SuSLik: deductive synthesis of safe programs with pointers

Nadia Polikarpova (UCSD)
Shachar Itzhaky (Technion)
follow along

https://github.com/TyGuS/suslik-tutorial
we’ve come a long way…

Checking a Large Routine

Alan Turing

```c
int fac (int n) {
    int s, r, u, v;
    for (u = r = 1; v = u, r < n; r++)
        for (s = 1; u += v, s++ < r; )
            return u;
}
```

1949

CompCert:
verified C compiler

Project Everest:
verified HTTPS stack

verified microkernel

verified OS kernel

today
... but this is still hard

have to write code, specs, and proofs!

Checking a Large Routine
Alan Turing

1949 today
what’s next for verified systems?

1949

Checking a Large Routine
Alan Turing

today

Security, Performance, Proof.
program synthesis
program synthesis with guarantees
program synthesis with guarantees

formal specification → ??? → code + proof
challenge 1: too many programs
challenge 1: too many programs
challenge 2: checking each program is hard
deductive synthesis
idea: to find the program, look for the proof

use spec to guide search!
deductive synthesis

A Deductive Approach to Program Synthesis

[Manna, Waldinger]

Leon

SuSLik

Synquid

VS3

[Kneuss et al’13]

[Polikarpova, Sergey’19]

[Polikarpova et al’16]

Srivastava et al’10

Fiat

[Delaware et al’15]

ImpSynth

[Qui, Solar-Lezama’18]

Jennisys

[Leino, Milicevic’12]

1980

today
deductive synthesis

of pointer-manipulating programs

A Deductive Approach to Program Synthesis

[Polikarpova, Sergey. POPL’19]

Leon

SuSLiK

Synquid

VS3

ImpSynth

Fiat

Jennisys

1980
today
SuSLik

specification → C code + proof
SuSLik

specification

? → C code + proof
SuSLik

specification

→

?-?

→

C code + proof

⚠ verbose
⚠ unstructured
⚠ pointers & aliasing
SuSLik

= Synthesis using Separation Logic

separation logic

deductive synthesis

C code + proof

😊 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

\texttt{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 \, \ast \, y \mapsto b \} \quad \text{precondition}
\{ x \mapsto b \, \ast \, y \mapsto a \} \quad \text{postcondition}
```

void \text{swap(loc x, loc y)}

“points-to”

heaplets

“and separately”

program variables

ghost variables
demo: swap

```c
void swap(loc x, loc y)
{x ↦ a * y ↦ b}
{x ↦ b * y ↦ a}
```

how did this happen?
deductive synthesis
idea: to find the program, look for the proof
\begin{align*}
\{ x \mapsto a \ast y \mapsto b \} \\
??
\{ x \mapsto b \ast y \mapsto a \}
\end{align*}
let a1 = *x;

{x ↦ a1 * y ↦ b}

??

{x ↦ b * y ↦ 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 \}
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:

\[ \begin{array}{ccc}
  a & b & c \\
  x & y & z \\
\end{array} \]

end state:

\[ \begin{array}{ccc}
  c & a & b \\
  x & y & z \\
\end{array} \]

\textbf{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”
separation logic (SL)

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

1. example: swap

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

3. deductive synthesis
separation logic (SL)

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

program
programs

do nothing  skip
programs

do nothing

read from heap

offset (natural number)

skip

let y = *(x + n)

variables
programs

- do nothing
- read from heap
- write to heap

skip

let y = *(x + n)

*(x + n) = e  expression
  (arithmetic, boolean)
programs

do nothing
skip

read from heap
let y = *(x + n)

write to heap
*(x + n) = e

allocate block
let y = malloc(n)
**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)**
programs

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

1. skip
2. `let y = *(x + n)`
3. `*(x + n) = e`
4. `let y = malloc(n)`
5. `free(x)`
6. `p(e_1, ..., e_n)`
programs

- do nothing: skip
- read from heap: \( y = *(x + n) \)
- write to heap: \( *(x + n) = e \)
- allocate block: \( y = \text{malloc}(n) \)
- free block: \( \text{free}(x) \)
- procedure call: \( p(e_1, ..., e_n) \)
- assignment: only heap is mutable, not stack variables!
programs

do nothing
read from heap
write to heap
allocate block
free block
procedure call
sequential composition
conditional

skip
let $y = \ast(x + n)$
$\ast(x + n) = e$
let $y = \text{malloc}(n)$
$\text{free}(x)$
$p(e_1, ..., e_n)$
$c_1; c_2$
if (e) \{c_1\} else \{c_2\}
separation logic (SL)

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

SL assertions
SL assertions

empty heap  {emp}
SL assertions

empty heap \{ \text{emp} \}

singleton heap \{ y \mapsto 5 \}

5

y
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 \{ \text{emp} \}

singleton heap \{ y \mapsto 5 \}

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

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

+ pure formula \{ a > 5 ; x \mapsto 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 *x = 5 \quad \{ x \mapsto 5 \}

\{ x \mapsto a \} \quad *(x + 1) = 5 \quad \text{\textcolor{red}{x}}

\{ x \mapsto a \} \quad \text{let } y = *x \quad \{ x \mapsto y \}

\{ \text{emp} \} \quad \text{let } y = \text{malloc}(2) \quad \{ [y, 2] * y \mapsto a * (y + 1) \mapsto b \}

\{ [x, 2] * x \mapsto 5 * (x + 1) \mapsto 7 \} \quad \text{free}(x + 1) \quad \text{\textcolor{red}{x}}
this tutorial

1. example: swap

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

3. deductive synthesis
example: dispose

```c
void dispose(loc x)
```
deallocate a linked list with head pointer x
dynamic data structures

how do we specify a linked list?

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)
  }
}
demo: dispose a list

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

what if we care about content?
example: copy

copy a linked list with head pointer x

```c
void copy(loc x, loc ret)
```
**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 ↦ y ; list(y, S') }
}

pure part  spatial part
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: copy a list

```c
void copy(loc x, loc ret)
{
  list(x, S) * ret ↦ _
}
{
  list(x, S) * ret ↦ y * list(y, S)
}
```

return location

existential
exercise 2: single to double

convert a singly-linked list into a doubly-linked list

option 1: in place

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

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

option 2: with re-allocation

```
void sll_to_dll(loc ret)
{
    ???
}  // singly-linked list and return location

{ ??? }  // doubly-linked list with same elements
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
deductive synthesis
idea: to find the program, look for the proof
this tutorial

1. example: swap
2. intro to separation logic

3. deductive synthesis
   3.1. proof system
   3.2. recursion and cyclic proofs
   3.3. discovering recursive auxiliaries
synthetic separation logic (SSL)

a proof system for program derivation relation

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

a state that satisfies $P$
can be transformed into a state that satisfies $Q$
using a program $c$
(Emp)

$\{\text{emp}\} \rightsquigarrow \{\text{emp}\} \mid ???$

empty heap
{emp} ⊸ {emp} | skip
(Frame)

\[ \{ P \ast R \} \xrightarrow{\text{same heap}} \{ Q \ast R \} \mid ?? \]
(Frame)

\[
\{ P \} \sim \rightarrow \{ Q \} \mid c \\
\hline
\{ P \ast R \} \sim \rightarrow \{ Q \ast R \} \mid c
\]

same heap
(Write)

\{ x \mapsto \_ \ast P \} \rightsquigarrow \{ x \mapsto e \ast Q \} \mid ??

points to non-ghost expression
(Write)

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

\[
\{ x \mapsto _\ast P \} \leadsto \{ x \mapsto e \ast Q \} \mid \ast x = e; \ c
\]

points to non-ghost expression
(Read)

\[
\{ \text{x} \mapsto a \ast P \} \sim \Rightarrow \{ Q \} \quad | \quad ??
\]

points to ghost variable
(Read)

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

\[
\begin{align*}
\{ x \mapsto a \ast P \} &\rightsquigarrow \{ Q \} \\
\text{let } y = *x; c
\end{align*}
\]

points to ghost variable
SSL: basic rules

(Emp)
\[
\{\text{emp}\} \xrightarrow{\text{skip}} \{\text{emp}\}
\]

(Frame)
\[
\begin{align*}
\{P\} & \xrightarrow{c} \{Q\} \\
\{P \ast R\} & \xrightarrow{c} \{Q \ast R\}
\end{align*}
\]

(Read)
\[
\begin{align*}
[y/a]\{x \mapsto A \ast P\} & \xrightarrow{c} [y/a]\{Q\} \\
\{x \mapsto a \ast P\} & \xrightarrow{\text{let } y = *x; c} \{Q\}
\end{align*}
\]

(Write)
\[
\begin{align*}
\{x \mapsto e \ast P\} & \xrightarrow{c} \{x \mapsto e \ast Q\} \\
\{x \mapsto _\ast P\} & \xrightarrow{c} \{x \mapsto e \ast Q\} \\
\ast x & = e; c
\end{align*}
\]
example: swap

$$\{ x \mapsto a \ast y \mapsto b \} \mapsto \{ x \mapsto b \ast y \mapsto a \} \mid ??$$
\{ x \mapsto a \ast y \mapsto b \} \quad \mapsto \quad \{ x \mapsto b \ast y \mapsto a \} \quad | \quad ??
\[ \{ x \mapsto a1 \cdot y \mapsto b \} \not\twoheadrightarrow \{ x \mapsto b \cdot y \mapsto a1 \} \mid \text{let } a1 = \ast x; \text{ ??} \]

\[ \{ x \mapsto a \cdot y \mapsto b \} \not\twoheadrightarrow \{ x \mapsto b \cdot y \mapsto a \} \mid \text{let } a1 = \ast x; \text{ ??} \]
\[
\begin{align*}
\{ x \mapsto a1 \cdot y \mapsto b1 \} & \implies \{ x \mapsto b1 \cdot y \mapsto a1 \} \mid ?? \\
\{ x \mapsto a1 \cdot y \mapsto b \} & \implies \{ x \mapsto b \cdot y \mapsto a1 \} \mid \text{let } b1 = ^*y; ?? \\
\{ x \mapsto a \cdot y \mapsto b \} & \implies \{ x \mapsto b \cdot y \mapsto a \} \mid \text{let } a1 = ^*x; ??
\end{align*}
\]
\[
\begin{align*}
\{ x \mapsto b1 \ast y \mapsto b1 \} & \leadsto \{ x \mapsto b1 \ast y \mapsto a1 \} \mid \text{??} \\
\{ x \mapsto a1 \ast y \mapsto b1 \} & \leadsto \{ x \mapsto b1 \ast y \mapsto a1 \} \mid \ast x = b1; \text{??} \\
\{ x \mapsto a1 \ast y \mapsto b \} & \leadsto \{ x \mapsto b \ast y \mapsto a1 \} \mid \text{let } b1 = \ast y; \text{??} \\
\{ x \mapsto a \ast y \mapsto b \} & \leadsto \{ x \mapsto b \ast y \mapsto a \} \mid \text{let } a1 = \ast x; \text{??}
\end{align*}
\]
\[
\begin{align*}
\{ y \mapsto b1 \} & \xrightarrow{\text{Read}} \{ y \mapsto a1 \} \mid \text{??} \\
\{ x \mapsto b1 \ast y \mapsto b1 \} & \xrightarrow{\text{Write}} \{ x \mapsto b1 \ast y \mapsto a1 \} \mid \text{??} \\
\{ x \mapsto a1 \ast y \mapsto b1 \} & \xrightarrow{\text{Frame}} \{ x \mapsto b1 \ast y \mapsto a1 \} \mid \text{??} \\
\{ x \mapsto a1 \ast y \mapsto b \} & \xrightarrow{\text{Read}} \{ x \mapsto b \ast y \mapsto a1 \} \mid \text{let } b1 = \ast y; \text{ ??} \\
\{ x \mapsto a \ast y \mapsto b \} & \xrightarrow{\text{Read}} \{ x \mapsto b \ast y \mapsto a \} \mid \text{let } a1 = \ast x; \text{ ??}
\end{align*}
\]
\[
\begin{array}{c}
\{ y \mapsto a_1 \} \leadsto \{ y \mapsto a_1 \} | \ ?? \\
\hline
\{ y \mapsto b_1 \} \leadsto \{ y \mapsto a_1 \} | *y = a_1; ?? \\
\hline
\{ x \mapsto b_1 \ast y \mapsto b_1 \} \leadsto \{ x \mapsto b_1 \ast y \mapsto a_1 \} | ?? \\
\hline
\{ x \mapsto a_1 \ast y \mapsto b_1 \} \leadsto \{ x \mapsto b_1 \ast y \mapsto a_1 \} | *x = b_1; ?? \\
\hline
\{ x \mapsto a_1 \ast y \mapsto b \} \leadsto \{ x \mapsto b \ast y \mapsto a_1 \} | \text{let } b_1 = *y; ?? \\
\hline
\{ x \mapsto a \ast y \mapsto b \} \leadsto \{ x \mapsto b \ast y \mapsto a \} | \text{let } a_1 = *x; ??
\end{array}
\]
\[
\{ \text{emp} \} \xrightarrow{\ } \{ \text{emp} \} \mid ?? \\
\{ y \mapsto a1 \} \xrightarrow{\ } \{ y \mapsto a1 \} \mid ?? \\
\hline
\{ y \mapsto b1 \} \xrightarrow{\ } \{ y \mapsto a1 \} \mid *y = a1; ?? \\
\hline
\{ x \mapsto b1 \ast y \mapsto b1 \} \xrightarrow{\ } \{ x \mapsto b1 \ast y \mapsto a1 \} \mid ?? \\
\hline
\{ x \mapsto a1 \ast y \mapsto b1 \} \xrightarrow{\ } \{ x \mapsto b1 \ast y \mapsto a1 \} \mid *x = b1; ?? \\
\hline
\{ x \mapsto a1 \ast y \mapsto b \} \xrightarrow{\ } \{ x \mapsto b \ast y \mapsto a1 \} \mid \text{let } b1 = *y; ?? \\
\hline
\{ x \mapsto a \ast y \mapsto b \} \xrightarrow{\ } \{ x \mapsto b \ast y \mapsto a \} \mid \text{let } a1 = *x; ??
\]
\[ \{ x \mapsto a \times y \mapsto b \} \]

let \( a_1 = *x \); let \( b_1 = *y \); \( *x = b_1 \); \( *y = a_1 \); skip

\[ \{ x \mapsto b \times y \mapsto a \} \]
demo: deriving swap interactively

```c
void swap(loc x, loc y)
{
    x ↦ a * y ↦ b
}
{ x ↦ b * y ↦ a }
SSL: other rules

Dynamic memory:
   (Free) Free a memory block we have
   (Alloc) Allocate a memory block we need

Predicates:
   (Open) Unfold a predicate in the precondition
   (Close) Unfold a predicate in the postcondition

Existentials:
   (Unify) Unify two heaplets in pre and post
   (Pick) Replace an existential with some variable
demo: bank account

**predicate** account (loc x, int bal) {
    | true ⇒ { bal ≥ 0 ; [x, 1] * x ↦ bal }
}

**void** delete(loc acc)
{ account(acc, b) }
{ emp }

**void** create(loc ret)
{ ret ↦ 0 }
{ ret ↦ acc * account(acc, 0) }
exercise 3: deposit

**predicate** account (loc x, int bal) {
  | true ⇒ { bal ≥ 0 ; [x, 1] * x ↦ bal }
}

void deposit(loc acc, int amt)

{ ??? }

{ ??? }
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
   3.1. proof system
   3.2. recursion and cyclic proofs
   3.3. discovering recursive auxiliaries
dispose a list

dispose(loc x)
{ list(x) } { emp }

predicate list (loc x) {
| x = 0 ⇒ { emp } |
| x ≠ 0 ⇒ { \([x, 2]\) ∗ x ↦ v ∗ (x + 1) ↦ y ∗ list(y) } { emp }
}

how do we generate a recursive call?

(Emp)

{ emp } { emp } | skip

(Free)

committee (Emp)

(Free) free(x) (Open)

dispose(loc x)
{ list(x) } { emp }

if (x == 0) {...} else {...}
cyclic proofs

proof “trees” with backlinks

verification of recursive programs:

[Brotherston et al. POPL’08]
[Rowe, Brotherston. CPP’17]

synthesis of recursive programs:

[Itzhaky, Peleg, Polikarpova, Rowe, Sergey. PLDI’21]
dispose a list

dispose(loc x)
{ list(x) }
{ emp }

bud

backlink generates a recursive call with what arguments?

companion

if x == 0 {...} else {...}
call abduction

\[
\begin{align*}
\{ \text{emp} \} & \leadsto \{ \text{emp} \} \quad \text{skip} \\
\{ \text{emp} \} & \leadsto \{ \text{emp} \} \quad \text{dispose(y) \{ emp \} \leadsto \{ emp \}} \\
\{ \text{list(y)} \} & \leadsto \{ \text{list(y)} \} \quad \text{dispose(y) \{ emp \} \leadsto \{ emp \}} \\
\{ \text{list(y)} \} & \leadsto \{ \text{list(x')} \} \quad \text{dispose(x')} \{ emp \} \leadsto \{ emp \} \\
\{ \text{list(y)} \} & \leadsto \{ \text{emp} \} \\
\{ \text{list(y)} \} & \leadsto \{ \text{emp} \} \quad \text{free(x)} \\
\{ \text{list(x)} \} & \leadsto \{ \text{emp} \} \quad \text{if } x = 0 \{ \ldots \} \text{ else } \{ \ldots \}
\end{align*}
\]
demo: deriving dispose

```c
void dispose(loc x)
{
  list(x)
}
{
  emp
}
```
exercise 4: deriving single-to-double

```c
void sll_to_dll(loc x, loc prev)
{
    sll_reserved(x, S)
}
{
    dll(x, prev, S)
}
```
this tutorial

1. example: swap

2. intro to separation logic

3. deductive synthesis
   3.1. proof system
   3.2. recursion and cyclic proofs
   3.3. discovering recursive auxiliaries
task: flatten a tree into a list
task: flatten a tree into a list (in place)
task: flatten a tree into a list (in place)
task: flatten a tree into a list (in place)
task: flatten a tree into a list (in place)
task: flatten a tree into a list (in place)
example: tree flattening

```c
void flatten(loc x)
{
  tree(x, S)
}
{
  dll(x, y, S)
}
```

set of elements

back pointer (irrelevant)
this task is challenging!

```c
void flatten(loc x)
{ tree(x, S) }
{ dll(x, y, S) }
```

Leon
[Kneuss et al’13]

Synquid

ImpSynth

SuSLik’19
[Polikarpova, Sergey’19]
tree flattening

\[
\text{flatten}(x) \begin{cases} 
\end{cases}
\]

\[
\xrightarrow{\text{äm}}
\]

\[
\begin{cases} 
\end{cases}
\]

\[
\xrightarrow{\text{äm}}
\]

\[
\begin{cases} 
\end{cases}
\]
tree flattening

\[
\text{flatten}(x) \{
\quad \text{if } (x \neq 0) \{
\quad \quad l = \ast(x + 1); \quad r = \ast(x + 2);
\quad \}\}
\]

\[
\begin{array}{c}
\{ \quad \{ \quad l \quad r \quad \} \quad \rightarrow \quad \{ \quad \}\quad x \\
\end{array}
\]
tree flattening

flatten(x) {
    if (x != 0) {
        l = *(x + 1); r = *(x + 2);
        flatten(l);
    }
}

\[
x \rightarrow \{ \begin{cases} 
  l \\
  r
\end{cases} \rightarrow \{ \begin{cases} 
  \text{ } \\
  \text{ }
\end{cases} \}
\]

\[
x \rightarrow \{ \begin{cases} 
  \text{ } \\
  \text{ }
\end{cases} \}
\]
tree flattening

```c
flatten(x) {
    if (x != 0) {
        l = *(x + 1); r = *(x + 2);
        flatten(l); flatten(r);
    }
}
```

needs recursive function to append two lists!
tree flattening with cyclic proofs

```
flatten(x) {
    if (x != 0) {
        ...
    }
}
```
tree flattening with cyclic proofs

```c
flatten(x) {
  if (x != 0) {
    ...
    if (l == 0) { ... } else {
      n = *(l + 1);
    }
  }

```

```
x
{ l n r } ~w~ { }
```

```c
}
}
```
does this goal look familiar?

```plaintext
flatten(x) {
    if (x != 0) {
        ...
        if (l == 0) { ... } else {
            n = *(l + 1);
        }
    }
}
```
let’s cycle back!

flatten(x) {
    if (x != 0) {
        ...
        if (l == 0) { ...
        } else {
            n = *(l + 1);
            ...
        }
    }
}

helper(n, r, l)
let’s cycle back!

```javascript
flatten(x) {
  if (x != 0) {
    ...
    if (l == 0) { ... } else {
      n = *(l + 1);
      helper(n, r, l);
    }
  }
}
```
extracting the auxiliary

```c
flatten(x) {
    if (x != 0) {
        ...
        if (l == 0) { ... } else {
            n = *(l + 1);
            helper(n, r, l);
            ...
        }
    }
}
```
extracting the auxiliary

```plaintext
flatten(x) {
    if (x != 0) {
        ...
        if (l == 0) { ... } else {
            n = *(l + 1);
            helper(n, r, l);
            ...
        }
    }
}

helper(l, r, x) {
    ...
}
```
extracting the auxiliary

\[
\text{flatten}(x) \{ \\
\quad \text{if } (x \neq 0) \{ \\
\quad \quad l = *(x + 1); \; r = *(x + 2); \\
\quad \quad \text{flatten}(l); \; \text{flatten}(r); \\
\quad \quad \text{helper}(l, r, x); \\
\quad \} \\
\} \\
\]

\[
\text{helper}(l, r, x) \{ \\
\quad \text{if } (l == 0) \{ \ldots \} \; \text{else} \{ \\
\quad \quad n = *(l + 1); \\
\quad \quad \text{n = } *(l + 1); \\
\quad \quad \text{helper}(n, r, l); \\
\quad \quad \text{helper}(n, r, l); \\
\quad \} \\
\} \\
\]
deductive synthesis

idea: to find the program, look for the proof
void flatten(loc x)

{ tree(x, S) }

{ dll(x, y, S) }
exercise 5: dispose a rose tree

```c
void dispose(loc x)
{
  rtree(x);

  emp;
}
```
SuSLik

= Synthesis using Separation Logik

- separation logic
- deductive synthesis
- C code + proof

- reasoning about pointers & aliasing
- uses specs to guide synthesis

2. Cyclic Program Synthesis. Itzhaky, Peleg, Polikarpova, Rowe, Sergey. [PLDI’21]


thanks to my collaborators!

Ilya Sergey (Yale-NUS)  Shachar Itzhaky (Technion)  Reuben Rowe (Royal Holloway)  Hila Peleg (Technion)  Andreea Costea (Yale-NUS)

Amy Zhu  Roman Shchedrin  Yasunari Watanabe  Kiran Gopinathan  George Pîrlea