OREGON PROGRAMMING LANGUAGES SUMMER SCHOOL

Modern Separation Logic — Derek Dreyer

Lecture 5 - *July 2, 2026*

Finishing the Proof of the Lock Predicate

We pick up where Lecture 4 left off in the proof of

$$\{\mathsf{lock}(\ell, P)\}\ \mathsf{lock}(\ell)\ \{\_.\ P\}.$$

with the following context and goal:

Context:
$(\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * P) * \square\, \mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$

Goal:
$\mathsf{wp}\, \mathsf{trylock}(\ell)\, \{v.\ (\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * P) * \mathsf{wp}\, (\texttt{if}\ v\ \texttt{then}\ ()\ \texttt{else}\ \mathsf{lock}(\ell))\, \{\_.\ P\}\}$

We case on whether $\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * P$:

Case $\ell \mapsto \mathsf{true}$. Since ℓ currently stores true, the $\texttt{CAS}(\ell, \mathsf{false}, \mathsf{true})$ hidden inside $\mathsf{trylock}(\ell)$ does not match, so by FailCas it fails: it leaves ℓ untouched and returns false. Substituting $v := \mathsf{false}$ into the goal gives us:

Context:
$\ell \mapsto \mathsf{true} * \square\, \mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$

Goal:
$(\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * P) * \mathsf{wp}\, (\texttt{if}\ \mathsf{false}\ \texttt{then}\ ()\ \texttt{else}\ \mathsf{lock}(\ell))\, \{\_.\ P\}$

The $\ell \mapsto \mathsf{true}$ still in our context is exactly the left disjunct the invariant needs, so we hand it back and close the invariant. This discharges the disjunction, leaving:

Context:
$\square\, \mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$

Goal:
$\mathsf{wp}\, (\texttt{if}\ \mathsf{false}\ \texttt{then}\ ()\ \texttt{else}\ \mathsf{lock}(\ell))\, \{\_.\ P\}$

The conditional reduces to its else branch, so the goal simplifies to $\mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$, which is identical to the fact already sitting in our context, so we use it to close the goal.

Compiled By:
Carolyn Zech

Case $\ell \mapsto \mathsf{false} * P$. Since ℓ stores false, the `CAS` succeeds: by SucCas it atomically updates ℓ to true and returns true. Substituting $v := \mathsf{true}$ into the goal gives us:

Context:	*Goal:*
$\ell \mapsto \mathsf{true} * P * \square\, \mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$	$(\ell \mapsto \mathsf{true} \vee \ell \mapsto \mathsf{false} * P) *$ $\mathsf{wp}\, (\texttt{if true then () else } \mathsf{lock}(\ell))\, \{\_.\ P\}$

As before, we hand back $\ell \mapsto \mathsf{true}$ and close the invariant, discharging the disjunction:

Context:	*Goal:*
$P * \square\, \mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$	$\mathsf{wp}\, (\texttt{if true then () else } \mathsf{lock}(\ell))\, \{\_.\ P\}$

The conditional reduces to its then branch and returns $()$, so $\mathsf{trylock}(\ell)$ (and hence $\mathsf{lock}(\ell)$) is finished, and the goal simplifies to just P:

Context:	*Goal:*
$P * \square\, \mathsf{wp}\, \mathsf{lock}(\ell)\, \{\_.\ P\}$	P

which is trivially met by the P in our context. □

Modeling `Arc<T>` in Rust

We now turn to a different case study: proving a model of `Arc<T>` in Rust using concurrent separation logic. From the documentation:

> The type `Arc<T>` provides shared ownership of a value of type T, allocated in the heap. Invoking clone on Arc produces a new Arc instance, which points to the same allocation on the heap as the source Arc, while increasing a reference count. When the last Arc pointer to a given allocation is destroyed, the value stored in that allocation (often referred to as "inner value") is also dropped.

The specification

We represent `Arc` in concurrent separation logic via the following two Hoare triples, where $\mathsf{Arc}(x)$ denotes the permission to access shared data through x:

$$\{\mathsf{Arc}(x)\}\ y = x.\texttt{clone}();\ \{\mathsf{Arc}(x) * \mathsf{Arc}(y)\} \qquad \{\mathsf{Arc}(x)\}\ x.\texttt{drop}()\ \{\mathsf{True}\}$$

At first glance, this looks broken. The separating conjunction $*$ is supposed to witness that its two conjuncts describe disjoint, non-aliasing resources. So how can $\mathsf{Arc}(x) * \mathsf{Arc}(y)$ possibly be a sound postcondition?

This representation is okay because $\mathsf{Arc}(x)$ does *not* describe physical ownership of the memory behind x. It is a purely *logical* permission: a token entitling its holder to access the shared data through x. Because $\mathsf{Arc}(x)$ lives in the logical, ghost layer of our reasoning rather than in the physical heap, we are free to split it between different threads without violating separation logic's guarantees about physical, disjoint heap ownership.

The axioms governing Arc

To make this precise we introduce a piece of logical state, $\mathsf{Count}(x, n)$, asserting that there are currently n outstanding copies of $\mathsf{Arc}(x)$. Two axioms govern how $\mathsf{Count}(x, n)$ and $\mathsf{Arc}(x)$ interact:

1. $\mathsf{Count}(x, n) * \mathsf{Arc}(x) \Rightarrow n > 0$: if n counts the live `Arc` instances and we hold one of them, then n must be at least one.
2. $\mathsf{Count}(x, n) \Leftrightarrow \mathsf{Count}(x, n+1) * \mathsf{Arc}(x)$: this is how we create a new `Arc`. Reading the rule left to right increments the count and produces a fresh $\mathsf{Arc}(x)$; reading it right to left consumes an $\mathsf{Arc}(x)$ and decrements the count.

The Arc invariant

Having introduced this logical state, we tie it to the physical heap with an invariant:

$$\exists n.\ \mathsf{Count}(x, n) * \big(n = 0 \vee x \mapsto (n, \text{-})\big)$$

This invariant says that whenever the logical count is nonzero, x points to a physical heap cell storing that same count. The "-" is just a placeholder for the actual inner value stored alongside the count; since we're only reasoning about reference counting here, we don't care what it is.

Proof of the clone spec

With the invariant connecting the logical and physical states, we can verify that our model of `Arc` is correct.

For this case, we'll focus on the clone rule with the Hoare triple $\{\mathsf{Arc}(x)\}\ y = x.\texttt{clone}();\ \{\mathsf{Arc}(x) * \mathsf{Arc}(y)\}$.

Since our precondition is $\mathsf{Arc}(x)$, Axiom 1 tells us that $n > 0$. The call to `clone()` is implemented as a single fetch-and-add, $\texttt{FAA}(x.\texttt{count}, 1)$. To reason about it, we temporarily open the invariant, borrowing its physical points-to fact into our local state, and execute the `FAA` against it:

$$\{\mathsf{Arc}(x) * x \mapsto (n, \text{-})\}\ y = x.\texttt{clone}();\ \{\mathsf{Arc}(x) * x \mapsto (n+1, \text{-})\}$$

Our logical state (still just $\mathsf{Count}(x, n)$ inside the invariant) no longer matches the physical state we just produced, so we invoke Axiom 2 to update it: incrementing the logical count to $n + 1$ creates a fresh $\mathsf{Arc}(x)$ in the postcondition:

$$\{\mathsf{Arc}(x) * x \mapsto (n, \text{-})\}\ y = x.\texttt{clone}();\ \{\mathsf{Arc}(x) * \mathsf{Arc}(x) * x \mapsto (n+1, \text{-})\}$$

Now that our logical and physical state agree once more, we can close the invariant back up, leaving only the two Arcs:

$$\{\mathsf{Arc}(x)\}\ y = x.\texttt{clone}();\ \{\mathsf{Arc}(x) * \mathsf{Arc}(x)\}$$

which is our target postcondition. □

Why the Arc Axioms Are Sound

We built a model of `Arc` on top of these two axioms, but how do we know that the axioms themselves are sound? The Iris recipe for answering this is to define logical state as a partial commutative monoid (PCM). Once we've done that, Iris gives you two primitive proof rules:

$$\mathsf{Own}(a \cdot b) \iff \mathsf{Own}(a) * \mathsf{Own}(b)$$

$$\frac{\forall f.\ \mathsf{Def}(a \cdot f) \Rightarrow \mathsf{Def}(b \cdot f)}{\mathsf{Own}(a) \Rrightarrow \mathsf{Own}(b)}$$

The first rule says that owning a composite resource $a \cdot b$ is the same as separately owning a and owning b. The second says that if we currently own a and want to update our ownership to b, we may do so, so long as that change never disturbs composability: for every possible frame f (i.e. whatever other resources the rest of the program or other threads might be holding alongside ours), if $a \cdot f$ was well-defined then $b \cdot f$ must be well-defined too. In other words, the update is safe exactly when it can't possibly invalidate resources owned elsewhere.

From these two rules, you can derive the soundness of the Arc axioms.