# Higher-Order Linearisability

**Andrzej Murawski**
University of Warwick

**Nikos Tzevelekos**
Queen Mary U. of London

me

Compositionality workshop, Simons Institute, Dec 2016

# What this talk is about

Linearisability:

- standard correctness criterion for concurrent libraries (in the context of programming languages)
- proves conformance to given behaviour specification

- applicable to first-order programs

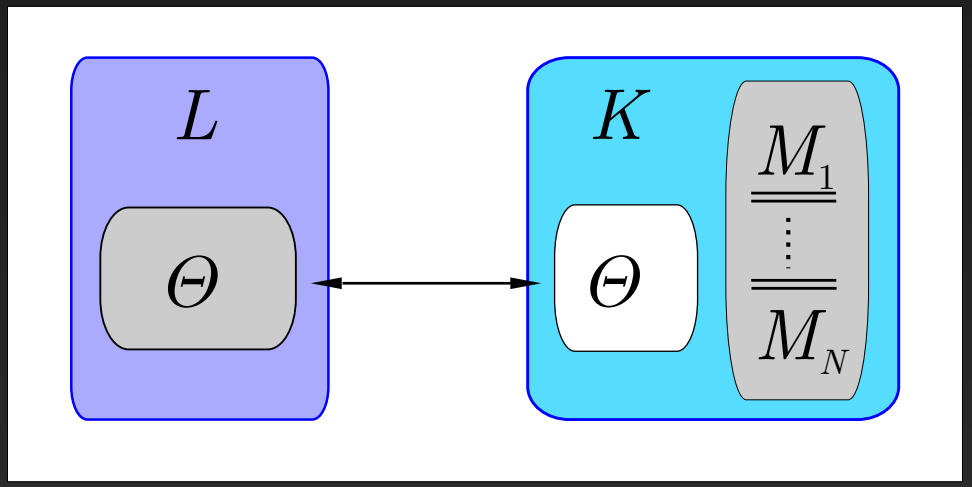
→ we examine linearisability using game semantics

→ extend it to general higher-order libraries

# Concurrent library behaviour

Consider a *queue* library $L$, invoked concurrently, with methods:

**enqueue:** int $\rightarrow$ void **dequeue:** void $\rightarrow$ int




Library $L : \Theta$, and $N$-threaded client $K$

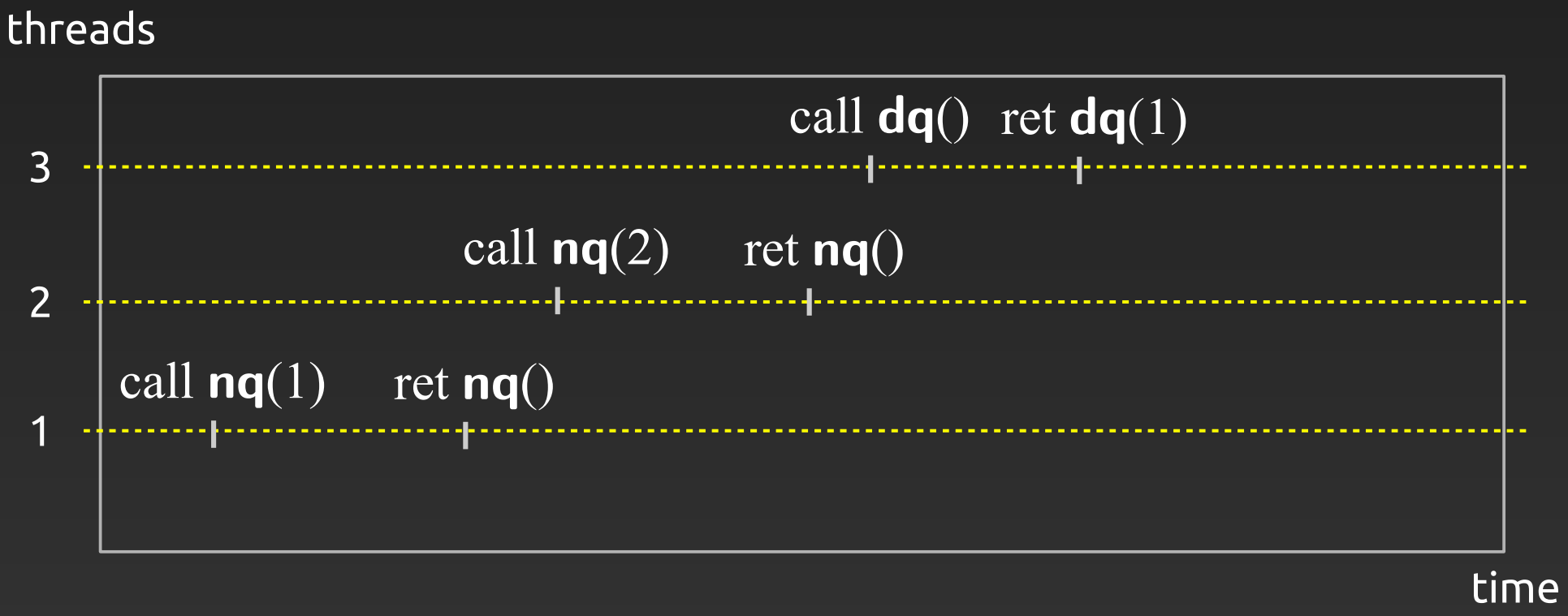
$\Theta = \{$ **nq:** int $\rightarrow$ void, **dq:** void $\rightarrow$ int $\}$

# Concurrent library behaviour

Consider a *queue* library $L$, invoked concurrently, with methods:

**enqueue:** int $\rightarrow$ void **dequeue:** void $\rightarrow$ int

A (correct) library behaviour is the following:

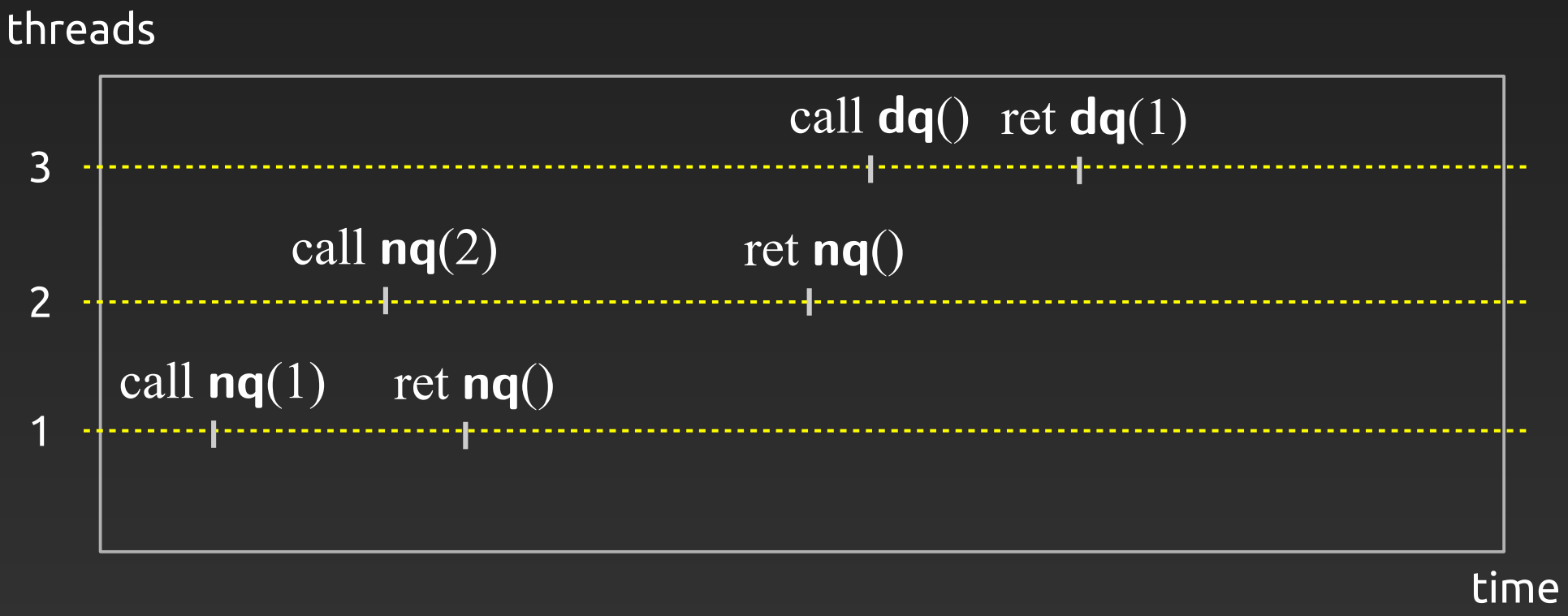


# Concurrent library behaviour

Consider a *queue* library $L$, invoked concurrently, with methods:

**enqueue:** int $\rightarrow$ void **dequeue:** void $\rightarrow$ int

A (correct) library behaviour is the following:

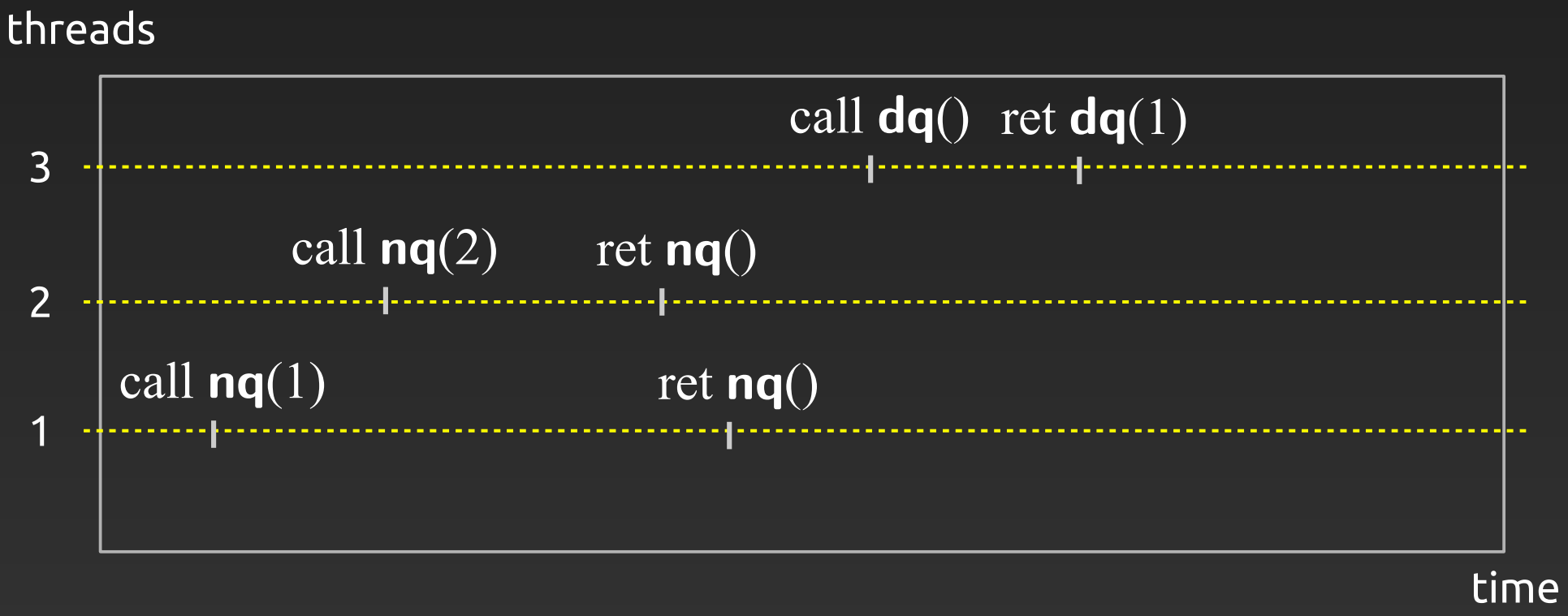


# Concurrent library behaviour

Consider a *queue* library $L$, invoked concurrently, with methods:

**enqueue:** int $\rightarrow$ void **dequeue:** void $\rightarrow$ int

A (correct) library behaviour is the following:

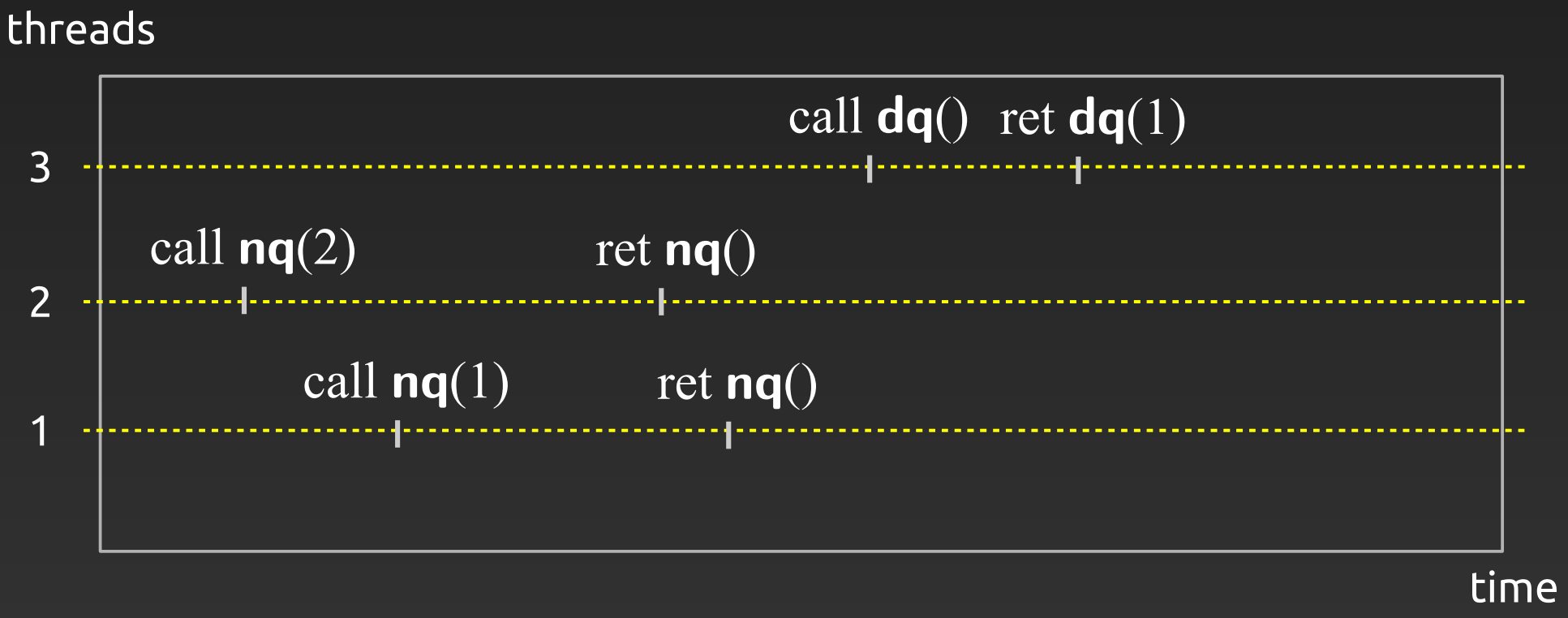


# Concurrent library behaviour

Consider a *queue* library $L$, invoked concurrently, with methods:

**enqueue:** int $\rightarrow$ void **dequeue:** void $\rightarrow$ int

A (correct) library behaviour is the following:

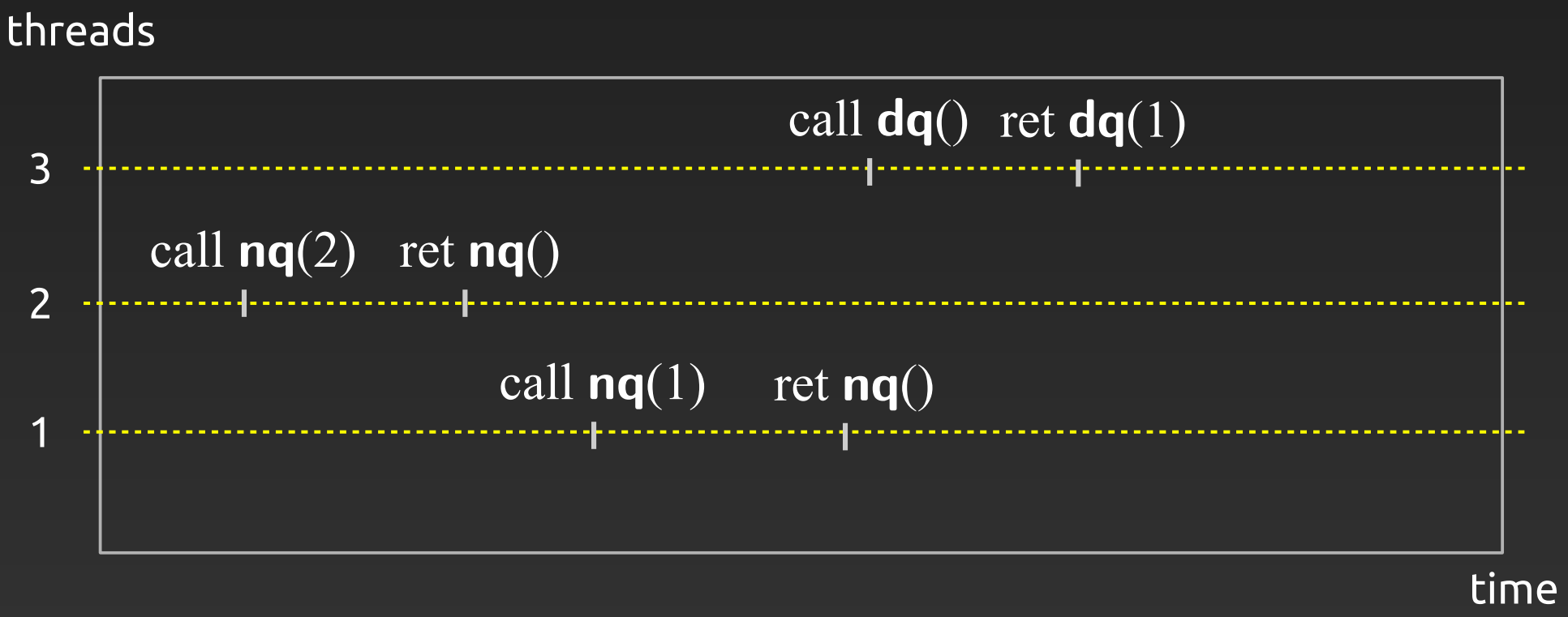


# Concurrent library behaviour

Consider a *queue* library $L$, invoked concurrently, with methods:

**enqueue:** int $\rightarrow$ void **dequeue:** void $\rightarrow$ int

A (~~correct~~) library behaviour is the following:



# Correctness formally

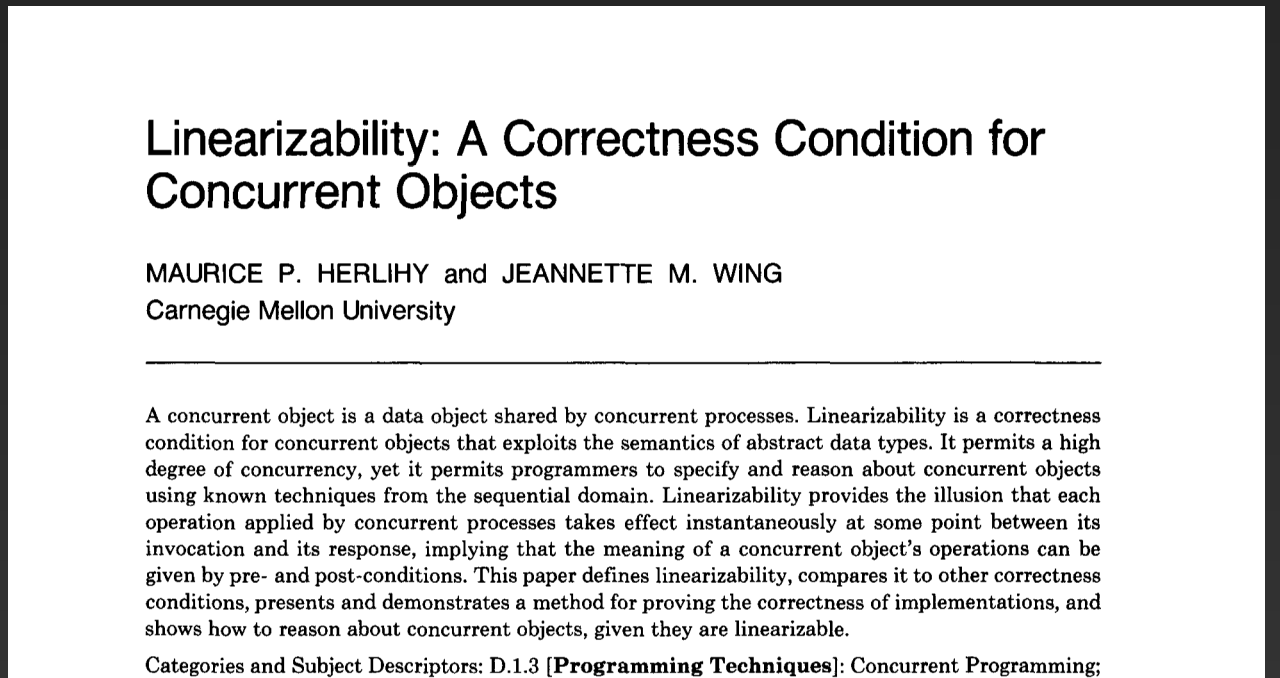
How can we characterise “correct” behaviours?

→ use linearisability:

*a behaviour is correct if it follows a given specification, modulo legal reordering of actions*

Linearisability introduced in 1990 by Herlihy and Wing

by now a standard correctness criterion



Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming;

# Linearisability formally

Concurrent setting of histories of observable actions

- Actions $a$ and $b$ can be transposed if they are in different threads and:

$$(a\ ,\ b) \text{ is } \textit{not} \text{ of the form } (\mathrm{ret}(...)\ ,\ \mathrm{call}(\ldots))$$

- History $h_1$ linearises to $h_2$ if we can get $h_2$ from $h_1$ by a series of such transpositions

# Linearisability formally

Concurrent setting of histories of observable actions

- Actions $a$ and $b$ can be transposed if they are in different threads and:

  $(a\ ,\ b)$ is *not* of the form $(\text{ret}(...)\ ,\ \text{call}(\ldots))$

- History $h_1$ linearises to $h_2$ if we can get $h_2$ from $h_1$ by a series of such transpositions

- A library $L$ linearises to some specification $A$ if every history $h_1$ of $L$ linearises into some $h_2$ in $A$.

  → $A$ is a set of sequential histories:

  $h = (t_1,\text{call}(\mathbf{m}_1,...))\ (t_1,\text{ret}(\mathbf{m}_1,...))\ (t_2,\text{call}(\mathbf{m}_2,...))\ (t_2,\text{ret}(\mathbf{m}_2,...))\ \ldots$

# Linearisability vs soundness

Linearisability can also be seen as a safety criterion:

$L$ is linearises into $L'$ $\implies$ $L$ obs. approximates $L'$

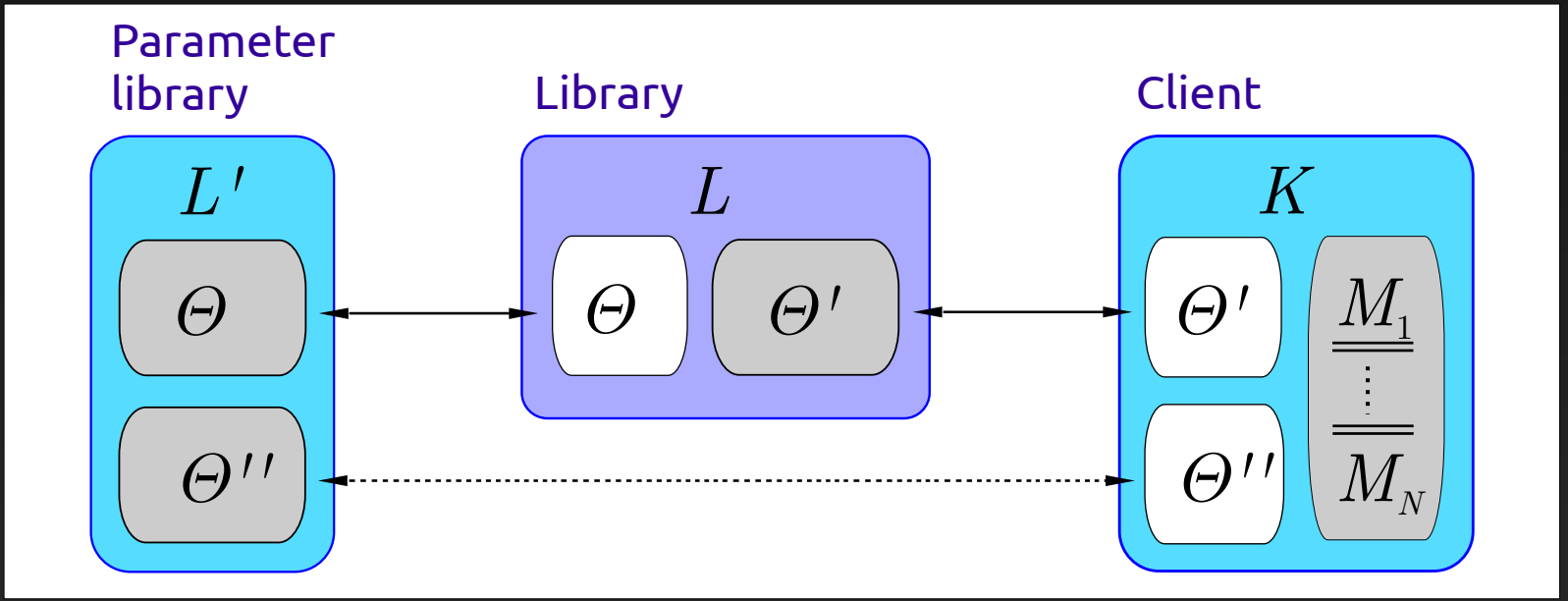
[Filipovic, O'Hearn, Rinetzky, Yang '10]

i.e:

- for any client $K$,

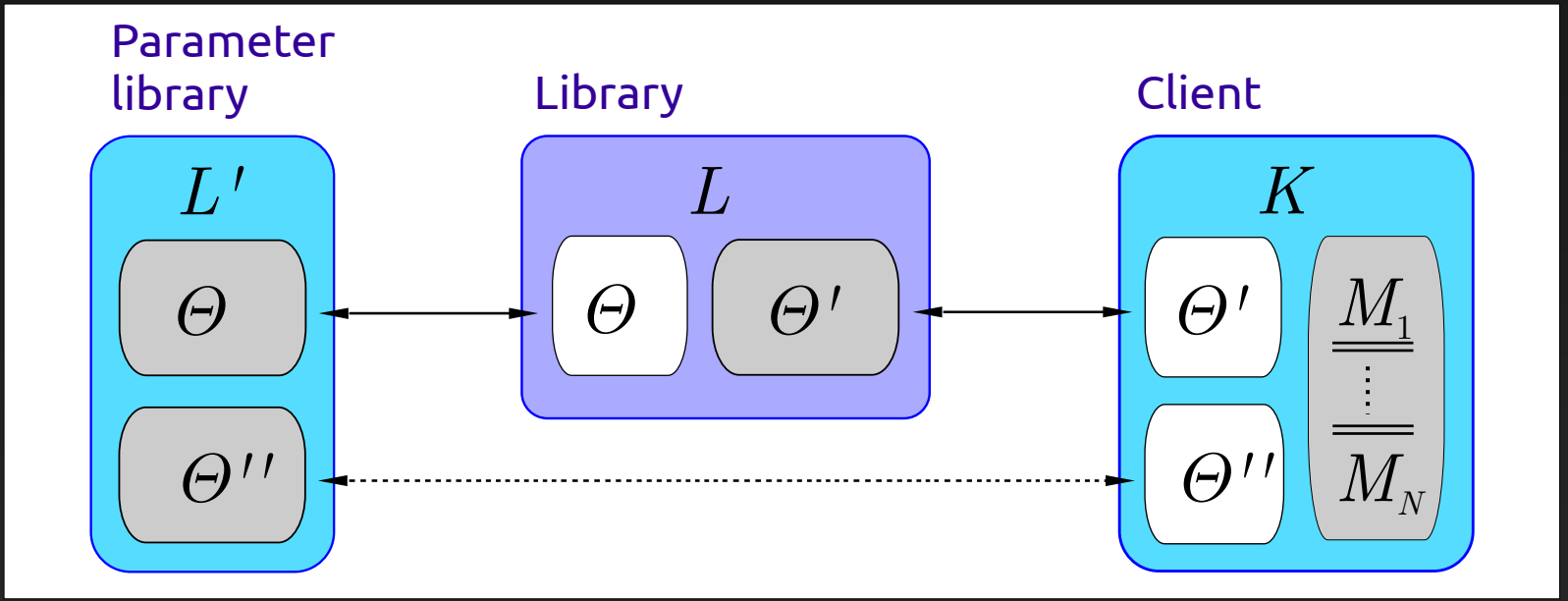
  (link $L$ in $K$) terminates $\implies$ (link $L'$ in $K$) terminates

# Higher-order libraries




- $\Theta, \Theta', \Theta''$ are sets of method names with their types
- types can be of higher order (i.e. general function types)
- $M_1, \ldots, M_N$ are terms/programs running in parallel

# Higher-order libraries




- $\Theta, \Theta', \Theta''$ are sets
- types can be of hig
- $M_1, \ldots, M_N$ are ter

So far the 1st order case studied (all methods int $\rightarrow$ int)

[Cerone, Gotsman, Yang '14]

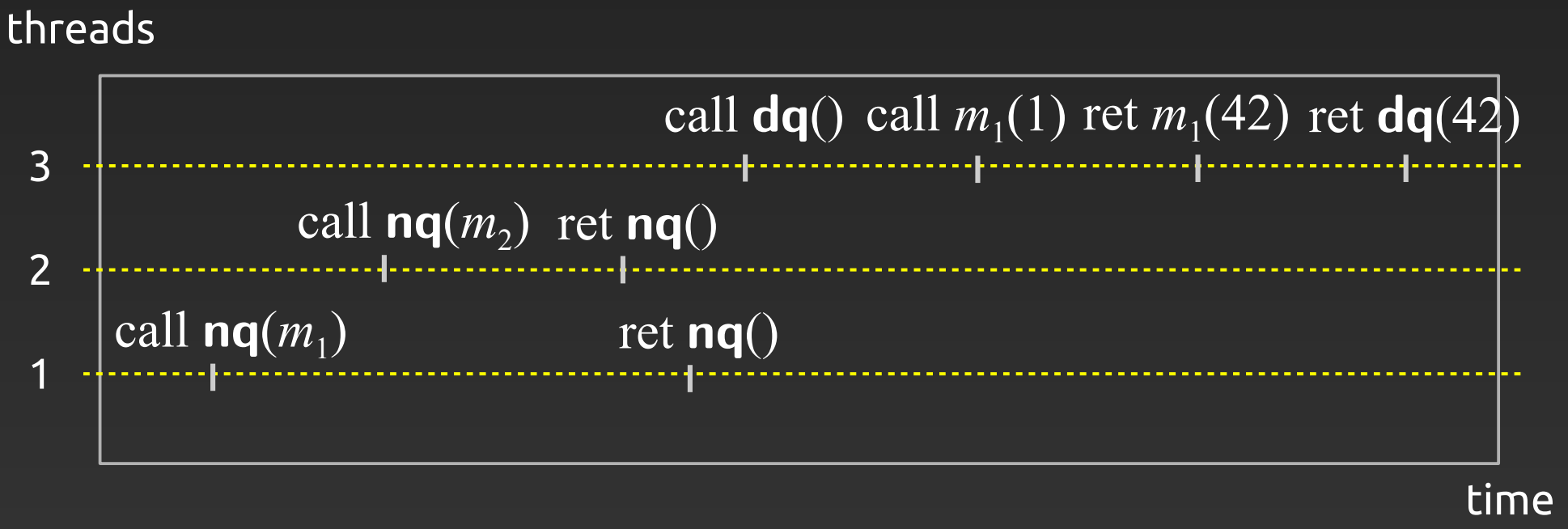
# HO queue

Suppose now $L$ is a HO queue:

**nq:** $(\text{int} \to \text{int}) \to \text{void}$ **dq:** $\text{void} \to \text{int}$

that, in each dequeue:

- it increases a counter and dequeues the first queued method
- returns the result of applying the method to the counter value



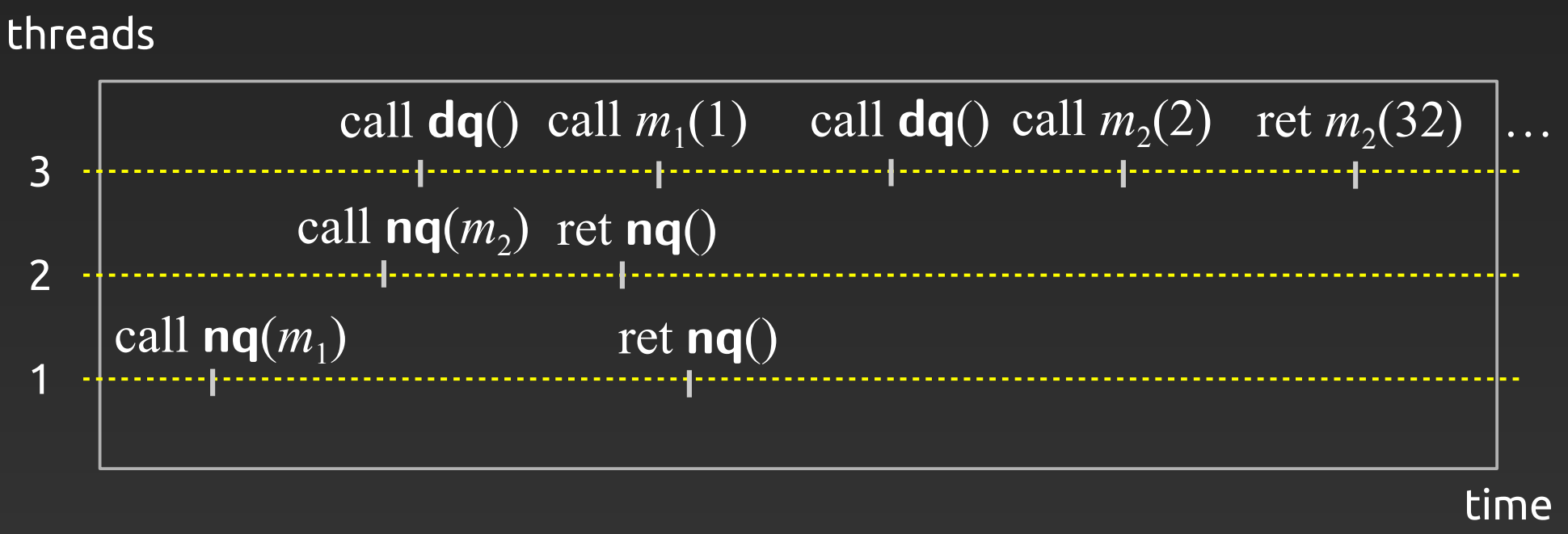
# HO queue

Suppose now $L$ is a HO queue:

**nq:** $(\text{int} \to \text{int}) \to \text{void}$ **dq:** $\text{void} \to \text{int}$

that, in each dequeue:

- it increases a counter and dequeues the first queued method
- returns the result of applying the method to the counter value



# Higher-order difficulties

The HO situation is much richer as histories are higher order:

$$h = (1,\text{call } \mathbf{m}_1(\widetilde{\widetilde{\mathbf{m}_1}},\widetilde{\widetilde{\mathbf{m}_2}}))\ (2,\text{call } \mathbf{m}_2(...))\ (2,\text{call } \widetilde{\widetilde{\mathbf{m}_1}}(...))\ (2,\text{ret } \widetilde{\widetilde{\mathbf{m}_1}}(\widetilde{\widetilde{\mathbf{m}_3}}))\ \ldots$$

- a method call need not be followed by its return
- method calls/returns can be issued by: the client, the library and the parameter library
- new methods can appear on-the-fly as arguments to calls/ret's
- sequential histories?

Approach: use game semantics

# Game semantics

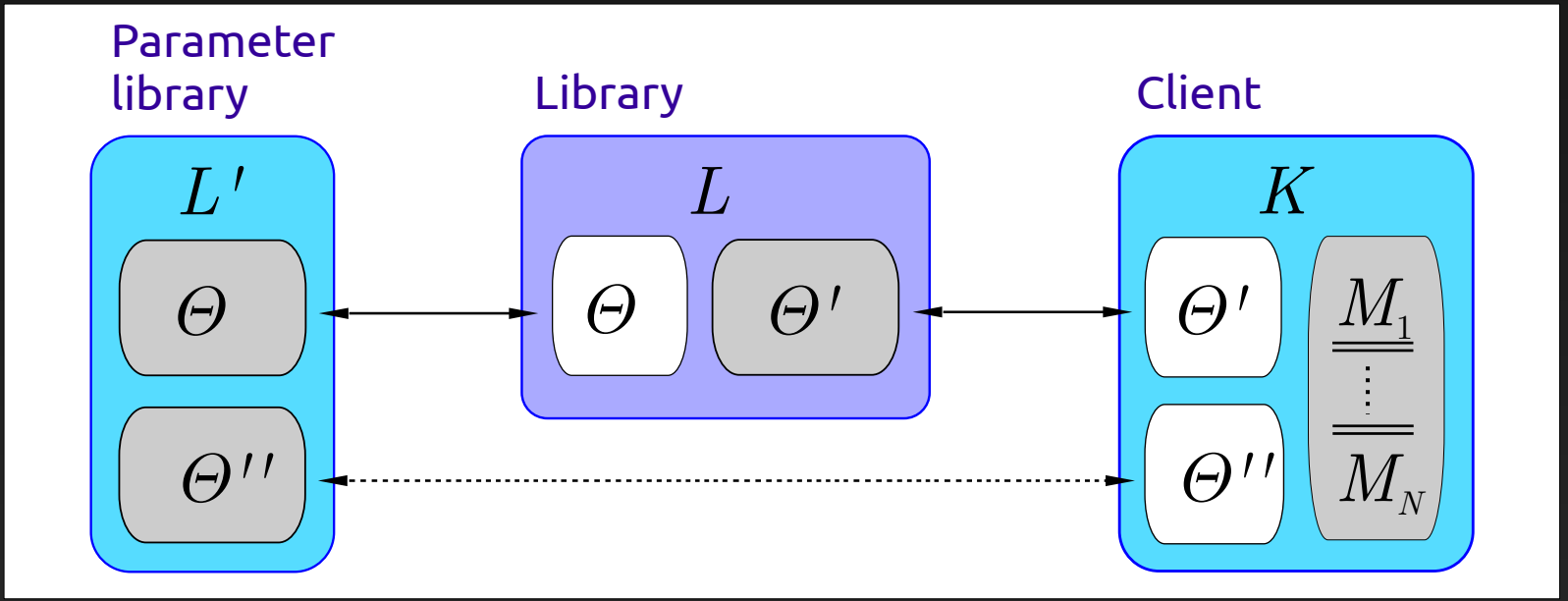
A theory assigning operational/denotational meaning to higher-order programs:

- computation modelled as a 2-player game between
  - *Opponent* (the environment), aka $O$
  - *Proponent* (the program), aka $P$
- Programs = *strategies* for $P$
- categories of games via strategy composition

Optimal level of intensionality: full abstraction

Can be formulated purely operationally

# Library/context game




link $L'; L$ in $M_1 || ... || M_N$

Here the game is played between:

- the library $L$ (*Proponent*)
- the context $(L', K)$ of $L$ (*Opponent*)

# Library syntax

$$
\begin{array}{rrl}
\textit{Libraries} & L ::= & B \mid \mathsf{abstract}\ m;\ L \mid \mathsf{public}\ m;\ L \\
\textit{Blocks} & B ::= & \epsilon \mid m = \lambda x.M;\ B \mid r := \lambda x.M;\ B \mid r := i;\ B \\
\textit{Terms} & M ::= & () \mid i \mid \mathsf{tid} \mid x \mid m \mid M \oplus M \mid \mathsf{cond}\, M M M \mid \langle M, M \rangle \mid \pi_i M \\
 & & \mid \lambda x^{\theta}.M \mid xM \mid mM \mid \mathsf{let}\, x = M\, \mathsf{in}\, M \mid r := M \mid\, !r \\
\textit{Clients} & K ::= & M \parallel \cdots \parallel M \\
\textit{Types} & \theta ::= & \mathsf{unit} \mid \mathsf{int} \mid \theta \times \theta \mid \theta \to \theta
\end{array}
$$

# Higher-order histories

A history over $\Theta \to \Theta'$ is a sequence of actions

$$h = (t_1, a_1)\ (t_2, a_2)\ \ldots$$

where each action is a *tagged* call or return

$$a_i \in \{\ \text{call}\ m(v)_X, \text{ret}\ m(v)_X \mid m \in \text{Meths}, v \in \text{Vals}, X \in \{O,P\}\}$$

# Higher-order histories

A history over $\Theta \to \Theta'$ is a sequence of actions

$$h = (t_1, a_1)\ (t_2, a_2)\ \ldots$$

where each action is a *tagged* call or return

$$a_i \in \{\ \text{call}\ m(v)_X, \text{ret}\ m(v)_X \mid m \in \text{Meths}, v \in \text{Vals}, X \in \{O,P\}\}$$

and such that, in each thread $t_i$:

- calls and returns are well matched (i.e. bracketed)
- $O$ and $P$ actions alternate
- every method called must be first introduced (or be in $\Theta, \Theta'$)
- $P$ only calls $O$-methods, and returns from $P$-methods
- $O$ only calls $P$-methods, and returns from $O$-methods

# History example

$L : \Theta \rightarrow \Theta', \quad \Theta = \emptyset$
$\Theta' = \{$ **nq:** int $\rightarrow$ void, **dq:** void $\rightarrow$ int $\}$

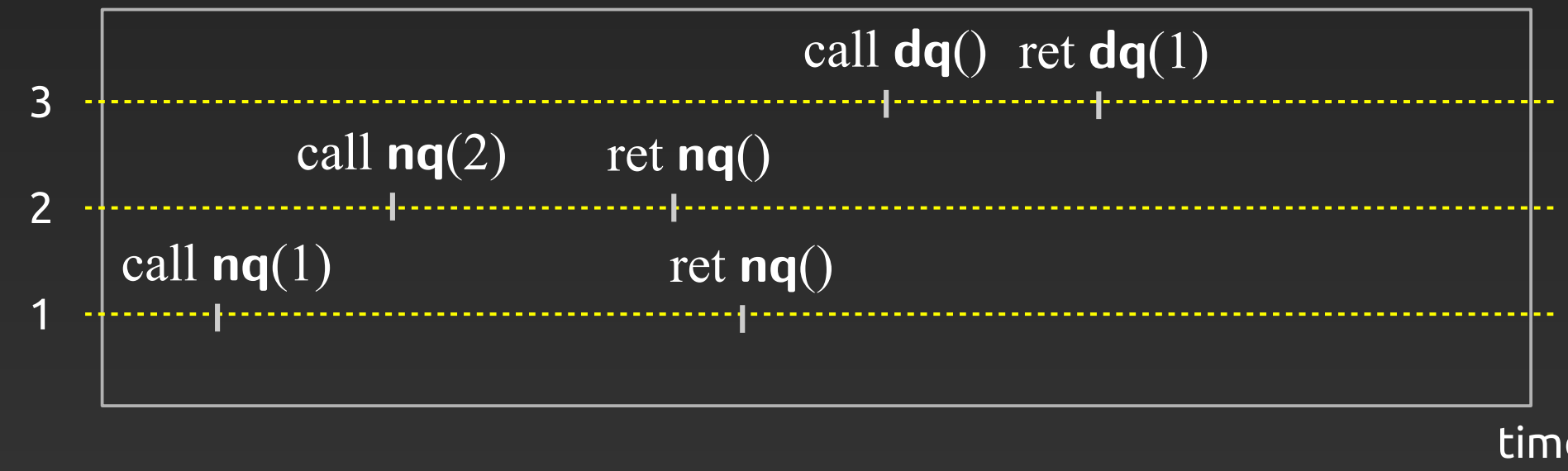
Recall the queue $L$ with methods:

**nq:** int $\rightarrow$ void

**dq:** void $\rightarrow$ int

a possible history is the following:

$$(1, \text{call } \mathbf{nq}(1)_O)\ (2, \text{call } \mathbf{nq}(2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_0)\ (3, \text{ret } \mathbf{dq}(1)_P)$$



# HO history example

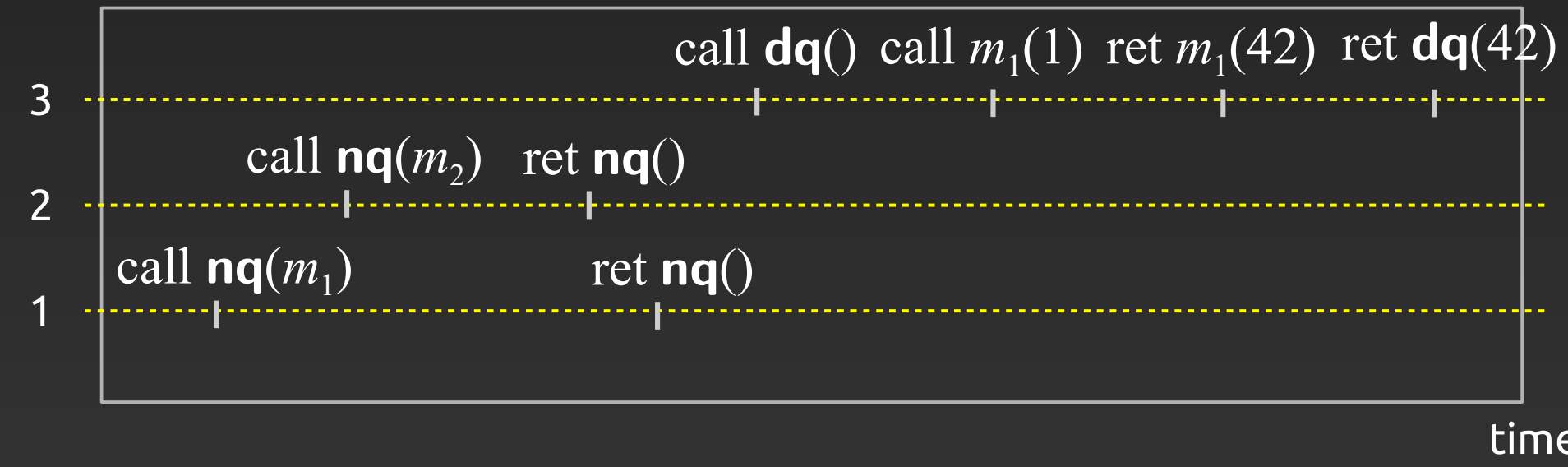
$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** (int→int)→void, **dq:** void→int $\}$

Suppose now $L$ is a HO queue:

**nq:** $(\text{int} \to \text{int}) \to \text{void}$ **dq:** $\text{void} \to \text{int}$

a possible history is:

$$(1, \text{call } \mathbf{nq}(m_1)_O)\ (2, \text{call } \mathbf{nq}(m_2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_O)\ (3, \text{call } m_1(1)_P)\ (3, \text{ret } m_1(42)_O)\ (3, \text{ret } \mathbf{dq}(42)_P)$$



# HO history example

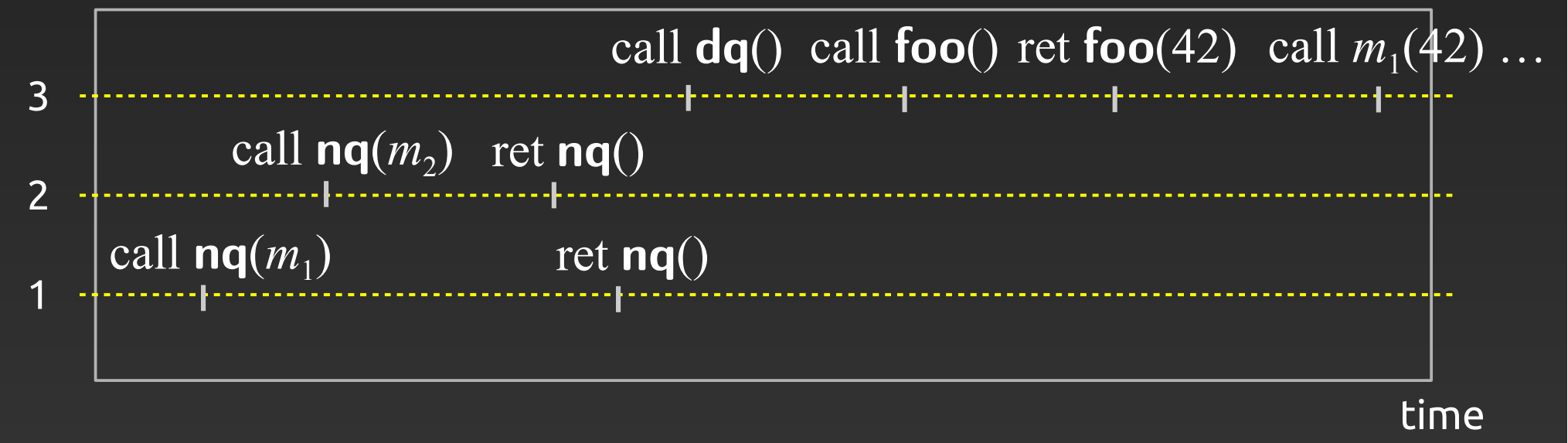
$L : \Theta \to \Theta'$, $\Theta$ = { **foo:** void→int }
$\Theta'$ = { **nq:** (int→int)→void, **dq:** void→int }

Suppose now $L$ is a HO queue:

**nq:** (int $\to$ int) $\to$ void  **dq:** void $\to$ int  **foo:** void $\to$ int

a possible history is:

(1, call **nq**$(m_1)_O$) (2, call **nq**$(m_2)_O$) (2, ret **nq**$()_P$) (1, ret **nq**$()_P$)
(3, call **dq**$()_O$) (3, call **foo**$()_P$) (3, ret **foo**$(42)_O$) (3, call $m_1(42)_P$)
(3, ret $m_1(32)_O$) (3, ret **dq**$(42)_P$)



# Higher-order linearisability

Action transposition is now the relation $\lhd_{PO}$ defined by:

$$(t, a_{(O)})\,(t', a') \;\lhd_{PO}\; (t', a')\,(t, a_{(O)})$$

$$(t', a')\,(t, a_{(P)}) \;\lhd_{PO}\; (t, a_{(P)})\,(t', a')$$

**Def:** History $h_1$ linearises to $h_2$ if we can get $h_2$ from $h_1$ by a series of $\lhd_{PO}$-transpositions.

A library $L$ linearises to some specification $A$ if every history $h_1$ of $L$ linearises into some $h_2$ in $A$.
The specification $A$ is a set of sequential histories:

$$h = (t_1, a_{1(O)})\,(t_1, a_{2(P)})\,(t_2, a_{3(O)})\,(t_2, a_{4(P)})\;\ldots$$

# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** int $\to$ void, **dq:** void $\to$ int $\}$

Recall the queue $L$ with methods:

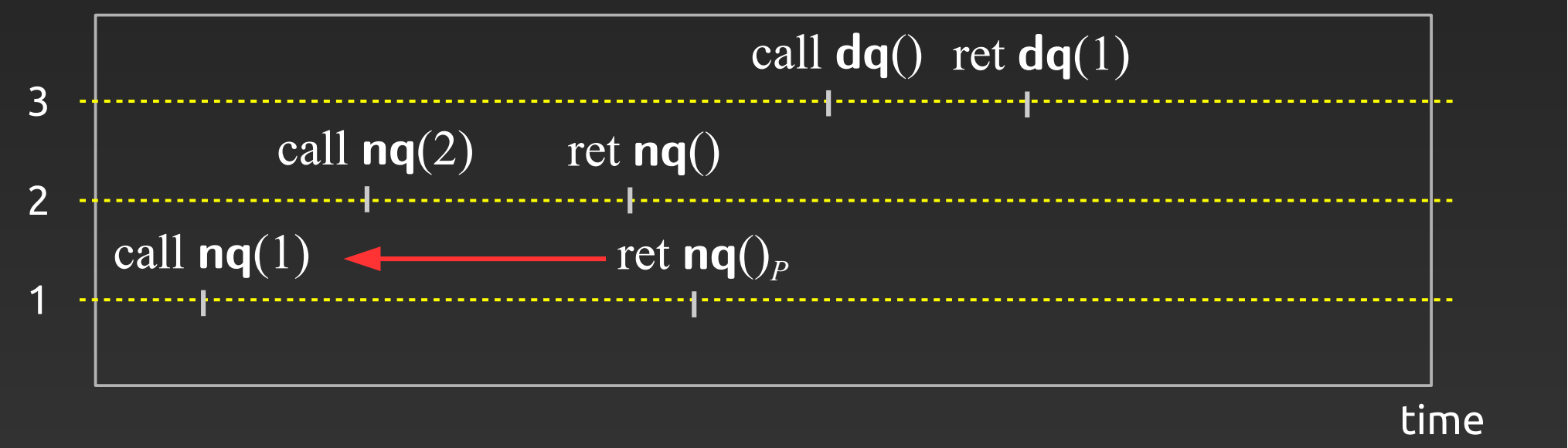
**nq:** int $\to$ void

**dq:** void $\to$ int

a possible history is the following:

$$(1, \text{call } \mathbf{nq}(1)_O)\ (2, \text{call } \mathbf{nq}(2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_0)\ (3, \text{ret } \mathbf{dq}(1)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** int $\to$ void, **dq:** void $\to$ int $\}$

Recall the queue $L$ with methods:

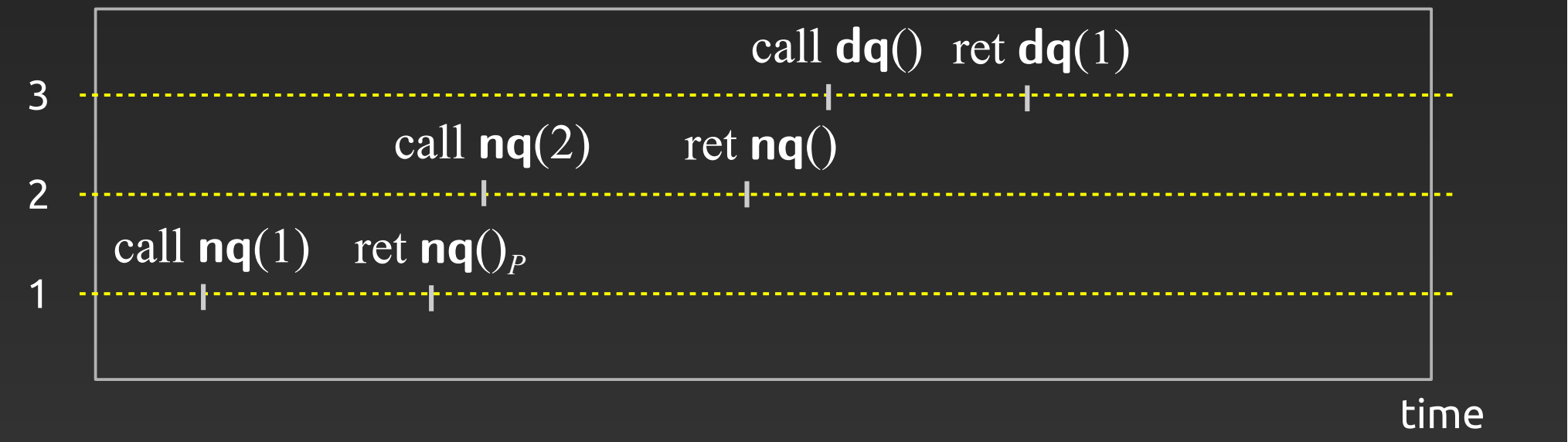
**nq:** int $\to$ void

**dq:** void $\to$ int

a possible history is the following:

$$(1, \text{call } \mathbf{nq}(1)_O)\ (1, \text{ret } \mathbf{nq}()_P)\ (2, \text{call } \mathbf{nq}(2)_O)\ (2, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_0)\ (3, \text{ret } \mathbf{dq}(1)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** int $\to$ void, **dq:** void $\to$ int $\}$

Recall the queue $L$ with methods:

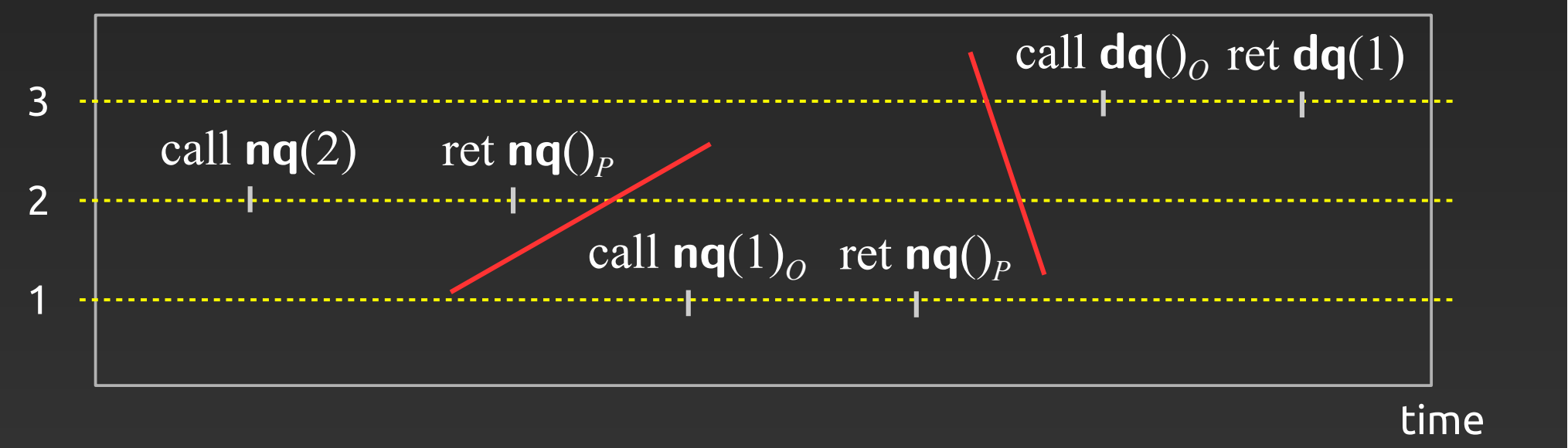
**nq:** int $\to$ void

**dq:** void $\to$ int

a possible (bad) history is the following:

$$(2, \text{call } \mathbf{nq}(2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{call } \mathbf{nq}(1)_O)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_0)\ (3, \text{ret } \mathbf{dq}(1)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** (int→int)→void, **dq:** void→int $\}$

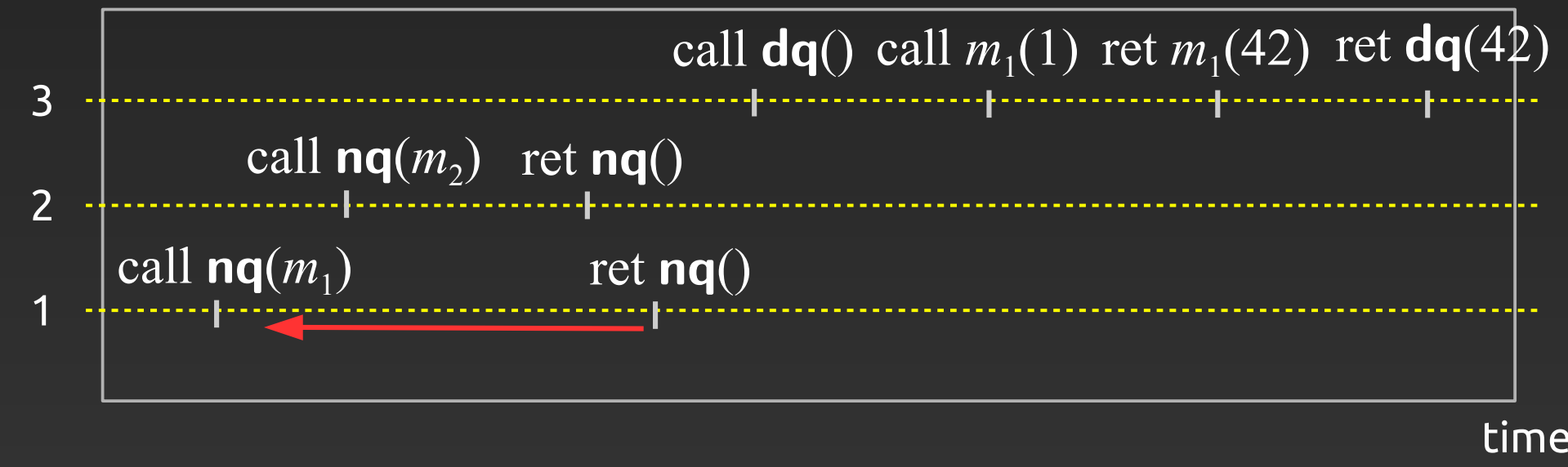
Suppose now $L$ is a HO queue:

**nq:** $(\text{int} \to \text{int}) \to \text{void}$ **dq:** $\text{void} \to \text{int}$

a possible history is:

$$(1, \text{call } \mathbf{nq}(m_1)_O)\ (2, \text{call } \mathbf{nq}(m_2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_O)\ (3, \text{call } m_1(1)_P)\ (3, \text{ret } m_1(42)_O)\ (3, \text{ret } \mathbf{dq}(42)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** (int→int)→void, **dq:** void→int $\}$

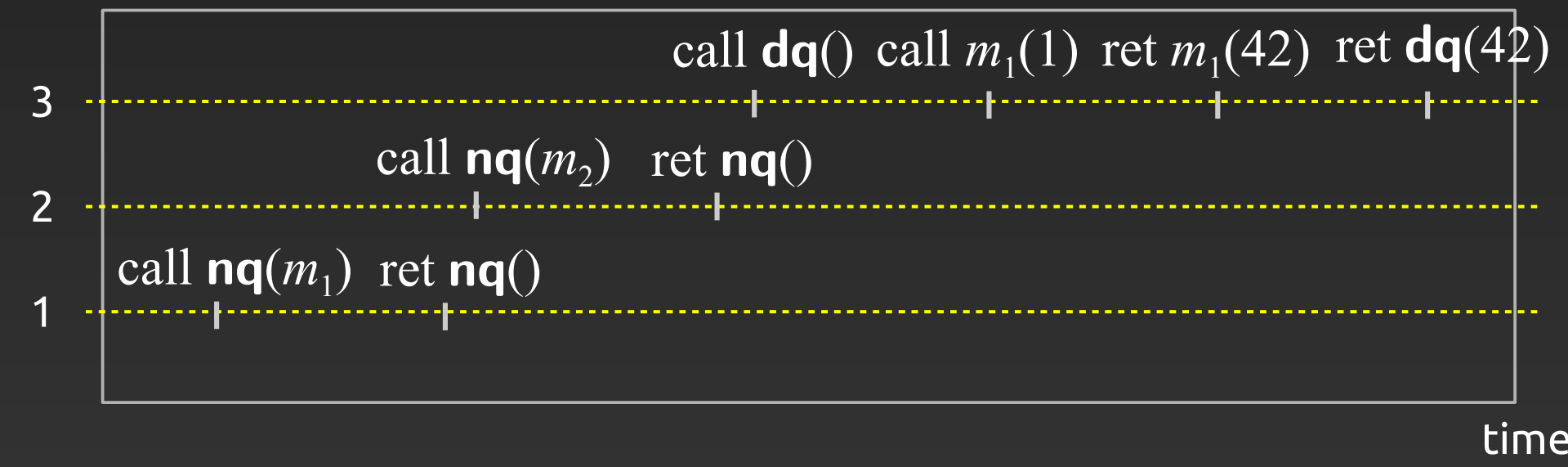
Suppose now $L$ is a HO queue:

**nq:** $(\text{int} \to \text{int}) \to \text{void}$ **dq:** $\text{void} \to \text{int}$

a possible history is:

$$(1, \text{call } \mathbf{nq}(m_1)_O)\ (1, \text{ret } \mathbf{nq}()_P)\ (2, \text{call } \mathbf{nq}(m_2)_O)\ (2, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_O)\ (3, \text{call } m_1(1)_P)\ (3, \text{ret } m_1(42)_O)\ (3, \text{ret } \mathbf{dq}(42)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** (int→int)→void, **dq:** void→int $\}$

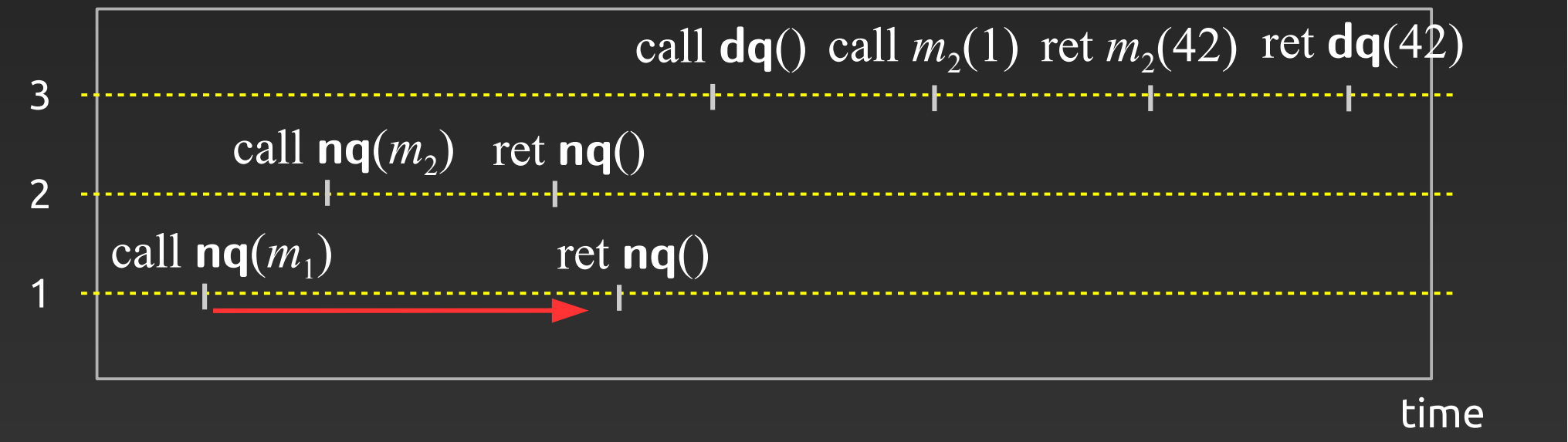
Suppose now $L$ is a HO queue:

**nq:** (int $\to$ int) $\to$ void  **dq:** void $\to$ int

a possible history is:

$$(1, \text{call } \mathbf{nq}(m_1)_O)\ (2, \text{call } \mathbf{nq}(m_2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_O)\ (3, \text{call } m_2(1)_P)\ (3, \text{ret } m_2(42)_O)\ (3, \text{ret } \mathbf{dq}(42)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** (int→int)→void, **dq:** void→int $\}$

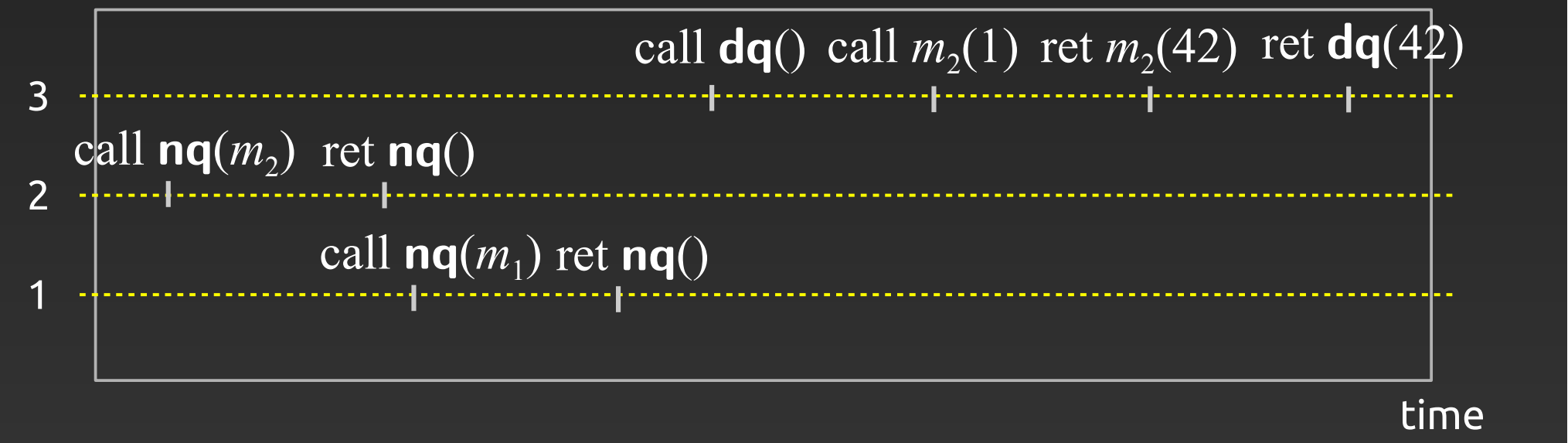
Suppose now $L$ is a HO queue:

**nq:** $(\text{int} \to \text{int}) \to \text{void}$ **dq:** $\text{void} \to \text{int}$

a possible history is:

$$(2, \text{call } \mathbf{nq}(m_2)_O)\ (2, \text{ret } \mathbf{nq}()_P)\ (1, \text{call } \mathbf{nq}(m_1)_O)\ (1, \text{ret } \mathbf{nq}()_P)$$
$$(3, \text{call } \mathbf{dq}()_O)\ (3, \text{call } m_2(1)_P)\ (3, \text{ret } m_2(42)_O)\ (3, \text{ret } \mathbf{dq}(42)_P)$$

$\lhd_{PO}$



# Example revisited

$L : \Theta \to \Theta',\ \ \Theta = \emptyset$
$\Theta' = \{$ **nq:** (int→int)→void, **dq:** void→int $\}$

Suppose now $L$ is a HO queue:

**nq:** (int $\to$ int) $\to$ void  **dq:** void $\to$ int

a *sequential* history specifying the queue is of the form:

$(t_1, \text{call } \mathbf{nq}(m_1)_O)\ (t_1, \text{ret } \mathbf{nq}()_P)\ \ldots\ (t_i, \text{call } \mathbf{nq}(m_i)_O)\ (t_i, \text{ret } \mathbf{nq}()_P)$

$(t_1', \text{call } \mathbf{dq}()_O)\ (t_1', \text{call } m_1(1)_P)$

$(t_{i+1}, \text{call } \mathbf{nq}(m_{i+1})_O)\ (t_{i+1}, \text{ret } \mathbf{nq}()_P)\ \ldots\ (t_j, \text{call } \mathbf{nq}(m_j)_O)\ (t_j, \text{ret } \mathbf{nq}()_P)$

$(t_2', \text{call } \mathbf{dq}()_O)\ (t_2', \text{call } m_2(2)_P)$

$\lhd_{PO}$

$\ldots$

$(t_2', \text{ret } m_2(x_2)_O)\ (t_2', \text{ret } \mathbf{dq}(x_2)_P)$

$\ldots$

$(t_1', \text{ret } m_1(x_1)_O)\ (t_1', \text{ret } \mathbf{dq}(x_1)_P)$

# Soundness

$L_1$ is linearises into $L_2$ $\implies$ $L_1$ obs. approximates $L_2$

i.e. for all $(L', K)$:

(link $L'; L_1$ in $K$) terminates $\implies$ (link $L'; L_2$ in $K$) terminates

We use a compositionality argument:

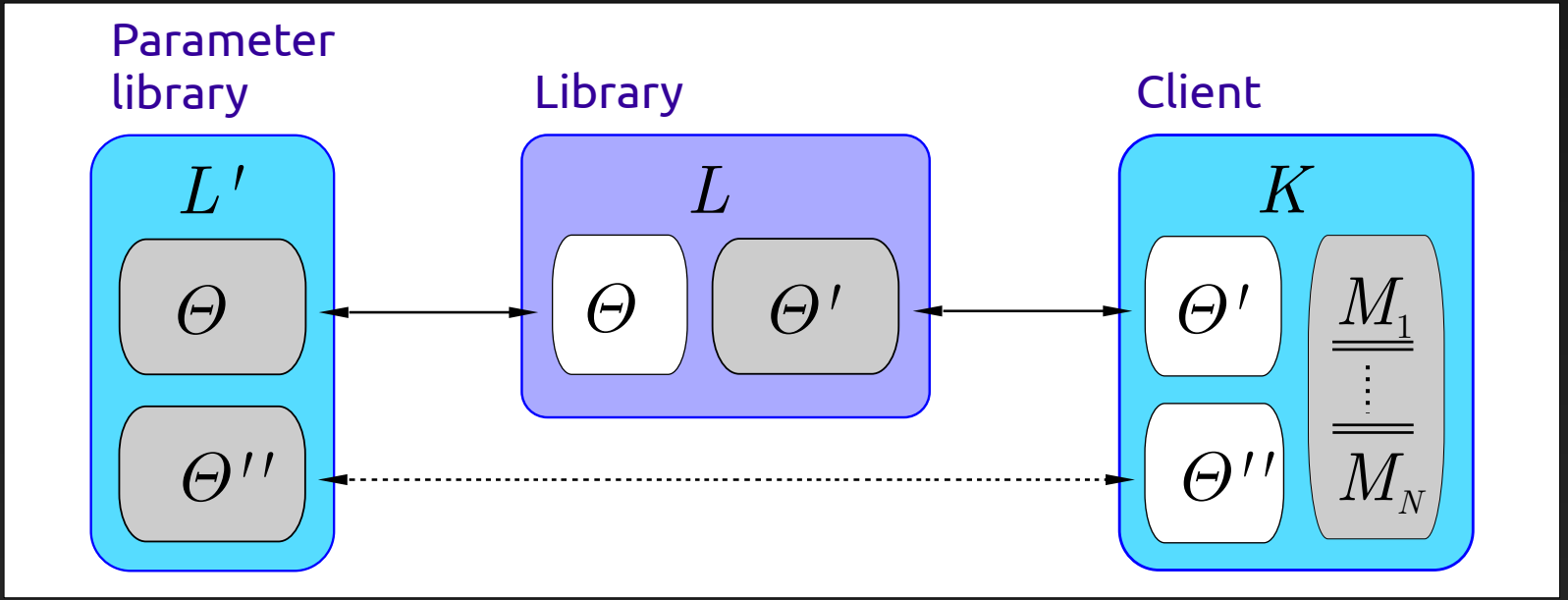
- we define a semantics $[\![\_]\!]$ for libraries and contexts (*operational game semantics*)
- show that $[\![$ link $L'; L$ in $K ]\!] = [\![ (L', K) ]\!]; [\![ L ]\!]$
- show that the semantics $[\![\_]\!]$ is closed under the dual of $\lhd_{PO}$

→ cf. with conditions for game semantics of concurrency

[Ghica, Murawski '08]

# Encapsulation

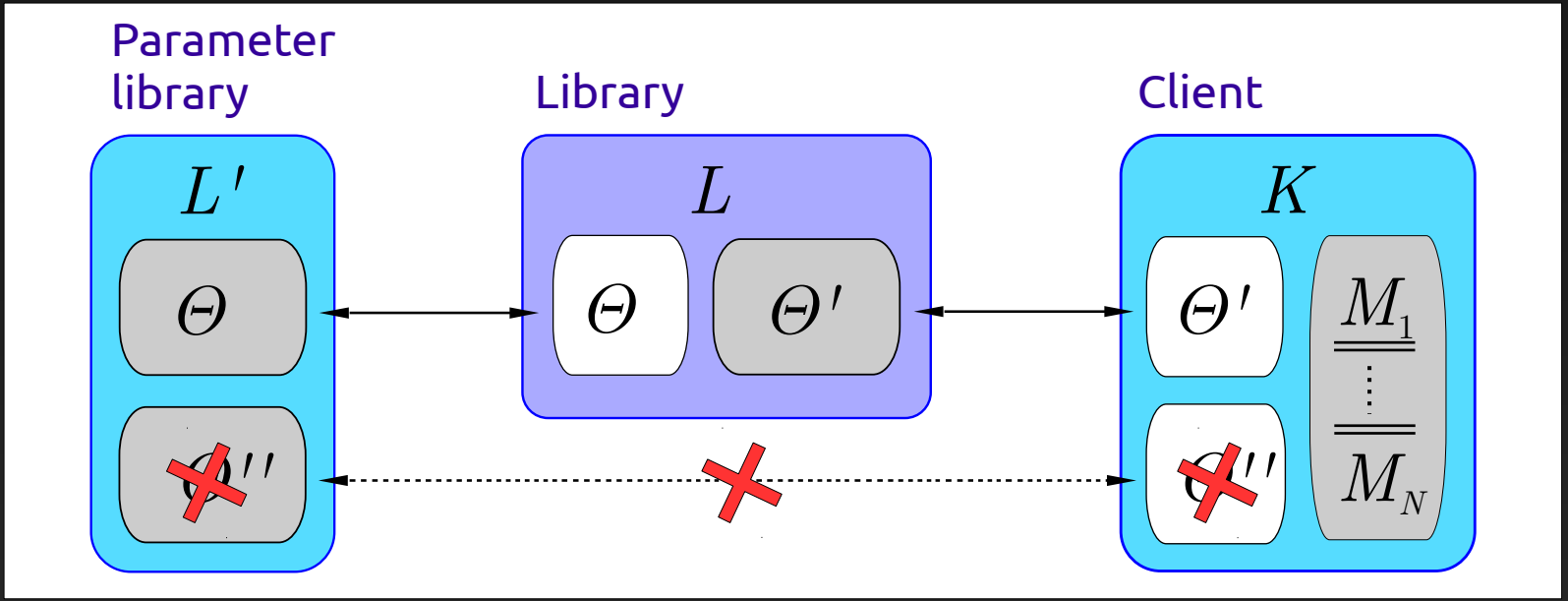
[Cerone, Gotsman, Yang '14]




- $\Theta, \Theta', \Theta''$ are sets of method names with their types
- types can be of higher order (i.e. general function types)
- $M_1, \ldots, M_N$ are terms/programs running in parallel

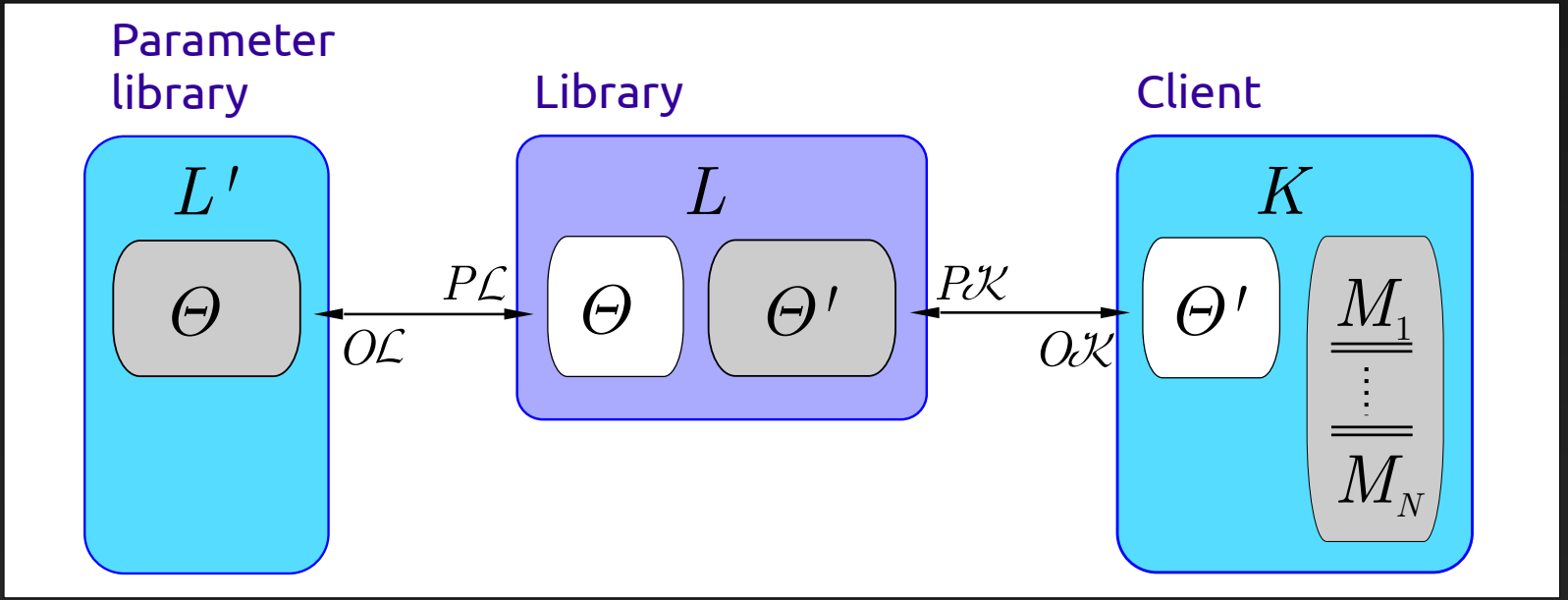
# Encapsulation

[Cerone, Gotsman, Yang '14]




- $\Theta, \Theta'$ are sets of method names with their types
- types can be of higher order (i.e. general function types)
- $M_1, \ldots, M_N$ are terms/programs running in parallel

# Encapsulation: separating Opponent




- $\Theta, \Theta'$ are sets of method names with their types
- types can be of higher order (i.e. general function types)
- $M_1, \ldots, M_N$ are terms/programs running in parallel
- We separate Opponent into $O\mathcal{K}$ and $O\mathcal{L}$ (dually for Proponent)

# Encapsulated linearisability

We tag actions wrt the part they appear in ($\mathcal{K}/\mathcal{L}$):

$$a \in \{\, \text{call } m(v)_{XY}, \text{ret } m(v)_{XY} \mid XY \in \{O\mathcal{K}, P\mathcal{K}, O\mathcal{L}, P\mathcal{L}\}\}$$

Action transposition is now the relation $\lhd_{PO} \cup \diamond$ where:

$$(t, a_{(XY)})\,(t', a'_{(X'Y')}) \quad \diamond \quad (t', a'_{(X'Y')})\,(t, a_{(XY)})$$

whenever $t \neq t'$ and $Y \neq Y'$.

**Def:** History $h_1$ enc-linearises to $h_2$ if we can get $h_2$ from $h_1$ by a series of $(\lhd_{PO} \cup \diamond)$-transpositions.

**Thm**: Enc-linearisability is sound for encapsulated contexts

# Concluding

We propose a natural extension of linearisability that captures higher-order libraries.

We prove it is sound and compositional

[ paper at `http://arxiv.org/abs/1610.07965` ]

Further on:

- weaker notions, allowing for richer orders between histories (cf. Radha's talk)
- shared state and weak memory behaviours