Hoare Logic and Model Checking

Kasper Svendsen
University of Cambridge

CST Part II – 2016/17

Acknowledgement: slides heavily based on previous versions by Mike Gordon and Alan Mycroft

Pointers

Pointers and state

So far, we have been reasoning about a language without pointers, where all values were numbers.

In this lecture we will extend the WHILE language with pointers and introduce an extension of Hoare logic, called Separation Logic, to simplify reasoning about pointers.

Pointers and state

$$
\begin{array}{lcll}
E & ::= & N \mid \textbf{null} \mid V \mid E_1 + E_2 \mid & \textit{expressions} \\
 & \mid & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \\
 & & & \\
B & ::= & T \mid F \mid E_1 = E_2 & \textit{boolean expressions} \\
 & \mid & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots & \\
 & & & \\
C & ::= & \textbf{skip} \mid C_1; C_2 \mid V := E & \textit{commands} \\
 & \mid & \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 & \\
 & \mid & \textbf{while } B \textbf{ do } C & \\
 & \mid & V := [E] \mid [E_1] := E_2 & \\
 & \mid & V := \textbf{cons}(E_1, ..., E_n) \mid \textbf{dispose}(E) &
\end{array}
$$

Pointers and state

Commands are now evaluated with respect to a **heap** h that stores the current value of allocated locations.

Reading, writing and disposing of pointers fails if the given location is not currently allocated.

Fetch assignment command: $V := [E]$

- evaluates E to a location l and assigns the current value of l to V; faults if l is not currently allocated

Pointers and state

Heap assignment command: $[E_1] := E_2$

- evaluates E_1 to a location l and E_2 to a value v and updates the heap to map l to v; faults if l is not currently allocated

Pointer disposal command, **dispose**(E)

- evaluates E to a location l and deallocates location l from the heap; faults if l is not currently allocated

Pointers and state

Allocation assignment command: $V := \mathbf{cons}(E_1, ..., E_n)$

- chooses n **consecutive** unallocated locations, say $l_1, ..., l_n$, evaluates $E_1, ..., E_n$ to values $v_1, ..., v_n$, updates the heap to map l_i to v_i for each i and assigns l_1 to V

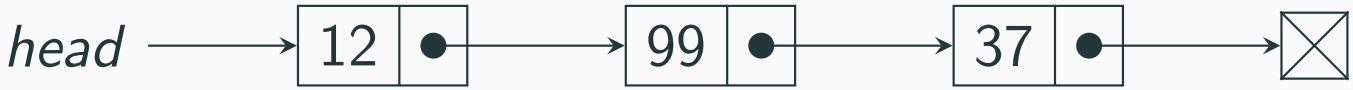
Allocation never fails.

The language supports pointer arithmetic: e.g.,

$$X := \mathbf{cons}(0, 1); Y := [X + 1]$$

Pointers and state

In this extended language we can work with proper data structures, like the following singly-linked list.



For instance, this operation deletes the first element of the list:

```
x := [head + 1];        // lookup address of second element
dispose(head);          // deallocate first element
dispose(head + 1);
head := x               // swing head to point to second element
```

Operational semantics

Pointers and state

For the WHILE language we modelled the state as a function assigning values (numbers) to all variables:

$$s \in \mathit{State} \stackrel{\mathit{def}}{=} \mathit{Var} \to \mathit{Val}$$

To model pointers we will split the state into a **stack** and a **heap**

- a stack assigns values to program variables, and
- a heap maps locations to values

$$\mathit{State} \stackrel{\mathit{def}}{=} \mathit{Store} \times \mathit{Heap}$$

Pointers and state

Values now includes both numbers and locations

$$Val \stackrel{def}{=} \mathbb{Z} + Loc$$

Locations are modelled as natural numbers

$$Loc \stackrel{def}{=} \mathbb{N}$$

To model allocation, we model the heap as a **finite** function

$$Store \stackrel{def}{=} Var \to Val \qquad\qquad Heap \stackrel{def}{=} Loc \stackrel{fin}{\to} Val$$

Pointers and state

WHILE_p programs can fail in several ways

- dereferencing an invalid pointer
- invalid pointer arithmetic

To model failure we introduce a distinguished failure value $\lightning$

$$\mathcal{E}[\![-]\!] : \mathit{Exp} \times \mathit{Store} \to \{\lightning\} + \mathit{Val}$$
$$\mathcal{B}[\![-]\!] : \mathit{BExp} \times \mathit{Store} \to \{\lightning\} + \mathbb{B}$$
$$\Downarrow : \mathcal{P}(\mathit{Cmd} \times \mathit{State} \times (\{\lightning\} \cup \mathit{State}))$$

Pointer dereference

$$\frac{\mathcal{E}[\![E]\!](s) = l \qquad l \in dom(h)}{\langle V := [E], (s, h)\rangle \Downarrow (s[V \mapsto h(l)], h)}$$

$$\frac{\mathcal{E}[\![E]\!](s) = l \qquad l \notin dom(h)}{\langle V := [E], (s, h)\rangle \Downarrow \lightning}$$

Pointer assignment

$$\frac{\mathcal{E}[\![E_1]\!](s) = l \qquad \mathcal{E}[\![E_2]\!](s) = v \qquad l \in dom(h) \qquad v \neq \lightning}{\langle [E_1] := E_2, (s,h)\rangle \Downarrow (s, h[l \mapsto v])}$$

$$\frac{\mathcal{E}[\![E_1]\!](s) = l \qquad l \notin dom(h)}{\langle [E_1] := E_2, (s,h)\rangle \Downarrow \lightning} \qquad \frac{\mathcal{E}[\![E_2]\!](s) = \lightning}{\langle [E_1] := E_2, (s,h)\rangle \Downarrow \lightning}$$

Reasoning about pointers

Reasoning about pointers

In standard Hoare logic we can syntactically approximate the set of program variables that might be affected by a command C.

$$
\begin{aligned}
mod(\textbf{skip}) &= \emptyset \\
mod(X := E) &= \{X\} \\
mod(C_1; C_2) &= mod(C_1) \cup mod(C_2) \\
mod(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2) &= mod(C_1) \cup mod(C_2) \\
mod(\textbf{while } B \textbf{ do } C) &= mod(C)
\end{aligned}
$$

The rule of constancy

The rule of constancy expresses that assertions that do not refer to variables modified by a command are automatically preserved during its execution.

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad mod(C) \cap FV(R) = \emptyset}{\vdash \{P \wedge R\}\ C\ \{Q \wedge R\}}$$

This rule derivable in standard Hoare logic.

This rule is important for **modularity** as it allows us to only mention the part of the state that we access.

Reasoning about pointers

Imagine we extended Hoare logic with a new assertion, $E_1 \hookrightarrow E_2$, for asserting that location E_1 currently contains the value E_2 and extend the proof system with the following axiom:

$$\frac{}{\vdash \{\top\}\ [E_1] := E_2\ \{E_1 \hookrightarrow E_2\}}$$

Then we loose the rule of constancy:

$$\frac{\vdash \{\top\}\ [X] := 1\ \{X \hookrightarrow 1\}}{\vdash \{\top \wedge Y \hookrightarrow 0\}\ [X] := 1\ \{X \hookrightarrow 1 \wedge Y \hookrightarrow 0\}}$$

(the post-condition is false if X and Y refer to the same location.)

Reasoning about pointers

In the presence of pointers, syntactically distinct variables can refer to the same location. Updates made through one variable can thus influence the state referenced by other variables.

This complicates reasoning as we explicitly have to track inequality of pointers to reason about updates:

$$\vdash \{E_1 \neq E_3 \land E_3 \hookrightarrow E_4\}\ [E_1] := E_2\ \{E_1 \hookrightarrow E_2 \land E_3 \hookrightarrow E_4\}$$

Separation logic

Separation logic

Separation logic is an extension of Hoare logic that simplifies reasoning about mutable state using new connectives to control aliasing.

Separation logic was proposed by John Reynolds in 2000 and developed further by Peter O'Hearn and Hongsek Yang around 2001. It is still a very active area of research.

Separation logic

Separation logic introduces two new concepts for reasoning about mutable state::

- **ownership**: Separation logic assertions do not just describe properties of the current state, they also assert ownership of part of the heap.
- **separation**: Separation logic introduces a new connective, written $P * Q$, for asserting that the part of the heap owned by P and Q are **disjoint**.

This makes it easy to describe data structures without sharing.

Separation logic

Separation logic introduces a new assertion, written $E_1 \mapsto E_2$, for reasoning about individual heap cells.

The points-to assertion, $E_1 \mapsto E_2$, asserts

- that the current value of heap location E_1 is E_2, and
- asserts ownership of heap location E_1.

Meaning of separation logic assertions

The semantics of a separation logic assertion, written $[\![P]\!](s)$, is a set of heaps that satisfy the assertion P.

The intended meaning is that if $h \in [\![P]\!](s)$ then P asserts ownership of any locations in $dom(h)$.

The heaps $h \in [\![P]\!](s)$ are thus referred to as **partial heaps**, since they only contain the locations owned by P.

The empty heap assertion, only holds for the empty heap:

$$[\![emp]\!](s) \stackrel{def}{=} \{[]\}$$

Meaning of separation logic assertions

The points-to assertion, $E_1 \mapsto E_2$, asserts ownership of the location referenced by E_1 and that this location currently contains E_2:

$$[\![E_1 \mapsto E_2]\!](s) \stackrel{def}{=} \{h \mid dom(h) = \{\mathcal{E}[\![E_1]\!](s)\} \\ \wedge\, h(\mathcal{E}[\![E_1]\!](s)) = \mathcal{E}[\![E_2]\!](s)\}$$

Separating conjunction, $P * Q$, asserts that the heap can be split into two distjoint parts such that one satisfies P and the other Q:

$$[\![P * Q]\!](s) \stackrel{def}{=} \{h \mid \exists h_1, h_2.\, h = h_1 \uplus h_2 \\ \wedge\, h_1 \in [\![P]\!](s) \wedge h_2 \in [\![Q]\!](s)\}$$

Here we use $h_1 \uplus h_2$ as shorthand for $h_1 \cup h_2$ where $h_1 \uplus h_2$ is only defined when $dom(h_1) \cap dom(h_2) = \emptyset$.

Examples of separation logic assertions

1. $X \mapsto E_1 * Y \mapsto E_2$

   This assertion is unsatisfiable in a state where X and Y refer to the same location, since $X \mapsto E_1$ and $Y \mapsto E_2$ would both assert ownership of the same location.

   The following heap satisfies the assertion:

2. $X \mapsto E * X \mapsto E$

   This assertion is not satisfiable.

Meaning of separation logic assertions

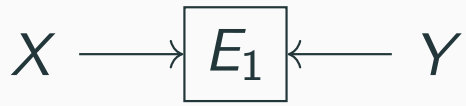
The first-order primitives are interpreted much like for Hoare logic:

$$
\begin{aligned}
[\![\bot]\!](s) &\stackrel{def}{=} \emptyset \\
[\![\top]\!](s) &\stackrel{def}{=} \mathit{Heap} \\
[\![P \wedge Q]\!](s) &\stackrel{def}{=} [\![P]\!](s) \cap [\![Q]\!](s) \\
[\![P \vee Q]\!](s) &\stackrel{def}{=} [\![P]\!](s) \cup [\![Q]\!](s) \\
[\![P \Rightarrow Q]\!](s) &\stackrel{def}{=} \{h \mid h \in [\![P]\!](s) \Rightarrow h \in [\![Q]\!](s)\} \\
&\vdots
\end{aligned}
$$

Examples of separation logic assertions

3. $X \mapsto E_1 \wedge Y \mapsto E_2$

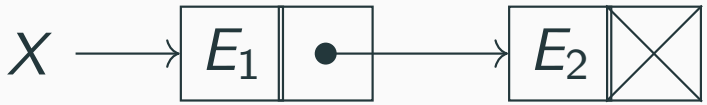
   This asserts that X and Y alias each other and $E_1 = E_2$:



Examples of separation logic assertions

4. $X \mapsto Y * Y \mapsto X$

5. $X \mapsto E_1, Y * Y \mapsto E_2, \mathbf{null}$



Here $X \mapsto E_1, ..., E_n$ is shorthand for

$$X \mapsto E_1 * (X + 1) \mapsto E_2 * \cdots * (X + n - 1) \mapsto E_n$$

Summary: Separation logic assertions

Separation logic assertions **describe** properties of the current state and assert **ownership** of parts of the current heap.

Separation logic controls aliasing of pointers by asserting that assertions own **disjoint** heap parts.

Separation logic triples

Separation logic triples

Separation logic (SL) extends the assertion language but uses the same Hoare triples to reason about the behaviour of programs

$$\vdash \{P\}\ C\ \{Q\} \qquad\qquad \vdash [P]\ C\ [Q]$$

but with a different meaning.

Our SL triples extend the meaning of our HL triples in two ways

- they ensure that our WHILE_p programs do not fail
- they require that we respect the ownership discipline associated with assertions

Separation logic triples

Separation logic triples require that we assert ownership in the precondition of any heap-cells modified.

For instance, the following triple asserts ownership of the location denoted by X and stores the value 2 at this location

$$\vdash \{X \mapsto 1\}\ [X] := 2\ \{X \mapsto 2\}$$

However, the following triple is not valid, because it updates a location that it may not be the owner of

$$\nvdash \{Y \mapsto 1\}\ [X] := 2\ \{Y \mapsto 1\}$$

Framing

How can we make this idea that triples must assert ownership of the heap-cells they modify precise?

The idea is to require that all triples must preserve any assertions disjoint from the precondition. This is captured by the frame-rule:

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad mod(C) \cap FV(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The assertion R is called the frame.

Framing

How does preserving all frames force triples to assert ownership of heap-cells they modify?

Imagine that the following triple did hold and preserved all frames:

$$\{Y \mapsto 1\}\ [X] := 2\ \{Y \mapsto 1\}$$

In particular, it would preserve the frame $x \mapsto 1$:

$$\{Y \mapsto 1 * X \mapsto 1\}\ [X] := 2\ \{Y \mapsto 1 * X \mapsto 1\}$$

This triple definitely does not hold, since the location referenced by X contains 2 in the terminal state.

Framing

This problem does not arise for triples that assert ownership of the heap-cells they modify, since triples only have to preserve frames **disjoint** from the precondition.

For instance, consider this triple which does assert ownership of X

$$\{X \mapsto 1\}\ [X] := 2\ \{X \mapsto 2\}$$

If we frame on $X \mapsto 1$ then we get the following triple which holds vacuously since no initial states satisfies $X \mapsto 1 * X \mapsto 1$.

$$\{X \mapsto 1 * X \mapsto 1\}\ [X] := 2\ \{X \mapsto 2 * X \mapsto 1\}$$

Meaning of Separation logic triples

The meaning of $\{P\}\ C\ \{Q\}$ in Separation logic is thus

- C does not fault when executed in an initial state satisfying P,
- if C terminates in a terminal state when executed from an initial heap $h_1 \uplus h_F$ where h_1 satisfies P then the terminal state has the form $h'_1 \uplus h_F$ where h'_1 satisfies Q

This bakes-in the requirement that triples must satisfy framing, by requiring that they preserve all disjoint frames h_F.

Meaning of Separation logic triples

Written formally, the meaning is:

$$
\begin{aligned}
&\models \{P\}\ C\ \{Q\} \stackrel{def}{=} \\
&\quad (\forall s, h.\, h \in [\![P]\!](s) \Rightarrow \neg(\langle C, (s,h)\rangle \Downarrow \lightning)) \wedge \\
&\quad (\forall s, s', h, h', h_F.\, dom(h) \cap dom(h_F) = \emptyset \wedge \\
&\qquad h \in [\![P]\!](s) \wedge \langle C, (s, h \uplus h_F)\rangle \Downarrow (s', h') \\
&\qquad\quad \Rightarrow \exists h_1'.\, h' = h_1' \uplus h_F \wedge h_1' \in [\![Q]\!](s'))
\end{aligned}
$$

Summary

Separation logic is an extension of Hoare logic with new primitives to simplify reasoning about pointers.

Separation logic extends Hoare logic with a notion of **ownership** and **separation** to control aliasing and reason about shared mutable data structures.

Suggested reading:

- John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures.