Deductive Synthesis of Programs with Pointers: Expressive, Trustworthy, Fast

Ilya Sergey

ilyasergey.net

NUS
National University of Singapore

demo.c — listcopy

demo.c

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* create(int arr[], int N)
{
    struct Node* head_ref = NULL;
    for (int i = N - 1; i >= 0; i--) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = arr[i];
        newNode->next = head_ref;
        head_ref = newNode;
    }
    return head_ref;
}
```

PROBLEMS OUTPUT TERMINAL

TERMINAL

```
ilya-thunderbolt:listcopy ilya$
```

master ⊗ 0 ⚠ 0 Ln 16, Col 28 Spaces: 4 UTF-8 LF C *programs*

On ~~theories~~ such as these we cannot rely.

Proof we need. Proof!

Program Synthesis that We Can Trust

Given a *specification*,
automatically generate a *program*
that *provably* satisfies it.

This Talk

Program Synthesis as automated *proof search*

(aka *Deductive Synthesis*)

Today's Agenda

- Deductive synthesis in a nutshell
- Trust in program synthesis
- Extensions and Applications

Today's Agenda

- Deductive synthesis in a nutshell
- Trust in program synthesis
- Extensions and Applications

Let’s *swap* values of two *distinct* pointers

Let's *swap* values of two *distinct* pointers

Let's *swap* values of two *distinct* pointers

$x \mapsto b$ $\quad$ $y \mapsto a$

swap

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

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

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

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

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

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

"separately"

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

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

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

Peter W. O'Hearn, John C. Reynolds, Hongseok Yang:
Local Reasoning about Programs that Alter Data Structures. CSL 2001

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

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

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

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

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

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

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

??

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

```
let a2 = *x;
```

$\{\ x \mapsto a2 * y \mapsto \boxed{b}\ \}$

??

$\{\ x \mapsto \boxed{b} * y \mapsto a2\ \}$

```
let a2 = *x;
let b2 = *y;
```

$\{\ x \mapsto a2 \ast y \mapsto b2\ \}$

??

$\{\ x \mapsto b2 \ast y \mapsto a2\ \}$

```
let a2 = *x;
let b2 = *y;
*x = b2;
```

$\{ x \mapsto b2 * y \mapsto b2 \}$

??

$\{ x \mapsto b2 * y \mapsto a2 \}$

```
let a2 = *x;
let b2 = *y;
*x = b2;
*y = a2;
```

$\{ x \mapsto b2 * y \mapsto a2 \}$

??

$\{ x \mapsto b2 * y \mapsto a2 \}$

```
let a2 = *x;
let b2 = *y;
*x = b2;
*y = a2;
```

$\{\ x \mapsto b2 * y \mapsto a2\ \}$

??

$\{\ x \mapsto b2 * y \mapsto a2\ \}$

$x \mapsto b2 * y \mapsto a2 \ \vdash\ x \mapsto b2 * y \mapsto a2$

```
let a2 = *x;
let b2 = *y;
*x = b2;
*y = a2;
```

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

??

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

$$x \mapsto b2 * y \mapsto a2 \ \vdash\ x \mapsto b2 * y \mapsto a2$$ ✓

```
void swap(loc x, loc y) {
        let a2 = *x;
        let b2 = *y;
        *x = b2;
        *y = a2;
}
```

Reasoning with Symbolic Heaps

Symbolic Heap Entailment

$$P \vdash Q$$

Any heap (state) that satisfies **P**, also satisfies **Q**.

Hoare-style Pre/Postcondition

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

If the initial state satisfies **P**, then, after **c** terminates, the final state satisfies **Q**.

Separation Logic

{ **P** } **c** { **Q** }

If the initial state satisfies **P**, then
program **c** will execute *without memory errors*
and after it terminates, the final state satisfies **Q**.

Transforming Entailment

(our invention)

$$P \rightsquigarrow Q$$

There *exists* a program **c**, such that
for any initial state satisfying **P**,
c, after it terminates,
will transform to a state satisfying **Q**.

$P \vdash Q$ implies $P \rightsquigarrow Q$

"Proof": `skip`

$x \mapsto a \quad \rightsquigarrow \quad x \mapsto 42$

“Proof”: `*x = 42`

$x \mapsto a \rightsquigarrow x \mapsto 42$ | `*x = 42`

$$P \rightsquigarrow Q \mid c$$

P transforms to Q via a program **c**.

Theorem:

$$P \rightsquigarrow Q \mid c \quad \text{implies} \quad \{\, P \,\}\ c\ \{\, Q \,\}$$

Synthetic Separation Logic

$$\Gamma \,;\, P \rightsquigarrow Q \mid c$$

$$\Gamma\ ;\ P \rightsquigarrow Q \mid c$$

- (Γ, P, Q) — *goal*
- **GV** (Γ, P, Q) — *ghost* variables (scope: *pre/postcondition*)
- **EV** (Γ, P, Q) — *existentials* (scope: *postcondition*)

$$\Gamma; \{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid \mathbf{??}$$

$$\Gamma; \{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid \texttt{skip} \qquad \text{(Emp)}$$

$$a \in \mathrm{GV}(\Gamma, P, Q)$$

$$\Gamma; \{\, x \mapsto a * P \,\} \rightsquigarrow \{\, Q \,\} \mid \texttt{??}$$

$$\frac{a \in \mathrm{GV}(\Gamma, P, Q) \qquad y \text{ is fresh} \qquad \Gamma, y\, ; [y/a]\{\, x \mapsto y * P \,\} \rightsquigarrow [y/a]\{\, Q \,\} \mid \mathtt{c}}{\Gamma; \{\, x \mapsto a * P \,\} \rightsquigarrow \{\, Q \,\} \mid \mathtt{let\ y\ =\ *x;\ c}}\ \text{(Read)}$$

$$\Gamma; \{ x \mapsto - * P \} \rightsquigarrow \{ x \mapsto e * Q \} \mid ??$$

$$\frac{\mathit{Vars}(e) \subseteq \Gamma \qquad \Gamma ; \{ x \mapsto e * P \} \rightsquigarrow \{ x \mapsto e * Q \} \mid \mathtt{c}}{\Gamma ; \{ x \mapsto - * P \} \rightsquigarrow \{ x \mapsto e * Q \} \mid \mathtt{*x = e;\ c}} \text{(Write)}$$

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

$$\frac{\begin{array}{c}\mathrm{EV}(\Gamma, P, Q) \cap \mathit{Vars}(R) = \varnothing \\ \Gamma ; \{ P \} \leadsto \{ Q \} \mid \mathbf{c}\end{array}}{\Gamma ; \{ P * R \} \leadsto \{ Q * R \} \mid \mathbf{c}} \;(\text{Frame})$$

$$\Gamma; \{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid \mathtt{skip} \qquad \text{(Emp)}$$

$$\frac{a \in GV(\Gamma, P, Q) \qquad y \text{ is fresh} \qquad \Gamma, y; [y/a]\{ x \mapsto y * P \} \rightsquigarrow [y/a]\{ Q \} \mid \mathtt{c}}{\Gamma; \{ x \mapsto a * P \} \rightsquigarrow \{ Q \} \mid \mathtt{let\ y\ =\ *x;\ c}} \ \text{(Read)}$$

$$\frac{EV(\Gamma, P, Q) \cap \mathit{Vars}(R) = \emptyset \qquad \Gamma; \{ P \} \rightsquigarrow \{ Q \} \mid \mathtt{c}}{\Gamma; \{ P * R \} \rightsquigarrow \{ Q * R \} \mid \mathtt{c}} \ \text{(Frame)}$$

$$\frac{\mathit{Vars}(e) \subseteq \Gamma \qquad \Gamma; \{ x \mapsto e * P \} \rightsquigarrow \{ x \mapsto e * Q \} \mid \mathtt{c}}{\Gamma; \{ x \mapsto - * P \} \rightsquigarrow \{ x \mapsto e * Q \} \mid \mathtt{*x\ =\ e;\ c}} \ \text{(Write)}$$

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

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

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

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

$$\frac{\{\, x, y, a2 \,\} \,;\, \{\, x \mapsto a2 * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a2 \,\} \;\big|\; \texttt{??}}{\{\, x, y \,\} \,;\, \{\, x \mapsto a * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a \,\} \;\big|\; \texttt{let a2 = *x; ??}} \;\text{(Read)}$$

$$\dfrac{\dfrac{\{\,x, y, a2, b2\,\}\,;\,\{\,x \mapsto a2 * y \mapsto b2\,\} \rightsquigarrow \{\,x \mapsto b2 * y \mapsto a2\,\} \;\big|\; \texttt{??}}{\{\,x, y, a2\,\}\,;\,\{\,x \mapsto a2 * y \mapsto b\,\} \rightsquigarrow \{\,x \mapsto b * y \mapsto a2\,\} \;\big|\; \texttt{let b2 = *y; ??}}\;\text{(Read)}}{\{\,x, y\,\}\,;\,\{\,x \mapsto a * y \mapsto b\,\} \rightsquigarrow \{\,x \mapsto b * y \mapsto a\,\} \;\big|\; \texttt{let a2 = *x; ??}}\;\text{(Read)}$$

$$\{\, x, y, a2, b2 \,\} \,;\, \{\, x \mapsto b2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{??}$$

(Write)

$$\{\, x, y, a2, b2 \,\} \,;\, \{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{*x = b2; ??}$$

(Read)

$$\{\, x, y, a2 \,\} \,;\, \{\, x \mapsto a2 * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{let b2 = *y; ??}$$

(Read)

$$\{\, x, y \,\} \,;\, \{\, x \mapsto a * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a \,\} \mid \texttt{let a2 = *x; ??}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{ x, y, a2, b2 \} ; \{ y \mapsto b2 \} \rightsquigarrow \{ y \mapsto a2 \} \mid \texttt{??}}{\{ x, y, a2, b2 \} ; \{ x \mapsto b2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{??}}\ (\text{Frame})}{\{ x, y, a2, b2 \} ; \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{*x = b2; ??}}\ (\text{Write})}{\{ x, y, a2 \} ; \{ x \mapsto a2 * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a2 \} \mid \texttt{let b2 = *y; ??}}\ (\text{Read})}{\{ x, y \} ; \{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \texttt{let a2 = *x; ??}}\ (\text{Read})$$

$$\{ x, y, a2, b2 \} ; \{ y \mapsto a2 \} \leadsto \{ y \mapsto a2 \} \mid \texttt{??}$$

$$\overline{}\ \text{(Write)}$$

$$\{ x, y, a2, b2 \} ; \{ y \mapsto b2 \} \leadsto \{ y \mapsto a2 \} \mid \texttt{*y = a2; ??}$$

$$\overline{}\ \text{(Frame)}$$

$$\{ x, y, a2, b2 \} ; \{ x \mapsto b2 * y \mapsto b2 \} \leadsto \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{??}$$

$$\overline{}\ \text{(Write)}$$

$$\{ x, y, a2, b2 \} ; \{ x \mapsto a2 * y \mapsto b2 \} \leadsto \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{*x = b2; ??}$$

$$\overline{}\ \text{(Read)}$$

$$\{ x, y, a2 \} ; \{ x \mapsto a2 * y \mapsto b \} \leadsto \{ x \mapsto b * y \mapsto a2 \} \mid \texttt{let b2 = *y; ??}$$

$$\overline{}\ \text{(Read)}$$

$$\{ x, y \} ; \{ x \mapsto a * y \mapsto b \} \leadsto \{ x \mapsto b * y \mapsto a \} \mid \texttt{let a2 = *x; ??}$$

$$\{ x, y, a2, b2 \} ; \{ \mathrm{emp} \} \rightsquigarrow \{ \mathrm{emp} \} \mid \texttt{??}$$

(Frame)

$$\{ x, y, a2, b2 \} ; \{ y \mapsto a2 \} \rightsquigarrow \{ y \mapsto a2 \} \mid \texttt{??}$$

(Write)

$$\{ x, y, a2, b2 \} ; \{ y \mapsto b2 \} \rightsquigarrow \{ y \mapsto a2 \} \mid \texttt{*y = a2; ??}$$

(Frame)

$$\{ x, y, a2, b2 \} ; \{ x \mapsto b2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{??}$$

(Write)

$$\{ x, y, a2, b2 \} ; \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{*x = b2; ??}$$

(Read)

$$\{ x, y, a2 \} ; \{ x \mapsto a2 * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a2 \} \mid \texttt{let b2 = *y; ??}$$

(Read)

$$\{ x, y \} ; \{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \texttt{let a2 = *x; ??}$$

(Emp)

$$\{ x, y, a2, b2 \} ; \{ \mathsf{emp} \} \rightsquigarrow \{ \mathsf{emp} \} \mid \texttt{skip}$$

(Frame)

$$\{ x, y, a2, b2 \} ; \{ y \mapsto a2 \} \rightsquigarrow \{ y \mapsto a2 \} \mid \texttt{??}$$

(Write)

$$\{ x, y, a2, b2 \} ; \{ y \mapsto b2 \} \rightsquigarrow \{ y \mapsto a2 \} \mid \boxed{\texttt{*y = a2;}}\ \texttt{??}$$

(Frame)

$$\{ x, y, a2, b2 \} ; \{ x \mapsto b2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto a2 \} \mid \texttt{??}$$

(Write)

$$\{ x, y, a2, b2 \} ; \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto b2 * y \mapsto a2 \} \mid \boxed{\texttt{*x = b2;}}\ \texttt{??}$$

(Read)

$$\{ x, y, a2 \} ; \{ x \mapsto a2 * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a2 \} \mid \boxed{\texttt{let b2 = *y;}}\ \texttt{??}$$

(Read)

$$\{ x, y \} ; \{ x \mapsto a * y \mapsto b \} \rightsquigarrow \{ x \mapsto b * y \mapsto a \} \mid \boxed{\texttt{let a2 = *x;}}\ \texttt{??}$$

```
void swap(loc x, loc y) {
    let a2 = *x;
    let b2 = *y;
    *x = b2;
    *y = a2;
}
```

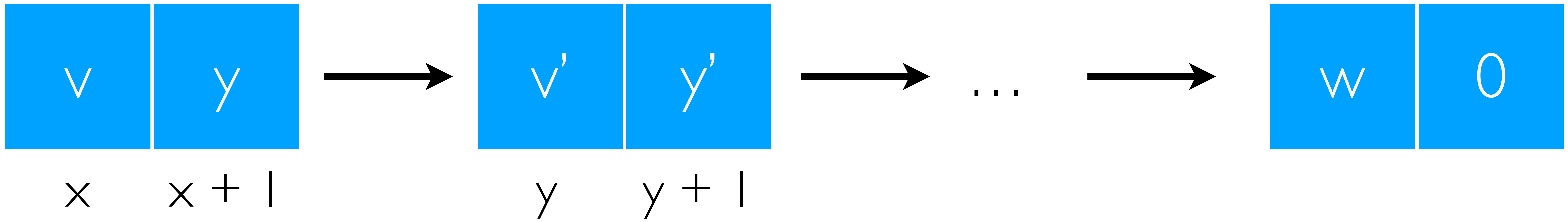
Constraints on Data

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

$$\Gamma \,;\, \{\phi; P\} \rightsquigarrow \{\psi; Q\} \mid c$$

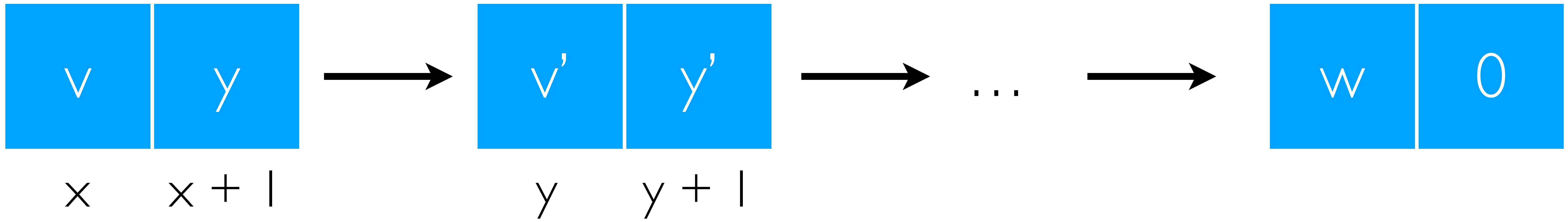
$$\{ a > 5 ; x \mapsto a \} \rightsquigarrow \{ b > a ; x \mapsto b \}$$

Inductive Predicates and Recursion



v
y
v'
y'
...
w
0
x
x + 1
y
y + 1

```
predicate sll (loc x, set s) {
  | x = 0  ∧ { s = ∅          ;  emp }
  | x ≠ 0 ∧  { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```

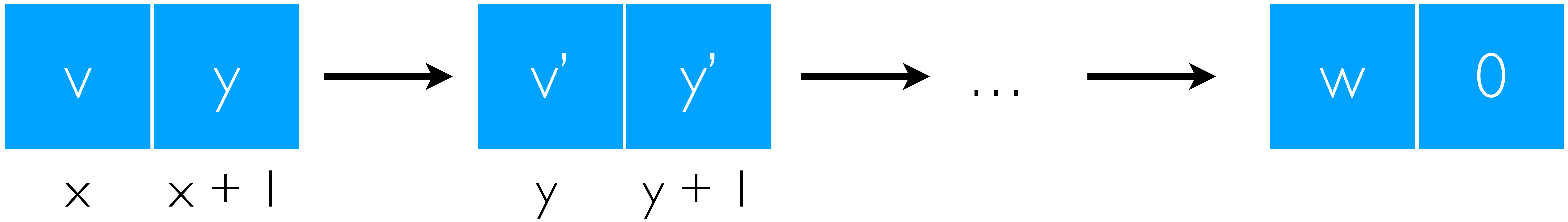


predicate sll (**loc** x, **set** s) {

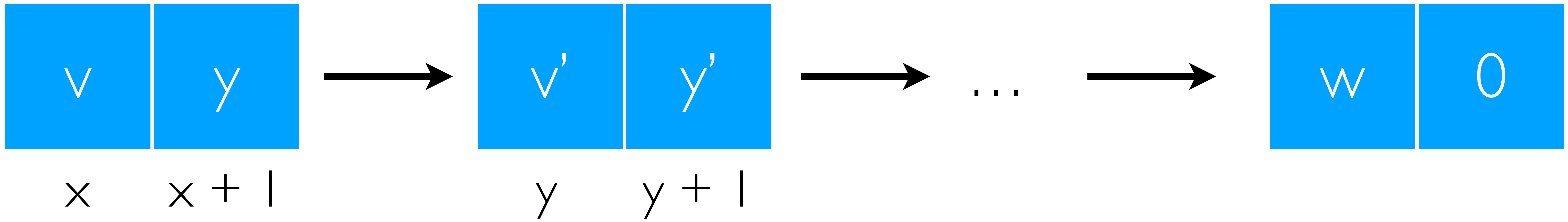
| $x = 0$ ∧ { $s = \emptyset$; emp }

| $x \neq 0$ ∧ { $s = \{v\} \cup s'$; [x, 2] $*$ $x \mapsto v$ $*$ $(x + 1) \mapsto y$ $*$ sll(y, s') }

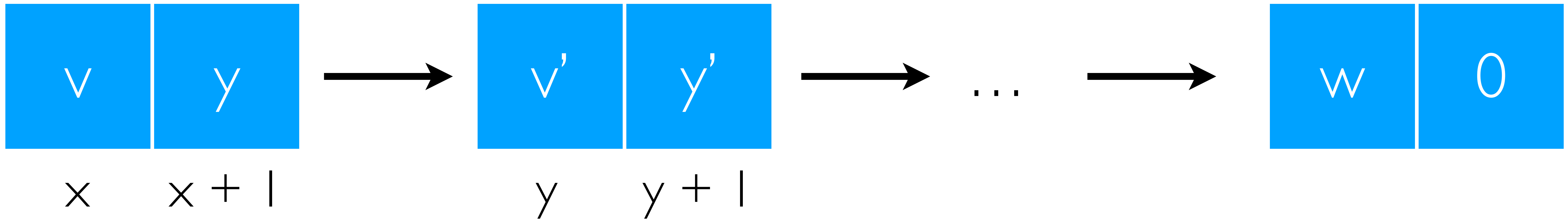
}



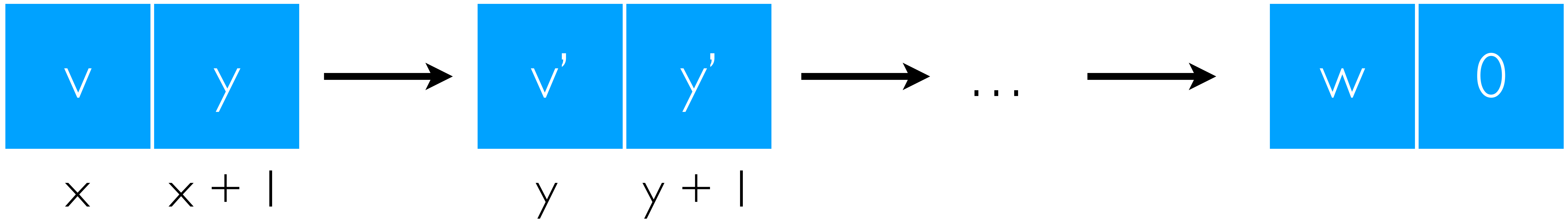
```
predicate sll (loc x, set s) {
  | x = 0  ∧  { s = ∅           ;  emp }
  | x ≠ 0 ∧   { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```



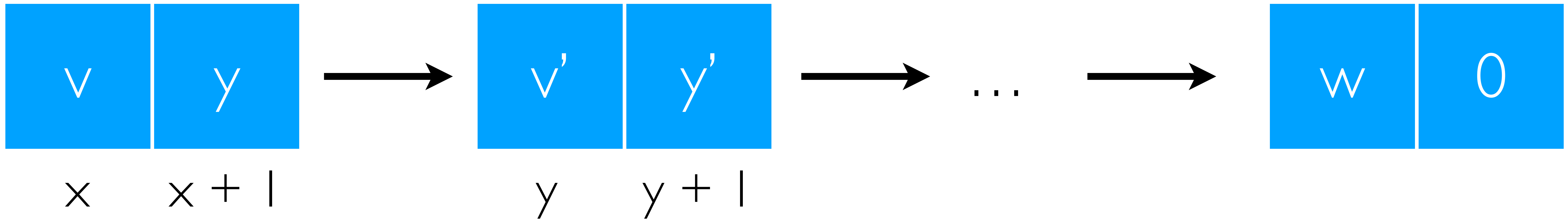
```
predicate sll (loc x, set s) {
  | x = 0  ∧  { s = ∅            ;  emp }
  | x ≠ 0 ∧  { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```



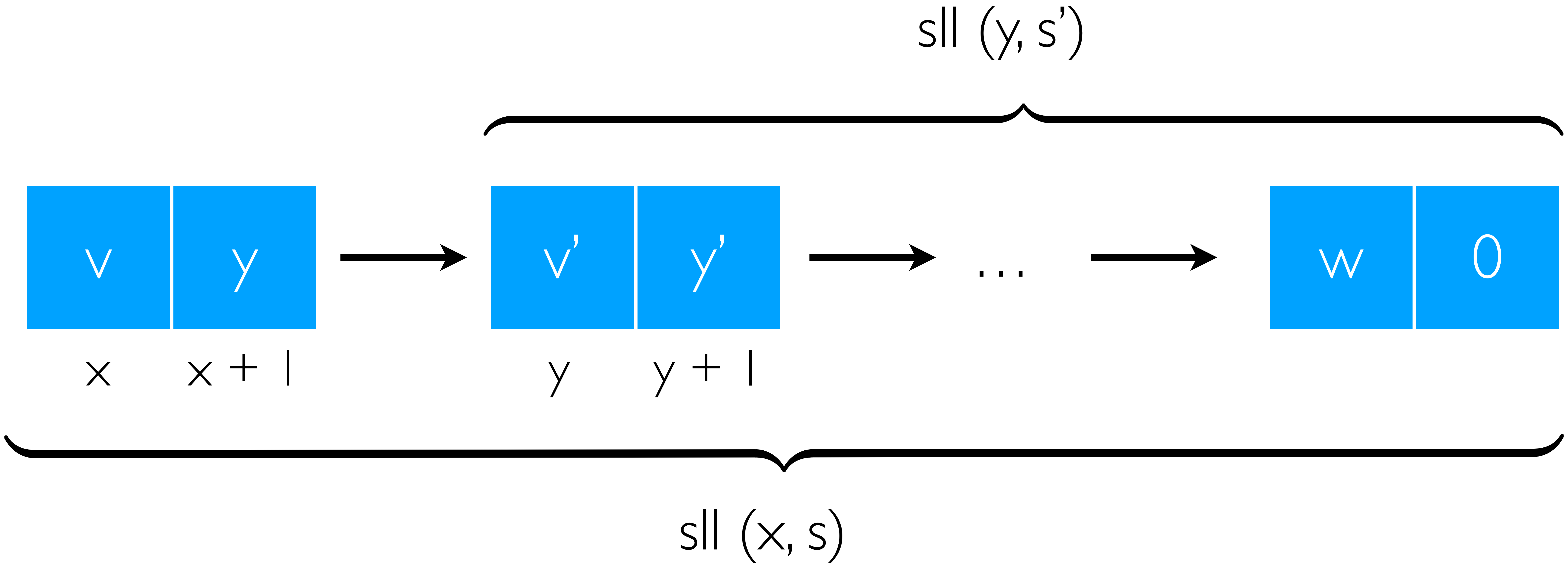
```
predicate sll (loc x, set s) {
  | x = 0  ∧  { s = ∅          ;  emp }
  | x ≠ 0 ∧   { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```



```
predicate sll (loc x, set s) {
  | x = 0  ∧  { s = ∅          ;  emp }
  | x ≠ 0 ∧   { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```



$$
\begin{aligned}
&\textbf{predicate}\ \mathrm{sll}\ (\textbf{loc}\ x,\ \textbf{set}\ s)\ \{\\
&\quad \mid\ x = 0\ \wedge\ \{\ s = \emptyset\ ;\ \mathrm{emp}\ \}\\
&\quad \mid\ x \neq 0\ \wedge\ \{\ s = \{v\} \cup s'\ ;\ [x, 2] * x \mapsto v * (x + 1) \mapsto y * \mathrm{sll}(y, s')\ \}\\
&\}
\end{aligned}
$$



```
predicate sll (loc x, set s) {
  | x = 0  ∧  { s = ∅          ;  emp }
  | x ≠ 0 ∧  { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```

{ sll (x, s) }

```
void listfree(loc x)
```

{ emp }

predicate sll (loc x, set s) {
  | $x = 0 \wedge$ { $s = \varnothing$; emp }
  | $x \neq 0 \wedge$ { $s = \{v\} \cup s'$; $[x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s')$ }
}

{ $sll^{1}(x, s)$ } `void listfree(loc x)` { emp }

$$\{ sll^{0}(x, s) \}$$

??

$$\{ emp \}$$

predicate sll (loc x, set s) {
| $x = 0$ ∧ { $s = \emptyset$; emp }
| $x \neq 0$ ∧ { $s = \{v\} \cup s'$; $[x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s')$ }
}

{ $sll^{1}(x, s)$ } `void listfree(loc x)` { emp }

{ $sll^{0}(x, s)$ }

??

{ emp }

predicate sll (loc x, set s) {
  | $x = 0 \wedge$ { $s = \varnothing$; emp }
  | $x \neq 0 \wedge$ { $s = \{v\} \cup s'$; $[x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s')$ }
}

{ $sll^{1}(x, s)$ } `void listfree(loc x)` { emp }

```
if (x == 0) {
```

{ $x = 0$; $sll^{0}(x, s)$ }

```
    ??
```

{ emp }

```
} else {
```

{ $x \neq 0$; $sll^{0}(x, s)$ }

```
    ??
```

{ emp }

```
}
```

```
predicate sll (loc x, set s) {
  | x = 0  ∧ { s = ∅          ; emp }
  | x ≠ 0 ∧ { s = {v} ∪ s'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ y ∗ sll(y, s') }
}
```

$\{ sll^1(x, s) \}$ `void listfree(loc x)` $\{ emp \}$

```
if (x == 0) {
```

$\{ x = 0 \wedge s = \emptyset ;\ emp \}$

```
    ??
```

$\{ emp \}$

```
} else {
```

$\{ x \neq 0 \wedge s = \{v\} \cup s' ;\ [x, 2] * x \mapsto v * (x + 1) \mapsto y * sll^1(y, s') \}$

```
    ??
```

$\{ emp \}$

```
}
```

predicate sll (loc x, set s) {
| $x = 0 \wedge \{ s = \varnothing \; ; \; \mathrm{emp} \}$
| $x \neq 0 \wedge \{ s = \{v\} \cup s' \; ; [x, 2] * x \mapsto v * (x + 1) \mapsto y * \mathrm{sll}(y, s') \}$
}

$\{ \mathrm{sll}^1 (x, s) \}$ `void listfree(loc x)` $\{ \mathrm{emp} \}$

```
if (x == 0) {
```

$\{ x = 0 \wedge s = \varnothing \; ; \; \mathrm{emp} \}$

```
    skip
```

$\{ \mathrm{emp} \}$

```
} else {
```

$\{ x \neq 0 \wedge s = \{v\} \cup s' \; ; [x, 2] * x \mapsto v * (x + 1) \mapsto y * \mathrm{sll}^1 (y, s') \}$

```
    ??
```

$\{ \mathrm{emp} \}$

```
}
```

predicate sll (loc x, set s) {
  | $x = 0 \wedge \{ s = \varnothing \quad ; \text{emp} \}$
  | $x \neq 0 \wedge \{ s = \{v\} \cup s' \quad ; [x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s') \}$
}

$\{ sll^1 (x, s) \}$ `void listfree(loc x)` $\{ \text{emp} \}$

```
if (x == 0) { } else {
```

$\{ x \neq 0 \wedge s = \{v\} \cup s' \quad ; [x, 2] * x \mapsto v * (x + 1) \mapsto y * sll^1 (y, s') \}$

```
    ??
```

$\{ \text{emp} \}$

```
}
```

predicate sll (loc x, set s) {
  | $x = 0 \wedge \{ s = \varnothing \; ; \; emp \}$
  | $x \neq 0 \wedge \{ s = \{v\} \cup s' \; ; [x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s') \}$
}

$\{ sll^1 (x, s) \}$ `void listfree(loc x)` $\{ emp \}$

```
if (x == 0) { } else {

    let nxt2 = *(x + 1);
```

$\{ x \neq 0 \wedge s = \{v\} \cup s' \; ; [x, 2] * x \mapsto v * (x + 1) \mapsto nxt2 * sll^1 (nxt2, s') \}$

```
    ??
```

$\{ emp \}$

```
}
```

predicate sll (loc x, set s) {
| $x = 0 \ \wedge \ \{ s = \varnothing \ ; \ emp \}$
| $x \neq 0 \wedge \{ s = \{v\} \cup s' \ ; [x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s') \}$
}

$\{ sll^{1} (x, s) \}$ void listfree(loc x) $\{ emp \}$

```
if (x == 0) { } else {
    let nxt2 = *(x + 1);
    free(x);
```

$\{ x \neq 0 \wedge s = \{v\} \cup s' \ ; sll^{1} (nxt2, s') \}$

```
    ??
```

$\{ emp \}$

```
}
```

predicate sll (loc x, set s) {
  | $x = 0 \land \{ s = \varnothing$; emp }
  | $x \neq 0 \land \{ s = \{v\} \cup s'$; $[x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s')$ }
}

{ $sll^{1}(x, s)$ } `void listfree(loc x)` { emp }

```
if (x == 0) { } else {

   let nxt2 = *(x + 1);

   free(x);

   listfree(nxt2);
```

{ $x \neq 0 \land s = \{v\} \cup s'$; emp }

```
   ??
```

{ emp }

```
}
```

predicate sll (loc x, set s) {
  | $x = 0 \;\wedge\; \{ s = \varnothing \;;\; emp \}$
  | $x \neq 0 \wedge \{ s = \{v\} \cup s' \;; [x, 2] * x \mapsto v * (x + 1) \mapsto y * sll(y, s') \}$
}

$\{ sll^{1} (x, s) \}$ `void listfree(loc x)` $\{ emp \}$

```
if (x == 0) { } else {

    let nxt2 = *(x + 1);

    free(x);

    listfree(nxt2);

    skip;

}
```

```
void listfree(loc x) {
  if (x == 0) { } else {
     let nxt2 = *(x + 1);
     free(x);
     listfree(nxt2);
  }
}
```

Rules of the Logic

StarPartial

$$\frac{\begin{array}{c} x + \iota \neq y + \iota' \notin \phi \qquad \phi' \triangleq \phi \wedge (x + \iota \neq y + \iota') \\ \Sigma; \Gamma;\ \{\phi'; \langle x, \iota\rangle \mapsto e * \langle y, \iota'\rangle \mapsto e' * P\} \leadsto \{Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; \langle x, \iota\rangle \mapsto e * \langle y, \iota'\rangle \mapsto e' * P\} \leadsto \{Q\} \,|\, c}$$

Open

$$\frac{\begin{array}{c} \mathcal{D} \triangleq p(\overline{x_i}) \overline{\langle \xi_j, \{\chi_j, R_j\}\rangle}_{j \in 1 \ldots N} \in \Sigma \\ \ell < \mathsf{MaxUnfold} \quad \sigma \triangleq [\overline{x_i \mapsto y_i}] \quad \mathsf{Vars}(\overline{y_i}) \subseteq \Gamma \\ \phi_j \triangleq \phi \wedge [\sigma]\xi_j \wedge [\sigma]\chi_j \qquad P_j \triangleq \lceil [\sigma]R_j \rceil^{\ell+1} * \lceil P \rceil \\ \forall j \in 1 \ldots N, \quad \Sigma; \Gamma;\ \{\phi_j; P_j\} \leadsto \{Q\} \,|\, c_j \\ c \triangleq \textbf{if}\ ([\sigma]\xi_1)\ \{c_1\}\ \textbf{else}\ \{\textbf{if}\ ([\sigma]\xi_2) \ldots \textbf{else}\ \{c_N\}\} \end{array}}{\Sigma; \Gamma;\ \{\phi; P * p^{\ell}(\overline{y_i})\} \leadsto \{Q\} \,|\, c}$$

AbduceCall

$$\frac{\begin{array}{c} \mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f * F_f\}\{\psi_f; Q_f\} \in \Sigma \\ F_f \text{ has no predicate instances} \qquad [\sigma]P_f = P \\ F_f \neq \mathsf{emp} \qquad F' \triangleq [\sigma]F_f \qquad \Sigma; \Gamma;\ \{\phi; F\} \leadsto \{\phi; F'\} \,|\, c_1 \\ \Sigma; \Gamma;\ \{\phi; P * F' * R\} \leadsto \{Q\} \,|\, c_2 \end{array}}{\Sigma; \Gamma;\ \{\phi; P * F * R\} \leadsto \{Q\} \,|\, c_1; c_2}$$

Read

$$\frac{\begin{array}{c} a \in \mathsf{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) \qquad y \notin \mathsf{Vars}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \Gamma \cup \{y\};\ [y/a]\{\phi; \langle x, \iota\rangle \mapsto a * P\} \leadsto [y/a]\{Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; \langle x, \iota\rangle \mapsto a * P\} \leadsto \{Q\} \,|\, \textbf{let}\ y = *(x + \iota); c}$$

Close

$$\frac{\begin{array}{c} \mathcal{D} \triangleq p(\overline{x_i}) \overline{\langle \xi_j, \{\chi_j, R_j\}\rangle}_{j \in 1 \ldots N} \in \Sigma \\ \ell < \mathsf{MaxUnfold} \qquad \sigma \triangleq [\overline{x_i \mapsto y_i}] \\ \text{for some } k,\ 1 \leq k \leq N \qquad R' \triangleq \lceil [\sigma]R_k \rceil^{\ell+1} \\ \Sigma; \Gamma;\ \{\mathcal{P}\} \leadsto \{\psi \wedge [\sigma]\xi_k \wedge [\sigma]\chi_k; Q * R'\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\mathcal{P}\} \leadsto \{\psi; Q * p^{\ell}(\overline{y_i})\} \,|\, c}$$

Call

$$\frac{\begin{array}{c} \mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f\}\{\psi_f; Q_f\} \in \Sigma \\ R =^{\ell} [\sigma]P_f \qquad \phi \Rightarrow [\sigma]\phi_f \\ \phi' \triangleq [\sigma]\psi_f \qquad R' \triangleq \lceil [\sigma]Q_f \rceil \qquad \overline{e_i} = [\sigma]\overline{x_i} \\ \mathsf{Vars}(\overline{e_i}) \subseteq \Gamma \qquad \Sigma; \Gamma;\ \{\phi \wedge \phi'; P * R'\} \leadsto \{Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; P * R\} \leadsto \{Q\} \,|\, f(\overline{e_i}); c}$$

Alloc

$$\frac{\begin{array}{c} R = [z, n] * \circledast_{0 \leq i \leq n} (\langle z, i\rangle \mapsto e_i) \qquad z \in \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ (\{y\} \cup \{\overline{t_i}\}) \cap \mathsf{Vars}(\Gamma, \mathcal{P}, \mathcal{Q}) = \emptyset \\ R' \triangleq [y, n] * \circledast_{0 \leq i \leq n} (\langle y, i\rangle \mapsto t_i) \\ \Sigma; \Gamma;\ \{\phi; P * R'\} \leadsto \{\psi; Q * R\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; P\} \leadsto \{\psi; Q * R\} \,|\, \textbf{let}\ y = \mathtt{malloc}(n); c}$$

Free

$$\frac{\begin{array}{c} R = [x, n] * \circledast_{0 \leq i \leq n} (\langle x, i\rangle \mapsto e_i) \\ \mathsf{Vars}(\{x\} \cup \{\overline{e_i}\}) \subseteq \Gamma \qquad \Sigma; \Gamma;\ \{\phi; P\} \leadsto \{Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; P * R\} \leadsto \{Q\} \,|\, \mathtt{free}(n); c}$$

Write

$$\frac{\mathsf{Vars}(e) \subseteq \Gamma \qquad \Gamma;\ \{\phi; \langle x, \iota\rangle \mapsto e * P\} \leadsto \{\psi; \langle x, \iota\rangle \mapsto e * Q\} \,|\, c}{\Gamma;\ \{\phi; \langle x, \iota\rangle \mapsto e' * P\} \leadsto \{\psi; \langle x, \iota\rangle \mapsto e * Q\} \,|\, *(x + \iota) = e; c}$$

UnifyHeaps

$$\frac{\begin{array}{c} [\sigma]R' = R \\ \mathsf{frameable}(R') \qquad \emptyset \neq \mathsf{dom}(\sigma) \subseteq \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \Gamma;\ \{P * R\} \leadsto [\sigma]\{\psi; Q * R'\} \,|\, c \end{array}}{\Gamma;\ \{\phi; P * R\} \leadsto \{\psi; Q * R'\} \,|\, c}$$

Frame

$$\frac{\begin{array}{c} \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \cap \mathsf{Vars}(R) = \emptyset \\ \mathsf{frameable}(R') \qquad \Gamma;\ \{\phi; P\} \leadsto \{\psi; Q\} \,|\, c \end{array}}{\Gamma;\ \{\phi; P * R\} \leadsto \{\psi; Q * R\} \,|\, c}$$

Induction

$$\frac{\begin{array}{c} f \triangleq \text{goal's name} \\ \overline{x_i} \triangleq \text{goal's formals} \\ P_f \triangleq p^1(\overline{y_i}) * \lceil P \rceil \qquad Q_f \triangleq \lceil Q \rceil \\ \mathcal{F} \triangleq f(\overline{x_i}) : \{\phi_f; P_f\}\{\psi_f; Q_f\} \\ \Sigma, \mathcal{F}; \Gamma;\ \{\phi; p^0(\overline{y_i}) * P\} \leadsto \{Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; p^0(\overline{y_i}) * P\} \leadsto \{Q\} \,|\, c}$$

Emp

$$\frac{\mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) = \emptyset \qquad \phi \Rightarrow \psi}{\Gamma;\ \{\phi; \mathsf{emp}\} \leadsto \{\psi; \mathsf{emp}\} \,|\, \mathtt{skip}}$$

Inconsistency

$$\frac{\phi \Rightarrow \bot}{\Gamma;\ \{\phi; P\} \leadsto \{Q\} \,|\, \mathtt{error}}$$

NullNotLVal

$$\frac{\begin{array}{c} x \neq 0 \notin \phi \qquad \phi' \triangleq \phi \wedge x \neq 0 \\ \Sigma; \Gamma;\ \{\phi'; \langle x, \iota\rangle \mapsto e * P\} \leadsto \{Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\phi; \langle x, \iota\rangle \mapsto e * P\} \leadsto \{Q\} \,|\, c}$$

SubstLeft

$$\frac{\begin{array}{c} \phi \Rightarrow x = y \\ \Gamma;\ [y/x]\{\phi; P\} \leadsto [y/x]\{Q\} \,|\, c \end{array}}{\Gamma;\ \{\phi; P\} \leadsto \{Q\} \,|\, c}$$

Pick

$$\frac{\begin{array}{c} y \in \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \mathsf{Vars}(e) \in \Gamma \cup \mathsf{GV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \Gamma;\ \{\phi; P\} \leadsto [e/y]\{\psi; Q\} \,|\, c \end{array}}{\Gamma;\ \{\phi; P\} \leadsto \{\psi; Q\} \,|\, c}$$

UnifyPure

$$\frac{\begin{array}{c} [\sigma]\psi' = \phi' \\ \emptyset \neq \mathsf{dom}(\sigma) \subseteq \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \Gamma;\ \{\mathcal{P}\} \leadsto [\sigma]\{\mathcal{Q}\} \,|\, c \end{array}}{\Gamma;\ \{\phi \wedge \phi'; P\} \leadsto \{\psi \wedge \psi'; Q\} \,|\, c}$$

SubstRight

$$\frac{\begin{array}{c} x \in \mathsf{EV}(\Gamma, \mathcal{P}, \mathcal{Q}) \\ \Sigma; \Gamma;\ \{\mathcal{P}\} \leadsto [e/x]\{\psi, Q\} \,|\, c \end{array}}{\Sigma; \Gamma;\ \{\mathcal{P}\} \leadsto \{\psi \wedge x = e; Q\} \,|\, c}$$

Synthesis Algorithm

Proof Search Algorithm

- Goal-driven, trying a fixed set of rules to build the program;
- *Branching* and *backtracking*: some rules emit many alternatives;
- Along with the program, emits the *complete proof tree*;
- Terminates assuming finite number of *predicate unfoldings.*

Implementation

SuSLik

(**S**ynthesis **u**sing **S**eparation **L**og**ik**)

Demo

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Integers	1	swap two	1	4	1.0x	0.2
	2	min of two[1]	1	3	1.1x	0.8
Singly Linked List	3	length[2]	1	6	1.4x	0.4
	4	max[2]	1	11	1.9x	3.0
	5	min[2]	1	11	1.9x	2.9
	6	singleton[1]	1	4	0.9x	0.2
	7	deallocate	1	4	5.5x	0.2
	8	initialize	1	4	1.6x	0.4
	9	copy[3]	1	11	2.7x	0.6
	10	append[3]	1	6	1.1x	0.4
	11	delete[3]	1	12	2.6x	1.2
	12	deallocate two	2	9	6.2x	0.2
	13	append three	2	14	2.3x	1.0
	14	non-destructive append	2	21	3.0x	8.0
	15	union	2	23	5.5x	4.3
	16	intersection[4]	3	32	7.0x	101.1
	17	difference[4]	2	21	5.1x	4.7
	18	deduplicate[4]	2	22	7.3x	1.8
Sorted list	19	prepend[2]	1	4	0.4x	0.2
	20	insert[2]	1	19	3.1x	1.0
	21	insertion sort[2]	1	7	1.2x	0.7
	22	sort[4]	2	13	4.9x	1.0
	23	reverse[4]	2	11	4.0x	0.7
	24	merge[2]	2	30	4.4x	55.6
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2
	26	copy	1	22	4.3x	7.2
	27	append[3]	1	10	1.6x	1.7
	28	delete[3]	1	19	3.7x	3.4
	29	single to double	1	23	6.0x	0.7

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2
	26	copy	1	22	4.3x	7.2
	27	append[3]	1	10	1.6x	1.7
	28	delete[3]	1	19	3.7x	3.4
	29	single to double	1	23	6.0x	0.7

List of Lists	30	deallocate	2	11	10.7x	0.2
	31	flatten[4]	2	17	4.4x	0.6
	32	length[5]	2	21	5.5x	22.8
Binary Tree	33	size	1	9	2.5x	0.4
	34	deallocate	1	6	8.0x	0.2
	35	deallocate two	1	16	11.8x	0.4
	36	copy	1	16	3.8x	2.5
	37	flatten w/append	1	17	4.8x	0.4
	38	flatten w/acc	1	12	2.1x	0.6
	39	flatten	2	23	7.1x	1.5
	40	flatten to dll in place	2	15	9.6x	11.3
	41	flatten to dll w/null[5]	2	17	11.2x	106.1
BST	42	insert[2]	1	19	2.8x	14.6
	43	rotate left[2]	1	5	0.2x	6.2
	44	rotate right[2]	1	5	0.2x	4.9
	45	find min[5]	1	11	1.4x	66.3
	46	find max[5]	1	18	2.2x	58.0
	47	delete root[2]	1	18	1.3x	13.9
	48	from list[4]	2	27	5.7x	10.0
	49	to sorted list[4]	3	32	7.7x	20.8
Rose Tree	50	deallocate	2	9	12.0x	0.2
	51	flatten	3	25	8.0x	11.0
	52	copy[5]	2	32	7.9x	-
Packed Tree	53	pack[5]	1	16	1.6x	-
	54	unpack[5]	1	23	2.9x	21.0

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Integers	1	swap two	1	4	1.0x	0.2
	2	min of two[1]	1	3	1.1x	0.8
Singly Linked List	3	length[2]	1	6	1.4x	0.4
	4	max[2]	1	11	1.9x	3.0
	5	min[2]	1	11	1.9x	2.9
	6	singleton[1]	1	4	0.9x	0.2
	7	deallocate	1	4	5.5x	0.2
	8	initialize	1	4	1.6x	0.4
	9	copy[3]	1	11	2.7x	0.6
	10	append[3]	1	6	1.1x	0.4
	11	delete[3]	1	12	2.6x	1.2
	12	deallocate two	2	9	6.2x	0.2
	13	append three	2	14	2.3x	1.0
	14	non-destructive append	2	21	3.0x	8.0
	15	union	2	23	5.5x	4.3
	16	intersection[4]	3	32	7.0x	101.1
	17	difference[4]	2	21	5.1x	4.7
	18	deduplicate[4]	2	22	7.3x	1.8
Sorted list	19	prepend[2]	1	4	0.4x	0.2
	20	insert[2]	1	19	3.1x	1.0
	21	insertion sort[2]	1	7	1.2x	0.7
	22	sort[4]	2	13	4.9x	1.0
	23	reverse[4]	2	11	4.0x	0.7
	24	merge[2]	2	30	4.4x	55.6
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2
	26	copy	1	22	4.3x	7.2
	27	append[3]	1	10	1.6x	1.7
	28	delete[3]	1	19	3.7x	3.4
	29	single to double	1	23	6.0x	0.7

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2
	26	copy	1	22	4.3x	7.2
	27	append[3]	1	10	1.6x	1.7
	28	delete[3]	1	19	3.7x	3.4
	29	single to double	1	23	6.0x	0.7

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
List of Lists	30	deallocate	2	11	10.7x	0.2
	31	flatten[4]	2	17	4.4x	0.6
	32	length[5]	2	21	5.5x	22.8
Binary Tree	33	size	1	9	2.5x	0.4
	34	deallocate	1	6	8.0x	0.2
	35	deallocate two	1	16	11.8x	0.4
	36	copy	1	16	3.8x	2.5
	37	flatten w/append	1	17	4.8x	0.4
	38	flatten w/acc	1	12	2.1x	0.6
	39	flatten	2	23	7.1x	1.5
	40	flatten to dll in place	2	15	9.6x	11.3
	41	flatten to dll w/null[5]	2	17	11.2x	106.1
BST	42	insert[2]	1	19	2.8x	14.6
	43	rotate left[2]	1	5	0.2x	6.2
	44	rotate right[2]	1	5	0.2x	4.9
	45	find min[5]	1	11	1.4x	66.3
	46	find max[5]	1	18	2.2x	58.0
	47	delete root[2]	1	18	1.3x	13.9
	48	from list[4]	2	27	5.7x	10.0
	49	to sorted list[4]	3	32	7.7x	20.8
Rose Tree	50	deallocate	2	9	12.0x	0.2
	51	flatten	3	25	8.0x	11.0
	52	copy[5]	2	32	7.9x	-
Packed Tree	53	pack[5]	1	16	1.6x	-
	54	unpack[5]	1	23	2.9x	21.0

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Integers	1	swap two	1	4	1.0x	0.2
	2	min of two[1]	1	3	1.1x	0.8
Singly Linked List	3	length[2]	1	6	1.4x	0.4
	4	max[2]	1	11	1.9x	3.0
	5	min[2]	1	11	1.9x	2.9
	6	singleton[1]	1	4	0.9x	0.2
	7	deallocate	1	4	5.5x	0.2
	8	initialize	1	4	1.6x	0.4
	9	copy[3]	1	11	2.7x	0.6
	10	append[3]	1	6	1.1x	0.4
	11	delete[3]	1	12	2.6x	1.2
	12	deallocate two	2	9	6.2x	0.2
	13	append three	2	14	2.3x	1.0
	14	non-destructive append	2	21	3.0x	8.0
	15	union	2	23	5.5x	4.3
	16	intersection[4]	3	32	7.0x	101.1
	17	difference[4]	2	21	5.1x	4.7
	18	deduplicate[4]	2	22	7.3x	1.8
Sorted list	19	prepend[2]	1	4	0.4x	0.2
	20	insert[2]	1	19	3.1x	1.0
	21	insertion sort[2]	1	7	1.2x	0.7
	22	sort[4]	2	13	4.9x	1.0
	23	reverse[4]	2	11	4.0x	0.7
	24	merge[2]	2	30	4.4x	55.6
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2
	26	copy	1	22	4.3x	7.2
	27	append[3]	1	10	1.6x	1.7
	28	delete[3]	1	19	3.7x	3.4
	29	single to double	1	23	6.0x	0.7

Data Structure	*Id*	*Description*	*Proc*	*Stmt*	*Code/Spec*	*Time*
Doubly Linked List	25	singleton[1]	1	5	1.1x	0.2
	26	copy	1	22	4.3x	7.2
	27	append[3]	1	10	1.6x	1.7
	28	delete[3]	1	19	3.7x	3.4
	29	single to double	1	23	6.0x	0.7
List of Lists	30	deallocate	2	11	10.7x	0.2
	31	flatten[4]	2	17	4.4x	0.6
	32	length[5]	2	21	5.5x	22.8
Binary Tree	33	size	1	9	2.5x	0.4
	34	deallocate	1	6	8.0x	0.2
	35	deallocate two	1	16	11.8x	0.4
	36	copy	1	16	3.8x	2.5
	37	flatten w/append	1	17	4.8x	0.4
	38	flatten w/acc	1	12	2.1x	0.6
	39	flatten	2	23	7.1x	1.5
	40	flatten to dll in place	2	15	9.6x	11.3
	41	flatten to dll w/null[5]	2	17	11.2x	106.1
BST	42	insert[2]	1	19	2.8x	14.6
	43	rotate left[2]	1	5	0.2x	6.2
	44	rotate right[2]	1	5	0.2x	4.9
	45	find min[5]	1	11	1.4x	66.3
	46	find max[5]	1	18	2.2x	58.0
	47	delete root[2]	1	18	1.3x	13.9
	48	from list[4]	2	27	5.7x	10.0
	49	to sorted list[4]	3	32	7.7x	20.8
Rose Tree	50	deallocate	2	9	12.0x	0.2
	51	flatten	3	25	8.0x	11.0
	52	copy[5]	2	32	7.9x	-
Packed Tree	53	pack[5]	1	16	1.6x	-
	54	unpack[5]	1	23	2.9x	21.0

Deductive Program Synthesis: Summary

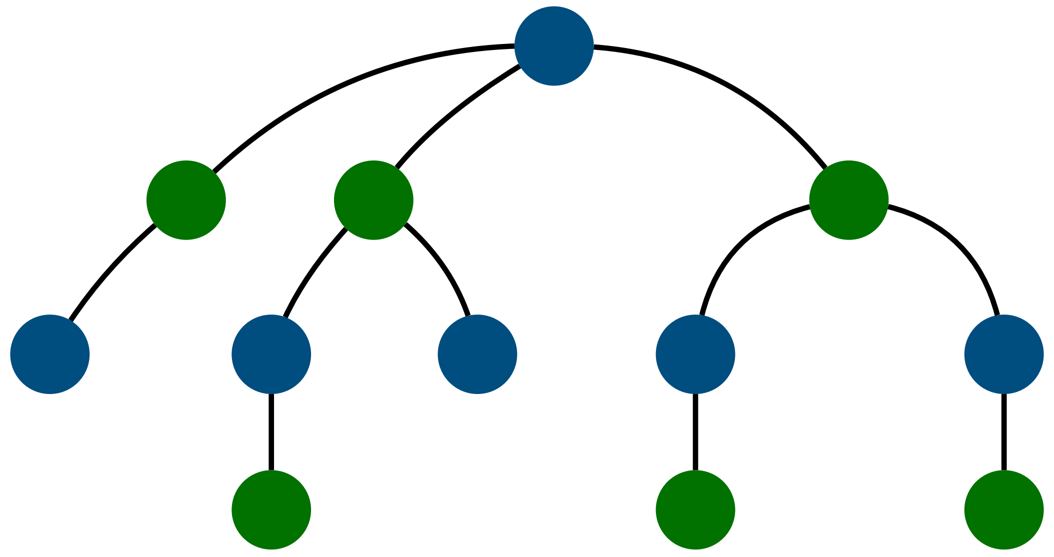
Initial specification

$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$

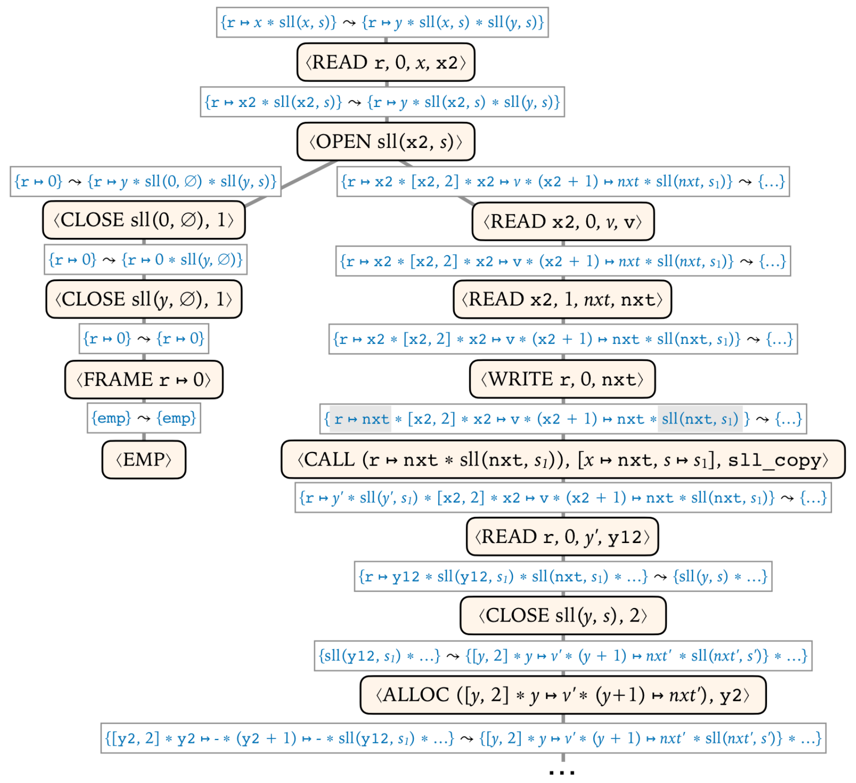
```
void sll_copy(loc r)
```

$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Proof search



Proof tree



Program (byproduct)

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

Deductive Program Synthesis: Summary

POPL'19

Structuring the Synthesis of Heap-Manipulating Programs

NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore

Initi

$\{$r

void

$\{$r $\mapsto y$

ct)

SuSLik

A **deductive synthesizer**
that uses inference rules of
Synthetic Separation Logic (SSL)
to generate imperative,
heap-manipulating programs

```
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

Today's Agenda

- Deductive synthesis in a nutshell
- Trust in program synthesis
- Extensions and Applications

Today's Agenda

- Deductive synthesis in a nutshell
- Trust in program synthesis
- Extensions and Applications

correct theory ≠
correct implementation

$$\{r \mapsto x * \mathsf{sll}(x, S)\}$$

```
void sll_copy(loc r)
```

$$\{r \mapsto y * \mathsf{sll}(x, S) * \mathsf{sll}(y, S)\}$$

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

$$\{r \mapsto x * \mathsf{sll}(x, S)\} \leadsto \{r \mapsto y * \mathsf{sll}(x, S) * \mathsf{sll}(y, S)\}$$

What's wrong?

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

$\{r \mapsto x * \mathsf{sll}(x, S)\} \rightsquigarrow \{\mathbf{r} \mapsto y * \mathsf{sll}(x, S) * \mathsf{sll}(y, S)\}$

There's a bug.

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *r = y2;
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

How can we trust what SuSLik gives us?

SuSLik codebase: too large to verify

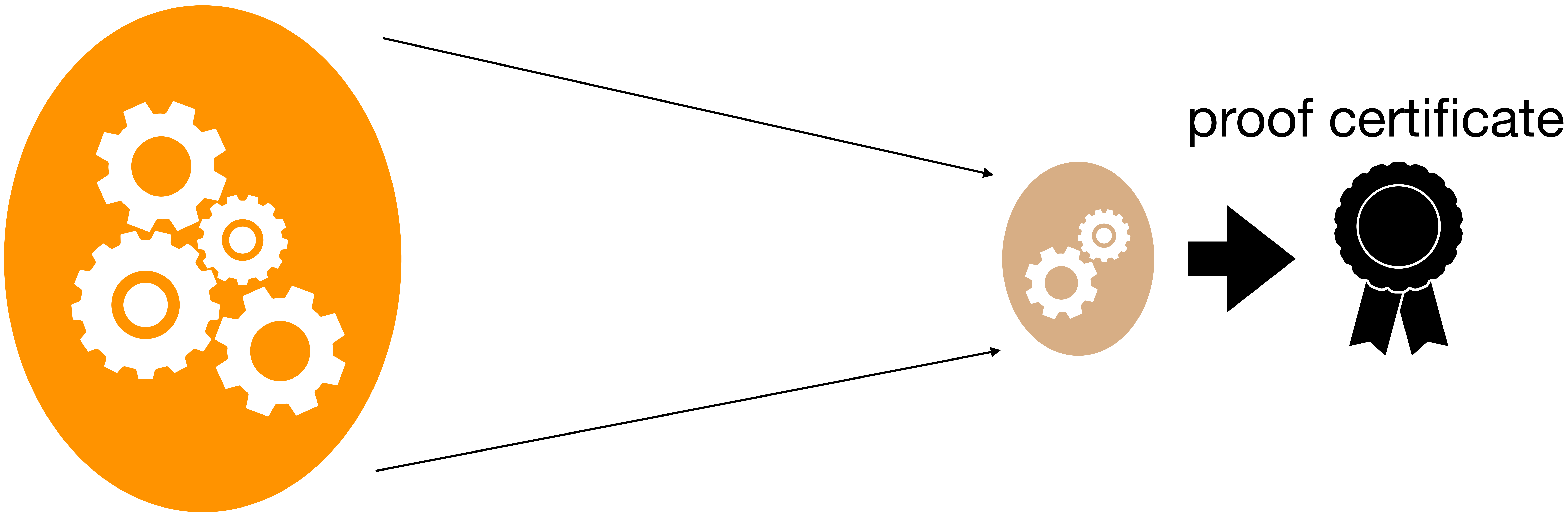
```
protected def synthesize(goal: Goal)
                        (stats: SynStats): Option[Solution] = {
  init(goal)
  processWorkList(stats, goal.env.config)
}

@tailrec final def processWorkList(implicit
                                   stats: SynStats,
                                   config: SynConfig): Option[Solution] = {
  // Check for timeouts
  if (!config.interactive && stats.timedOut) {
    throw SynTimeOutException(s"\n\nThe derivation took too long: more than ${config.timeOut} seconds.\n")
  }

  val sz = worklist.length
  log.print(s"Worklist ($sz): ${worklist.map(n => s"${n.pp()}[${n.cost}]").mkString(" ")}", Console.YELLOW
  log.print(s"Succeeded leaves (${successLeaves.length}): ${successLeaves.map(n => s"${n.pp()}").mkStri
  log.print(s"Memo (${memo.size}) Suspended (${memo.suspendedSize})", Console.YELLOW, 2)
  stats.updateMaxWLSize(sz)

  if (worklist.isEmpty) None // No more goals to try: synthesis failed
  else {
    val (node, addNewNodes) = popNode // Select next node to expand
    val goal = node.goal
    implicit val ctx: log.Context = log.Context(goal)
    stats.addExpandedGoal(node)
    log.print(s"Expand: ${node.pp()}[${node.cost}]", Console.YELLOW) //      <goal: ${node.goal.label.pp}>
    log.print(s"${goal.pp}", Console.BLUE)
    trace.add(node)

    // Lookup the node in the memo
    val res = memo.lookup(goal) match {
      case Some(Failed) => { // Same goal has failed before: record as failed
        log.print("Recalled FAIL", Console.RED)
        trace.add(node, Failed, Some("cache"))
        node.fail
        None
      }
      case Some(Succeeded(sol, id)) =>
      { // Same goal has succeeded before: return the same solution
        log.print(s"Recalled solution ${sol._1.pp}", Console.RED)
```

```
object OperationalRules extends SepLogicUtils with RuleUtils {

  val exceptionQualifier: String = "rule-operational"

  import Statements._

  /*
  Write rule: create a new write from where it's possible

  Γ ; {φ ; x.f -> l' * P} ; {ψ ; x.f -> l' * Q} ---> S   GV(l) = GV(l') = Ø
  ------------------------------------------------------------------------ [write]
  Γ ; {φ ; x.f -> l * P} ; {ψ ; x.f -> l' * Q} ---> *x.f := l' ; S

  */
  object WriteRule extends SynthesisRule with GeneratesCode with InvertibleRule {

    ide def toString: Ident = "Write"

    y(goal: Goal): Seq[RuleResult] = {
      = goal.pre
       = goal.post

      ets have no ghosts
      hosts: Heaplet => Boolean = {
      e PointsTo(x@Var(_), _, e) => !goal.isGhost(x) && e.vars.forall(v => !goal.isGhost(v))
      case _ => false
    }

    // When do two heaplets match
    def isMatch(hl: Heaplet, hr: Heaplet) = sameLhs(hl)(hr) && !sameRhs(hl)(hr) && noGhosts(hr)

    findMatchingHeaplets(_ => true, isMatch, goal.pre.sigma, goal.post.sigma) match {
      case None => Nil
      case Some((hl@PointsTo(x@Var(_), offset, e1), hr@PointsTo(_, _, e2))) =>
        val newPre = Assertion(pre.phi, goal.pre.sigma - hl)
        val newPost = Assertion(post.phi, goal.post.sigma - hr)
        val subGoal = goal.spawnChild(newPre, newPost)
        val kont: StmtProducer = PrependProducer(Store(x, offset, e2)) >> ExtractHelper(goal)

        List(RuleResult(List(subGoal), kont, this, goal))
      case Some((hl, hr)) =>
        ruleAssert(assertion = false, s"Write rule matched unexpected heaplets ${hl.pp} and ${hr.pp}")
        Nil
```

Meet the Coq Proof Assistant

- *State-of-the* art verification framework
- Based on *dependently typed functional language*
- *Interactive* — requires a human in the loop
- Very small *trusted code base*
- Used to implement fully verified
  - *compilers*
  - *operating systems*
  - *distributed protocols*

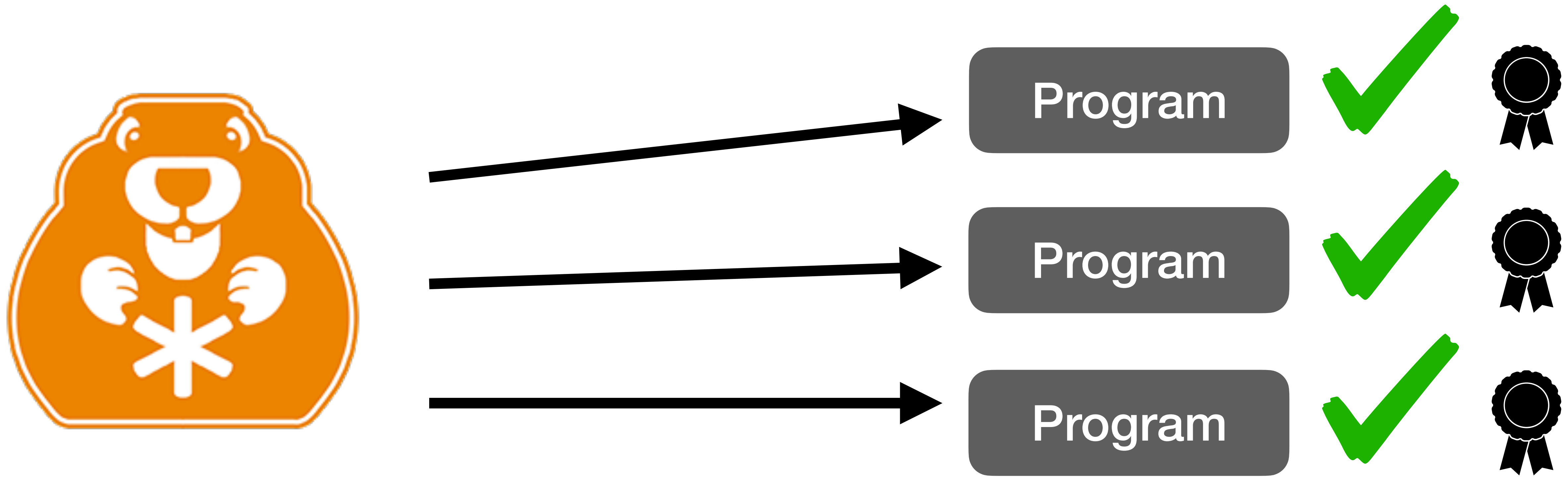
Shifting the burden of trust



SuSLik: Large TCB

Coq: Small TCB

Deductive insight → post-hoc certification



The Design and Implementation of a Certifying Compiler

George C. Necula   Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891
{necula,petel}@cs.cmu.edu

PLDI'98

ICFP'21

Certifying the Synthesis of Heap-Manipulating Programs

YASUNARI WATANABE, Yale-NUS College, Singapore and National University of Singapore, Singapore
KIRAN GOPINATHAN, National University of Singapore, Singapore
GEORGE PÎRLEA, National University of Singapore, Singapore
NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore

Another quick demo?

Today's Agenda

- Deductive synthesis in a nutshell
- Trust in program synthesis
- Extensions and Applications

Today's Agenda

- Deductive synthesis in a nutshell
- Trust in program synthesis
- Extensions and Applications

Extensions and Applications

- Synthesis with *immutability annotations*
  more precise specifications, more "natural" programs (ESOP'20)
- Automated synthesis of *mutually-recursive functions* (PLDI'21)
- Synthesis for *program repair* generating *provably correct patches* (VMCAI'21)
- Deductive synthesis of *Rust programs* from types (PLDI'23)
- Combining deductive synthesis and *synthesis by example* (WIP)

Extensions and Applications

- Synthesis with *immutability annotations*
  more precise specifications, more "natural" programs (ESOP'20)

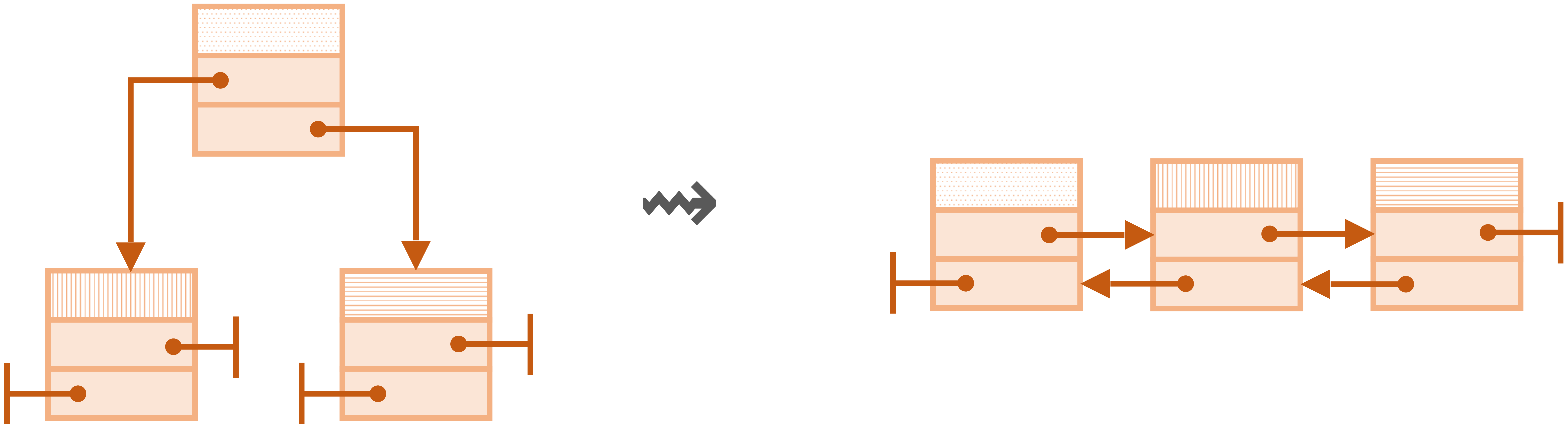
- Automated synthesis of *mutually-recursive functions* (PLDI'21)

- Synthesis for *program repair* generating *provably correct patches* (VMCAI'21)

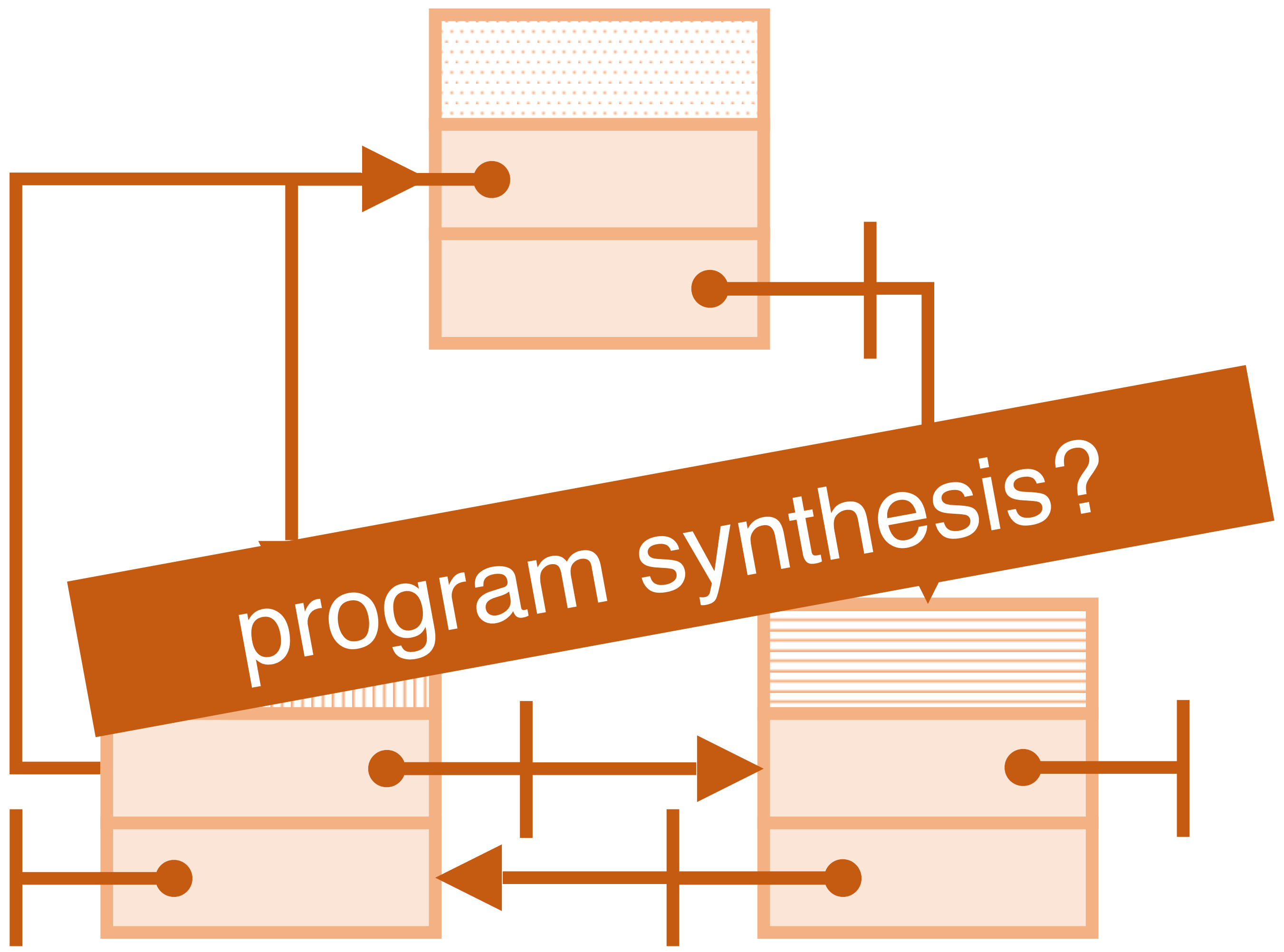
- Deductive synthesis of *Rust programs* from types (PLDI'23)

- Combining deductive synthesis and *synthesis by example* (WIP)

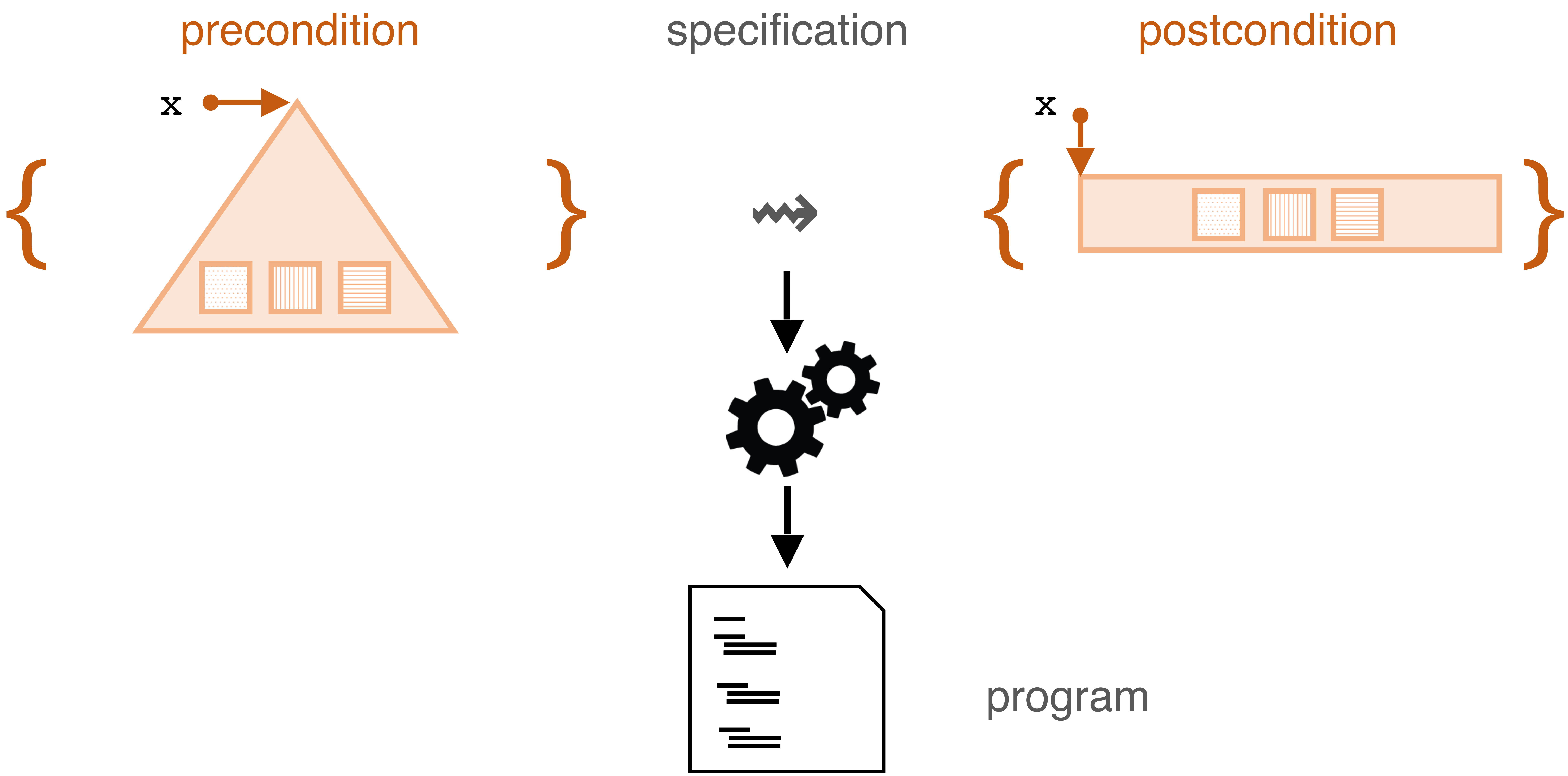
task: flatten a tree into a list



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



we know what to do!



we know what to do!

separation logic

{ tree(x, S) } ⇝ { dll(x, _, S) } specification

culprit: recursion!

program

why SuSLik fails

```
flatten(x) {
```


```
}
```

why SuSLik fails

```
flatten(x) {
  if (x != 0) {
    l = *x.l; r = *x.r;



  }
}
```

why SusLik fails

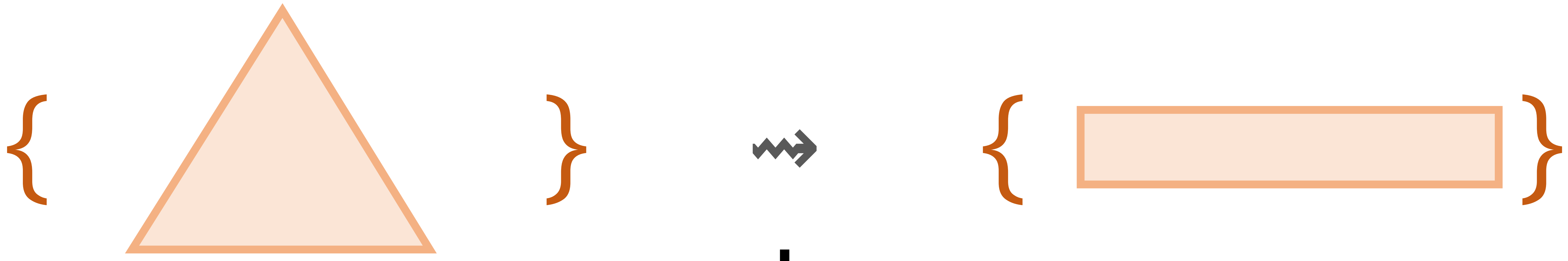
```
flatten(x) {
  if (x != 0) {
    l = *x.l; r = *x.r;
    flatten(l);


  }
}
```

why SuSLik fails

```
flatten(x) {
  if (x != 0) {
    l = *x.l; r = *x.r;
    flatten(l); flatten(r);

  }
}
```

needs recursive function to append two lists!

existing synthesizers



specification

Leon — [Kneuss et a…

Synquid — […olikarpova et al'16]

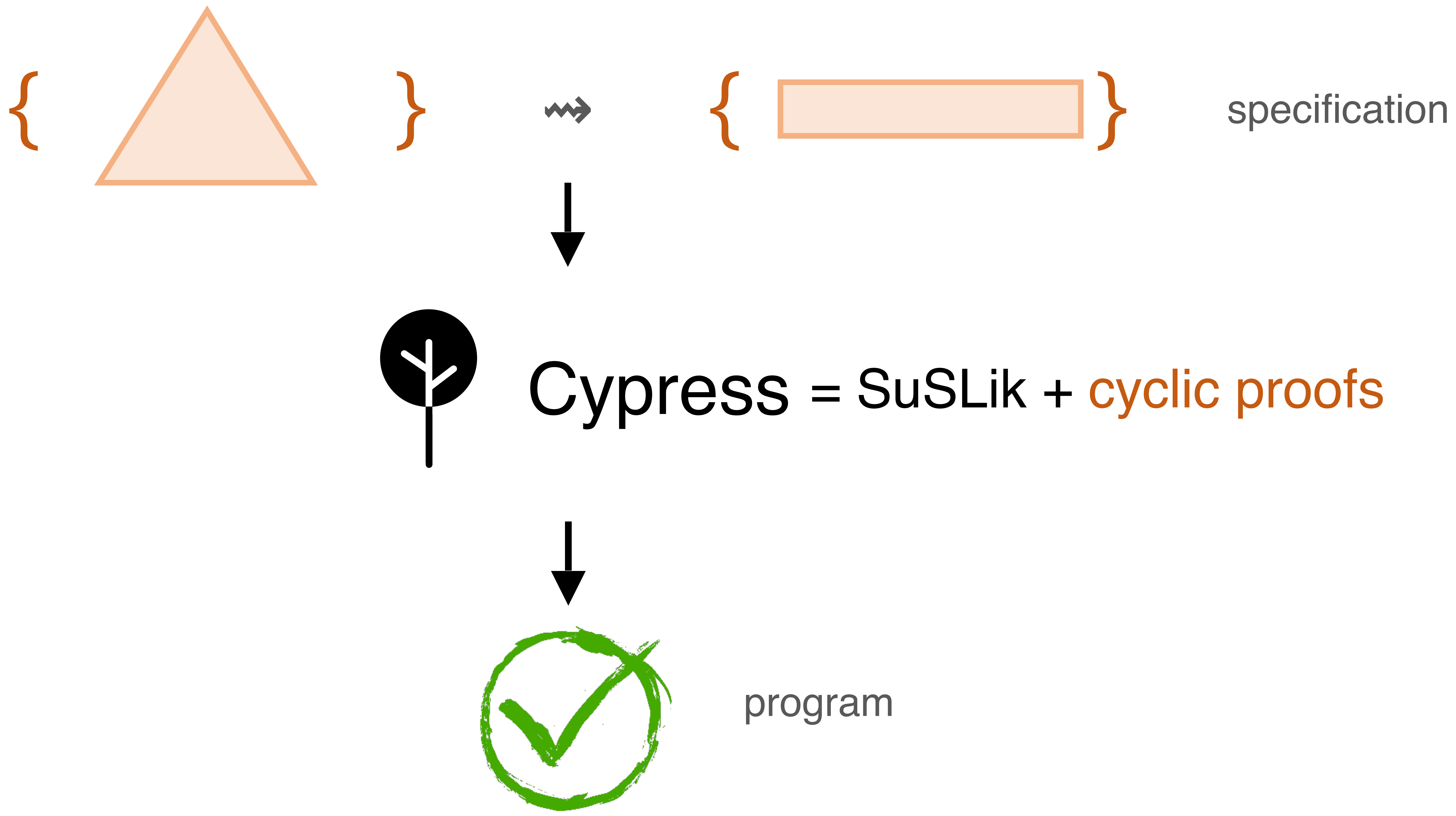
S…Sl… — [Polikarpova, Sergey'19]

…npSynth — [Qui, Solar-Lezama'18]

unable to discover recursive auxiliaries!

program

Idea: cyclic synthesis



tree flattening in Cypress

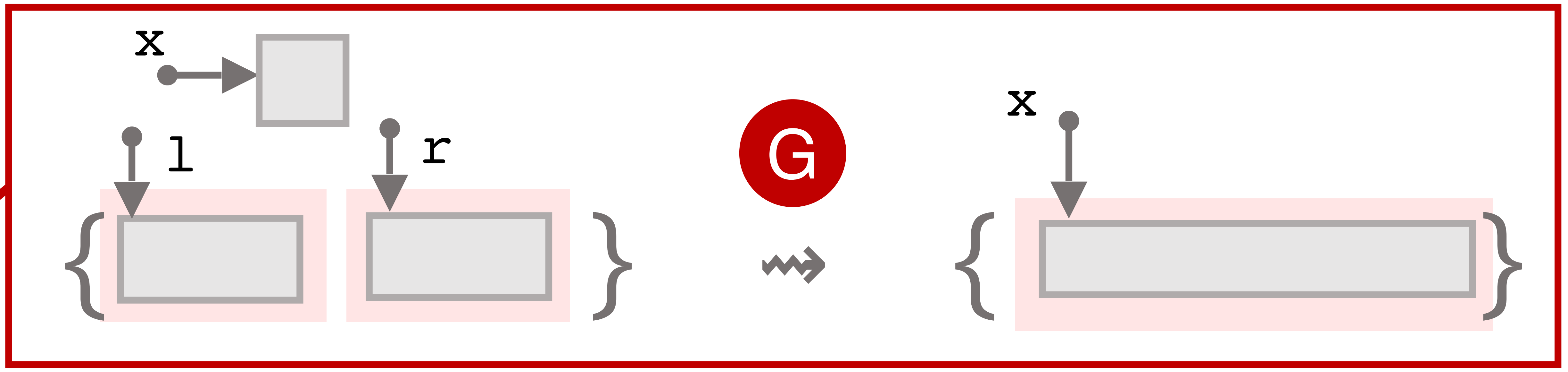
```
flatten(x) {
  if (x != 0) {
    ...


  }
}
```

tree flattening in Cypress

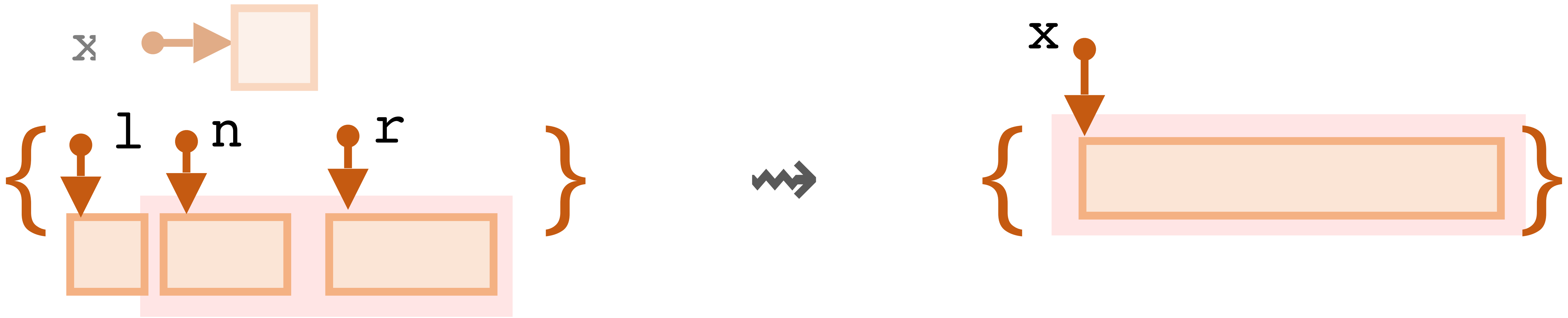
```
flatten(x) {
   if (x != 0) {
      …
      if (l == 0) { … } else {
         n = *l.nxt;
```


```
      }
   }
}
```

does this goal look familiar?



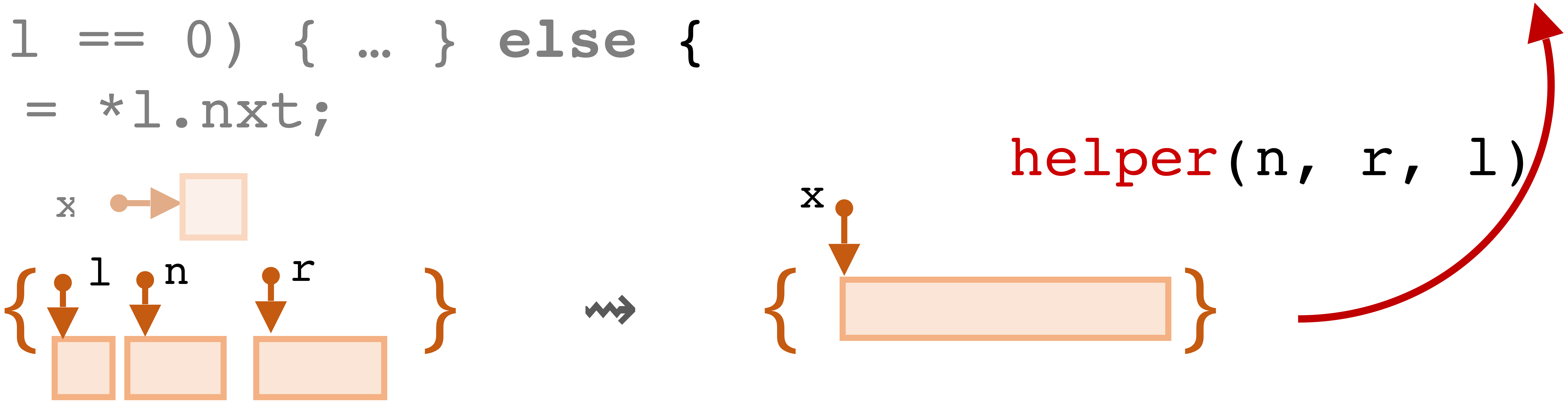

```
flatten(x) {
  if (x != 0) {
    …
    if (l == 0) { … } else {
      n = *l.nxt;
```



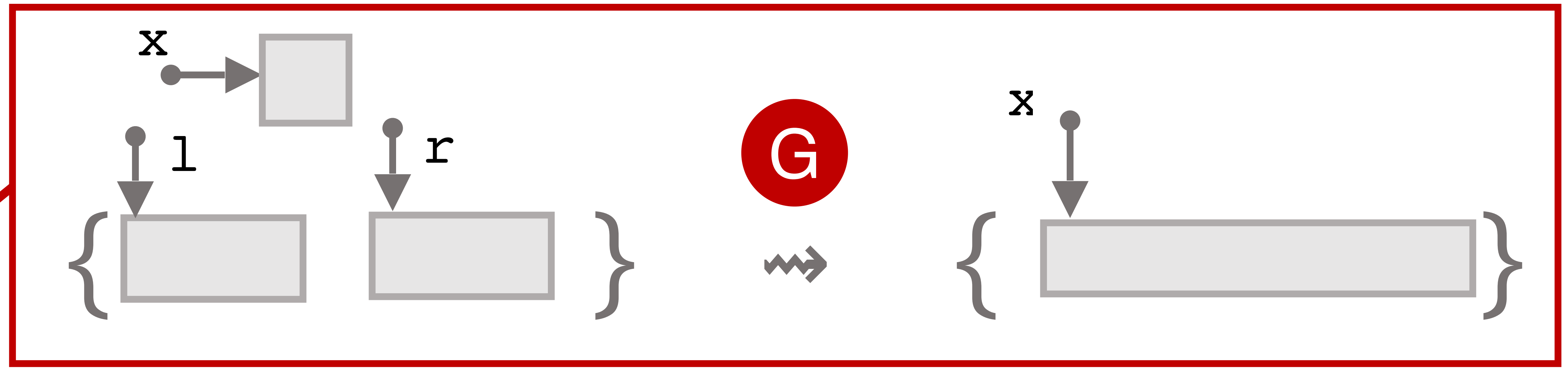

```
    }
  }
}
```

let's cycle back!

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

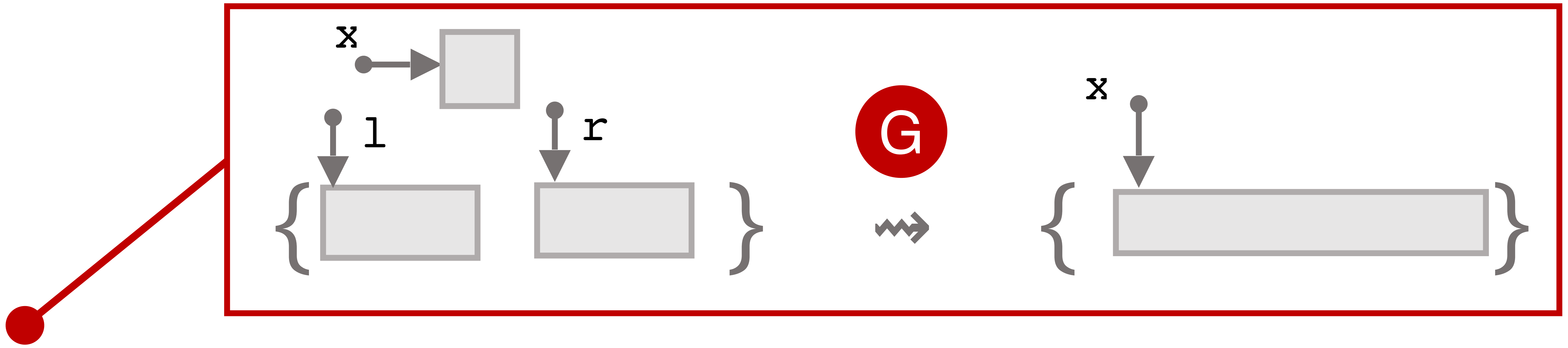


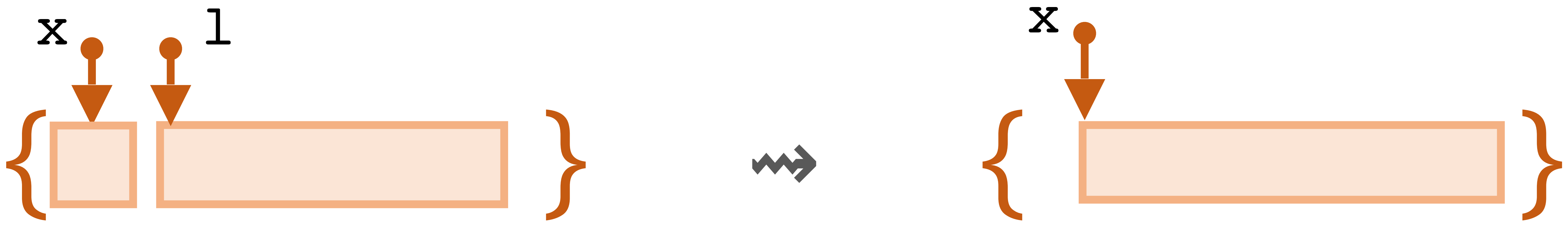

```
        }
    }
}
```



let’s cycle back!

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

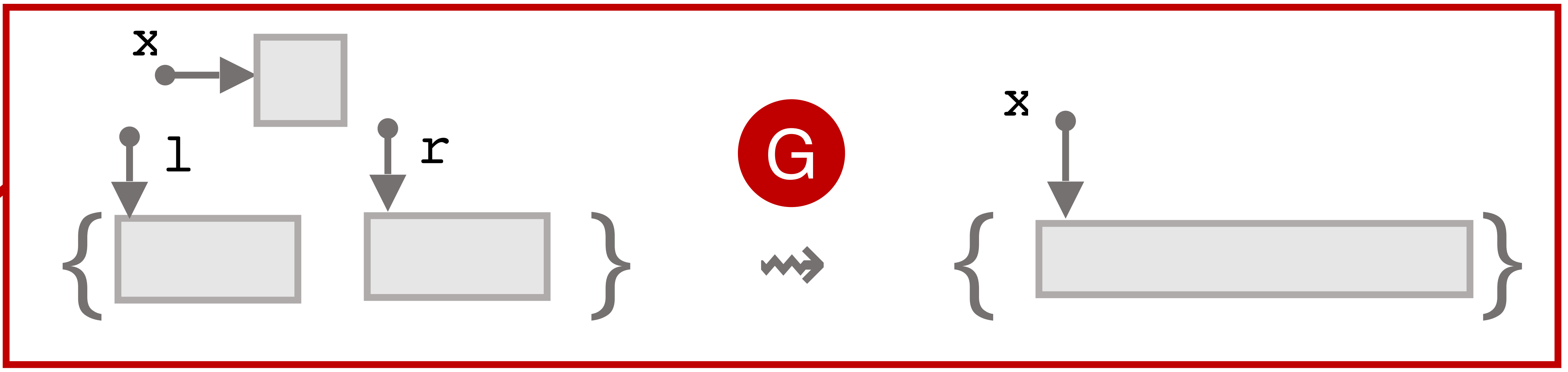




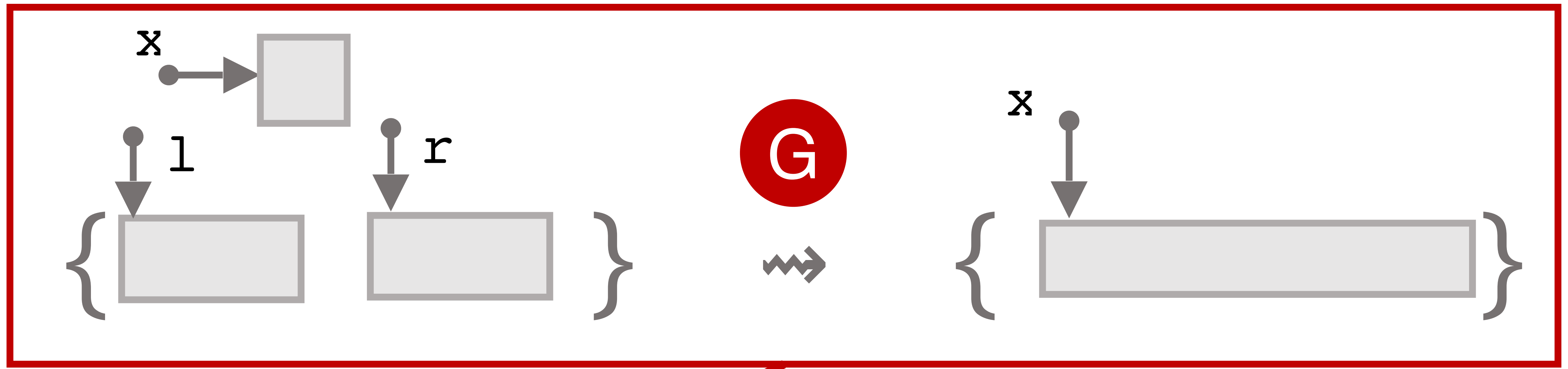

```
        }
    }
}
```

extracting the auxiliary

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



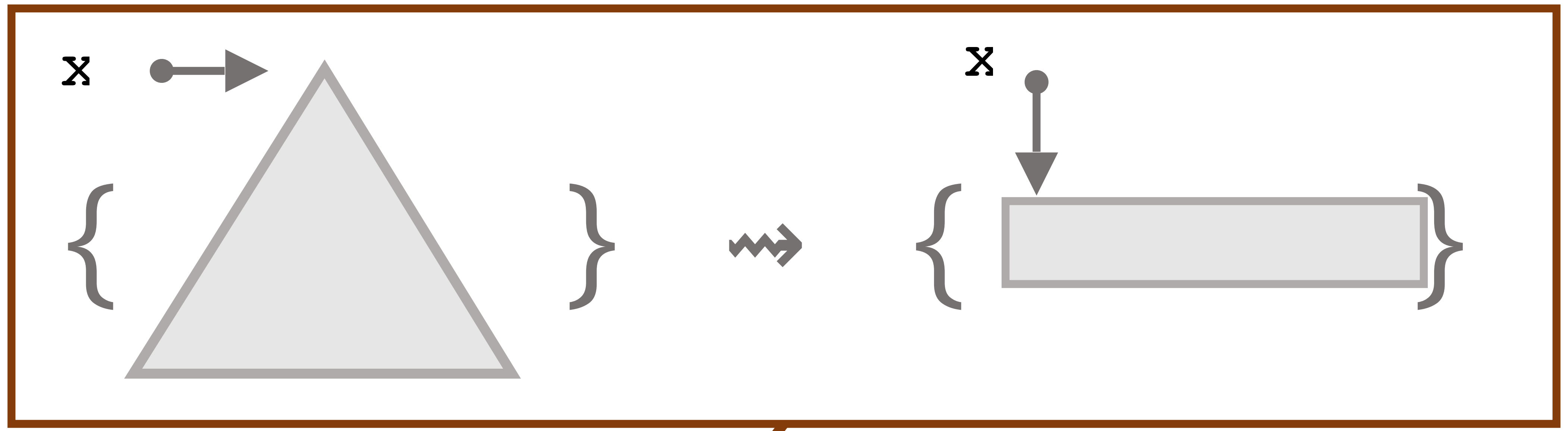
extracting the auxiliary

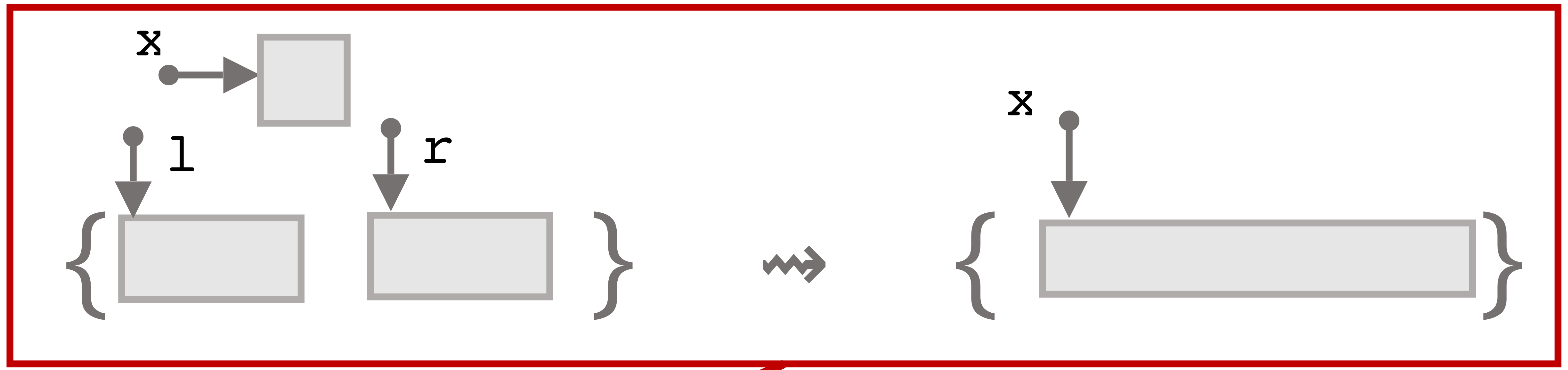



```
flatten(x) {
   if (x != 0) {
      …
      if (l == 0) { … } else
      {
         n = *l.nxt;
         helper(n, r, l);
         …
   } }
}
```

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

}
```

extracting the auxiliary






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

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

Yet another demo?

what else can it do?

nested traversals

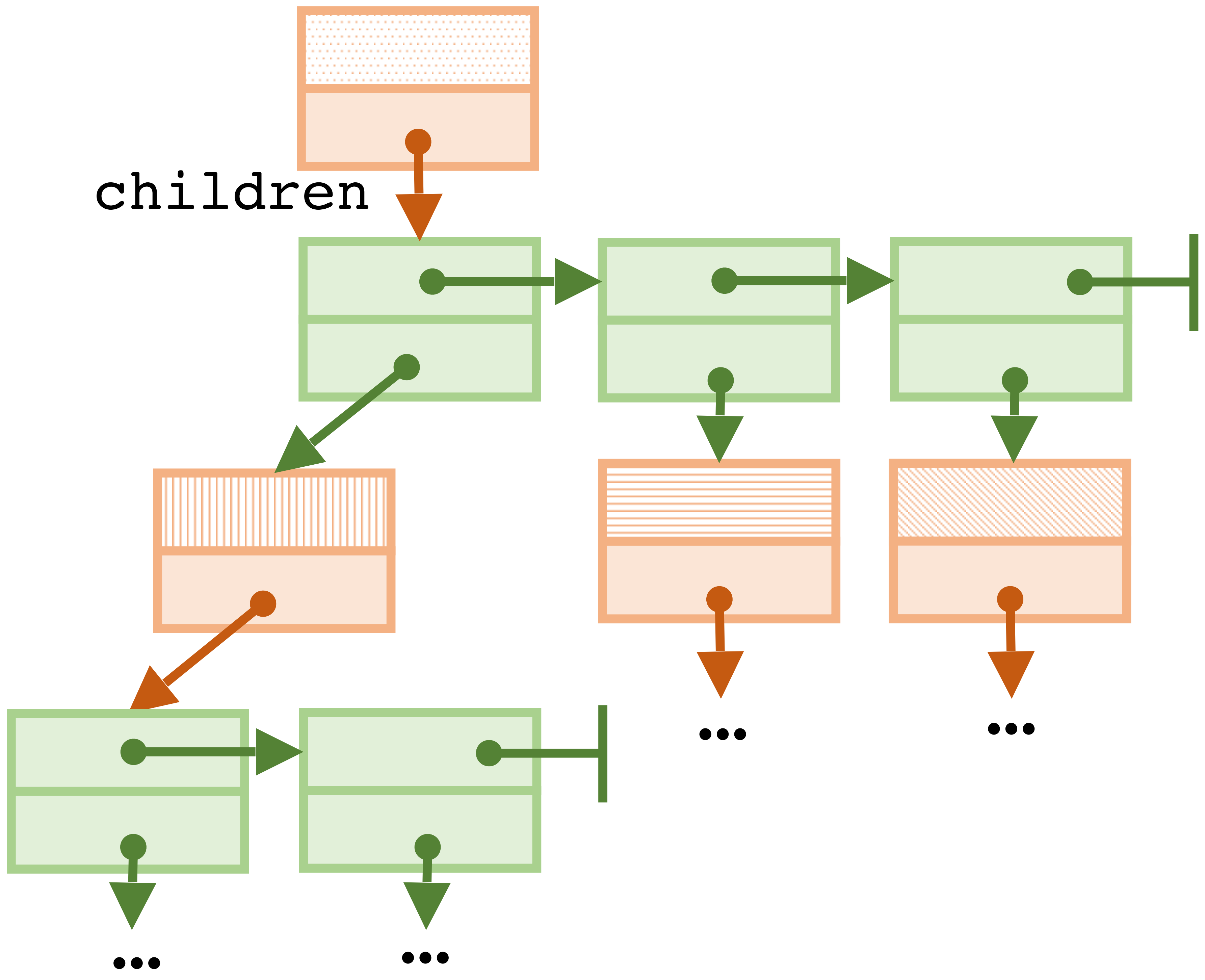
e.g. list sorting, deduplication

non-trivial termination metrics

e.g. sorted list merge

mutual recursion

e.g. n-ary tree flattening



what else can it do?

nested traversals

e.g. list sorting, deduplication

non-trivial termination metrics

e.g. sorted list merge

mutual recursion

e.g. n-ary tree flattening

2–40 sec

Artifacts Available V1.1 · Artifacts Evaluated Reusable V1.1 · Artifacts Evaluated Functional V1.1

Cyclic Program Synthesis

Shachar Itzhaky
Technion
Israel
shachari@cs.technion.ac.il

Hila Peleg
University of California, San Diego
USA
hpeleg@eng.ucsd.edu

Nadia Polikarpova
University of California, San Diego
USA
nadia.polikarpova@ucsd.edu

Reuben N. S. Rowe
Royal Holloway, University of London
United Kingdom
reuben.rowe@rhul.ac.uk

Ilya Sergey
Yale-NUS College
National University of Singapore
Singapore
ilya.sergey@yale-nus.edu.sg

Today's Agenda

- Proof-based synthesis in a nutshell
- Trust in program synthesis
- Extensior

Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities
(Invited Paper)

Shachar Itzhaky[1], Hila Peleg[2], Nadia Polikarpova[2], Reuben N. S. Rowe[3], and Ilya Sergey[4]

To Take Away

- Program Synthesis —
  generating a program given a *specification*.

- Deductive Program Synthesis —
  synthesising a program as a *proof in a domain-specific logic*.

- Deductive Synthesis via Separation Logic —
  synthesising *correct-by-construction heap-manipulating programs*.

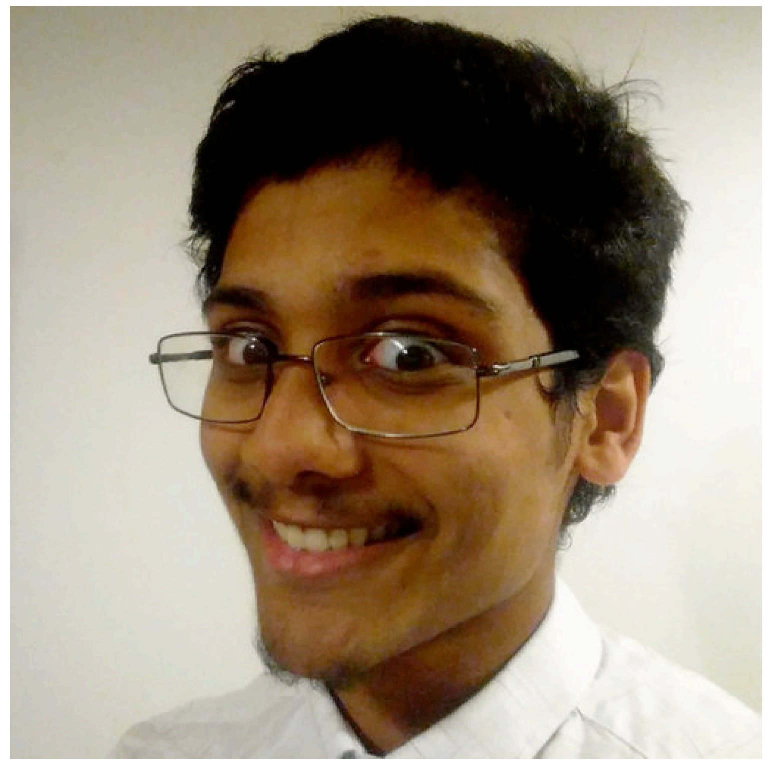

Nadia Polikarpova

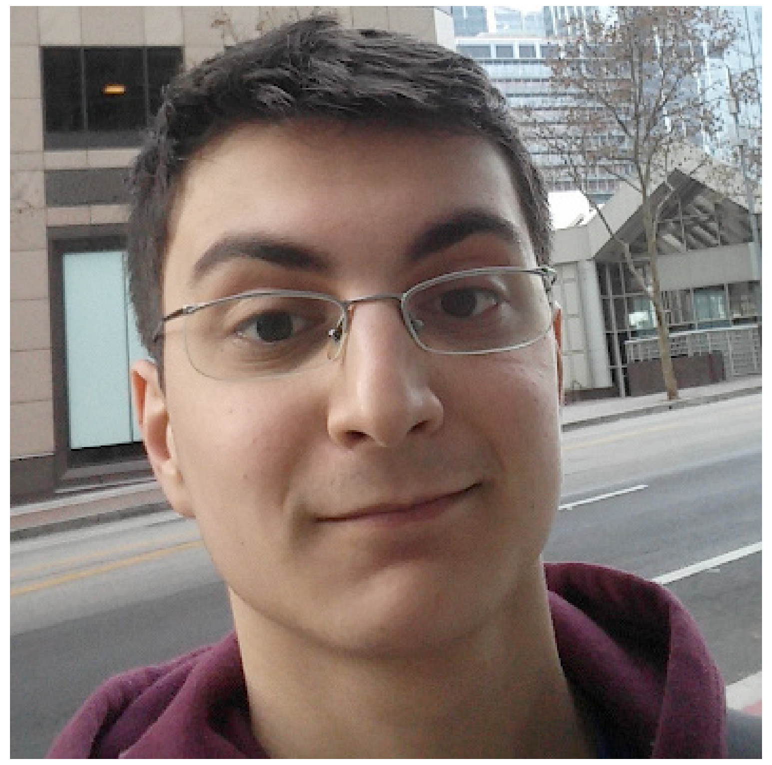

Shachar Itzhaky

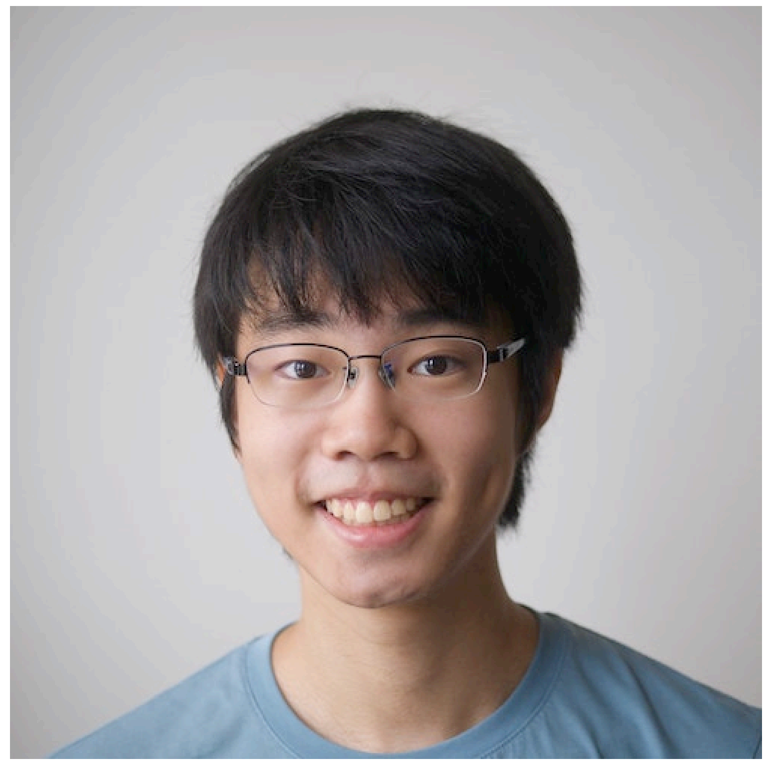

Hila Peleg


Reuben Rowe


Andreea Costea


Kiran Gopinathan


George Pîrlea


Yasunari Watanabe


Amy Zhu

Resources

- Papers:
  - *Structuring the Synthesis of Heap-Manipulating Programs*, POPL'19
  - *Cyclic Program Synthesis*, PLDI'21
  - *Certifying the Synthesis of Heap-Manipulating Programs*, ICFP'21
  - *Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities*, CAV'21
  - *Leveraging Rust Types for Program Synthesis*, PLDI'23, to appear
- On GitHub: https://github.com/TyGuS/suslik
- Google: "**suslik synthesis**"

Thanks!

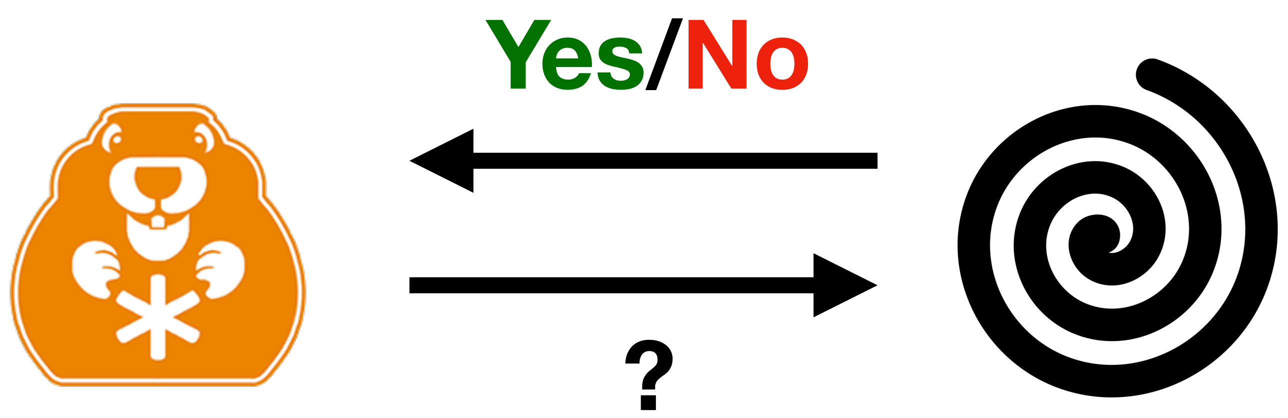
Backup Slides

SuSLik solves pure assertions with SMT

pure assertions

$$\vdash \Phi \Rightarrow \Psi$$

Synthesis

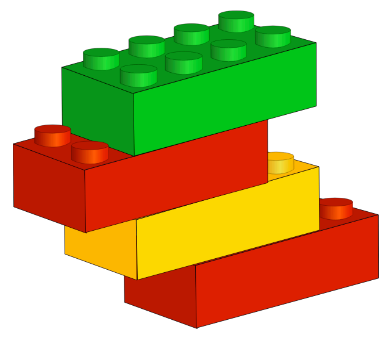


SMT solver

Verification

HTT

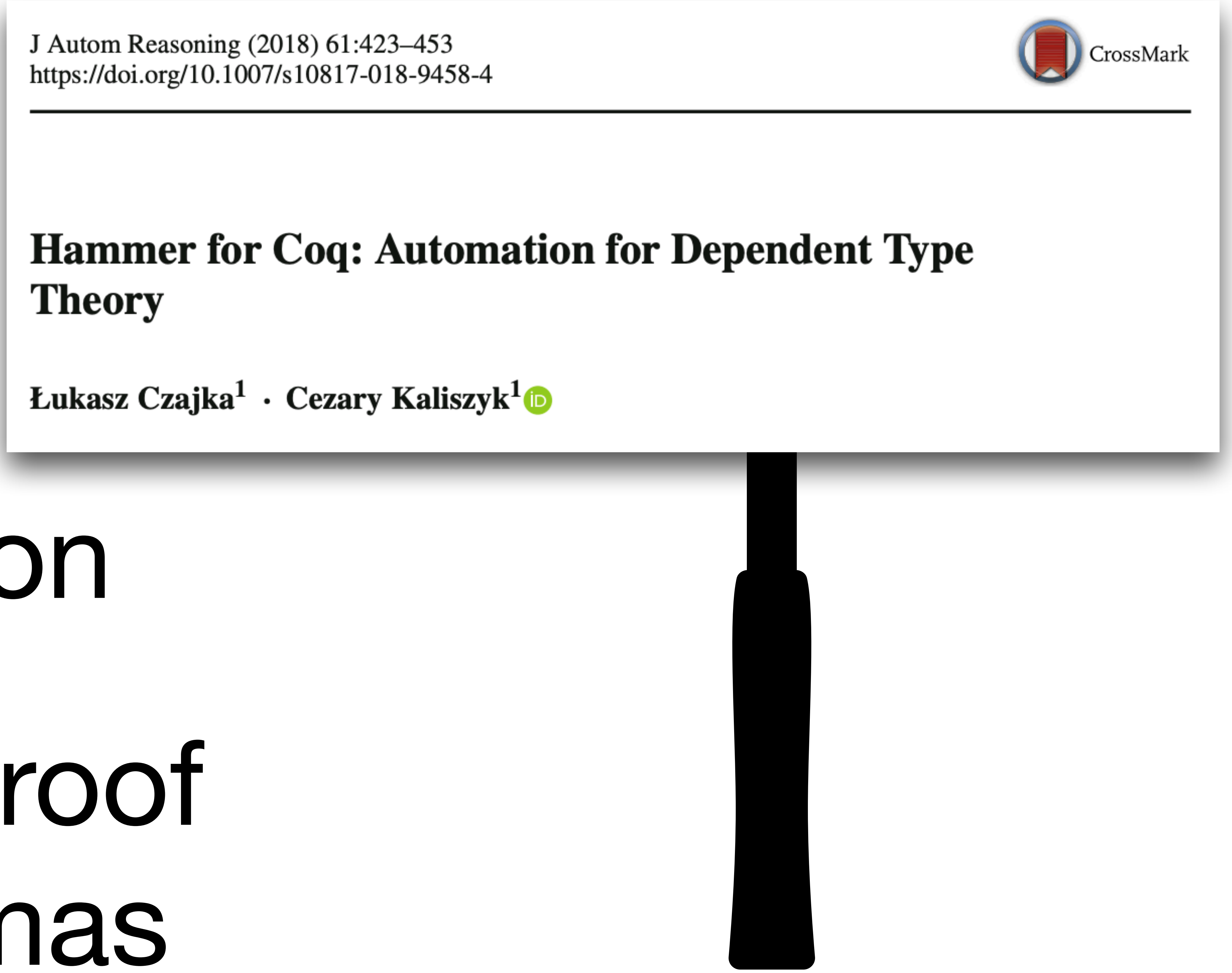
```
apply …
rewrite …
apply …
```



constructive proof

Solution: certified solvers (hammers)

- Single-line commands
- Powerful proof automation
- Advanced ATP-guided proof search on available lemmas


J Autom Reasoning (2018) 61:423–453
https://doi.org/10.1007/s10817-018-9458-4

CrossMark

Hammer for Coq: Automation for Dependent Type Theory

Łukasz Czajka[1] · **Cezary Kaliszyk**[1]

Hammer time!

Capture and extract entailments into lemmas

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).
```

Hammer time!

Prove extracted lemma with CoqHammer[2]

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).

Proof. intros. hammer. Qed.
```

Lemma becomes usable for automation

Main proof

???

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).

Proof. intros. hammer. Qed.
```

Lemma becomes usable for automation

Main proof

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).

Proof. intros. hammer. Qed.
```