Automating Separation Logic Reasoning

- Separation Logic is a bad fit for SMT solvers
  - Predicates are higher-order
  - Predicates are often recursive
  - Relies on Associative-Commutative (AC) reasoning

$$P \star Q \star R \Leftrightarrow R \star P \star Q$$

- Automation through a cooperation between SMT solving and custom separation logic decision procedures

A Syntax-Directed Frame Rule

- **Problem**: Applications of the frame rule are non-deterministic

$$\frac{\Gamma \vdash \boxed{c} : \{Q\}\ t\ \{R\}}{\Gamma \vdash c : \{?\,P \star Q\}\ t\ \{?\,P \star R\}}$$

- **Solution:** Deterministically apply framing at the "leaf" only, during function calls

$$\frac{\Gamma \vdash v : a. \qquad \Gamma \vdash f : a \ \rightarrow \{Q\}\ t\ \{R\}}{\Gamma \vdash \boxed{f\ v} : \{?\,P \star Q\}\ t\ \{?\,P \star R\}}$$

Automating Frame Inference: An Example

```
val write (r:ref a) (x:a) : Steel unit (ptr r) (ptr r)
let two_writes (r1 r2:ref int) : Steel unit (ptr r1 ⋆ ptr r2) (ptr r1 ⋆ ptr r2)

  = write r1 0;          // : {?F1 ⋆ ptr r1} unit {?F1 ⋆ ptr r1}
    write r1 1           // : {?F2 ⋆ ptr r1} unit {?F2 ⋆ ptr r1}
```

- **Observation:** Separation logic VCs can be seen as AC-unification problems

  for instance, ptr r1 ⋆ ptr r2 ⇔ ?F1 ⋆ ptr r1

- **Observation:** A scheduling of equivalences where each problem contains at most one metavariable exists

  ptr r1 ⋆ ptr r2 ⇔ ?F1 ⋆ ptr r1,

  ?F1 ⋆ ptr r1 ⇔ ?F2 ⋆ ptr r1

  ?F2 ⋆ ptr r1 ⇔ ptr r1 ⋆ ptr r2

Solving Frame Metavariables

We reduced the problem to solving equivalences of the shape

$$?F \star P1 \star P2 \Leftrightarrow Q1 \star Q2$$

We provide a decision procedure for these problems as an F* tactic, which:

- Supports existentially quantified ghost variables
- Can query the SMT solver for equalities on subterms
- Sacrifices completeness for speed and user interaction

Steel Example: Spinlocks

val lock (p:slprop) : Type

Transferring ownership

val acquire (l:lock p) : Steel unit emp (λ _ → p)

Initially, no ownership

val release (l:lock p) : Steel unit p (λ _ → emp)

Transferring ownership back

Steel Example: Invariants (“Ghost locks”)

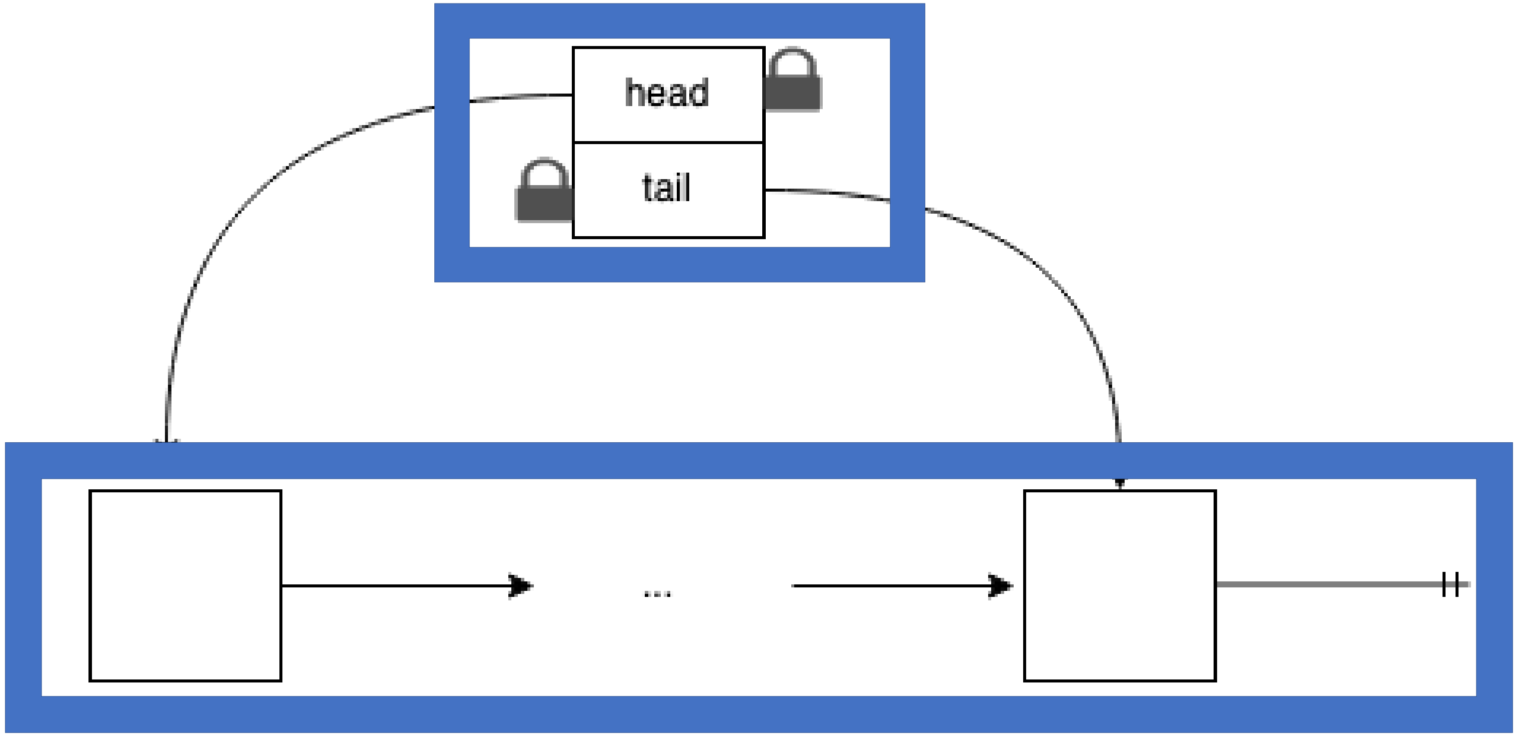
val inv (p:slprop) : Type

Invariants can be accessed inside **atomic commands**

Composition of a physical action and a finite number of ghost operations

This enables lock-free shared-memory concurrency

Steel Example: Michael-Scott 2-lock queues



- The *shape* of the queue is captured by an ***invariant***
- The dequeuer and the enqueuer both have a ***lock*** on the head and tail pointers respectively

Steel Example: Message-Passing Concurrency

val chan : Type

val endpoint (ch:chan) (p:prot) : slprop

Returns a new channel

Returns permissions for two parties to use the new channel for protocol p

val new (p:prot) : Steel chan emp (λc → endpoint c p ★ endpoint c (dual p))

At this stage of the protocol, we must send a message

The message x is compatible with the current state of the protocol

val send (c:chan{is_send_next next}) (x:msg_t next) :
  Steel unit (endpoint c next) (λ_ → endpoint c (step next x))

We initially are at the stage *next* of the protocol on channel c

After executing send, we advanced the state of the protocol by sending x

val recv …

val close …

Steel Example: PingPong Protocol

```
let pingpong : prot =
  let x = Protocol.send int in
  let y = Protocol.recv (y:int{y > x}) in
  Protocol.done


let client (c:chan) : Steel unit (endpoint c pingpong) (λ_ → emp)
  = send c 17;
    let y = recv c in
    assert (y > 17);
    close ()
```

Statically checked assert, erased at runtime

Separating Separation and First-Order Logic

- We associate to each separation logic predicate a *self-framing selector*

  For example, a reference's selector is the value it contains

- First-order logic predicates about *selectors* can be discharged by SMT

```
val swap (p1 p2:ref int) : Steel unit (ptr p1 ★ ptr p2) (ptr p1 ★ ptr p2)
  (requires λ _→ T)
  (ensures λ s0 _ s1 → s0.[p1] == s1.[p2] /\ s0.[p2] == s1.[p1])
```

Steel Example: Binary Trees

Steel a p q: a computation that has return type a, under the precondition p, and with the postcondition q

```
type node a = {data : a; left : t a; right: t a}
and t a = ref (node a)

val tree (ptr:t a) : slprop
```

An abstract predicate capturing a tree-shaped structure

Expects a tree-shaped structure

```
let rec height (ptr:t a) : Steel int (tree ptr) (λ _ → tree ptr)
  (requires λ_ → T)
  (ensures λs0 x s1 → s0.[ptr] == s1.[ptr] ∧ Spec.height s0.[ptr] == x)
= if is_null ptr then (unroll_leaf ptr; 0) else (
     let node = unroll_tree ptr in
     let hleft = height node.left in
     let hright = height node.right in roll_tree ptr node.left node.right;
     if hleft > hright then (hleft + 1) else (hright + 1))
```

Returns a tree-shaped structure

The contents of the tree are unchanged

Functional correctness

Steel Example: AVL Trees

```
type node a = {data : a; left : t a; right: t a}
and t a = ref (node a)

val tree (ptr:t a) : slprop

val insert_avl (cmp:Spec.cmp a) (ptr:t a) (v:a)
  : Steel (t a) (tree ptr) (λ ptr' → tree ptr')
    (requires λ s →            Spec.is_avl cmp s.[ptr])
    (ensures λs0 ptr' s1 → Spec.is_avl cmp s1.[ptr] ∧
                            s1.[ptr'] == Spec.insert_avl cmp s0.[ptr] v)
```

Same abstract predicate as before

The AVL invariant is preserved

Functional correctness