Linearizability with ownership transfer

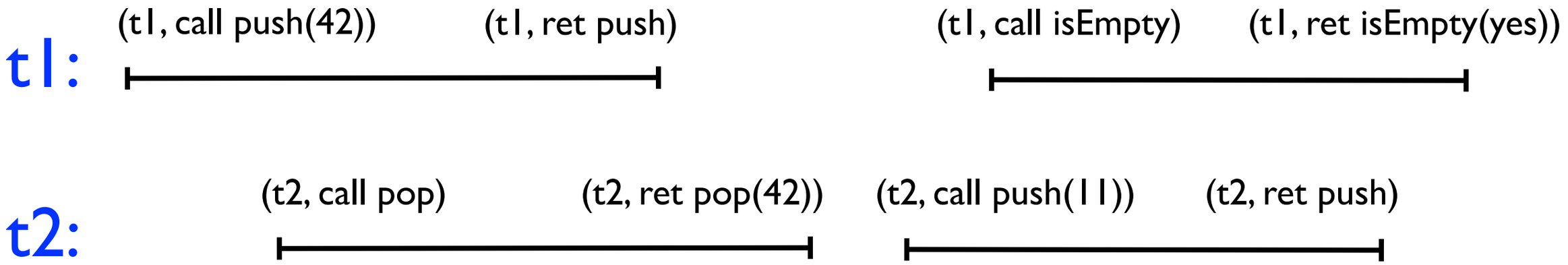
Hongseok Yang
University of Oxford

Joint work with Alexey Gotsman
(IMDEA Software Institute, Spain)

Concurrent libraries

- Encapsulate efficient concurrent algorithms.
  - Java: java.util.concurrent
  - C++: Threading Building Blocks
  - C#: System.Collections.Concurrent
- Implement stacks, queues, skip lists, hash tables, etc.
- But it is not easy to understand why they are correct.

Non-blocking stack:

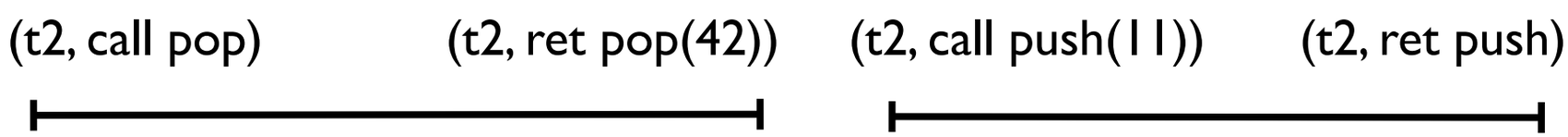


Non-blocking stack:

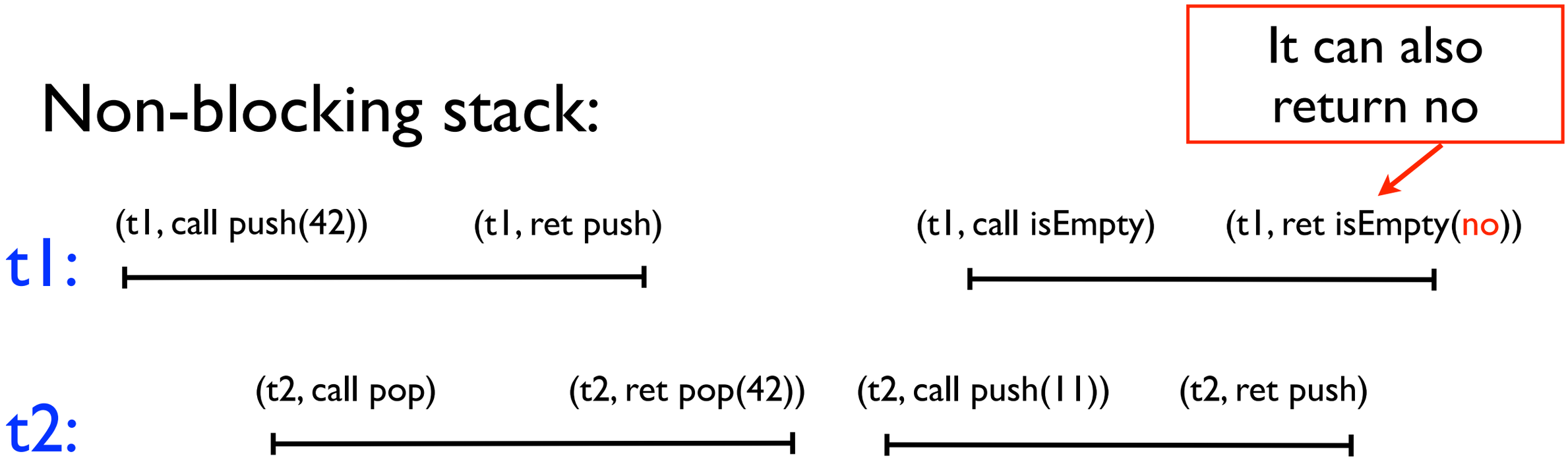
Why is it OK to return 'empty' here?

t1:

t2:



Non-blocking stack:



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; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

1. INTRODUCTION

1.1 Overview

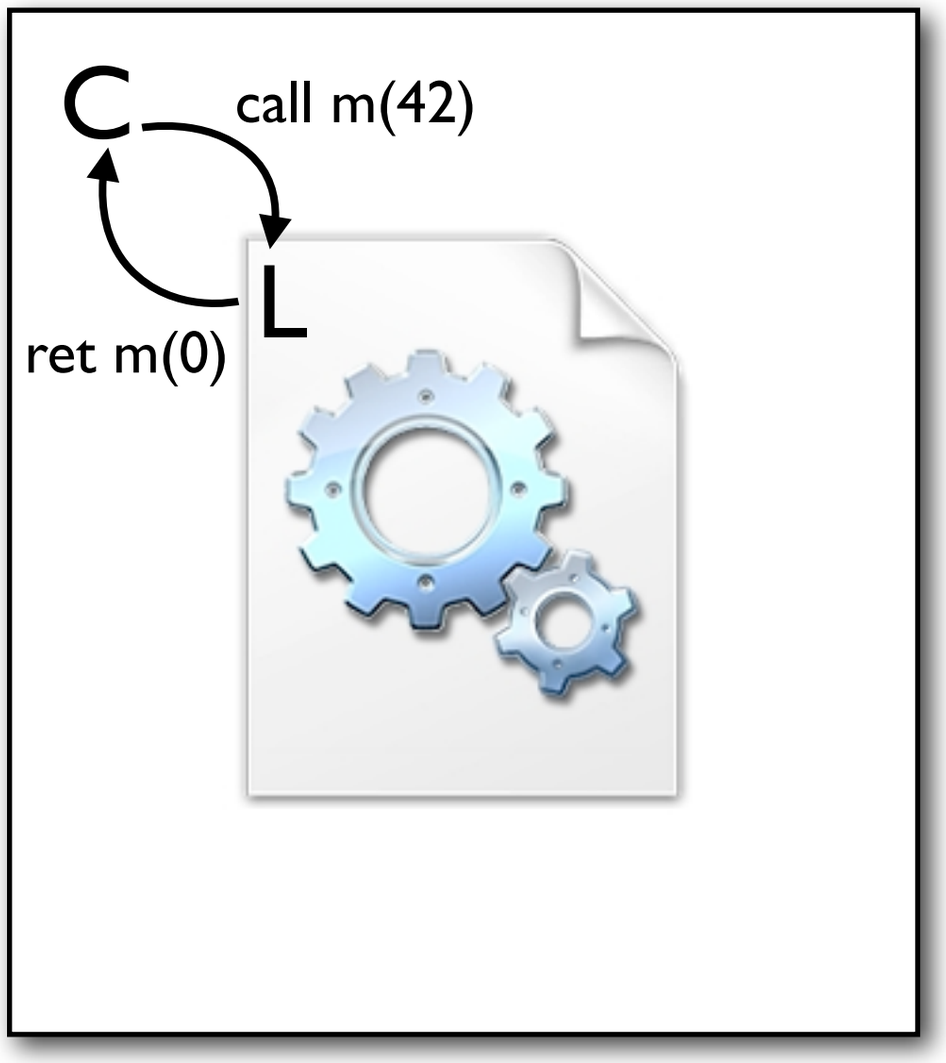
Informally, a concurrent system consists of a collection of sequential processes that communicate through shared typed objects. This model encompasses both message-passing architectures in which the shared objects are message queues,

A preliminary version of this paper appeared in the Proceedings of the 14th ACM Symposium on Principles of Programming Languages, January 1987 [21].
This research was sponsored by IBM and the Defense Advanced Research Projects Agents (DOD), ARPA order 4976 (Amendment 20), under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. Additional spport for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.
Authors' address: Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

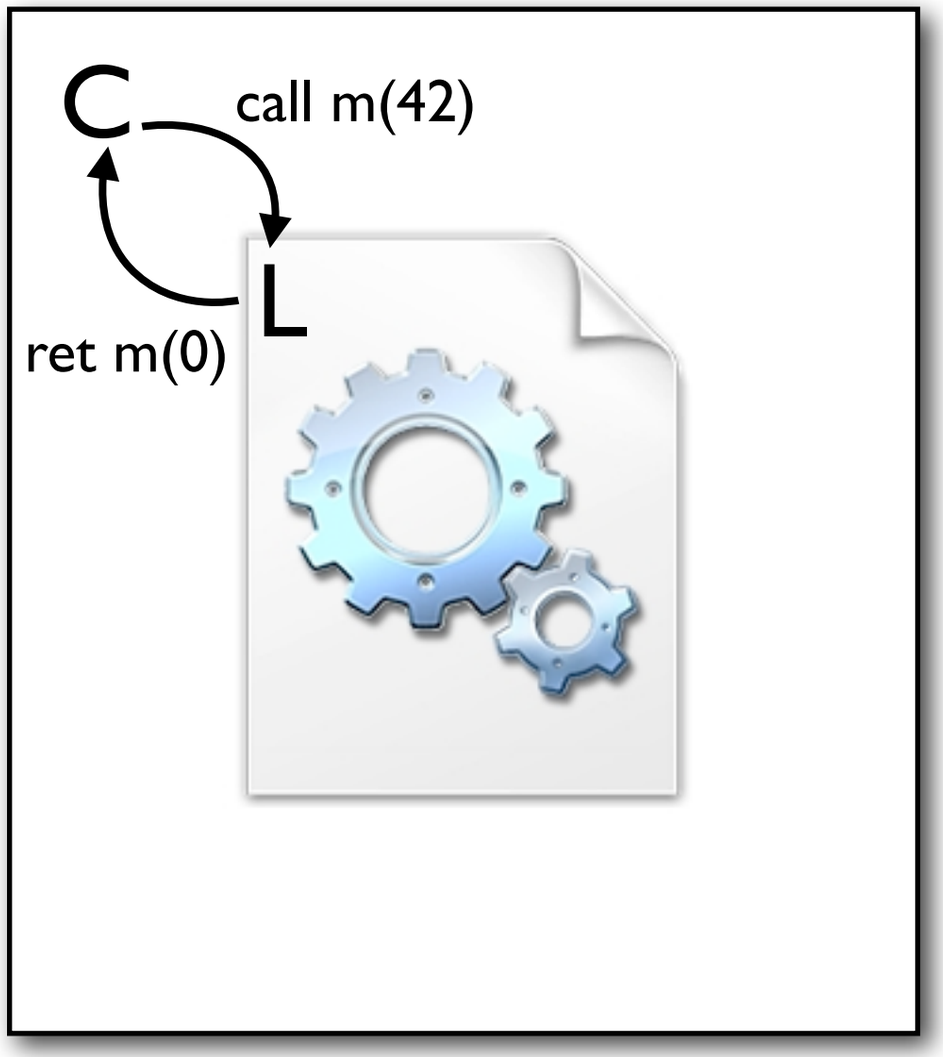
- A binary relation L ⊑ L' on libraries L and L'.
- Usually, L is an impl. and L' is a spec.

Problem

Shared address space

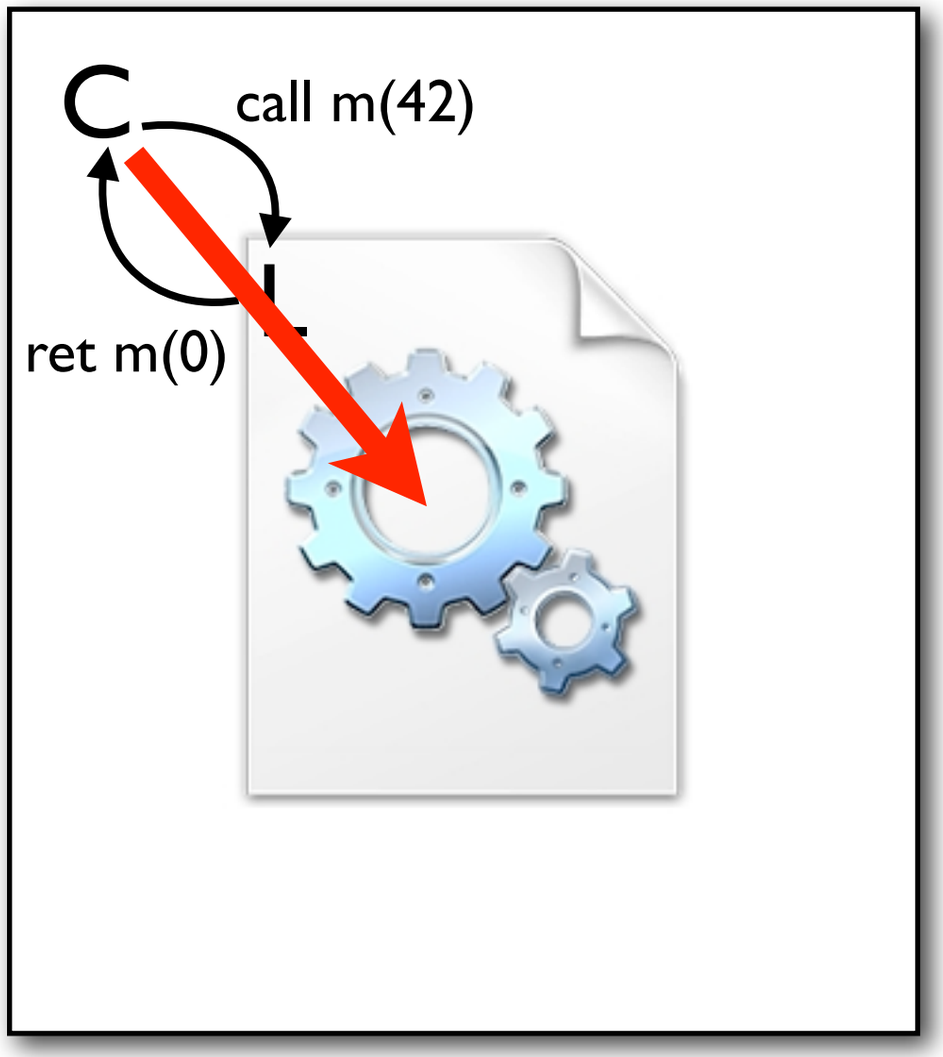


Shared address space



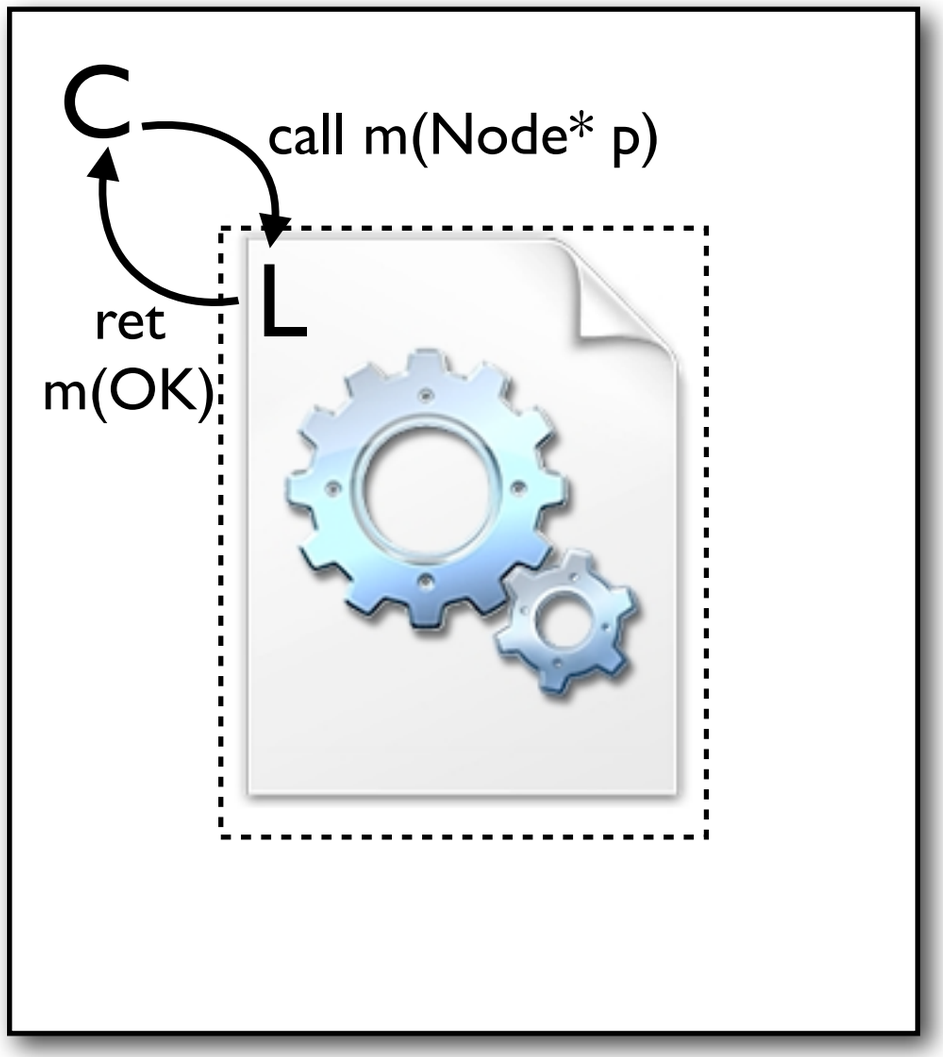
- Linearizability assumes a complete isolation between the library and its client.

Shared address space



- Linearizability assumes a complete isolation between the library and its client.
- If client and library share address space, they can corrupt each other.

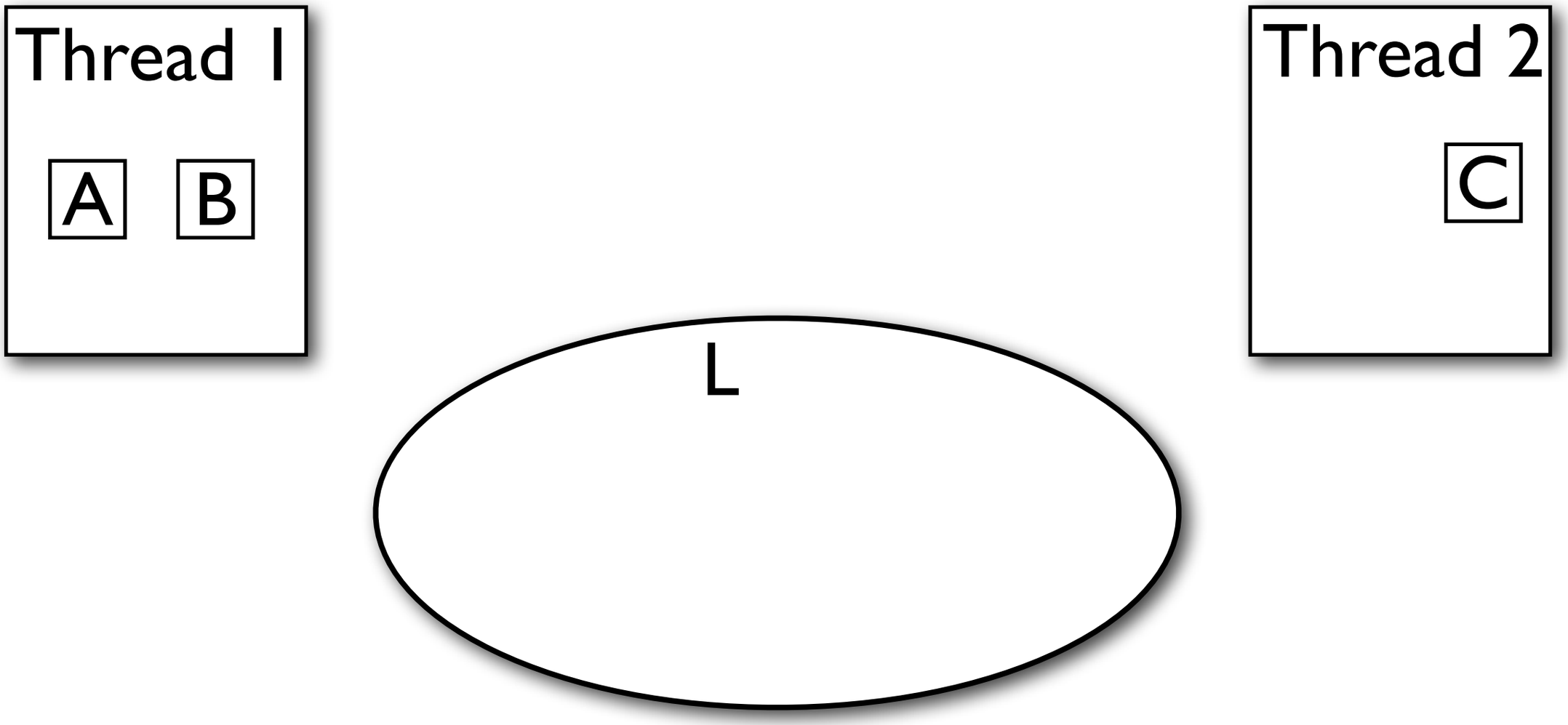
Ownership transfer



- Boundary between data structures owned by the library and its client is not static.
- Ownership = right to access.
- Ownership of cells transferred between library and client.

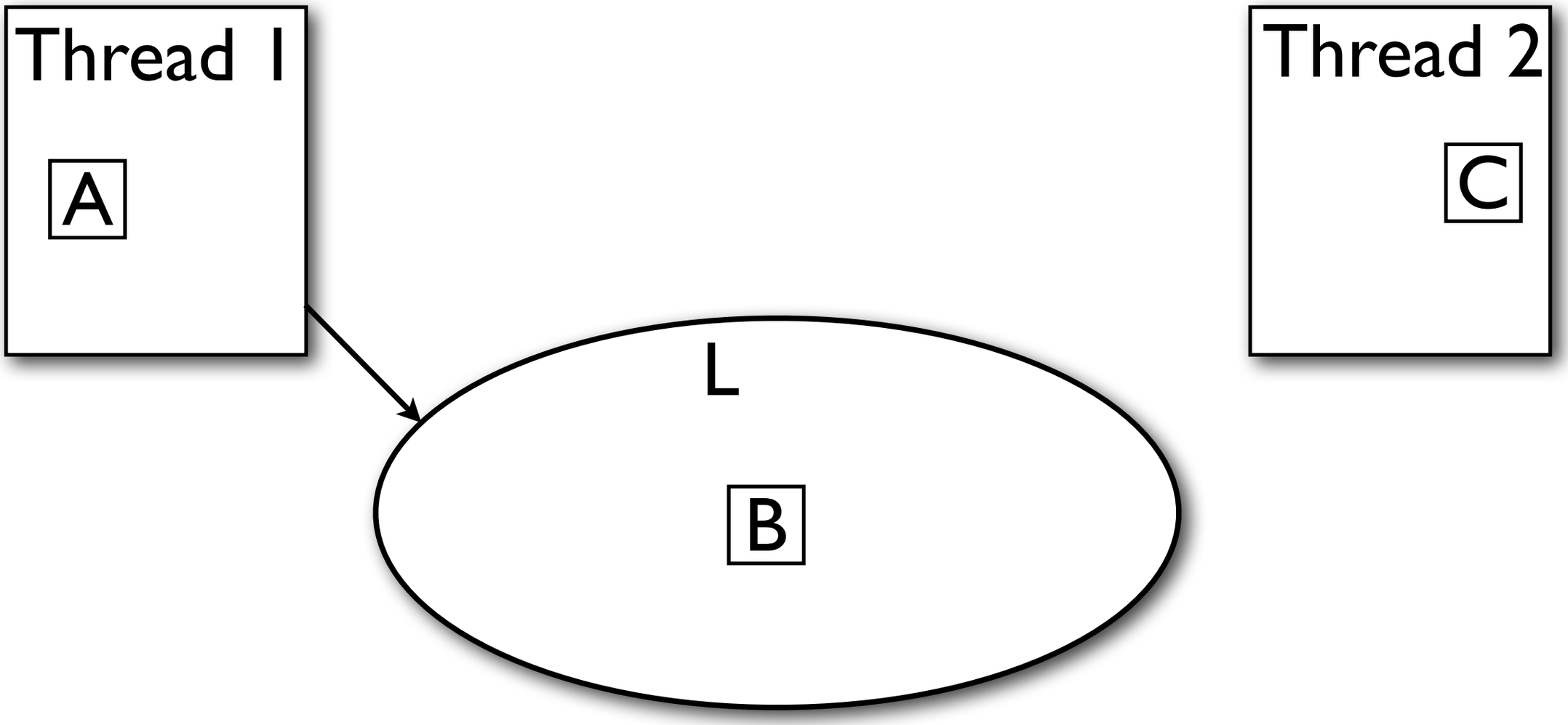
Example: container

- Typically stores pointers to data structures.



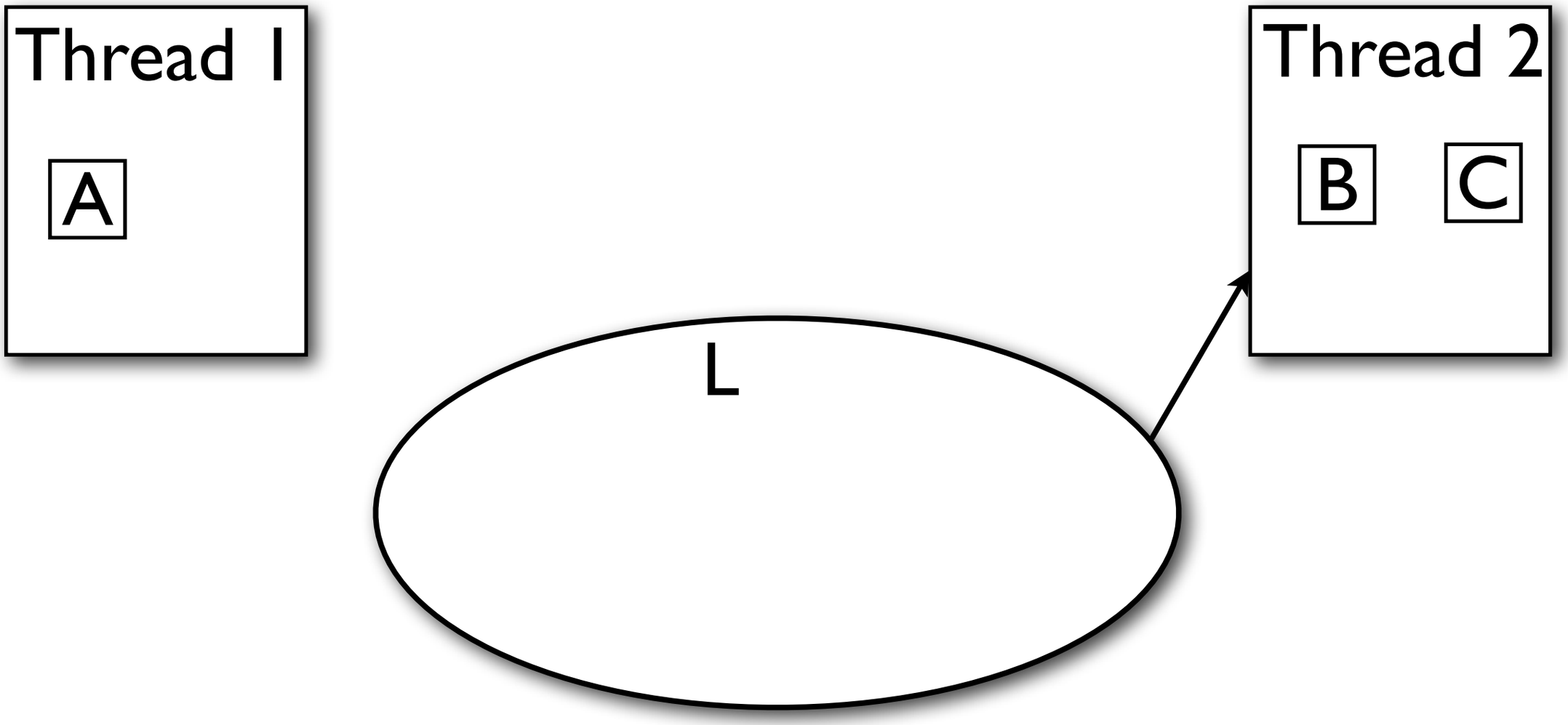
Example: container

- Typically stores pointers to data structures.



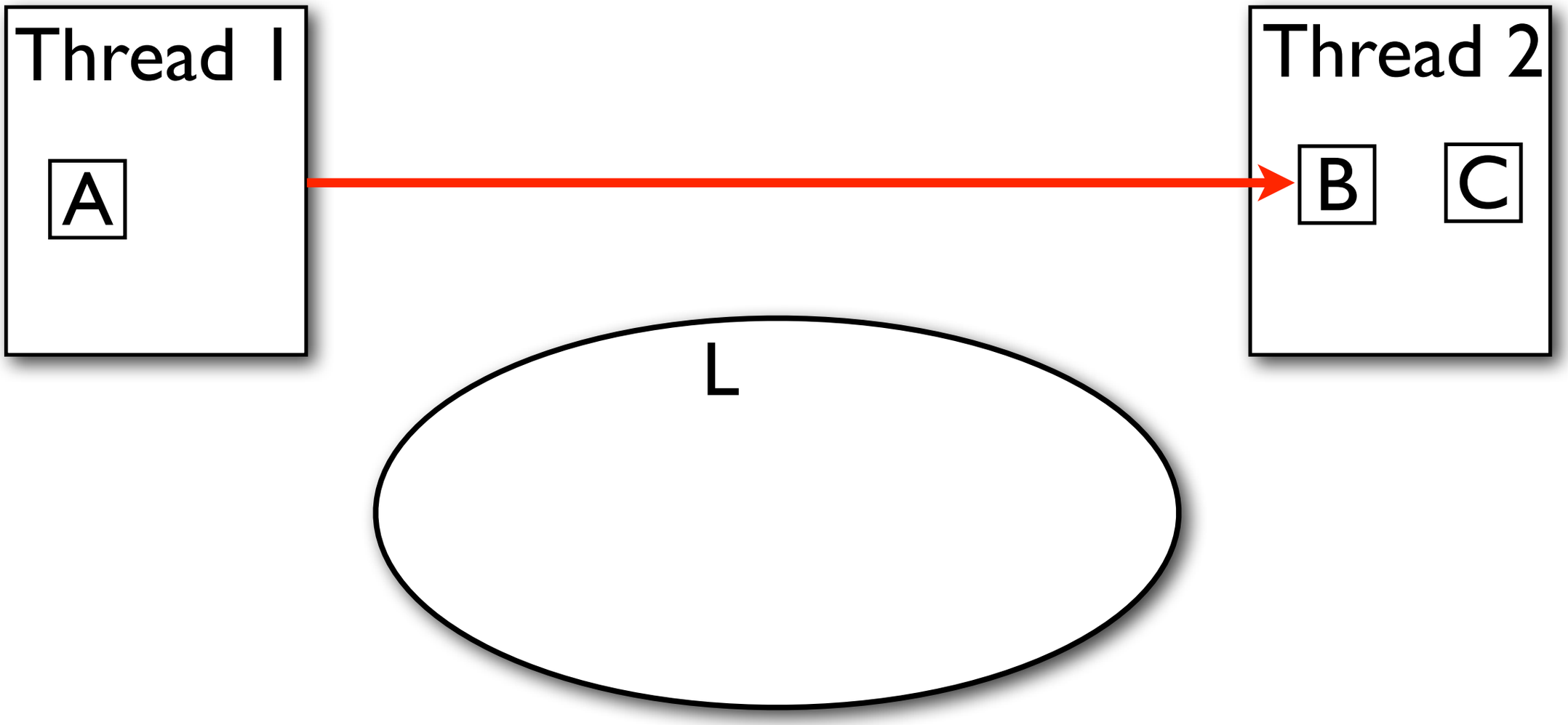
Example: container

- Typically stores pointers to data structures.



Example: container

- Typically stores pointers to data structures.

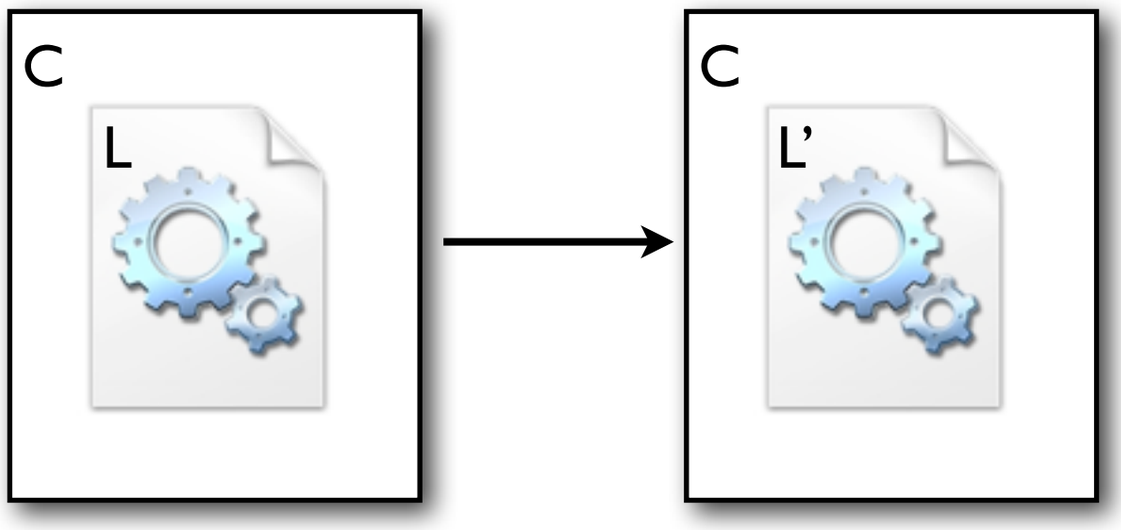


Can’t access B after transferring its ownership!

Our paper

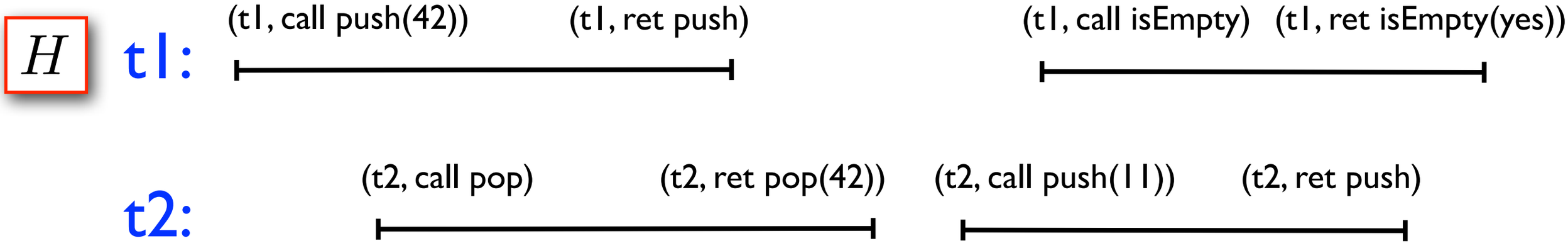
1. We generalised linearizability to a realistic setting: shared address space, ownership transfer.
2. Our generalisation gives the Abstraction Theorem:

$L \sqsubseteq L'$ $\Rightarrow$ Can replace L by L' in a proof of C in C(L).



Review of linearizability

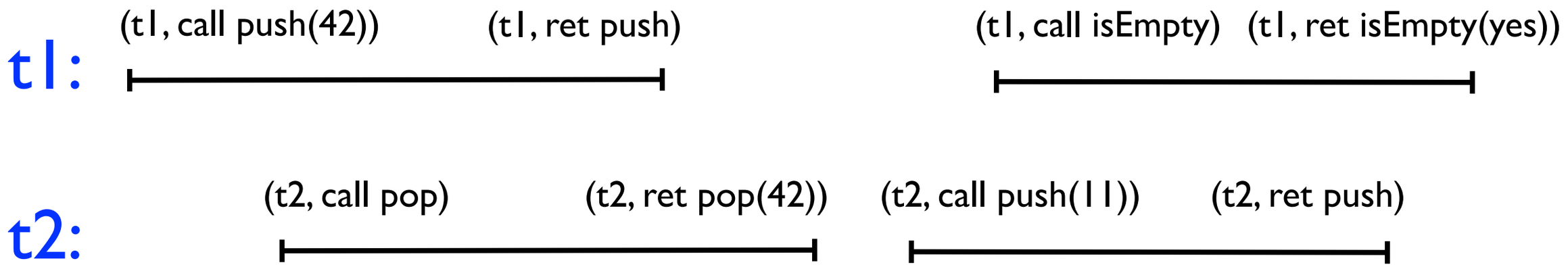
Histories



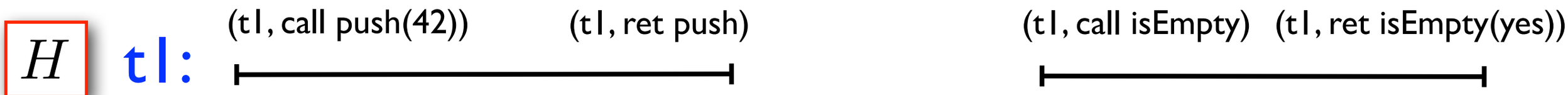
(t1, call push(42))(t2, call pop)(t1, ret push)(t2, ret pop(42))(t2,call push(11)) ...

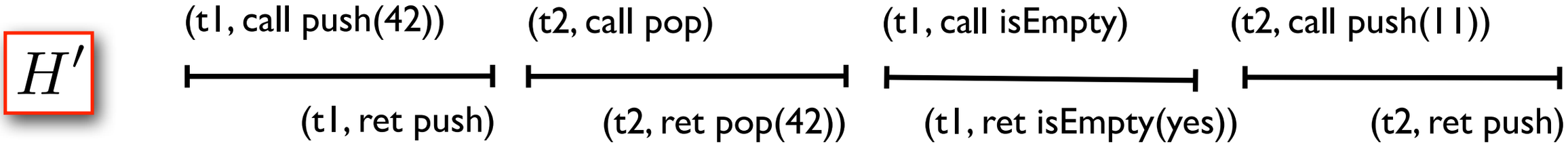
Histories

H


t1:
(t1, call push(42))
(t1, ret push)
(t1, call isEmpty)
(t1, ret isEmpty(yes))
t2:
(t2, call pop)
(t2, ret pop(42))
(t2, call push(11))
(t2, ret push)

Linearization relation

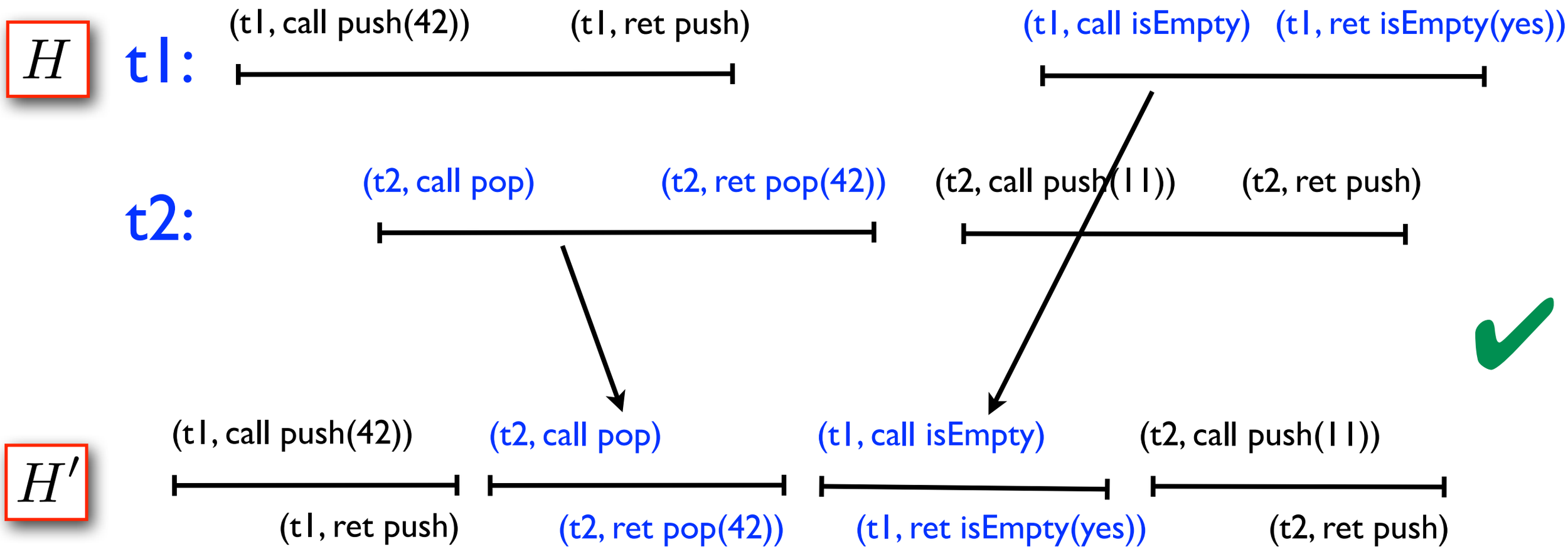




$H \sqsubseteq H'$

- We can permute calls and returns by different threads.
- But non-overlapping method invocations can't be rearranged.

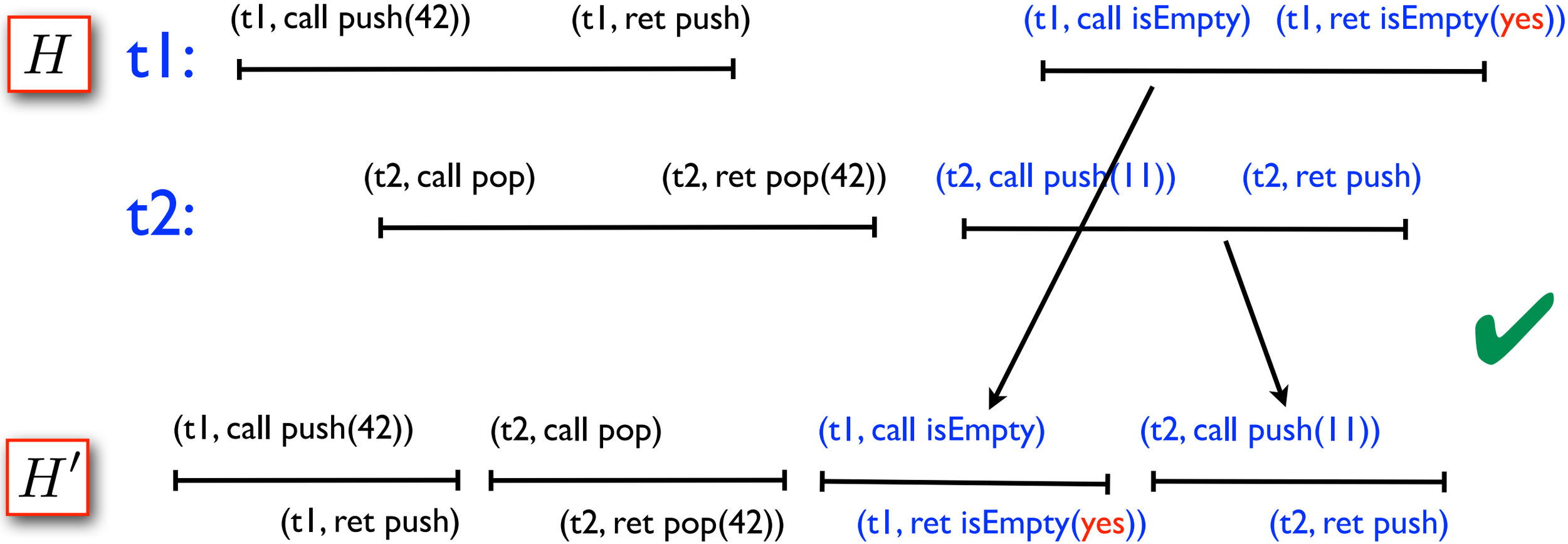
Linearization relation



$H \sqsubseteq H'$

- We can permute calls and returns by different threads.
- But non-overlapping method invocations can't be rearranged.

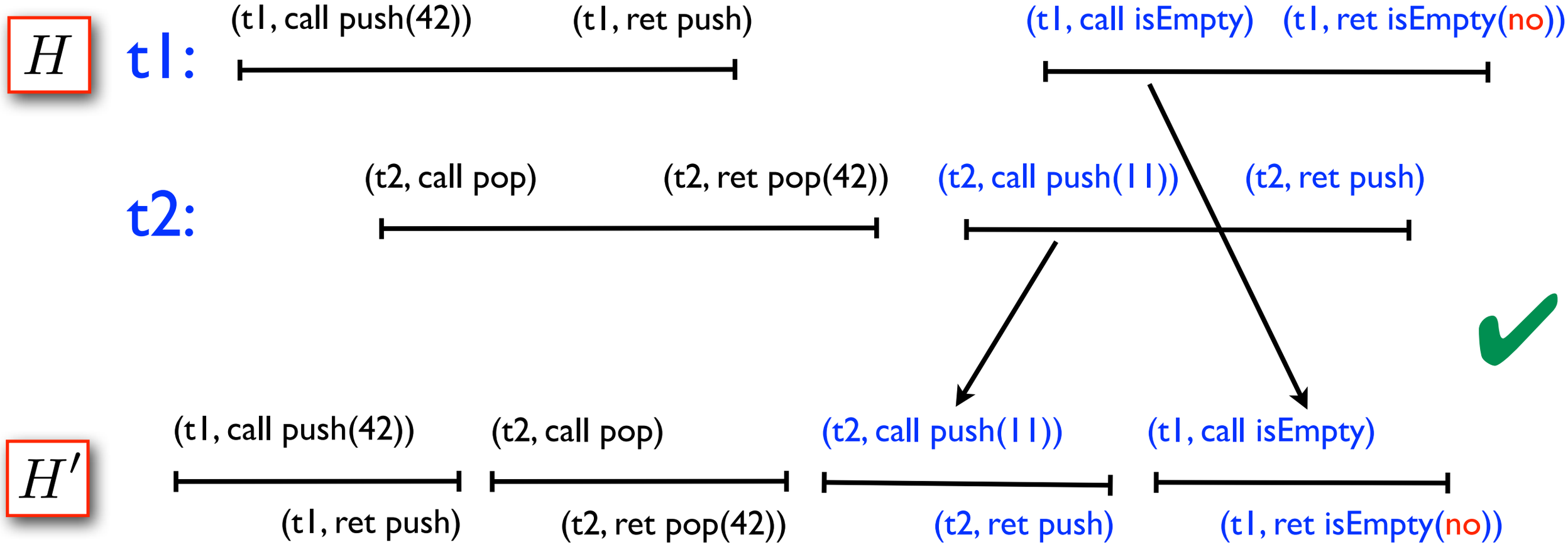
Linearization relation



$H \sqsubseteq H'$

- We can permute calls and returns by different threads.
- But non-overlapping method invocations can't be rearranged.

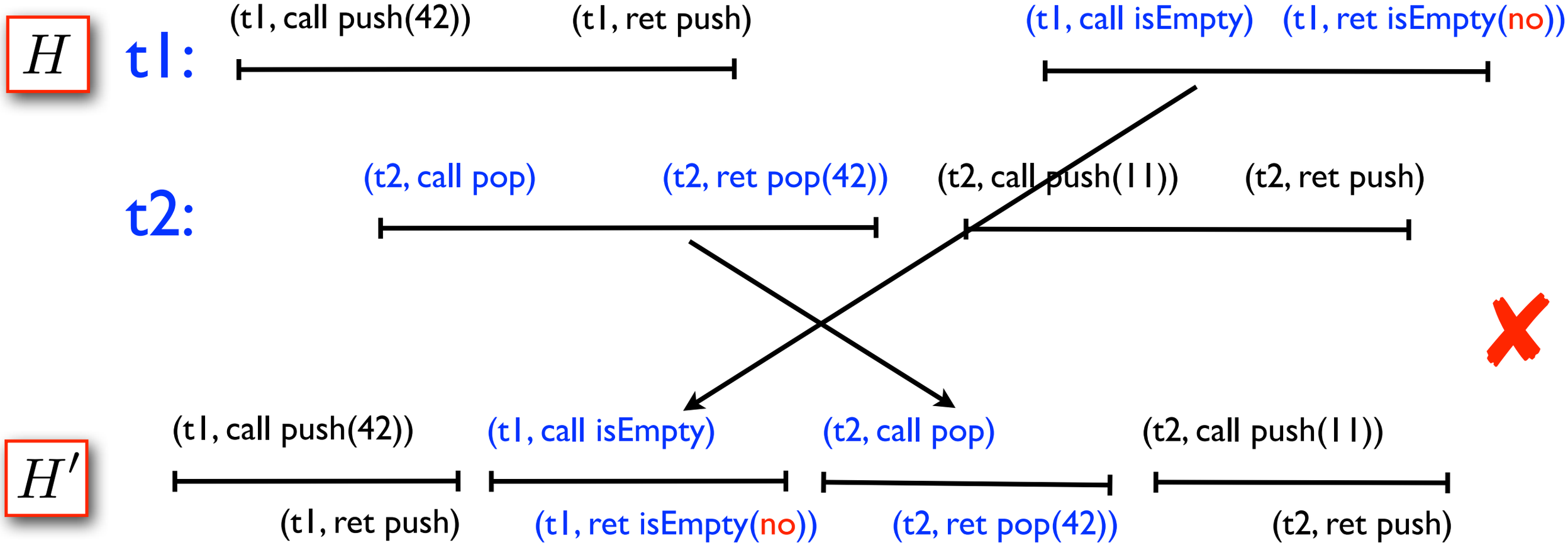
Linearization relation



$H \sqsubseteq H'$

- We can permute calls and returns by different threads.
- But non-overlapping method invocations can't be rearranged.

Linearization relation



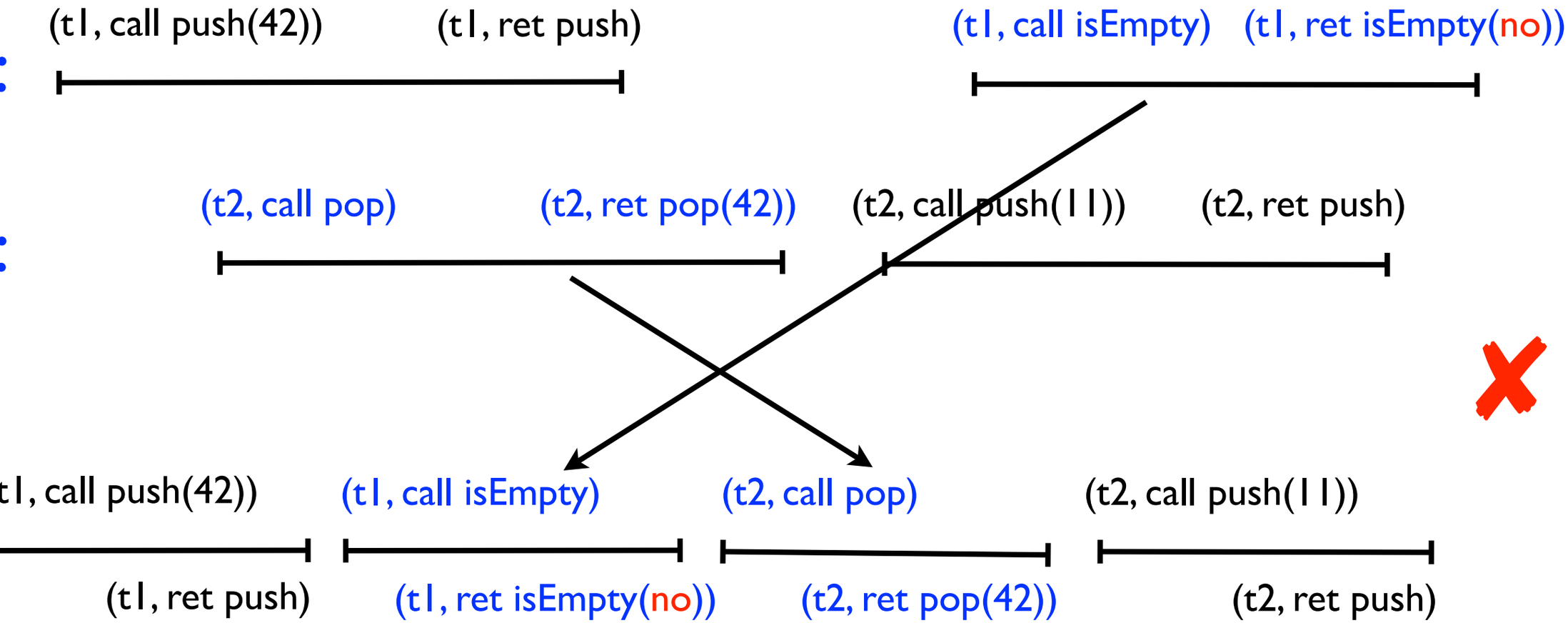
$H \sqsubseteq H'$

- We can permute calls and returns by different threads.
- But non-overlapping method invocations can't be rearranged.

Li

$$L \sqsubseteq L' \iff \forall H \in [\![L]\!].\, \exists H' \in [\![L']\!].\, H \sqsubseteq H'$$

H



H'

$H \sqsubseteq H'$

- We can permute calls and returns by different threads.
- But non-overlapping method invocations can't be rearranged.

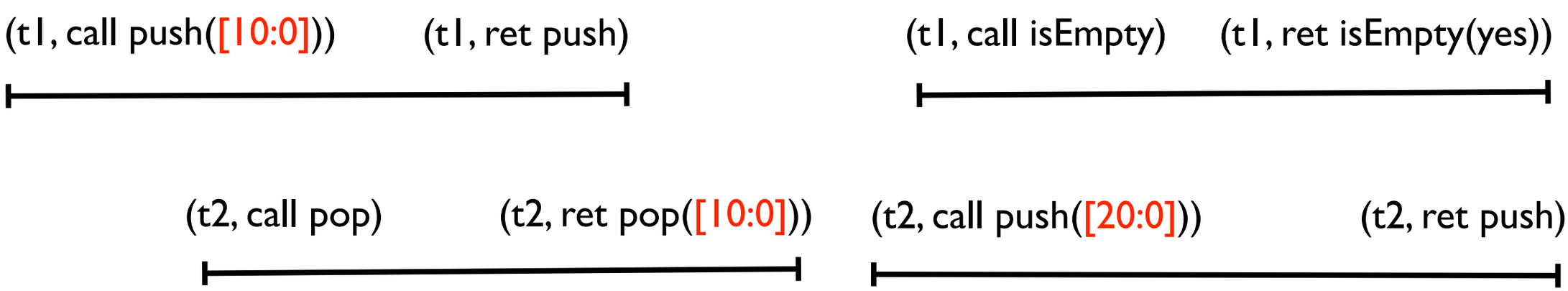
Our results

Specified library "L : Γ"

- Γ defines a light-weight specification $\{p_m\}m\{q_m\}$ for every method m in L.
- E.g. {n↦_}push(n){emp}, {emp}pop{res↦_}.

- p_m determines cells transferred from client to library.
- q_m determines cells transferred from library to client.

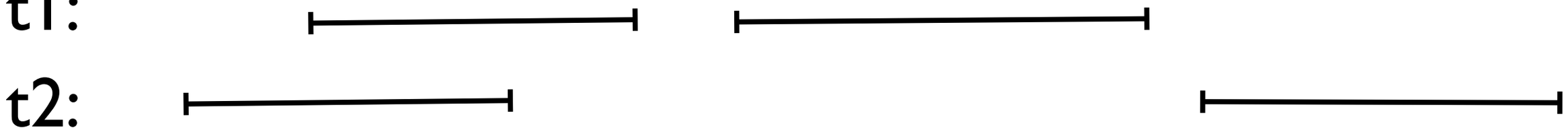
Histories with ownership transfer



- Before: (t, call m(k)), (t, ret m(k)) for k in Integer.
- Now: (t, call m(θ)), $\theta \in p_m$; (t, ret m(θ)), $\theta \in q_m$.
- Linearizability as before.

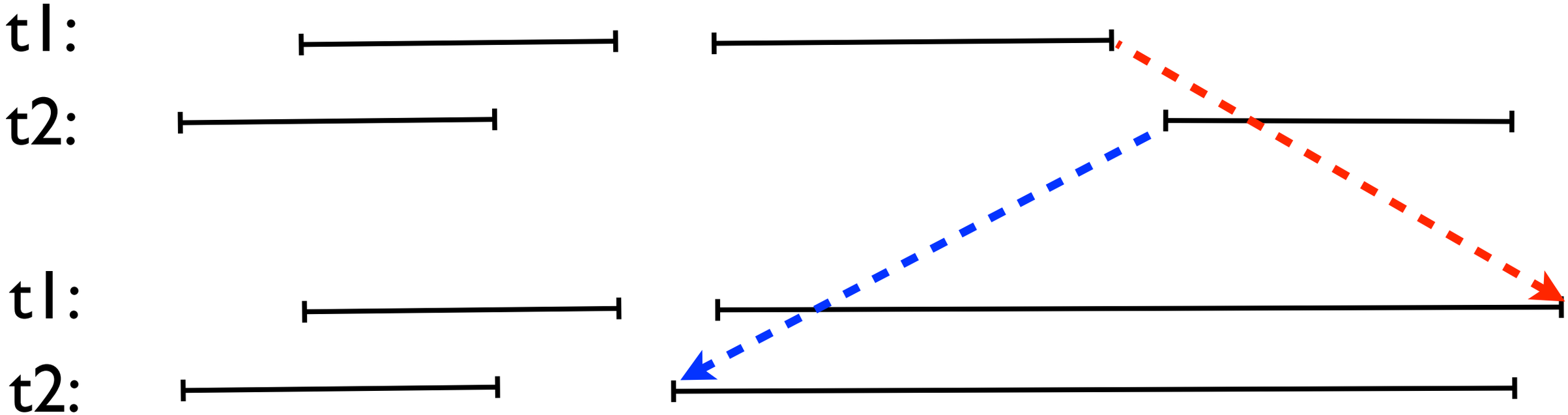
Why were we surprised?

- Important properties of standard linearizability rely on the movability of call and ret actions.



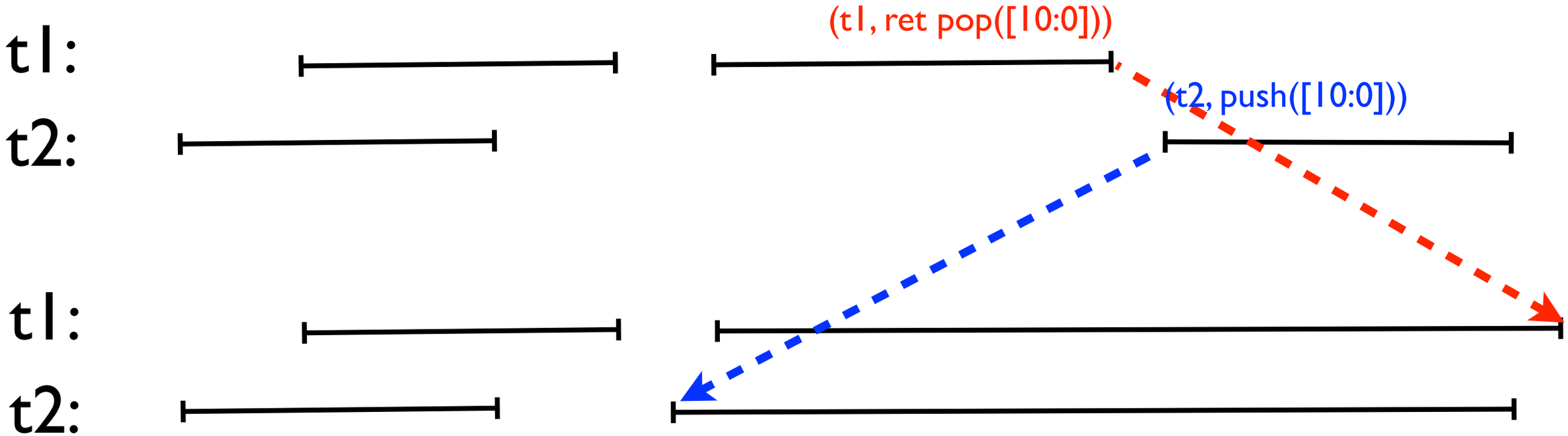
Why were we surprised?

- Important properties of standard linearizability rely on the movability of call and ret actions.



Why were we surprised?

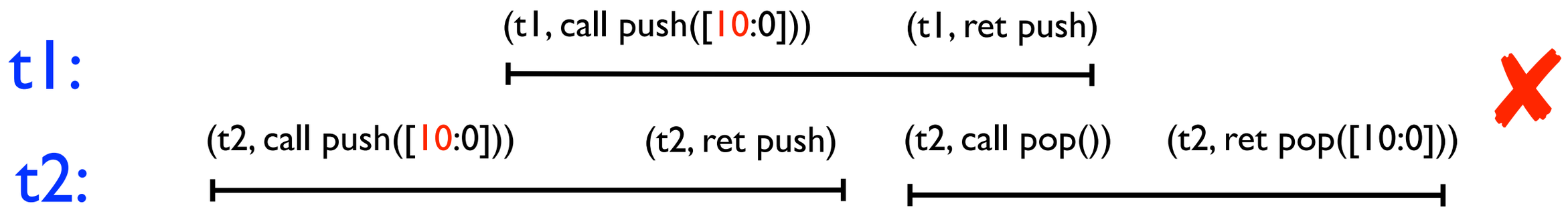
- Important properties of standard linearizability rely on the movability of call and ret actions.



- But this movability might not hold in the presence of transferred cells.

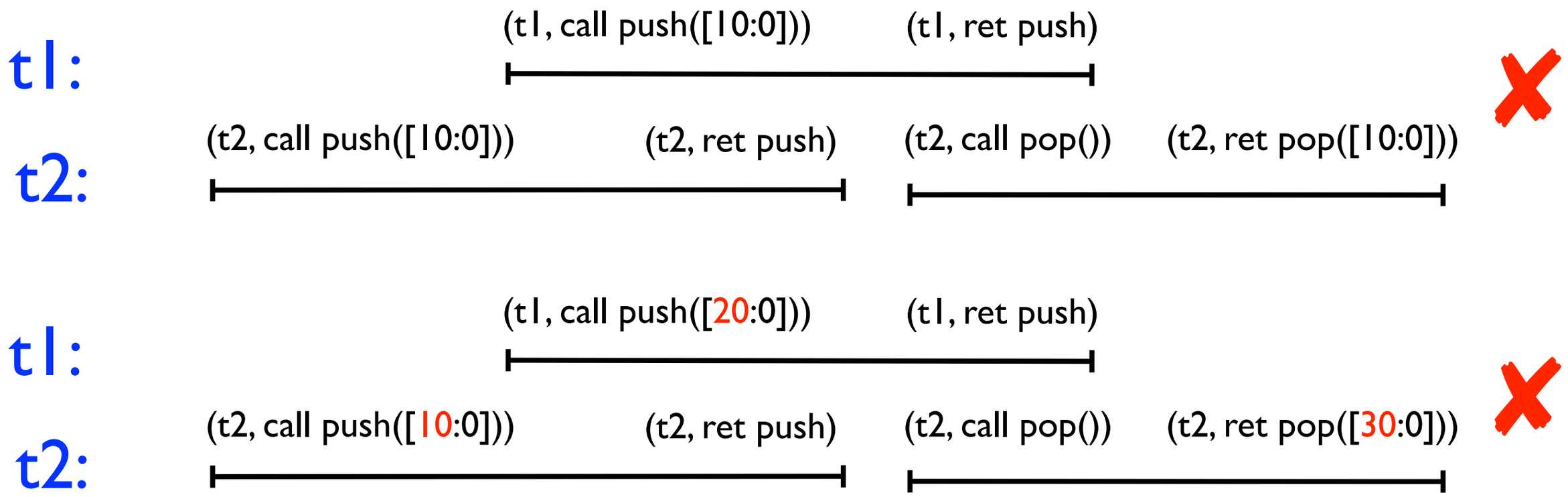
Well-balanced histories

- Histories that have correct accounting of cells owned by library and its client.



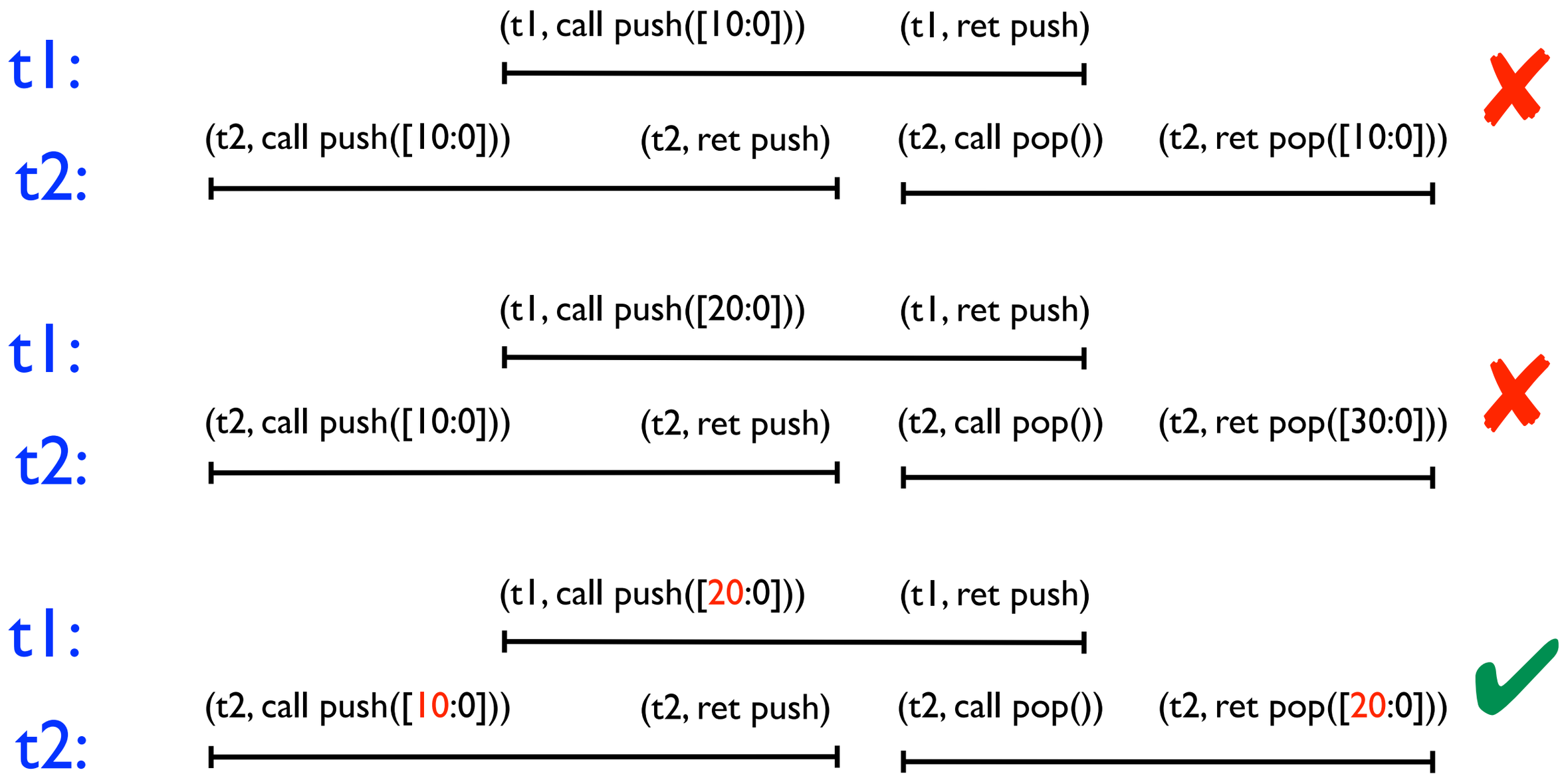
Well-balanced histories

- Histories that have correct accounting of cells owned by library and its client.



Well-balanced histories

- Histories that have correct accounting of cells owned by library and its client.



Semantics of specified library ⟦L:Γ⟧

$$\Big\|_{k=1}^{n} \left(\begin{array}{l} \texttt{while (true) \{} \\ \quad \texttt{if (nondet())} \qquad \texttt{\{new } p_1\texttt{; } m_1\texttt{(nondet()); delete } q_1\texttt{;\}} \\ \quad \texttt{else if (nondet())} \ \texttt{\{new } p_2\texttt{; } m_2\texttt{(nondet()); delete } q_2\texttt{;\}} \\ \quad \texttt{...} \\ \quad \texttt{else} \qquad\qquad\qquad\quad \texttt{\{new } p_l\texttt{; } m_l\texttt{(nondet()); delete } q_l\texttt{;\}} \\ \texttt{\}} \end{array} \right)$$

Semantics of specified library ⟦L:Γ⟧

Any number of threads

Any methods, in any order, with any parameters

$$\left(\begin{array}{l}
\texttt{while (true) \{} \\
\quad \texttt{if (nondet())} \qquad\quad \texttt{\{new } p_1\texttt{; } m_1\texttt{(nondet()); delete } q_1\texttt{;\}} \\
\quad \texttt{else if (nondet())} \; \texttt{\{new } p_2\texttt{; } m_2\texttt{(nondet()); delete } q_2\texttt{;\}} \\
\quad \ldots \\
\quad \texttt{else} \qquad\qquad\qquad\quad \texttt{\{new } p_l\texttt{; } m_l\texttt{(nondet()); delete } q_l\texttt{;\}} \\
\texttt{\}}
\end{array}\right)^{\Big\| n}_{k=1}$$

Semantics of specified library ⟦L:Γ⟧

Any number of threads

Any methods, in any order, with any parameters

$$\left(\begin{array}{l}
\texttt{while (true) \{} \\
\quad \texttt{if (nondet())} \quad\;\; \texttt{\{new } p_1\texttt{; } m_1\texttt{(nondet()); delete } q_1\texttt{;\}} \\
\quad \texttt{else if (nondet())} \; \texttt{\{new } p_2\texttt{; } m_2\texttt{(nondet()); delete } q_2\texttt{;\}} \\
\quad \texttt{...} \\
\quad \texttt{else} \qquad\qquad\qquad \texttt{\{new } p_l\texttt{; } m_l\texttt{(nondet()); delete } q_l\texttt{;\}} \\
\texttt{\}}
\end{array}\right)^{\|_{k=1}^{n}}$$

- new p: allocates an arbitrary state satisfying p.
- delete q: removes the part of state satisfying q.

Semantics of specified library ⟦L:Γ⟧

$$\Big\|_{k=1}^{n} \left(\begin{array}{l} \texttt{while (true) \{} \\ \quad \texttt{if (nondet())} \qquad \texttt{\{new } p_1\texttt{; } m_1\texttt{(nondet()); delete } q_1\texttt{;\}} \\ \quad \texttt{else if (nondet())} \ \texttt{\{new } p_2\texttt{; } m_2\texttt{(nondet()); delete } q_2\texttt{;\}} \\ \quad \texttt{...} \\ \quad \texttt{else} \qquad\qquad\qquad\qquad \texttt{\{new } p_l\texttt{; } m_l\texttt{(nondet()); delete } q_l\texttt{;\}} \\ \texttt{\}} \end{array} \right)$$

- ⟦L:Γ⟧ consists of histories generated by this most general client.
- Lemma: Only well-behaved histories are generated.

Linearizability

$$L \sqsubseteq L' \iff \forall H \in [\![L]\!].\, \exists H' \in [\![L']\!].\, H \sqsubseteq H'$$

Generated from the most-general client

- Libraries L and L' have the same specifications Γ.

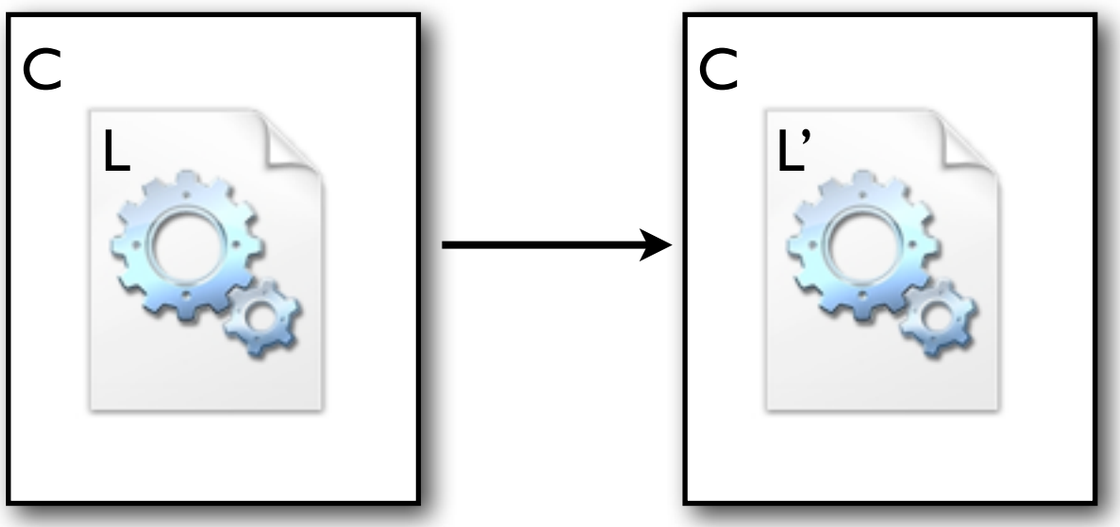
Abstraction Theorem

Assume:

- Client C and libraries L, L' : Γ are safe.
- L $\sqsubseteq$ L'.

Then client($[\![C(L)]\!]$) $\subseteq$ client($[\![C(L')]\!]$)

Can replace L by L' in a proof of C:



$$C(L') \models P \Rightarrow C(L) \models P$$

Abstraction Theore

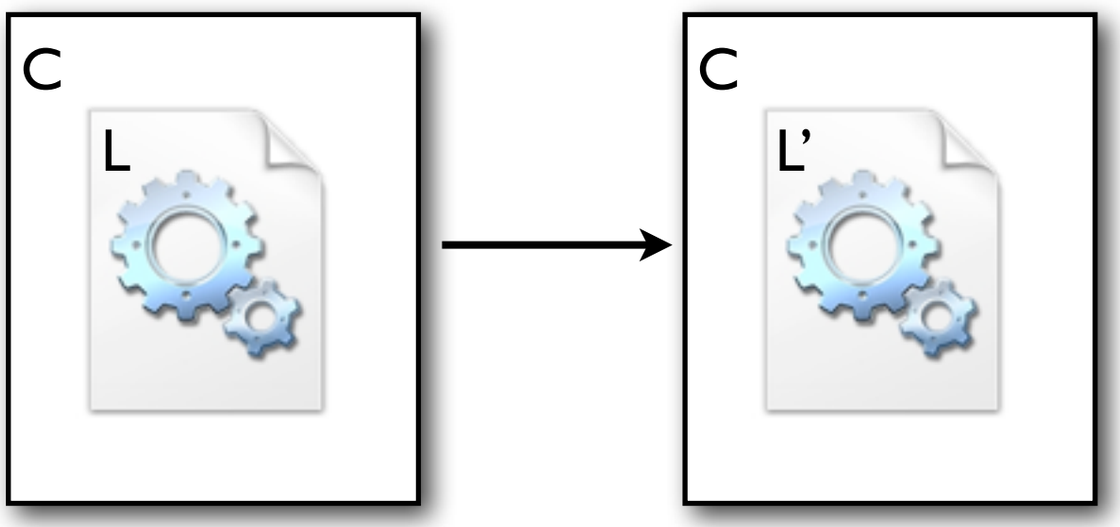
Client and library do not corrupt each other.

Assume:

- Client C and libraries L, L' : Γ are safe.
- L ⊑ L'.

Then client($[\![C(L)]\!]$) ⊆ client($[\![C(L')]\!]$)

Can replace L by L' in a proof of C:



$$C(L') \models P \Rightarrow C(L) \models P$$