

# synthesis of safe pointer-manipulating programs

Nadia Polikarpova

Workshop on Dependable and Secure Software Systems  
Oct 26, 2021  
ETH Zurich

# 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;
}
```



# 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

1949

today

# ... but this is still hard

Checking a Large  
Routine  
  
Alan Turing



have to write code, specs, and proofs!



Security. Performance. Proof.



CERTIKOS



1949



today

# what's next for verified systems?

Checking a Large  
Routine  
  
Alan Turing



1949



today

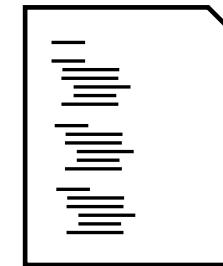


# program synthesis

specification

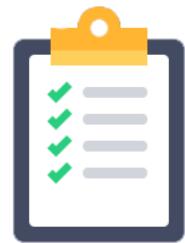


code



# 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



# 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]

1980

today

# deductive synthesis

**A Deductive Approach  
to Program Synthesis**

[Manna, Waldinger]

**Leon**

[Kneuss et al'13] [Polikarpova, Sergey'19] [Polikarpova et al'16]

**VS3**

[Srivastava et al'10]

**Fiat**

[Delaware et al'15]

**SuSLik**

**Synquid**

**ImpSynth**

[Qui, Solar-Lezama'18]

**Jennisys**

[Leino, Milicevic'12]



1980



today

# deductive synthesis of pointer-manipulating programs

A Deductive Approach  
to Program Synthesis

[Manna, Waldinger]

1980

Leon

[Polikarpova, Sergey. POPL'19]

VS3

Fiat

SuSLik



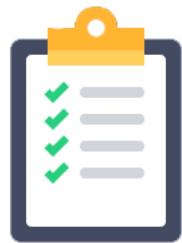
ImpSynth

Jennisys

today

# SuSLik

specification



C code  
+ proof



# 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**og**i**k

[Reynolds. LICS'02]

specification



C code  
+ proof

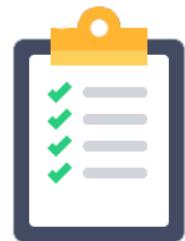


# SuSLik

= **S**ynthesis **u**sing **S**eparation **L**og**i**k

[Reynolds. LICS'02]

separation  
logic



C code  
+ proof

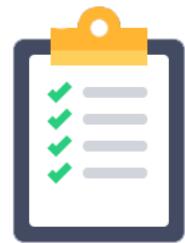


# SuSLik

= **S**ynthesis **u**sing **S**eparation **L**og**i**k

[Reynolds. LICS'02]

separation  
logic



C code  
+ proof



⌚ reasoning about  
pointers & aliasing

# SuSLik

= **S**ynthesis **u**sing **S**eparation **L**og**i**k

[Reynolds. LICS'02]

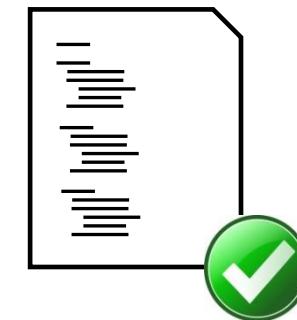
separation  
logic



deductive  
synthesis



C code  
+ proof



⌚ reasoning about  
pointers & aliasing

# SuSLik

= **Synthesis using Separation Logik**

[Reynolds. LICS'02]

separation  
logic



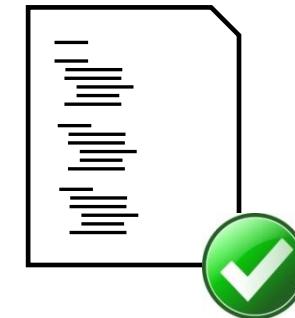
☺ reasoning about  
pointers & aliasing

deductive  
synthesis



☺ uses specs  
to guide synthesis

C code  
+ proof



# this talk

## 1. a taste of SuSLik

# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs

# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs
3. learning auxiliary functions

# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs
3. learning auxiliary functions

# example 1: swap

swap values of two *distinct* pointers

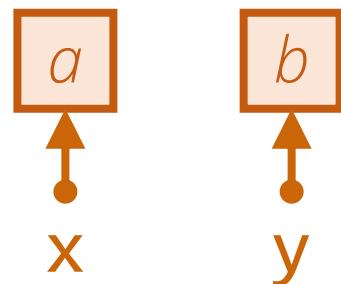
```
void swap(loc x, loc y)
```

# example 1: swap

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

# example 1: swap

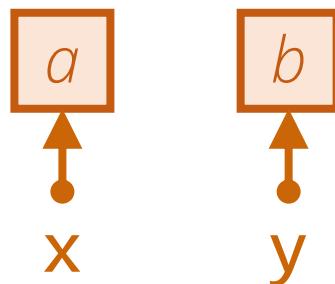
start state:



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

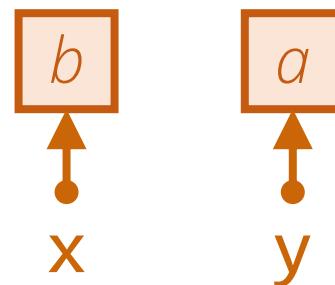
# example 1: swap

start state:



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

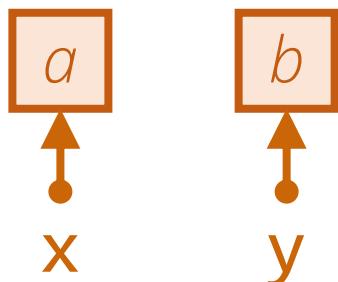
end state:



# example 1: swap

in separation logic:

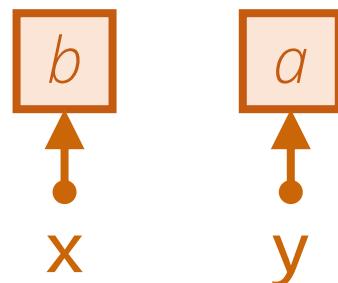
start state:



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

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

end state:

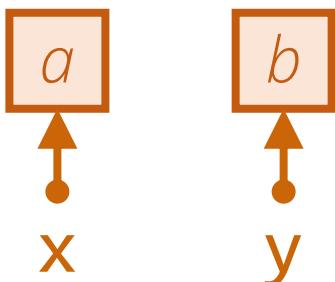


$\{ x \mapsto b * y \mapsto a \}$  postcondition

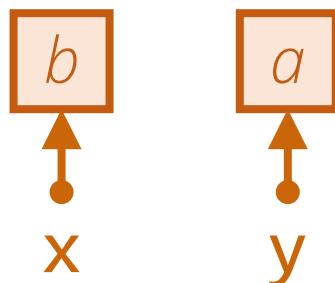
# example 1: swap

in separation logic:

start state:



end state:



heaplets  
↓  
 $\{ x \mapsto a * y \mapsto b \}$  precondition

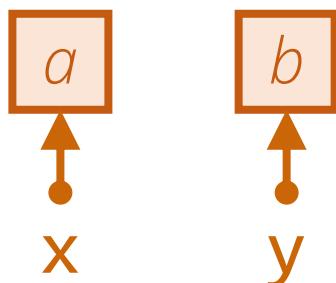
**void swap(loc x, loc y)**

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

# example 1: swap

in separation logic:

start state:



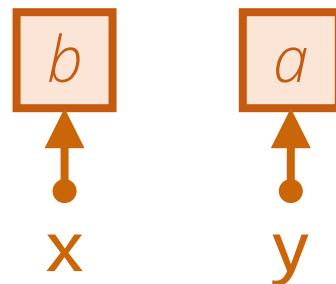
“points-to”

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

precondition

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

end state:



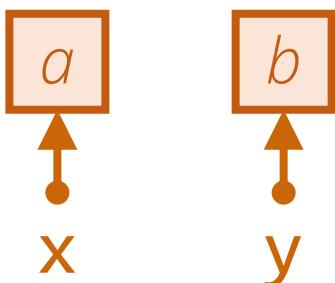
$\{x \mapsto b * y \mapsto a\}$

postcondition

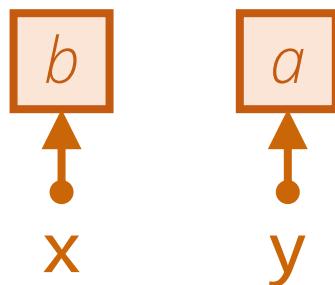
# example 1: swap

in separation logic:

start state:



end state:



"and  
separately"

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

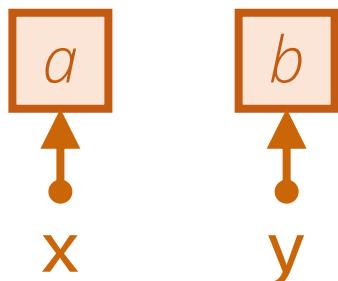
**void swap(loc x, loc y)**

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

# example 1: swap

in separation logic:

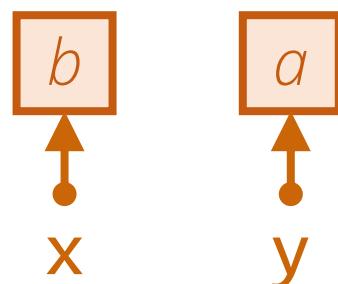
start state:



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

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

end state:



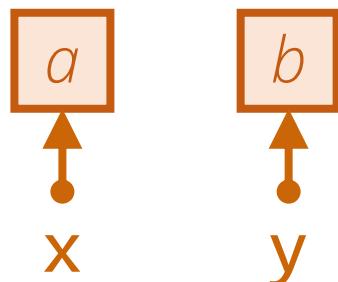
$\{ x \mapsto b * y \mapsto a \}$  postcondition

program variables

# example 1: swap

in separation logic:

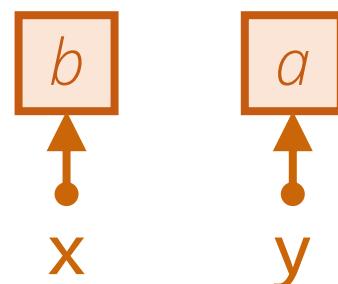
start state:



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

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

end state:



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

program variables      ghost variables

# demo 1: swap

swap values of two *distinct* pointers

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

# demo 1: swap

swap values of two *distinct* pointers

how did this happen?

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

# deductive synthesis

idea: to find the program, look for the proof



# synthetic separation logic (SSL)

a proof system for program derivation relation

$$P \xrightarrow{\sim} Q \mid C$$

# synthetic separation logic (SSL)

a proof system for program derivation relation

$$P \xrightarrow{\sim} Q \mid C$$

↑  
precondition

# synthetic separation logic (SSL)

a proof system for program derivation relation

$$P \xrightarrow{\sim} Q \mid C$$

↑                      ↑  
precondition          postcondition

# synthetic separation logic (SSL)

a proof system for program derivation relation

$$P \xrightarrow{\sim} Q \mid C$$

↑                    ↑                    ↑  
precondition      postcondition      program

(Emp)

{emp} ↣ {emp} | ??

(Emp)

{emp} ↵ {emp} | ??



(Emp)

{emp} ↗ {emp} | **skip**

(Frame)

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

(Frame)

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

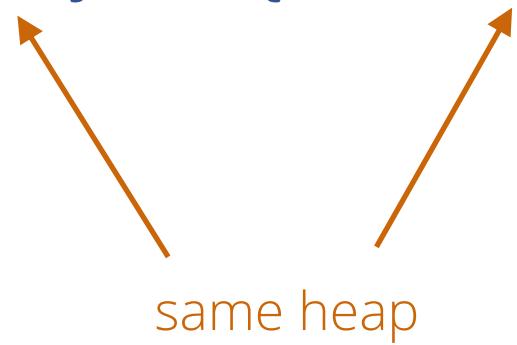
same heap

(Frame)

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

---

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



(Write)

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

(Write)

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



points to non-ghost expression

(Write)

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

---

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



points to non-ghost expression

(Read)

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

(Read)

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

| ??



points to ghost variable

(Read)

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

---

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

 points to ghost variable

# SSL: basic rules

(Emp)

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

(Read)

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

---

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

(Frame)

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

(Write)

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

---

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

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

# example 1: swap

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

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

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

---

(Read)

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

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

---

(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)

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

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

---

(Write)

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

---

(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)

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

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

---

(Frame)

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

---

(Write)

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

---

(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)

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

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

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

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

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

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

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

$$\frac{\{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \text{let } a1 = *x; ??}{\quad} \quad (\text{Read})$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs
3. learning auxiliary functions

# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs
  - inductive predicates
  - recursion via cyclic proofs
3. learning auxiliary functions

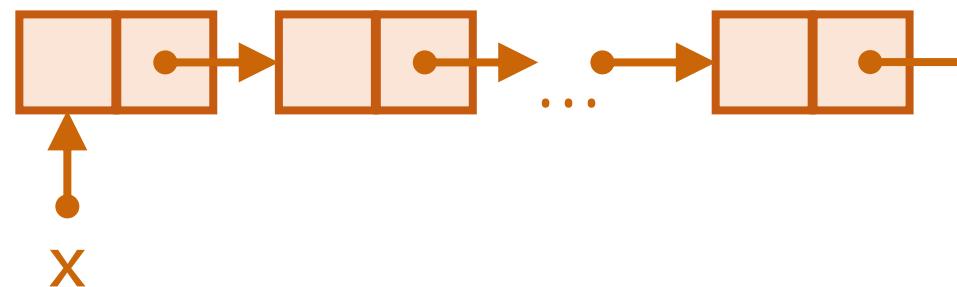
# **example 2: dispose**

deallocate a linked list with head pointer x

**void dispose(loc x)**

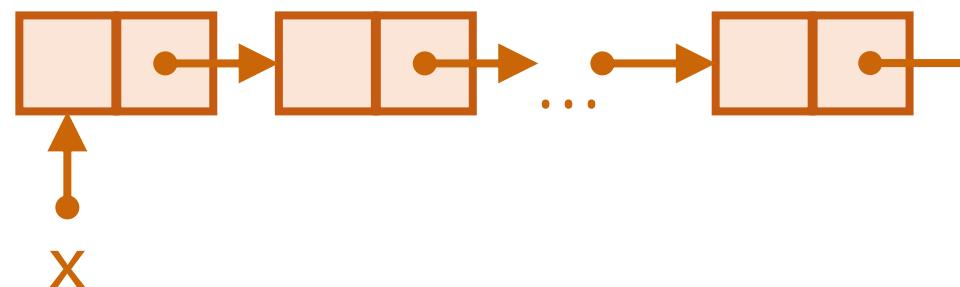
# dynamic data structures

how do we specify a linked list?



# dynamic data structures

how do we specify a linked list?



inductive predicates to the rescue!

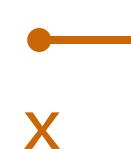
# the linked list predicate

**predicate** list (loc x) {

}

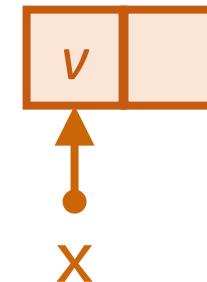
# the linked list predicate

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



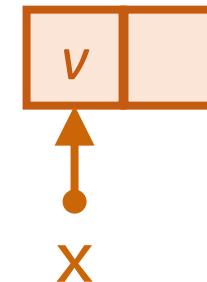
# the linked list predicate

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



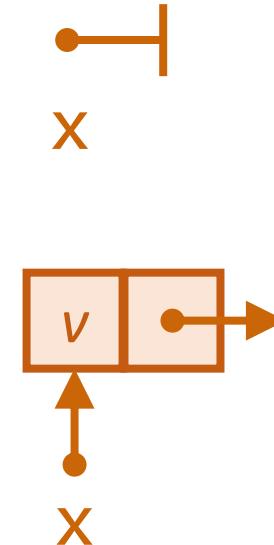
# the linked list predicate

```
predicate list (loc x) {  
    | x = 0 ⇒ { emp }  
    | x ≠ 0 ⇒ { [x, 2]  
                  * x ↪ v  
                }  
}
```



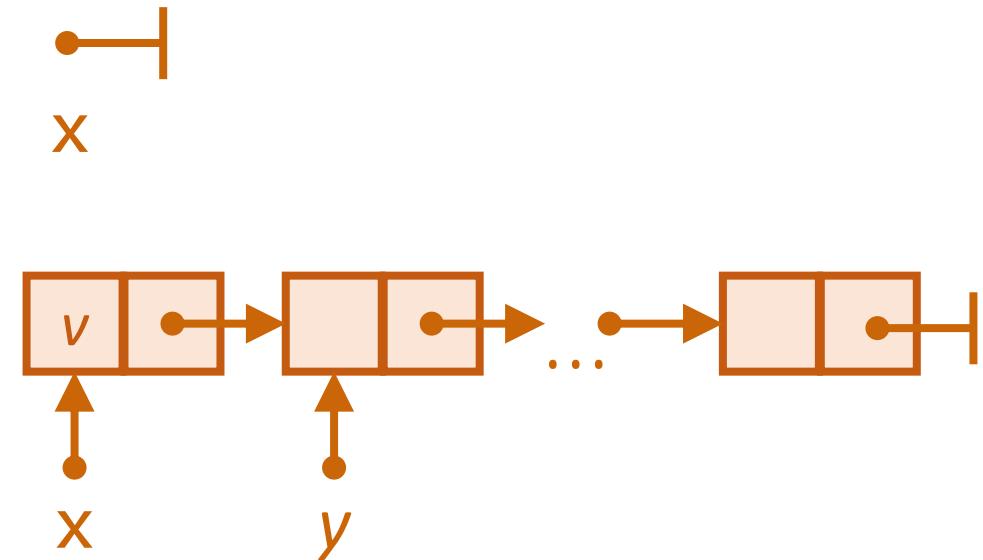
# the linked list predicate

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



# the linked list predicate

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



# demo 2: dispose a list

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

# demo 2: dispose a list

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

# demo 2: dispose a list

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

# demo 2: dispose a list

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

what if we care about content?

# **example 3: 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) {

}

# linked list with elements

```
predicate list (loc x, set S) {  
    | x = 0 ⇒ { S = ∅ ; emp }  
}  
}
```



# linked list with elements

```
predicate list (loc x, set S) {  
    | x = 0 ⇒ { S = ∅ ; emp }  
    |  
        ↑          ↑  
    pure part    spatial part  
}  
}
```



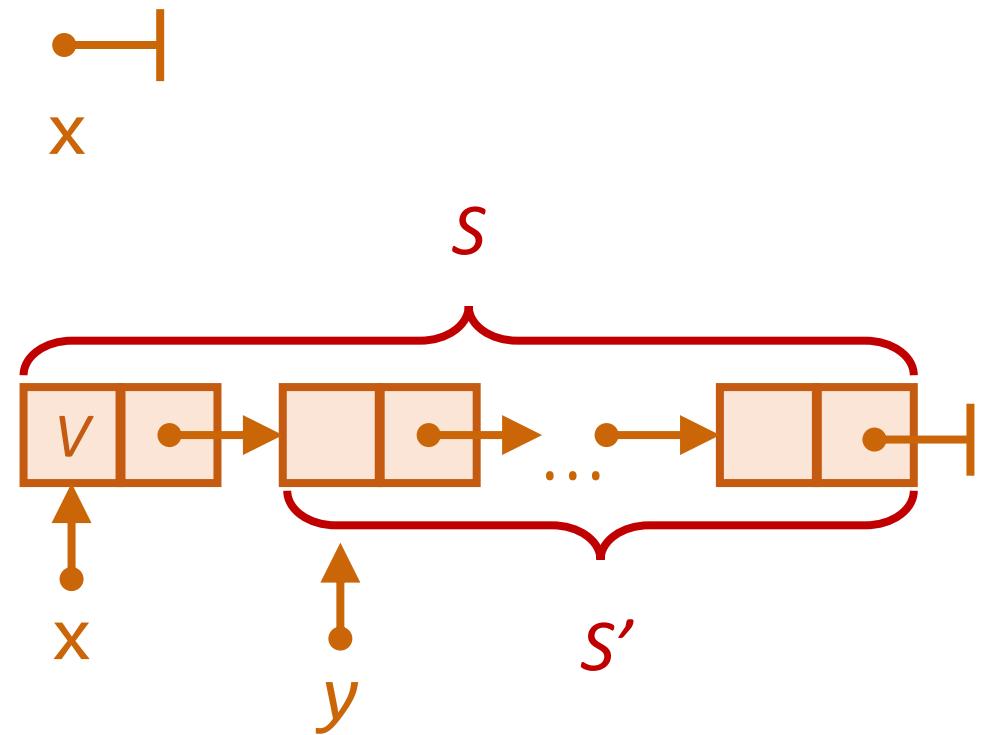
# linked list with elements

```
predicate list (loc x, set S) {  
    | x = 0 ⇒ { S = ∅ ; 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 ret)
```

# demo 3: copy a list

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

# demo 3: copy a list

```
return location  
void copy(loc x, loc ret)  
{ list(x, S) * ret ↪ _ }
```

# demo 3: copy a list

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

# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs
  - inductive predicates
  - recursion via cyclic proofs
3. learning auxiliary functions

# dispose a list

```
dispose(loc x)
```

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

```
{ emp }
```

# dispose a list

```
dispose(loc x)
```

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

```
{ emp }
```

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

# dispose a list

```
dispose(loc x)
```

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

```
{ emp }
```

---

(Open)

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

# 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) }
}
```

---

(Open)

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

# 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) }
}
```

---

(Open)

```
{ list(x) } ↨ { emp } | if (x == 0) {...} else {...}
```

# 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 }  $\rightsquigarrow$  { emp }

---

(Open)

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

# 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)

---

```
{ emp } ↣ { emp } | skip
```

(Open)

---

```
{ list(x) } ↣ { emp } | if (x == 0) {...} else {...}
```

# 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)

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

# 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)

---

```
{ emp } ↣ { emp } | skip
```

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

(Open)

---

```
{ list(x) } ↣ { emp } | if (x == 0) {...} else {...}
```

# 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) }
}
```

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

(Free)  
(Open)

# 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) }
}
```

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

# 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) }
}
```

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

# dispose a list

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

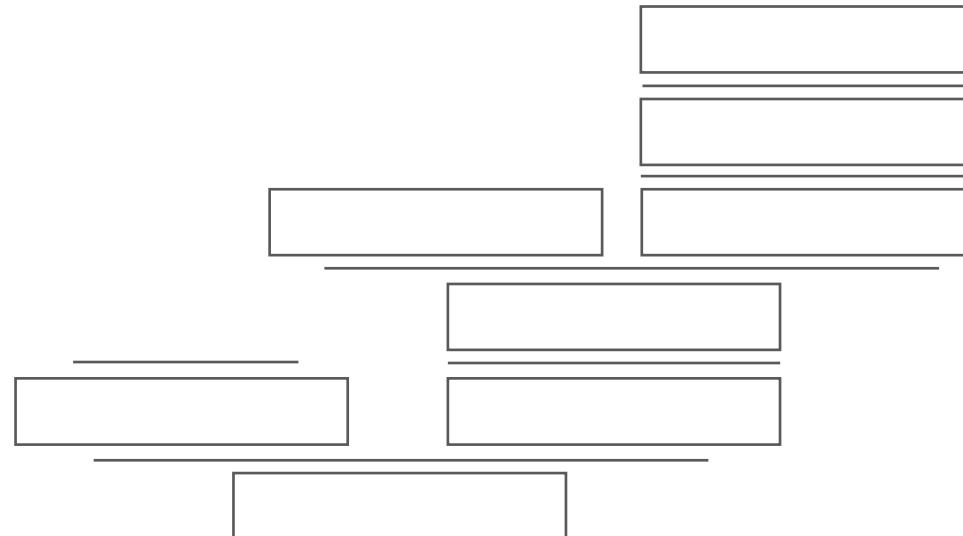
```
predicate list (loc x) {
| x = 0 ⇒ { emp }
| x ≠ 0 ⇒ { [x, 2] * x ↦ v * (x + 1) ↦ y * list(y) }
}
```

how do we generate a recursive call?

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

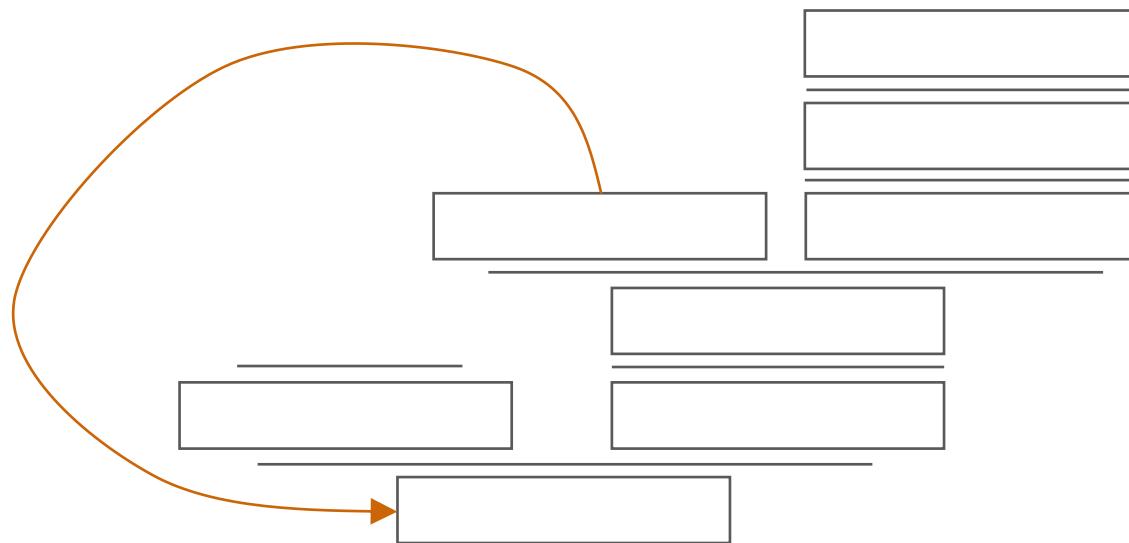
# cyclic proofs

proof “trees” with backlinks



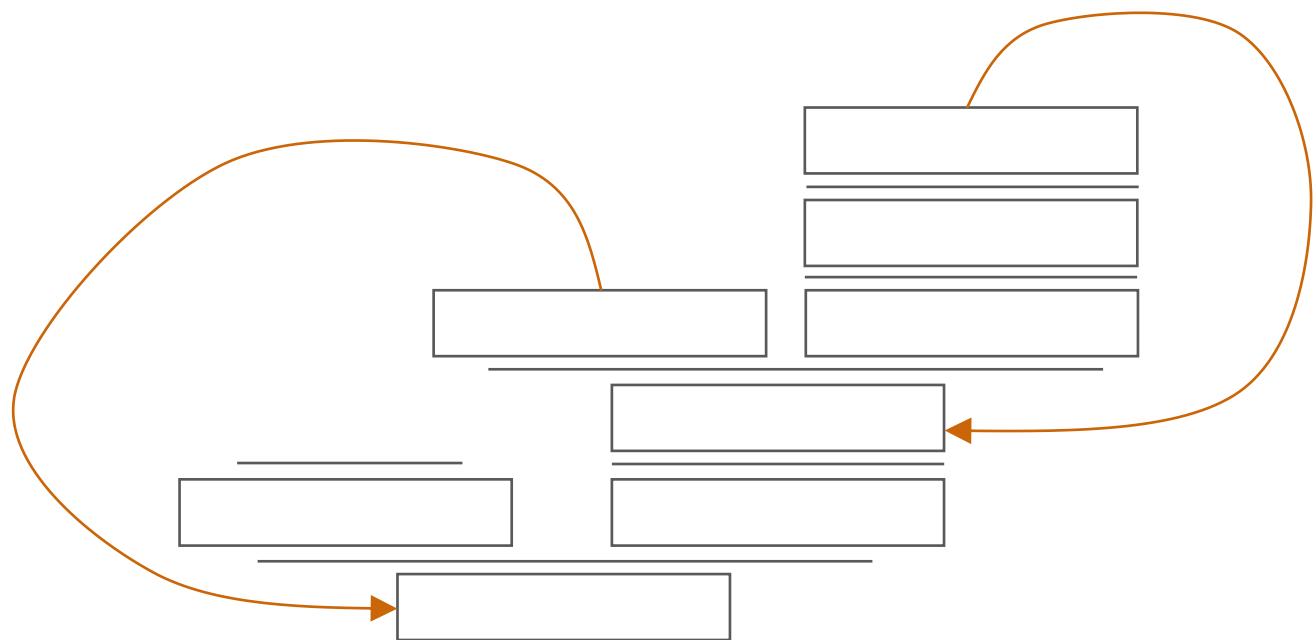
# cyclic proofs

proof “trees” with backlinks



# cyclic proofs

proof “trees” with backlinks



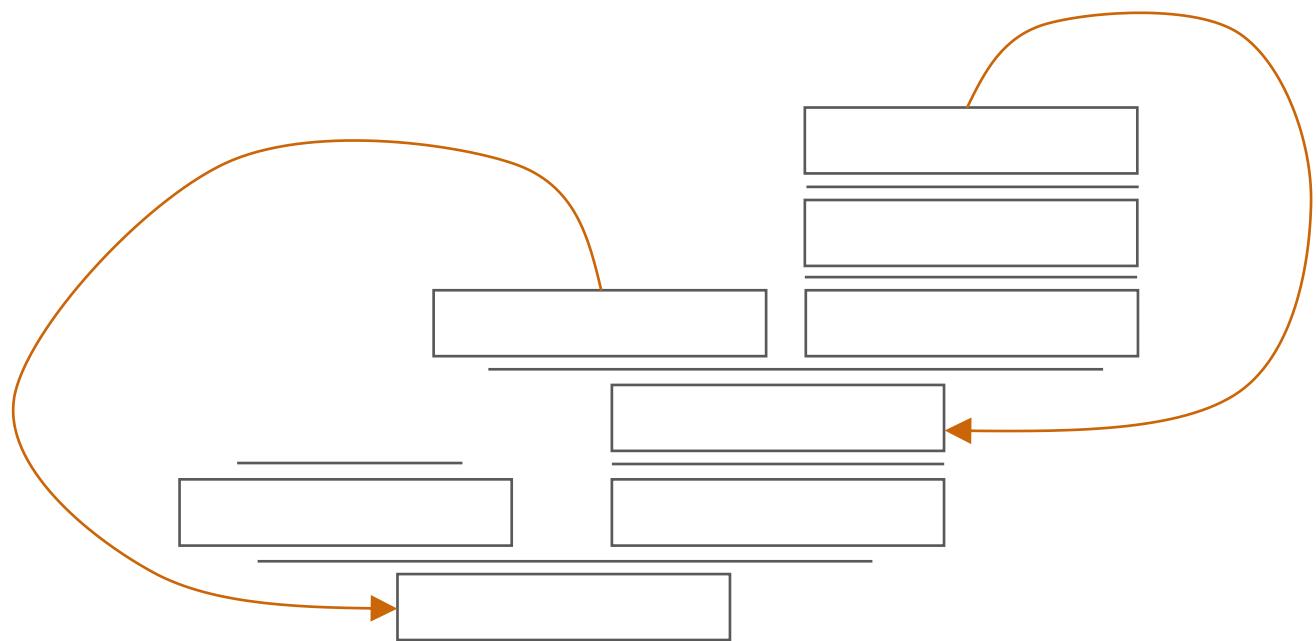
# cyclic proofs

proof “trees” with backlinks

verification of recursive programs:

[Brotherston et al. POPL'08]

[Rowe, Brotherston. CPP'17]



# cyclic proofs

proof “trees” with backlinks

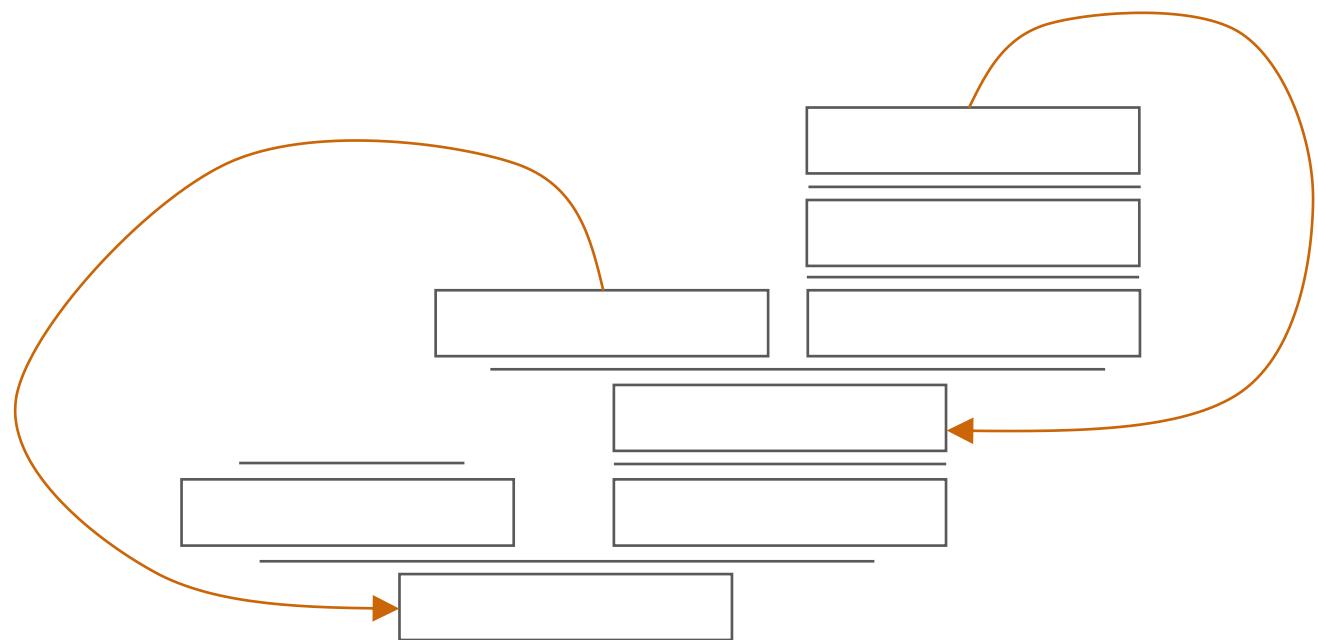
verification of recursive programs:

[Brotherston et al. POPL'08]

[Rowe, Brotherston. CPP'17]

synthesis of recursive programs:

SuSLik!



# dispose a list

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

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

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

# dispose a list

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

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

(Open)

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

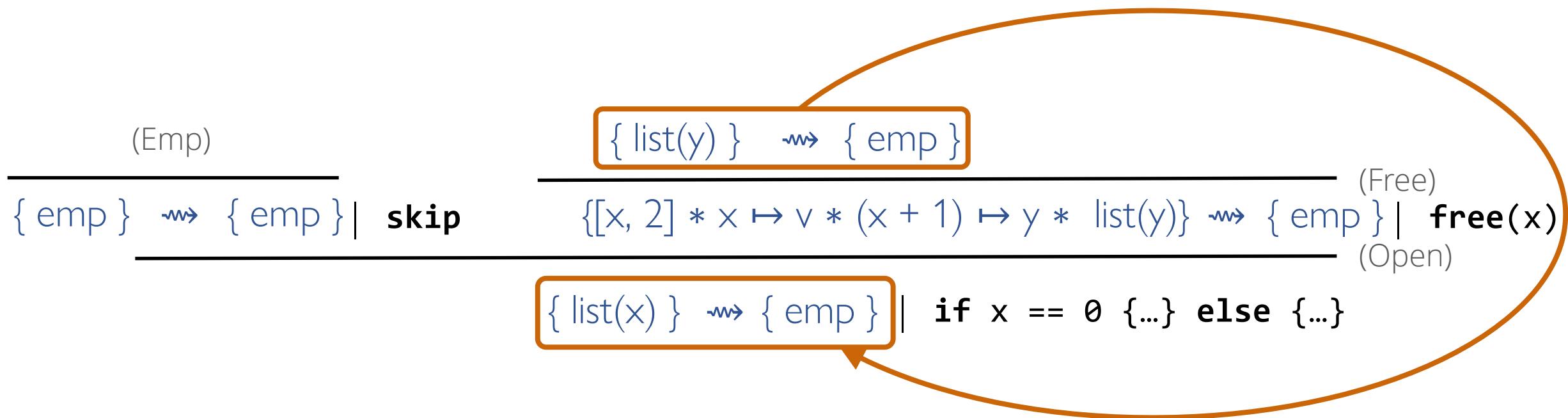
# dispose a list

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

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

# dispose a list

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



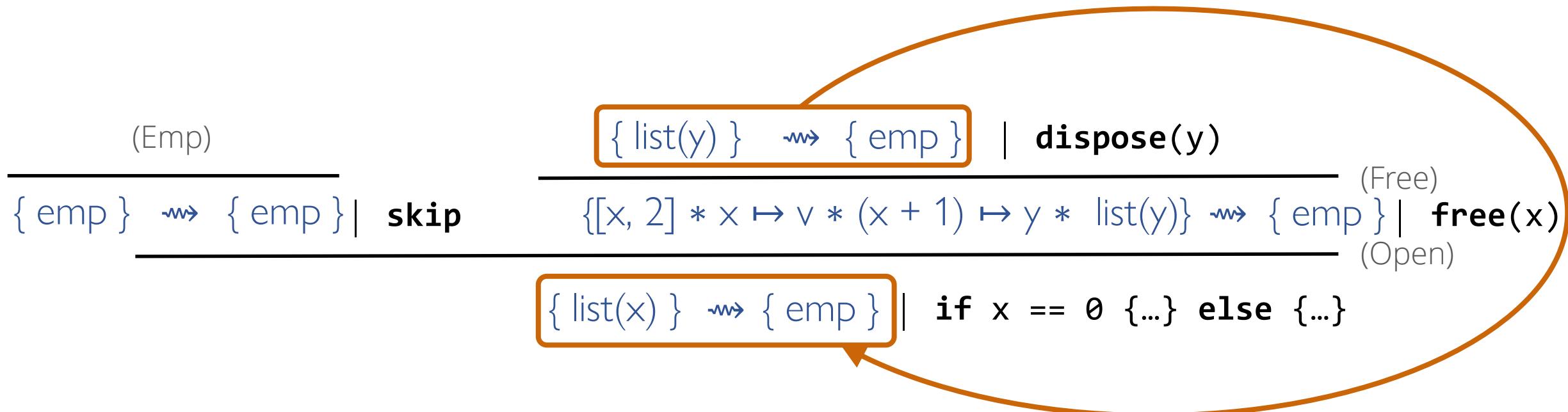
# dispose a list

```
dispose(loc x)
```

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

```
{ emp }
```

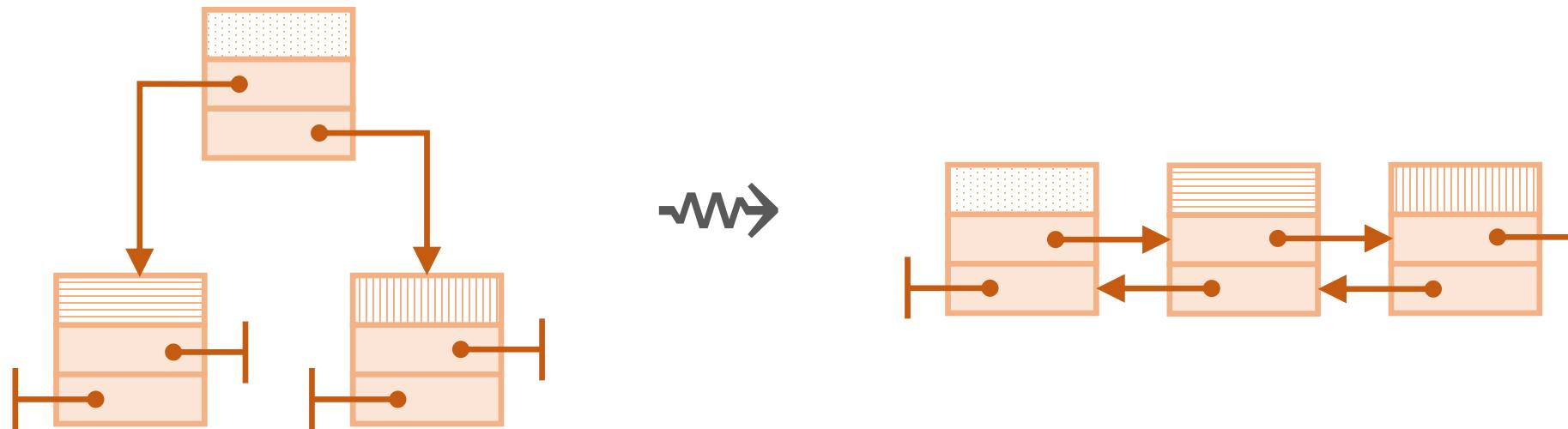
backlink generates a recursive call



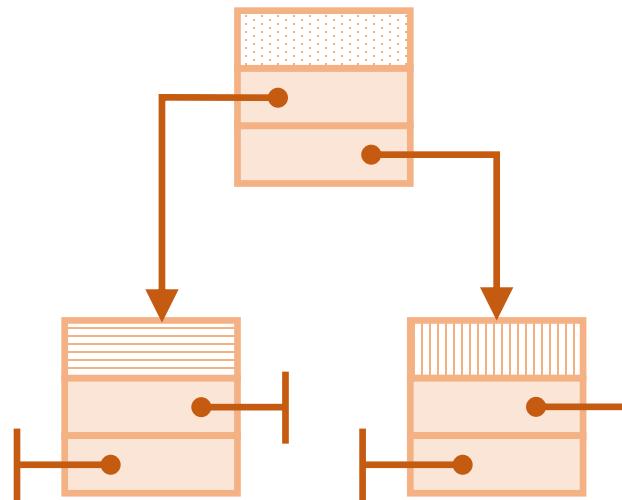
# this talk

1. a taste of SuSLik
2. recursion and cyclic proofs
3. learning auxiliary functions

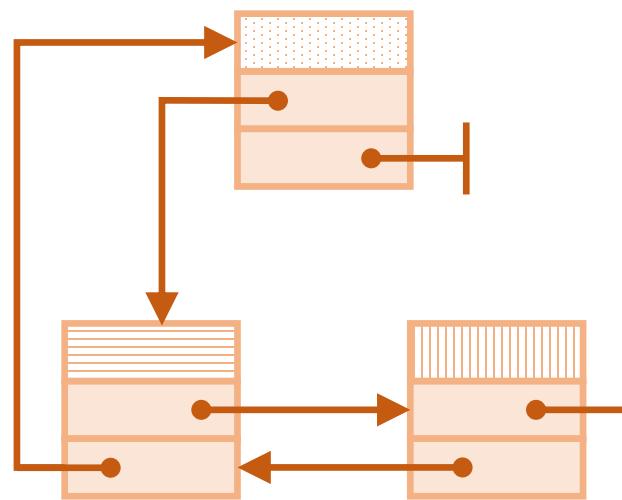
# task: flatten a tree into a list



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



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



# example 4: tree flattening

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

# example 4: tree flattening

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

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

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

set of elements

# example 4: 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**

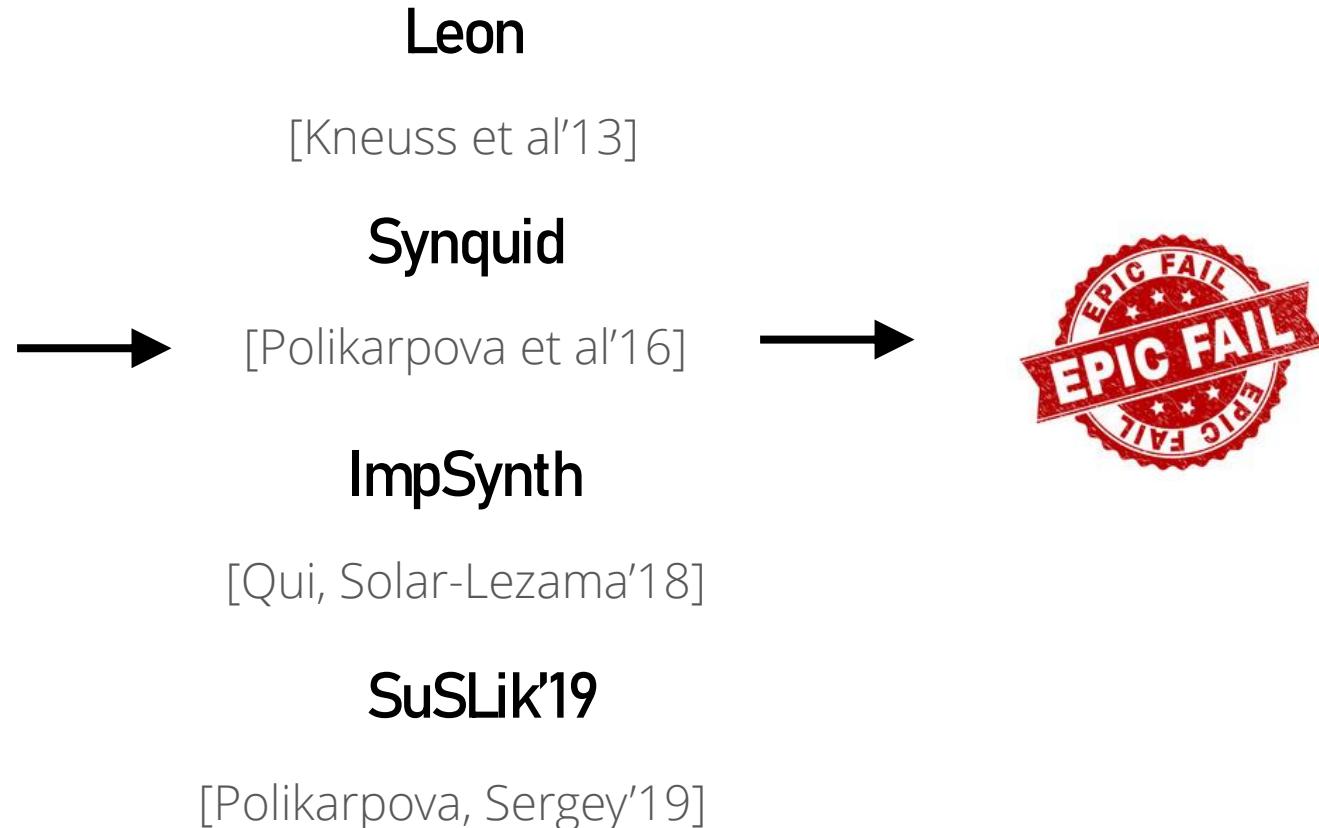
[Qui, Solar-Lezama'18]

**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]

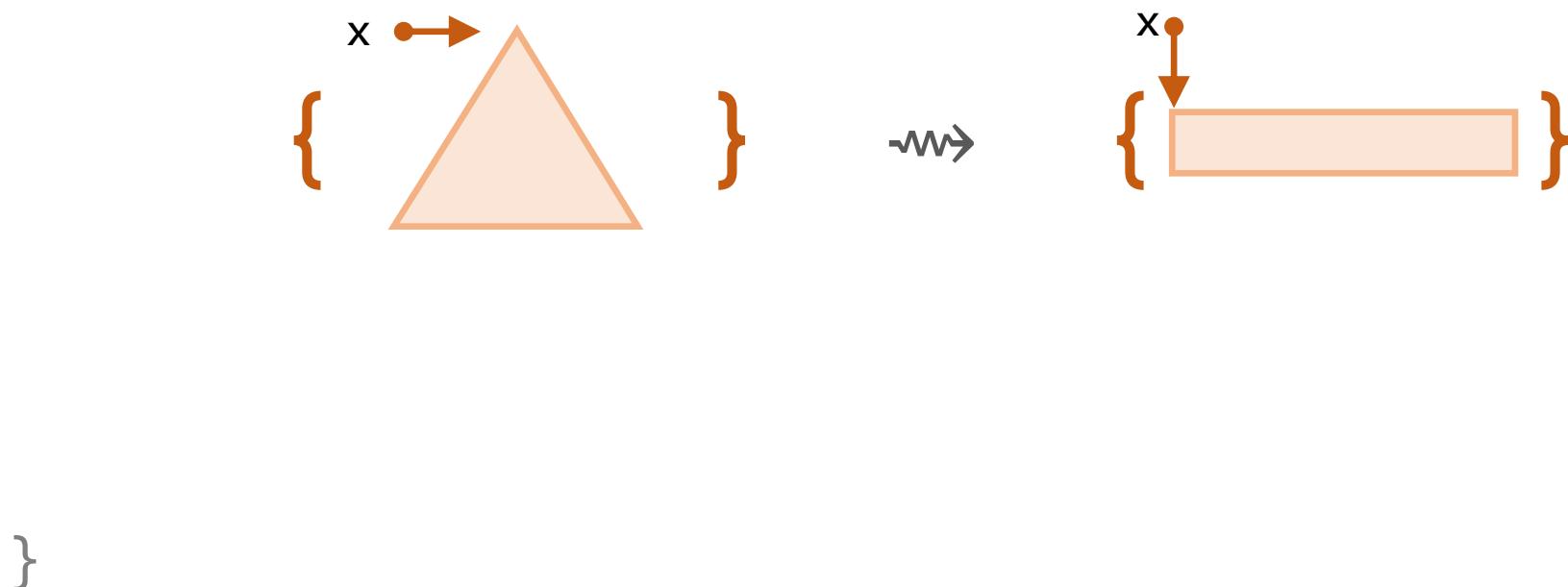
SuSLik'19

[Polikarpova, Sergey'19]



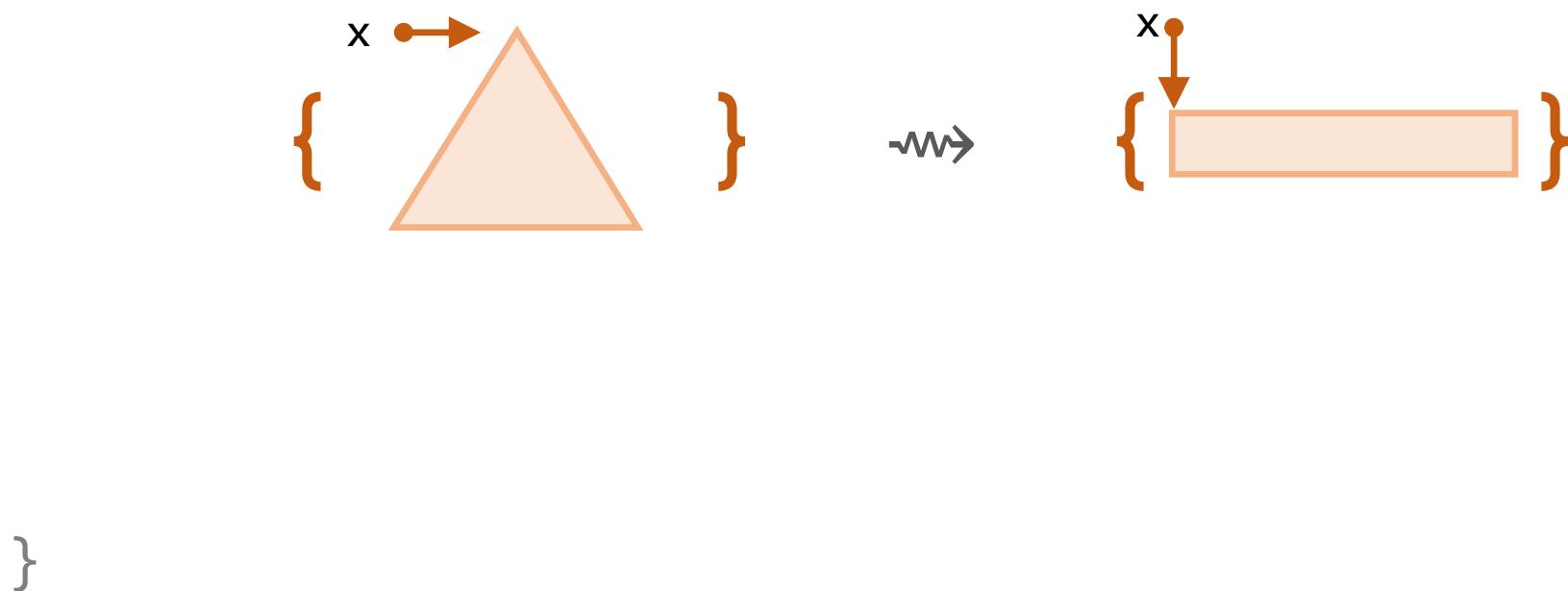
# tree flattening

```
flatten(x) {
```



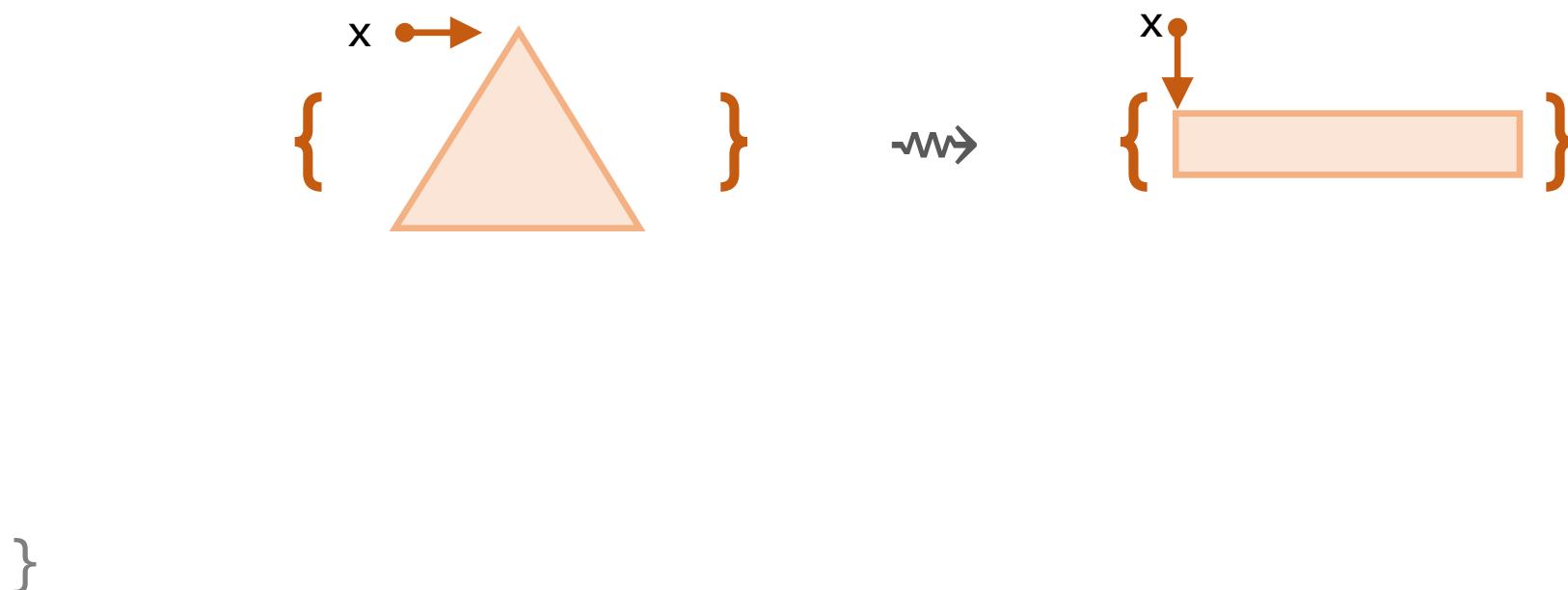
# step 1: pattern-match on input

```
flatten(x) {
```



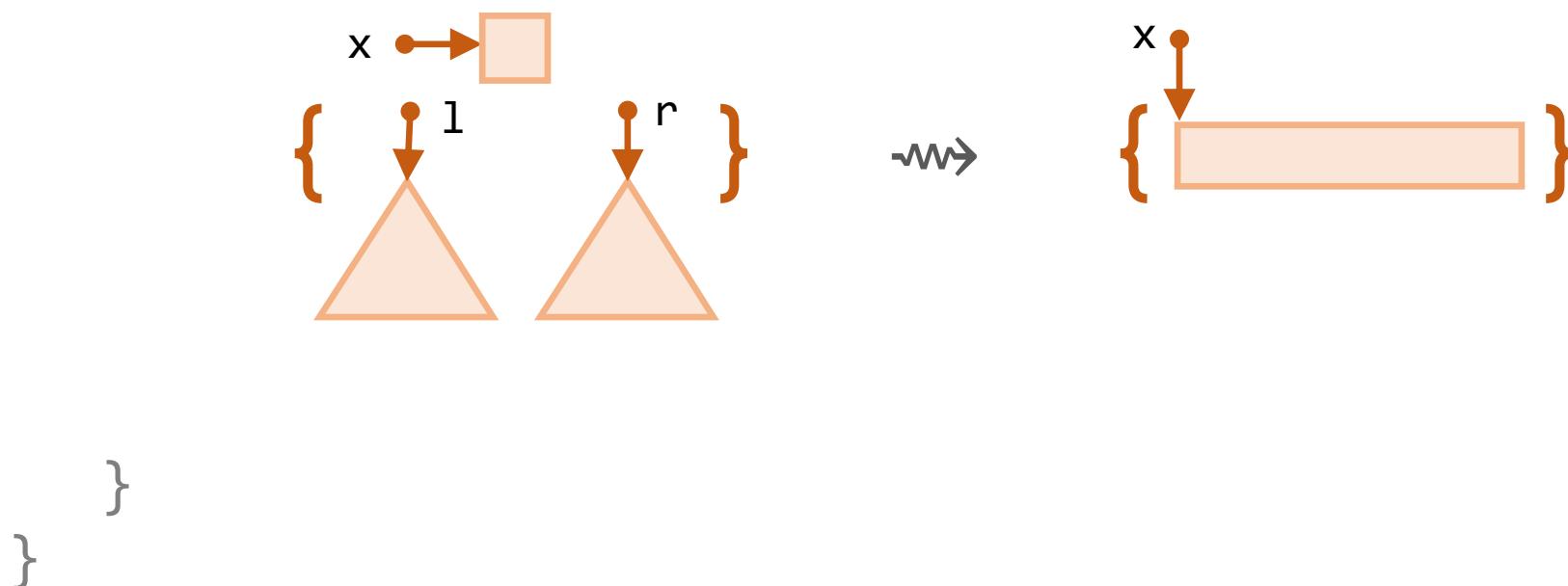
# step 1: pattern-match on input

```
flatten(x) {
```



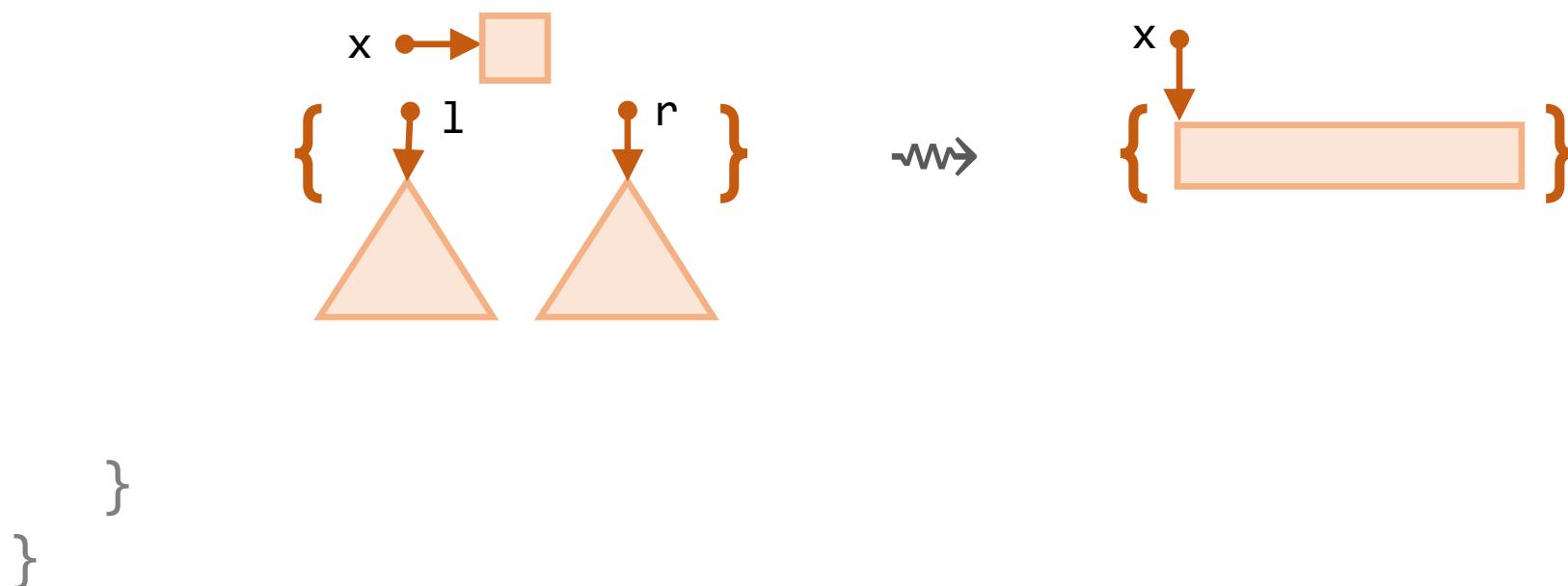
# step 1: pattern-match on input

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



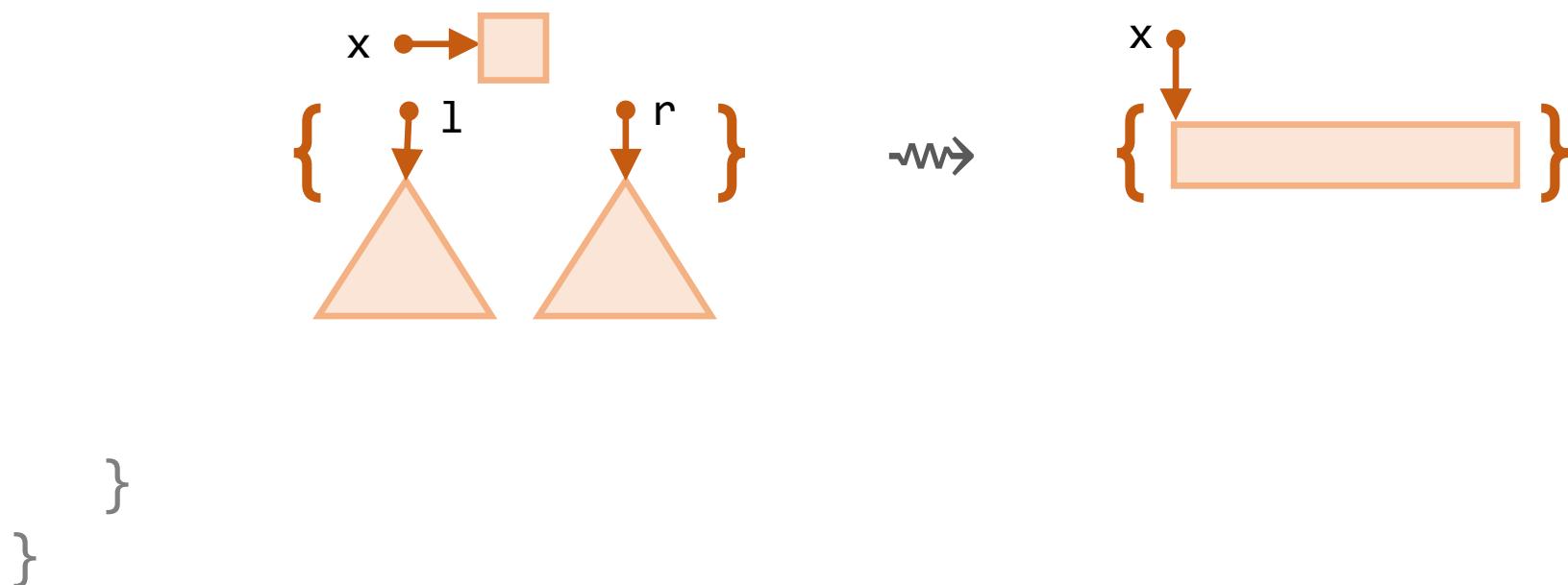
# step 1: pattern-match on input

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



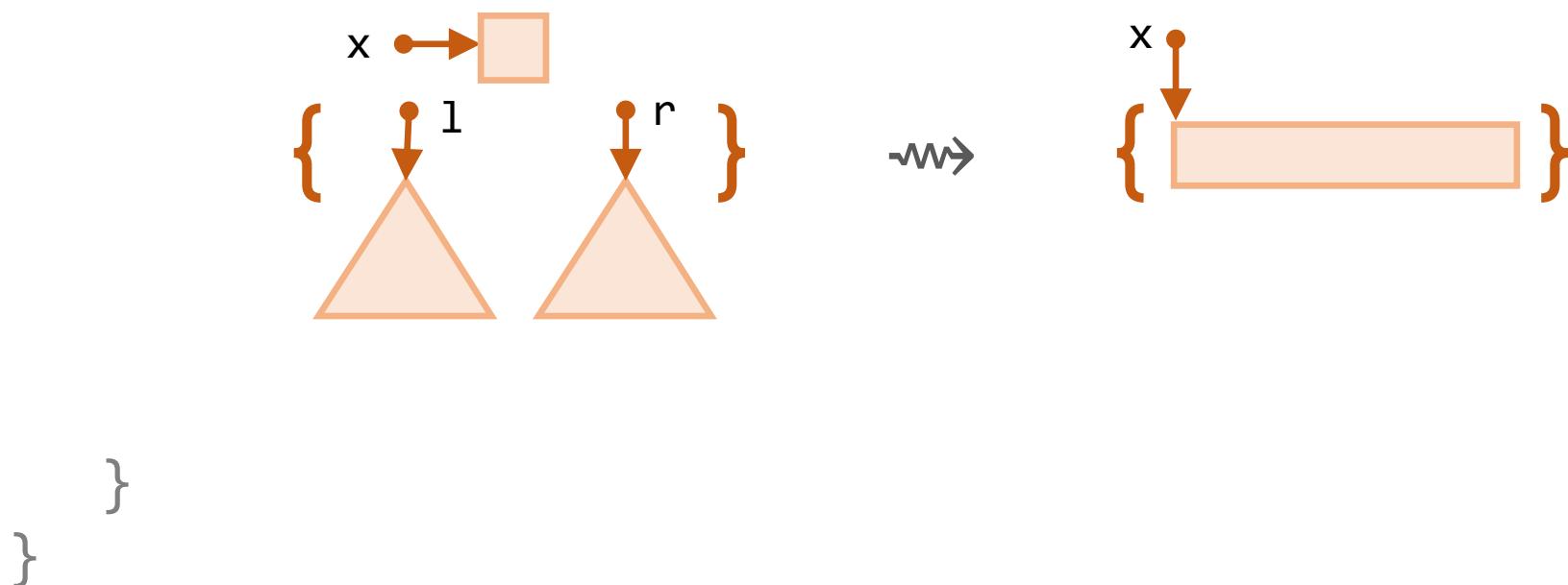
# step 1: pattern-match on input

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



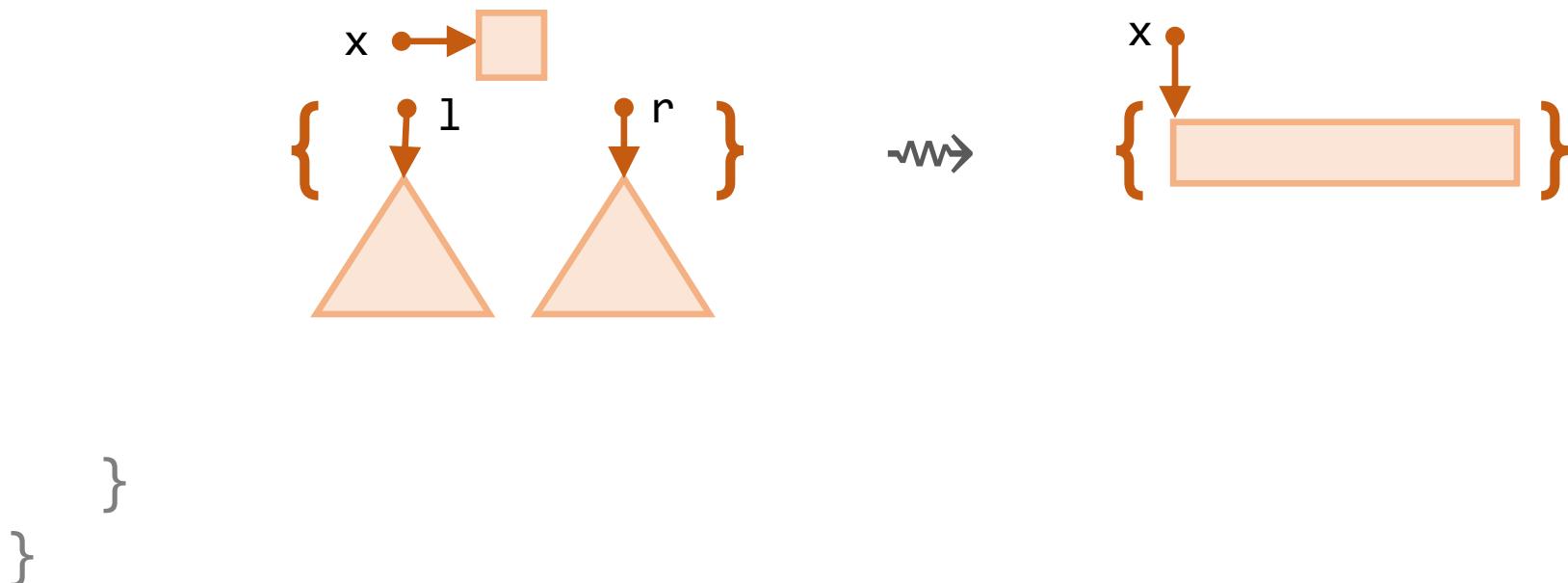
# step 1: pattern-match on input

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



# step 2: recursive call on l

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



# step 2: recursive call on l

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



# step 3: recursive call on r

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



# step 3: recursive call on r

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



# now what?

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



# now what?

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



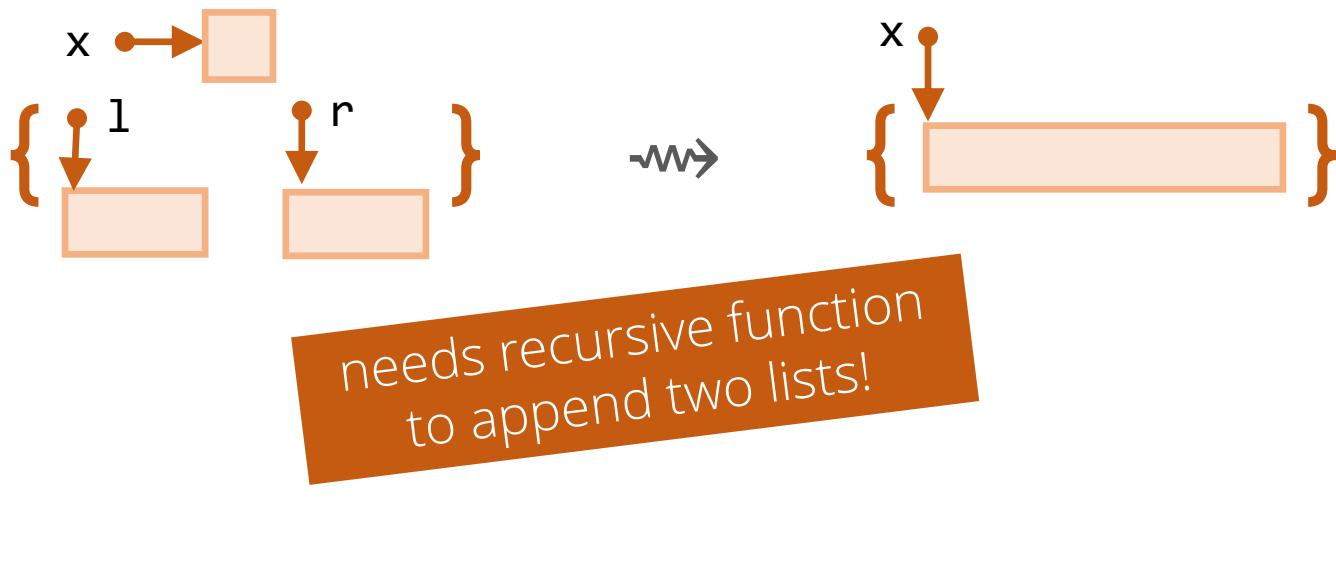
# now what?

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



# now what?

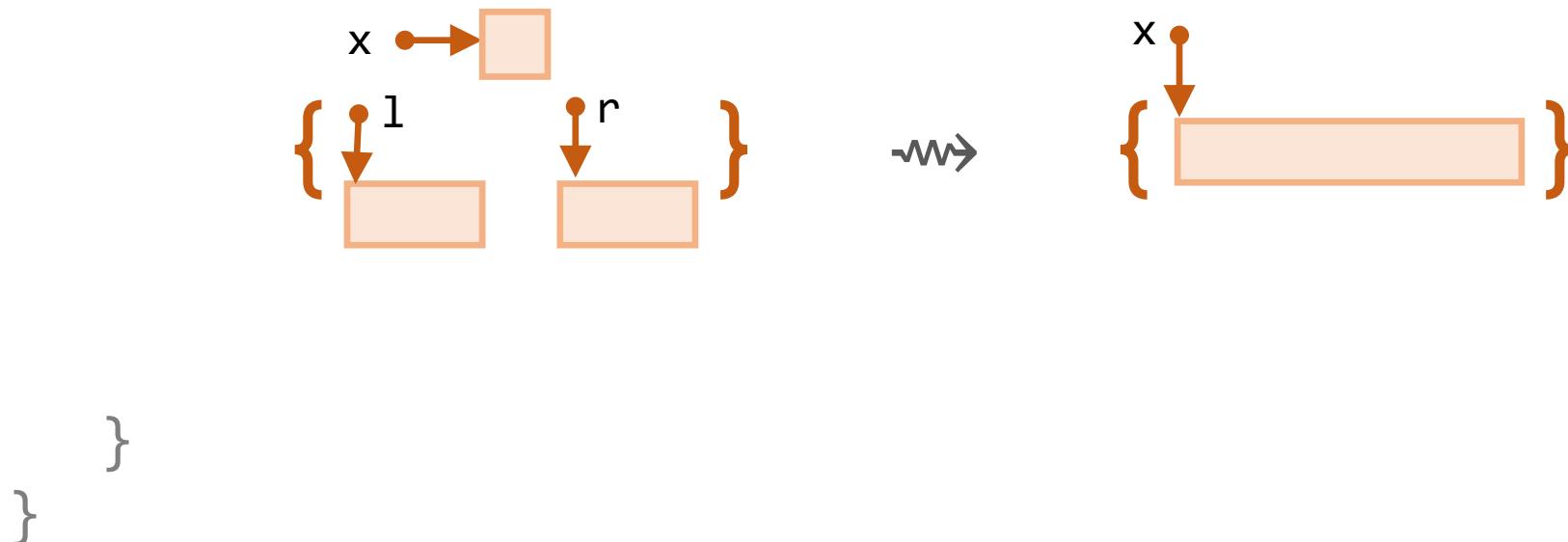
```
flatten(x) {  
    if (x != 0) {  
        l = *(x + 1); r = *(x + 2);  
        flatten(l); flatten(r);  
    }  
}
```



# tree flattening with cyclic proofs

```
flatten(x) {  
    if (x != θ) {
```

...

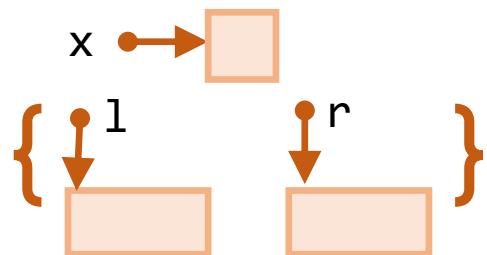


# tree flattening with cyclic proofs

```
flatten(x) {  
    if (x != θ) {
```

...

```
}
```



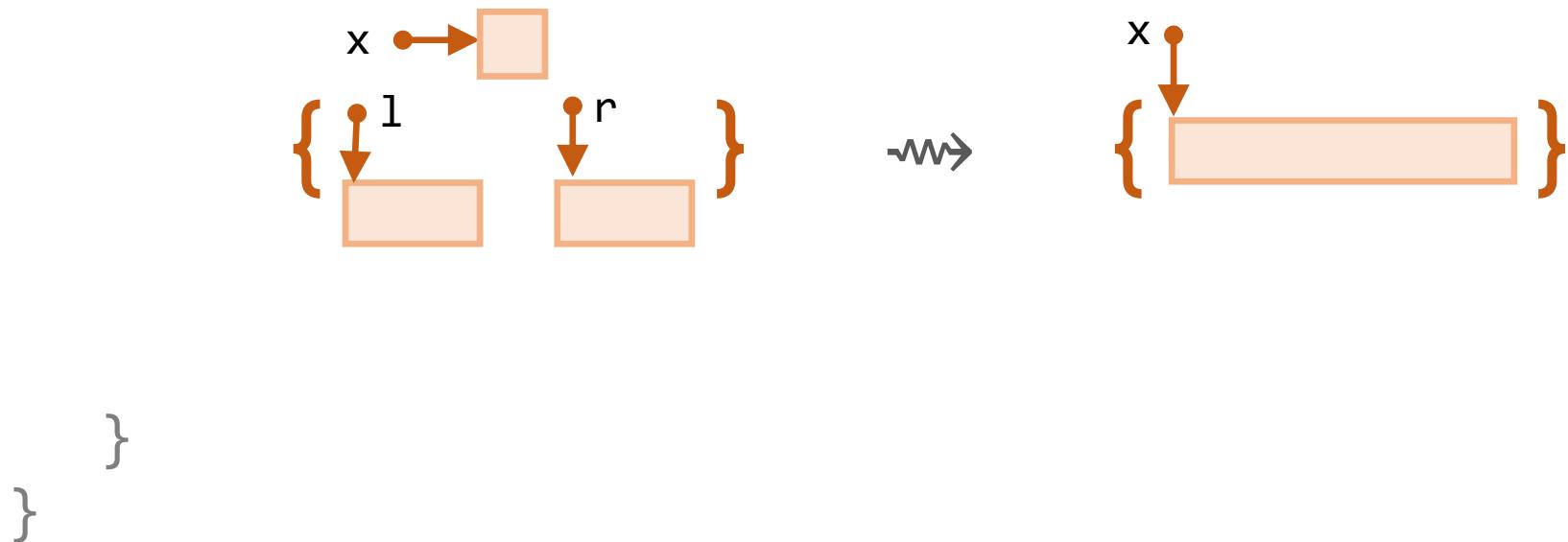
G



# step 4: pattern match on list l

```
flatten(x) {  
    if (x != θ) {
```

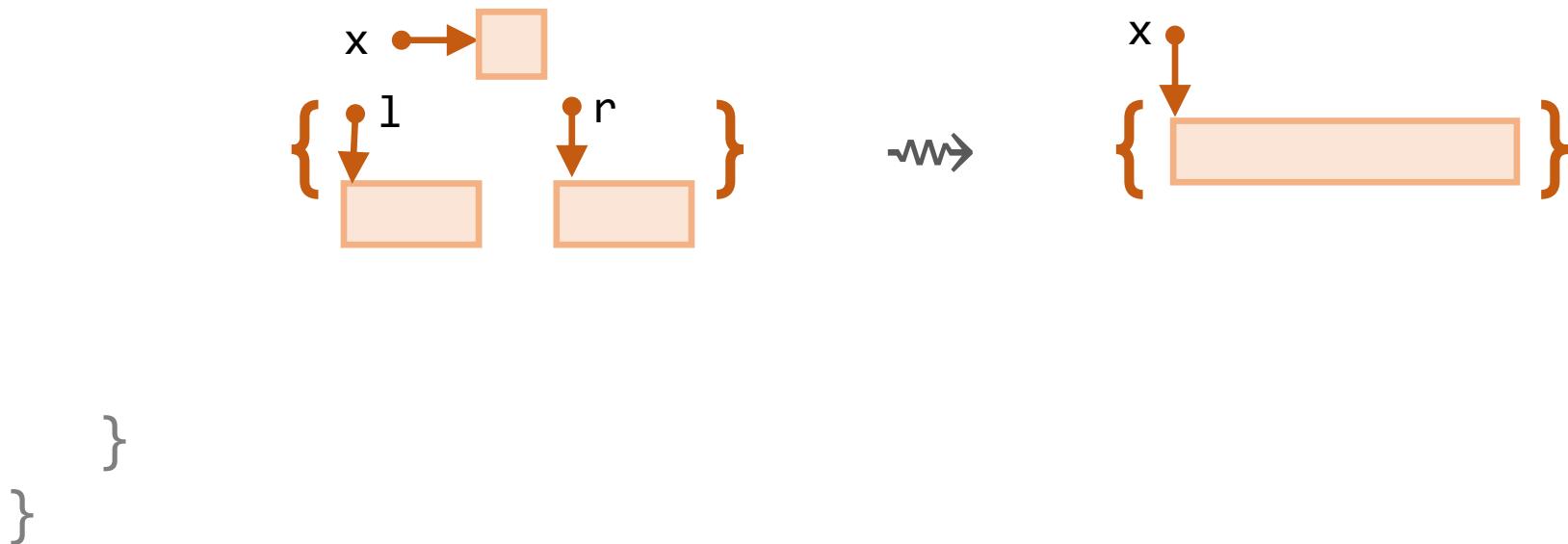
...



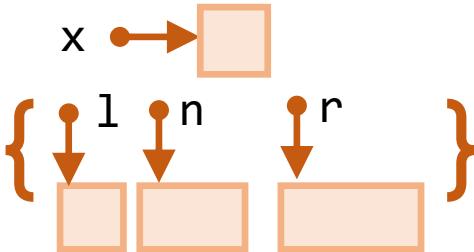
# step 4: pattern match on list l

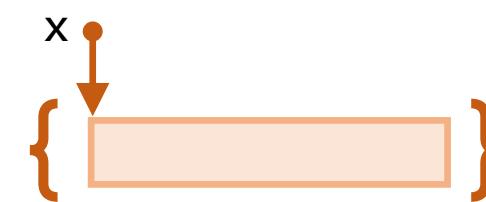
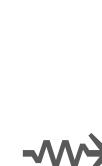
```
flatten(x) {  
    if (x != θ) {
```

...

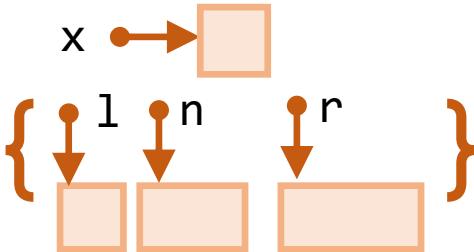


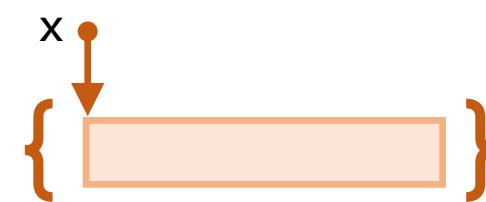
# step 4: pattern match on list l

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



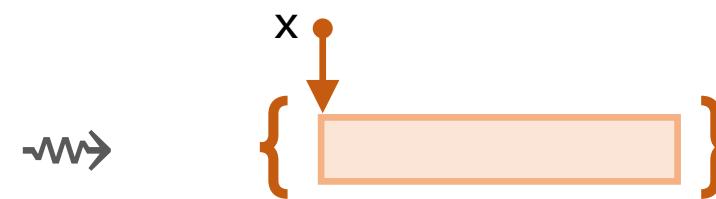
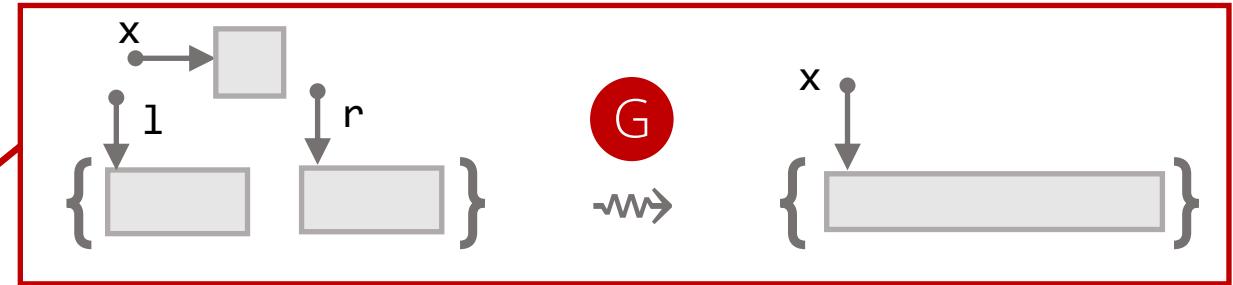
# does this goal look familiar?

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



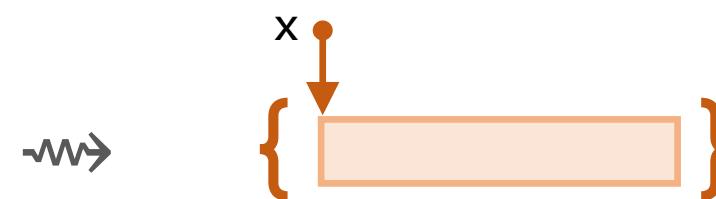
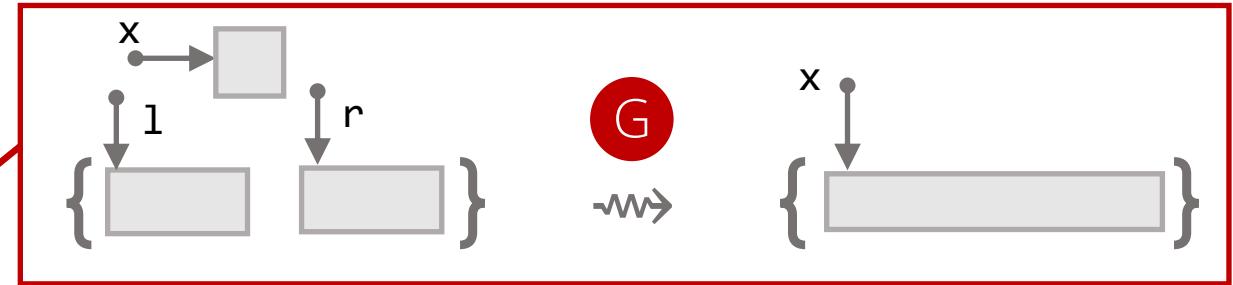
# does this goal look familiar?

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



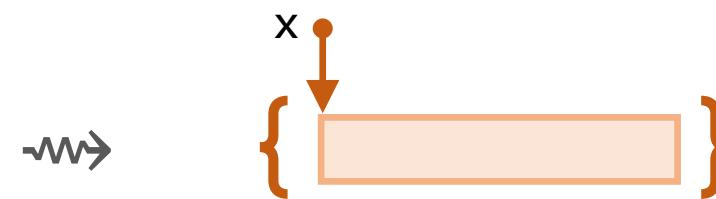
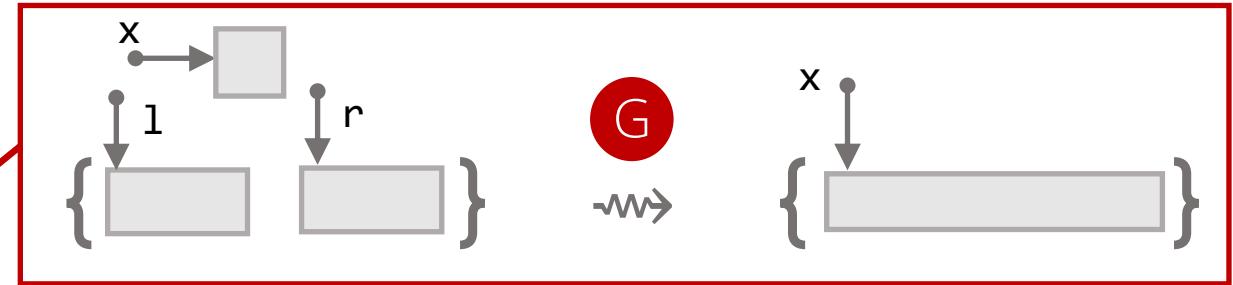
# does this goal look familiar?

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



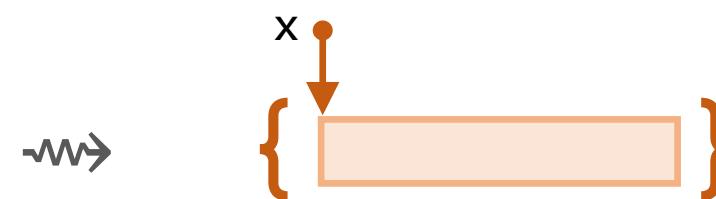
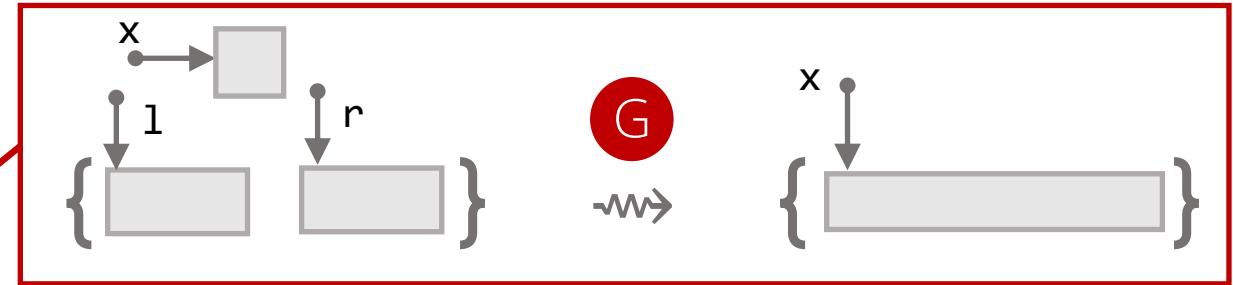
# does this goal look familiar?

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



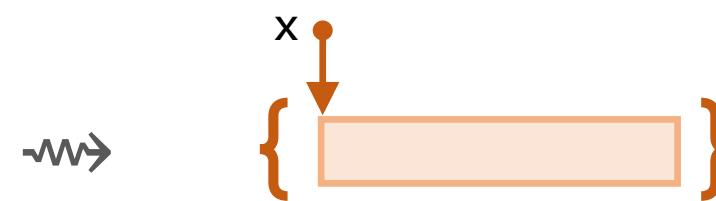
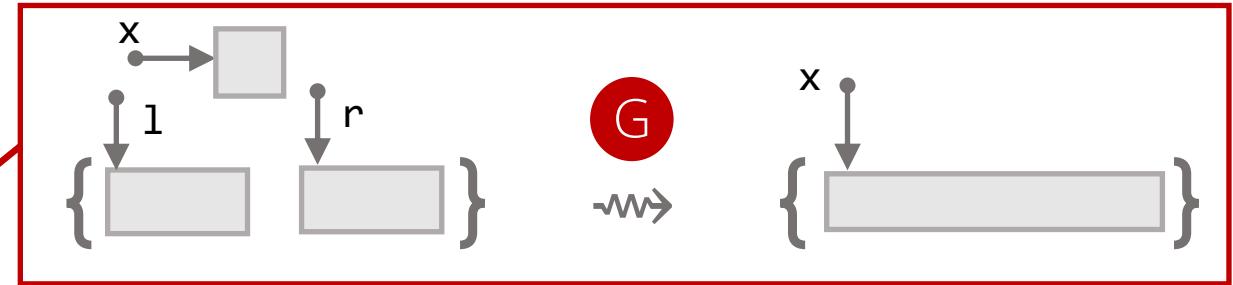
# does this goal look familiar?

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



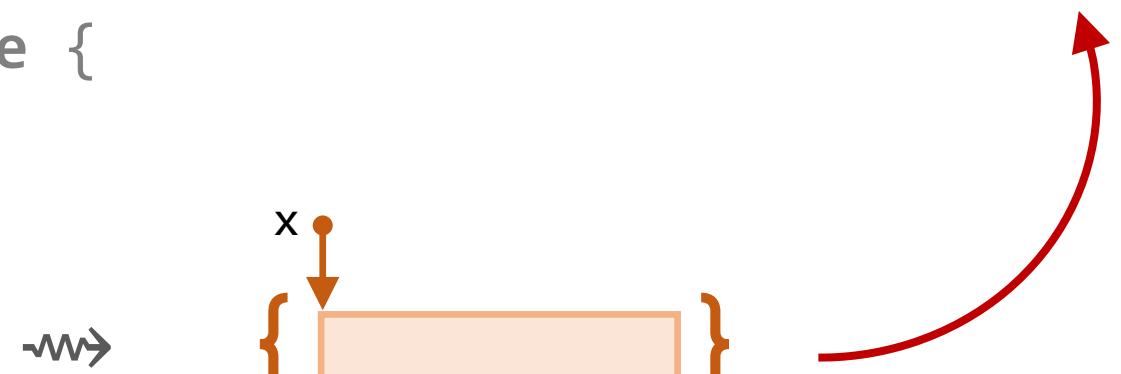
# does this goal look familiar?

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



# link it back!

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



# link it back!

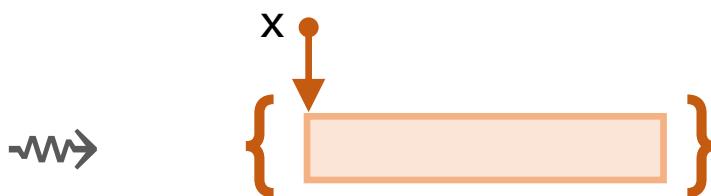
```
flatten(x) {  
    if (x != 0) {  
        ...  
        if (l == 0) { ... } else {  
            n = *(l + 1);  
  
            x →   
            { l n r }  
        }  
    }  
}
```



helper(n, r, l)

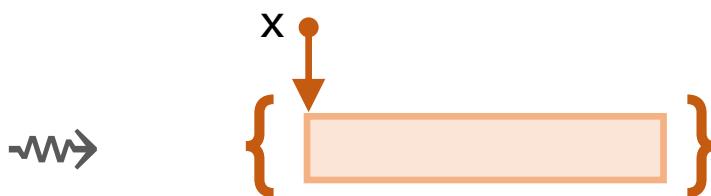
# link it back!

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



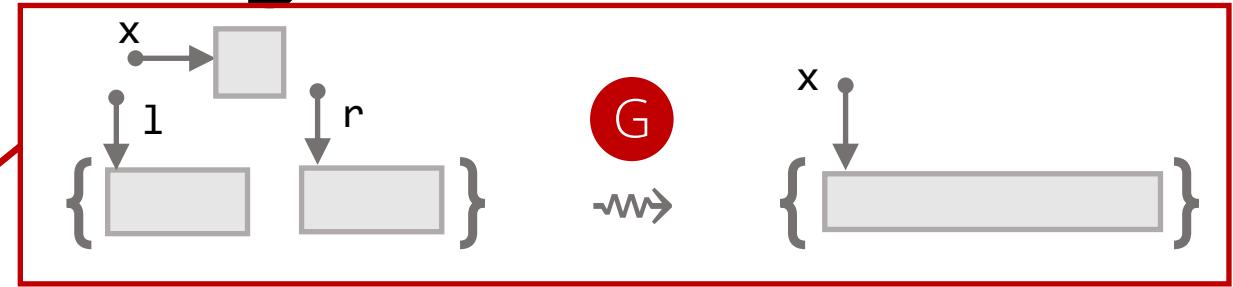
# link it back!

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



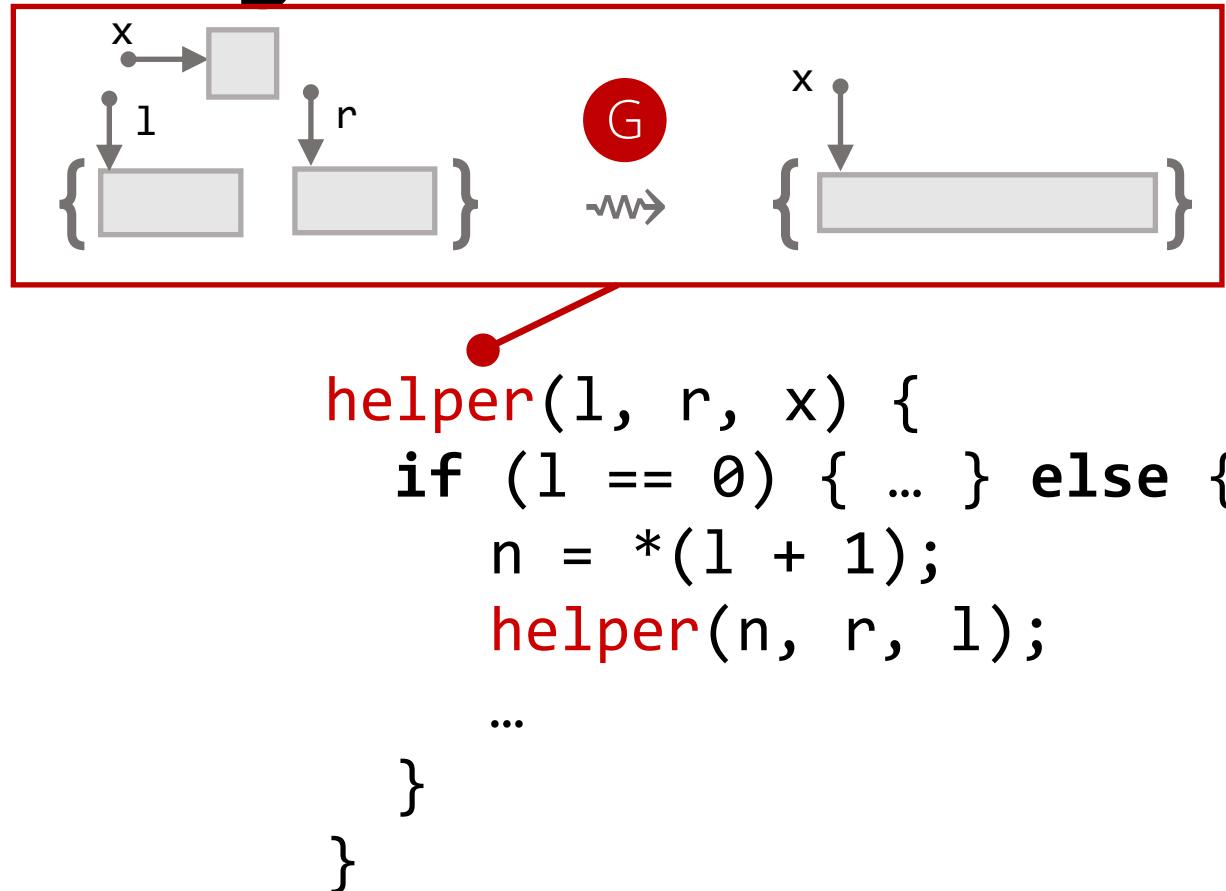
# extracting the auxiliary

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

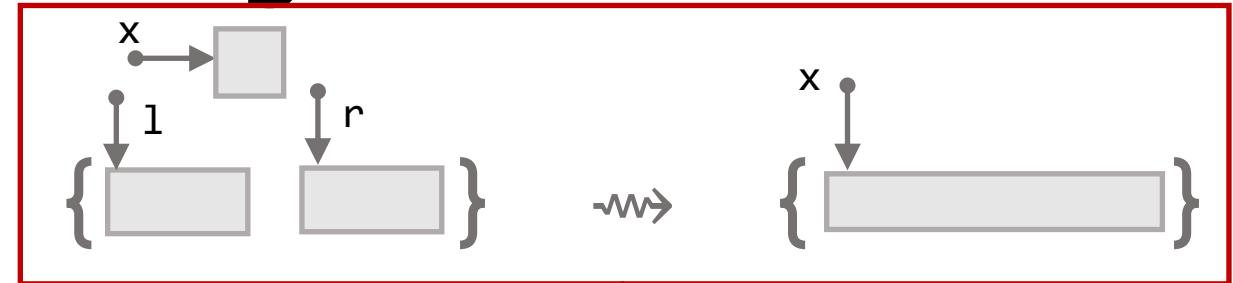


# extracting the auxiliary

```
flatten(x) {  
    if (x != 0) {  
        ...  
        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 4: tree flattening

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

# **what else can SuSLik do?**

# **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 trees

# 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 trees

54 benchmarks

# 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 trees

54 benchmarks

up to 32 statements

# 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 trees

54 benchmarks

up to 32 statements

up to 12x code/spec ratio

# 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 trees

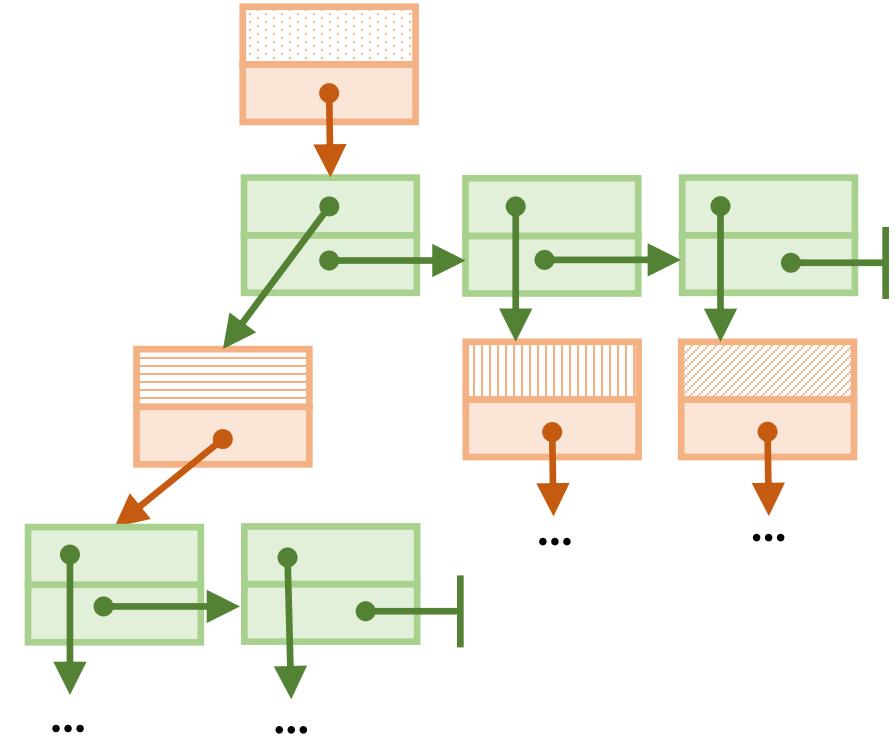
54 benchmarks

up to 32 statements

up to 12x code/spec ratio

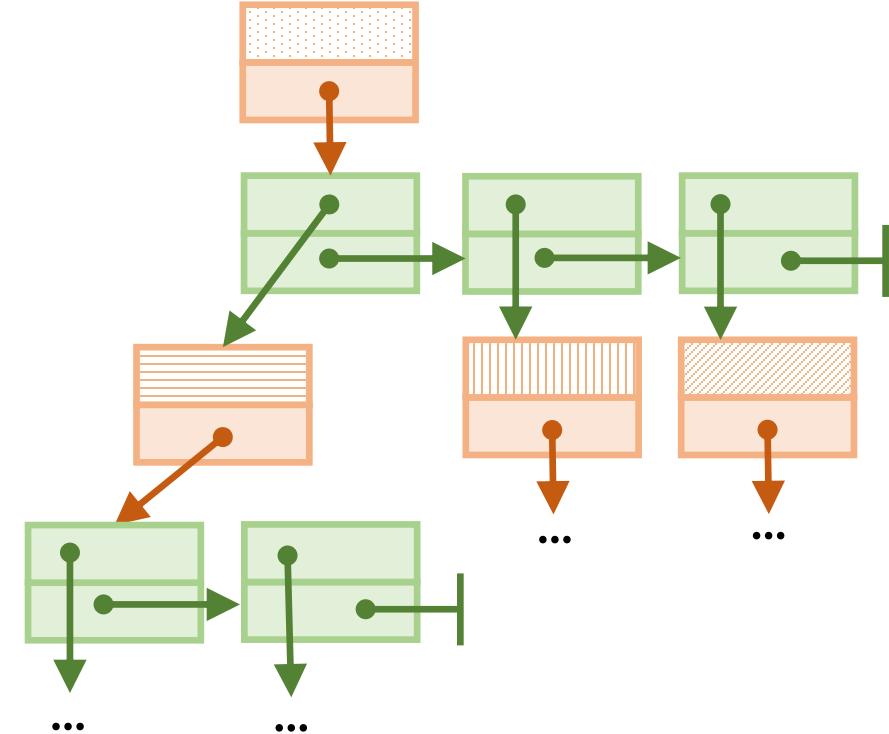
synthesis times: < 2 min

# example: dispose rose tree



# example: dispose rose tree

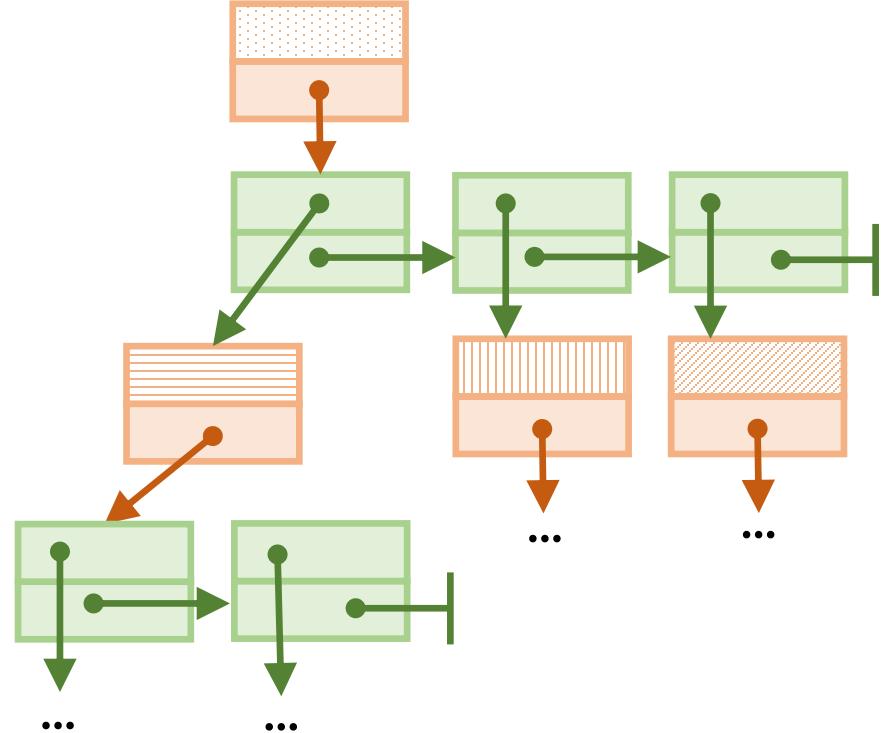
{ rtree(x) }  $\rightsquigarrow$  { emp }



# example: dispose rose tree

{ rtree(x) }  $\rightsquigarrow$  { emp }

```
dispose(x) {
    if (x != 0) {
        dispose_children(x);
    }
}
```

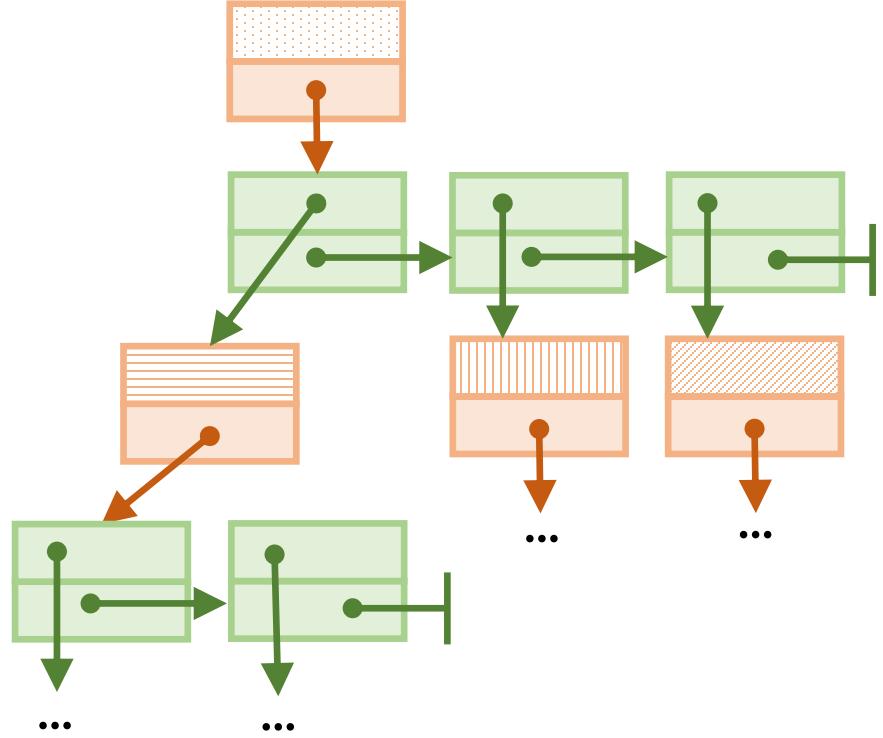


# example: dispose rose tree

{ rtree(x) }  $\rightsquigarrow$  { emp }

```
dispose(x) {
    if (x != 0) {
        dispose_children(x);
    }
}
```

```
dispose_children(x) {
    c = *(x + 1);
    free(x);
    if (c != 0) {
        r = *c; dispose(r); dispose_children(c);
    }
}
```



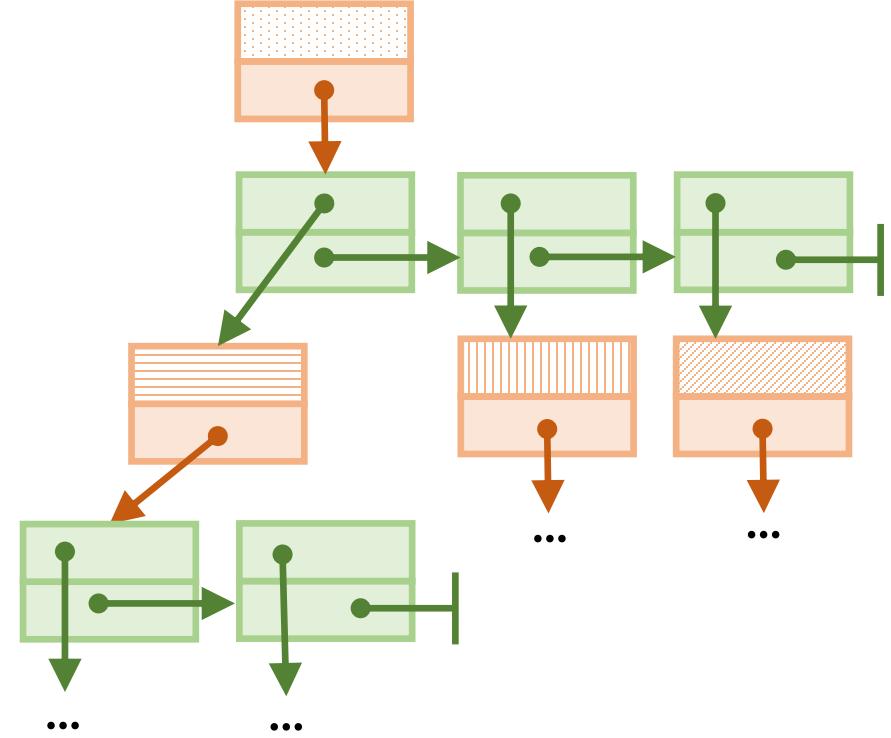
# example: dispose rose tree

{ rtree(x) }  $\rightsquigarrow$  { emp }

```
dispose(x) {
    if (x != 0) {
        dispose_children(x);
    }
}

dispose_children(x) {
    c = *(x + 1);
    free(x);
    if (c != 0) {
        r = *c; dispose(r); dispose_children(c);
    }
}
```

mutual recursion



# **what is out of scope?**

auxiliaries with accumulators

e.g. linear-time list reversal, merge sort

# what is out of scope?

auxiliaries with accumulators

e.g. linear-time list reversal, merge sort

future work: can we allow more auxiliaries w/o exploding search space?

# what is out of scope?

auxiliaries with accumulators

e.g. linear-time list reversal, merge sort

future work: can we allow more auxiliaries w/o exploding search space?

search space too large

e.g. rose tree copy

current search relies on brittle cost heuristic

# what is out of scope?

auxiliaries with accumulators

e.g. linear-time list reversal, merge sort

**future work:** can we allow more auxiliaries w/o exploding search space?

search space too large

e.g. rose tree copy

current search relies on brittle cost heuristic

**future work:** can we learn an adaptive cost function?

# what is out of scope?

auxiliaries with accumulators

e.g. linear-time list reversal, merge sort

**future work:** can we allow more auxiliaries w/o exploding search space?

search space too large

e.g. rose tree copy

current search relies on brittle cost heuristic

**future work:** can we learn an adaptive cost function?

**future work:** combine automatic and interactive synthesis?

# thanks to my collaborators!



Ilya Sergey  
(Yale-NUS)



Shachar Itzhaky  
(Technion)



Reuben Rowe  
(Royal Holloway)



Hila Peleg  
(UCSD/Technion)



Andreea Costea  
(Yale-NUS)

Amy Zhu

Roman Shchedrin

Yasunari Watanabe

Kiran Gopinathan

George Pîrlea

# learn more

Polikarpova, Sergey [POPL'19]: deductive synthesis meets separation logic

Costea et al. [ESOP'20]: immutability annotations for more robust synthesis

Itzhaky et al. [PLDI'21]: SuSLik meets cyclic proofs

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

Watanabe at al. [ICFP'21]: translating SuSLik derivations into Coq

<https://suslik.programming.systems/>