

SuSLik: deductive synthesis of safe programs with pointers

Nadia Polikarpova (UCSD)
Shachar Itzhaky (Technion)



UCSDCSE
Computer Science and Engineering

follow along

<https://github.com/TyGuS/suslik-tutorial>

we've come a long way...

Checking a Large Routine

Alan Turing

```
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;  
}
```



CompCert:
verified C compiler



Project Everest:
verified HTTPS stack



verified microkernel



CERTIKOS

verified OS kernel

1949

today

... but this is still hard

Checking a Large
Routine

Alan Turing



CERTIKOS

have to write code, specs, and proofs!

1949

today

what's next for verified systems?

Checking a Large
Routine

Alan Turing



Security. Performance. Proof.



CERTIKOS



1949

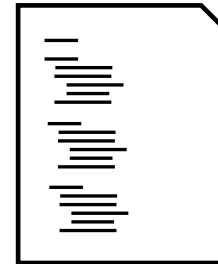
today

program synthesis

specification



code



program synthesis **with guarantees**

formal
specification



code + proof



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

[Kneuss et al'13]

SuSLik

[Polikarpova, Sergey'19]

Synquid

[Polikarpova et al'16]

VS3

[Srivastava et al'10]

ImpSynth

[Qui, Solar-Lezama'18]

Fiat

[Delaware et al'15]

Jennisys

[Leino, Milicevic'12]



deductive synthesis of **pointer-manipulating** programs

A Deductive Approach
to Program Synthesis

[Manna, Waldinger]

Leon

[Polikarpova, Sergey. POPL'19]

VS3

Fiat

SuSLik



Synquid

ImpSynth

Jennisys

1980

today

SuSLik

specification



C code
+ proof



SuSLik

specification



C code
+ proof



SuSLik

specification



C code
+ proof



- ☹ verbose
- ☹ unstructured
- ☹ pointers & aliasing

SuSLik

= **S**ynthesis **u**sing **S**eparation **L**ogik

[Reynolds. LICS'02]

separation
logic



☺ reasoning about
pointers & aliasing



deductive
synthesis



☺ uses specs
to guide synthesis



C code
+ proof



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

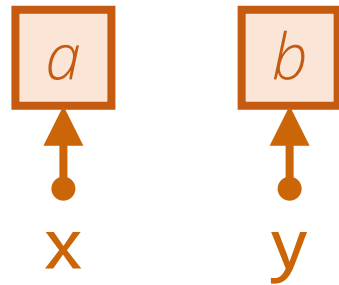
example: swap

swap values of two *distinct* pointers

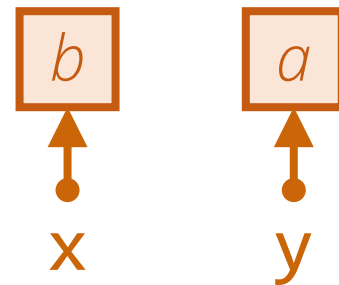
```
void swap(loc x, loc y)
```

example: swap

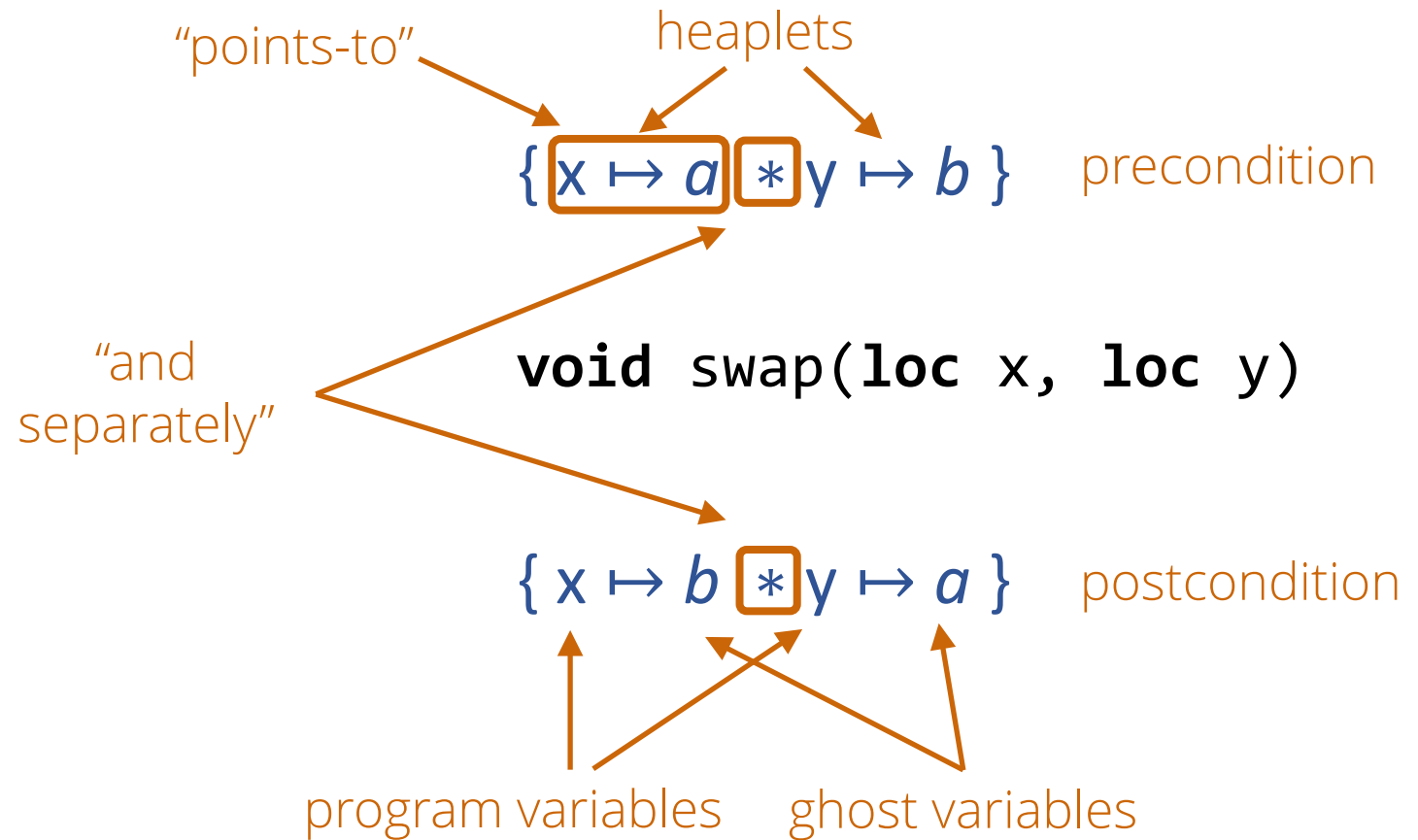
start state:



end state:



in separation logic:



demo: swap

```
void swap(loc x, loc y)
```

```
{  $x \mapsto a * y \mapsto b$  }
```

```
{  $x \mapsto b * y \mapsto a$  }
```

how did this happen?

deductive synthesis

idea: to find the program, look for the proof



use spec to guide search!



$$\{x \mapsto a * y \mapsto b\}$$

??

$$\{x \mapsto b * y \mapsto a\}$$

let a1 = *x;

{ x $\mapsto$ a1 * y $\mapsto$ b }

??

{ x $\mapsto$ b * y $\mapsto$ a1 }

let a1 = *x;

let b1 = *y;

{ x $\mapsto$ a1 * y $\mapsto$ b1 }

??

{ x $\mapsto$ b1 * y $\mapsto$ a1 }

let a1 = *x;

let b1 = *y;

*x = b1;

{ x $\mapsto$ b1 * y $\mapsto$ b1 }

??

{ x $\mapsto$ b1 * y $\mapsto$ a1 }

let a1 = *x;

let b1 = *y;

*x = b1;

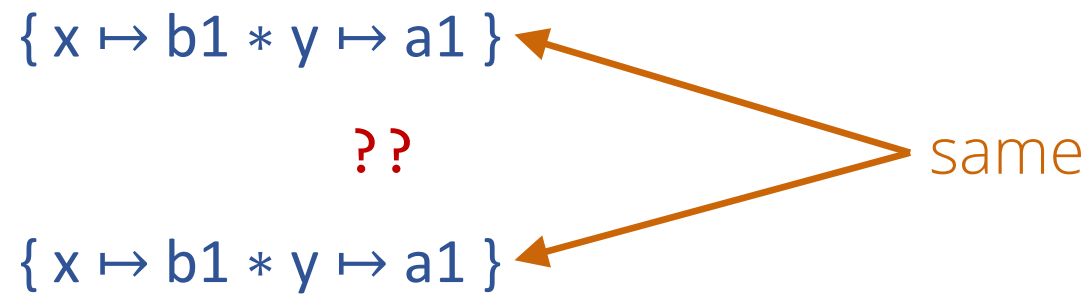
*y = a1;

{ x $\mapsto$ b1 * y $\mapsto$ a1 }

??

{ x $\mapsto$ b1 * y $\mapsto$ a1 }

same



```
let a1 = *x;
```

```
let b1 = *y;
```

```
*x = b1;
```

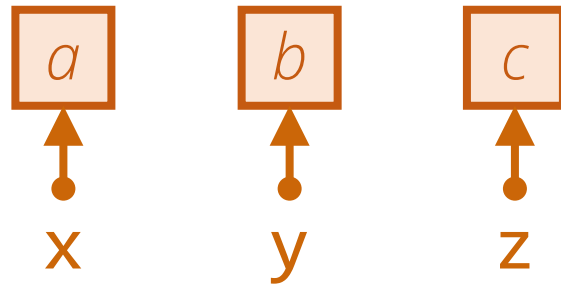
```
*y = a1;
```



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

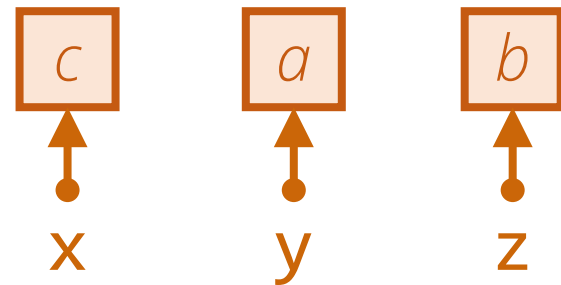
exercise 1: rotate three

start state:



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

end state:



this tutorial

1. example: swap

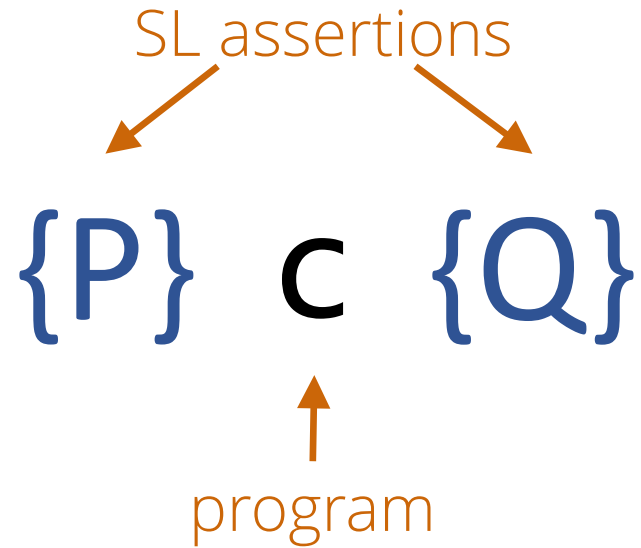
2. intro to separation logic

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)

$$\{P\} \quad c \quad \{Q\}$$

↑
program

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

allocate block

skip

let $y = *(x + n)$

$*(x + n) = e$

let $y = \text{malloc}(n)$

programs

do nothing

read from heap

write to heap

allocate block

free block

skip

let $y = *(x + n)$

$*(x + n) = e$

let $y = \text{malloc}(n)$

free(x)

programs

do nothing

read from heap

write to heap

allocate block

free block

procedure call

skip

let $y = *(x + n)$

$*(x + n) = e$

let $y = \mathbf{malloc}(n)$

free(x)

$p(e_1, \dots, e_n)$

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)

$p(e_1, \dots, e_n)$

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 $y = *(x + n)$

$*(x + n) = e$

let $y = \mathbf{malloc}(n)$

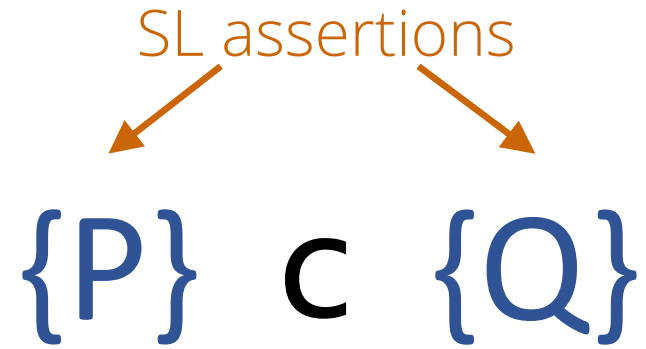
free(x)

$p(e_1, \dots, e_n)$

$c_1 ; c_2$

if (e) $\{c_1\}$ else $\{c_2\}$

separation logic (SL)



SL assertions

empty heap { emp }

SL assertions

empty heap $\{ \text{emp} \}$

singleton heap $\{ y \mapsto 5 \}$



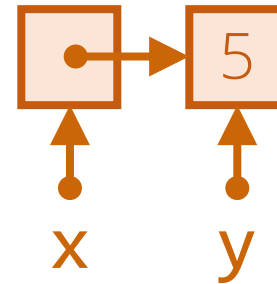
SL assertions

empty heap $\{ \text{emp} \}$

singleton heap $\{ y \mapsto 5 \}$

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

 ↙ ↘
 heaplets



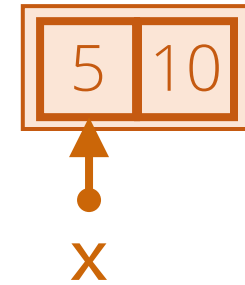
SL assertions

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 $\{ x \mapsto y * y \mapsto 5 \}$

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

+ pure formula $\{ a > 5 ; x \mapsto a \}$



separation logic (SL)

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

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

$\{x \mapsto a\}$

$*x = 5$

$\{x \mapsto 5\}$

$\{x \mapsto a\}$

$*(x + 1) = 5$



$\{x \mapsto a\}$

let $y = *x$

$\{x \mapsto y\}$

$\{\text{emp}\}$

let $y = \text{malloc}(2)$

$\{[y, 2] * y \mapsto a * (y + 1) \mapsto b\}$

$\{[x, 2] * x \mapsto 5 * (x + 1) \mapsto 7\}$

free $(x + 1)$



this tutorial

1. example: swap
2. intro to separation logic
 - 2.1. programs and assertions
 - 2.2. specifying data transformations
3. deductive synthesis

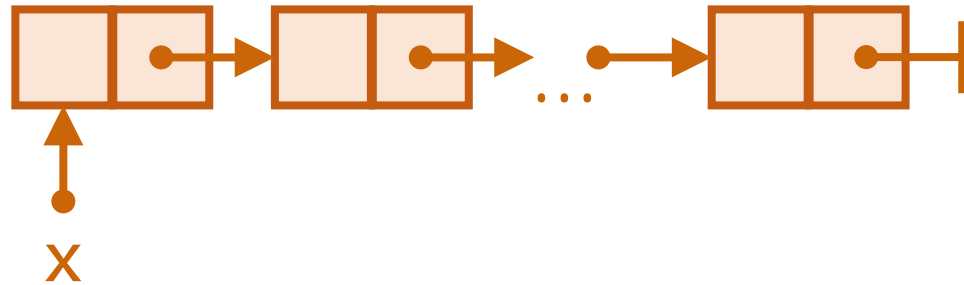
example: dispose

deallocate a linked list with head pointer x

```
void dispose(loc x)
```

dynamic data structures

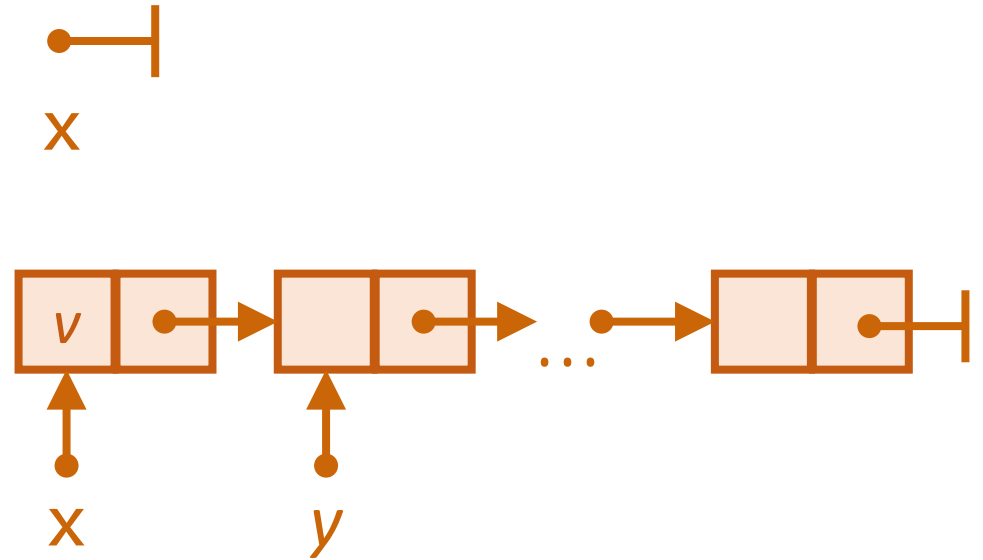
how do we specify a linked list?



inductive predicates to the rescue!

the linked list predicate

```
predicate list (loc x) {  
  |  $x = 0 \Rightarrow \{ \text{emp} \}$   
  |  $x \neq 0 \Rightarrow \{ [x, 2]$   
      *  $x \mapsto v$   
      *  $(x + 1) \mapsto y$   
      * list(y)  
  }  
}
```



demo: dispose a list

```
void dispose(loc x)
```

```
{ list(x) }
```

```
{ emp }
```

what if we care about content?

example: copy

copy a linked list with head pointer x

```
void copy(loc x, loc ret)
```

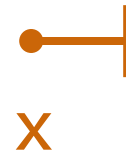
linked list with elements

predicate list (loc x , set S) {
| $x = 0 \Rightarrow \{ S = \emptyset ; \text{emp} \}$

pure part

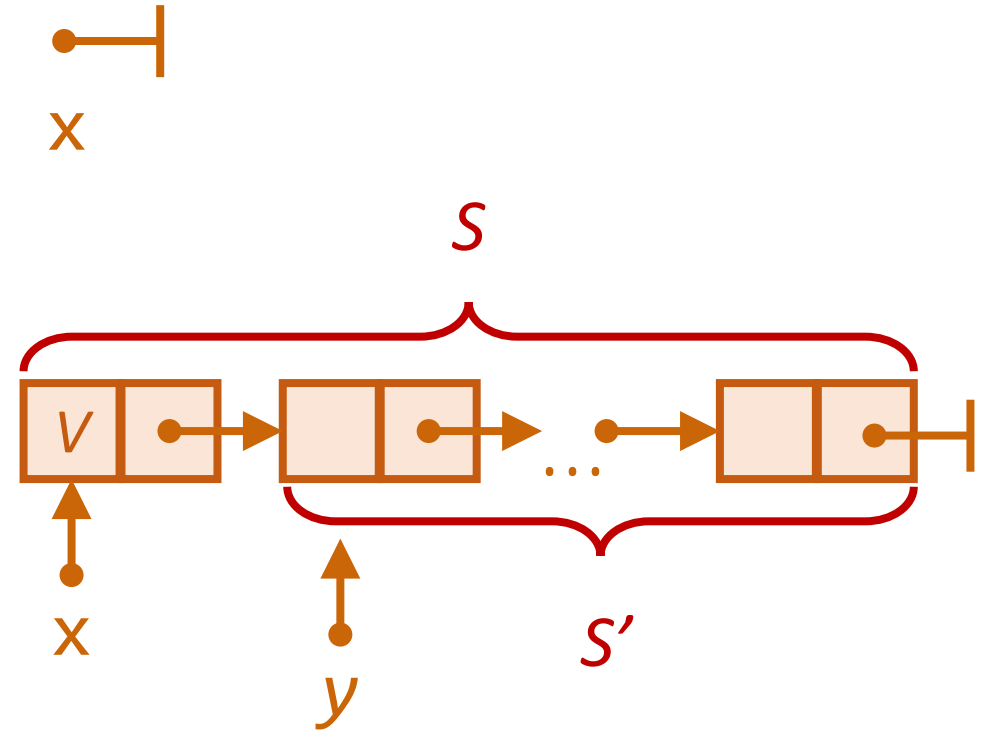
spatial part

}



linked list with elements

```
predicate list (loc x, set S) {  
  |  $x = 0 \Rightarrow \{ S = \emptyset ; \text{emp} \}$   
  |  $x \neq 0 \Rightarrow \{ S = \{v\} + S' ;$   
     $[x, 2]$   
     $* x \mapsto v * (x + 1) \mapsto y$   
     $* \text{list}(y, S') \}$   
}
```



demo: copy a list

return location
↓
void copy(**loc** x, **loc** ret)
{ list(x, S) * ret $\mapsto$ _ }
{ list(x, S) * ret $\mapsto$ *y* * list(*y*, S) }
↑
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

$$P \rightsquigarrow Q \mid c$$

a state that satisfies P
can be transformed into a state that satisfies Q
using a program c

(Emp)

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

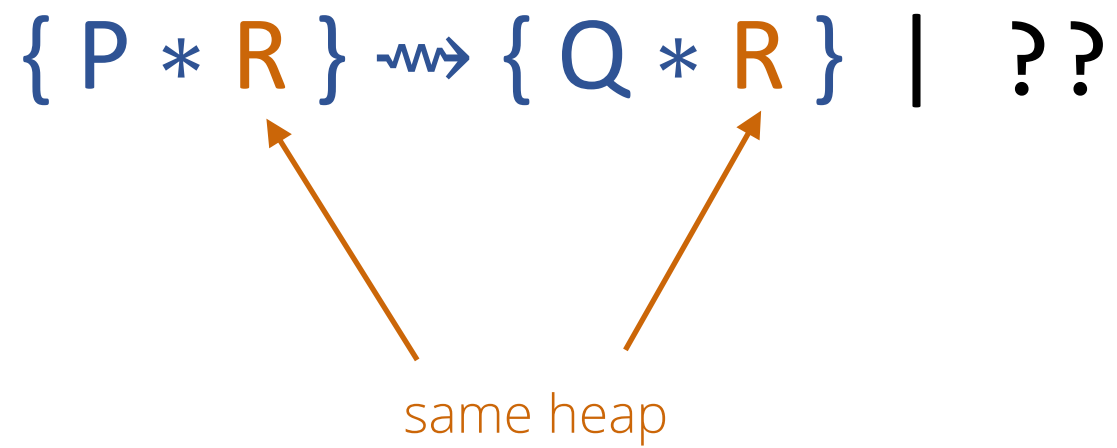
empty heap



(Emp)

$\{\text{emp}\} \rightsquigarrow \{\text{emp}\} \mid \mathbf{skip}$

(Frame)

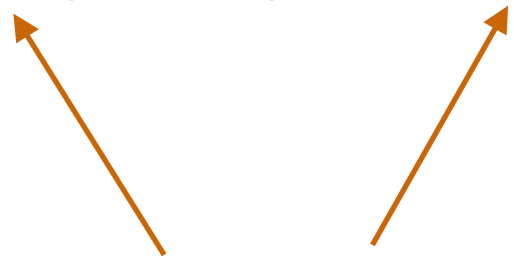


(Frame)

$$\{ P \} \rightsquigarrow \{ Q \} \mid c$$

$$\{ P * R \} \rightsquigarrow \{ Q * R \} \mid c$$

same heap



(Write)

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

points to non-ghost expression

(Write)

$$\frac{\{x \mapsto e * P\} \rightsquigarrow \{x \mapsto e * Q\} \mid c}{\{x \mapsto _ * P\} \rightsquigarrow \{x \mapsto e * Q\} \mid *x = e; c}$$

↑
points to non-ghost expression

(Read)

$$\{ x \mapsto a * P \} \rightsquigarrow \{ Q \}$$

| ??

points to ghost variable

(Read)

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

$$\{ x \mapsto a * P \} \rightsquigarrow \{ Q \}$$

| let $y = *x$; c

points to ghost variable

SSL: basic rules

(Emp)

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

(Frame)

$$\frac{\{P\} \rightsquigarrow \{Q\} \mid c}{\{P * R\} \rightsquigarrow \{Q * R\} \mid c}$$

(Read)

$$\frac{[y/a]\{x \mapsto A * P\} \rightsquigarrow [y/a]\{Q\} \mid c}{\{x \mapsto a * P\} \rightsquigarrow \{Q\} \mid \text{let } y = *x; c}$$

(Write)

$$\frac{\{x \mapsto e * P\} \rightsquigarrow \{x \mapsto e * Q\} \mid c}{\{x \mapsto _ * P\} \rightsquigarrow \{x \mapsto e * Q\} \mid *x = e; c}$$

example: swap

$$\{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid \quad ??$$

$$\{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \quad ??$$

$$\{x \mapsto a1 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a1\} \mid \quad ??$$

$$\{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid \quad \mathbf{let} \ a1 = *x; \quad ??$$

(Read)

$$\{x \mapsto a1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid ??$$

(Read)

$$\{x \mapsto a1 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a1\} \mid \text{let } b1 = *y; ??$$

(Read)

$$\{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid \text{let } a1 = *x; ??$$

$$\{x \mapsto b1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid ??$$

(Write)

$$\{x \mapsto a1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid *x = b1; ??$$

(Read)

$$\{x \mapsto a1 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a1\} \mid \text{let } b1 = *y; ??$$

(Read)

$$\{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid \text{let } a1 = *x; ??$$

$$\{y \mapsto b1\} \rightsquigarrow \{y \mapsto a1\} \mid ??$$

(Frame)

$$\{x \mapsto b1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid ??$$

(Write)

$$\{x \mapsto a1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid *x = b1; ??$$

(Read)

$$\{x \mapsto a1 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a1\} \mid \text{let } b1 = *y; ??$$

(Read)

$$\{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid \text{let } a1 = *x; ??$$

$$\{y \mapsto a1\} \rightsquigarrow \{y \mapsto a1\} \mid ??$$

$$\{y \mapsto b1\} \rightsquigarrow \{y \mapsto a1\} \mid *y = a1; ??$$

$$\{x \mapsto b1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid ??$$

$$\{x \mapsto a1 * y \mapsto b1\} \rightsquigarrow \{x \mapsto b1 * y \mapsto a1\} \mid *x = b1; ??$$

$$\{x \mapsto a1 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a1\} \mid \text{let } b1 = *y; ??$$

$$\{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\} \mid \text{let } a1 = *x; ??$$

$$\begin{array}{c}
\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid ?? \\
\hline
\{ y \mapsto a1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid ?? \quad \text{(Frame)} \\
\hline
\{ y \mapsto b1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid *y = a1; ?? \quad \text{(Write)} \\
\hline
\{ x \mapsto b1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid ?? \quad \text{(Frame)} \\
\hline
\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid *x = b1; ?? \quad \text{(Write)} \\
\hline
\{ x \mapsto a1 * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a1 \} \mid \text{let } b1 = *y; ?? \quad \text{(Read)} \\
\hline
\{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \text{let } a1 = *x; ?? \quad \text{(Read)}
\end{array}$$

$\frac{}{\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \text{skip}}$	(Emp)
$\frac{}{\{ y \mapsto a1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid ??}$	(Frame)
$\frac{}{\{ y \mapsto b1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid *y = a1; ??}$	(Write)
$\frac{}{\{ x \mapsto b1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid ??}$	(Frame)
$\frac{}{\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid *x = b1; ??}$	(Write)
$\frac{}{\{ x \mapsto a1 * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a1 \} \mid \text{let } b1 = *y; ??}$	(Read)
$\frac{}{\{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \text{let } a1 = *x; ??}$	(Read)

$\{ x \mapsto a * y \mapsto b \}$

let a1 = *x; **let** b1 = *y; *x = b1; *y = a1; **skip**

$\{ x \mapsto b * y \mapsto a \}$

demo: deriving swap interactively

```
void swap(loc x, loc y)
```

```
{  $x \mapsto a * y \mapsto b$  }
```

```
{  $x \mapsto b * y \mapsto a$  }
```

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  $\Rightarrow$  { bal  $\geq$  0 ; [x, 1] * x  $\mapsto$  bal }  
}
```

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

```
void create(loc ret)  
{ ret  $\mapsto$  0 }  
{ ret  $\mapsto$  acc * account(acc, 0) }
```

exercise 3: deposit

```
predicate account (loc x, int bal) {  
  | true  $\Rightarrow$  { bal  $\geq$  0 ; [x, 1] * x  $\mapsto$  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 ⇒
  | x ≠ 0 ⇒
}
```

how do we generate a recursive call?

(Emp)	{ emp } $\rightsquigarrow$ { emp } skip	$\{ [x, 2] * x \mapsto v * (x + 1) \mapsto y \} * \text{list}(y) \}$ $\rightsquigarrow$ { emp } free(x)	(Free) (Open)
	$\{ \text{list}(x) \} \rightsquigarrow \{ \text{emp} \} \mid \text{if } (x == 0) \{ \dots \} \text{ else } \{ \dots \}$		

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?

(Emp)

$\{ \text{list}(y) \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{dispose}(y)$

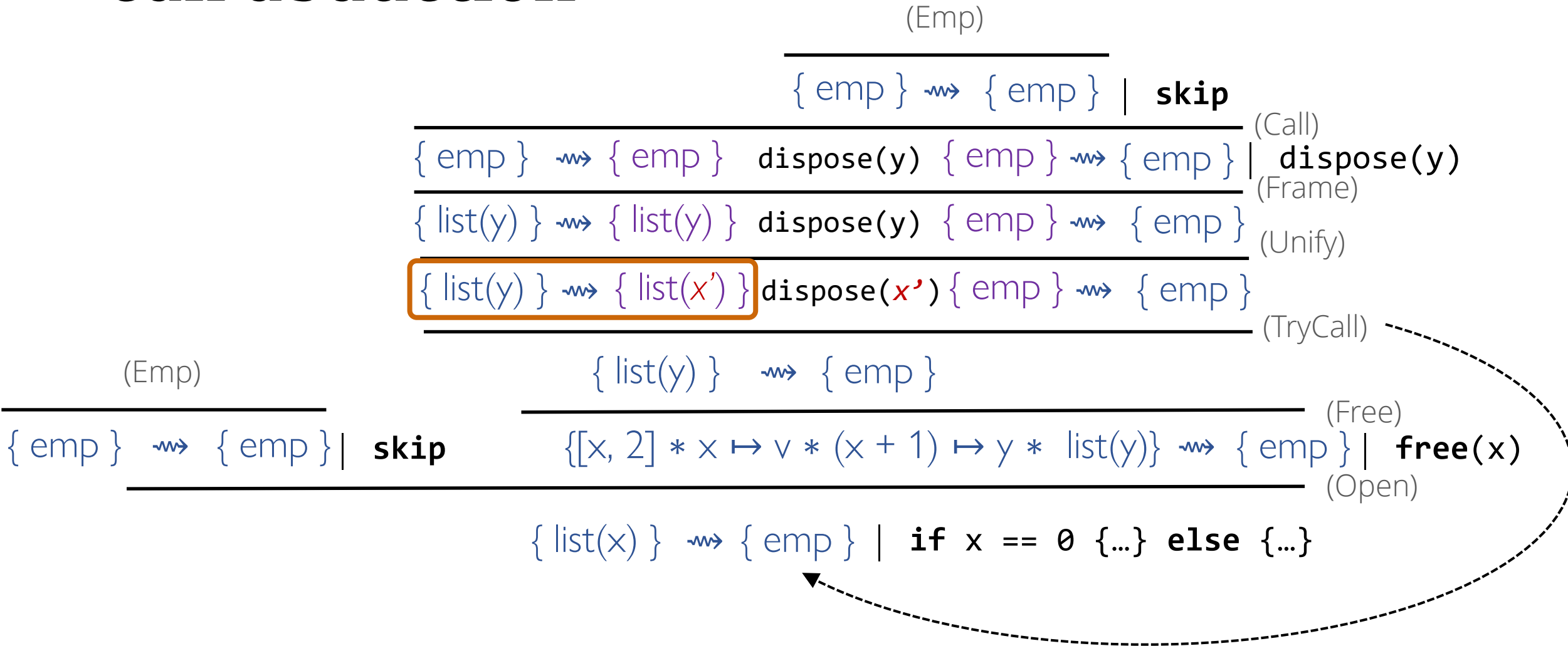
$\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip}$

$\{ [x, 2] * x \mapsto v * (x + 1) \mapsto y * \text{list}(y) \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{free}(x)$
(Free)
(Open)

$\{ \text{list}(x) \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{if } x == 0 \{ \dots \} \mathbf{else } \{ \dots \}$

companion

call abduction



demo: deriving dispose

```
void dispose(loc x)
```

```
{ list(x) }
```

```
{ emp }
```

hard!

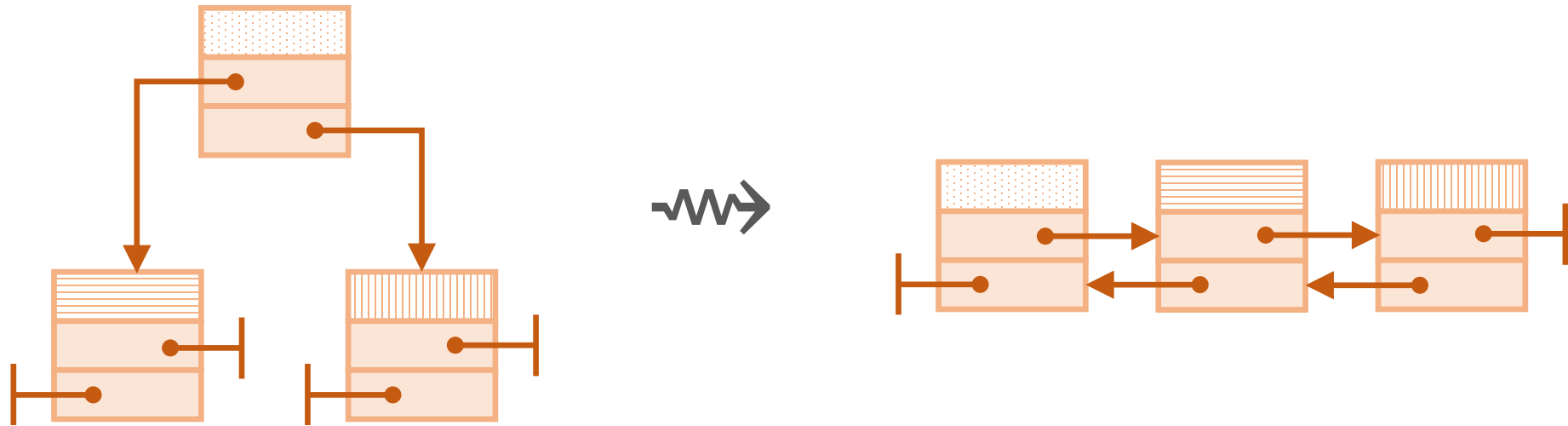
exercise 4: deriving single-to-double

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

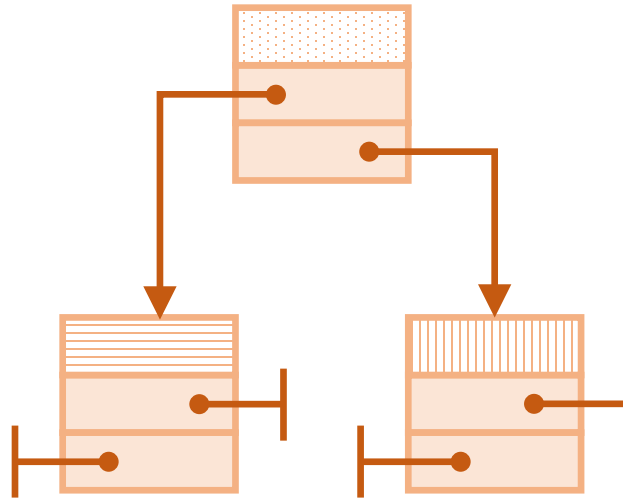
this tutorial

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

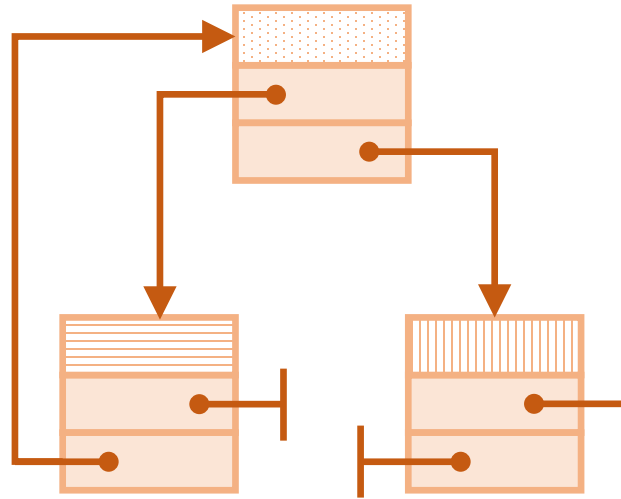
task: flatten a tree into a list



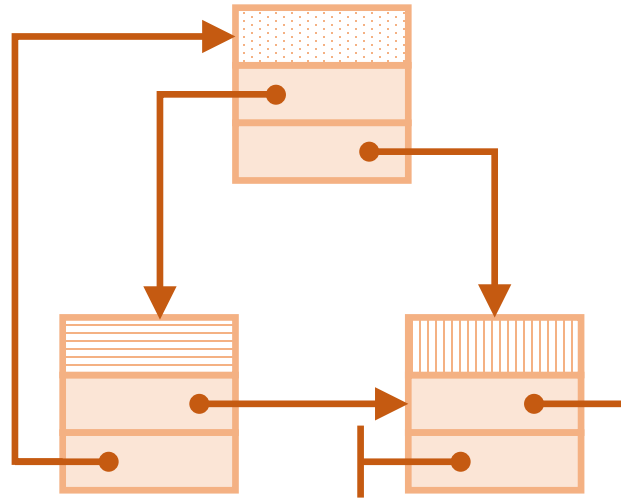
task: flatten a tree into a list (in place)



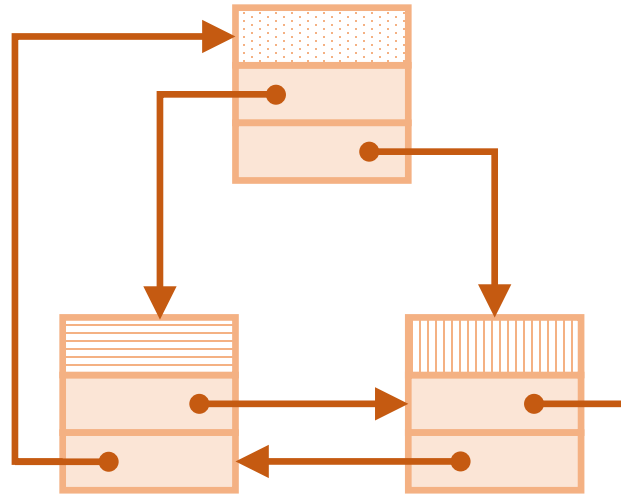
task: flatten a tree into a list (in place)



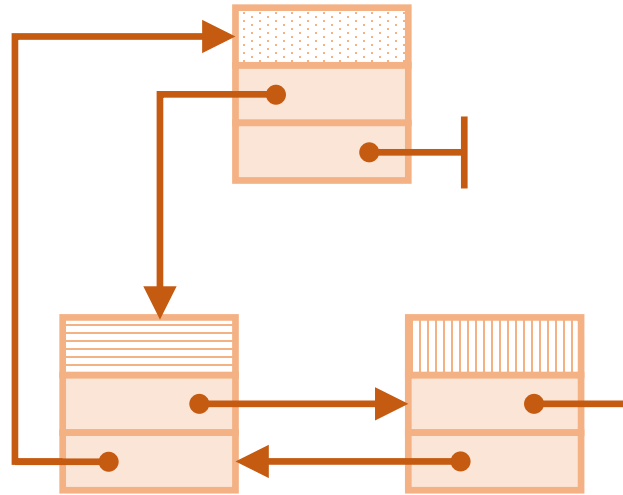
task: flatten a tree into a list (in place)



task: flatten a tree into a list (in place)



task: flatten a tree into a list (in place)



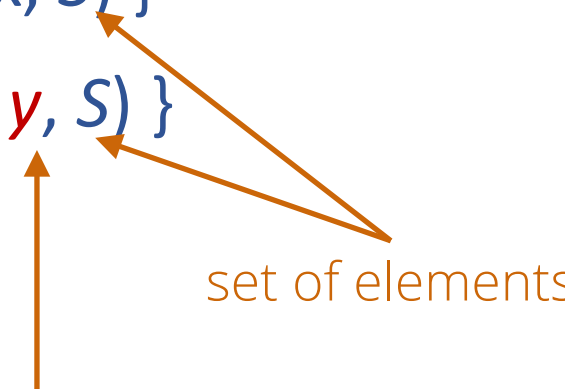
example: tree flattening

```
void flatten(loc x)
```

```
{ tree(x, S) }
```

```
{ dll(x, y, S) }
```

back pointer
(irrelevant)



set of elements

this task is challenging!

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

Leon

[Kneuss et al'13]

Synquid

unable to discover recursive auxiliaries!

ImpSynth

[Qui, Solar-Lezama'18]

SuSLik'19

[Polikarpova, Sergey'19]



tree flattening

```
flatten(x) {
```



```
}
```

tree flattening

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



```
}  
}
```

tree flattening

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



```
}  
}
```

tree flattening

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

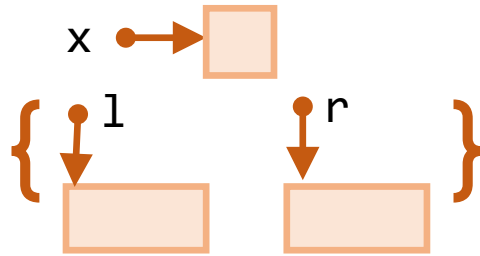


needs recursive function
to append two lists!

}

tree flattening **with cyclic proofs**

```
flatten(x) {  
  if (x != 0) {  
    ...  
  }  
}
```

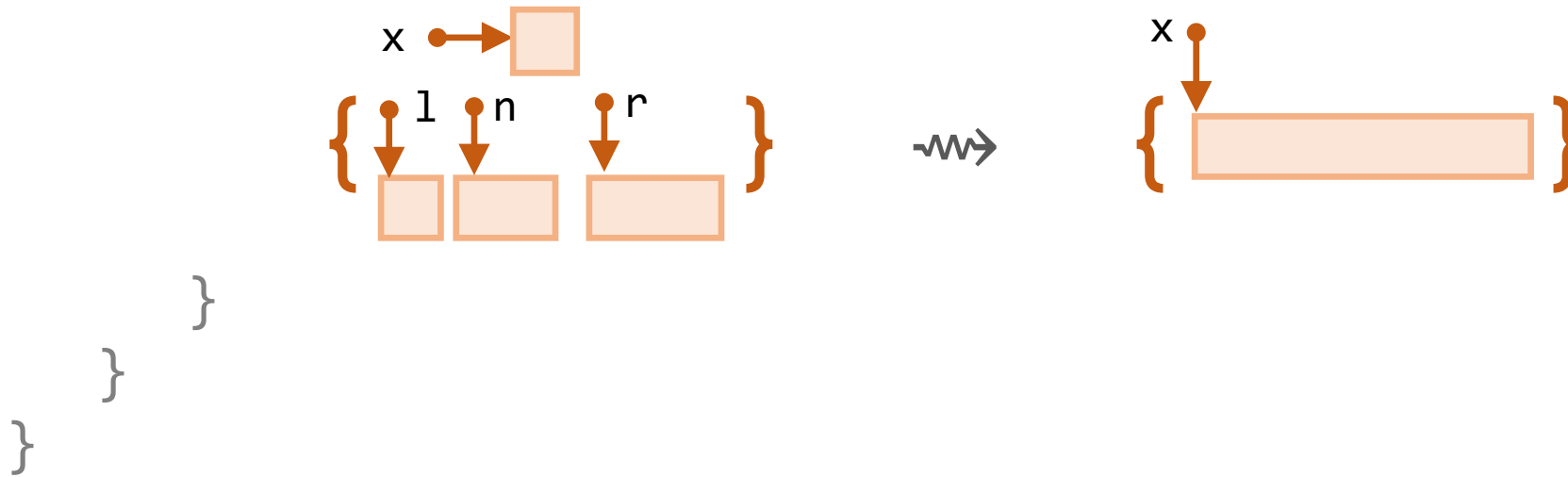


```
}
```

```
}
```

tree flattening **with cyclic proofs**

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



does this goal look familiar?

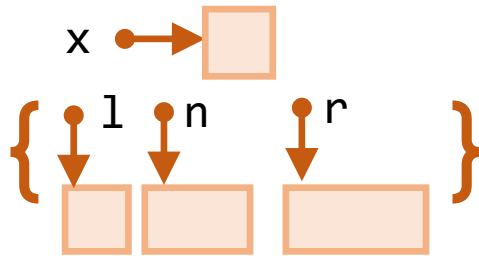
```
flatten(x) {  
  if (x != 0) {  
    ...  
    if (l == 0) { ... } else {  
      n = *(l + 1);  
    }  
  }  
}
```



```
}  
}  
}
```

let's cycle back!

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



$\rightsquigarrow$



helper(n, r, l)



let's cycle back!

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



```
}  
}  
}
```

extracting the auxiliary

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



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

extracting the auxiliary

```
flatten(x) {  
  if (x != 0) {  
    ...  
  }
```

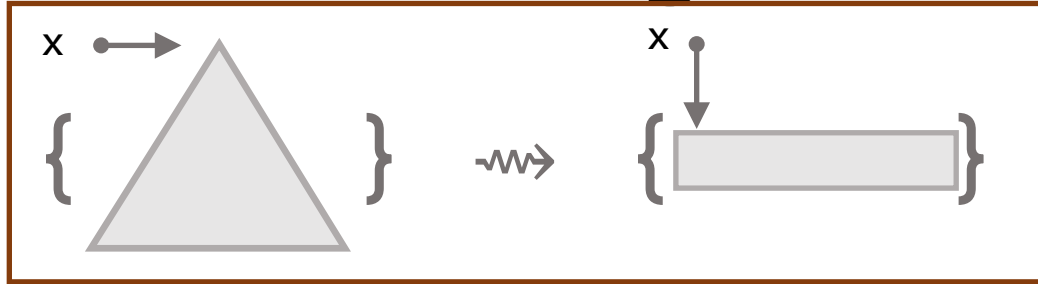
```
    if (l == 0) { ... } else {  
      n = *(l + 1);  
      helper(n, r, l);  
      ...  
    }
```

```
  }  
}
```

```
helper(l, r, x) {
```



extracting the auxiliary



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



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

deductive synthesis

idea: to find the program, look for the proof



demo: tree flattening

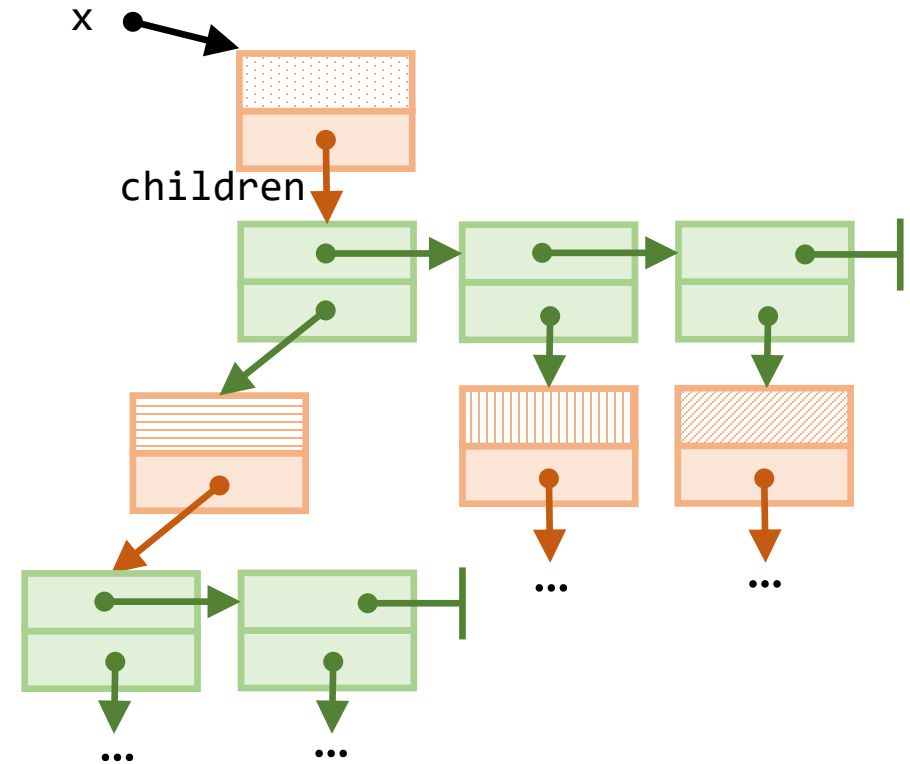
```
void flatten(loc x)
```

```
{ tree(x, S) }
```

```
{ dll(x, y, S) }
```

exercise 5: dispose a rose tree

```
void dispose(loc x)
{ rtree(x) }
{ emp }
```



SuSLik

= **S**ynthesis **u**sing **S**eparation **L**ogik

separation
logic



☺ reasoning about
pointers & aliasing

deductive
synthesis



☺ uses specs
to guide synthesis

C code
+ proof



read more

1. *Structuring the Synthesis of Heap-Manipulating Programs.*
Polikarpova, Sergey [POPL'19]
2. *Cyclic Program Synthesis.*
Itzhaky, Peleg, Polikarpova, Rowe, Sergey. [PLDI'21]
3. *Concise Read-Only Specifications for Better Synthesis of Programs with Pointers.*
Costea, Zhu, Polikarpova, Sergey [ESOP'20]
4. *Certifying the Synthesis of Heap-Manipulating Programs.*
Watanabe, Gopinathan, Pîrlea, Polikarpova, Sergey [ICFP'21]
5. *Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities (Invited Paper).*
Itzhaky, Peleg, Polikarpova, Rowe, Sergey. [CAV'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