In Lecture 7 we saw Hoare Logic, a *program logic* that gave us the ability to reason about the particular behaviors of a program we care about without running it. Using only predicates over states, Hoare Logic worked very nicely on Imp because Imp had only mutable variables and each variable was its own mapping.

In Fl!, however, things are a bit more complicated. Recall the following example from Lecture 14.

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in}\\ \mathsf{let}\ y = x\ \mathsf{in}\\ \quad x := 2\ ;\ !y\end{array}$$

This example shows that we can *alias* references. That is, we can have two variables (here x and y) that both refer to the same reference and therefore point to the same thing. When using Hoare Logic, we therefore need to be careful about this aliasing; If we had the predicate $(x \mapsto 1) \land (y \mapsto 1)$ as a precondition to $x := 2$, we would need to somehow track that we are changing both with the assignment. Unfortunately, if we're not careful, we will just end up with the following partial correctness assertion (PCA):

$$\{(x \mapsto 1) \land (y \mapsto 1)\}\ x := 2\ ;\ !y\ \{P\}\,.$$

But the precondition doesn't have enough information for us to know if x and y are actually the same reference, so we have no way to know if this returns 1 or 2!

To address this problem, we turn to *separation logic*. Separation logic is designed to reason about references in a heap with knowledge about (non-)aliasing. To do so, it uses a small set of new logical constructs that we will discuss in detail below, including formalizing the "points-to" predicate $x \mapsto v$.

1 Hoare Logic in λ-Calculus

Before we talk about separation logic, we first need to adapt Hoare Logic as we saw it in Lecture 7 to λ-calculus. In Imp, there were no output values, only the current store, so predicates were just over the store. We will now have predicates over the store—which will separation logic structures—but our predicates must also talk about return values and other computations in our program. To handle that, our postconditions will be functions of a return value. That is, a PCA will now take the form

$$\{P\}\ e\ \{\lambda x.\, Q(x)\}\,.$$

To get a sense of how these work in a language like λ-calculus or Fl, we look at two of the core rules.

$$[\textsc{Val}]\ \frac{}{\vdash \{P(v)\}\ v\ \{\lambda x.\, P(x)\}} \qquad [\textsc{Let}]\ \frac{\vdash \{P\}\ e_1\ \{\lambda x.\, Q(x)\} \qquad \forall v. \vdash \{Q(v)\}\ e_2[x \mapsto v]\ \{\lambda y.\, R(y)\}}{\vdash \{P\}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \{\lambda y.\, R(y)\}}$$

Note also that we can rewrite expressions using let to avoid having complex compound expressions. For example, we could rewrite

$$e_1\ e_2 \ \rightsquigarrow\ \mathsf{let}\ f = e_1\ \mathsf{in}\ \mathsf{let}\ x = e_2\ \mathsf{in}\ f\ x.$$

This rewriting to separate all of the interesting expressions using let is known as a translation into *A-normal form* and is common in compilers. It can also help tremendously with analysis, as we no longer need to analyze the behavior of compound expressions, just individual expressions and sequencing. We will use this strategy for the rest of this lecture for simplicity. We could have avoided the need for an A-normal form transformation, but the program logic rules would be considerably more complicated.

Soundness. Soundness for this functional version of Hoare Logic is nearly the same as the Imp version. First, we note that the definition of semantic entailment now uses the output value of the expression. Seocnd, since there now exist stuck terms, we demand that the PCA $\{P\}\ e\ \{Q\}$ ensure that e not get stuck. Formally,

$$\begin{aligned} \vDash \{P\}\ e\ \{\lambda x.\, Q(x)\} \iff^{\triangle} \forall \sigma.\, \sigma \vDash P \text{ and } \langle e, \sigma\rangle \longrightarrow^* \langle e', \sigma'\rangle \\ \implies \text{either } \langle e', \sigma'\rangle \text{ can step} \\ \text{or } e' = v' \text{ and } \sigma' \vDash Q(v'). \end{aligned}$$

Soundness then operates the same was as it did before.

Definition 1 (Soundness)**.** The logic is *sound* if, whenever $\vdash \{P\}\ e\ \{\lambda x.\, Q(x)\}$ then $\vDash \{P\}\ e\ \{\lambda x.\, Q(x)\}$.

2 Separation Logic Propositions

To talk about locations in the heap while remaining aware of the problems of aliasing, we introduce three new propositions into our predicate logic.

$$P \quad ::= \quad \cdots \mid \text{emp} \mid \ell \mapsto v \mid P \star Q$$

Let's go through each of these predicates.

- The predicate emp is true only on the empty store: $\varnothing \vDash \text{emp}$. This may seem like a silly predicate, but we will see later how it is, in fact, very useful.
- The predicate $\ell \mapsto v$ is true on exactly one store: the store that maps ℓ to v and nothing else. That is,

  $$\sigma \vDash \ell \mapsto v \iff^{\triangle} \text{dom}(\sigma) = \{\ell\} \text{ and } \sigma(\ell) = v.$$

  Again, the restriction that σ contain *only* the mapping $\ell \mapsto v$ may seem very limiting, but we will see how it is important soon.
- The last proposition is the *separating conjunction* $\star$. The proposition $P \star Q$, read "P and separately Q" or simply "P star Q," is similar to a normal logical conjunction in that it requires both P and Q to hold. However, it requires them to do so *disjointly*, meaning there are two disjoint parts of the store, one where P holds and the other where Q holds.

  Formally, two stores σ_1 and σ_2 are *disjoint* if their domains are disjoint. For simplicity, we write $\sigma_1 \perp \sigma_2$ when $\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \varnothing$. The separating conjunction is then defined by

  $$\sigma \vDash P \star Q \iff^{\triangle} \exists \sigma_1, \sigma_2.\, (\sigma = \sigma_1 \cup \sigma_2) \text{ and } (\sigma_1 \perp \sigma_2) \text{ and } \sigma_1 \vDash P \text{ and } \sigma_2 \vDash Q.$$

  Notably, the separating conjunction rules unexpected aliasing. The predicate $\ell \mapsto v \star \ell \mapsto v$ is false for all stores! As a result, unlike with regular $\wedge$, if $(\ell_1 \mapsto v) \star (\ell_2 \mapsto v)(\sigma)$ holds, we know that ℓ_1 and ℓ_2 must be different, so changing one cannot impact the other.

Pure Predicates. We also have a notion of *pure predicates*, which are predicates that are true about the program we are working in and do not interact with the store. For instance, we might record the value of a variable $x = a$, which does not care about what locations point to what in a store. We denote these predicates $\lceil\varphi\rceil$ to indicate their purity, and note that, as heap predicates, they are equivalent to emp. That is, for any pure predicate φ, the heap predicate $\lceil\varphi\rceil \star P$ holds over any heap where P holds.

The combination of pure predicates and separation logic connectives is immensely useful for building Hoare Logic deduction rules for stateful operations. We will discuss what are know as the "small rules" here, which assume the simplest form of expressions. We can get to those using A-normal form or by proving that the subterms reduce in the way we expect.

$$\text{[H-Ref]}\ \frac{}{\vdash \{\mathsf{emp}\}\ \mathsf{ref}\ v\ \{\lambda x.\, \exists \ell.\, \lceil x = \ell \rceil \star \ell \mapsto v\}} \qquad \text{[H-Store]}\ \frac{}{\vdash \{\ell \mapsto v_0\}\ \ell \coloneqq v\ \{\lambda\_.\, \ell \mapsto v\}}$$

$$\text{[H-Deref]}\ \frac{}{\vdash \{\ell \mapsto v\}\ !\ell\ \{\lambda x.\, \lceil x = v \rceil \star \ell \mapsto v\}}$$

By only considering the minimal possible stores, these rules remain small, simple, and self-contained. However, we need more to use them in real programs where they are likely other locations in the store.

3 Using Separation Logic: The Frame Rule

To handle stores with more locations, we need a rule that allows us to combine the small rules above with larger stores. Rather than gluing an additional arbitrary predicate to all of these rules, we capture the ability to do so in a uniform way using the *frame rule*, which forms the heart of separation logic.

$$\text{[H-Frame]}\ \frac{\vdash \{P\}\ e\ \{\lambda x.\, Q(x)\}}{\vdash \{P \star F\}\ e\ \{\lambda x.\, Q(x) \star F\}}$$

Frame is critical. It not only allows the small rules to be used in larger contexts with more locations in the heap, it supports separation logic's *ownership reasoning*. The idea is that having an assertion in the precondition expresses "ownership."

For example, the precondition $\ell \mapsto v$ means the function starts out owning the location ℓ, meaning no other part of the program can read from it or write to it. In the triple $\{\ell \mapsto 0\}\ f(\ell, \ell')\ \{\lambda\_.\, \ell \mapsto 42\}$, the function f owns the location ℓ for the duration of its execution and we can be sure that nothing interferes with it. Moreover, because the pre- and post-conditions do not reference ℓ', we know that f cannot access it at all. The frame rule coupled with the separating nature of $\star$ captures these requirements.

As an example, consider using separation logic to prove that the following expression returns true

$$\begin{aligned} e_{\text{own}} \ = \ & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\ & \mathsf{let}\ y = \mathsf{ref}\ 42\ \mathsf{in} \\ & f(x, y)\ ;\ (!x = !y) \end{aligned}$$

where f satisfies the PCA $\{\ell \mapsto 0\}\ f(\ell, \ell')\ \{\lambda\_.\, \ell \mapsto 42\}$.

We might accomplish this by proving $\{\top\}\ e_{\text{own}}\ \{\lambda v.\, \lceil v = \mathsf{true} \rceil \star \top\}$ where $\top$ is the trivial predicate that

holds on all stores. The following sketches out the proof.

$\{\top\} = \{\top \star \mathrm{emp}\}$
  $\{\mathrm{emp}\}$ (H-Frame)
    $\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}$ (H-Ref)
  $\{x \mapsto 0\} = \{x \mapsto 0 \star \mathrm{emp}\}$
    $\{\mathrm{emp}\}$ (H-Frame)
      $\mathsf{let}\ y = \mathsf{ref}\ 42\ \mathsf{in}$ (H-Ref)
    $\{y \mapsto 42\}$
  $\{x \mapsto 0 \star y \mapsto 42\}$
    $\{x \mapsto 0\}$ (H-Frame)
      $f(x, y)$ (f as shown above)
    $\{x \mapsto 42\}$
  $\{x \mapsto 42 \star y \mapsto 42\}$
    $!x = !y$ (H-Frame and H-Deref twice)
  $\{\lambda v.\ \lceil v_x = 42 \wedge v_y = 42 \wedge v = (v_x = v_y)\rceil$
      $\star\, x \mapsto 42 \star y \mapsto 42\}$
  $\Rrightarrow \{\lambda v.\ \lceil v = \mathsf{true}\rceil \star x \mapsto 42 \star y \mapsto 42\}$ (H-Weaken)
$\{\lambda v.\ \lceil v = \mathsf{true}\rceil \star x \mapsto 42 \star y \mapsto 42 \star \top\}$
$\Rrightarrow \{\lambda v.\ \lceil v = \mathsf{true}\rceil \star \top\}$ (H-Weaken)

Note that every step in this proof using H-Frame in some way or another! This example shows how core the H-Frame rule is and how we can combine its ownership reasoning to produce a useful result.