# #27: Separation Logic and Deductive Synthesis

**Sankha Narayan Guria**

EECS 700: Introduction to Program Synthesis

# Program synthesis with guarantees

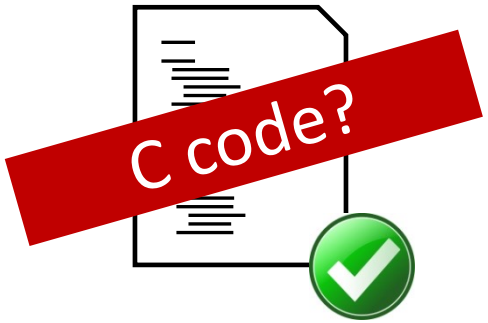
specification

code + proof

# Program synthesis with guarantees

specification

code + proof




☹ verbose
☹ unstructured
☹ pointers

# The trouble with pointers

- Can we naively apply Hoare logic to programs with pointers?

$\{* x = 10 \wedge * y = 10\}$

$\Rightarrow$

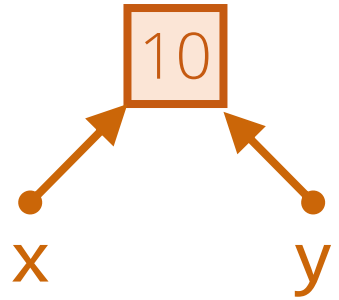
$\{(* x) + 5 = 15 \wedge (* y) - 5 = 5\}$

```
*x = *x + 5;
```

$\{* x = 15 \wedge (* y) - 5 = 5\}$

```
*y = *y - 5;
```

$\{* x = 15 \wedge * y = 5\}$



# SuSLik

Synthesis Using Separation Logik

# The SuSLik approach

separation logic

☺ reasoning about pointers & aliasing

deductive synthesis

☺ uses specs to guide synthesis

code

# Outline

1. example: swap
   a taste of SuSLik
2. separation logic
   specifying pointer-manipulating programs
3. deductive synthesis
   from SL specifications to programs

# Outline

1. example: swap
2. separation logic
3. deductive synthesis

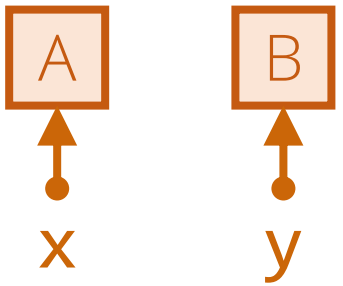
# Example: swap

Swap values of two *distinct* pointers

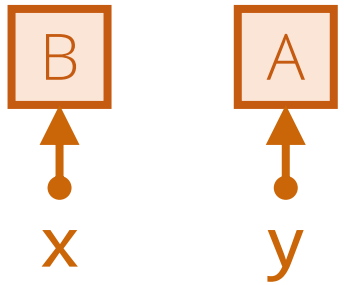
```
void swap(loc x, loc y)
```

# Example: swap

start state:




end state:




in separation logic:

precondition

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

separately

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

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

postcondition

logical variables

# Demo: swap

Swap values of two *distinct* pointers

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

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

??

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

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

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

??

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

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

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

??

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

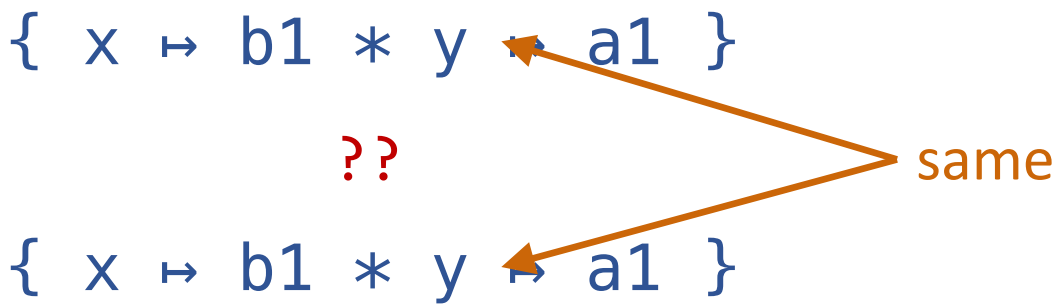
```
let a1 = *x;
let b1 = *y;
*x = b1;
```

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

??

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

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



```
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;
}
```

# Outline

1. example: swap

2. separation logic

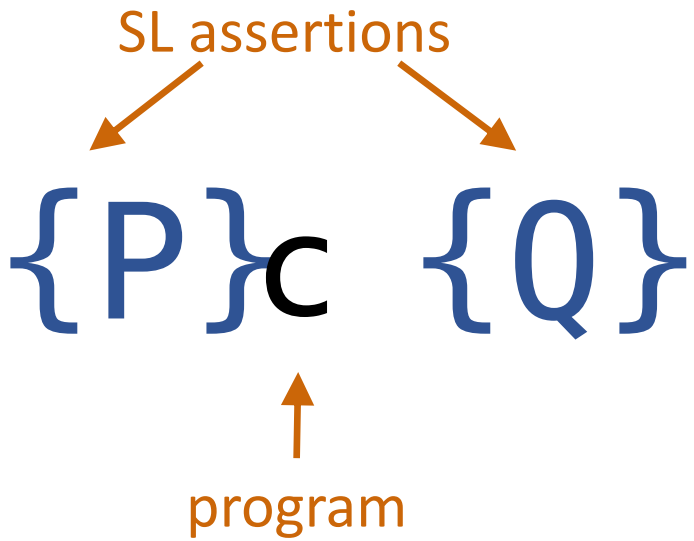
3. deductive synthesis

# Separation logic (SL)

Hoare logic

"about the heap"

# Separation logic (SL)




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

# Outline

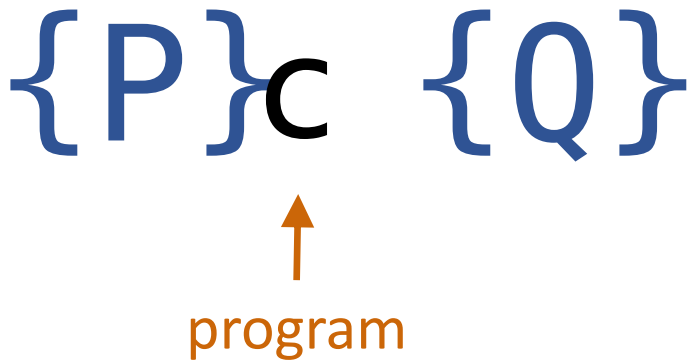
1. example: swap

# 2. separation logic

   2.1. programs
   2.2. assertions
   2.3. specifying data transformations

3. deductive synthesis

# Separation logic (SL)



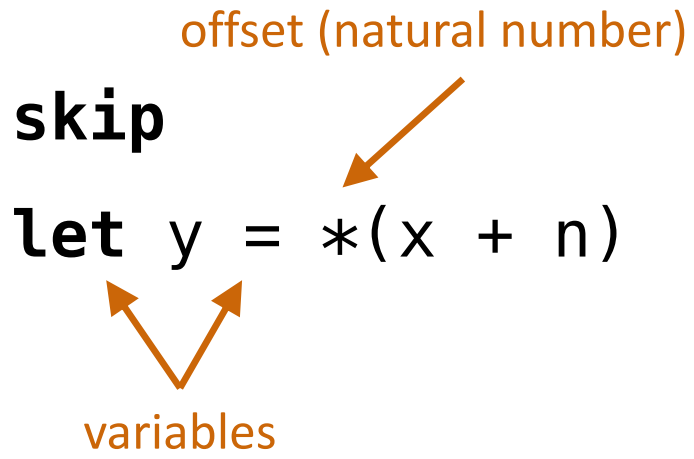
# Programs

| | |
|---|---|
| do nothing | **`skip`** |

# Programs

do nothing

read from heap



# 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($e_1$, …, $e_n$) |

# 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 {$c_2$} |

# Outline

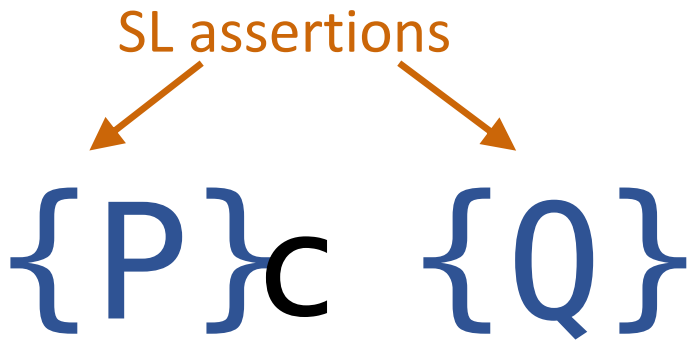
1. example: swap

## 2. separation logic

- 2.1. programs
- 2.2. assertions
- 2.3. specifying data transformations

3. deductive synthesis

# Separation logic (SL)



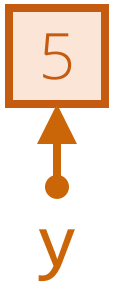
# SL assertions

empty heap { emp }

# SL assertions

empty heap $\{ \; \mathsf{emp} \; \}$

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



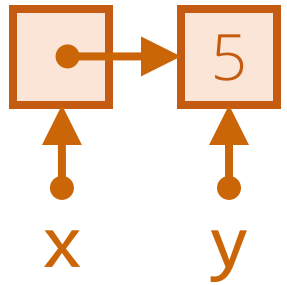
# SL assertions

empty heap $\{ \; emp \; \}$

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

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

heaplets



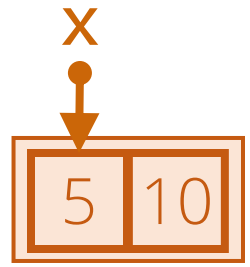
# SL assertions

empty heap

$$\{\ \mathsf{emp}\ \}$$

singleton heap

$$\{\ y \mapsto 5\ \}$$




separating conjunction

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

memory block

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

# SL assertions

empty heap

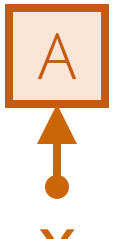
$$\{\ emp\ \}$$

singleton heap

$$\{\ y \mapsto 5\ \}$$

separating conjunction

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



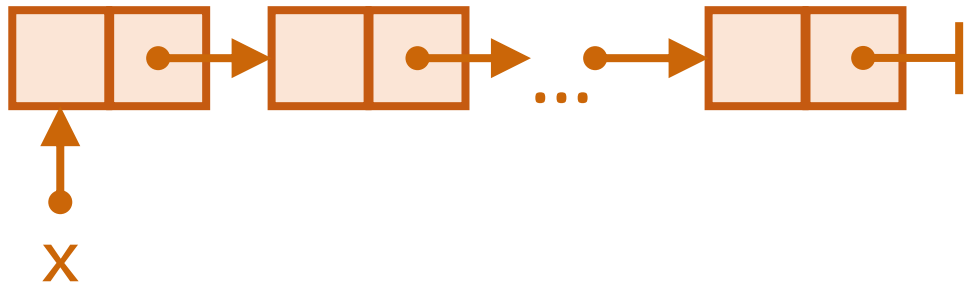

memory block

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

+ pure formula

$$\{\ A > 5\ ;\ x \mapsto A\ \}$$

# SL assertions: linked structures

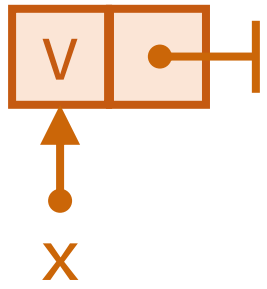


# SL assertions: linked structures

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

# SL assertions: linked structures

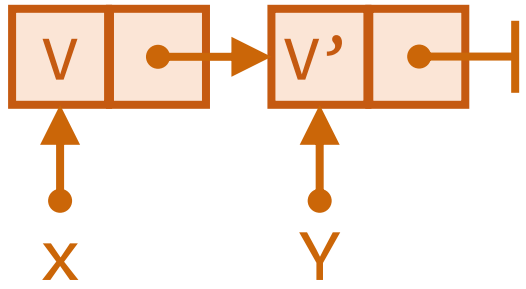
linked list $\{\ [x, 2] * x \mapsto V * (x + 1) \mapsto 0\ \}$



# SL assertions: linked structures

linked list

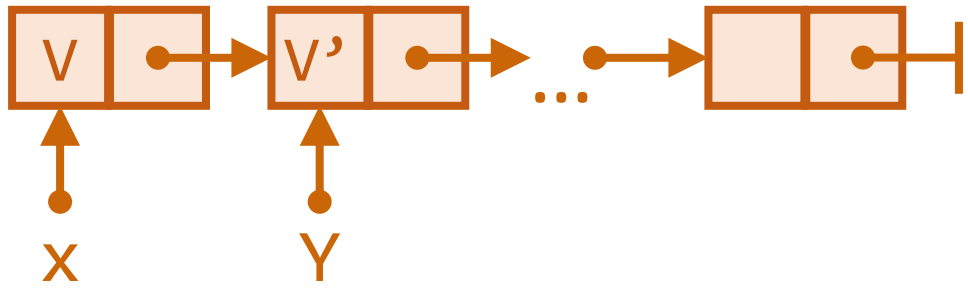
$$\{\ [x, 2] * x \mapsto V * (x + 1) \mapsto Y *$$
$$[Y, 2] * Y \mapsto V' * (Y + 1) \mapsto 0$$
$$\}$$



# SL assertions: linked structures

linked list

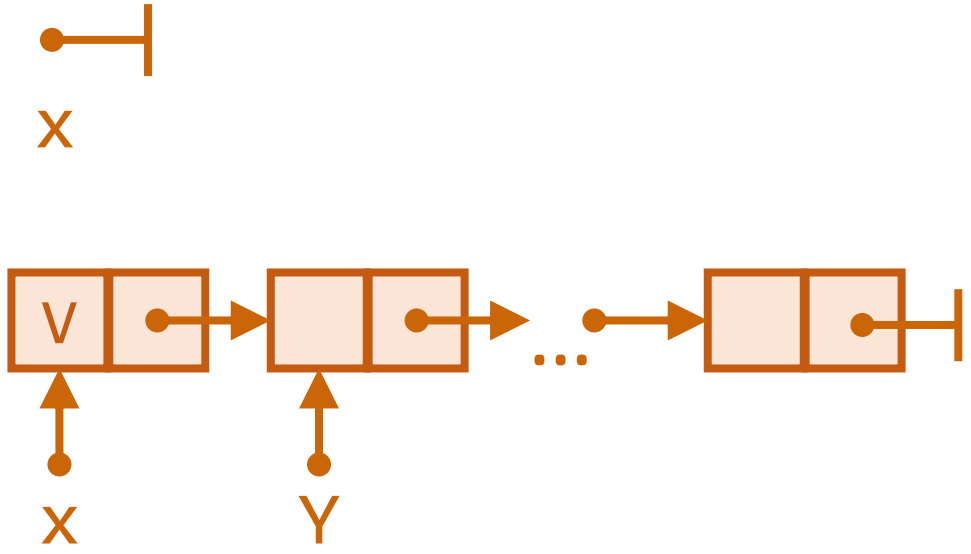
$$\{\ [x, 2] * x \mapsto V * (x + 1) \mapsto Y *$$
$$[Y, 2] * Y \mapsto V' * (Y + 1) \mapsto Y' *$$
$$\ldots$$
$$\}$$




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)
                }
}
```



# Outline

1. example: swap

## 2. separation logic

- 2.1. programs
- 2.2. assertions
- 2.3. specifying data transformations

3. deductive synthesis

# Example: dispose a list

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

# Example: copy a list

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

return location

# Outline

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

# Deductive synthesis

synthesis as proof search

# Outline

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

# 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

$$\{emp\} \twoheadrightarrow \{emp\} \mid ??$$

(Emp)

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

(Frame)

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

(Write)

$$\frac{\{\ x \mapsto e \ast P\ \} \twoheadrightarrow \{\ x \mapsto e \ast Q\ \} \mid c}{\{\ x \mapsto \_ \ast P\ \} \twoheadrightarrow \{\ x \mapsto e \ast Q\ \}\,??}$$

(Read)

$$\frac{[y/A]\{\ x \mapsto A \ast P\ \} \twoheadrightarrow [y/A]\{\ Q\ \} \quad | \ c}{\{\ x \mapsto A \ast P\ \} \twoheadrightarrow \{\ Q\ \} \quad ?? \quad ; \ c}$$

# SSL: basic rules

(Emp)

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

(Read)

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

(Frame)

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

(Write)

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

# Example: swap

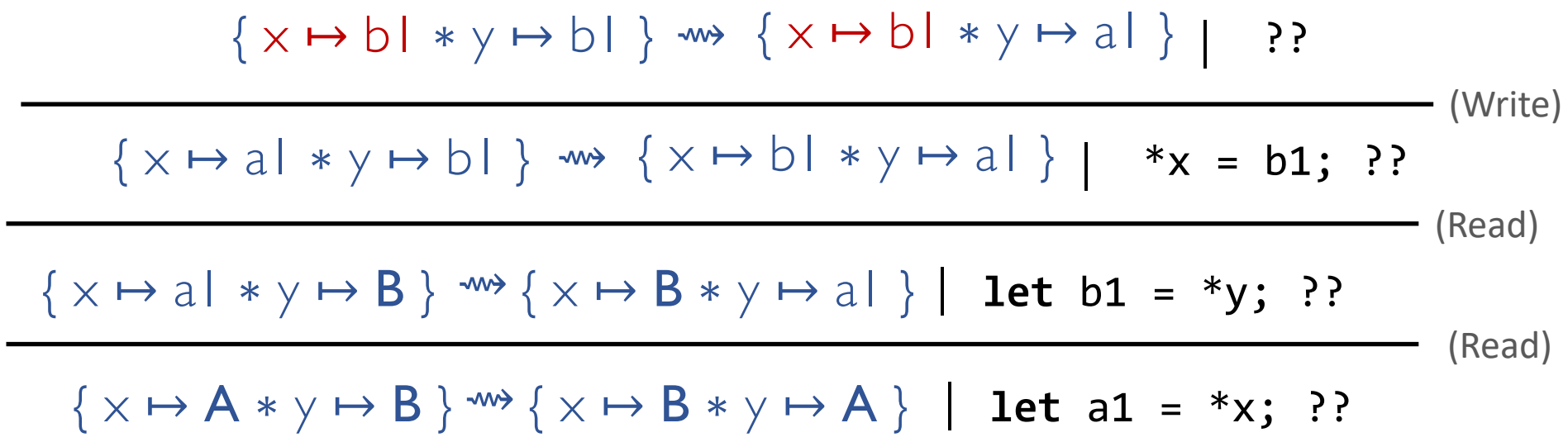
$$\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \quad ??$$

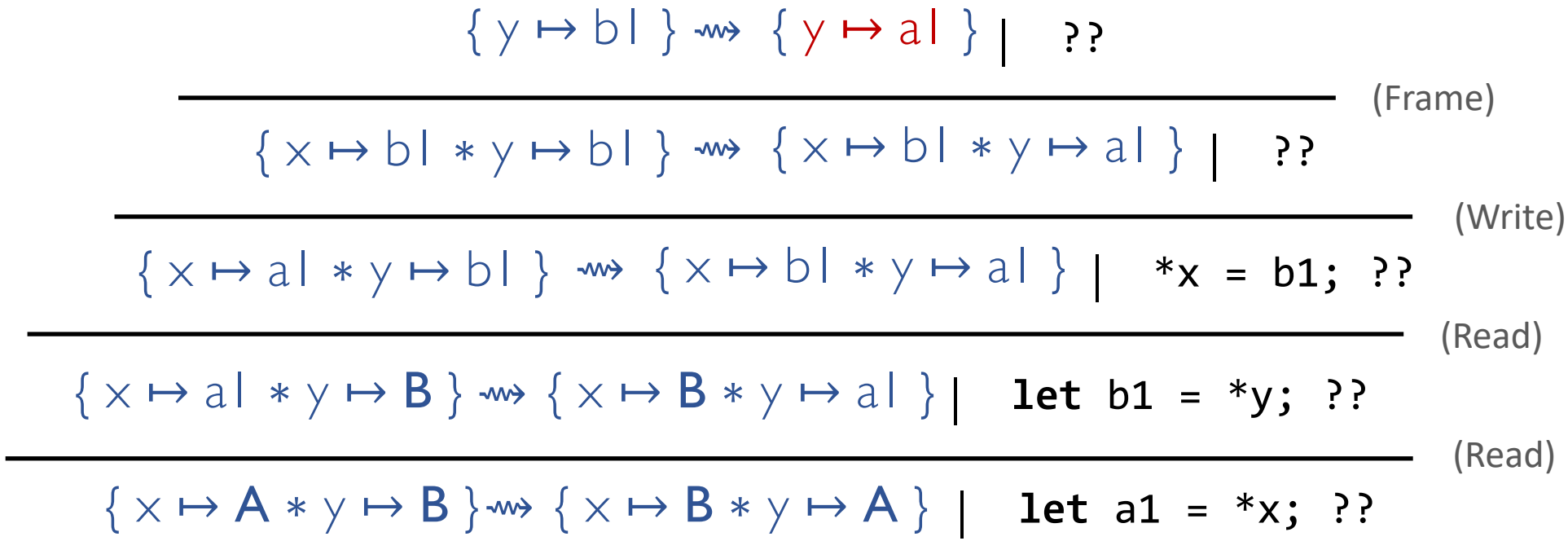
$$\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \quad ??$$

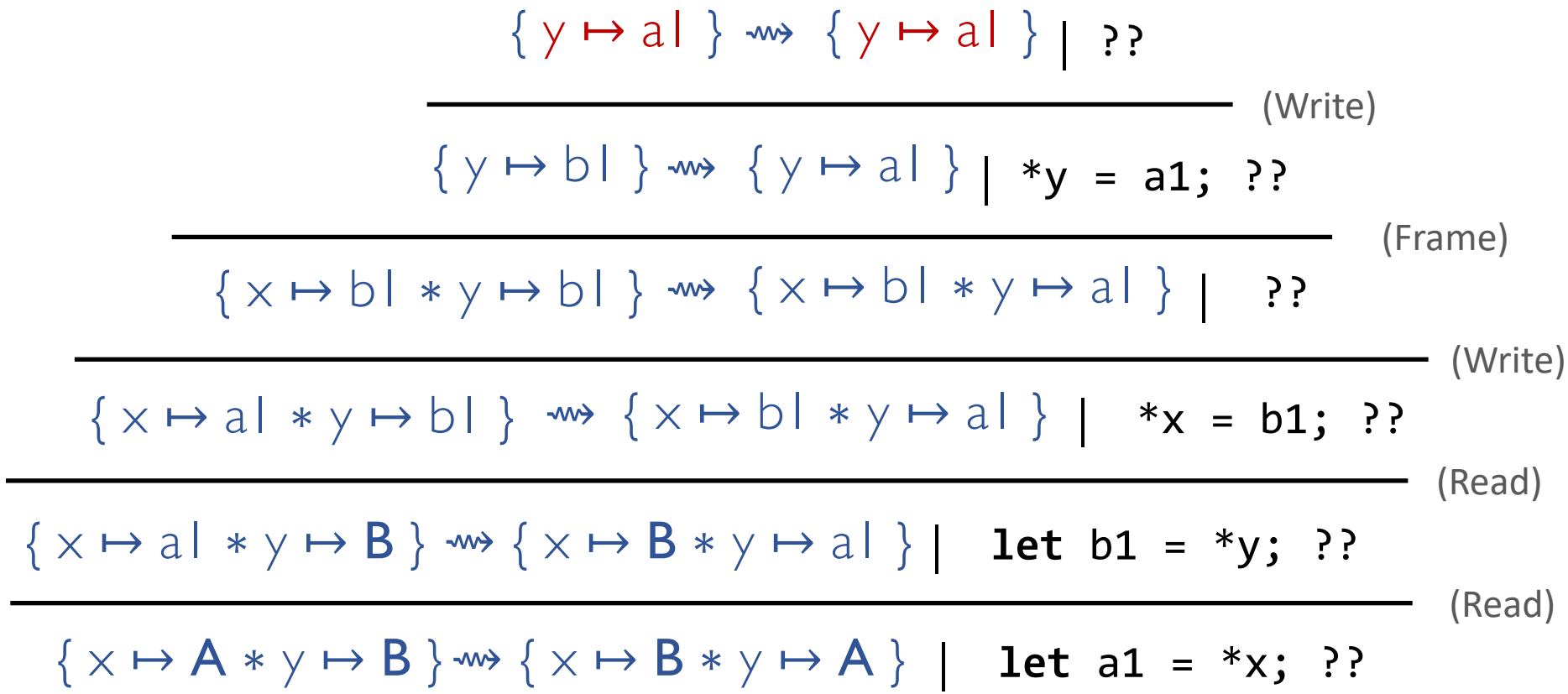
$$\frac{\{\, x \mapsto a1 \ast y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B \ast y \mapsto a1 \,\} \mid \; ??}{\{\, x \mapsto A \ast y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B \ast y \mapsto A \,\} \mid \; \texttt{let a1 = *x; ??}} \text{ (Read)}$$

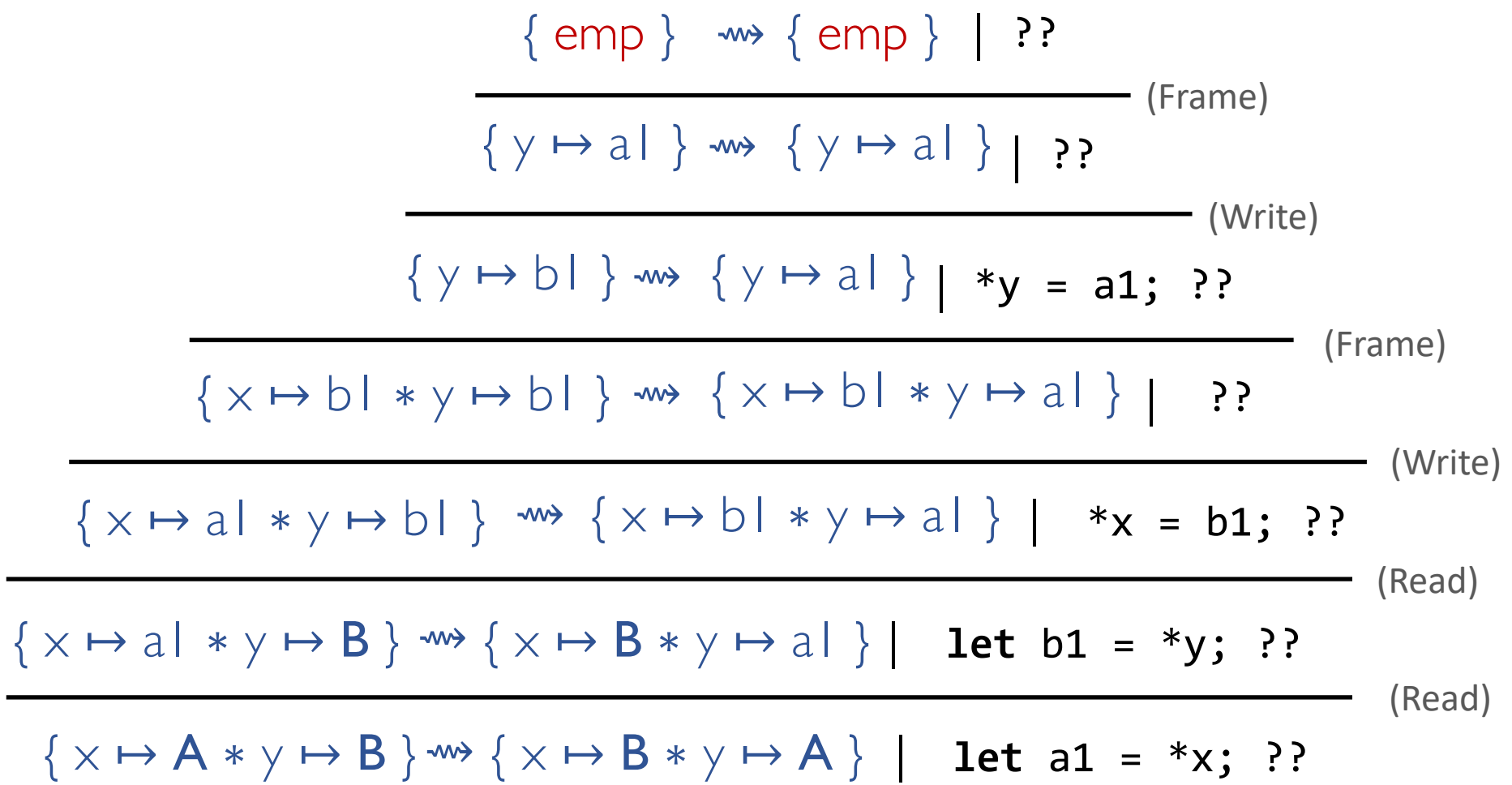
$$\frac{\{\, x \mapsto a1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \; ??}{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \; \texttt{let b1 = *y; ??}}\ \text{(Read)}$$

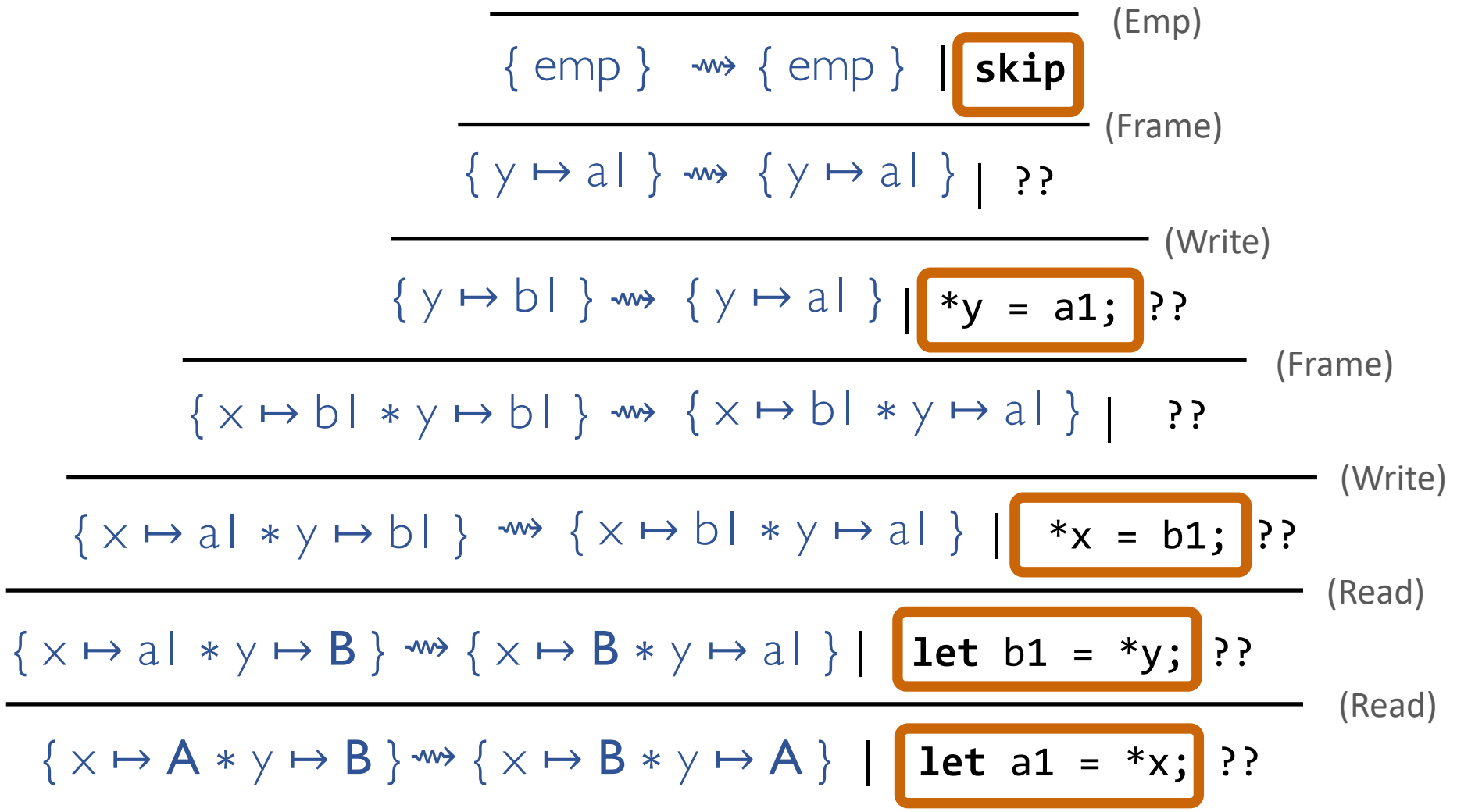
$$\frac{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \; \texttt{let b1 = *y; ??}}{\{\, x \mapsto A * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto A \,\} \mid \; \texttt{let a1 = *x; ??}}\ \text{(Read)}$$


{ 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; ??


{ 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; ??


{ 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 } | ??
(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
```

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