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

Philippa Gardner

Lecture 2 - *June 25, 2026*

These are the notes for the second lecture of the OPLSS 2026 course on Separation Logic and Compositional Symbolic Execution, taught by Philippa Gardner. The official lecture slides are available at `https://gillianplatform.github.io/docs/oplss26/Lecture2.pdf`.

1 Describing Structures Beyond Classic Linked Lists

The lecture gives the following add-node example for Collections-C singly linked lists:

$$\{x = x \star v = v \star \mathsf{CClist}(x, \alpha)\}\ \mathsf{CCllist\ addnode}(x, v)\ \{\mathsf{CClist}(ret, \alpha \cdot [v])\}$$

```
CCllist addnode(x, v) {
   n := new(2);
   [n] := v;
   l := [x];
   if (l = 0) {
      [x] := l + 1; [x + 1] := n; [x + 2] := n
   } else {
      t := [x + 2];
      [t + 1] := n; [x + 2] := n; [x] := l + 1
   };
   return x
}
```

The postcondition states that the returned Collections-C list contains the original sequence α extended by the singleton list $[v]$.

Separation logic also allows us to describe complex structures, such as N-ary trees:

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

$$
\begin{array}{rcl}
tree(x,c) & \stackrel{def}{=} & (x \mapsto 0, \_ \star c = 1) \\
 & & \vee \left(\exists\, n > 1, \vec{p}, \vec{c}. x \mapsto n \star c = \sum_{i=1}^{n} c_i \star P(n,c) \right) \\
P(n,c) & = & \circledast_{1 \le i \le n} (x + i \mapsto p_i \star tree(p_i, c_i) \wedge 1 \le c_i < c)
\end{array}
$$

The predicate $tree(x, c)$ describes a tree x with c total cells used by either constructing a leaf with value 0 (and an empty box for uniformity), or by establishing what the head x is pointing to, and ensuring we have enough space for each of the children (c_i) and describing them recursively.

An example of an N-ary Tree described by the specification is the one portrayed in Figure 1.1

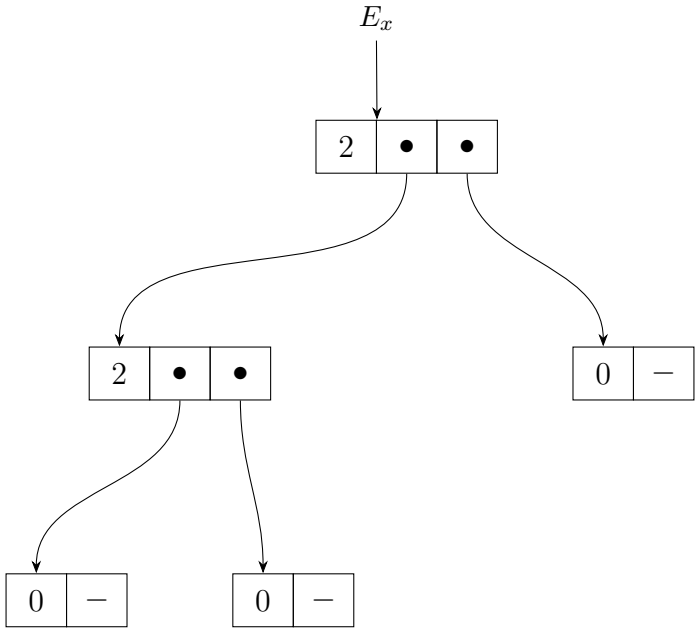

Figure 1.1: N-ary Tree

Following the same schema, we could also define Hash Tables, Doubly Linked Lists, Collections-C linked lists, ...

2 Formalizing Proof Sketches

In the previous lecture we saw how we could describe informally the behaviour of a program, such as `LLen` (which computed the length of a linked list recursively) via Hoare triples with preconditions and postconditions encoded in separation logic. But how can we reason about the program in a formal way?

For that, we'll use the proof rules defined in Figure 2 (**frame**, **cons**equence, **exist**ential elimination, and **seq**uencing) to do proof derivations.

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \mathsf{mod}(C) \cap fv(R) = \emptyset}{\vdash \{P \star R\}\ C\ \{Q \star R\}}\ \text{frame} \qquad \frac{\models P \Rightarrow P' \qquad \vdash \{P'\}\ C\ \{Q'\} \qquad \models Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}\ \text{cons}$$

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{\exists x.P\}\ C\ \{\exists x.Q\}}\ \text{exists} \qquad \frac{\vdash \{P\}\ C_1\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}\ \text{seq}$$

Figure 2.1: Proof rules for reasoning about Hoare triples

Notation: $\mathsf{mod} : C \to \mathsf{Var}$ is a function which maps commands to the (program) variables which are (possibly) modified, while fv returns the set of free variables (program or logical) from an assertion expression. In particular, x is modified by assignments such as $x := E$, $x := [E]$, $x := new(E)$, and $x := f(\vec{E})$, but not by mutation commands such as $[E] := E$. Recall that the proof rules are sound: $\vdash \{P\}C\{Q\} \Rightarrow \models \{P\}C\{Q\}$.

Let's consider the example of the previous lecture, in particular:

0 } else {
1 $\{\text{x} = x \star list(x, \alpha) \star n, t = null \star x \neq null\}$
2 // unfold list predicate to recursive case, identify the resource needed
3 $\{\exists v, \beta, y. x \mapsto v, y \star \alpha = v : \beta \star list(y, \beta) \star n, t = null\}$
4 exists,frame+cons
5 $\{x \mapsto v, y \star list(y, \beta) \star n, t = null\}$
6 t := [x + 1];
7 // Use the fcall rule to consume precondition and produce postcondition
8 $\{x \mapsto v, y \star list(y, \beta) \star n = null \star t = y\}$
9 n := llen(t);
10 $\{x \mapsto v, y \star list(y, \beta) \star n = |\beta|\}$
11 n := n + 1;
12 $\{x \mapsto v, y \star list(y, \beta) \star n = |\beta| + 1\}$
13 // frame back assertions, add exists and fold back list predicate
14 $\{\exists v, \beta, y. x \mapsto v, y \star \alpha = v : \beta \star list(y, \beta) \star n = |\beta| + 1\}$
15 $\{list(x, \alpha) \star n = |\alpha|\}$
16 }

Let's use proof rules to verify the else body after unfolding the list predicate.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma \vdash \{P\}\ t := [x+1]\ \{S\}}\ \mathbf{1} \quad \dfrac{\cdots}{\Gamma \vdash \{S\}\ n := llen(t)\ \{R\}}\ \mathbf{2}}{\Gamma \vdash \{P\}\ t := [x+1]; n := llen(t)\ \{R\}}\ \text{seq} \quad \dfrac{\cdots}{\Gamma \vdash \{R\}\ n := n+1\ \{Q\}}}{\Gamma \vdash \{P\}\ t := [x+1]; n := llen(t); n := n+1\ \{Q\}}\ \text{seq}}{\Gamma \vdash \{\exists v, \beta, y.P\}\ t := [x+1]; n := llen(t); n := n+1\ \{\exists v, \beta, y.Q\}}\ \text{exists} \times 3$$

Figure 2.2: Proof for LLen inner block

If we allow P, S, R, Q to stand for:

- $P = x \mapsto v, y \star \alpha = v : \beta \star list(y, \beta) \star n, t = null$
- $S = x \mapsto v, y \star \alpha = v : \beta \star list(y, \beta) \star n = null \star t = y$
- $R = x \mapsto v, y \star \alpha = v : \beta \star list(y, \beta) \star n = |\beta|$
- $Q = x \mapsto v, y \star \alpha = v : \beta \star list(y, \beta) \star n = |\beta| + 1$

The two remaining subderivations used in Figure 2 are:

$$\dfrac{\dfrac{\dfrac{x \notin \mathsf{pv}(\mathsf{null})}{\Gamma \vdash \{t = \mathsf{null} \star x + 1 \mapsto y\}\ t := [x+1]\ \{t = y \star x + 1 \mapsto y\}}\ \text{lookup}}{\Gamma \vdash \{t = \mathsf{null} \star x + 1 \mapsto y \star F\}\ t := [x+1]\ \{t = y \star x + 1 \mapsto y \star F\}}\ \text{frame}}{\Gamma \vdash \{P\}\ t := [x+1]\ \{S\}}\ \text{cons}$$
$$F = x \mapsto v \star \alpha = v : \beta \star list(y, \beta) \star n = \mathsf{null}$$

Figure 2.3: LLen proof derivation, recursive case 1

$$\dfrac{\dfrac{\dfrac{\{x = x \star list(x, \alpha)\}\ \mathsf{LLen}(x)\ \{list(x, \alpha) \star ret = |\alpha|\} \in \Gamma}{\Gamma \vdash \{y \in \mathsf{Val} \star t = y \star list(y, \beta)\}\ n := llen(t)\ \{list(y, \beta) \star n = |\beta|\}}\ \text{fcall}}{\Gamma \vdash \{y \in \mathsf{Val} \star t = y \star list(y, \beta) \star F\}\ n := llen(t)\ \{list(y, \beta) \star n = |\beta| \star F\}}\ \text{frame}}{\Gamma \vdash \{S\}\ n := llen(t)\ \{R\}}\ \text{cons}$$
$$F = x \mapsto v, y \star \alpha = v : \beta$$

Figure 2.4: LLen proof derivation, recursive case 2

Then we can see most of the work is just either joining commands naturally via the **seq** rule, or unfolding definitions so we can meet the precondition of a function call, as can be seen by the S condition.

3 Compositional Symbolic Execution

The second half of the lecture explains how tools bridge the gap between proof sketches and automated reasoning. Compositional symbolic execution symbolically executes a partial symbolic state, uses *produce* to assume the resources described by a precondition, and uses *consume* to check that the resources required by a command or a postcondition are available. First-order path constraints are checked by SMT solvers such as Z3.

The lecture separates the theory and practice of compositional symbolic execution: the theory is tool-independent, parametric in the memory model, and supports both compositional verification and true bug detection, while Gillian[1] is the maintained platform implementation.

The symbolic state used in Gillian has four parts:

$$\hat{\sigma} = (\hat{s}, \hat{\mu}, \widehat{\mathsf{pred}}, \hat{\pi})$$

where $\hat{s} : \mathsf{PVar} \rightharpoonup_{\text{fin}} \mathsf{LExp}$ is the symbolic store, $\hat{\mu} : \mathsf{LExp} \rightharpoonup_{\text{fin}} \mathsf{LExp}$ is the symbolic linear heap, $\widehat{\mathsf{pred}}$ stores user-defined predicate instances, and $\hat{\pi}$ is the symbolic path condition.

[1] Demo video: https://www.youtube.com/watch?v=5TTBX4ecZkk.

For the recursive case of LLen, compositional symbolic execution follows the same logical structure as the proof sketch:

1. produce the initial symbolic state from the precondition $x = x \star list(x, \alpha)$;
2. split on $x = \mathsf{null}$;
3. in the non-null branch, unfold $list(x, \alpha)$ to obtain $x \mapsto v, y \star list(y, \beta) \star \alpha = v : \beta$;
4. execute the lookup $t := [x + 1]$ and update the symbolic store with $t = y$;
5. for $n := \mathsf{LLen}(t)$, consume the callee precondition for t and produce its postcondition;
6. execute $n := n + 1$, fold the list predicate back, and consume the final postcondition.