Synthesis of Programs with Pointers via Read-Only Specifications

Andreea Costea *Amy Zhu* *Nadia Polikarpova* *Ilya Sergey*

NUS National University of Singapore School of Computing

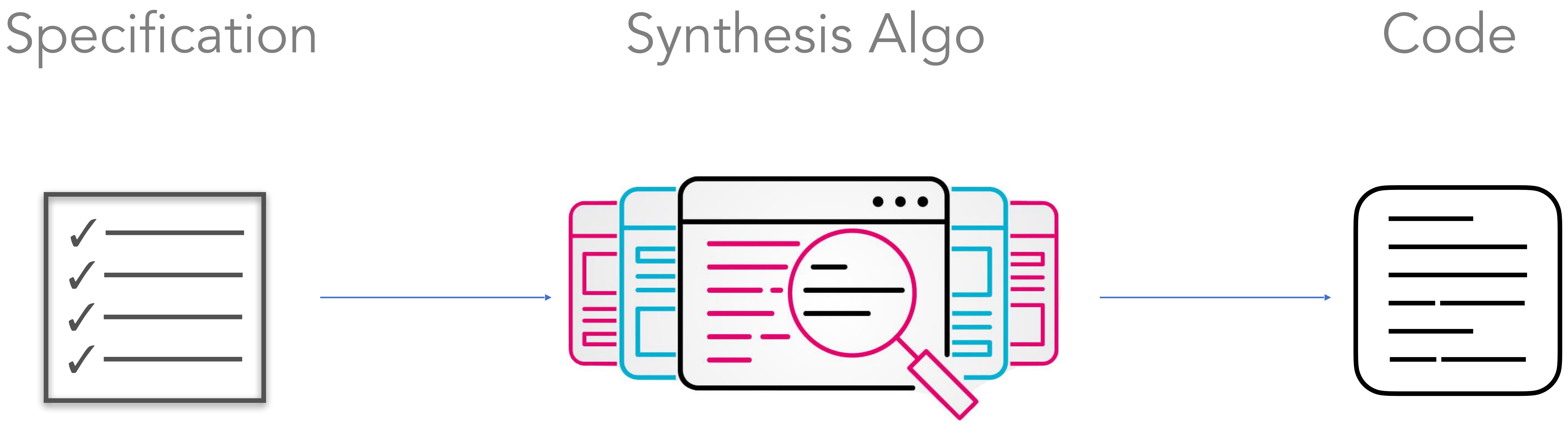
PLSE

University of California San Diego

Yale NUS College

Synthesis of Programs with Pointers

via Read-Only Specifications

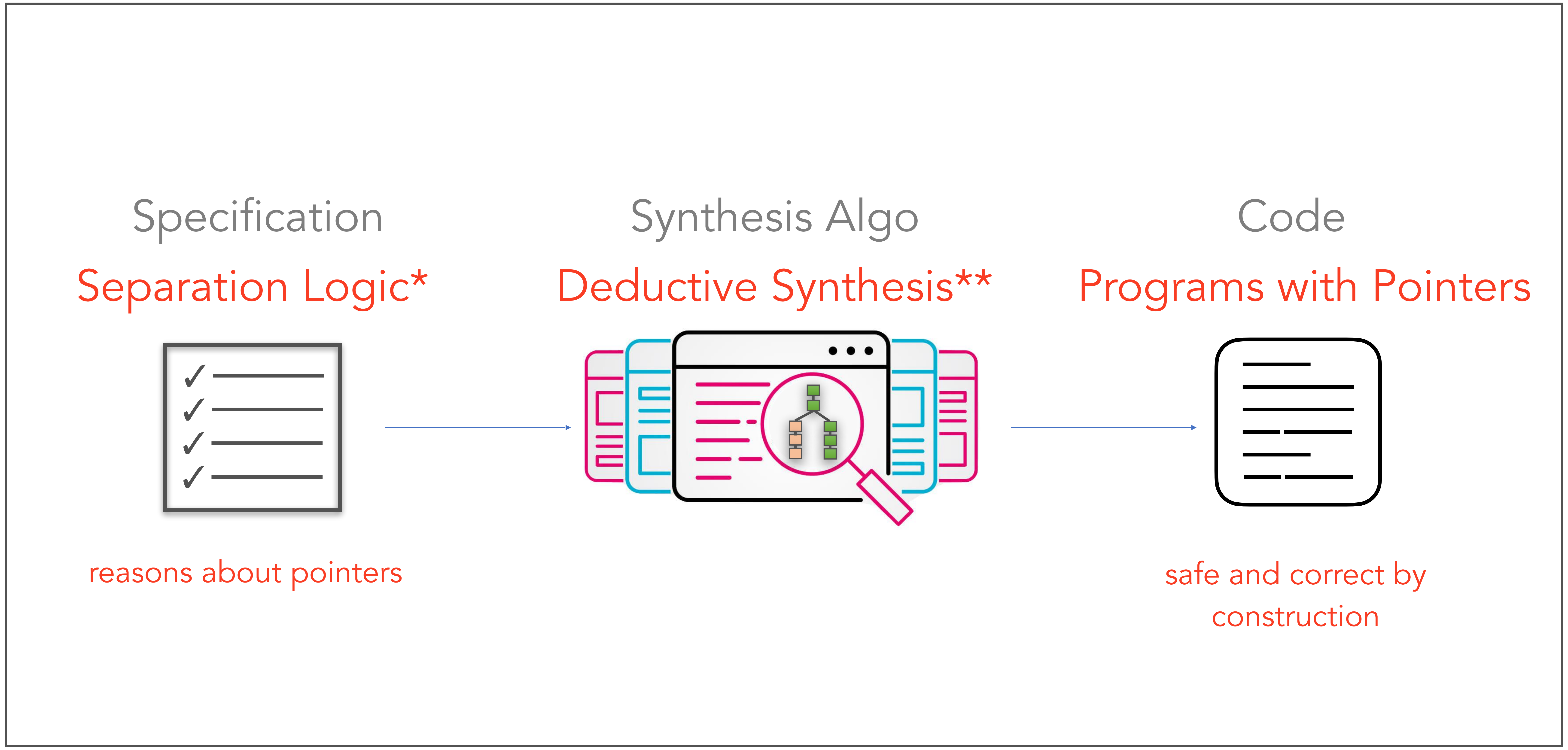


Specification

Synthesis Algo

Code

SSL: Synthetic Separation Logic



Specification

Separation Logic*

reasons about pointers

Synthesis Algo

Deductive Synthesis**

Code

Programs with Pointers

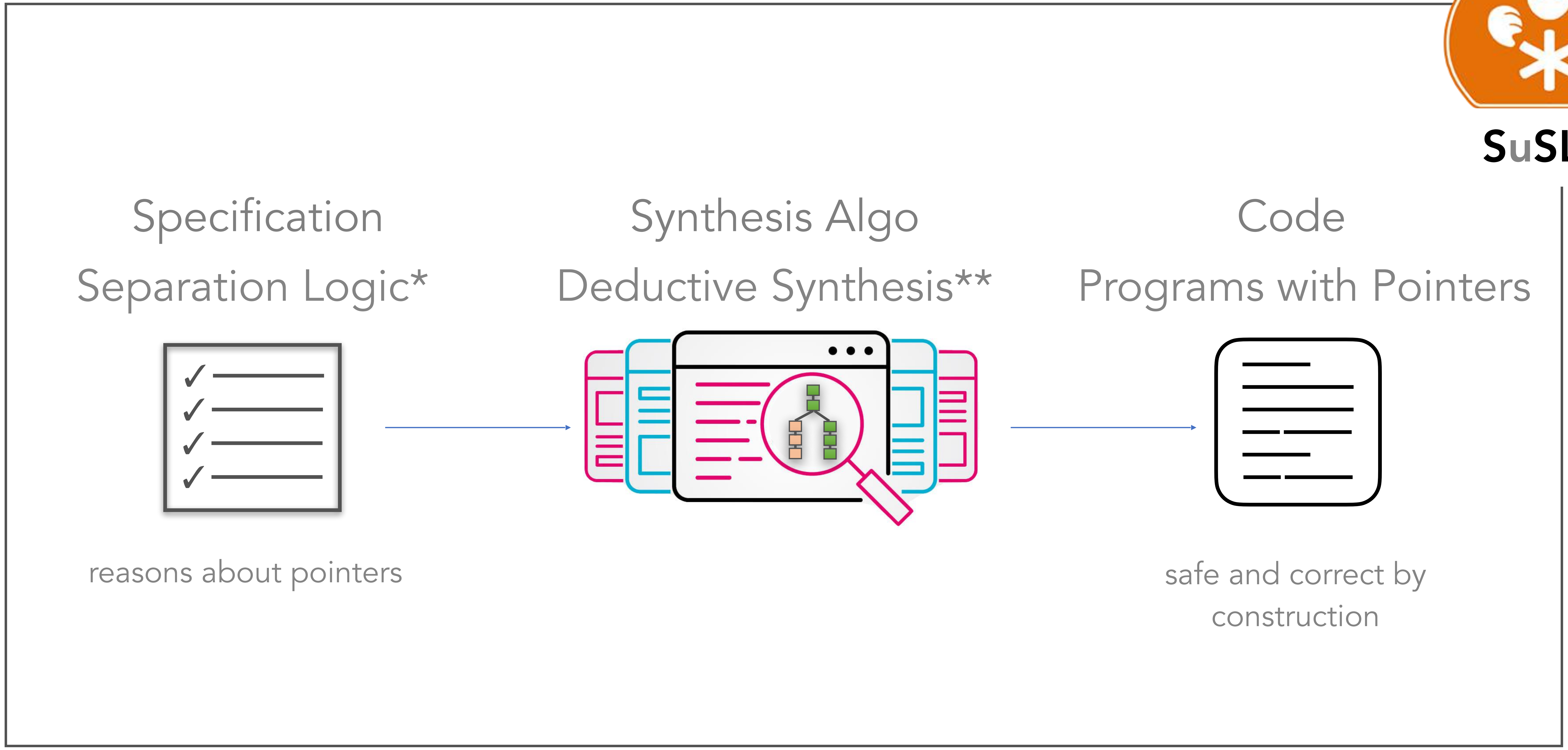
safe and correct by construction

* *Local Reasoning about Programs that Alter Data Structures*, O'Hearn, Reynolds, Yang:. CSL 2001

** *Structuring the Synthesis of Heap-Manipulating Programs*, Polikarpova & Sergey @POPL'19

SSL: Synthetic Separation Logic

SuSLik



Specification

Separation Logic*

reasons about pointers

Synthesis Algo

Deductive Synthesis**

Code

Programs with Pointers

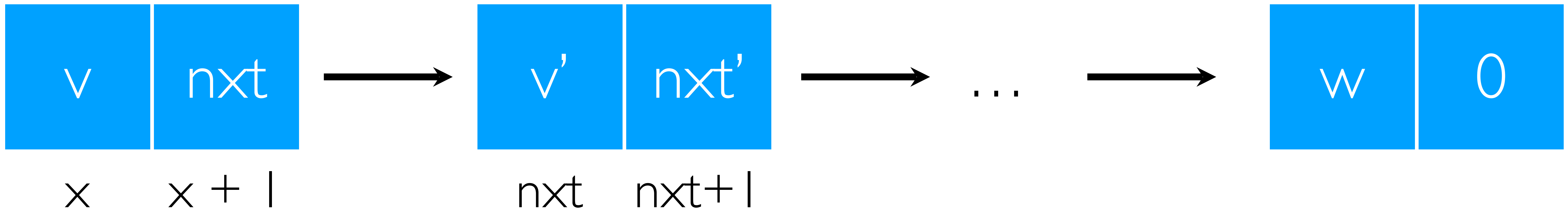
safe and correct by construction

* *Local Reasoning about Programs that Alter Data Structures*, O'Hearn, Reynolds, Yang:. CSL 2001

** *Structuring the Synthesis of Heap-Manipulating Programs*, Polikarpova & Sergey @POPL'19

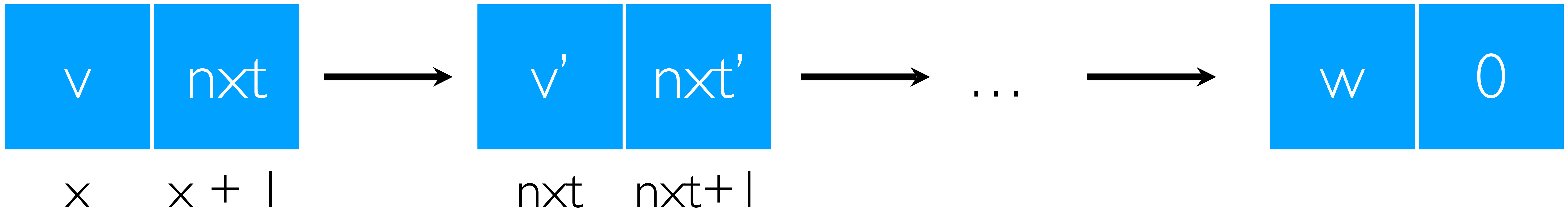
Example: copy a linked list

Example: copy a **linked list**



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

Example: copy a **linked list**

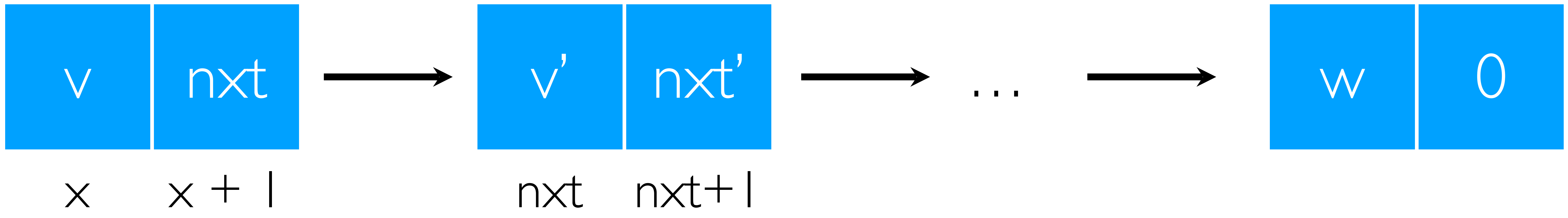


v nxt → v' nxt' → … → w 0

x x + 1 nxt nxt+1

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

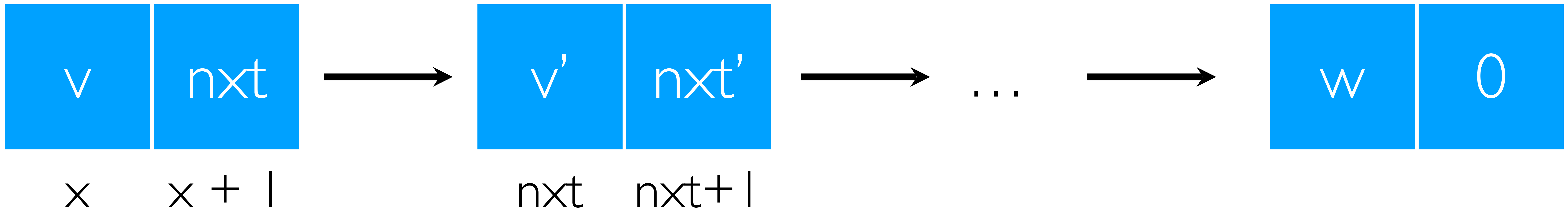
Example: copy a **linked list**



pure constraints

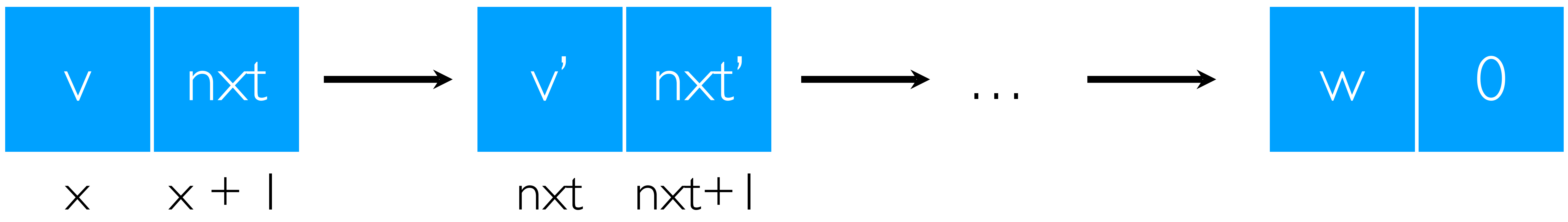
```
predicate ls (loc x, set S) {
   | x = 0  ∧  { S = ∅          ; emp }
   | x ≠ 0 ∧  { S = {v} ∪ S’   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ nxt ∗ ls(nxt, S’) }
}
```

Example: copy a **linked list**



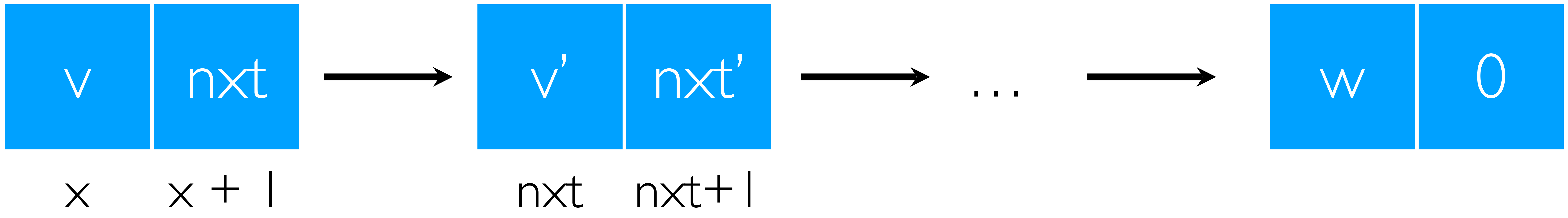
```
predicate ls (loc x, set S) {
   | x = 0  ∧  { S = ∅             ; emp }
   | x ≠ 0 ∧  { S = {v} ∪ S’   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ nxt ∗ ls(nxt, S’) }
}
```

Example: copy a **linked list**



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

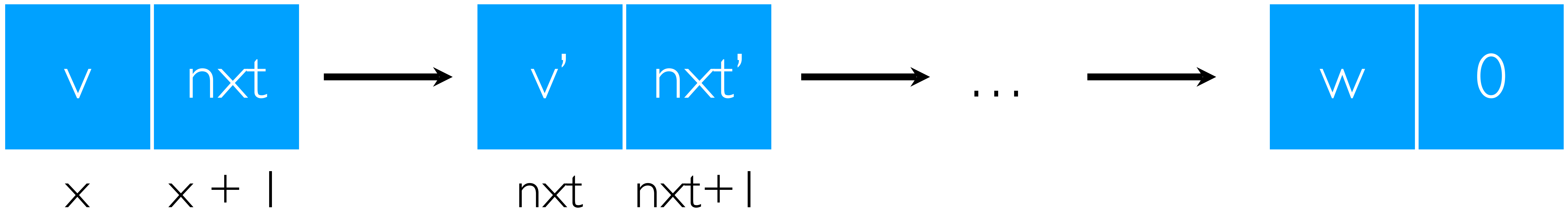
Example: copy a **linked list**



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

memory block

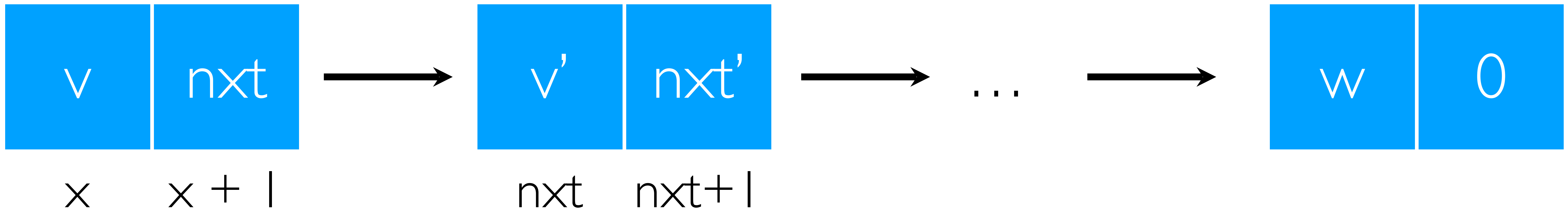
Example: copy a **linked list**



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

points-to

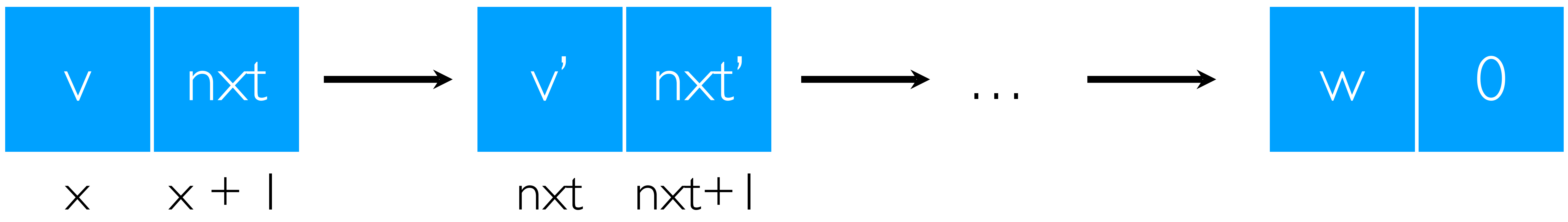
Example: copy a **linked list**



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

points-to

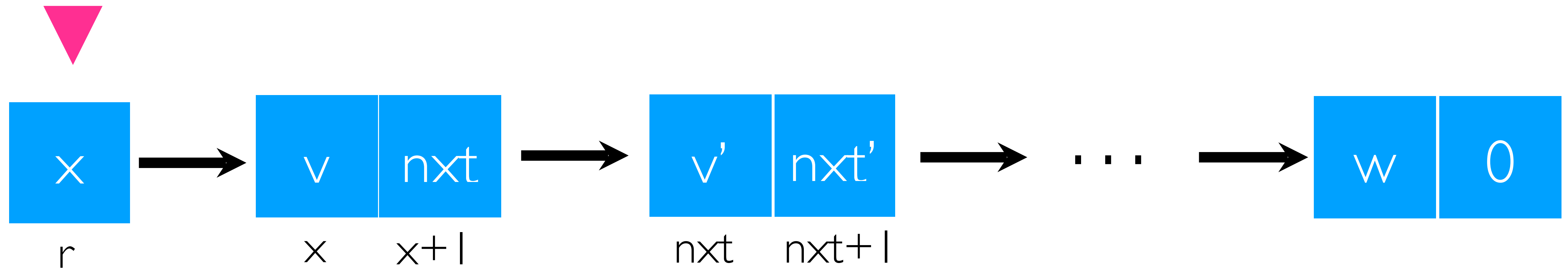
Example: copy a **linked list**



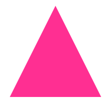
```
predicate ls (loc x, set S) {
   | x = 0  ∧  { S = ∅           ; emp }
   | x ≠ 0 ∧  { S = {v} ∪ S'   ; [x, 2] ∗ x ↦ v ∗ (x + 1) ↦ nxt ∗ ls(nxt, S') }
}
```

separating conjunction

Example: copy a linked list

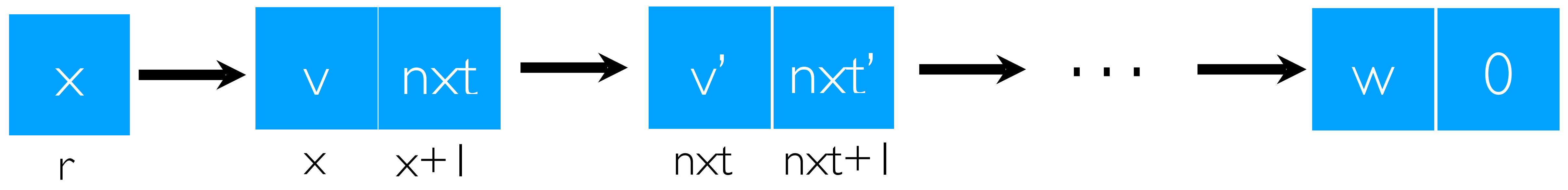


Precondition: `{r ↦ x * lseg(x, S)}`



```
void listcopy (loc r)
```

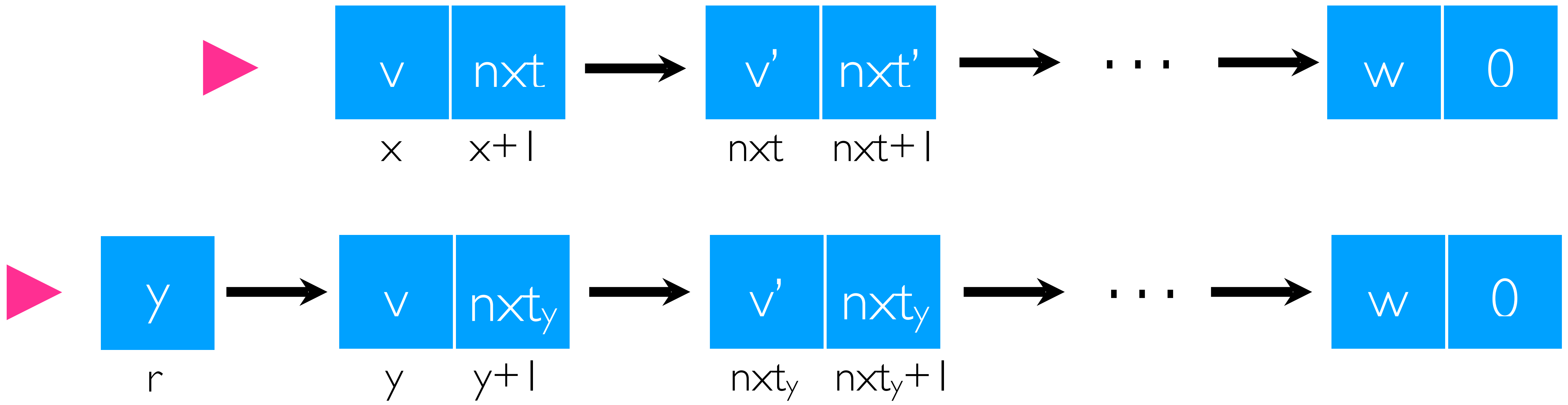
Example: copy a linked list



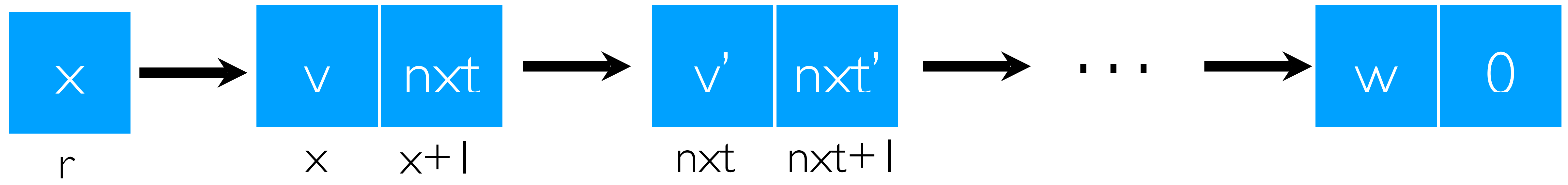
Precondition: {r ↦ x * lseg(x, S)}

```
void listcopy (loc r)
```

Postcondition: {r ↦ y * lseg(x, S) * lseg(y, S)}

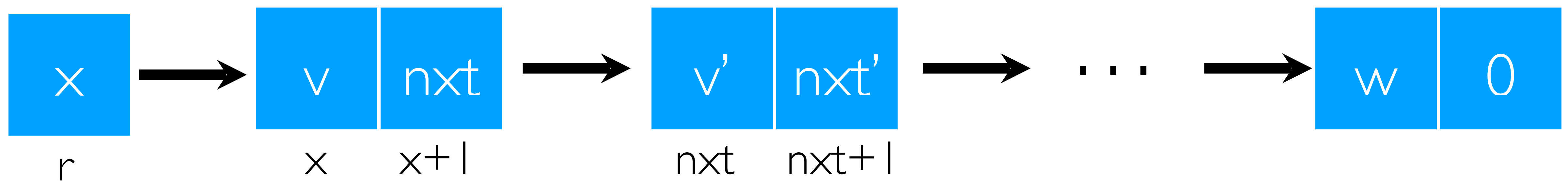


Example: copy a linked list



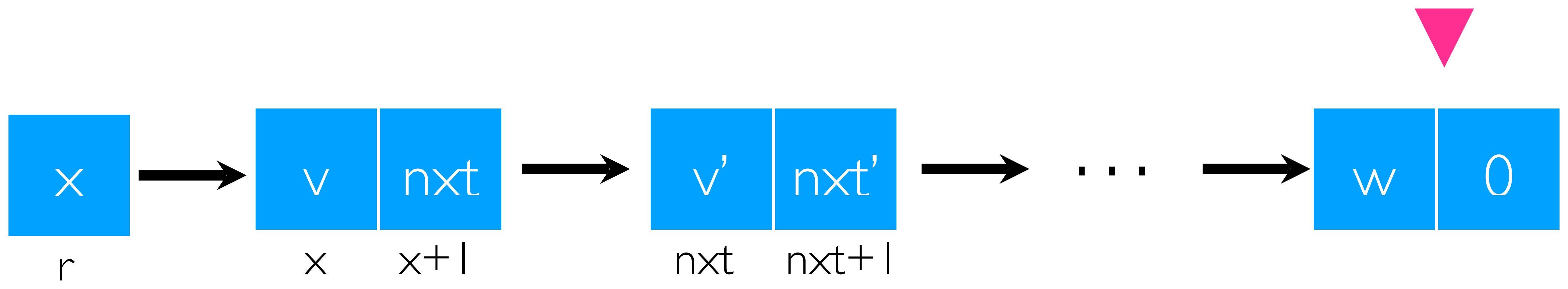
```
{r ↦ x * lseg(x, S)}

  void listcopy (loc r)

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



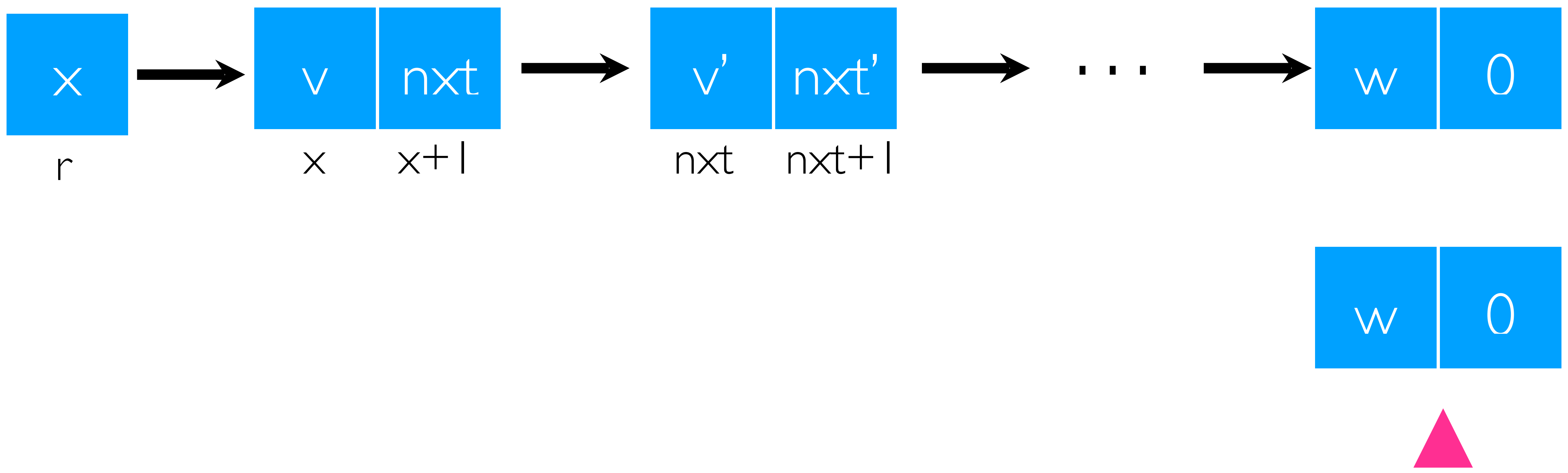
```
{r ↦ x * lseg(x, S)}

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
  ▶ listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



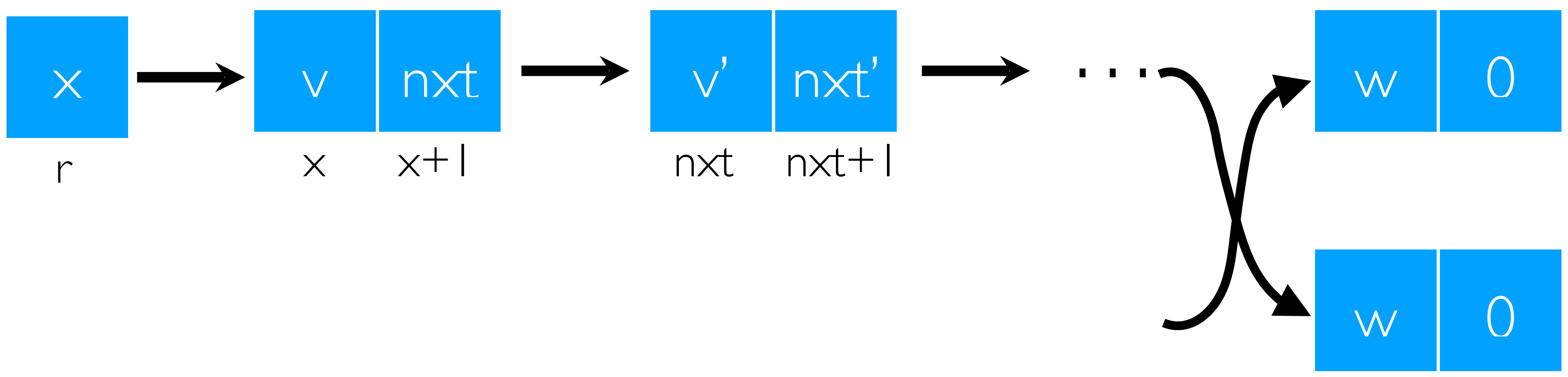
```
{r ↦ x * lseg(x, S)}

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



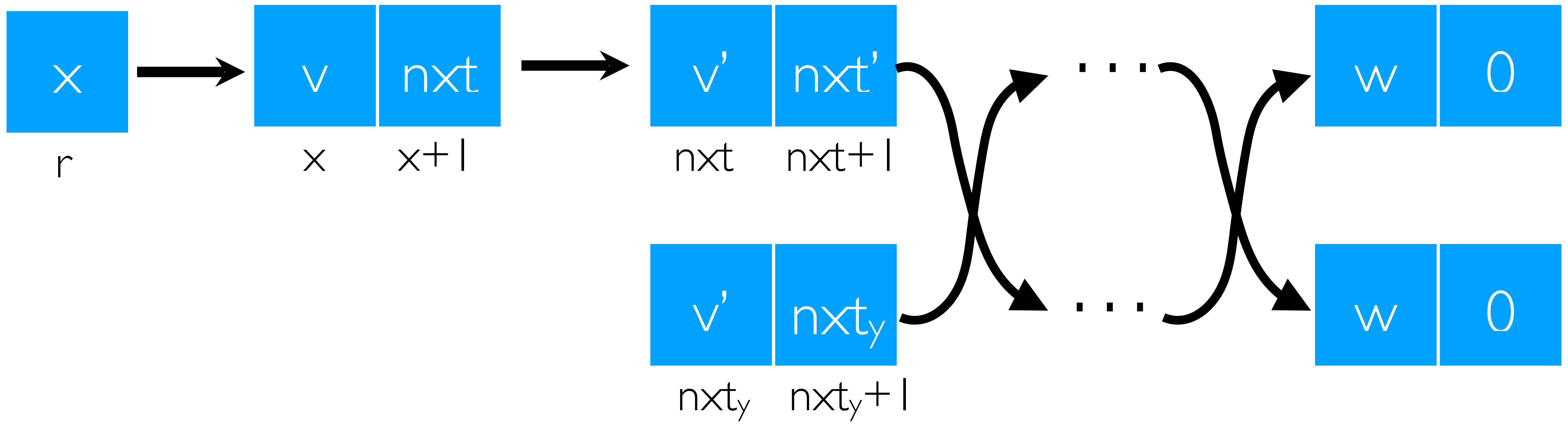
```
{r ↦ x * lseg(x, S)}

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
  ▶ let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



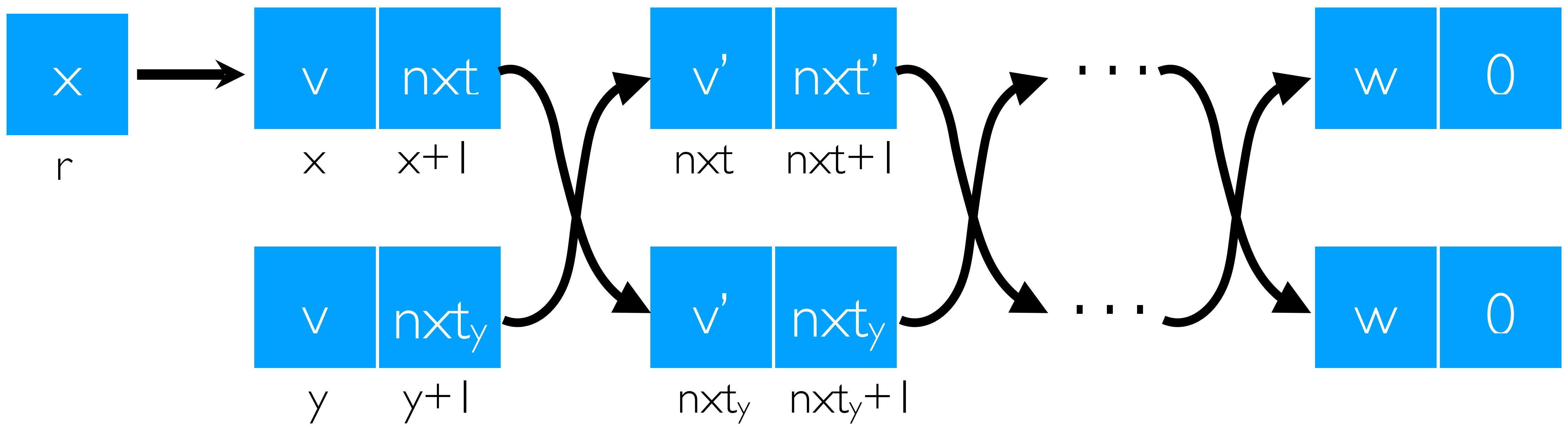
```
{r ↦ x * lseg(x, S)}

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
 ▶  *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



```
{r ↦ x * lseg(x, S)}

void listcopy (loc r) {
  let x = ∗r;
  if (x == 0) {
  } else {
    let v = ∗x;
    let nxt = ∗(x + 1);
    ∗r = nxt;
    listcopy(r);
    let y1 = ∗r;
    let y = malloc(2);
    ∗(x + 1) = y1;
    ∗r = y;
    ∗(y + 1) = nxt;
    ∗y = v;
  } }

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list

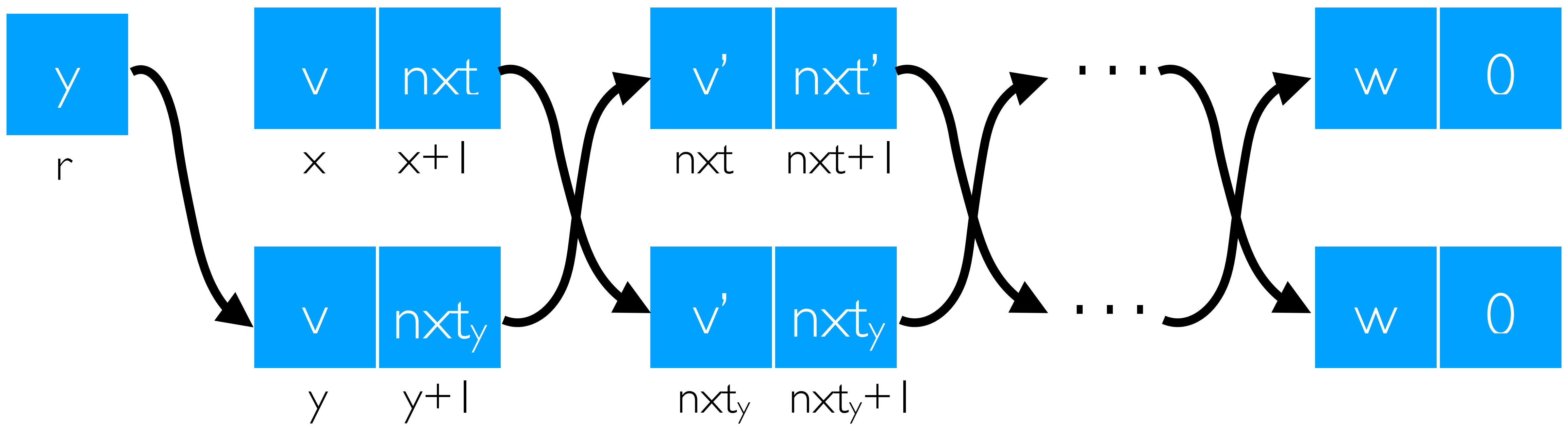


```
{r ↦ x * lseg(x, S)}
```

```
void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }
```

```
{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list

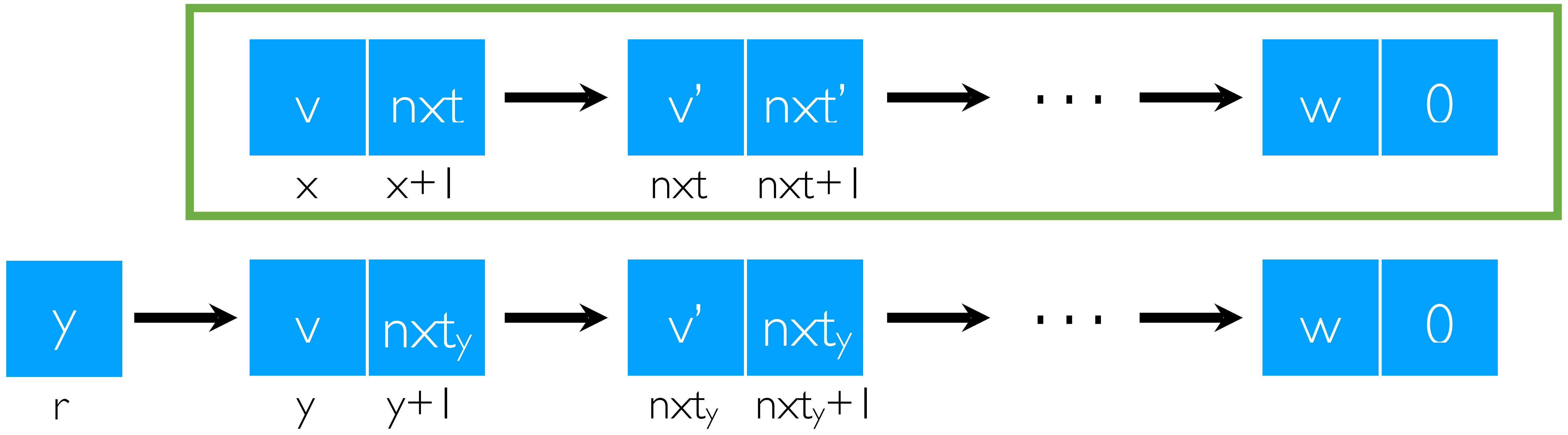


```
{r ↦ x * lseg(x, S)}
```

```
void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }
```

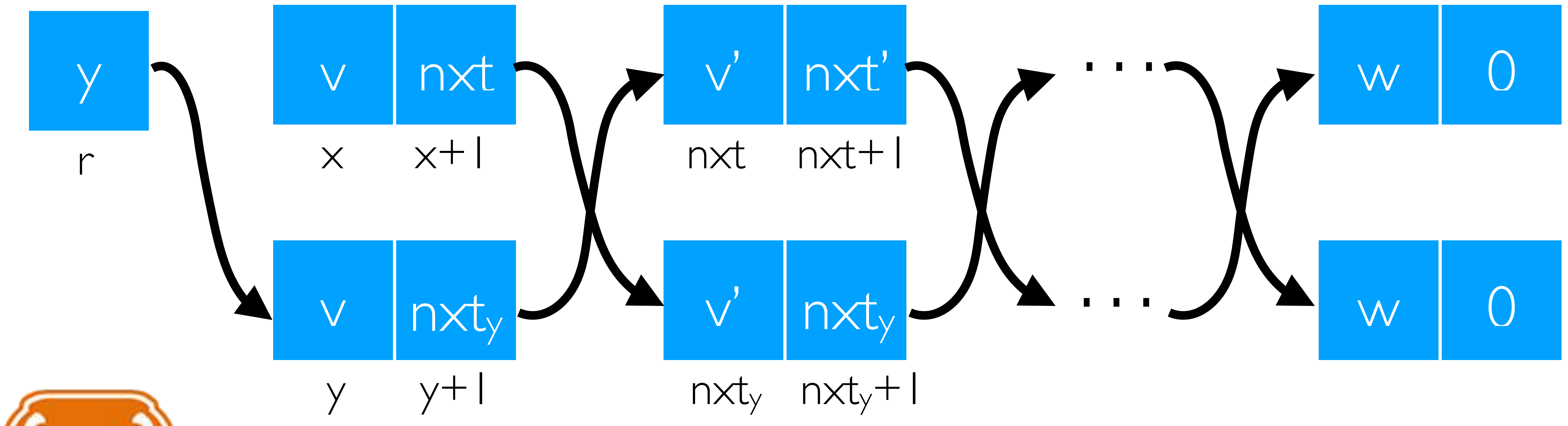
```
{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



expected

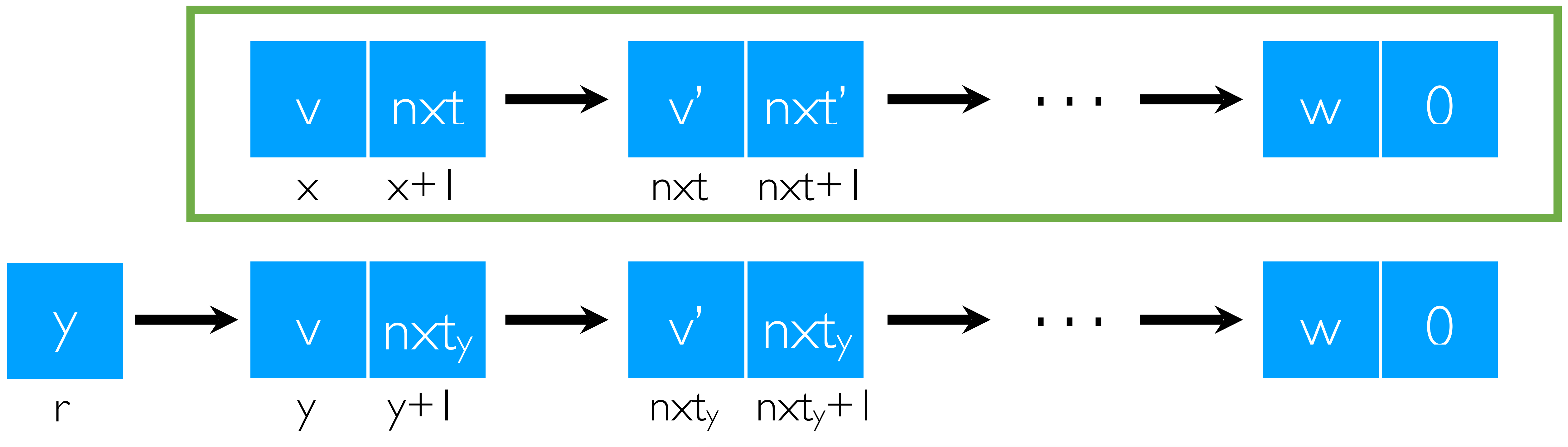
result



Spurious writes.

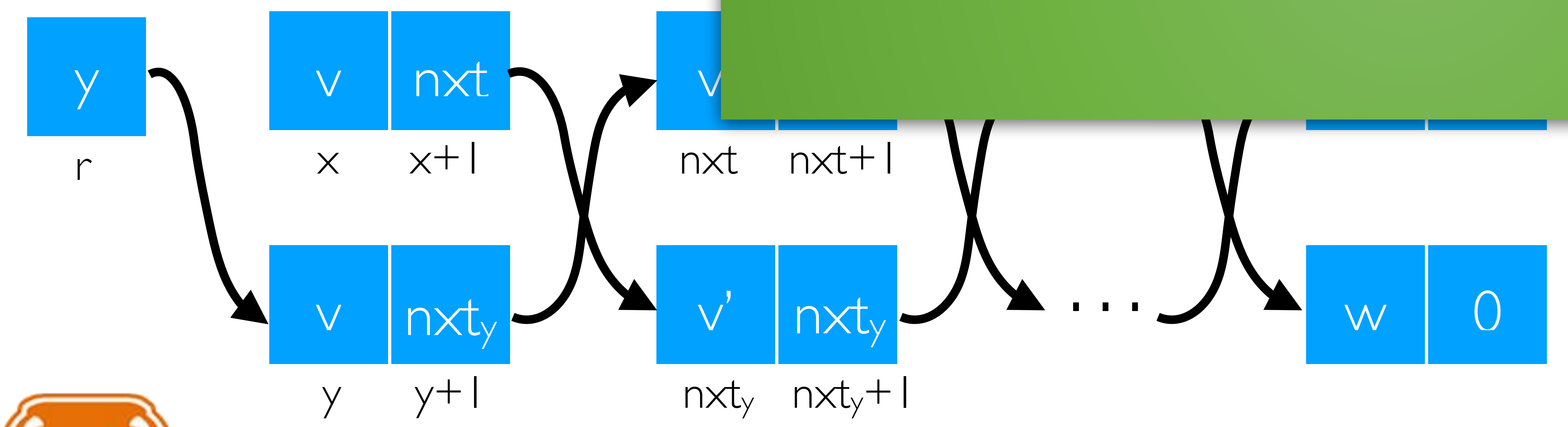
```
{r ↦ x * lseg(x, S)}

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
} }

{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Example: copy a linked list



expected

result



Make the initial list *Read-Only*:
it **must not be altered**

Spurious writes.

```
{r ↦ x * lseg(x, S)}
```

```
void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
} }
```

```
{r ↦ y * lseg(x, S) * lseg(y, S)}
```

Synthesis of Programs with Pointers
via Read-Only Specifications

(our contribution)

Effective:	more natural and shorter programs
Efficient:	smaller search space—faster synthesis
Robust:	better performance in “worst case scenarios”

A Primer on
Synthetic Separation Logic

Syntactic Separation Logic

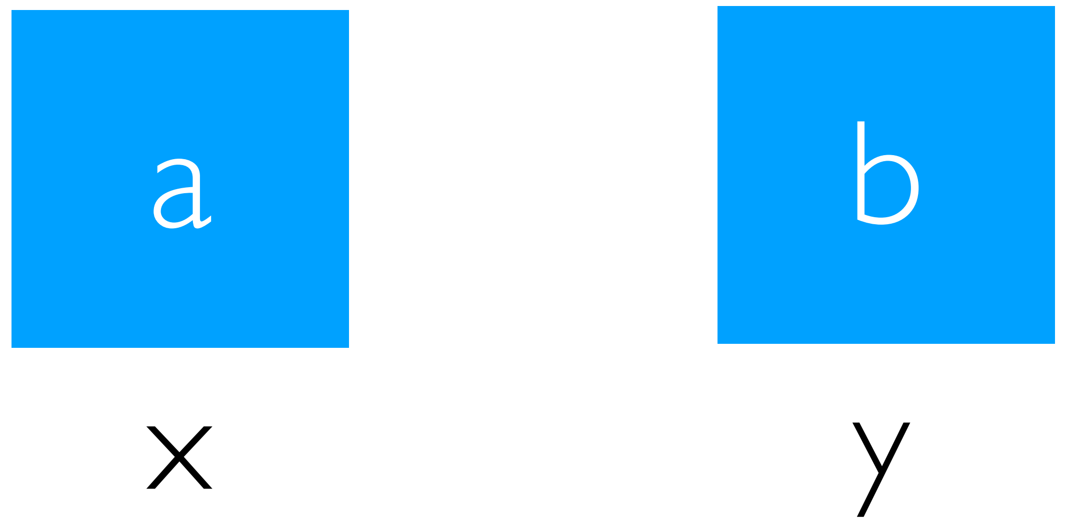
$$\begin{tikzcd} \{P\}\ c\ \{Q\} \arrow[d] \\ P \rightsquigarrow Q \mid c \end{tikzcd}$$

Syntactic Separation Logic

$$\begin{tikzcd} & & \{P\}\ c\ \{Q\} \arrow[d] \\ P \vdash Q & \text{implies} & P \rightsquigarrow Q \mid c \end{tikzcd}$$

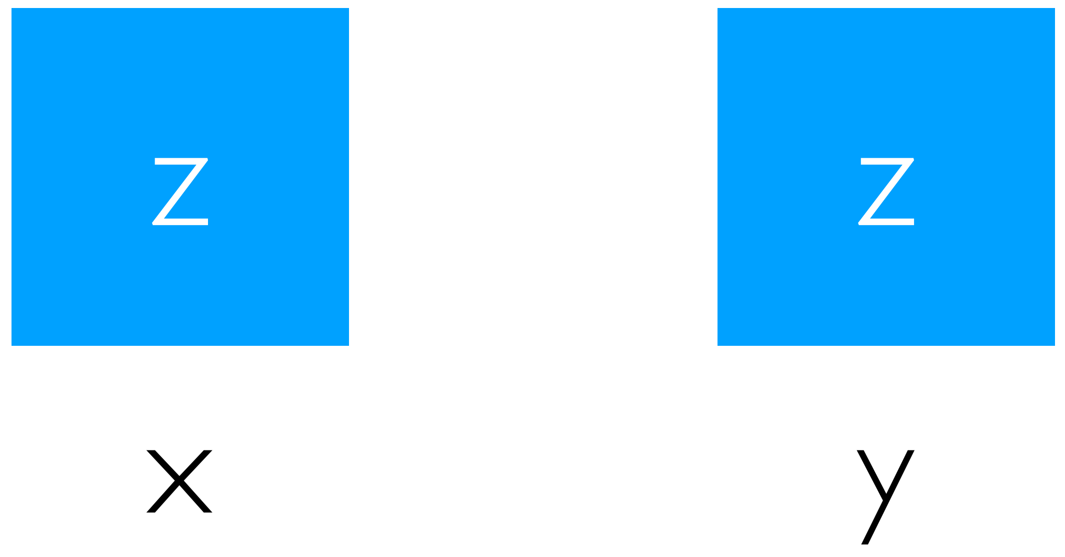
Example: **pick** - equalises the values of two distinct memory locations

Precondition:



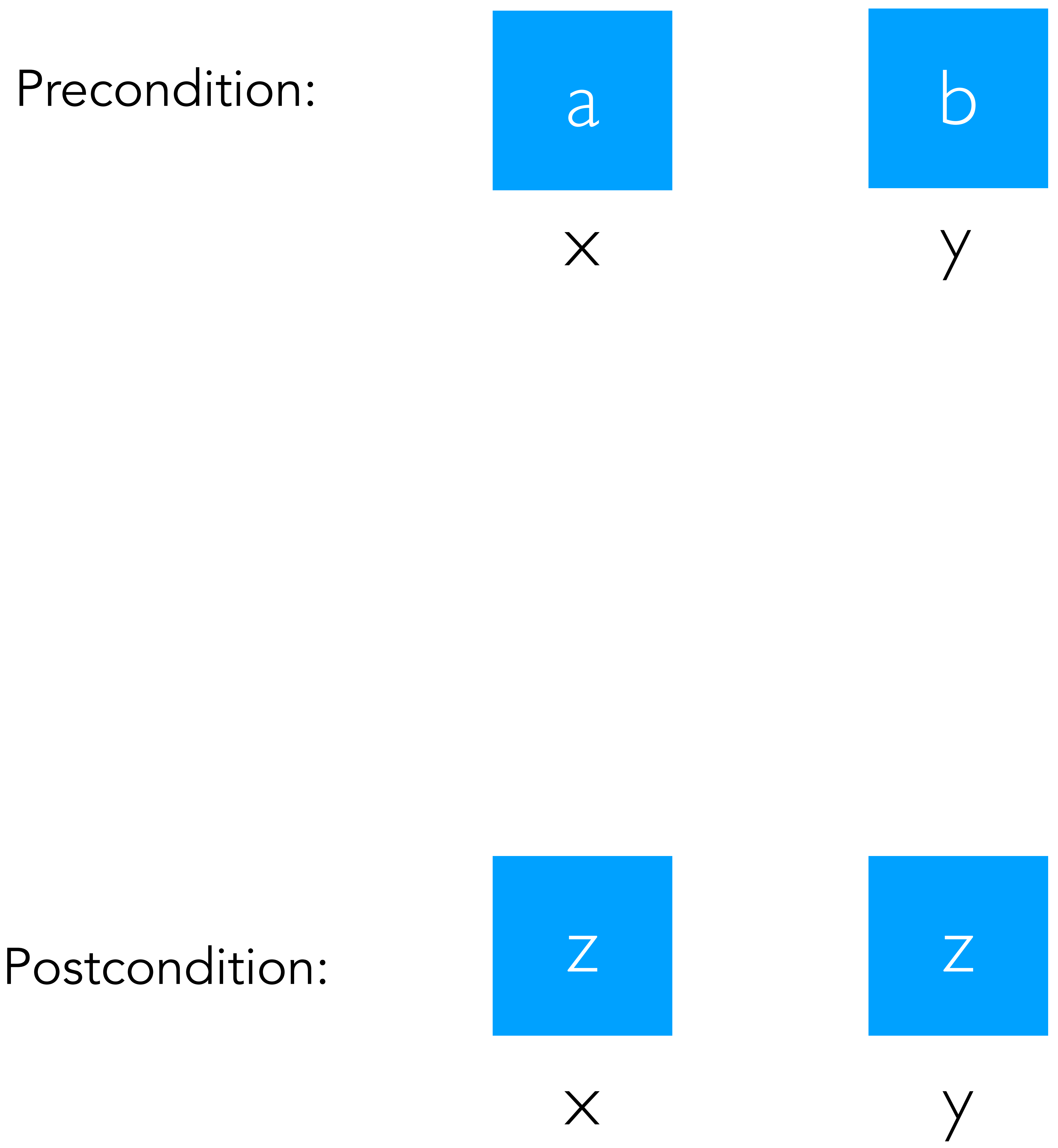
$$\{ x \mapsto a * y \mapsto b \}$$

Postcondition:



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

Example: **pick** - equalises the values of two distinct memory locations

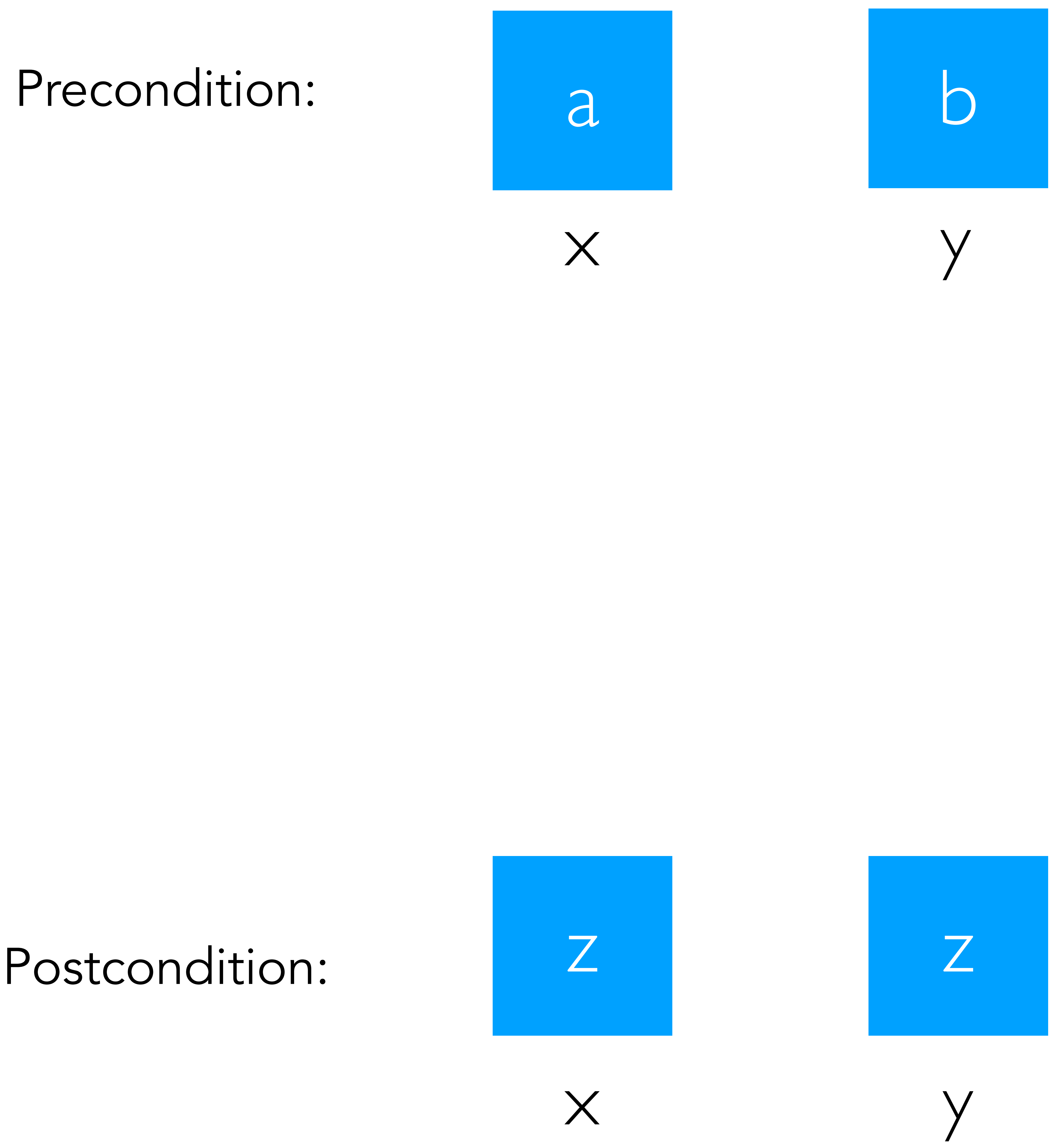


$\{\ x \mapsto a * y \mapsto b\ \}$

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

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

Example: **pick** - equalises the values of two distinct memory locations



Precondition:

a b

x y

Postcondition:

z z

x y

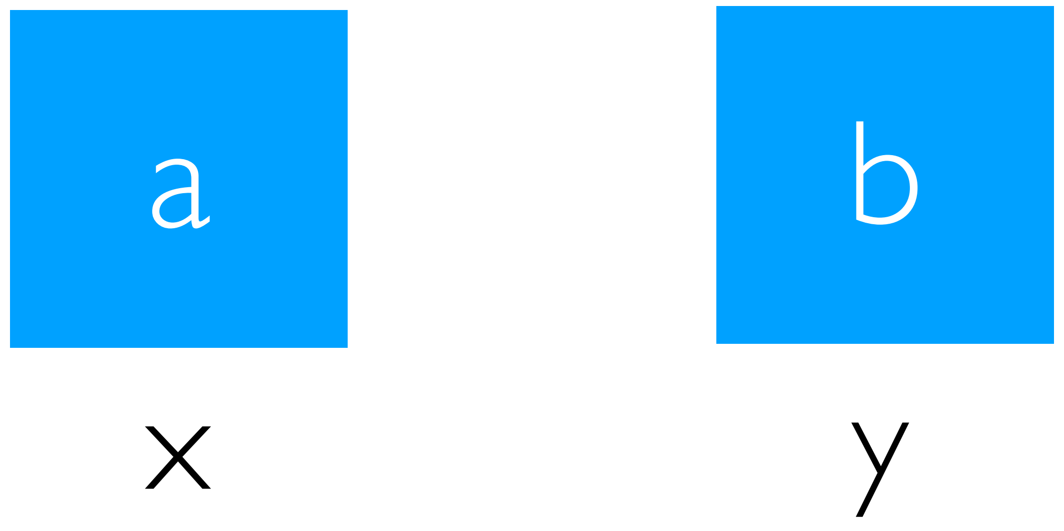
$\{\ x \mapsto a * y \mapsto b\ \}$

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

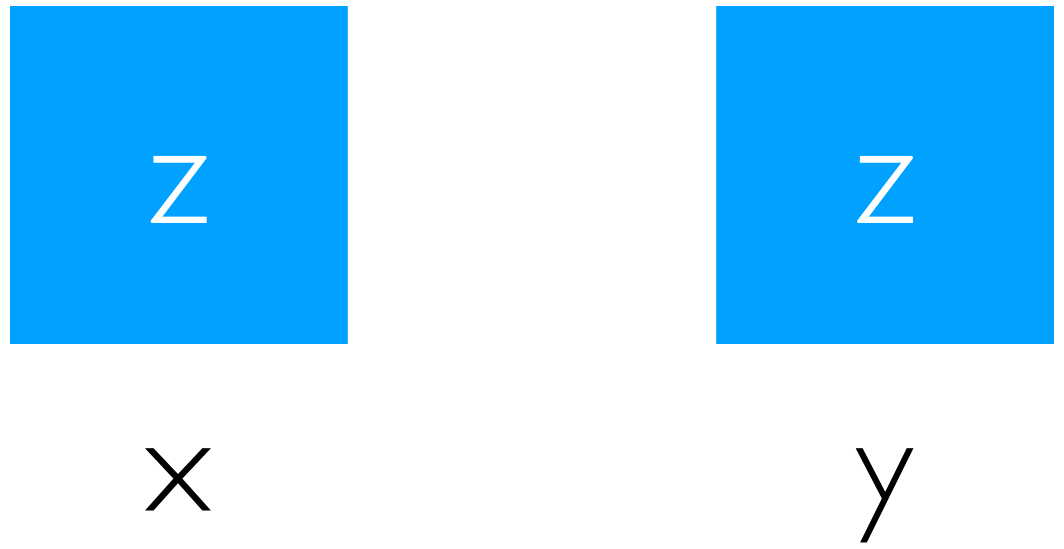
$\{\ x \mapsto z * y \mapsto z\ \}$

Example: **pick** - equalises the values of two distinct memory locations

Precondition:



Postcondition:

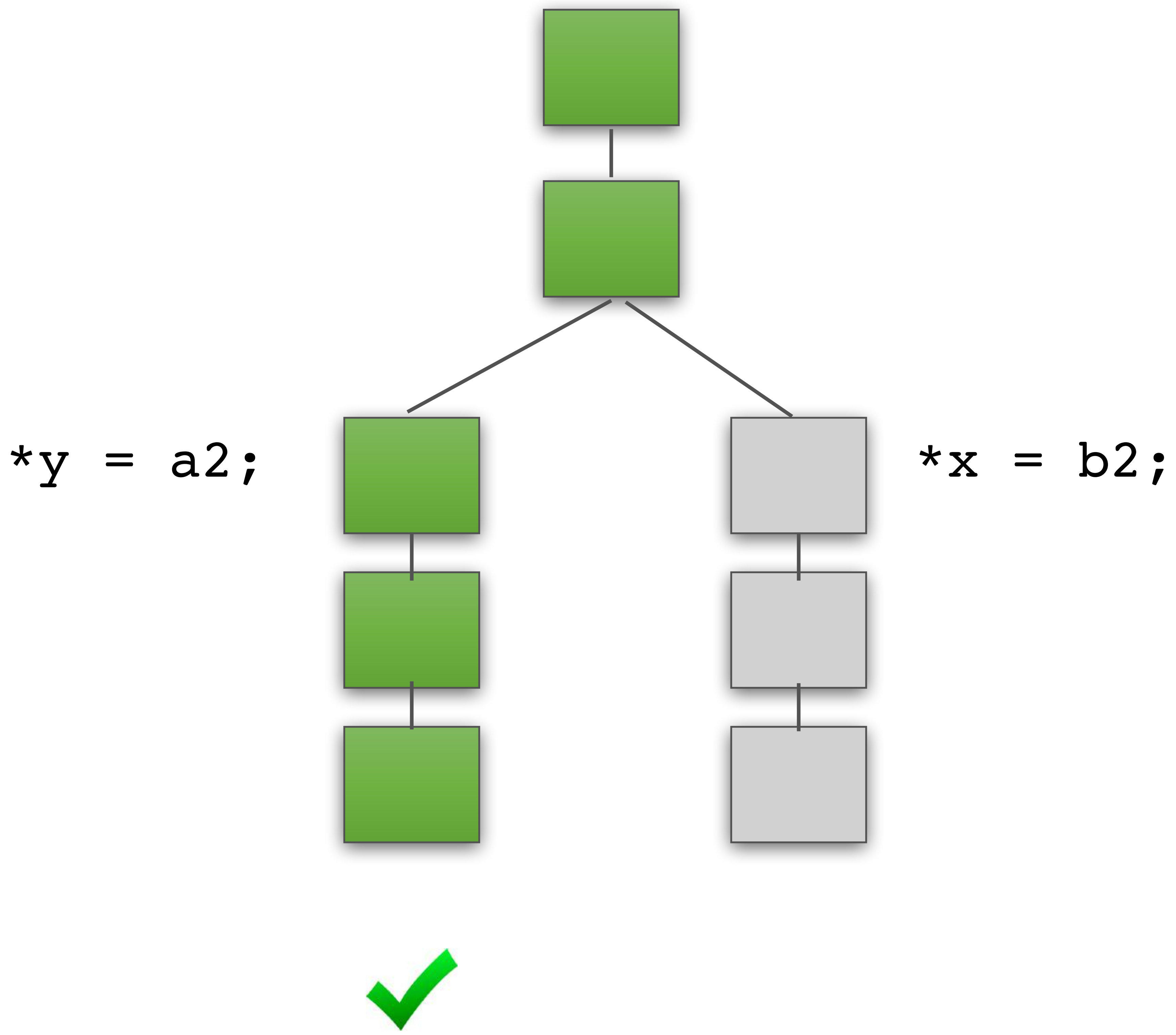


$\{ \ x \mapsto a * y \mapsto b \ \}$

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

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

Example: **pick** - equalises the values of two distinct memory locations

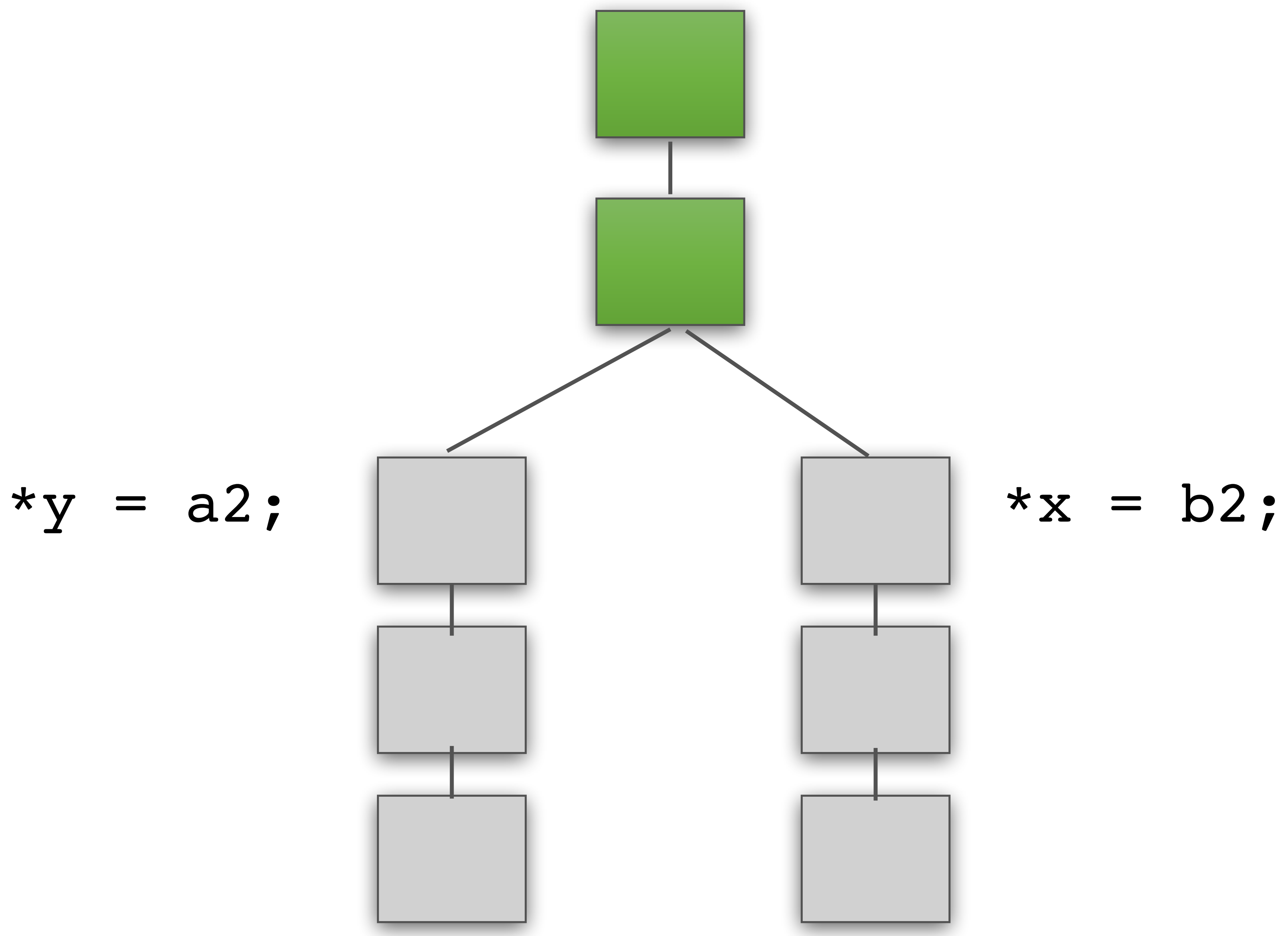


$$\{\ \mathsf{x} \mapsto a * \mathsf{y} \mapsto b\ \}$$

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

$$\{\ \mathsf{x} \mapsto z * \mathsf{y} \mapsto z\ \}$$

Example: **pick** - equalises the values of two distinct memory locations

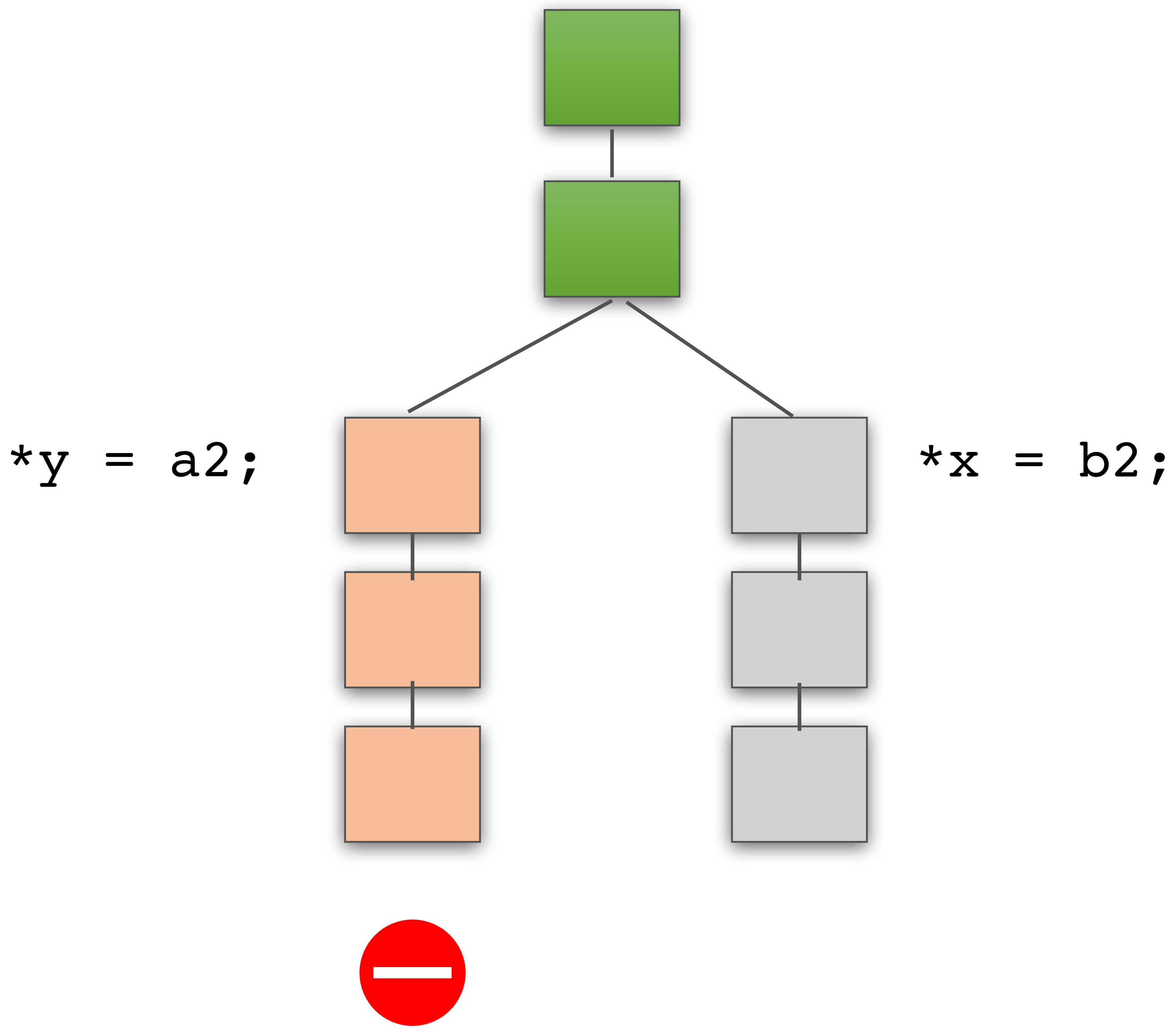


$\{\ x \mapsto a * \boxed{y \mapsto b}\,🔒 \}$

```
void pick(loc x, loc y) {

      let a2 = *x;

      let b2 = *y;


}
```

$\{\ x \mapsto z * \boxed{y \mapsto z}\,🔒 \}$

Example: **pick** - equalises the values of two distinct memory locations

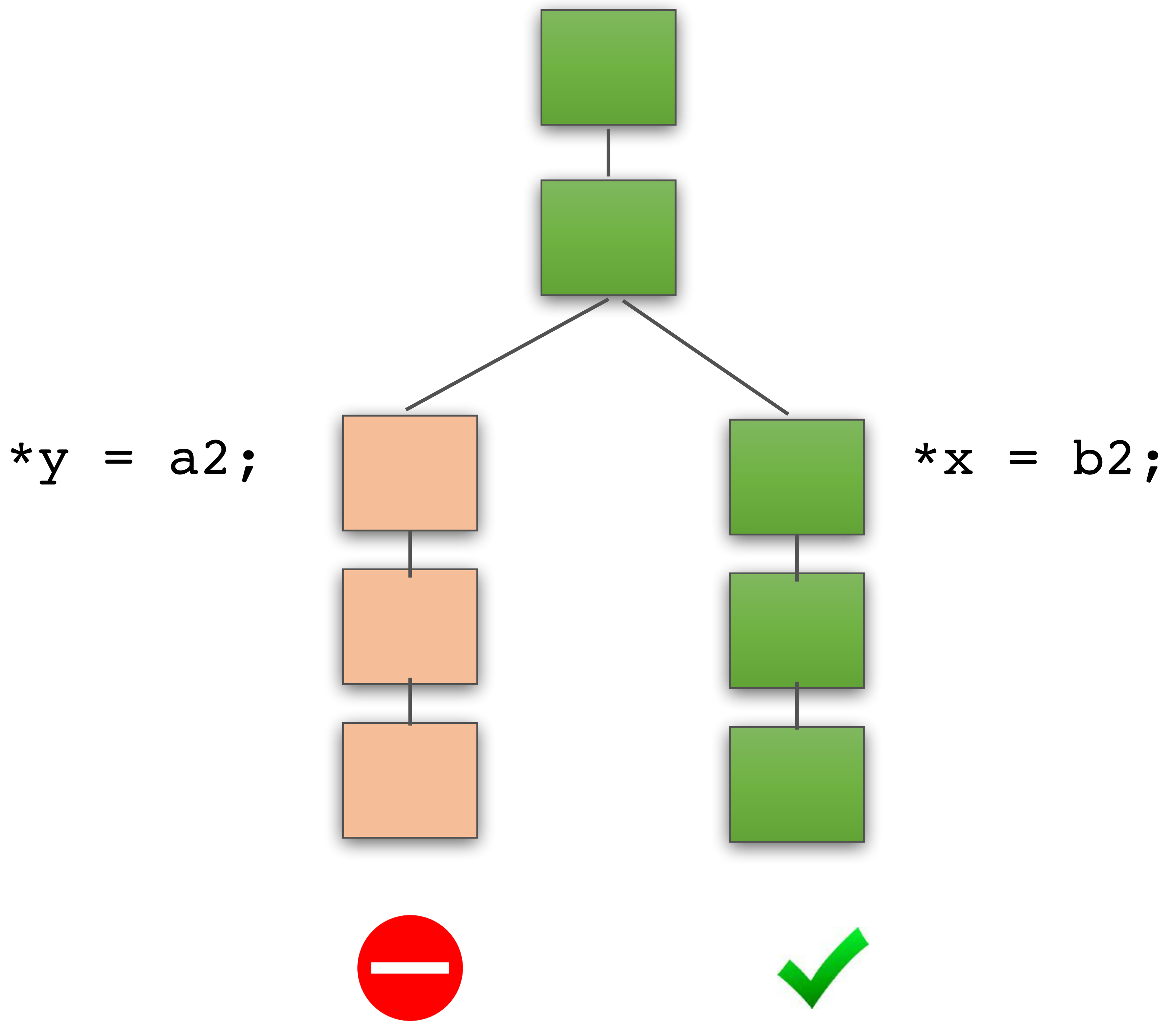


$\{\ x \mapsto a * \boxed{y \mapsto b}$🔒$\}$

```
void pick(loc x, loc y) {

        let a2 = *x;

        let b2 = *y;


}
```

$\{\ x \mapsto z * \boxed{y \mapsto z}$🔒$\}$

Example: **pick** - equalises the values of two distinct memory locations

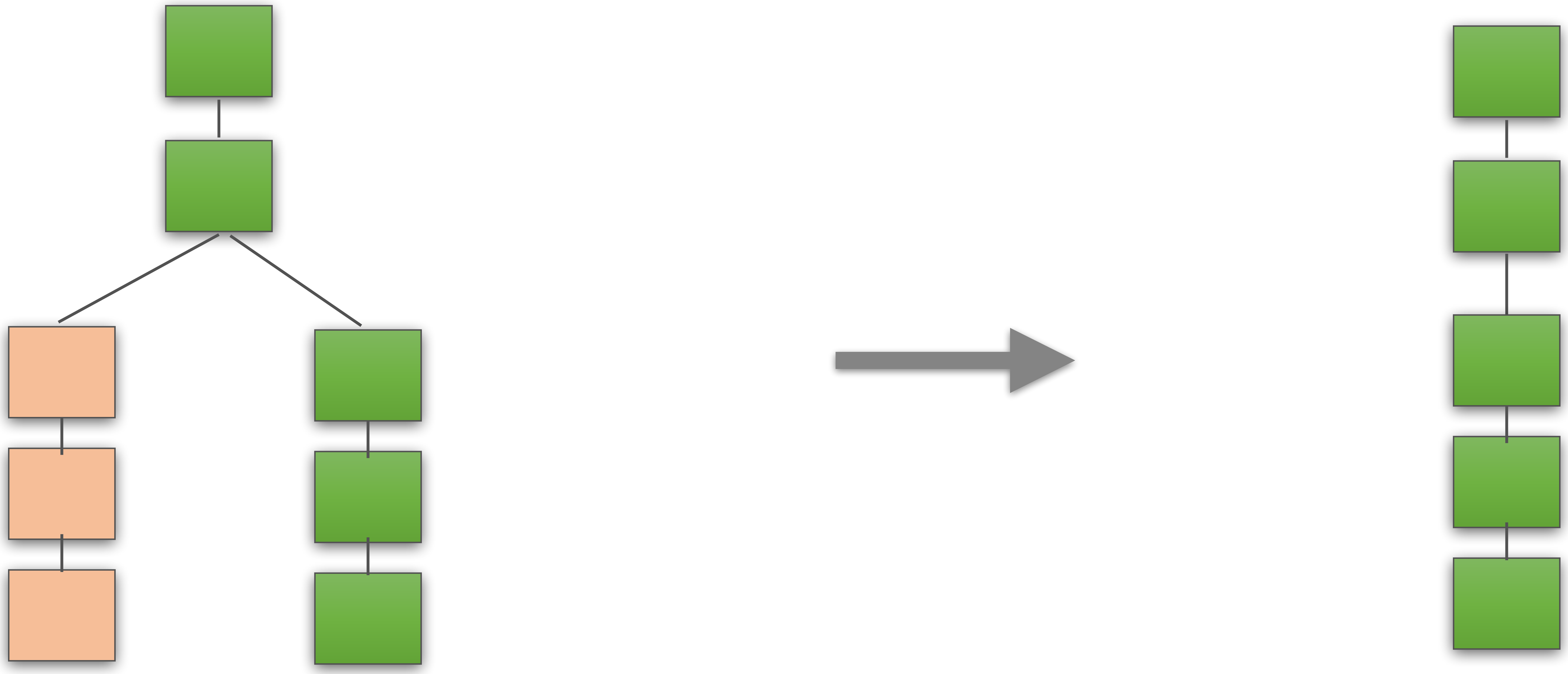


$\{\ x \mapsto a * \boxed{y \mapsto b}$🔒$\}$

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

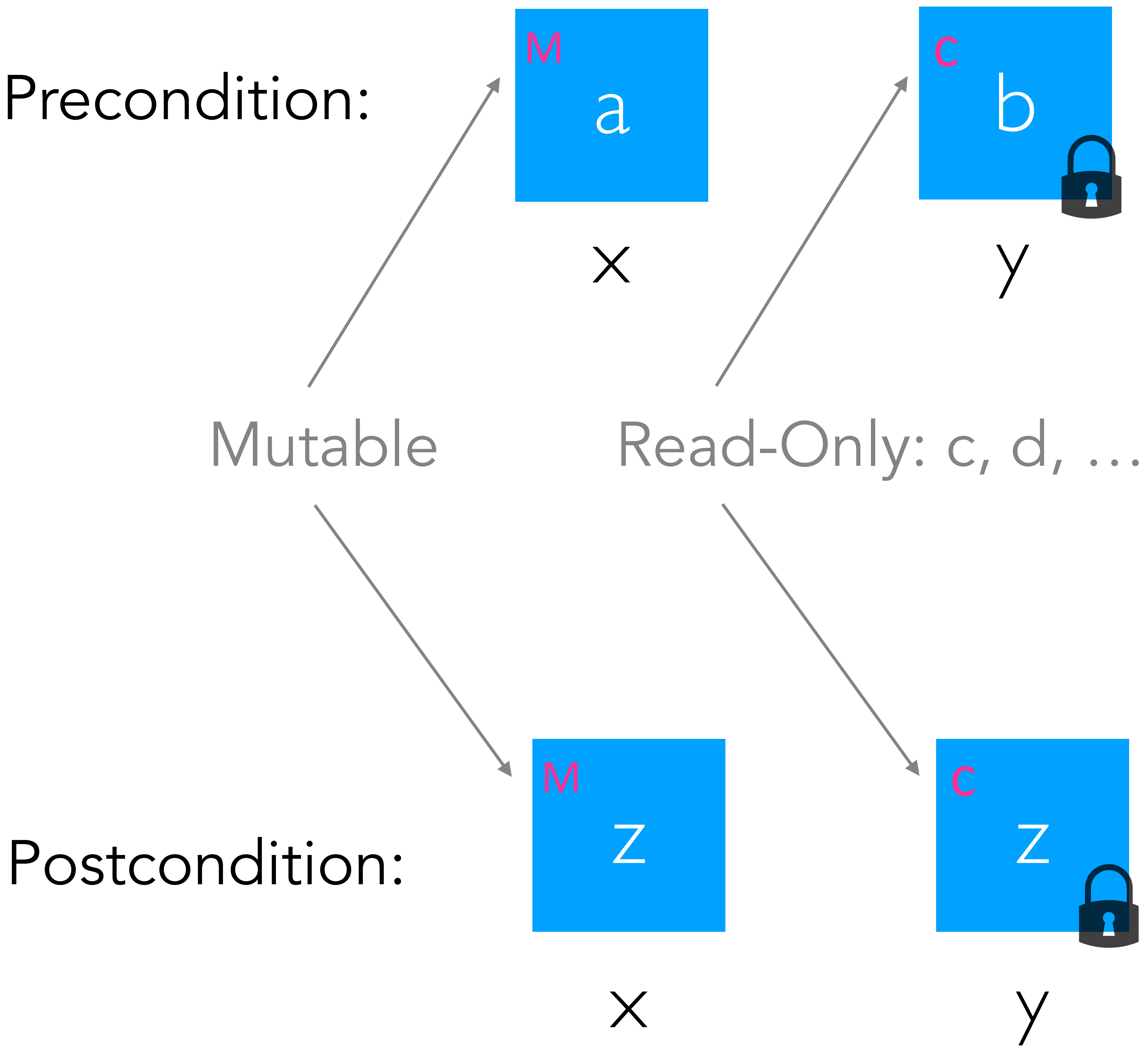
$\{\ x \mapsto z * \boxed{y \mapsto z}$🔒$\}$

Example: **pick** - equalises the values of two distinct memory locations



Read-Only Specifications

Example: pick with **Read-Only Specifications**



$$\{\ x \overset{M}{\mapsto} a * \boxed{y \overset{c}{\mapsto} b}\ \}$$

```
void pick(loc x, loc y){

      let a2 = *x;

      let b2 = *y;

      *y = a2;

}
```

$$\{\ x \overset{M}{\mapsto} z * \boxed{y \overset{c}{\mapsto} z}\ \}$$

Example: pick with **Read-Only Specifications**

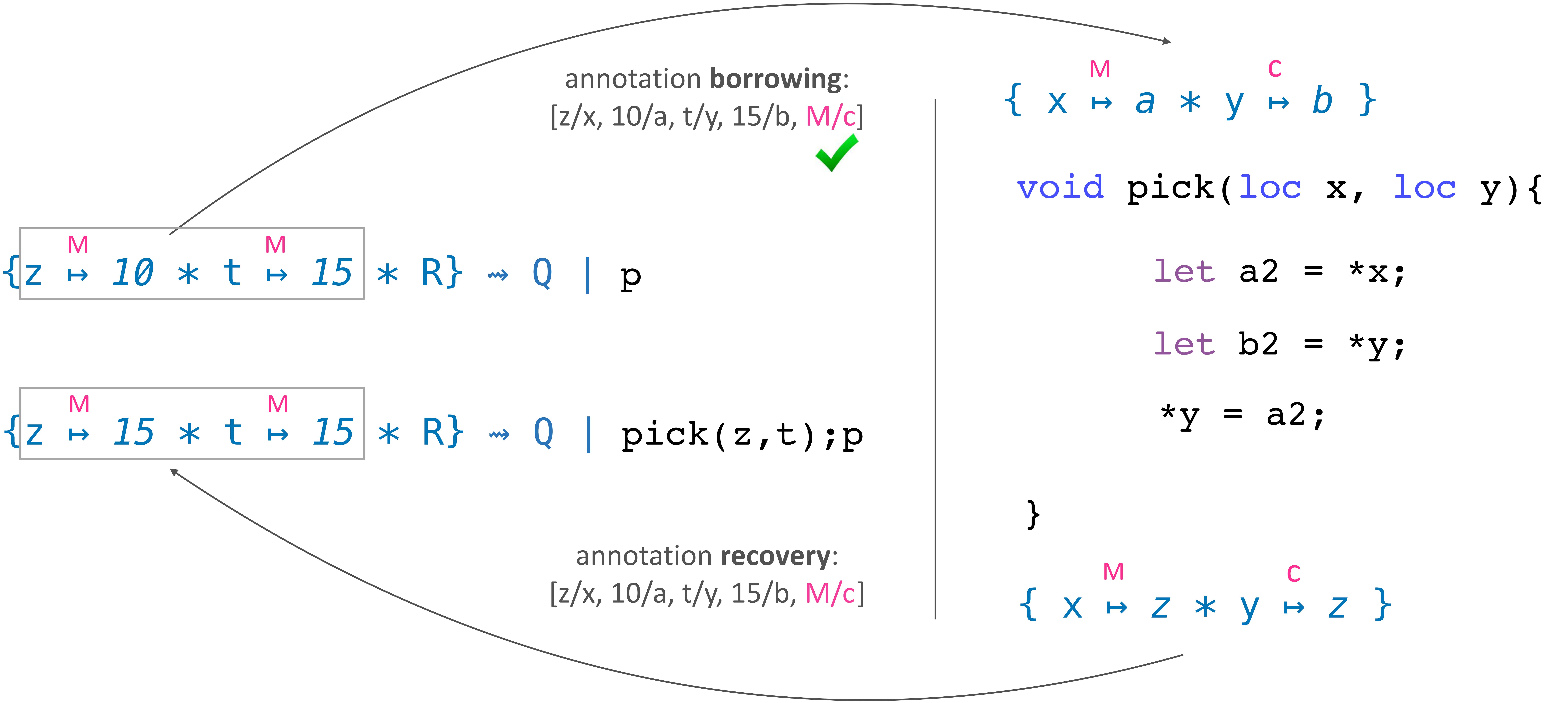
$\{z \overset{M}{\mapsto} \mathit{10} * t \overset{M}{\mapsto} \mathit{15} * R\} \rightsquigarrow Q \mid p$

$\{ x \overset{M}{\mapsto} a * y \overset{C}{\mapsto} b \}$

```
void pick(loc x, loc y){

      let a2 = *x;

      let b2 = *y;

      *y = a2;

}
```

$\{ x \overset{M}{\mapsto} z * y \overset{C}{\mapsto} z \}$

Example: pick with **Read-Only Specifications**



annotation **borrowing**:
[z/x, 10/a, t/y, 15/b, M/c] ✓

$\{\boxed{z \overset{M}{\mapsto} \mathit{10} * t \overset{M}{\mapsto} \mathit{15}} * R\} \rightsquigarrow Q \mid \texttt{p}$

$\{\boxed{z \overset{M}{\mapsto} \mathit{15} * t \overset{M}{\mapsto} \mathit{15}} * R\} \rightsquigarrow Q \mid \texttt{pick(z,t);p}$

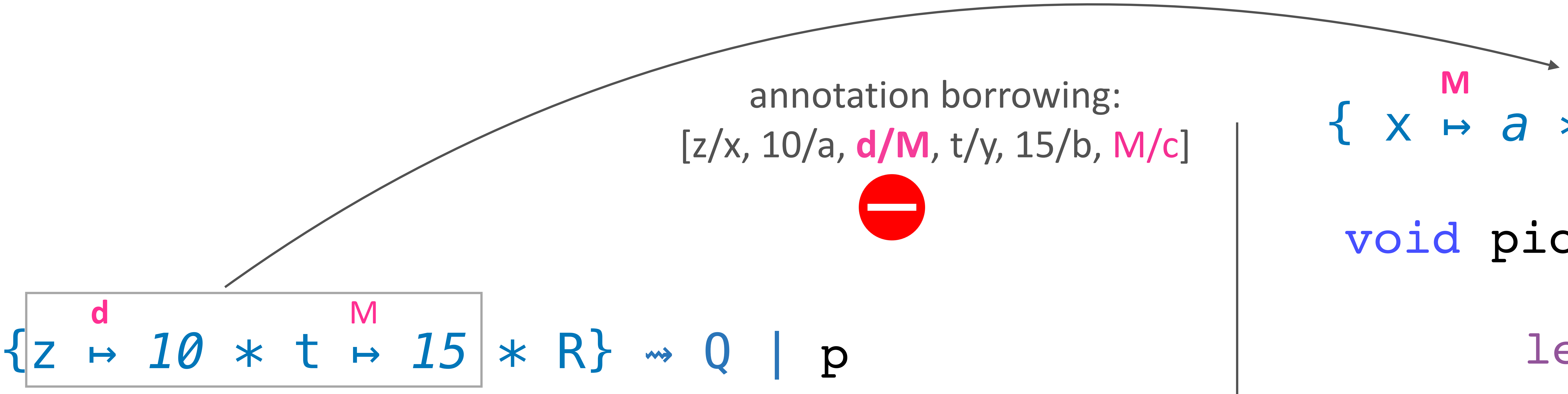
annotation **recovery**:
[z/x, 10/a, t/y, 15/b, M/c]

$\{\ x \overset{M}{\mapsto} a * y \overset{c}{\mapsto} b\ \}$

```
void pick(loc x, loc y){

      let a2 = *x;

      let b2 = *y;

      *y = a2;

}
```

$\{\ x \overset{M}{\mapsto} z * y \overset{c}{\mapsto} z\ \}$

Example: pick with **Read-Only Specifications**



annotation borrowing:
[z/x, 10/a, **d/M**, t/y, 15/b, M/c]

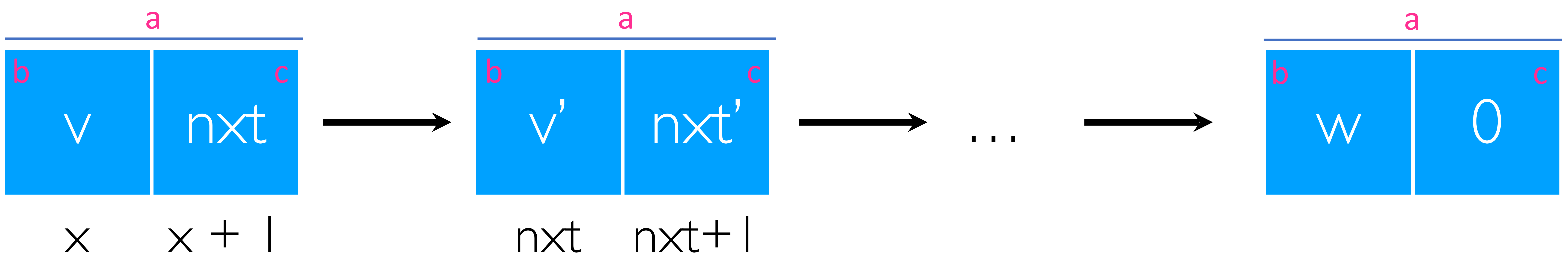
$\{\boxed{z \overset{d}{\mapsto} \mathit{10} * t \overset{M}{\mapsto} \mathit{15}} * R\} \leadsto Q \mid p$

$\{\ x \overset{M}{\mapsto} a * y \overset{c}{\mapsto} b\ \}$

```
void pick(loc x, loc y){

      let a2 = *x;

      let b2 = *y;

      *y = a2;

}
```

$\{\ x \overset{M}{\mapsto} z * y \overset{c}{\mapsto} z\ \}$

Example: copy of a linked list Read-Only Specifications



Example: copy of a linked list

read-only

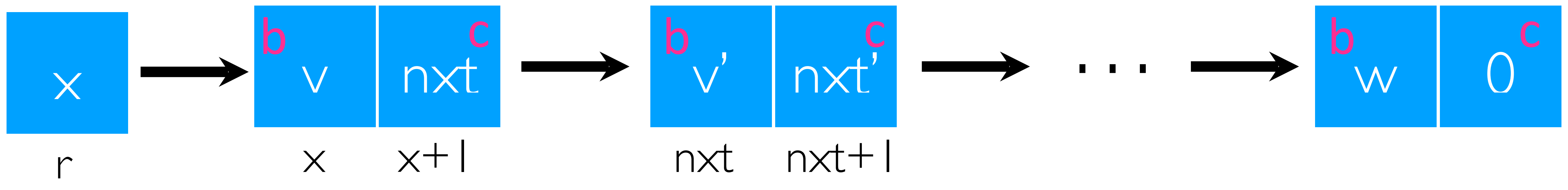
```
{r ↦ x * lseg(x,S,a,b,c) }

     void listcopy (loc r)

{r ↦ y * lseg(x,S,a,b,c) * lseg(y,S,M,M,M)}
```

mutable

Example: copy of a linked list

```
{r ↦ x * lseg(x,S,a,b,c) }

     void listcopy (loc r)

{r ↦ y * lseg(x,S,a,b,c) * lseg(y,S,M,M,M)}
```

Example: copy of a linked list

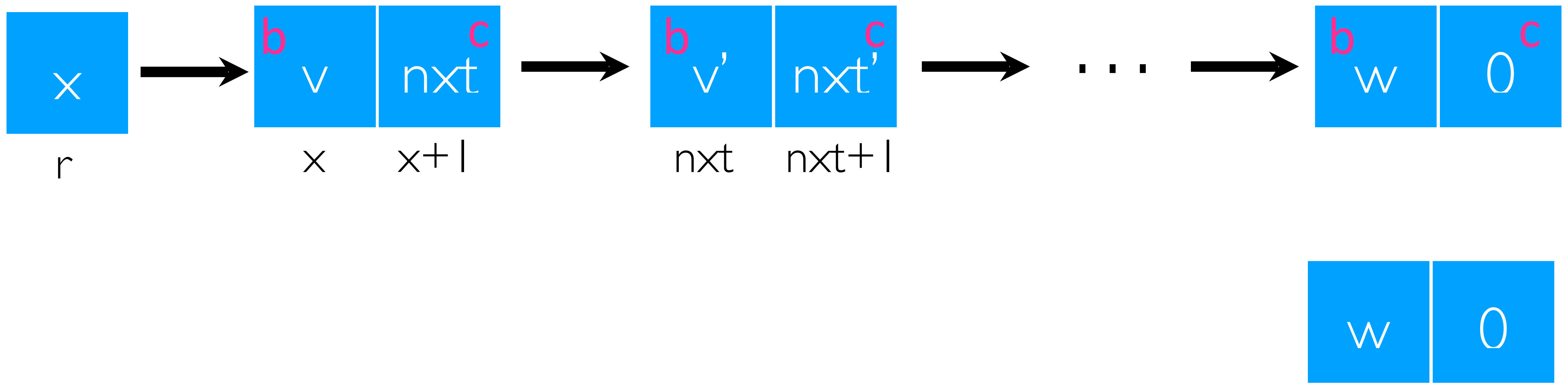


```
{r ↦ x * lseg(x,S,a,b,c) }
```

```
void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }
```

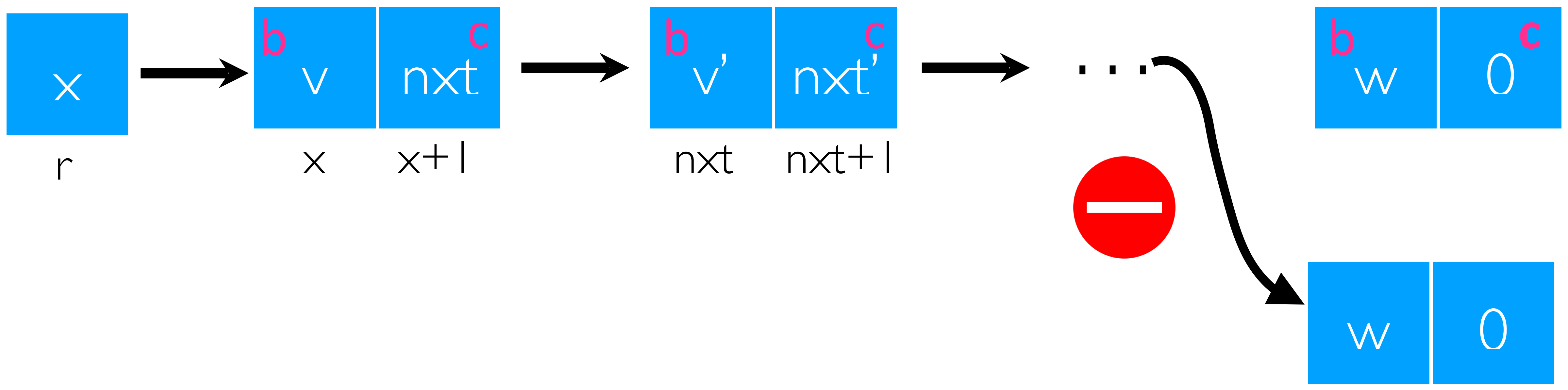
```
{r ↦ y * lseg(x,S,a,b,c) * lseg(y,S,M,M,M)}
```

Example: copy of a linked list



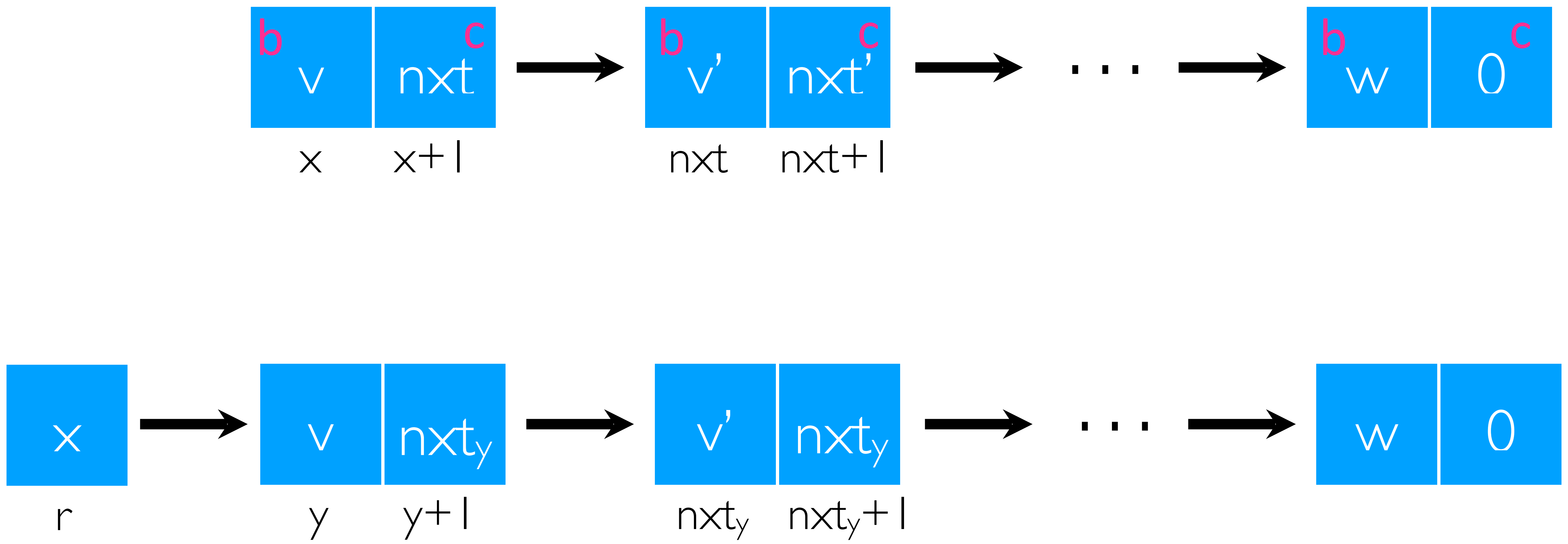
```
{r ↦ x * lseg(x,S,a,b,c) }

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
  } }

{r ↦ y * lseg(x,S,a,b,c) * lseg(y,S,M,M,M)}
```

Example: copy of a linked list



```
{r ↦ x * lseg(x,S,a,b,c) }

void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);
    *(x + 1) = y1;
    *r = y;
    *(y + 1) = nxt;
    *y = v;
} }

{r ↦ y * lseg(x,S,a,b,c) * lseg(y,S,M,M,M)}
```

Example: copy of a linked list



{r ↦ x * lseg(x,S,a,b,c) }

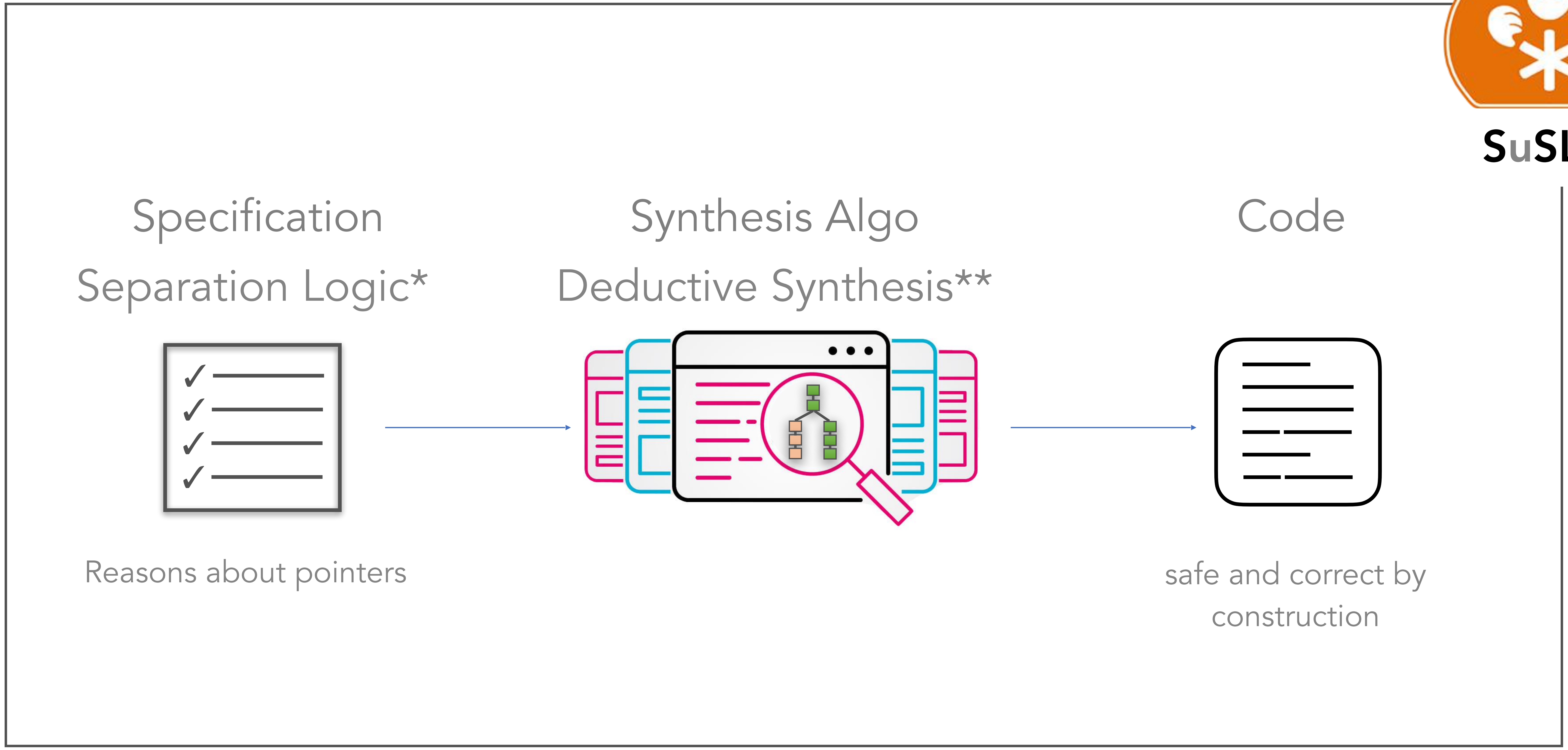
```
void listcopy (loc r) {
  let x = *r;
  if (x == 0) {
  } else {
    let v = *x;
    let nxt = *(x + 1);
    *r = nxt;
    listcopy(r);
    let y1 = *r;
    let y = malloc(2);

    *r = y;
    *(y + 1) =  y1 ;
    *y = v;
} }
```

✓

{r ↦ y * lseg(x,S,a,b,c) * lseg(y,S,M,M,M)}

SSL: **S**ynthetic **S**eparation **L**ogic

SuSLik



Specification

Separation Logic*

Synthesis Algo

Deductive Synthesis**

Code

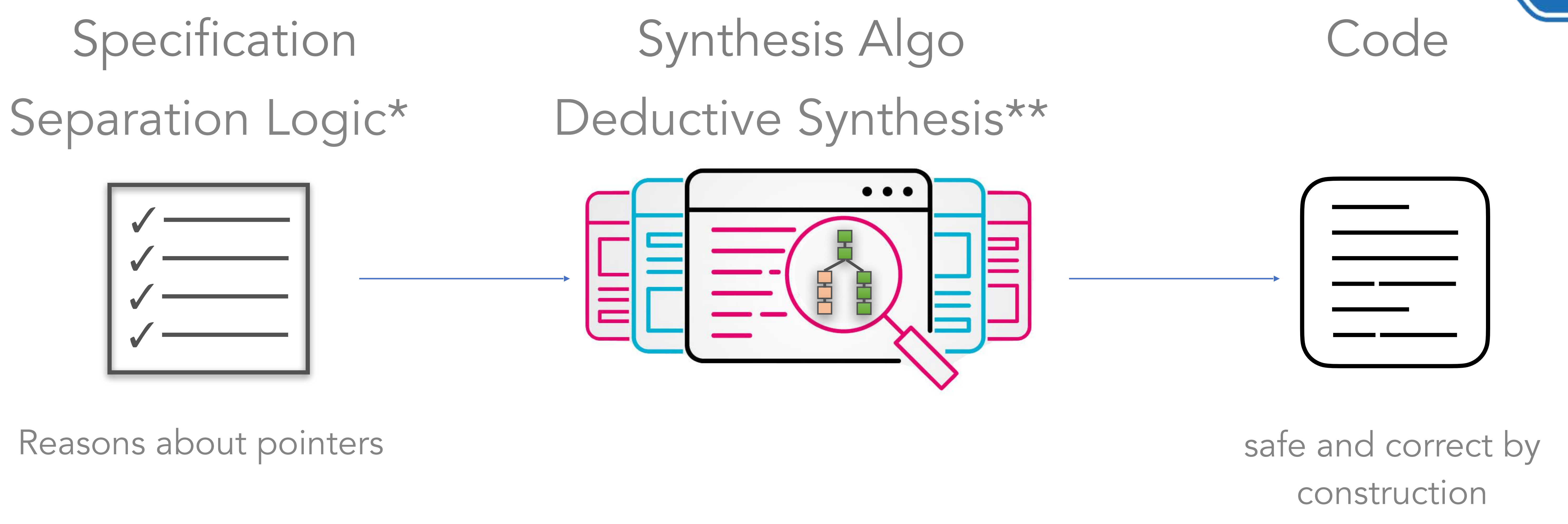
Reasons about pointers

safe and correct by construction

* *Local Reasoning about Programs that Alter Data Structures*, O'Hearn, Reynolds, Yang:. CSL 2001

** *Structuring the Synthesis of Heap-Manipulating Programs*, Polikarpova & Sergey @POPL'19

BoSSL: Borrowing Synthetic Separation Logic



Specification

Separation Logic*

Reasons about pointers

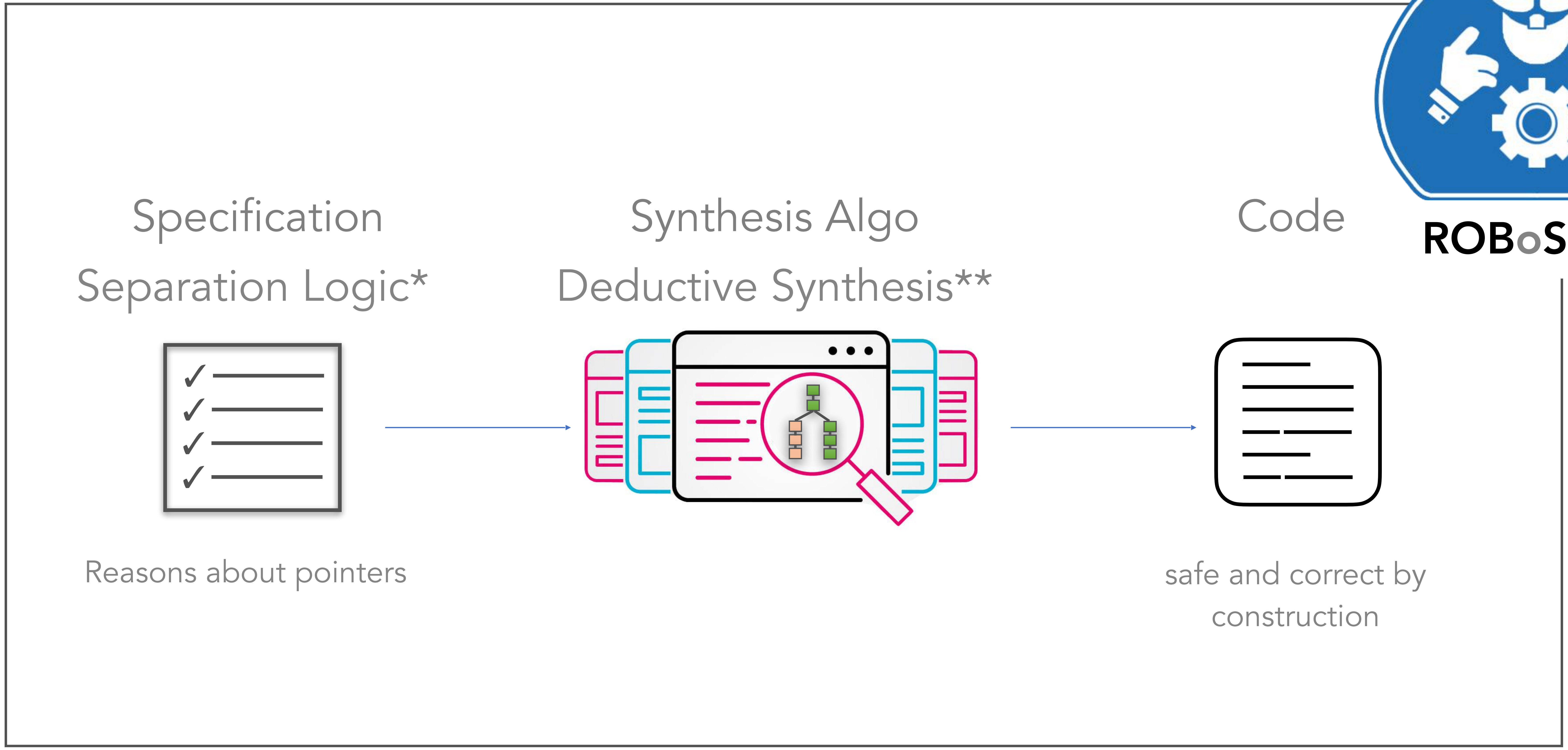
Synthesis Algo

Deductive Synthesis**

Code

safe and correct by construction

BoSSL: Borrowing Synthetic Separation Logic



Specification

Separation Logic*

Synthesis Algo

Deductive Synthesis**

Code

Reasons about pointers

safe and correct by construction

ROBoSuSLik

https://github.com/TyGuS/robosuslik

Synthesis of Programs with Pointers via Read-Only Specifications

(our contribution)

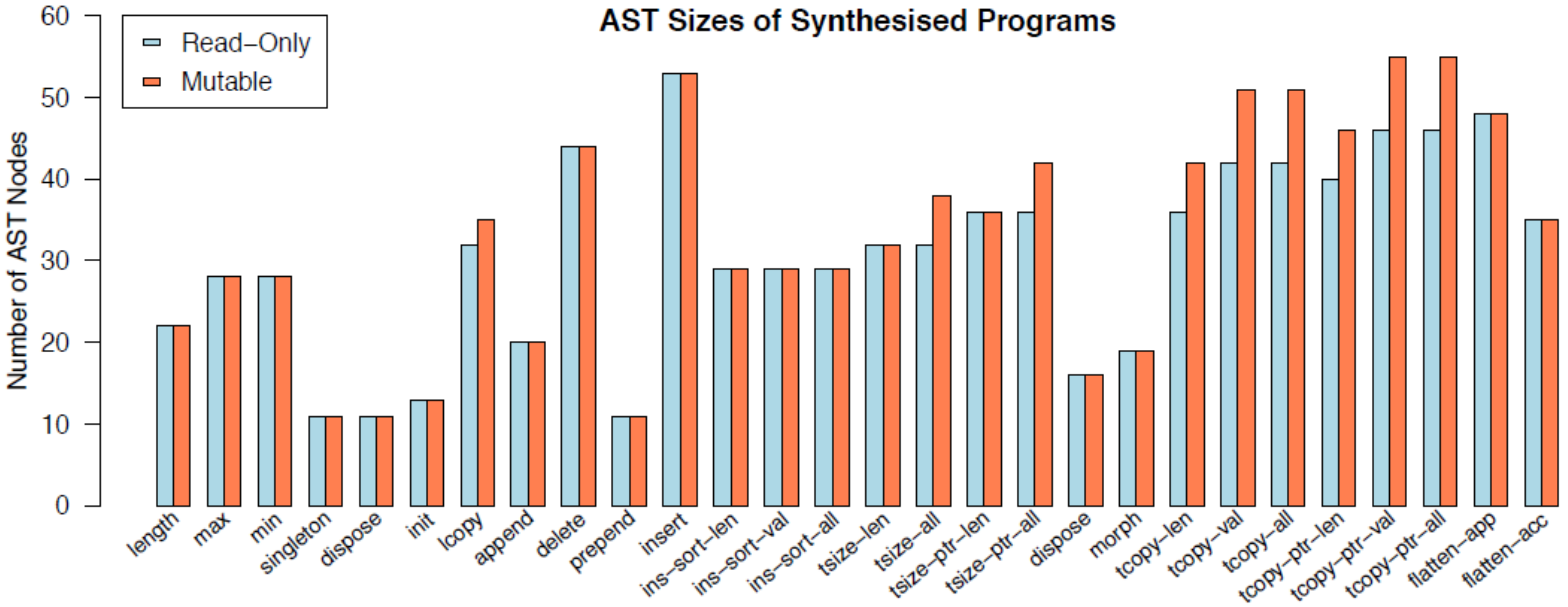
Effective:	more natural and shorter programs
Efficient:	smaller search space—faster synthesis
Robust:	better performance in “worst case scenarios”

Synthesis of Programs with Pointers via Read-Only Specifications

(our contribution)

Effective:	more natural and shorter programs
Efficient:	smaller search space—faster synthesis
Robust:	better performance in "worst case scenarios"

Results 1 – AST size



Synthesis of Programs with Pointers via Read-Only Specifications

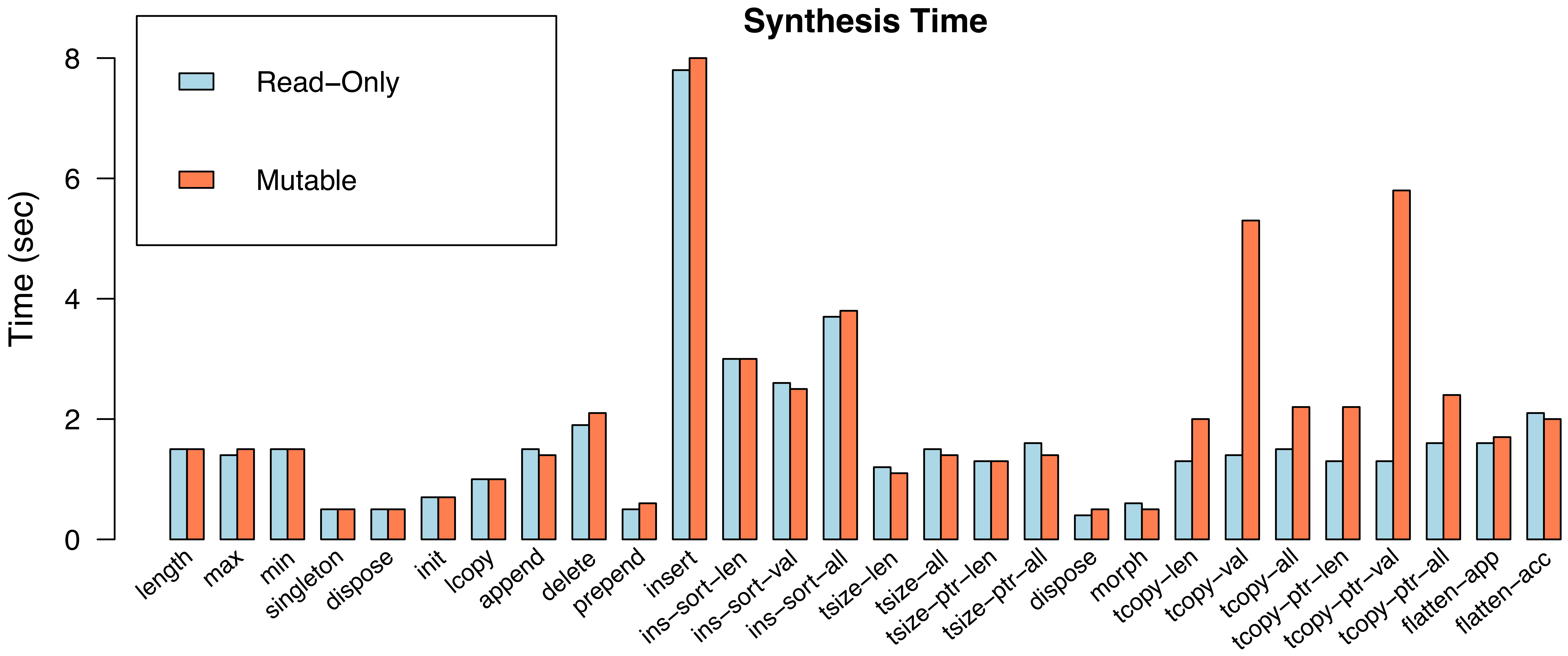
(our contribution)

Effective: more natural and shorter programs

Efficient: smaller search space—faster synthesis

Robust: better performance in "worst case scenarios"

Results 2 – Synthesis time

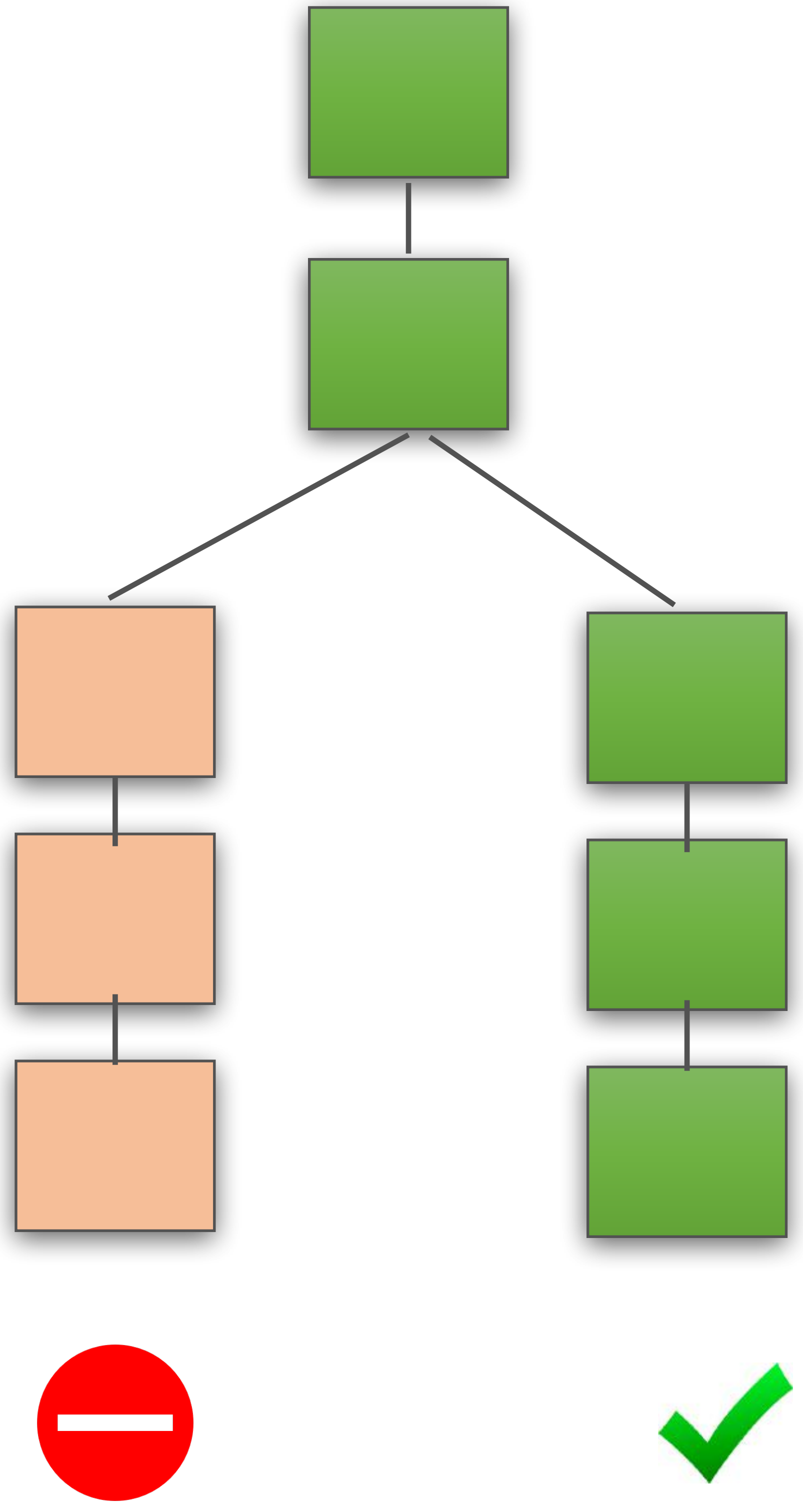


Synthesis of Programs with Pointers via Read-Only Specifications

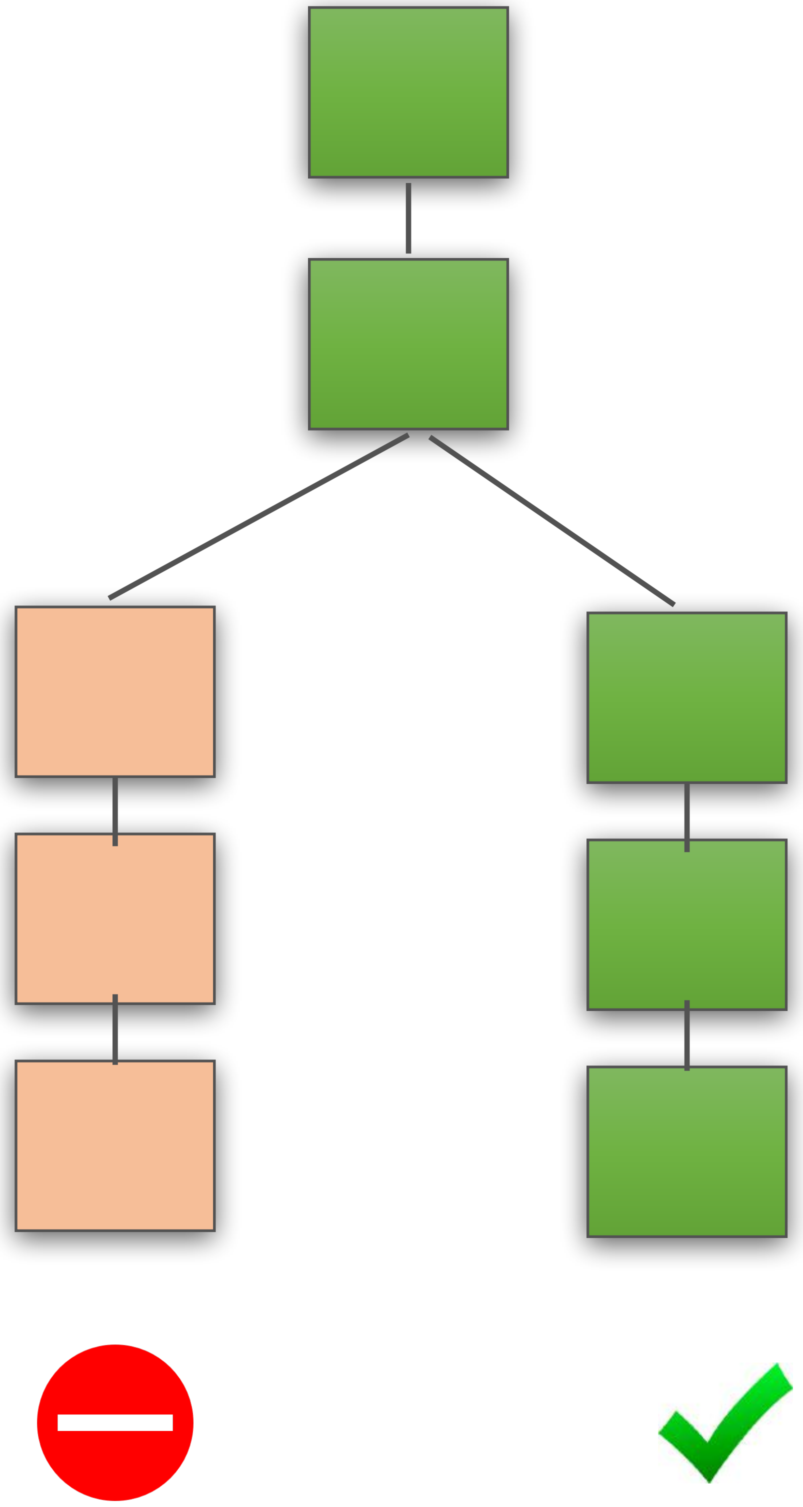
(our contribution)

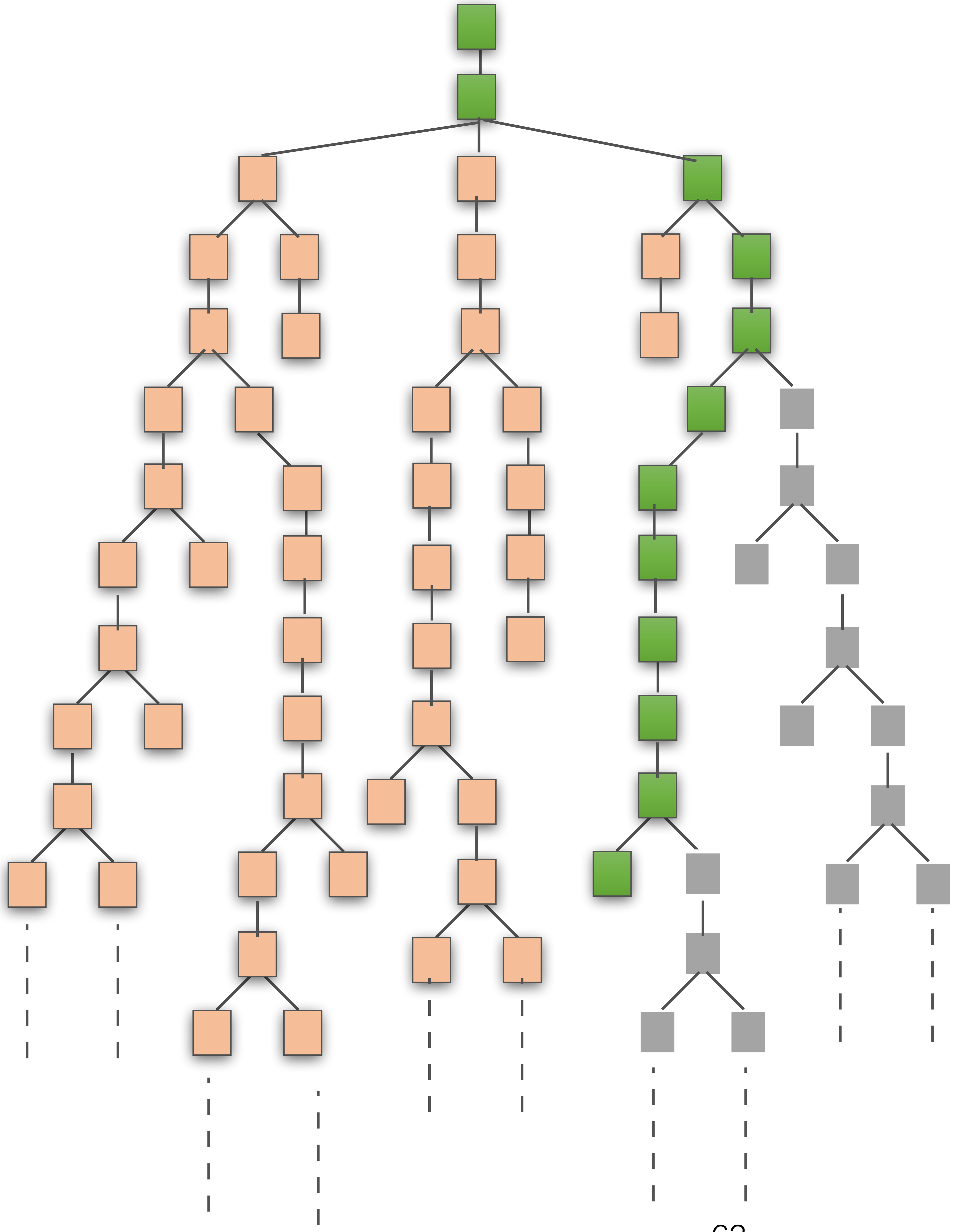
Effective:	more natural and shorter programs
Efficient:	smaller search space—faster synthesis
Robust:	better performance in “worst case scenarios”

Robustness



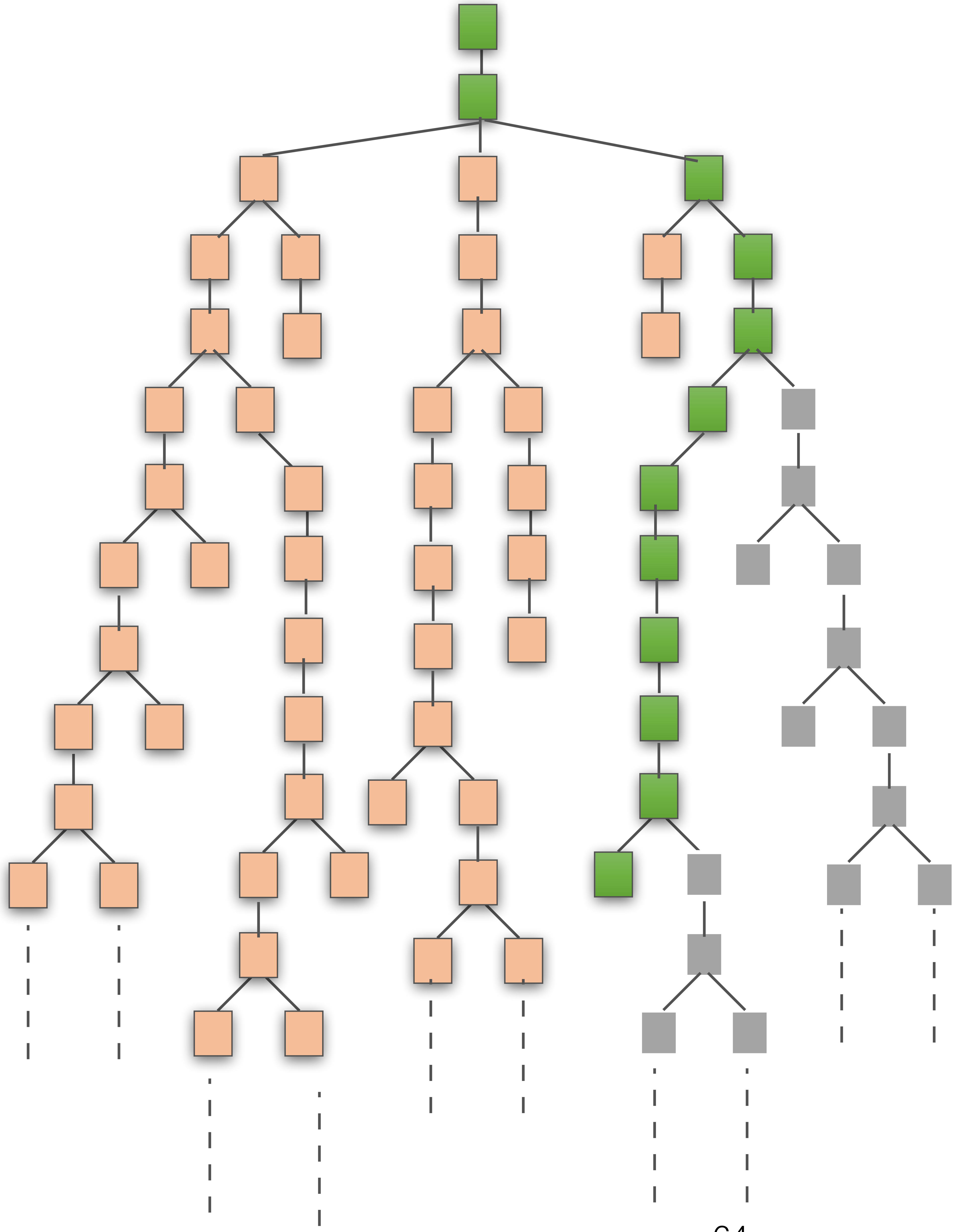
Robustness





Robustness

Is ROBoSuSLik always outperforming SuSLik irrespective of the employed search heuristic?

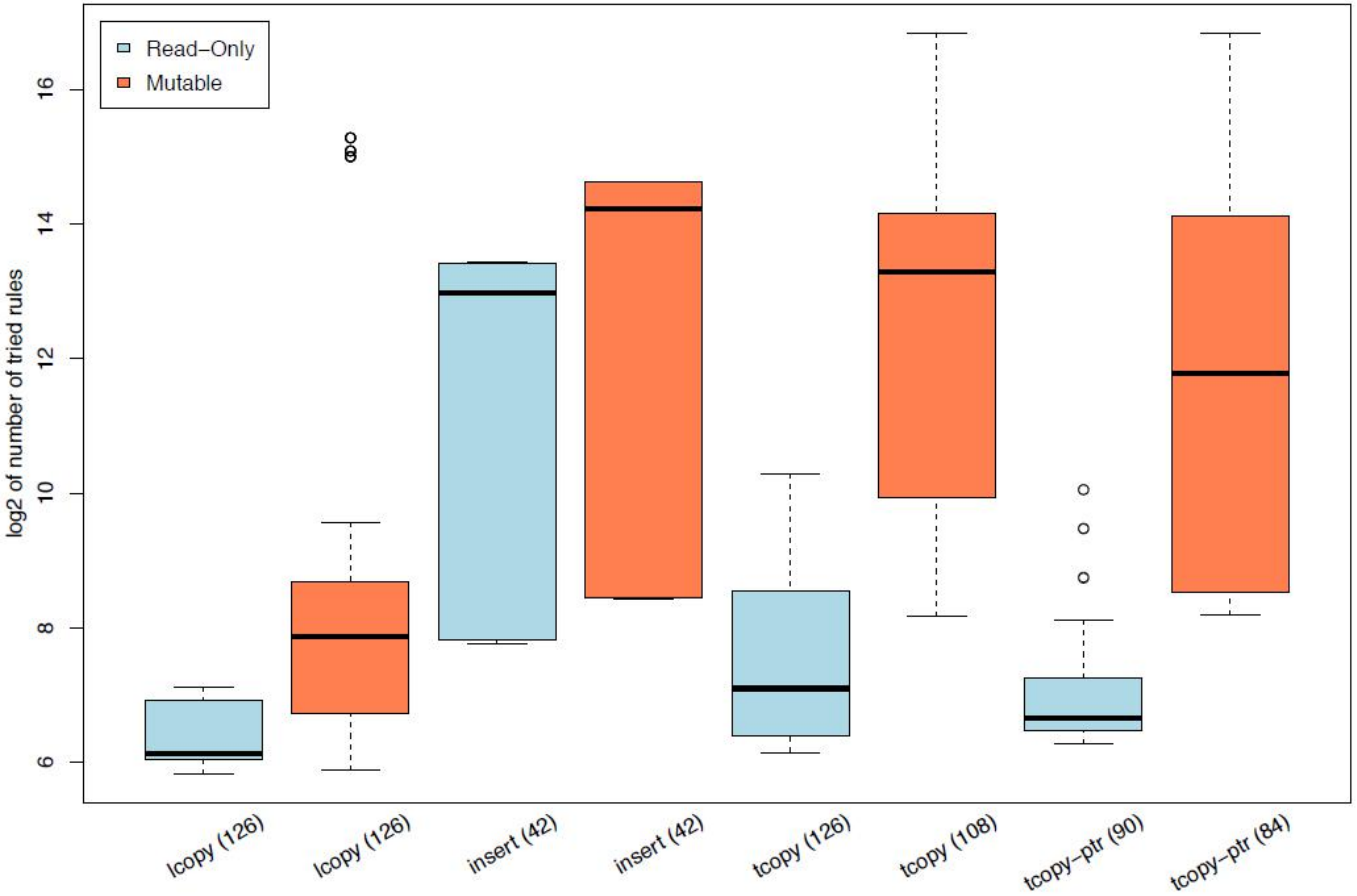


Robustness - serach heuristics variations

We explored:

- 3 variants of specification
- 6 different unification orders strategies
- 7 different search strategies

Results 3 – No of fired rules



the shorter the boxplots the better

Synthesis of Programs with Pointers via Read-Only Specifications

(our contribution)

Effective:	more natural and shorter programs
Efficient:	smaller search space—faster synthesis
Robust:	better performance in "worst case scenarios"

Read-Only Specifications: Related Work

Fractional Permissions [Boyland 2003]

Chalice [Leino et al. 2009],
Verifast [Jacobs et al. 2011]

Abstract permissions [Heule et al., 2013]

Viper [Muller et al. 2016]

Immutable Specifications [David et al. 2011]

Read-Only Assertions [Chargueraud et al. 2017]

Disjoint Permissions [Bach et al. 2018]

Tailored for verification,
not for synthesis!

Synthesis of Programs with Pointers via Read-Only Specifications

(our contribution)

Effective: more natural and shorter programs

Efficient: smaller search space—faster synthesis

Robust: better performance in “worst case scenarios”

Thank You!

To Take-Away

Adding borrows to SSL improves the synthesis efficiency:

synthesised programs of better quality

improved synthesis performance

stronger correctness guarantees

robust synthesis

Thank You!

References

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierho. A type system for borrowing permissions. In POPL, pages 557 - 570. ACM, 2012.

John Boyland. Checking Interference with Fractional Permissions. In SAS, volume 2694 of LNCS, pages 55 - 72. Springer, 2003.

K. Rustan M. Leino and Peter Muller. A Basis for Verifying Multi-threaded Programs. In ESOP, volume 5502 of LNCS, pages 378-393. Springer, 2009.

K. Rustan M. Leino, Peter Muller, and Jan Smans. Verification of Concurrent Programs with Chalice. In Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures, volume 5705 of LNCS, pages 195-222. Springer, 2009.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frederic Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In NASA Formal Methods, volume 6617 of LNCS, pages 41-55. Springer, 2011.

Peter Muller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In VMCAI, volume 9583 of LNCS, pages 41-62. Springer, 2016.

Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In OOPSLA, pages 359 - 374. ACM, 2011.

Arthur Chargueraud and Francois Pottier. Temporary Read-Only Permissions for Separation Logic. In ESOP, volume 10201 of LNCS, pages 260 - 286. Springer, 2017.

Xuan Bach Le and Aquinas Hobor. Logical reasoning for disjoint permissions. InESOP, volume 10801 of LNCS, pages 385{414. Springer, 2018.

Proof Search Algorithm

- Goal-driven, with *backtracking* (in CPS), trying a fixed set of rules;
- *Branching*: some rules emit many alternatives;
- Along with the program, emits the ***complete proof tree.***
- ***Optimisations***: Invertible Rules (*cf. Focusing* in Proof Theory),
- phased search, “Early Failure” rules

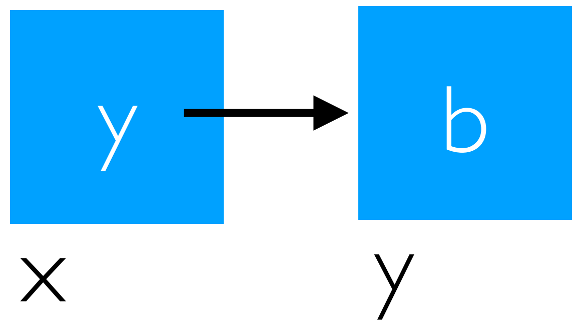
Separation Logic

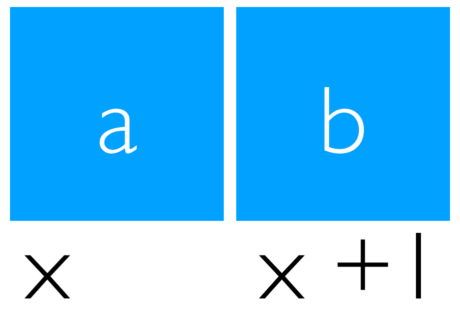
starting in a state that satisfies **P**,
program **c** will execute *without memory errors*, and
upon its termination the state will satisfy **Q.**

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

empty heap	$\{ \text{emp} \}$
singleton heap	$\{ x \mapsto a \}$
separating conjunction	$\{ x \mapsto y * y \mapsto b \}$
memory block	$\{ [x,2] * x \mapsto a * (x+1) \mapsto b \}$
pure constraints	$\{ a > 0 ; x \mapsto a \}$

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(e1, …, en)
sequential composition	c1; c2
conditional	if (e) {c1} else {c2}





Read-Only Specifications: Related Work

Fractional Permissions [Boyland 2003]

Chalice [Leino et al. 2009],
Verifast [Jacobs et al. 2011]

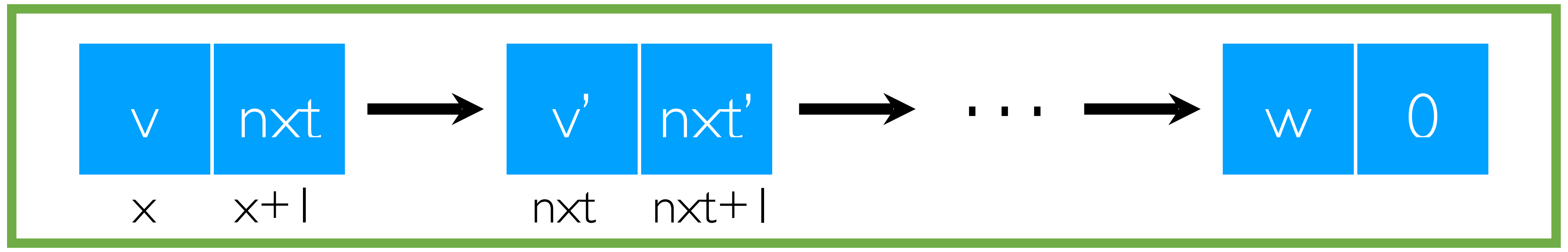
Abstract permissions [Heule et al., 2013]

Viper [Muller et al. 2016]

Immutable Specifications [David et al. 2011]

Read-Only Assertions [Chargueraud et al. 2017]

Tailored for verification,
not for synthesis!



Example

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

void listcopy (loc r)

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

!

R: Add RO permissions.

Example

$\{r \mapsto x * ls(x, S)[RO,RO] \}$

void listcopy (loc r)

$\{r \mapsto y * ls(x, S)[RO,RO] * ls(y, S)[M,M] \}$

!

R: Add RO permissions.

Example

{r ↦ x * ls(x, S)[RO,RO] }

void listcopy (loc r)

{r ↦ y * ls(x, S)[RO,RO] * ls(y, S)[M,M] }

Example

{r ↦ x * ls(x, S)[RO,RO] }

void listcopy (loc r)

{r ↦ y * ls(x, S)[RO,RO] * ls(y, S)[M,M] }

// … <caller>…:

// z ↦ x’ * ls(x’, S’)[M,M]

listcopy(z) ⛔

// z ↦ y * ls(x’, S’)[RO,RO] * ls(y, S’)[M,M]

Example

{r ↦ x * ls(x, S)[a,b] }

void listcopy (loc r)

{r ↦ y * ls(x, {0})[a,b] * ls(y, S)[M,M] }

// … <caller>…:

// r ↦ x * ls(x, S)[M,M]

listcopy(z)

// r ↦ x * ls(x, S)[M,M]

Setup

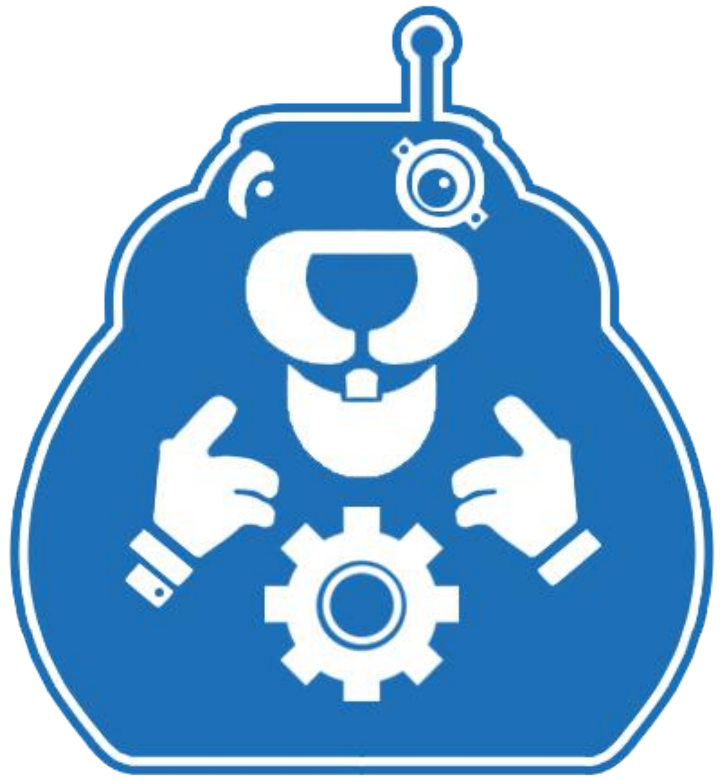
Varied the properties captured in the inductive definitions.

Applied 42 kinds of *perturbations* to stress the proof search strategy.

SuSLik -> ROBoSuSLik

https://github.com/TyGuS/suslik

https://github.com/TyGuS/suslik/tree/borrows



(**S**ynthesis **u**sing **S**eparation **L**og**ik**)[1]

(**R**ead-**O**nly **Bo**rrows for **S**ynthesis **u**sing **S**eparation **L**og**ik**)[2]

[1] [Polikarpova & Sergey @POPL’19]

[2] [Costea, Zhu, Polikarpova,Sergey @ESOP’20]

SSL: basic rules

(Emp)

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

SSL: basic rules

(Emp)

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

SSL: basic rules

$$\text{(Read)}\quad \frac{}{\{\, x \mapsto A * P \,\} \rightsquigarrow \{\, Q \,\} \mid}$$

SSL: basic rules

(Read)

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

SSL: basic rules

(Read)

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

(Write)

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

SSL: basic rules

(Read)

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

(Write)

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

SSL: basic rules

(Frame)

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

(UnifyHeaps)

$$\frac{[\sigma]R' = R \qquad \{\, P * R \,\} \rightsquigarrow [\sigma]\{\, Q * R' \,\} \mid c}{\{\, P * R \,\} \rightsquigarrow \{\, Q * R' \,\} \mid c}$$

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

$$\{ x \mapsto a \ast y \mapsto b \} \rightsquigarrow \{ x \mapsto z \ast y \mapsto z \} \quad \Big| \quad \texttt{??}$$

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

$$\dfrac{\dfrac{\dfrac{\sigma = [a2/z] \qquad \{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto a2 * y \mapsto a2 \} \mid \texttt{??}}{\{ x \mapsto a2 * y \mapsto b2 \} \rightsquigarrow \{ x \mapsto z * y \mapsto z \} \mid \texttt{??}}\,\text{(UnifyHeaps)}}{\{ x \mapsto a2 * y \mapsto \boldsymbol{b} \} \rightsquigarrow \{ x \mapsto z * y \mapsto z \} \mid \texttt{let b2 = *y; ??}}\,\text{(Read)}}{\{ x \mapsto \boldsymbol{a} * y \mapsto \boldsymbol{b} \} \rightsquigarrow \{ x \mapsto z * y \mapsto z \} \mid \texttt{let a2 = *x; ??}}\,\text{(Read)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\, y \mapsto b2 \,\} \rightsquigarrow \{\, y \mapsto a2 \,\} \mid \texttt{??}}{\sigma = [a2/z] \qquad \{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto a2 * y \mapsto a2 \,\} \mid \texttt{??}}\text{(Frame)}}{\{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto z * y \mapsto z \,\} \mid \texttt{??}}\text{(UnifyHeaps)}}{\{\, x \mapsto a2 * y \mapsto \boldsymbol{b} \,\} \rightsquigarrow \{\, x \mapsto z * y \mapsto z \,\} \mid \texttt{let b2 = *y; ??}}\text{(Read)}}{\{\, x \mapsto \boldsymbol{a} * y \mapsto \boldsymbol{b} \,\} \rightsquigarrow \{\, x \mapsto z * y \mapsto z \,\} \mid \texttt{let a2 = *x; ??}}\text{(Read)}$$

$$\begin{array}{c}
\{\, y \mapsto a2 \,\} \leadsto \{\, y \mapsto a2 \,\} \mid \texttt{??} \\
\hline
\{\, y \mapsto b2 \,\} \leadsto \{\, y \mapsto a2 \,\} \mid \texttt{*y = a2; ??} \quad (\text{Write}) \\
\hline
\sigma = [a2/z] \quad \{\, x \mapsto a2 * y \mapsto b2 \,\} \leadsto \{\, x \mapsto a2 * y \mapsto a2 \,\} \mid \texttt{??} \quad (\text{Frame}) \\
\hline
\{\, x \mapsto a2 * y \mapsto b2 \,\} \leadsto \{\, x \mapsto z * y \mapsto z \,\} \mid \texttt{??} \quad (\text{UnifyHeaps}) \\
\hline
\{\, x \mapsto a2 * y \mapsto \boldsymbol{b} \,\} \leadsto \{\, x \mapsto z * y \mapsto z \,\} \mid \texttt{let b2 = *y; ??} \quad (\text{Read}) \\
\hline
\{\, x \mapsto \boldsymbol{a} * y \mapsto \boldsymbol{b} \,\} \leadsto \{\, x \mapsto z * y \mapsto z \,\} \mid \texttt{let a2 = *x; ??} \quad (\text{Read})
\end{array}$$

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

$$\frac{}{\{\ y \mapsto \mathrm{a2}\ \} \rightsquigarrow \{\ y \mapsto \mathrm{a2}\ \} \mid \texttt{??}}\ \text{(Frame)}$$

$$\frac{}{\{\ y \mapsto \mathrm{b2}\ \} \rightsquigarrow \{\ y \mapsto \mathrm{a2}\ \} \mid \texttt{*y = a2; ??}}\ \text{(Write)}$$

$$\frac{}{\sigma = [\mathrm{a2}/z] \quad \{\ x \mapsto \mathrm{a2} * y \mapsto \mathrm{b2}\ \} \rightsquigarrow \{\ x \mapsto \mathrm{a2} * y \mapsto \mathrm{a2}\ \} \mid \texttt{??}}\ \text{(Frame)}$$

$$\frac{}{\{\ x \mapsto \mathrm{a2} * y \mapsto \mathrm{b2}\ \} \rightsquigarrow \{\ x \mapsto z * y \mapsto z\ \} \mid \texttt{??}}\ \text{(UnifyHeaps)}$$

$$\frac{}{\{\ x \mapsto \mathrm{a2} * y \mapsto \boldsymbol{b}\ \} \rightsquigarrow \{\ x \mapsto z * y \mapsto z\ \} \mid \texttt{let b2 = *y; ??}}\ \text{(Read)}$$

$$\frac{}{\{\ x \mapsto \boldsymbol{a} * y \mapsto \boldsymbol{b}\ \} \rightsquigarrow \{\ x \mapsto z * y \mapsto z\ \} \mid \texttt{let a2 = *x; ??}}\ \text{(Read)}$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\ \mathrm{emp}\ \} \rightsquigarrow \{\ \mathrm{emp}\ \} \mid \texttt{skip}}\,(\text{Emp})}{\{\ \mathrm{y} \mapsto \mathrm{a2}\ \} \rightsquigarrow \{\ \mathrm{y} \mapsto \mathrm{a2}\ \} \mid \texttt{??}}\,(\text{Frame})}{\{\ \mathrm{y} \mapsto \mathrm{b2}\ \} \rightsquigarrow \{\ \mathrm{y} \mapsto \mathrm{a2}\ \} \mid \boxed{\texttt{*y = a2;}}\ \texttt{??}}\,(\text{Write})}{\sigma = [\mathrm{a2}/\mathrm{z}] \quad \{\ \mathrm{x} \mapsto \mathrm{a2} \ast \mathrm{y} \mapsto \mathrm{b2}\ \} \rightsquigarrow \{\ \mathrm{x} \mapsto \mathrm{a2} \ast \mathrm{y} \mapsto \mathrm{a2}\ \} \mid \texttt{??}}\,(\text{Frame})}{\{\ \mathrm{x} \mapsto \mathrm{a2} \ast \mathrm{y} \mapsto \mathrm{b2}\ \} \rightsquigarrow \{\ \mathrm{x} \mapsto \mathrm{z} \ast \mathrm{y} \mapsto \mathrm{z}\ \} \mid \texttt{??}}\,(\text{UnifyHeaps})}{\{\ \mathrm{x} \mapsto \mathrm{a2} \ast \mathrm{y} \mapsto b\ \} \rightsquigarrow \{\ \mathrm{x} \mapsto \mathrm{z} \ast \mathrm{y} \mapsto \mathrm{z}\ \} \mid \boxed{\texttt{let b2 = *y;}}\ \texttt{??}}\,(\text{Read})}{\{\ \mathrm{x} \mapsto a \ast \mathrm{y} \mapsto b\ \} \rightsquigarrow \{\ \mathrm{x} \mapsto \mathrm{z} \ast \mathrm{y} \mapsto \mathrm{z}\ \} \mid \boxed{\texttt{let a2 = *x;}}\ \texttt{??}}\,(\text{Read})
$$