CS 839: Systems Verification

Lecture 8: Separation Logic (part 2)

this lecture will be mostly on the board

Learning outcomes

1. Appreciate why separation logic works
2. Be ready to think *in* separation logic

Recall: meaning of $P * Q$

Write down the definition from memory

Answer

$(P * Q)(h) \triangleq \exists h_1, h_2. \, (h = h_1 \cup h_2) \wedge (h_1 \perp h_2) \wedge P(h_1) \wedge Q(h_2)$

Recall: separation logic predicates

$\ell_3 \mapsto \ell_4 * \ell_2 \mapsto \ell_2 * \ell_1 \mapsto 3$

Separation logic rules

$$\{\mathrm{emp}\}\ \mathrm{alloc}\ v\ \{\lambda w.\, \exists \ell.\, \lceil w = \ell \rceil \star \ell \mapsto v\} \quad \text{alloc-spec}$$

$$\{\ell \mapsto v\}\ !\,\ell\ \{\lambda w.\, \lceil w = v \rceil \star \ell \mapsto v\} \quad \text{load-spec}$$

$$\{\ell \mapsto v_0\}\ \ell \leftarrow v\ \left\{\lambda\_.\, \ell \mapsto v\right\} \quad \text{store-spec}$$

Now we get to the *program* logic part of separation logic, reasoning about actual programs.

Frame rule

$$\frac{\{P\}\ e\ \{\lambda v.\, Q(v)\}}{\{P \star F\}\ e\ \{\lambda v.\, Q(v) \star F\}}\ \text{frame}$$

Key to separation logic

Recall how in regular Hoare logic, we would prove a specification for a function, then use it whenever needed by adapting the pre- and post-condition with the rule of consequence.

The frame rule lets us reason about a function over a "small footprint" and then use it in a larger heap. It shows that in the larger context, the function doesn't modify "anything else."

Illustrative example of framing

suppose we've proven

$\{\ell_1 \mapsto \overline{0}\}\, f(\ell_1, \ell_2)\, \{\ell_1 \mapsto \overline{42}\}$

$$
\begin{aligned}
e_{\text{own}} ::= \\
&\textbf{let } x := \text{alloc } \overline{0} \textbf{ in} \\
&\textbf{let } y := \text{alloc } \overline{42} \textbf{ in} \\
&f\,(x, y); \\
&\text{assert}\,(!\,x == !\,y)
\end{aligned}
$$

Proof outline for e_{own}

Rewrite the code (in "A-normal form") to put !x and !y onto their own lines, so the proof outline can be written clearly.

Exercise: prove swap correct

$$\text{swap } \ell_1\, \ell_2 ::= \textbf{let } t := !\,\ell_1 \textbf{ in } \ell_1 \leftarrow !\,\ell_2;\ \ell_2 \leftarrow t$$

$$\begin{array}{l} \{x \mapsto a \star y \mapsto b\} \\ \quad \text{swap } x\, y \\ \{\lambda\_.\, x \mapsto b \star y \mapsto a\} \end{array}$$

5-min break

Magic wand

$P \mathrel{-\!\!*} Q$

There is another separation logic operator that turns out to be useful to mechanize proofs in Rocq: the "magic wand", or more formally the "separating implication".

Intuition for magic wand

If you remember only one thing about wand, remember $P \star (P \mathrel{-\!\!\star} Q) \vdash Q$.

Consider the Prop equivalent to get a sense for this as a form of implication, but with separating conjunction rather than regular conjunction

Characterization of magic wand

$$P \vdash (Q \mathrel{-\!\!\star} R) \iff P \star Q \vdash R$$

Exercise: defining magic wand in the model

"if we extend the heap with P, then we get Q"

Answer

$$(P \mathrel{-\!\!*} Q)(h) \triangleq \forall h', (h \perp h') \land P(h') \implies Q(h \cup h')$$

Properties of wand

$$\frac{}{P \star (P \mathrel{-\!\!\star} Q) \vdash Q} \text{ wand-elim}$$

$$\frac{P \star Q \vdash R}{P \vdash (Q \mathrel{-\!\!\star} R)} \text{ wand-intro}$$

Wand implication

$$\frac{P' \vdash P \quad Q \vdash Q'}{(P \mathrel{-\!\!\star} Q) \vdash (P' \mathrel{-\!\!\star} Q')} \text{ wand-impl}$$

Extracting from an array

Application 1: single element of an array

If we had `array(l, xs)` meaning a sequence of consecutive points-to facts, we could prove `array(l, xs) -* l |-> x[n] * (l + n |-> x[n] -* array(l, xs))` (if n is in-bounds!). More fancy: prove `array(l, xs) -* l |-> x[n] * (forall v, l + n |-> v -* array(l, <|n := v>| xs))`

Extracting from a HashMap

Rust Entry API

```
impl<'a, K, V> Entry<'a, K, V>
```

Source

```
pub fn or_insert(self, default: V) -> &'a mut V
```

1.0.0 · Source

Ensures a value is in the entry by inserting the default if empty, and returns a mutable reference to the value in the entry.

Examples

```
use std::collections::HashMap;

let mut map: HashMap<&str, u32> = HashMap::new();

map.entry("poneyland").or_insert(3);
assert_eq!(map["poneyland"], 3);

*map.entry("poneyland").or_insert(10) *= 2;
assert_eq!(map["poneyland"], 6);
```