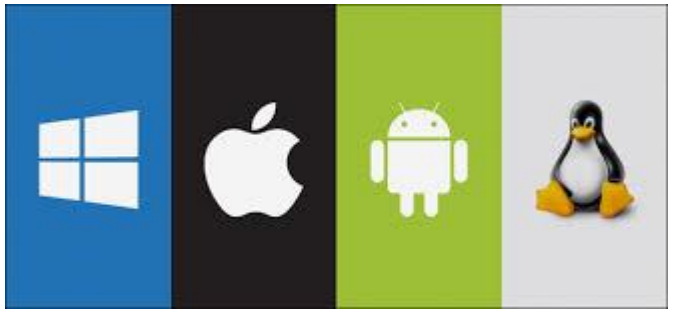
synthesis of safe pointer-manipulating programs

Nadia Polikarpova
joint work with Ilya Sergey (Yale-NUS)

UCSDCSE
Computer Science and Engineering

pointer-manipulating programs



operating systems

network / security protocols

browsers

☺ efficient

☹ hard to write

☹ memory safety bugs

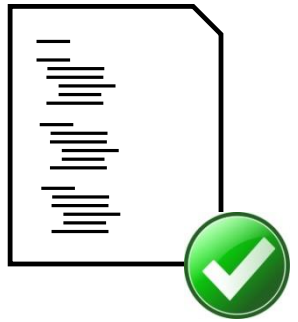
program synthesis to the rescue

specification

☺ easy to write

☹ verbose
☹ unstructured
☹ pointers & aliasing

code



☺ efficient
☺ backwards compatible
☺ provably memory-safe

our solution

separation logic

deductive synthesis

code

☺ reasoning about pointers & aliasing

☺ uses specs to guide synthesis

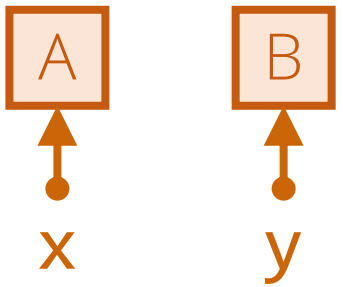
example: swap

Swap values of two *distinct* pointers

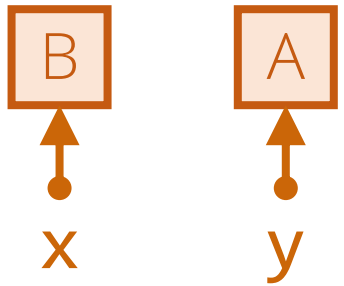
```
void swap(loc x, loc y)
```

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

example: swap

specification

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

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

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

code

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

insight: the spec tells us what to do!

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

??

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

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

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

??

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

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

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

??

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

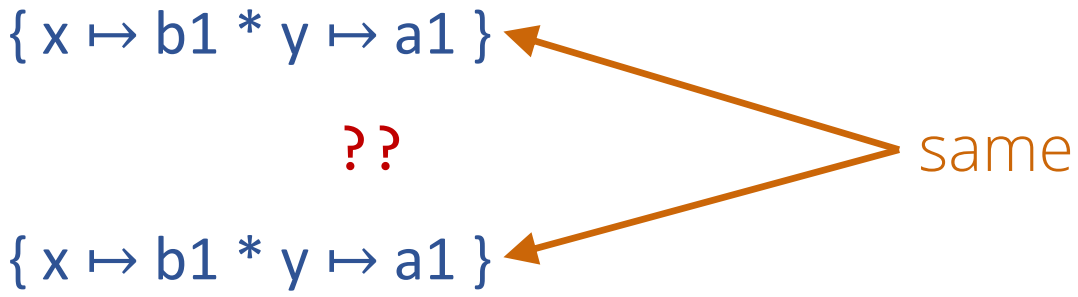
```
let a1 = *x;
let b1 = *y;
*x = b1;
```

{ 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

I. separation logic
   reasoning about pointer-manipulating programs (prior work)

II. deductive synthesis
   from SL specifications to programs

III. the SuSLik synthesizer
   results and future work

this talk

I. separation logic

II. deductive synthesis

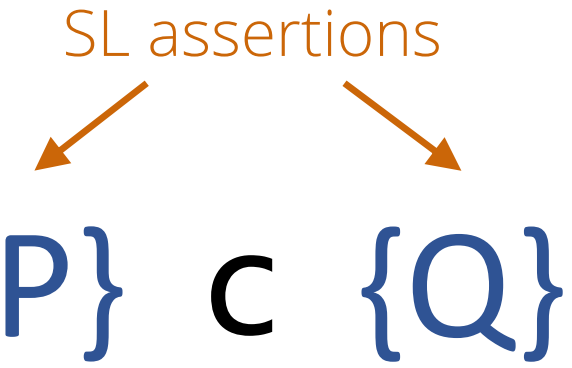
III. the SuSLik synthesizer

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

separation logic (SL)

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

$\{ x \mapsto A \}$ `*x = 5` $\{ x \mapsto 5 \}$ 

$\{ x \mapsto A \}$ `*(x + 1) = 5` $\{ (x + 1) \mapsto 5 \}$ 

SL assertions

empty heap { emp }

SL assertions

empty heap { emp }

singleton heap { y ↦ 5 }

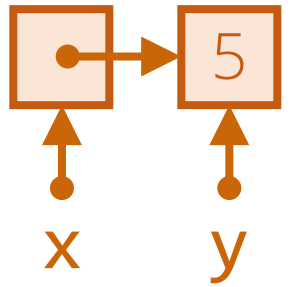
SL assertions

empty heap { emp }

singleton heap { y ↦ 5 }

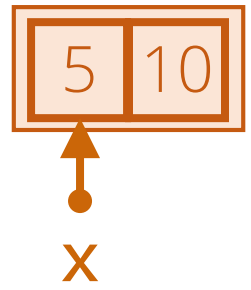
separating conjunction { x ↦ y * y ↦ 5 }

heaplets



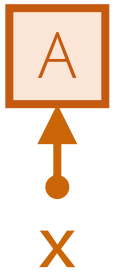
SL assertions

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

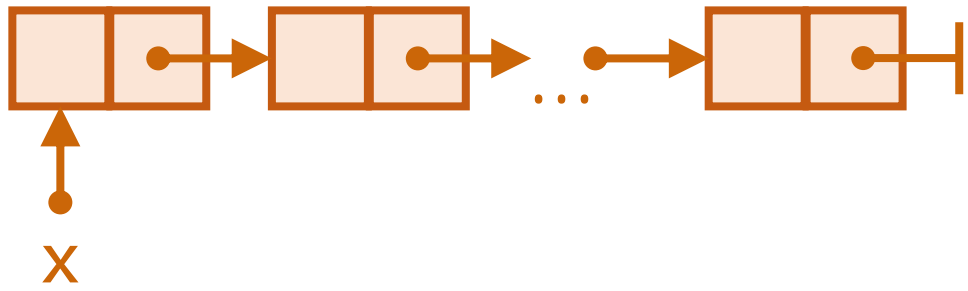


SL assertions

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

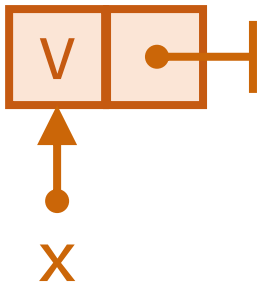


SL assertions: linked structures

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

SL assertions: linked structures

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

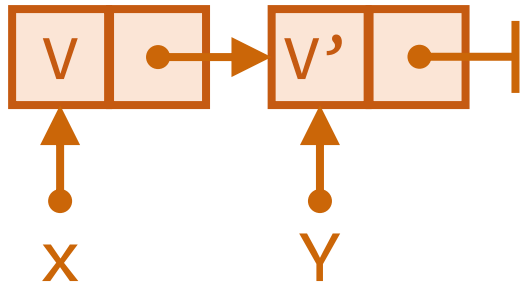


SL assertions: linked structures

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

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

$\}$



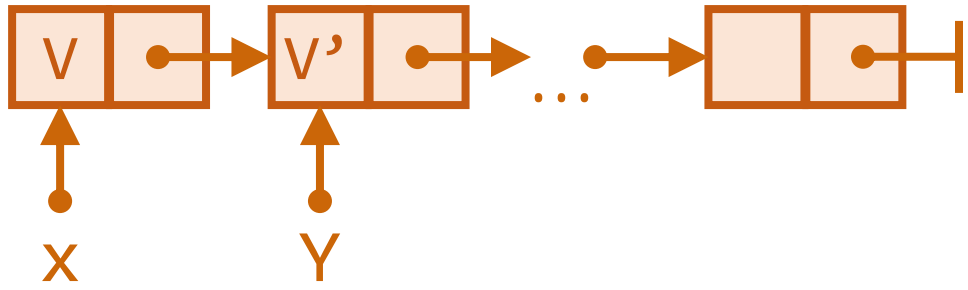
SL assertions: linked structures

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

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

…

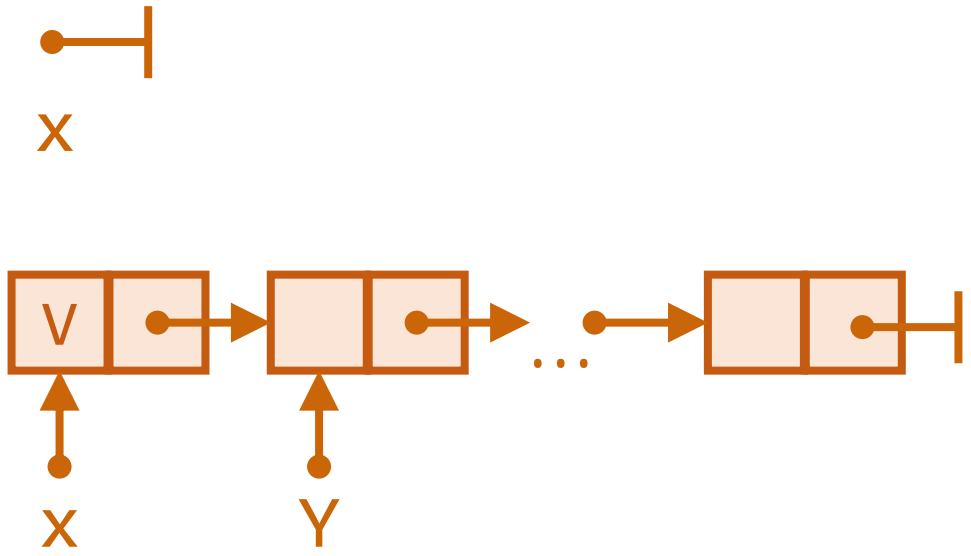
$\}$



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)
                }
}
```



example: dispose a list

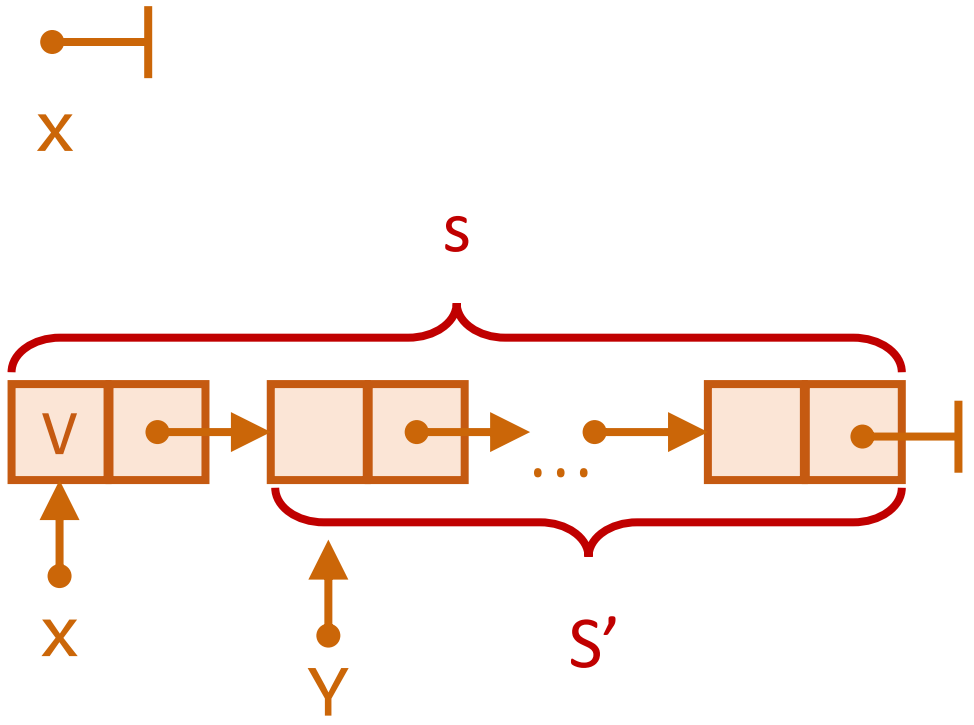
{ list(x) }

```
void dispose(loc 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') }
}
```



example: copy a list

{ list(x, S) * r ↦ _ }

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

{ list(x, S) * r ↦ Y * list(Y, S) }

example: single to double

{ list(x, S) * r ↦ _ }

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

{r ↦ Y * dlist(Y, P, S) }

doubly-linked list

list with reserved space in each node

{ list_reserved(x, S) }

```
void to_dll_in_place(loc x)
```

{dlist(x, P, S) }

this talk

I. separation logic

II. deductive synthesis

III. the SuSLik synthesizer

deductive synthesis

synthesis as proof search

proof of what?

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)

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

(Frame)

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

(Write)

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

(Read)

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

SSL: basic rules

(Emp)

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

(Read)

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

(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

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

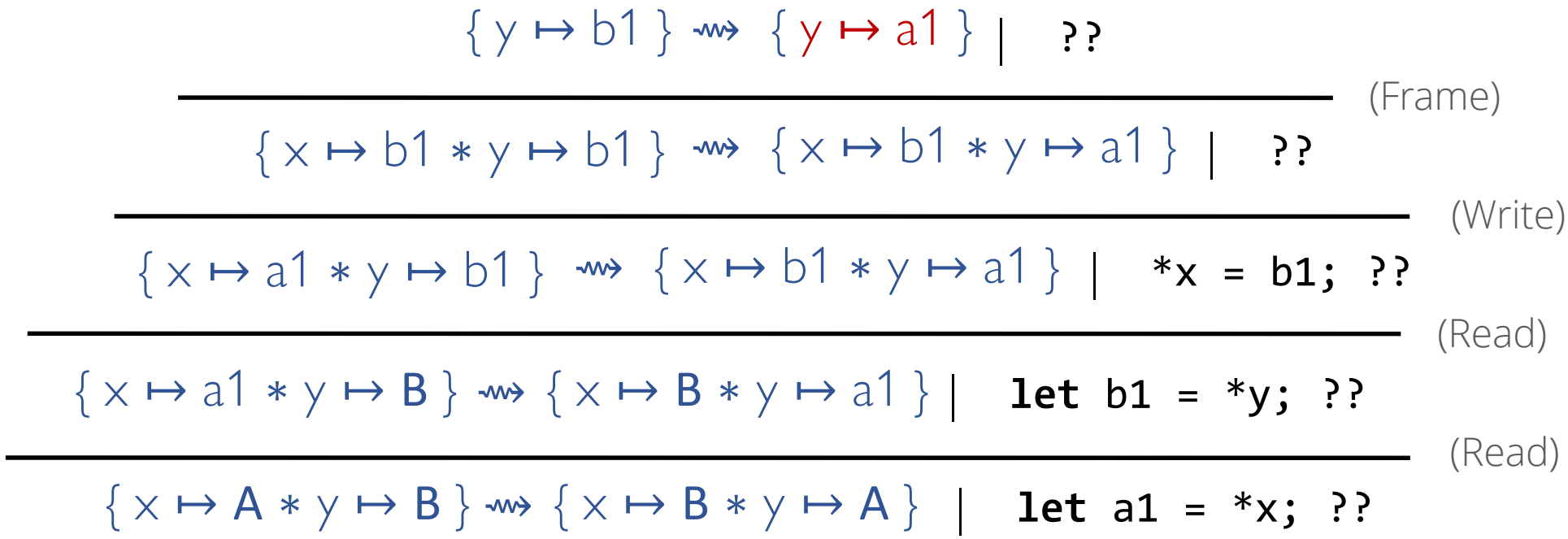
$$\{ x \mapsto A * y \mapsto B \} \rightsquigarrow \{ x \mapsto B * y \mapsto A \} \mid \quad ??$$

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

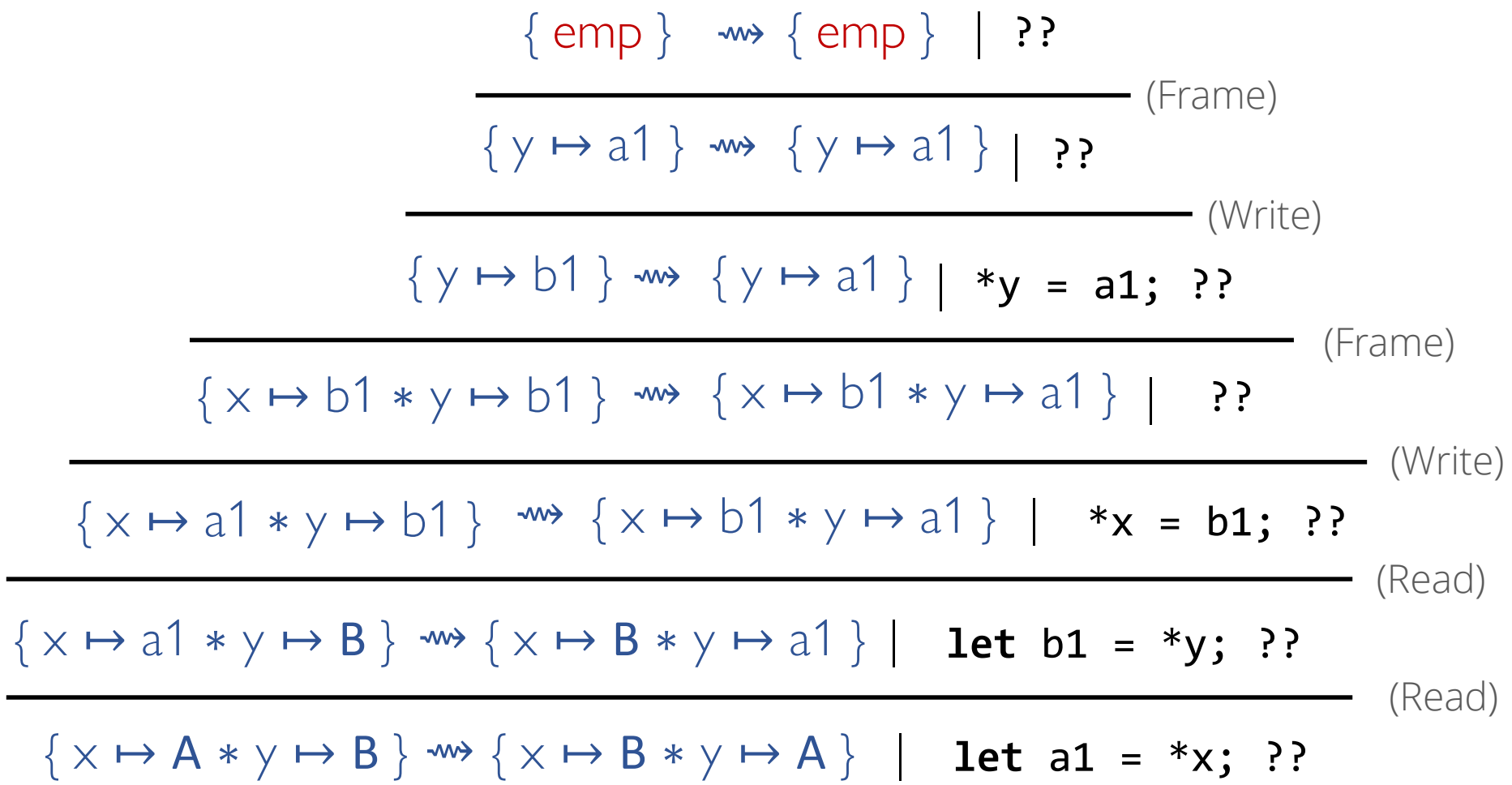
$$\frac{\{\, x \mapsto a1 * y \mapsto b1 \,\} \rightsquigarrow \{\, x \mapsto b1 * y \mapsto a1 \,\} \mid \texttt{??}}{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \texttt{let b1 = *y; ??}}\ \text{(Read)}$$

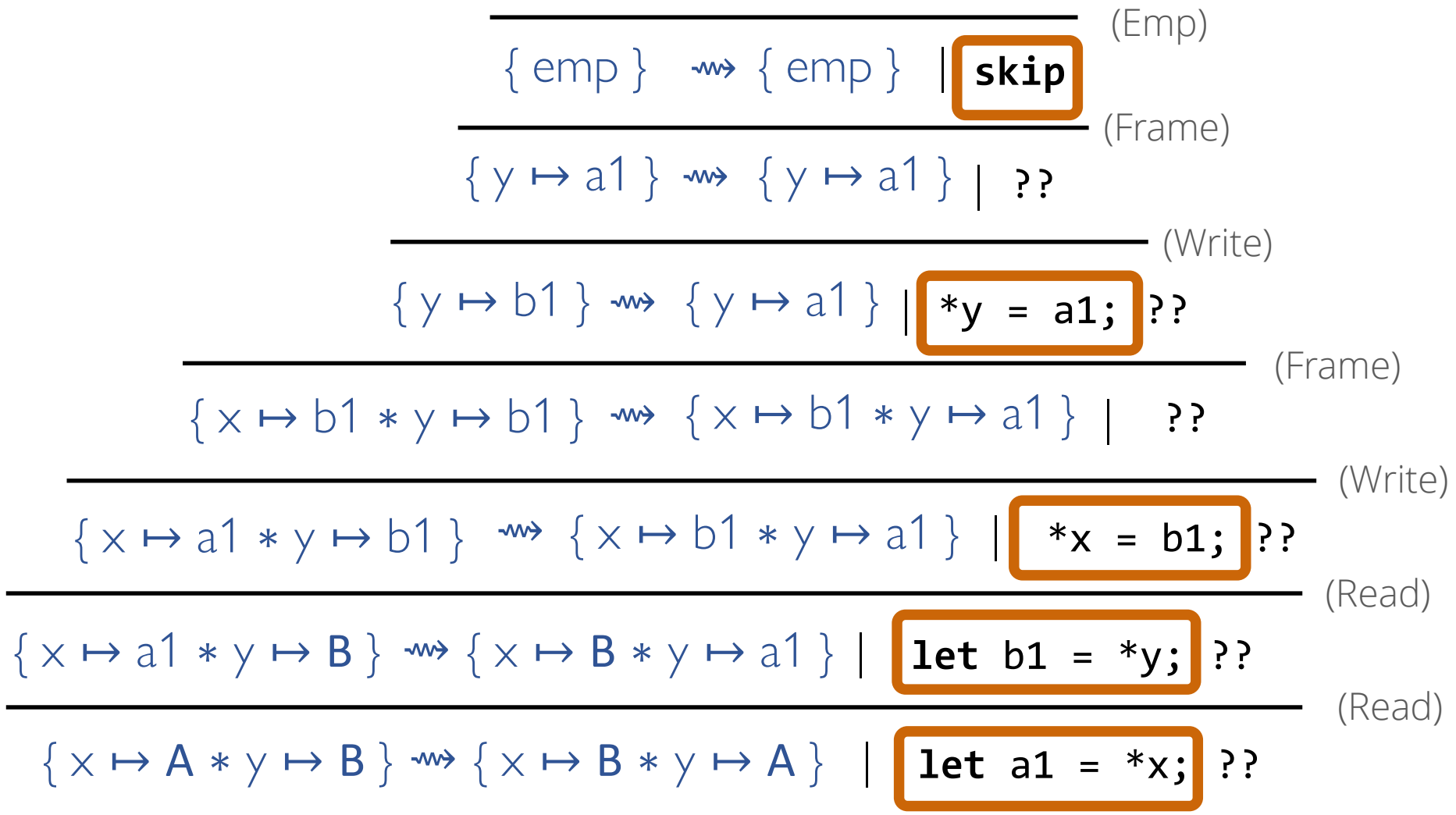
$$\frac{\{\, x \mapsto a1 * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto a1 \,\} \mid \texttt{let b1 = *y; ??}}{\{\, x \mapsto A * y \mapsto B \,\} \rightsquigarrow \{\, x \mapsto B * y \mapsto A \,\} \mid \texttt{let a1 = *x; ??}}\ \text{(Read)}$$

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


{ 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
```

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

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

{ list(x) }

```
void dispose(loc 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)

guarantees

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

and c terminates

this talk

I. separation logic

II. deductive synthesis

III. the SuSLik synthesizer

SuSLik

(Synthesis using Separation Logik)

SuSLik

backtracking search in SSL
+ optimizations

Group	*Description*	*Code*	*Code/Spec*	*Time*	*T-phase*	*T-inv*	*T-fail*	*T-com*	*T-all*	*T-IS*
Integers	swap two	12	0.9x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	min of two[2]	10	0.7x	0.1	0.1	0.1	< 0.1	0.1	0.2	
Linked List	length[1,2]	21	1.2x	0.4	0.9	0.5	0.4	0.6	1.4	29x
	max[1]	27	1.7x	0.6	0.8	0.5	0.4	0.4	0.8	20x
	min[1]	27	1.7x	0.5	0.9	0.5	0.4	0.5	1.2	49x
	singleton[2]	11	0.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	dispose	11	2.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	initialize	13	1.4x	< 0.1	0.1	0.1	< 0.1	0.1	< 0.1	
	copy[3]	35	2.5x	0.2	0.3	0.3	0.1	0.2	-	
	append[3]	19	1.1x	0.2	0.3	0.3	0.2	0.3	0.7	
	delete[3]	44	2.6x	0.7	0.5	0.3	0.2	0.3	0.7	
Sorted list	prepend[1]	11	0.3x	0.2	1.4	83.5	0.1	0.1	-	48x
	insert[1]	58	1.2x	4.8	-	-	-	5.0	-	6x
	insertion sort[1]	28	1.3x	1.1	1.8	1.3	1.2	1.2	74.2	82x
Tree	size	38	2.7x	0.2	0.3	0.2	0.2	0.2	0.3	
	dispose	16	4.0x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	copy	55	3.9x	0.4	49.8	-	0.8	1.4	-	
	flatten w/append	48	4.0x	0.4	0.6	0.5	0.4	0.4	0.6	
	flatten w/acc	35	1.9x	0.6	1.7	0.7	0.5	0.6	-	
BST	insert[1]	58	1.2x	31.9	-	-	-	-	-	11x
	rotate left[1]	15	0.1x	37.7	-	-	-	-	-	0.5x
	rotate right[1]	15	0.1x	17.2	-	-	-	-	-	0.8x

[1] From (Qiu and Solar-Lezama 2017) [2] From (Leino and Milicevic 2012) [3] From (Qiu et al. 2013)

Group	*Description*	*Code*	*Code/Spec*	*Time*	*T-phase*	*T-inv*	*T-fail*	*T-com*	*T-all*	*T-IS*
Integers	swap two	12	0.9x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	min of two[2]	10	0.7x	0.1	0.1	0.1	< 0.1	0.1	0.2	
Linked List	length[1,2]	21	1.2x	0.4	0.9	0.5	0.4	0.6	1.4	29x
	max[1]	27	1.7x	0.6	0.8	0.5	0.4	0.4	0.8	20x
	min[1]	27	1.7x	0.5	0.9	0.5	0.4	0.5	1.2	49x
	singleton[2]	11	0.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	dispose	11	2.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	initialize	13	1.4x	< 0.1	0.1	0.1	< 0.1	0.1	< 0.1	
	copy[3]	35	2.5x	0.2	0.3	0.3	0.1	0.2	-	
	append[3]	19	1.1x	0.2	0.3	0.3	0.2	0.3	0.7	
	delete[3]	44	2.6x	0.7	0.5	0.3	0.2	0.3	0.7	
Sorted list	prepend[1]	11	0.3x	0.2	1.4	83.5	0.1	0.1	-	48x
	insert[1]	58	1.2x	4.8	-	-	-	5.0	-	6x
	insertion sort[1]	28	1.3x	1.1	1.8	1.3	1.2	1.2	74.2	82x
Tree	size	38	2.7x	0.2	0.3	0.2	0.2	0.2	0.3	
	dispose	16	4.0x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	copy	55	3.9x	0.4	49.8	-	0.8	1.4	-	
	flatten w/append	48	4.0x	0.4	0.6	0.5	0.4	0.4	0.6	
	flatten w/acc	35	1.9x	0.6	1.7	0.7	0.5	0.6	-	
BST	insert[1]	58	1.2x	31.9	-	-	-	-	-	11x
	rotate left[1]	15	0.1x	37.7	-	-	-	-	-	0.5x
	rotate right[1]	15	0.1x	17.2	-	-	-	-	-	0.8x

[1] From (Qiu and Solar-Lezama 2017) [2] From (Leino and Milicevic 2012) [3] From (Qiu et al. 2013)

Group	*Description*	*Code*	*Code/Spec*	*Time*	*T-phase*	*T-inv*	*T-fail*	*T-com*	*T-all*	*T-IS*
Integers	swap two	12	0.9x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	min of two[2]	10	0.7x	0.1	0.1	0.1	< 0.1	0.1	0.2	
Linked List	length[1,2]	21	1.2x	0.4	0.9	0.5	0.4	0.6	1.4	29x
	max[1]	27	1.7x	0.6	0.8	0.5	0.4	0.4	0.8	20x
	min[1]	27	1.7x	0.5	0.9	0.5	0.4	0.5	1.2	49x
	singleton[2]	11	0.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	dispose	11	2.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	initialize	13	1.4x	< 0.1	0.1	0.1	< 0.1	0.1	< 0.1	
	copy[3]	35	2.5x	0.2	0.3	0.3	0.1	0.2	-	
	append[3]	19	1.1x	0.2	0.3	0.3	0.2	0.3	0.7	
	delete[3]	44	2.6x	0.7	0.5	0.3	0.2	0.3	0.7	
Sorted list	prepend[1]	11	0.3x	0.2	1.4	83.5	0.1	0.1	-	48x
	insert[1]	58	1.2x	4.8	-	-	-	5.0	-	6x
	insertion sort[1]	28	1.3x	1.1	1.8	1.3	1.2	1.2	74.2	82x
Tree	size	38	2.7x	0.2	0.3	0.2	0.2	0.2	0.3	
	dispose	16	4.0x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	copy	55	3.9x	0.4	49.8	-	0.8	1.4	-	
	flatten w/append	48	4.0x	0.4	0.6	0.5	0.4	0.4	0.6	
	flatten w/acc	35	1.9x	0.6	1.7	0.7	0.5	0.6	-	
BST	insert[1]	58	1.2x	31.9	-	-	-	-	-	11x
	rotate left[1]	15	0.1x	37.7	-	-	-	-	-	0.5x
	rotate right[1]	15	0.1x	17.2	-	-	-	-	-	0.8x

[1] From (Qiu and Solar-Lezama 2017) [2] From (Leino and Milicevic 2012) [3] From (Qiu et al. 2013)

Group	*Description*	*Code*	*Code/Spec*	*Time*	*T-phase*	*T-inv*	*T-fail*	*T-com*	*T-all*	*T-IS*
Integers	swap two	12	0.9x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	min of two[2]	10	0.7x	0.1	0.1	0.1	< 0.1	0.1	0.2	
Linked List	length[1,2]	21	1.2x	0.4	0.9	0.5	0.4	0.6	1.4	29x
	max[1]	27	1.7x	0.6	0.8	0.5	0.4	0.4	0.8	20x
	min[1]	27	1.7x	0.5	0.9	0.5	0.4	0.5	1.2	49x
	singleton[2]	11	0.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	dispose	11	2.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	initialize	13	1.4x	< 0.1	0.1	0.1	< 0.1	0.1	< 0.1	
	copy[3]	35	2.5x	0.2	0.3	0.3	0.1	0.2	-	
	append[3]	19	1.1x	0.2	0.3	0.3	0.2	0.3	0.7	
	delete[3]	44	2.6x	0.7	0.5	0.3	0.2	0.3	0.7	
Sorted list	prepend[1]	11	0.3x	0.2	1.4	83.5	0.1	0.1	-	48x
	insert[1]	58	1.2x	4.8	-	-	-	5.0	-	6x
	insertion sort[1]	28	1.3x	1.1	1.8	1.3	1.2	1.2	74.2	82x
Tree	size	38	2.7x	0.2	0.3	0.2	0.2	0.2	0.3	
	dispose	16	4.0x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	copy	55	3.9x	0.4	49.8	-	0.8	1.4	-	
	flatten w/append	48	4.0x	0.4	0.6	0.5	0.4	0.4	0.6	
	flatten w/acc	35	1.9x	0.6	1.7	0.7	0.5	0.6	-	
BST	insert[1]	58	1.2x	31.9	-	-	-	-	-	11x
	rotate left[1]	15	0.1x	37.7	-	-	-	-	-	0.5x
	rotate right[1]	15	0.1x	17.2	-	-	-	-	-	0.8x

[1] From (Qiu and Solar-Lezama 2017) [2] From (Leino and Milicevic 2012) [3] From (Qiu et al. 2013)

limitations & future work

- pure synthesis

  use an off-the-shelf pure synthesizer

- reasoning about inductive predicates

  identify decidable fragment

- goal must be inductive

  cyclic proofs to the rescue!

$\{ r \mapsto \_ \} \rightsquigarrow \{ r \geq x \wedge r \geq y ; r \mapsto M \}$ | `c`

$\{ P \} \rightsquigarrow \{ Q \}$ | `skip`

$\{ tree(t, S) * r \mapsto \_ \}$

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

$\{ r \mapsto X * list(X, S) \}$

deductive synthesis with SuSLik

separation logic

☺ reasoning about pointers & aliasing

deductive synthesis

☺ uses specs to guide synthesis

code

☺ provably memory-safe

backup slides

optimizations

- invertible rules
- early failure
- multi-phase search
- symmetry reduction

optimization: invertible rules

- invertible rules do not restrict the set of derivable programs
- idea: invertible rules need not be backtracked

(Read)

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

optimization: early failure

- idea: sometimes you know that a goal is unsatisfiable

(Post-Inconsistent)

$$\frac{\psi \neq \bot \qquad \vdash \phi \wedge \psi \Rightarrow \bot \qquad \{\ \mathsf{emp}\ \} \rightsquigarrow \{\ \bot\ ;\ \mathsf{emp}\ \} \mid c}{\{\ \phi\ ;\ P\ \} \rightsquigarrow \{\ \psi\ ;Q\ \} \mid c}$$

optimization: multi-phase search

- unfolding phase: deals with inductive predicates
  (Open), (Close), (Call), (Frame)
- flat phase: deals with points-to and blocks
  (Write), (Call), (Alloc), (Free), (Frame)
- idea: if unfolding phase cannot eliminate all predicates, give up

optimization: symmetry reduction

- sometimes rule applications commute

$$\dfrac{\dfrac{\{\, a \mapsto 0 \,\} \rightsquigarrow \{\, b \mapsto 0 \,\} \mid ??}{\{\, y \mapsto b * a \mapsto 0 \,\} \rightsquigarrow \{\, y \mapsto b * b \mapsto 0 \,\} \mid ??}\ \text{(Frame)}}{\{\, x \mapsto a * y \mapsto b * a \mapsto 0 \,\} \rightsquigarrow \{\, x \mapsto a * y \mapsto b * b \mapsto 0 \,\} \mid ??}\ \text{(Frame)}$$

- idea: only allow one order of commuting applications

example: flatten a tree

```
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, r);
    …
}
```

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