Compositional Software Verification

Philippa Gardner
Imperial College London

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. Klee; 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 Verifast, Viper, Gillian, CN; the true bug-finding tools Infer-Pulse, Gillian.

These techniques **do scale**.

This Summer School

Lecture 1: Compositional Verification

- Separation logic
- Specification and verification of sequential programs for mutating data structures
- Tools inspired from separation logic, based on compositional symbolic execution

Lab session 1: An Introduction to Gillian

- List algorithms
- Some fun, harder examples of data-structure algorithms

Lecture 2: Compositional Symbolic Execution

- Foundations of compositional symbolic execution, parametric on the state
- State combinators for compositional symbolic execution

Lab session 2: Experience with Gillian

- Some fun, harder examples of data-structure algorithms
- Examples transferred from the Collection-C library

Some of the Gang

Philippa Gardner

Diego Cupello

Andreas Lööw

Nat Karmios

Daniele Nantes

Opale Sjöstedt

A Problem with 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.

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

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

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

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

This reasoning does not scale.

A Solution using Separation Logic

Local Hoare Triples

This reasoning works with **partial** memory.

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

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

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

A Solution using Separation Logic

Local Hoare Triples

This reasoning works with **partial** memory.

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

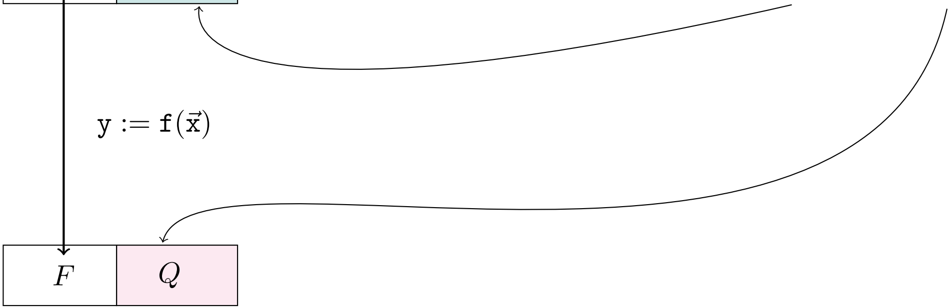
$$\vdash \{\,\mathsf{list}(x) * \mathsf{list}(y)\,\}\ \mathtt{LDispose}(x)\ \{\,\mathtt{ret} = \mathtt{null} * \mathsf{list}(y)\,\}$$

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

This reasoning does scale.

Function Compositionality

$$\{\, \vec{\mathtt{x}} = \vec{x} \star \boxed{P} \,\}\ \mathtt{f}(\vec{\mathtt{x}})\ \{\, \boxed{Q} \,\}$$



Simple While Language

Expressions

$$
\begin{array}{rll}
\textbf{Boolean values} & b \in \mathsf{Bool} & \stackrel{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\} \\
\textbf{Values} & v \in \mathsf{Val} & \supseteq \mathbb{N} \cup \mathsf{Bool} \cup \{\mathtt{null}\} \\
\textbf{Program Variables} & \mathtt{x} \in \mathsf{Var} & \\
\textbf{Expressions} & E \in \mathsf{Exp} & ::= v \mid \mathtt{x} \mid E + E \mid E - E \mid E = E \mid E > E \mid E \wedge E \mid \neg E \mid \ldots
\end{array}
$$

Program State

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

Notation

$\emptyset$ denotes the empty store. $s[x \mapsto v]$ 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 $C ::= \texttt{skip} \mid \texttt{x} := E \mid C; C \mid \texttt{if } (E)\ C \texttt{ else } C \mid \texttt{while } (E)\ C \mid \texttt{x} := f(\vec{E}) \mid$
$\texttt{x} := [E] \mid [E] := E \mid \texttt{x} := \texttt{new}(E) \mid \texttt{dispose}(E)$

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

Operational Semantics $(s, h), C \Downarrow (s', h')$

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

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

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:

$$
\begin{array}{rll}
P \in \mathsf{Assert} ::= & P \land P \mid P \Rightarrow P \mid \mathsf{True} \mid \mathsf{False} & \text{classical connectives} \\
& \mid E = E \mid E > E \mid E \in X \mid \ \dots & \text{Boolean assertions} \\
& \mid \exists x.\, P & \text{existential quantification} \\
& \mid P \star P \mid \mathsf{emp} \mid \circledast_{E_1 \leq x < E_2} P & \text{separating connectives} \\
& \mid E \mapsto E & \text{linear heap cell assertion}
\end{array}
$$

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

Satisfiability Relation

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

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

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

Some Properties

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

Some properties of $*$:

$$
\begin{aligned}
&\models P \star Q \;\Leftrightarrow\; Q \star P \\
&\models P \star (Q \star R) \;\Leftrightarrow\; (P \star Q) \star R \\
&\models P \star \mathsf{emp} \Leftrightarrow P \\
&\models (P_1 \wedge P_2) \star Q \Rightarrow (P_1 \star Q) \wedge (P_2 \star Q) \\
&\models (P_1 \vee P_2) \star Q \;\Leftrightarrow\; (P_1 \star Q) \vee (P_2 \star Q) \\
&\models (\exists x.P) \star Q \Leftrightarrow \exists x.(P \star Q) \text{ if no variable clash}
\end{aligned}
$$

Some properties of the cell assertion:

$$\models E_1 \mapsto E_2 \Rightarrow E_1 \in \mathbb{N} \wedge E_2 \in \mathsf{Val}$$

$$\models E_1 \mapsto E_2 \star E_3 \mapsto E_4 \Rightarrow E_1 \neq E_3$$

$$\models E_1 \mapsto E_2 \wedge E_3 \mapsto E_4 \Rightarrow E_1 = E_3 \wedge E_2 = E_4$$

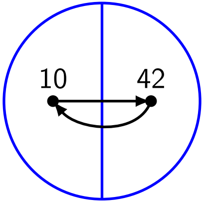
Exercise

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. $10 \mapsto 1 \wedge 11 \mapsto 1$
4. $10 \mapsto 1 \wedge 11 \mapsto 2$
5. $10 \mapsto 1 \vee 10 \mapsto 2$
6. $10 \mapsto - \wedge 10 \mapsto 1$
7. $10 \mapsto - \star 10 \mapsto 1$
8. $(10 \mapsto 11 \star 11 \mapsto -) \vee 10 \mapsto 0$
9. $(10 \mapsto 1 \star \mathsf{true}) \wedge (11 \mapsto 2 \star \mathsf{true})$

Example

Assertion: $\mathtt{x} \mapsto \mathtt{y} \star \mathtt{y} \mapsto \mathtt{x}$



Variable Store: $\{\mathrm{x} \rightarrow 10, \mathrm{y} \rightarrow 42\}$

Heap: $\{10 \mapsto 42, 42 \mapsto 10\}$

Example

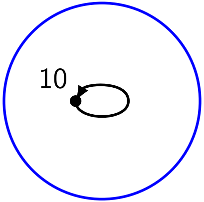
Assertion: $\mathrm{x} \mapsto \mathrm{y} \wedge \mathrm{y} \mapsto \mathrm{x}$

Variable Store: $\{\mathrm{x} \to 10, \mathrm{y} \to 42\}$

No heap satisfies this assertion.

Example

Assertion: $\mathtt{x} \mapsto \mathtt{y} \wedge \mathtt{y} \mapsto \mathtt{x}$



Variable Store: $\{\mathtt{x} \to 10, \mathtt{y} \to 10\}$

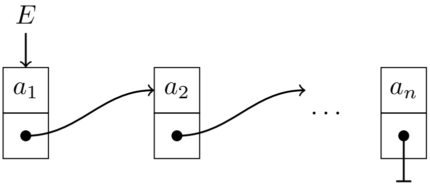
Heap: $\{10 \mapsto 10\}$

The heap $\{10 \mapsto 10\}$ satisfies assertion $\mathtt{x} \mapsto \mathtt{y} \wedge \mathtt{y} \mapsto \mathtt{x}$.

Derived Assertions and Predicate Definitions

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

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

- $\models$ $\mathsf{list}(E, [\,]) \Rightarrow$ What does that tell us about E?
- $\models$ $\mathsf{list}(E, a{:}\alpha) \Rightarrow$ What does that tell us about E?
- $\models$ $\mathsf{list}(\mathtt{null}, \alpha) \Rightarrow$ What does that tell us about α?
- $\models$ $E_1 \mapsto E_2 \star \mathsf{list}(E_3, \alpha) \star E_3 \neq \mathtt{null} \Rightarrow$ 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 **holds**, written

$$\models \{P\}C\{Q\}$$

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 are given on our webpage for SSFT'25.

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 $x + 1$.

$\texttt{LLen}(\texttt{x})$ Proof Sketch: function entry and base case

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

$\texttt{LLen}(\texttt{x})\ \{$

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

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

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

$\texttt{if}\ (\texttt{x} = \texttt{null})\ \{$

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

// As $\texttt{x} = \texttt{null}$, unfolding the list predicate yields the base case.

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

$\texttt{n} := 0$

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

// Forget $\texttt{x}$, $\texttt{t}$ as no longer used, connect $\texttt{n}$ to α, and fold back list predicate.

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

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

$\}\ \texttt{else}\ \{$

$\texttt{LLen(x)}$ 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 and simplify assertion.

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

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.

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

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

```
}
```

$\mathtt{LLen(x)}$ Proof Sketch: end of conditional statement

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

```
LLen(x) {
```

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

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

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

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

```
} else {
  ...
```

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

```
}
```

// As the post-condition of both the if and else bodies are the same,
// we infer the post-condition of the conditional statement.

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

```
return n
}
```

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

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

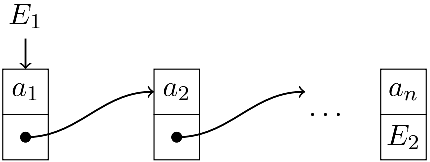
Consider an iterative list-length algorithm where $|\alpha|$ is the length of the list α

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

List-segment Predicate

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

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

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

$\mathtt{LLen(x)}$ {

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

$\mathtt{y} := \mathtt{x};$

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

// Forget x as it is no longer used

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

$\mathtt{n} := 0;$

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

// 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

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

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

// 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 (y ≠ null) {
```

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

// 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 $\mathtt{y}$ as loop body changes $\mathtt{y}$

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

// Isolate the resource of the loop

ex + fr + co

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

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

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

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

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

// Bring back the existentials and frame

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

$\mathtt{LLen(x)}$ 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 $\star$ node at end $\rightarrow$ extended list segment

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

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

}

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

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

// 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

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

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

Some Current Tools inspired by Separation Logic

VeriFast

- Research prototype from KU Leuven [Jacobs et al., NFM 2011]
- Compositional verification for Java, C, Rust,
- Partially formalised in Featherweight VeriFast [Vogels et al., LMCS 2015]
- Supports concurrency, fractional permissions, higher-order predicates.

Viper

- Verification infrastructure from ETH Zurich [Müller et al., VMCAI 2016]
- Provides an intermediate language and two verification backends, one based on CSE
- Frontends for Go, Python, Rust, and more
- Formalisation introduced [Dardinier et al., POPL 2025]

Some Current Tools inspired by Separation Logic

CN

- Recently introduced by the University of Cambridge [Pulte et al., POPL 2023]
- Aims at an intuitive developer experience
- Compositional verification of C programs using the Cerberus compiler

Infer-Pulse

- Automatic industry tool from Meta [Le et. al, OOPSLA 2022]
- Under-approximate true bug finding for C/C++/Obj C, Java, Erlang, Hack
- Loosely underpinned by Incorrectness Separation Logic [Raad et al., CAV 2020]

The Gillian Platform

Gillian

José Fragoso Santos

Petar Maksimović

Sacha-Élie Ayoun

- Academic tool from Imperial College London [Fragoso Santos et al., PLDI 2020]
- Parametric on the memory model
- Language-agnostic compositional verification and true bug finding [ECOOP 2024]
- Instantiated to C and JS [PLDI 2020] and Rust [PLDI 2025]
- Used to compositionally verify (part of) the JS and C AWS SDK headers [Maksimović et al., CAV 2021]
- Supported by "matching plans" for automatic frame inference [Lööw et al., ECOOP 2024]

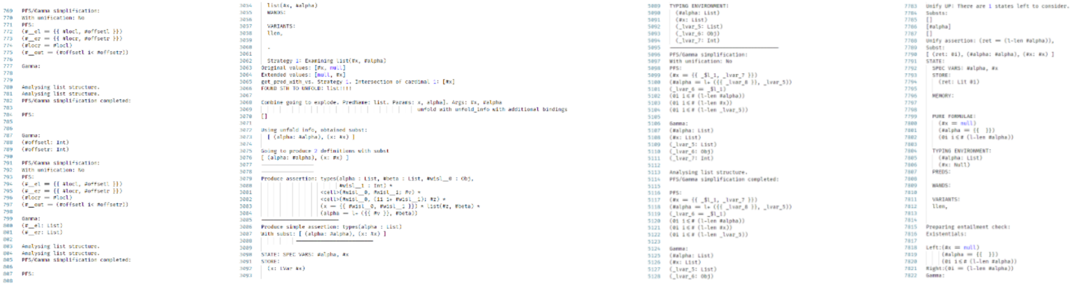
Gillian Instantiations

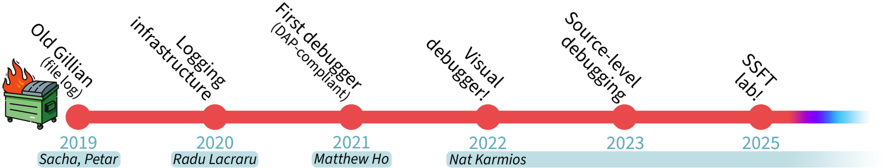
Gillian Instantiations	
WiSL	For teaching and experimentation
Gillian-JS	Extensible-object memory model, JaVerT compiler
Gillian-C	Block-offset memory model, CompCert and CBMC compiler
Gillian-Rust	Step up from Gillian-C: partial laid-out heap, Iris ghost resources; Rust compiler + ours
Gillian-CHERI	Extension of Gillian-C memory model with support for CHERI capabilities

Gillian Applications	
WiSL	Data-structure algorithms
Gillian-JS	Symbolic testing for Buckets-JS; verification of code from AWS Encryption SDK message header
Gillian-C	Symbolic for Collections-JS; verification of code from AWS Encryption SDK message header
Gillian-Rust	Small bits from standard library (LinkedList & Vec)

Gillian Student Lab

Gillian Verification Debugging





This Summer School

Lecture 1: Compositional Verification

- Separation logic
- Specification and verification of sequential programs for mutating data structures
- Tools inspired from separation logic, based on compositional symbolic execution

Lab session 1: An Introduction to Gillian

- list algorithms
- some fun, harder examples of data-structure algorithms

Lecture 2: Compositional Symbolic Execution

- foundations of compositional symbolic execution, parametric on the state
- State combinators for compositional symbolic execution

Lab session 2: Experience with Gillian

- Some fun, harder examples of data-structure algorithms
- Examples transferred from the Collection-C library