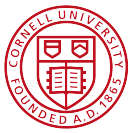
CS 4110 – Programming Languages and Logics
Lecture #16: Separation Logic



So far, we have looked at a version of Hoare logic that supports reasoning about IMP programs in terms of pre- and post-conditions on stores. In this lecture, we consider a language with pointers and dynamic memory allocation. We define a big-step operational semantics for the language as a relation on program states (σ, h) with a store and a heap. We show how Hoare logic fails to support modular reasoning due to aliasing between pointers. Finally, we introduce *separation logic*, which extends Hoare logic's assertions with predicates on heaps and provides a *frame* rule that supports modular proofs.

1 IMP with Heaps

As a first step, we extend IMP with standard commands for manipulating values on the heap. Note: in this lecture we will write e for arithmetic expressions rather than a, as it is more intuitive:

$$\begin{array}{rcl} c & ::= & \textbf{skip} \mid x := e \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c \\ & & \mid x := \textbf{new}(e) \mid \textbf{free}(e) \mid x := *e \mid *e_1 := e_2 \end{array}$$

The new commands in this language include:

- $x := \textbf{new}(e)$, which evaluates e to a value v, allocates storage for v on the heap, and assigns the pointer to that storage to x in the store;
- $x := *e$ which evaluates e to a pointer p, loads the value v associated with p from the heap, and assigns v to x in the store;
- $*e := e_1$ which evaluates e_1 to a pointer p and e_2 to a value v, and then stores v at the location associated with p on the heap; and
- $\textbf{free}(e)$ which evaluates e to a pointer p and deallocates the heap storage associated with p.

2 Big-Step Operational Semantics

To define the semantics of this extension of IMP, we will need states that keep track both of the local variables as well as the heap. We will model states as pairs (σ, h) where

- $\sigma \in \text{Var} \to \mathbb{Z}$ is the store, and
- $h \in \text{Addr} \rightharpoonup_{\text{fin}} \mathbb{Z}$ is the heap.

As usual, we write $dom(h)$ for the domain of the heap and $h[p \mapsto v]$ for the update operation that maps p to v and otherwise behaves like h.

In the formal semantics, commands evaluate in one big step

$$\langle \sigma, h, c \rangle \Downarrow (\sigma', h')$$

while arithmetic and boolean expressions remain pure,

$$\langle \sigma, e \rangle \Downarrow v$$

Basic commands

$$\frac{}{\langle \sigma, h, \textbf{skip} \rangle \Downarrow (\sigma, h)}\ \textsc{Skip} \qquad \frac{\langle \sigma, e \rangle \Downarrow v}{\langle \sigma, h, x := e \rangle \Downarrow (\sigma[x \mapsto v], h)}\ \textsc{Assign}$$

$$\frac{\langle \sigma, h, c_1 \rangle \Downarrow (\sigma_1, h_1) \qquad \langle \sigma_1, h_1, c_2 \rangle \Downarrow (\sigma_2, h_2)}{\langle \sigma, h, c_1; c_2 \rangle \Downarrow (\sigma_2, h_2)}\ \textsc{Seq}$$

Conditionals and loops

$$\frac{\langle \sigma, b \rangle \Downarrow \textbf{true} \qquad \langle \sigma, h, c_1 \rangle \Downarrow (\sigma', h')}{\langle \sigma, h, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \rangle \Downarrow (\sigma', h')}\ \textsc{If-True} \qquad \frac{\langle \sigma, b \rangle \Downarrow \textbf{false} \qquad \langle \sigma, h, c_2 \rangle \Downarrow (\sigma', h')}{\langle \sigma, h, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \rangle \Downarrow (\sigma', h')}\ \textsc{If-False}$$

$$\frac{\langle \sigma, b \rangle \Downarrow \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \rangle \Downarrow (\sigma, h)}\ \textsc{While-False}$$

$$\frac{\langle \sigma, b \rangle \Downarrow \textbf{true} \qquad \langle \sigma, h, c \rangle \Downarrow (\sigma_1, h_1) \qquad \langle \sigma_1, h_1, \textbf{while } b \textbf{ do } c \rangle \Downarrow (\sigma_2, h_2)}{\langle \sigma, h, \textbf{while } b \textbf{ do } c \rangle \Downarrow (\sigma_2, h_2)}\ \textsc{While-True}$$

Heap commands

$$\frac{\langle \sigma, e \rangle \Downarrow p \qquad p \in dom(h) \qquad h(p) = v}{\langle \sigma, h, x := *e \rangle \Downarrow (\sigma[x \mapsto v], h)}\ \textsc{Load} \qquad \frac{\langle \sigma, e_1 \rangle \Downarrow p \qquad \langle \sigma, e_2 \rangle \Downarrow v \qquad p \in dom(h)}{\langle \sigma, h, *e_1 := e_2 \rangle \Downarrow (\sigma,\ h[p \mapsto v])}\ \textsc{Store}$$

$$\frac{\langle \sigma, e \rangle \Downarrow v \qquad p \notin dom(h)}{\langle \sigma, h, x := \textbf{new}(e) \rangle \Downarrow (\sigma[x \mapsto p],\ h[p \mapsto v])}\ \textsc{New} \qquad \frac{\langle \sigma, e \rangle \Downarrow p \qquad p \in dom(h)}{\langle \sigma, h, \textbf{free}(e) \rangle \Downarrow (\sigma,\ h \setminus \{p\})}\ \textsc{Free}$$

3 The Rule of Constancy

The following rule is admissible in standard Hoare logic:

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad fvs(R) \cap \quad \text{mod } c = \emptyset}{\vdash \{P \wedge R\}\ c\ \{Q \wedge R\}}\ \textsc{Const}$$

Here $fvs(R)$ are the free variables of R and $\quad \text{mod } (c)$ are the variables that c may modify. For example, $fvs(y = 0) = \{y\}$ and $\quad \text{mod } (x := x{+}1) = \{x\}$.

Intuitively, the rule of constancy captures a form of local reasoning. It allows us to first prove a simple Hoare triple and then extend it to a more complicated triple by conjoining the same predicate to the pre- and post-conditions. For instance, using this rule so we can strengthen

$$\{x > 0\}\ x := x{+}1\ \{x > 1\}$$

to

$$\{x > 0 \wedge y = 0\}\ x := x{+}1\ \{x > 1 \wedge y = 0\}$$

Free and Modified Variables

The function $fv(P)$ returns the set of program variables that occur free in assertion P.

$$\begin{array}{rcl} fv(a_1 \leq a_2) & = & fv(a_1) \cup fv(a_2) \\ fv(P \wedge Q) & = & fv(P) \cup fv(Q) \\ fv(P \vee Q) & = & fv(P) \cup fv(Q) \\ fv(P \implies Q) & = & fv(P) \cup fv(Q) \\ fv(\neg P) & = & fv(P) \\ fv(\forall i.\, P) & = & fv(P) \setminus \{i\} \\ fv(\exists i.\, P) & = & fv(P) \setminus \{i\} \end{array}$$

The function $mod(c)$ returns the set of program variables modified by command c.

$$\begin{array}{rcl} mod(\textbf{skip}) & = & \emptyset \\ mod(x := e) & = & \{x\} \\ mod(c_1; c_2) & = & mod(c_1) \cup mod(c_2) \\ mod(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) & = & mod(c_1) \cup mod(c_2) \\ mod(\textbf{while } b \textbf{ do } c) & = & mod(c) \\ mod(x := *e) & = & \{x\} \\ mod(*e_1 := e_2) & = & \emptyset \\ mod(x := \textbf{new}(e)) & = & \{x\} \\ mod(\textbf{free}(e)) & = & \emptyset \end{array}$$

Issues Related to Aliasing Unfortunately, the rule of constancy is not sound when we extend our language of assertions with heap predicates. For example, suppose that we add an assertion $p \rightharpoonup v$, which says that the heap maps p to v. Now consider trying to use the rule of constancy to add a "constant fact" about the heap. We might first prove the following triple:

$$\{\, x \rightharpoonup - \,\}\, *x := 4 \,\{\, x \rightharpoonup 4 \,\}.$$

and then use the rule of constancy to obtain:

$$\{\, x \rightharpoonup - \wedge\, y \rightharpoonup 3 \,\}\, *x := 4 \,\{\, x \rightharpoonup 4 \wedge\, y \rightharpoonup 3 \,\}.$$

However, if x and y are aliases for each other, then the postcondition is false. Of course, it is possible to complicate the pre- and post-condition to capture disjointness predicates, but this quickly becomes impractical—in principle, every pointer on the heap might be aliased to every other pointer, which leads to a combinatorial explosion. Such complications motivated the development of separation logic.

4 Heap Assertions

Before defining separation logic, let us extend our language of assertions with heap predicates:

$$H, J, K ::= \textbf{emp} \mid [P] \mid e_1 \hookrightarrow e_2 \mid H_1 \star H_2 \mid H_1 \mathrel{-\!\!*} H_2 \mid H_1 \mathrel{-\!\!\!\wedge} H_2 \mid \forall x.\, H \mid \exists x.\, H$$

Intuitively, these predicates can be understood as follows.

- **emp** : heap is empty,
- $[P]$: heap is empty and P holds,
- $e_1 \hookrightarrow e_2$: heap consists of exactly one cell with pointer e_1 and value e_2,
- $H_1 \star H_2$: heap can be split into two disjoint pieces, one satisfying H_1 and the other satisfying H_2,
- $H_1 \curlywedge H_2$ and $H_1 \curlyvee H_2$ are heap assertion versions of the standard boolean connectives, and
- $\forall x.\, H$ and $\exists x.\, H$: are heap assertion versions of the standard quantifiers.

More formally, we can model their semantics as follows:

$$
\begin{array}{rll}
(\sigma, h) \models_I \mathbf{emp} & \text{if} & h = \emptyset \\
(\sigma, h) \models_I [P] & \text{if} & h = \emptyset \wedge \sigma \models_I P \\
(\sigma, h) \models_I e_1 \hookrightarrow e_2 & \text{if} & h = \{(p, v)\} \text{ where } \langle \sigma, e_1 \rangle_I \Downarrow p \text{ and } \langle \sigma, e_2 \rangle_I \Downarrow v \\
(\sigma, h) \models_I H_1 \star H_2 & \text{if} & \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge (\sigma, h_1) \models_I P_1 \wedge (\sigma, h_2) \models_I P_2 \\
(\sigma, h) \models_I H_1 \curlywedge H_2 & \text{if} & (\sigma, h) \models_I H_1 \text{ and } (\sigma, h) \models_I H_2 \\
(\sigma, h) \models_I H_1 \curlyvee H_2 & \text{if} & (\sigma, h) \models_I H_1 \text{ or } (\sigma, h) \models_I H_2 \\
(\sigma, h) \models_I \forall x.\, H & \text{if} & (\sigma, h) \models_{I[x \mapsto v]} H \text{ for all } v \\
(\sigma, h) \models_I \exists x.\, H & \text{if} & (\sigma, h) \models_{I[x \mapsto v]} H \text{ for some } v
\end{array}
$$

We will often abbreviate $e \rightharpoonup - \triangleq \exists v.\, e \rightharpoonup v$.

Note that predicates like $x \rightharpoonup 5 \star x \rightharpoonup 7$ are *unsatisfiable* as the separating conjunction operator, $\star$, enforces disjointness.

5 Triples and the Frame Rule

Now we will define the notion of valid triples, in both hoare and separation logic. We will present total correctness versions, as we want programs that have been verified to be free of memory errors, and our big-step semantics models memory errors as getting stuck.

We first define valid Hoare triples in the usual way.

Definition (Hoare Triple (Total Correctness))**.** A Hoare logic triple is valid, written $\models_{\text{Hoare}} \{H\}\ c\ \{J\}$, if for all σ, h, and I, if $\sigma, h \models_I H$ then $\langle \sigma, h, c \rangle \Downarrow (\sigma', h')$ and $\sigma', h' \models_I J$

Next, we define valid separation logic triples in terms of valid Hoare triples. Note that K is universally quantified, which essentially bakes in a form of modularity, since a valid triple must remain valid when combined with any other heap predicate.

Definition (Separation Logic Triple (Total Correctness))**.** A separation logic triple is valid, written $\models \{H\}\ c\ \{J\}$, if for all K we have $\models_{\text{Hoare}} \{H \star K\}\ c\ \{J \star K\}$

Writing $h_1 \perp h_2$ to mean that h_1 and h_2 are disjoint heaps, we can also give an equivalent definition of valid separation logic triples.

Definition (Separation Logic Triple (Total Correctness), Alternate)**.** A Hoare logic triple is valid, written $\models_{\text{Hoare}} \{H\}\ c\ \{J\}$, if for all σ, h_1, h_2 and I, if $\sigma, h_1 \models_I H$ and $h_1 \perp h_2$ then $\langle \sigma, h_1 \uplus h_2, c \rangle \Downarrow (\sigma', h_1' \uplus h_2)$ and $\sigma', h_1' \models_I J$

6 The Frame Rule

A major appeal of separation logic is that it supports the so-called frame rule:

$$\frac{\vdash \{H\}\ c\ \{J\} \quad \mathit{fvs}(K) \cap \quad \mathrm{mod}\ c = \emptyset}{\vdash \{H \star K\}\ c\ \{J \star K\}}\ \textsc{Frame}$$

The rule captures modular reasoning about the heap. Intuitively, we first prove a specification for c using only the "heap" footprint it needs. Then the command can be framed in any larger context obtained by combining the footprint with a disjoint heap.

7 Small Axioms for Heap Commands

We can define "small" axioms for the heap-manipulating commands as follows.

$$\frac{}{\{\, e \hookrightarrow e' \,\}\ x := *e\ \{\, [x = e'] \star e \hookrightarrow e' \,\}}\ \textsc{Load}$$

$$\frac{}{\{\, x \hookrightarrow - \,\}\ *x := e\ \{\, x \hookrightarrow e \,\}}\ \textsc{Store}$$

$$\frac{}{\{\, \mathbf{emp} \,\}\ x := \mathbf{new}(e)\ \{\, [x = p] \star p \hookrightarrow e \,\}}\ \textsc{Alloc}$$

$$\frac{}{\{\, e \hookrightarrow - \,\}\ \mathbf{free}(e)\ \{\, \mathbf{emp} \,\}}\ \textsc{Free}$$

A full treatment of separation logic has other proof rules similar to Hoare logic. Interested students are referred to two canonical treatments:

- Reynold's lecture notes, or
- Charguéraud's textbook, which this lecture is based on.