SuSLik: synthesis of safe pointer-manipulating programs

Nadia Polikarpova
joint work with Ilya Sergey (Yale-NUS)
follow along

https://github.com/TyGuS/suslik-tutorial
pointer-manipulating programs

operating systems

network / security protocols

browsers

😊 efficient

😊 hard to write

😊 memory safety bugs
how to make them safe?

write in a high-level language

😊 easy to write
😊 have to rewrite everything

write in C, verify in Coq

😀 hard to write
😀 backwards compatible
program synthesis to the rescue

specification

😊 easy to write

😢 verbose
😢 unstructured
😢 pointers & aliasing

code

😊 efficient
😊 backwards compatible
😊 provably memory-safe
SuSLik

(Synthesis using Separation Logik)
the SuSLik approach

- separation logic
- deductive synthesis
- code

😊 reasoning about pointers & aliasing
😊 uses specs to guide synthesis
this tutorial

1. example: swap
   a taste of SuSLik

2. intro to separation logic
   reasoning about pointer-manipulating programs

3. deductive synthesis
   from SL specifications to programs
**this tutorial**

1. example: swap
2. intro to separation logic
3. deductive synthesis
example: swap

Swap values of two distinct pointers

void swap(loc x, loc y)
example: swap

start state:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

end state:

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

in separation logic:

\[
\{ x \mapsto A \quad y \mapsto B \} \\
\{ x \mapsto B \quad y \mapsto A \} \\
\]

void swap(loc x, loc y)

precondition

postcondition

ghost variables
demo 1: swap

Swap values of two distinct pointers

```c
void swap(loc x, loc y)
```
\{ x \mapsto A \ast y \mapsto B \}

??

\{ x \mapsto B \ast y \mapsto A \}
let a1 = *x;

\{ x \mapsto a1 \land y \mapsto B \}

??

\{ x \mapsto B \land y \mapsto a1 \}
let a1 = *x;
let b1 = *y;

{x ← a1 * y ← b1}

??

{x ← b1 * y ← a1}
let a1 = *x;
let b1 = *y;
*x = b1;

{x ↦ b1 * y ↦ b1}

??

{x ↦ b1 * y ↦ a1}
let a1 = *x;
let b1 = *y;
*x = b1;
*y = a1;

\{ x \mapsto b1 \, y \mapsto a1 \}

??

\{ x \mapsto b1 \, y \mapsto a1 \}
same
let a1 = *x;
let b1 = *y;
*x = b1;
*y = a1;
void swap(loc x, loc y) {
    let a1 = *x;
    let b1 = *y;
    *x = b1;
    *y = a1;
}

exercise 1: rotate three

start state:

```
A
x

B
y

C
z
```

end state:

```
C
x

A
y

B
z
```

```
void rotate(loc x, loc y, loc z)
```
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
separation logic (SL)

Hoare logic
“about the heap”
starting in a state that satisfies $P$
program $c$ will execute \textit{without memory errors},
and upon its termination the state will satisfy $Q$
1. example: swap

2. intro to separation logic
   2.1. programs and assertions
   2.2. inductive predicates
   2.3. specifying data transformations

3. deductive synthesis
separation logic (SL)
programs

do nothing  
skip
programs

do nothing

read from heap

skip

let y = *(x + n)

offset (natural number)

variables
programs

- do nothing
- read from heap
- write to heap

skip

let $y = *(x + n)$

$*(x + n) = e$  

(expression
(arithmetic, boolean))
programs

do nothing  
read from heap  
write to heap  
an allocate block  

skip

let y = *(x + n)

*(x + n) = e

let y = malloc(n)
programs

do nothing
read from heap
write to heap
allocate block
free block

skip
let y = *(x + n)
*(x + n) = e
let y = malloc(n)
free(x)
programs

- do nothing: skip
- read from heap: let $y = *(x + n)$
- write to heap: $*(x + n) = e$
- allocate block: let $y = malloc(n)$
- free block: free(x)
- procedure call: $p(e_1, \ldots, e_n)$
programs

do nothing
read from heap
write to heap
allocate block
free block
procedure call
assignment

skip
let \( y = *(x + n) \)
\( *(x + n) = e \)
let \( y = \text{malloc}(n) \)
free(\( x \))
\( p(e_1, \ldots, e_n) \)

only heap is mutable, not stack variables!
programs

- do nothing: `skip`
- read from heap: `let y = *(x + n)`
- write to heap: `*(x + n) = e`
- allocate block: `let y = malloc(n)`
- free block: `free(x)`
- procedure call: `p(e_1, ..., e_n)`
- sequential composition: `c_1 ; c_2`
- conditional: `if (e) {c_1} else {c_2}`
separation logic (SL)

{\text{P}} \text{ c } {\text{Q}}

SL assertions
SL assertions

empty heap

{ emp }
SL assertions

empty heap \{ \text{emp} \}

singleton heap \{ y \mapsto 5 \}
SL assertions

empty heap

\{ \text{emp} \} 

singleton heap

\{ y \mapsto 5 \} 

separating conjunction

\{ x \mapsto y \ast y \mapsto 5 \} 

heaplets
SL assertions

empty heap      \{ \text{emp} \}  

singleton heap \{ y \mapsto 5 \}  

separating conjunction  \{ x \mapsto y * y \mapsto 5 \}  

memory block \{ [x, 2] * x \mapsto 5 * (x + 1) \mapsto 10 \}
SL assertions

empty heap  { emp }

singleton heap  { y ↦ 5 }

separating conjunction  { x ↦ y * y ↦ 5 }

memory block  { [x, 2] * x ↦ 5 * (x + 1) ↦ 10 }

+ pure formula  { A > 5 ; x ↦ A }
separation logic (SL)

\{P\} \ c \ \{Q\}

starting in a state that satisfies $P$
program $c$ will execute without memory errors,
and upon its termination the state will satisfy $Q$
example triples

\{ x \mapsto A \} \quad \quad *x = 5 \quad \quad \{ x \mapsto 5 \}\\
\{ x \mapsto A \} \quad \quad *(x + 1) = 5 \quad \quad \text{✗}\\
\{ x \mapsto A \} \quad \quad \text{let } y = *x \quad \quad \{ x \mapsto y \}\\
\{ \text{emp} \} \quad \quad \text{let } y = \text{malloc}(2) \quad \quad \{ [y, 2] * y \mapsto A * (y + 1) \mapsto B \}\\
\{ [x, 2] * x \mapsto 5 * (x + 1) \mapsto 7 \} \quad \quad \text{free}(x + 1) \quad \quad \text{✗}
1. example: swap

2. intro to separation logic
   2.1. programs and assertions
   2.2. inductive predicates
   2.3. specifying data transformations

3. deductive synthesis
SL assertions: linked structures
SL assertions: linked structures

linked list \{ x = 0 ; emp \}
SL assertions: linked structures

linked list \( \{ [x, 2] \ast x \mapsto V \ast (x + 1) \mapsto 0 \} \)
SL assertions: linked structures

linked list \{ [x, 2] * x \rightarrow V * (x + 1) \rightarrow Y * \\
[Y, 2] * Y \rightarrow V' * (Y + 1) \rightarrow 0 \}

\[
\begin{array}{c}
\text{V} \\
\downarrow \\
\text{X}
\end{array}
\quad
\begin{array}{c}
\text{V'} \\
\downarrow \\
\text{Y}
\end{array}
\]
SL assertions: linked structures

{ \[x, 2\] * x ↦ V * (x + 1) ↦ Y * \\
[Y, 2] * Y ↦ V’ * (Y + 1) ↦ Y’ *

... 

}

inductive predicates to the rescue!
the linked list predicate

**predicate list (loc x) {**

| x = 0 => { emp } |
| x ≠ 0 => { [x, 2] |
| * x ↦ V |
| * (x + 1) ↦ Y |
| * list(Y) |

}
this tutorial

1. example: swap

2. intro to separation logic
   2.1. programs and assertions
   2.2. inductive predicates
   2.3. specifying data transformations

3. deductive synthesis
demo 2: dispose a list

```c
void dispose(loc x)
{
    list(x)
}
{ emp }
```
linked list with elements

**predicate** list (loc x, set s) {
  | x = 0  => { s = ∅ ; emp } 
  | x ≠ 0  => { s = {V} + S’ ; 
  [x, 2]
  * x ↦ V * (x + 1) ↦ Y 
  * list(Y, S’) }
}
demo 3: copy a list

void copy(loc x, loc r)

{ list(x, S) * r \mapsto _ }  

{ list(x, S) * r \mapsto Y * \text{list}(Y, S) }

return location
exercise 2: append two lists

```c
void append( ??? )
{
    ???
    ???
}
```
exercise 3: schema migration

```c
void single_to_double(loc x)
{
    ???
}  // singly-linked list with reserved space in each node

{ ??? }  // doubly-linked list at the same address
```
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
deductive synthesis

synthesis as proof search
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
   3.1. proof system
   3.2. proof search
transforming entailment

\[ P \xrightarrow{c} Q \mid c \]

a state that satisfies $P$
can be transformed into a state that satisfies $Q$
using a program $c$
synthetic separation logic (SSL)

proof system for transforming entailment
\{emp\} \xrightarrow{\sim} \{emp\} \mid ??
(Emp)

\{emp\} \xrightarrow{\sim} \{emp\} \mid \text{skip}
(Frame)

\[
\begin{array}{c}
\{ P \} \rightsquigarrow \{ Q \} \mid c \\
\hline
\{ P \ast R \} \rightsquigarrow \{ Q \ast R \} \mid ??
\end{array}
\]
(Write)

\[
\{ x \mapsto e \ast P \} \xrightarrow{\ \ast \ } \{ x \mapsto e \ast Q \} \mid c
\]

\[
\{ x \mapsto \_ \ast P \} \xrightarrow{\ \ast \ } \{ x \mapsto e \ast Q \} \mid ??
\]
(Read)

\[[y/A]\{ x \mapsto A \ast P \} \leadsto [y/A]\{ Q \} \mid c\]

\[
\{ x \mapsto A \ast P \} \leadsto \{ Q \} \mid ??
\]
SSL: basic rules

(Emp)

\[
\{\text{emp}\} \xrightarrow{\text{skip}} \{\text{emp}\}
\]

(Read)

\[
[y/A]\{ x \mapsto A \ast P \} \xrightarrow{\text{let } y = *x} [y/A]\{ Q \} | \ c
\]

(Frame)

\[
\{ P \} \xrightarrow{\ c} \{ Q \}
\]

\[
\{ P \ast R \} \xrightarrow{\ c} \{ Q \ast R \}
\]

(Write)

\[
\{ x \mapsto e \ast P \} \xrightarrow{\ c} \{ x \mapsto e \ast Q \}
\]

\[
\{ x \mapsto _\ast P \} \xrightarrow{\ c} \{ x \mapsto e \ast Q \} | *x = e; \ c
\]
example: swap

\{ x \mapsto A \ast y \mapsto B \} \iff \{ x \mapsto B \ast y \mapsto A \} | ??
\{ x \mapsto A \ast y \mapsto B \} \iff \{ x \mapsto B \ast y \mapsto A \} | ??
\[
\{ x \mapsto a1 \ast y \mapsto B \} \mapsto \{ x \mapsto B \ast y \mapsto a1 \}\ |
\text{let } a1 = *x; \quad ??
\]
\[
\{ x \mapsto a_1 \times y \mapsto b \} \xrightarrow{??} \{ x \mapsto b \times y \mapsto a_1 \} \mid ??
\]

\[
\{ x \mapsto a_1 \times y \mapsto B \} \xrightarrow{??} \{ x \mapsto B \times y \mapsto a_1 \} \mid \text{let } b_1 = *y; ??
\]

\[
\{ x \mapsto A \times y \mapsto B \} \xrightarrow{??} \{ x \mapsto B \times y \mapsto A \} \mid \text{let } a_1 = *x; ??
\]
\[
\begin{align*}
\{ x \mapsto b1 \ast y \mapsto b1 \} &\rightsquigarrow \{ x \mapsto b1 \ast y \mapsto a1 \} \quad | \quad ?? \\
\{ x \mapsto a1 \ast y \mapsto b1 \} &\rightsquigarrow \{ x \mapsto b1 \ast y \mapsto a1 \} \quad | \quad *x = b1; ?? \\
\{ x \mapsto a1 \ast y \mapsto B \} &\rightsquigarrow \{ x \mapsto B \ast y \mapsto a1 \} \quad | \quad \text{let } b1 = *y; ?? \\
\{ x \mapsto A \ast y \mapsto B \} &\rightsquigarrow \{ x \mapsto B \ast y \mapsto A \} \quad | \quad \text{let } a1 = *x; ??
\end{align*}
\]
\[\{ y \mapsto a_1 \} \xrightarrow{\text{Frame}} \{ y \mapsto a_1 \} | \text{??}\]

\[\{ y \mapsto b_1 \} \xrightarrow{\text{Write}} \{ y \mapsto a_1 \} | *y = a_1; \text{??}\]

\[\{ x \mapsto b_1 \ast y \mapsto b_1 \} \xrightarrow{\text{Write}} \{ x \mapsto b_1 \ast y \mapsto a_1 \} | \text{??}\]

\[\{ x \mapsto a_1 \ast y \mapsto b_1 \} \xrightarrow{\text{Write}} \{ x \mapsto b_1 \ast y \mapsto a_1 \} | *x = b_1; \text{??}\]

\[\{ x \mapsto a_1 \ast y \mapsto B \} \xrightarrow{\text{Read}} \{ x \mapsto B \ast y \mapsto a_1 \} | \text{let } b_1 = *y; \text{??}\]

\[\{ x \mapsto A \ast y \mapsto B \} \xrightarrow{\text{Read}} \{ x \mapsto B \ast y \mapsto A \} | \text{let } a_1 = *x; \text{??}\]
let a1 = *x;
let b1 = *y;

{x \mapsto a1, y \mapsto b1} \rightarrow \{ x \mapsto b1 \cdot y \mapsto b1 \} | \text{skip}

{x \mapsto a1, y \mapsto b1} \rightarrow \{ x \mapsto b1 \cdot y \mapsto a1 \} | \text{*x = b1;}

{x \mapsto a1, y \mapsto B} \rightarrow \{ x \mapsto B \cdot y \mapsto a1 \} | \text{let b1 = *y;}

{x \mapsto A \cdot y \mapsto B} \rightarrow \{ x \mapsto B \cdot y \mapsto A \} | \text{let a1 = *x;}

{y \mapsto a1} \rightarrow \{ y \mapsto a1 \} | ??

{y \mapsto b1} \rightarrow \{ y \mapsto a1 \} | \text{*y = a1; ??}

{emp} \rightarrow \{ emp \} | \text{skip}
\{ x \mapsto A, y \mapsto B \}

\textbf{let} \ a_1 = \star x; \ \textbf{let} \ b_1 = \star y; \ \star x = b_1; \ \star y = a_1; \ \textbf{skip}

\{ x \mapsto B, y \mapsto A \}
demo 4: tracing swap
synthetic separation logic (SSL)

• basic rules
  (Emp), (Read), (Write), (Frame)
  (Alloc), (Free)

• pure reasoning and unification

• inductive predicates and recursion
synthetic separation logic (SSL)

• basic rules
  (Emp), (Read), (Write), (Frame)
  (Alloc), (Free)

• pure reasoning and unification

• inductive predicates and recursion
example: dispose a list

```c
void dispose(loc x)
{
    list(x)
}
{ emp }
```
\{ \text{list}^0(x) \} \quad \text{(Induction)}

\{ \text{emp} \}
predicate list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

{ list^0(x) }

?? (Open)

{ emp }

{ list^1(x) } void dispose(loc x) { emp }
predicate list(loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => {{x, 2}: x ↦ V * (x + 1) ↦ Y * list(Y)}
}

if (x == 0) {
  { x = 0 ; emp }

  ??

  { emp }
}

else {
  { x ≠ 0 ; [x, 2]: x ↦ V * (x + 1) ↦ Y * list^1(Y) }

  ??

  { emp }
}

{ list^1 (x) } void dispose(loc x) { emp }
```c
predicate list (loc x) {
    | x == 0 => { emp }
    | x != 0 => { [x, 2] * x => V * (x + 1) => Y * list(Y) }
}

if (x == 0) {
    { x == 0 ; emp }
        ??
        (Emp)
    { emp }
}
else {
    { x != 0 ; [x, 2] * x => V * (x + 1) => Y * list(Y) }
        ??
    { emp }
}
```

```c
{ list^1 (x) } void dispose(loc x) { emp }
```
predicate list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

if (x == 0) {
  { x = 0 ; emp }

  skip

  { emp }

} else {
  { x ≠ 0 ; [x, 2] * x ↦ V * (x + 1) ↦ Y * list¹(Y) }

  ??

  { emp }
}

{ list¹ (x) } void dispose(loc x) { emp }
predicate list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

    if (x == 0) { skip } else {
    { x ≠ 0 ; [x, 2] * x ↦ V * (x + 1) ↦ Y * list¹(Y) }

        ??

    { emp }
}

{ list¹ (x) } void dispose(loc x) { emp }
predicate list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

if (x == 0) { skip } else {
  { x ≠ 0 ; [x, 2] * x ↦ V * (x + 1) ↦ Y * list¹(Y) }

  ??

  { emp }
}

{ list¹ (x) } void dispose(loc x) { emp }
\textbf{predicate} list (loc x) {
| x = 0  => { emp } \\
| x ≠ 0  => { [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ Y ∗ list(Y) }
}

if (x == 0) { skip } else {
  let y1 = *(x + 1);

  \{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ y1 ∗ list¹(y1) \}

  ??

  \{ emp \}

}
**Predicate** list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

if (x == 0) { skip } else {
  let y1 = *(x + 1);
  { x ≠ 0 ; [x, 2] * x ↦ V * (x + 1) ↦ y1 * list¹(y1) }

  ??

  { emp }
}

{ list¹ (x) } void dispose(loc x) { emp }

(Free)
predicate list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

if (x == 0) { skip } else {
  let y1 = *(x + 1);
  free x;

  { x ≠ 0 ; list¹(y1) }
}

{ emp }

{ list¹ (x) } void dispose(loc x) { emp }
predicate list (loc x) {
    | x = 0 => { emp }
    | x ≠ 0 => {[x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

    if (x == 0) { skip } else {
        let y1 = *(x + 1);
        free x;

        { x ≠ 0 ; list¹ (y1) }

        ??

        { emp }
    }
predicate list (loc x) {
| x = 0 => {emp}
| x ≠ 0 => {[x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y)}
}

if (x == 0) { skip } else {
  let y1 = *(x + 1);
  free x;
  dispose(y1);
  {x ≠ 0; emp}
}

{ list^1(x) } void dispose(loc x) { emp }
predicate list (loc x) {
  | x = 0 => { emp }
  | x ≠ 0 => { [x, 2] * x ↦ V * (x + 1) ↦ Y * list(Y) }
}

if (x == 0) { skip } else {
  let y1 = *(x + 1);
  free x;
  dispose(y1);
}

{ x ≠ 0 ; emp } ?? { emp } (Emp)

{ list¹ (x) } void dispose(loc x) { emp }
predicate list (loc x) {
    | x = 0 => { emp }
    | x ≠ 0 => {[x, 2] * x \mapsto V * (x + 1) \mapsto Y * list(Y) }
}

if (x == 0) { skip } else {
    let y1 = *(x + 1);
    free x;
    dispose(y1);
    skip
}

{ list^{1}(x) } void dispose(loc x) { emp }
void dispose(loc x) {
    if (x == 0) {} else {
        let y1 = *(x + 1);
        free x;
        dispose(y1)
    }
}
synthetic separation logic (SSL)

• basic rules
  (Emp), (Read), (Write), (Frame), (Alloc), (Free)

• pure reasoning and unification

• inductive predicates and recursion
  (Open), (Close), (Induction), (Call)
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
   3.1. proof system
   3.2. proof search
SuSLik

backtracking search in SSL
+ optimizations
demo 5: backtracking
optimizations

• invertible rules
• early failure
• multi-phase search
• symmetry reduction
optimization: invertible rules

• invertible rules do not restrict the set of derivable programs
• idea: invertible rules need not be backtracked

(Read)

\[
[y/A]\{ x \mapsto A \ast P \} \rightsquigarrow [y/A]\{ Q \} \mid c
\]

\[
\{ x \mapsto A \ast P \} \rightsquigarrow \{ Q \} \mid \text{let } y = \ast x; \ c
\]
optimization: early failure

- idea: sometimes you know that a goal is unsatisfiable

\[
\psi \neq \bot \quad \vdash \phi \land \psi \Rightarrow \bot \quad \{ \text{emp} \} \rightsquigarrow \{ \bot ; \text{emp} \} \mid c
\]

\[
\{ \phi ; P \} \rightsquigarrow \{ \psi ; Q \} \mid c
\]
optimization: multi-phase search

• unfolding phase: deals with inductive predicates
  (Open), (Close), (Call), (Frame)
• flat phase: deals with points-to and blocks
  (Write), (Call), (Alloc), (Free), (Frame)
• idea: if unfolding phase cannot eliminate all predicates, give up
optimization: symmetry reduction

• sometimes rule applications commute

\[
\begin{align*}
\{ a \mapsto 0 \} & \rightarrow \{ b \mapsto 0 \} | ?? \\
\{ y \mapsto b \cdot a \mapsto 0 \} & \rightarrow \{ y \mapsto b \cdot b \mapsto 0 \} | ?? \\
\{ x \mapsto a \cdot y \mapsto b \cdot a \mapsto 0 \} & \rightarrow \{ x \mapsto a \cdot y \mapsto b \cdot b \mapsto 0 \} | ??
\end{align*}
\]

• idea: only allow one order of commuting applications
limitations & future work

• pure synthesis
  use an off-the-shelf pure synthesizer

• reasoning about inductive predicates
  identify decidable fragment

• goal must be inductive
  cyclic proofs to the rescue!

\[
\begin{align*}
\{r \mapsto \_\} & \Rightarrow \{r \geq x \land r \geq y; r \mapsto M\} \mid c \\
\{P\} & \Rightarrow \{Q\} \mid \text{skip}
\end{align*}
\]

\[
\begin{align*}
\{\text{tree}(t, S) \ast r \mapsto \_\} \\
\text{void flatten}(\text{loc } t, \text{loc } r) \\
\{r \mapsto X \ast \text{list}(X, S)\}
\end{align*}
\]
deductive synthesis with SuSLik

- separation logic
- deductive synthesis
- code

😊 reasoning about pointers & aliasing
😊 uses specs to guide synthesis
😊 provably memory-safe