Compositional C11 Program Transformation

Mark Batty - Mike Dodds - Alexey Gotsman

Imperial Concurrency Workshop, July 2015

@miike

Overview

The C11 model is (arguably) broken: we omit problem features, most importantly *no RLX.*

- Context: relaxed memory, axiomatic semantics, fragment of C11 / C++11.
- Immediate aim: program transformation, eg compiler optimisations.
- Approach: summarise context interactions using a set of *histories* (denotational-*ish*).
- Under construction!

Overview

1. **Objective: compositional transformation**
2. C11 semantics primer
3. Defining execution histories
4. Cutting down contexts

Fragments and Contexts

C - whole program

$[\![C]\!]$ - semantics (defined by a set of executions)

$$C(-) \quad + \quad P \quad = \quad C(P)$$

Context *(program with hole)*

Program fragment

Whole program

Motivation: Compiler Optimisations

$P_1 \rightsquigarrow P_2$ - replace one fragment with another

$P_1:$ r=read(x);
r=read(x);
$\rightsquigarrow$ $P_2:$ r=read(x);

Assume operations are **release-acquire** unless otherwise mentioned

Is this a sound transformation on C11?

Motivation: Compiler Optimisations

$P_1 \rightsquigarrow P_2$ - replace one fragment with another

Soundness:

$$\forall C.\, \forall X_2 \in [\![C(P_2)]\!].\, \exists X_1 \in [\![C(P_1)]\!].\, \mathsf{obsv}(X_2) = \mathsf{obsv}(X_1)$$

all contexts

executions of transformed program

executions of prior program

equivalent observed behaviour

Approach

$[\![P]\!]_d$ - summarises of all possible interactions

(…a kind of *denotation*)

Adequacy:

$$[\![P_2]\!]_d \subseteq [\![P_1]\!]_d \implies$$
$$\forall C. \forall X_2 \in [\![C(P_2)]\!]. \exists X_1 \in [\![C(P_1)]\!]. \mathsf{obsv}(X_2) = \mathsf{obsv}(X_1)$$

Thus:

$[\![P_2]\!]_d \subseteq [\![P_1]\!]_d \implies P_1 \rightsquigarrow P_2$ is sound

Approach

We'd also like *finiteness:*

P is loop-free code $\implies$ $[\![P]\!]_d$ is finite

(…possibly with symbolic values?)

This would support e.g. automated checking

1. Objective: compositional transformation
2. **C11 semantics primer**
3. Defining execution histories
4. Cutting down contexts

C11 concurrency semantics

Executions: multiple partial orders on memory actions.

$$\langle \mathcal{A}, \mathsf{rf}, \mathsf{hb}, \mathsf{mo} \rangle \ \in \ [\![C]\!]$$

memory actions *read-writes* *happens before* *per-location coherence*

C11 concurrency semantics (II)

C11 semantics is *very* non-compositional:

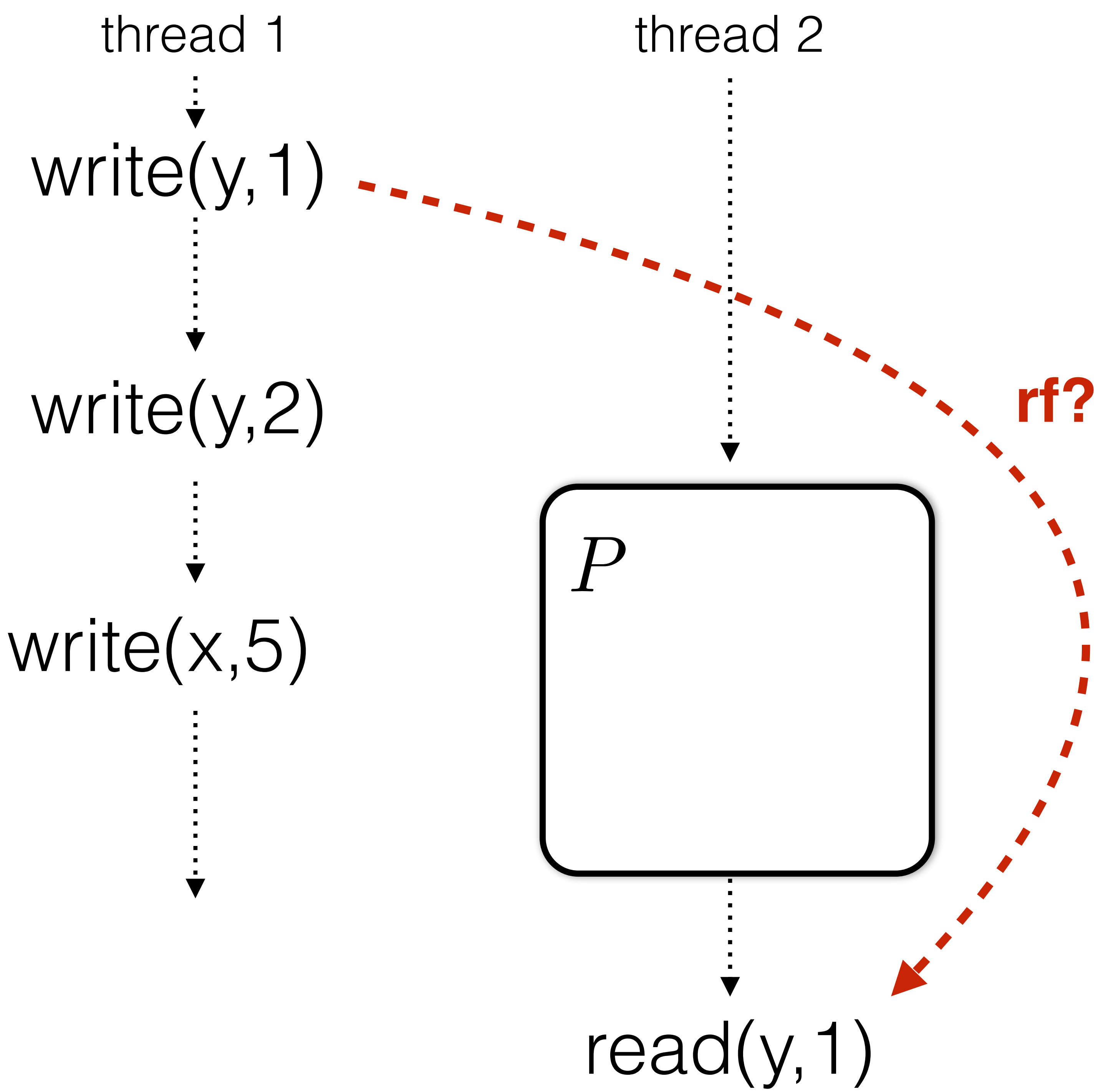
1. Generate whole-program *execution candidates*.
2. Filter on the basis of *validity axioms.*

Validity forbids eg:

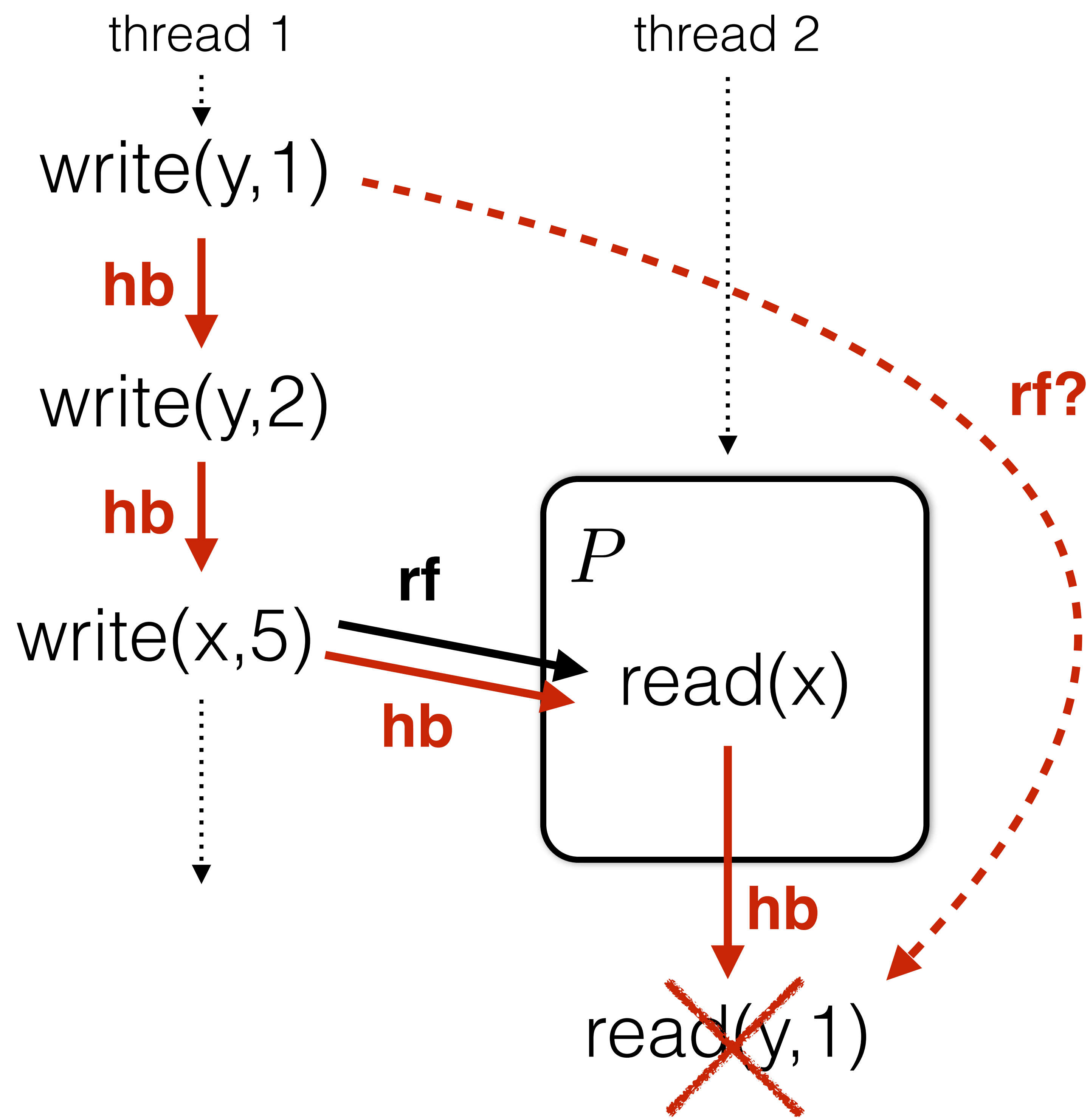
$$\begin{tikzcd}
\mathrm{write}(x,v) \arrow[r, "\mathbf{hb}"'] \arrow[rr, bend left, "\mathbf{rf}"] & \mathrm{write}(x,v') \arrow[r, "\mathbf{hb}"'] & \mathrm{read}(x,v)
\end{tikzcd}$$

forbidden!

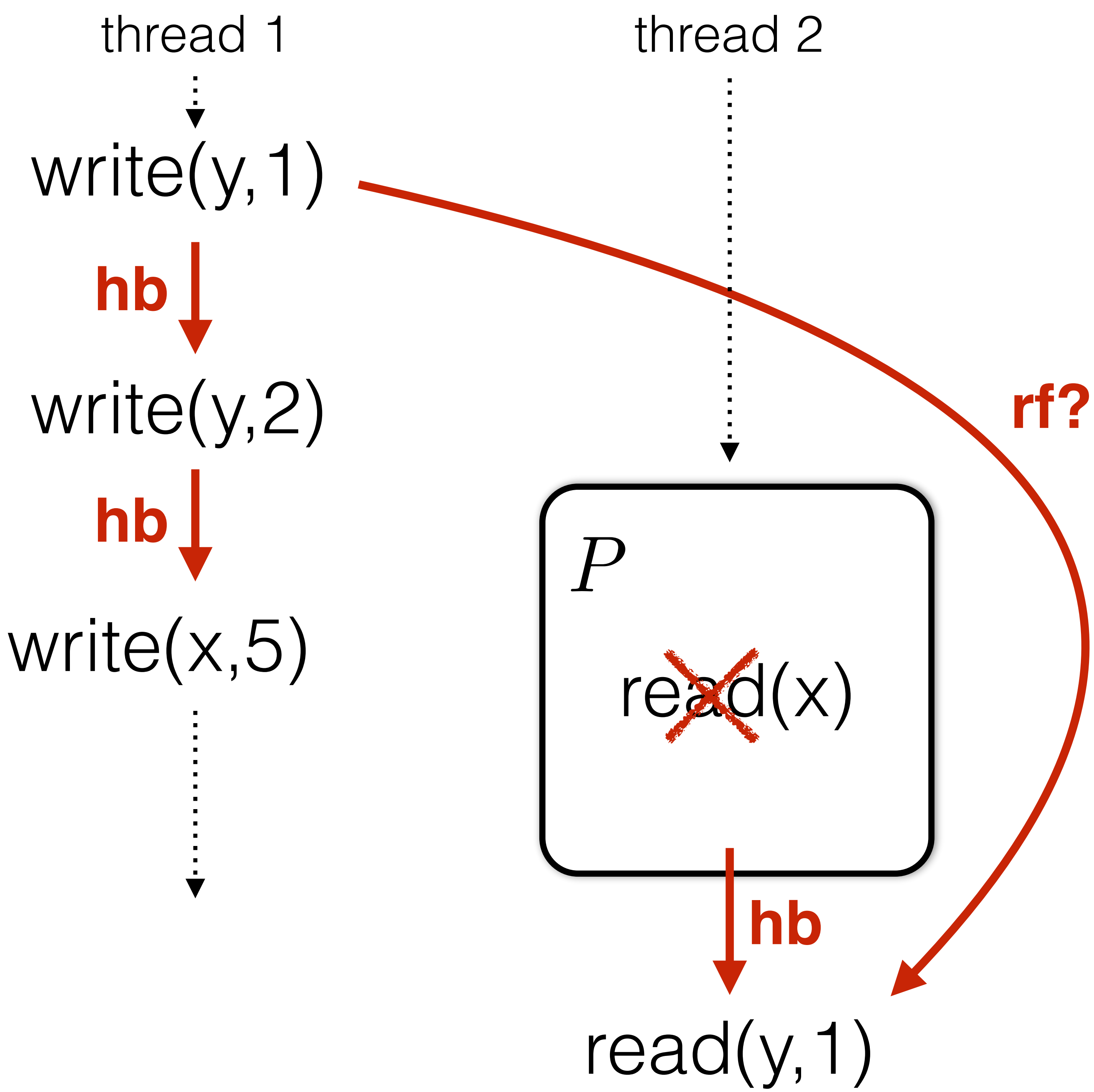
Observable behaviour



Observable behaviour



Observable behaviour

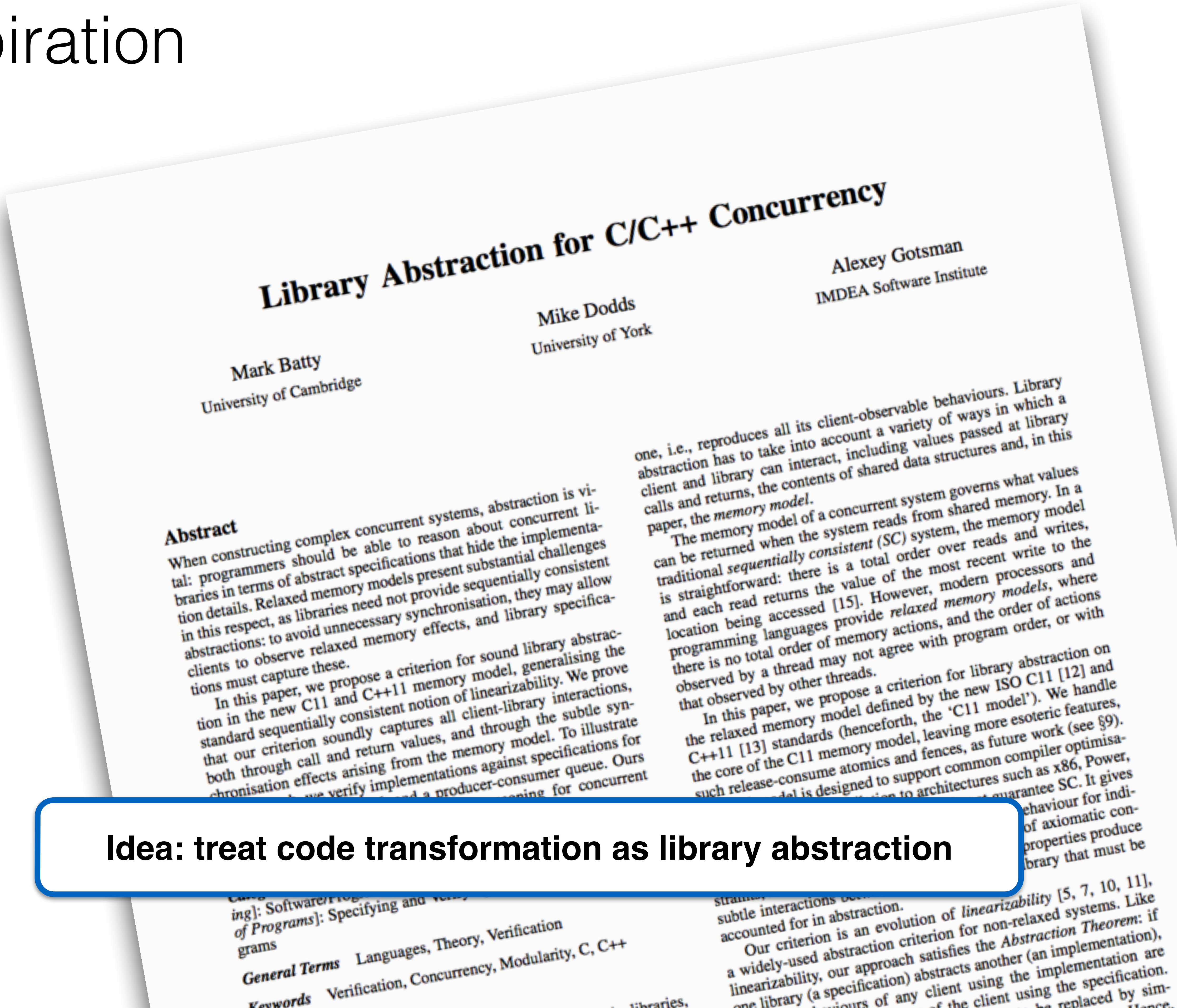


C11 Challenges

1. Semantics is whole-program and axiomatic.
2. No notion of a global state
3. Unclear when orders can be observed

1. Objective: compositional transformation
2. C11 semantics primer
3. **Defining execution histories**
4. Cutting down contexts

Inspiration



Library Abstraction for C/C++ Concurrency

Mark Batty
University of Cambridge

Mike Dodds
University of York

Alexey Gotsman
IMDEA Software Institute

Abstract

When constructing complex concurrent systems, abstraction is vital: programmers should be able to reason about concurrent libraries in terms of abstract specifications that hide the implementation details. Relaxed memory models present substantial challenges in this respect, as libraries need not provide sequentially consistent abstractions: to avoid unnecessary synchronisation, they may allow clients to observe relaxed memory effects, and library specifications must capture these.

In this paper, we propose a criterion for sound library abstraction in the new C11 and C++11 memory model, generalising the standard sequentially consistent notion of linearizability. We prove that our criterion soundly captures all client-library interactions, both through call and return values, and through the subtle synchronisation effects arising from the memory model. To illustrate our criterion, we verify implementations against specifications for ... producer-consumer queue. Ours ... for concurrent

ing]: Software/Prog... of Programs]: Specifying and ... grams

General Terms Languages, Theory, Verification

Keywords Verification, Concurrency, Modularity, C, C++

one, i.e., reproduces all its client-observable behaviours. Library abstraction has to take into account a variety of ways in which a client and library can interact, including values passed at library calls and returns, the contents of shared data structures and, in this paper, the *memory model*.

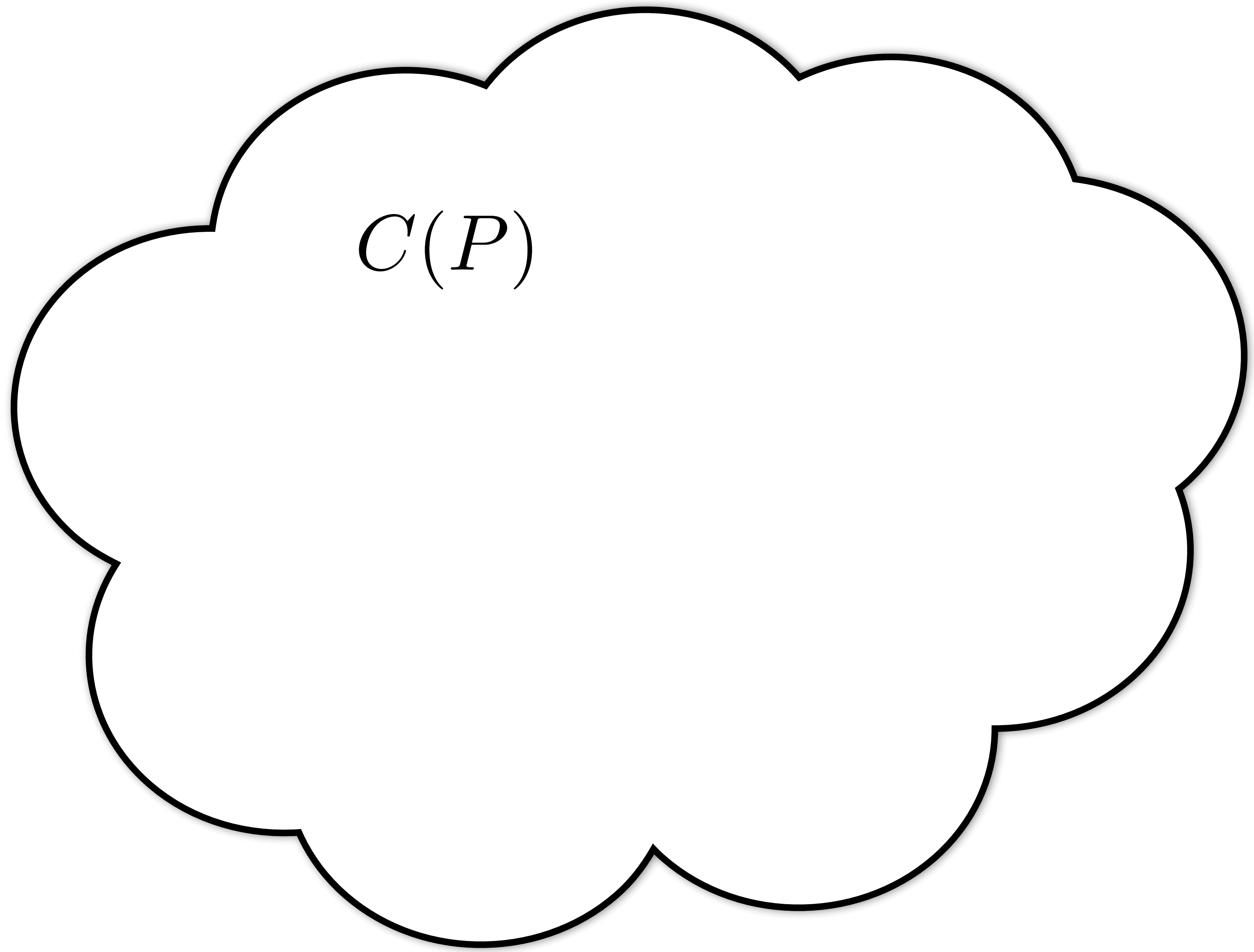
The memory model of a concurrent system governs what values can be returned when the system reads from shared memory. In a traditional *sequentially consistent (SC)* system, the memory model is straightforward: there is a total order over reads and writes, and each read returns the value of the most recent write to the location being accessed [15]. However, modern processors and programming languages provide *relaxed memory models*, where there is no total order of memory actions, and the order of actions observed by a thread may not agree with program order, or with that observed by other threads.

In this paper, we propose a criterion for library abstraction on the relaxed memory model defined by the new ISO C11 [12] and C++11 [13] standards (henceforth, the 'C11 model'). We handle the core of the C11 memory model, leaving more esoteric features, such release-consume atomics and fences, as future work (see §9). ... is designed to support common compiler optimisa... to architectures such as x86, Power, ... guarantee SC. It gives ... behaviour for indi... of axiomatic con... properties produce ... library that must be ... subtle interactions ... accounted for in abstraction.

Our criterion is an evolution of *linearizability* [5, 7, 10, 11], a widely-used abstraction criterion for non-relaxed systems. Like linearizability, our approach satisfies the *Abstraction Theorem*: if one library (a specification) abstracts another (an implementation), the behaviours of any client using the implementation are ... of the client using the specification. ... be replaced by sim...

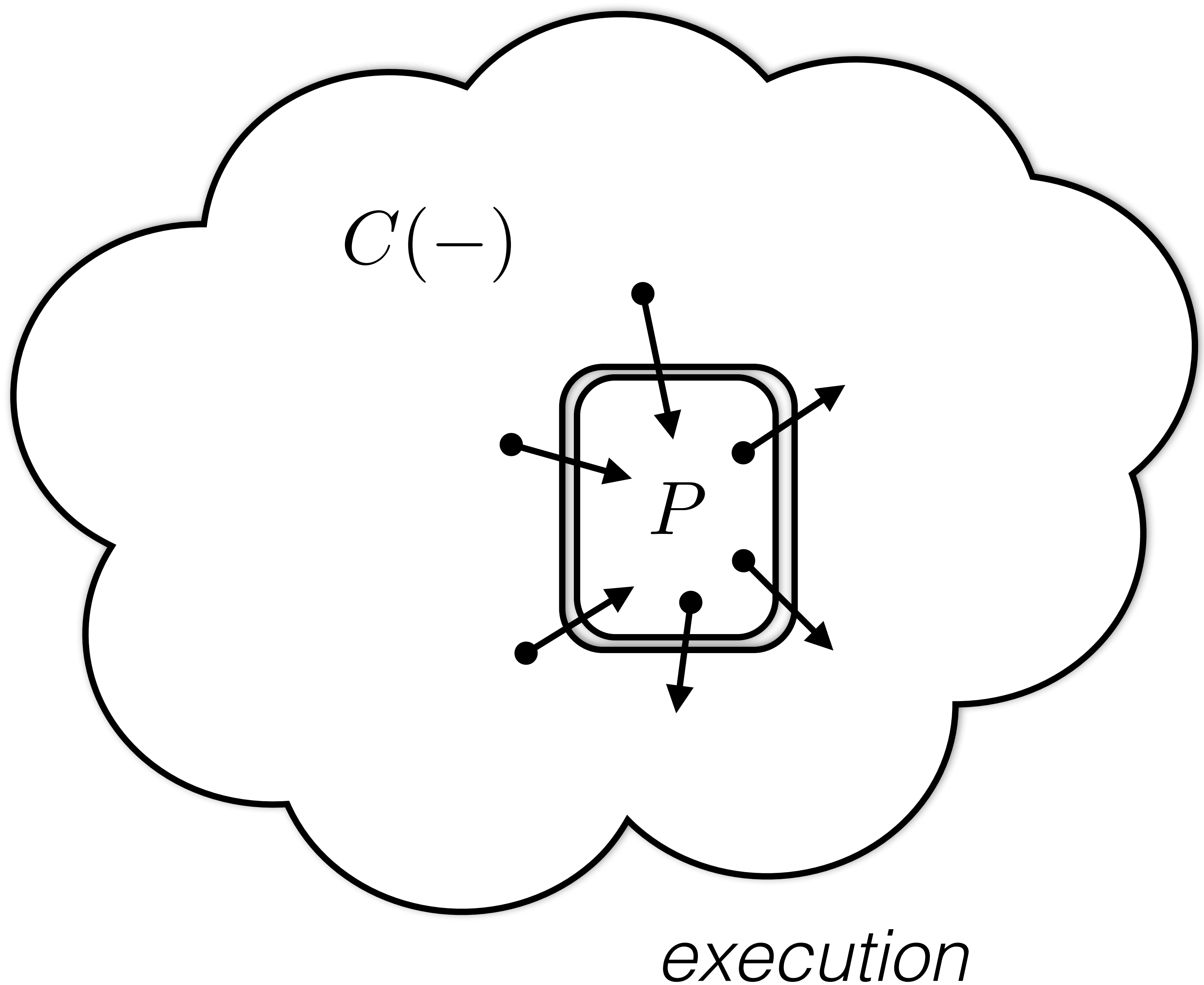
Idea: treat code transformation as library abstraction

Intuition

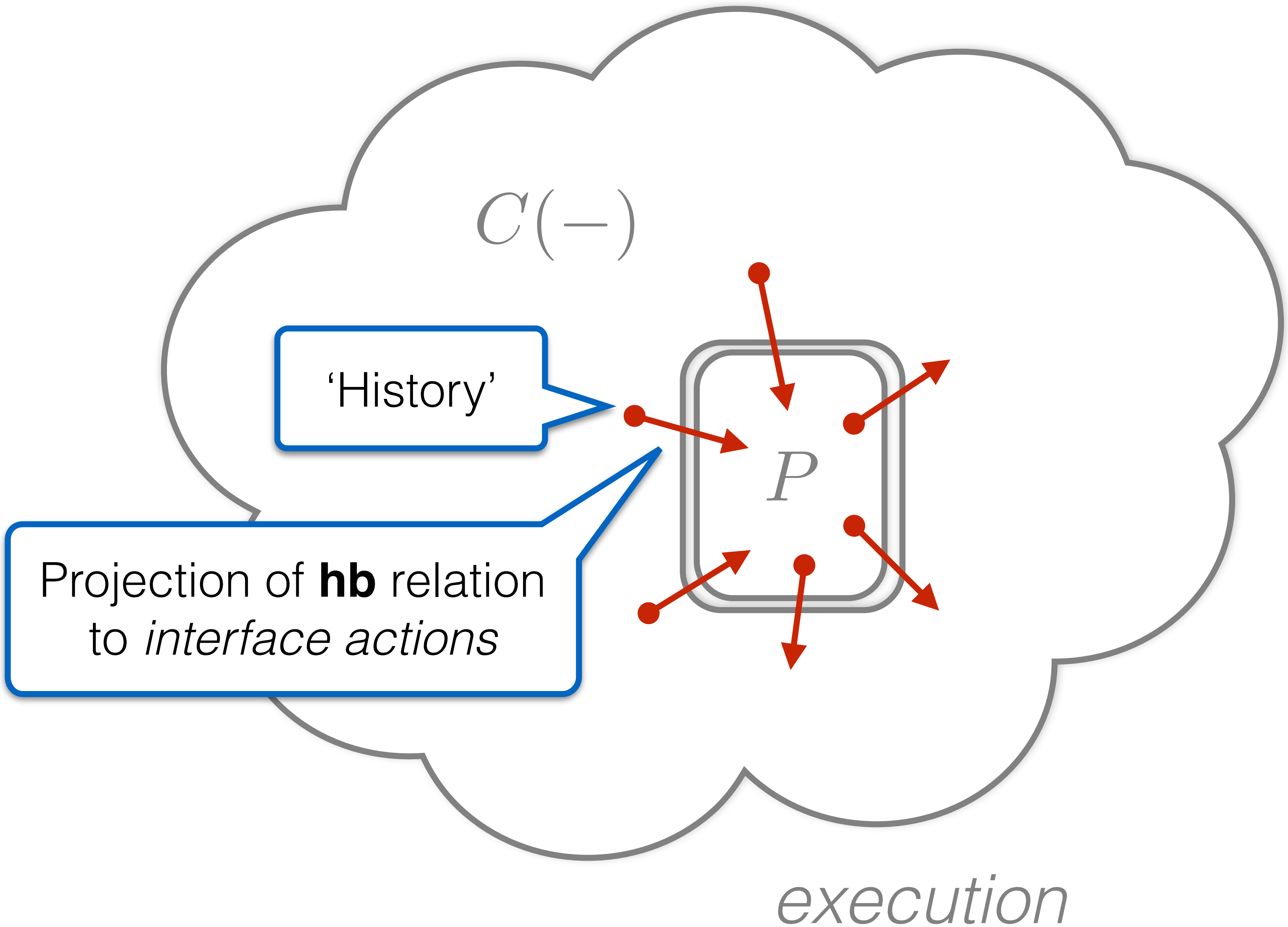


execution

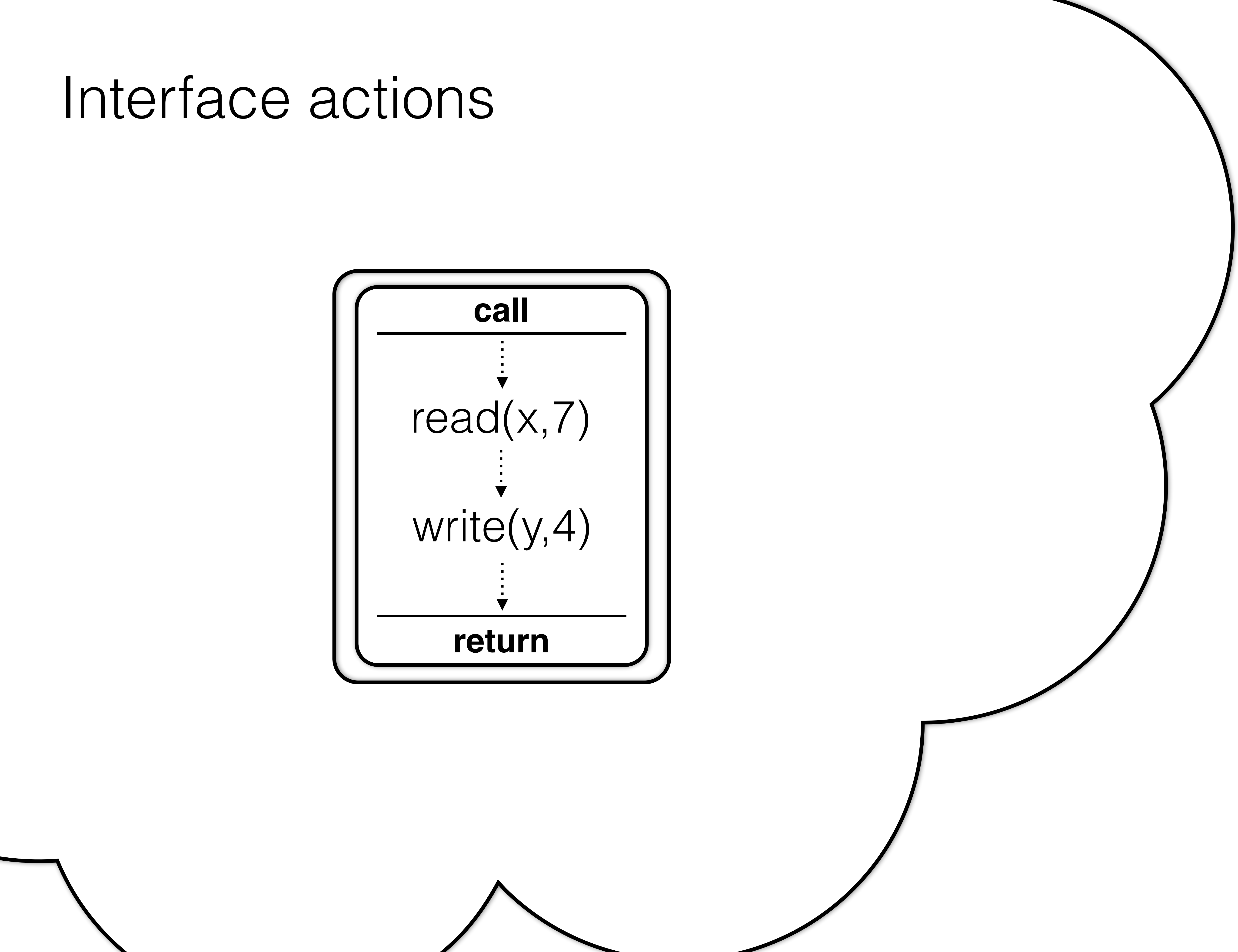
Intuition



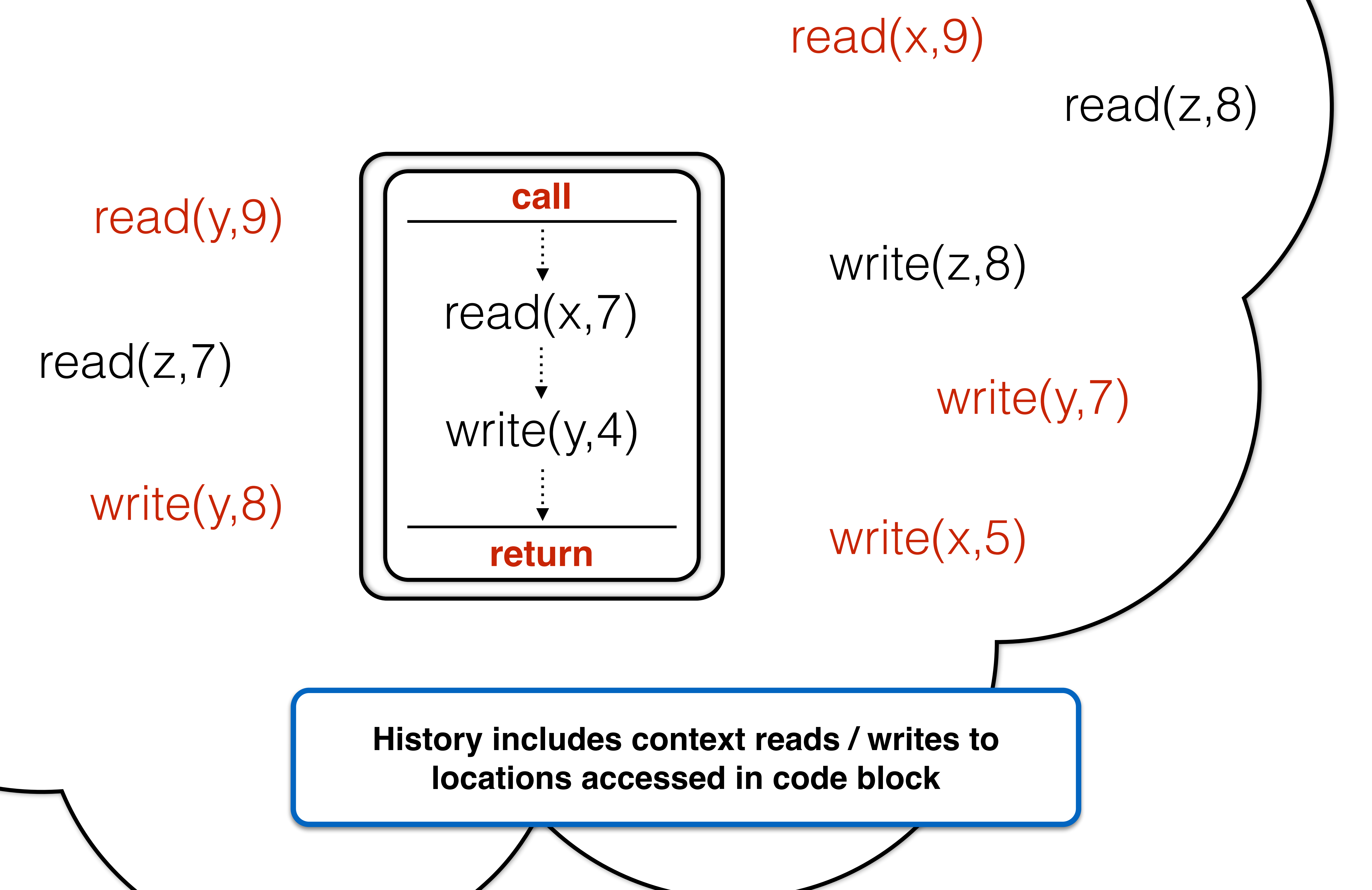
Intuition


$C(-)$
'History'
Projection of **hb** relation to *interface actions*
P
execution

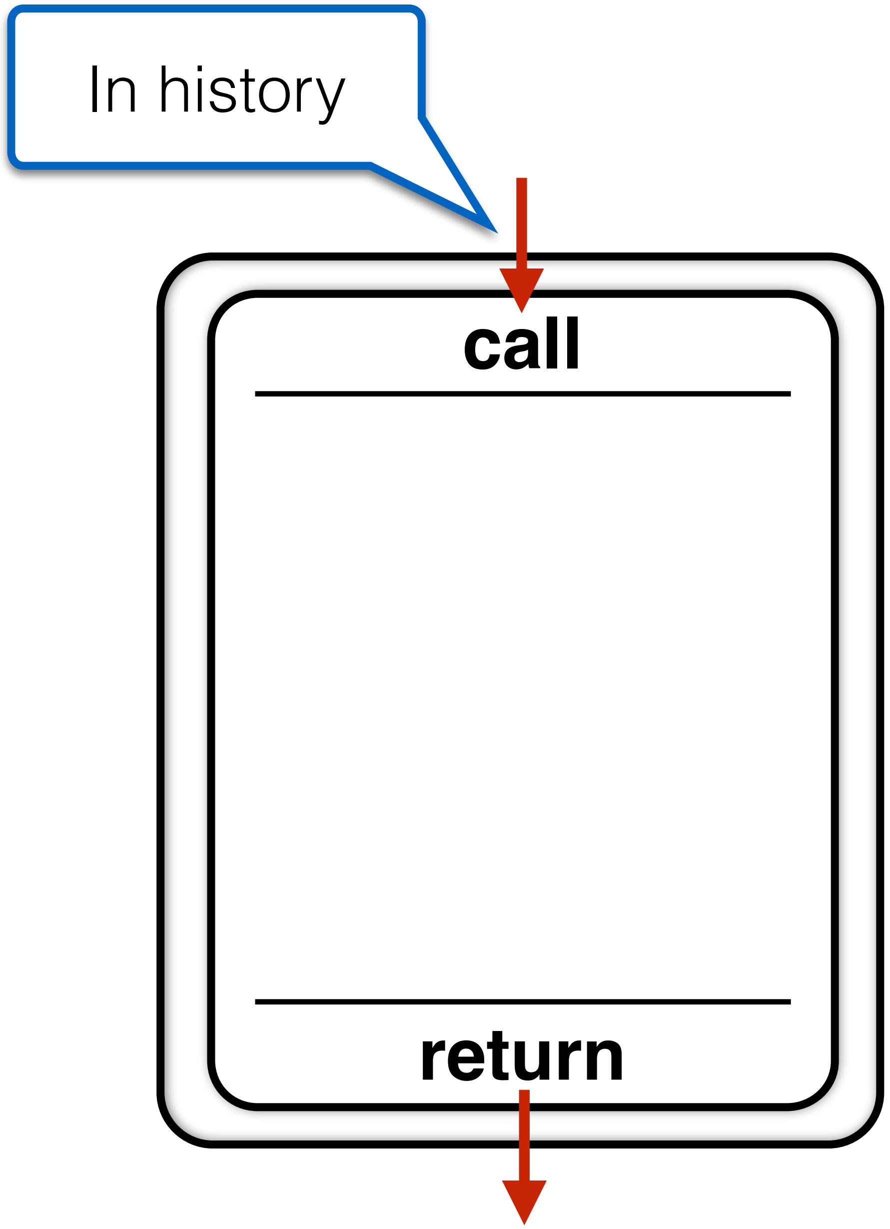
Interface actions



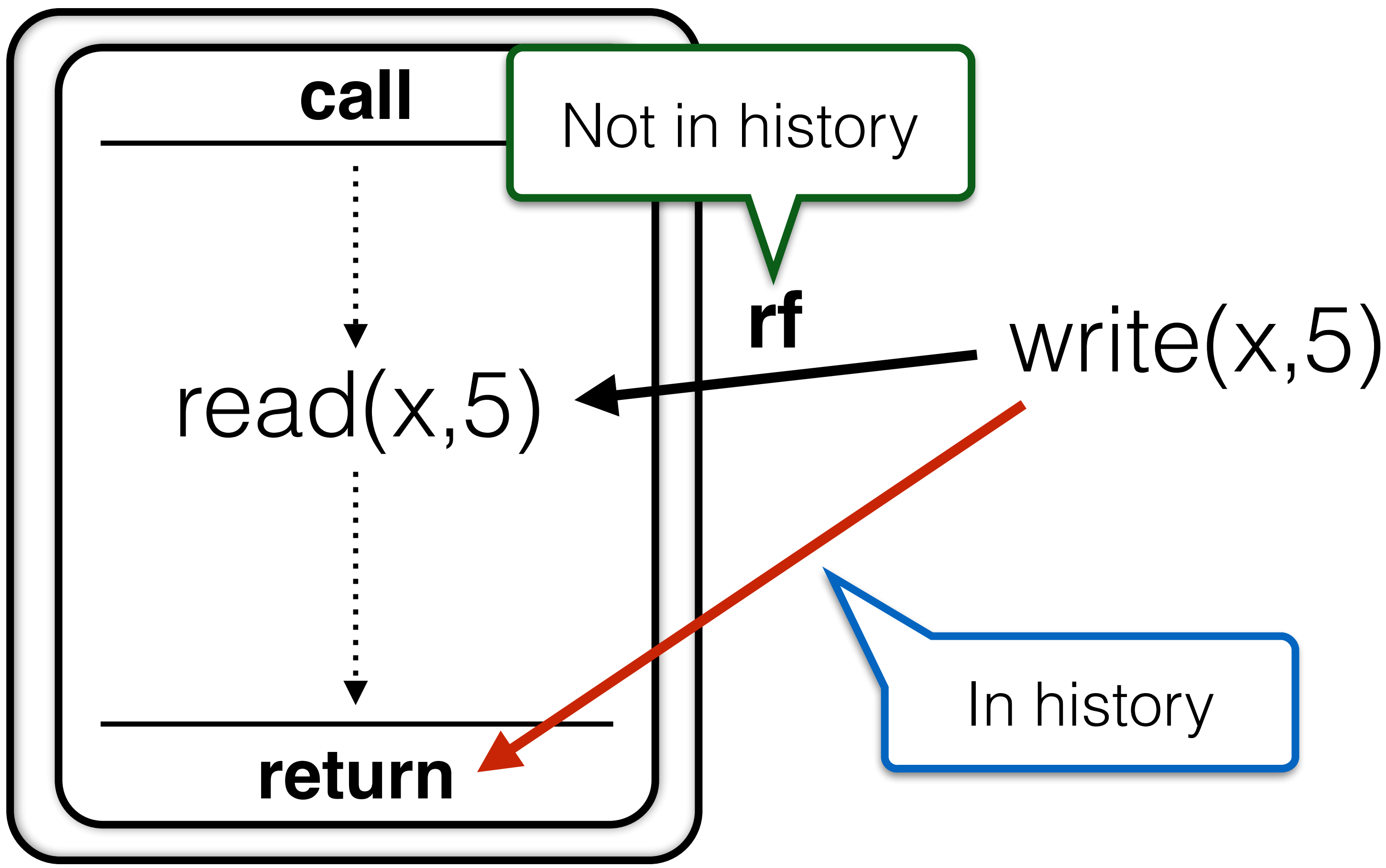
Interface actions



History

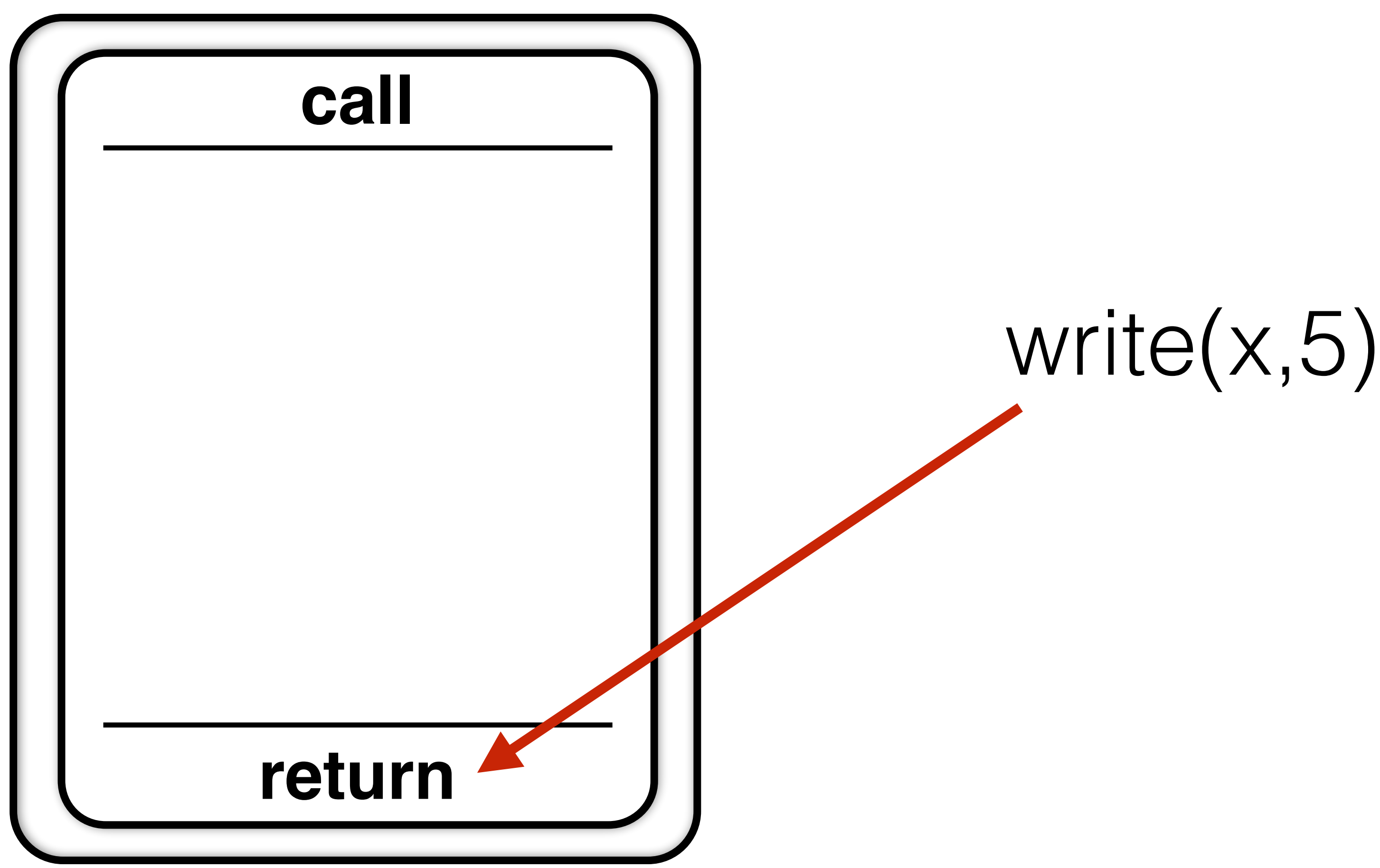


History



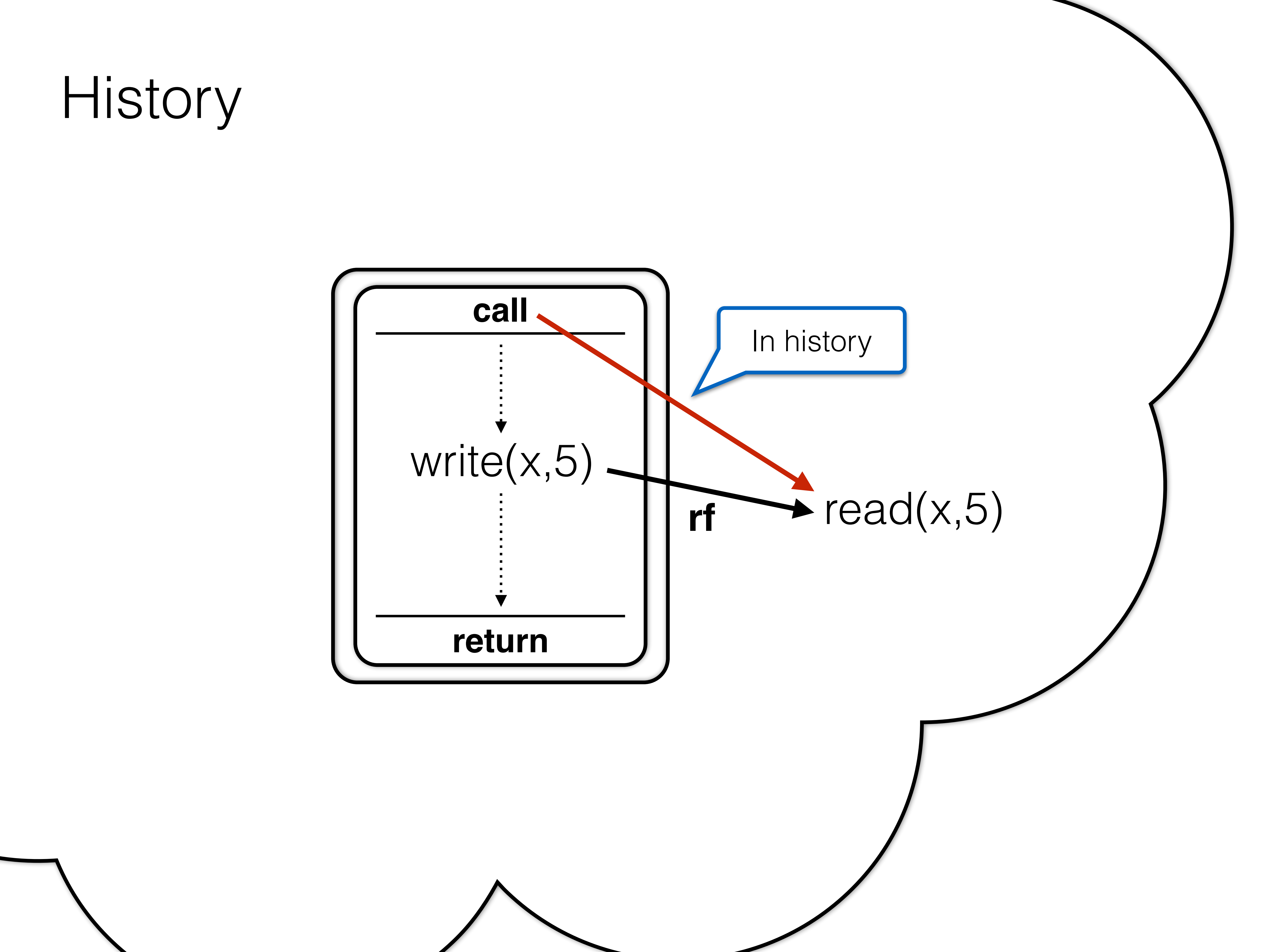
Don’t record internal actions in history

History

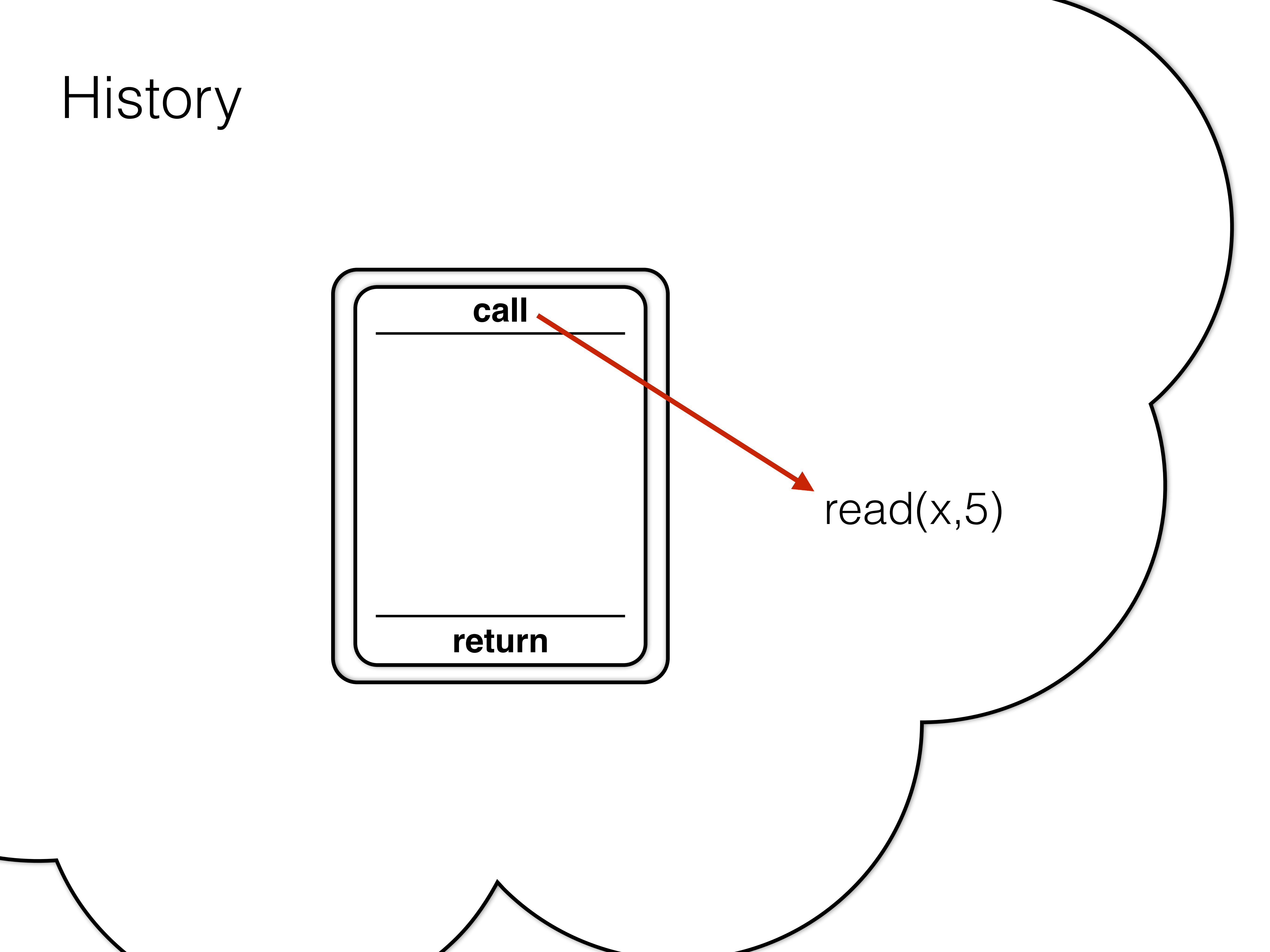


Don’t record internal actions in history

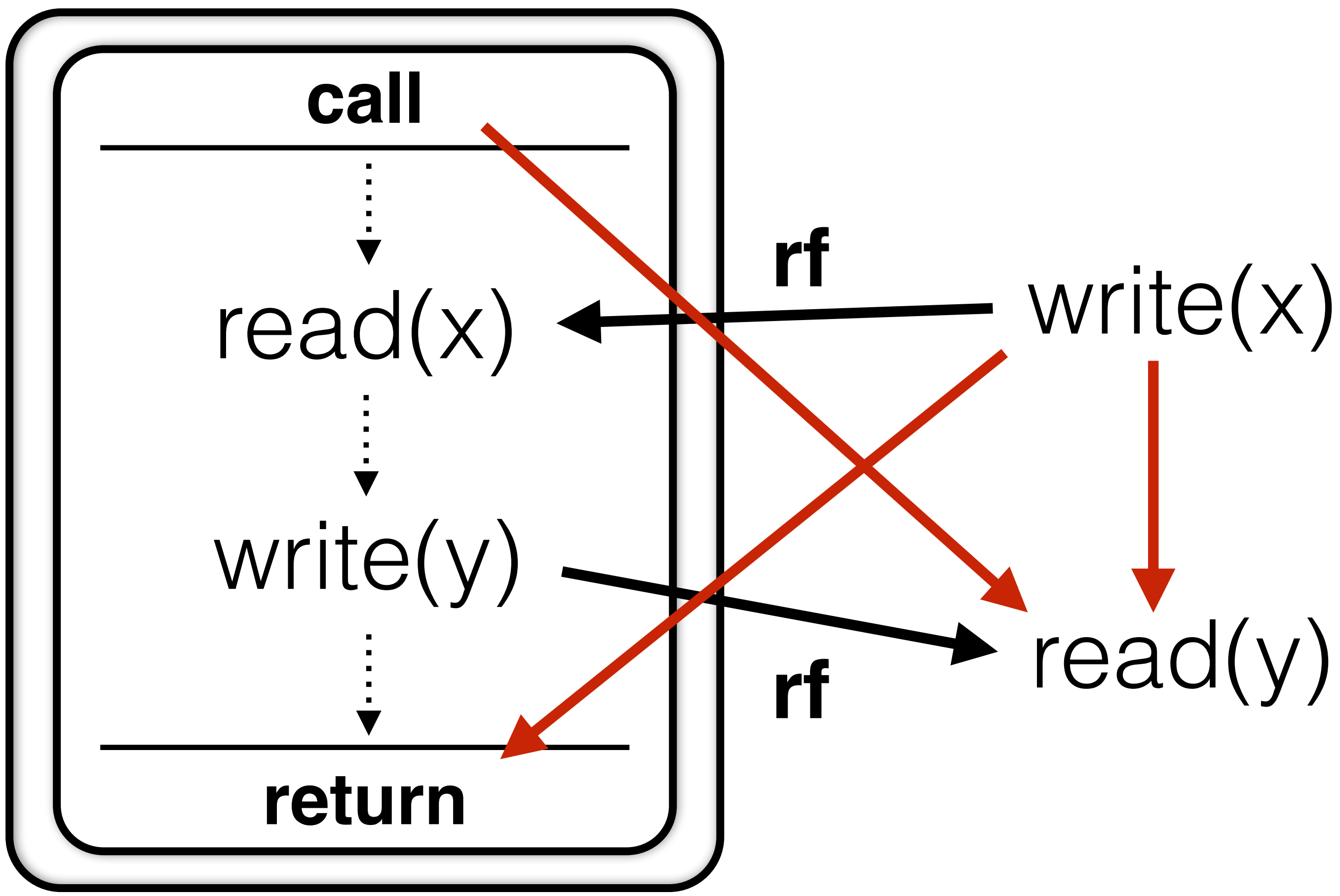
History



History

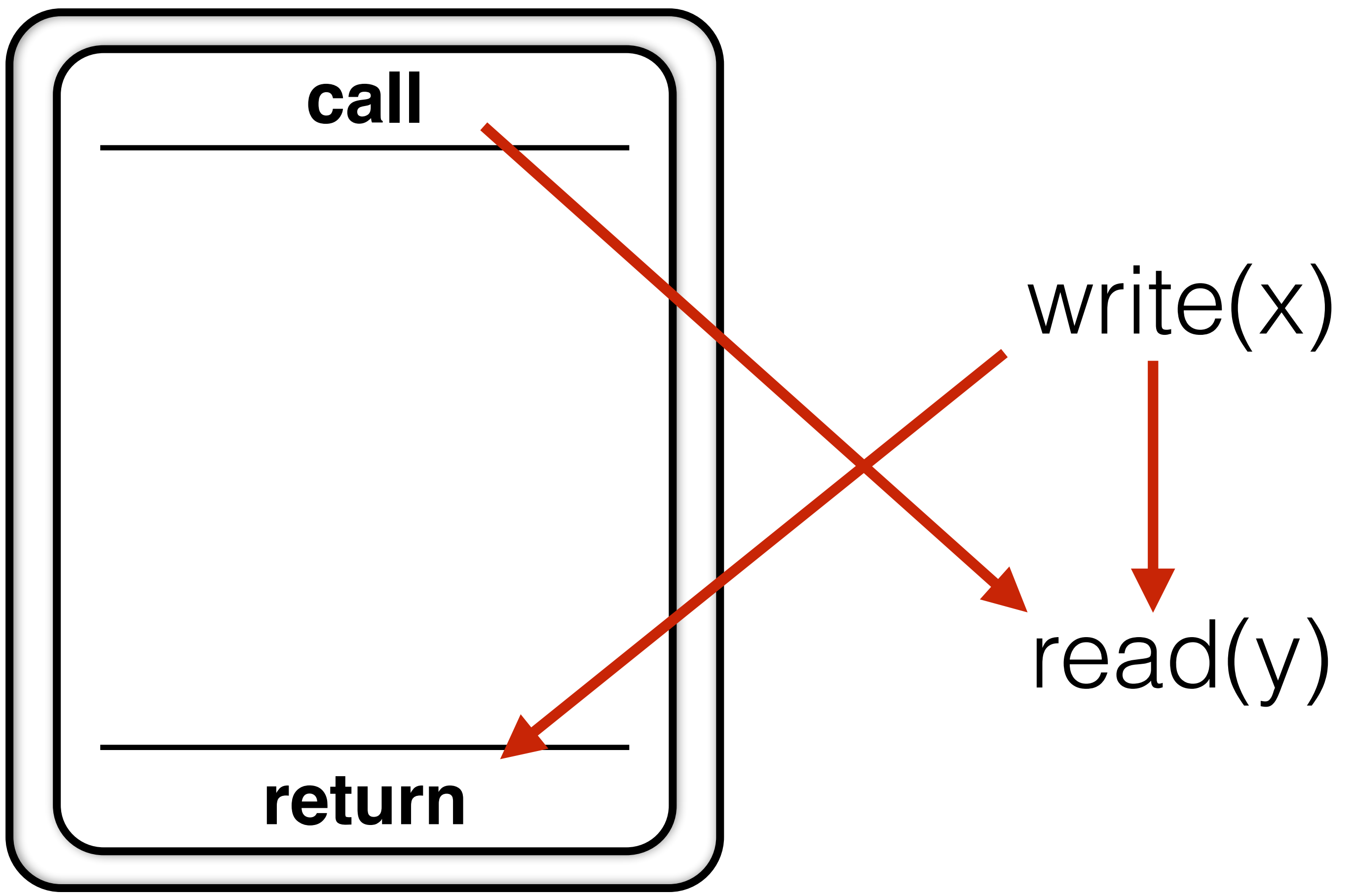


History



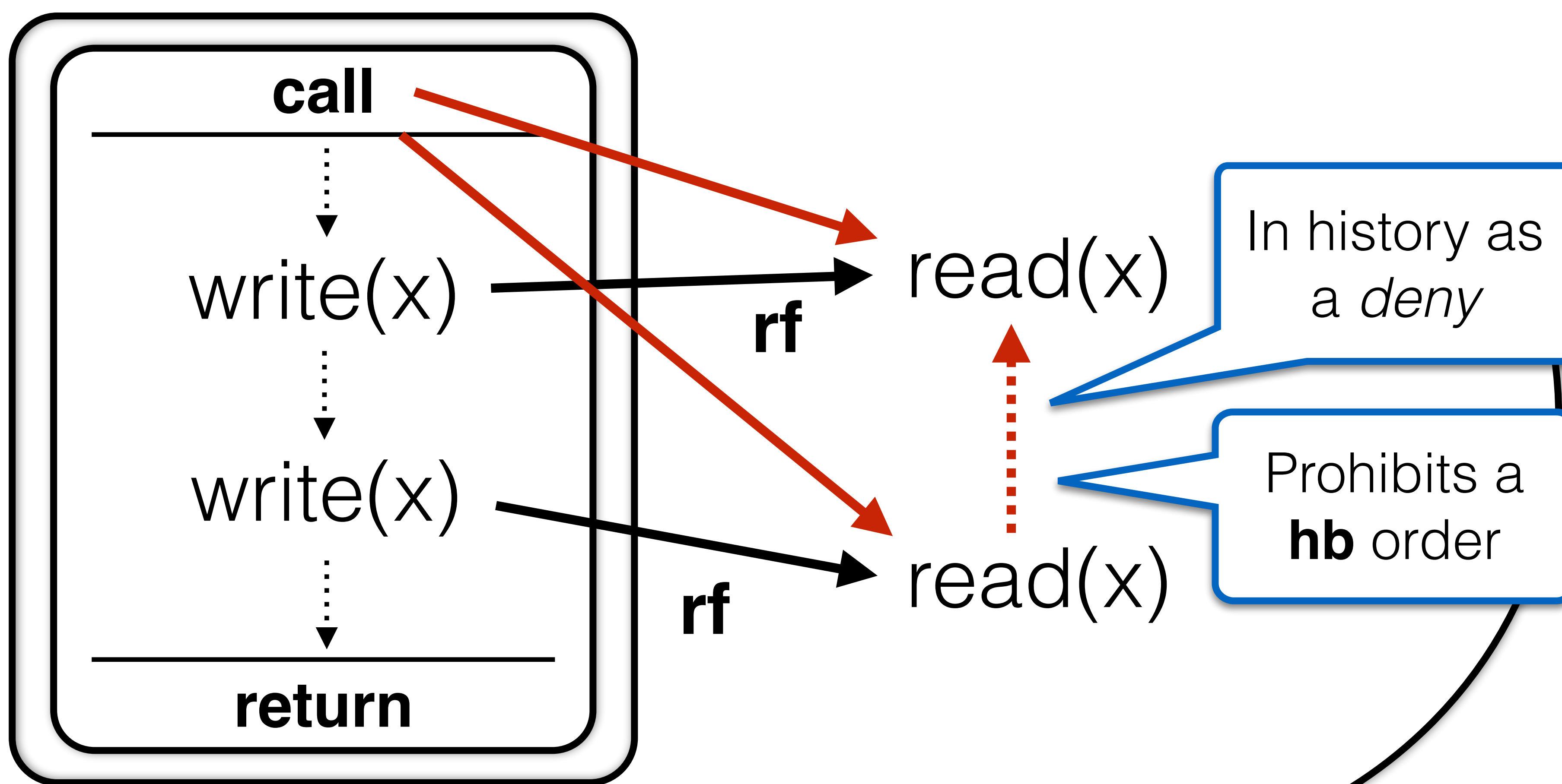
Some internal order matters!

History



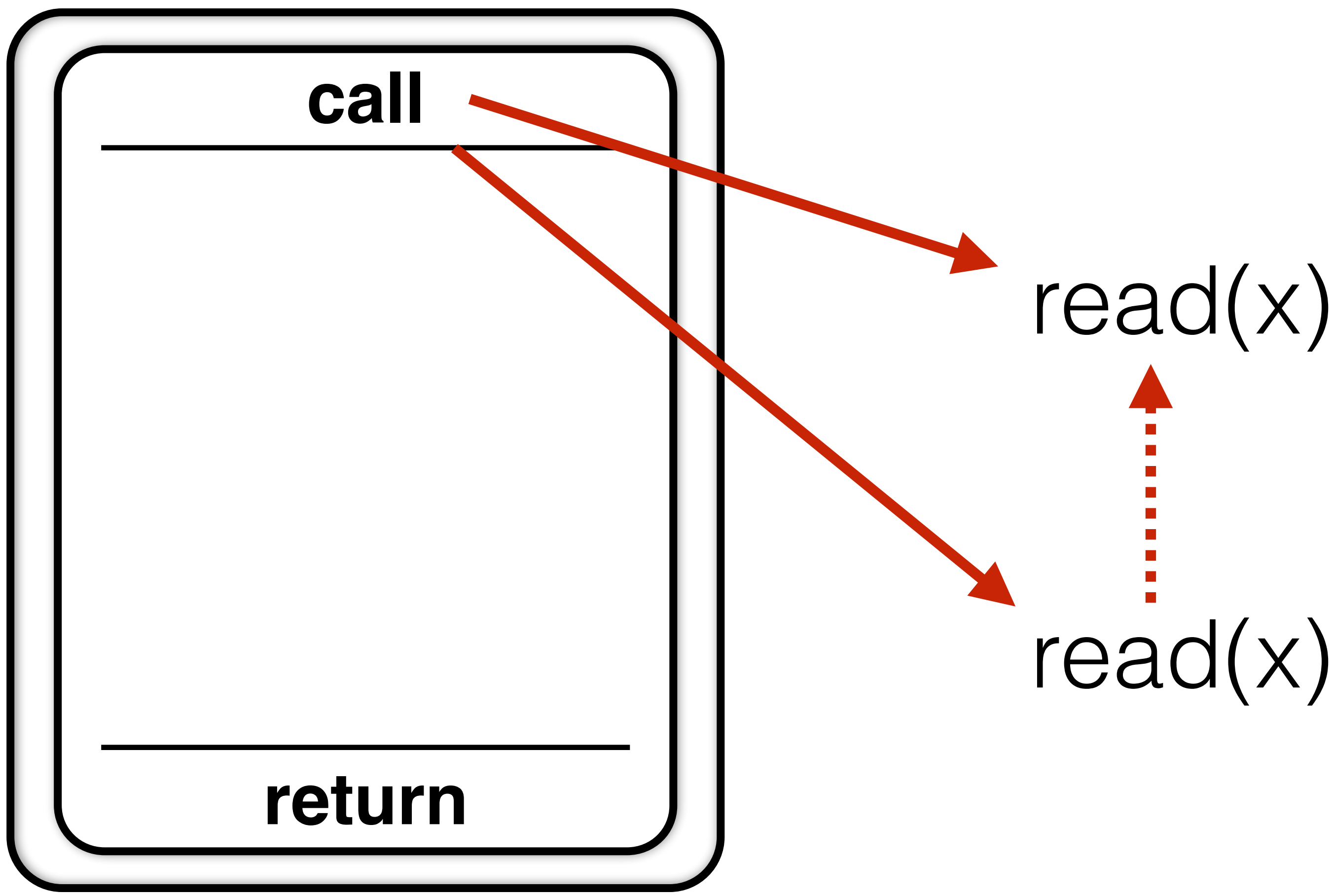
Some internal order matters!

History



Some internal order matters!

History



Some internal order matters!

Building $[\![P]\!]_d$

Interface actions

hb projection (‘*guarantee*’)

forbidden hb (‘*deny*’)

$$\text{History} : \mathcal{P}(\text{Action}) \times \mathcal{P}(\text{Action} \times \text{Action}) \times \mathcal{P}(\text{Action} \times \text{Action})$$

$$[\![P]\!]_d : \mathcal{P}(\text{History})$$

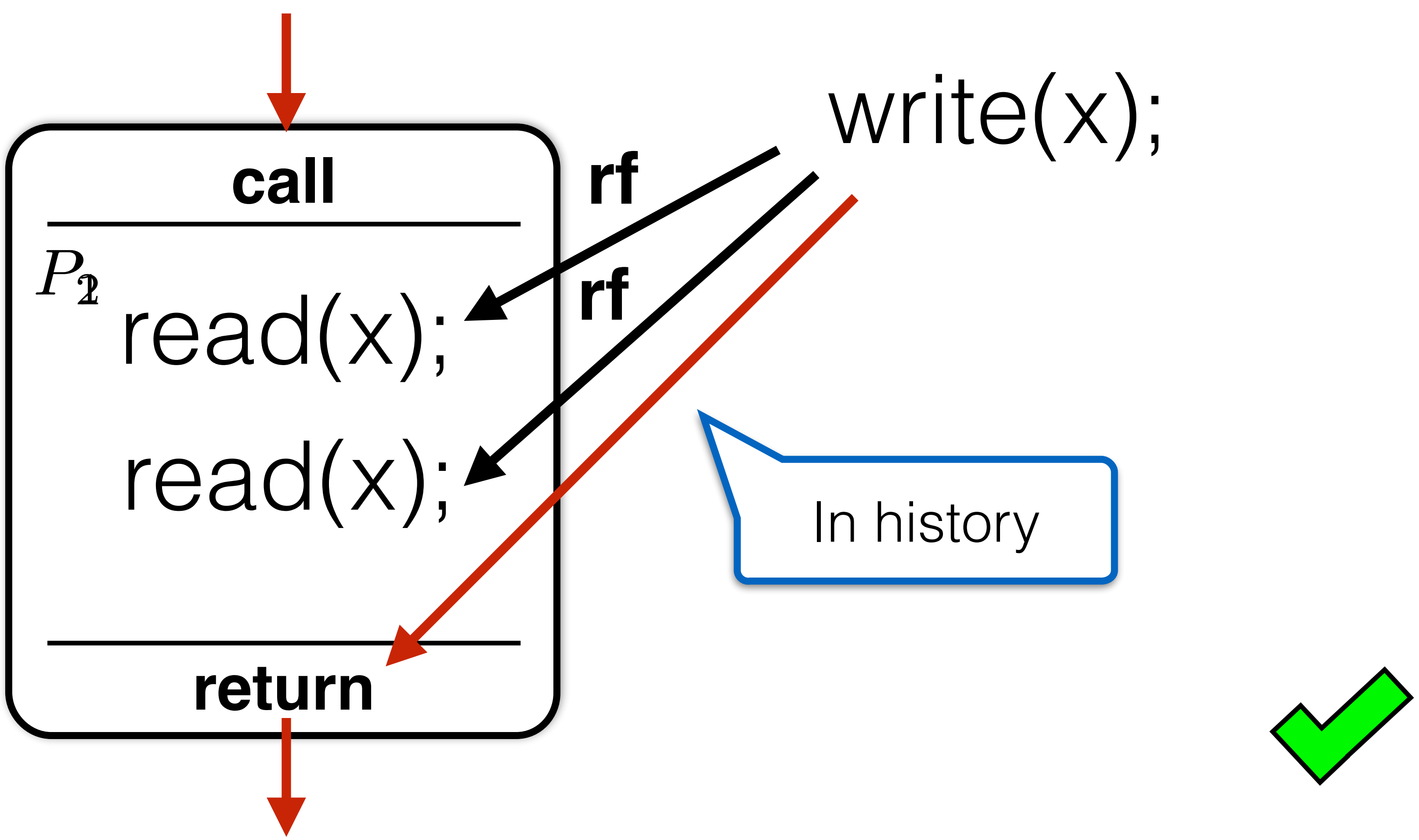
To build $[\![P]\!]_d$:

1. Generate executions in $[\![P]\!]$ for a *limited collection* of contexts.
2. Extract the history from each execution.

Validating the example

$$P_1: \begin{array}{l}\text{read(x);}\\ \text{read(x);}\end{array} \quad \rightsquigarrow \quad P_2: \text{ read(x);}$$

Show $[\![P_2]\!]_d \subseteq [\![P_1]\!]_d$:



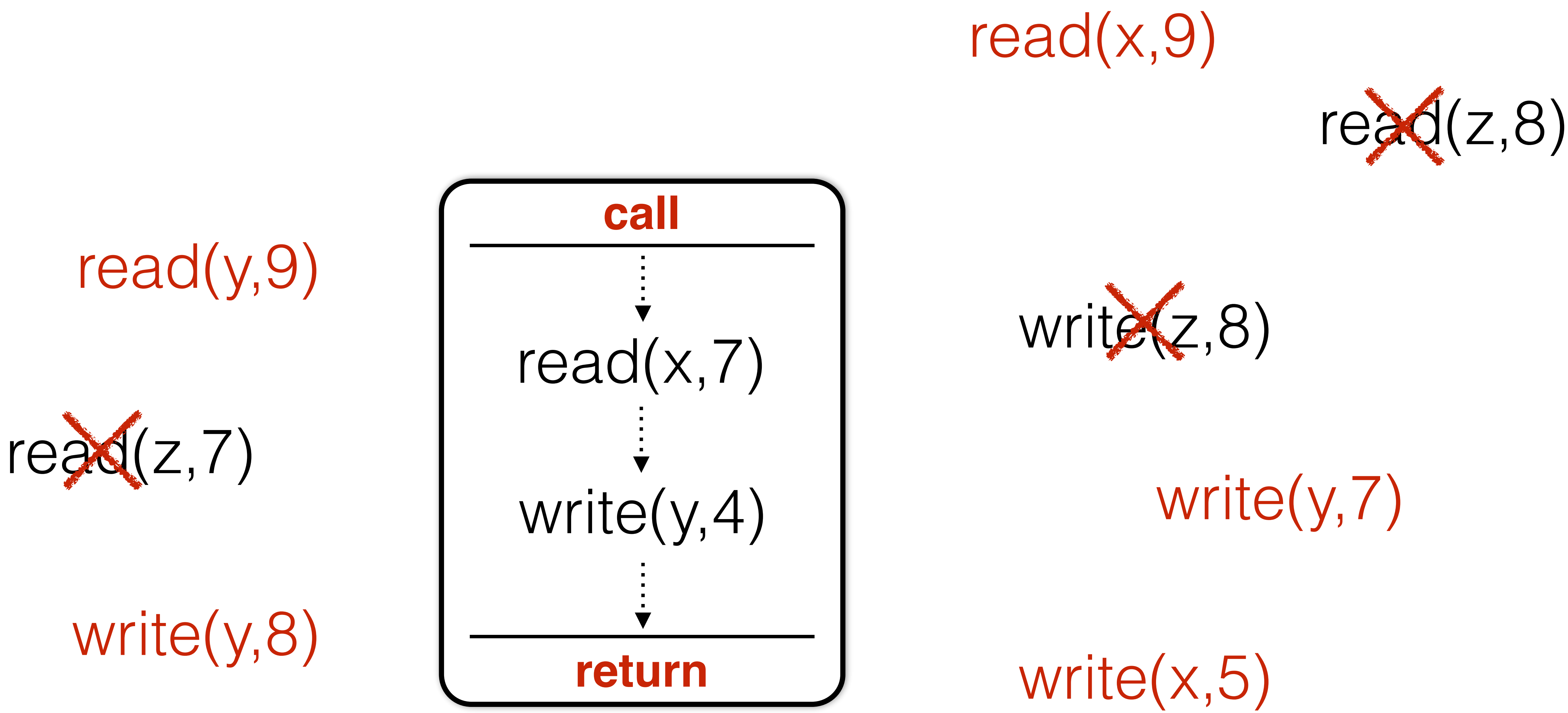
1. Objective: compositional transformation
2. C11 semantics primer
3. Defining execution histories
4. **Cutting down contexts**

Which contexts matter?

$$\forall C.\, \forall X_2 \in [\![C(P_2)]\!].\, \exists X_1 \in [\![C(P_1)]\!].\, \mathsf{obsv}(X_2) = \mathsf{obsv}(X_1)$$

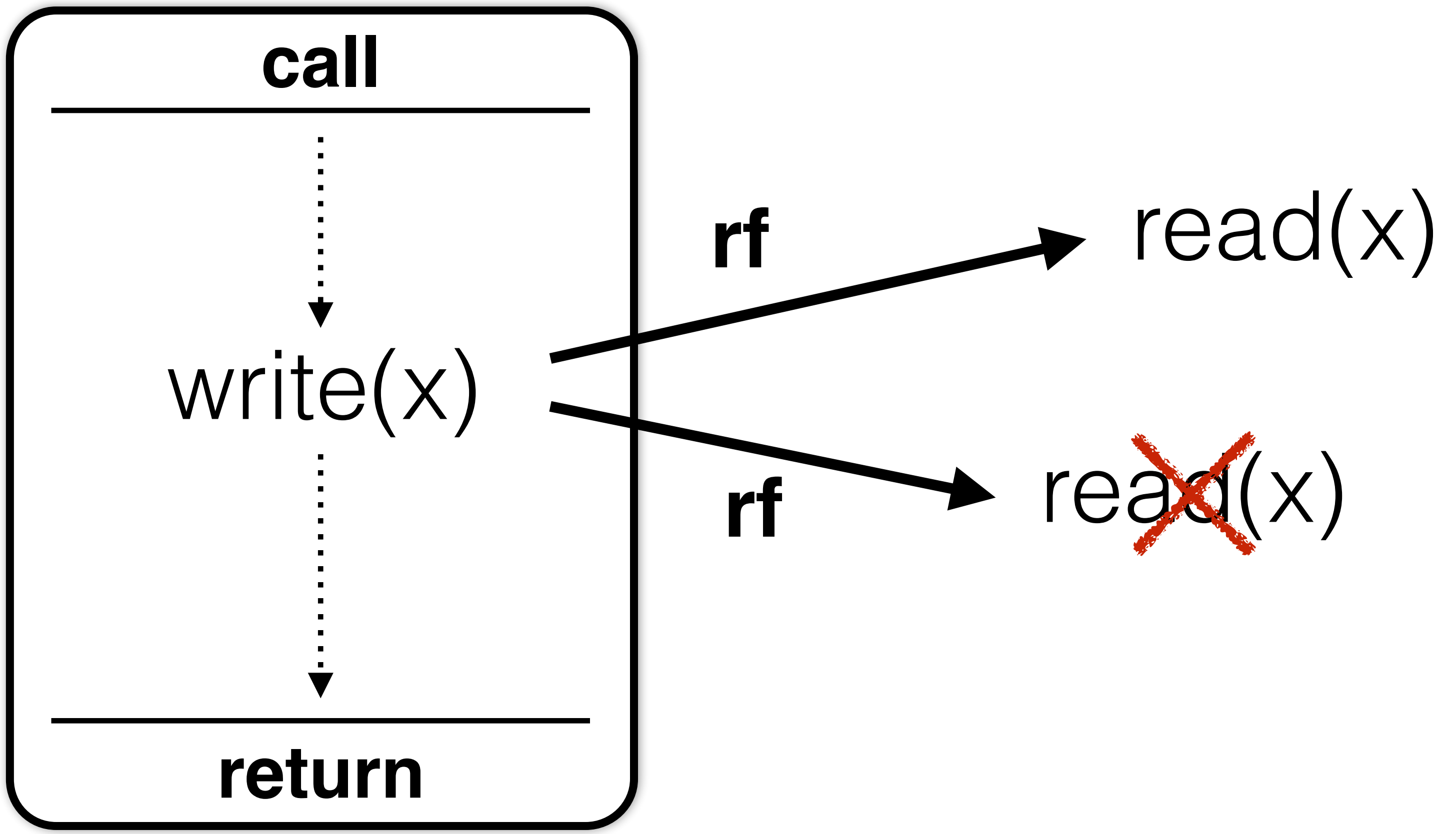
all contexts?

Which contexts matter?



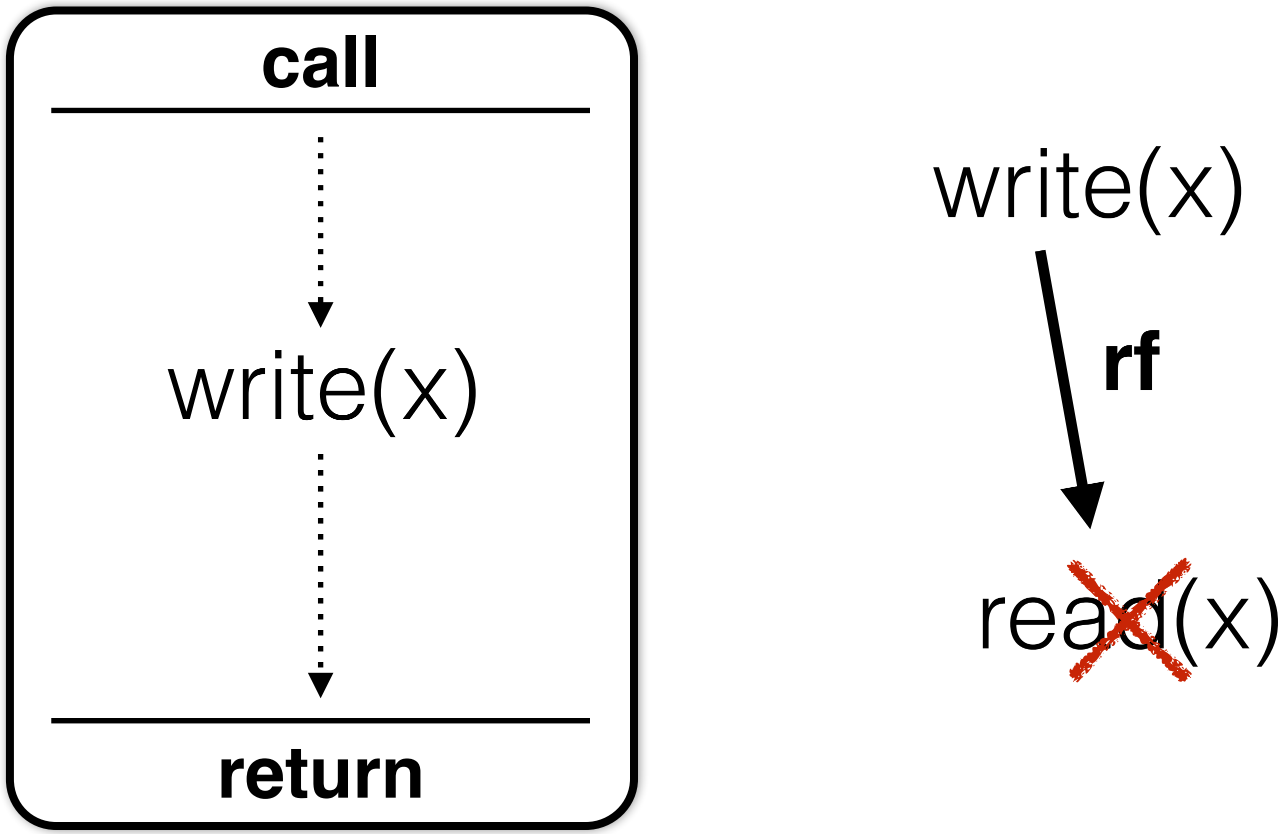
Drop non-interface context actions

Which contexts matter?



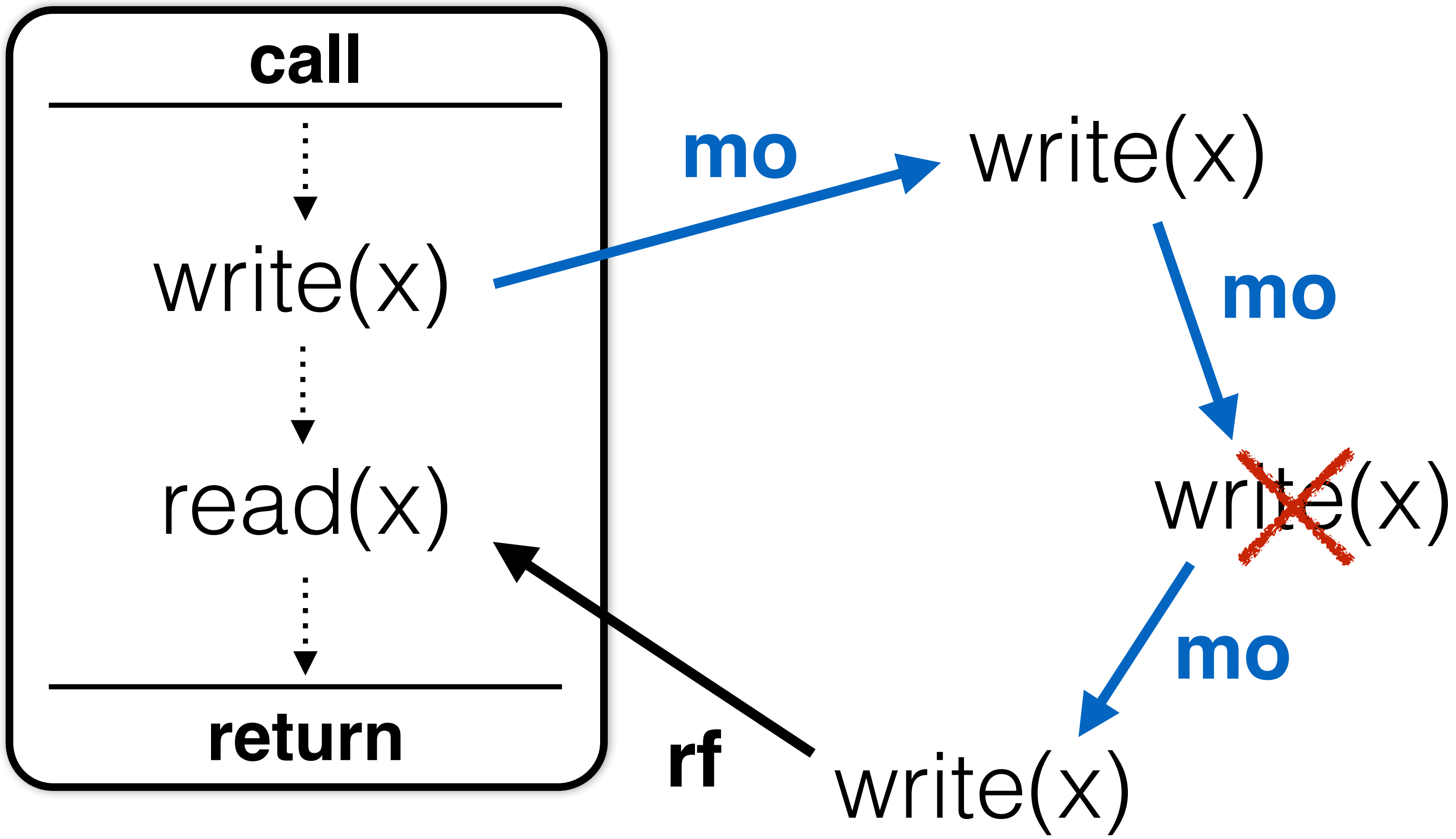
Drop duplicate interface reads

Which contexts matter?



Drop interface reads from interface writes

Which contexts matter?



Drop interface writes with siblings in modification-order

Which contexts matter?

$$\mathsf{cut}(X) \overset{\Delta}{\iff} \text{“only interface actions”} \wedge \text{“only rf-distinguished reads”} \wedge \text{“only mo-distinguished writes”}$$

P is loop-free code $\implies \{X \in [\![P]\!] \mid \mathsf{cut}(X)\}$ is finite

$\implies [\![P]\!]_d$ is finite

Current status

- Proved adequacy for a fragment of C11.
  (release-acquire, NA, working on SC)

- Validated a collection of optimisations.

- Finiteness theorem (mostly done).

- *Full abstraction* (in progress).

- *Checking tool* (planning stages).

Towards a compositional semantics?

Would like to define parallel composition:

$$[\![P_1 \| P_2]\!]_d \stackrel{\mathsf{def}}{=} [\![P_1]\!]_d \oplus [\![P_2]\!]_d$$

Would (maybe) like full abstraction:

$$\forall C.\, \forall X_2 \in [\![C(P_2)]\!].\, \exists X_1 \in [\![C(P_1)]\!].\, \mathsf{obsv}(X_2) = \mathsf{obsv}(X_1)$$
$$\implies \quad [\![P_2]\!]_d \subseteq [\![P_1]\!]_d$$