Compositional Symbolic Execution

Compositional Symbolic Execution in a nutshell

- Underpins tools based on ideas from separation logic: e.g. Verifast, Viper (one backend), Gillian, CN, Infer-Pulse
- Successful in both academia and industry
- Symbolic execution of **partial** symbolic state; partial symbolic heap manipulated using **consume** and **produce** operations
- First-order path constraints checked with SMT solvers (we use z3)
- **Compositionality**: possible to analyse a function separately from the rest of the codebase and record the results in function specifications in a separation logic

Compositional Symbolic Execution for Correctness and Incorrectness Reasoning

Andreas Lööw
Imperial College London, UK
Daniele Nantes-Sobrinho
Imperial College London, UK
Sacha-Élie Ayoun
Imperial College London, UK
Caroline Cronjäger
Ruhr-Universität Bochum, Germany
Petar Maksimović
Imperial College London, UK
Runtime Verification Inc., Chicago, IL, USA
Philippa Gardner
Imperial College London, UK

Artifacts Available ECOOP V1.1

Artifacts Evaluated ECOOP Reusable V1.1

distinguished paper

Abstract

The introduction of separation logic has led to the development of symbolic execution techniques and tools that are (functionally) compositional with function specifications that can be used in broader calling contexts. Many of the compositional symbolic execution tools developed in academia and industry have been grounded on a formal foundation, but either the function specifications are not validated with respect to the underlying separation logic of the theory, or there is a large gulf between the theory and the implementation of the tool.

We introduce a formal compositional symbolic execution engine which creates and uses function specifications from an underlying separation logic and provides a sound theoretical foundation for, and indeed was partially inspired by, the Gillian symbolic execution platform. This is achieved by providing an axiomatic interface which describes the properties of the consume and produce operations used in the engine to update compositionally the symbolic state, for example when calling function specifications. This consume-produce technique is used by VeriFast, Viper, and Gillian, but has not been previously characterised independently of the tool. As part of our result, we give consume and produce operations inspired by the Gillian implementation that satisfy the properties described by our axiomatic interface. A surprising property is that our engine semantics provides a common foundation for both correctness and incorrectness reasoning, with the difference in the underlying engine only amounting to the choice to use satisfiability or validity. We use this property to extend the Gillian platform, which previously only supported correctness reasoning, with incorrectness reasoning and automatic true bug-finding using incorrectness bi-abduction. We evaluate our new Gillian platform by using the Gillian instantiation to C. This instantiation is the first tool grounded on a common formal compositional symbolic execution engine to support both correctness and incorrectness reasoning.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Program analysis; Theory of computation → Separation logic; Theory of computation → Automated reasoning

Keywords and phrases

Theory and practice of CSE

We are interested in both the theory and practice of CSE.

- **Theory**: we have developed the foundations of CSE theory, mechanised in Rocq, that is independent of a particular tool development.

- **Practice**: we develop and maintain the Gillian symbolic execution platform.

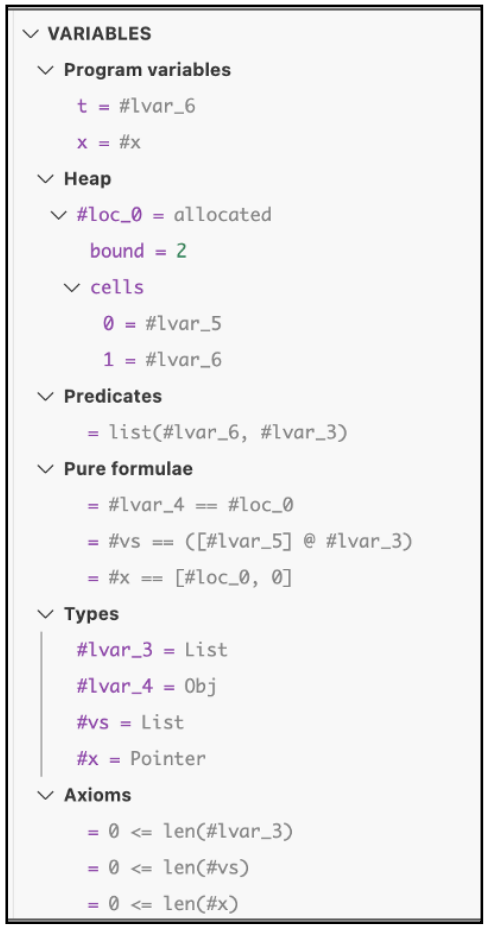
- Both feed into each other and inform each other!

Our CSE Theory

We differ from previous CSE theories in multiple ways.

1. **Foundational theory** independent of any tool
2. **Parametric** on the memory model
3. **Unified foundation** for compositional verification and true bug detection
4. **Standard logical definitions** used to specify functions (e.g. separation logic and incorrectness separation logic)

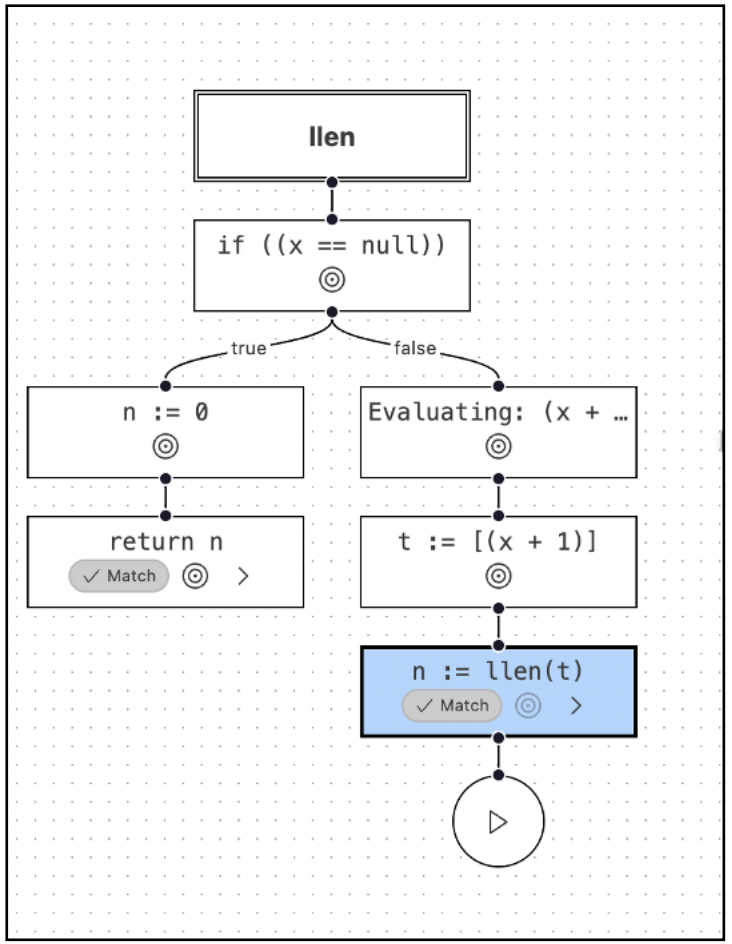
CSE Components in Gillian



Symbolic State

```
13
14  predicate list(+x, vs) {
15      (x == null) * (vs == nil);
16      (x -b> #v, #z) * list(#z, #vss) * (vs == #v::#vss)
17  }
18
19  // === EDIT BELOW HERE ===
20
21  { (x == #x) * list(#x, #vs) }
    Verify llen
22  function llen(x) {
23      if (x == null) {
24          n := 0
25      } else {
26          t := [x + 1];
27          n := llen(t);
28          n := n + 1
29      };
30      return n
31  }
32  { list(#x, #vs) * (ret == len(#vs)) }
33
```

Program Specification



Symbolic Execution

Symbolic state:
$\hat{\sigma} = (\hat{s}, \hat{\mu}, \widehat{\wp}, \hat{\pi})$

VARIABLES

Program variables
- t = #lvar_6
- x = #x

Heap
- #loc_0 = allocated
  - bound = 2
  - cells
    - 0 = #lvar_5
    - 1 = #lvar_6

Predicates
- = list(#lvar_6, #lvar_3)

Pure formulae
- = #lvar_4 == #loc_0
- = #vs == ([#lvar_5] @ #lvar_3)
- = #x == [#loc_0, 0]

Types
- #lvar_3 = List
- #lvar_4 = Obj
- #vs = List
- #x = Pointer

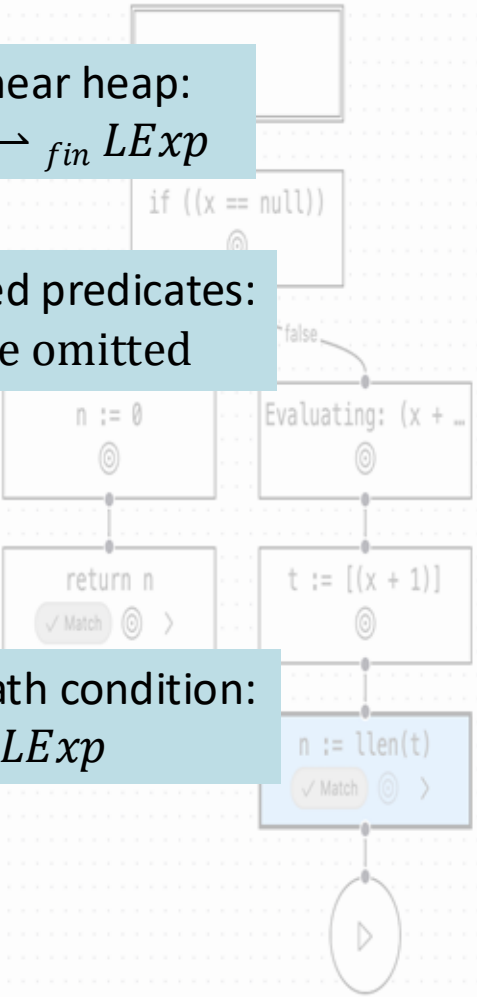
Axioms
- = 0 <= len(#lvar_3)
- = 0 <= len(#vs)
- = 0 <= len(#x)

Symbolic store:
$\hat{s} : PVar \rightharpoonup_{fin} LExp$

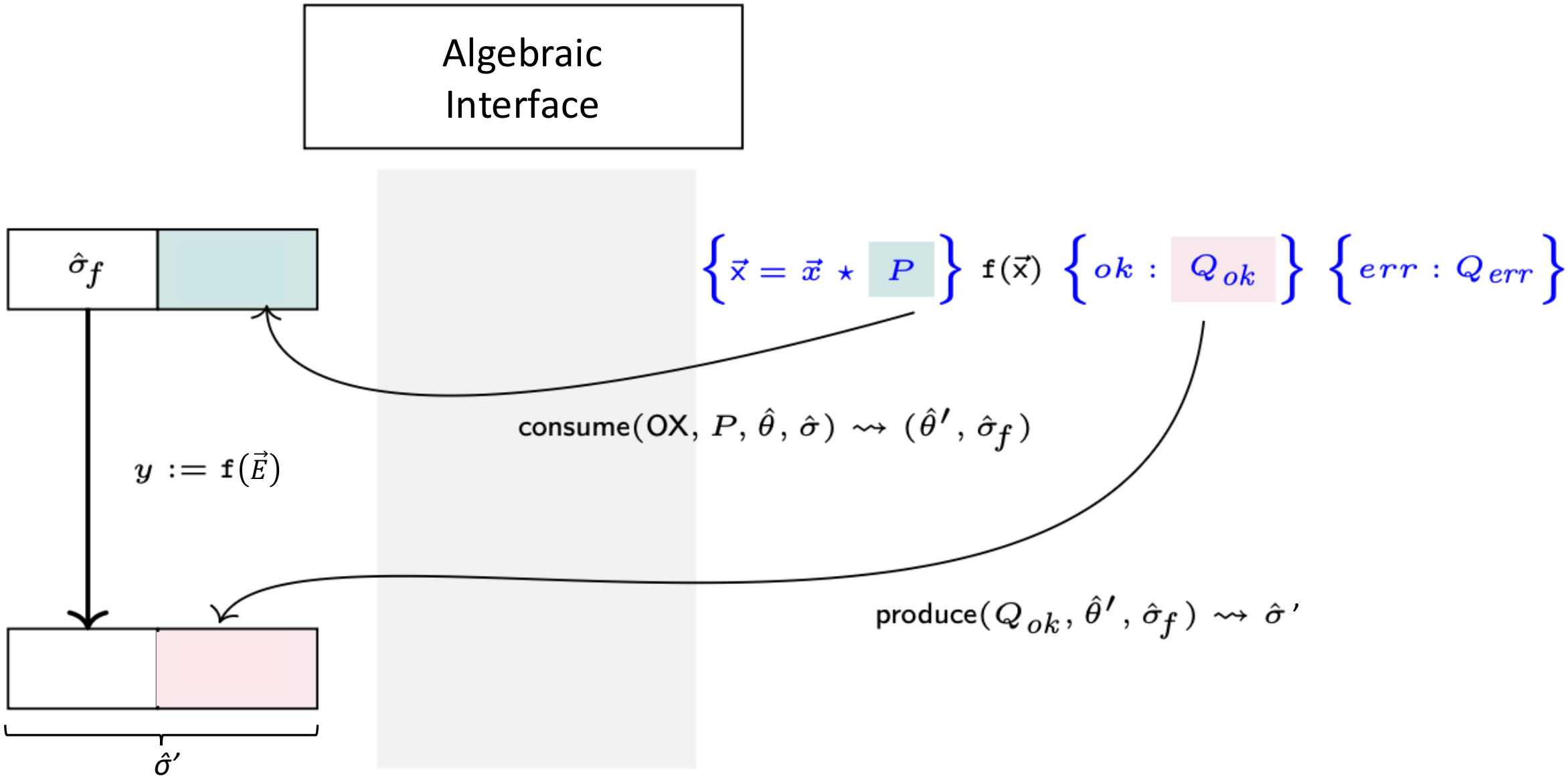
Symbolic linear heap:
$\hat{\mu} : LExp \rightharpoonup_{fin} LExp$

User-defined predicates:
$\widehat{\wp}$: type omitted

Symbolic path condition:
$\hat{\pi} : LExp$



Compositional Symbolic Execution: Function Compositionality



CSE for $LLen$ – Recursive case

$\{\boldsymbol{x} = x \ * list(x, \alpha)\}$
$LLen(\boldsymbol{x})\ \{$
  $if\,(\boldsymbol{x} = null)\{$
    $\boldsymbol{n} \coloneqq 0;$
  $\}\ else\ \{$
    $\boldsymbol{t} \coloneqq [\boldsymbol{x} + 1];$
    $\boldsymbol{n} \coloneqq LLen(\boldsymbol{t});$
    $\boldsymbol{n} \coloneqq \boldsymbol{n} + 1;$
  $\}$
  $return\ \boldsymbol{n};$
$\}$
$\{ok:\ list(x, \alpha) * \boldsymbol{ret} = |\alpha|\}$

Approach: to verify this function specification using compositional symbolic execution

Produce (i.e. assume) initial symbolic state from assertion to start symbolic execution

For function call, **consume** precondition and then **produce** postcondition

Consume (i.e. assert) final symbolic state and check it matches specification in the assertion

Produce the initial symbolic state given $list(x, \alpha)$ in the precondition

Initial symbolic state also initialises parameter variable ($\boldsymbol{x}$) and variables in the function body ($\boldsymbol{n}, \boldsymbol{t}$)

$\{\boldsymbol{x} = x * list(x, \alpha)\}$
$LLen(\boldsymbol{x})\{$
$// \ \widehat{\sigma_1} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], true) : produce$
$if(\boldsymbol{x} = null)\{$
$// \ \widehat{\sigma_2} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null) :$ if branch
$// \ \widehat{\sigma_3} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, \emptyset, \hat{x} = null \wedge \hat{\alpha} = []) :$ unfolding list predicate
$\boldsymbol{n} := 0;$
$// \ \widehat{\sigma_4} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, \emptyset$ $ull \wedge \hat{\alpha} = []) :$ assignment
$// \ \widehat{\sigma_5} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}$ $], \hat{x} = null \wedge \hat{\alpha} = []) :$ folding list predicate
$\}\ else\ \{$
$\ldots$

Analogous to intermediate assertions in proof sketches

$\{\boldsymbol{x} = x * \boldsymbol{n}, \boldsymbol{t} = null * list(x, \alpha) * x = null\}$

$\{\boldsymbol{x} = x \ * list(x, \alpha)\}$

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

// $\widehat{\sigma_1} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], true)$: produce

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

// $\widehat{\sigma_2} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null)$: if branch

// $\widehat{\sigma_3} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, \emptyset, \hat{x} = null \wedge \hat{\alpha} = [])$: unfolding list predicate

$\boldsymbol{n} := 0;$

// $\widehat{\sigma_4} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, \emptyset, \hat{x} = null \wedge \hat{\alpha} = [])$: assignment

// $\widehat{\sigma_5} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null \wedge \hat{\alpha} = [])$: folding list predicate

$\}\ else\ \{$

...

} *else* {

// $\widehat{\sigma_6} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null)$: else branch

// $\widehat{\sigma_7} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta})$: unfold list

$\boldsymbol{t} \coloneqq [\boldsymbol{x} + 1];$

// $\widehat{\sigma_8} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta})$: lookup

$\boldsymbol{n} \coloneqq LLen(\boldsymbol{t});$

// $\widehat{\sigma_9} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}|, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta})$: fcall

$\boldsymbol{n} \coloneqq \boldsymbol{n} + 1;$

// $\widehat{\sigma_{10}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta})$: assign

// $\widehat{\sigma_{11}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta})$: fold list

}

return $\boldsymbol{n}$;

// note: we now have two branches, so we must consider changes in both symbolic states

/* $\widehat{\sigma_{5.1}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null, \boldsymbol{ret} \mapsto 0], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null \wedge \hat{\alpha} = [])$

$\widehat{\sigma_{11.1}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}, \boldsymbol{ret} \mapsto |\hat{\beta}| + 1], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta})$ */

}

$\{ok: list(x, \alpha) * \boldsymbol{ret} = |\alpha|\}$

Consume part of $\widehat{\sigma_8}$ that matches the precondition when LLen is given $\boldsymbol{t}$ as input

Results in the following intermediate symbolic state:

$$\widehat{\sigma'} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \emptyset, \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$$

} *else* {

// ...fold list

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

// $\widehat{\sigma_8} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$

$\boldsymbol{n} := LLen(\boldsymbol{t});$

// $\widehat{\sigma_9} = \left([\boldsymbol{x} \mapsto \hat{x}, n \mapsto |\hat{\beta}|, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$

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

// ... $\mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], [list(\hat{y}, \hat{\beta})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$: assign

... $= \hat{v} :: \hat{\beta}\right)$: fold list

Produce postcondition given $\widehat{\sigma'}$ ($\boldsymbol{ret}$ maps to $\boldsymbol{n}$), which yields $\widehat{\sigma_9}$

// note: we now have two branches, so we must consider changes in both symbolic states

/* $\widehat{\sigma_{5.1}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null, \boldsymbol{ret} \mapsto 0], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null \wedge \hat{\alpha} = [])$

$\widehat{\sigma_{11.1}} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}, \boldsymbol{ret} \mapsto |\hat{\beta}| + 1], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$

}

{ok: $list(x, \alpha) * \boldsymbol{ret} = |\alpha|$}

Recall:

$\{\boldsymbol{x} = x * list(x, \alpha)\}$

$LLen(\boldsymbol{x})\ \{\ldots\}$

$\{list(x, \alpha) * \boldsymbol{ret} = |\alpha|\}$

} *else* {

// $\widehat{\sigma_6} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null)$: else branch

// $\widehat{\sigma_7} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$: unfold list

$\boldsymbol{t} \coloneqq [\boldsymbol{x} + 1]$;

// $\widehat{\sigma_8} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$: lookup

$\boldsymbol{n} \coloneqq LLen(\boldsymbol{t})$;

// $\widehat{\sigma_9} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}|, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$: fcall

$\boldsymbol{n} \coloneqq \boldsymbol{n} + 1$;

// $\widehat{\sigma_{10}} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$: assign

// $\widehat{\sigma_{11}} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$: fold list

}

return $\boldsymbol{n}$;

// note: we now have two branches, so we must consider changes in both symbolic states

/* $\widehat{\sigma_{5.1}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null, \boldsymbol{ret} \mapsto 0], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null \wedge \hat{\alpha} = [])$

$\widehat{\sigma_{11.1}} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}, \boldsymbol{ret} \mapsto |\hat{\beta}| + 1], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right)$ */

}

$\{ok: list(x, \alpha) * \boldsymbol{ret} = |\alpha|\}$

$\}\ else\ \{$

$// \ \widehat{\sigma_6} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null) : \text{else branch}$

$// \ \widehat{\sigma_7} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto null], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right) : \text{unfold list}$

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

$// \ \widehat{\sigma_8} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto null, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right) : \text{lookup}$

$\boldsymbol{n} := LLen(\boldsymbol{t});$

$// \ \widehat{\sigma_9} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}|, \boldsymbol{t} \mapsto \hat{y}], [\hat{x} \mapsto \hat{v}, \hat{x}+1 \mapsto \hat{y}], \left[list\left(\hat{y}, \hat{\beta}\right)\right], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right) : \text{fcall}$

r

sign

$/$

$\}$

$return\ \boldsymbol{n};$

$/* \ \widehat{\sigma_{5.1}} = ([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto 0, \boldsymbol{t} \mapsto null, \boldsymbol{ret} \mapsto 0], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} = null \wedge \hat{\alpha} = [])$

$\widehat{\sigma_{11.1}} = \left([\boldsymbol{x} \mapsto \hat{x}, \boldsymbol{n} \mapsto |\hat{\beta}| + 1, \boldsymbol{t} \mapsto \hat{y}, \boldsymbol{ret} \mapsto |\hat{\beta}| + 1], \widehat{\mu_\emptyset}, [list(\hat{x}, \hat{\alpha})], \hat{x} \neq null \wedge \hat{\alpha} = \hat{v} :: \hat{\beta}\right) \ */$

$\}$

$\{ok: list(x, \alpha) * \boldsymbol{ret} = |\alpha|\}$

Consume postcondition from $\widehat{\sigma_{5.1}}$ and check that the result satisfies the empty assertion (i.e. $heap = \widehat{\mu_\emptyset}$ and $pred = \emptyset$)

Consume postcondition from $\widehat{\sigma_{11.1}}$ and check that the result satisfies the empty assertion (i.e. $heap = \widehat{\mu_\emptyset}$ and $pred = \emptyset$)

CSE Parametric on Memory Model

Compositional Symbolic Execution

$$\textsc{Skip}\quad \frac{}{\hat{\sigma}, \texttt{skip} \Downarrow^m_\Gamma ok : \hat{\sigma}}$$

$$\textsc{Assign}\quad \frac{[\![\mathsf{E}]\!]_{\hat{s}} = \hat{E} \qquad \hat{s}' = \hat{s}[\mathsf{x} \mapsto \hat{E}] \qquad \hat{\pi}' = (\hat{s}(\mathsf{x}) \in \mathsf{Val} \wedge \hat{\pi})}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \mathsf{x} := \mathsf{E} \Downarrow^m_\Gamma ok : (\hat{s}', \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}')}$$

$$\textsc{Assign-Err-Eval}\quad \frac{[\![\mathsf{E}]\!]_{\hat{s}} = \hat{E} \qquad \hat{s}' = \hat{s}[\mathsf{err} \mapsto \text{“ExprEval”}] \qquad \hat{\pi}' = (\hat{E} \notin \mathsf{Val} \wedge \hat{\pi})}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \mathsf{x} := \mathsf{E} \Downarrow^m_\Gamma err : (\hat{s}', \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}')}$$

$$\textsc{If-Then}\quad \frac{[\![\mathsf{E}]\!]_{\hat{s}} = \hat{E} \qquad \hat{\pi}' = (\hat{E} = \mathsf{true} \wedge \hat{\pi}) \qquad (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}'), \mathsf{C}_1 \Downarrow^m_\Gamma o : \hat{\sigma}'}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \texttt{if } (\mathsf{E}) \ \mathsf{C}_1 \ \texttt{else } \mathsf{C}_2 \Downarrow^m_\Gamma o : \hat{\sigma}'}$$

$$\textsc{If-Else}\quad \frac{[\![\mathsf{E}]\!]_{\hat{s}} = \hat{E} \qquad \hat{\pi}' = (\hat{E} = \mathsf{false} \wedge \hat{\pi}) \qquad (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}'), \mathsf{C}_2 \Downarrow^m_\Gamma o : \hat{\sigma}'}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \texttt{if } (\mathsf{E}) \ \mathsf{C}_1 \ \texttt{else } \mathsf{C}_2 \Downarrow^m_\Gamma o : \hat{\sigma}'}$$

$$\textsc{If-Err-Eval}\quad \frac{[\![\mathsf{E}]\!]_{\hat{s}} = \hat{E} \qquad \hat{\pi}' = (\hat{E} \notin \mathsf{Val} \wedge \hat{\pi}) \qquad \hat{s}' = \hat{s}[\mathsf{err} \mapsto \text{“ExprEval”}]}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \texttt{if } (\mathsf{E}) \ \mathsf{C}_1 \ \texttt{else } \mathsf{C}_2 \Downarrow^m_\Gamma err : (\hat{s}', \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}')}$$

$$\textsc{If-Err-Type}\quad \frac{[\![\mathsf{E}]\!]_{\hat{s}} = \hat{E} \qquad \hat{\pi}' = (\hat{E} \notin \mathsf{Bool} \wedge \hat{E} \in \mathsf{Val} \wedge \hat{\pi}) \qquad \hat{s}' = \hat{s}[\mathsf{err} \mapsto \text{“Type”}]}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \texttt{if } (\mathsf{E}) \ \mathsf{C}_1 \ \texttt{else } \mathsf{C}_2 \Downarrow^m_\Gamma err : (\hat{s}', \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}')}$$

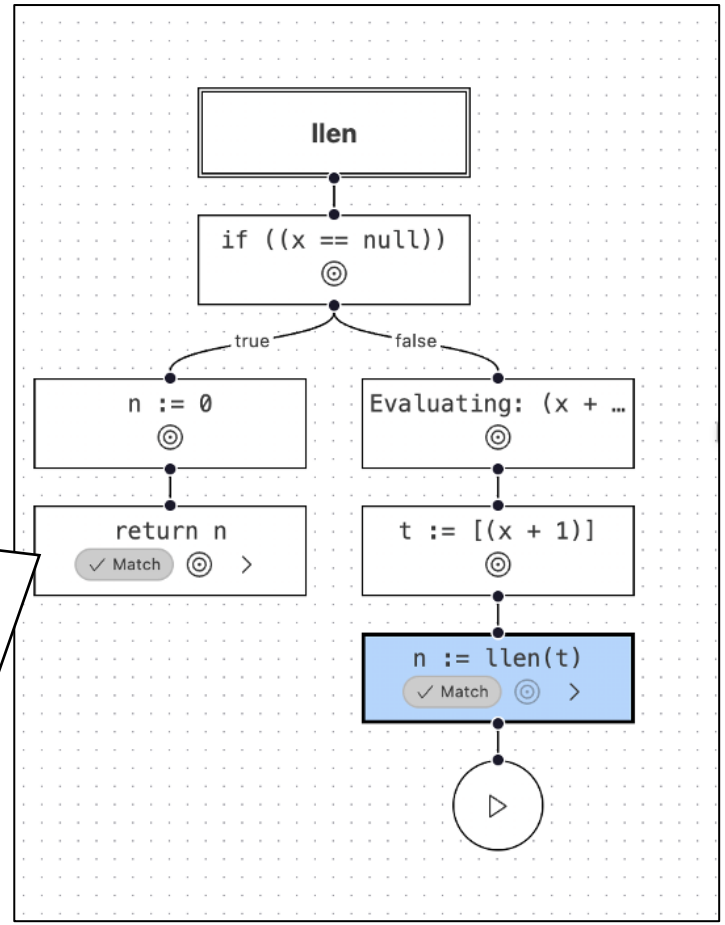
$$\textsc{Seq}\quad \frac{\hat{\sigma}, \mathsf{C}_1 \Downarrow^m_\Gamma ok : \hat{\sigma}' \qquad \hat{\sigma}', \mathsf{C}_2 \Downarrow^m_\Gamma o : \hat{\sigma}''}{\hat{\sigma}, \mathsf{C}_1; \mathsf{C}_2 \Downarrow^m_\Gamma o : \hat{\sigma}''}$$

$$\textsc{Seq-Err}\quad \frac{\hat{\sigma}, \mathsf{C}_1 \Downarrow^m_\Gamma o : \hat{\sigma}' \qquad o \neq ok}{\hat{\sigma}, \mathsf{C}_1; \mathsf{C}_2 \Downarrow^m_\Gamma o : \hat{\sigma}'}$$

$$\textsc{Fold}\quad \frac{\ldots}{\hat{\sigma}, \texttt{fold } p(\vec{E}) \Downarrow^m_\Gamma ok : \hat{\sigma}'}$$

$$\textsc{Unfold}\quad \frac{\ldots}{\hat{\sigma}, \texttt{unfold } p(\vec{E}) \Downarrow^m_\Gamma ok : \hat{\sigma}'}$$

$$\textsc{Lookup}\quad \frac{\hat{\mu}(\hat{E}'_l) = (\hat{\mu}_b, \_) \qquad \hat{\mu}_b(\hat{E}'_o) = \hat{E} \qquad \hat{\pi}' = (\hat{E}'_l = \hat{E}_l \wedge \hat{E}'_o = \hat{E}_o)}{\hat{\mu}.\texttt{lookup}([\hat{E}_l, \hat{E}_o]) \rightsquigarrow ok : (\hat{\mu}, \hat{\pi}', [\hat{E}])}$$



Memory Models in Program Reasoning

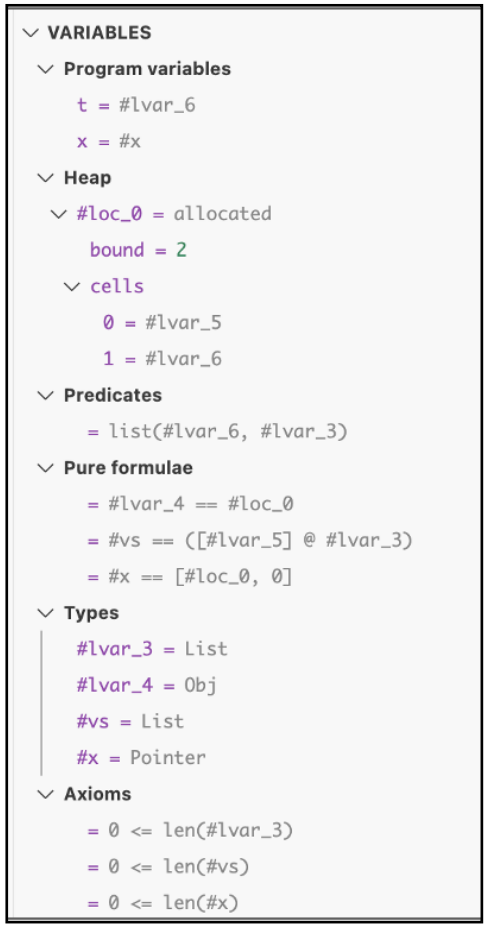
Memory model = Memory structure + Read and write actions

Memory Models in Program Reasoning

Requirement : Different applications require different memory models

- 1. Different memory models for different languages
  - E.g.: C = bytes, raw pointers, etc.
  - E.g.: JavaScript = objects, prototype chains, etc.
- 2. Different types of analyses require different types of ghost state
  - E.g.: exclusive ownership vs. fractional ownership
- 3. Other different requirements
  - E.g.: performance, implementation effort, automation support, etc.

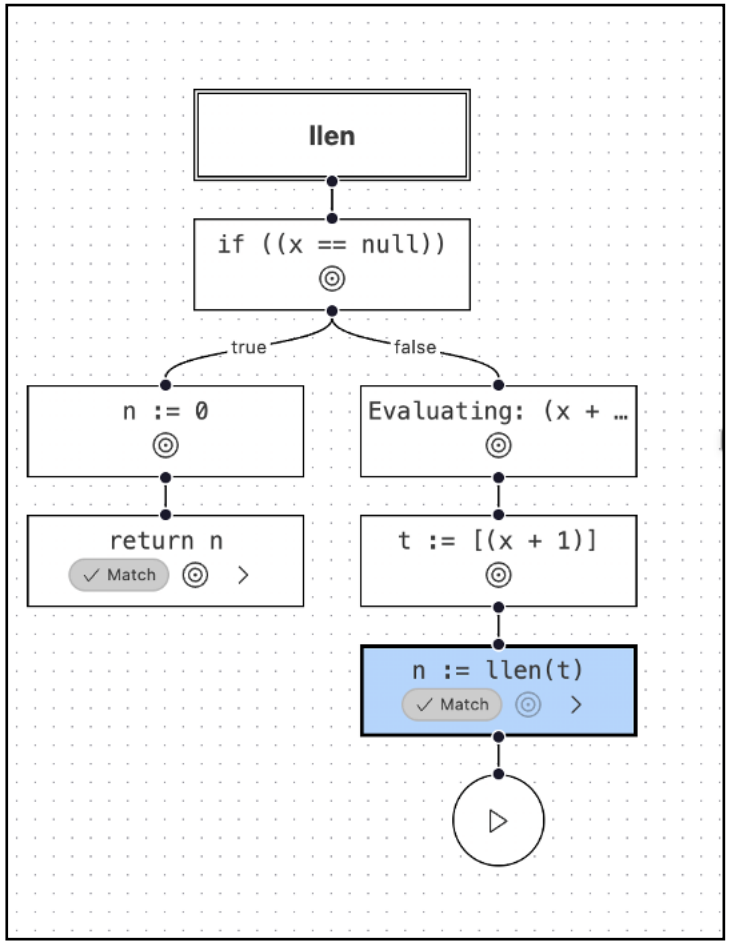
Gillian-While (WiSL)



Symbolic State

```
13
14  predicate list(+x, vs) {
15      (x == null) * (vs == nil);
16      (x -b> #v, #z) * list(#z, #vss) * (vs == #v::#vss)
17  }
18
19  // === EDIT BELOW HERE ===
20
21  { (x == #x) * list(#x, #vs) }
    Verify llen
22  function llen(x) {
23      if (x == null) {
24          n := 0
25      } else {
26          t := [x + 1];
27          n := llen(t);
28          n := n + 1
29      };
30      return n
31  }
32  { list(#x, #vs) * (ret == len(#vs)) }
33
```

Program Specification



Symbolic Execution

Symbolic state:
$\hat{\sigma} = (\hat{s}, \hat{\mu}, \widehat{\wp}, \hat{\pi})$

VARIABLES

Program variables

t = #lvar_6

x = #x

Heap

#loc_0 = allocated

bound = 2

cells

0 = #lvar_5

1 = #lvar_6

Predicates

= list(#lvar_6, #lvar_3)

Pure formulae

= #lvar_4 == #loc_0

= #vs == ([#lvar_5] @ #lvar_3)

= #x == [#loc_0, 0]

Types

#lvar_3 = List

#lvar_4 = Obj

#vs = List

#x = Pointer

Axioms

= 0 <= len(#lvar_3)

= 0 <= len(#vs)

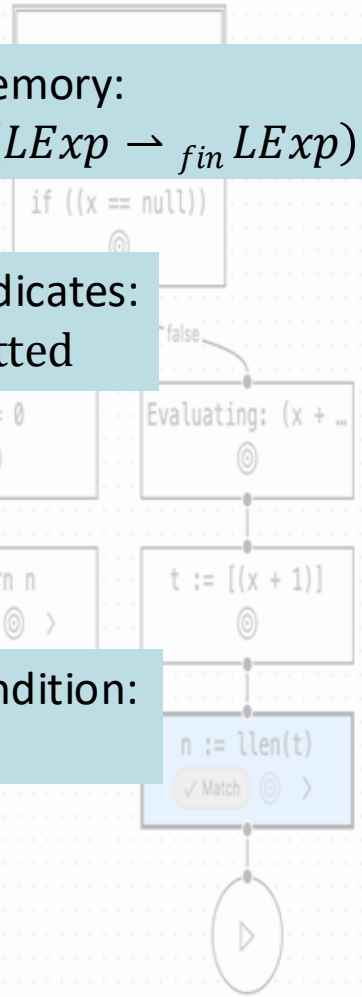
= 0 <= len(#x)

Symbolic store:
$\hat{s} : PVar \rightharpoonup_{fin} LExp$

Symbolic WiSL memory:
$\hat{\mu} : LExp \rightharpoonup_{fin} ((LExp \rightharpoonup_{fin} LExp) \times LExp^{?})_{\emptyset}$

User-defined predicates:
$\widehat{\wp}$: type omitted

Symbolic path condition:
$\hat{\pi} : LExp$



Example CSE Instantiation: WiSL

- Concrete partial memory model:
$$\mu : Loc \rightharpoonup_{fin} ((Nat \rightharpoonup_{fin} Val) \times Nat^{?})_{\emptyset}$$
- Well-formedness condition
(e.g. each cell's offset is less than the block's bound)
- Resource assertions: $E \mapsto E \mid bound(E, E) \mid E \mapsto \emptyset$
- Symbolic partial memory model:
$$\hat{\mu} : LExp \rightharpoonup_{fin} \left((LExp \rightharpoonup_{fin} LExp) \times LExp^{?}\right)_{\emptyset}$$
- $\text{consume}_{\text{Res}}$ and $\text{produce}_{\text{Res}}$
(consuming & producing resource assertions)

Memory Soundness

$\theta : LVar \rightharpoonup_{fin} LVal$ is the logical environment

$$\theta, (s, \mu) \models (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}) \stackrel{\text{def}}{=} \exists \mu_1, \mu_2.$$

$$\mu = \mu_1 \cdot \mu_2$$

In the general definition of partial memory, memory composition is a requirement of the instance.

$$\text{and } \theta, s \models_{\text{Sto}} \hat{s}$$

$$\text{and } \theta, \mu_1 \models_{\text{Mem}} \hat{\mu}$$

$$\text{and } \theta, \mu_2 \models_{\text{Pred}} \hat{\mathcal{P}}$$

Similarly, the satisfaction relations for memory and predicates are a requirement of the instance.

$$\text{and } [\![\hat{\pi}]\!]_\theta = \text{true}$$

WISL actions

WISL Actions

WISL memory actions — lookup, mutate, allocation and deallocation — are particular instances of general CSE actions.

$$\textsc{Lookup}\quad \frac{\hat{\mu}(\hat{E}'_l) = (\hat{\mu}_b, \_) \qquad \hat{\mu}_b(\hat{E}'_o) = \hat{E} \qquad \hat{\pi}' = (\hat{E}'_l = \hat{E}_l \wedge \hat{E}'_o = \hat{E}_o)}{\hat{\mu}.\texttt{lookup}([\hat{E}_l, \hat{E}_o]) \rightsquigarrow ok : (\hat{\mu}, \hat{\pi}', [\hat{E}])}$$

Frame Properties for Memory Actions

OX frame — *"If memory is removed by an action, (non-missing) behaviour is unaffected"*

$$\begin{array}{ll} \text{If} & (\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o : (\mu', \vec{v}') \\ \text{then} & \exists \mu'', \vec{v}'', o'.\ \mu.\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o' : (\mu'', \vec{v}'') \text{ and} \\ & \quad (o' \neq miss \Rightarrow (o' = o \text{ and } \vec{v}'' = \vec{v}' \text{ and } \mu' = \mu'' \cdot \mu_f)) \end{array}$$

UX frame — *"If memory is added by an action, (non-missing) behaviour is unaffected"*

$$\begin{array}{ll} \text{If} & \mu.\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o : (\mu', \vec{v}') \text{ and } o \neq miss \text{ and } \mu' \cdot \mu_f \text{ is defined} \\ \text{then} & (\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o : (\mu' \cdot \mu_f, \vec{v}') \end{array}$$

Result

The WiSL memory model satisfies the OX and UX frame properties.

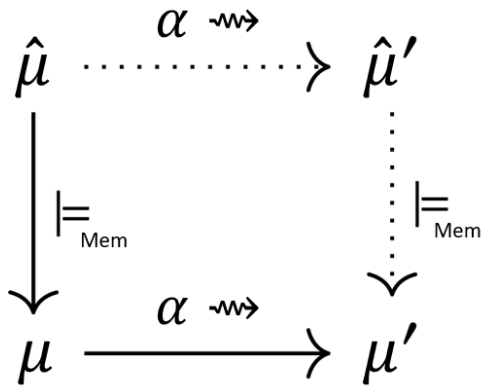
OX and UX Soundness for actions

OX soundness

All *concrete* actions can be simulated by a corresponding *symbolic* action

If $\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$ and $\theta, \mu \models_{\text{Mem}} \hat{\mu}$ and $[\![\vec{\hat{E}}]\!]_\theta = \vec{v}$ and
$(\forall o, \hat{\mu}', \hat{\pi}', \vec{\hat{E}}'.\ \hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}')$ and $[\![\hat{\pi}']\!]_\theta = \text{true}$ and $[\![\vec{\hat{E}}']\!]_\theta \neq \lightning \Rightarrow o \neq abort)$

then $\exists \hat{\mu}', \hat{\pi}', \vec{\hat{E}}', \theta'.\ \hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}')$ and $\theta'|_{\text{lv}(\hat{\mu})} = \theta$
and $[\![\hat{\pi}']\!]_{\theta'} = \text{true}$ and $\theta', \mu' \models_{\text{Mem}} \hat{\mu}'$ and $[\![\vec{\hat{E}}']\!]_{\theta'} = \vec{v}'$

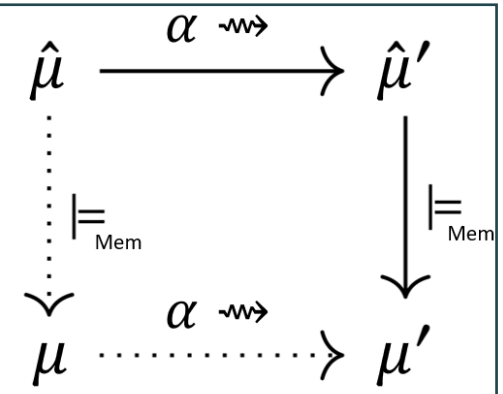


UX soundness

All *symbolic* actions can be simulated by a corresponding *concrete* action

If $\hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}')$ and $o \neq abort$ and $[\![\vec{\hat{E}}]\!]_\theta = \vec{v}$ and
$[\![\hat{\pi}']\!]_\theta = \text{true}$ and $\theta, \mu' \models_{\text{Mem}} \hat{\mu}'$ and $[\![\vec{\hat{E}}']\!]_\theta = \vec{v}'$

then $\exists \mu.\ \theta, \mu \models_{\text{Mem}} \hat{\mu}$ and $\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$



Result
WiSL actions are sound.

Reasoning with function specifications

CSE

Functions with rich function specifications expressed using user-defined predicates to capture operations over e.g. pointer-based data structures

Consume and produce

- Reasoning about function calls using function specifications
- Verification of function specifications
- Folding and unfolding of user-defined predicates
- Using and proving lemmas (similar to functions)

Compositionality: Function Call Rule

$\Gamma(f)|_m = \langle\!\langle \vec{\mathrm{x}} = \vec{x} \star P \rangle\!\rangle\ f(\vec{\mathrm{x}})\ \langle\!\langle ok : Q_{ok} \rangle\!\rangle\ \langle\!\langle err : Q_{err} \rangle\!\rangle$ — get function specification

$[\![\vec{\mathrm{E}}]\!]_{\hat{s}} \Downarrow \vec{\hat{E}}$ and $\hat{\theta} = [\vec{x} \mapsto \vec{\hat{E}}]$ — evaluate function parameters

$\hat{\pi}' = (\vec{\hat{E}} \in \mathrm{Val} \wedge \hat{\pi})$ — successful evaluation in path condition

$\mathrm{consume}(m, P, \hat{\theta}, (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}')) \rightsquigarrow (ok, \hat{\theta}', (\hat{s}, \hat{\mu}_f, \hat{\mathcal{P}}_f, \hat{\pi}''))$ — consume pre-condition

$\hat{r}$ fresh — fresh variable for return value

$Q'_{ok} = Q_{ok}[\hat{r}/\mathrm{ret}]$ and $\hat{\theta}'' = \hat{\theta}'[r \mapsto \hat{r}]$ — set up return value

$\hat{\pi}''' = (\hat{r} \in \mathrm{Val} \wedge \hat{\pi}'')$ — include $\hat{r}$ in path condition

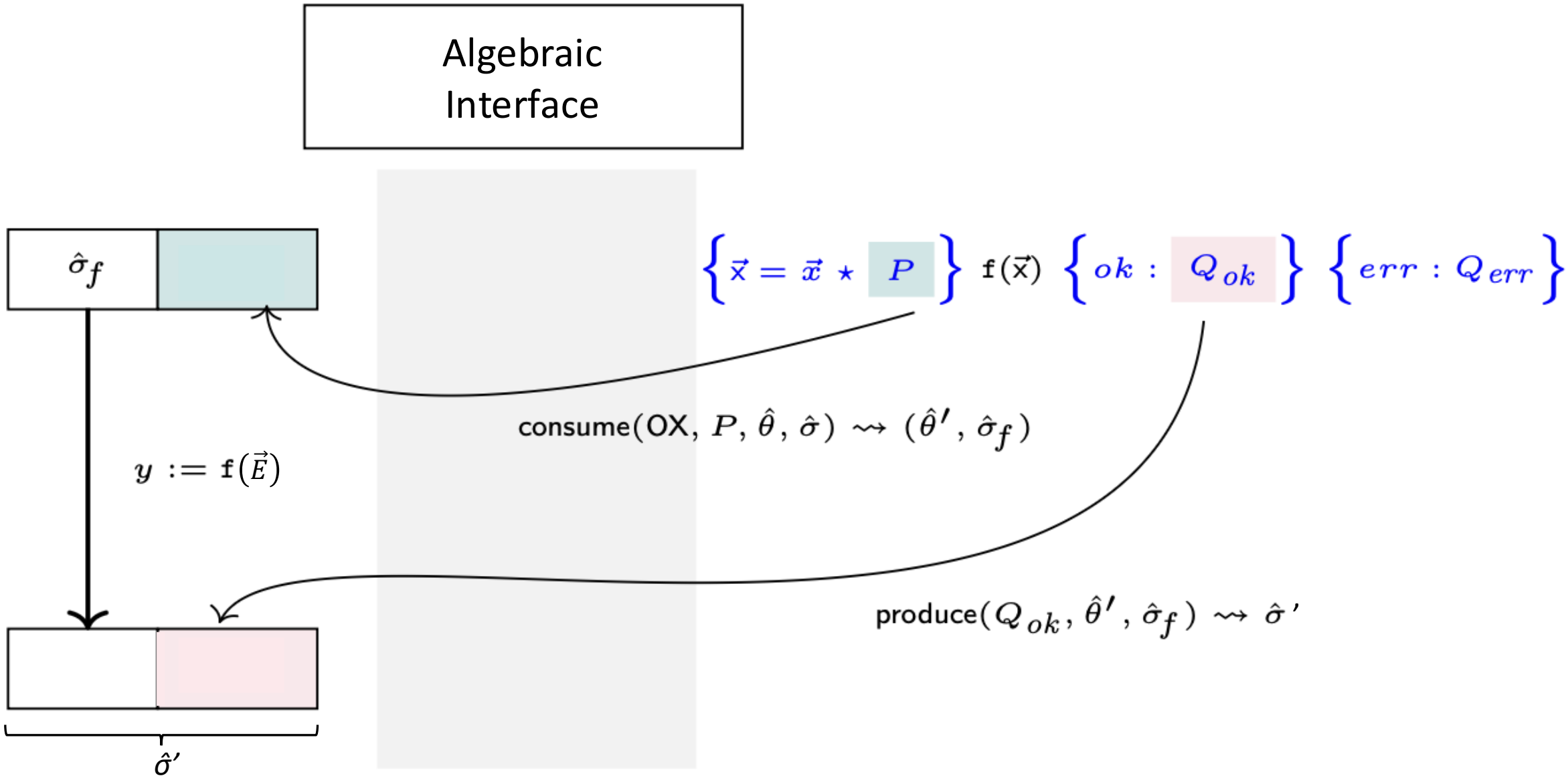
$\mathrm{produce}(Q'_{ok}, \hat{\theta}'', (\hat{s}, \hat{\mu}_f, \hat{\mathcal{P}}_f, \hat{\pi}''')) \rightsquigarrow (\hat{s}, \hat{\mu}', \hat{\mathcal{P}}', \hat{\pi}'''')$ — produce post-condition

$\hat{\pi}''''' = (\hat{s}(\mathrm{y}) \in \mathrm{Val} \wedge \hat{\pi}'''')$ — include y in path condition

---

$$(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \mathrm{y} := f(\vec{\mathrm{E}}) \Downarrow^m_\Gamma ok : (\hat{s}[\mathrm{y} \mapsto \hat{r}], \hat{\mu}', \hat{\mathcal{P}}', \hat{\pi}''''')$$

Compositional Symbolic Execution: Function Compositionality



Soundness of Consume and Produce

How do we prove soundness of constructs defined in terms of consume and produce, such as function call and fold/unfold?

Our answer: algebraic interface for consume and produce

Our interface captures mathematically what has been illustrated using pictures in the previous slides

Mathematical definitions allow us to show:

- Consume-and-produce operations are sound assuming the interface holds and the memory actions satisfy frame.
- The Gillian implementation of consume and produce satisfies the interface.

Our Consume and Produce Interface

Property 1 (OX Soundness: consume).

$$\begin{array}{ll} \textit{If} & m \in \{OX, EX\} \textit{ and} \\ & (\forall \hat{\theta}', \hat{\sigma}', o.\ \textit{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, \hat{\theta}', \hat{\sigma}') \ \textit{ and } \ \mathsf{SAT}(\hat{\sigma}') \Rightarrow o \neq \textit{abort}) \\ & \textit{and } \ \theta, s, \mu \models \hat{\sigma} \\ \textit{then} & \exists \mu_P, \mu_f, \hat{\theta}', \hat{\sigma}_f.\ \textit{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (ok, \hat{\theta}', \hat{\sigma}_f) \ \textit{ and } \ \mu = \mu_P \cdot \mu_f \\ & \textit{and } \ \theta, s, \mu_P \models \hat{\theta}'(P) \textit{ and } \ \theta, s, \mu_f \models \hat{\sigma}_f \end{array}$$

Property 2 (UX Soundness: consume).

$$\begin{array}{ll} \textit{If} & \textit{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (ok, \hat{\theta}', \hat{\sigma}_f) \\ & \textit{and } \ \theta, s, \mu_P \models \hat{\theta}'(P) \textit{ and } \theta, s, \mu_f \models \hat{\sigma}_f \ \textit{ and } \ (\mu_P \cdot \mu_f) \textit{ is defined} \\ \textit{then} & \theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma} \end{array}$$

Property 3 (OX Soundness: produce). *Let* $\mathrm{lv}(P) \subseteq \mathrm{dom}(\hat{\theta})$.

$$\begin{array}{ll} \textit{If} & \theta, s, \mu_P \models \hat{\theta}'(P) \textit{ and } \theta, s, \mu \models \hat{\sigma} \textit{ and } (\mu_P \cdot \mu_f) \textit{ is defined} \\ \textit{then} & \exists \theta, \hat{\sigma}'.\ \theta'|_{\mathrm{lv}(\hat{\sigma})} = \theta \ \textit{ and } \ \textit{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}' \ \textit{ and } \ \theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}' \end{array}$$

Property 4 (UX Soundness: produce).

$$\begin{array}{ll} \textit{If} & \textit{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}' \textit{ and } \theta, s, \mu \models \hat{\sigma}' \\ \textit{then} & \exists \mu_P, \mu_f.\ \mu = \mu_P \cdot \mu_f \textit{ and } \ \theta, s, \mu_P \models \hat{\theta}(P) \ \textit{ and } \ \theta, s, \mu_f \models \hat{\sigma} \end{array}$$

Our Consume and Produce Interface

Property 1 (OX Soundness: consume).

If $m \in \{OX, EX\}$ *and*
$(\forall \hat{\theta}', \hat{\sigma}', o.\ consume(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, \hat{\theta}', \hat{\sigma}')$ *and* $\mathsf{SAT}(\hat{\sigma}') \Rightarrow o \neq abort)$
and $\theta, s, \mu \models \hat{\sigma}$
then $\exists \mu_P, \mu_f, \hat{\theta}', \hat{\sigma}_f.\ consume(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (ok, \hat{\theta}', \hat{\sigma}_f)$ *and* $\mu = \mu_P \cdot \mu_f$

Remark: The interface is independent of the
- memory model
- language
- implementation

Property 2
If
then $\theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}$

Property 3 (OX Soundness: produce). *Let* $\mathrm{lv}(P) \subseteq \mathrm{dom}(\hat{\theta})$.
If $\theta, s, \mu_P \models \hat{\theta}'(P)$ *and* $\theta, s, \mu \models \hat{\sigma}$ *and* $(\mu_P \cdot \mu_f)$ *is defined*
then $\exists \theta, \hat{\sigma}'.\ \theta'|_{\mathrm{lv}(\hat{\sigma})} = \theta$ *and* $produce(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *and* $\theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}'$

Property 4 (UX Soundness: produce).
If $produce(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *and* $\theta, s, \mu \models \hat{\sigma}'$
then $\exists \mu_P, \mu_f.\ \mu = \mu_P \cdot \mu_f$ *and* $\theta, s, \mu_P \models \hat{\theta}(P)$ *and* $\theta, s, \mu_f \models \hat{\sigma}$

Our Consume and Produce Interface

Property 1 (OX Soundness: consume).

If $m \in \{OX, EX\}$ *and*
$(\forall \hat{\theta}', \hat{\sigma}', o.\ consume(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, \hat{\theta}', \hat{\sigma}')$ *and* $\mathsf{SAT}(\hat{\sigma}') \Rightarrow o \neq abort)$
and $\theta, s, \mu \models \hat{\sigma}$
then $\exists \mu_P, \mu_f, \hat{\theta}', \hat{\sigma}_f.\ consume(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (ok, \hat{\theta}', \hat{\sigma}_f)$ *and* $\mu = \mu_P \cdot \mu_f$
and $\theta, s, \mu_P \models \hat{\theta}'(P)$ *and* $\theta, s, \mu_f \models \hat{\sigma}_f$

Result: Function Call Soundness

If the consume and produce operations satisfy these OX/UX properties, then the function call rule is OX/UX sound.

Property (illegible)
If
and $\theta, s, \mu_P \models \hat{\theta}'(P)$ *and* $\theta, s, \mu_f \models \hat{\sigma}_f$ *and* $(\mu_P \cdot \mu_f)$ *is defined*
then $\theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}$

Property 3 (OX Soundness: produce). *Let* $\mathrm{lv}(P) \subseteq \mathrm{dom}(\hat{\theta})$.
If $\theta, s, \mu_P \models \hat{\theta}'(P)$ *and* $\theta, s, \mu \models \hat{\sigma}$ *and* $(\mu_P \cdot \mu_f)$ *is defined*
then $\exists \theta, \hat{\sigma}'.\ \theta'|_{\mathrm{lv}(\hat{\sigma})} = \theta$ *and* $produce(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *and* $\theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}'$

Property 4 (UX Soundness: produce).
If $produce(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *and* $\theta, s, \mu \models \hat{\sigma}'$
then $\exists \mu_P, \mu_f.\ \mu = \mu_P \cdot \mu_f$ *and* $\theta, s, \mu_P \models \hat{\theta}(P)$ *and* $\theta, s, \mu_f \models \hat{\sigma}$

Our Consume and Produce Interface

Property 1 (OX Soundness: consume).

If $m \in \{OX, EX\}$ *and*
$(\forall \hat{\theta}', \hat{\sigma}', o.\ consume(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, \hat{\theta}', \hat{\sigma}')$ *and* $\mathsf{SAT}(\hat{\sigma}') \Rightarrow o \neq abort)$
and $\theta, s, \mu \models \hat{\sigma}$
then $\exists \mu_P, \mu_f, \hat{\theta}', \hat{\sigma}_f.\ consume(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (ok, \hat{\theta}', \hat{\sigma}_f)$ *and* $\mu = \mu_P \cdot \mu_f$
and $\theta, s, \mu_P \models \hat{\theta}'(P)$ *and* $\theta, s, \mu_f \models \hat{\sigma}_f$

Result

Gillian's consume and produce operations satisfy these properties.

Property 2 (UX S

If *consu*
and $\theta, s, \mu_P \models \theta'(P)$ *and* $\theta, s, \mu_f \models \sigma_f$ *and* $(\mu_P \cdot \mu_f)$ *is defined*
then $\theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}$

Property 3 (OX Soundness: produce). *Let* $\mathrm{lv}(P) \subseteq \mathrm{dom}(\hat{\theta})$.
If $\theta, s, \mu_P \models \hat{\theta}'(P)$ *and* $\theta, s, \mu \models \hat{\sigma}$ *and* $(\mu_P \cdot \mu_f)$ *is defined*
then $\exists \theta, \hat{\sigma}'.\ \theta'|_{\mathrm{lv}(\hat{\sigma})} = \theta$ *and* $produce(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *and* $\theta, s, (\mu_P \cdot \mu_f) \models \hat{\sigma}'$

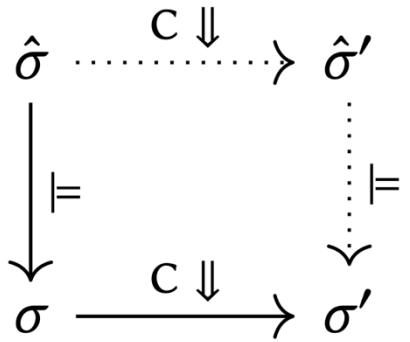
Property 4 (UX Soundness: produce).
If $produce(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *and* $\theta, s, \mu \models \hat{\sigma}'$
then $\exists \mu_P, \mu_f.\ \mu = \mu_P \cdot \mu_f$ *and* $\theta, s, \mu_P \models \hat{\theta}(P)$ *and* $\theta, s, \mu_f \models \hat{\sigma}$

General Soundness Definition

OX soundness

All *concrete* executions can be simulated by a corresponding *symbolic* execution

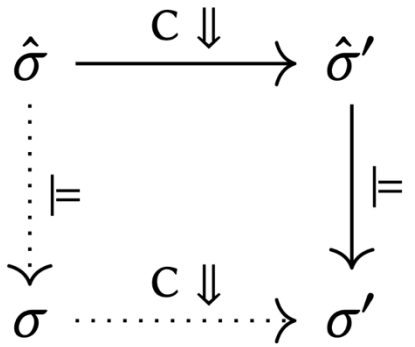
$$\sigma, \mathrm{C} \Downarrow \sigma' \wedge \sigma \models \hat{\sigma} \Rightarrow \exists \hat{\sigma}', \hat{\sigma}, \mathrm{C} \Downarrow \hat{\sigma}' \wedge \sigma' \models \hat{\sigma}'.$$



UX soundness

All *symbolic* executions can be simulated by a corresponding *concrete* execution

$$\hat{\sigma}, \mathrm{C} \Downarrow \hat{\sigma}' \wedge \sigma' \models \hat{\sigma}' \Rightarrow \exists \sigma, \sigma, \mathrm{C} \Downarrow \sigma' \wedge \sigma \models \hat{\sigma}.$$



General OX Soundness Result

Concrete memory actions satisfy OX frame property

If $(\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$
then $\exists \mu'', o', \vec{v}''.\ \mu.\alpha(\vec{v}) \rightsquigarrow o' : (\mu'', \vec{v}'')$ and
$(o' \neq \mathit{miss} \Rightarrow (\mu' = \mu'' \cdot \mu_f \text{ and } o' = o \text{ and } \vec{v}'' = \vec{v}'))$

+

Symbolic memory action OX soundness

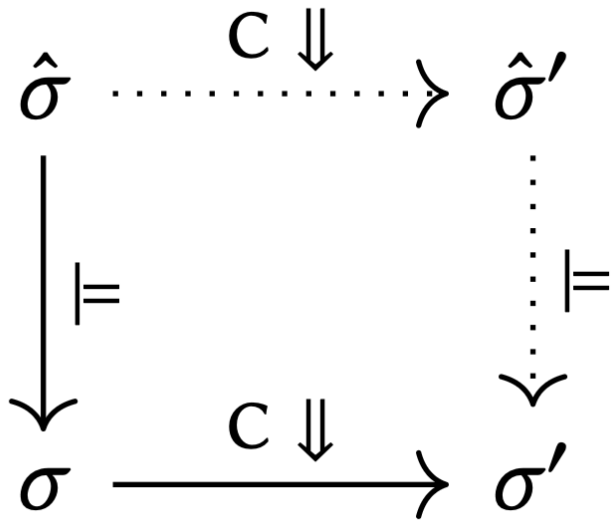
If $\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$ and $\theta, \mu \models_{\text{Mem}} \hat{\mu}$ and $[\![\vec{\hat{E}}]\!]_\theta = \vec{v}$ and
$(\forall o, \hat{\mu}', \hat{\pi}', \vec{\hat{E}}'.\ \hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}')$ and $[\![\hat{\pi}']\!]_\theta = \text{true}$ and $[\![\vec{\hat{E}}']\!]_\theta \neq \lightning \Rightarrow o \neq \mathit{abort})$
then $\exists \hat{\mu}', \hat{\pi}', \vec{\hat{E}}', \theta'.\ \hat{\mu}.\alpha(\vec{\hat{E}}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{\hat{E}}')$ and $\theta'|_{\text{lv}(\hat{\mu})} = \theta$
and $[\![\hat{\pi}']\!]_{\theta'} = \text{true}$ and $\theta', \mu' \models_{\text{Mem}} \hat{\mu}'$ and $[\![\vec{\hat{E}}']\!]_{\theta'} = \vec{v}'$

+

$\text{consume}_{\text{Res}}$ & $\text{produce}_{\text{Res}}$ OX soundness

implies

OX soundness for symbolic execution *with function calls and fold/unfold*



Result

WiSL symbolic execution is sound

Memory-model instances: CSE and Gillian

Memory model name	§	OX	UX	Rocq kLoC	Gillian
Linear model	6.1	✓	✓	1	✗
Linear model with unique-match branching	6.1	✓	✓	1	✗
Linear model with angelic branching	6.1	✗	✓	≈ 0.5	✗
Linear model without negative information	6.1	✓	✗	≈ 0.5	✗
Fractional ownership model	6.2	✓	✓	2	✓*
VeriFast-inspired model for C	6.3	✓	✗	≈ 0.5	✗
Block-offset model for C	6.4	✓	✓	4	✓
Models for OOP languages (e.g., JavaScript)	6.5	✓	✓	✗	✓
CHERI-assembly model	6.6	✓	✗	19	✗
Block-of-trees model for C	6.7	✓	✓	✓*	✓

Yellow checkmark = not applicable, red checkmark = future work, * = variant implemented in Gillian

WiSL demo

Gillian-While with block-offset memory

`https://youtu.be/3lLeZB_91L4`

Diego Cupello

C demo

Gillian-C with heap tree memory
(see OPLSS26 fourth lecture)

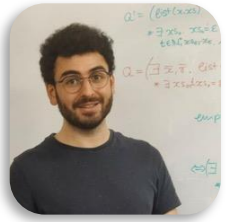
`https://youtu.be/zsJLskHjmG0`

Nat Karmios

Published work, ECOOP’24

- **Unified Compositional** Symbolic Execution

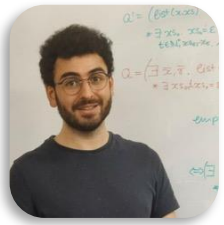




- Haskell Implementation

- Gillian Platform

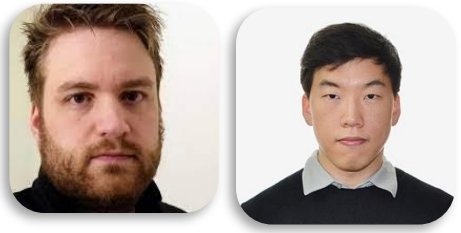




Published Work, OOPSLA'25

- Unified **Parametric** Compositional Symbolic Execution

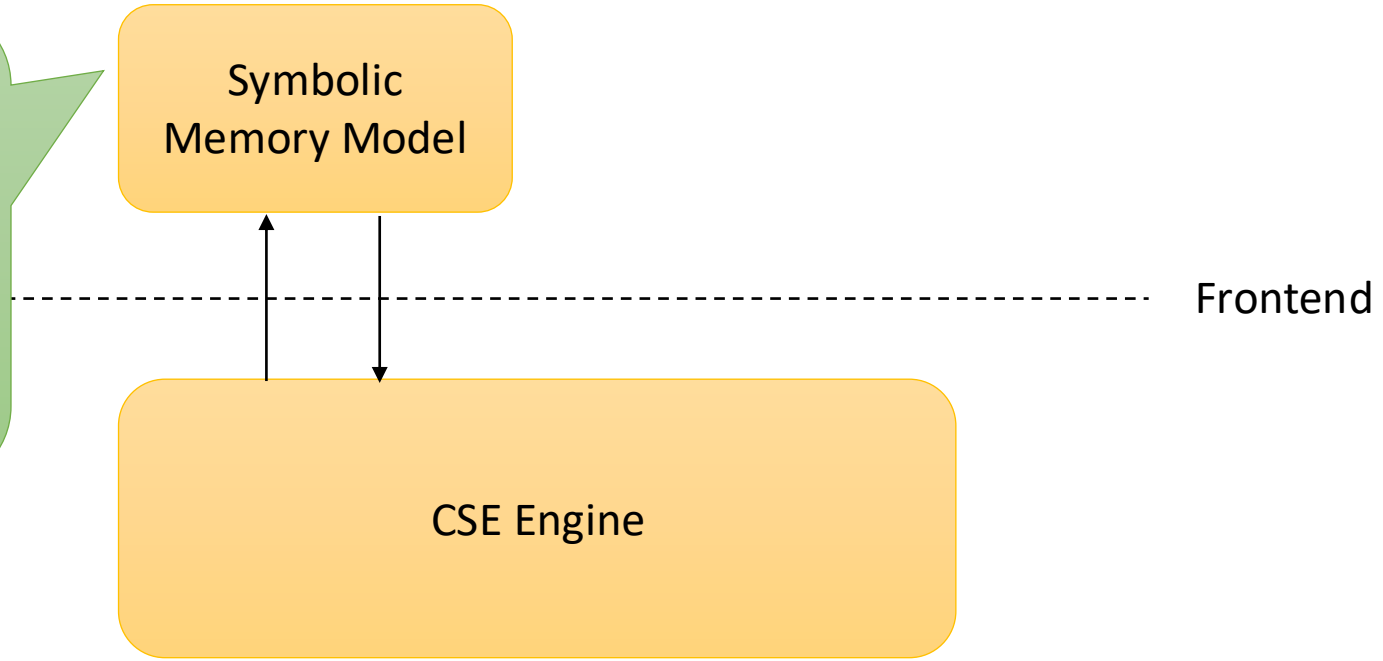
- **Coq Formalism and Verification**



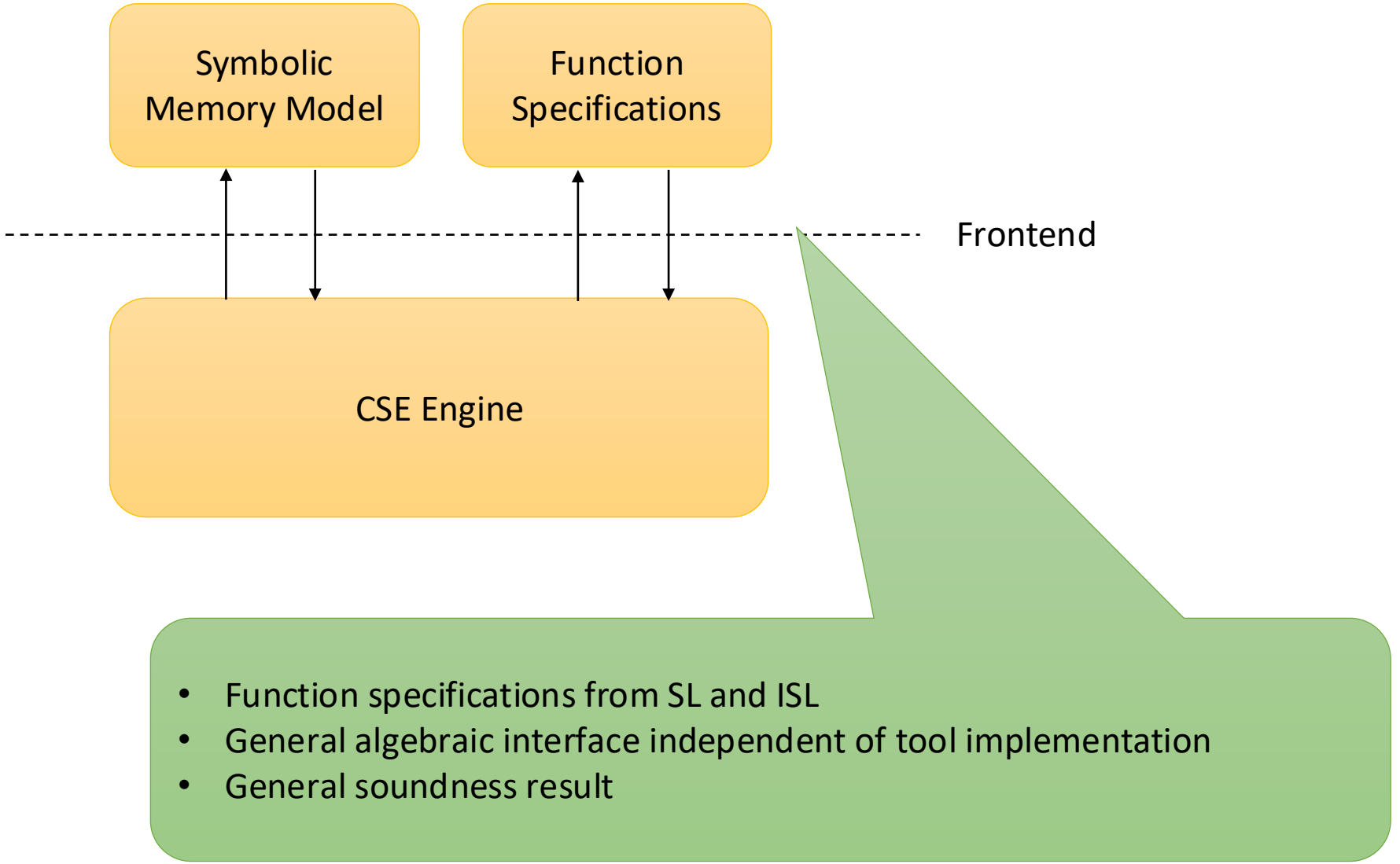
- Gillian Platform

Compositional Symbolic Execution: Foundations

- Symbolic Memory
- Symbolic Actions
- $\text{consume}_{\text{res}}$, $\text{produce}_{\text{res}}$
- OX/UX frame property
- Memory model soundness



Compositional Symbolic Execution: Foundations



CSE Analysis Standardisation….