On abstraction and compositionality for weak-memory linearisability

Brijesh Dongol Radha Jagadeesan James Riely
Alasdair Armstrong

Atomicity abstraction and weak memory

- How do we provide reliable atomicity abstractions?
  - Concurrent objects, e.g., locks, stacks, queues etc
  - Transactional memory

Atomicity abstraction and weak memory

- How do we provide reliable atomicity abstractions?
  - Concurrent objects, e.g., locks, stacks, queues etc
  - Transactional memory
- What does a programmer require from an atomicity abstraction?
  - Abstraction (or contextual refinement)
  - Compositionality

Atomicity abstraction and weak memory

- How do we provide reliable atomicity abstractions?
  - Concurrent objects, e.g., locks, stacks, queues etc
  - Transactional memory
- What does a programmer require from an atomicity abstraction?
  - Abstraction (or contextual refinement)
  - Compositionality
- The above well studied assuming sequentially consistent (SC) memory
- How do atomicity abstractions behave under weak (or relaxed) memory?

Atomicity abstraction and weak memory

- How do we provide reliable atomicity abstractions?
  - Concurrent objects, e.g., locks, stacks, queues etc
  - Transactional memory
- What does a programmer require from an atomicity abstraction?
  - Abstraction (or contextual refinement)
  - Compositionality
- The above well studied assuming sequentially consistent (SC) memory
- How do atomicity abstractions behave under weak (or relaxed) memory?
  - Use framework of Alglave, Maranget and Tautschnig (AMT)
    — captures a large number of memory models (TSO, Power, ARM)

Atomicity abstraction and weak memory

- How do we provide reliable atomicity abstractions?
  - Concurrent objects, e.g., locks, stacks, queues etc
  - Transactional memory

- What does a programmer require from an atomicity abstraction?
  - Abstraction (or contextual refinement)
  - Compositionality

- The above well studied assuming sequentially consistent (SC) memory

- How do atomicity abstractions behave under weak (or relaxed) memory?
  - Use framework of Alglave, Maranget and Tautschnig (AMT)
    — captures a large number of memory models (TSO, Power, ARM)
  - Concurrent objects (this paper)
  - Transactional memory (Dongol, Jagadeesan and Riely, POPL 2018)

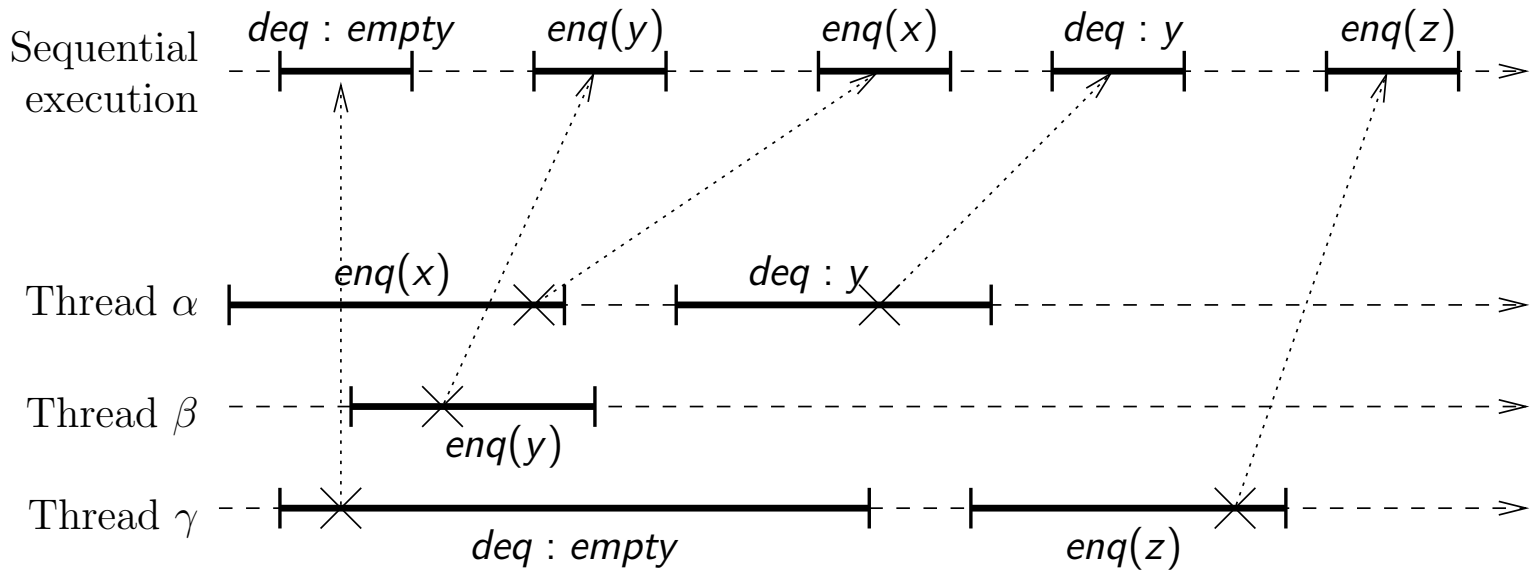
Linearisability for SC

- Correctness of concurrent object defined by linearisability
  - each operation takes effect between invocation and return
  - order of effects legal for sequential specification

Linearisability for SC

- Correctness of concurrent object defined by linearisability
  - each operation takes effect between invocation and return
  - order of effects legal for sequential specification

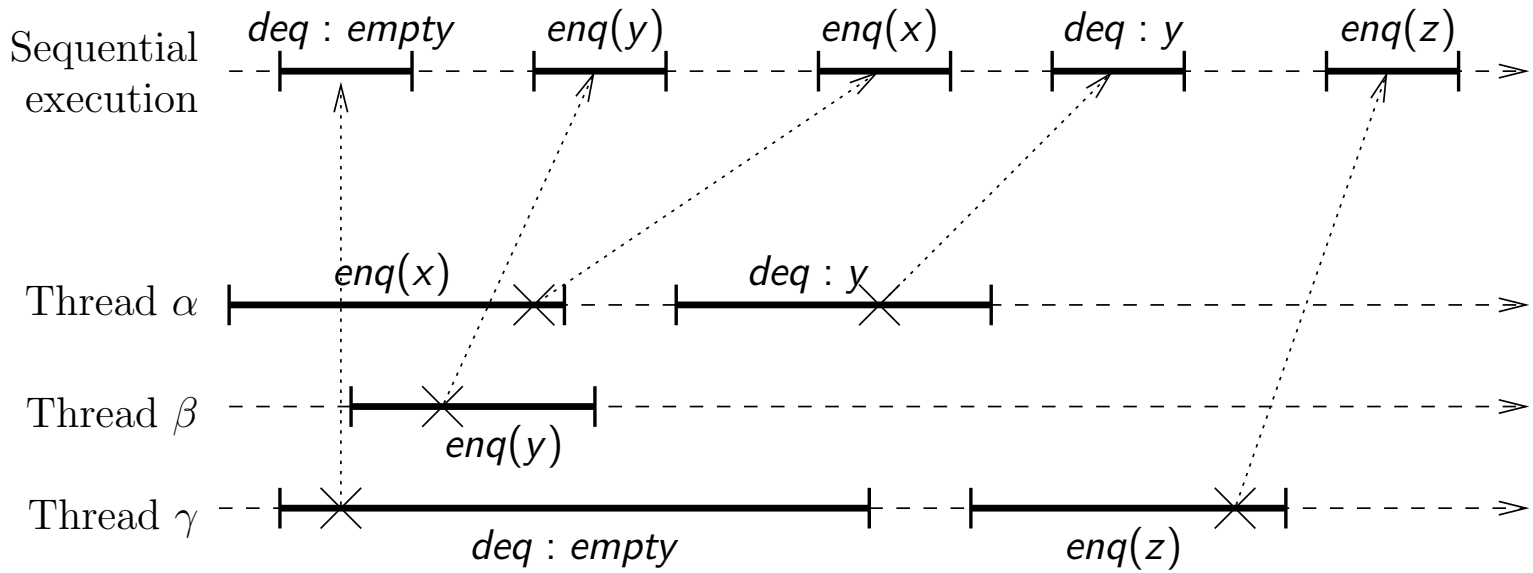
- **Example.** Concurrent queue



Linearisability for SC

- Correctness of concurrent object defined by linearisability
  - each operation takes effect between invocation and return
  - order of effects legal for sequential specification

- **Example.** Concurrent queue

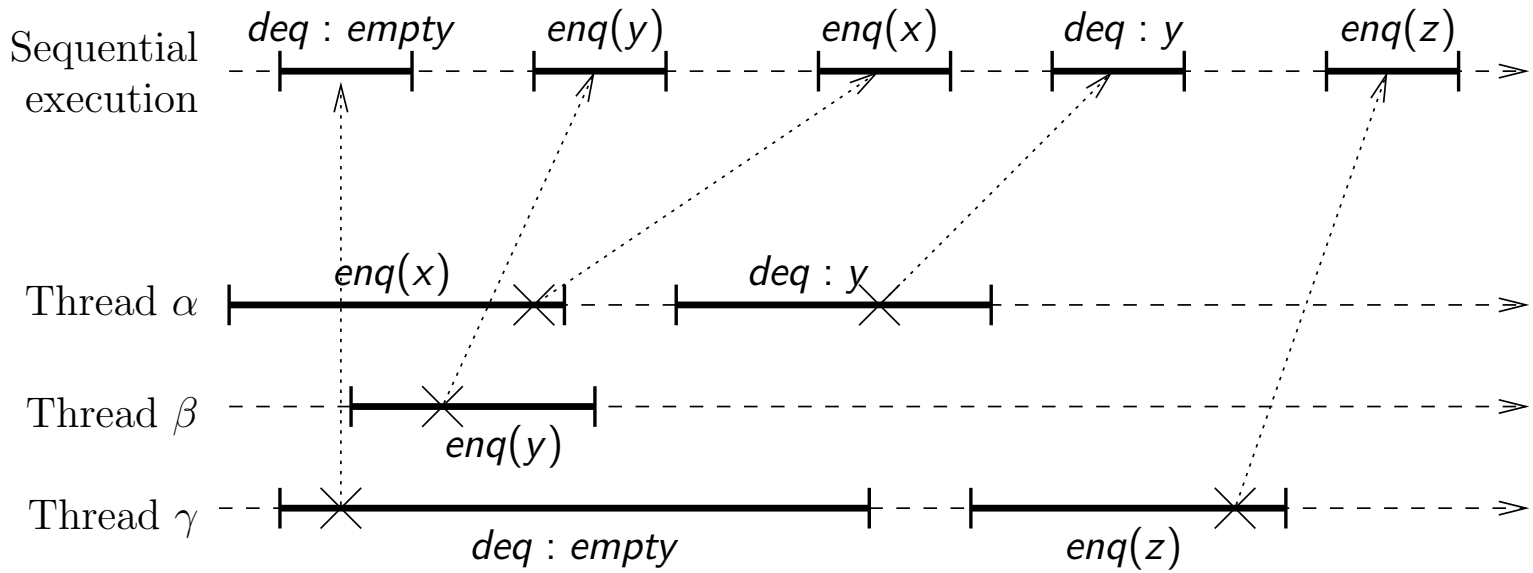


- Properties of linearisability:
  - necessary and sufficient for contextual refinement (Filipovic, 2010)
  - compositional (Herlihy and Wing, 1990)

Linearisability for SC

- Correctness of concurrent object defined by linearisability
  - each operation takes effect between invocation and return
  - order of effects legal for sequential specification

- **Example.** Concurrent queue



- Properties of linearisability:
  - necessary and sufficient for contextual refinement (Filipovic, 2010)
  - compositional (Herlihy and Wing, 1990)

- What about weak memory?

AMT's axiomatic models

Executions defined by:

- Set of (read/write) *events* $\mathbb{E}$ and *orders* over $\mathbb{E}$, e.g.,
  - co (coherence order) — total order on the writes of each location
  - rf (reads from dependency) — maps writes to reads
  - ppo (preserved program order), — program order po with commuting events in architecture removed
  - ...

AMT's axiomatic models

Executions defined by:

- Set of (read/write) *events* $\mathbb{E}$ and *orders* over $\mathbb{E}$, e.g.,
  - co (coherence order) — total order on the writes of each location
  - rf (reads from dependency) — maps writes to reads
  - ppo (preserved program order), — program order po with commuting events in architecture removed
  - ...
- Other relations are derived, e.g.,
  - Happens-before $\mathsf{hb} = \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}$
  - From-read anti-dependency $\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$

AMT's axiomatic models

Executions defined by:

- Set of (read/write) *events* $\mathbb{E}$ and *orders* over $\mathbb{E}$, e.g.,
  - co (coherence order) — total order on the writes of each location
  - rf (reads from dependency) — maps writes to reads
  - ppo (preserved program order), — program order po with commuting events in architecture removed
  - ...
- Other relations are derived, e.g.,
  - Happens-before $\mathsf{hb} = \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}$
  - From-read anti-dependency $\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$

Execution is correct if it satisfies four axioms:

$$\mathsf{acyclic}(\mathsf{hb}) \qquad \text{(No-Thin-Air)}$$

$$\mathsf{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}) \qquad \text{(SC-Per-Location)}$$

$$\mathsf{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*) \qquad \text{(Observation)}$$

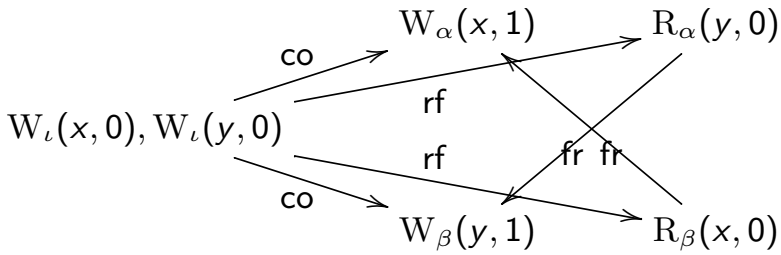
$$\mathsf{acyclic}(\mathsf{co} \cup \mathsf{prop}) \qquad \text{(Propagation)}$$

Example: TSO (load buffering)

```
Init:  x, y = 0, 0
Thread α:  x := 1; r1 := y;
Thread β:  y := 1; r2 := x;
```

Example: TSO (load buffering)

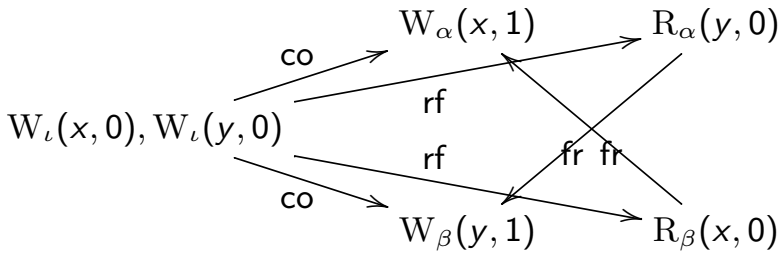
```
Init:  x, y = 0, 0
Thread α:  x := 1; r1 := y;
Thread β:  y := 1; r2 := x;
```



Allowed execution

Example: TSO (load buffering)

```
Init:  x, y = 0, 0
Thread α:  x := 1; r1 := y;
Thread β:  y := 1; r2 := x;
```

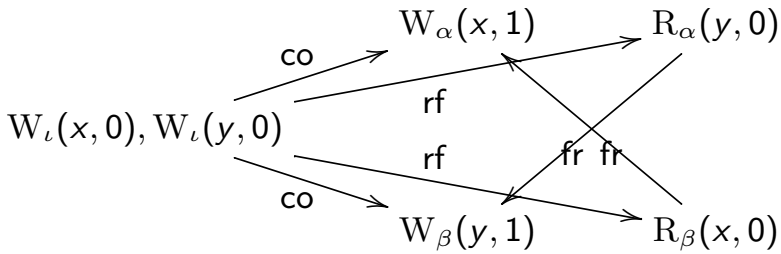


Allowed execution

```
Init:  x, y = 0, 0
Thread α: x := 1; FF; r1 := y;
Thread β: y := 1; FF; r2 := x;
```

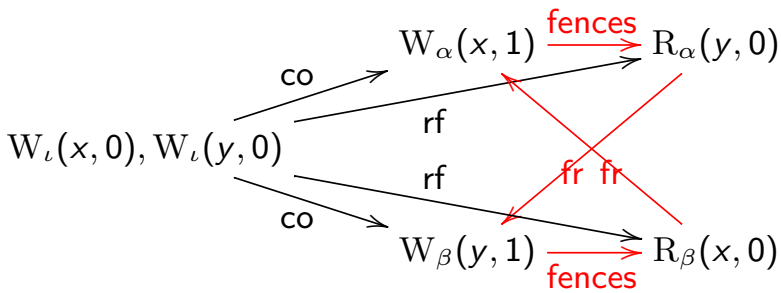
Example: TSO (load buffering)

```
Init:  x, y = 0, 0
Thread α:  x := 1; r1 := y;
Thread β:  y := 1; r2 := x;
```



Allowed execution

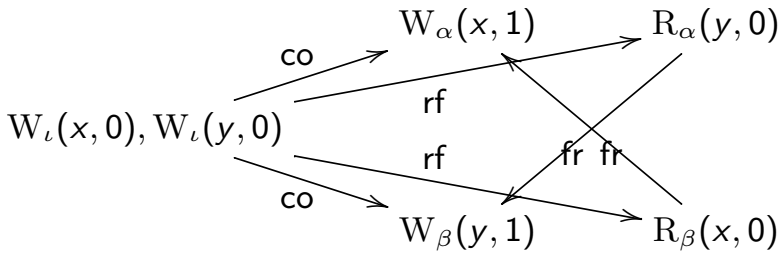
```
Init:  x, y = 0, 0
Thread α: x := 1; FF; r1 := y;
Thread β: y := 1; FF; r2 := x;
```



Disallowed execution

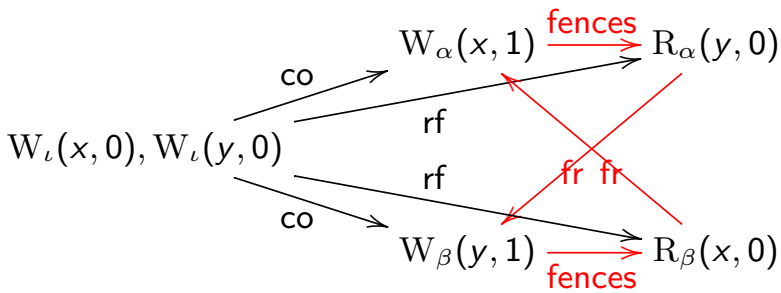
Example: TSO (load buffering)

```
Init:  x, y = 0, 0
Thread α:  x := 1; r1 := y;
Thread β:  y := 1; r2 := x;
```



Allowed execution

```
Init:  x, y = 0, 0
Thread α: x := 1; FF; r1 := y;
Thread β: y := 1; FF; r2 := x;
```



Disallowed execution

Let's apply this framework to a setting with concurrent objects

Abstract objects and weak memory

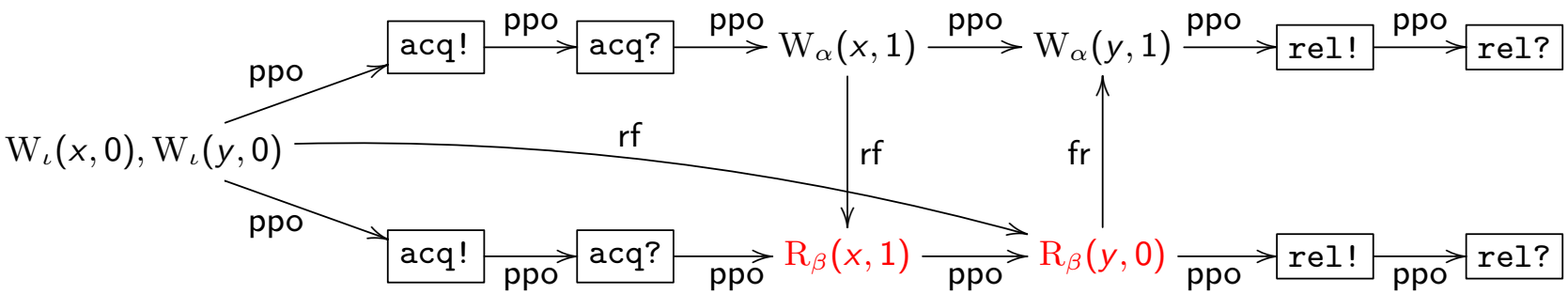
```
Init:  x, y = 0, 0
Thread α:  lock.acq(); x := 1; y := 1; lock.rel();
Thread β:  lock.acq(); print x; print y; lock.rel();
```

- Expected behaviour: Thread β either prints 0 0 or 1 1

Abstract objects and weak memory

```
Init:  x, y = 0, 0
Thread α:  lock.acq(); x := 1; y := 1; lock.rel();
Thread β:  lock.acq(); print x; print y; lock.rel();
```

- Expected behaviour: Thread β either prints 0 0 or 1 1
- Naive use of AMT axioms permits a bad behaviour: β prints 1 0

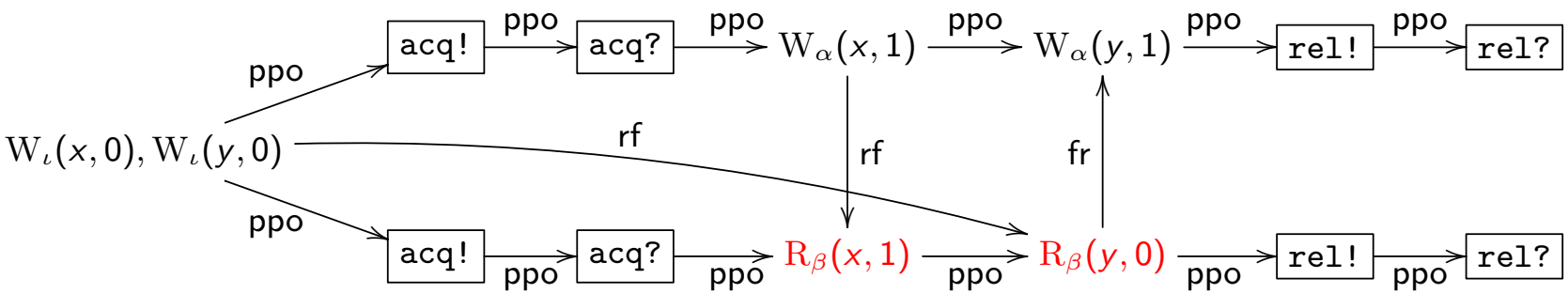


Allowed execution

Abstract objects and weak memory

```
Init:  x, y = 0, 0
Thread α:  lock.acq(); x := 1; y := 1; lock.rel();
Thread β:  lock.acq(); print x; print y; lock.rel();
```

- Expected behaviour: Thread β either prints 0 0 or 1 1
- Naive use of AMT axioms permits a bad behaviour: β prints 1 0



Allowed execution

- Two problems:
  1. Typical SC lock specification is not strong enough
     - Only describes allowable order of operations
     - Doesn't describe memory effects
  2. Weak memory axioms ignore interaction with lock object

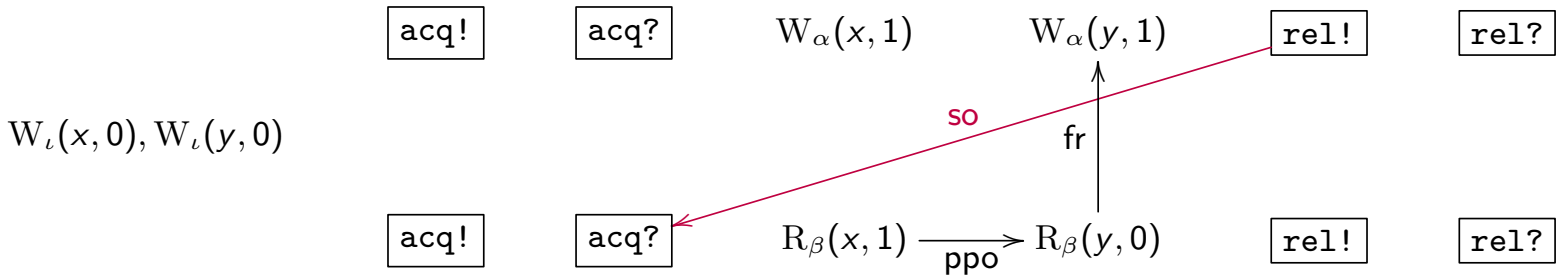
Abstract objects and weak memory

```
Init:  x, y = 0, 0
Thread α:  lock.acq(); x := 1; y := 1; lock.rel();
Thread β:  lock.acq(); print x; print y; lock.rel();
```

Abstract objects and weak memory

```
Init:  x, y = 0, 0
Thread α:  lock.acq(); x := 1; y := 1; lock.rel();
Thread β:  lock.acq(); print x; print y; lock.rel();
```

Disallowing bad behaviour (only showing relevant edges):

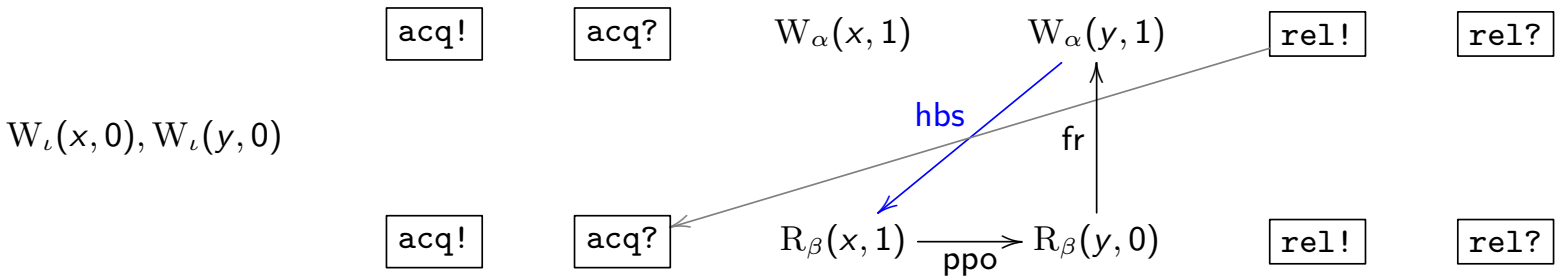


1. Strengthen the lock specification to include specification order (so) from release to acquire

Abstract objects and weak memory

```
Init:  x, y = 0, 0
Thread α:  lock.acq(); x := 1; y := 1; lock.rel();
Thread β:  lock.acq(); print x; print y; lock.rel();
```

Disallowing bad behaviour (only showing relevant edges):



Violates OBSERVATION axiom

1. Strengthen the lock specification to include specification order (so) from release to acquire
2. Strengthen the weak memory axioms to induce additional happens before (hbs)

Strengthening specifications

- Under SC, a specification defined by legal history

  Example. Legal stack history (ensures LIFO order)

  $$a : push!(5) \cdot a : push? \cdot b : push!(6) \cdot b : push? \cdot c : pop! \cdot c : pop?(6)$$

Strengthening specifications

- Under SC, a specification defined by legal history

  Example. Legal stack history (ensures LIFO order)

  $$a : push!(5) \cdot a : push? \cdot b : push!(6) \cdot b : push? \cdot c : pop! \cdot c : pop?(6)$$

- In weak memory, stack specification orders push invocation and corresponding pop return

  Example. For stack history above

  $$b : push!(6) \xrightarrow{so} c : pop?(6)$$

Strengthening specifications

- Under SC, a specification defined by legal history

  Example. Legal stack history (ensures LIFO order)

  $$a : push!(5) \cdot a : push? \cdot b : push!(6) \cdot b : push? \cdot c : pop! \cdot c : pop?(6)$$

- In weak memory, stack specification orders push invocation and corresponding pop return

  Example. For stack history above

  $$b : push!(6) \xrightarrow{so} c : pop?(6)$$

- Specification order is used to build additional happens-before

  $$\mathsf{hbs} \quad = \quad \mathsf{po}; \mathsf{so}; \mathsf{po}$$

Abstract executions via example (publication)

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Possible (bad) abstract trace of this client/object program:

$$\mathrm{W}_\alpha(x,5) \quad \mathit{push}!_\alpha(5) \quad \mathit{push}?_\alpha$$

$$\mathrm{W}_\iota(x,0)$$

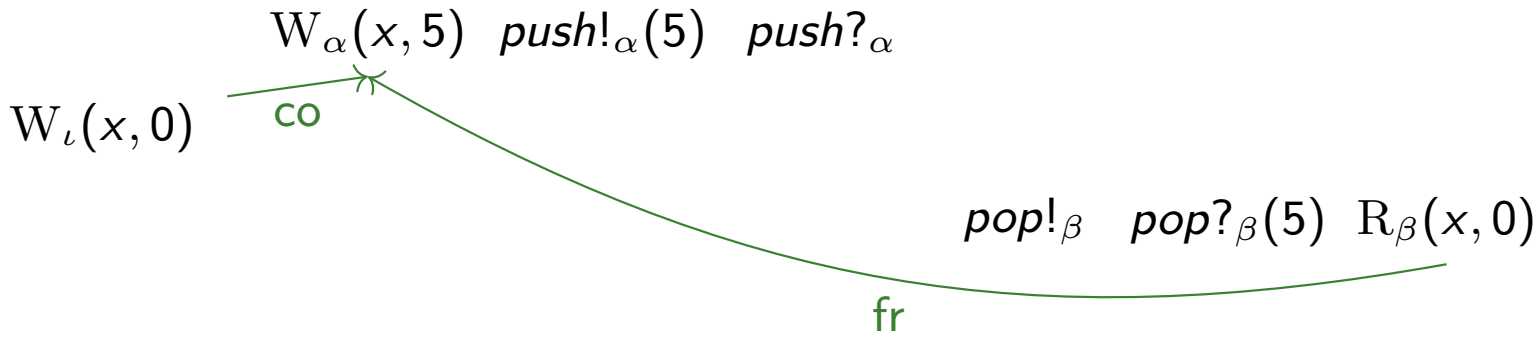
$$\mathit{pop}!_\beta \quad \mathit{pop}?_\beta(5) \quad \mathrm{R}_\beta(x,0)$$

How to show trace invalid for memory model?

Abstract executions via example (publication)

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Possible (bad) abstract trace of this client/object program:



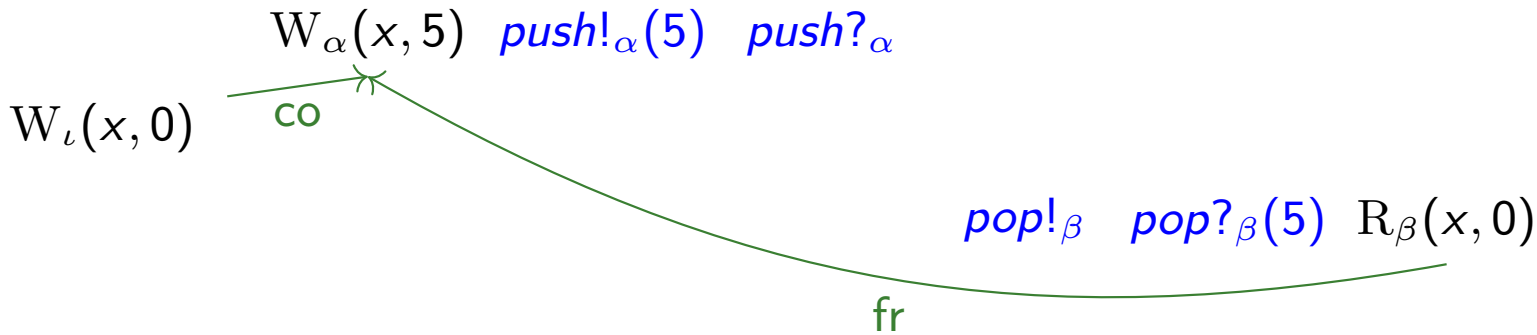
How to show trace invalid for memory model?

1. AMT framework gives execution orders

Abstract executions via example (publication)

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Possible (bad) abstract trace of this client/object program:



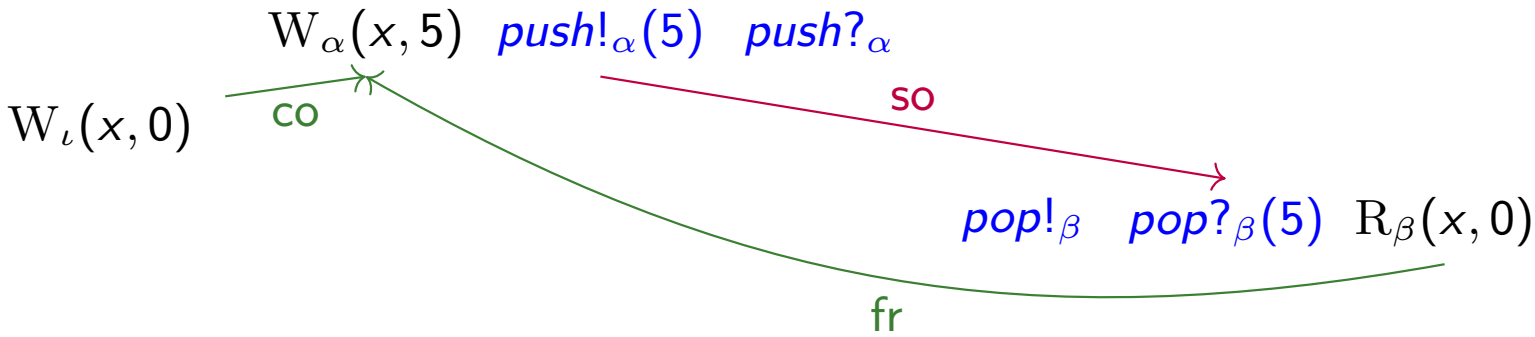
How to show trace invalid for memory model?

1. AMT framework gives execution orders
2. Check history (i.e., trace restricted to specification) is valid

Abstract executions via example (publication)

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Possible (bad) abstract trace of this client/object program:



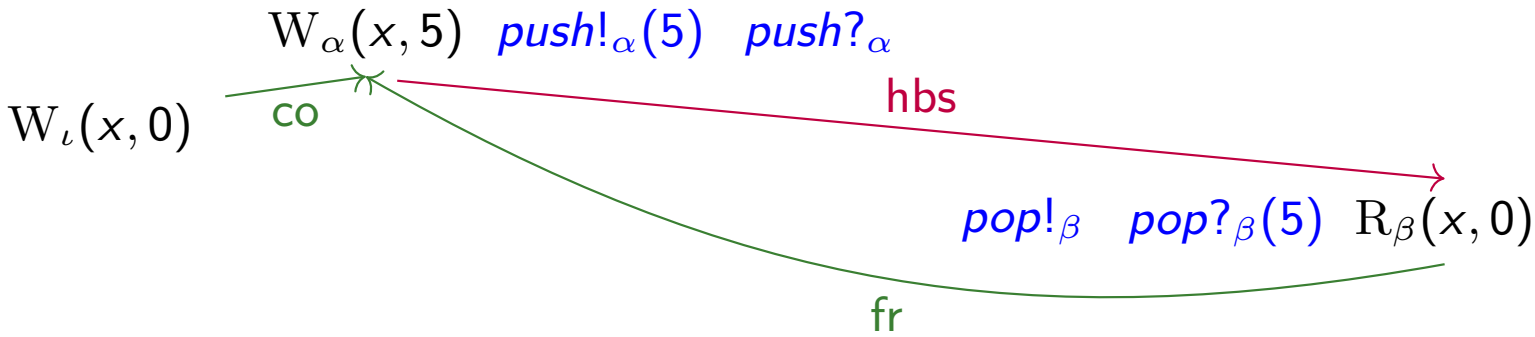
How to show trace invalid for memory model?

1. AMT framework gives execution orders
2. Check history (i.e., trace restricted to specification) is valid
3. Weak memory stack specification has order so for stack history

Abstract executions via example (publication)

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Possible (bad) abstract trace of this client/object program:



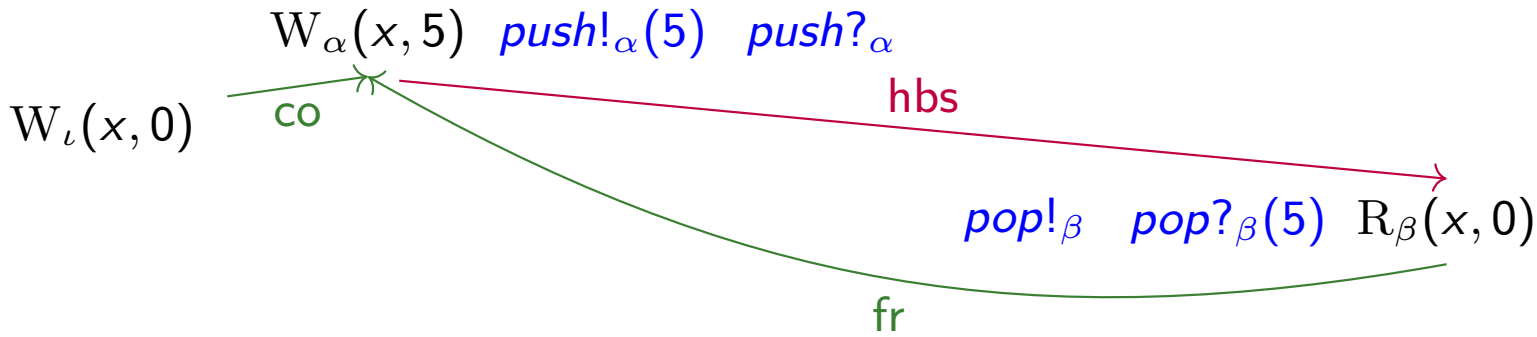
How to show trace invalid for memory model?

1. AMT framework gives execution orders
2. Check history (i.e., trace restricted to specification) is valid
3. Weak memory stack specification has order so for stack history
4. This induces a lifted specification order hbs = po ; so; po

Abstract executions via example (publication)

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Possible (bad) abstract trace of this client/object program:



How to show trace invalid for memory model?

1. AMT framework gives execution orders
2. Check history (i.e., trace restricted to specification) is valid
3. Weak memory stack specification has order so for stack history
4. This induces a lifted specification order hbs = po ; so; po
5. Execution is invalid (as desired) according to AMT axioms using

$$\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{hbs}$$

Concrete implementations

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

Concrete implementations

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

How to show concrete trace of this client/object program is invalid?

$\mathrm{W}_\alpha(x,5)$ $\mathit{push}!_\alpha(5)$ S $\mathit{push}?_\alpha$

$\mathrm{W}_\iota(x,0)$

$\mathit{pop}!_\beta$ T $\mathit{pop}?_\beta(5)$ $\mathrm{R}_\beta(x,0)$

Concrete implementations

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

How to show concrete trace of this client/object program is invalid?

$$\mathrm{W}_\alpha(x,5)\ \ \mathit{push}!_\alpha(5)\ \ S\ \ \mathit{push}?_\alpha$$

$$\mathrm{W}_\iota(x,0)$$

$$\mathit{pop}!_\beta\ \ T\ \ \mathit{pop}?_\beta(5)\ \mathrm{R}_\beta(x,0)$$

- If the stack is correct, then there must be memory actions in S, T that invalidates the execution

Concrete implementations

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

How to show concrete trace of this client/object program is invalid?

$$\mathrm{W}_\alpha(x,5)\ \ push!_\alpha(5)\ \ S\ \ push?_\alpha$$

$$\mathrm{W}_\iota(x,0)$$

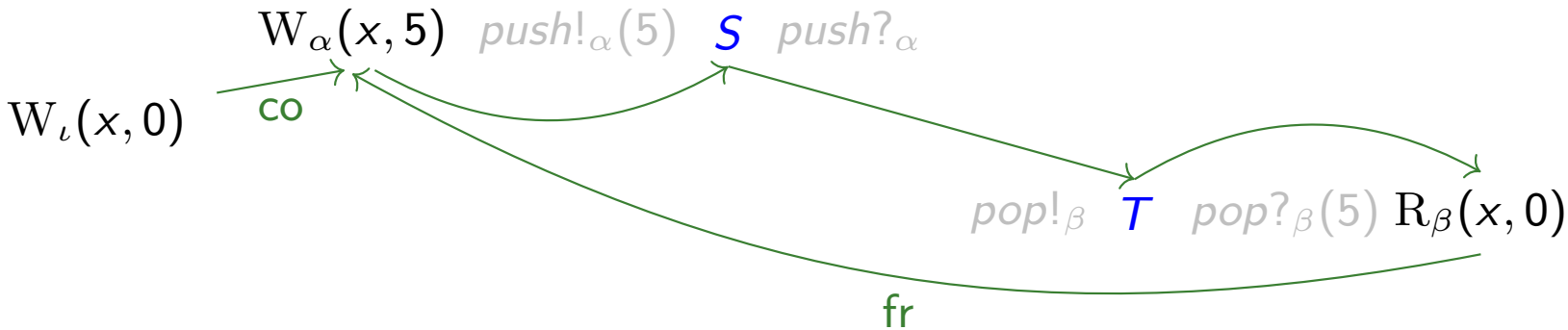
$$pop!_\beta\ \ T\ \ pop?_\beta(5)\ \mathrm{R}_\beta(x,0)$$

- If the stack is correct, then there must be memory actions in S, T that invalidates the execution
- Can simply
  - ignore the method calls/returns,

Concrete implementations

```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

How to show concrete trace of this client/object program is invalid?

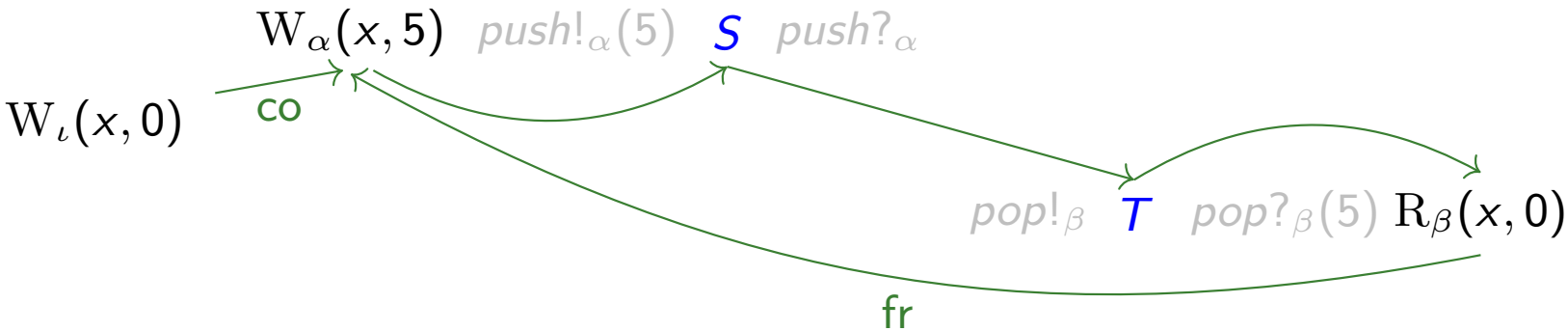


- If the stack is correct, then there must be memory actions in S, T that invalidates the execution
- Can simply
  - ignore the method calls/returns,
  - apply AMT framework

Concrete implementations

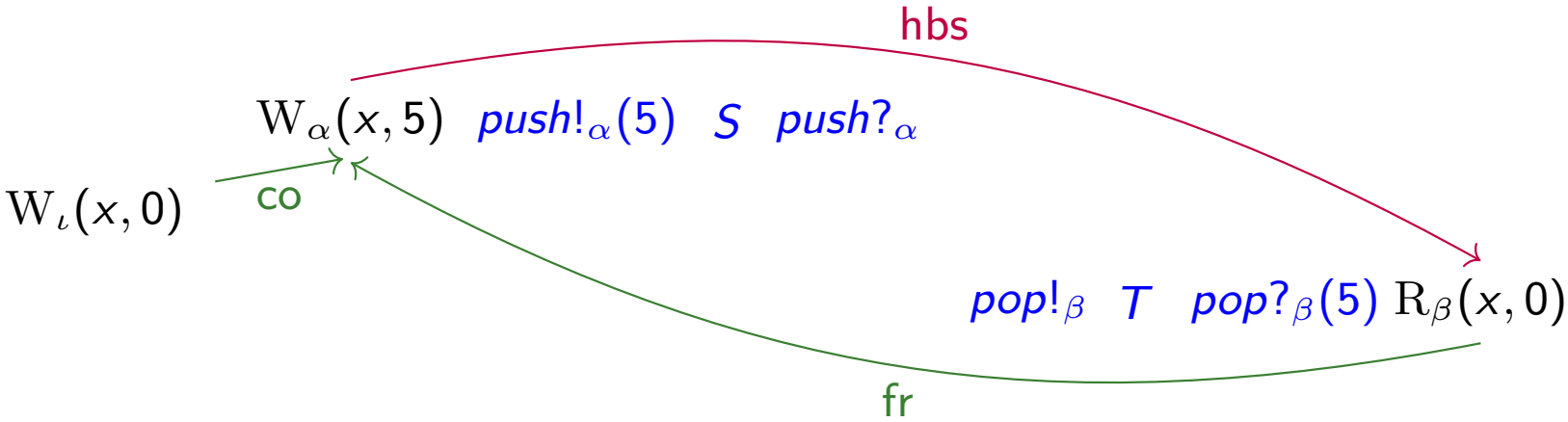
```
Init:  x = 0
Thread α:  x := 5; push(5);
Thread β:  r1 := pop(); r2 := x;
```

How to show concrete trace of this client/object program is invalid?



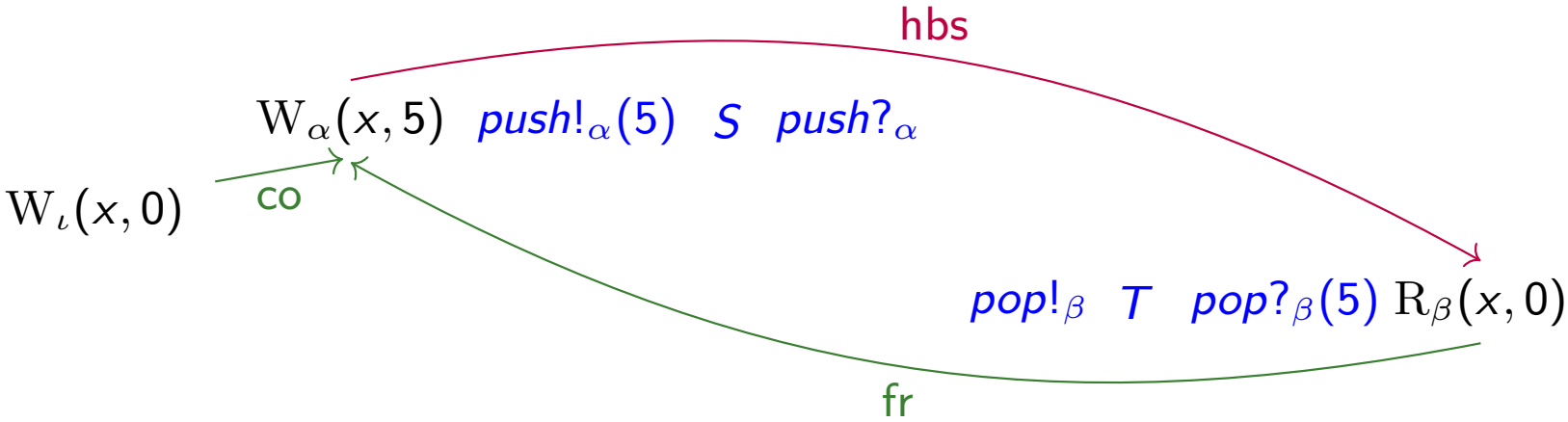
- If the stack is correct, then there must be memory actions in S, T that invalidates the execution
- Can simply
  - ignore the method calls/returns,
  - apply AMT framework
- But we want to think about clients and object implementations separately
- Use the abstract specification as the glue

Concrete implementations



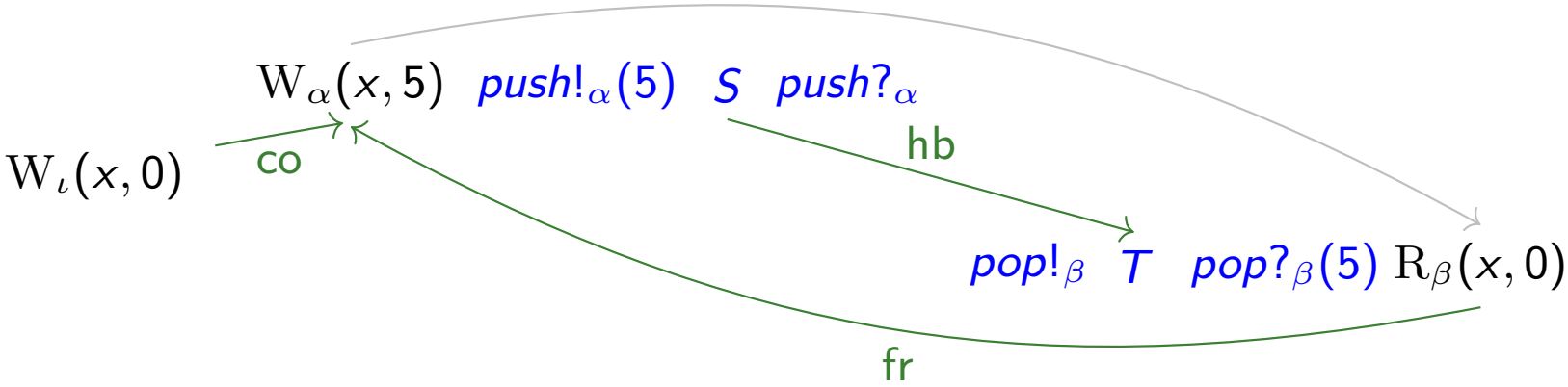
- Recall: We require hbs to create a cycle using a from read anti-depedency

Concrete implementations



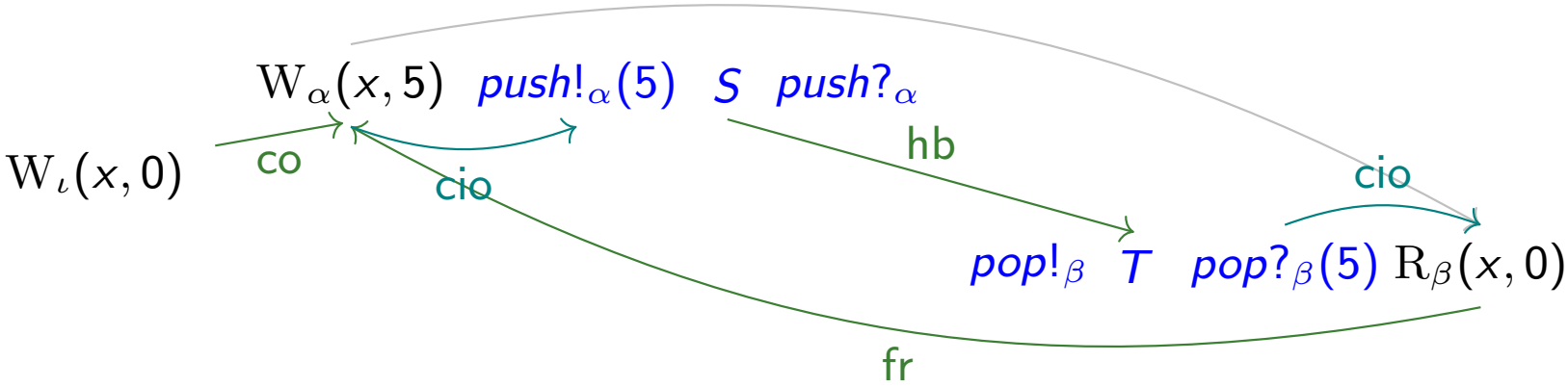
- Recall: We require hbs to create a cycle using a from read anti-depedency
- Problem: How do we ensure hbs exists
  - without knowledge of client (reasoning about the object only),
  - generically for any memory model?

Concrete implementations



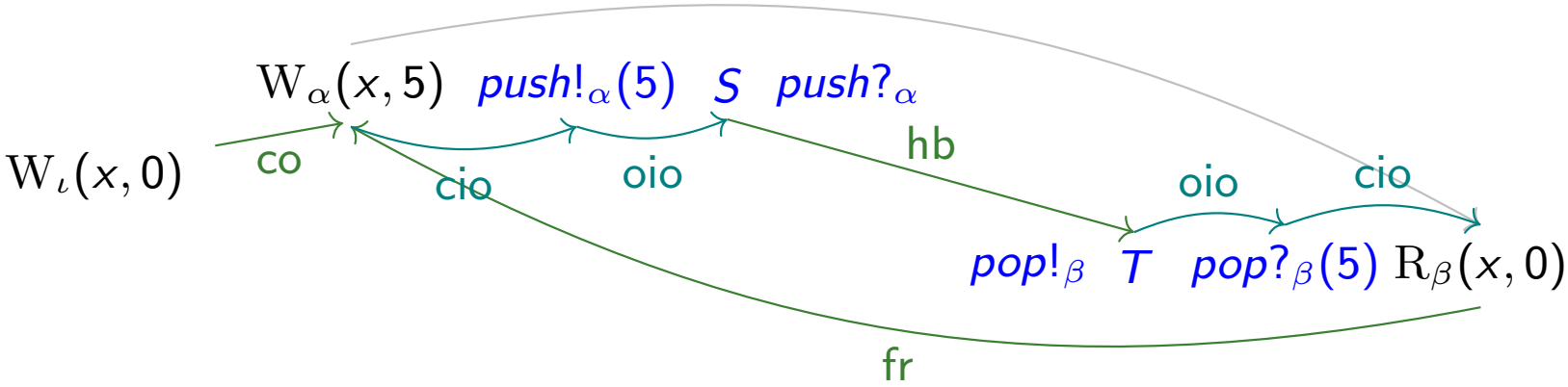
- Recall: We require hbs to create a cycle using a from read anti-depedency
- Problem: How do we ensure hbs exists
  - without knowledge of client (reasoning about the object only),
  - generically for any memory model?
- Three sets of edges to consider
  1. hb order from S to T (given by the memory model)

Concrete implementations



- Recall: We require hbs to create a cycle using a from read anti-depedency
- Problem: How do we ensure hbs exists
  - without knowledge of client (reasoning about the object only),
  - generically for any memory model?
- Three sets of edges to consider
  1. hb order from S to T (given by the memory model)
  2. client-interface order (cio):
     - client events to invocations and
     - responses to client events

Concrete implementations



- Recall: We require hbs to create a cycle using a from read anti-depedency
- Problem: How do we ensure hbs exists
  - without knowledge of client (reasoning about the object only),
  - generically for any memory model?
- Three sets of edges to consider
  1. hb order from S to T (given by the memory model)
  2. client-interface order (cio):
     - client events to invocations and
     - responses to client events
  3. object-interface order (oio):
     - invocations to object events, and
     - object events to responses

Preventing artificial order

- We cannot, by default, include both $\xrightarrow{\text{cio}}$ and $\xrightarrow{\text{oio}}$

  This gives us “artificial” order, which may not exist in memory model

Preventing artificial order

- We cannot, by default, include both $\xrightarrow{\text{cio}}$ and $\xrightarrow{\text{oio}}$

  This gives us "artificial" order, which may not exist in memory model

- **Example.** Empty method with no memory events creates extra order

Program 1 in TSO

`Thread` α`:  x:=1; r:=y`

$\mathrm{W}_\alpha(x,1)$ $\quad$ $\mathrm{R}_\alpha(y,0)$

Preventing artificial order

- We cannot, by default, include both $\xrightarrow{\text{cio}}$ and $\xrightarrow{\text{oio}}$

  This gives us "artificial" order, which may not exist in memory model

- **Example.** Empty method with no memory events creates extra order

Program 1 in TSO

```
Thread α:  x:=1; r:=y
```

$\mathrm{W}_\alpha(x,1) \qquad \mathrm{R}_\alpha(y,0)$

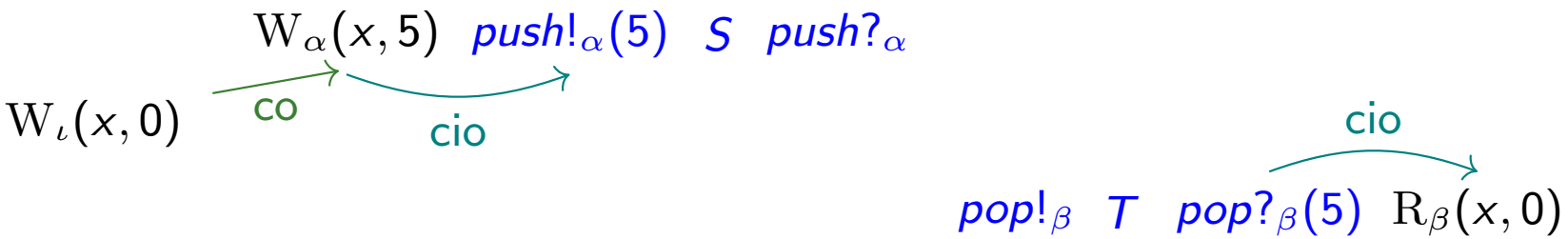
Program 2 in TSO

```
Thread α:  x:=1; empty(); r:=y
```

$\mathrm{W}_\alpha(x,1) \xrightarrow{\text{cio}} E! \xrightarrow{\text{oio}} E? \xrightarrow{\text{cio}} \mathrm{R}_\alpha(y,0)$

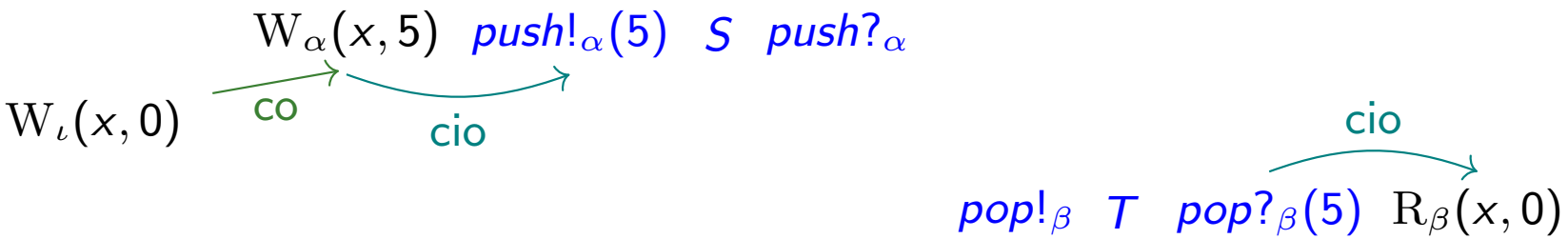
- **Solution.**
  - By default assume: ClientEvent $\xrightarrow{\text{cio}}$ Invocation
  - Conditionally have: Invocation $\xrightarrow{\text{oio}}$ ObjectEvent

Object interface orders (oio) in example



- A correct stack implementation must guarantee:
  1. $S \xrightarrow{\text{hb}} T$
  2. $push!_\alpha(5) \xrightarrow{\text{oio}} S$
  3. $T \xrightarrow{\text{oio}} pop?_\beta(5)$

Object interface orders (oio) in example



- A correct stack implementation must guarantee:
  1. $S \xrightarrow{\text{hb}} T$
  2. $push!_\alpha(5) \xrightarrow{\text{oio}} S$
  3. $T \xrightarrow{\text{oio}} pop?_\beta(5)$
- Reasoning above entirely contained within the object
- All orders stem from the memory model

Programmer expectation

- Want a condition Z such that for any
  - abstract object AS and
  - concrete object CS

$$Z(AS, CS) \quad \Rightarrow \quad \forall C \in \mathit{Client}.\ C[AS] \sqsubseteq C[CS] \qquad \text{(Abstraction)}$$

Programmer expectation

- Want a condition Z such that for any
  - abstract object AS and
  - concrete object CS

$$Z(AS, CS) \quad \Rightarrow \quad \forall C \in \mathit{Client}.\ C[AS] \sqsubseteq C[CS] \qquad \text{(Abstraction)}$$

- We develop two instantiations of Z:
  - real-time hb-linearisability
  - causal hb-linearisability

Real-time hb-linearisability

Definition

Client-implementation trace t *real-time* hb-*linearisable* with respect to (h, so) if

$$\forall \alpha \in \mathit{Threads}.\ t|(\mathbb{I} \cup \mathbb{R})|\alpha = h|\alpha \qquad \text{(Permutation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{t} i \ \Rightarrow\ r \xrightarrow{h} i \qquad \text{(RTO-Preservation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ i \xrightarrow{\mathsf{so}} r \ \Rightarrow\ i \xrightarrow{\mathsf{hb}^+} r \qquad \text{(HB-Satisfaction)}$$

Real-time hb-linearisability

Definition

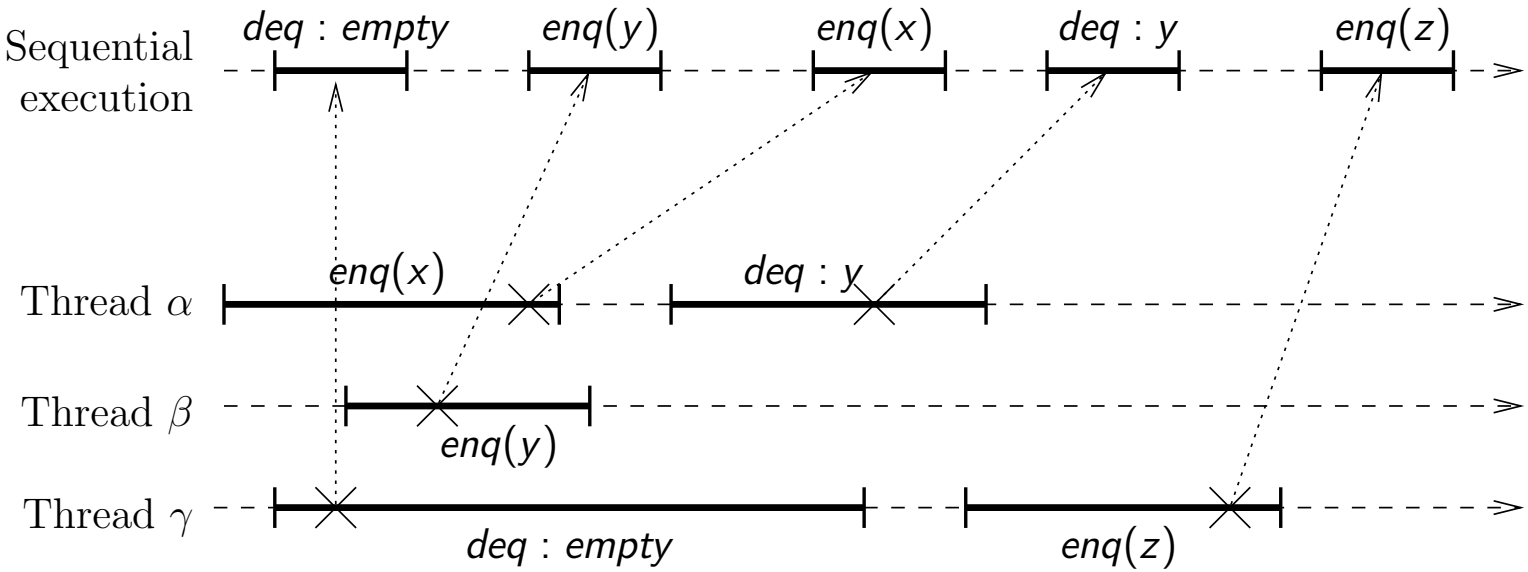
Client-implementation trace t *real-time* hb-*linearisable* with respect to (h, so) if

$$\forall \alpha \in \mathit{Threads}.\ t|(\mathbb{I} \cup \mathbb{R})|\alpha = h|\alpha \qquad \text{(Permutation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{t} i \ \Rightarrow\ r \xrightarrow{h} i \qquad \text{(RTO-Preservation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ i \xrightarrow{\mathsf{so}} r \ \Rightarrow\ i \xrightarrow{\mathsf{hb}^+} r \qquad \text{(HB-Satisfaction)}$$

Example. Concurrent queue (linearisability)



Real-time hb-linearisability

Definition

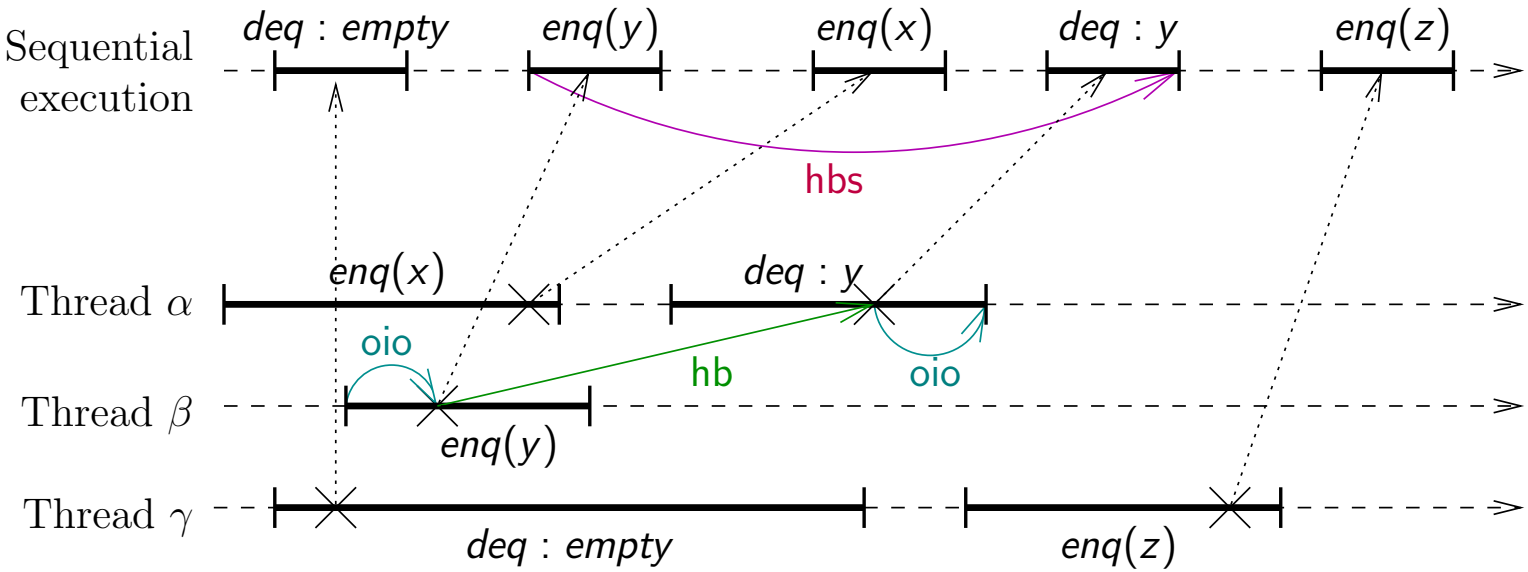
Client-implementation trace t *real-time* hb-*linearisable* with respect to (h, so) if

$$\forall \alpha \in \mathit{Threads}.\ t|(\mathbb{I} \cup \mathbb{R})|\alpha = h|\alpha \qquad \text{(Permutation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{t} i \ \Rightarrow\ r \xrightarrow{h} i \qquad \text{(RTO-Preservation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ i \xrightarrow{\mathsf{so}} r \ \Rightarrow\ i \xrightarrow{\mathsf{hb}^+} r \qquad \text{(HB-Satisfaction)}$$

Example. Concurrent queue (hb-linearisability)



Causal hb-linearisability

- In the weak memory setting, there is an opportunity to relax the real-time order constraint
- Two operations are ordered (in an implementation) iff they are ordered by hb

Causal hb-linearisability

- In the weak memory setting, there is an opportunity to relax the real-time order constraint
- Two operations are ordered (in an implementation) iff they are ordered by hb

Definition

Implementation trace t is *causal* hb-*linearisable* with respect to (h, so) iff

$$\forall \alpha \in \mathit{Threads}.\ t|(\mathbb{I} \cup \mathbb{R})|\alpha = h|\alpha \qquad \text{(Permutation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{\mathsf{hb}^+} i \ \Rightarrow\ r \xrightarrow{h} i \qquad \text{(HB-Preservation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ i \xrightarrow{\mathsf{so}} r \ \Rightarrow\ i \xrightarrow{\mathsf{hb}^+} r \qquad \text{(HB-Satisfaction)}$$

Abstraction and compositionality

- Both real-time and causal hb-linearisability guarantee abstraction
- Real-time hb-linearisability ensures compositionality
- Compositionality for causal hb-linearisability requires either
  - an unobtrusive client, or
  - a commutative specification

Our paper

Contributions:

1. Extension of AMT model to cope with client-object programs
2. Enable objects to be developed independently of client:
   - Real-time hb-linearisability
   - Causal hb-linearisability
3. Abstraction and compositionality theorems for both forms of linearisability

Benefit: Applicable to many different memory models: TSO, Power, ARMv7 ...

Questions?