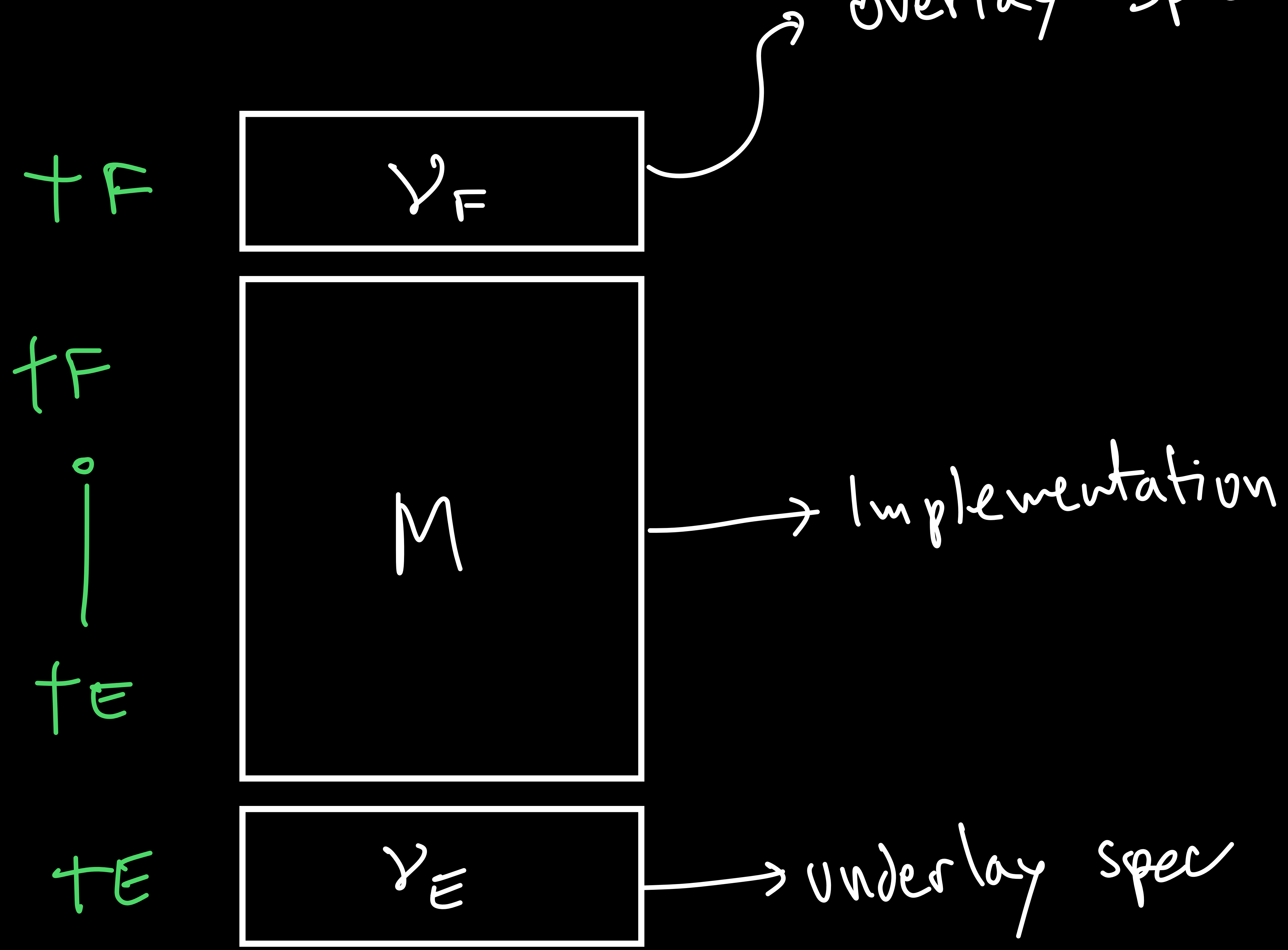
Linearizability and Compositional CCAL

Arthur Oliveira Vale - Feb 22nd 2024

Object-Based CAL Review

A certified layer is:



A spec (strategy) is a set of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

alternating + type-dependent

Implementations are regular:

$$M : tE \multimap F$$

$$\rightsquigarrow$$

$$\hat{M} : tE \multimap tF$$

Certified when refinement condition holds:

$$\gamma_E ; \hat{M} \sqsupseteq \gamma_F$$

What should a concurrent implementation look like?

Simple idea: several "threads" running sequential code in parallel

```
Import x : Var[Nat]

Nat get() {
    v ← x.rd();
    ret v
}

unit inc() {
    v ← x.rd();
    _ ← x.wrt(v+1);
    ret ok
}
```

∥ . . . ∥

```
Import x : Var[Nat]

Nat get() {
    v ← x.rd();
    ret v
}

unit inc() {
    v ← x.rd();
    _ ← x.wrt(v+1);
    ret ok
}
```

What should a concurrent implementation look like?

Simple idea: several "threads" running sequential code in parallel

$\Upsilon \rightsquigarrow$ set of agent names

$\alpha \in \Upsilon \rightsquigarrow$ agent α

$M[\alpha] : \mathcal{T}E \to \mathcal{T}F \rightsquigarrow$ implementation that α runs

$M[\Upsilon] = (M[\alpha])_{\alpha \in \Upsilon} \rightsquigarrow$ concurrent implementation

Vertical composition:

$$(N[\alpha])_{\alpha \in \Upsilon} \circ (M[\alpha])_{\alpha \in \Upsilon} \triangleq (N[\alpha] \circ M[\alpha])_{\alpha \in \Upsilon}$$
$$= (\hat{M}[\alpha] ; N[\alpha])_{\alpha \in \Upsilon}$$

What is a concurrent spec?

A spec (strategy) is a set of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

alternating
+
type-dependent

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

alternating + type-dependent

What does it mean to be "well-formed"

Simple idea: interleaving of well-formed sequential traces of each thread

T_{tE} = "well-formed sequential tE traces"

$$\gamma : T_{tE} = \|_{\alpha \in \Gamma} \alpha : T_{tE}$$

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

alternating + type-dependent

What does it mean to be "well-formed"?

Simple idea: interleaving of well-formed sequential traces of each thread

T_{tE} = "well-formed sequential tE traces"

$$r : T_{tE} = \|_{\alpha \in r}\ \alpha : T_{tE}$$

Example: $r = \{0, 1\}$ $E = \text{Counter}$

$$\begin{array}{l} 1.\text{inc} \\ 1:\text{ok} \\ 1:\text{set} \end{array} \ \Big\|\ \begin{array}{l} 2:\text{get} \\ 2:0 \end{array} =$$

1: inc 1: ok 1: set 2: get 2: 0
1: inc 1: ok 2: set 1: get 2: 0
1: inc 2: set 1: ok 1. get 2: 0
2: set 1: inc 1: ok 1. get 2: 0
2: set 1: inc 1: ok 2: 0 1. get
⋮
2: set 2: 0 1: inc 1: ok 1. get

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

~~alternating~~ + type-dependent

What does it mean to be "well-formed"?

Simple idea: interleaving of well-formed sequential traces of each thread

T_{tE} = "well-formed sequential tE traces"

$$r : T_{tE} = \|_{\alpha \in r} \, \alpha : T_{tE}$$

Example: $r = \{1, 2\}$ E = Counter

$$\begin{array}{l} 1.\ inc \\ 1:ok \\ 1:set \end{array} \;\Big\|\; \begin{array}{l} 2:get \\ 2:0 \end{array} \;=\; \begin{array}{l} 1:inc\ 1:ok\ 1:set\ 2:get\ 2:0 \\ 1:inc\ 1:ok\ 2:set\ 1:get\ 2:0 \\ 1:inc\ 2:set\ 1{,}ok\ 1.get\ 2:0 \\ 2:set\ 1:inc\ 1{,}ok\ 1.get\ 2:0 \\ 2:set\ 1:inc\ 1{,}ok\ 2:0\ 1.get \\ \vdots \\ 2:set\ 2:0\ 1:inc\ 1{,}ok\ 1.get \end{array}$$

NOT Alternating

What is a concurrent spec?

- sequential counter

  get

A spec (strategy) is a set o of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

type-dependent + interleavings of well-formed sequential traces

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

type-dependent + interleavings of well-formed sequential traces

- sequential counter

get → 0

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

type-dependent + interleavings of well-formed sequential traces

- sequential counter

  get $\to$ 0

- Concurrent counter

  1:get

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- deterministic

type-dependent + interleavings of well-formed sequential traces

- sequential counter

get → 0

- Concurrent counter

1:get 1:0

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive
- ~~deterministic~~

type-dependent + interleavings of well-formed sequential traces

- sequential counter

get → 0

- Concurrent counter

1:get 1:0

1:get 2:inc 2:ok 1:1

1:get 2:inc 2:ok 2:inc 2:ok 1:2

1:get 2:inc

⋮

1:get (2:inc 2:ok)* 1:n

NOT Deterministic

What is a concurrent spec?

A spec (strategy) is a set o of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive

type-dependent + interleavings of well-formed sequential traces

Receptive?

What if I want to describe an atomic counter?

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- receptive

type-dependent + interleavings of well-formed sequential traces

Receptive?

What if I want to describe an atomic counter?

1: inc

1: inc 1: ok

1: inc 1: ok 2: get

1: inc 1: ok 2: get 2: 1

⋮

"alternating but threaded"

What is a concurrent spec?

A spec (strategy) is a set σ of "well-formed" traces s.t.

- non-empty
- prefix-closed
- ~~receptive~~

type-dependent + interleavings of well-formed sequential traces

Receptive?

What if I want to describe an atomic counter?

1: inc 2: set

1: inc 1: ok 2: set 1: inc

not atomic but required by receptivity

NOT Receptive

What is a concurrent spec?

A spec (strategy) is a set of "well-formed" traces s.t.

- non-empty
- prefix-closed

type-dependent + interleavings of well-formed sequential traces

Lock Concurrent Spec

$\text{Lock} \triangleq \{ \text{acq} : \text{unit} , \ \text{rel} : \text{unit} \}$

States: $\mathcal{P}(\Upsilon) \times \text{Maybe } \Upsilon \times \mathcal{P}(\alpha)$

$(\text{Waiting}, \text{owner?}, \text{releasing}) \xrightarrow{\alpha : \text{acq}} (\text{Waiting} \uplus \{\alpha\}, \text{owner?}, \text{releasing})$

$(\text{Waiting}, \text{None}, \text{releasing}) \xrightarrow{\alpha : \text{ok}^{\text{acq}}} (\text{Waiting}, \text{Some } \alpha, \text{releasing})$

$(\text{Waiting}, \text{Some } \alpha, \text{releasing}) \xrightarrow{\alpha . \text{rel}} (\text{Waiting}, \text{None}, \text{releasing} \uplus \{\alpha\})$

$(\text{Waiting}, \text{owner?}, \text{releasing} \uplus \{\alpha\}) \xrightarrow{\alpha : \text{ok}^{\text{rel}}} (\text{Waiting}, \text{owner?}, \text{releasing})$

Conditional Spec : Assumes agents alternate acquire calls with release calls

Executions of Concurrent Implementations

Given a concurrent implementation

$$M[\Gamma] = \left(M[\alpha] \right)_{\alpha \in \Gamma}$$

we define its set of executions as

$$\widehat{M[\Gamma]} \triangleq \|_{\alpha \in \Gamma} \widehat{M[\alpha]}$$

The identity

Identity for (sequential) Implementation composition:

$\mathrm{Id} : \mathcal{F} \multimap \mathcal{F}$

```
f(a_1, ..., a_n) {
    v ← f(a_1, ..., a_n);
    ret v
}
```

The identity

Identity for (sequential) Implementation composition:

$\mathrm{Id} : \uparrow F \multimap F$

```
f(a_1, ..., a_n) {
    v ← f(a_1, ..., a_n);
    ret v
}
```

Example:

```
inc() {
    v ← inc();
    ret v
}
```

```
get() {
    v ← get();
    ret v
}
```

$\mathrm{Id} : \uparrow \mathrm{Counter} \multimap \mathrm{Counter}$

inc $\qquad$ ok

↳ inc → ok

get $\qquad$ v

↳ get → v

$\widehat{\mathrm{Id}} : \uparrow \mathrm{Counter} \multimap \uparrow \mathrm{Counter}$

get $\qquad$ 3 → inc $\qquad$ ok → ...

↳ get → 3 $\qquad$ ↳ inc → ok

The identity

Identity for (sequential) Implementation composition:

$Id : \dagger F \multimap F$

```
f(a_1,...,a_n) {
    v <- f(a_1,...,a_n);
    ret v
}
```

Example:

```
inc() {              inc() {
  v <- inc();          v <- inc();
  ret v                ret v
}          || ... || }

get() {              get() {
  v <- get();          v <- get();
  ret v                ret v
}                    }
```

Concurrent Identity

$$Id[r] \triangleq (Id)_{t \in r}$$

$\widehat{Id[r]} : \dagger counter \multimap \dagger counter \qquad r = \{1, 2\}$

1:inc 2:get 2:get 2:v 1:inc 2:v 1:ok 2:inc

$\dagger counter$

$\dagger counter$

Spec + Implementation Composition

$\gamma_E : tE$

$\widehat{M[r]} : tE \multimap tF$

$\Big\} \quad \gamma_E ; \widehat{M[r]} : tF$

$$\gamma_E ; \widehat{M[r]} = \{ t\restriction_F \mid t \in \widehat{M[r]} \wedge t\restriction_E \in \gamma_E \}$$

↳ projection of t to only F events

Example:

$\widehat{Id[r]} : t\text{Counter} \multimap t\text{Counter} \qquad r = \{1,2\}$

2:inc

$t\text{Counter}$

$t\text{Counter}$

1:inc 2: get

2:0

$\underbrace{\text{2 : get} \quad \text{2:0} \quad \text{1: inc} \quad \text{1: ok}}_{\gamma_{\text{atomic Counter}}}$

Spec + Implementation Composition

$\gamma_E : tE$

$\widehat{M[r]} : tE \multimap tF$ $\Bigg\}$ $\gamma_E ; \widehat{M[r]} : tF$

Sidebar:

$(N[\alpha])_{\alpha \in r} \circ (M[\alpha])_{\alpha \in r}$

$=$

$\widehat{M[r]} ; \widehat{N[r]}$

$$\gamma_E ; \widehat{M[r]} = \{ t\restriction_F \mid t \in \widehat{M[r]} \wedge t\restriction_E \in \gamma_E \}$$

projection of t to only F events

Example:

$\widehat{\mathrm{Id}[r]} : t\mathrm{Counter} \multimap t\mathrm{Counter}$ $r = \{1, 2\}$

1:inc 2: get 2:0 2: inc

$t\mathrm{counter}$

$t\mathrm{Counter}$

$\underbrace{\text{2 : get} \quad \text{2:0} \quad \text{1: inc} \quad \text{1: ok}}_{\gamma^{\text{atomic}}_{\text{Counter}}}$

A Problem: Identity?

2 : get 2 : 0 1 : inc 1 : ok $\in \gamma^{\text{atomic}}_{\text{counter}}$

$\wedge$

2 : 0 2 : inc

1 : inc 2 : get

2 : get 2 : 0 1 : inc 1 : ok $\in \mathrm{Id}[r]$

$\Downarrow$

1 : inc 2 : get 2 : 0 2 : inc $\in r^{\text{atomic}}_{\text{counter}} ; \mathrm{Id}[r]$

but

1 : inc 2 : get 2 : 0 2 : inc $\notin r^{\text{atomic}}_{\text{counter}}$

A Problem: Identity?

Id[r] is NOT the identity

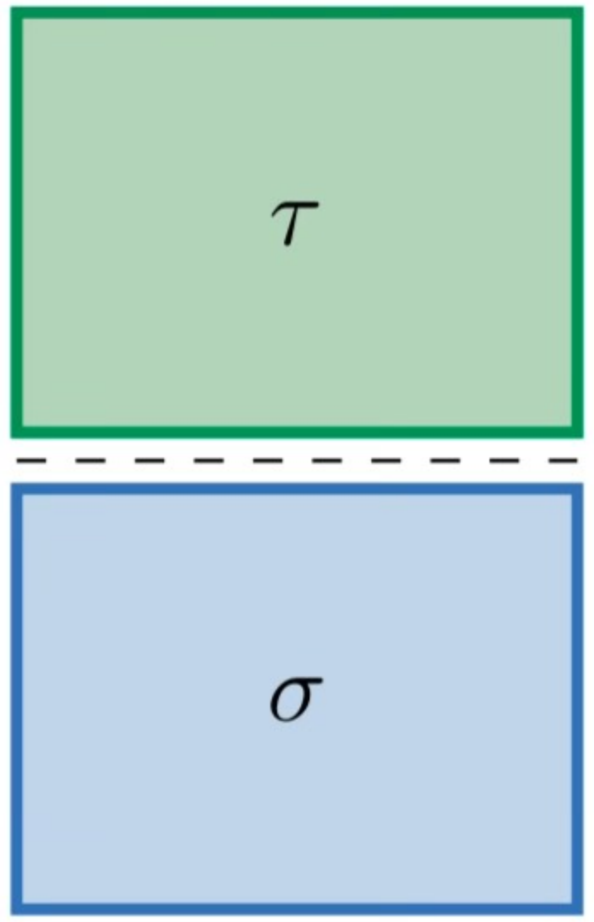
Algebra of Composition

There is a composition operation denoted by

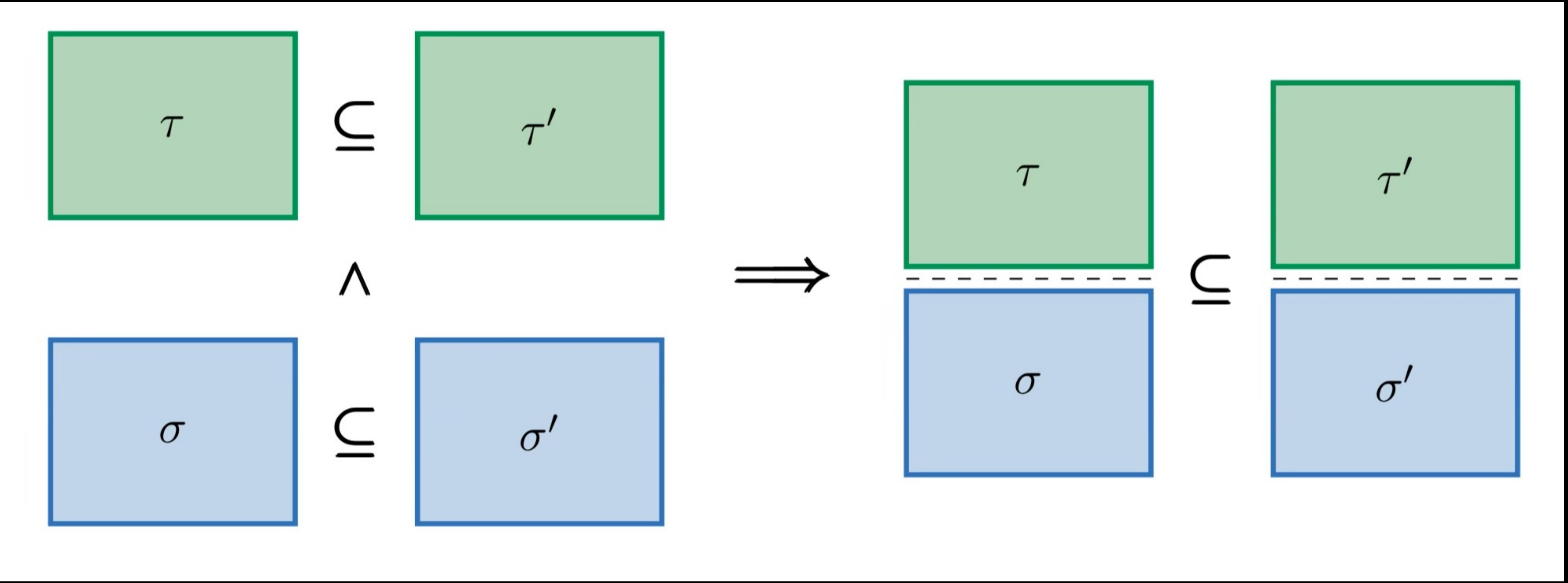
$$\sigma : \boldsymbol{A} \multimap \boldsymbol{B} \qquad \tau : \boldsymbol{B} \multimap \boldsymbol{C} \longmapsto \sigma ; \tau : \boldsymbol{A} \multimap \boldsymbol{C}$$

Which is **associative** ... but there is **no identity element**!

$$\forall \sigma : \boldsymbol{A} \multimap \boldsymbol{B} . \mathbf{id}_{\boldsymbol{A}} ; \sigma ; \mathbf{id}_{\boldsymbol{B}} = \sigma$$

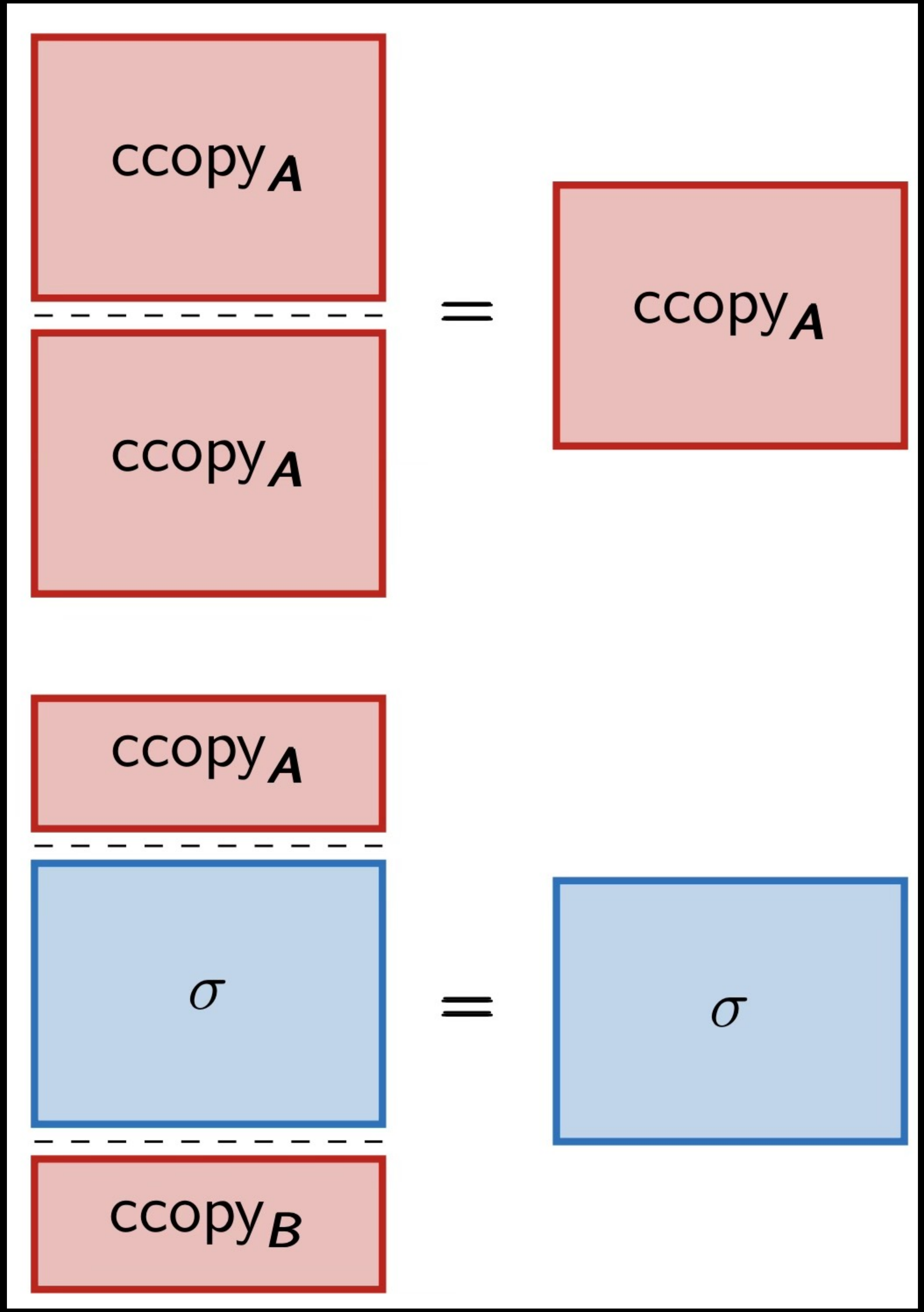


Algebra of Composition



Algebra of Composition

$$ccopy_{tE} = \mathrm{Id}[\gamma] : tE \multimap tE$$



idempotency

when $\sigma = \big\Vert_{\alpha \in \gamma} \widehat{M}[\alpha]$

Observational Refinement

Suppose

Concrete Object

$$\boxed{\nu'_E : tE} \subseteq \begin{array}{c}\boxed{Id[r]}\\ \boxed{\nu_E : tE}\end{array}$$

Abstract Spec

Then

$$\begin{array}{c}\boxed{M[r]}\\ \boxed{\nu'_E}\end{array} \subseteq \begin{array}{c}\boxed{M[r]}\\ \boxed{Id[r]}\\ \boxed{\nu_E}\end{array} = \begin{array}{c}\boxed{M[r]}\\ \boxed{\nu_E}\end{array}$$

Linearizability

ν'_E is linearizable to ν_E when

$$\boxed{\nu'_E : tE} \subseteq \begin{array}{c}\boxed{\mathrm{Id}[\tau]} \\ \boxed{\nu_E : tE}\end{array}$$

Notation

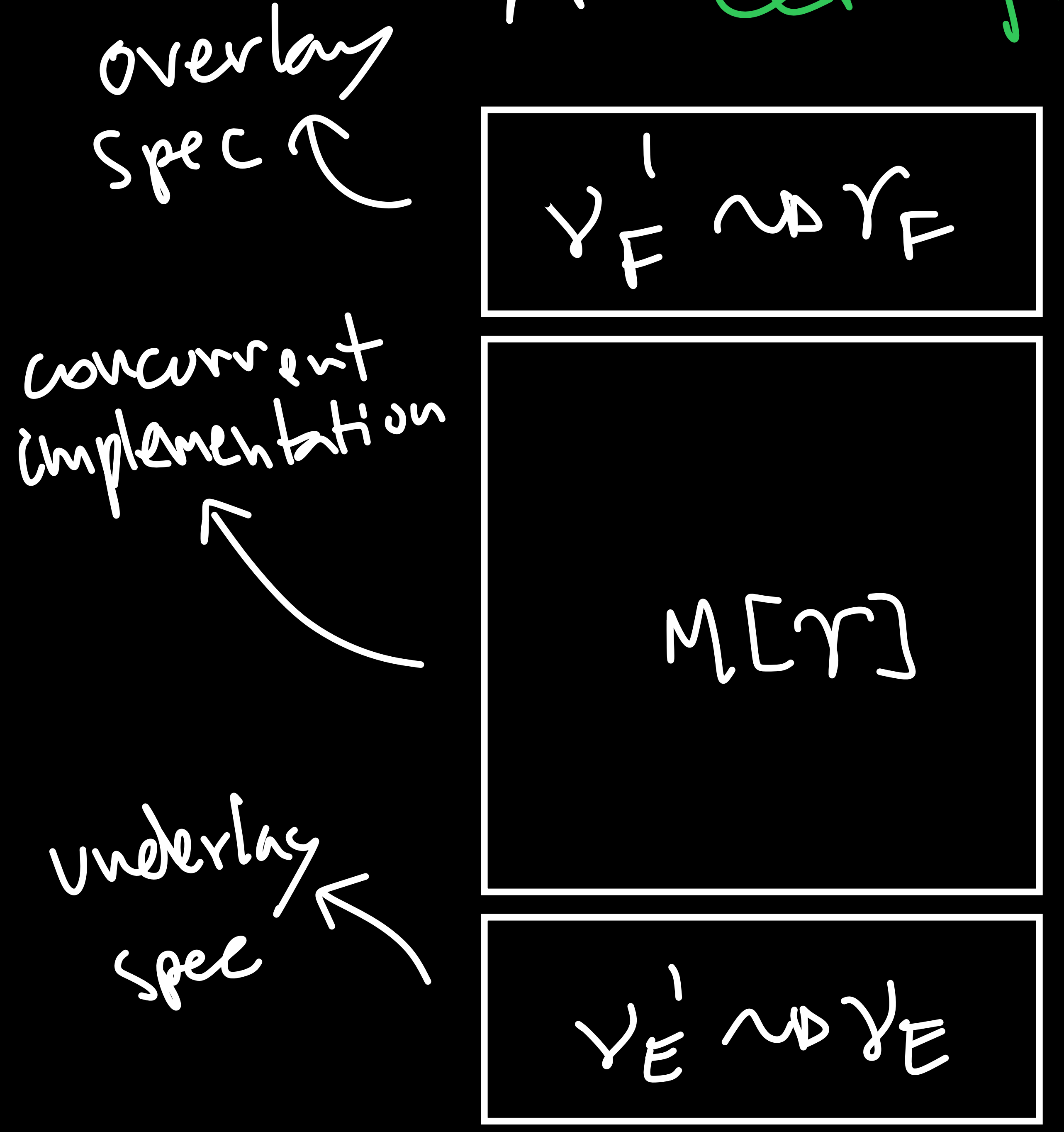
$$\nu'_E \rightsquigarrow \nu_E$$

Certified Concurrent Layers

A layer interface is pair

$$\nu'_E \leadsto \nu_E$$

A certified concurrent layer consists of



$$\nu'_F \subseteq \nu'_E ; \widehat{M[\gamma]}$$

s.t. $\wedge$

$$\nu'_E ; \widehat{M[\gamma]} \leadsto \nu_F$$

Linearizability

ν'_E is linearizable to ν_E
when

$$\boxed{\nu'_E : tE} \subseteq \begin{array}{c} \boxed{ld[r]} \\ \boxed{\nu_E : tE} \end{array}$$

Notation

$$\nu'_E \rightsquigarrow \nu_E$$

When does this hold?

i.e. when does $t \in \nu_E ; \widehat{ld[r]}$?

Examples: Consider atomic counter as the abstract specification

Examples: Consider atomic counter as the abstract specification

1: inc 2: inc 1: inc 1: ok 2: inc 2: ok 1: ok 2: ok

→ more concurrent

→ less concurrent

allows to decrease concurrency

Examples: Consider atomic counter as the abstract specification

1: inc 2: inc

1: inc 1: ok 2: inc 2: ok

1: ok 2: ok → more concurrent

→ less concurrent

allow to decrease concurrency

2: set 1: inc 2: 0

2: get 2: 0

→ 1: inc appears

→ 1: inc removed

allowed to remove pending invocations

Examples: Consider atomic counter as the abstract specification

1: inc 2: inc 1: OK 2: OK → more concurrent

1: inc 1: OK 2: inc 2: OK → less concurrent

allowed to decrease concurrency

2: set 1: inc 2: 0 → 1: inc appears

2: get 2: 0 → 1: inc removed

allowed to remove pending invocations

1: inc 2: get 2: 1 → pending 1: inc

1: inc 1: OK 2: get 2: 1 → completed 1: inc

allowed to complete pending invocations

Examples: Consider atomic counter as the abstract specification

1:inc 2:inc 1:OK 2:OK → more concurrent

1:inc 1:OK 2:inc 2:OK → less concurrent

allowed to decrease concurrency

2:set 1:inc 2:0 → 1:inc appears

2:get 2:0 → 1:inc removed

allowed to remove pending invocations

1:inc 2:get 2:1 → pending 1:inc

1:inc 1:OK 2:get 2:1 → completed 1:inc

allowed to complete pending invocations

1:inc 2:get 1:OK 2:0 2:inc 2:OK

2:get 2:0 1:inc 1:OK 2:inc 2:OK

CANNOT reorder events that happen before each other

Another Example:

ν'_{squeue} : α_0:deq α_1:enq(1) α_0:1 α_2:enq(2) α_1:ok α_0:deq α_2:ok

α_0:deq $\rightarrow$ α_0:1 $\rightarrow$ α_0:deq

α_1:enq(1) $\rightarrow$ α_1:ok

α_2:enq(2) $\rightarrow$ α_2:ok

α_0:deq $\rightarrow$ α_0:1 $\rightarrow$ α_0:deq

α_1:enq(1) $\rightarrow$ α_1:ok

α_2:enq(2) $\rightarrow$ α_2:ok

ν_{squeue} : α_1:enq(1) $\rightarrow$ α_1:ok α_0:deq $\rightarrow$ α_0:1 α_2:enq(2) $\rightarrow$ α_2:ok

Another Example:

ν'_{squeue} : α_0:deq α_1:enq(1) α_0:1 α_2:enq(2) α_1:ok α_0:deq α_2:ok

α_0:deq $\to$ α_0:1 $\to$ α_0:deq

α_1:enq(1) $\to$ α_1:ok

α_2:enq(2) $\to$ α_2:ok

$\rightsquigarrow$

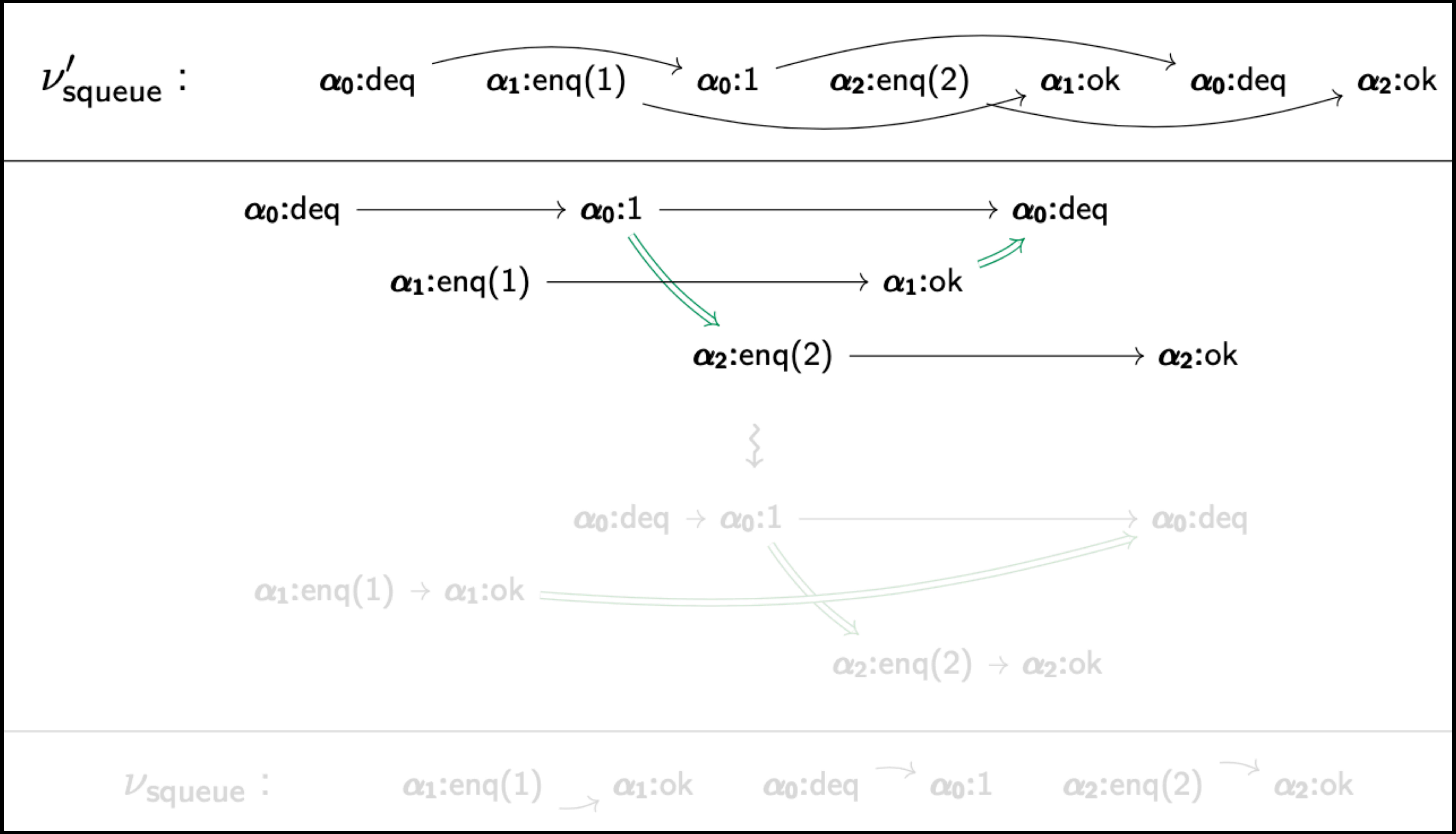
α_0:deq $\to$ α_0:1 $\to$ α_0:deq

α_1:enq(1) $\to$ α_1:ok

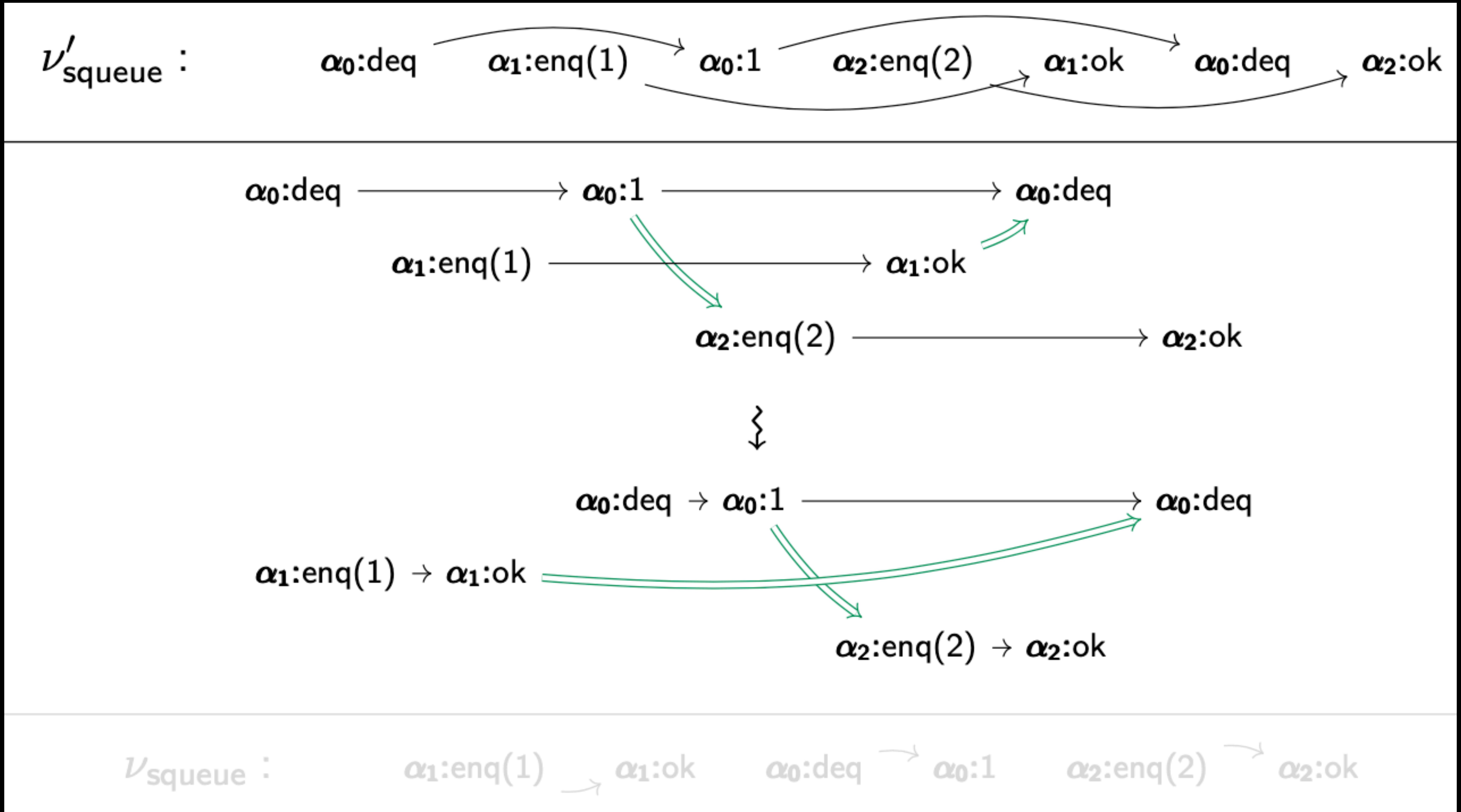
α_2:enq(2) $\to$ α_2:ok

ν_{squeue} : α_1:enq(1) α_1:ok α_0:deq α_0:1 α_2:enq(2) α_2:ok

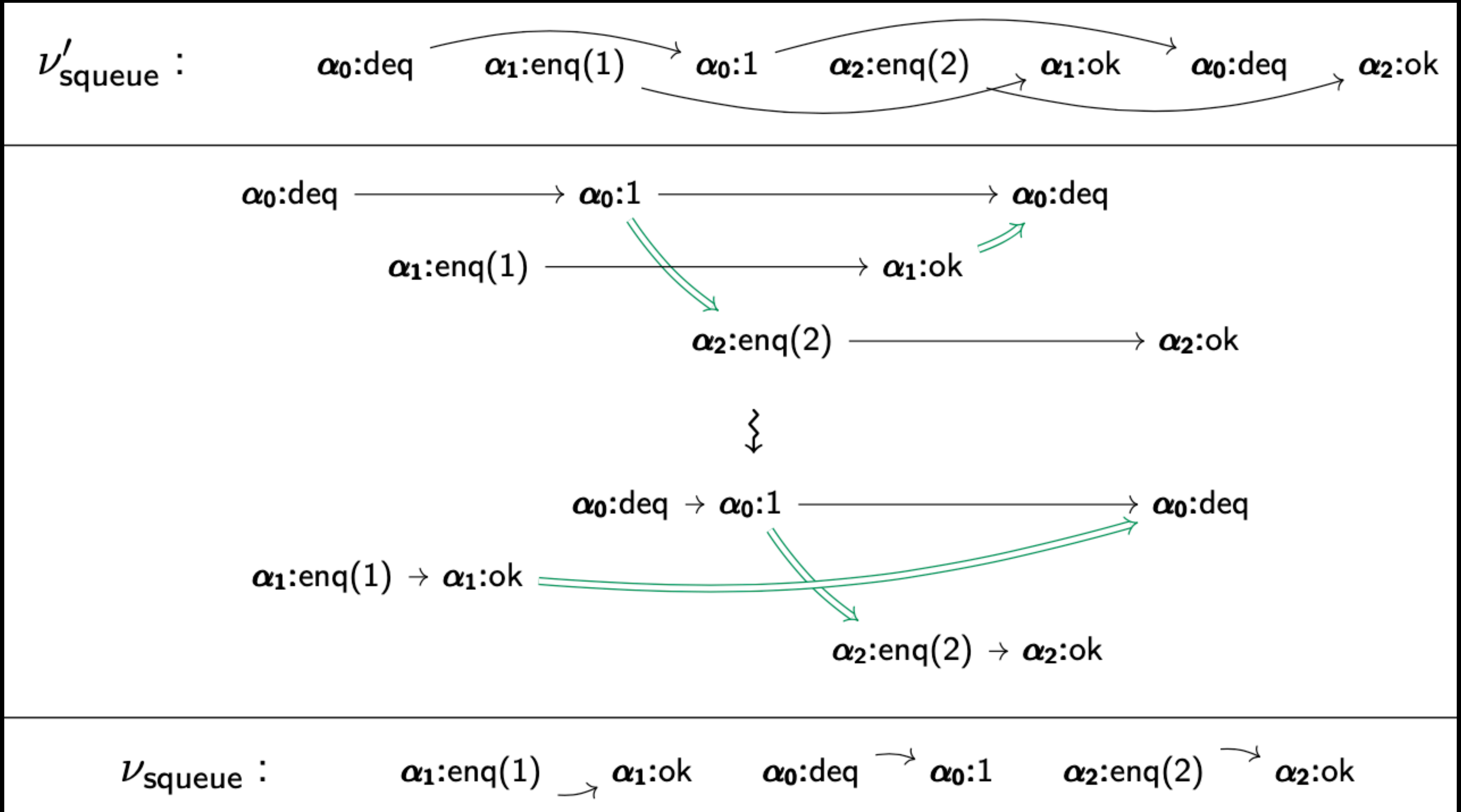
Another Example:



Another Example:



Another Example:



Concrete Linearizability

DEFINITION

s is linearizable to t when there exists a sequence s_O of invocations and a sequence s_P of responses such that

$$s \cdot s_P \rightsquigarrow_{\boldsymbol{A}} t \cdot s_O$$

- t need not be atomic (coincides with Herlihy-Wing when it is);
- s_P = returns;
- s_O = removed pending invocations (not all need be removed);
- $\rightsquigarrow_{\boldsymbol{A}}$ = happens-before order preservation.

$t' \in \nu_{E;\mathrm{Id}\widehat{[r]}} \iff \exists t \in \nu_E.\ t'$ is linearizable w.r.t. ν_E

The Trap of Atomicity

Not every object is atomic

1: exch(X)   2: exch(Y)   1: Y   2: X

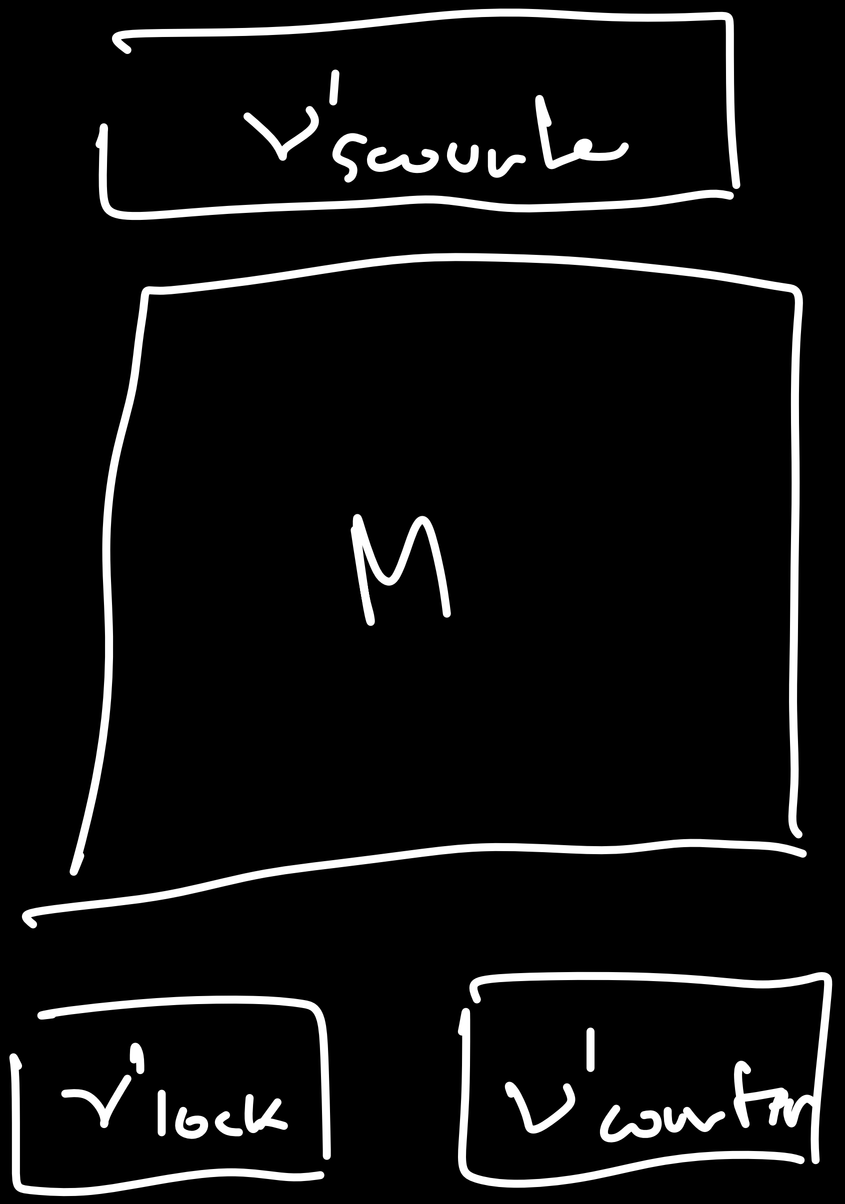
linearizes to

1: exch(X)   1: Y   2, exch(Y)   2: X

2, exch(Y)   2: X   1: exch(X)   1: Y

The Trap of Atomicity

$\nu'_{scounter} \rightsquigarrow \nu_{atomic\ counter}$




```
inc(x) {
  _ ← acq();
  v ← inc(x);
  _ ← rel()
  ret v
}
```

$\nu'_{lock} \rightsquigarrow \nu_{atomic\ lock}$

$\nu'_{counter} \rightsquigarrow \nu_{racy\ counter}$

$\nu_{racy\ counter}$

every trace is

$\underbrace{s}_{\text{atomic counter trace}} \cdot \underbrace{t}_{\text{arbitrary trace that starts with two invocations}}$

```
inc() {
  v ← x.rd();
  _ ← x.wrt(v+1);
  ret OK
}
```