Separation Logic, Compositional Symbolic Execution and the Gillian Platform

Philippa Gardner
Imperial College London

Challenge: to develop techniques and tools for verification and true bug detection that scale.

Link: `https://gillianplatform.github.io/labs/standv26/`

Lecture 1

Separation Logic and the Gillian Platform

Scalable Software Verification

Some program-analysis techniques based on first-order logic:

- verification based on Hoare logic: e.g. Frama-C;
- symbolic execution: e.g. CBMC.

These techniques are not compositional with respect to the machine state, and **do not scale**.

Some program-analysis techniques, compositional with respect to the machine state:

- separation logics, originally O'Hearn and Reynolds: e.g. the concurrent higher-order Iris framework.
- compositional symbolic execution inspired by separation logics: e.g. the compositional verification tools and platforms Verifast, Viper, Gillian, CN; the true platforms for true bug detection Infer-Pulse, Gillian.

These techniques **do scale**.

The CSE and Gillian Team: Past and Current Members


Philippa Gardner


Diego Cupello


Andreas Lööw


Simon Park


Sacha-Élie Ayoun


Caroline Cronjäger


Petar Maksimović


José Fragoso Santos


Nat Karmios


Daniele Nantes


Opale Sjöstedt


Shiva Tamil Kumaran

Traditional Hoare Logic

Traditional Hoare Triples

This reasoning works with the **whole** memory, using conjunction and the conjunction rule to describe different properties of the memory.

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

$$\not\vdash \{\, \mathsf{List}(\mathtt{x}) \wedge \mathsf{List}(\mathtt{y}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} \wedge \mathsf{List}(\mathtt{y}) \,\}$$

Traditional Hoare Logic

Traditional Hoare Triples

This reasoning works with the **whole** memory, using conjunction and the conjunction rule to describe different properties of the memory.

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

$$\not\vdash \{\, \mathsf{List}(\mathtt{x}) \land \mathsf{List}(\mathtt{y}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} \land \mathsf{List}(\mathtt{y}) \,\}$$

$$\vdash \{\, \mathsf{List}(\mathtt{x}) \land \mathsf{List}(\mathtt{y}) \land \mathsf{NReach}(\mathtt{x}, \mathtt{y}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} \land \mathsf{List}(\mathtt{y}) \,\}$$

$$\vdash \{\, \mathsf{List}(\mathtt{x}) \land \mathsf{List}(\mathtt{y}) \land \mathsf{List}(\mathtt{z}) \land \mathsf{NReach}(\mathtt{x}, \mathtt{y}) \land \mathsf{NReach}(\mathtt{y}, \mathtt{z}) \land \mathsf{NReach}(\mathtt{z}, \mathtt{y}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} \land \mathsf{List}(\mathtt{y}) \land \mathsf{List}(\mathtt{z}) \,\}$$

This reasoning does not scale.

Local Hoare Triples

This reasoning works with **partial** memory.

$$\vdash \{\, \mathsf{list}(\mathtt{x}) \,\}\ \mathtt{LDispose(x)}\ \{\, \mathtt{ret} = \mathtt{null} \,\}$$

It uses **separating conjunction** and **frame rule** to disjointly extend the memory.

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \text{side-condition}}{\vdash \{P \star R\}\ C\ \{Q \star R\}}\ \text{frame}$$

Local Hoare Triples

This reasoning works with **partial** memory.

$$\vdash \{\, \mathsf{list}(\mathtt{x}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} \,\}$$

$$\vdash \{\, \mathsf{list}(\mathtt{x}) * \mathsf{list}(\mathtt{y}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} * \mathsf{list}(\mathtt{y}) \,\}$$

$$\vdash \{\, \mathsf{list}(\mathtt{x}) * \mathsf{list}(\mathtt{y}) * \mathsf{list}(\mathtt{z}) \,\}\ \mathtt{LDispose}(\mathtt{x})\ \{\, \mathtt{ret} = \mathtt{null} * \mathsf{list}(\mathtt{y}) * \mathsf{list}(\mathtt{z}) \,\}$$

This reasoning does scale.

A Simple While Language

Expressions

Boolean values $b \in \mathsf{Bool} \stackrel{\mathsf{def}}{=} \{\mathsf{true}, \mathsf{false}\}$

Values $v \in \mathsf{Val} \supseteq \mathbb{N} \cup \mathsf{Bool} \cup \{\texttt{null}\}$

Program Variables $\texttt{x} \in \mathsf{Var}$

Simple Expressions $E \in \mathsf{Exp} ::= v \mid \texttt{x} \mid E + E \mid E - E \mid E \times E \mid E \,/\, E$
$E = E \mid E > E \mid E \wedge E \mid \neg E \mid \ldots$

Program State

Variable Store $s \in \mathsf{Store} \stackrel{\mathsf{def}}{=} \mathsf{Var} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$

Linear Heap $h \in \mathsf{Heap} \stackrel{\mathsf{def}}{=} \mathbb{N} \rightharpoonup_{\mathsf{fin}} \mathtt{Val}$

State $\sigma \in \mathsf{State} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{Heap}$

Notation

$\emptyset$ denotes the empty store and $s[x \mapsto v]$ denotes the store with x updated with v.
$\emptyset$ denotes the empty heap and $h_1 \uplus h_2$ denotes the disjoint union of heaps.

Expression Evaluation

$\mathcal{E}[\![E]\!]_s \in \mathtt{Val}$ denotes the value of expression E with respect to the variable store s.

When $\mathcal{E}[\![E]\!]_s \in \mathsf{Bool}$, we say that E is a **Boolean expression**.

Commands

Commands $C ::= \mathtt{x} := E \mid \mathtt{x} := [E] \mid [E] := E \mid \mathtt{x} := \mathtt{new}(E) \mid \mathtt{dispose}(E) \mid \mathtt{x} := f(\vec{E}) \mid \mathtt{skip} \mid C; C \mid \mathtt{if}\ (E)\ C\ \mathtt{else}\ C \mid \mathtt{while}\ (E)\ C$

$[E]$ denotes the value stored at the heap cell with address given by the value of expression E.

Function definition context, γ, maps function identifiers to their implementations:

$$\gamma(f) = (\vec{\mathtt{x}}, C, E) \text{ where } \mathtt{pv}(E) \subseteq \{\vec{\mathtt{x}}\} \cup \mathtt{pv}(C).$$

We tend to write $f(\vec{\mathtt{x}})\{C; \mathtt{return}\ E\} \in \gamma$ for $\gamma(f) = (\vec{\mathtt{x}}, C, E)$.

Operational Semantics $(s, h), C \Downarrow_\gamma (s', h')$

Assertion Language

The **assertion language** for separation logic provides **assertions** (formulae) for describing the pre- and post-conditions for the Hoare triples.

Logical Expressions and Logical State

Logical Values	$a, a_1, \ldots \in \mathsf{LVal}$	$\supseteq$	Val
Logical Variables	$x, y, \ldots \in \mathsf{LVar}$		
Logical Expressions	$E \in \mathsf{LExp}$	$::=$	$a \mid \mathtt{x} \mid x \mid E + E \mid E - E \mid E \times E \mid E \,/\, E \mid$ $E = E \mid E > E \mid E \wedge E \mid \neg E \mid \ldots,$ for $\mathtt{x} \in \mathsf{PVar}$
Logical Environment	$e \in \mathsf{LEnv}$	$\stackrel{\text{def}}{=}$	$\mathsf{LVar} \rightharpoonup_{\mathsf{fin}} \mathsf{LVal}$
Logical State	$(e, s, h) \in \mathsf{LState}$	$\stackrel{\text{def}}{=}$	$\mathsf{LEnv} \times \mathsf{Store} \times \mathsf{Heap}$

Logical expression evaluation

$\mathcal{E}[\![E]\!]_{e,s} \in \mathsf{Val}$ describes the value of logical expression E with respect to logical store e and variable store s.

Assertions

The set of **assertions**, Assert, is defined by the grammar:

$$
\begin{array}{rll}
P \in \mathsf{Assert} ::= & P \wedge P \mid P \Rightarrow P \mid \mathsf{True} \mid \mathsf{False} & \text{classical connectives} \\
& \mid E = E \mid E > E \mid E \in X \mid \ \ldots & \text{Boolean assertions} \\
& \mid \exists x.\, P & \text{existential quantification} \\
& \mid P \star P \mid \mathsf{emp} & \text{separating connectives} \\
& \mid E \mapsto E & \text{linear heap cell assertion} \\
& \mid \mathsf{pred}(\vec{E}) & \text{predicate assertion}
\end{array}
$$

where $X \subseteq \mathsf{LVal}$ and $x \in \mathsf{LVar}$.

Predicate definition: $\mathsf{pred}(\vec{x}) \stackrel{\mathsf{def}}{=} P$, where the free logical variables of P are in $\{\vec{x}\}$.

Satisfiability Relation

The logical state (e, s, h) **satisfies** P, written $e, s, h \vDash P$, is defined by:

$$
\begin{array}{lcl}
e,s,h \vDash P_1 \wedge P_2 & \iff & e,s,h \vDash P_1 \wedge e,s,h \vDash P_2 \\
e,s,h \vDash P_1 \Rightarrow P_2 & \iff & e,s,h \vDash P_1 \implies e,s,h \vDash P_2 \\
e,s,h \vDash \mathsf{True} & \iff & \text{always} \\
e,s,h \vDash \mathsf{False} & \iff & \text{never} \\
e,s,h \vDash E_1 = E_2 & \iff & (\mathcal{E}[\![E_1 = E_2]\!]_{e,s} = \mathsf{true}) \wedge h = \emptyset \\
e,s,h \vDash E_1 > E_2 & \iff & (\mathcal{E}[\![E_1 > E_2]\!]_{e,s} = \mathsf{true}) \wedge h = \emptyset \\
e,s,h \vDash E \in X & \iff & (\mathcal{E}[\![E \in X]\!]_{e,s} = \mathsf{true}) \wedge h = \emptyset \\
e,s,h \vDash \exists x.\, P & \iff & \exists v \in \mathsf{Val}.\, e[x \mapsto v], s, h \vDash P \\
e,s,h \vDash P_1 \star P_2 & \iff & \exists h_1, h_2 \in \mathsf{Heap}.\, h = h_1 \uplus h_2 \wedge e,s,h_1 \vDash P_1 \wedge e,s,h_2 \vDash P_2 \\
e,s,h \vDash \mathsf{emp} & \iff & h = \emptyset \\
e,s,h \vDash E_1 \mapsto E_2 & \iff & h = \{\mathcal{E}[\![E_1]\!]_{e,s} \mapsto \mathcal{E}[\![E_2]\!]_{e,s}\} \\
e,s,h \vDash \mathsf{pred}(\vec{E}) & \iff & \mathsf{pred}(\vec{x}) \stackrel{\mathsf{def}}{=} P \wedge e,s,h \vDash P[\vec{E}/\vec{x}]
\end{array}
$$

We write $[\![P]\!]_{e,s} \stackrel{\text{def}}{=} \{h : e, s, h \vDash P\}$.

Exercise

State whether each assertion is satisfiable or unsatisfiable; when satisfiable, describe the heaps that satisfy the assertion.

1. $10 \mapsto 1 \star 10 \mapsto 1$
2. $10 \mapsto 1 \star 10 \mapsto 2$
3. $10 \mapsto 1 \wedge 11 \mapsto 1$

Additional questons and all answers given on our webpage.

Some Properties

An assertion P is **valid**, written $\vDash P$, if $e, s, h \vDash P$ for all e, s and h.

Some properties of $*$: commutativity, associativity, unit emp, scope extrusion of the existential and some distributivity:

$$\vDash (P_1 \wedge P_2) \star Q \Rightarrow (P_1 \star Q) \wedge (P_2 \star Q)$$
$$\vDash (P_1 \vee P_2) \star Q \;\Leftrightarrow\; (P_1 \star Q) \vee (P_2 \star Q)$$

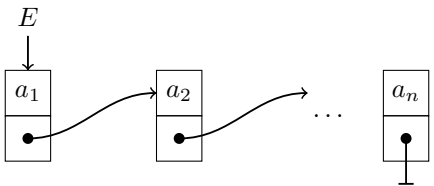
Some properties of the cell assertion:

$$\vDash E_1 \mapsto E_2 \Rightarrow E_1 \in \mathbb{N} \wedge E_2 \in \mathsf{Val}$$
$$\vDash E_1 \mapsto E_2 \star E_3 \mapsto E_4 \Rightarrow E_1 \neq E_3$$
$$\vDash E_1 \mapsto E_2 \wedge E_3 \mapsto E_4 \Rightarrow E_1 = E_3 \wedge E_2 = E_4$$

Derived Assertions and Predicate Definitions

$$
\begin{array}{rcl}
E \mapsto - & \stackrel{\text{def}}{=} & \exists x.\, E \mapsto x, \text{ for } x \text{ not free in } E \\
E \mapsto E_1, E_2 & \stackrel{\text{def}}{=} & (E \mapsto E_1) \star (E+1 \mapsto E_2) \\
\mathsf{list}(x, \alpha) & \stackrel{\text{def}}{=} & (x = \texttt{null} \star \alpha = [\,]) \vee (\exists a, l_1, y.\, x \mapsto a, y \star \mathsf{list}(y, \alpha_1) \star \alpha = a{:}\alpha_1)
\end{array}
$$

Predicate $\mathsf{list}(E, \alpha)$ is satisfied by a singly-linked list that has the shape



where $\alpha = [a_1, a_2, \cdots, a_n]$.

List Predicate with Values: Properties

Exercise

- $\vDash \mathsf{list}(E, [\,]) \Rightarrow$ What does that tell us about E?
- $\vDash \mathsf{list}(E, a{:}\alpha) \Rightarrow$ What does that tell us about E?
- $\vDash \mathsf{list}(\mathtt{null}, \alpha) \Rightarrow$ What does that tell us about α?
- $\vDash E_1 \mapsto E_2 \star \mathsf{list}(E_3, \alpha) \star E_3 \neq \mathtt{null} \Rightarrow$ What is the relationship between E_1 and E_3?

Separation Logic

Hoare Triples

A **Hoare triple**, $\{P\}C\{Q\}$, is a relation between two assertions P and Q and command C where P is called the **pre-condition** and Q the **post-condition**.

A Hoare triple of a command C is **valid**, written

$$\vDash \{P\}C\{Q\}$$

if and only if, starting in any logical state in which the assertion P holds, no execution of the simple command C aborts and, for any execution of C that terminates, the assertion Q holds in the final logical state.

The proof rules of separation logic provide a way of discovering valide Hoare triples, written $\vdash \{P\}C\{Q\}$. These rules are given on our webpage for OPLSS'26.

Soundness $\quad \vdash \{P\}C\{Q\} \Rightarrow \vDash \{P\}C\{Q\}$

LLen(x): List Length

```
LLen(x) {
  if (x = null) {
    n := 0
  } else {
    t := [x + 1];
    n := LLen(t);
    n := n + 1
  };
  return n
}
```

```
LLen(x) {
  y := x;
  n := 0;
  while (y ≠ null) {
    y := [y + 1];
    n := n + 1
  };
  return n
}
```

where $[x + 1]$ denotes the contents of the storage at the address given by expression $x + 1$.

LLen(x) Proof Sketch: function entry and base case

$\vdash$ $\{\mathtt{x} = x \star \mathsf{list}(x, \alpha)\}$

```
LLen(x) {
```

// Initialise the local variables and ensure the Boolean condition is evaluable.

$\{\mathtt{x} = x \star \mathsf{list}(x, \alpha) \star \mathtt{n}, \mathtt{t} = \mathtt{null}\}$

$\{\mathtt{x} = x \star \mathsf{list}(x, \alpha) \star \mathtt{n}, \mathtt{t} = \mathtt{null} \star (\mathtt{x} = \mathtt{null}) \in \mathsf{Bool}\}$

```
  if (x = null) {
```

$\{\mathtt{x} = x \star \mathsf{list}(x, \alpha) \star \mathtt{n}, \mathtt{t}, \mathtt{x} = \mathtt{null}\}$

// As x = null, unfolding the list predicate yields the base case.

$\{\mathtt{x} = x \star (x = \mathtt{null} \star \alpha = [\,]) \star \mathtt{n}, \mathtt{t} = \mathtt{null}\}$

$\mathtt{n} := 0$

$\{\mathtt{x} = x \star (x = \mathtt{null} \star \alpha = [\,]) \star \mathtt{t} = \mathtt{null} \star \mathtt{n} = 0\}$

// Forget x and t as no longer used, connect n to α, and fold back list predicate.

$\{(x = \mathtt{null} \star \alpha = [\,]) \star \mathtt{n} = |\alpha|\}$

$\{\mathsf{list}(x, \alpha) \star \mathtt{n} = |\alpha|\}$

```
  } else {
```

LLen(x) Proof Sketch: recursive case

} else {

$\{\mathtt{x} = x \star \mathsf{list}(x, \alpha) \star \mathtt{n}, \mathtt{t} = \mathtt{null} \star \mathtt{x} \neq \mathtt{null}\}$

// Unfold list predicate to recursive case, identify the resource needed

$\{\exists v, \beta, y.\ x \mapsto v, y \star \alpha = v{:}\beta \star \mathsf{list}(y, \beta) \star \mathtt{n}, \mathtt{t} = \mathtt{null}\}$

exists, frame + cons

$\{x \mapsto v, y \star \mathsf{list}(y, \beta) \star \mathtt{n}, \mathtt{t} = \mathtt{null}\}$

$\mathtt{t} := [\mathtt{x} + 1];$

// Use the fcall rule to **consume** precondition and **produce** postcondition.

$\{x \mapsto v, y \star \mathsf{list}(y, \beta) \star \mathtt{n} = \mathtt{null} \star \mathtt{t} = y\}$

$\mathtt{n} := \mathtt{llen}(\mathtt{t});$

$\{x \mapsto v, y \star \mathsf{list}(y, \beta) \star \mathtt{n} = |\beta|\}$

$\mathtt{n} := \mathtt{n} + 1$

$\{x \mapsto v, y \star \mathsf{list}(y, \beta) \star \mathtt{n} = |\beta| + 1\}$

// frame back assertions, add exists and fold back list predicate.

$\{\exists v, \beta, y.\ x \mapsto v, y \star \alpha = v{:}\beta \star \mathsf{list}(y, \beta) \star \mathtt{n} = |\beta| + 1\}$

$\{\mathsf{list}(\mathtt{x}, \alpha) \star \mathtt{n} = |\alpha|\}$

}

LLen(x) Proof Sketch: end of conditional statement

$\vdash \{\mathtt{x} = x \star \mathsf{list}(x, \alpha)\}$

```
LLen(x) {
  {x = x ⋆ list(x, α) ⋆ n, t = null}
  {x = x ⋆ list(x, α) ⋆ n, t = null ⋆ (x = null) ∈ Bool}
  if (x = null) {
    ...
    {list(x, α) ⋆ n = |α|}
  } else {
    ...
    {list(x, α) ⋆ n = |α|}
  }
  // As the post-condition of both the if and else bodies are the same,
  // we infer the post-condition of the conditional statement.
  {list(x, α) ⋆ n = |α|}
  return n
}
{list(x, α) ⋆ ret = |α|}
```

LLen(x): List Length

```
LLen(x) {
  if (x = null) {
    n := 0
  } else {
    t := [x + 1];
    n := LLen(t);
    n := n + 1
  };
  return n
}
```

```
LLen(x) {
  y := x;
  n := 0;
  while (y ≠ null) {
    y := [y + 1];
    n := n + 1
  };
  return n
}
```

where $[x + 1]$ denotes the contents of the storage at the address given by expression $x + 1$.

Iterative List-length Algorithm

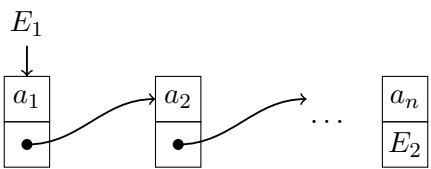
Consider an iterative list-length algorithm where $|\alpha|$ is the length of the list α

$$
\begin{array}{l}
\vdash \{\mathtt{x} = x \star \mathsf{list}(x, \alpha)\} \\
\quad \mathtt{LLen(x)}\ \{ \\
\qquad \{\mathtt{x} = x \star \mathsf{list}(\mathtt{x}, \alpha) \star \mathtt{y}, \mathtt{n} = \mathtt{null}\} \\
\qquad \mathtt{y} := \mathtt{x};\ \mathtt{n} := 0; \\
\qquad \{\exists \alpha_1, \alpha_2.\ ??? \star \mathsf{list}(\mathtt{y}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1|\} \\
\qquad \mathtt{while}\ (\mathtt{y} \neq \mathtt{null})\ \{ \\
\qquad\quad \mathtt{y} := [\mathtt{y} + 1]; \\
\qquad\quad \mathtt{n} := \mathtt{n} + 1; \\
\qquad \} \\
\qquad \{\mathsf{list}(x, \alpha) \star \mathtt{n} = |\alpha|\} \\
\qquad \mathtt{return\ n}; \\
\quad \} \\
\quad \{\mathsf{list}(x, \alpha) \star \mathsf{ret} = |\alpha|\}
\end{array}
$$

List-segment Predicate

$$\mathsf{lseg}(x, z, l) \stackrel{\text{def}}{=} (x = z \star l = [\,]) \vee (\exists a, y, l_1.\ x \mapsto a, y \star \mathsf{lseg}(y, z, l_1) \star l = a{:}l_1)$$

Predicate $\mathsf{lseg}(E_1, E_2, \alpha)$ is satisfied by an incomplete singly-linked list that has the shape



List-segment Predicate: Properties

The following properties hold for the list-segment predicate:

$$\models \mathsf{list}(E, \alpha) \Leftrightarrow \mathsf{lseg}(E, \mathtt{null}, \alpha)$$
$$\models \mathsf{lseg}(E_1, E_2, \alpha) \star \mathsf{lseg}(E_2, E_3, \beta) \Rightarrow \mathsf{lseg}(E_1, E_3, \alpha \cdot \beta)$$
$$\models \mathsf{lseg}(E_1, E_2, \alpha) \star \mathsf{list}(E_2, \beta) \Rightarrow \mathsf{list}(E_1, \alpha \cdot \beta)$$
$$\models \mathsf{lseg}(E_1, E_2, \alpha) \star E_2 \mapsto a, E_3 \Rightarrow \mathsf{lseg}(E_1, E_3, \alpha \cdot [a])$$

where $[a]$ denotes the single-element list containing a.

LLen(x) Proof Sketch: function entry and loop invariant

$\vdash \{\mathtt{x} = x \star \mathsf{list}(x, \alpha)\}$

```
LLen(x) {
```

$\{\mathtt{x} = x \star \mathsf{list}(\mathtt{x}, \alpha) \star \mathtt{y}, \mathtt{n} = \mathtt{null}\}$

```
  y := x;
```

$\{\mathtt{x} = x \star \mathsf{list}(\mathtt{x}, \alpha) \star \mathtt{y} = x \star \mathtt{n} = \mathtt{null}\}$

```
  // Forget x as it is no longer used
```

$\{\mathsf{list}(x, \alpha) \star \mathtt{y} = x \star \mathtt{n} = \mathtt{null}\}$

```
  n := 0;
```

$\{\mathsf{list}(x, \alpha) \star \mathtt{y} = x \star \mathtt{n} = 0\}$

```
  // Set up the loop invariant: the traversed part (initially, nothing) is a list segment from x to y,
  // the untraversed part (initially, everything) is a list at y, the contents of the two parts (initially, [ ] and α)
  // form the full list contents α, and n is counting the length of the traversed part
```

$\{\exists \alpha_1, \alpha_2 \,.\, \mathsf{lseg}(x, \mathtt{y}, \alpha_1) \star \mathsf{list}(\mathtt{y}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1| \star (\mathtt{y} \neq \mathtt{null} \mathrel{\dot\in} \mathsf{Bool})\}$

```
  while (y ≠ null) {
    // Loop entry: invariant and loop condition
```

$\{\exists \alpha_1, \alpha_2 \,.\, \mathsf{lseg}(x, \mathtt{y}, \alpha_1) \star \mathsf{list}(\mathtt{y}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1| \wedge \mathtt{y} \neq \mathtt{null}\}$

```
      ...
```

LLen(x) Proof Sketch: proving the loop body

`while` $(\mathtt{y} \neq \mathtt{null})$ {

$\{\exists \alpha_1, \alpha_2 \,.\, \mathsf{lseg}(x, \mathtt{y}, \alpha_1) \star \mathsf{list}(\mathtt{y}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1| \wedge \mathtt{y} \neq \mathtt{null}\}$

// Unfold the shaded predicate to gain access to $[\mathtt{y}+1]$

$\{\exists \alpha_1, \alpha_2, \alpha_3, a, z \,.\, \mathsf{lseg}(x, \mathtt{y}, \alpha_1) \star \mathtt{y} \mapsto a, z \star \mathsf{list}(z, \alpha_3) \star \alpha_2 = a{:}\alpha_3 \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1|\}$

// Record previous value of y as loop body changes y

$\{\exists \alpha_1, \alpha_3, y, a, z \,.\, \mathtt{y} = y \star \mathsf{lseg}(x, y, \alpha_1) \star y \mapsto a, z \star \mathsf{list}(z, \alpha_3) \star \alpha = \alpha_1 \cdot (a{:}\alpha_3) \star \mathtt{n} = |\alpha_1|\}$

// Isolate the resource of the loop

exists, frame, cons

$\{\mathtt{y} = y \star y \mapsto a, z \star \mathtt{n} = |\alpha_1|\}$

$\mathtt{y} := [\mathtt{y}+1];$

$\{\mathtt{y} = z \star y \mapsto a, z \star \mathtt{n} = |\alpha_1|\}$

$\mathtt{n} := \mathtt{n}+1;$

$\{\mathtt{y} = z \star y \mapsto a, z \star \mathtt{n} = |\alpha_1| + 1\}$

// Bring back the existentials and frame

$\{\exists \alpha_1, \alpha_3, y, a, z. \ \mathtt{y} = z \star y \mapsto a, z \star \mathtt{n} = |\alpha_1| + 1 \star \mathsf{lseg}(x, y, \alpha_1) \star \mathsf{list}(z, \alpha_3) \star \alpha = \alpha_1 \cdot (a{:}\alpha_3)\}$

LLen(x) Proof Sketch: exiting the loop and function exit

// Bring back the existentials and frame

$\{\exists \alpha_1, \alpha_3, y, a, z.\ \mathtt{y} = z \star y \mapsto a, z \star \mathtt{n} = |\alpha_1| + 1 \star \mathsf{lseg}(x, y, \alpha_1) \star \mathsf{list}(z, \alpha_3) \star \alpha = \alpha_1 \cdot (a{:}\alpha_3)\}$

// Re-establish the loop invariant

$\{\exists \alpha_1, \alpha_3, y, a.\ \mathsf{lseg}(x, y, \alpha_1) \star y \mapsto a, \mathtt{y} \star \mathsf{list}(\mathtt{y}, \alpha_3) \star \alpha = \alpha_1 \cdot (a{:}\alpha_3) \star \mathtt{n} = |\alpha_1| + 1\}$

// Apply lemma: (shaded) list segment ⋆ node at end → extended list segment

$\{\exists \alpha_1, \alpha_3, y, a\,.\,\mathsf{lseg}(x, \mathtt{y}, \alpha_1 \cdot [a]) \star \mathsf{list}(\mathtt{y}, \alpha_3) \star \alpha = (\alpha_1 \cdot [a]) \cdot \alpha_3 \star \mathtt{n} = |\alpha_1 \cdot [a]|\}$

$\{\exists \alpha_1, \alpha_2\,.\,\mathsf{lseg}(x, \mathtt{y}, \alpha_1) \star \mathsf{list}(\mathtt{y}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1| \star (\mathtt{y} \neq \mathtt{null} \dot{\in} \mathsf{Bool})\}$

}

$\{(\exists \alpha_1, \alpha_2\,.\,\mathsf{lseg}(x, \mathtt{y}, \alpha_1) \star \mathsf{list}(\mathtt{y}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1|) \wedge \mathtt{y} = \mathtt{null}\}$

$\{\exists \alpha_1, \alpha_2\,.\,\mathsf{lseg}(x, \mathtt{null}, \alpha_1) \star \mathsf{list}(\mathtt{null}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1|\}$

// Apply lemma: (shaded) list segment ending with null is a list

$\{\exists \alpha_1, \alpha_2\,.\,\mathsf{list}(x, \alpha_1) \star \alpha_2 = [\,] \star \alpha = \alpha_1 \cdot \alpha_2 \star \mathtt{n} = |\alpha_1|\}$

$\{\mathsf{list}(x, \alpha) \star \mathtt{n} = |\alpha|\}$

`return n;`

}

// Assign return value

$\{\mathsf{list}(x, \alpha) \star \mathtt{n} = |\alpha| \star \mathsf{ret} = \mathtt{n}\}$

$\{\mathsf{list}(x, \alpha) \star \mathsf{ret} = |\alpha|\}$

Exercise: List Concatenation

```
list_concat(x, y) {
    if (x = null) {
      x := y
    } else {
      t := x;
      n := [x + 1];
      while (n ≠ null) {
        t := n;
        n := [n + 1]
      };
      [t + 1] := y
    };
    return x
  ;}
```

A Gillian Taster

Link: `https://youtu.be/5TTBX4ecZkk`

The Gillian Platform

Semi-automatic Verification & Automatic True Bug Detection

Gillian Platform: PLDI'20, CAV'21

A general platform for the development of compositional analysis tools with strong mathematical foundations:

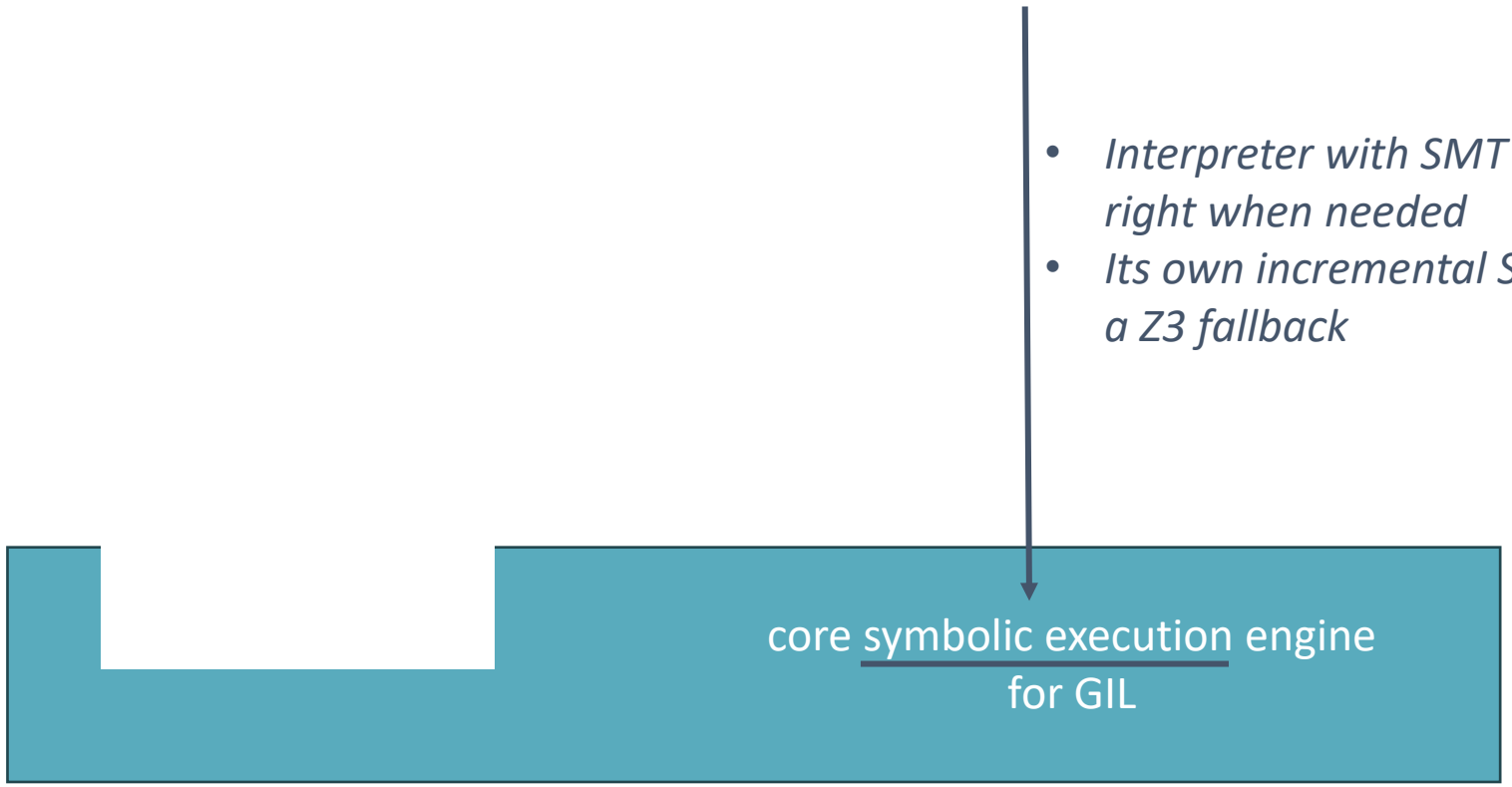
- **Parametric on the partial symbolic state**
- **Gillian-specific match/consume and produce algorithms satisfy the properties of the algebraic interface**
- **Gillian-JavaScript and Gillian-C and Gillian-Rust [PLDI'25] tool instantiations**
- **Applications to semi-automatic verification and true bug detection for real-world code:**
  e.g., verification of the JS implementation (200 loc) and C implementation (900 loc) of critical AWS encryption SDK code

Gillian

Gillian: Parametric Symbolic Execution

Gillian: Parametric Symbolic Execution

quite literally speaking



- *Interpreter with SMT queries sent right when needed*
- *Its own incremental SMT solver with a Z3 fallback*

Gillian: Parametric Symbolic Execution

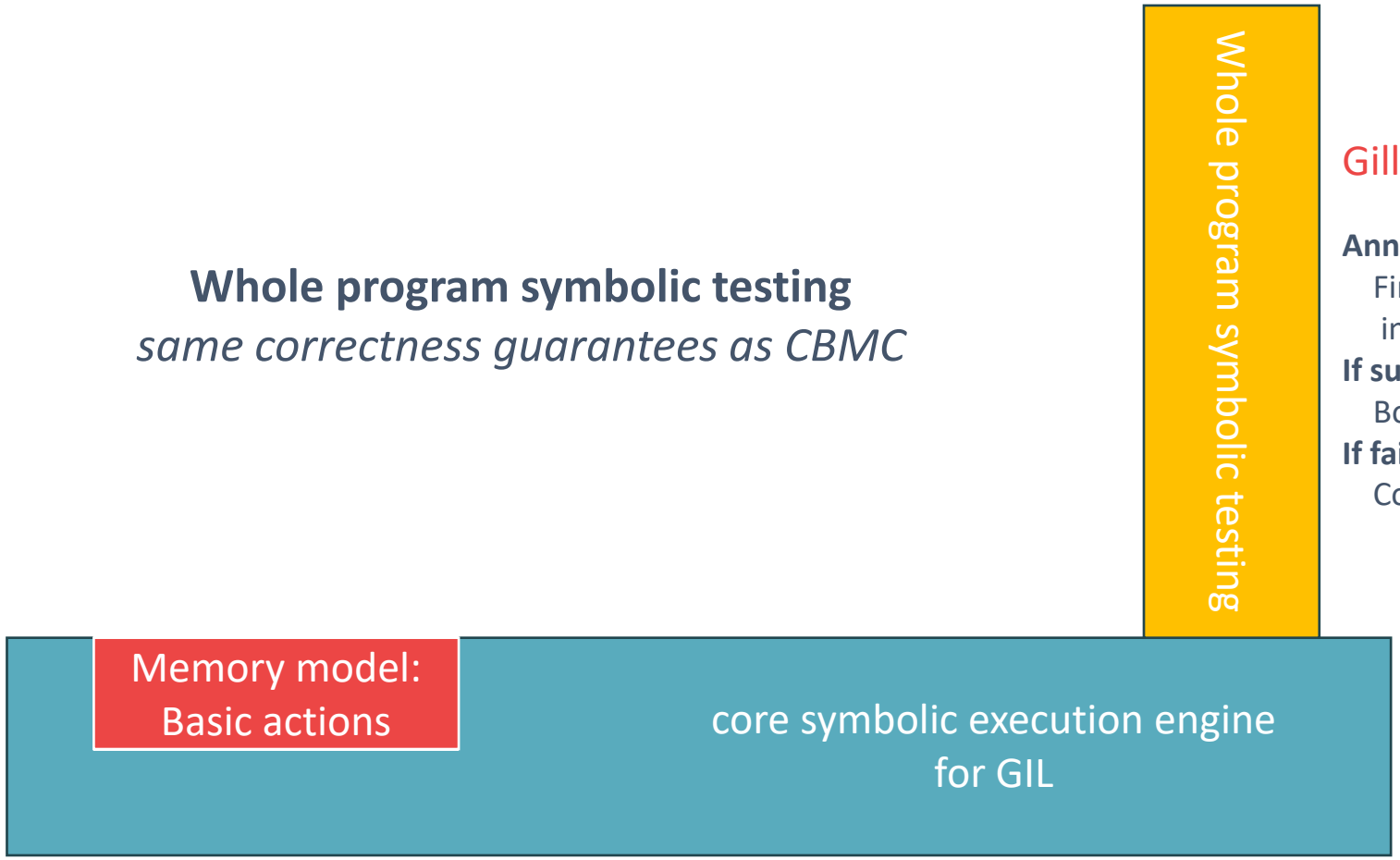
Semantic hole: memory operations
e.g. load, store, pointer validity checks in C,
prototype access in JS…

quite literally speaking

- *Interpreter with SMT queries sent right when needed*
- *Its own incremental SMT solver with a Z3 fallback*

Gillian: Parametric Symbolic Execution

Whole program symbolic testing
same correctness guarantees as CBMC

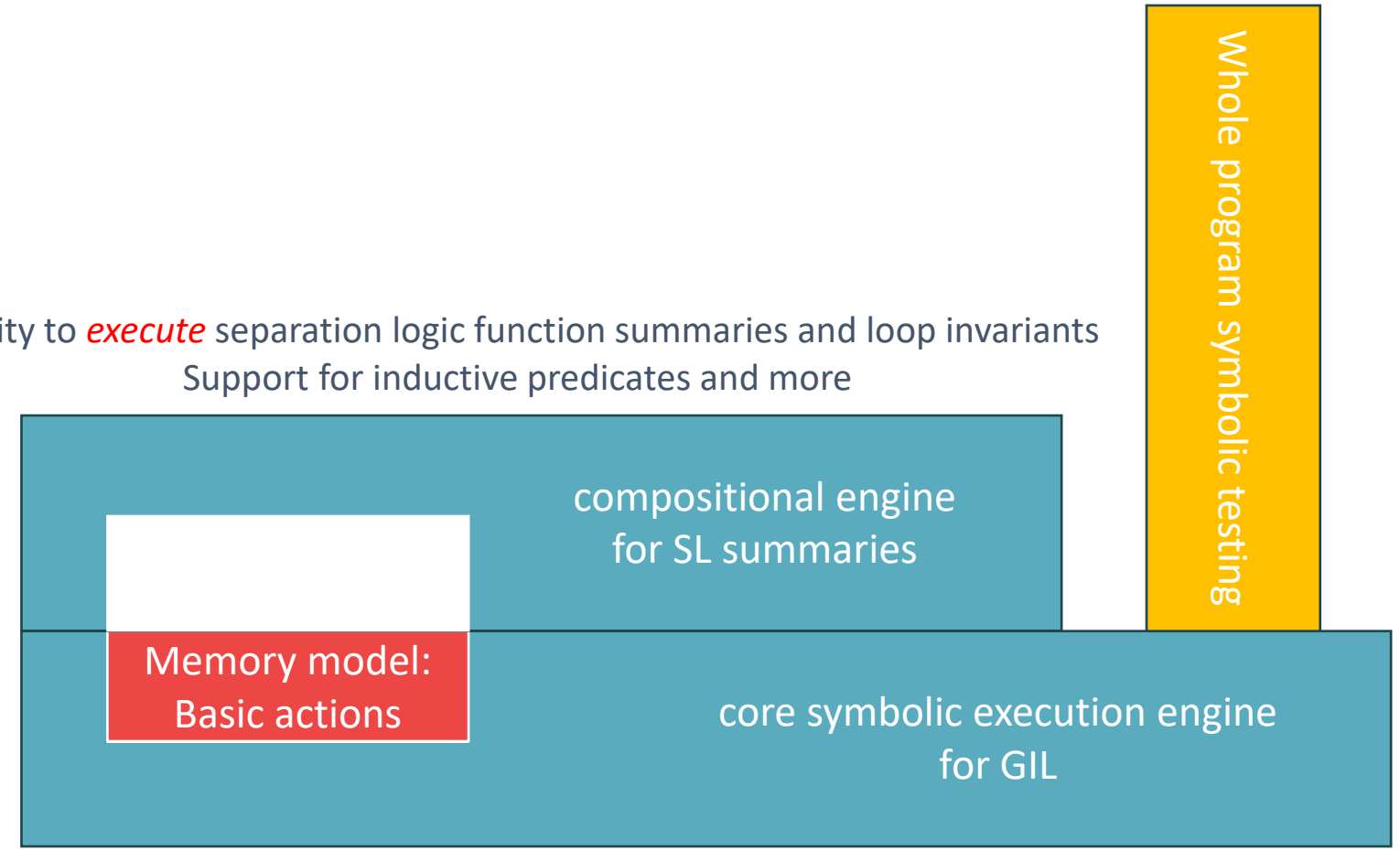


Gillian: Part 1, PLDI'20

Annotations:
First-order assertions about symbolic inputs/outputs
If successful:
Bounded correctness guarantees (loops unrolled)
If failure:
Concrete counter-model

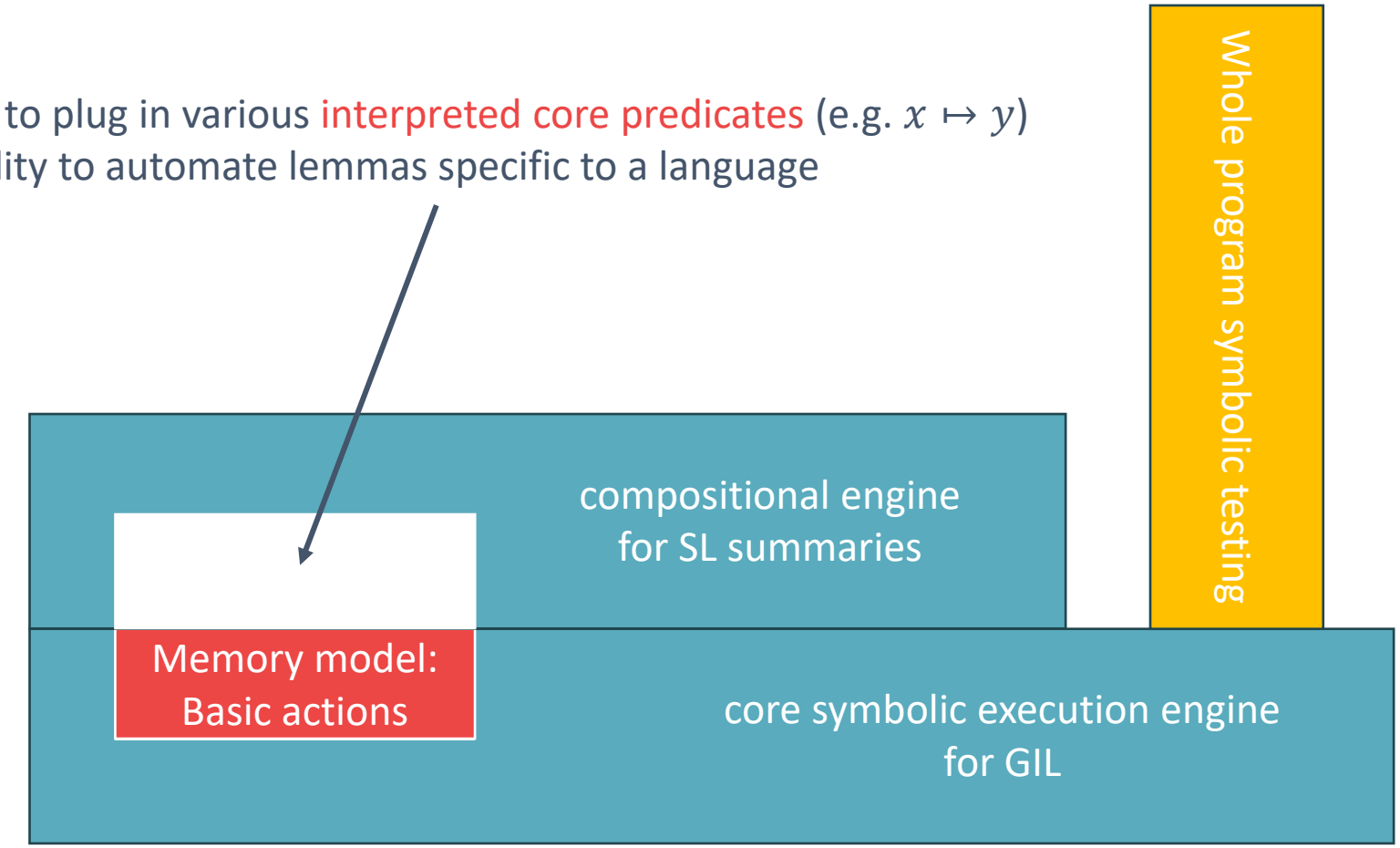
Gillian: Compositional Parametric Symbolic Execution

Ability to *execute* separation logic function summaries and loop invariants
Support for inductive predicates and more



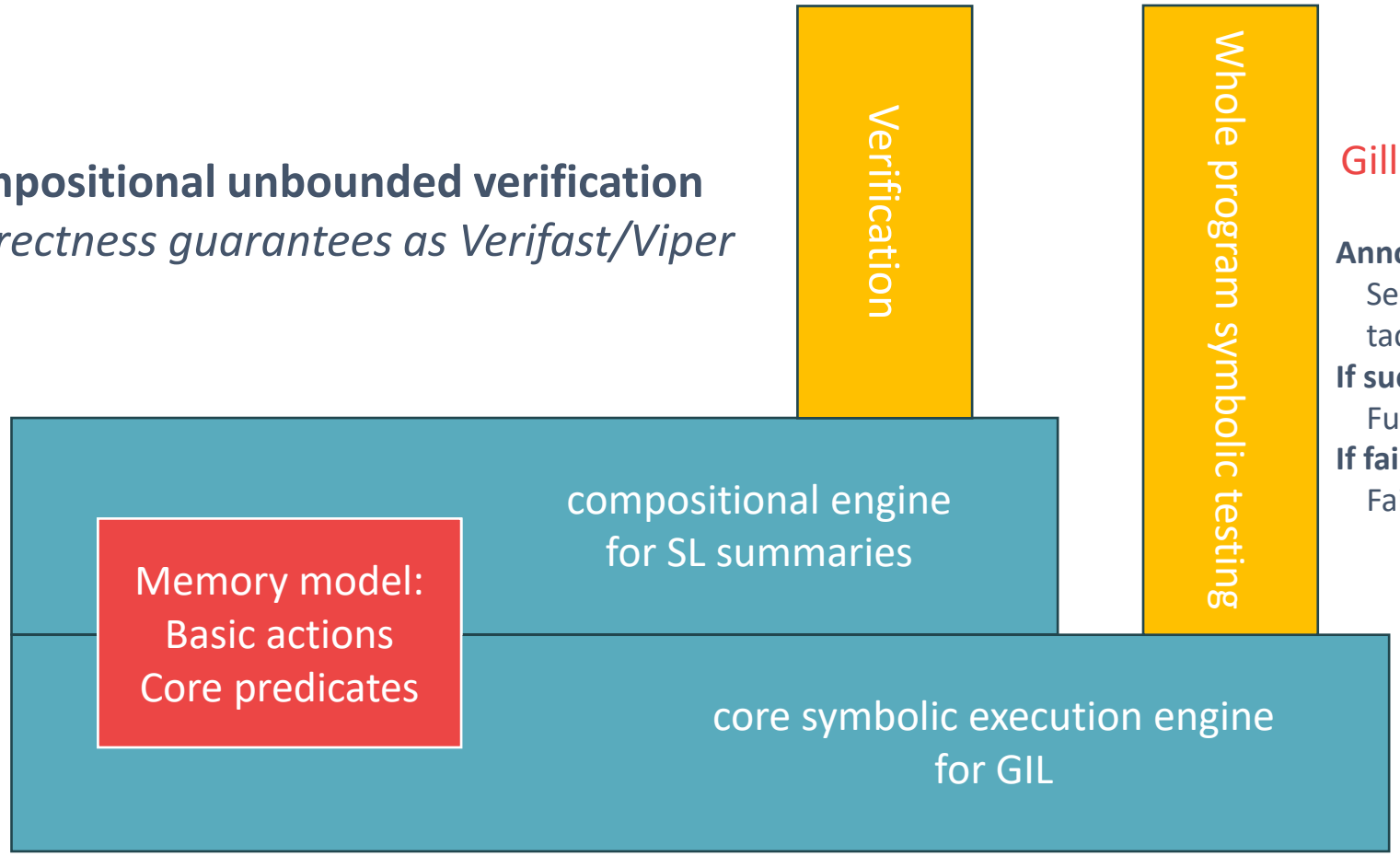
Gillian: Compositional Parametric Symbolic Execution

Ability to plug in various interpreted core predicates (e.g. $x \mapsto y$)
=> Ability to automate lemmas specific to a language



Gillian: Compositional Parametric Symbolic Execution

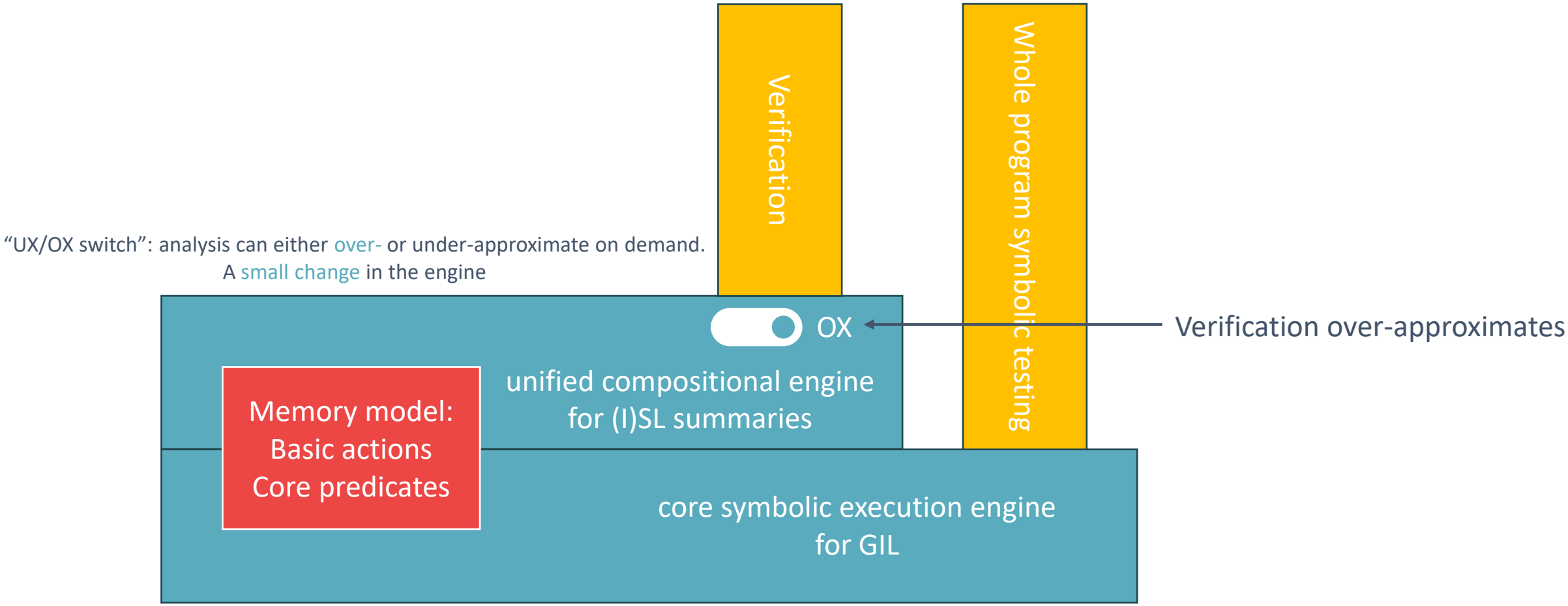
Full compositional unbounded verification
same correctness guarantees as Verifast/Viper



Gillian: Part 2, CAV'21

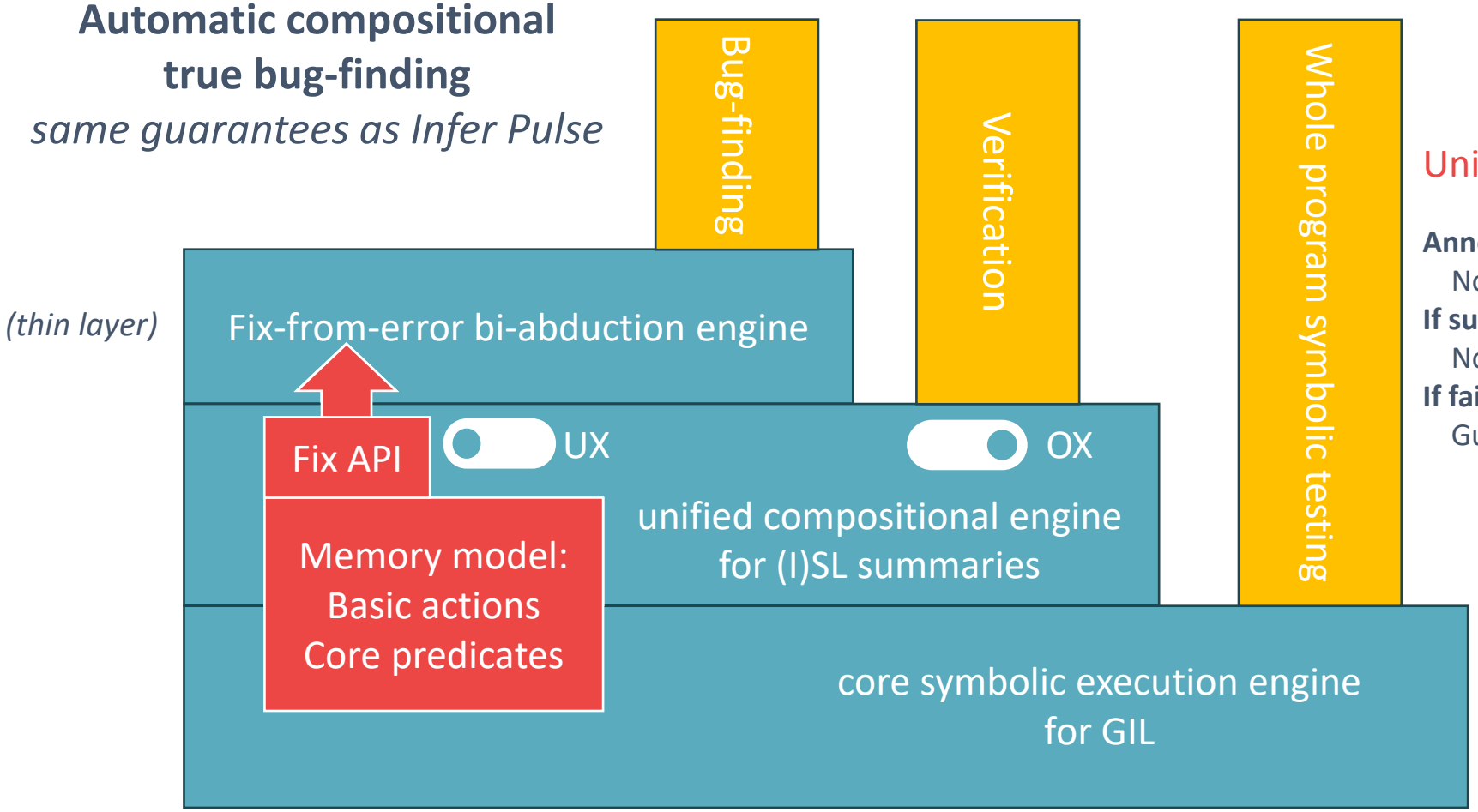
Annotations:
Separation logic specs, invariants, lemmas, tactics…
If successful:
Functional correctness
If failure:
Failing symbolic trace

Gillian: Unified Compositional Parametric Symbolic Execution



Gillian: Unified Compositional Parametric Symbolic Execution

Automatic compositional true bug-finding
same guarantees as Infer Pulse



Unified CSE: Submitted

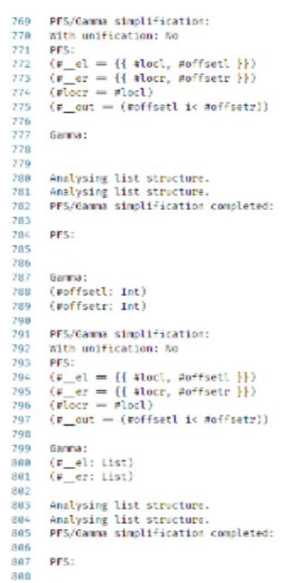
Annotations:
None
If successful:
No guarantee
If failure:
Guarantee of a true bug

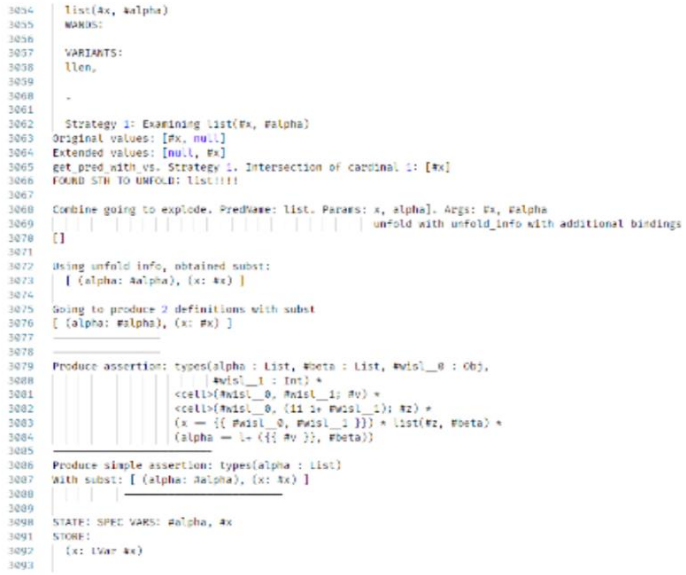
Gillian Instantiations

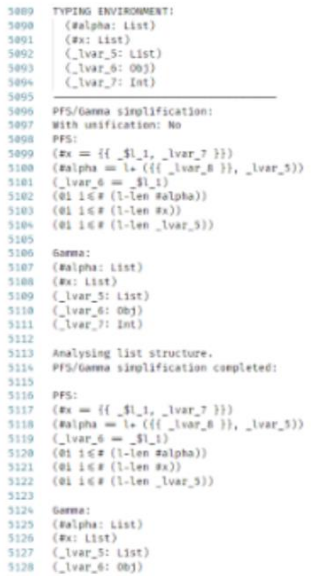
WiSL (Gillian-While)	Simple block-offset model for teaching and experimentation Applied to data-structure algorithms
Gillian-JS	Extensible object memory model, JaVerT compiler Used for symbolic testing of Buckets-JS & verification of code from AWS Encryption SDK
Gillian-C	Block-offset memory model, CompCert and CBMC compilers Used for symbolic testing of Collections-C & verification of code from AWS Encryption SDK
Gillian-Rust	Step up from Gillian-C; partially laid-out heap, Iris ghost resources; Rust compiler + ours Used to verify parts of standard library (LinkedList & Vec)
Gillian-CHERI	Extension of Gillian-C memory model with support for CHERI capabilities

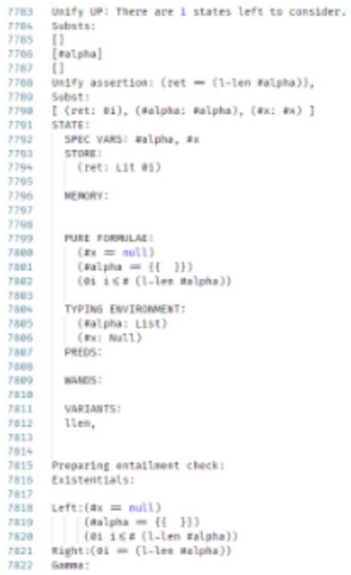
Gillian Student Lab

Gillian Verification Debugging









Old Gillian (file log)

Logging infrastructure

First debugger (DAP-compliant)

Visual debugger!

Source-level debugging

SSFT lab

TACAS paper!

2019 Sacha, Petar

2020 Radu Lacraru

2021 Matthew Ho

2022 Nat Karmios

2023

2025

2026

A Gillian Taster

Link: https://youtu.be/5TTBX4ecZkk