Software Verification (Autumn 2014)
Lecture 5: Separation Logic
Part III

Chris Poskitt

Chair of Software Engineering

ETH zürich

In the previous lecture we saw that:

- separation logic is an extension of Hoare logic for shared mutable data structures
- program states are now modelled by variable stores and heaps
- spatial connectives allow assertions to focus on resources used by programs
- frame rule enables local reasoning

Next on the agenda

(1) model of program states for separation logic 

(2) assertions and spatial connectives 

(3) axioms and inference rules 

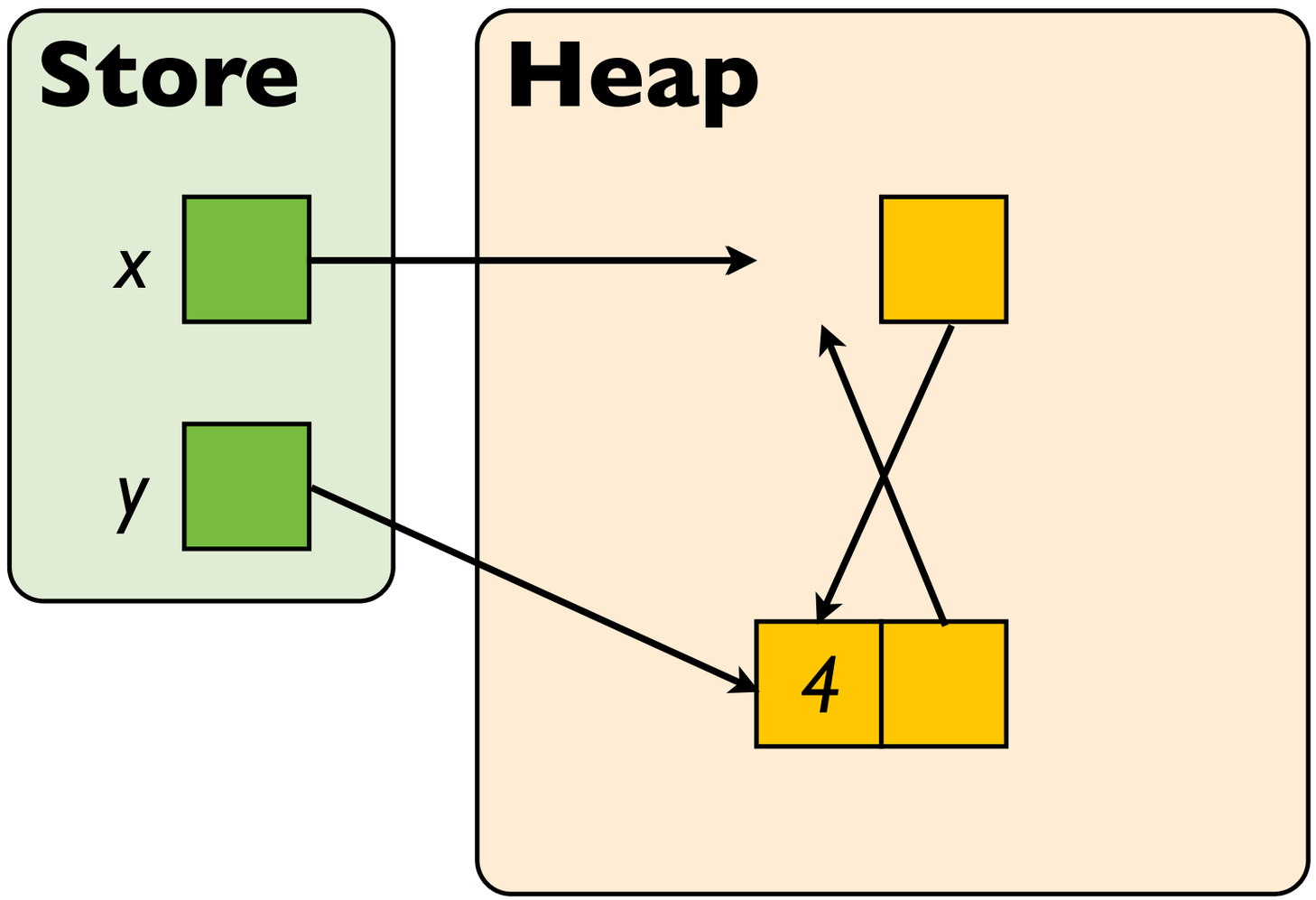
(4) program proofs

Exercise: prove this!

{emp}

x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];

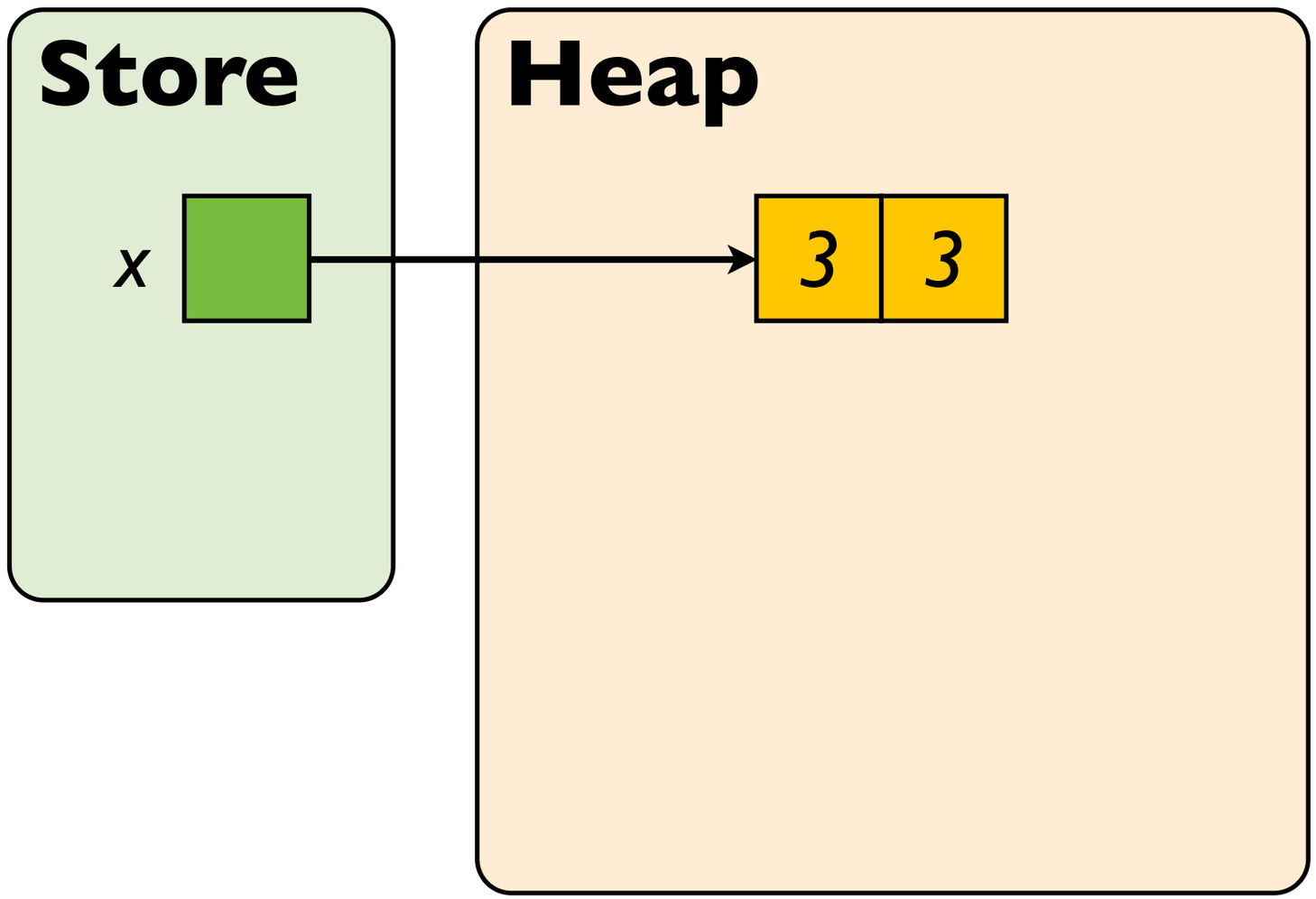
{y|->4 * true}



Proof outline

{emp}

x := cons(3,3);

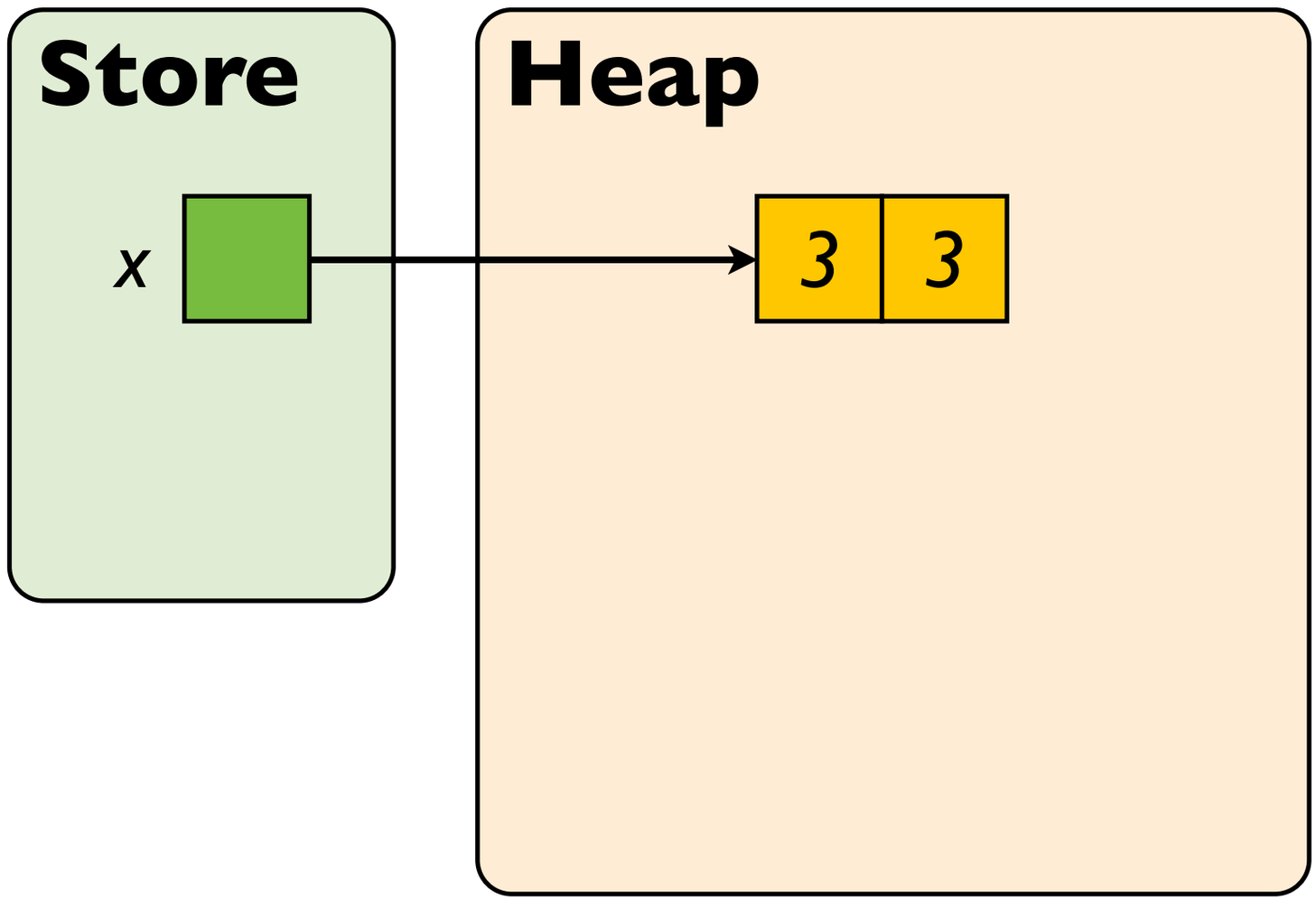


Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}



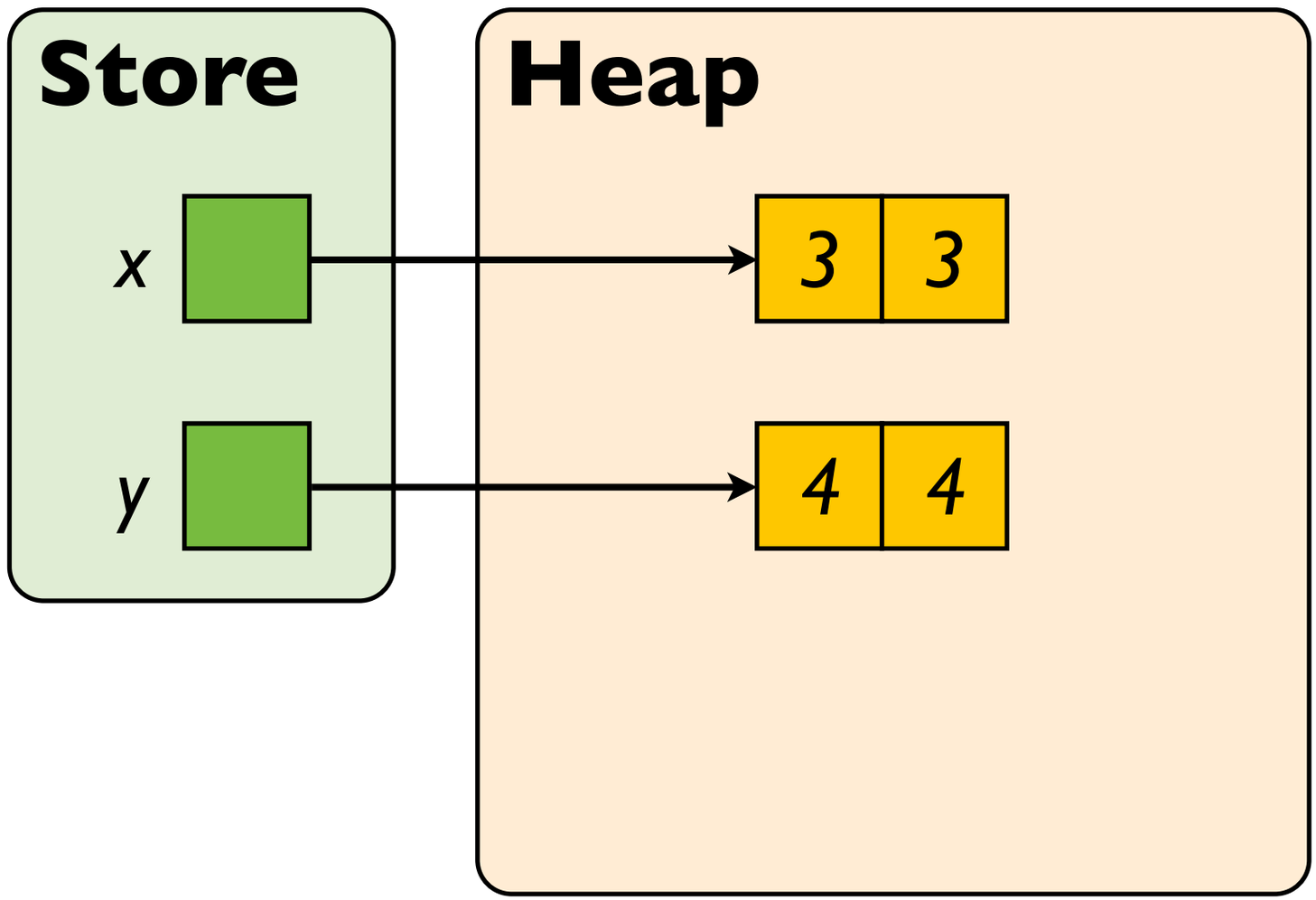
Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}

y := cons(4,4);



Proof outline

{emp}

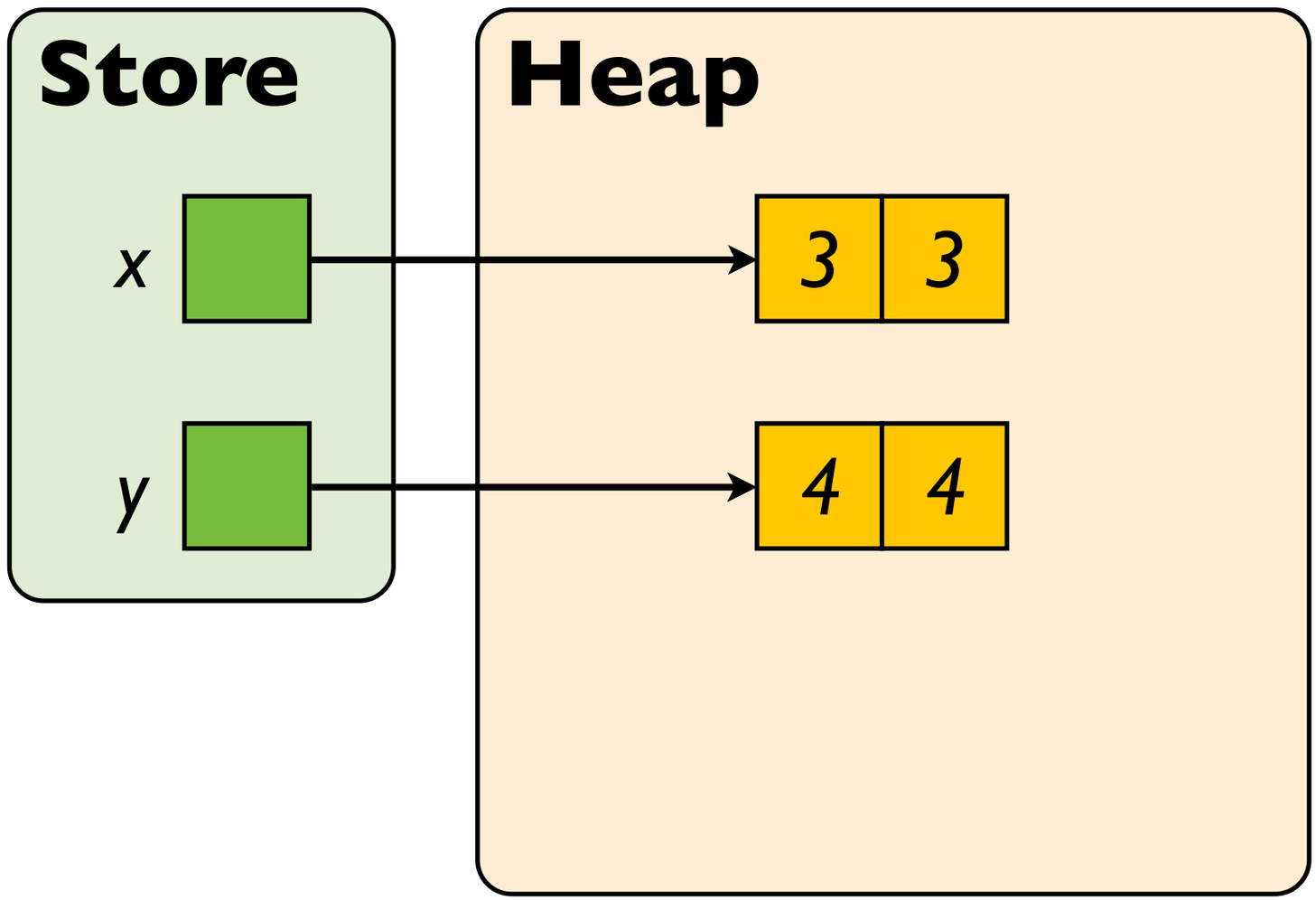
x := cons(3,3);

{x |-> 3,3}

{x |-> 3,3 * emp}

y := cons(4,4);

rule of consequence



Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}

{x |-> 3,3 * emp}

{emp}

y := cons(4,4);

{y |-> 4,4}

frame rule!

{x |-> 3,3 * y |-> 4,4}

Proof outline

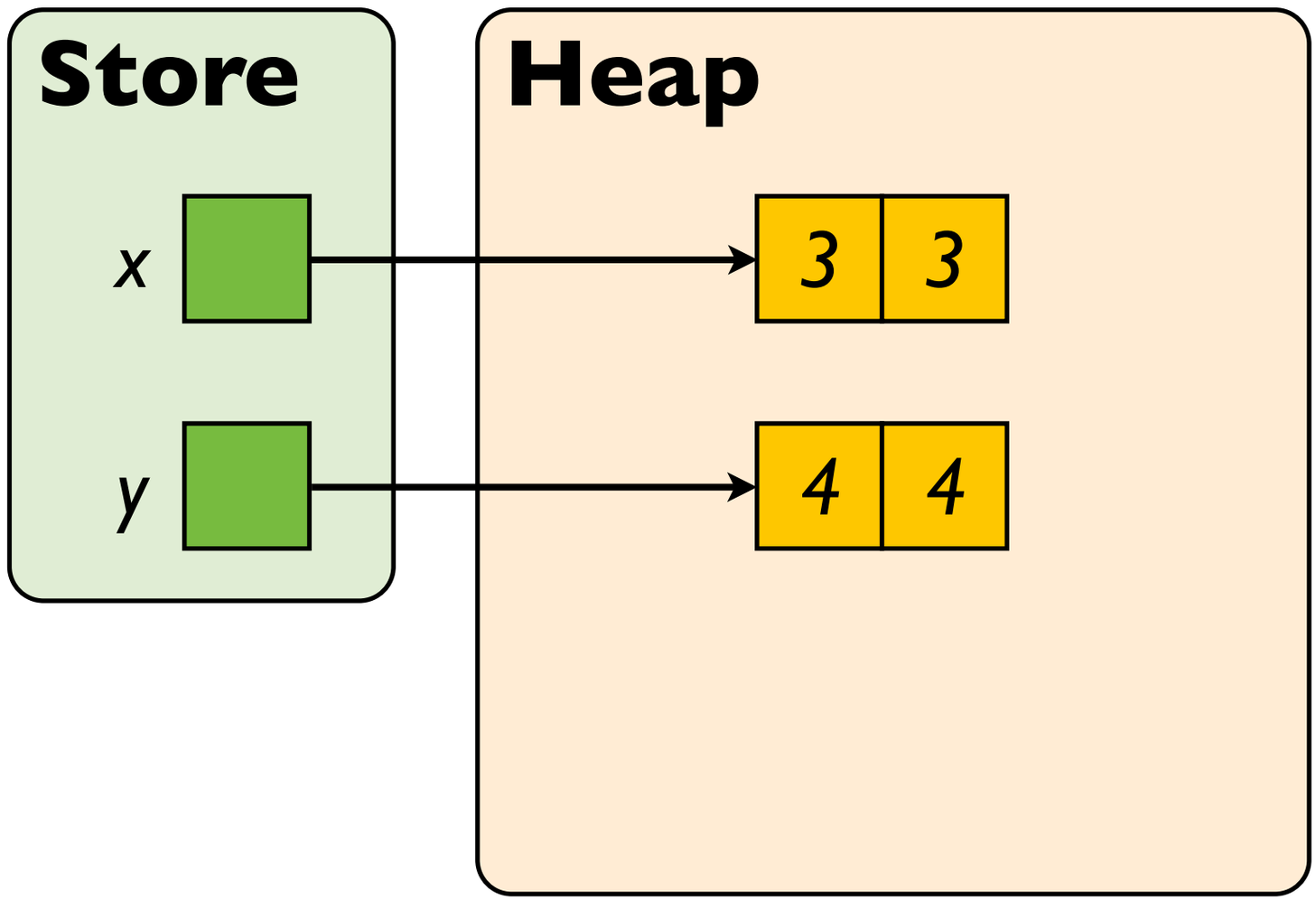
{emp}

x := cons(3,3);

{x |-> 3,3}

y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}



Proof outline

{emp}

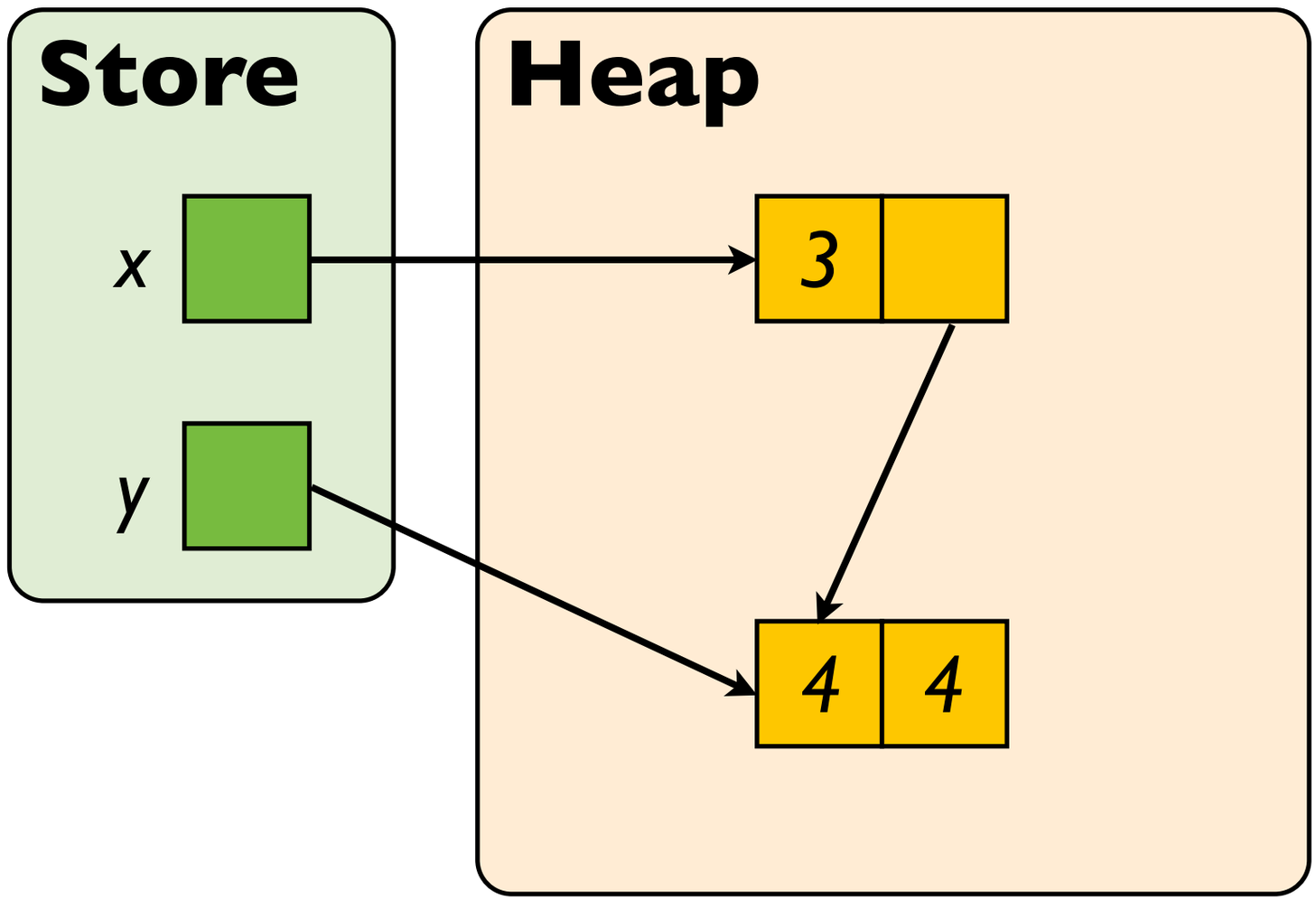
x := cons(3,3);

{x |-> 3,3}

y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}

[x+1] := y;



Proof outline

{emp}

x := cons(3,3);

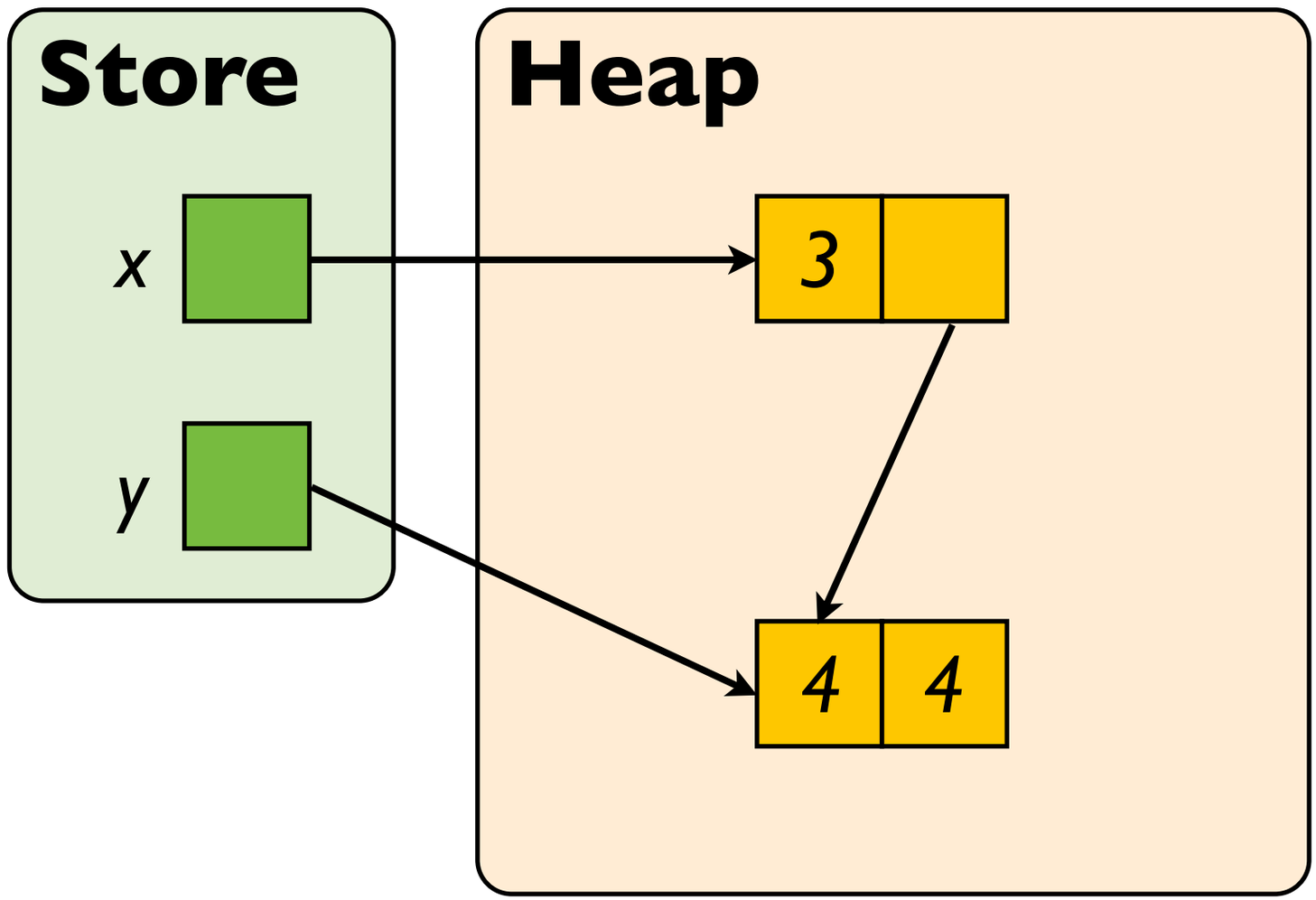
{x |-> 3,3}

y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}

{x |-> 3 * x+1 |-> 3
* y |-> 4,4}

[x+1] := y;



Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}

y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}

{x |-> 3 * x+1 |-> 3
* y |-> 4,4}

[x+1] := y;

{x |-> 3 * x+1 |-> y
* y |-> 4,4}

{x+1 |-> 3}

[x+1] := y;

{x+1 |-> y}

frame rule!

Proof outline

{emp}

x := cons(3,3);

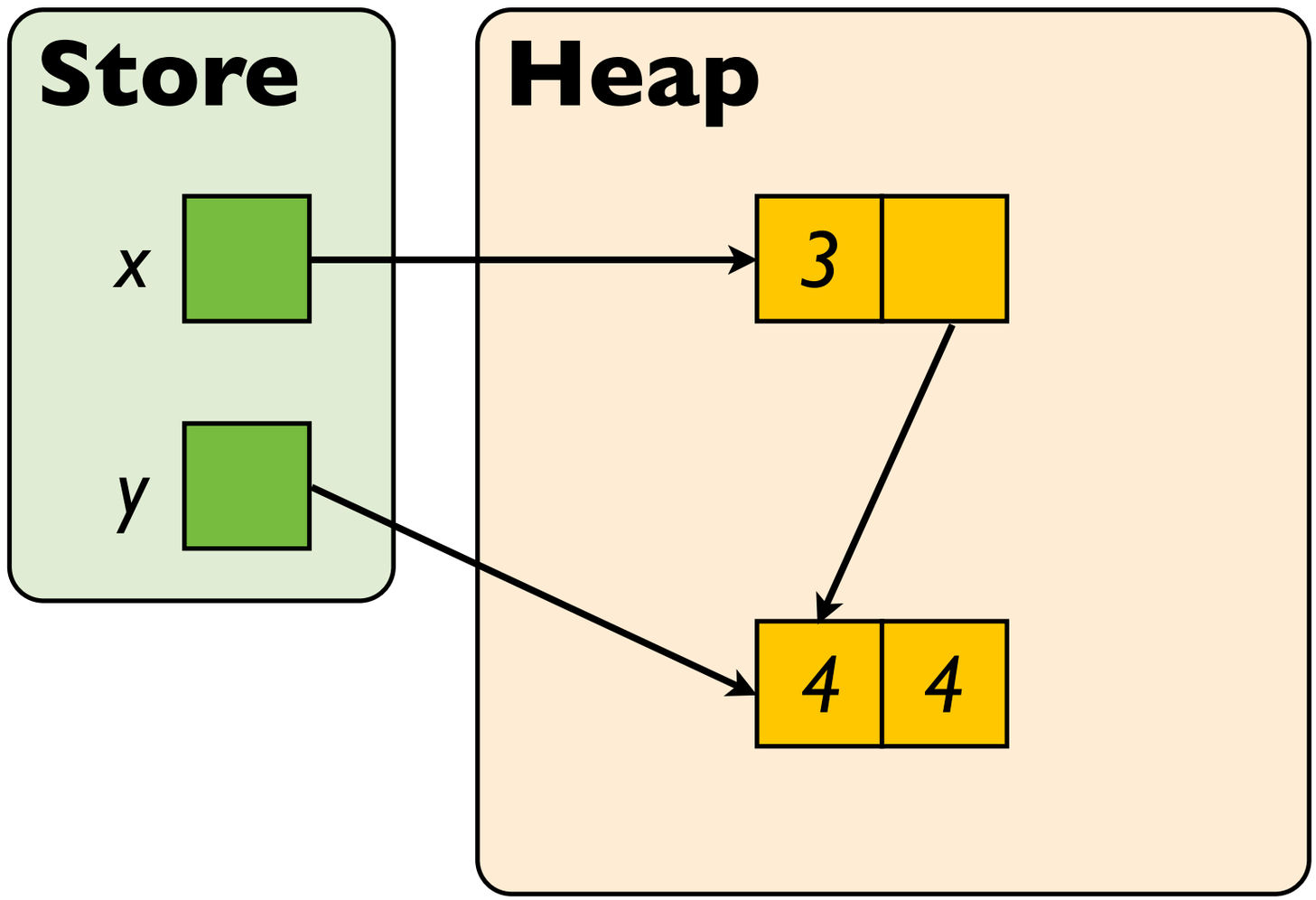
{x |-> 3,3}

y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}

[x+1] := y;

{x |-> 3,y * y |-> 4,4}



Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}

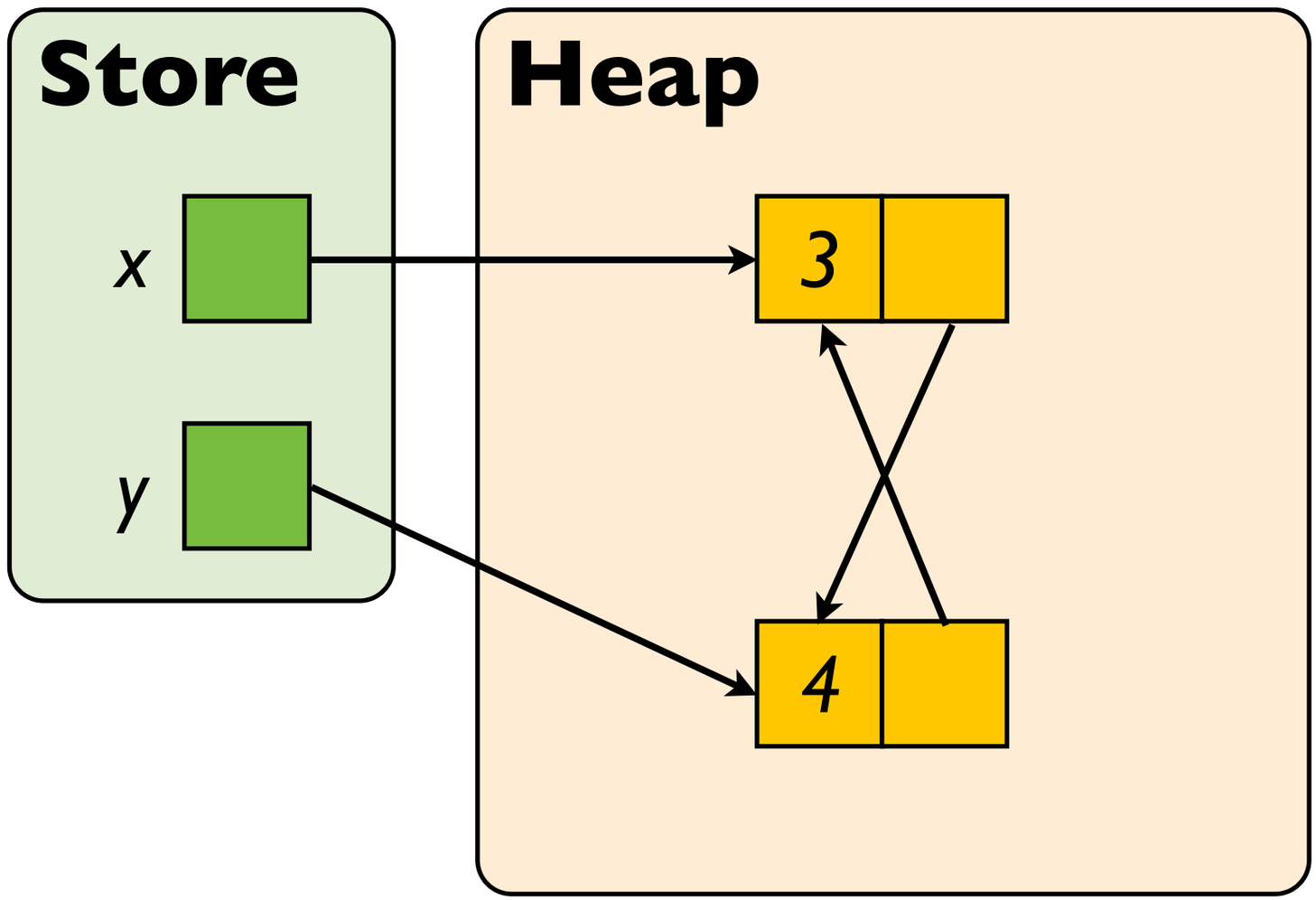
y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}

[x+1] := y;

{x |-> 3,y * y |-> 4,4}

[y+1] := x;



Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}

y := cons(4,4);

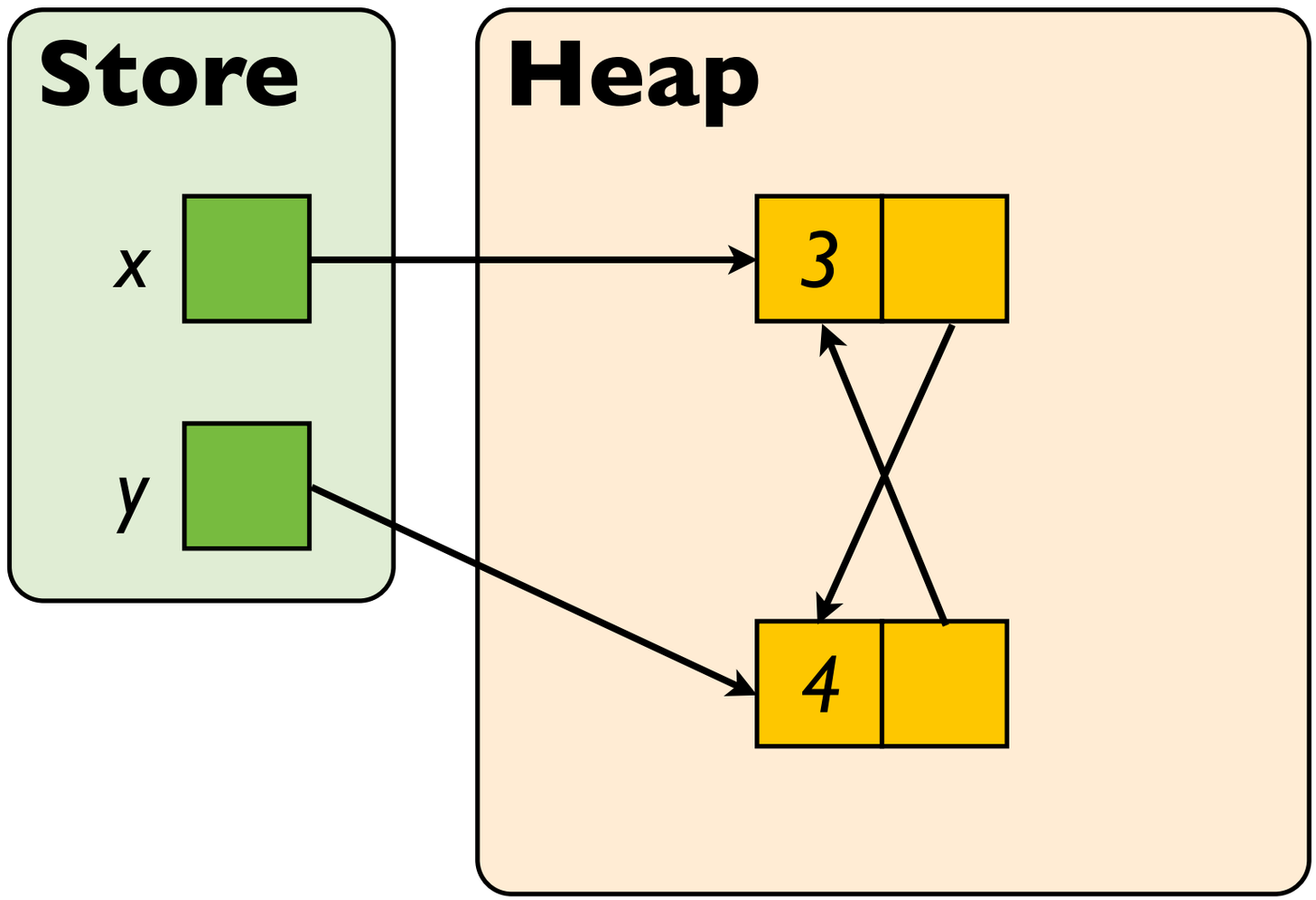
{x |-> 3,3 * y |-> 4,4}

[x+1] := y;

{x |-> 3,y * y |-> 4,4}

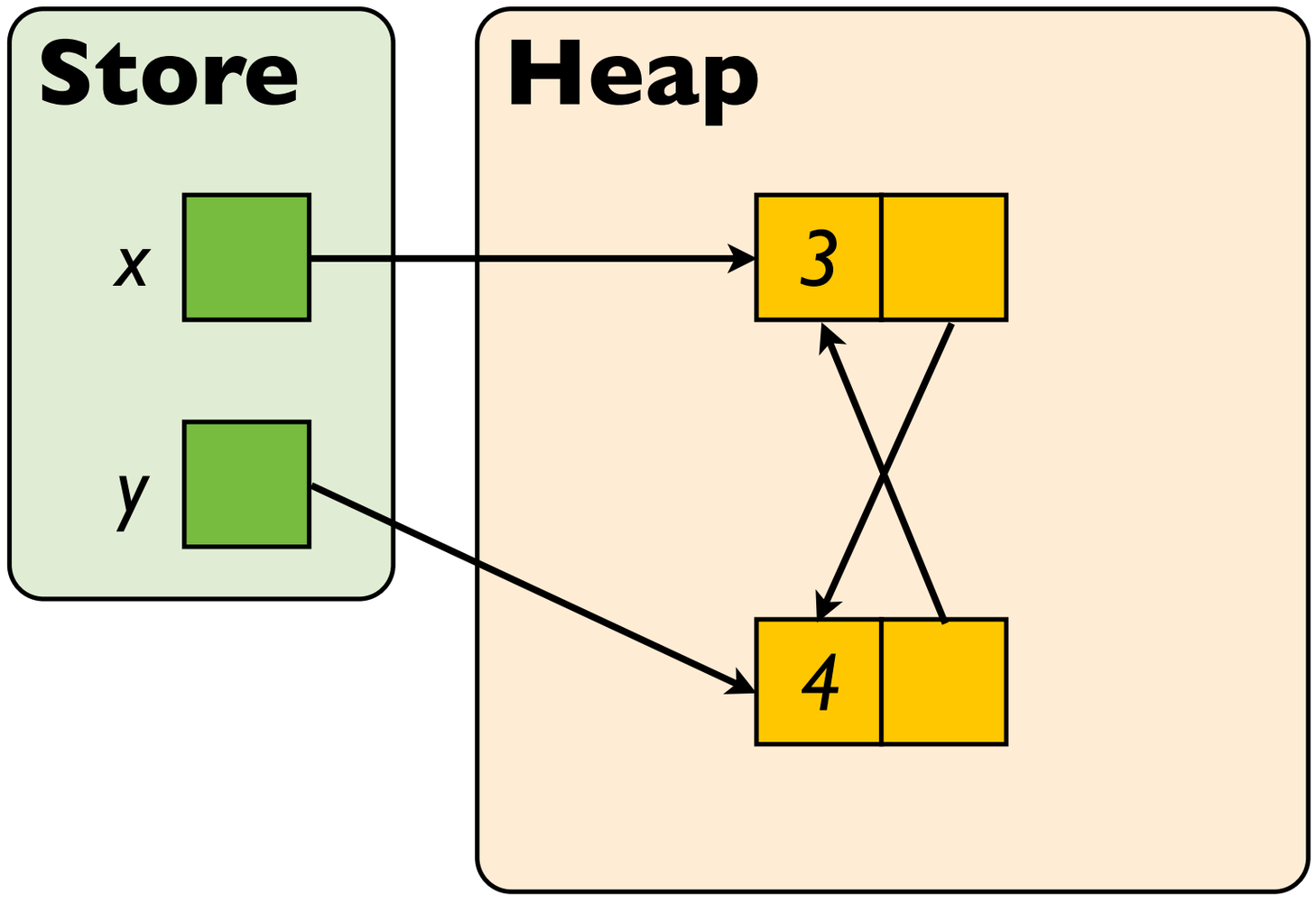
[y+1] := x;

{x |-> 3,y * y |-> 4,x}



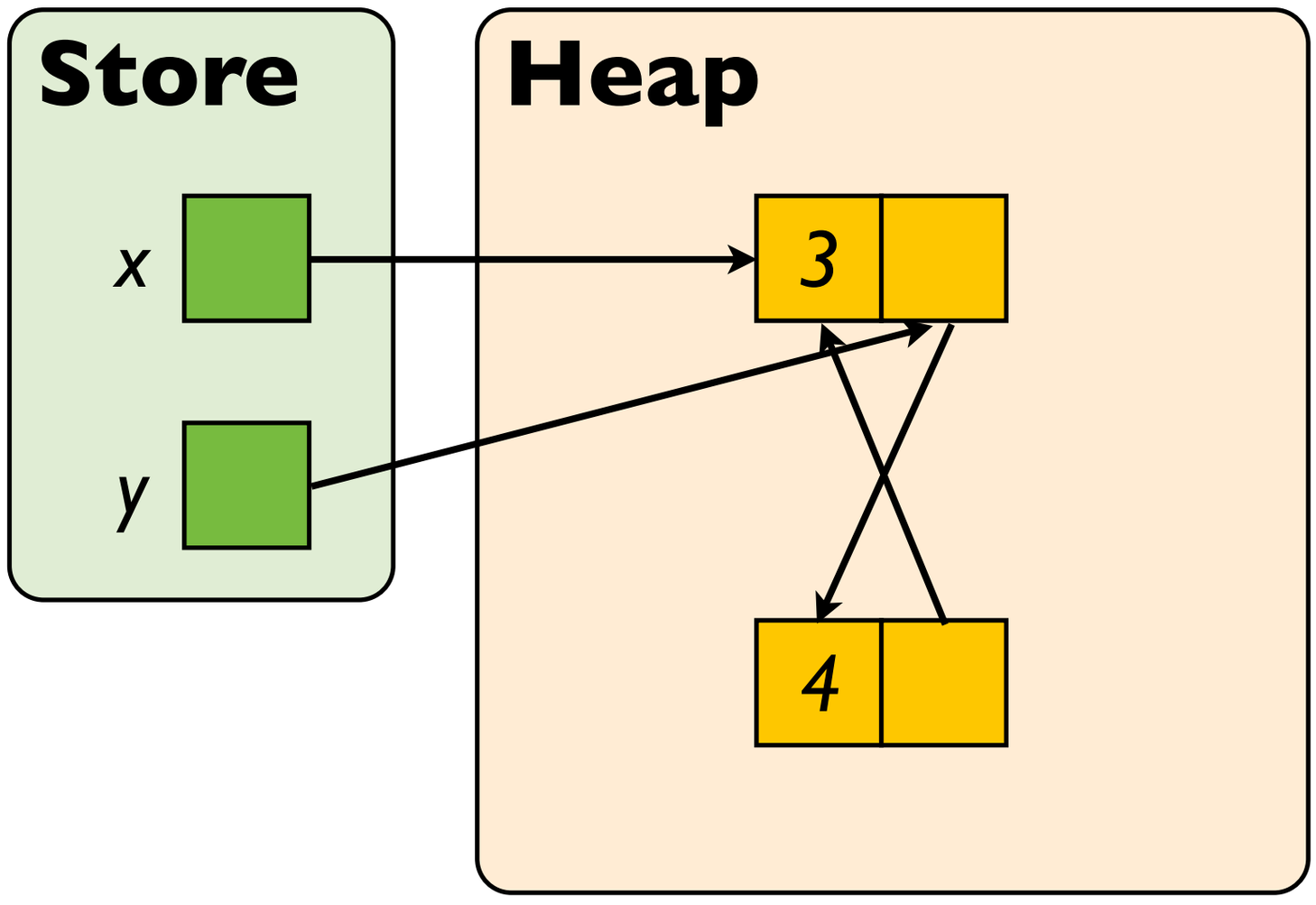
{emp}
  x := cons(3,3);
{x |-> 3,3}
  y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
  [x+1] := y;
{x |-> 3,y * y |-> 4,4}
  [y+1] := x;
{x |-> 3,y * y |-> 4,x}

Proof outline



{emp}
  x := cons(3,3);
{x |-> 3,3}
  y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
  [x+1] := y;
{x |-> 3,y * y |-> 4,4}
  [y+1] := x;
{x |-> 3,y * y |-> 4,x}
  y := x+1;

Proof outline



Proof outline

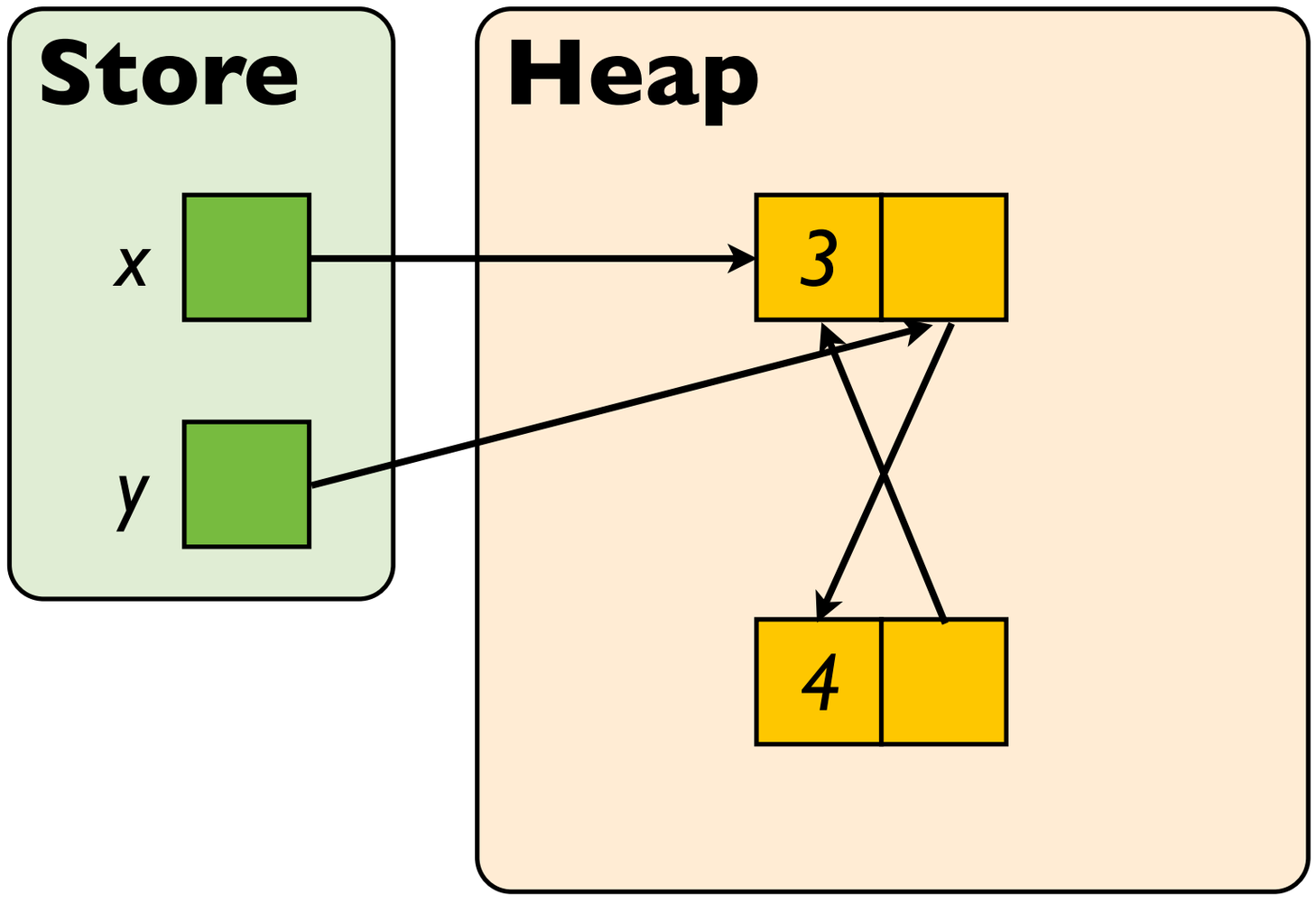
{emp}
  x := cons(3,3);
{x |-> 3,3}
  y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
  [x+1] := y;
{x |-> 3,y * y |-> 4,4}
  [y+1] := x;
{x |-> 3,y * y |-> 4,x}
  y := x+1;

via "forward" assignment axiom (from Hoare logic)

{x |-> 3,y * y |-> 4,x}
  y := x+1
{x |-> 3,y^{old} * y^{old} |-> 4,x ∧ y = x+1}

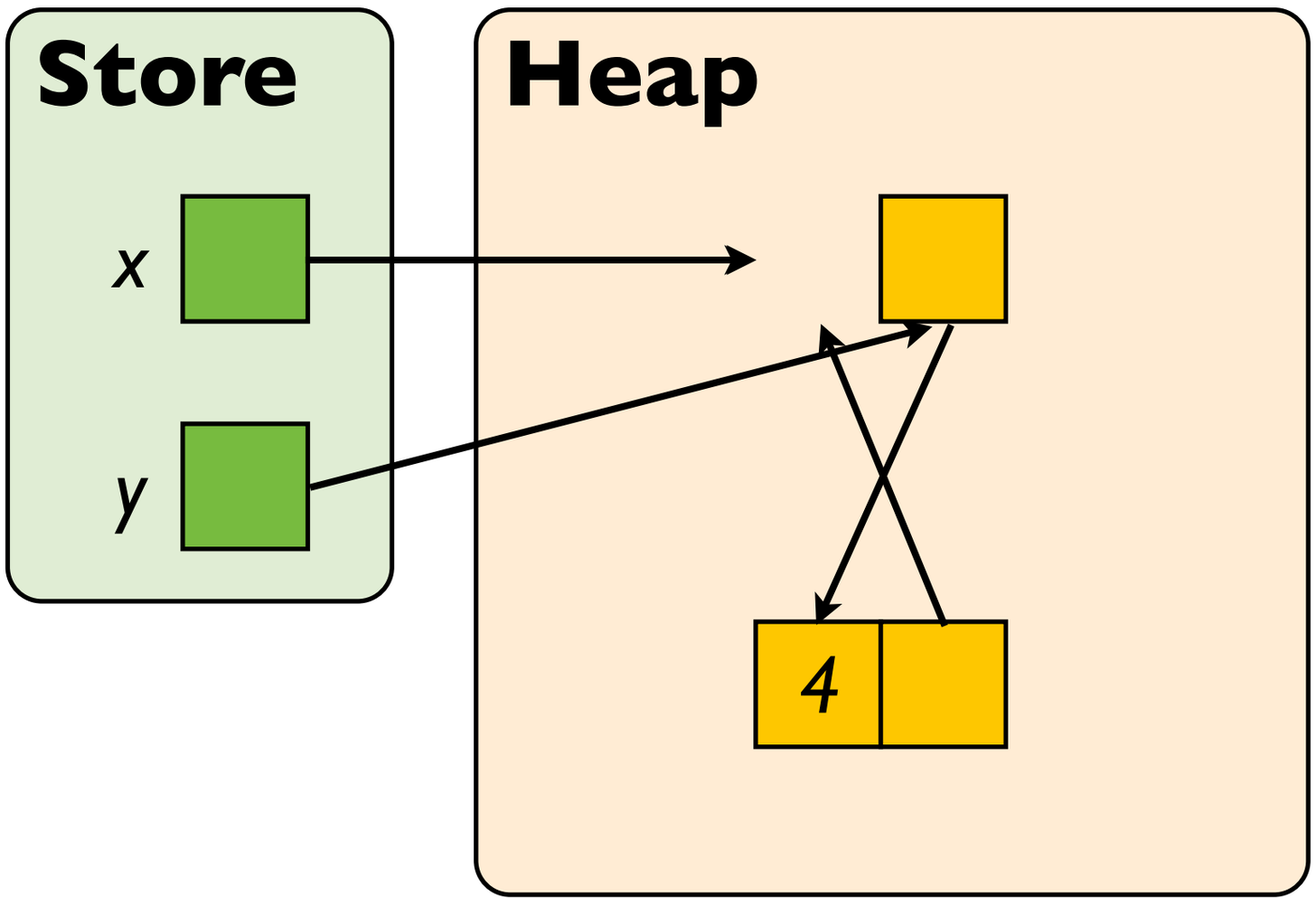
{emp}
  x := cons(3,3);
{x |-> 3,3}
  y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
  [x+1] := y;
{x |-> 3,y * y |-> 4,4}
  [y+1] := x;
{x |-> 3,y * y |-> 4,x}
  y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
  ∧ y = x+1}

Proof outline



Proof outline

{emp}
x := cons(3,3);
{x |-> 3,3}
y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
[x+1] := y;
{x |-> 3,y * y |-> 4,4}
[y+1] := x;
{x |-> 3,y * y |-> 4,x}
y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
∧ y = x+1}
dispose x;



Proof outline

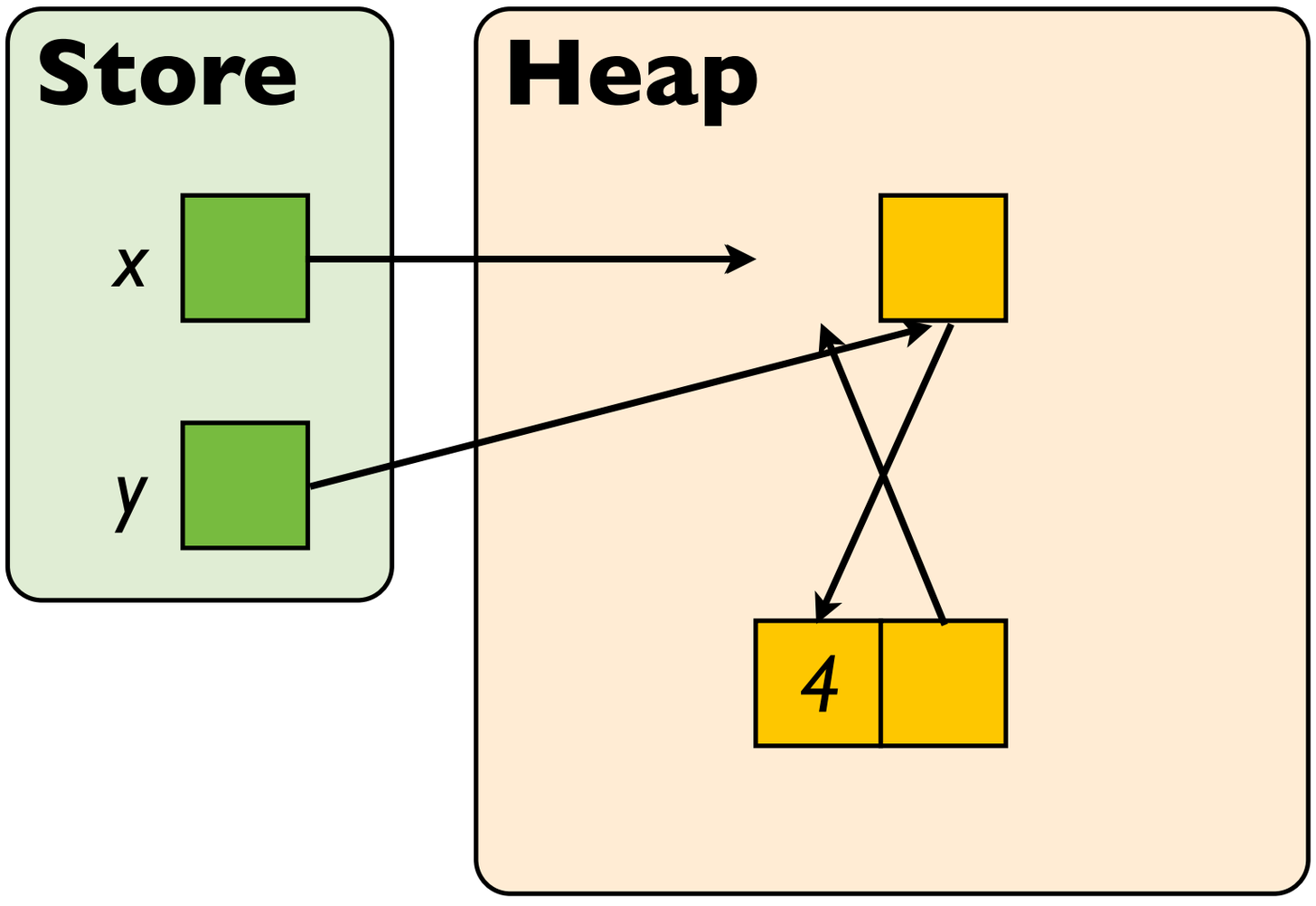
{emp}
x := cons(3,3);
{x |-> 3,3}
y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
[x+1] := y;
{x |-> 3,y * y |-> 4,4}
[y+1] := x;
{x |-> 3,y * y |-> 4,x}
y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
∧ y = x+1}
dispose x;
{emp * x+1 |-> y^{old} *
y^{old} |-> 4,x ∧ y = x+1}

{x |-> 3}
dispose x;
{emp}

frame rule!

Proof outline

{emp}
x := cons(3,3);
{x |-> 3,3}
y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
[x+1] := y;
{x |-> 3,y * y |-> 4,4}
[y+1] := x;
{x |-> 3,y * y |-> 4,x}
y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
∧ y = x+1}
dispose x;
{x+1 |-> y^{old} * y^{old} |-> 4,x
∧ y = x+1}



Proof outline

{emp}

x := cons(3,3);

{x |-> 3,3}

y := cons(4,4);

{x |-> 3,3 * y |-> 4,4}

[x+1] := y;

{x |-> 3,y * y |-> 4,4}

[y+1] := x;

{x |-> 3,y * y |-> 4,x}

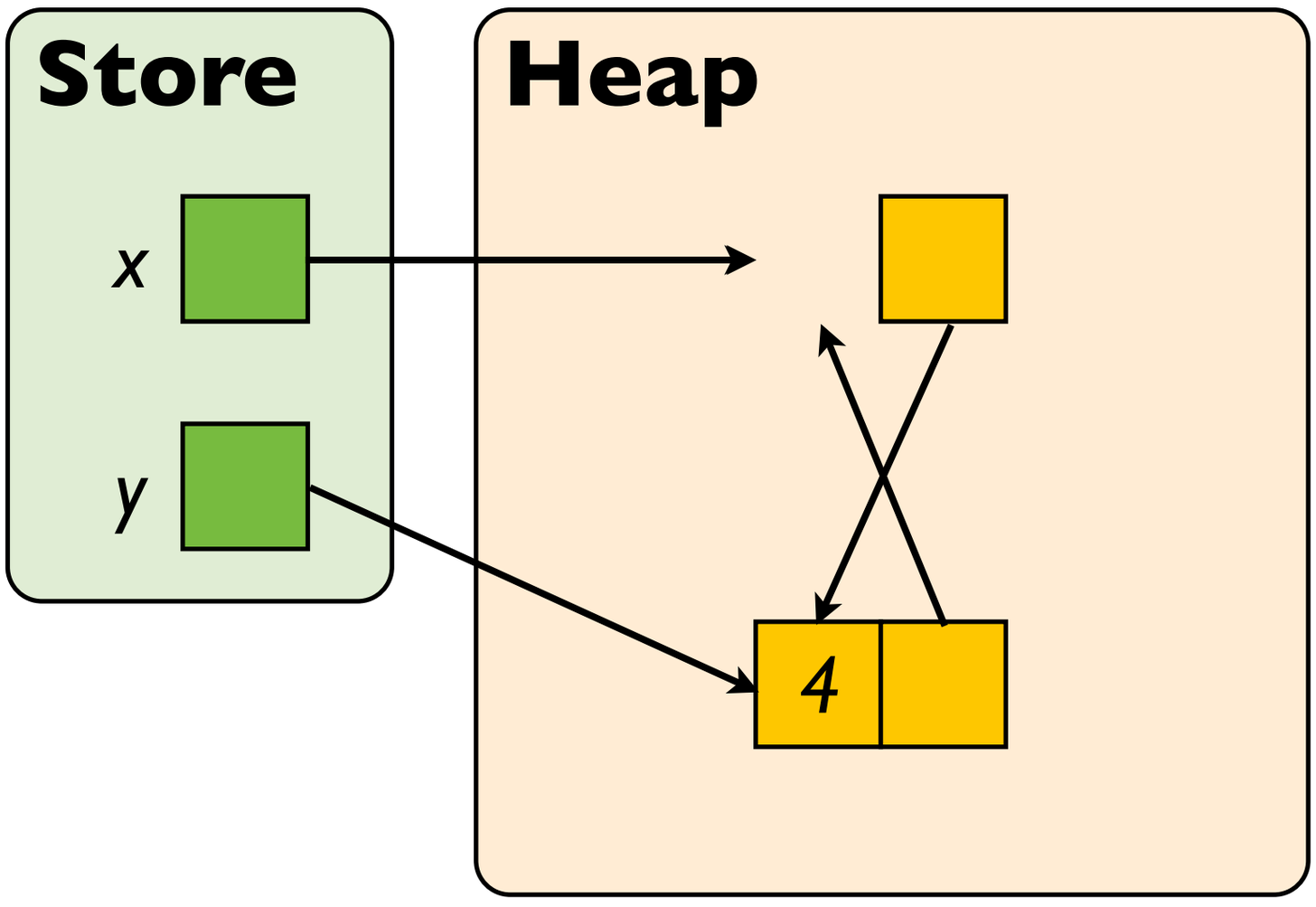
y := x+1;

{x |-> 3,y^{old} * y^{old} |-> 4,x
∧ y = x+1}

dispose x;

{x+1 |-> y^{old} * y^{old} |-> 4,x
∧ y = x+1}

y := [y];



{emp}
  x := cons(3,3);
{x |-> 3,3}
  y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
  [x+1] := y;
{x |-> 3,y * y |-> 4,4}
  [y+1] := x;
{x |-> 3,y * y |-> 4,x}
  y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
        ∧ y = x+1}
  dispose x;
{x+1 |-> y^{old} * y^{old} |-> 4,x
        ∧ y = x+1}
  y := [y];

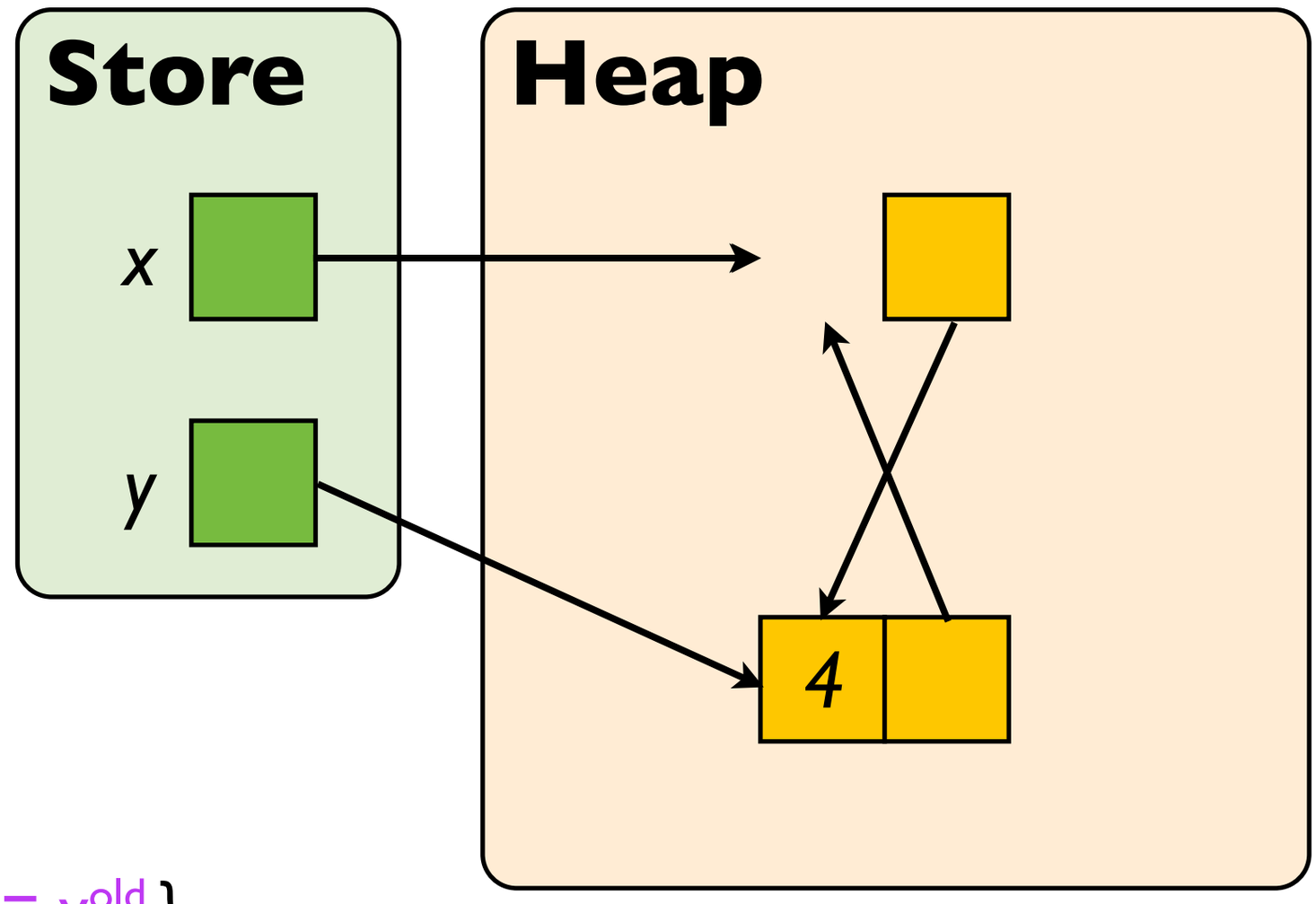
Proof outline

frame rule and consequence!

{x+1 = y ∧ y |-> y^{old} }
  y := [y];
{x+1 |-> y^{old} ∧ y^{old} = y}

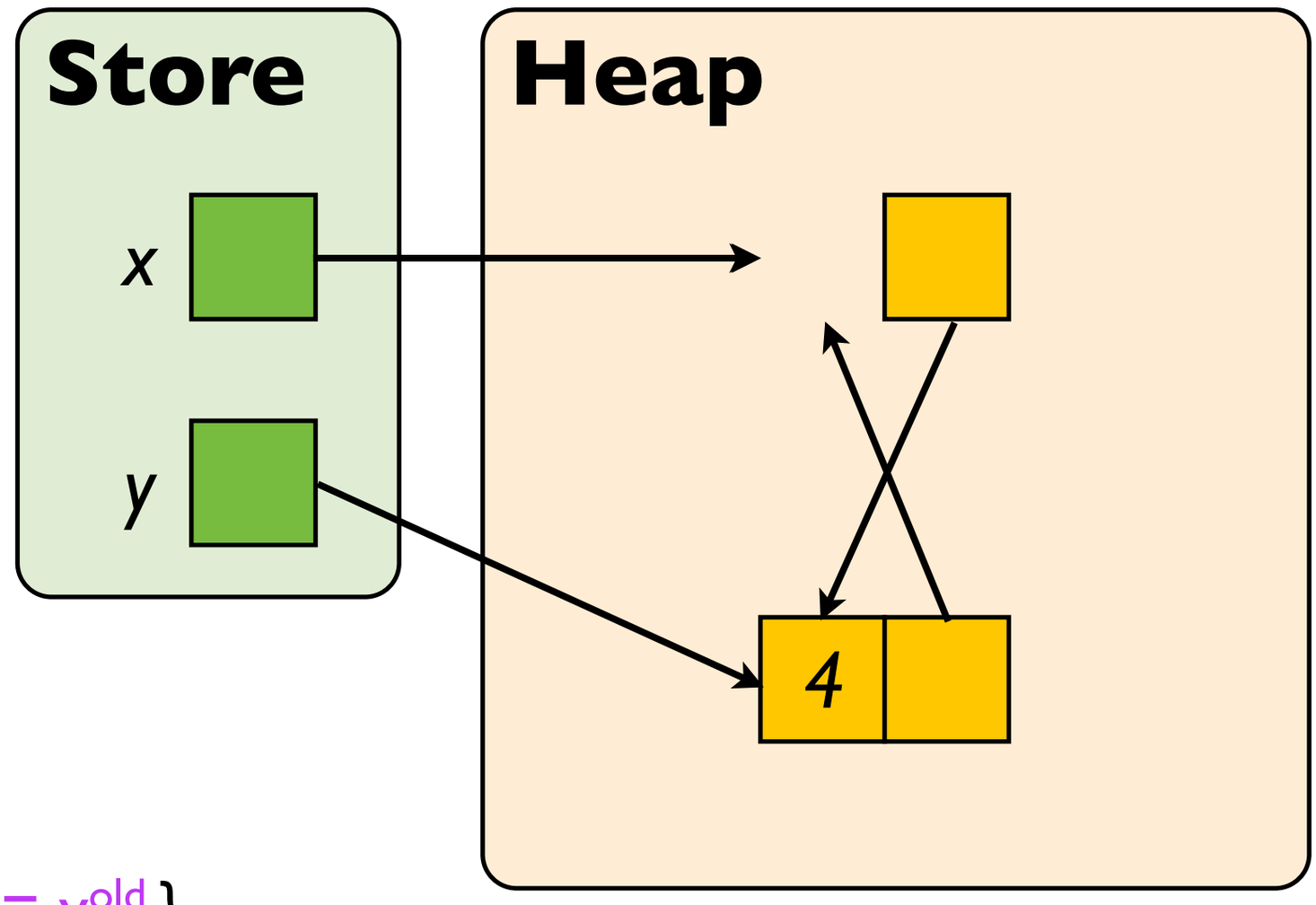
Proof outline

{emp}
x := cons(3,3);
{x |-> 3,3}
y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
[x+1] := y;
{x |-> 3,y * y |-> 4,4}
[y+1] := x;
{x |-> 3,y * y |-> 4,x}
y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
∧ y = x+1}
dispose x;
{x+1 |-> y^{old} * y^{old} |-> 4,x
∧ y = x+1}
y := [y];
{x+1 |-> y^{old} * y^{old} |-> 4,x ∧ y = y^{old} }



Proof outline

{emp}
x := cons(3,3);
{x |-> 3,3}
y := cons(4,4);
{x |-> 3,3 * y |-> 4,4}
[x+1] := y;
{x |-> 3,y * y |-> 4,4}
[y+1] := x;
{x |-> 3,y * y |-> 4,x}
y := x+1;
{x |-> 3,y^{old} * y^{old} |-> 4,x
∧ y = x+1}
dispose x;
{x+1 |-> y^{old} * y^{old} |-> 4,x
∧ y = x+1}
y := [y];
{x+1 |-> y^{old} * y^{old} |-> 4,x ∧ y = y^{old} }
{y |-> 4 * true}

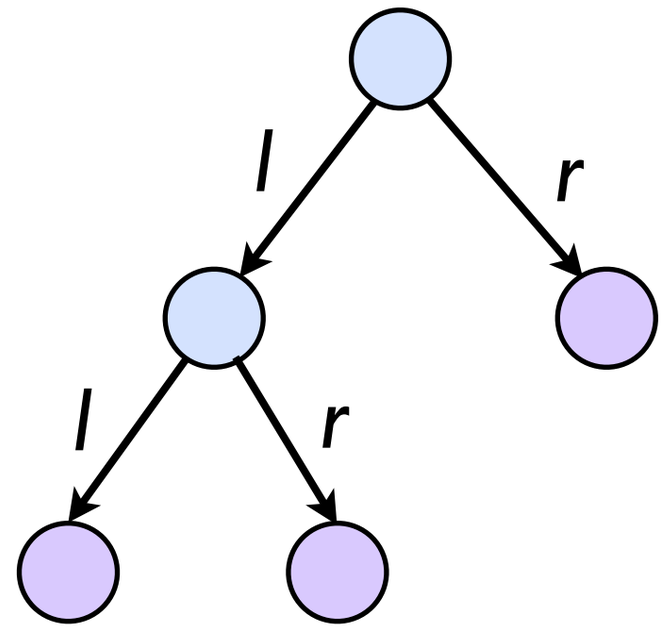


Inductive definitions in assertions

- heap portions in more realistic programs might comprise e.g. a tree, linked list
- helpful and concise to define such structures as predicates

Tree disposal

```
procedure DispTree(p)
local i, j;
if ¬isatom?(p) then
    i := p→l;
    j := p→r;
    DispTree(i)
    DispTree(j)
    dispose(p)
```



Tree predicate

$$\text{tree}(e) \iff$$
$$\texttt{if } \text{isAtom}(e) \texttt{ then } \text{emp}$$
$$\texttt{else } \exists x, y.\ e \mapsto x, y * \text{tree}(x) * \text{tree}(y)$$

- notes:
  - isAtom(e) returns true if e is an atomic value (e.g. characters) and not a location
  - if-then-else is easily compilable to logic (how?)

Tree disposal

```
procedure DispTree(p)
local i, j;
if ¬isatom?(p) then
    i := p→l;
    j := p→r;
    DispTree(i)
    DispTree(j)
    dispose(p)
```

{tree(p)} DispTree(p) {emp}

Tree disposal

```
procedure DispTree(p)
local i, j;
if ¬isatom?(p) then
    i := p→l;
    j := p→r;
    DispTree(i)
    DispTree(j)
    dispose(p)
```

we first:

- *adjust for our store/heap model*
- *focus the proof on the crucial part*

{tree(p)} DispTree(p) {emp}

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
    j := [p+1];
    DispTree(i);
    DispTree(j);
    dispose(p);
    dispose(p+1);
{emp}
```

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
```

Tree disposal proof

(from O'Hearn)

{p |-> x,y * tree(x) * tree(y)}

i := [p];

{p |-> x}

i := [p]

{p |-> x ∧ x = i }

frame rule!

{p |-> x,y * tree(x) * tree(y) ∧ x = i}

{p |-> x,y * tree(i) * tree(y)}

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}

    i := [p];
{p |-> x,y * tree(i) * tree(y)}
j := [p+1];
```

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
```

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
```

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
```

frame rule!

```
{tree(i)}
    DispTree(i)
{emp}
```

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
    DispTree(j);
```

Tree disposal proof

(from O'Hearn)

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
    DispTree(j);
{p |-> x,y * emp * emp}
```

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
    DispTree(j);
{p |-> x,y * emp * emp}
```

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
    DispTree(j);
{p |-> x,y * emp * emp}
    dispose(p);
    dispose(p+1);
```

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
    DispTree(j);
{p |-> x,y * emp * emp}
    dispose(p);
    dispose(p+1);
{emp * emp * emp}
```

```
{p |-> x,y * tree(x) * tree(y)}
    i := [p];
{p |-> x,y * tree(i) * tree(y)}
    j := [p+1];
{p |-> x,y * tree(i) * tree(j)}
    DispTree(i);
{p |-> x,y * emp * tree(j)}
    DispTree(j);
{p |-> x,y * emp * emp}
    dispose(p);
    dispose(p+1);
{emp * emp * emp}
{emp}
```

Next on the agenda

(1) model of program states for separation logic 

(2) assertions and spatial connectives 

(3) axioms and inference rules 

(4) program proofs 

In a nutshell 

The frame rule is absolutely key to separation logic proofs

$$\frac{\{p\}\ \ C\ \ \{q\}}{\{p * r\}\ \ C\ \ \{q * r\}}$$

Thank you! Questions?