

synthesis of safe pointer-manipulating programs

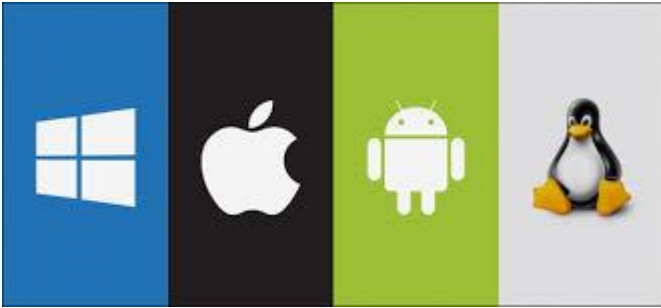
Nadia Polikarpova

with Ilya Sergey (Yale-NUS), Shachar Itzhaky (Technion),
Reuben Rowe (Royal Holloway), and Hila Peleg (UCSD/Technion)



UCSDCSE
Computer Science and Engineering

pointer-manipulating programs



operating systems

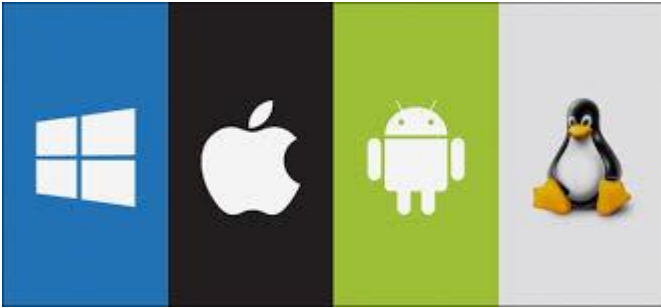


network / security
protocols



browsers

pointer-manipulating programs



operating systems



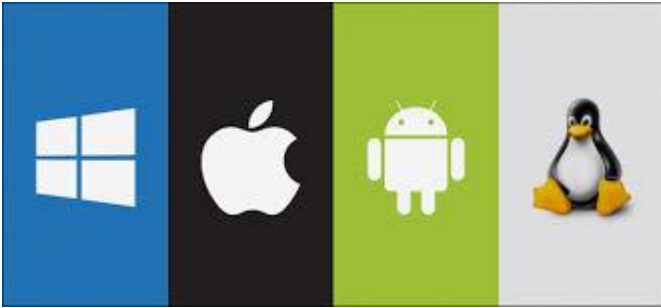
network / security
protocols



browsers

😊 efficient

pointer-manipulating programs



operating systems

😊 efficient



network / security
protocols

☹️ have bugs



browsers

solution: verified systems



CompCert:
verified C compiler



SEL4:
verified microkernel



Project Everest:
verified HTTPS stack



DeepSpec (CertiKOS,
Vellvm, DeepWeb)

solution: verified systems



CompCert:
verified C compiler



SEL4:
verified microkernel



Project Everest:
verified HTTPS stack



DeepSpec (CertiKOS,
Vellvm, DeepWeb)

😊 no bugs

solution: verified systems



CompCert:
verified C compiler



SEL4:
verified microkernel



Project Everest:
verified HTTPS stack



DeepSpec (CertiKOS,
Vellvm, DeepWeb)

😊 no bugs

😞 hard to write

(have to write code, specs, and proofs!)

program synthesis to the rescue

specification



program synthesis to the rescue

specification



code + proof



program synthesis to the rescue

specification



code + proof



😊 easy to write

😊 verified

program synthesis to the rescue

specification



😊 easy to write



code + proof



😊 verified

program synthesis to the rescue

specification



😊 easy to write



code + proof



😊 verified

- ☹ verbose
- ☹ unstructured
- ☹ pointers & aliasing

SuSLik



SuSLik



(Synthesis using Separation Logik)

the SuSLik approach

specification



code + proof



the SuSLik approach

specification



code + proof



the SuSLik approach

separation
logic



code + proof



the SuSLik approach

separation
logic



code + proof



☺ reasoning about
pointers & aliasing

the SuSLik approach

separation
logic



☺ reasoning about
pointers & aliasing



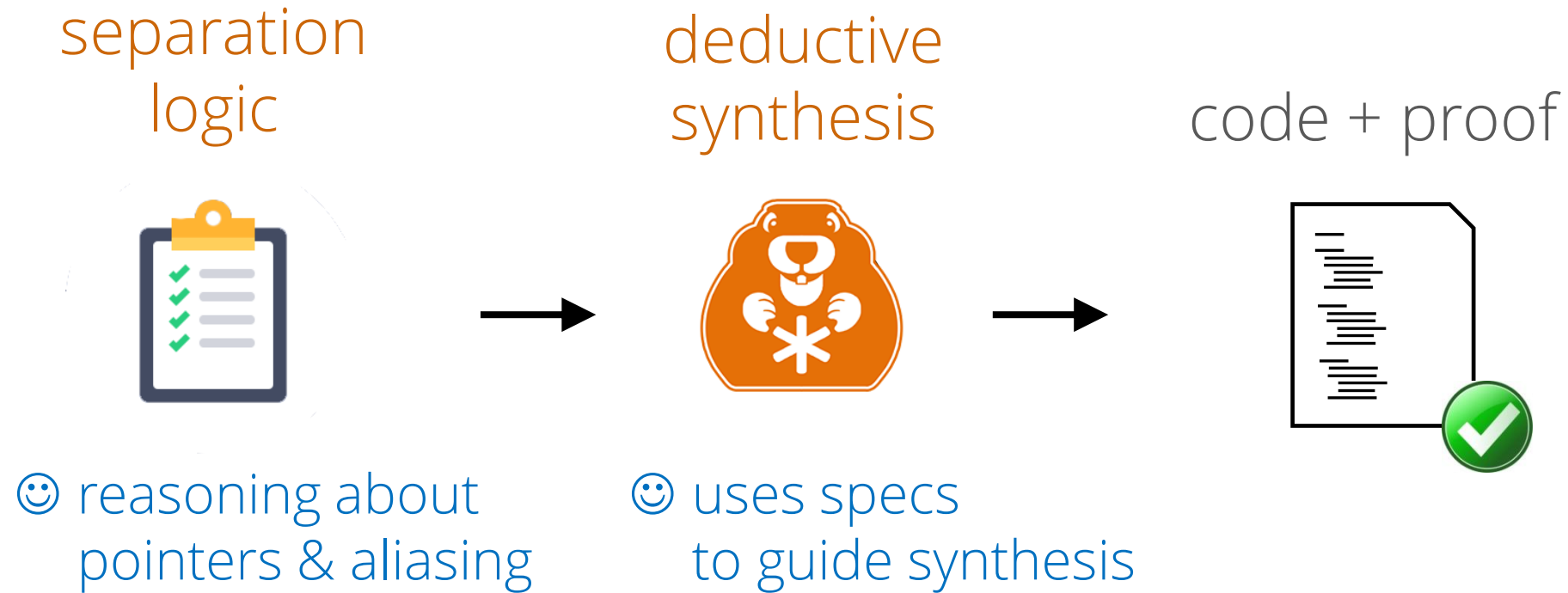
deductive
synthesis



code + proof



the SuSLik approach



this talk

1. example: swap a taste of SuSLik

Polikarpova, Sergey [POPL'19]

this talk

1. example: swap
a taste of SuSLik

Polikarpova, Sergey [POPL'19]

2. cyclic program synthesis
how SuSLik deals with recursion and discovers auxiliaries

Itzhaky et al. [PLDI'21]

this talk

1. example: swap
2. cyclic program synthesis

example 1: swap

swap values of two *distinct* pointers

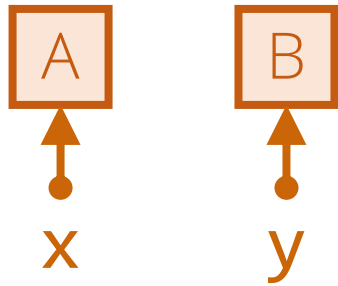
```
void swap(loc x, loc y)
```

example 1: swap

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

example 1: swap

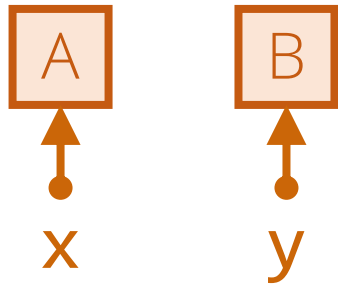
start state:



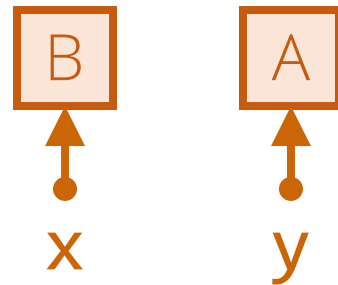
```
void swap(loc x, loc y)
```

example 1: swap

start state:



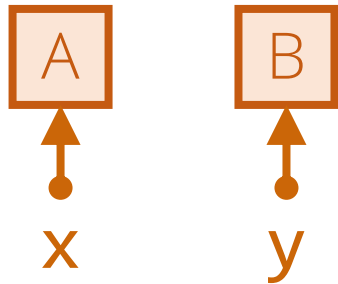
end state:



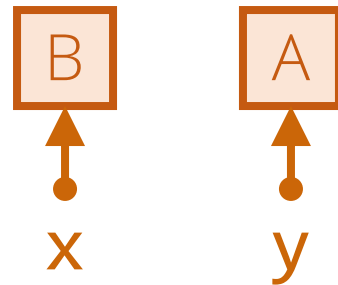
```
void swap(loc x, loc y)
```

example 1: swap

start state:



end state:



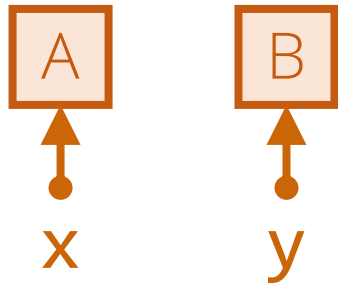
in separation logic:

$$\{ x \mapsto A * y \mapsto B \}$$

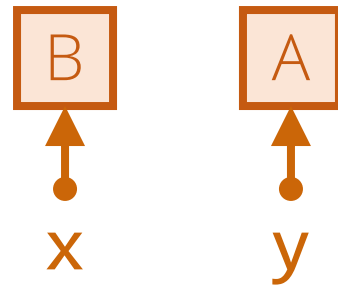
```
void swap(loc x, loc y)
```

example 1: swap

start state:



end state:



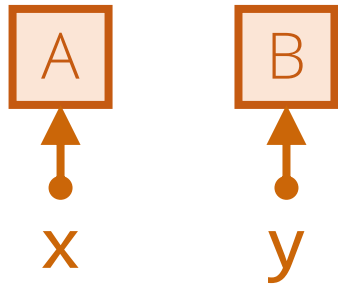
in separation logic:

$\{ x \mapsto A * y \mapsto B \}$ precondition

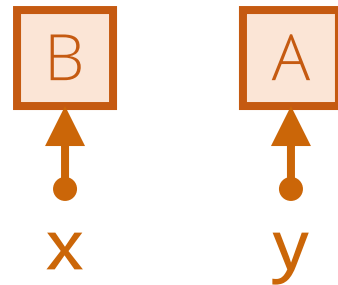
void swap(**loc** x , **loc** y)

example 1: swap

start state:



end state:



in separation logic:

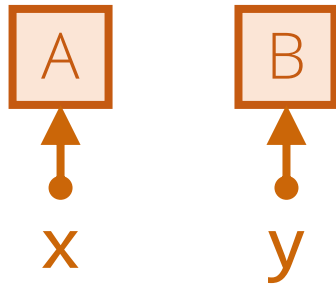
$\{ x \mapsto A * y \mapsto B \}$ precondition

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

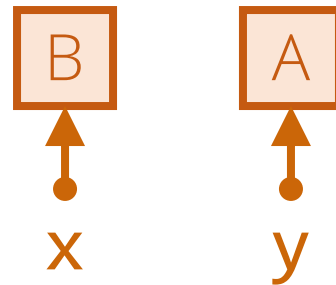
$\{ x \mapsto B * y \mapsto A \}$

example 1: swap

start state:



end state:



in separation logic:

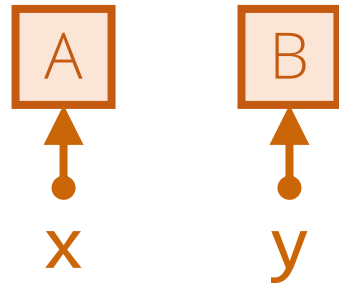
$\{ x \mapsto A * y \mapsto B \}$ precondition

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

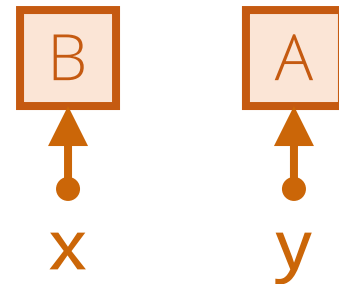
$\{ x \mapsto B * y \mapsto A \}$ postcondition

example 1: swap

start state:



end state:



in separation logic:

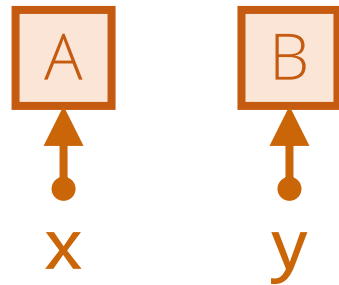
"points-to" $\longrightarrow$ $\{x \mapsto A * y \mapsto B\}$ precondition

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

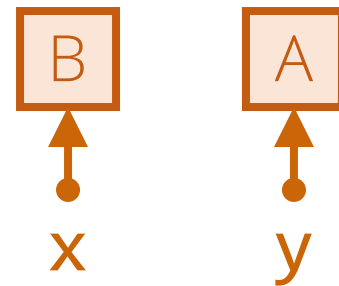
$\{x \mapsto B * y \mapsto A\}$ postcondition

example 1: swap

start state:



end state:



in separation logic:

"and
separately"

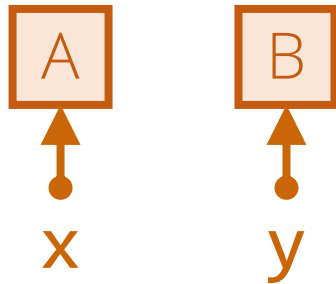
$\{ x \mapsto A \boxed{*} y \mapsto B \}$ precondition

void swap(loc x, loc y)

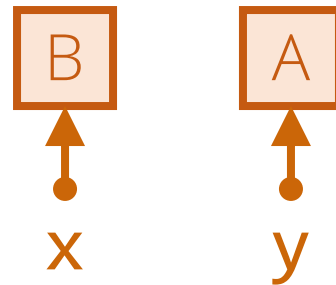
$\{ x \mapsto B \boxed{*} y \mapsto A \}$ postcondition

example 1: swap

start state:



end state:



in separation logic:

$\{ x \mapsto A * y \mapsto B \}$ precondition

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

$\{ x \mapsto B * y \mapsto A \}$ postcondition

ghost variables

demo 1: swap

swap values of two *distinct* pointers

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

deductive synthesis

synthesis as proof search

synthetic separation logic (SSL)

proof system for
transforming entailment

transforming entailment

$$P \rightsquigarrow Q \mid c$$

transforming entailment

$$P \rightsquigarrow Q \mid c$$

a state that satisfies P

transforming entailment

$$P \rightsquigarrow Q \mid c$$

a state that satisfies P
can be transformed into a state that satisfies Q

transforming entailment

$$P \rightsquigarrow Q \mid c$$

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

$\{\text{emp}\} \not\Rightarrow \{\text{emp}\} \mid ??$

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

↑
empty heap

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

(Emp)

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

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

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

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

(Frame)

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

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

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

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

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

(Write)

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

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

$$\{ x \mapsto A * P \} \rightsquigarrow \{ Q \} \quad | \quad ??$$

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

$$\{x \mapsto A * P\} \rightsquigarrow \{Q\} \mid \text{let } y = *x; c$$

(Read)

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

$$\{x \mapsto A * P\} \rightsquigarrow \{Q\} \mid \mathbf{let} \ y = *x; \ c$$

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 1: swap

$$\{x \mapsto A * y \mapsto B\} \rightsquigarrow \{x \mapsto B * y \mapsto A\} \mid ??$$

$$\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \quad ??$$

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

$$\{x \mapsto A * y \mapsto B\} \rightsquigarrow \{x \mapsto B * y \mapsto A\} \mid \quad \text{let } a1 = *x; ??$$

(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)

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

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

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

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

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

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

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

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

$$\frac{}{\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip}} \text{ (Emp)}$$

$$\frac{}{\{ y \mapsto a1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid ??} \text{ (Frame)}$$

$$\frac{}{\{ y \mapsto b1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid *y = a1; ??} \text{ (Write)}$$

$$\frac{}{\{ x \mapsto b1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid ??} \text{ (Frame)}$$

$$\frac{}{\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid *x = b1; ??} \text{ (Write)}$$

$$\frac{}{\{ x \mapsto a1 * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto a1 \} \mid \mathbf{let} \ b1 = *y; ??} \text{ (Read)}$$

$$\frac{}{\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \mathbf{let} \ a1 = *x; ??} \text{ (Read)}$$

$$\begin{array}{c}
\frac{}{\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip}} \text{ (Emp)} \\
\frac{}{\{ y \mapsto a1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid ??} \text{ (Frame)} \\
\frac{}{\{ y \mapsto b1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid *y = a1; ??} \text{ (Write)} \\
\frac{}{\{ x \mapsto b1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid ??} \text{ (Frame)} \\
\frac{}{\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid *x = b1; ??} \text{ (Write)} \\
\frac{}{\{ x \mapsto a1 * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto a1 \} \mid \mathbf{let\ } b1 = *y; ??} \text{ (Read)} \\
\frac{}{\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \boxed{\mathbf{let\ } a1 = *x; ??}} \text{ (Read)}
\end{array}$$

$$\begin{array}{c}
\frac{}{\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip}} \text{ (Emp)} \\
\frac{}{\{ y \mapsto a1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid ??} \text{ (Frame)} \\
\frac{}{\{ y \mapsto b1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid *y = a1; ??} \text{ (Write)} \\
\frac{}{\{ x \mapsto b1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid ??} \text{ (Frame)} \\
\frac{}{\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid *x = b1; ??} \text{ (Write)} \\
\frac{}{\{ x \mapsto a1 * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto a1 \} \mid \boxed{\mathbf{let\ } b1 = *y;} ??} \text{ (Read)} \\
\frac{}{\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \boxed{\mathbf{let\ } a1 = *x;} ??} \text{ (Read)}
\end{array}$$

$\frac{}{\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \text{skip}}$	(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)
$\frac{}{\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid \boxed{*x = b1;} ??}$	(Write)
$\frac{}{\{ x \mapsto a1 * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto a1 \} \mid \boxed{\text{let } b1 = *y;} ??}$	(Read)
$\frac{}{\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \boxed{\text{let } a1 = *x;} ??}$	(Read)

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

$$\begin{array}{c}
\frac{}{\{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \boxed{\text{skip}}} \text{(Emp)} \\
\frac{}{\{ y \mapsto a1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid ??} \text{(Frame)} \\
\frac{}{\{ y \mapsto b1 \} \rightsquigarrow \{ y \mapsto a1 \} \mid \boxed{*y = a1; ??}} \text{(Write)} \\
\frac{}{\{ x \mapsto b1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid ??} \text{(Frame)} \\
\frac{}{\{ x \mapsto a1 * y \mapsto b1 \} \rightsquigarrow \{ x \mapsto b1 * y \mapsto a1 \} \mid \boxed{*x = b1; ??}} \text{(Write)} \\
\frac{}{\{ x \mapsto a1 * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto a1 \} \mid \boxed{\text{let } b1 = *y; ??}} \text{(Read)} \\
\frac{}{\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \boxed{\text{let } a1 = *x; ??}} \text{(Read)}
\end{array}$$

$\{ x \mapsto A * y \mapsto B \}$

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

$\{ x \mapsto B * y \mapsto A \}$

this talk

1. example: swap

2. cyclic program synthesis

this talk

1. example: swap

2. cyclic program synthesis

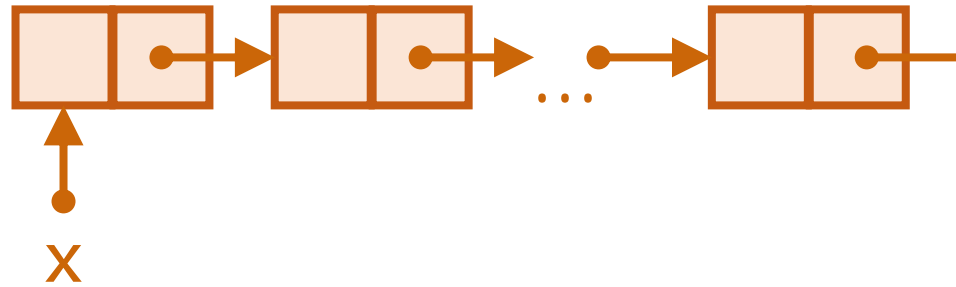
2.1. inductive predicates

2.2. recursion via cyclic proofs

2.3. discovering recursive auxiliaries

linked structures

how do we handle unbounded data structures?



the linked list predicate

```
predicate list (loc x) {
```

```
}
```

the linked list predicate

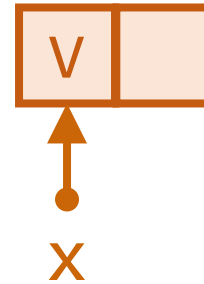
```
predicate list (loc x) {  
  | x = 0 => { emp }
```



```
}
```

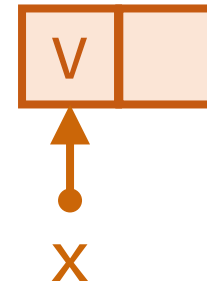
the linked list predicate

```
predicate list (loc x) {  
  | x = 0 => { emp }  
  | x ≠ 0 => { [x, 2]  
  }  
}
```



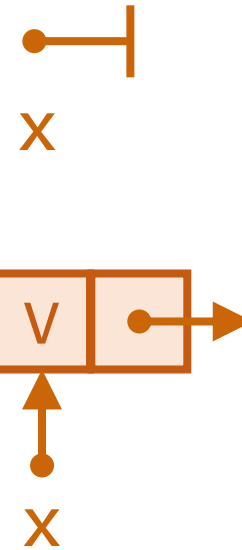
the linked list predicate

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



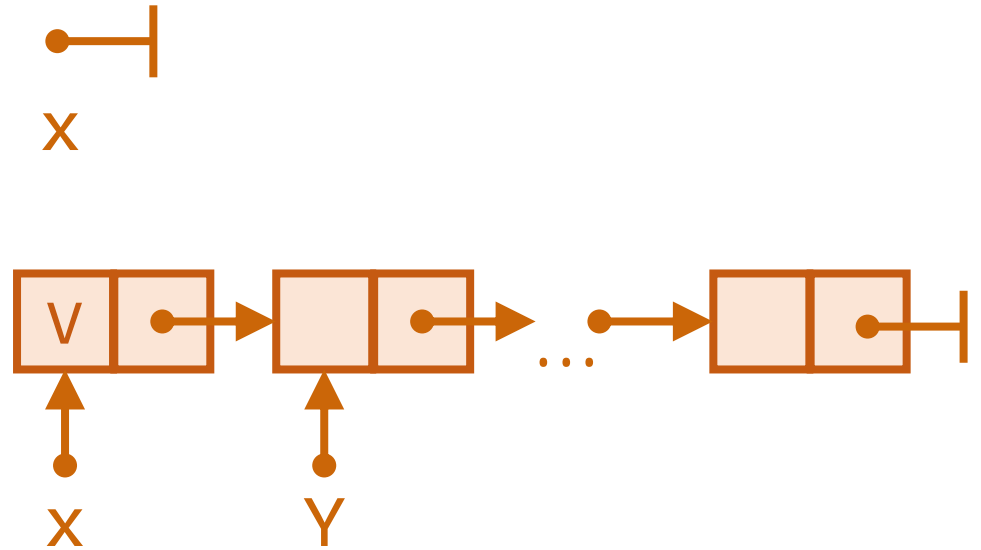
the linked list predicate

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



the linked list predicate

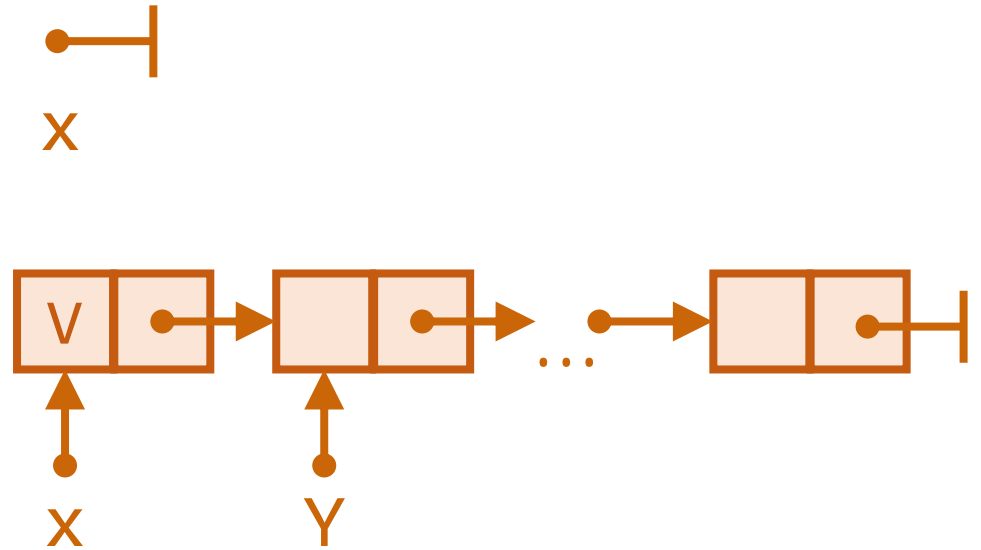
```
predicate list (loc x) {  
  |  $x = 0 \Rightarrow \{ \text{emp} \}$   
  |  $x \neq 0 \Rightarrow \{ [x, 2]$   
    *  $x \mapsto V$   
    *  $(x + 1) \mapsto Y$   
    * list(Y)  
  }  
}
```



the linked list predicate

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

malloc-
allocated
block



demo 2: dispose a list

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

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

```
{ emp }
```

dispose a list

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

`{ list(x) }  $\rightsquigarrow$  { emp }`

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

$\{ \text{list}(x) \} \rightsquigarrow \{ \text{emp} \}$

(Open)

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

```
predicate list (loc x) {  
  | x = 0 =>  
  | x ≠ 0 =>  
}
```

$\{ \text{list}(x) \} \rightsquigarrow \{ \text{emp} \}$

(Open)

dispose a list

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

predicate `list (loc x) {`
 `| x = 0 =>`
 `| x ≠ 0 =>`
`}`

`{ list(x) }  $\rightsquigarrow$  { emp } | if x == 0 {...} else {...}`

(Open)

dispose a list

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

```
predicate list (loc x) {  
  | x = 0 => { emp }  
  | x ≠ 0 =>  
}
```

`{ emp }  $\rightsquigarrow$  { emp }`

`{ list(x) }  $\rightsquigarrow$  { emp } | if x == 0 {...} else {...}`

(Open)

dispose a list

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

```
predicate list (loc x) {  
  | x = 0 => { emp }  
  | x ≠ 0 =>  
}
```

(Emp)

`{ emp }  $\rightsquigarrow$  { emp }`

(Open)

`{ list(x) }  $\rightsquigarrow$  { emp } | if x == 0 {...} else {...}`

dispose a list

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

```
predicate list (loc x) {  
  | x = 0 => { emp }  
  | x ≠ 0 =>  
}
```

(Emp)

`{ emp }  $\rightsquigarrow$  { emp } | skip`

(Open)

`{ list(x) }  $\rightsquigarrow$  { emp } | if x == 0 { ... } else { ... }`

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

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

(Emp)

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

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

(Open)

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

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

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

(Emp)

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

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

(Open)

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

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

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

(Emp)

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

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

(Open)

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

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

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

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

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

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

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

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

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

how do we generate a recursive call?

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

this talk

1. example: swap

2. cyclic program synthesis

2.1. inductive predicates

2.2. recursion via cyclic proofs

2.3. discovering recursive auxiliaries

cyclic proofs

proof “trees” with
backlinks

Brotherston, Bornat, Calcagno: *Cyclic proofs of program termination in separation logic*. [POPL'08]

Rowe, Brotherston: *Automatic cyclic termination proofs for recursive procedures in separation logic*. [CPP'17]

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

$$\begin{array}{c}
 \text{(Emp)} \\
 \hline
 \{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip} \\
 \hline
 \end{array}
 \quad
 \begin{array}{c}
 \{ \text{list}(y) \} \rightsquigarrow \{ \text{emp} \} \\
 \hline
 \{ [x, 2] * x \mapsto v * (x + 1) \mapsto y * \text{list}(y) \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{free}(x) \\
 \hline
 \text{(Free)} \\
 \text{(Open)}
 \end{array}$$

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

dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

$$\begin{array}{c}
 \text{(Emp)} \\
 \hline
 \{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip} \\
 \hline
 \end{array}
 \quad
 \begin{array}{c}
 \boxed{\{ \text{list}(y) \} \rightsquigarrow \{ \text{emp} \}} \\
 \hline
 \{ [x, 2] * x \mapsto v * (x + 1) \mapsto y * \text{list}(y) \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{free}(x) \\
 \hline
 \text{(Free)} \\
 \text{(Open)}
 \end{array}$$

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

dispose a list

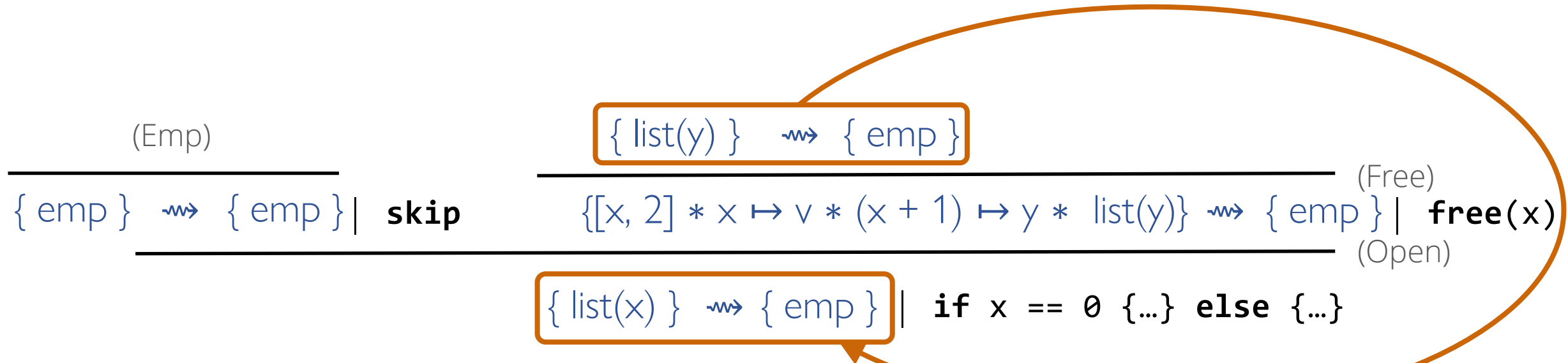
$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

$$\begin{array}{c}
 \text{(Emp)} \\
 \hline
 \{ \text{emp} \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{skip} \\
 \hline
 \end{array}
 \quad
 \begin{array}{c}
 \boxed{\{ \text{list}(y) \} \rightsquigarrow \{ \text{emp} \}} \\
 \hline
 \{ [x, 2] * x \mapsto v * (x + 1) \mapsto y * \text{list}(y) \} \rightsquigarrow \{ \text{emp} \} \mid \mathbf{free}(x) \\
 \hline
 \text{(Free)} \\
 \text{(Open)}
 \end{array}$$

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

dispose a list

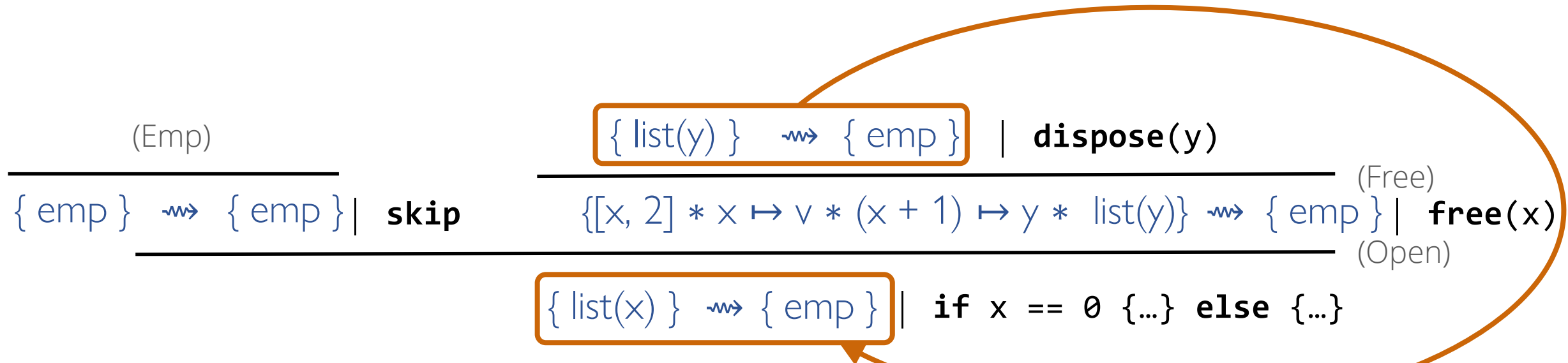
$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$



dispose a list

$\{ \text{list}(x) \} \text{dispose}(x) \{ \text{emp} \}$

backlink generates a recursive call



this talk

1. example: swap

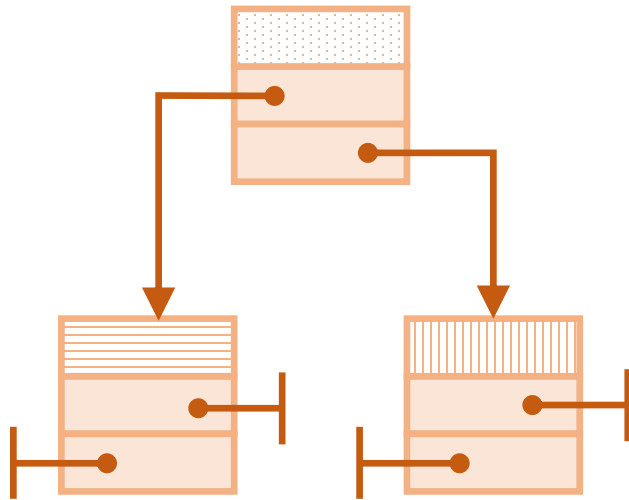
2. cyclic program synthesis

2.1. inductive predicates

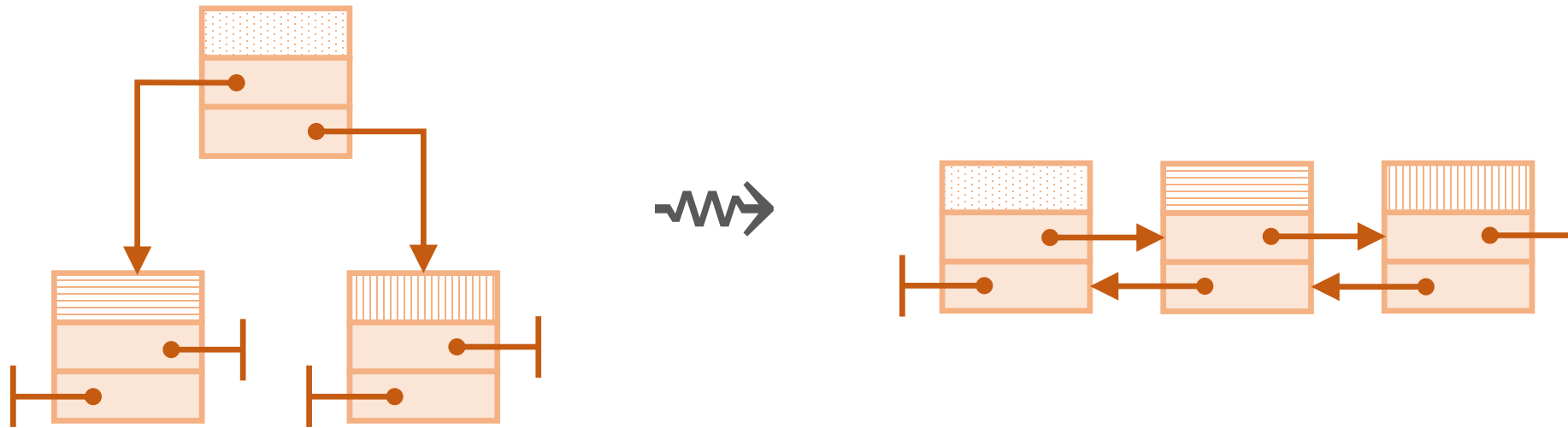
2.2. recursion via cyclic proofs

2.3. discovering recursive auxiliaries

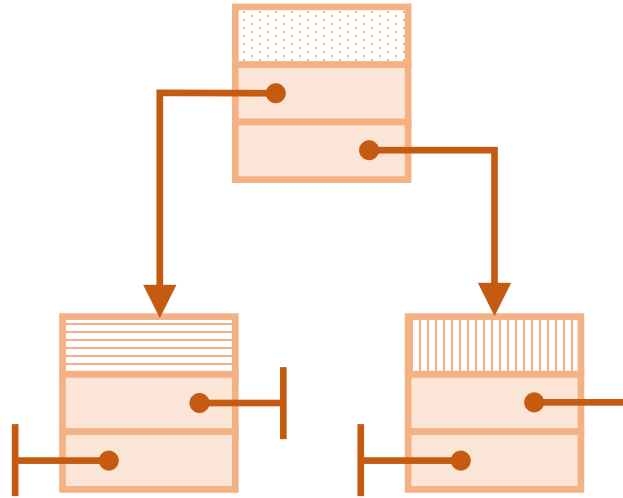
task: flatten a tree into a list



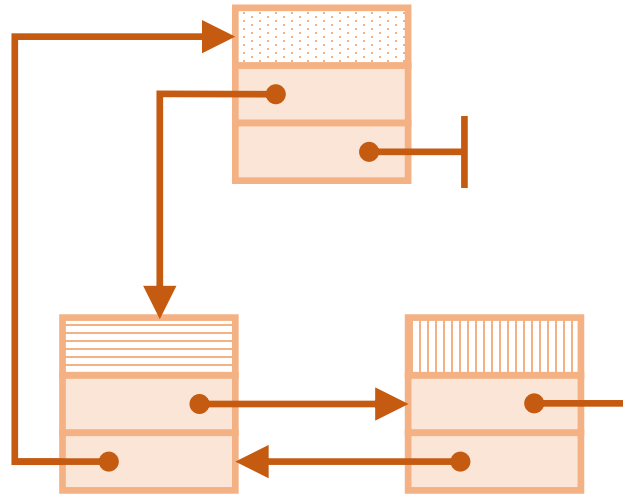
task: flatten a tree into a list



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



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



example 3: tree flattening

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

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

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

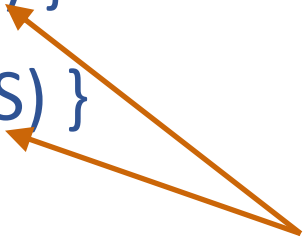
example 3: tree flattening

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

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

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

set of elements



example 3: tree flattening

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

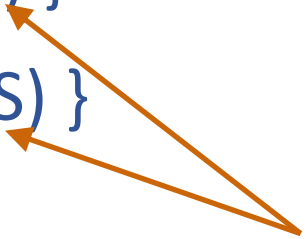
```
{ tree(x, S) }
```

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

back pointer
(irrelevant)



set of elements



this task is challenging!

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

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

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



Leon

[Kneuss et al'13]

Synquid

[Polikarpova et al'16]

ImpSynth

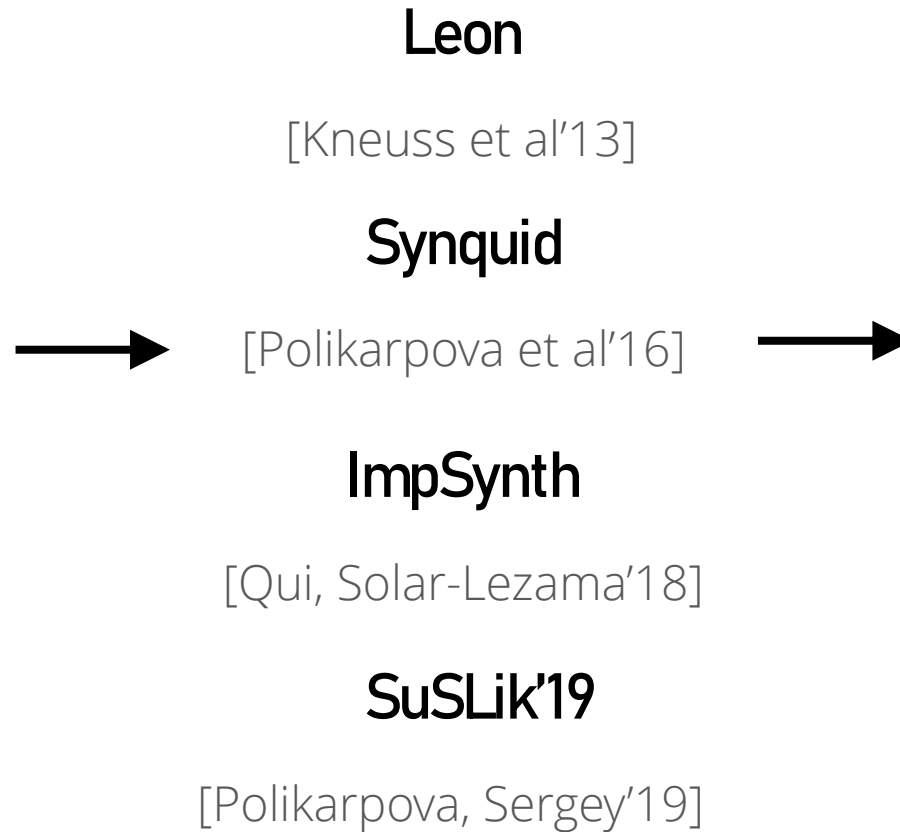
[Qui, Solar-Lezama'18]

SuSLik'19

[Polikarpova, Sergey'19]

this task is challenging!

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



this task is challenging!

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



```
}  
}
```

three flattening

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



```
}  
}
```

three flattening

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



```
}  
}
```

three flattening

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



```
}  
}
```

three flattening

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

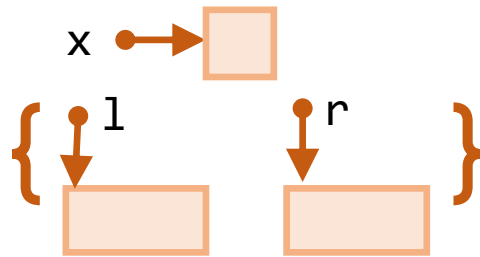


needs recursive function
to append two lists!

```
}  
}
```

tree flattening with cyclic proofs

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



$\rightsquigarrow$

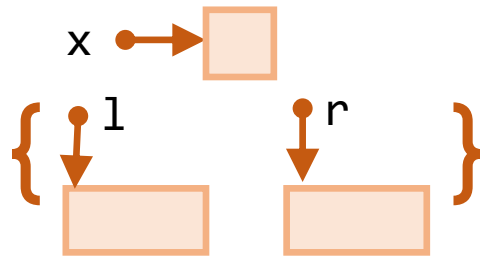


}

}

tree flattening with cyclic proofs

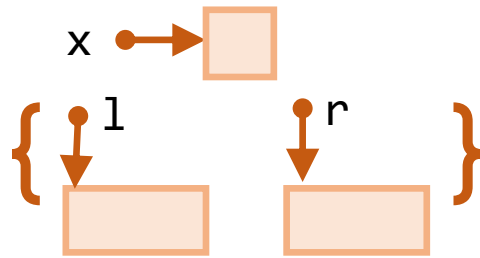
```
flatten(x) {  
  if (x != 0) {  
    ...  
  }  
}
```



```
}  
}
```

tree flattening with cyclic proofs

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

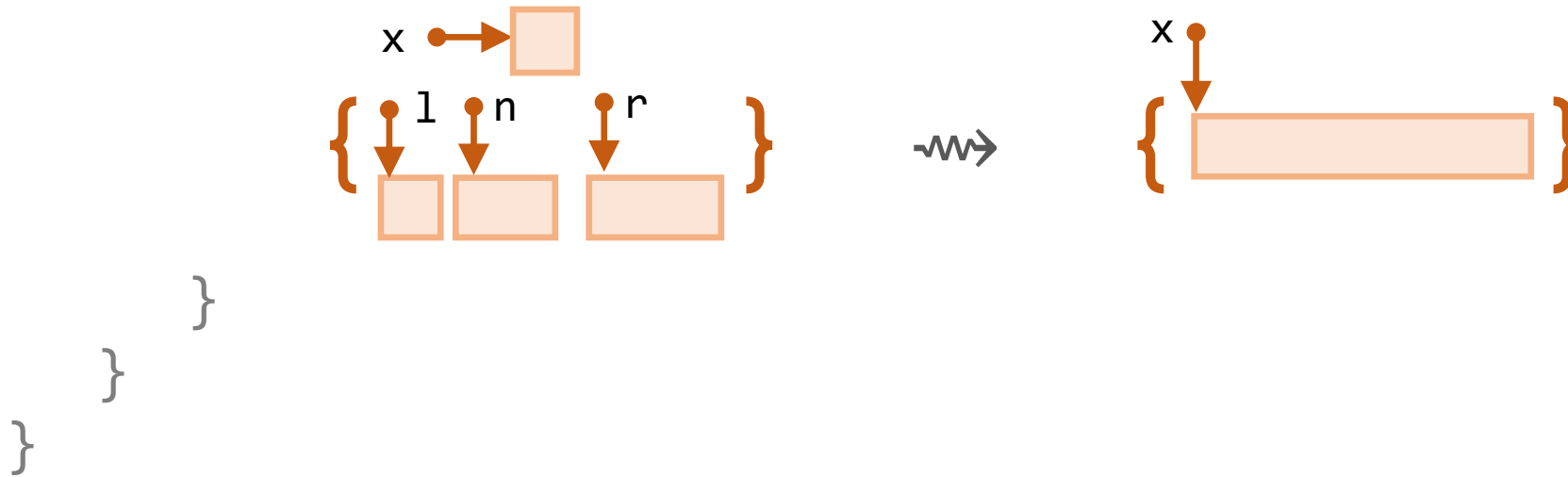


}

}

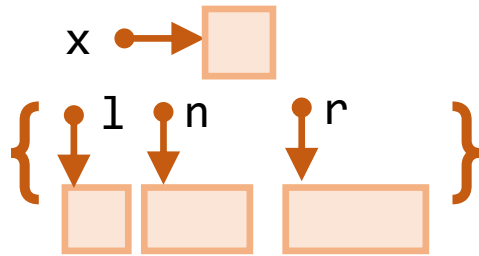
tree flattening with cyclic proofs

```
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);
```



$\rightsquigarrow$



```
}  
}  
}
```

does this goal look familiar?

```
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);
```



```
}  
}  
}
```

does this goal look familiar?

```
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);
```



```
}  
}  
}
```

does this goal look familiar?

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



```
}  
}  
}
```

let's cycle back!

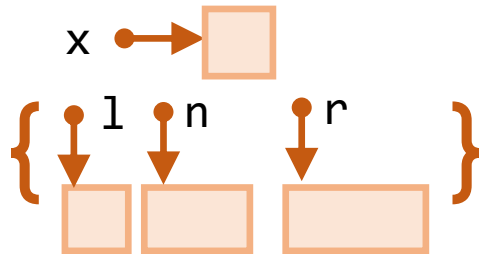
```
flatten(x) {  
  if (x != 0) {  
    ...  
    if (l == 0) { ... } else {  
      n = *(l + 1);  
    }  
  }  
}
```



```
}  
}  
}
```

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);  
    }  
  }  
}
```



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);  
  ...  
}
```

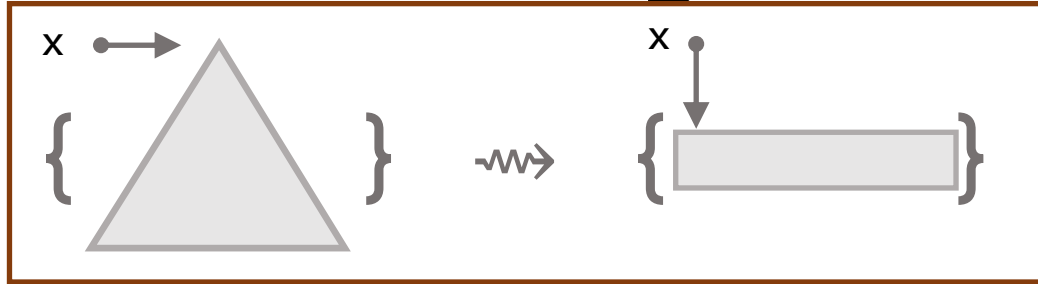
extracting the auxiliary

```
flatten(x) {  
  if (x != 0) {  
    ...  
    helper(1, r, x);  
  }  
}
```



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

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);  
    ...  
  }  
}
```

demo 3: tree flattening

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

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

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

what else can SuSLik do?

nested traversals

e.g. list sorting, deduplication

what else can SuSLik do?

nested traversals

e.g. list sorting, deduplication

non-trivial termination metrics

e.g. sorted list merge

what else can SuSLik do?

nested traversals

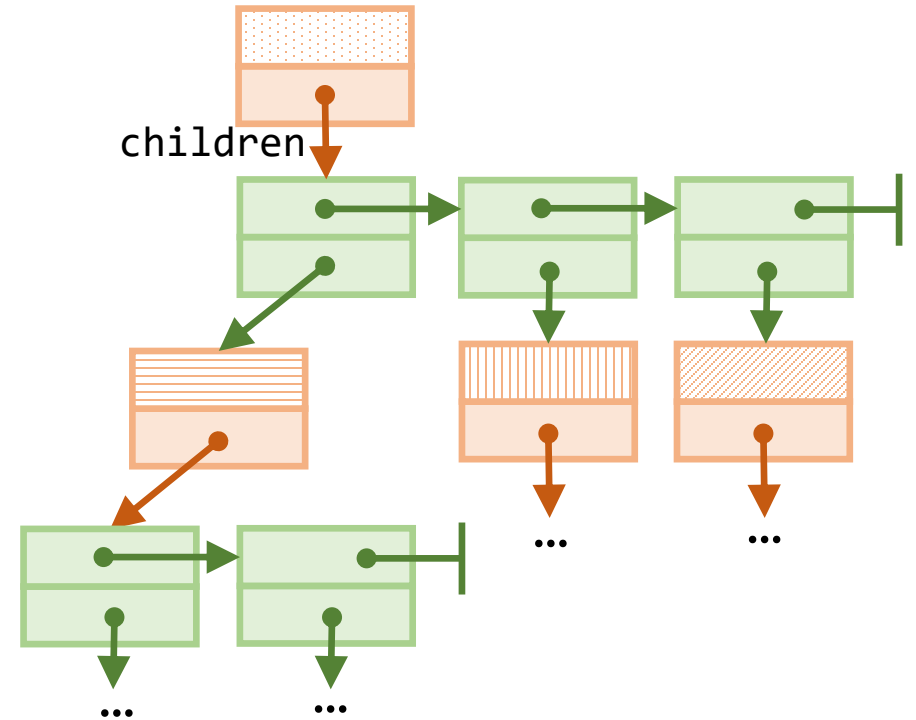
e.g. list sorting, deduplication

non-trivial termination metrics

e.g. sorted list merge

mutual recursion

e.g. rose tree flattening



what else can SuSLik do?

54 benchmarks

Itzhaky et al. [CAV'21] (invited paper)

what else can SuSLik do?

54 benchmarks

Itzhaky et al. [CAV'21] (invited paper)

(sorted, nested) lists, trees, BSTs, rose trees, packed trees

what else can SuSLik do?

54 benchmarks

Itzhaky et al. [CAV'21] (invited paper)

(sorted, nested) lists, trees, BSTs, rose trees, packed trees

synthesized programs with up to

32 statements (BST to sorted list)

12x code/spec ratio (rose tree deallocate)

what else can SuSLik do?

54 benchmarks

Itzhaky et al. [CAV'21] (invited paper)

(sorted, nested) lists, trees, BSTs, rose trees, packed trees

synthesized programs with up to

32 statements (BST to sorted list)

12x code/spec ratio (rose tree deallocate)

synthesis times < 2 minutes

all but five < 30 sec

future work

more classes of auxiliaries

Itzhaky et al. [CAV'21] (invited paper)

e.g. linear-time list reversal requires an accumulator parameter
this will enable loops!

future work

more classes of auxiliaries

Itzhaky et al. [CAV'21] (invited paper)

e.g. linear-time list reversal requires an accumulator parameter
this will enable loops!

program efficiency / readability

e.g. synthesizes quadratic-time list reversal, tree flattening
code distribution between procedures is sub-optimal

future work

more classes of auxiliaries

Itzhaky et al. [CAV'21] (invited paper)

e.g. linear-time list reversal requires an accumulator parameter
this will enable loops!

program efficiency / readability

e.g. synthesizes quadratic-time list reversal, tree flattening
code distribution between procedures is sub-optimal

better search

current search relies on brittle cost heuristic
can we learn an adaptive cost function?

deductive synthesis with SuSLik

