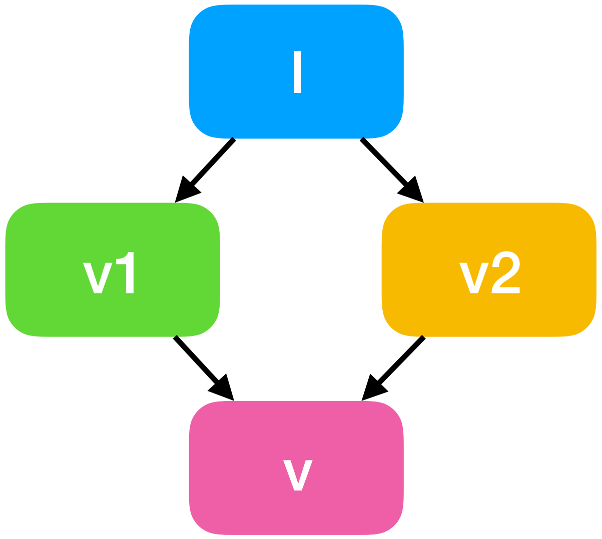


$$R_{mem}(v) = R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \cup R_{mem}(v_1) - R_{mem}(l) \cup R_{mem}(v_2) - R_{mem}(l)$$

1. Any element popped in either v1 or v2 does not remain in v
2. Any element pushed into either v1 or v2 appears in v
3. An element that remains untouched in l, v1, v2 remains in v

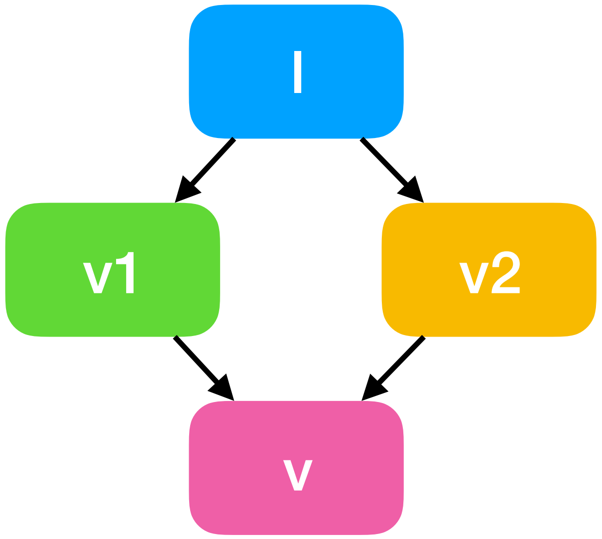
Relational Specification

$$
\begin{aligned}
R_{ob}(v) \quad \supseteq \quad & (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \\
& \quad \cup \quad R_{ob}(v_1) - R_{ob}(l) \quad \cup \quad R_{ob}(v_2) - R_{ob}(l)) \\
\cap \quad & (R_{mem}(v) \times R_{mem}(v))
\end{aligned}
$$



Relational Specification

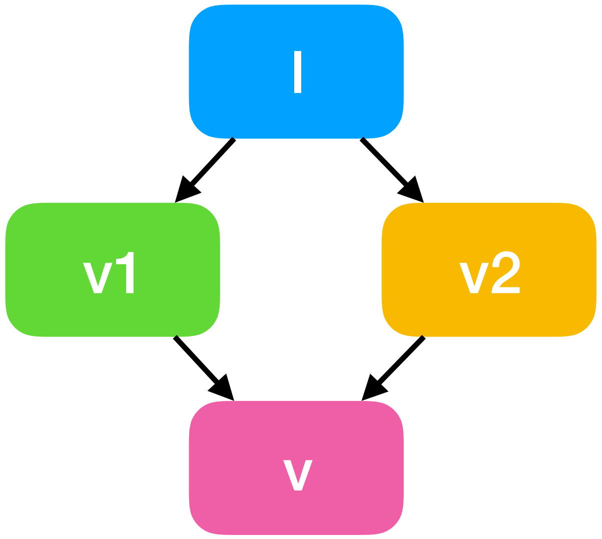
$$
\begin{aligned}
R_{ob}(v) \quad \supseteq \quad & (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \\
& \cup \quad R_{ob}(v_1) - R_{ob}(l) \quad \cup \quad R_{ob}(v_2) - R_{ob}(l)) \\
& \cap \quad (R_{mem}(v) \times R_{mem}(v))
\end{aligned}
$$



- RHS has to be confined to $R_{mem}(v)$ x $R_{mem}(v)$ since certain orders might be missing
  - ★ Consider l = [0], v1= [0,1], v2 = [ ], v = [1]

Relational Specification

$$
\begin{aligned}
\boxed{R_{ob}(v) \quad \supseteq} \quad & (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \\
& \quad \cup \quad R_{ob}(v_1) - R_{ob}(l) \quad \cup \quad R_{ob}(v_2) - R_{ob}(l)) \\
\cap \quad & (R_{mem}(v) \times R_{mem}(v))
\end{aligned}
$$

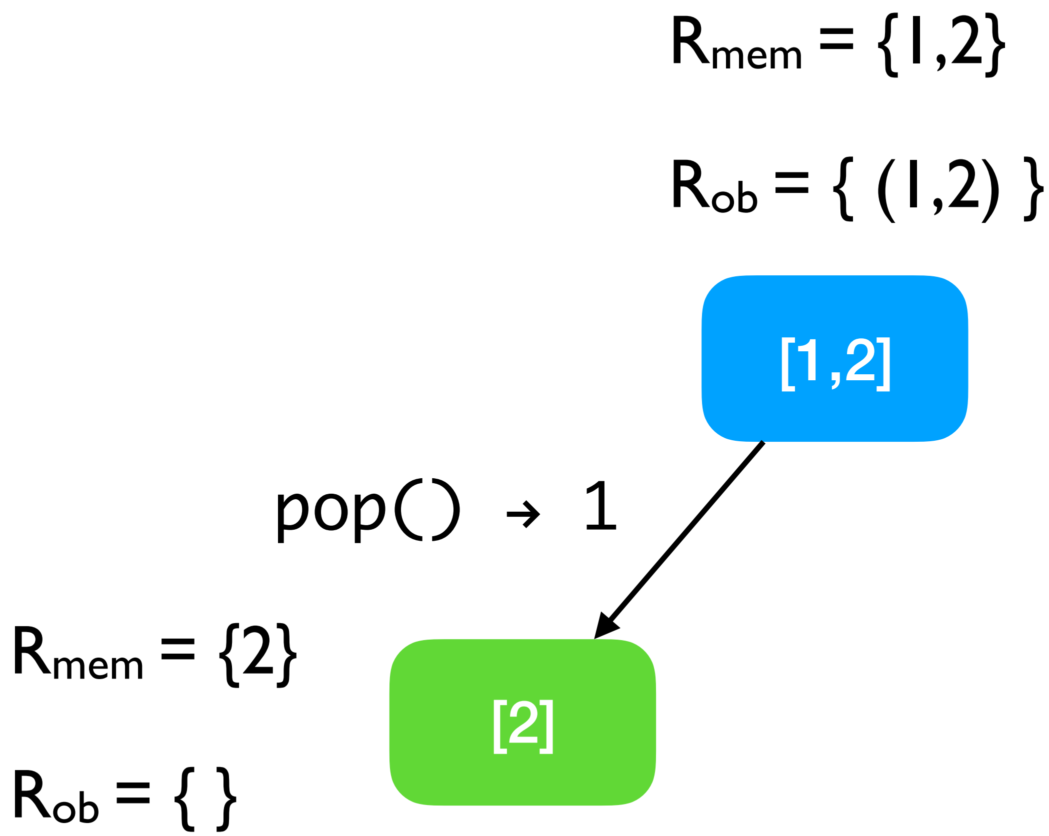


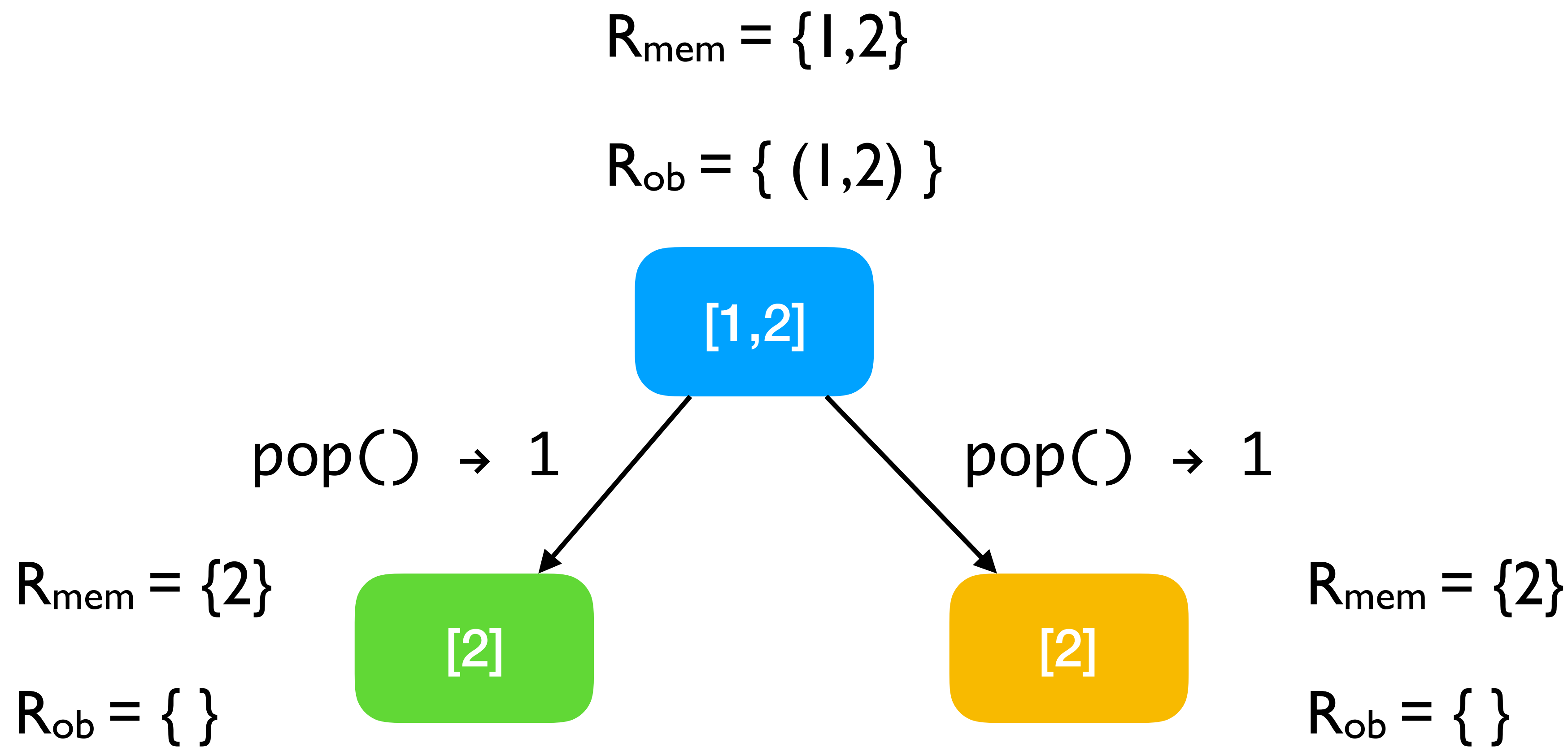
- RHS has to be confined to $R_{mem}(v)$ x $R_{mem}(v)$ since certain orders might be missing
  - ★ Consider l = [0], v1= [0,1], v2 = [ ], v = [1]
- RHS is an underspecification since orders between concurrent insertions will only be present in $R_{ob}(v)$
  - ★ Consider l = [ ], v1= [0], v2 = [1], v = [0,1]

$R_{mem} = \{1,2\}$

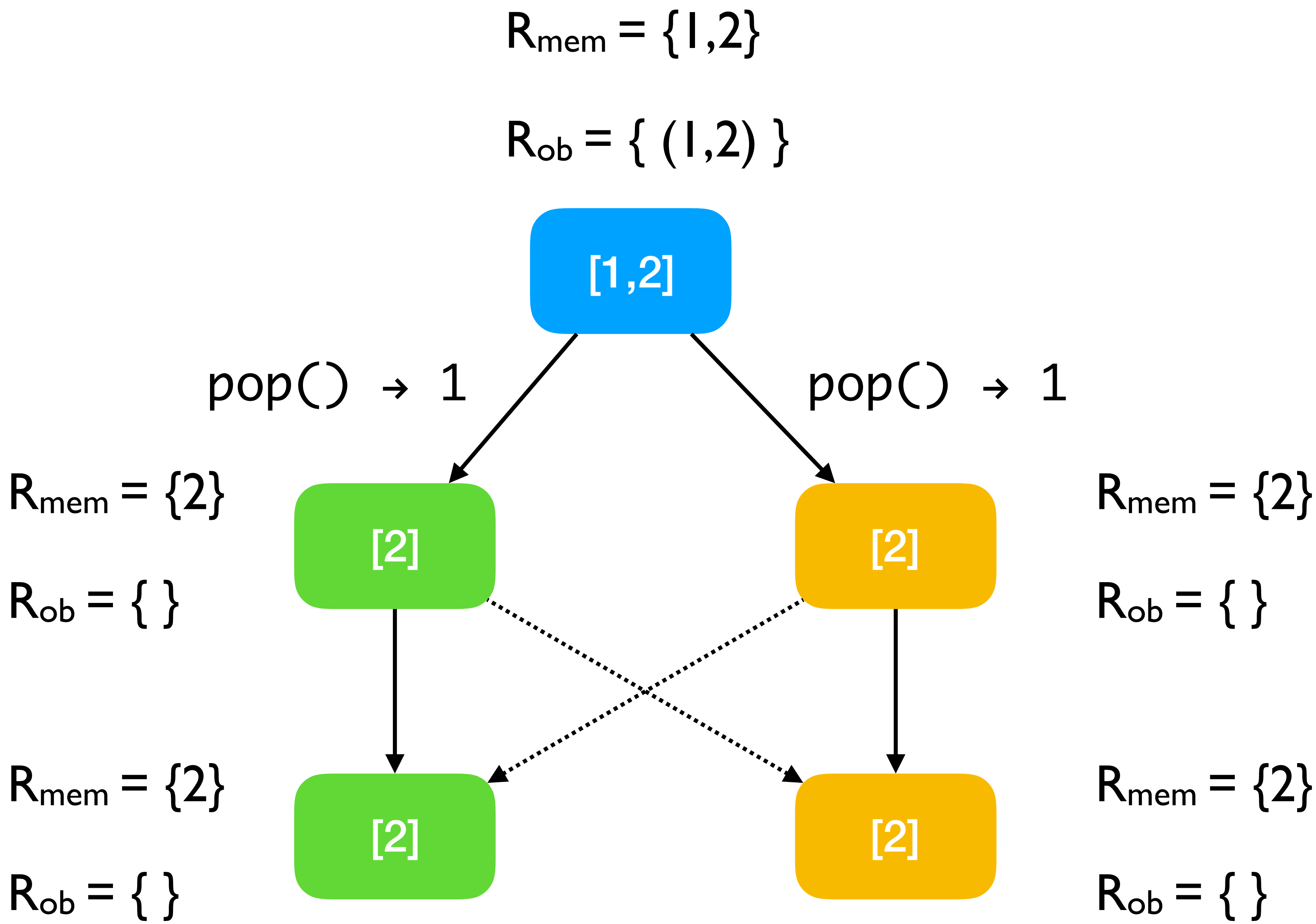
$R_{ob} = \{ (1,2) \}$

[1,2]


$R_{mem} = \{1,2\}$
$R_{ob} = \{ (1,2) \}$
[1,2]
pop() → 1
$R_{mem} = \{2\}$
$R_{ob} = \{ \}$
[2]



R_{mem} = {1,2}
R_{ob} = { (1,2) }
[1,2]
pop() → 1
pop() → 1
R_{mem} = {2}
R_{ob} = { }
[2]
[2]
R_{mem} = {2}
R_{ob} = { }

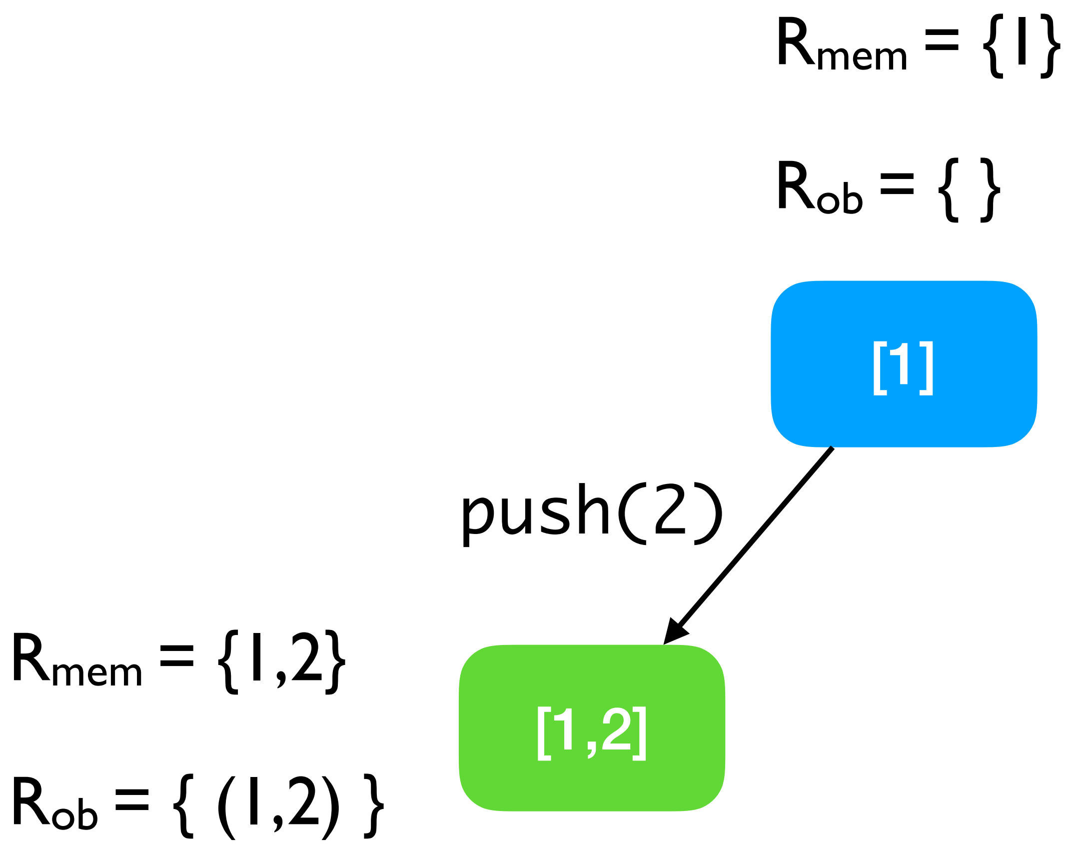


R_{mem} = {1,2}
R_{ob} = { (1,2) }
[1,2]
pop() → 1
pop() → 1
R_{mem} = {2}
R_{ob} = { }
[2]
[2]
R_{mem} = {2}
R_{ob} = { }
R_{mem} = {2}
R_{ob} = { }
[2]
[2]
R_{mem} = {2}
R_{ob} = { }

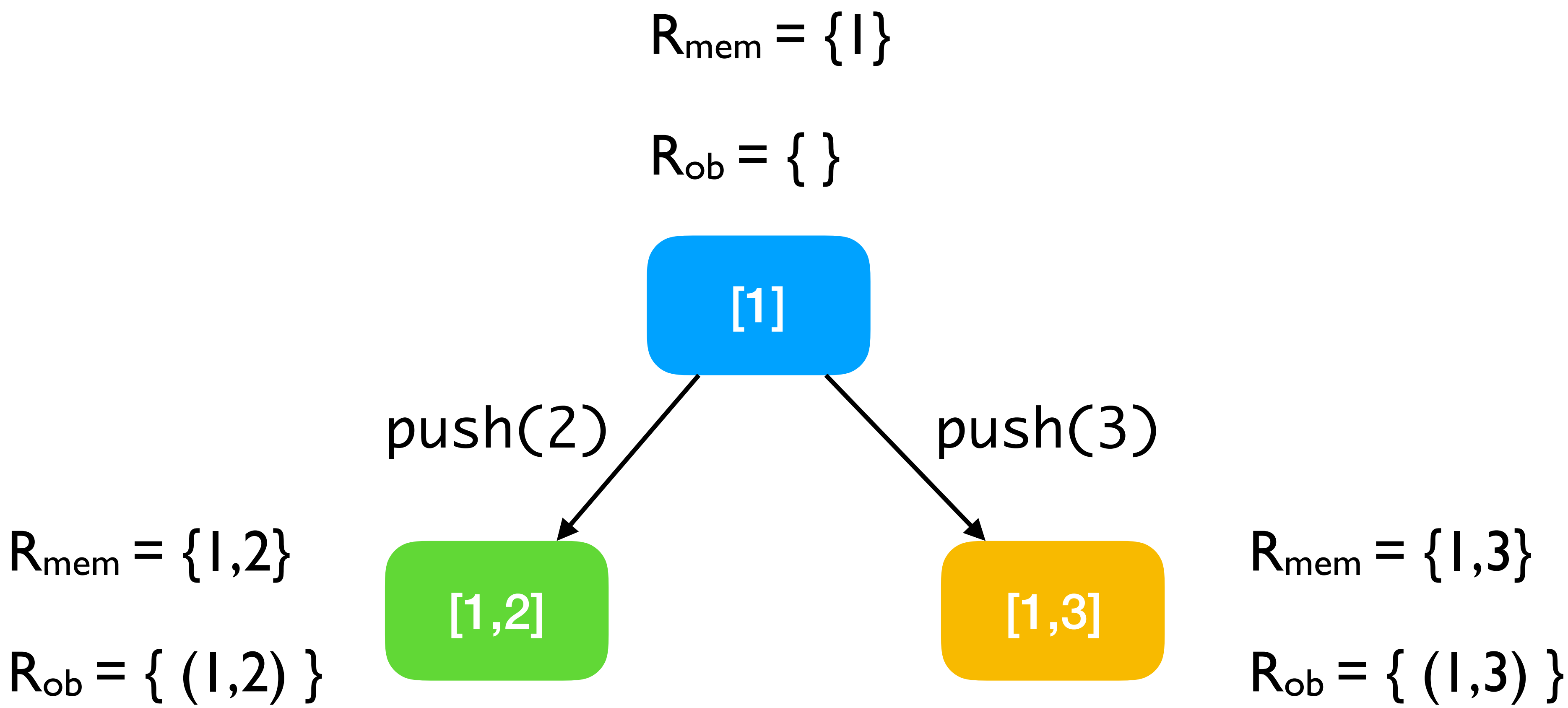
$R_{mem} = \{1\}$

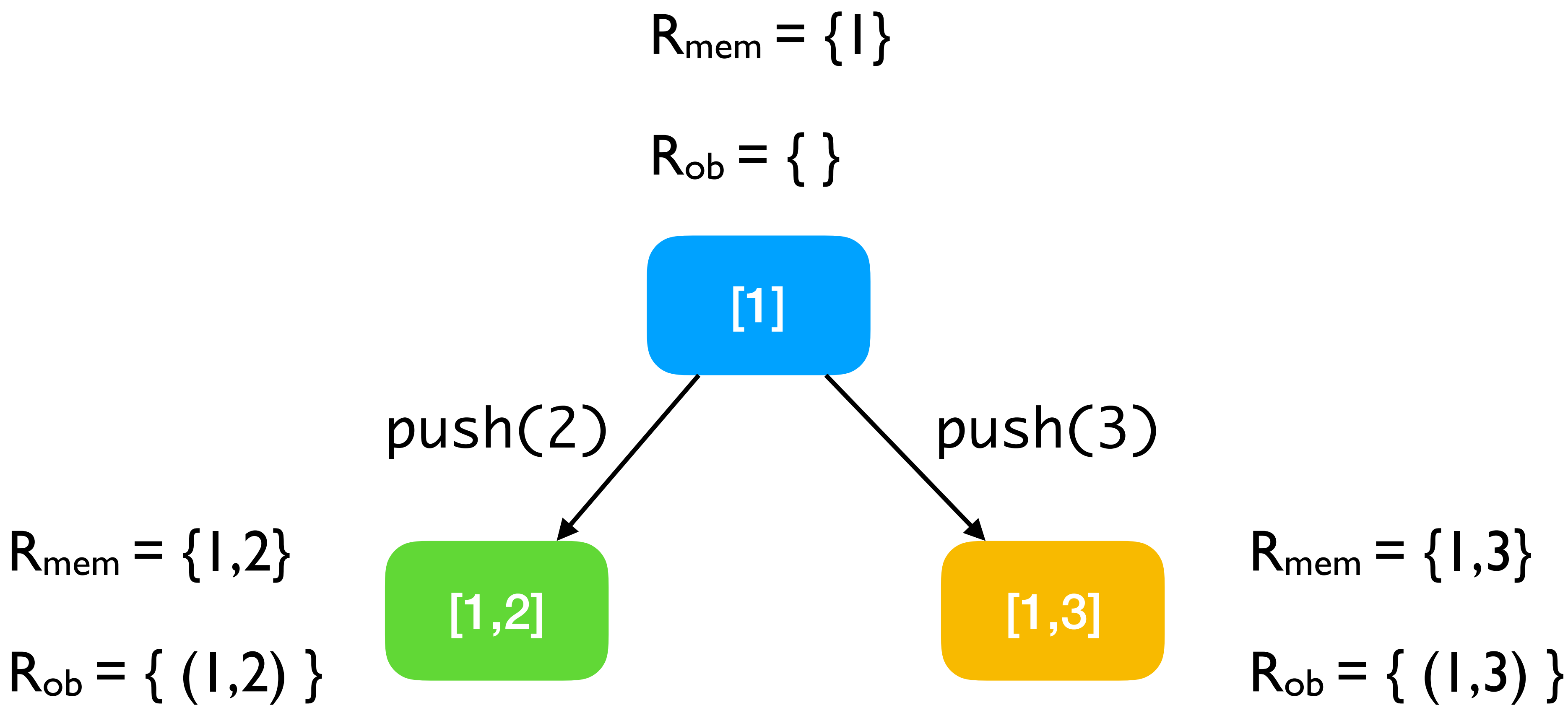
$R_{ob} = \{\ \}$

[1]



R_mem = {1}
R_ob = { }
[1]
push(2)
R_mem = {1,2}
R_ob = { (1,2) }
[1,2]

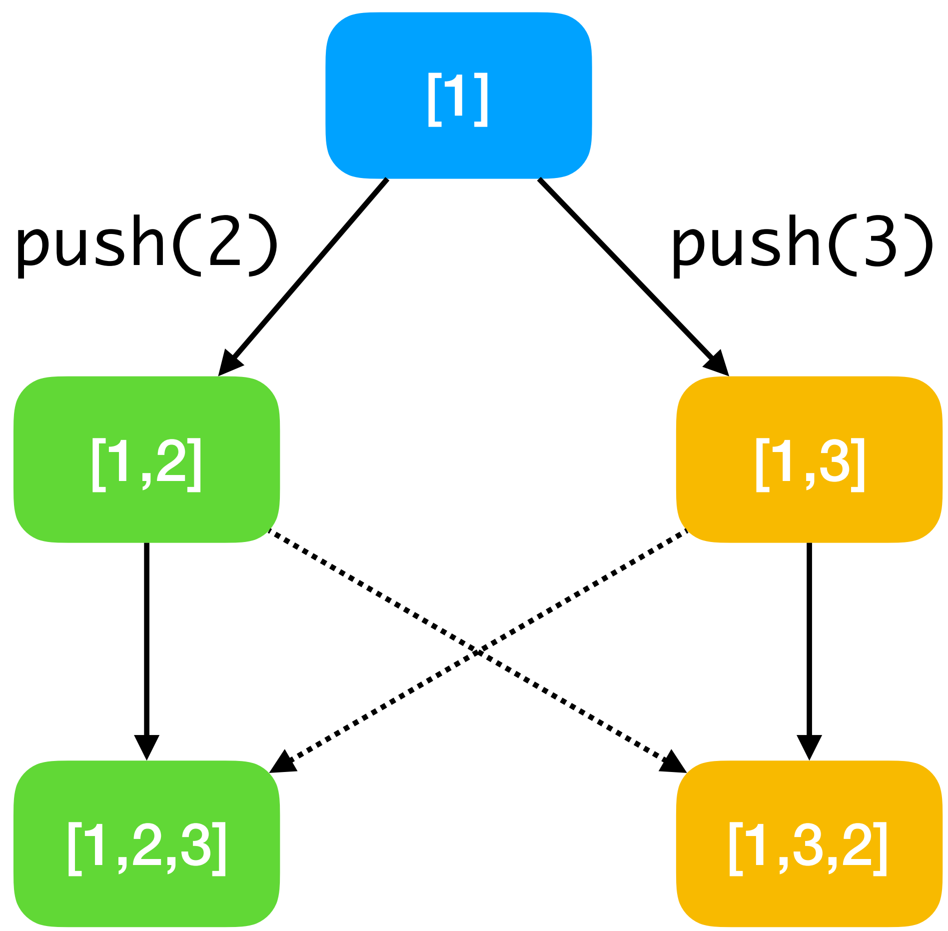

$R_{mem} = \{1\}$
$R_{ob} = \{\ \}$
[1]
push(2)
push(3)
$R_{mem} = \{1,2\}$
$R_{ob} = \{\ (1,2)\ \}$
[1,2]
[1,3]
$R_{mem} = \{1,3\}$
$R_{ob} = \{\ (1,3)\ \}$



Use < as an arbitration function between concurrent insertions

$R_{mem} = \{1\}$

$R_{ob} = \{ \}$



$R_{mem} = \{1,2\}$

$R_{ob} = \{ (1,2) \}$

$R_{mem} = \{1,3\}$

$R_{ob} = \{ (1,3) \}$

$R_{mem} = \{1,2,3\}$

$R_{ob} = \{ (1,2), (1,3), (2,3) \}$

$R_{mem} = \{1,2,3\}$

$R_{ob} = \{ (1,2), (1,3). (2,3) \}$

Use $<$ as an arbitration function between concurrent insertions

Characteristic Relations

A sequence of relations $\overline{R}_T$ is called a characteristic relation of a data type T, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ iff x and y are extensionally equal as interpreted under T.

Characteristic Relations

A sequence of relations $\overline{R}_T$ is called a characteristic relation of a data type T, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ iff x and y are extensionally equal as interpreted under T.

- R_{mem} and R_{ob} are the characteristic relations of queue

Characteristic Relations

A sequence of relations $\overline{R}_T$ is called a characteristic relation of a data type T, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ iff x and y are extensionally equal as interpreted under T.

- R_{mem} and R_{ob} are the characteristic relations of queue
- Appeals only to the sequential properties of the data type
  - ★ Ignore distribution when defining characteristic relations.

Synthesizing Merge

Synthesizing Merge

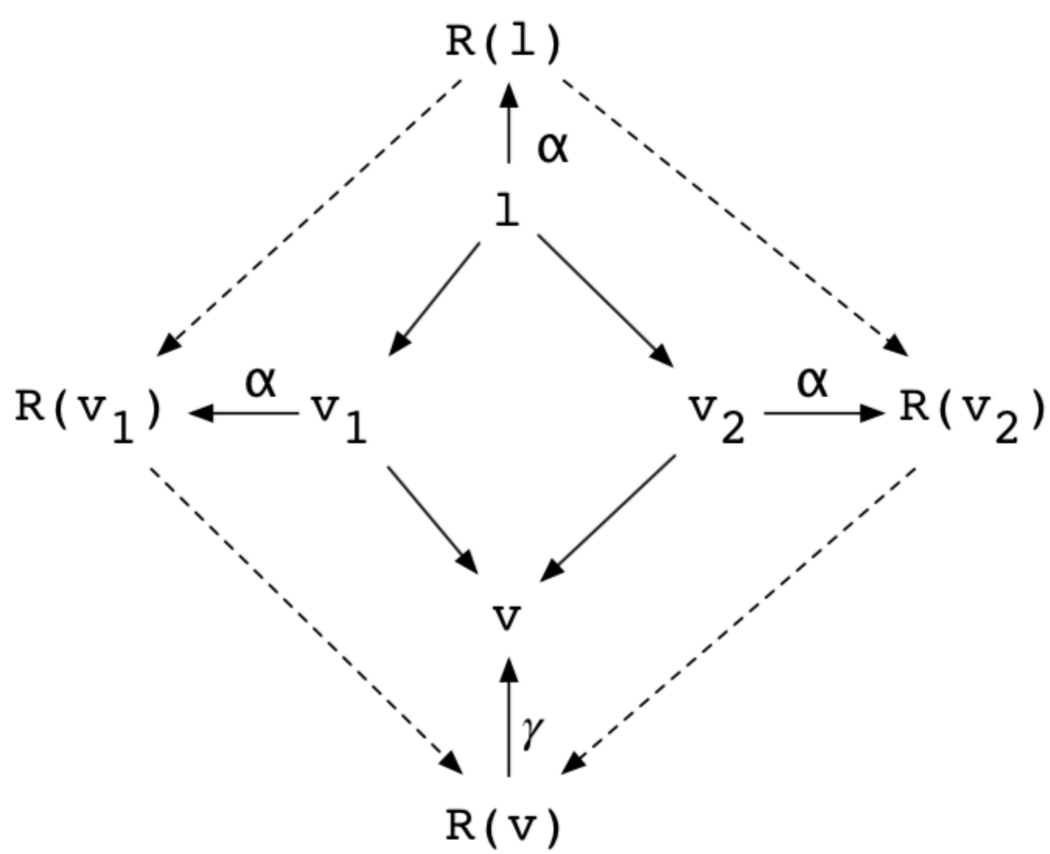
- Semantics of merge in relational domain is quite standard across data types

Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
    - ★ Can we synthesise merge functions for arbitrary data type?

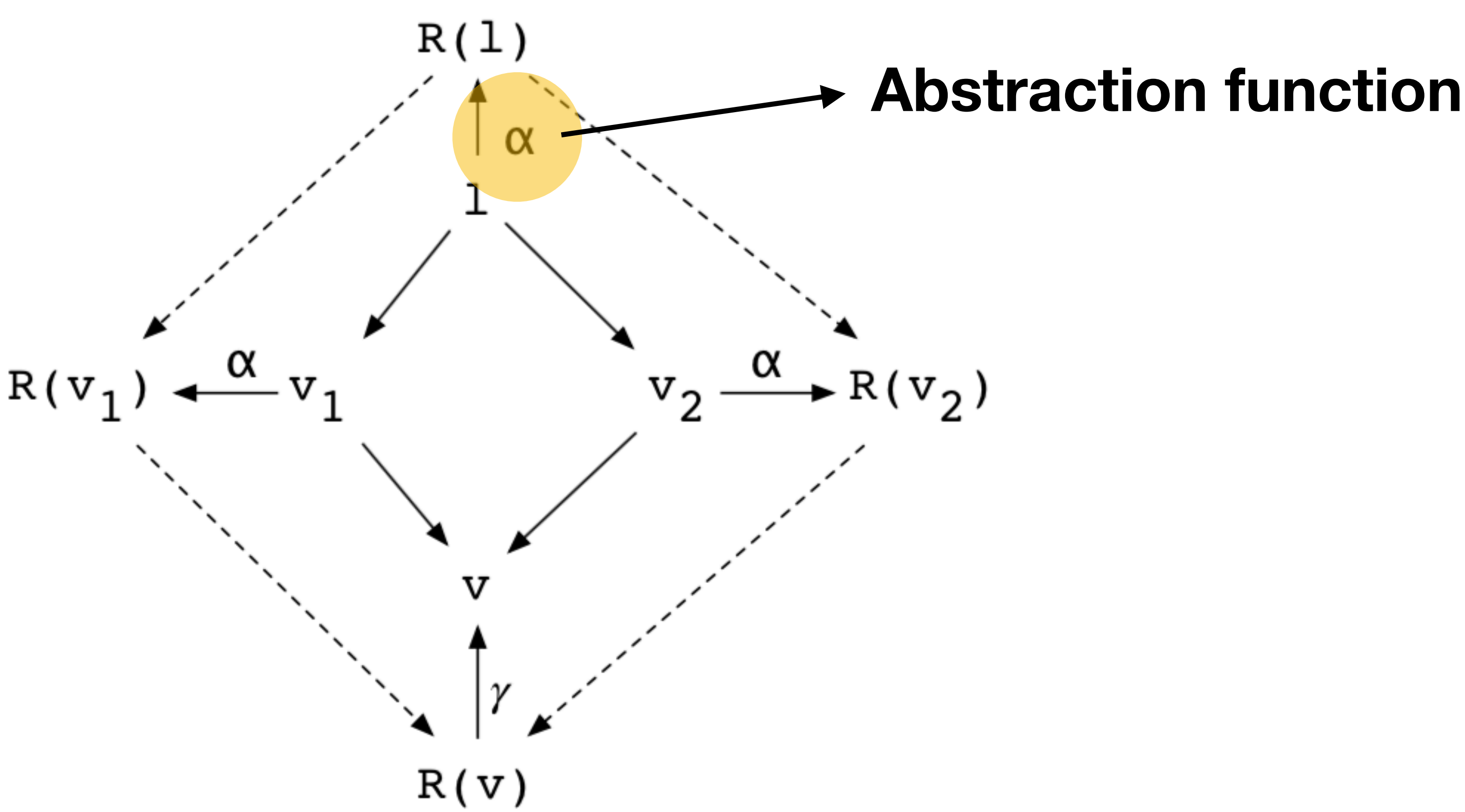
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - ★ Can we synthesise merge functions for arbitrary data type?



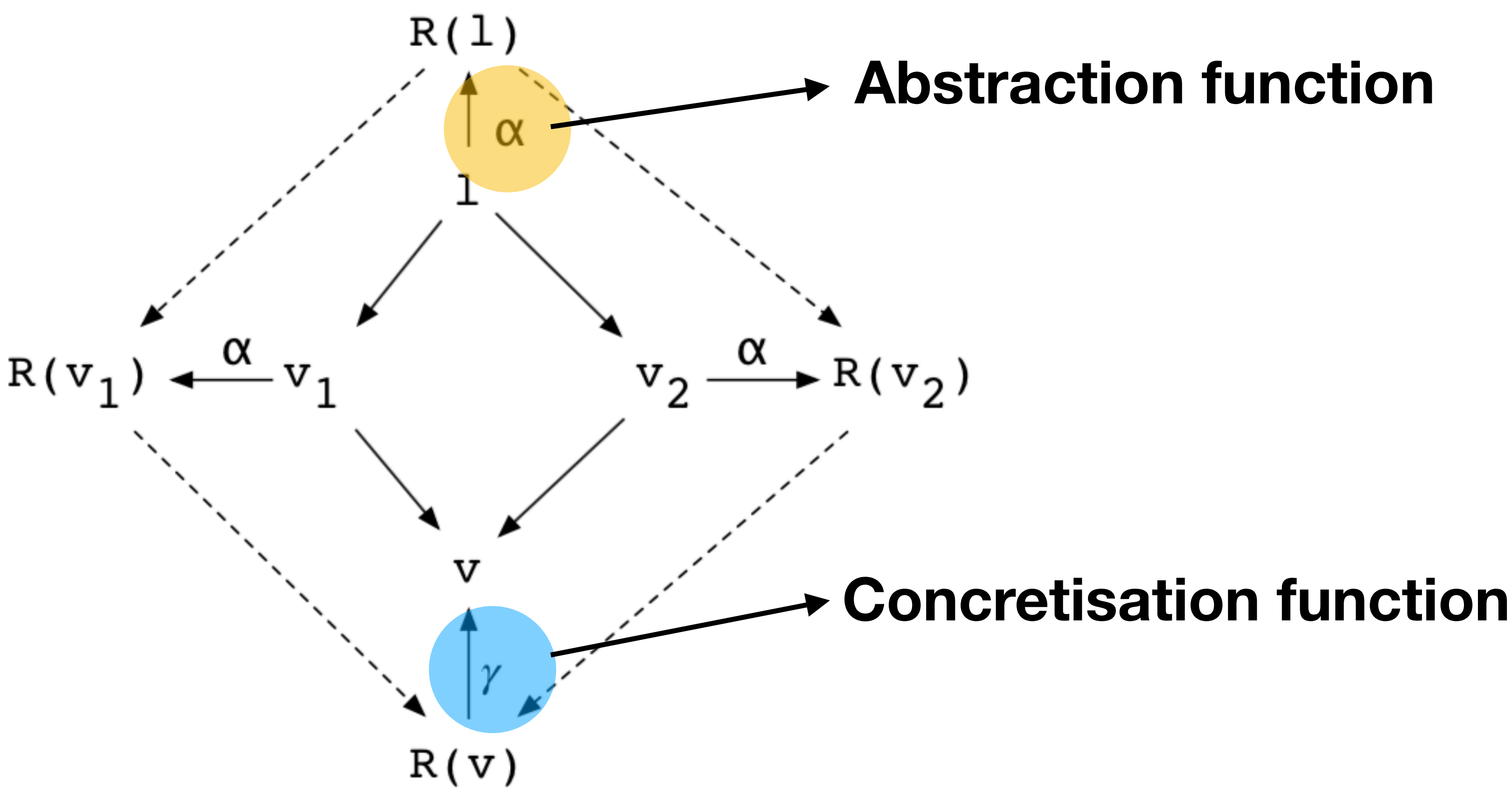
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - ★ Can we synthesise merge functions for arbitrary data type?



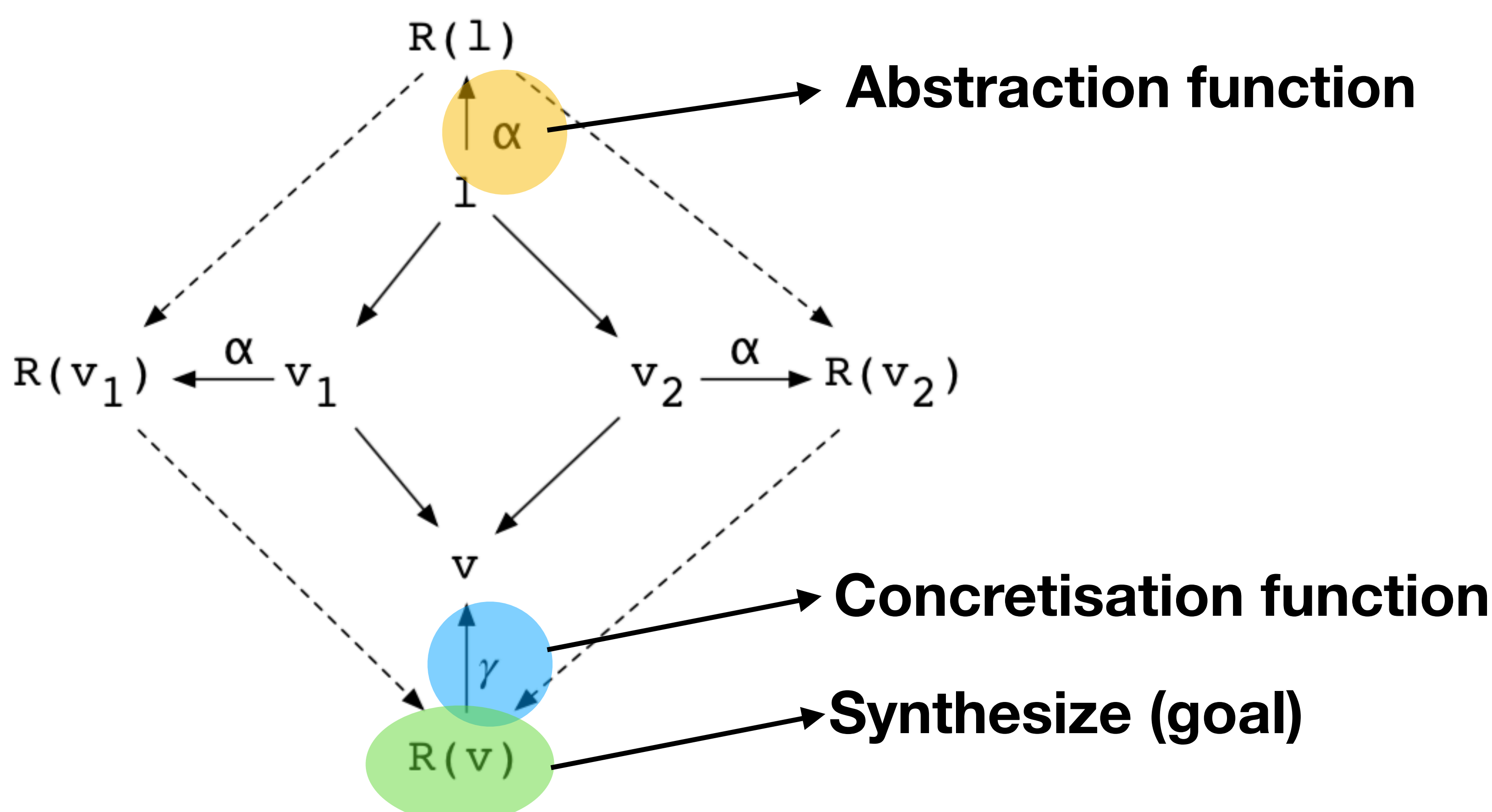
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - ★ Can we synthesise merge functions for arbitrary data type?



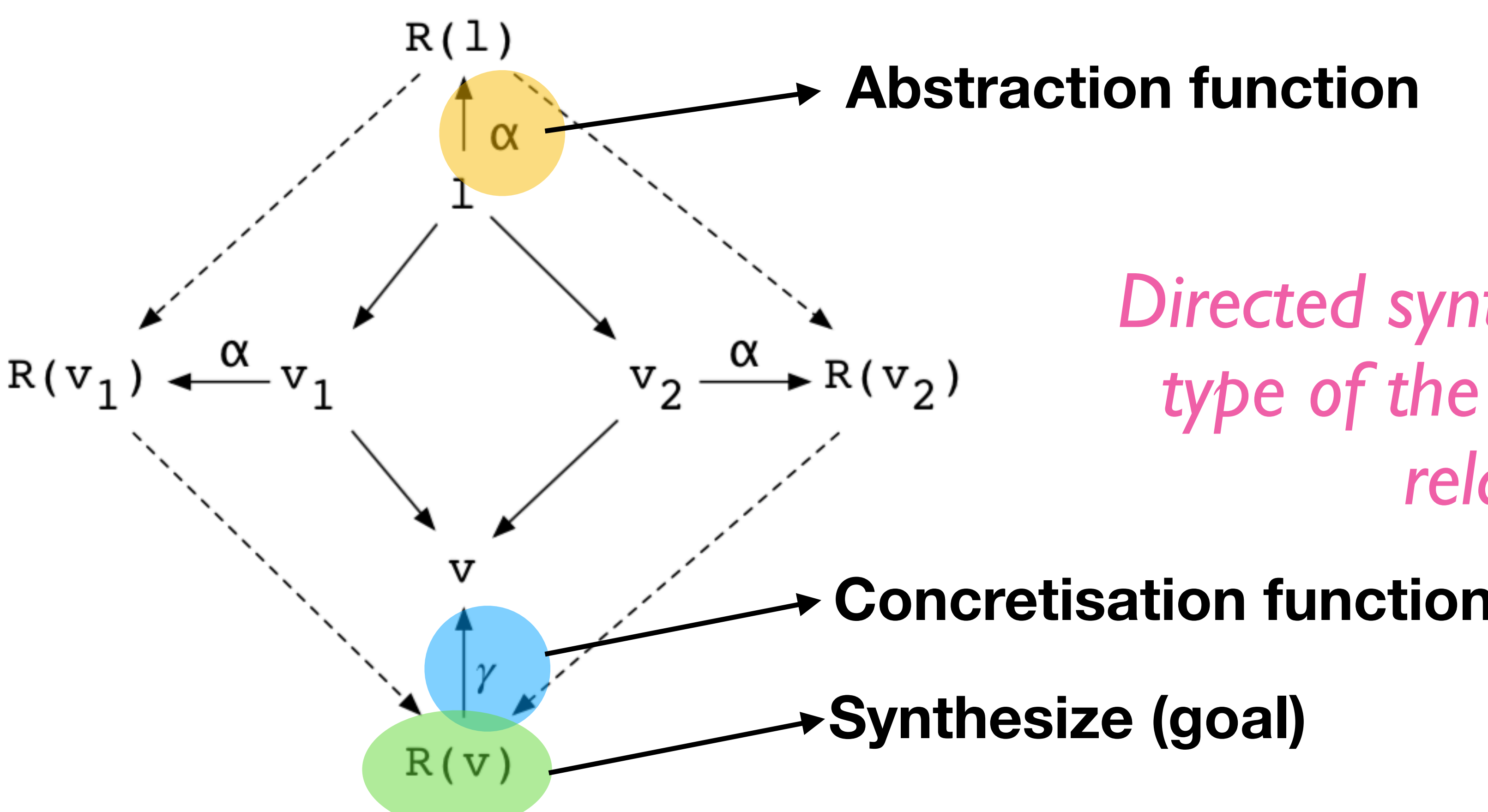
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - ★ Can we synthesise merge functions for arbitrary data type?



Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - ★ Can we synthesise merge functions for arbitrary data type?



Directed synthesis using the type of the characteristic relations

Abstraction Function: Queue

Abstraction Function: Queue

```
let rec R_mem = function
    | [] -> ∅
    | x::xs -> {x} ∪ R_mem(xs)
```

Abstraction Function: Queue

```
let rec R_mem = function
    | [] -> ∅
    | x::xs -> {x} ∪ R_mem(xs)
```

R_{mem} : {ν : int list} $\rightarrow$ $\mathcal{P}$(int)

Abstraction Function: Queue

```
let rec R_mem = function
    | [] -> ∅
    | x::xs -> {x} ∪ R_mem(xs)
```

R_{mem} : {ν : int list} $\to$ $\mathcal{P}$(int)

```
let rec R_ob = function
  | [] -> ∅
  | x::xs -> ({x} × R_mem(xs)) ∪ R_ob(xs)
```

Abstraction Function: Queue

$$
\begin{array}{l}
\texttt{let rec } R_{mem} \texttt{ = function} \\
\quad\quad \texttt{| [] -> } \emptyset \\
\quad\quad \texttt{| x::xs -> \{x\} } \cup R_{mem}\texttt{(xs)}
\end{array}
$$

$$R_{mem} : \{\nu : \texttt{int list}\} \rightarrow \mathcal{P}(\texttt{int})$$

$$
\begin{array}{l}
\texttt{let rec } R_{ob} \texttt{ = function} \\
\quad \texttt{| [] -> } \emptyset \\
\quad \texttt{| x::xs -> (\{x\} } \times R_{mem}\texttt{(xs))} \cup R_{ob}\texttt{(xs)}
\end{array}
$$

$$R_{ob} : \{\nu : \texttt{int list}\} \rightarrow \mathcal{P}(R_{mem}(\nu) \times R_{mem}(\nu))$$

Abstraction Function: Binary Tree

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree
```

Abstraction Function: Binary Tree

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree
```

```
let rec R_mem = function
  | E -> ∅
  | N(l,x,r) -> R_mem(l) ∪ {x} ∪ R_mem(r)
```

$$\texttt{let rec } R_{mem} = \texttt{function}$$
$$\texttt{| E -> } \emptyset$$
$$\texttt{| N(l,x,r) -> } R_{mem}\texttt{(l)} \cup \{\texttt{x}\} \cup R_{mem}\texttt{(r)}$$

Abstraction Function: Binary Tree

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree
```

```
let rec R_mem = function
  | E -> ∅
  | N(l,x,r) -> R_mem(l) ∪ {x} ∪ R_mem(r)
```

```
type label = L | R
let rec R_to = function
  | E -> ∅
  | N(l,x,r) ->
    let l_des = {x} × {L} × R_mem(l) in
    let r_des = {x} × {R} × R_mem(r) in
    R_to(l) ∪ l_des ∪ r_des ∪ R_to(r)
```

Abstraction Function: Binary Heap

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree

let rec R_mem = function
  | E -> ∅
  | N(l,x,r) -> R_mem(l) ∪ {x} ∪ R_mem(r)
```

Abstraction Function: Binary Heap

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree
```

```
let rec R_mem = function
  | E -> ∅
  | N(l,x,r) -> R_mem(l) ∪ {x} ∪ R_mem(r)
```

```
let rec R_ans = function
  | E -> ∅
  | N(l,x,r) ->
    let des_x = R_mem(l) ∪ R_mem(r) in
    let r_ans = {x} × des_x in
    R_ans(l) ∪ r_ans ∪ R_ans(r)
```

Data Type	Characteristic Relations
Binary Heap	Membership (R_{mem}), Ancestor ($R_{ans} \subseteq R_{mem} \times R_{mem}$)
Priority Queue	Membership (R_{mem})
Set	Membership (R_{mem})
Graph	Vertex (R_V), Edge (R_E)
Functional Map	Key-Value (R_{kv})
List	Membership (R_{mem}), Order (R_{ob})
Binary Tree	Membership (R_{mem}), Tree-order ($R_{to} \subseteq R_{mem} \times \texttt{label} \times R_{mem}$)
Binary Search Tree	Membership (R_{mem})

Table 1. Characteristic relations for various data types

Compositionality: Pair

Compositionality: Pair

- The merge of a pair is the merge of the corresponding constituents

Compositionality: Pair

- The merge of a pair is the merge of the corresponding constituents
- A pair data type is defined by the relations:

let R_{fst} = fun (x,_) -> {x}   let R_{snd} = fun (_,y) -> {y}

Compositionality: Pair

- The merge of a pair is the merge of the corresponding constituents
- A pair data type is defined by the relations:

  let R_{fst} = fun (x,_) -> {x}   let R_{snd} = fun (_,y) -> {y}

- Assume that the pair is composed of 2 counters. The counter merge spec is: $\phi_c(l, v_1, v_2, v) \Leftrightarrow v = l + (v_1 - l) + (v_2 - l)$

Compositionality: Pair

- The merge of a pair is the merge of the corresponding constituents
- A pair data type is defined by the relations:

```
let R_fst = fun (x,_) -> {x}      let R_snd = fun (_,y) -> {y}
```

- Assume that the pair is composed of 2 counters. The counter merge spec is: $\phi_c(l, v_1, v_2, v) \Leftrightarrow v = l + (v_1 - l) + (v_2 - l)$
- Then, pair merge spec is:

$$
\begin{aligned}
\phi_{c\times c}(l, v_1, v_2, v) \quad \Leftrightarrow \quad & \forall x, y, z, s.\ x \in R_{fst}(l) \ \wedge\ y \in R_{fst}(v_1) \ \wedge\ z \in R_{fst}(v_2) \\
& \qquad \wedge\ \phi_c(x, y, z, s) \Rightarrow s \in R_{fst}(v) \\
& \wedge\ \forall s.\ s \in R_{fst}(v) \Rightarrow \exists x, y, z.\ x \in R_{fst}(l) \ \wedge\ y \in R_{fst}(v_1) \\
& \qquad \wedge\ z \in R_{fst}(v_2) \ \wedge\ \phi_c(x, y, z, s) \\
& \wedge\ \ldots(\texttt{respectively for } R_{snd})
\end{aligned}
$$

Generalising Pairs to Ordinates

Generalising Pairs to Ordinates

- An alternative characteristic relation for a pair is:

```
let Rpair (x,y) = {(1,x), (2,y)}
```

Generalising Pairs to Ordinates

- An alternative characteristic relation for a pair is:

```
let R_pair (x,y) = {(1,x), (2,y)}
```

$$R_{pair} : \{v : \mathtt{counter} * \mathtt{counter}\} \rightarrow \mathcal{P}\,(\mathtt{int} \times \mathtt{counter})$$

Generalising Pairs to Ordinates

- An alternative characteristic relation for a pair is:

```
let R_pair (x,y) = {(1,x), (2,y)}
```

$$R_{pair} : \{v : \mathtt{counter} * \mathtt{counter}\} \rightarrow \mathcal{P}\,(\mathtt{int} \times \mathtt{counter})$$

- Corresponding merge specification is:

Generalising Pairs to Ordinates

- An alternative characteristic relation for a pair is:

```
let R_pair (x,y) = {(1,x), (2,y)}
```

$$R_{pair} : \{v : \texttt{counter} * \texttt{counter}\} \rightarrow \mathcal{P}\,(\texttt{int} \times \texttt{counter})$$

- Corresponding merge specification is:

$$\begin{aligned}
\phi_{c\times c} \quad = \quad & \forall (k : \texttt{int}).\forall (x, y, z, s : \texttt{counter}).\ (k, x) \in R_{pair}(l) \ \wedge\ (k, y) \in R_{pair}(v_1) \\
& \qquad \wedge\ (k, z) \in R_{pair}(v_2) \ \wedge\ \phi_c(x, y, z, s) \Rightarrow (k, s) \in R(v) \\
\wedge \quad & \forall (k : \texttt{int}).\forall (s : \texttt{counter}).\ (k, s) \in R_{pair}(v) \Rightarrow \exists (x, y, z : \texttt{counter}).\ (k, x) \in R_{pair}(l) \\
& \qquad \wedge\ (k, y) \in R_{pair}(v_1) \ \wedge\ (k, z) \in R_{pair}(v_2) \ \wedge\ \phi_c(x, y, z, s)
\end{aligned}$$

Generalising Pairs to Ordinates

- An alternative characteristic relation for a pair is:

```
let R_pair (x,y) = {(1,x), (2,y)}
```

$$R_{pair} : \{\nu : \texttt{counter} * \texttt{counter}\} \rightarrow \mathcal{P}\,(\texttt{int} \times \texttt{counter})$$

- Corresponding merge specification is:

$$\begin{aligned}\phi_{c\times c} \quad = \quad & \forall (k : \texttt{int}).\forall (x, y, z, s : \texttt{counter}).\ (k, x) \in R_{pair}(l) \ \wedge\ (k, y) \in R_{pair}(v_1) \\ & \qquad \wedge\ (k, z) \in R_{pair}(v_2) \ \wedge\ \phi_c(x, y, z, s) \Rightarrow (k, s) \in R(v) \\ \wedge \quad & \forall (k : \texttt{int}).\forall (s : \texttt{counter}).\ (k, s) \in R_{pair}(v) \Rightarrow \exists (x, y, z : \texttt{counter}).\ (k, x) \in R_{pair}(l) \\ & \qquad \wedge\ (k, y) \in R_{pair}(v_1) \ \wedge\ (k, z) \in R_{pair}(v_2) \ \wedge\ \phi_c(x, y, z, s)\end{aligned}$$

- Given appropriate characteristic relation for a n-tuple ($R_{n\text{-tuple}}$), the same merge specification can be used.

Generalising Pairs to Ordinates

- Similar encoding can be given to maps with non-mergeable types as keys and mergeable types as values

$$
\begin{aligned}
R_k &: \{\nu : (\texttt{string}, \texttt{int})\ \texttt{map}\} \rightarrow \mathcal{P}(\texttt{string}), \\
R_{kv} &: \{\nu : (\texttt{string}, \texttt{int})\ \texttt{map}\} \rightarrow \mathcal{P}(R_k(\nu) \times \mathsf{counter})
\end{aligned}
$$

Types of Characteristic Relations

Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:

Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
  - ★ Membership (R_{mem})

$$R : \{v : T\} \to \mathcal{P}\left(\overline{T}\right), \text{ where } T \text{ is a non-mergeable type}$$

Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
  - ★ Membership (R_{mem})

$$R : \{v : T\} \to \mathcal{P}\left(\overline{T}\right), \text{ where } T \text{ is a non-mergeable type}$$

  - ★ Ordering (R_{ob}, R_{ans}, R_{to})

$$R : \{v : T\} \to \mathcal{P}\left(\rho\right)$$

    - ✦ where ρ is a sequence of non-mergeable types and other relations, which flattens to a sequence of non-mergeable types

Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
  - ★ Membership (R_{mem})

    $$R : \{v : T\} \to \mathcal{P}\left(\overline{T}\right), \text{ where } T \text{ is a non-mergeable type}$$

  - ★ Ordering (R_{ob}, R_{ans}, R_{to})

    $$R : \{v : T\} \to \mathcal{P}\left(\rho\right)$$

    - ✦ where ρ is a sequence of non-mergeable types and other relations, which flattens to a sequence of non-mergeable types
  - ★ Ordinates (R_{pair}, R_{kv})

    $$R : \{v : T\} \to \mathcal{P}\left(\overline{T} \times \overline{\tau}\right), \text{ where } \tau \text{ is a mergeable type}$$

Deriving merge spec

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\overline{T}\right)}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)) \Leftrightarrow \overline{x} \in R(v)} \quad \text{[Set-Merge]}$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}\,(\rho) \qquad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)\ \cap\ \rho) \Rightarrow \overline{x} \in R(v)} \quad \text{[Order-Merge-1]}$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}\,(\rho) \qquad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in R(v) \Rightarrow \overline{x} \in \rho} \quad \text{[Order-Merge-2]}$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}\,(\rho) \qquad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \qquad \overline{\tau} \neq \emptyset}{\begin{array}{l}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{x}, \overline{y}, \overline{z}, \overline{s} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l)\ \wedge\ (\overline{k}, \overline{y}) \in R_+(v_1)\ \wedge\ (\overline{k}, \overline{z}) \in R_+(v_2) \\ \qquad \wedge\ \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2))\ \wedge\ \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\ \wedge\ (\overline{k}, \overline{s}) \in \rho \Rightarrow (\overline{k}, \overline{s}) \in R(v)\end{array}} \quad \text{[Rel-Merge-1]}$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}\,(\rho) \qquad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \qquad \overline{\tau} \neq \emptyset}{\begin{array}{l}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{s} : \overline{\tau}).\ (\overline{k}, \overline{s}) \in R(v) \Rightarrow (\overline{k}, \overline{s}) \in \rho \\ \qquad \wedge\ \exists(\overline{x}, \overline{y}, \overline{z} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l)\ \wedge\ (\overline{k}, \overline{y}) \in R_+(v_1)\ \wedge\ (\overline{k}, \overline{z}) \in R_+(v_2) \\ \qquad\qquad \wedge\ \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2))\ \wedge\ \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\end{array}} \quad \text{[Rel-Merge-2]}$$

Deriving merge spec

$$\frac{R : \{\nu : T\} \to \mathcal{P}\left(\overline{T}\right)}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)) \Leftrightarrow \overline{x} \in R(v)} \quad [\textsc{Set-Merge}]$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}(\rho) \quad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)\ \cap\ \rho) \Rightarrow \overline{x} \in R(v)} \quad [\textsc{Order-Merge-1}]$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}(\rho) \quad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in R(v) \Rightarrow \overline{x} \in \rho} \quad [\textsc{Order-Merge-2}]$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}(\rho) \quad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \quad \overline{\tau} \neq \emptyset}{\begin{array}{l}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{x}, \overline{y}, \overline{z}, \overline{s} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l)\ \wedge\ (\overline{k}, \overline{y}) \in R_+(v_1)\ \wedge\ (\overline{k}, \overline{z}) \in R_+(v_2) \\ \quad \wedge\ \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2))\ \wedge\ \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\ \wedge\ (\overline{k}, \overline{s}) \in \rho \Rightarrow (\overline{k}, \overline{s}) \in R(v)\end{array}} \quad [\textsc{Rel-Merge-1}]$$

$$\frac{R : \{\nu : T\} \to \mathcal{P}(\rho) \quad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \quad \overline{\tau} \neq \emptyset}{\begin{array}{l}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{s} : \overline{\tau}).\ (\overline{k}, \overline{s}) \in R(v) \Rightarrow (\overline{k}, \overline{s}) \in \rho \\ \quad \wedge\ \exists(\overline{x}, \overline{y}, \overline{z} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l)\ \wedge\ (\overline{k}, \overline{y}) \in R_+(v_1)\ \wedge\ (\overline{k}, \overline{z}) \in R_+(v_2) \\ \quad\quad \wedge\ \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2))\ \wedge\ \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\end{array}} \quad [\textsc{Rel-Merge-2}]$$

- Not complete, but practical
- Can derive merge spec for
  - ★ Data structures: Set, Heap, Graph, Queue, TreeDoc
  - ★ Larger apps: TPC-C, TPC-E, Twissandra, Rubis

Distributed Implementation

Distributed Implementation

- For making this programming model practical, we need to:
  - ★ Quickly compute LCA
  - ★ Optimise storage through sharing
  - ★ Optimise network transmissions (state based merge)

Distributed Implementation

- For making this programming model practical, we need to:
  - ★ Quickly compute LCA
  - ★ Optimise storage through sharing
  - ★ Optimise network transmissions (state based merge)
- Irmin
  - ★ A reimplementation of Git in pure OCaml
  - ★ Arbitrary OCaml objects, not just files + User-defined 3-way merges
  - ★ Only transmit diffs over the network
  - ★ Multiple storage backends including in-memory, filesystems, log-structured-merge database, distributed databases

Performance

Performance

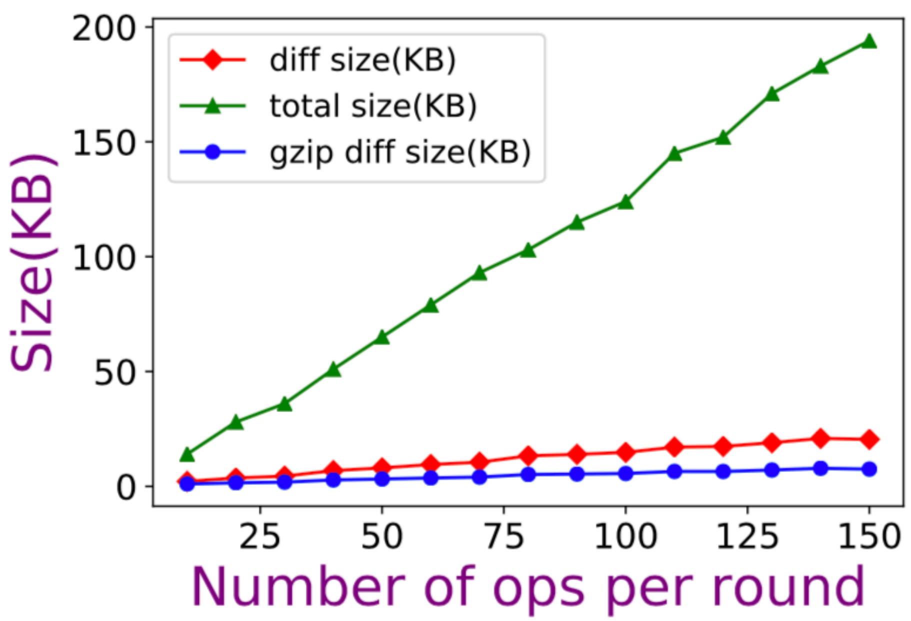
- What is the size of diff compared to the size of data structure?

Performance

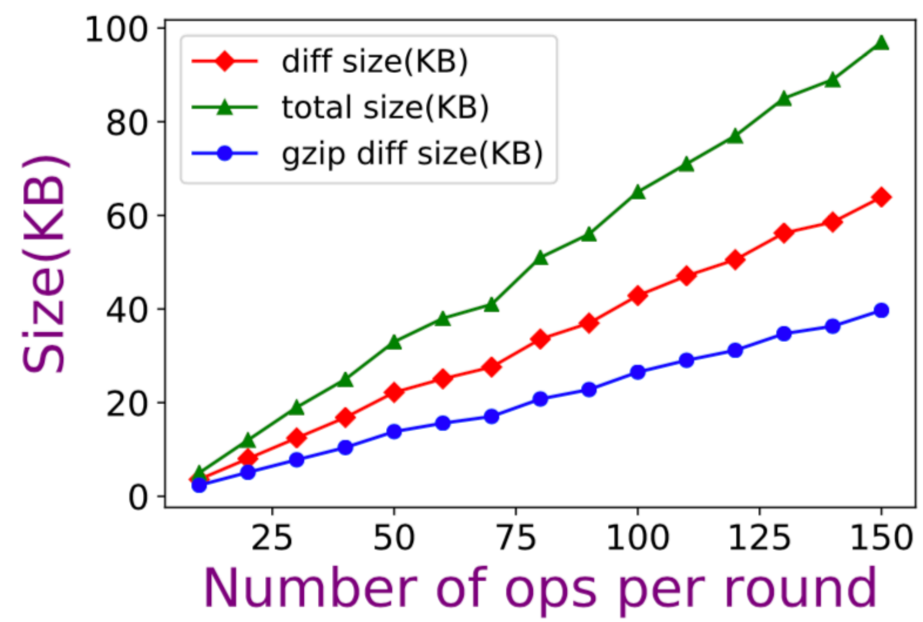
- What is the size of diff compared to the size of data structure?
- Setup
    - ★ 2 Replicas, fixed number of rounds, each round has N operations
    - ★ 75% inserts, 25% deletions
    - ★ Synchronise after each round

Performance

- What is the size of diff compared to the size of data structure?
- Setup
    - ★ 2 Replicas, fixed number of rounds, each round has N operations
    - ★ 75% inserts, 25% deletions
    - ★ Synchronise after each round



Binary Heap



Growable Array

Thanks for listening!