Modern Separation Logic — Derek Dreyer

Lecture 4 - Invariants & Concurrency
July 1, 2026

1 From Persistency to Invariants

Continued from lecture 3... Regarding the program :

$$\begin{aligned} \mathit{MutBitImpl} \triangleq\ & \text{let } x = \text{new } 0 \text{ in} \\ & \langle \lambda\langle\rangle.\, x \leftarrow 1 - !x, \\ & \ \lambda\langle\rangle.\, \text{let } y = !x \text{ in} \\ & \quad \text{if } y = 0 \text{ then false} \\ & \quad \text{else if } y = 1 \text{ then true} \\ & \quad \text{else assert false}\rangle \end{aligned}$$

$$\text{assert } e \triangleq \text{if } e \text{ then } \langle\rangle \text{else } 4242$$

We want to prove:

$$\{\text{True}\}\ \mathit{MutBitImpl}\ \{v.\, \mathit{MutBit}(v)\}$$

where

$$\begin{aligned} \mathit{MutBit}(v) \triangleq\ & \{\text{True}\}(\pi_1 v)\langle\rangle\{w.\, w = \langle\rangle\} \\ & \wedge \{\text{True}\}(\pi_2 v)\langle\rangle\{w.\, w = \text{true} \vee w = \text{false}\} \end{aligned}$$

We internalized Hoare triples as an Iris proposition

$$\{P\}\ e\ \{v.\, Q(v)\} \triangleq \Box(P \mathrel{-\!\!*} \text{wp } e\ \{v.\, Q(v)\}),$$

using the persistence modality $\Box$ so that a specification, once proved, could be freely duplicated and reused.

Compiled By:
Ricky Cheng, Kalab Assefa, Ziyang Wang

How can we duplicate the ownership of resource $\ell \mapsto 0$ into the two clause of the $\wedge$ in $MutBit$, where ℓ is the location of x? After allocating the reference ℓ, our context holds $\ell \mapsto 0$. To prove the (persistent) triple for $\lambda\langle\rangle.\, x \leftarrow 1 - !x$, we would need to give up $\ell \mapsto 0$ underneath a $\Box$, but $\ell \mapsto 0$ is not a persistent proposition, so it cannot survive being put "inside a box". Wrapping a non-persistent resource in $\Box$ does not launder it into something duplicable; it is simply not provable. This is the real problem left over from last time, and persistency alone does not solve it. What we need is a way to make the *ownership itself* shared and duplicable, while still allowing us to temporarily and exclusively use what it protects. This is exactly what *invariants* give us.

2 Invariants

2.1 State Propositions

Not every proposition is safe to share as an invariant , in particular, an invariant must not secretly smuggle in more separation-logic machinery (like another invariant, or a weakest precondition) that could break the bookkeeping we are about to set up. The propositions we are allowed to put into an invariant are drawn from a restricted fragment, the *state propositions*.

2.2 Invariant Assertions

Given a state proposition F and a *namespace* N (a name used to keep invariants apart, more on this below), we write $\boxed{F}^{\mathcal{N}}$ for the assertion that F is maintained as a *shared* invariant. Owning $\boxed{F}^{\mathcal{N}}$ does not mean we personally own F: it means the ambient program, as a whole, guarantees that F holds at all times (except for brief windows we will discuss shortly). No single thread, and no single part of our proof, gets to hold F exclusively anymore , ownership has become collective.

For `MutBitImpl`, we no longer try to keep $\ell \mapsto 0$ to ourselves. Instead we agree, once and for all, to a protocol for how ℓ may be used: it always stores either 0 or 1.

$$I_{\text{MutBit}} := \boxed{\exists n \in \{0,1\}.\ \ell \mapsto n}^{\mathcal{N}}$$

2.3 Invariant Rules

InvPers
$$\frac{}{\boxed{F}^{\mathcal{N}} \vdash \Box \boxed{F}^{\mathcal{N}}}$$

InvAlloc
$$\frac{P * \boxed{F}^{\mathcal{N}} \vdash \text{wp}_E\ e\ \{v.\, Q(v)\}}{P * F \vdash \text{wp}_E\ e\ \{v.\, Q(v)\}}$$

InvOpen
$$\frac{P * F \vdash \text{wp}_{E \setminus N}\ e\ \{v.\, F * Q(v)\} \qquad N \subseteq E}{P * \boxed{F}^{\mathcal{N}} \vdash \text{wp}_E\ e\ \{v.\, Q(v)\}}$$

InvPers says that owning knowledge *that* F is maintained as an invariant is itself persistent: this

knowledge can be freely duplicated and handed out to as many parts of the proof (or, later, as many threads) as we like, even though F itself need not be persistent.

InvAlloc says we may convert local, exclusive ownership of a state proposition F into the persistent, shared knowledge $\boxed{F}^{\mathcal{N}}$. We trade away the right to unilaterally read or modify F, but in exchange we get to share it everywhere.

InvOpen is the interesting rule. If we own $\boxed{F}^{\mathcal{N}}$, we may *temporarily* get our hands on F during the execution of e , but we must hand back F (or a re-established version of it) in the postcondition. The reason we must restore it is that other parts of the proof (and, once we add concurrency, other threads) are relying on F continuing to hold whenever they look at it.

2.4 Namespaces and Masks

If we could open the same invariant twice, the logic would be unsound: two copies of $\ell \mapsto n$ from the same invariant would let us derive False via PointsToSep. To prevent this, every invariant carries a *namespace* N, and every weakest precondition carries a *mask* E recording which namespaces are still available to be opened ($E = \top$, the full mask, when omitted). Opening $\boxed{F}^{\mathcal{N}}$ requires $N \subseteq E$, and removes N from the mask for the duration of e; the mask is restored once the invariant is closed again. In the Iris Proof Mode this bookkeeping is entirely automatic: opening an invariant is a single tactic that checks the namespace side-condition and updates the mask for you.

2.5 Proof of `MutBitImpl`

We prove $\{\mathsf{True}\}$ `MutBitImpl` $\{v.\,\mathrm{MutBit}(v)\}$.

Both branches are proved underneath $\Box$, and both only ever needed the shared, persistent I_{MutBit}, never exclusive ownership of ℓ, Thus the argument goes through for $\lambda\langle\rangle.\, x \leftarrow 1 - !x$ and `get` arbitrarily many times.

2.6 Rocq Mechanization

The proof above corresponds closely to the following Rocq/Iris Proof Mode script.

```
(** Setup **)
Definition hoare (P : iProp) (e : expr) (Phi : val -> iProp) : iProp :=
  square (P -* WP e {{ Phi }}).

(* MutBit example *)
Definition assert : val :=
```

```
  lambda: "x", if: "x" then #() else #42 #42.

Definition MutBitImpl : expr :=
  let: "x" := ref #0 in
  (lambda: <>, "x" <- #1 - !"x",
   lambda: <>, let: "y" := !"x" in
              if: "y" = #0 then #false
              else if: "y" = #1 then #true
              else assert #false).

Definition MutBitSpec v : iProp :=
  {{ True }} (Fst v) #() {{ w, [| w = #() |] }} *
  {{ True }} (Snd v) #() {{ w, [| w = #true |] \/ [| w = #false |] }}.

Definition mutbitN := nroot .@ "mutbit".
Lemma MutBitImpl_proof :
  |- {{ True }} MutBitImpl {{ v, MutBitSpec v }}.
Proof.
  iIntros "!> _". unfold MutBitImpl.
  wp_alloc loc as "Hl". wp_pures.
  iApply (inv_alloc mutbitN (loc |-> #0 \/ loc |-> #1) with "[Hl]").
  { eauto with iFrame. }
  iIntros "#HInv".
  iApply wp_value. unfold MutBitSpec. iSplit.
  - iIntros "!>_". wp_pures.
    iApply (inv_open with "HInv"); first set_solver.
    iIntros "[Hl | Hl]".
    + wp_load. wp_store. iApply wp_value. eauto with iFrame.
    + wp_load. wp_store. iApply wp_value. eauto with iFrame.
  - iIntros "!> _". wp_pures.
    iApply (inv_open with "HInv"); first set_solver.
    iIntros "[Hl | Hl]".
    + wp_load. wp_pures. iApply wp_value. eauto with iFrame.
    + wp_load. wp_pures. iApply wp_value. eauto with iFrame.
Qed.
```

A few lines are worth matching up to the rules above:

- `wp_alloc` corresponds to the NEW rule from Lecture 2, giving us `loc` $\mapsto 0$.
- `inv_alloc mutbitN (...) with "[Hl]"` is exactly INVALLOC: we hand over our exclusive resource `Hl` and get back persistent knowledge of the invariant, named `HInv`.

- The `iIntros "!>"` pattern discharges a $\Box$ goal , it is how we *prove* a persistent proposition once its underlying content has been established (this is what lets `HInv` move into the persistent context, marked with `#`).
- Because `HInv` is persistent, it survives `iSplit` unchanged: it is available, in full, in *both* resulting subgoals ($\lambda\langle\rangle.\, x \leftarrow 1{-}!x$ and `get`). This is InvPers / duplicability in action.
- `inv_open with "HInv"` is InvOpen; the side-condition `set_solver` discharges the namespace check $N \subseteq E$ automatically.
- The two branches `[Hl | Hl]` are the case split on $\ell \mapsto 0 \vee \ell \mapsto 1$, and are symmetric, matching the $\lambda\langle\rangle.\, x \leftarrow 1{-}!x$/`get` cases above; `eauto with iFrame` both restores the invariant and discharges the postcondition.

3 Concurrency

So far, our language has had no way to run two pieces of code "at the same time". We now add that ability.

3.1 Extending the Language

$$\text{Terms } e ::= \cdots \mid \texttt{CAS}(e_1, e_2, e_3) \mid \texttt{fork}\{e\}$$

- $\texttt{fork}\{e\}$ spawns a new thread executing e, and immediately returns () in the thread that called it. In $\texttt{fork}\{e_1\}; e_2$, after $\texttt{fork}\{e_1\}$, both e_1 and e_2 are running.
- $\texttt{CAS}(e_1, e_2, e_3)$ ("compare-and-set") evaluates e_1, e_2, e_3 to a location ℓ and values v, w. If ℓ *currently* stores v, it atomically updates ℓ to store w and returns true; otherwise it leaves ℓ untouched and returns false.

Crucially, `CAS` performs its check-and-update as a *single, indivisible* step. This is what makes it useful as a synchronization primitive: no other thread can run "in between" the comparison and the update.

3.2 Operational Semantics

$$\frac{h(\ell) = v \quad v \text{ comparable}}{(\texttt{CAS}(\ell, v, w), h) \rightsquigarrow_b (\mathsf{true}, h[\ell \mapsto w])} \qquad \frac{h(\ell) \neq v \quad v \text{ comparable}}{(\texttt{CAS}(\ell, v, w), h) \rightsquigarrow_b (\mathsf{false}, h)}$$

$$\frac{}{(K[\texttt{fork}\{e\}], h) \rightsquigarrow (K[()], h, [e])}$$

Two new base reductions for `CAS` mirror the two possible outcomes above. Forking off a thread extends the contextual step with a list of newly spawned expressions, which then get folded into a thread pool; the thread pool may reduce *any* thread at each step, so any interleaving of threads is possible.

3.3 Concurrent Separation Logic Rules

SucCAS

$$\frac{v_1 \text{ comparable}}{\ell \mapsto v_1 * (\ell \mapsto v_2 \mathrel{-\!\!*} Q(\mathsf{true})) \vdash \text{wp } \ \texttt{CAS}(\ell, v_1, v_2)\ \{v.\, Q(v)\}}$$

FailCAS

$$\frac{v_1 \neq v_3 \quad v_1 \text{ comparable}}{\ell \mapsto v_3 * (\ell \mapsto v_3 \mathrel{-\!\!*} Q(\mathsf{false})) \vdash \text{wp } \ \texttt{CAS}(\ell, v_1, v_2)\ \{v.\, Q(v)\}}$$

Fork

$$\frac{}{\text{wp } \ e\ \{\_.\, \mathsf{True}\} * Q() \vdash \text{wp } \ \texttt{fork}\{e\}\ \{v.\, Q(v)\}}$$

SucCAS and FailCAS correspond exactly to the two operational rules above: to run $\texttt{CAS}(\ell, v_1, v_2)$ we need to already know what ℓ currently stores, and depending on whether it matches v_1, we either get to update our ownership to v_2 (and true), or keep it as-is (and false). Fork says that to justify forking off e, we only need e to be safe to run on its own (with an uninteresting postcondition, True) , we do not need to know anything about *when* it finishes, since we get $Q()$ for the parent immediately, before e has even necessarily started running.

3.4 Invariants Must Be Opened Atomically

Concurrency breaks InvOpen as stated above. With other threads running, we can no longer assume that e in wp $\ e\ \{v.\, Q(v)\}$ executes from start to finish without interruption: other threads may run at arbitrary points *during* e, and if we have left an invariant broken while e is still in progress, those other threads could observe an inconsistent state. We fix this by only allowing invariants to be opened around *atomic* expressions , expressions that reduce to a value in a single step, such as $\texttt{CAS}(\ell, v, w)$, $\ell \leftarrow v$, or $!\ell$:

InvOpen (concurrent)

$$\frac{P * F \vdash \text{wp}_{E \setminus N}\ e\ \{v.\, F * Q(v)\} \qquad N \subseteq E \qquad e \text{ atomic}}{P * \boxed{F}^{\mathcal{N}} \vdash \text{wp}_E\ e\ \{v.\, Q(v)\}}$$

Because an atomic expression cannot be interrupted, it is safe to momentarily break the invariant around it , no other thread ever gets to look "inside" that single step.

3.5 Example: A Spin Lock

We implement the classic "hello world" of concurrent separation logic: a spin lock, represented as a single memory cell holding a boolean (true while held, false while free).

$$\begin{aligned}
\texttt{mklock()} &:= \text{new false} \\
\texttt{trylock}(l) &:= \texttt{CAS}(l, \text{false}, \text{true}) \\
\texttt{lock}(l) &:= \text{if } \texttt{trylock}(l) \text{ then } () \text{ else } \texttt{lock}(l) \\
\texttt{unlock}(l) &:= l \leftarrow \text{false}
\end{aligned}$$

`trylock` makes a single attempt to atomically flip the lock from free to held, reporting whether it succeeded. `lock` calls `trylock` in a loop ("spins") until it succeeds. `unlock` simply writes false directly , no `CAS` is needed, because by the specification below, only the thread currently holding the lock is ever in a position to call `unlock`, so there is no race to guard against.

3.5.1 Specification

We introduce a predicate $\text{lock}(\ell, P)$, read "ℓ is a lock protecting the resource P", and aim to prove:

$$\begin{aligned}
&\{P\}\ \texttt{mklock()}\ \{v.\, \exists \ell.\, v = \ell * \text{lock}(\ell, P)\} \\
&\{\text{lock}(\ell, P)\}\ \texttt{lock}(\ell)\ \{_.\, P\} \\
&\{\text{lock}(\ell, P)\}\ \texttt{trylock}(\ell)\ \{v.\, (v = \text{false}) \vee (v = \text{true} * P)\} \\
&\{\text{lock}(\ell, P) * P\}\ \texttt{unlock}(\ell)\ \{_.\, \text{True}\}
\end{aligned}$$

Intuitively, $\text{lock}(\ell, P)$ behaves like an invariant that is tied to program discipline: `lock` "opens" it, handing us P, and `unlock` "closes" it again, taking P back. We define it directly as an invariant:

$$\text{lock}(\ell, P) := \boxed{\ell \mapsto \text{true} \vee (\ell \mapsto \text{false} * P)}^{\mathcal{N}}$$

Either the lock is currently held ($\ell \mapsto \text{true}$, and P is out in the world with whoever holds it), or it is free ($\ell \mapsto \text{false}$), in which case the invariant itself is holding P in escrow for the next thread to acquire it. Since $\text{lock}(\ell, P)$ is defined as an invariant, InvPers gives us, for free, that it is persistent:

$$\text{lock}(\ell, P) \vdash \Box\, \text{lock}(\ell, P),$$

which is exactly what lets a single lock be shared and used by arbitrarily many threads.

3.5.2 Proof Sketch of `lock`

Unfolding one call, our goal becomes

$$\mathrm{lock}(\ell, P) \vdash \mathrm{wp}\ \ (\text{if } \texttt{trylock}(\ell) \text{ then } () \text{ else } \texttt{lock}(\ell))\ \{\_.\, P\}.$$

Binding on the `CAS` inside `trylock` and opening $\mathrm{lock}(\ell, P)$ (a single `CAS` step is atomic, so this is allowed) splits the proof into the two disjuncts of the invariant:

- **Case** $\ell \mapsto$ **true** and
- **Case** $\ell \mapsto$ **false** $* P$

To be continued ...