

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.

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

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

$$\vdash \{ \text{List}(x) \wedge \text{List}(y) \wedge \text{NReach}(x, y) \} \text{LDispose}(x) \{ \text{ret} = \text{null} \wedge \text{List}(y) \}$$

$$\vdash \{ \text{List}(x) \wedge \text{List}(y) \wedge \text{List}(z) \wedge \text{NReach}(x, y) \wedge \text{NReach}(y, z) \wedge \text{NReach}(z, y) \} \text{LDispose}(x) \{ \text{ret} = \text{null} \wedge \text{List}(y) \wedge \text{List}(z) \}$$

This reasoning does not scale.

A Solution using Separation Logic

Local Hoare Triples

This reasoning works with **partial** memory.

$$\vdash \{ \text{list}(x) \} \text{LDispose}(x) \{ \text{ret} = \text{null} \}$$

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

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

A Solution using Separation Logic

Local Hoare Triples

This reasoning works with **partial** memory.

$$\vdash \{ \text{list}(x) \} \text{LDispose}(x) \{ \text{ret} = \text{null} \}$$

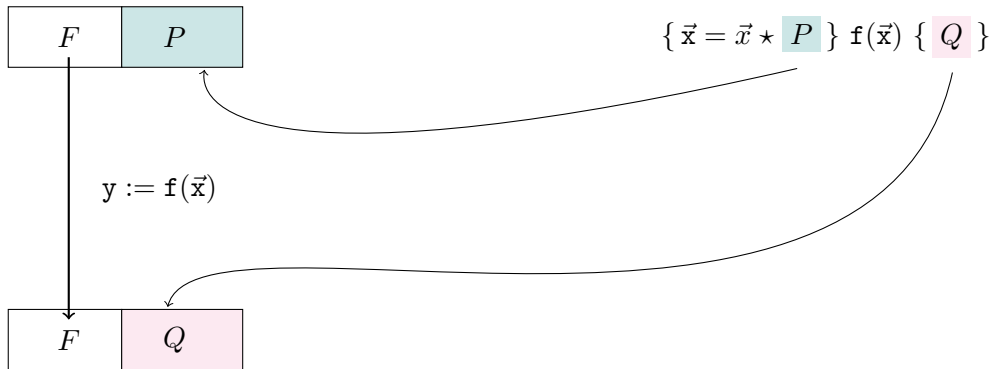
$$\vdash \{ \text{list}(x) * \text{list}(y) \} \text{LDispose}(x) \{ \text{ret} = \text{null} * \text{list}(y) \}$$

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

This reasoning does scale.

A solution using Separation Logic

Function Compositionality



Simple While Language

Expressions

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

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

Program Variables $x \in \text{Var}$

Expressions $E \in \text{Exp} ::= v \mid x \mid E + E \mid E - E \mid E = E \mid E > E \mid E \wedge E \mid \neg E \mid \dots$

Program State

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

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

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

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

Commands

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

$[E]$ 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

Logical Values $a, a_1, \dots \in \text{LVal} \quad \supseteq \text{Val}$

Logical Variables $x, y, \dots \in \text{LVar}$

Logical Expressions $E \in \text{LExp} ::= a \mid \mathbf{x} \mid x \mid E + E \mid E - E \mid$
 $E = E \mid E > E \mid E \wedge E \mid \neg E \mid \dots, \text{ for } \mathbf{x} \in \text{PVar}$

Logical Environment $e \in \text{LEnv} \stackrel{\text{def}}{=} \text{LVar} \rightarrow_{\text{fin}} \text{LVal}$

Logical State $(e, s, h) \in \text{LState} \stackrel{\text{def}}{=} \text{LEnv} \times \text{Store} \times \text{Heap}$

Logical expression evaluation

$\mathcal{E}[E]_{e,s} \in \text{Val}$ 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:

$P \in \text{Assert} ::= P \wedge P \mid P \Rightarrow P \mid \text{True} \mid \text{False}$	classical connectives
$\mid E = E \mid E > E \mid E \in X \mid \dots$	Boolean assertions
$\mid \exists x. P$	existential quantification
$\mid P \star P \mid \text{emp} \mid \bigotimes_{E_1 \leq x < E_2} P$	separating connectives
$\mid E \mapsto E$	linear heap cell assertion

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

Satisfiability Relation

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

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

We write $\llbracket P \rrbracket_{e,s} \stackrel{\text{def}}{=} \{h : e, s, h \models P\}$.

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 $\star$:

$$\models P \star Q \Leftrightarrow Q \star P$$

$$\models P \star (Q \star R) \Leftrightarrow (P \star Q) \star R$$

$$\models P \star \text{emp} \Leftrightarrow P$$

$$\models (P_1 \wedge P_2) \star Q \Rightarrow (P_1 \star Q) \wedge (P_2 \star Q)$$

$$\models (P_1 \vee P_2) \star Q \Leftrightarrow (P_1 \star Q) \vee (P_2 \star Q)$$

$$\models (\exists x.P) \star Q \Leftrightarrow \exists x.(P \star Q) \text{ if no variable clash}$$

Some properties of the cell assertion:

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

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

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

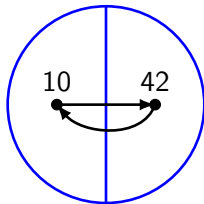
Exercise

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 $(10 \mapsto 1 \star \text{true}) \wedge (11 \mapsto 2 \star \text{true})$

Example

Assertion: $x \mapsto y \star y \mapsto x$



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

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

Example

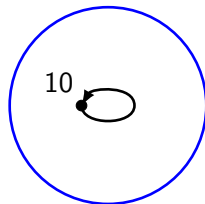
Assertion: $x \mapsto y \wedge y \mapsto x$

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

No heap satisfies this assertion.

Example

Assertion: $x \mapsto y \wedge y \mapsto x$



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

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

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

Derived Assertions and Predicate Definitions

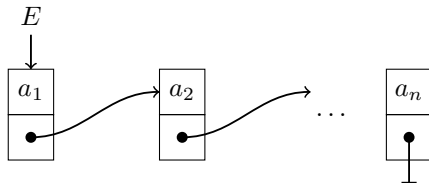
$$E \mapsto - \stackrel{\text{def}}{=} \exists x. E \mapsto x, \text{ for } x \text{ not free in } E$$

$$E \mapsto E_1, E_2 \stackrel{\text{def}}{=} (E \mapsto E_1) \star (E + 1 \mapsto E_2)$$

$$\text{list}(x) \stackrel{\text{def}}{=} (x = \text{null}) \vee (\exists z. x \mapsto -, z \star \text{list}(z))$$

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

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



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

List Predicate with Values: Properties

Exercise

$\models \text{list}(E, []) \Rightarrow$ What does that tell us about E ?

$\models \text{list}(E, a:\alpha) \Rightarrow$ What does that tell us about E ?

$\models \text{list}(\text{null}, \alpha) \Rightarrow$ What does that tell us about α ?

$\models E_1 \mapsto E_2 \star \text{list}(E_3, \alpha) \star E_3 \neq \text{null} \Rightarrow$ 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 $x + 1$.

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

```
⊢ {x = x ★ list(x, α)}  
  LLen(x) {  
    // Initialise the local variables and ensure the while condition is evaluable.  
    {x = x ★ list(x, α) ★ n, t = null}  
    {x = x ★ list(x, α) ★ n, t = null ★ (x = null) ∈ Bool}  
    if (x = null) {  
      {x = x ★ list(x, α) ★ n, t, x = null}  
      // As x = null, unfolding the list predicate yields the base case.  
      {x = x ★ (x = null ★ α = [ ]) ★ n, t = null}  
      n := 0  
      {x = x ★ (x = null ★ α = [ ]) ★ t = null ★ n = 0}  
      // Forget x, t as no longer used, connect n to α, and fold back list predicate.  
      {(x = null ★ α = [ ]) ★ n = |α|}  
      {list(x, α) ★ n = |α|}  
    } else {
```

LLen(x) Proof Sketch: recursive case

```
} else {  
  { $x = x \star \text{list}(x, \alpha) \star n, t = \text{null} \star x \neq \text{null}$ }  
  // Unfold list predicate to recursive case and simplify assertion.  
  { $\exists v, \beta, y. x \mapsto v, y \star \alpha = v:\beta \star \text{list}(y, \beta) \star n, t = \text{null}$ }  
  cons | { $x \mapsto v, y \star \text{list}(y, \beta) \star n, t = \text{null}$ }  
    t := [x + 1];  
  + | // Use the fcall rule to consume precondition and produce postcondition.  
    { $x \mapsto v, y \star \text{list}(y, \beta) \star n = \text{null} \star t = y$ }  
    n := llen(t);  
  exists, frame | { $x \mapsto v, y \star \text{list}(y, \beta) \star n = |\beta|$ }  
    n := n + 1  
    { $x \mapsto v, y \star \text{list}(y, \beta) \star n = |\beta| + 1$ }  
  // frame back assertions, add exists and fold back list predicate.  
  { $\exists v, \beta, y. x \mapsto v, y \star \alpha = v:\beta \star \text{list}(y, \beta) \star n = |\beta| + 1$ }  
  { $\text{list}(x, \alpha) \star n = |\alpha|$ }  
}
```

LLen(x) Proof Sketch: end of conditional statement

```
⊢ {x = x ★ list(x, α)}  
  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

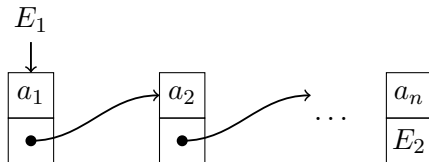
Consider an iterative list-length algorithm where $|\alpha|$ is the length of the list α

```
⊢ {x = x ★ list(x, α)}  
  LLen(x) {  
    {x = x ★ list(x, α) ★ y, n = null}  
    y := x; n := 0;  
    {∃α1, α2. ??? ★ list(y, α2) ★ α = α1 · α2 ★ n = |α1|}  
    while (y ≠ null) {  
      y := [y + 1];  
      n := n + 1;  
    }  
    {list(x, α) ★ n = |α|}  
    return n;  
  }  
  {list(x, α) ★ ret = |α|}
```

List-segment Predicate

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

Predicate $\text{lseg}(E_1, E_2, \alpha)$ 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:

$$\models \text{list}(E, \alpha) \Leftrightarrow \text{lseg}(E, \text{null}, \alpha)$$

$$\models \text{lseg}(E_1, E_2, \alpha) \star \text{lseg}(E_2, E_3, \beta) \Rightarrow \text{lseg}(E_1, E_3, \alpha \cdot \beta)$$

$$\models \text{lseg}(E_1, E_2, \alpha) \star \text{list}(E_2, \beta) \Rightarrow \text{list}(E_1, \alpha \cdot \beta)$$

$$\models \text{lseg}(E_1, E_2, \alpha) \star E_2 \mapsto a, E_3 \Rightarrow \text{lseg}(E_1, E_3, \alpha \cdot [a])$$

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

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

```
⊢ {x = x * list(x, α)}
  LLen(x) {
    {x = x * list(x, α) * y, n = null}
    y := x;
    {x = x * list(x, α) * y = x * n = null}
    // Forget x as it is no longer used
    {list(x, α) * y = x * n = null}
    n := 0;
    {list(x, α) * y = x * 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
    {∃α1, α2. lseg(x, y, α1) * list(y, α2) * α = α1 · α2 * n = |α1| * (y ≠ null ∈ Bool)}
    while (y ≠ null) {
      // Loop entry: invariant and loop condition
      {∃α1, α2. lseg(x, y, α1) * list(y, α2) * α = α1 · α2 * n = |α1| ∧ y ≠ null}
      ...
    }
  }
```

LLen(x) Proof Sketch: proving the loop body

```
while (y ≠ null) {  
  { $\exists \alpha_1, \alpha_2. \text{lseg}(x, y, \alpha_1) \star \text{list}(y, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star n = |\alpha_1| \wedge y \neq \text{null}$ }  
  // Unfold the shaded predicate to gain access to [y + 1]  
  { $\exists \alpha_1, \alpha_2, \alpha_3, a, z. \text{lseg}(x, y, \alpha_1) \star y \mapsto a, z \star \text{list}(z, \alpha_3) \star \alpha_2 = a:\alpha_3 \star \alpha = \alpha_1 \cdot \alpha_2 \star n = |\alpha_1|$ }  
  // Record previous value of y as loop body changes y  
  { $\exists \alpha_1, \alpha_3, y, a, z. y = y \star \text{lseg}(x, y, \alpha_1) \star y \mapsto a, z \star \text{list}(z, \alpha_3) \star \alpha = \alpha_1 \cdot (a:\alpha_3) \star n = |\alpha_1|$ }  
  // Isolate the resource of the loop  
  co | { $y = y \star y \mapsto a, z \star n = |\alpha_1|$ }  
  + | y := [y + 1];  
  fr | { $y = z \star y \mapsto a, z \star n = |\alpha_1|$ }  
  + | n := n + 1;  
  ex | { $y = z \star y \mapsto a, z \star n = |\alpha_1| + 1$ }  
  // Bring back the existentials and frame  
  { $\exists \alpha_1, \alpha_3, y, a, z. y = z \star y \mapsto a, z \star n = |\alpha_1| + 1 \star \text{lseg}(x, y, \alpha_1) \star \text{list}(z, \alpha_3) \star \alpha = \alpha_1 \cdot (a:\alpha_3)$ }
```

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

```
// Bring back the existentials and frame
{ $\exists \alpha_1, \alpha_3, y, a, z. y = z \star y \mapsto a, z \star n = |\alpha_1| + 1 \star \text{lseg}(x, y, \alpha_1) \star \text{list}(z, \alpha_3) \star \alpha = \alpha_1 \cdot (a:\alpha_3)$ }
// Re-establish the loop invariant
{ $\exists \alpha_1, \alpha_3, y, a. \text{lseg}(x, y, \alpha_1) \star y \mapsto a, y \star \text{list}(y, \alpha_3) \star \alpha = \alpha_1 \cdot (a:\alpha_3) \star n = |\alpha_1| + 1$ }
// Apply lemma: (shaded) list segment  $\star$  node at end  $\rightarrow$  extended list segment
{ $\exists \alpha_1, \alpha_3, y, a. \text{lseg}(x, y, \alpha_1 \cdot [a]) \star \text{list}(y, \alpha_3) \star \alpha = (\alpha_1 \cdot [a]) \cdot \alpha_3 \star n = |\alpha_1 \cdot [a]|$ }
{ $\exists \alpha_1, \alpha_2. \text{lseg}(x, y, \alpha_1) \star \text{list}(y, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star n = |\alpha_1| \star (y \neq \text{null} \in \text{Bool})$ }
}
{ $(\exists \alpha_1, \alpha_2. \text{lseg}(x, y, \alpha_1) \star \text{list}(y, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star n = |\alpha_1|) \wedge y = \text{null}$ }
{ $\exists \alpha_1, \alpha_2. \text{lseg}(x, \text{null}, \alpha_1) \star \text{list}(\text{null}, \alpha_2) \star \alpha = \alpha_1 \cdot \alpha_2 \star n = |\alpha_1|$ }
// Apply lemma: (shaded) list segment ending with null is a list
{ $\exists \alpha_1, \alpha_2. \text{list}(x, \alpha_1) \star \alpha_2 = [] \star \alpha = \alpha_1 \cdot \alpha_2 \star n = |\alpha_1|$ }
{ $\text{list}(x, \alpha) \star n = |\alpha|$ }
return n;
}
// Assign return value
{ $\text{list}(x, \alpha) \star n = |\alpha| \star \text{ret} = n$ }
{ $\text{list}(x, \alpha) \star \text{ret} = |\alpha|$ }
```

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



José Frago 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 [Löow 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

```

776 PPS/Name substitution:
777 NFN annotations: No
778
779 if (c_1 == [ [ char, noFset ] ] )
780 if (c_2 == [ [ char, noFset ] ] )
781 if (c_3 == [ char ] )
782 if (c_4 == [ noFset ] )
783
784 if (c_5 == [ noFset ] )
785
786 if (c_6 == [ noFset ] )
787
788
789 Analyzing list structure.
790 Analyzing list structure.
791 PPS/Name substitution completed:
792
793 PPS:
794
795 if (c_1 == [ [ char, noFset ] ] )
796 if (c_2 == [ [ char, noFset ] ] )
797 if (c_3 == [ char ] )
798 if (c_4 == [ noFset ] )
799
800 if (c_5 == [ noFset ] )
801
802 if (c_6 == [ noFset ] )
803
804
805 if (c_1 == [ [ char, noFset ] ] )
806 if (c_2 == [ [ char, noFset ] ] )
807 if (c_3 == [ char ] )
808 if (c_4 == [ noFset ] )
809
810 if (c_5 == [ noFset ] )
811
812 if (c_6 == [ noFset ] )
813
814
815 Analyzing list structure.
816 Analyzing list structure.
817 PPS/Name substitution completed:
818
819 PPS:

```

```

1001 list(xw, alpha)
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
18
```

```

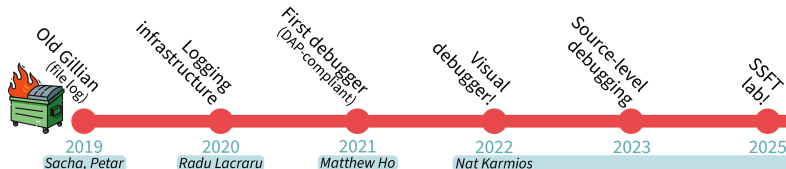
3080 TYPING ENVIRONMENT:
3081   (lambda: list)
3082   (x: list)
3083     (x, _var_5: list)
3084     (x, _var_6: list)
3085     (x, _var_7: list)
3086
3087 PPS/Gamma simplification:
3088 With unification: No
3089
3090 (x = if {x}_1, _var_7: list)
3091 (lambda = lambda (x) {({x}_0, _var_8: list), _var_9: list})
3092   (x = if {x}_1, _var_7: list)
3093   (x) (x) (x) (x) (x) (x)
3094   (x) (x) (x) (x) (x) (x)
3095   (x) (x) (x) (x) (x) (x)
3096
3097 Gamma:
3098   (lambda: list)
3099   (x: list)
3100   (x, _var_10: list)
3101   (x, _var_11: list)
3102   (x, _var_12: list)
3103
3104 Analysing list structure.
3105 PPS/Gamma simplification completed:
3106
3107 PPS:
3108 (x = if {x}_1, _var_7: list)
3109 (lambda = lambda (x) {({x}_0, _var_8: list), _var_9: list})
3110   (x = if {x}_1, _var_7: list)
3111   (x) (x) (x) (x) (x) (x)
3112   (x) (x) (x) (x) (x) (x)
3113
3114 Gamma:
3115   (lambda: list)
3116   (x: list)
3117   (x, _var_13: list)
3118   (x, _var_14: list)
3119
3120

```

```

7763 unify @P: There are k states left to consider.
7764 subtext:
7765   [alpha]
7766   [alpha]
7767   []
7768 unify assertion: (ret = (1-len alpha)),
7769 subtext:
7770   [ ret: (x), (alpha: alpha), (x: x) ]
7771 STATE:
7772   SPAC (x:0: alpha, x:
7773     STATE:
7774       (ret: list x)
7775     )
7776 MEMORY:
7777   [ ]
7778   [ ]
7779 PURE FORMULAE:
7780   (x = null)
7781   (alpha = [ ])
7782   (0 <= X (1-len alpha))
7783   [ ]
7784 TYPING ENVIRONMENT:
7785   (alpha: list)
7786   (x: null)
7787 PREDS:
7788   [ ]
7789 NAMES:
7790   [ ]
7791 VARIANTS:
7792   list,
7793   list,
7794   list,
7795   list
7796 Preparing entailment:
7797 existential:
7798   [ ]
7799 left: (x = null)
7800   (alpha = [ ])
7801   (0 <= X (1-len alpha))
7802 right: (0 = (1-len alpha))
7803 Ques:

```



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