Hoare logic

Lecture 4: Introduction to separation logic

Christopher Pulte cp526
University of Cambridge

CST Part II – 2021/2022

Recap

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 look at the challenges Hoare logic faces when trying to reason about this language.

This will motivate introducing 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:

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

The heap

Commands are now evaluated also with respect to a **heap** that stores the current values of allocated locations.
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, the heap is a **finite** function, that is, a partial function with a finite domain:

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

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

Failure

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.

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

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

Why failure?

Instead of modelling failure explicitly, we could just leave the configuration stuck, but explicit failure makes things clearer and easier to state.

The following holds in WHILE_p:

$$\forall C, s, h. \left(\begin{array}{l} \langle C, \langle s, h \rangle \rangle \to^* \lightning \vee \\ \langle C, \langle s, h \rangle \rangle \to^\omega \vee \\ \exists h', s'. \langle C, \langle s, h \rangle \rangle \to^* \langle \mathbf{skip}, \langle s', h' \rangle \rangle \end{array} \right)$$

Adapting the base constructs to handle the heap

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

$$\frac{\mathcal{E}[\![E]\!](s) = N}{\langle X := E, \langle s, h\rangle\rangle \to \langle \textbf{skip}, \langle s[X \mapsto N], h\rangle\rangle}$$

$$\frac{}{\langle \textbf{skip}; C_2, \langle s, h\rangle\rangle \to \langle C_2, \langle s, h\rangle\rangle} \qquad \frac{\langle C_1, \langle s, h\rangle\rangle \to \langle C_1', \langle s', h'\rangle\rangle}{\langle C_1; C_2, \langle s, h\rangle\rangle \to \langle C_1'; C_2, \langle s', h'\rangle\rangle}$$

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

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

$$\frac{\langle C_1, \langle s, h\rangle\rangle \to \lightning}{\langle C_1; C_2, \langle s, h\rangle\rangle \to \lightning}$$

Heap dereferencing

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

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

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

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

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

Heap assignment

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

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

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

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

Deallocation

Deallocating an allocated location removes that location from the heap:

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

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

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

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:

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

Because the heap has a finite domain, it is always possible to pick a suitable ℓ, so allocation never faults. A real machine would run out of memory at some point.

Because of allocation, WHILE_p is not deterministic.

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:
  - $X := \textbf{alloc}(37, 42); Y := [X + 1]$ ends with $Y = 42$
  - $X := \textbf{alloc}(0); \textbf{if } X = 3 \textbf{ then } [3] := 1 \textbf{ else } [X] := 2$ is safe

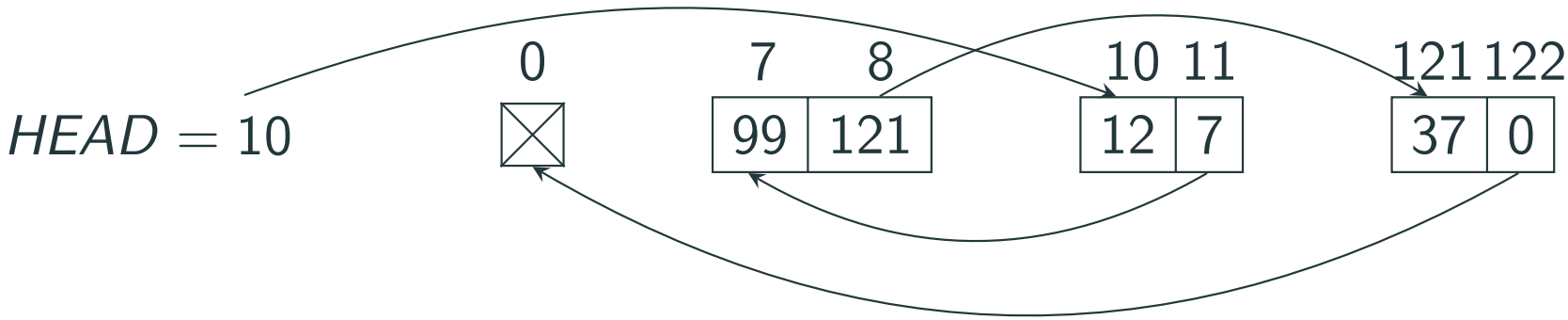
We do not have a separate type of pointers: we use integers as pointers.

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

Pointers and data structures

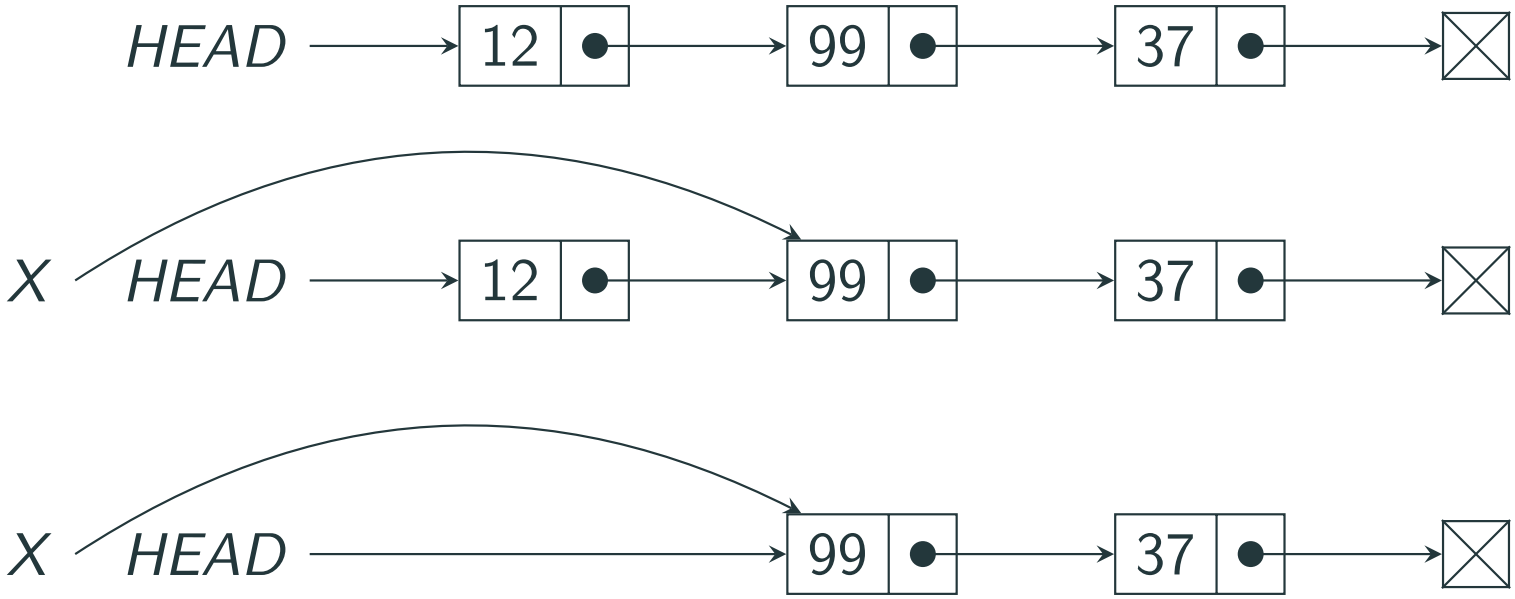
In WHILE_p, we can encode data structures in the heap. For example, we can encode the mathematical list $[12, 99, 37]$ with the following singly-linked list:

More concretely:



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).

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*

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 (in the absence of 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:

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

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

For reference: free variables

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

$$FV(-) : \mathit{Term} \to \mathcal{P}(\mathit{Var})$$

$$FV(\chi) \stackrel{def}{=} \{\chi\}$$

$$FV(f(t_1, \ldots, t_n)) \stackrel{def}{=} FV(t_1) \cup \ldots \cup FV(t_n)$$

and

$$FV(-) : \mathit{Assertion} \to \mathcal{P}(\mathit{Var})$$

$$FV(\top) = FV(\bot) \stackrel{def}{=} \emptyset$$

$$FV(P \wedge Q) = FV(P \vee Q) = FV(P \Rightarrow Q) \stackrel{def}{=} FV(P) \cup FV(Q)$$

$$FV(\forall x.\, P) = FV(\exists x.\, P) \stackrel{def}{=} FV(P) \setminus \{x\}$$

$$FV(t_1 = t_2) \stackrel{def}{=} FV(t_1) \cup FV(t_2)$$

$$FV(p(t_1, \ldots, t_n)) \stackrel{def}{=} FV(t_1) \cup \ldots FV(t_n)$$

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:

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

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 rule:

$$\frac{}{\vdash \{\exists v.\, E_1 \hookrightarrow v\}\ [E_1] := E_2\ \{E_1 \hookrightarrow E_2\}}$$

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

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

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:

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

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 enables **modular** reasoning about **resources**.
It introduces new connectives to reason about the combination of **disjoint** resources.

We will use separation logic to reason about pointers in WHILE_p.
Our resources will be parts of the heap, and we will use the new connectives of separation logic to control aliasing.

Where a Hoare logic assertion refers to a (freely duplicable) property of the current state, a separation logic assertion asserts **ownership** of resources. Resources can be combined or compared (and exchanged), but need to be accounted for.

History and terminology

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.

There are many variants of separation logic.

In WHILE_p, the heap is explicitly managed: the program is meant to dispose of heap locations itself. To be able to show that our programs do not leak memory, we are going to consider a so-called linear (or classical) separation logic. If we were not interested in reasoning about deallocation, for example because there is a garbage collector, we could use an affine (or intuitionistic) separation logic.

The points-to assertion

We introduce a new assertion, written $t_1 \mapsto t_2$, and read "t_1 points to t_2", to reason 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 extends Hoare logic with a new connective, the separating conjunction '$*$', to reason about disjoint resources.

The assertion $P * Q$ asserts that P and Q hold (somewhat like $P \wedge Q$); however, it also asserts that the resources (the parts of the heap) owned by P and Q are **disjoint**.

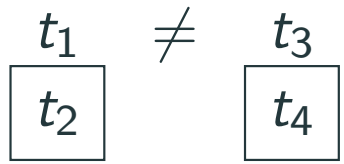
The separating conjunction has a neutral element, *emp*, which describes the empty resource (the empty heap):
$emp * P \Leftrightarrow P \Leftrightarrow P * emp$.

Examples of separation logic assertions

1. $(t_1 \mapsto t_2) * (t_3 \mapsto t_4)$

   This assertion is unsatisfiable in a state where $t_1 = t_3$, since $t_1 \mapsto t_2$ and $t_3 \mapsto t_4$ would both assert ownership of the same location.

   A heap satisfying this assertion is of the following shape:



Examples of separation logic assertions

2. For example,

$$((X \mapsto 101) * (Y \mapsto 102)) \wedge X = 7 \wedge Y = 41$$

is satisfied by the following heap:

Examples of separation logic assertions

3. $(t_1 \mapsto t_2) * (t_1 \mapsto t_3)$

   This assertion is not satisfiable, as t_1 is not disjoint from itself.

4. $t_1 \mapsto t_2 \wedge t_3 \mapsto t_4$

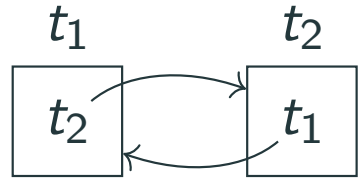
   This asserts that the heap is described by $t_1 \mapsto t_2$, and also by $t_3 \mapsto t_4$.
   Therefore, $t_1 = t_3$, and so $t_2 = t_4$

Examples of separation logic assertions

5. A heap satisfying

$$(t_1 \mapsto t_2) * (t_2 \mapsto t_1)$$

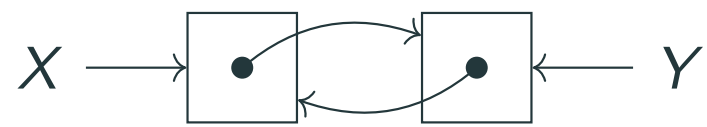
is of the following shape:



6. For instance, a heap satisfying

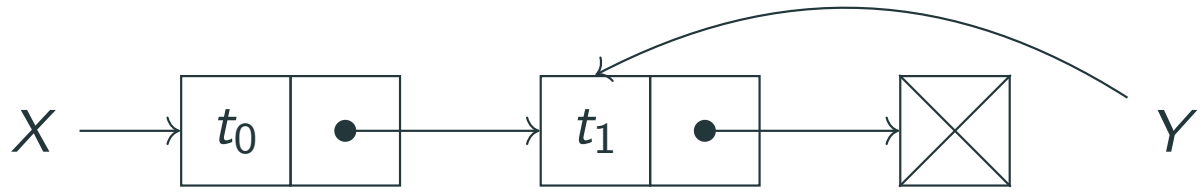
$$(X \mapsto Y) * (Y \mapsto X)$$

is of the following shape:



Examples of separation logic assertions

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

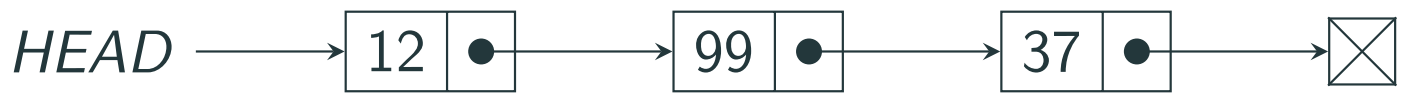


Here, $X \mapsto t_0, ..., t_n$ is shorthand for

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

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

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 stack 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 (with the extra indirection):

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

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:

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

$t_1 \mapsto t_2$ asserts ownership of location ℓ, so to capture ownership, requires $\{\ell\} \subseteq \mathit{dom}(h)$. Moreover, to prevent memory leaks, we require $\mathit{dom}(h) = \{\ell\}$.

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 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:

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

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

$$\not\models \{100 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1\}$$

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 state satisfies $37 \mapsto 1 * 37 \mapsto 1$:

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

Informal semantics of separation logic triples

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

- if h_1 satisfies P, when C is executed from an initial state with an initial heap $h_1 \uplus h_F$, then
  - C does not fault, and
  - if C terminates, then the terminal heap has the form $h_1' \uplus h_F$, where h_1' satisfies Q.

The first condition ensures that the precondition asserts ownership of all the locations that might be accessed.
The second condition 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:

$$\models \{P\}\ C\ \{Q\} \stackrel{def}{=}$$
$$\forall s, h_1, h_F.\, dom(h_1) \cap dom(h_F) = \emptyset \wedge h_1 \in [\![P]\!](s) \Rightarrow$$
$$\left(\begin{array}{l} (\neg(\langle C, \langle s, h_1 \uplus h_F \rangle\rangle \rightarrow^* \lightning)) \wedge \\ \left(\forall s', h'.\ \begin{array}{r} \langle C, \langle s, h_1 \uplus h_F \rangle\rangle \rightarrow^* \langle \mathbf{skip}, \langle s', h' \rangle\rangle \Rightarrow \\ \exists h'_1.\, h' = h'_1 \uplus h_F \wedge h'_1 \in [\![Q]\!](s') \end{array} \right) \end{array} \right)$$

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 that enables modular reasoning about resources. It extends Hoare logic with new assertions, and refines the semantics of assertions to reason about ownership and separation.

We leverage this to control aliasing, which enables practical reasoning about pointers and 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.