Hoare logic

Lecture 5: Introduction to separation logic

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Introduction

In the previous lectures, we have considered a language, WHILE, where mutability only concerned program variables.

In this lecture, we will extend the WHILE language with pointer operations on a heap, and introduce an extension of Hoare logic, called separation logic, to enable practical reasoning about pointers.

WHILE$_p$, a language with pointers

Syntax of WHILE$_p$

We introduce new commands to manipulate the heap:

$$
\begin{array}{rcll}
E & ::= & N \mid V \mid E_1 + E_2 & \textit{arithmetic expressions} \\
 & \mid & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \\
\textbf{null} & \stackrel{\textit{def}}{=} & 0 & \\
 & & & \\
B & ::= & \mathbf{T} \mid \mathbf{F} \mid E_1 = E_2 & \textit{boolean expressions} \\
 & \mid & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots & \\
 & & & \\
C & ::= & \textbf{skip} \mid C_1; C_2 \mid V := E & \textit{commands} \\
 & \mid & \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 & \\
 & \mid & \textbf{while } B \textbf{ do } C & \\
 & \mid & V := [E] \mid [E_1] := E_2 & \\
 & \mid & V := \textbf{alloc}(E_0, ..., E_n) & \\
 & \mid & \textbf{dispose}(E) &
\end{array}
$$

The heap

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

Heap assignment, dereferencing, and deallocation fail if the given locations are not currently allocated.

This is a design choice that makes WHILE_p more like a programming language, whereas having a heap with all locations always allocated would make WHILE_p more like assembly.

It allows us to consider faults, and how separation logic can be used to prevent faults, and it also makes things clearer.

Heap usage commands

Heap assignment command $[E_1] := E_2$

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

Heap dereferencing command $V := [E]$

- evaluates E to a location ℓ, and assigns the value that ℓ maps to to V; faults if ℓ is not currently allocated.

We could have heap dereferencing be an expression, but then expressions would fault, which would add complexity.

Heap management commands

Allocation assignment command: $V := \textbf{alloc}(E_0, ..., E_n)$

- chooses $n + 1$ **consecutive** unallocated locations starting at location ℓ, evaluates $E_0, ..., E_n$ to values $N_0, ..., N_n$, updates the heap to map $\ell + i$ to N_i for each i, and assigns ℓ to V.

In WHILE_p, allocation never faults.
A real machine would run out of memory at some point.

Deallocation command $\textbf{dispose}(E)$

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

Pointers

WHILE_p has proper pointer operations, as opposed for example to references:

- pointers can be invalid: $X := [\textbf{null}]$ faults
- we can perform pointer arithmetic:
  - $X := \textbf{alloc}(0, 1); Y := [X + 1]$
  - $X := \textbf{alloc}(0); \textbf{if } X = 3 \textbf{ then } [3] := 1 \textbf{ else } [X] := 2$

We do not have a separate type of pointers: we use integers as pointers.

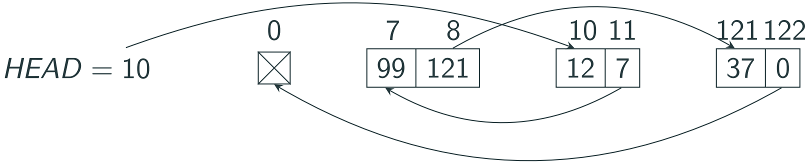
Pointers in C have many more subtleties. For example, in C, pointers can point to the stack.

Pointers and data structures

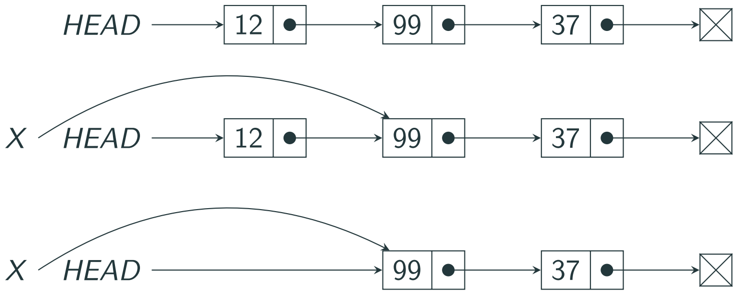
In WHILE_p, we can encode data structures in the heap. For example, we can encode the mathematical list $[12, 99, 37]$ with the following singly-linked list:

In WHILE, we would have had to encode that in integers, for example as $HEAD = 2^{12} \times 3^{99} \times 5^{37}$ (as in Part IB Computation theory).

More concretely:



Operations on mutable data structures



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*

Dynamic semantics of $\mathbf{WHILE}_p$

States of WHILE$_p$

For the WHILE language, we modelled the state as a function mapping program variables to values (integers):

$$s \in \mathit{Stack} \stackrel{\mathit{def}}{=} \mathit{Var} \to \mathbb{Z}$$

For WHILE_p, we extend the state to be composed of a **stack** and a **heap**, where

- the stack maps program variables to values (as before), and
- the heap maps allocated locations to values.

We have

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

Heaps

We elect for locations to be non-negative integers:

$$\ell \in \mathit{Loc} \stackrel{\mathit{def}}{=} \{\ell \in \mathbb{Z} \mid 0 \leq \ell\}$$

null is a location, but a “bad” one, that is never allocated.

To model the fact that only a finite number of locations is allocated at any given time, we model the heap as a **finite** function, that is, a partial function with a finite domain:

$$h \in \mathit{Heap} \stackrel{\mathit{def}}{=} (\mathit{Loc} \setminus \{\textbf{null}\}) \stackrel{\mathit{fin}}{\rightarrow} \mathbb{Z}$$

Failure of commands

WHILE_p commands can fail by:

- dereferencing an invalid pointer,
- assigning to an invalid pointer, or
- deallocating an invalid pointer.

because the location expression we provided does not evaluate to a location, or evaluates to a location that is not allocated (which includes **null**).

To explicitly model failure, we introduce a distinguished failure value $\lightning$, and adapt the semantics:

$$\Downarrow : \mathcal{P}(\mathit{Cmd} \times \mathit{State} \times (\{\lightning\} + \mathit{State}))$$

We could instead just leave the configuration stuck, but explicit failure makes things clearer and easier to state.

Adapting the base constructs to handle the heap

The base constructs can be adapted to handle the extended state in the expected way:

$$\frac{\mathcal{E}[\![E]\!](s) = N}{\langle V := E, (s, h)\rangle \Downarrow (s[V \mapsto N], h)} \qquad \frac{\langle C_1, (s, h)\rangle \Downarrow (s', h') \qquad \langle C_2, (s', h')\rangle \Downarrow (s'', h'')}{\langle C_1; C_2, (s, h)\rangle \Downarrow (s'', h'')}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C_1, (s, h)\rangle \Downarrow (s', h')}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \Downarrow (s', h')} \qquad \frac{\mathcal{B}[\![B]\!](s) = \bot \qquad \langle C_2, s\rangle \Downarrow (s', h')}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, (s, h)\rangle \Downarrow (s', h')}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C, (s, h)\rangle \Downarrow (s', h') \qquad \langle \textbf{while } B \textbf{ do } C, (s', h')\rangle \Downarrow (s'', h'')}{\langle \textbf{while } B \textbf{ do } C, (s, h)\rangle \Downarrow (s'', h'')}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \bot}{\langle \textbf{while } B \textbf{ do } C, (s, h)\rangle \Downarrow (s, h)} \qquad \frac{}{\langle \textbf{skip}, (s, h)\rangle \Downarrow (s, h)}$$

Adapting the base constructs to handle failure

They can also be adapted to handle failure in the expected way:

$$\frac{\langle C_1, (s, h)\rangle \Downarrow \lightning}{\langle C_1; C_2, (s, h)\rangle \Downarrow \lightning} \qquad \frac{\langle C_1, s\rangle \Downarrow (s', h') \qquad \langle C_2, (s', h')\rangle \Downarrow \lightning}{\langle C_1; C_2, (s, h)\rangle \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C_1, (s, h)\rangle \Downarrow \lightning}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, (s, h)\rangle \Downarrow \lightning} \qquad \frac{\mathcal{B}[\![B]\!](s) = \bot \qquad \langle C_2, (s, h)\rangle \Downarrow \lightning}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, (s, h)\rangle \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C, (s, h)\rangle \Downarrow \lightning}{\langle \textbf{while } B \textbf{ do } C, (s, h)\rangle \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C, (s, h)\rangle \Downarrow (s', h') \qquad \langle \textbf{while } B \textbf{ do } C, (s', h')\rangle \Downarrow \lightning}{\langle \textbf{while } B \textbf{ do } C, (s, h)\rangle \Downarrow \lightning}$$

Heap dereferencing

Dereferencing an allocated location stores the value at that location to the target program variable:

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

Dereferencing an unallocated location and dereferencing something that is not a location lead to a fault:

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

Heap assignment

Assigning to an allocated location updates the heap at that location with the assigned value:

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

Assigning to an unallocated location or to something that is not a location leads to a fault:

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

For reference: deallocation

Deallocating an allocated location removes that location from the heap:

$$\frac{\mathcal{E}[\![E]\!](s) = \ell \qquad \ell \in dom(h)}{\langle \textbf{dispose}(E), (s, h)\rangle \Downarrow (s, h \setminus \{(\ell, h(\ell))\})}$$

Deallocating an unallocated location or something that is not a location leads to a fault:

$$\frac{\mathcal{E}[\![E]\!](s) = \ell \qquad \ell \notin dom(h)}{\langle \textbf{dispose}(E), (s, h)\rangle \Downarrow \lightning} \qquad \frac{\nexists \ell.\, \mathcal{E}[\![E]\!](s) = \ell}{\langle \textbf{dispose}(E), (s, h)\rangle \Downarrow \lightning}$$

For reference: allocation

Allocating finds a block of unallocated locations of the right size, updates the heap at those locations with the initialisation values, and stores the start-of-block location to the target program variable:

$$\frac{\mathcal{E}[\![E_0]\!](s) = N_0 \qquad \ldots \qquad \mathcal{E}[\![E_n]\!](s) = N_n \qquad \forall i \in \{0, \ldots, n\}.\, \ell + i \notin dom(h) \qquad \ell \neq \mathbf{null}}{\langle V := \mathbf{alloc}(E_0, \ldots, E_n), (s, h)\rangle \Downarrow (s[V \mapsto \ell], h[\ell \mapsto N_1, \ldots, \ell + n \mapsto N_n])}$$

Because the heap has a finite domain, it is always possible to pick a suitable ℓ, so allocation never faults.

Attempting to reason about pointers in Hoare logic

Attempting to reason about pointers in Hoare logic

We will show that reasoning about pointers in Hoare logic is not practicable.

To do so, we will first show what makes compositional reasoning possible in standard Hoare logic (without pointers), and then show how it fails when we introduce pointers.

Approximating modified program variables

We can syntactically overapproximate the set of program variables that might be modified by a command C:

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

$$\begin{aligned}
mod([E_1] := E_2) &= \emptyset \\
mod(V := [E]) &= \{V\} \\
mod(V := \textbf{alloc}(E_0, \ldots, E_n)) &= \{V\} \\
mod(\textbf{dispose}(E)) &= \emptyset
\end{aligned}$$

For reference: free variables

The set of free variables of a term and of an assertion is given by

$$\begin{aligned} FV(-) &: Term \to \mathcal{P}(Var) \\ FV(\nu) &\stackrel{def}{=} \{\nu\} \\ FV(f(t_1, \ldots, t_n)) &\stackrel{def}{=} FV(t_1) \cup \ldots \cup FV(t_n) \end{aligned}$$

and

$$\begin{aligned} FV(-) &: Assertion \to \mathcal{P}(Var) \\ FV(\top) = FV(\bot) &\stackrel{def}{=} \emptyset \\ FV(P \wedge Q) = FV(P \vee Q) = FV(P \Rightarrow Q) &\stackrel{def}{=} FV(P) \cup FV(Q) \\ FV(\forall v.\, P) = FV(\exists v.\, P) &\stackrel{def}{=} FV(P) \setminus \{v\} \\ FV(t_1 = t_2) &\stackrel{def}{=} FV(t_1) \cup FV(t_2) \\ FV(p(t_1, \ldots, t_n)) &\stackrel{def}{=} FV(t_1) \cup \ldots FV(t_n) \end{aligned}$$

respectively.

The rule of constancy

In standard Hoare logic (without the rules that we will introduce later, and thus without the new commands we have introduced), the rule of constancy expresses that assertions that do not refer to program 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 is admissible in standard Hoare logic.

Modularity and the rule of constancy

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

Using the rule of constancy, we can **separately** verify two complicated commands:

$$\vdash \{P\}\ C_1\ \{Q\} \qquad\qquad \vdash \{R\}\ C_2\ \{S\}$$

and then, as long as they use different program variables, we can compose them.

For example, if $mod(C_1) \cap FV(R) = \emptyset$ and $mod(C_2) \cap FV(Q) = \emptyset$, we can compose them sequentially:

$$\dfrac{\dfrac{\vdash \{P\}\ C_1\ \{Q\} \qquad mod(C_1) \cap FV(R) = \emptyset}{\vdash \{P \wedge R\}\ C_1\ \{Q \wedge R\}} \qquad \dfrac{\vdash R \wedge Q \Rightarrow Q \wedge R \qquad \dfrac{\vdash \{R\}\ C_2\ \{S\} \qquad mod(C_2) \cap FV(Q) = \emptyset}{\vdash \{R \wedge Q\}\ C_2\ \{S \wedge Q\}} \qquad \vdash S \wedge Q \Rightarrow Q \wedge S}{\vdash \{Q \wedge R\}\ C_2\ \{Q \wedge S\}}}{\vdash \{P \wedge R\}\ C_1; C_2\ \{Q \wedge S\}}$$

A bad rule for reasoning about pointers

Imagine we extended Hoare logic with a new assertion, $t_1 \hookrightarrow t_2$, for asserting that location t_1 currently contains the value t_2, and extended the proof system with the following (sound) rule:

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

Then we would lose the rule of constancy, as using it, we would be able to derive

$$\frac{\vdash \{\top\}\ [37] := 42\ \{37 \hookrightarrow 42\} \qquad \mathit{mod}([37] := 42) \cap FV(Y \hookrightarrow 0) = \emptyset}{\vdash \{\top \wedge Y \hookrightarrow 0\}\ [37] := 42\ \{37 \hookrightarrow 42 \wedge Y \hookrightarrow 0\}}$$

even if $Y = 37$, in which case the postcondition would require 0 to be equal to 42.

There is a problem!

Reasoning about pointers

In the presence of pointers, we can have **aliasing**: syntactically distinct expressions can refer to the same location. Updates made through one expression can thus influence the state referenced by other expressions.

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

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

We have to assume that any location is possibly modified unless stated otherwise in the precondition. This is not compositional at all, and quickly becomes unmanageable.

Separation logic

Separation logic

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

The variant of separation logic that we are going to consider, which is suited to reason about an explicitly managed heap (as opposed to a heap with garbage collection), is called classical separation logic (as opposed to intuitionistic separation logic).

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

Concepts of separation logic

Separation logic introduces two new concepts for reasoning about pointers:

- **ownership**: separation logic assertions not only describe properties of the current state (as Hoare logic assertions did), but also assert ownership of part of the heap.
- **separation**: separation logic introduces a new connective for reasoning about the combination of **disjoint** parts of the heap.

The points-to assertion

Separation logic introduces a new assertion, written $t_1 \mapsto t_2$, and read "t_1 points to t_2", for reasoning about individual heap cells.

The points-to assertion $t_1 \mapsto t_2$

- asserts that the current value that heap location t_1 maps to is t_2 (like $t_1 \hookrightarrow t_2$), and
- asserts ownership of heap location t_1.

For example, $X \mapsto Y + 1$ asserts that the current value of heap location X is $Y + 1$, and moreover asserts ownership of that heap location.

The separating conjunction

Separation logic introduces a new connective, the separating conjunction $*$, for reasoning about disjointedness.

The assertion $P * Q$ asserts that P and Q hold (like $P \wedge Q$), and that moreover the parts of the heap owned by P and Q are **disjoint**.

The separating conjunction has a neutral element, *emp*, which describes the empty heap: $emp * P \Leftrightarrow P \Leftrightarrow P * emp$.

Examples of separation logic assertions

1. $(X \mapsto t_1) * (Y \mapsto t_2)$

   This assertion is unsatisfiable in a state where X and Y refer to the same location, since $X \mapsto t_1$ and $Y \mapsto t_2$ would both assert ownership of the same location.

   The following heap satisfies the assertion:

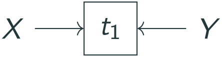
2. $(X \mapsto t) * (X \mapsto t)$

   This assertion is not satisfiable, as X is not disjoint from itself.

Examples of separation logic assertions

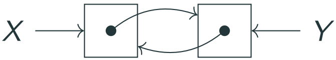
3. $X \mapsto t_1 \wedge Y \mapsto t_2$

   This asserts that X and Y alias each other and $t_1 = t_2$:

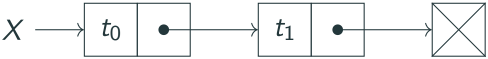


Examples of separation logic assertions

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



5. $(X \mapsto t_0, Y) * (Y \mapsto t_1, \mathbf{null})$



Here, $X \mapsto t_0, ..., t_n$ is shorthand for

$$(X \mapsto t_0) * ((X + 1) \mapsto t_1) * \cdots * ((X + n) \mapsto t_n)$$

Example use of the separating conjunction

6. $\exists x, y.\ (HEAD \mapsto 12, x) * (x \mapsto 99, y) * (y \mapsto 37, \textbf{null})$

   This describes our singly linked list from earlier:

Semantics of separation logic assertions

Semantics of separation logic assertions

The semantics of a separation logic assertion P, $[\![P]\!]$, is the set of states (that is, pairs of a stack and a heap) that satisfy P.

It is simpler to define it indirectly, through the semantics of P given a store s, written $[\![P]\!](s)$, which is the set of heaps that, together with stack s, satisfy P.

Recall that we want to capture the notion of ownership:
if $h \in [\![P]\!](s)$, then P should assert 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.

Semantics of separation logic assertions

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

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

Semantics of separation logic assertions: points-to

The points-to assertion $t_1 \mapsto t_2$ asserts ownership of the location referenced by t_1, and that this location currently contains t_2:

$$[\![t_1 \mapsto t_2]\!](s) \stackrel{def}{=} \left\{ h \in Heap \;\middle|\; \exists \ell, N. \begin{array}{l} [\![t_1]\!](s) = \ell \wedge \\ \ell \neq \textbf{null} \wedge \\ [\![t_2]\!](s) = N \wedge \\ dom(h) = \{\ell\} \wedge \\ h(\ell) = N \end{array} \right\}$$

$t_1 \mapsto t_2$ only asserts ownership of location ℓ, so to capture ownership, $dom(h) = \{\ell\}$.

Semantics of separation logic assertions: $*$

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

$$[\![P * Q]\!](s) \stackrel{def}{=} \left\{ h \in Heap \;\middle|\; \exists h_1, h_2. \;\begin{array}{l} h_1 \in [\![P]\!](s) \wedge \\ h_2 \in [\![Q]\!](s) \wedge \\ h = h_1 \uplus h_2 \end{array} \right\}$$

where $h = h_1 \uplus h_2$ is equal to $h = h_1 \cup h_2$, but only holds when $dom(h_1) \cap dom(h_2) = \emptyset$.

Semantics of separation logic assertions: *emp*

The empty heap assertion only holds for the empty heap:

$$[\![emp]\!](s) \stackrel{def}{=} \{h \in Heap \mid dom(h) = \emptyset\}$$

emp does not assert ownership of any location, so to capture ownership, $dom(h) = \emptyset$.

Summary: separation logic assertions

Separation logic assertions not only **describe** properties of the current state (as Hoare logic assertions did), but also assert **ownership** of parts of the current heap.

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

Semantics of separation logic triples

Semantics of separation logic triples

Separation logic not only extends the assertion language, but strengthens the semantics of correctness triples in two ways:

- they ensure that commands do not fail;
- they ensure that the ownership discipline associated with assertions is respected.

Ownership and separation logic triples

Separation logic triples ensure that the ownership discipline is respected by requiring that the precondition asserts ownership of any heap cells that the command might use.

For instance, we want the following triple, which asserts ownership of location 37, stores the value 42 at this location, and asserts that after that location 37 contains value 42, to be valid:

$$\vdash \{37 \mapsto 1\}\ [37] := 42\ \{37 \mapsto 42\}$$

However, we do not want the following triple to be valid, because it updates a location that it is not the owner of:

$$\nvdash \{100 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1\}$$

even though the precondition ensures that the postcondition is true!

Framing

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

The idea is to require that all triples must preserve any assertion that asserts ownership of a part of the heap disjoint from the part of the heap that their precondition asserts ownership of.

This is exactly what the separating conjunction, $*$, allows us to express.

The frame rule

This intent that all triples preserve any assertion R disjoint from the precondition, called the frame, 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 frame rule is similar to the rule of constancy, but uses the separating conjunction to express separation.

We still need to be careful about program variables (in the stack), so we need $mod(C) \cap FV(R) = \emptyset$.

Examples of 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:

$$\{100 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1\}$$

In particular, it would preserve the frame $37 \mapsto 1$:

$$\{100 \mapsto 1 * 37 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1 * 37 \mapsto 1\}$$

This triple definitely does not hold, since location 37 contains 42 in the terminal state.

Examples of 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 asserts ownership of location 37:

$$\{37 \mapsto 1\}\ [37] := 42\ \{37 \mapsto 42\}$$

If we frame on $37 \mapsto 1$, then we get the following triple, which holds vacuously since no initial states satisfies $37 \mapsto 42 * 37 \mapsto 1$:

$$\{37 \mapsto 1 * 37 \mapsto 1\}\ [37] := 42\ \{37 \mapsto 42 * 37 \mapsto 1\}$$

Informal semantics 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, and
- if h_1 satisfies P, and if when executed from an initial state with an initial heap $h_1 \uplus h_F$, C terminates, then the terminal heap 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 heaps h_F.

Formal semantics of separation logic triples

Written formally, the semantics is:

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

We then have the semantic version of the frame rule baked in:
If $\models \{P\}\ C\ \{Q\}$ and $mod(C) \cap FV(R) = \emptyset$, then
$\models \{P * R\}\ C\ \{Q * R\}$.

Summary

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

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

In the next lecture, we will look at a proof system for separation logic, and apply separation logic to examples.

Papers of historical interest:

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

For reference: failure of expressions

We can also allow failure in expressions:

$$\mathcal{E}[\![-]\!](=) : \mathit{Exp} \times \mathit{Store} \to \{\lightning\} + \mathbb{Z}$$

$$\mathcal{E}[\![E_1 + E_2]\!](s) \stackrel{def}{=} \begin{cases} \text{if } \exists N_1, N_2. \begin{array}{l} \mathcal{E}[\![E_1]\!](s) = N_1 \wedge \\ \mathcal{E}[\![E_2]\!](s) = N_2 \end{array} & , \quad N_1 + N_2 \\ \text{otherwise,} & \quad \lightning \end{cases}$$

$$\mathcal{E}[\![E_1 / E_2]\!](s) \stackrel{def}{=} \begin{cases} \text{if } \exists N_1, N_2. \begin{array}{l} \mathcal{E}[\![E_1]\!](s) = N_1 \wedge \\ \mathcal{E}[\![E_2]\!](s) = N_2 \wedge \\ N_2 \neq 0 \end{array} & , \quad N_1 / N_2 \\ \text{otherwise,} & \quad \lightning \end{cases}$$

$$\vdots$$

$$\mathcal{B}[\![-]\!] : \mathit{BExp} \times \mathit{Store} \to \{\lightning\} + \mathbb{B}$$

$$\vdots$$

For reference: handling failures of expressions

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

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

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

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

$$\frac{\mathcal{B}[\![B]\!](s) = \lightning}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, (s,h)\rangle \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \lightning}{\langle \textbf{while } B \textbf{ do } C, (s,h)\rangle \Downarrow \lightning}$$

$$\frac{\mathcal{E}[\![E]\!](s) = \lightning}{\langle \textbf{dispose}(E), (s,h)\rangle \Downarrow \lightning}$$

For reference: semantics with failure of expressions

The definitions we give work without modifications, because implicitly, by writing N and ℓ, we assume $N \neq \lightning$ and $\ell \neq \lightning$.

However, the separation logic rules have to be modified to prevent faulting of expressions (see next lecture).