Separation Logic and Compositional Symbolic Execution: Verification and Bug Detection at Scale

Philippa Gardner

Lecture 1 - *June 24, 2026*

This is the notes for the first lecture of the OPLSS 2026 course on Separation Logic and Compositional Symbolic Execution, taught by Philippa Gardner.

1 Hoare Logic and Its Scalability Problem

1.1 Overview of Hoare Logic

Hoare logic is a reasoning system for proving the specifications of programs. It uses Hoare triples, written $\{P\}C\{Q\}$, to describe program specifications, where C is the program, and P and Q are assertions that respectively describe pre- and post-conditions. A Hoare triple is true if, whenever the precondition P holds before executing the program C, the postcondition Q will hold after executing C. The Hoare logic provides rules for how to derive valid Hoare triples based on the structure of the program.

For example, the Hoare triple $\{x = 0\}\ x := x + 1\ \{x = 1\}$ says that if x is 0 before executing the assignment $x := x + 1$, then x is 1 afterward.

1.2 Separation Matters in Program Verification

For programs that manipulate heap-allocated data structures (like linked lists), awareness of difference heap locations (pointers) is crucial for reasoning about program behavior. For example, if we want to reason about a list deallocation program,

Compiled By:
Dinghong Zhong, Samuel Dodson, Daniel Andres Pinto Alvarado

```
LDispose(x) {
  while (x != null) {
    y := x;
    x := [x + 1];
    dispose(y);
    dispose(y + 1)
  };
  return null
}
```

Figure 1.1: List Dispose

the Hoare triple we can prove is:

$$\vdash \{\mathsf{List}(x)\}\mathsf{LDispose}(x)\{x = \mathsf{null}\}$$

which means if x points to a valid list before executing LDispose, then after executing LDispose, x will be null.

However, Hoare logic has scalability issues in a sense that it does not provide a clear and simple way to extend already proven specifications with richer assertions.

For example, suppose we have proved the Hoare triple $\vdash \{\mathsf{List}(x)\}\mathsf{LDispose}(x)\{x = \mathsf{null}\}$. How can we apply it to conclude the postcondition after executing LDispose in a program state that contains another $\mathsf{List}(y)$?

With a simple glance, the precondition is $\mathsf{List}(x) \wedge \mathsf{List}(y)$, and the postcondition should be $x = \mathsf{null} \wedge \mathsf{List}(y)$. But this is not necessarily true, because x and y could alias! We can conclude that $x = \mathsf{null}$, but we cannot say anything about y with certainty.

To make the specification describe the program correctly and precisely, we need to use a more complex assertion that describes the separation of the two lists in the precondition:

$$(\mathsf{List}(x) \wedge \mathsf{List}(y)) \wedge (\forall u.\ \mathsf{reach}(x, u) \wedge \mathsf{reach}(y, u) \Rightarrow u = \mathsf{null})$$

The $\forall u.\ \mathsf{reach}(x, u) \wedge \mathsf{reach}(y, u) \Rightarrow u = \mathsf{null}$ part of the assertion says that the two lists x and y are separated, i.e., they do not share any common nodes.

Only then can we conclude the postcondition $x = \mathsf{null} \wedge \mathsf{List}(y)$.

If we have three lists x, y and z, our precondition grows:

$$
\begin{array}{l}
(\mathsf{List}(x,\alpha) \wedge \mathsf{List}(y,\beta)) \\
\wedge\ (\forall u.\ \mathsf{reach}(x,u) \wedge \mathsf{reach}(y,u) \Rightarrow u = \mathsf{null}) \\
\wedge\ \mathsf{List}(z) \\
\wedge\ (\forall u.\ \mathsf{reach}(z,u) \wedge (\mathsf{reach}(x,u) \vee \mathsf{reach}(y,u)) \Rightarrow u = \mathsf{null})\,,
\end{array}
$$

as does the postcondition:

$$x = \mathsf{null} \wedge \mathsf{List}(y) \wedge \mathsf{List}(z) \wedge (\forall u.\ \mathsf{reach}(z,u) \wedge \mathsf{reach}(y,u) \Rightarrow u = \mathsf{null})\,.$$

This approach does not scale; not only is it unclear where to add the separation assertion, but when we add more separation assertions, the complexity of the condition grows quadratically (as does the time needed to solve them).

2 Separation Logic

The scalability problem of Hoare logic is caused by the fact that the formulae in the Hoare triples always describe the *entire* program heap. Hence, when we want to extend the specification to describe more states, we must write more complex formulae that assert the separation of added states with the existing states. Separation logic is a an approach that addresses this problem directly by introducing a new logical connective called the *separating conjunction*, written $\star$. Instead of describing the entire heap, a formula in separation logic describes only a *part* of the heap, and uses $\star$ to combine two formulae that describe two strictly separate parts of the heap. With this separating connective, we can extend the specification in a compositional way, without needing to write extra formulae that assert the separations.

Note. The separating conjunction operator bears superficial resemblance to the disjoint union; however, where the disjoint union re-indexes members of sets to reconcile intersection, the separating conjunction operator enforces that the two operands are disjoint.

Case Study: The While Language

We'll use a small imperative language to show how separation logic allows for meaningful a meaningful specification that talks about how the heap resource is managed.

$$\begin{array}{rl}
\textbf{Boolean Values} & b \in \mathsf{Bool} \stackrel{def}{=} \{\mathsf{True}, \mathsf{False}\} \\
\textbf{Values} & v \in \mathsf{Val} \stackrel{def}{=} \mathbb{N} \cup \mathsf{Bool} \cup \{\mathsf{null}\} \\
\textbf{Program Variables} & x \in \mathsf{Var} \\
\textbf{Simple Expressions} & E \in \mathsf{Exp} ::= v \mid x \mid E + E \mid E - E \mid E \times E \mid E/E \ldots
\end{array}$$

Figure 2.1: Expressions of the While Language

The expression language defined in Figure 2.1 is left underspecified so that it can later be extended. While Figure 2 uses $[E]$ to denote a value stored at the heap cell pointed by the value of the expression E, and $\vec{E}$ to denote the arguments of a function.

$$\begin{array}{lrl}
\mathbf{C} ::= & \mathbf{x} := E & \text{(assignment)} \\
\mid & \mathbf{x} := [E] & \text{(lookup)} \\
\mid & [E] := E & \text{(mutation)} \\
\mid & \mathbf{x} := \text{new}(E) & \text{(allocation)} \\
\mid & \text{dispose}(E) & \text{(deallocation)} \\
\mid & \text{skip} & \text{(skip)} \\
\mid & C; C & \text{(sequential composition)} \\
\mid & \text{if } (E)\ C \text{ else } C & \text{(if)} \\
\mid & \text{while } (E)\ C & \text{(while)} \\
\mid & y := f(\vec{E}) & \text{(function call)}
\end{array}$$

Figure 2.2: Commands of the While Language

Program State Notation

Figure 2.3 brings into scope the Variable Store and the Linear Heap. Finally, we introduce the **notations**:

- $\emptyset \in$ Store for the empty store.
- $x[x \mapsto v]$ for the Store s with x updated by v.

- $\emptyset \in \text{Heap}$ for the empty heap.
- $h1 \uplus h2$ for the union of the disjoint heaps h_1 and h_2.
- $\mathcal{E}[\![E]\!]_s \in \mathsf{Val}$ for the value of the expression E with respect to the variable store s.
- $\gamma(f) = (\vec{x}, C, E)$ for the **Function Definition Context** which maps function identifiers to their implementations.
- $f(\vec{x})\{C; return\ E\} \in \gamma$ for $\gamma(f) = (\vec{x}, C, E)$
- $(s,h), C \downarrow_\gamma (s',h')$ for the usual big step operational semantics which states that under the store s and the heap h, if C is executed, does not abort, and terminates, it produces a new state (s',h').
- $[E]$ for the contents of the storage at the address given by the expression E.

$$\begin{aligned} \textbf{Variable Store} \quad & s \in \text{Store} \stackrel{def}{=} \mathsf{Var} \rightharpoonup_{fin} \mathsf{Val} \\ \textbf{Linear Heap} \quad & h \in \text{Heap} \stackrel{def}{=} \mathbb{N} \rightharpoonup_{fin} \mathsf{Val} \\ \textbf{State} \quad & \sigma \in \text{State} \stackrel{def}{=} \text{Store} \times \text{Heap} \end{aligned}$$

Figure 2.3: Program State

Syntax of Separation Logic Assertions

Assertions. The set of *assertions*, Assert, ranged over by $P, P_1, Q, Q_1, \ldots$, is defined by:

$$\begin{array}{rcll} P \in \mathsf{Assert} & ::= & P \wedge P \mid P \Rightarrow P \mid \mathsf{True} \mid \mathsf{False} & \text{classical connectives} \\ & \mid & E = E \mid E > E \mid E \in X \mid \cdots & \text{boolean assertions} \\ & \mid & \exists x.\ P & \text{quantification} \\ & \mid & P \star P \mid P \mathrel{-\!\!*} P \mid \mathsf{emp} \mid \circledast_{E_1 \le x < E_2} P & \text{separating connectives} \\ & \mid & E \mapsto E & \text{cell assertion} \end{array}$$

2.1 Syntax of Separation Logic Assertions

Assertions. The set of *assertions*, Assert, ranged over by $P, P_1, Q, Q_1, \ldots$, is defined by:

$$
\begin{array}{rll}
P \in \mathsf{Assert} ::= & P \wedge P \mid P \Rightarrow P \mid \mathsf{True} \mid \mathsf{False} & \text{classical connectives} \\
& \mid E = E \mid E > E \mid E \in X \mid \cdots & \text{boolean assertions} \\
& \mid \exists x.\ P & \text{quantification} \\
& \mid P \star P \mid P \mathrel{-\!\!*} P \mid \mathsf{emp} \mid \circledast_{E_1 \leq x < E_2} P & \text{separating connectives} \\
& \mid E \mapsto E & \text{cell assertion}
\end{array}
$$

The syntax includes the classical logical connectives and quantifiers; the newly added separating connectives are highlighted.

Note. Many conventional operators are excluded here because they're derivable from the connectives given here; for example, $\neg P = P \implies \mathsf{False}$, with negation can use DeMorgan's to get $\vee$, and $\forall x.P = \neg\exists x.\neg P$.

2.2 Meaning of Separation Logic Assertions

What kind of program states does an assertion describe? The meaning of the main assertions is given by the satisfaction relation $e, s, h \models P$, which says that the assertion P is satisfied in the program state (e, s, h), where e is the environment of logical variables, s is the environment of program variables, and h is the heap.

Note. One can think of $P_1 \star P_2$ as separation, and $P_1 \wedge P_2$ as overlap. So, for example, $10 \mapsto 1 \star 10 \mapsto 2$ is unsatisfiable, as separate heaps cannot have any intersection in their domains—conversely, $10 \mapsto 1 \wedge 11 \mapsto 1$ is unsatisfiable because the two heaps do not have exactly identical domains. Note that $\wedge$ might superficially seem equivalent to $=$, but the distinction is best revealed by the idiom $P \star \mathsf{True}$.

For example, $(10 \mapsto 1 \star \mathsf{True}) \wedge (11 \mapsto 2 \star \mathsf{True})$ is satisfiable by heap $\{10 \mapsto 1, 11 \mapsto 2\}$, because True represents any heap (analagously to how $10 \mapsto 1 \wedge 10 \mapsto \_$ is satisfiable by heap $\{10 \mapsto 1\}$, because $\_$ is any value). Roughly, instead of "are these things equal," it's "can these things be made equal."

2.3 Separation Logic Rules

So far fig. 2.4 defines the programs that satisfy a separation logic Hoare triple. Now we define how to derive such triples from program syntax in fig. 2.5.

The soundness condition defined in fig. 2.6 guarantees that the derivation rules are correct (the proof is omitted here).

$$
\begin{aligned}
e,s,h \models P_1 \wedge P_2 &\iff e,s,h \models P_1 \wedge e,s,h \models P_2 \\
e,s,h \models P_1 \Rightarrow P_2 &\iff e,s,h \models P_1 \Rightarrow e,s,h \models P_2 \\
e,s,h \models \mathsf{True} &\iff \text{always} \\
e,s,h \models \mathsf{False} &\iff \text{never} \\
e,s,h \models E_1 = E_2 &\iff (\mathcal{E}[\![E_1 = E_2]\!]_{e,s} = \mathsf{True}) \wedge h = \emptyset \\
e,s,h \models E_1 > E_2 &\iff (\mathcal{E}[\![E_1 > E_2]\!]_{e,s} = \mathsf{True}) \wedge h = \emptyset \\
e,s,h \models E \in X &\iff (\mathcal{E}[\![E \in X]\!]_{e,s} = \mathsf{True}) \wedge h = \emptyset \\
e,s,h \models \exists x.\ P &\iff \exists v \in \mathsf{Val}.\ e[x \mapsto v], s, h \models P \\
e,s,h \models P_1 \star P_2 &\iff \exists h_1, h_2 \in \mathsf{Heap}.\ h = h_1 \uplus h_2 \wedge \\
&\qquad e,s,h_1 \models P_1 \wedge e,s,h_2 \models P_2 \\
e,s,h \models \mathsf{emp} &\iff h = \emptyset \\
e,s,h \models \circledast_{E_1 \le x < E_2} P &\iff \mathcal{E}[\![E_1]\!]_{e,s} = i \in \mathbb{N} \wedge \mathcal{E}[\![E_2]\!]_{e,s} = k \in \mathbb{N} \wedge \\
&\qquad \big(i < k \Rightarrow \exists h_i, \ldots, h_{k-1}.\ h = h_i \uplus \cdots \uplus h_{k-1} \wedge \\
&\qquad\qquad \forall j.\ i \le j < k.\ e,s,h_j \models P[j/x]\big) \wedge \\
&\qquad \big(i \ge k \Rightarrow h = \emptyset\big) \\
e,s,h \models E_1 \mapsto E_2 &\iff h = \{\mathcal{E}[\![E_1]\!]_{e,s} \mapsto \mathcal{E}[\![E_2]\!]_{e,s}\} \\
e,s,h \models \mathsf{pred}(\vec{E}) &\iff \cdots
\end{aligned}
$$

Figure 2.4: Satisfaction Relation for Separation Logic Assertions

$$\frac{x \notin \mathsf{pv}(E')}{\vdash \{x = E' \star E \in \mathsf{Val}\}\ x := E\ \{x = E[E'/x]\}}\ \text{ASSIGN}$$

$$\frac{x \notin \mathsf{pv}(E')}{\vdash \{x = E' \star E \mapsto E_1\}\ x := [E]\ \{x = E_1[E'/x] \star E[E'/x] \mapsto E_1[E'/x]\}}\ \text{LOOKUP}$$

$$\frac{}{\vdash \{E_1 \mapsto E \star E_2 \in \mathsf{Val}\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}}\ \text{MUTATE}$$

$$\frac{x \notin \mathsf{pv}(E')}{\vdash \{x = E' \star E \in \mathbb{N}\}\ x := \mathsf{new}(E)\ \{\exists y.\ x = y \star \circledast_{0 \le i < E[E'/x]} ((y + i) \mapsto \mathsf{null})\}}\ \text{NEW}$$

$$\frac{}{\vdash \{E \mapsto E_1\}\ \mathsf{dispose}(E)\ \{\mathsf{emp}\}}\ \text{DISPOSE} \qquad \frac{}{\vdash \{\mathsf{emp}\}\ \mathsf{skip}\ \{\mathsf{emp}\}}\ \text{SKIP}$$

$$\frac{\vdash \{P\}\ C_1\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1;\ C_2\ \{R\}}\ \text{SEQ} \qquad \frac{\vdash \{P \wedge E\}\ C_1\ \{Q_1\} \qquad \vdash \{P \wedge \neg E\}\ C_2\ \{Q_2\}}{\vdash \{P \star E \in \mathsf{Bool}\}\ \mathsf{if}\ (E)\ C_1\ \mathsf{else}\ C_2\ \{Q_1 \vee Q_2\}}\ \text{IF}$$

$$\frac{\vdash \{P \wedge E\}\ C\ \{P \star E \in \mathsf{Bool}\}}{\vdash \{P \star E \in \mathsf{Bool}\}\ \mathsf{while}\ (E)\ C\ \{P \wedge \neg E\}}\ \text{WHILE}$$

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \mathsf{mod}(C) \cap \mathsf{pv}(R) = \emptyset}{\vdash \{P \star R\}\ C\ \{Q \star R\}}\ \text{FRAME}$$

$$\frac{\models P \Rightarrow P' \qquad \vdash \{P'\}\ C\ \{Q'\} \qquad \models Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}\ \text{CONS} \qquad \frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{\exists x.\ P\}\ C\ \{\exists x.\ Q\}}\ \text{EXISTS}$$

$$\frac{\vdash \{P_1\}\ C\ \{Q\} \qquad \vdash \{P_2\}\ C\ \{Q\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q\}}\ \text{DISJ}$$

Figure 2.5: Separation Logic Rules

$$\forall P, C, Q, e, s, h. \vdash \{P\}\ C\ \{Q\} \Longrightarrow e, s, h \models P \\ \Longrightarrow \big(\neg\big((s,h), C \Downarrow \lightning\big) \wedge \big(\forall s', h'.\ (s,h), C \Downarrow (s',h') \Longrightarrow e, s', h' \models Q\big)\big)$$

Figure 2.6: Soundness of Separation Logic

It says if we can infer Hoare triple $\vdash \{P\}\ C\ \{Q\}$ from the program C, then whenever we run the C with a state s, h that satisfies the precodition P, after executing the command C the outcome states s', h' satisfies the postcondition Q.

2.4 Example: Proof sketch for LLen to compute the length of a list

$\vdash \quad \{\mathrm{x} = x \star \mathsf{List}(x, \alpha)\}$
$\mathsf{LLen}(x)$ {
  // Initialise the local variables and ensure the Boolean condition is evaluable.
  $\{\mathrm{x} = x \star \mathsf{List}(x, \alpha) \star \mathrm{n}, \mathrm{t} = \mathsf{null}\}$
  $\{\mathrm{x} = x \star \mathsf{List}(x, \alpha) \star \mathrm{n}, \mathrm{t} = \mathsf{null} \star (\mathrm{x} = \mathsf{null}) \in \mathsf{Bool}\}$
  if $(x = \mathsf{null})$ {
    $\{\mathrm{x} = x \star \mathsf{List}(x, \alpha) \star \mathrm{n}, \mathrm{t}, \mathrm{x} = \mathsf{null}\}$
    // Since x = null, unfold the list predicate to get the base case.
    $\{\mathrm{x} = x \star (\mathrm{x} = \mathsf{null} \star \alpha = [\,]) \star \mathrm{n}, \mathrm{t} = \mathsf{null}\}$
    $n := 0$
    $\{x = x \star (x = \mathsf{null} \star \alpha = [\,]) \star t = \mathsf{null} \star n = 0\}$
    // Forget x and t, connect n to alpha, and fold the list predicate.
    $\{(x = \mathsf{null} \star \alpha = [\,]) \star n = |\alpha|\}$
    $\{\mathsf{List}(x, \alpha) \star n = |\alpha|\}$
  } else {
    $\{\mathrm{x} = x \star \mathsf{List}(x, \alpha) \star n, t = \mathsf{null} \star x \neq \mathsf{null}\}$
    // Unfold the list predicate to the recursive case and identify the resource needed.
    $\{\exists v, \beta, y.\ x \mapsto v, y \star \alpha = v{:}\beta \star \mathsf{List}(y, \beta) \star n, t = \mathsf{null}\}$
    // Use frame, consequence, and the recursive-call rule.
    $\{x \mapsto v, y \star \mathsf{List}(y, \beta) \star n, t = \mathsf{null}\}$
    $t := [x + 1]$
    $\{x \mapsto v, y \star \mathsf{List}(y, \beta) \star n = \mathsf{null} \star t = y\}$
    $n := \mathsf{LLen}(t)$
    $\{x \mapsto v, y \star \mathsf{List}(y, \beta) \star n = |\beta|\}$
    $n := n + 1$
    $\{x \mapsto v, y \star \mathsf{List}(y, \beta) \star n = |\beta| + 1\}$
    // Frame back assertions, add existentials, and fold back the list predicate.
    $\{\exists v, \beta, y.\ x \mapsto v, y \star \alpha = v{:}\beta \star \mathsf{List}(y, \beta) \star n = |\beta| + 1\}$
    $\{\mathsf{List}(x, \alpha) \star n = |\alpha|\}$
  }
  // As the post-condition of both the if and else bodies are the same,
  // we infer the post-condition of the conditional statement.
  $\{\mathsf{List}(x, \alpha) \star n = |\alpha|\}$
  return n
}
$\{\mathsf{List}(x, \alpha) \star ret = |\alpha|\}$