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...

```plaintext
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

```ruby
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

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

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

Separation Logic

Hoare doubles (CPS style)

Library of verified fine-grained concurrent data structures (Semaphore, Treiber stack, Harris-Michael set, etc.)

Erased semantics (one shared heap)

Client programs

Coq
Trusted Computing Base

(a) Syntax

\[
\begin{align*}
    v & \in V \quad \text{Value} \\
    a & \in A \quad \text{Address} \\
    \alpha & \in O \quad ::= \, \text{read } a \mid \text{write } a \, v \mid \text{cas } a \, v_0 \, v_1 \\
    s & \in S \quad ::= \, x \leftarrow \alpha; \, s \mid \text{nil} \mid \text{abort}
\end{align*}
\]

\[
\begin{align*}
    h(a) &= v \\
    (h, \text{read } a) &\to_h (h, v) \\
    h(a) &= v_1 \\
    (h, \text{cas } a \, v_1 \, v_2) &\to_h (h[a \mapsto v_2], 1) \\
    a &\in \text{dom}(h) \\
    (h, \text{write } a \, v) &\to_h (h[a \mapsto v], 1) \\
    a &\in \text{dom}(h) \\
    h(a) &\neq v_1 \\
    (h, \text{cas } a \, v_1 \, v_2) &\to_h (h, 0) \\
    (h, \alpha) &\to_h (h', v) \\
    (h, P \uplus [i \mapsto x \leftarrow \alpha; \, s]) &\to (h', P \uplus [i \mapsto s[v/x]])
\end{align*}
\]

(b) Erased operational semantics

\[
\begin{align*}
    \forall i. \, P(i) &\neq \text{nil} \Rightarrow \exists h', P'. \, (h, P) \to (h', P') \\
    \forall h', P'. \, (h, P) &\to (h', P') \Rightarrow \text{safe-program } h' \, P'
\end{align*}
\]

(c) Safety

\[
\begin{align*}
\text{safe-program } h \, P \\ \text{SAFE}
\end{align*}
\]
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”

Queue $a \{5, 3, 6\}$

thread $i$: cas $a \leftarrow x_1 x_2$

$x_1$ $y_1$ $x_2$ $y_2$ $x_3$ $y_3$
Instrumented Global State
“Phantom Monitors”

Queue $a [5,3,6]$

DEQUEUE 5

thread $i$: cas $a$ $x_1$ $x_2$
Instrumented Global State
“Phantom Monitors”

Queue $a (5,3,6)$

thread $j$: write $y_2$ 0

HALT!
Instrumented Global State
“Phantom Monitors”

Queue $a \{5, 3, 6\}$

thread $k$: read $y_2$
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 ($s_0 \rightarrow s_1$)
  - collects the finished value ($s_1 \rightarrow s_3$)

- **worker thread**: $w$
  - checks the stack for (unfinished) values ($s_1 \rightarrow s_2$)
  - pushes the computed value of each pop ($s_2 \rightarrow s_1$)
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): // $\zeta_2 \rightarrow \zeta_1$
    assert processing[i] = (Out v)
    $\forall \exists v'. \text{ processing}[i] = (\text{In } v') \land v = \text{compute}(v')$
    processing.remove(i)
  case x = (In v): // $\zeta_0 \rightarrow \zeta_1$
    assert $\exists l'. \text{ loading} = v :: l'$
    loading := tail(loading)

onPop(i, x)
  if i = client_pid then
    case x = (Out v): assert True // $\zeta_1 \rightarrow \zeta_3$
    case x = (In v): loading := v :: loading // $\zeta_1 \rightarrow \zeta_0$
  else // $\zeta_1 \rightarrow \zeta_2$
    assert i $\notin$ dom(processing)
    processing.add(i, x)
Stack Specification

\[ \alpha ::= \text{push } v \mid \text{pop } v \text{ and:} \]

\[ \begin{align*}
\| \sigma_0 \| &= \epsilon \\
\| \sigma \| &= v :: \| \sigma' \|
\end{align*} \]

\[ \sigma \xrightarrow{\text{pop } v}_i \sigma' \]

\[ \sigma \xrightarrow{\text{push } v}_i \sigma' \]

\[ \sigma \not\equiv^* \sigma' \]

\[ \text{stable } (\text{Stack}_\Sigma a \sigma) \quad \text{Stack}_\Sigma a \sigma' * \text{pid } i \vdash \text{Stack}_\Sigma a \sigma * \text{pid } i \]

\[ \begin{align*}
\forall \sigma'. \sigma \not\equiv^* \sigma' & \Rightarrow \sigma' \xrightarrow{\text{push } v}_i \bullet \\
\forall \sigma'. \sigma \not\equiv^* \sigma' & \Rightarrow \| \sigma' \| = \epsilon \Rightarrow \forall v. \sigma' \xrightarrow{\text{pop } v}_i \bullet \\
\forall \sigma', v. \sigma \not\equiv^* \sigma' & \Rightarrow \forall \sigma', v. \sigma \not\equiv^* \sigma' \Rightarrow V \models_i \{\text{Stack}_\Sigma a \sigma' * \mathcal{W}\} \, s \quad \text{None} \\
\forall \sigma', v. \sigma \not\equiv^* \sigma' & \Rightarrow V \models_i \{\text{Stack}_\Sigma a \sigma' * \mathcal{W}\} \, s \quad \text{(Some } v) \\
\end{align*} \]

\[ V \models_i \{\text{Stack}_\Sigma a \sigma * \mathcal{W}\} \, x \leftarrow \text{pop } a; \, s\, x \]
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 \( \Sigma \) (address head, \( \sigma_0 \)) implements Monitor
\( \Sigma \ \sigma = \sigma_0 \) (* 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
    \( \sigma \).onPop(i, h(oldHead.item))
    assert hAcq = empty
  else if hAcq(newHead.next) = oldHead then
    \( \sigma \).onPush(i, hAcq(newHead.item))
    assert newHead 6= 0 \& dom(hAcq) = \{newHead.item, newHead.next\}
  else
    assert False
else
  assert hAcq = empty