SuSLik: synthesis of safe pointer-manipulating programs

Nadia Polikarpova

joint work with Ilya Sergey (Yale-NUS)

UCSDCSE
Computer Science and Engineering

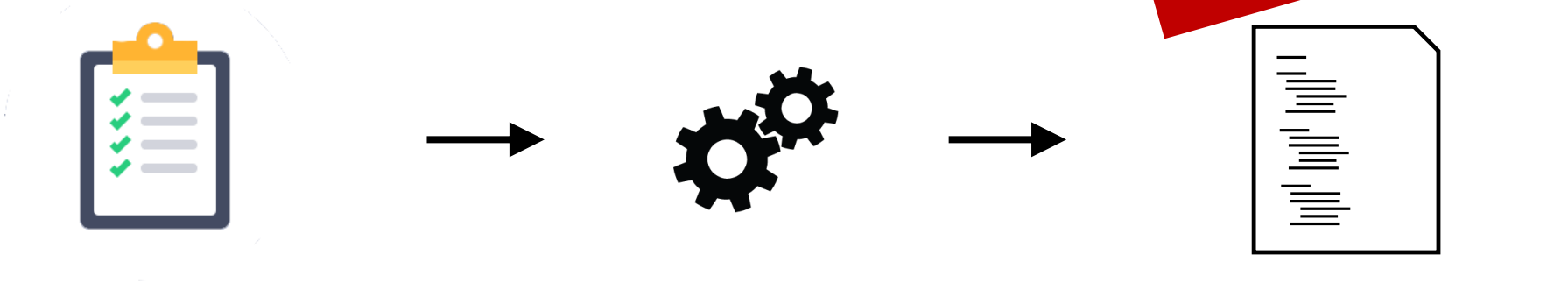
program synthesis

specification

code

program synthesis

specification



☹ verbose
☹ unstructured
☹ pointers & aliasing

SuSLik

Synthesis Using Separation Logik

the SuSLik approach

separation logic

☺ reasoning about pointers & aliasing

deductive synthesis

☺ uses specs to guide synthesis

code

this talk

1. example: swap
   a taste of SuSLik
2. the logic
   specifying pointer-manipulating programs
3. deductive synthesis
   from SL specifications to programs
4. cyclic synthesis
   complex programs via cyclic proofs

Polikarpova, Sergey [POPL'19]

ongoing work

this talk

1. example: swap
2. the logic
3. deductive synthesis
4. cyclic synthesis

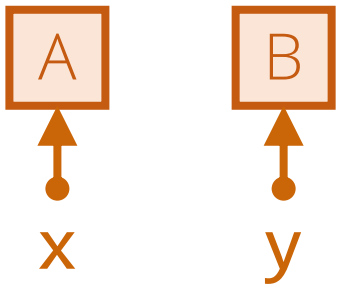
example: swap

Swap values of two *distinct* pointers

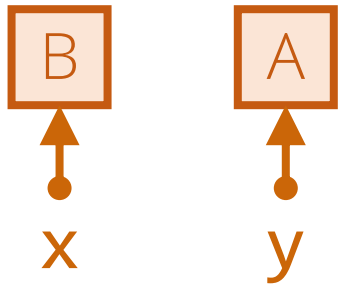
```
void swap(loc x, loc y)
```

example: swap

start state:



end state:



in separation logic:

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

separately

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

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

ghost variables

demo: swap

Swap values of two *distinct* pointers

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

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

??

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

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

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

??

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

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

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

??

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

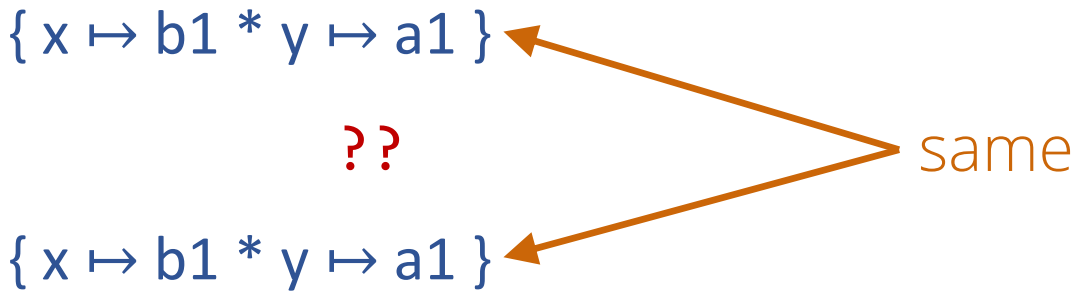
```
let a1 = *x;
let b1 = *y;
*x = b1;
```

{ x ↦ b1 * y ↦ b1 }

??

{ x ↦ b1 * y ↦ a1 }

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



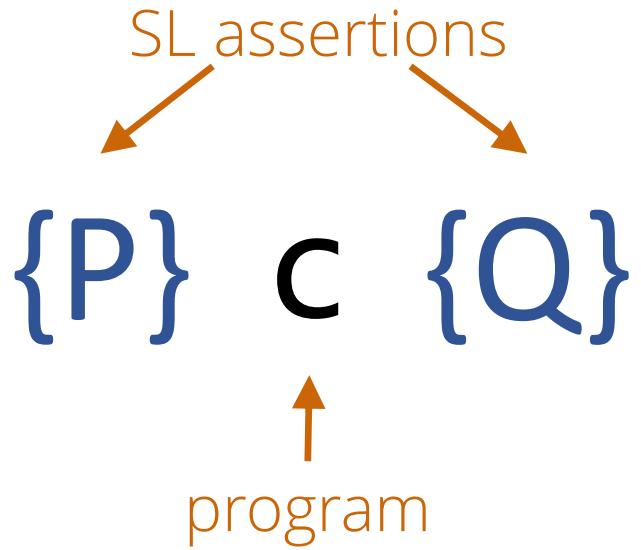
```
let a1 = *x;
let b1 = *y;
*x = b1;
*y = a1;
```

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

this talk

1. example: swap
2. the logic
3. deductive synthesis
4. cyclic synthesis

separation logic (SL)

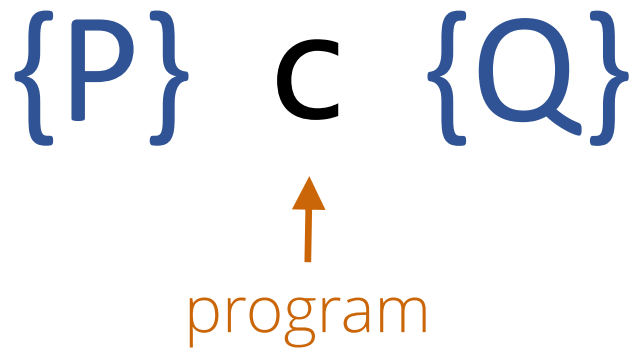


starting in a state that satisfies P
program **c** will execute without memory errors,
and upon its termination the state will satisfy Q

this talk

1. example: swap
2. the logic
   - 2.1. programs
   - 2.2. assertions
   - 2.3. specifying data transformations
3. deductive synthesis
4. cyclic synthesis

separation logic (SL)



programs

do nothing **skip**

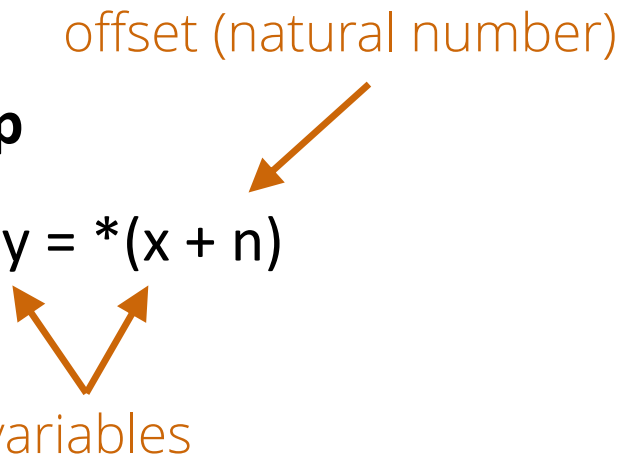
programs

do nothing

skip

read from heap

let y = *(x + n)



programs

do nothing	**skip**
read from heap	**let** y = *(x + n)
write to heap	*(x + n) = e ← expression (arithmetic, boolean)

programs

do nothing	**skip**
read from heap	**let** y = *(x + n)
write to heap	*(x + n) = e
allocate block	**let** y = **malloc**(n)

programs

do nothing	**skip**
read from heap	**let** y = *(x + n)
write to heap	*(x + n) = e
allocate block	**let** y = **malloc**(n)
free block	**free**(x)

programs

do nothing	**skip**
read from heap	**let** y = *(x + n)
write to heap	*(x + n) = e
allocate block	**let** y = **malloc**(n)
free block	**free**(x)
procedure call	$p(e_1, \ldots, e_n)$

programs

do nothing	**skip**
read from heap	**let** y = *(x + n)
write to heap	*(x + n) = e
allocate block	**let** y = **malloc**(n)
free block	**free**(x)
procedure call	$p(e_1, \ldots, e_n)$
~~assignment~~	only heap is mutable, not stack variables!

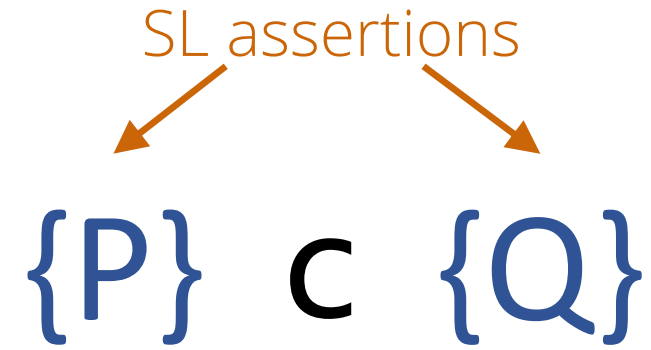
programs

do nothing	**skip**
read from heap	**let** y = *(x + n)
write to heap	*(x + n) = e
allocate block	**let** y = **malloc**(n)
free block	**free**(x)
procedure call	$p(e_1, \ldots, e_n)$
sequential composition	$c_1 ; c_2$
conditional	**if** (e) $\{c_1\}$ else $\{c_2\}$

this talk

1. example: swap
2. the logic
   - 2.1. programs
   - 2.2. assertions
   - 2.3. specifying data transformations
3. deductive synthesis
4. cyclic synthesis

separation logic (SL)



SL assertions

empty heap { emp }

SL assertions

empty heap { emp }

singleton heap { y ↦ 5 }

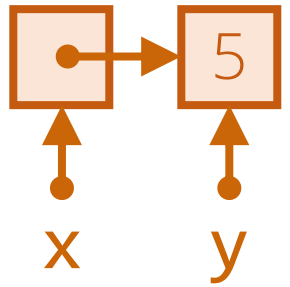
SL assertions

empty heap { emp }

singleton heap { y ↦ 5 }

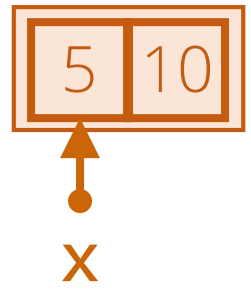
separating conjunction { x ↦ y * y ↦ 5 }

heaplets



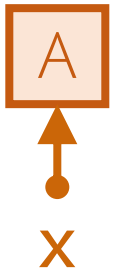
SL assertions

empty heap	{ emp }
singleton heap	{ y ↦ 5 }
separating conjunction	{ x ↦ y * y ↦ 5 }
memory block	{ [x, 2] * x ↦ 5 * (x + 1) ↦ 10 }

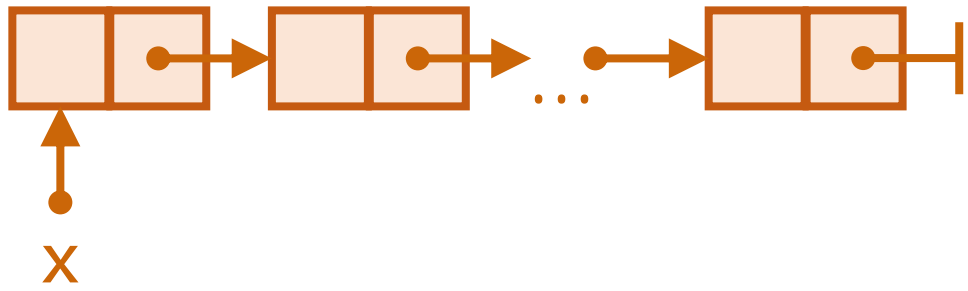


SL assertions

empty heap	{ emp }
singleton heap	{ y ↦ 5 }
separating conjunction	{ x ↦ y * y ↦ 5 }
memory block	{ [x, 2] * x ↦ 5 * (x + 1) ↦ 10 }
+ pure formula	{ A > 5 ; x ↦ A }



SL assertions: linked structures

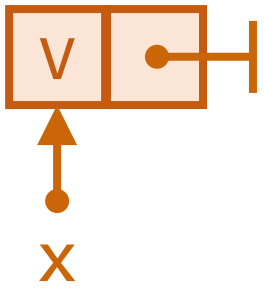


SL assertions: linked structures

linked list $\{ x = 0 ; emp \}$

SL assertions: linked structures

linked list $\{ [x, 2] * x \mapsto V * (x + 1) \mapsto 0 \}$

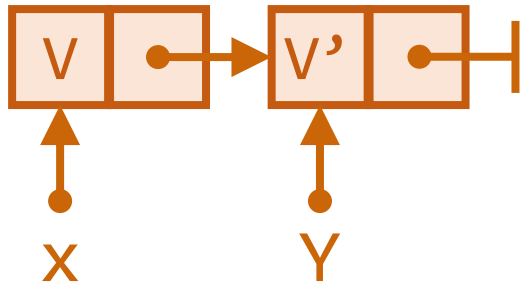


SL assertions: linked structures

linked list $\{ [x, 2] * x \mapsto V * (x + 1) \mapsto Y *$

$[Y, 2] * Y \mapsto V' * (Y + 1) \mapsto 0$

$\}$



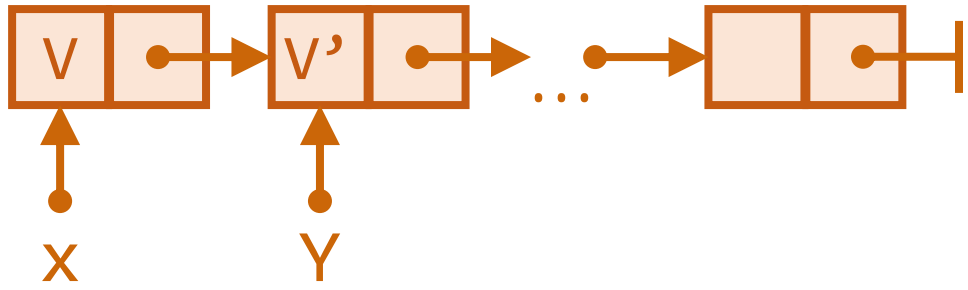
SL assertions: linked structures

linked list $\{ [x, 2] * x \mapsto V * (x + 1) \mapsto Y *$

$[Y, 2] * Y \mapsto V' * (Y + 1) \mapsto Y' *$

…

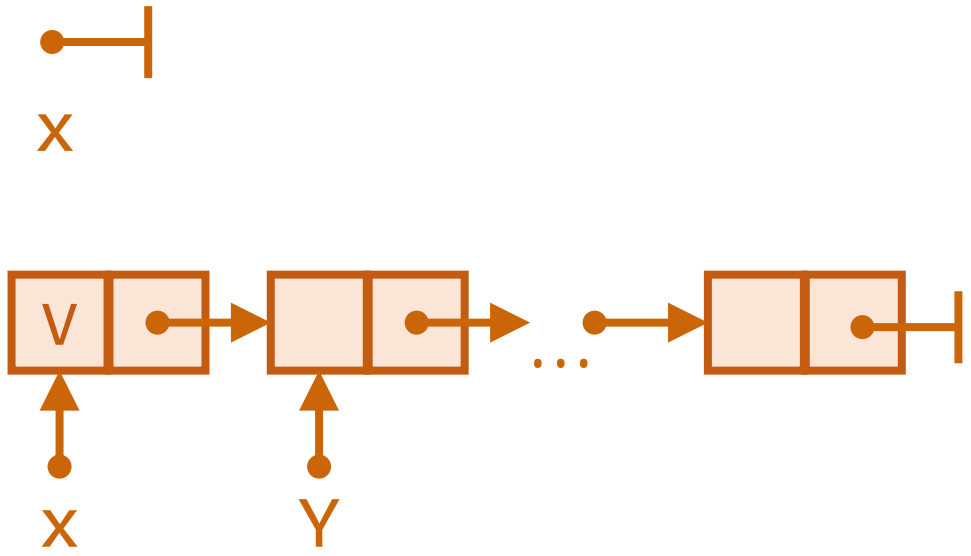
$\}$



inductive predicates to the rescue!

the linked list predicate

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



this talk

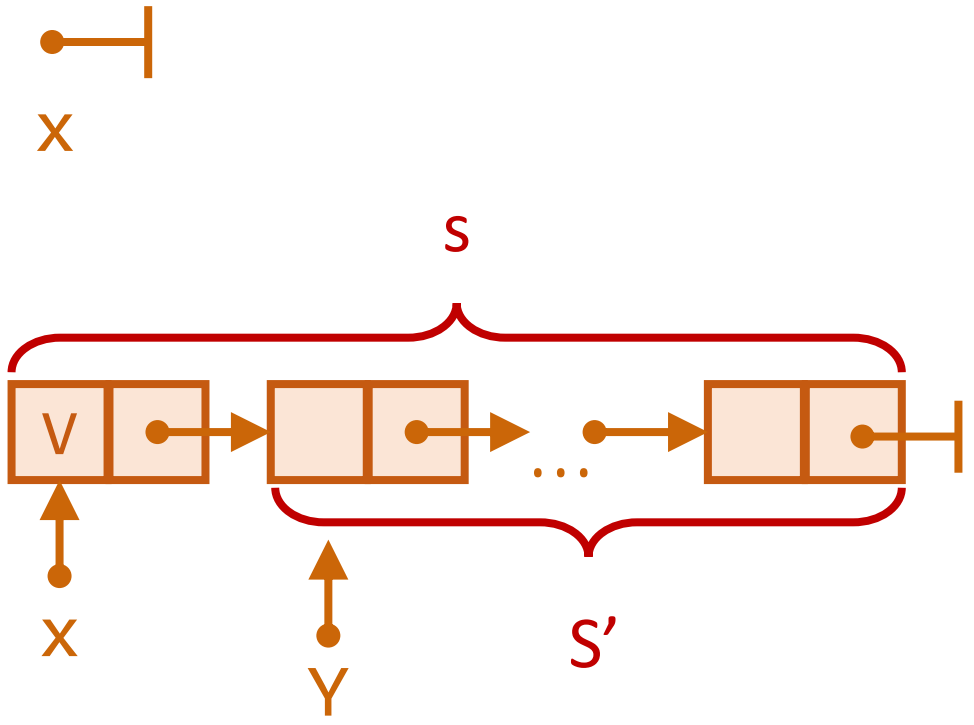
1. example: swap
2. the logic
   - 2.1. programs
   - 2.2. assertions
   - 2.3. specifying data transformations
3. deductive synthesis
4. cyclic synthesis

demo: dispose a list

```
void dispose(loc x)
{ list(x) }
{ 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: copy a list

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

{ list(x, S) * r ↦ _ }

{ list(x, S) * r ↦ Y * list(Y, S) }

return location

this talk

1. example: swap
2. the logic
3. deductive synthesis
4. cyclic synthesis

deductive synthesis

synthesis as proof search

this talk

1. example: swap
2. the logic
3. deductive synthesis
   3.1. proof system
   3.2. proof search
4. cyclic synthesis

transforming entailment

$$P \rightsquigarrow Q \mid c$$

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

synthetic separation logic (SSL)

proof system for
transforming entailment

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

(Emp)

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

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

(Frame)

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

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

(Write)

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

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

(Read)

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

SSL: basic rules

(Emp)

$$\{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid \texttt{skip}$$

(Read)

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

(Frame)

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

(Write)

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

example: swap

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

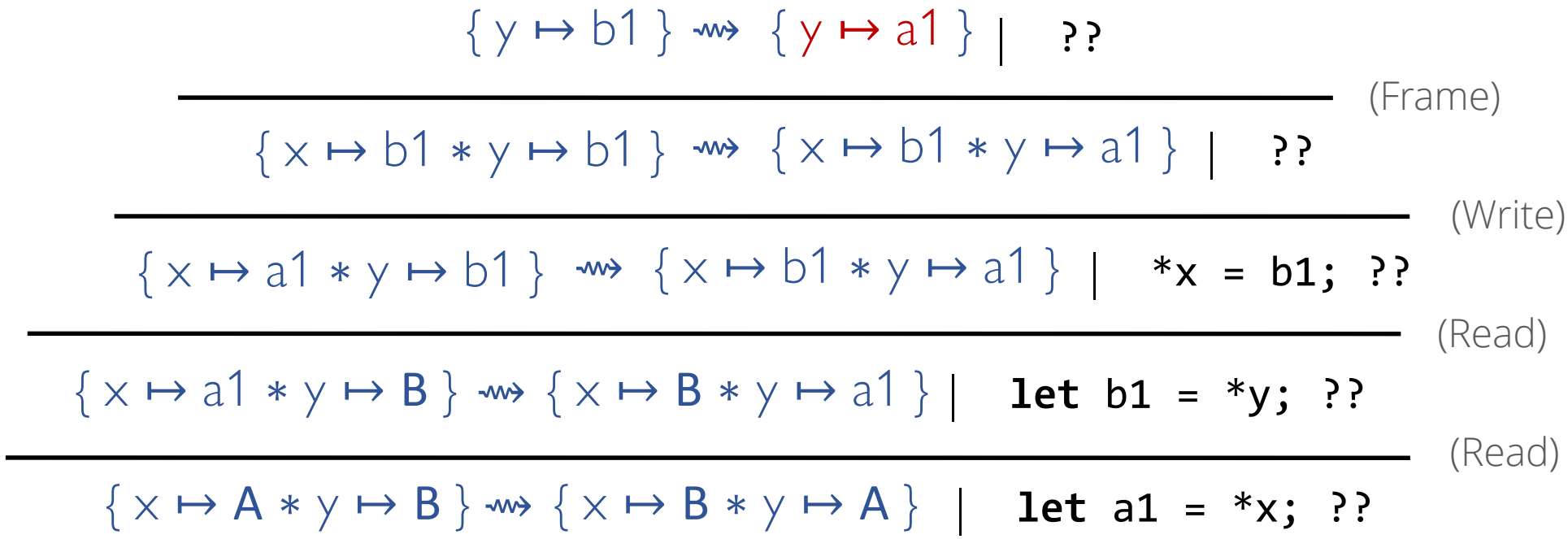
$$\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \quad ??$$

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

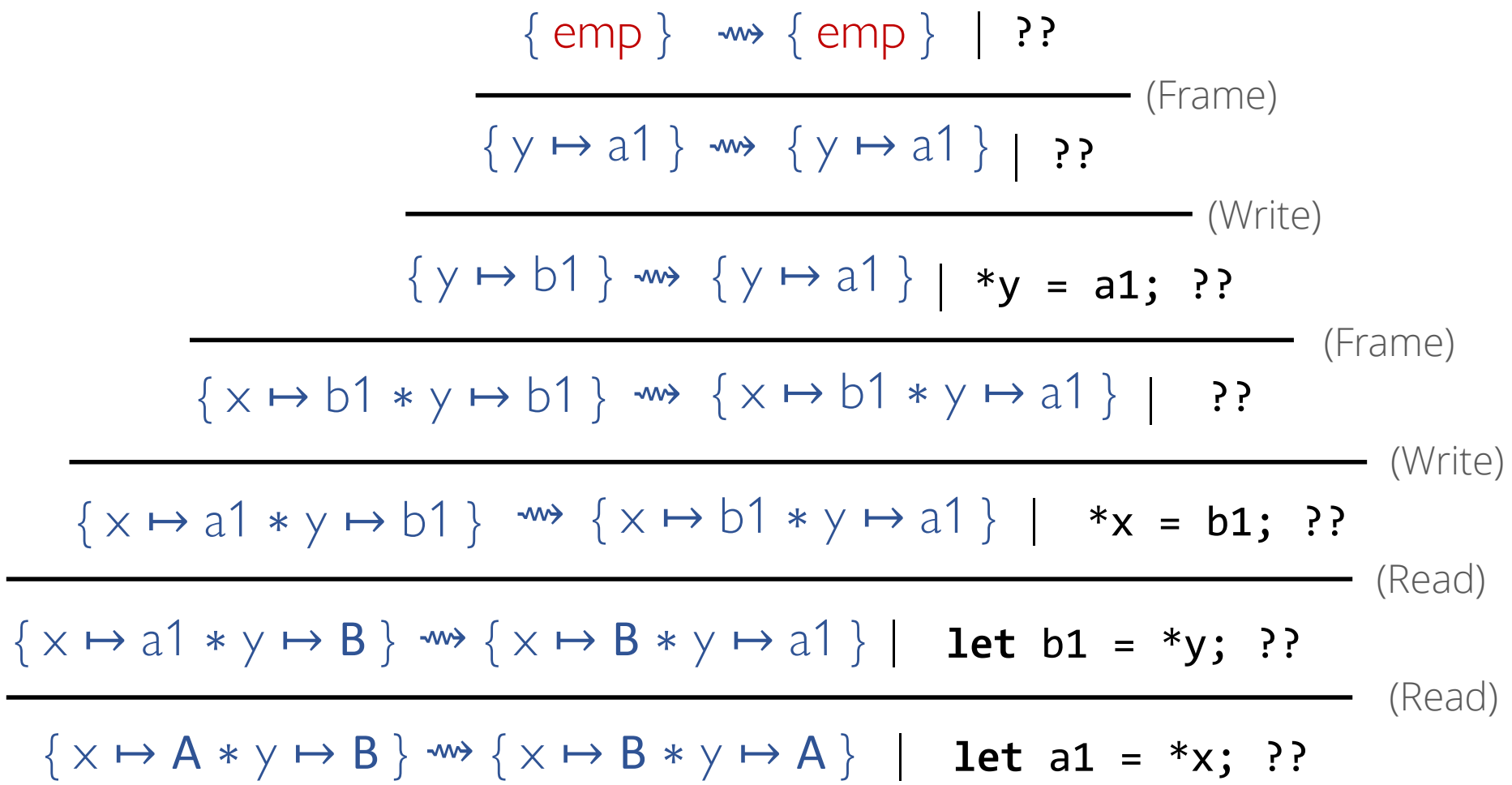
$$\frac{\{\, x \mapsto a1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \texttt{??}}{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \texttt{let b1 = *y; ??}}\ \text{(Read)}$$

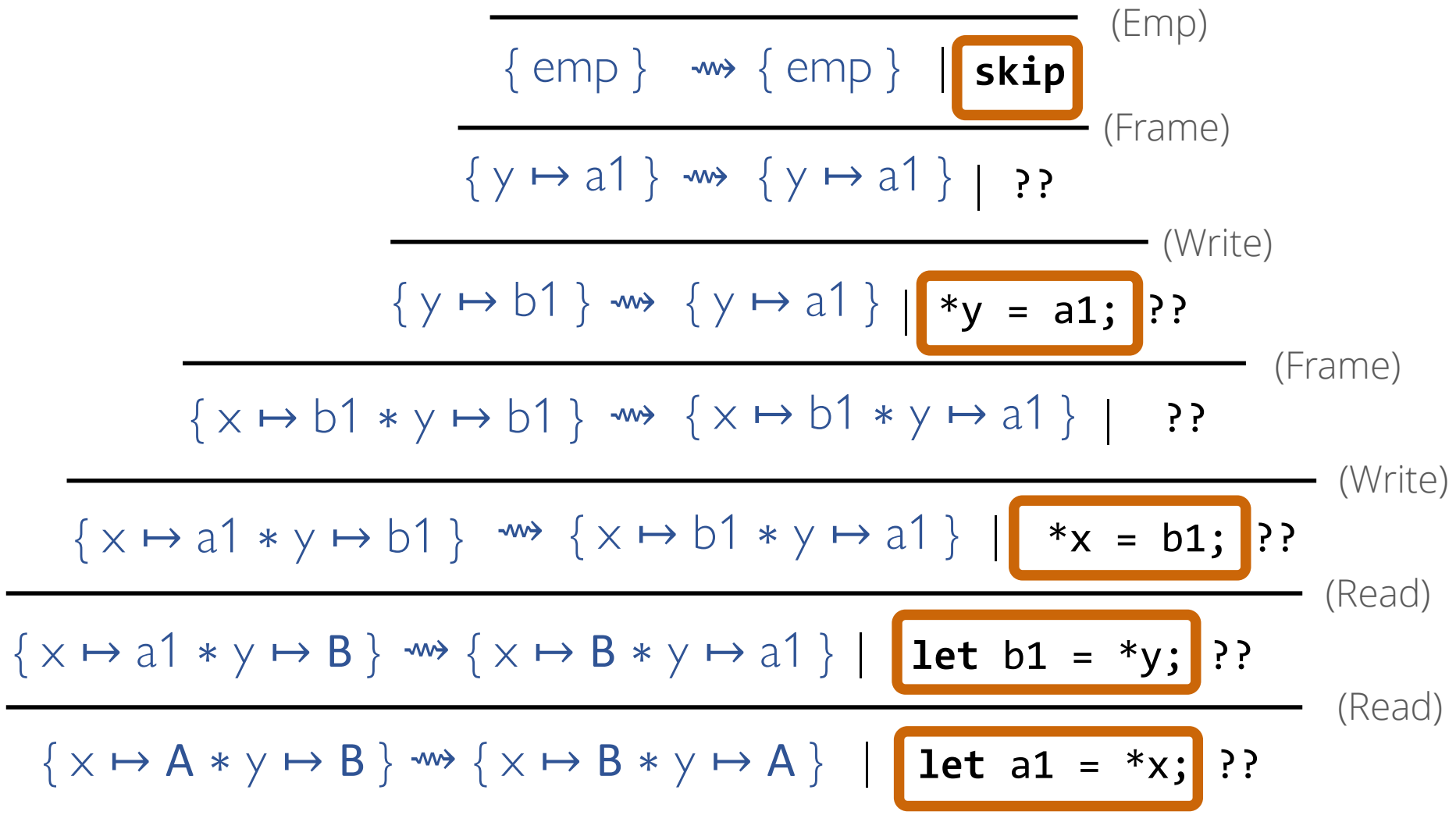
$$\frac{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \texttt{let b1 = *y; ??}}{\{\, x \mapsto A * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto A \,\} \mid \texttt{let a1 = *x; ??}}\ \text{(Read)}$$

$$\dfrac{\dfrac{\dfrac{\{\, x \mapsto b1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \texttt{??}}{\{\, x \mapsto a1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \texttt{*x = b1; ??}}\ \text{(Write)}}{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \texttt{let b1 = *y; ??}}\ \text{(Read)}}{\{\, x \mapsto A * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto A \,\} \mid \texttt{let a1 = *x; ??}}\ \text{(Read)}$$


{ y ↦ b1 } ⇝ { y ↦ a1 } | ??
(Frame)
{ x ↦ b1 * y ↦ b1 } ⇝ { x ↦ b1 * y ↦ a1 } | ??
(Write)
{ x ↦ a1 * y ↦ b1 } ⇝ { x ↦ b1 * y ↦ a1 } | *x = b1; ??
(Read)
{ x ↦ a1 * y ↦ B } ⇝ { x ↦ B * y ↦ a1 } | let b1 = *y; ??
(Read)
{ x ↦ A * y ↦ B } ⇝ { x ↦ B * y ↦ A } | let a1 = *x; ??

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\, y \mapsto a1 \,\} \rightsquigarrow \{\, y \mapsto a1 \,\} \mid \texttt{??}}{\{\, y \mapsto b1 \,\} \rightsquigarrow \{\, y \mapsto a1 \,\} \mid \texttt{*y = a1; ??}}\ \text{(Write)}}{\{\, x \mapsto b1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \texttt{??}}\ \text{(Frame)}}{\{\, x \mapsto a1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \texttt{*x = b1; ??}}\ \text{(Write)}}{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \texttt{let b1 = *y; ??}}\ \text{(Read)}}{\{\, x \mapsto A * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto A \,\} \mid \texttt{let a1 = *x; ??}}\ \text{(Read)}$$


{ emp } ⇝ { emp } | ??
(Frame)
{ y ↦ a1 } ⇝ { y ↦ a1 } | ??
(Write)
{ y ↦ b1 } ⇝ { y ↦ a1 } | *y = a1; ??
(Frame)
{ x ↦ b1 ∗ y ↦ b1 } ⇝ { x ↦ b1 ∗ y ↦ a1 } | ??
(Write)
{ x ↦ a1 ∗ y ↦ b1 } ⇝ { x ↦ b1 ∗ y ↦ a1 } | *x = b1; ??
(Read)
{ x ↦ a1 ∗ y ↦ B } ⇝ { x ↦ B ∗ y ↦ a1 } | let b1 = *y; ??
(Read)
{ x ↦ A ∗ y ↦ B } ⇝ { x ↦ B ∗ y ↦ A } | let a1 = *x; ??


{ emp } ⇝ { emp } | skip
(Emp)
(Frame)
{ y ↦ a1 } ⇝ { y ↦ a1 } | ??
(Write)
{ y ↦ b1 } ⇝ { y ↦ a1 } | *y = a1; ??
(Frame)
{ x ↦ b1 * y ↦ b1 } ⇝ { x ↦ b1 * y ↦ a1 } | ??
(Write)
{ x ↦ a1 * y ↦ b1 } ⇝ { x ↦ b1 * y ↦ a1 } | *x = b1; ??
(Read)
{ x ↦ a1 * y ↦ B } ⇝ { x ↦ B * y ↦ a1 } | let b1 = *y; ??
(Read)
{ x ↦ A * y ↦ B } ⇝ { x ↦ B * y ↦ A } | let a1 = *x; ??

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

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

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

demo: tracing swap

synthetic separation logic (SSL)

- basic rules

  (Emp), (Read), (Write), (Frame)
  (Alloc), (Free)
- pure reasoning and unification
- inductive predicates and recursion

synthetic separation logic (SSL)

- basic rules

  (Emp), (Read), (Write), (Frame)
  (Alloc), (Free)
- pure reasoning and unification
- inductive predicates and recursion

example: dispose a list

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

{ list(x) }

{ emp }

{ $\text{list}^1(x)$ } **void** dispose(**loc** x) { emp }

{ $\text{list}^0(x)$ }

??

(Induction)

{ emp }

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

{ $list^1$ (x) } **void** dispose(**loc** x) { emp }

{ $list^0(x)$ }

??

(Open)

{ emp }

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

{ list^{1} (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) {
```
{ x = 0 ; emp }

```
??
```
{ emp }

```
} else {
```
{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ Y ∗ list^{1} (Y) }

```
??
```
{ emp }

```
}
```

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

{ list^1 (x) } **void** dispose(**loc** x) { emp }

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

{ x = 0 ; emp }

```
??
```

(Emp)

{ emp }

```
} else {
```

{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ Y ∗ list^1 (Y) }

```
??
```

{ emp }

```
}
```

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

{ list^{1} (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) {
```
{ x = 0 ; emp }

```
    skip
```
{ emp }

```
} else {
```
{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ Y ∗ list^{1} (Y) }

```
    ??
```

{ emp }

```
}
```

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

{ list^1 (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
```
{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ Y ∗ list^1 (Y) }

```
    ??
```

{ emp }

```
}
```

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

{ list^{1} (x) } **void** dispose(**loc** x) { emp }

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

{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ Y ∗ list^{1} (Y) }

```
    ??
```

(Read)

{ emp }

```
}
```

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

{ list^1 (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
```

{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ y1 ∗ list^1 (y1) }

```
    ??
```

{ emp }

```
}
```

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

{ list^1 (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
```

{ x ≠ 0 ; [x, 2] ∗ x ↦ V ∗ (x + 1) ↦ y1 ∗ list^1 (y1) }

```
    ??
```

(Free)

{ emp }

```
}
```

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

{ list^1 (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
    free x;
```

{ x ≠ 0 ; list^1 (y1) }

```
    ??
```

{ emp }

```
}
```

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

{ list^{1} (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
    free x;
```

{ x ≠ 0 ; list^{1} (y1) }

(Call)

```
    ??
```

{ emp }

```
}
```

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

{ $list^1$ (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
    free x;
    dispose(y1);
```

{ x ≠ 0 ; emp }

```
    ??
```

{ emp }

```
}
```

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

{ list^{1} (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
    free x;
    dispose(y1);
```

{ x ≠ 0 ; emp }

```
    ??
```

{ emp }

```
}
```

(Emp)

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

{ $list^1$ (x) } **void** dispose(**loc** x) { emp }

```
if (x == 0) { skip } else {
    let y1 = *(x + 1);
    free x;
    dispose(y1);
    skip


}
```

```
void dispose(loc x) {
  if (x == 0) { }
  else {
    let y1 = *(x + 1);
    free x;
    dispose(y1)
  }
}
```

synthetic separation logic (SSL)

- basic rules
  (Emp), (Read), (Write), (Frame), (Alloc), (Free)
- pure reasoning and unification
- inductive predicates and recursion
  (Open), (Close), (Induction), (Call)

this talk

1. example: swap
2. the logic
3. deductive synthesis
   - 3.1. proof system
   - 3.2. proof search
4. cyclic synthesis

SuSLik

best-first and-or search in SSL

+ optimizations

demo: search

sources of incompleteness

- pure synthesis is hard!

  $\{ r \mapsto \_ \} \rightsquigarrow \{ \psi(\text{inputs}, M) ; r \mapsto M \}$

  use an off-the-shelf pure synthesizer

- goal must be inductive in one predicate

  can't handle nontrivial termination metrics

    e.g. merge sorted lists

  can't generate recursive auxiliary functions

    e.g. sort, reverse, tree flatten

this talk

1. example: swap
2. the logic
3. deductive synthesis
4. cyclic synthesis

cyclic synthesis

SSL + cyclic proofs

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

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

dispose revisited

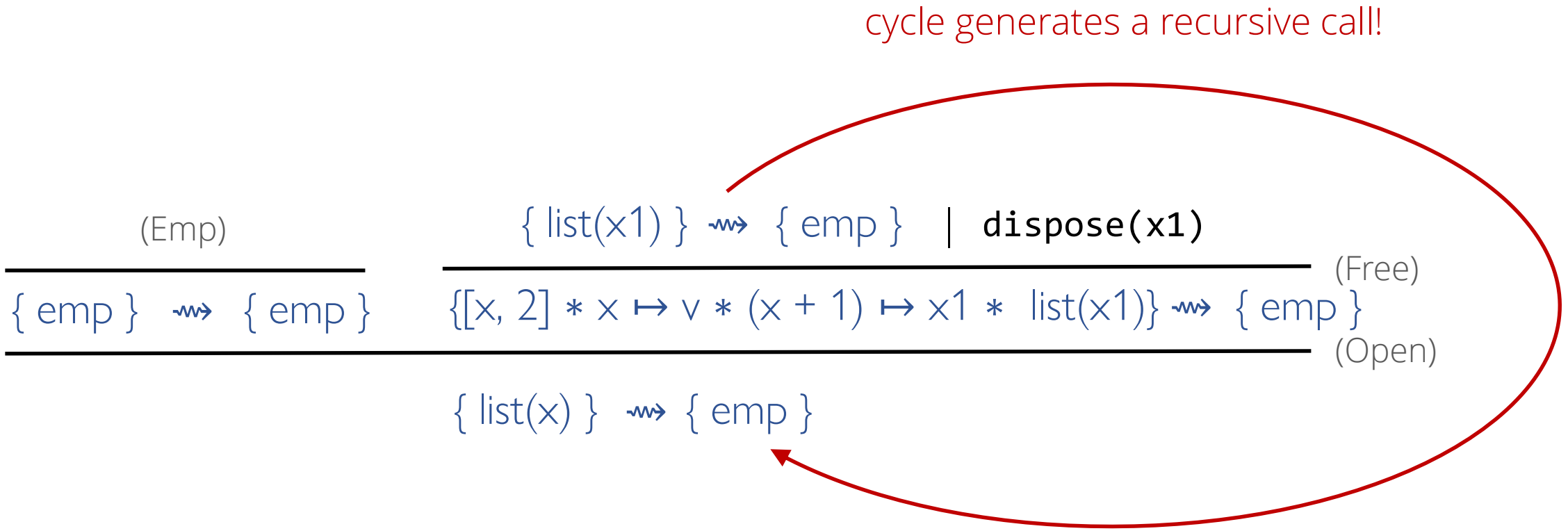
```
void dispose(loc x)
```

{ list(x) }

{ emp }

dispose: cyclic proof

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



dispose two

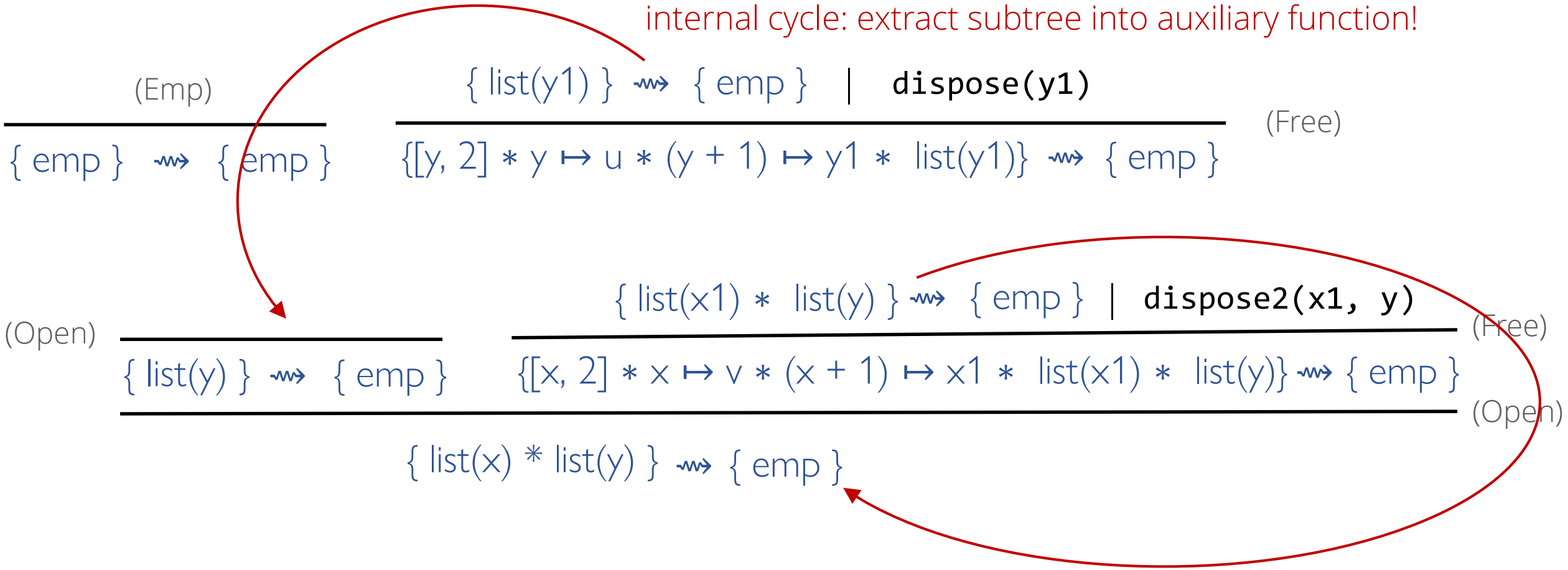
```
void dispose2(loc x, loc y)
```

{ list(x) * list(y) }

{ emp }

dispose two

$\{ \text{list}(x) * \text{list}(y) \}$ `dispose2(x, y)` $\{ \text{emp} \}$



synthesis with auxiliary functions

```
void dispose2(loc x, loc y) {
  if (x == 0) {
    dispose(y)
  } else {
    let x1 = *(x + 1);
    free x;
    dispose(x1, y)
  }
}
```

```
void dispose(loc y) {
  if (y == 0) {
  } else {
    let y1 = *(y + 1);
    free y;
    dispose(y1)
  }
}
```

example: flatten a tree

```
void flatten(loc t, loc r)
```

{ r ↦ 0 * tree(t) }

{ r ↦ x * list(x) }

example: flatten a tree

$\{r \mapsto 0 * \text{tree}(t)\}$ `flatten(t, r)` $\{r \mapsto x * \text{list}(x)\}$

```
void flatten(loc t, loc r) {
  if (t == 0) { } else {
    let v = *(t + 1);
    let t1 = *(t + 1);
    let t2 = *(t + 2);
    flatten(t1, r);
    let y = *r;
    flatten(t2, r);
    let z = *r;
    aux(y, z, v, r);
    free(t)
}
```

```
void aux(loc y, loc z, int v, loc r) {
  if (y == 0) {
    let x = malloc(2);
    *x = v;
    *(x + 1) = z;
    *r = x;
  } else {
    let y1 = *(y + 1);
    aux(y1, z, v, r);
    let z1 = *r;
    *(y + 1) = z1;
    *r = y;
  }
}
```

deductive synthesis with SuSLik

separation logic

☺ reasoning about pointers & aliasing

deductive synthesis

☺ uses specs to guide synthesis

code

☺ provably memory-safe