Phantom Monitors:
A Simple Foundation for Modular Proofs of Fine-Grained Concurrent Programs

Christian J. Bell, Mohsen Lesani, Adam Chlipala,
Stephan Boyer, Gregory Malecha, Peng Wang

MIT CSAIL

cj@csail.mit.edu

Nov 13th 2015

Observes all operations on a
shared data structure and
enforces invariants

Phantom Monitor

Exists only in the instrumented
semantics and is removed
during “erasure”

Top-level Goals

- verifying fine-grained concurrent **data structures**
- verifying concurrent **clients** of shared data structures
- use a program logic that supports thread-local reasoning
    - but allows global reasoning when necessary
- end-to-end machine-checked proofs

Outline

- Motivation: what does the verification problem look like?

- Our framework
    - Method
    - Architecture
    - Phantom monitors

Motivation

Goal: verification of fine-grained concurrent data structures…

```
fun push(head, v)
    node = alloc 2
    write node.item v
    while true
        oldHead = read head
        write node.next oldHead
        if cas head oldHead node = 1 then
            return


fun pop(head)
    while true
        oldHead = read head
        if oldHead = 0 then
            return None
        else
            newHead = read oldHead.next
            if cas head oldHead newHead = 1 then
                v = read oldHead.item
            return (Some v)
```

Goal: verification of fine-grained concurrent data structures…

```
{Stack head σ * F}
fun push(head, v)
    .
    .
    .
    .
    .
{Stack head (v::σ) * F}

{Stack head (v::σ) * F}
fun pop(head)
    .
    .
    .
    .
    .
    .
    .
    .
{Stack head σ * F}
```

Goal: … and their clients

```
global stack jobs

fun client(input)
    foreach v in input
        push jobs (In v)
    results = []
    while length(results) < length(input)
        x = pop jobs
        case x = Some (Out v): results := v :: results
        case x = Some (In v): push jobs (In v)
    return results

fun worker(compute)
    while true
        x = pop jobs
        case x = Some (In v): push jobs (Out (compute v))
        case x = Some (Out v): push jobs (Out v)

fun testCase()
    for 1..4 do
        fork worker(factorial)
    client_pid = fork client([1,2,3,4])
    results = join client_pid
    assert (sum results = 33)
```

Goal: … and their clients

```
{Stack jobs $\sigma$ * F}
fun client$_f$(input)
    .
    .
    .
    .
    .
{Stack jobs $\sigma'$ * results $=_{perm}$ map $f$ input * F}
```

```
{Stack jobs $\sigma$ * F}
fun worker(compute)
    .
    .
{False}
```

```
fun testCase()
    for 1..4 do
        fork worker(factorial)
    client_pid = fork client([1,2,3,4])
    results = join client_pid
    assert (sum results = 33)
```

What’s missing?

- The stack specification is insufficient to prove

`{...}` $\texttt{client}_f$`(input) {Stack jobs` σ' `* results` $=_{perm}$ `map` f `input * F}`

where

`{Stack head` σ `* F} push(head, v) {Stack head (v::`σ`) * F}`

`{Stack head (v::`σ`) * F} pop(head) {Stack head` σ `* F}`

- Some challenges:
  - the stack is multiplexed for different uses (In vs Out)
  - specialized thread roles (client vs worker)
  - the properties we need to enforce are *global*
    - client: how does interference compute *f*?

General Challenges

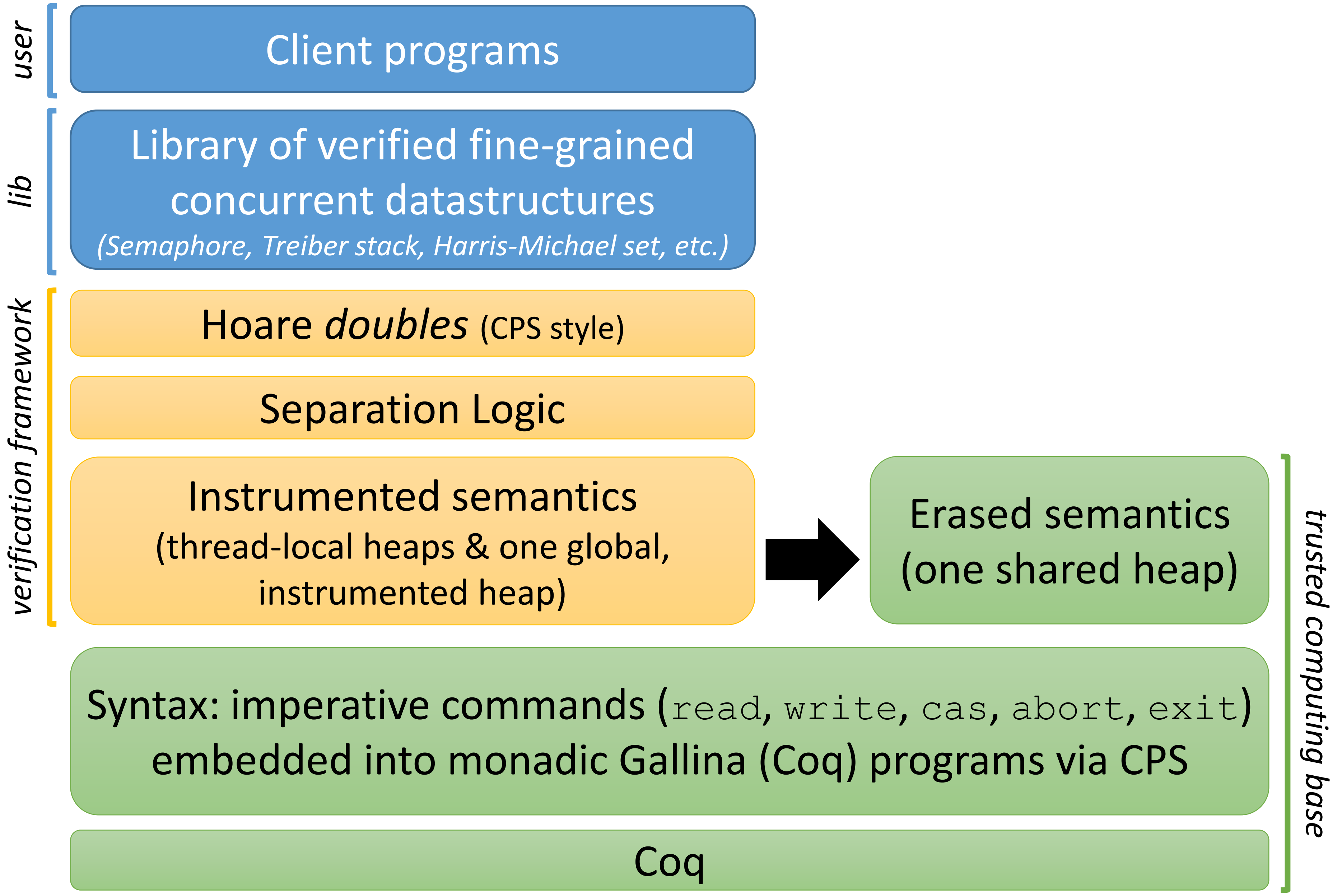
1. What does the concurrent program logic look like?
    - **abstraction**: high level
    - **local reasoning**: modular/manageable proofs
    - **generality**: can we prove real & interesting programs?
2. End-to-end verification
    - what does the **machine code** actually do?
    - can we **trust** our program logic?
3. Verification framework development
    - how do we **quickly test** new ideas?

Framework

Method

- Minimal operational semantics
  - Syntax: Imperative commands + Gallina programs
  - Erased & Instrumented semantics
- Minimal instrumentation for global state
  - “Phantom monitors” vs ghost state
- Verification framework is built on top
- Machine-checked proofs in Coq

Architecture



user

Client programs

lib

Library of verified fine-grained concurrent datastructures

(Semaphore, Treiber stack, Harris-Michael set, etc.)

verification framework

Hoare *doubles* (CPS style)

Separation Logic

Instrumented semantics (thread-local heaps & one global, instrumented heap)

Erased semantics (one shared heap)

trusted computing base

Syntax: imperative commands (`read`, `write`, `cas`, `abort`, `exit`) embedded into monadic Gallina (Coq) programs via CPS

Coq

Trusted Computing Base

$$\begin{array}{rcl} v \in V & & \text{Value} \\ a \in A & & \text{Address} \\ \alpha \in O & ::= & \texttt{read}\ a \mid \texttt{write}\ a\ v \mid \texttt{cas}\ a\ v_0\ v_1 \\ s \in S & :=: & x \leftarrow \alpha; s \mid \texttt{nil} \mid \texttt{abort} \end{array}$$

(a) Syntax

$$\begin{array}{rcll} i \in I & & & \text{Thread ID} \\ h \in H & = & A \rightharpoonup V & \text{Heap} \\ P & = & I \rightharpoonup S & \text{Processes} \end{array}$$

(b) Semantic domains

$$\frac{h(a) = v}{(h, \texttt{read}\ a) \rightarrow_{\mathrm{h}} (h, v)} \qquad \frac{h(a) = v_1}{(h, \texttt{cas}\ a\ v_1\ v_2) \rightarrow_{\mathrm{h}} (h[a \mapsto v_2], 1)}$$

$$\frac{a \in \mathrm{dom}(h)}{(h, \texttt{write}\ a\ v) \rightarrow_{\mathrm{h}} (h[a \mapsto v], 1)} \qquad \frac{a \in \mathrm{dom}(h) \qquad h(a) \neq v_1}{(h, \texttt{cas}\ a\ v_1\ v_2) \rightarrow_{\mathrm{h}} (h, 0)}$$

$$\frac{(h, \alpha) \rightarrow_{\mathrm{h}} (h', v)}{(h, P \uplus [i \mapsto x \leftarrow \alpha; s]) \underset{i}{\rightarrow} (h', P \uplus [i \mapsto s[v/x]])}$$

$$\frac{(h, P) \underset{i}{\rightarrow} (h', P')}{(h, P) \rightarrow (h', P')}$$

(c) Erased operational semantics

$$\begin{array}{c} \forall i.\ P(i) \neq \texttt{nil} \Rightarrow \exists h', P'.\ (h, P) \underset{i}{\rightarrow} (h', P') \\ \forall h', P'.\ (h, P) \rightarrow (h', P') \Rightarrow \text{safe-program}\ h'\ P' \\ \hline\hline \text{safe-program}\ h\ P \end{array}\ \text{\textsc{Safe}}$$

(d) Safety

TCB: Syntax

```
Inductive action: Set :=
    | read: action
    | write: address -> value -> action
    | cas: address -> value -> value -> action.

CoInductive proc : Set :=
    (* safely terminated thread *)
    | p_nil: proc
    (* crashed thread *)
    | p_abort: proc
    (* perform action, then call the continuation with its result *)
    | p_act: action -> (value -> proc) -> proc.

Definition act_to_proc act:= p_act act (fun _ => p_nil).
Coercion act_to_proc : action >-> proc.
Notation “x <- a ; p" := (a (fun x => p)) (...) : proc.
Notation “a ; p" := (a (fun _ => p)) (...) : proc.
```

Treiber Stack in Coq

```
Definition try_push (head node: address) (kont: value -> proc) : proc :=
    oldHead <- read head;
    write (a_next node) oldHead;
    m <- cas head oldHead node;
    kont m.

Definition push {h:alloc_handler} (head: address) (item: value) (kont: proc) : proc :=
    node <- alloc 2;
    write (a_item node) item;
    cofix loop:=
        m <- try_push head node;
        if m =? 0 then
            loop
        else
            kont.

Definition pop (head: address) (kont: option value->proc) : proc :=
    cofix loop:=
        oldHead <- read head;
        if oldHead =? 0 then
            kont None
        else
            newHead <- read (a_next oldHead);
            m <- cas head oldHead newHead;
            if m =? 0 then
                loop
            else
                x <- read (a_item oldHead);
                kont (Some x).
```

Client in Coq

```
Fixpoint client_load_input {h: alloc_handler} (input: list value) jobs kont:=
    match input with
    | nil => kont
    | x::input' =>
        push jobs (In x);
        client_load_input input' jobs kont
    end.

Definition client_collect_results {h: alloc_handler} N jobs kont:=
    (cofix loop results :=
        if length results =? N then
            kont results
        else
            x <- pop jobs;
            match x with
            | Some (Out item) =>
                loop (item::results)
            | Some (In item) =>
                push jobs item;
                loop results
            | None =>
                loop results
            end
    ) nil.

Definition client {h: alloc_handler} (input: list value) jobs kont:=
    client_load_input input jobs;
    results <- client_collect_results (length input) jobs;
    kont h results.
```

Instrumented Global State

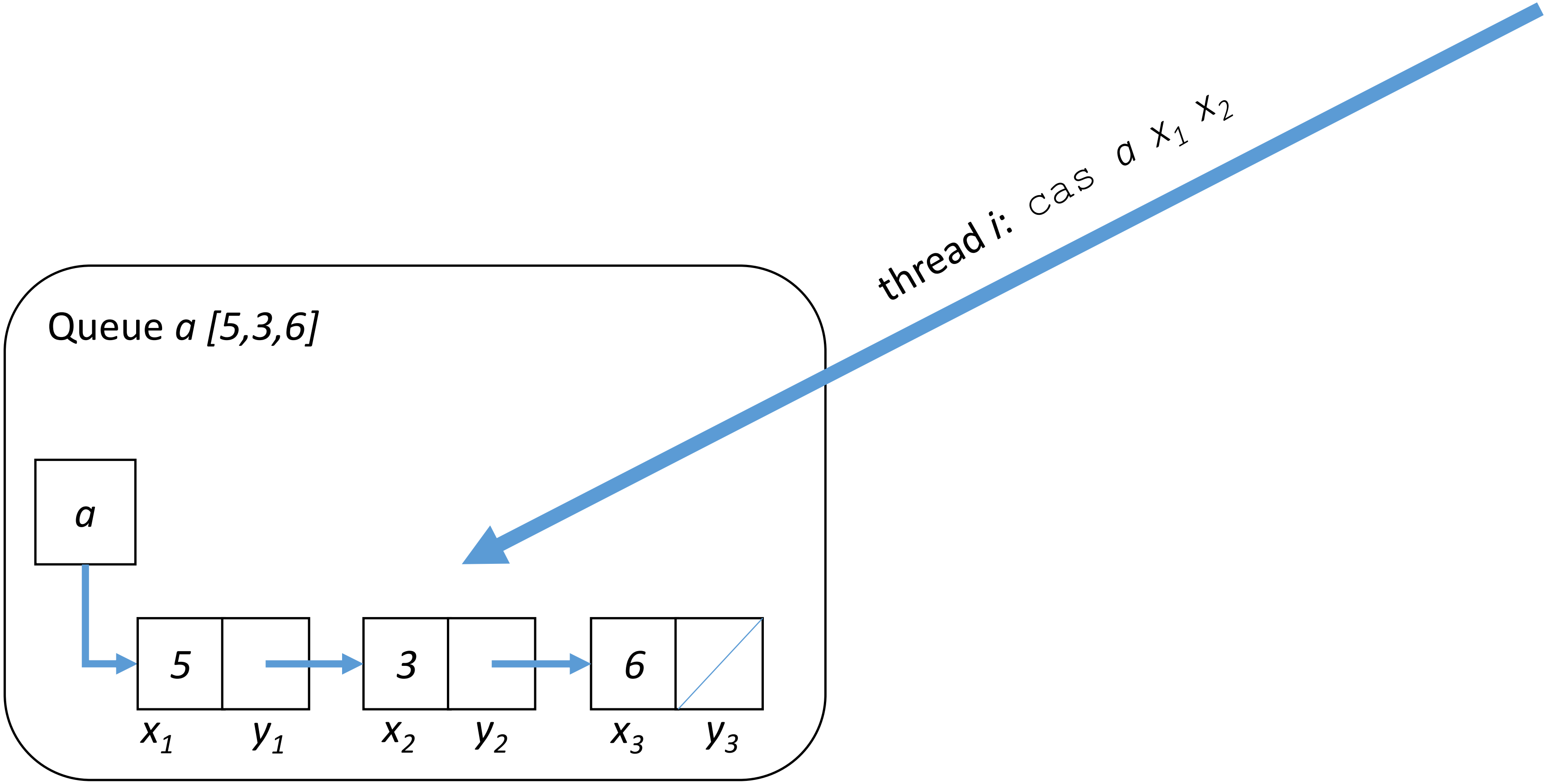
“Phantom Monitors”

- General idea: access to shared data structures is coordinated by a *global policy*:
  - what can the current thread do?
  - what can interfering threads do?

- We write a policy for a shared datastructure as a monadic [corecursive] Coq function that *monitors* every operation acting on the structure, rejecting any operation that violates the protocol, and evolving over time.

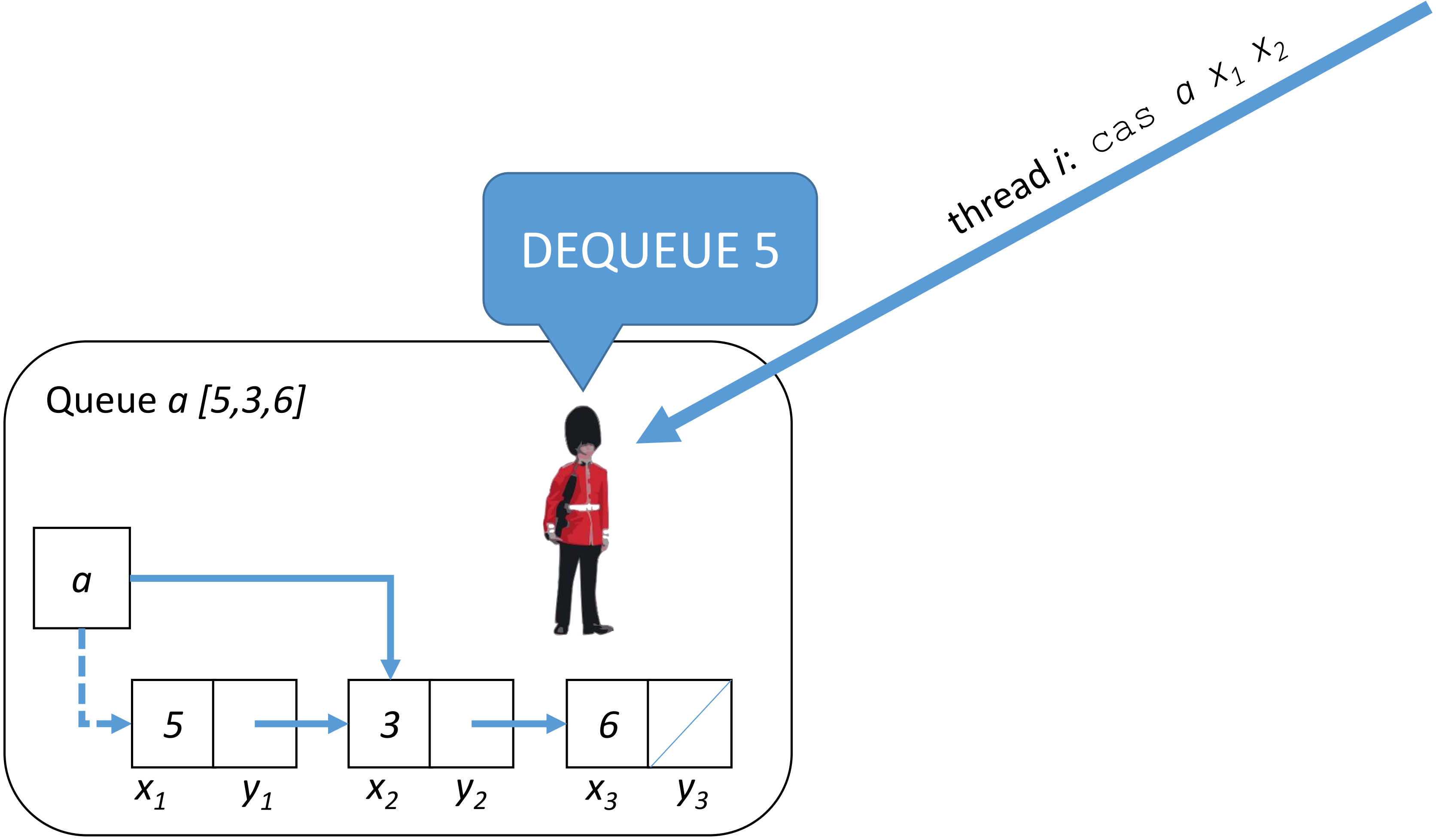
Instrumented Global State

“Phantom Monitors”



Instrumented Global State

“Phantom Monitors”



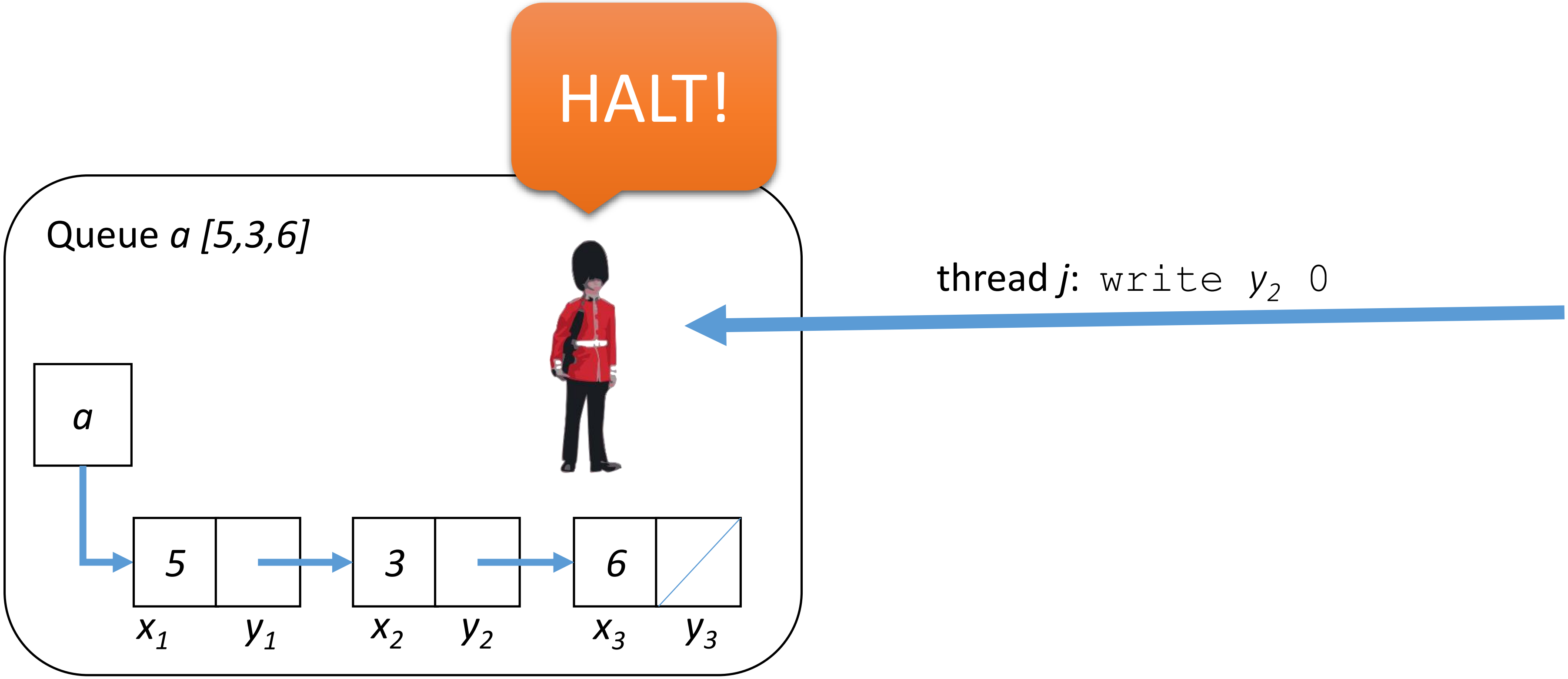
thread *i*: `cas` $a\ x_1\ x_2$

DEQUEUE 5

Queue *a [5,3,6]*

Instrumented Global State

“Phantom Monitors”



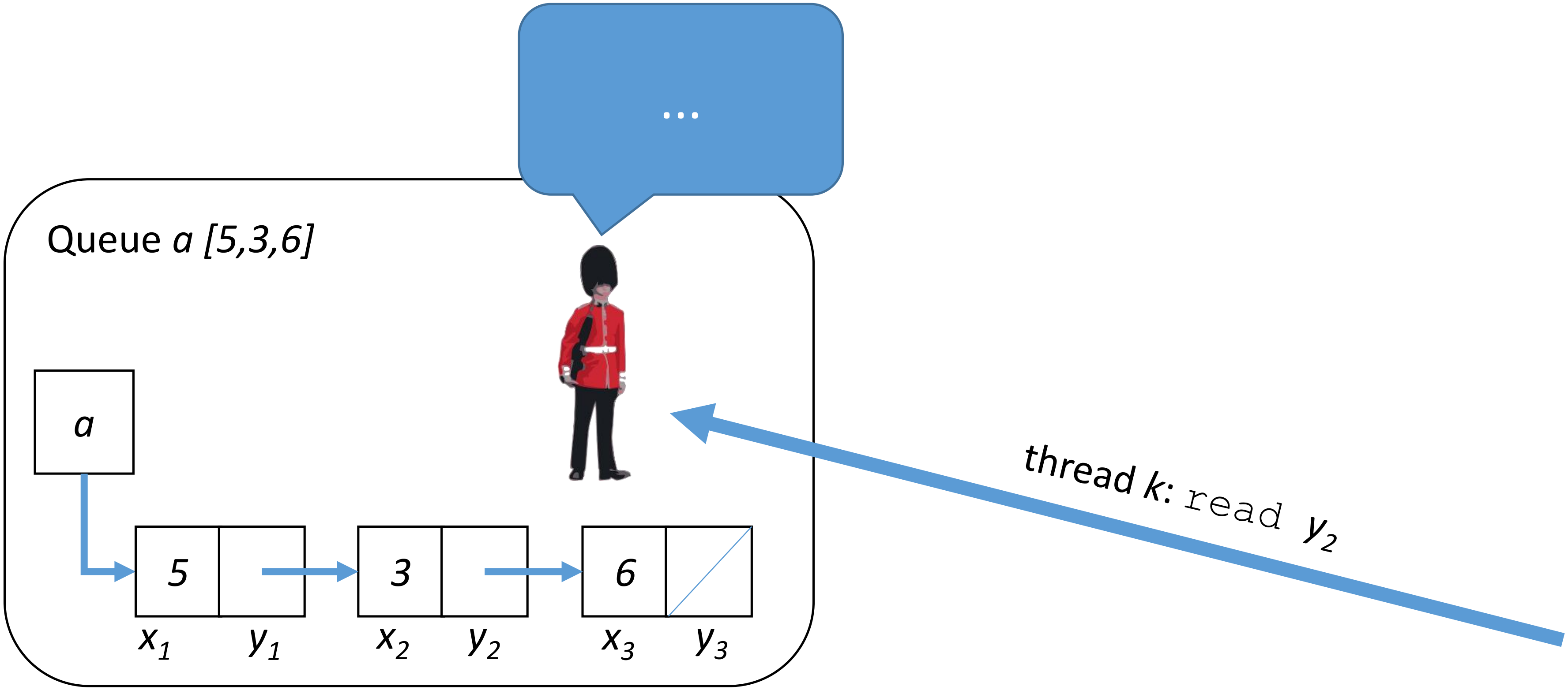
HALT!

Queue *a [5,3,6]*

thread ***j*****:** `write` y_2 `0`

Instrumented Global State

“Phantom Monitors”

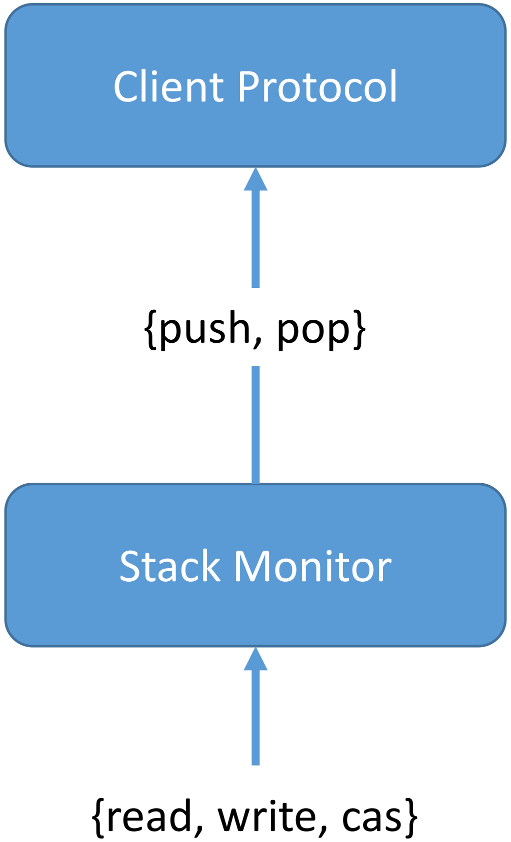


Phantom Monitors

Is a Coq function that:

1. Observes all operations on a data structure
2. Accepts or rejects each operation
3. May generate an abstract operation (“dequeue”) or silently accept it
4. Can change state
5. *Can be composed together*

Vertical Composition



Client Protocol

{push, pop}

Stack Monitor

{read, write, cas}

Abstract Client Protocol

- Definition: $\Sigma: (\mathbb{S}, \rightarrow, \sigma_0, [\![\cdot]\!])$
- Transition function: $\rightarrow \subseteq \mathbb{S} \times I \times \mathbb{A} \times \mathbb{S}$
  - $\sigma \xrightarrow[i]{\alpha} \sigma'$
- Interference: $\sigma \underset{\neq i}{\rightsquigarrow}^{*} \sigma'$

Stack Specification

$\alpha ::= \mathsf{push}\ v \mid \mathsf{pop}\ v$ and:

$$\frac{\vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma' * \mathcal{W}\}\ s}{\vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma * \mathcal{W}\}\ \mathsf{push}\ a\ v;\ s}$$

$$\frac{\begin{array}{r}\vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma\ * \mathcal{W}\}\ s\ \mathsf{None} \\ \vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma' * \mathcal{W}\}\ s\ (\mathsf{Some}\ v)\end{array}}{\vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma * \mathcal{W}\}\ x \leftarrow \mathsf{pop}\ a;\ s\ x}$$

Stack Monitor

```
protocol StackMonitor Σ (address head, $\sigma_0$) implements Monitor
  Σ σ = $\sigma_0$ (* abstract client protocol *)
  onRead(i, a, h, hAcq, hRel)
      assert hAcq = hRel = empty
  onWrite(i, a, v, h, hAcq, hRel)
      assert False
  onCAS(i, a, oldHead, newHead, h, hAcq, hRel)
      assert a = head ∧ hRel = empty
      if h(head) = oldHead then
          if h(oldHead.next) = newHead ∧ oldHead ≠ 0 then
              σ.onPop(i, h(oldHead.item))
              assert hAcq = empty
          else if hAcq(newHead.next) = oldHead then
              σ.onPush(i, hAcq(newHead.item))
              assert newHead ≠ 0 ∧ dom(hAcq) = {newHead.item, newHead.next}
          else
              assert False
      else
          assert hAcq = empty
```

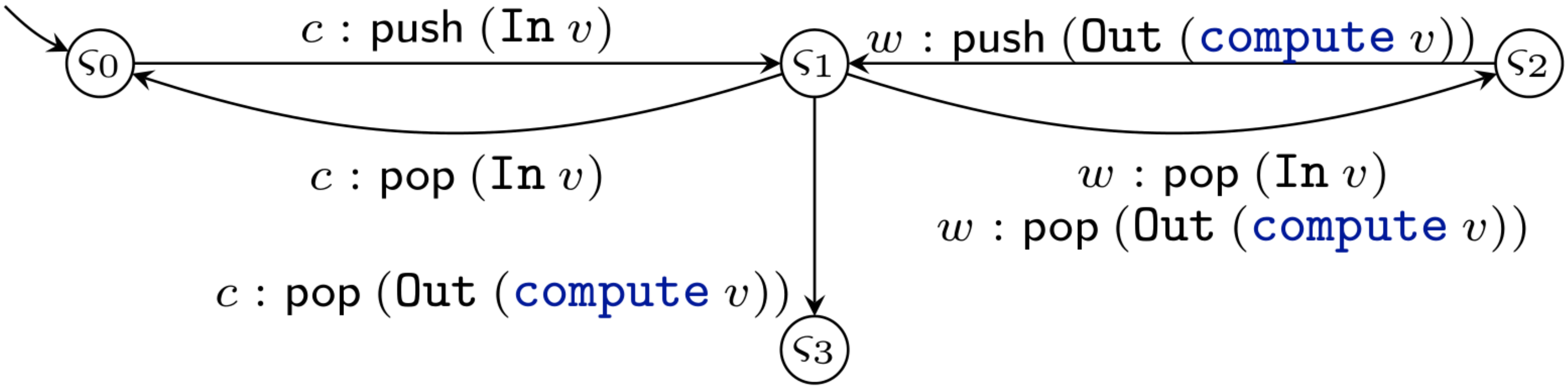
Stack

```
Definition Stack_ head hist s : predicate :=
    Ex_ node: address, Ex_ i, Ex_ s',
      pred_global
        (StackMonitor head s')
        (pts head node * ((llist node (model s') * top)
          && (and_list (observed_nodes hist))))
      * pred_pid i
      * !!(istep_star i s s').


Definition Stack head s : predicate := Stack_ head nil s.
```

Client Program Policy
(single-value case)

- client thread: c
    - pushes unfinished value into a shared stack ($\varsigma_0 \rightarrow \varsigma_1$)
    - collects the finished value ($\varsigma_1 \rightarrow \varsigma_3$)
- worker thread: w
    - checks the stack for (unfinished) values ($\varsigma_1 \rightarrow \varsigma_2$)
    - pushes the computed value of each pop ($\varsigma_2 \rightarrow \varsigma_1$)



Client Program Policy

(general case)

```
protocol JobsProto(input, client_pid, compute) implements StackProtocol
  list loading = input     // unfinished values to be pushed
  map processing = empty   // values held by worker threads

  onPush(i, x)
      case x = (Out v): // $\varsigma_2 \rightarrow \varsigma_1$
          assert processing[i] = (Out v)
                ∨  ∃ v’. processing[i] = (In v’) ∧ v = compute(v’)
          processing.remove(i)
      case x = (In v): // $\varsigma_0 \rightarrow \varsigma_1$
          assert ∃ l’. loading = v :: l’
          loading := tail(loading)

  onPop(i, x)
      if i = client_pid then
          case x = (Out v): assert True // $\varsigma_1 \rightarrow \varsigma_3$
          case x = (In v): loading := v :: loading // $\varsigma_1 \rightarrow \varsigma_0$
      else // $\varsigma_1 \rightarrow \varsigma_2$
          assert i ∉ dom(processing)
          processing.add(i, x)
```

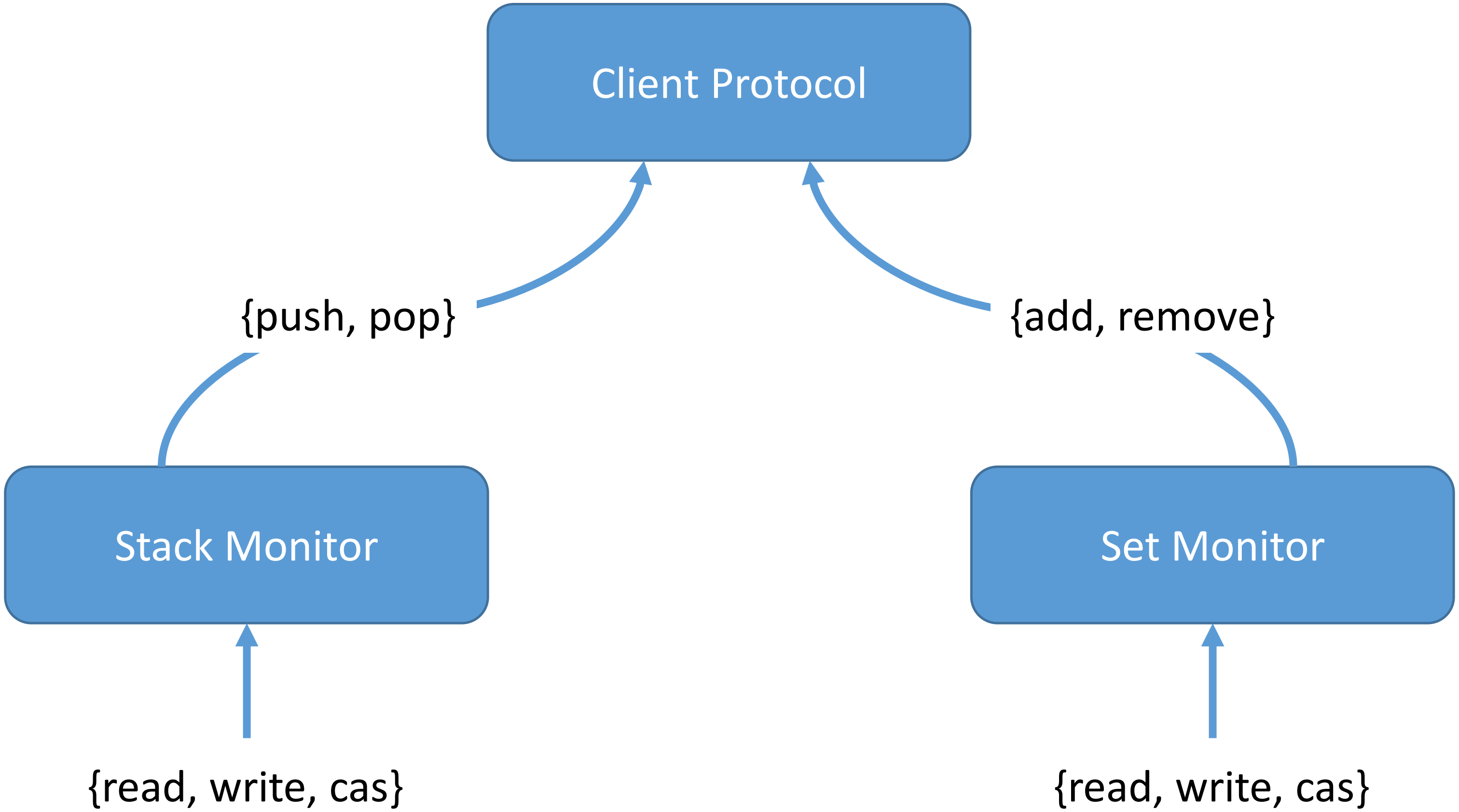
Summary

- Framework: a minimal TCB, semantically derived, proved in Coq
  - use built-in features of Coq when possible
  - avoid baking in features
    - derive permissions, PCM monitors, etc. as necessary
- Phantom monitors
  - global policies are describes by pure (monadic) functions
  - lightweight; straightforward erasure
  - when we want to see how the policy evolves, we simply run the function

What else?

- Harris-Michael lazy lock-free set algorithm
- Horizontal composition of monitors
- Coinductive Hoare doubles
- Ltac automation

Horizontal Composition



Client Protocol

{push, pop}

{add, remove}

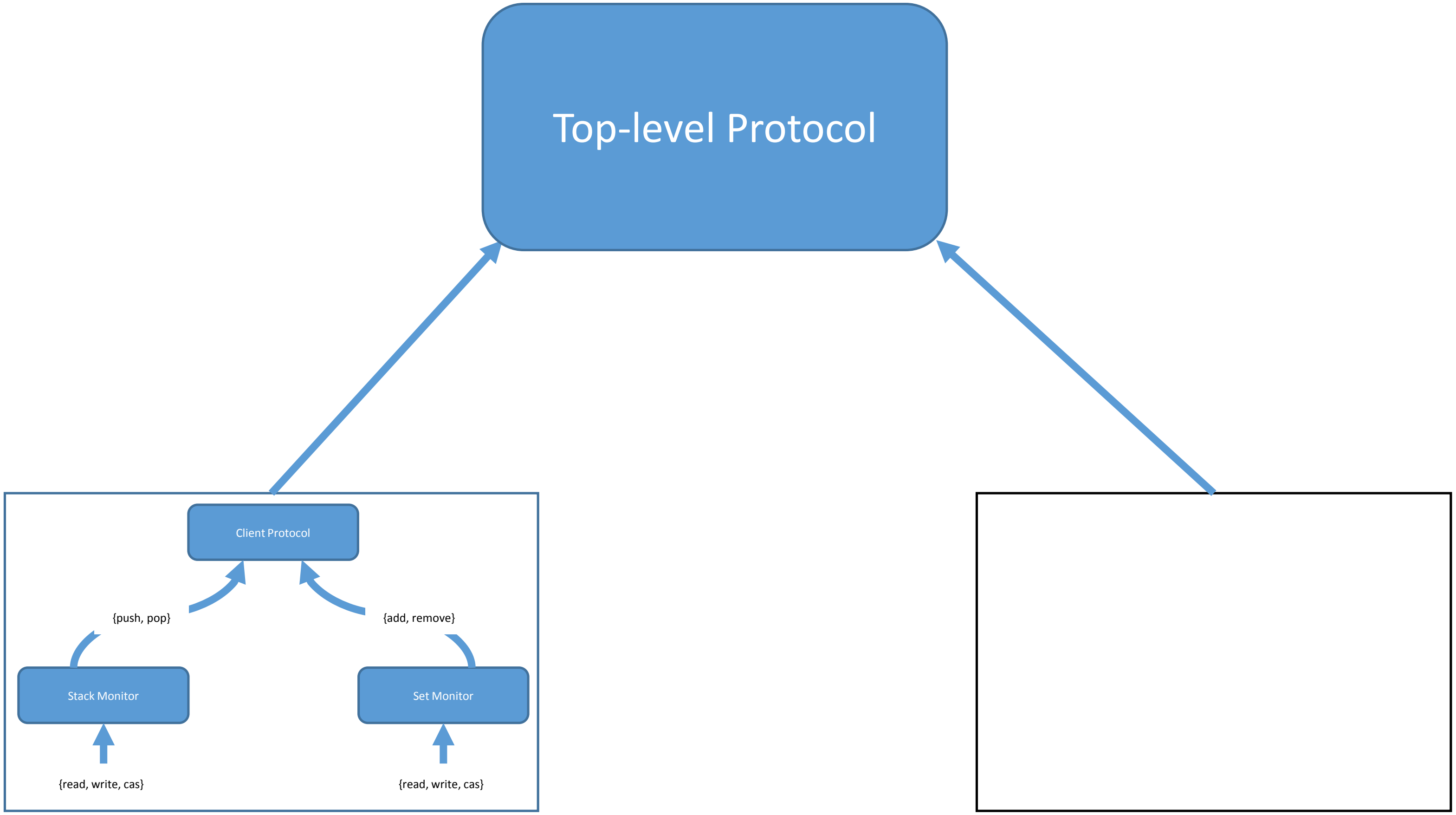
Stack Monitor

Set Monitor

{read, write, cas}

{read, write, cas}

Horizontal Composition



What else?

- Harris-Michael lazy lock-free set algorithm
- Horizontal composition of monitors
- *Logical* rule for monitor allocation
- Coinductive Hoare doubles
- Ltac automation

Thanks!