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

Goal: verification of concurrent client programs…

```
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 verification of fine-grained concurrent datastructures

```
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)
```

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?

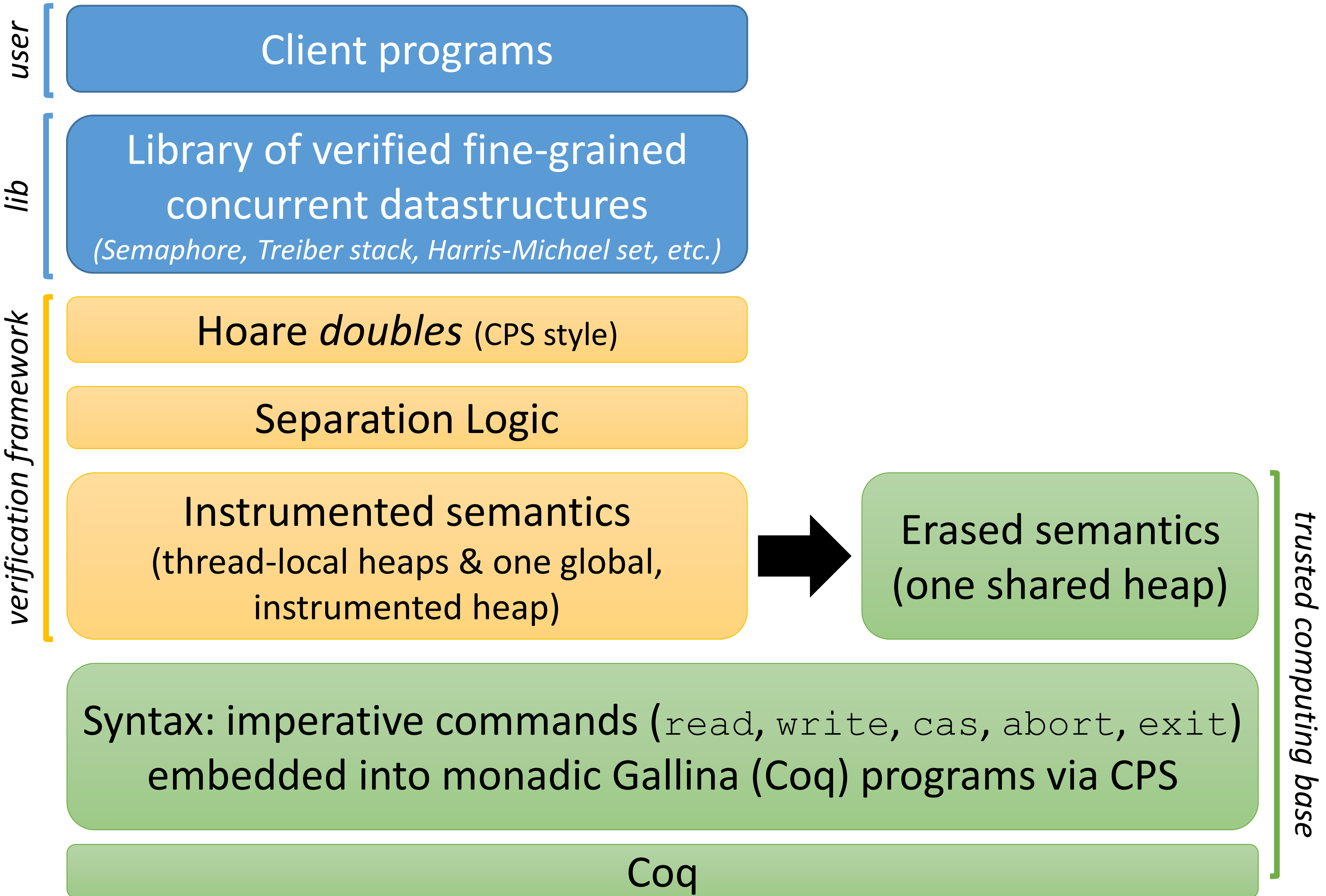
Our Focus

1. What does the concurrent program logic look like?
    - **abstraction**: high level
    - **local reasoning**: modular/manageable proofs
    - **generality**: can we prove real 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?

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

Method



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

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

(a) Semantic domains

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

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

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

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

(b) Erased operational semantics

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

(c) 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.
```

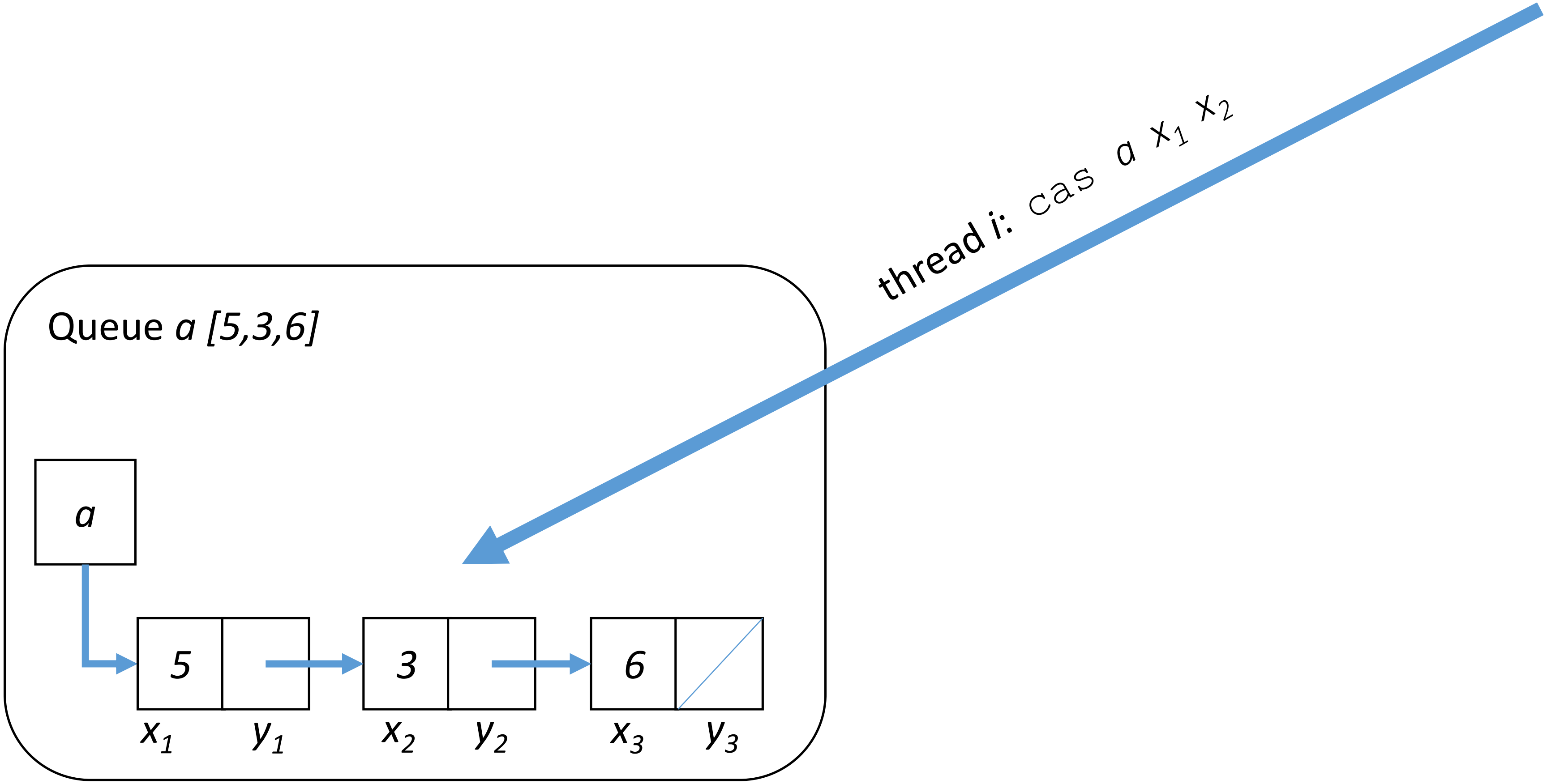
Instrumented Global State
“Phantom Monitors”

- 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”

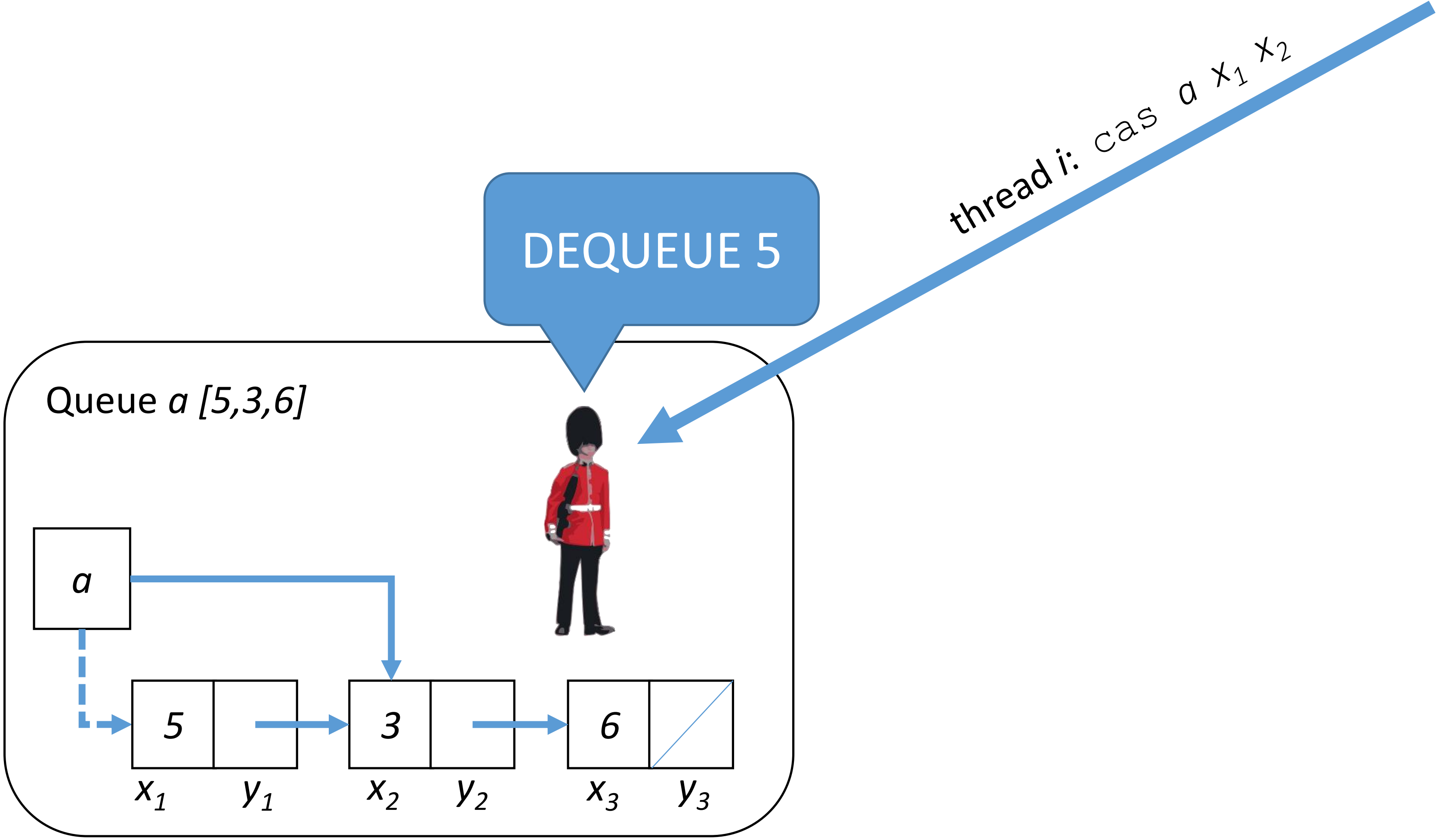


thread *i*: `cas` a x_1 x_2

Queue *a [5,3,6]*

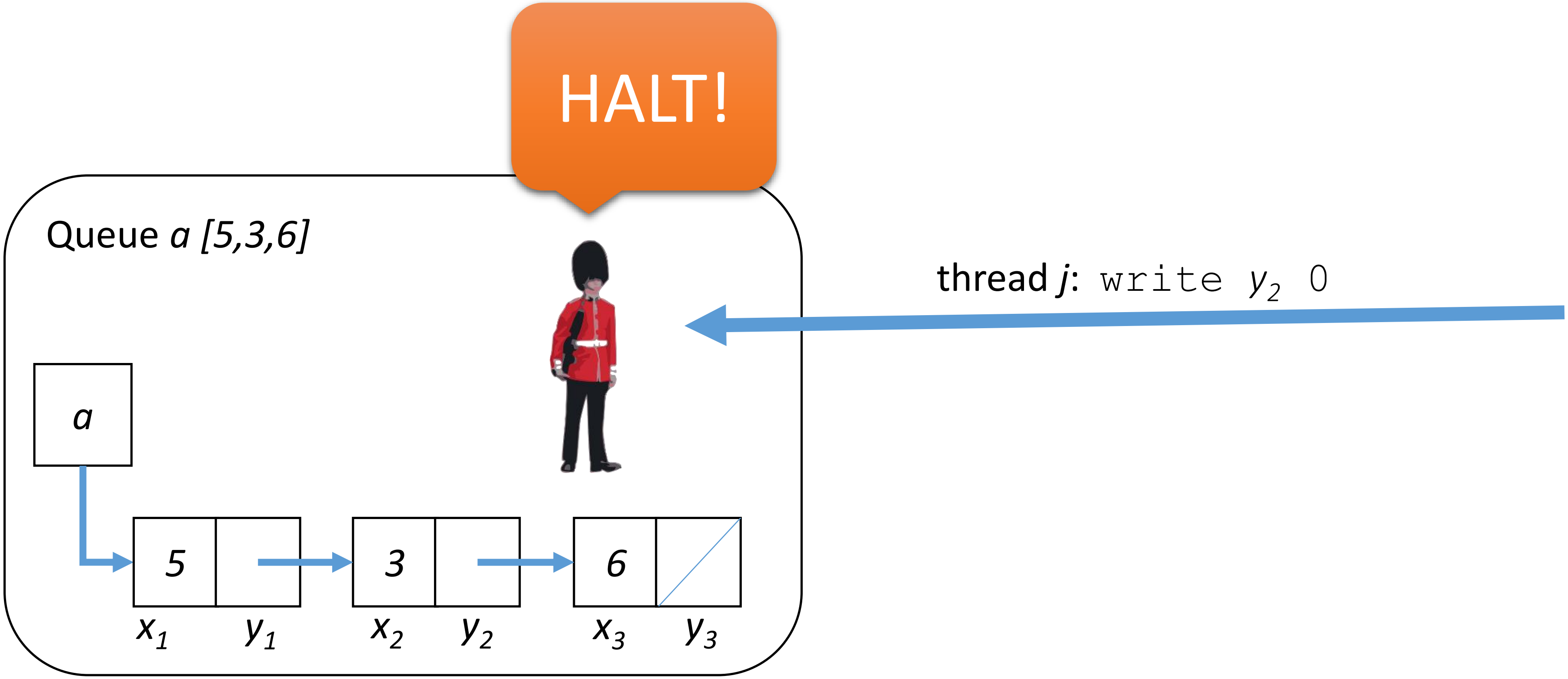
Instrumented Global State

"Phantom Monitors"



Instrumented Global State

“Phantom Monitors”



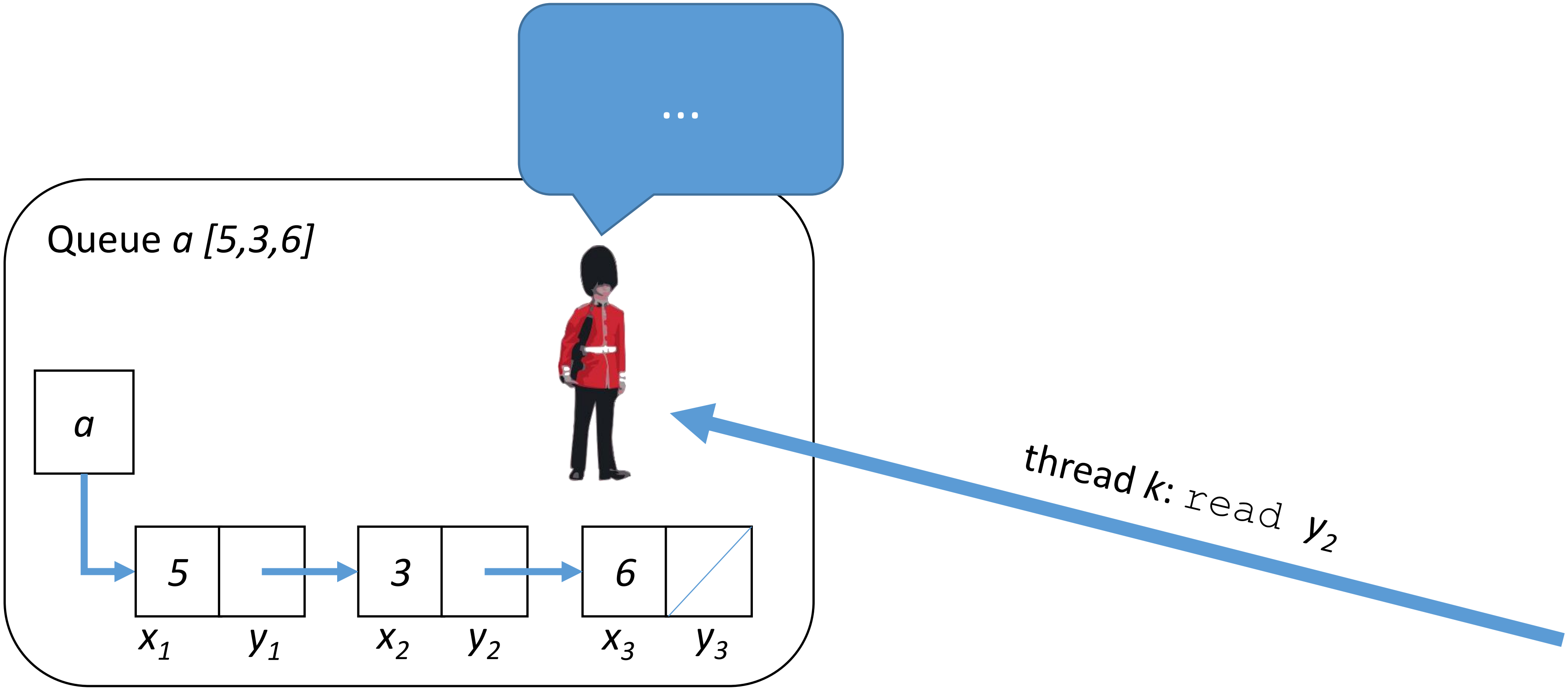
HALT!

Queue *a [5,3,6]*

thread j: `write` y_2 `0`

Instrumented Global State

“Phantom Monitors”



Phantom Monitors

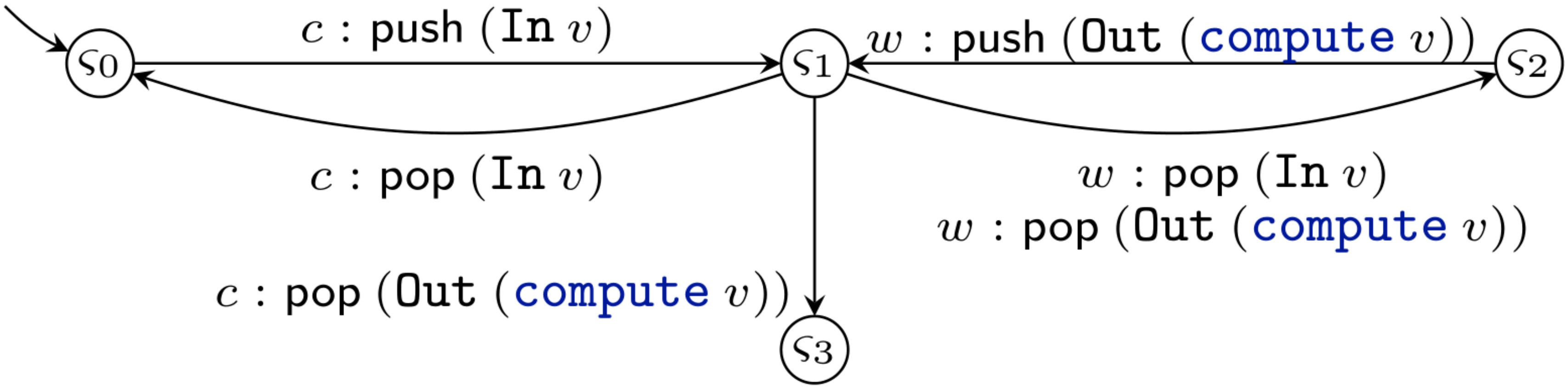
Is a Coq function that:

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*

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)
```

Stack Specification

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

$$\frac{}{\llbracket \sigma_0 \rrbracket = \epsilon} \qquad \frac{\sigma \overset{\mathsf{pop}\ v}{\underset{i}{\twoheadrightarrow}} \sigma'}{\llbracket \sigma \rrbracket = v :: \llbracket \sigma' \rrbracket} \qquad \frac{\sigma \overset{\mathsf{push}\ v}{\underset{i}{\twoheadrightarrow}} \sigma'}{\llbracket \sigma' \rrbracket = v :: \llbracket \sigma \rrbracket}$$

$$\frac{}{\mathsf{stable}\ (\mathsf{Stack}_\Sigma\ a\ \sigma)} \qquad \frac{\sigma \underset{\neq i}{\rightsquigarrow}^{*} \sigma'}{\mathsf{Stack}_\Sigma\ a\ \sigma' * \mathsf{pid}\ i \vdash \mathsf{Stack}_\Sigma\ a\ \sigma * \mathsf{pid}\ i}$$

$$\frac{\begin{array}{c}\mathsf{stable}\ \mathcal{W} \qquad \forall \sigma'.\ \sigma \underset{\neq i}{\rightsquigarrow}^{*} \sigma' \Rightarrow \sigma' \overset{\mathsf{push}\ v}{\underset{i}{\twoheadrightarrow}} \bullet \\ \forall \sigma'.\ \sigma \underset{\neq i}{\rightsquigarrow}^{*} \overset{\mathsf{push}\ v}{\underset{i}{\twoheadrightarrow}} \sigma' \Rightarrow \mathcal{V} \Vdash\!\models_i \{\mathsf{Stack}_\Sigma\ a\ \sigma' * \mathcal{W}\}\ s\end{array}}{\mathcal{V} \Vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma * \mathcal{W}\}\ \mathsf{push}\ a\ v;\ s}$$

$$\frac{\begin{array}{c}\mathsf{stable}\ \mathcal{W} \qquad \forall \sigma'.\ \sigma \underset{\neq i}{\rightsquigarrow}^{*} \sigma' \Rightarrow (\llbracket \sigma' \rrbracket = \epsilon\ \vee\ \exists v.\ \sigma' \overset{\mathsf{pop}\ v}{\underset{i}{\twoheadrightarrow}} \bullet) \\ \forall \sigma'.\ \sigma \underset{\neq i}{\rightsquigarrow}^{*} \sigma' \Rightarrow \llbracket \sigma' \rrbracket = \epsilon \Rightarrow \mathcal{V} \Vdash\!\models_i \{\mathsf{Stack}_\Sigma\ a\ \sigma' * \mathcal{W}\}\ s\ \mathsf{None} \\ \forall \sigma', v.\ \sigma \underset{\neq i}{\rightsquigarrow}^{*} \overset{\mathsf{pop}\ v}{\underset{i}{\twoheadrightarrow}} \sigma' \Rightarrow \mathcal{V} \Vdash\!\models_i \{\mathsf{Stack}_\Sigma\ a\ \sigma' * \mathcal{W}\}\ s\ (\mathsf{Some}\ v)\end{array}}{\mathcal{V} \Vdash_i \{\mathsf{Stack}_\Sigma\ a\ \sigma * \mathcal{W}\}\ x \leftarrow \mathsf{pop}\ a;\ s\ x}$$

Hypotheses

Our minimal TCB, semantically derived framework, and Coq proofs enable:

- quick(-ish*) development cycle of our logical framework
- automated proofs (via Ltac)
- exploring *general* logics
    - ex: do not bake in:
        - composable global reasoning
        - permission accounting
    - derive restricted principles as needed
- verifying challenging concurrent programs

* Coq proofs take time, but concurrency is tricky enough that it is easy to make mistakes with pen & paper proofs of logics

Thanks!

Client Program Policy

(general case)

```
protocol StackMonitor Σ (address head, σ0) implements Monitor
  Σ σ = σ0 (* 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 6= 0 ∧ dom(hAcq) = {newHead.item, newHead.next}
          else
              assert False
      else
          assert hAcq = empty
```