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
**Phantom Monitor**

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

Observes all operations on a shared data structure and enforces invariants
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...

```haskell
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 \text{ head } \sigma \ast F\}
fun push(head, v)
  
  
  
  
  
  
  
  
  
  
  
  
  \{Stack \text{ head } (v:\sigma) \ast F\}

\{Stack \text{ head } (v:\sigma) \ast F\}
fun pop(head)
  
  
  
  
  
  
  
  
  
  
  
  
  \{Stack \text{ head } \sigma \ast 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 $\sigma' * \text{results} =_{perm} \text{map} f \text{input} * F$}

{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
  \[ \ldots \] \text{client}_f(\text{input}) \{ \text{Stack jobs } \sigma' \ast \text{ results } =_{\text{perm}} \text{ map } f \text{ input } \ast F \}

  \text{where}

  \{ \text{Stack head } \sigma \ast F \} \text{ push}(\text{head}, v) \{ \text{Stack head } (v::\sigma) \ast F \}

  \{ \text{Stack head } (v::\sigma) \ast F \} \text{ pop}(\text{head}) \{ \text{Stack head } \sigma \ast 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 \textit{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

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

Client programs

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

Hoare doubles (CPS style)

Separation Logic

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

Erased semantics (one shared heap)

Coq
Trusted Computing Base

\[ v \in V \quad \text{Value} \]
\[ a \in A \quad \text{Address} \]
\[ \alpha \in O \quad ::= \ \text{read} \ a \ | \ \text{write} \ a \ v \ | \ \text{cas} \ a \ v_0 \ v_1 \]
\[ s \in S \quad ::= \ \ x \leftarrow \alpha; \ s \ | \ \text{nil} \ | \ \text{abort} \]

(a) Syntax

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

(b) Semantic domains

\[
\begin{align*}
(i \in I) &\quad \text{Thread ID} \\
h \in H &\quad = A \rightarrow V \quad \text{Heap} \\
P &\quad = I \rightarrow S \quad \text{Processes}
\end{align*}
\]

(c) Erased operational semantics

\[
\forall i. \ P(i) \neq \text{nil} \Rightarrow \exists h', P'. \ (h, P) \rightarrow (h', P')
\]
\[
\forall h', P'. \ (h, P) \rightarrow (h', P') \Rightarrow \text{safe-program} \ h' \ P'
\]
\[
\text{safe-program} \ h \ P \quad \text{SAFE}
\]

(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.
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”

Queue $a [5,3,6]$

\[
\begin{array}{ccc}
  \quad & 5 & x_1 \\
  a & \quad & \quad \downarrow \\
  \quad & 3 & x_2 \\
  \quad & 6 & x_3 \\
  \quad & \quad \downarrow \\
  \quad & y_1 & y_2 \\
  \quad & \quad \downarrow \\
  \quad & y_3 & \quad \\
\end{array}
\]

thread $i$: cas $a$ $x_1$ $x_2$
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
Vertical Composition

Client Protocol

{push, pop}

Stack Monitor

{read, write, cas}
Abstract Client Protocol

• Definition: $\Sigma: (S, \rightarrow, \sigma_0, \llbracket \cdot \rrbracket)$

• Transition function: $\rightarrow \subseteq S \times I \times A \times S$
  - $\sigma \xrightarrow{\alpha_i} \sigma'$

• Interference: $\sigma \xrightarrow{\bigcirc_{\neq i}}^{\star} \sigma'$
Stack Specification

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

\[ \vdash_i \{ \text{Stack}_\Sigma a \sigma' \ast \mathcal{W} \} s \]

\[ \vdash_i \{ \text{Stack}_\Sigma a \sigma \ast \mathcal{W} \} \text{ push } a \; v; \; s \]

\[ \vdash_i \{ \text{Stack}_\Sigma a \sigma \ast \mathcal{W} \} s \text{ None} \]

\[ \vdash_i \{ \text{Stack}_\Sigma a \sigma' \ast \mathcal{W} \} s (\text{Some } v) \]

\[ \vdash_i \{ \text{Stack}_\Sigma a \sigma \ast \mathcal{W} \} x \leftarrow \text{pop } a; \; s \; x \]
Stack Monitor

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 ≠ 0 ∧ dom(hAcq) = {newHead.item, newHead.next}
        else
            assert False
    else
        assert False
    assert hAcq = empty
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 (\( \varsigma_2 \rightarrow \varsigma_1 \))

![Diagram showing the interaction between client and worker threads]
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): // $\Sigma_2 \rightarrow \Sigma_1$
    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): // $\Sigma_0 \rightarrow \Sigma_1$
    assert $\exists l'. \text{loading} = v :: l'$
    loading := tail(loading)

onPop(i, x)
  if i = client_pid then
    case x = (Out v): assert True // $\Sigma_1 \rightarrow \Sigma_3$
    case x = (In v): loading := v :: loading // $\Sigma_1 \rightarrow \Sigma_0$
  else // $\Sigma_1 \rightarrow \Sigma_2$
    assert i $\notin$ 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
- {read, write, cas}

Set Monitor
- {read, write, cas}
Horizontal Composition

Top-level Protocol

Client Protocol
- (push, pop)
  - Stack Monitor
    - (read, write, cas)
  - Set Monitor
    - (read, write, cas)
- (add, remove)
What else?

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