Software Verification (Fall 2013)
Lecture 5: Separation Logic
Parts I - II

Chris Poskitt

Chair of Software Engineering

ETH zürich

A recent separation logic success story

MONOIDICS

facebook

theguardian

News | Sport | Comment | Culture | Business | Money | Life & style

News › Technology › Facebook

Facebook buys code-checking Silicon Roundabout startup Monoidics

Acquisition of company which carries out tests to find crashing bugs will see its technology applied to mobile apps and site

The Telegraph

Home News World Sport Finance Comment Blogs Culture Travel Life Women

Technology News | Technology Companies | Technology Reviews | Video Games | Technology

HOME » TECHNOLOGY » FACEBOOK

Facebook buys UK startup Monoidics

Facebook has acquired assets behind Monoidics, a London-based startup whose technology is used to detect coding errors.

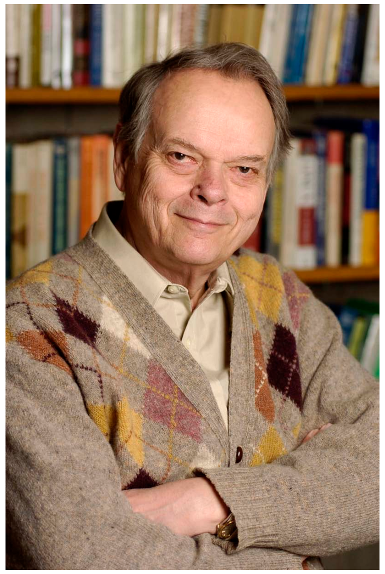
Main sources for these lectures

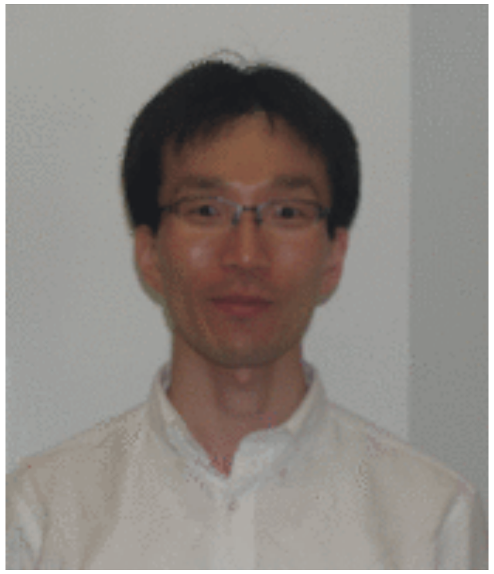
Peter W. O'Hearn:

A primer on separation logic
(and automatic program verification and analysis)

In: *Software Safety and Security: Tools for Analysis and Verification*. NATO Science for Peace and Security Series, vol. 33, pages 286-318, 2012

Main sources for these lectures





Peter W. O'Hearn, John C. Reynolds, Hongseok Yang

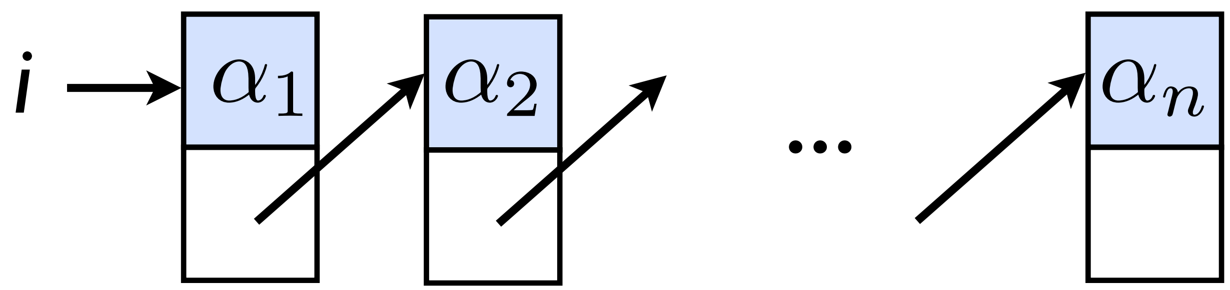
Local Reasoning about Programs that Alter Data Structures

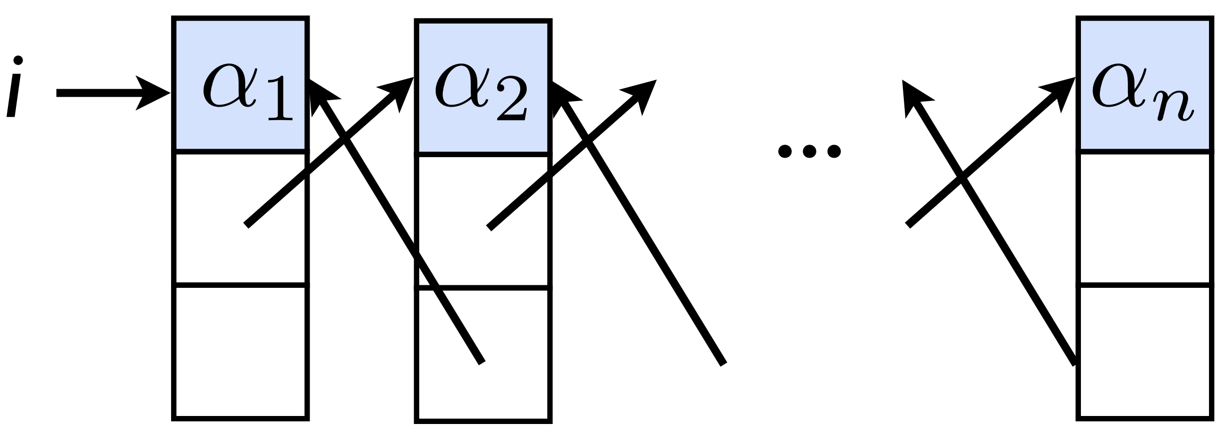
CSL '01. Volume 2142 of LNCS, pages 1-19. Springer, 2001.

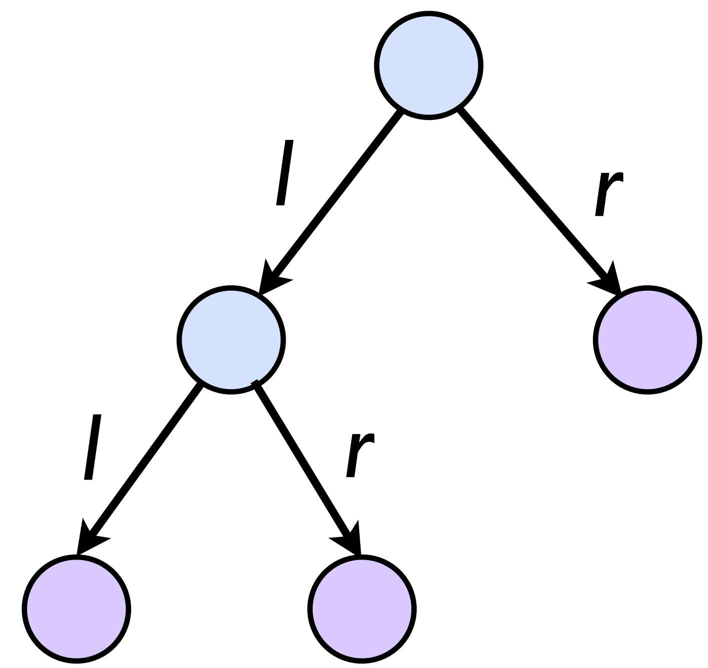
What is separation logic for?

- for reasoning about shared mutable data structures in imperative programs
  - structures where an updatable field can be referenced from <u>more than one point</u>
  - correctness of such programs depends upon complex restrictions on sharing
  - classical methods like Hoare logic suffer from extreme complexity; reasoning does not match programmers' intuitions

Some shared mutable data structures





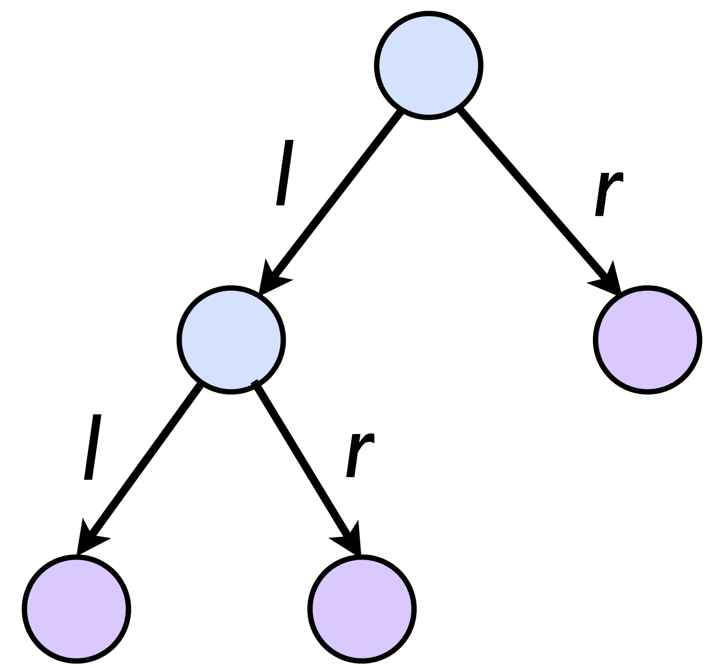


Problem illustration

(from O'Hearn)

- the following program disposes the elements of a tree

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



- can we prove its correctness using classical Hoare logic?

Problem illustration: Hoare logic

- here is a possible specification:

  { *tree(p)* $\wedge$ *reach(p,n)* }
  DispTree(*p*)
  { $\neg$*allocated(n)* }

  i.e. if before execution there is a node *n* in the tree that *p* points to, then after execution, *n* is not allocated

have we specified enough?

Problem illustration: Hoare logic

- what does DispTree(*p*) do to nodes outside of the tree *p*?

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

specification too weak!
does not rule out that DispTree(i)
did not alter subtree j...
...might no longer be a tree!
(precondition violation)

{ *tree(i)* ∧ *reach(i,n)* }
DispTree(*i*)
{ *¬allocated(n)* }

Problem illustration: Hoare logic

- can strengthen the specification with frame axioms
  i.e. clauses specifying what does not change

$\{$ *tree(p)* $\wedge$ *reach(p,n)* $\wedge$ *¬reach(p,m)* $\wedge$ *allocated(m)*
$\wedge$ *m.f = m'* $\wedge$ *¬allocated(q)* $\}$
  DispTree(*p*)
$\{$ *¬allocated(n)* $\wedge$ *¬reach(p,m)* $\wedge$ *allocated(m)*
$\wedge$ *m.f = m'* $\wedge$ *¬allocated(q)* $\}$

- complicated; certainly does not scale!
- does not match the intuition that programmers use

How does separation logic help?

- separation logic extends Hoare logic to facilitate local reasoning
- assertion language offers spatial connectives, allowing one to reason about smaller parts of the program state

$$\boxed{p * q}$$

- this locality allows us to:
  - avoid mentioning the frame in specifications
  - but to bring the frame condition in when needed

Next on the agenda

(1) model of program states for separation logic

(2) assertions and spatial connectives

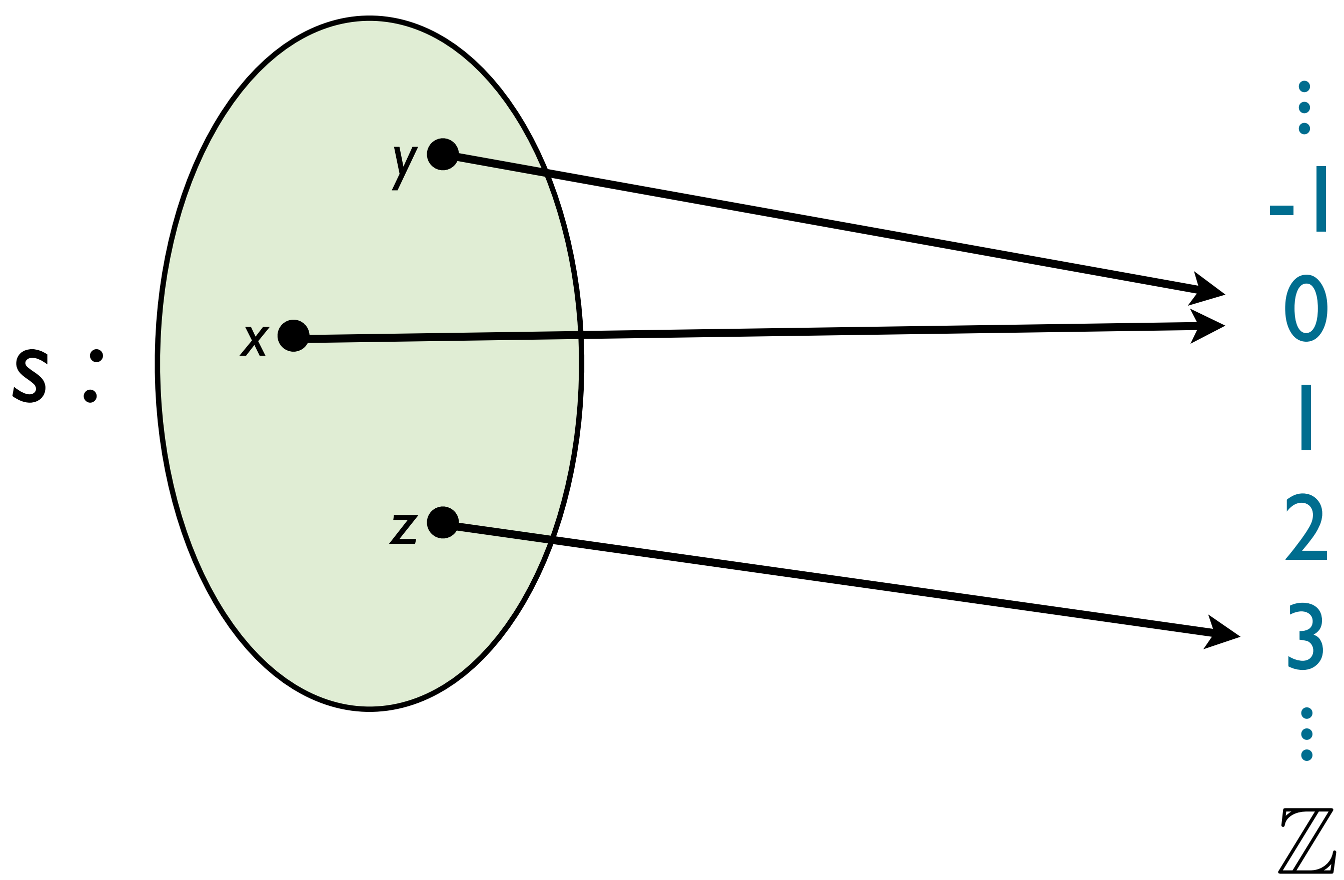
(3) axioms and inference rules

(4) program proofs

Recap: program states

- in Hoare logic a program state comprises a variable store

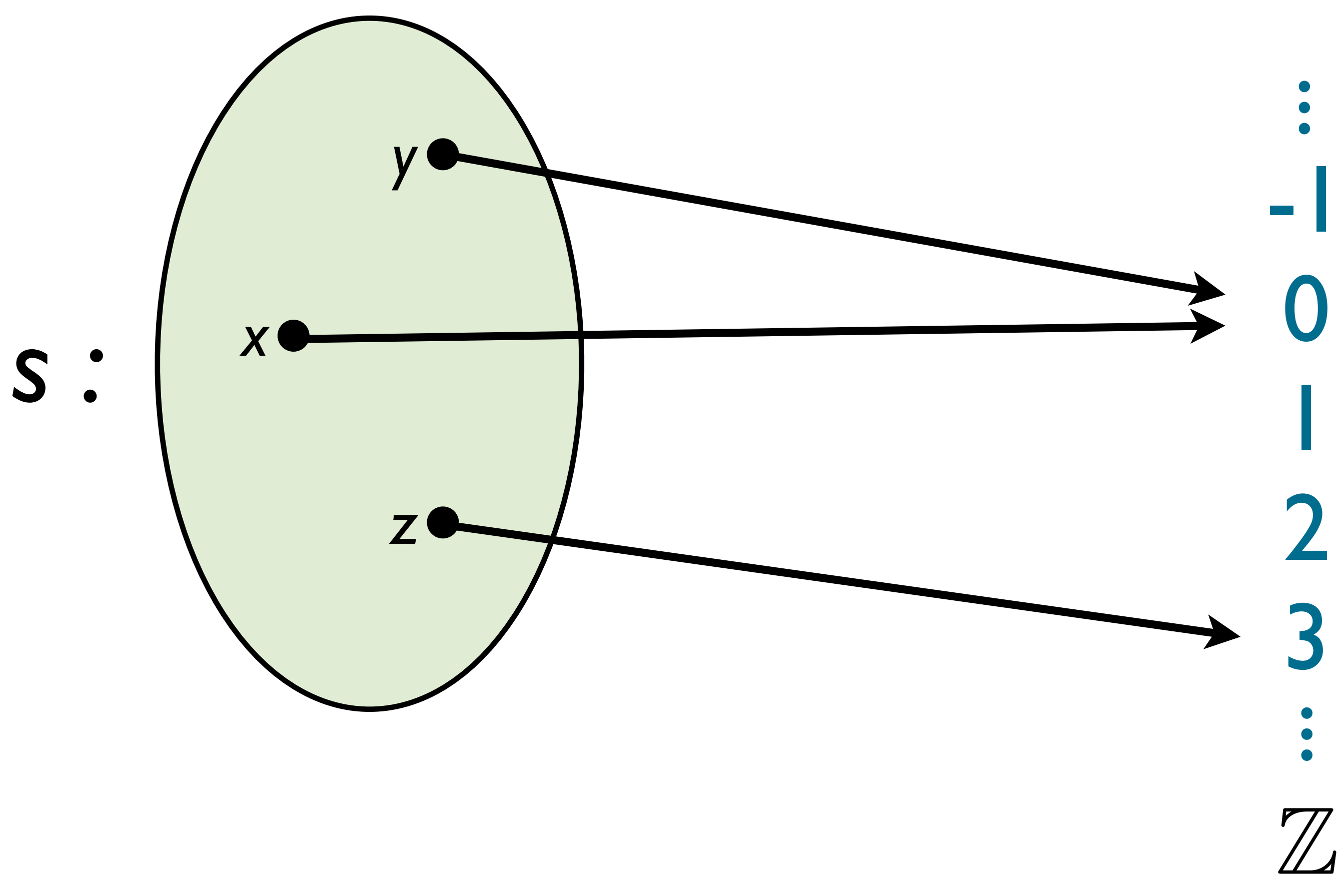
  i.e. a partial function mapping variables to integers



Recap: program states

- in Hoare logic a program state comprises a variable store

  i.e. a partial function mapping variables to integers



$s(x) = 0$
$s(y) = 0$
$s(z) = 3$

Recap: satisfaction of assertions

- we write $s \models p$ if store s (i.e. a program state) satisfies assertion p

- typically $\models$ is defined inductively

$$
\begin{aligned}
s \models p \wedge q \quad & \text{if} \quad s \models p \text{ and } s \models q \\
s \models \exists x.\ p \quad & \text{if} \quad \text{there exists some integer } v \text{ such that } s[x \mapsto v] \models p \\
& \vdots \\
s \models B \quad & \text{if} \quad [|B|]s = \text{true}
\end{aligned}
$$

(where $[|B|]s$ denotes the evaluation of B w.r.t. s)

Recap: satisfaction of assertions

- For example:

$$(x \mapsto 5, y \mapsto 10) \models x < y \land x > 0$$

$$(x \mapsto 25) \models \exists y.\ y > x$$

$$(x \mapsto 0) \not\models \exists y.\ y < x \land y \geq 0$$

The Heaplet model

- in separation logic, program states comprise both a variable store <u>and</u> a heap

    i.e. a function mapping <u>locations</u> (pointers) to integers

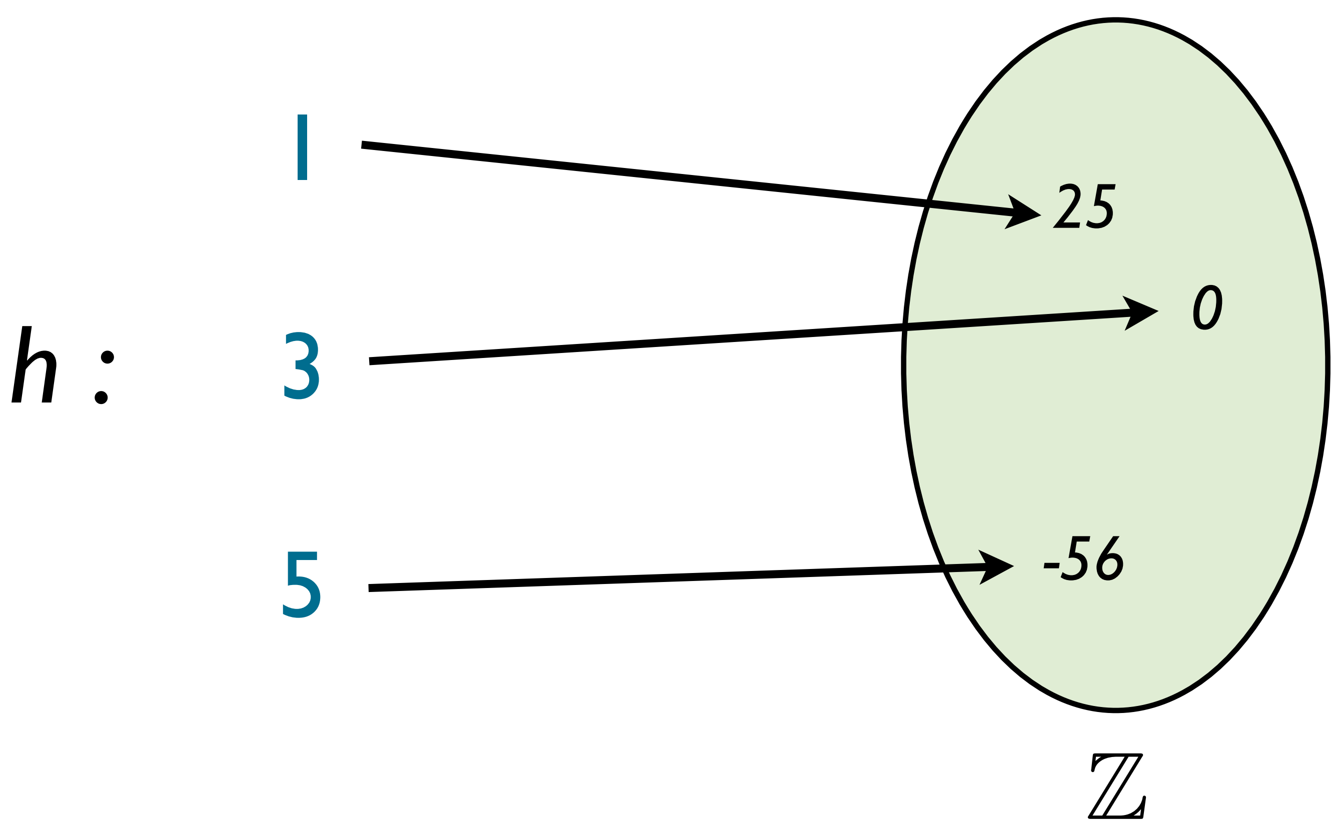
we define <u>locations</u> as a subset of the integers

The Heaplet model

- in separation logic, program states comprise both a variable store <u>and</u> a heap

i.e. a function mapping <u>locations</u> (pointers) to integers

we define <u>locations</u> as a subset of the integers



The Heaplet model

- the <u>store</u>: state of the local variables

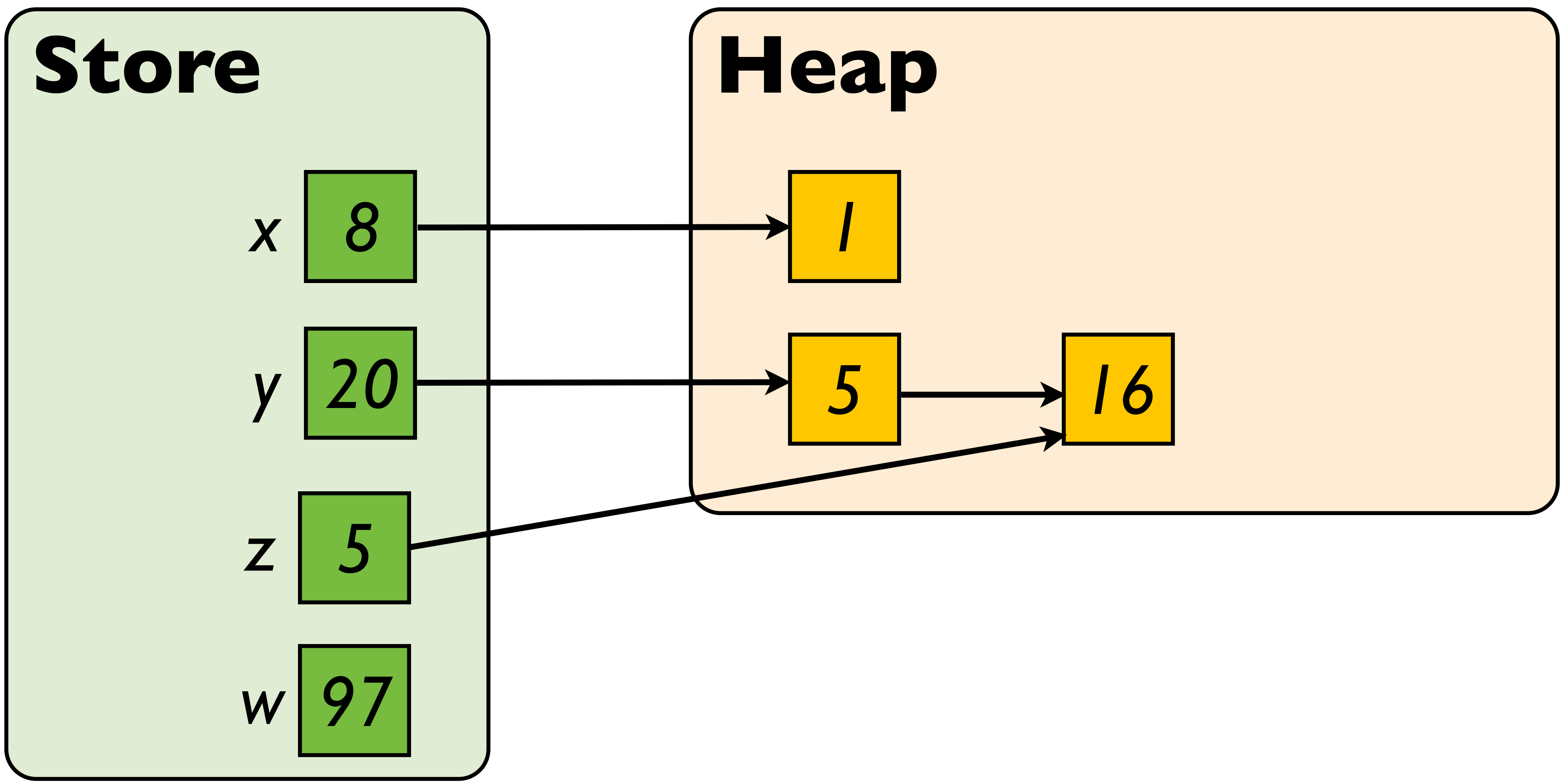
$$\text{Variables} \to \text{Integers}$$

- the <u>heap</u>: state of dynamically-allocated objects

$$\text{Locations} \to \text{Integers}$$

where: $\text{Locations} \subseteq \text{Integers}$

Example store and heap



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

Syntax of assertions

false	logical false	
$p \wedge q$	classical conjunction	
$p \vee q$	classical disjunction	
$p \Rightarrow q$	classical implication	
$p * q$	separating conjunction	*spatial assertions*
$p \mathrel{-\!\!*} q$	separating implication	
$e = f$	equality of expressions	
$e \mapsto f$	points to (in the heap)	*heap assertions*
emp	empty heap	
$\exists x.\ p$	existential quantifier	

(*e*,*f* range over integer expressions; *x* over variables; *p*,*q* over assertions)

Semantics of assertions

- we write $s,h \models p$ if store s and heap h (together the program state) satisfies assertion p

$$
\begin{array}{rll}
s,h \models \text{false} & \text{never} & \\
s,h \models p \wedge q & \text{if} & s,h \models p \text{ and } s,h \models q \\
s,h \models p \vee q & \text{if} & s,h \models p \text{ or } s,h \models q \\
s,h \models p \Rightarrow q & \text{if} & s,h \models p \text{ implies } s,h \models q \\
s,h \models e = f & \text{if} & [|e|]s = [|f|]s
\end{array}
$$

(where $[|e|]s$ denotes the evaluation of e with respect to s)

Semantics of empty heap

- the semantics of the remaining assertions all rely on the heap *h*

$$s, h \models \mathrm{emp} \quad \text{if} \quad h = \{\}$$

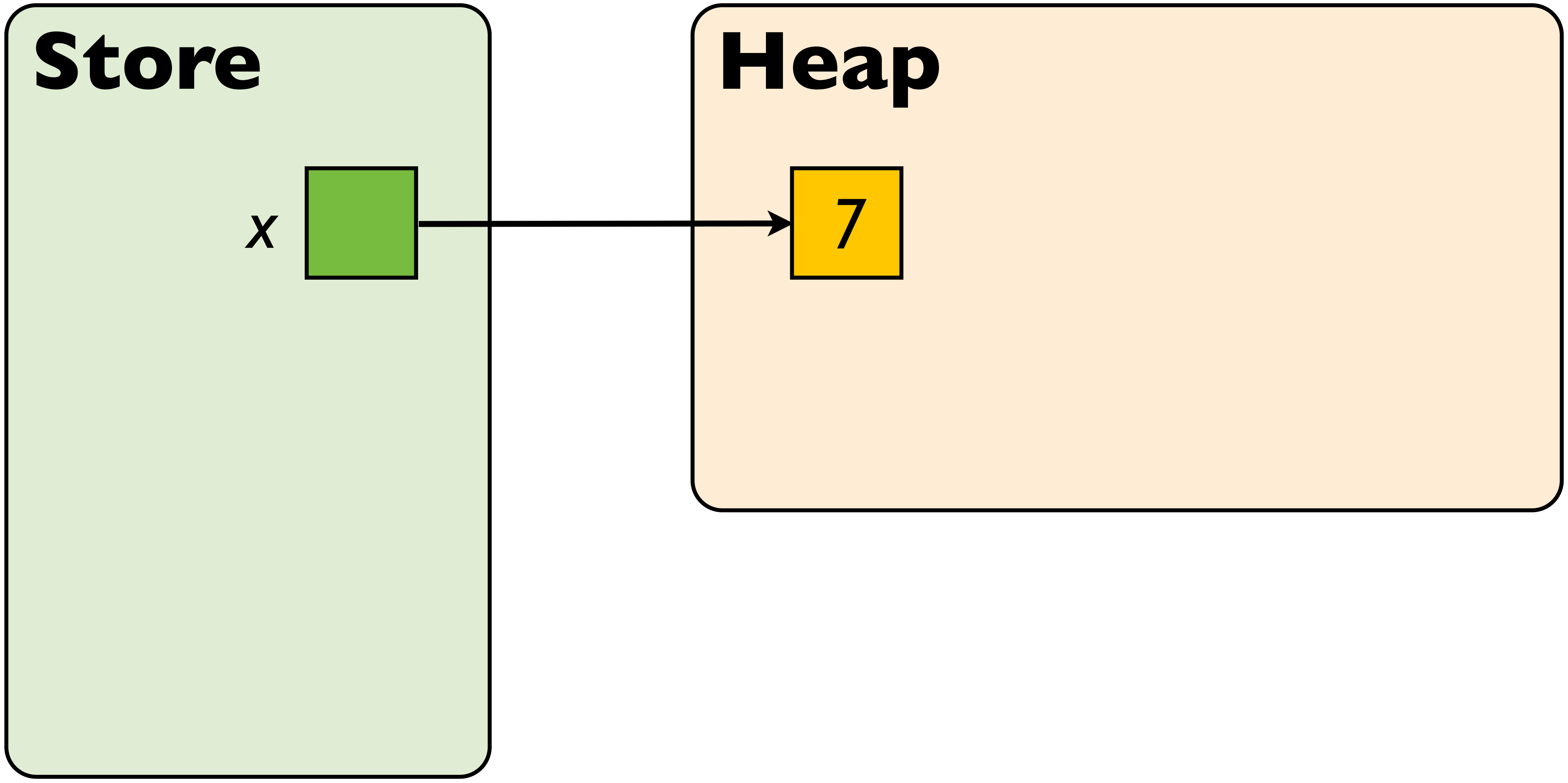
Semantics of points to

$$s, h \models e \mapsto f \quad \text{if} \quad h = \{[\![e]\!]s \to [\![f]\!]s\}$$

the heap h has <u>exactly</u> one location: the value of e...
...and the contents at that location is the value of f

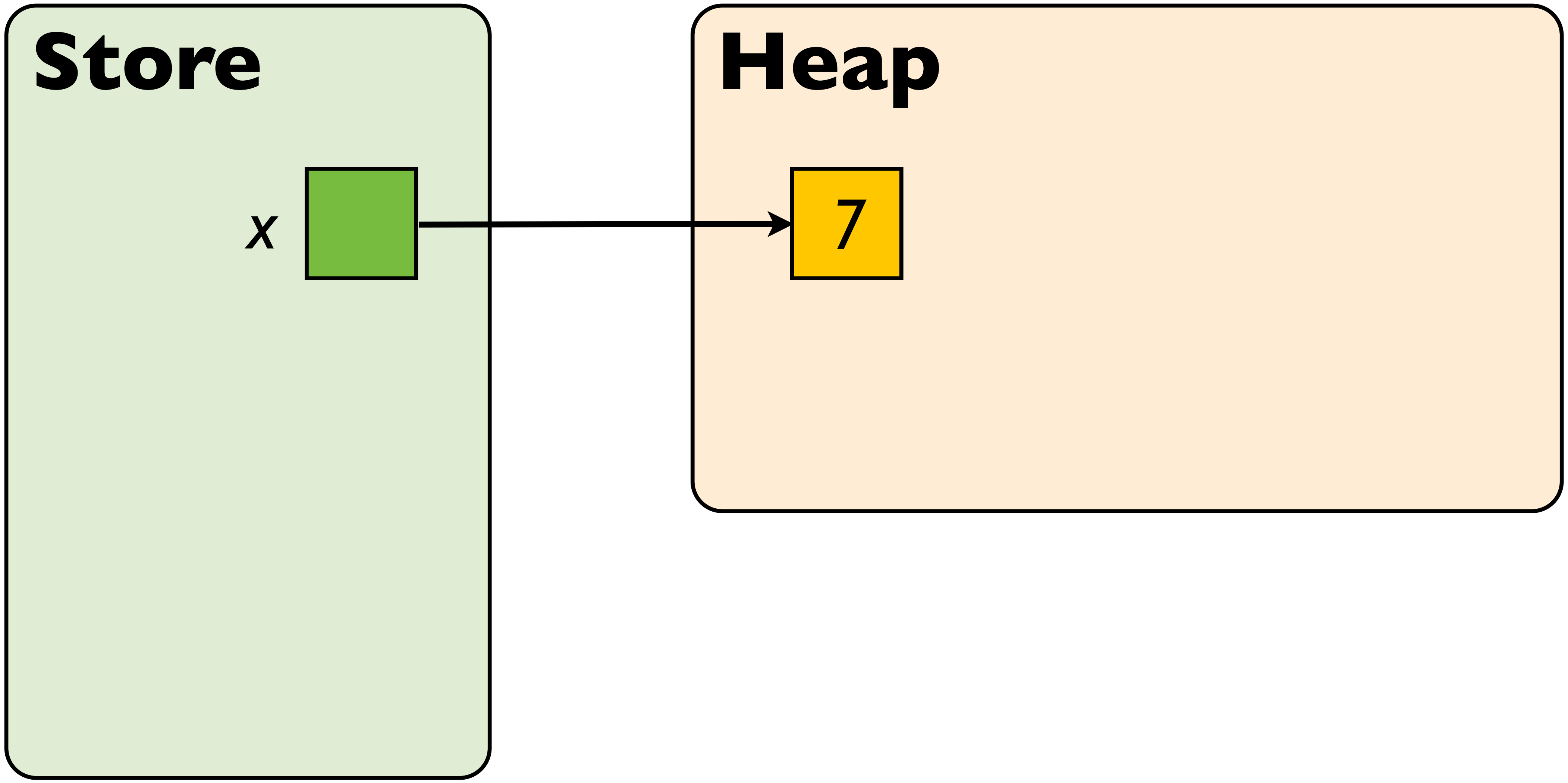
what about larger heaps?

Example of points to



Example of points to

$$x \mapsto 7$$



Semantics of separating conjunction

$$\boxed{s, h \models p * q}$$

- informally: the heap *h* can be divided in two so that *p* is true of one partition and *q* of the other

Semantics of separating conjunction

$$\boxed{s, h \models p * q}$$

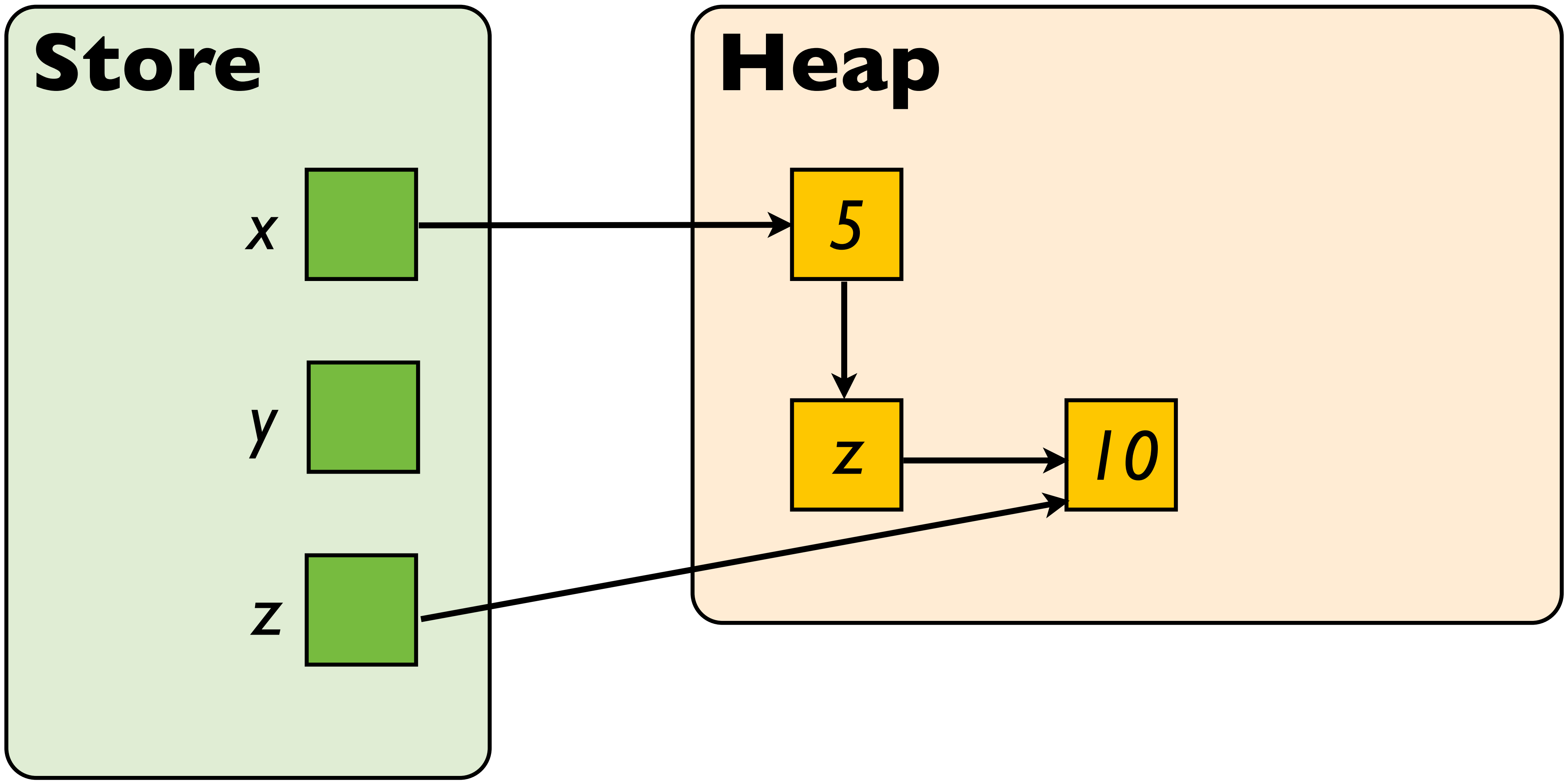
- informally: the heap *h* can be divided in two so that *p* is true of one partition and *q* of the other

disjoint domains of definition

disjoint function composition

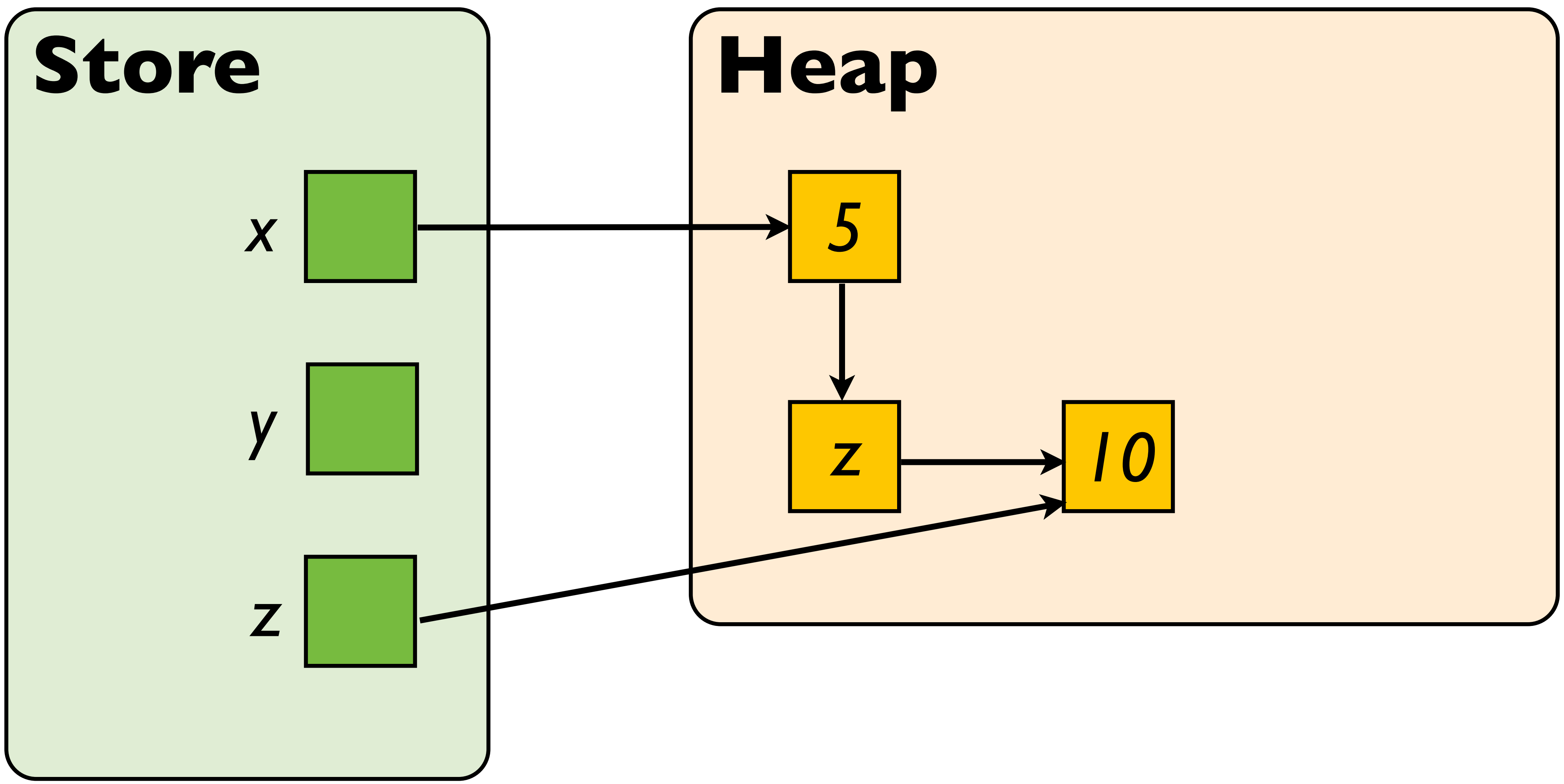
$$s, h \models p * q \quad \text{if} \quad \exists h_1, h_2.\ (h_1 \perp h_2),\ (h_1 \circ h_2 = h),$$
$$s, h_1 \models p \text{ and } s, h_2 \models q$$

Example of separating conjunction

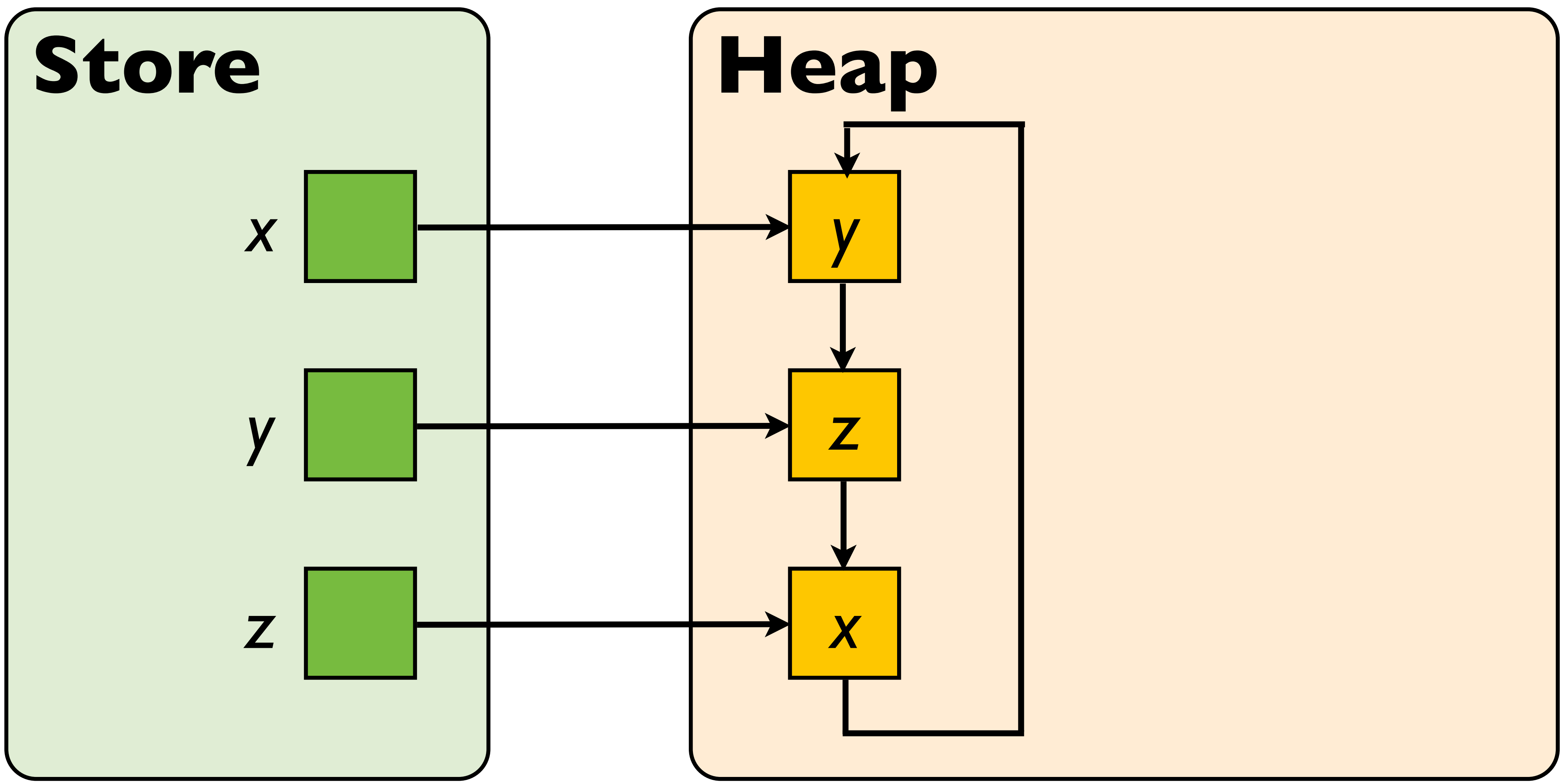


Example of separating conjunction

$$x \mapsto 5 * 5 \mapsto z * z \mapsto 10$$



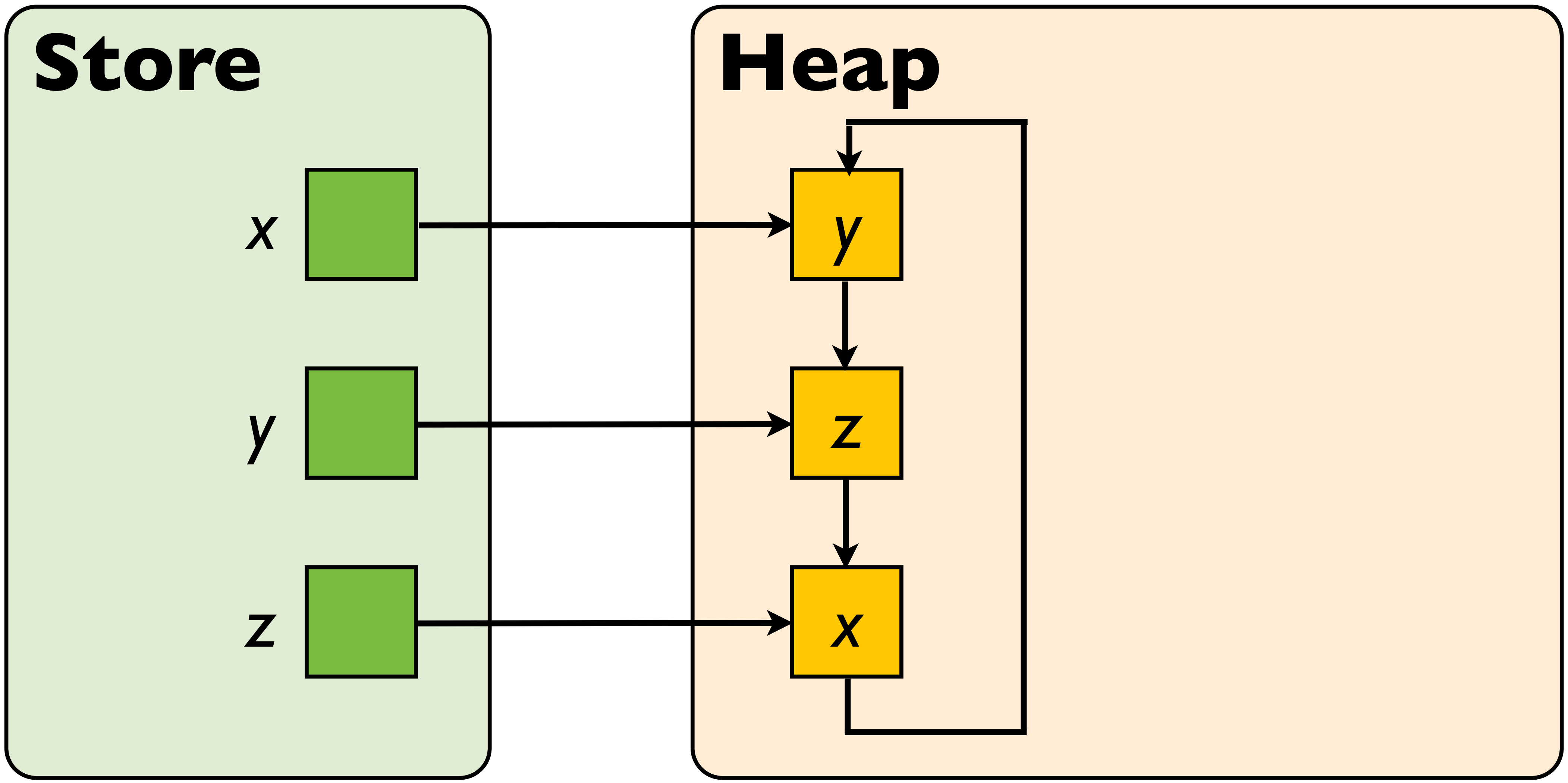
Example of separating conjunction
(from Calcagno)



Example of separating conjunction

(from Calcagno)

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



Notation

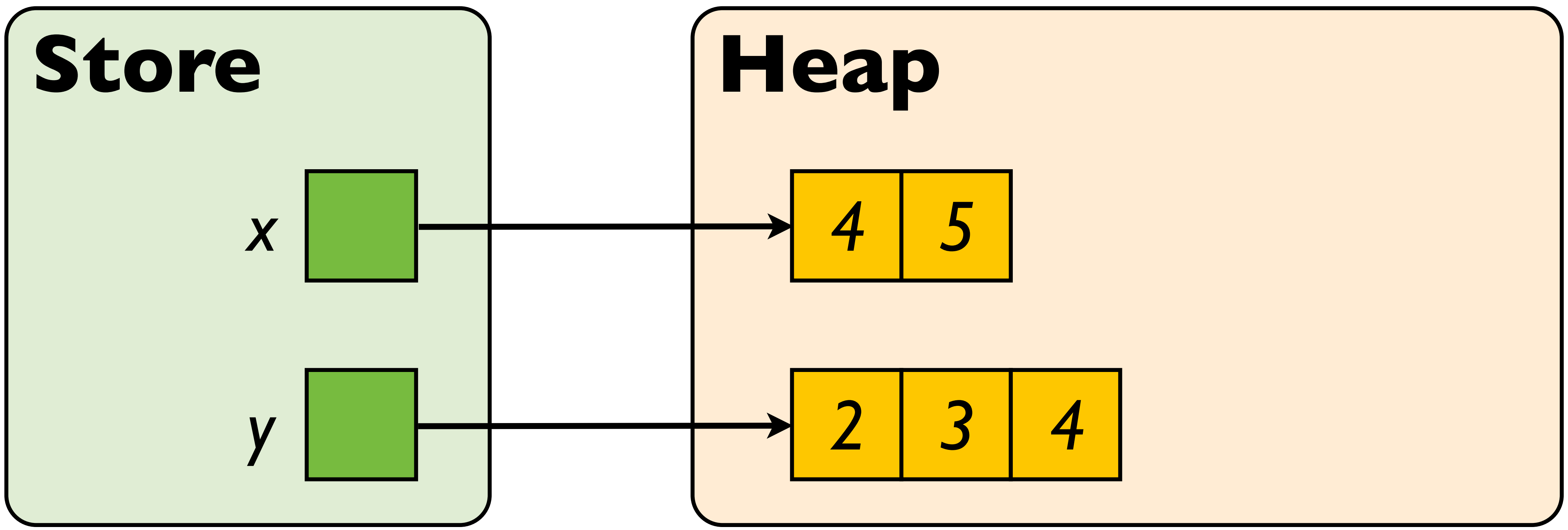
let $e \mapsto f_0, \ldots, f_n$

abbreviate $e \mapsto f_0 * e + 1 \mapsto f_1 * \cdots * e + n \mapsto f_n$

Notation

let $\quad e \mapsto f_0, \ldots, f_n$

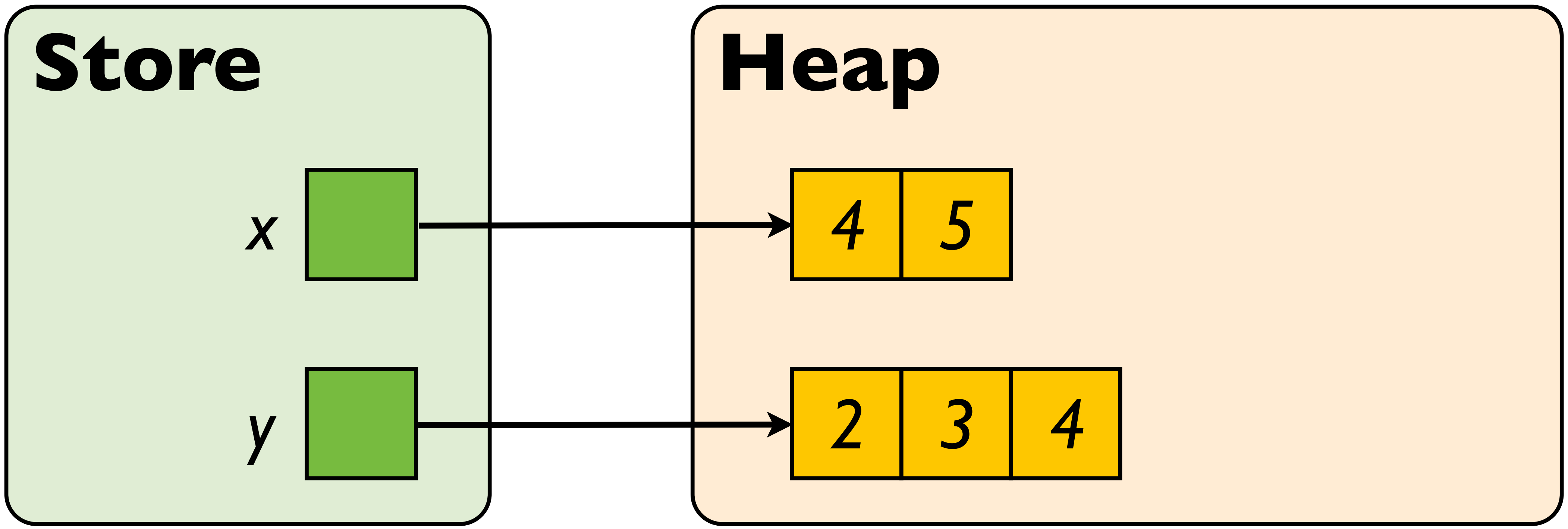
abbreviate $\quad e \mapsto f_0 * e+1 \mapsto f_1 * \cdots * e+n \mapsto f_n$



Notation

let $e \mapsto f_0, \ldots, f_n$

abbreviate $e \mapsto f_0 * e+1 \mapsto f_1 * \cdots * e+n \mapsto f_n$

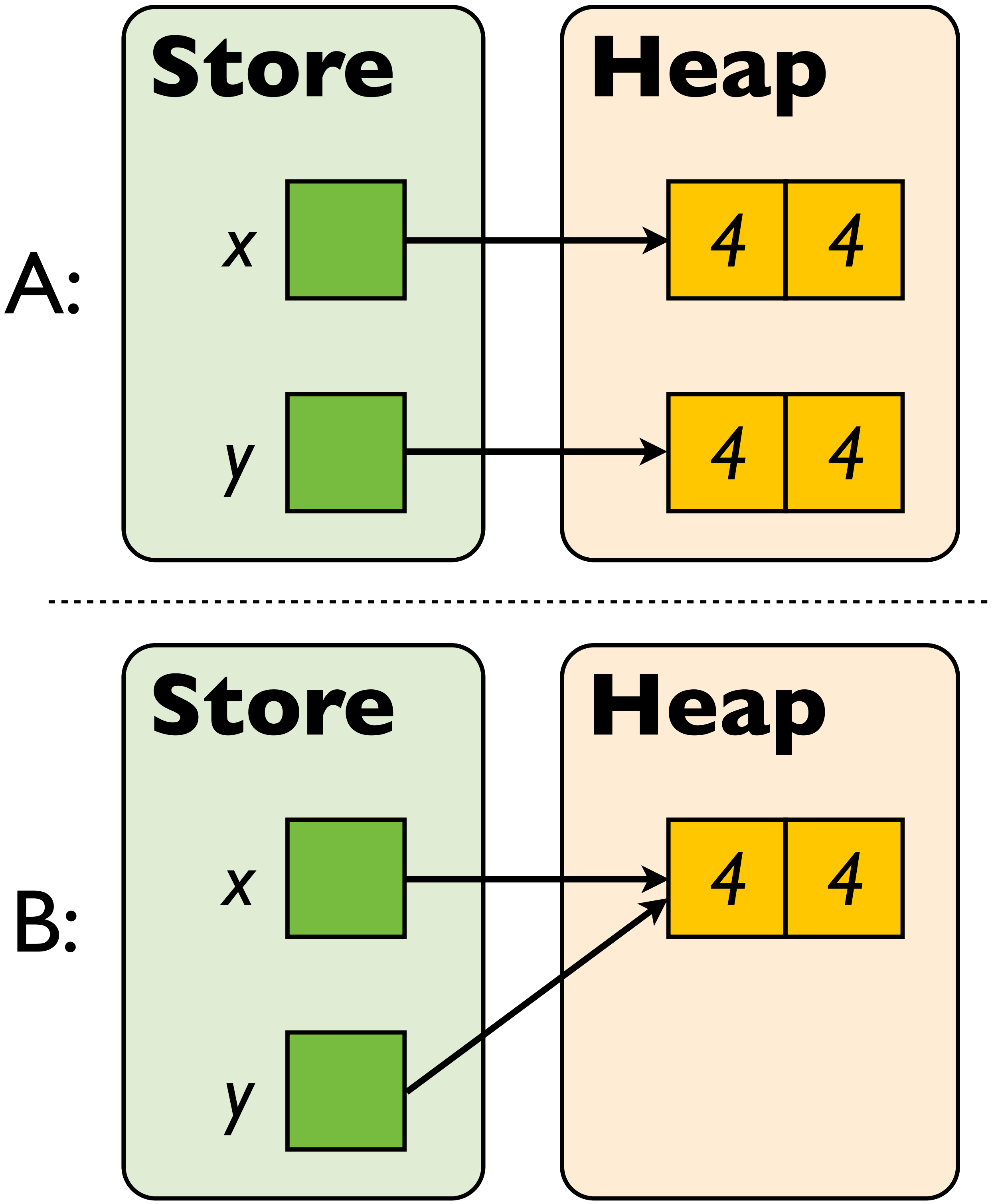


$$x \mapsto 4, 5 * y \mapsto 2, 3, 4$$

$$x \mapsto 4, 5 * true$$

Exercises

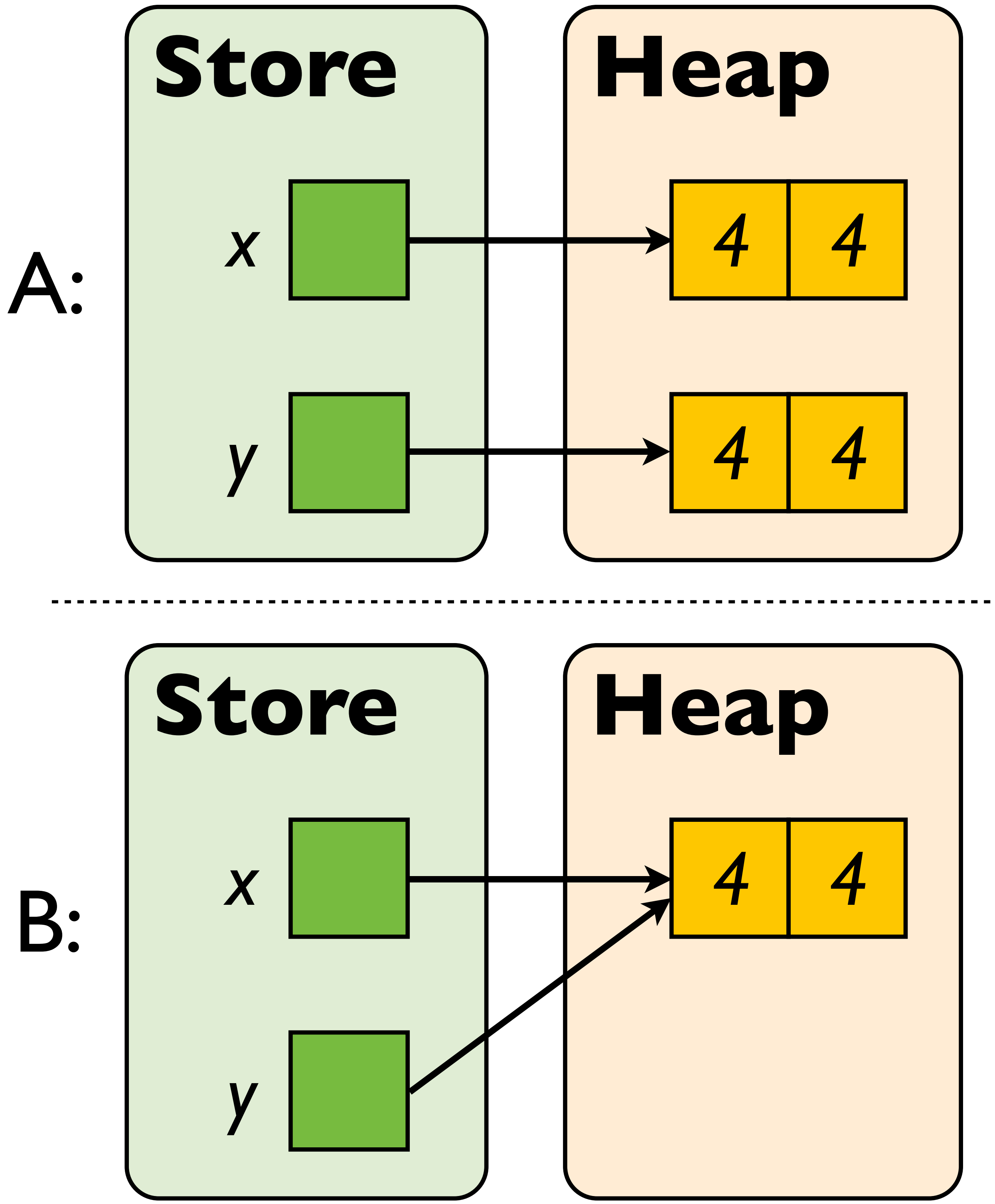
(from Parkinson)



	A	B
$x \mapsto 4, 4$		
$x \mapsto 4, 4 * \mathrm{true}$		
$x \mapsto 4, 4 * y \mapsto 4, 4$		
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$		
$(x \mapsto 4, 4 * \mathrm{true}) \wedge (y \mapsto 4, 4 * \mathrm{true})$		

Exercises

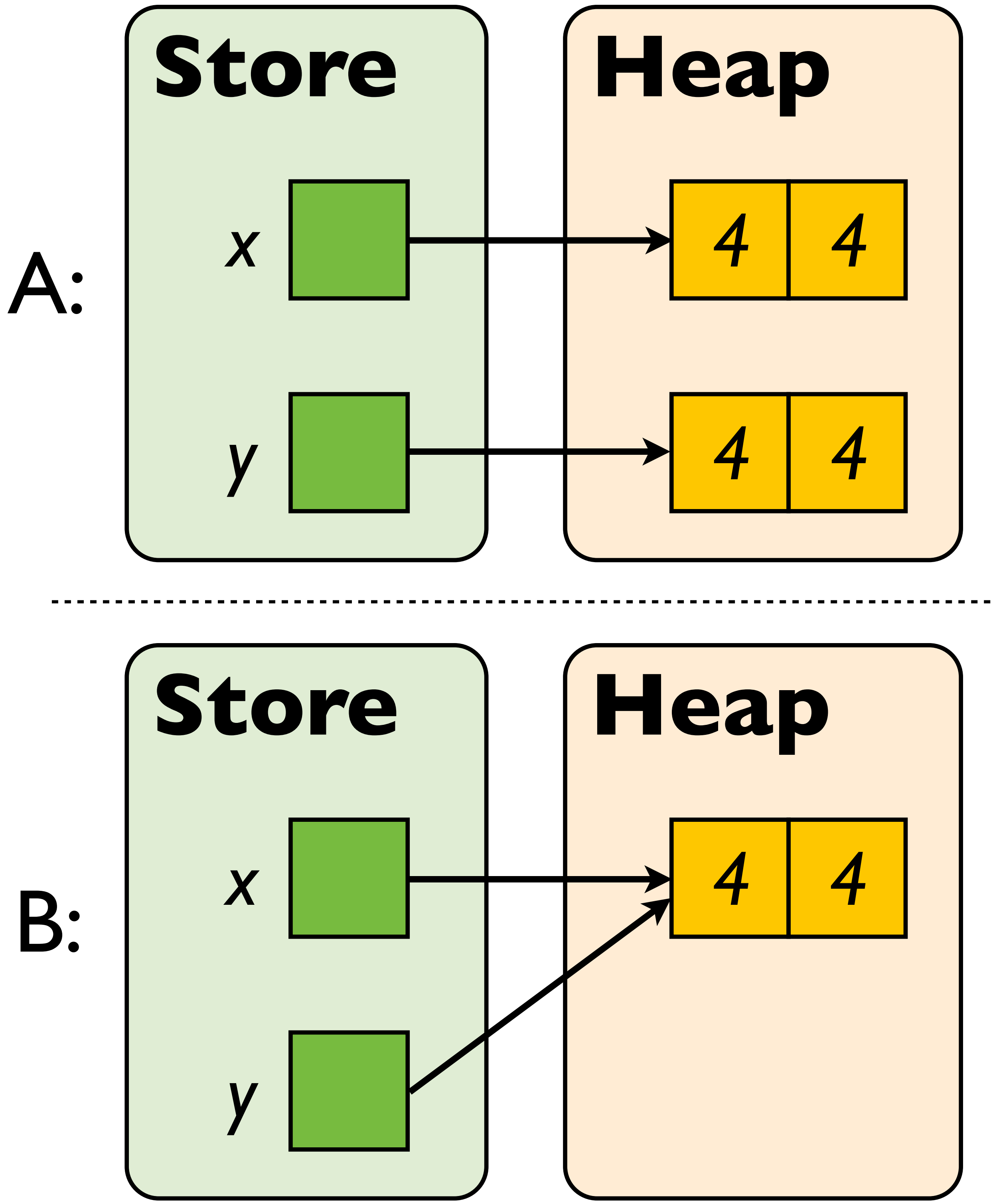
(from Parkinson)



	A	B
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 * \mathrm{true}$		
$x \mapsto 4, 4 * y \mapsto 4, 4$		
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$		
$(x \mapsto 4, 4 * \mathrm{true}) \wedge (y \mapsto 4, 4 * \mathrm{true})$		

Exercises

(from Parkinson)



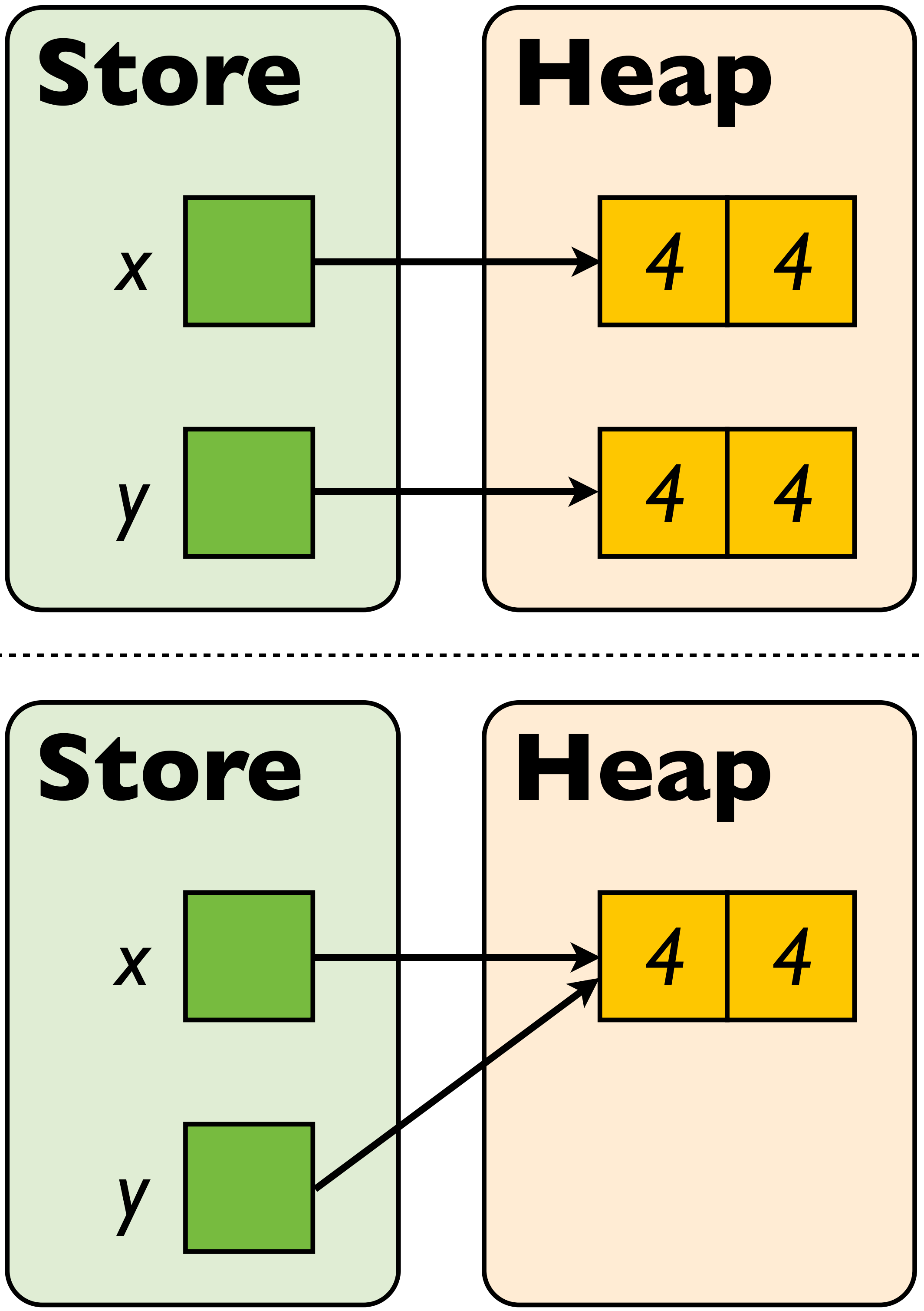
	A	B
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 * \mathrm{true}$	✓	✓
$x \mapsto 4, 4 * y \mapsto 4, 4$		
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$		
$(x \mapsto 4, 4 * \mathrm{true}) \wedge (y \mapsto 4, 4 * \mathrm{true})$		

Exercises

(from Parkinson)

A:

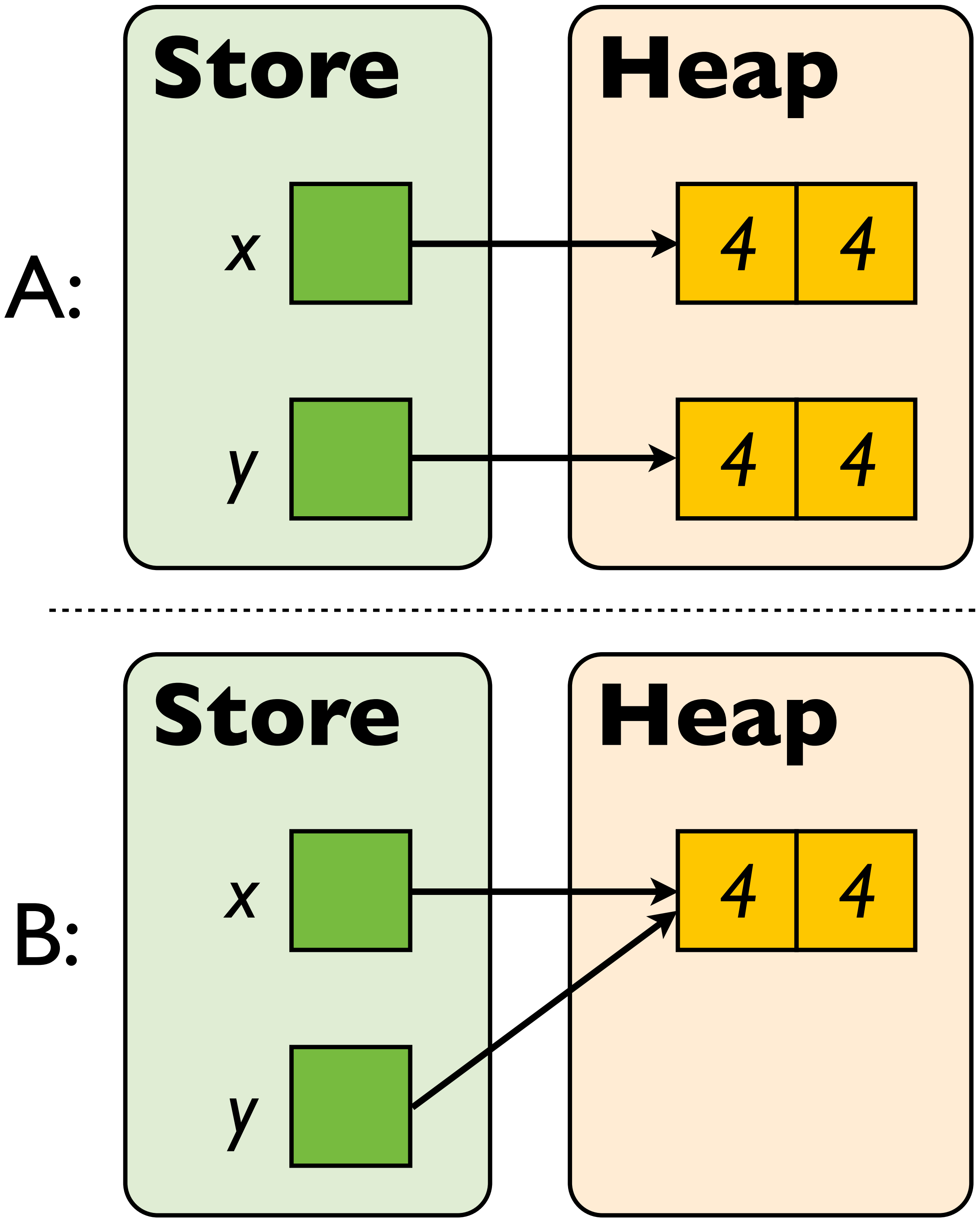
B:



	A	B
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 * \text{true}$	✓	✓
$x \mapsto 4, 4 * y \mapsto 4, 4$	✓	✗
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$		
$(x \mapsto 4, 4 * \text{true}) \wedge (y \mapsto 4, 4 * \text{true})$		

Exercises

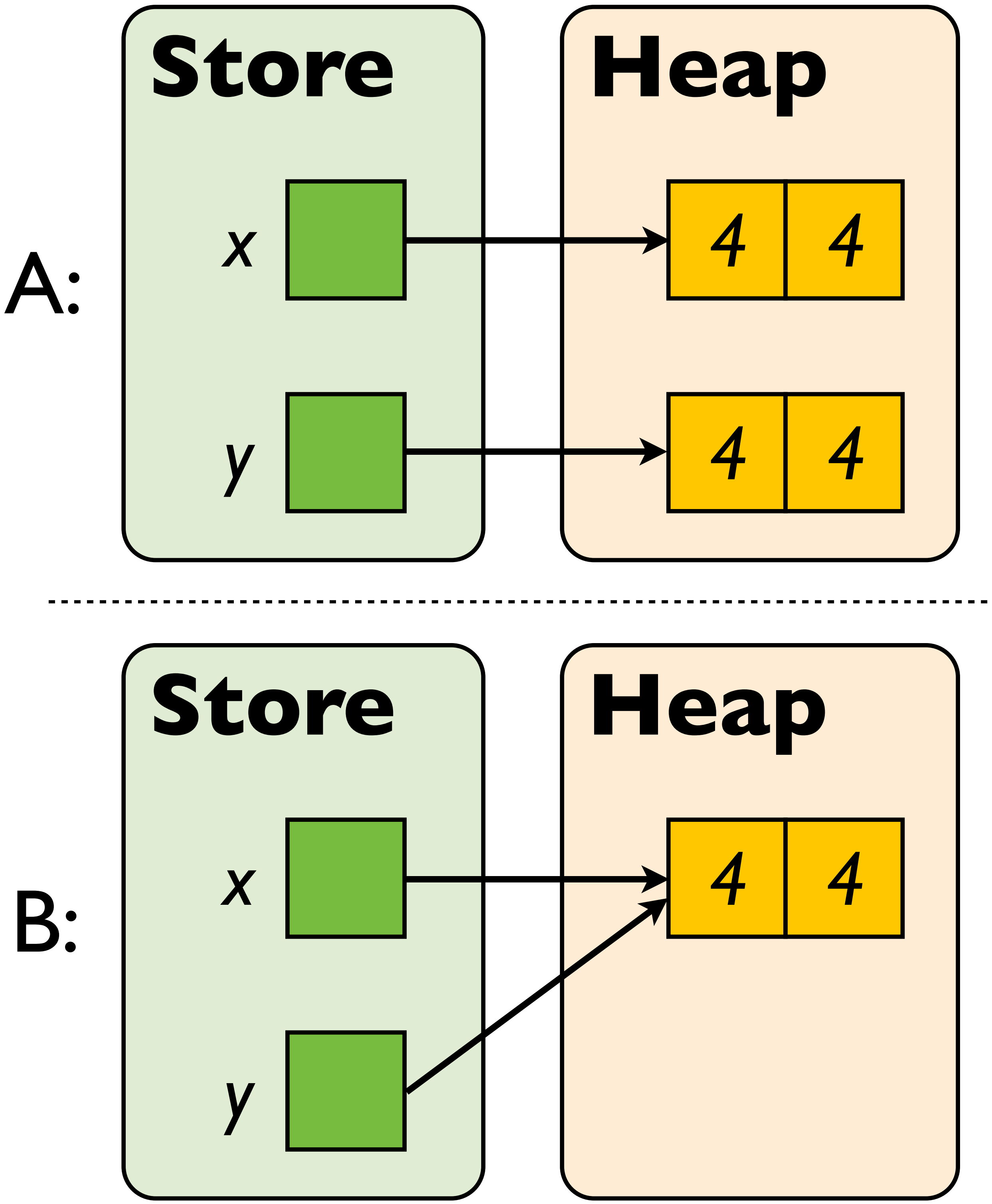
(from Parkinson)



	A	B
$x \mapsto 4, 4$	X	✓
$x \mapsto 4, 4 * \mathrm{true}$	✓	✓
$x \mapsto 4, 4 * y \mapsto 4, 4$	✓	X
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$	X	✓
$(x \mapsto 4, 4 * \mathrm{true}) \wedge (y \mapsto 4, 4 * \mathrm{true})$		

Exercises

(from Parkinson)



	A	B
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 * \mathrm{true}$	✓	✓
$x \mapsto 4, 4 * y \mapsto 4, 4$	✓	✗
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$	✗	✓
$(x \mapsto 4, 4 * \mathrm{true}) \wedge (y \mapsto 4, 4 * \mathrm{true})$	✓	✓

Semantics of separating implication

$$\boxed{s, h \models p \mathrel{-\!\!*} q}$$

- aka the magic wand
- informally: asserts that extending *h* with a disjoint part *h'* that satisfies *p* results in a new heap satisfying *q*
- metatheoretic uses, e.g. proving completeness results

$\wedge$ versus $*$

(from Parkinson)

Similarities

$$p \wedge q \quad \text{iff} \quad q \wedge p \qquad\qquad p * q \quad \text{iff} \quad q * p$$

$$p \wedge \text{true} \quad \text{iff} \quad p \qquad\qquad p * \text{emp} \quad \text{iff} \quad p$$

$$p \wedge (p \Rightarrow q) \quad \text{implies} \quad q \qquad\qquad p * (p \mathrel{-\!\!*} q) \quad \text{implies} \quad q$$

Differences

$$p \quad \text{implies} \quad p \wedge p \qquad\qquad \mathit{one} \quad \text{does not imply} \quad \mathit{one} * \mathit{one}$$

$$p \wedge p \quad \text{implies} \quad p \qquad\qquad \mathit{one} * \mathit{one} \quad \text{does not imply} \quad \mathit{one}$$

where *one* is defined by: $\exists x, y.\ x \mapsto y$

Unsatisfiable...?

$p \wedge \neg p$ $\qquad\qquad$ $p * \neg p$

Unsatisfiable...?

$p \wedge \neg p$ $\qquad\qquad$ $p * \neg p$

"to understand separation logic assertions you should always think locally"

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

Some program constructs for pointers

$v := e$	variable assignment
$v := [e]$	fetch assignment
$[e] := f$	heap assignment
$v :=$ cons$(e1, \ldots, en)$	allocation assignment
dispose(e)	pointer disposal

Some program constructs for pointers

$v := e$ variable assignment

$v := [e]$ fetch assignment

- *evaluate e (with respect to store) to get a location l*
- *fault if l is not in the heap*
- *otherwise assign contents of l in heap to variable v*

Some program constructs for pointers

v := e — variable assignment

[e] := f — heap assignment

- *evaluate e (with respect to store) to get a location l*
- *fault if l is not in the heap*
- *otherwise assign value of f as contents of l in the heap*

Some program constructs for pointers

$v := e$ variable assignment

- *choose n consecutive locations not in the heap*
- *...say l, l+1, ...*
- *extend the heap by adding l, l+1, ... to it*
- *assign l to variable v in the store*
- *assign values of e1, ..., en to contents of l, l+1, ...*

$v :=$ cons$(e1, \ldots, en)$ allocation assignment

Some program constructs for pointers

v := e — variable assignment

- *evaluate e (with respect to store) to get a location l*
- *fault if l is not in the heap*
- *otherwise remove l from the heap*

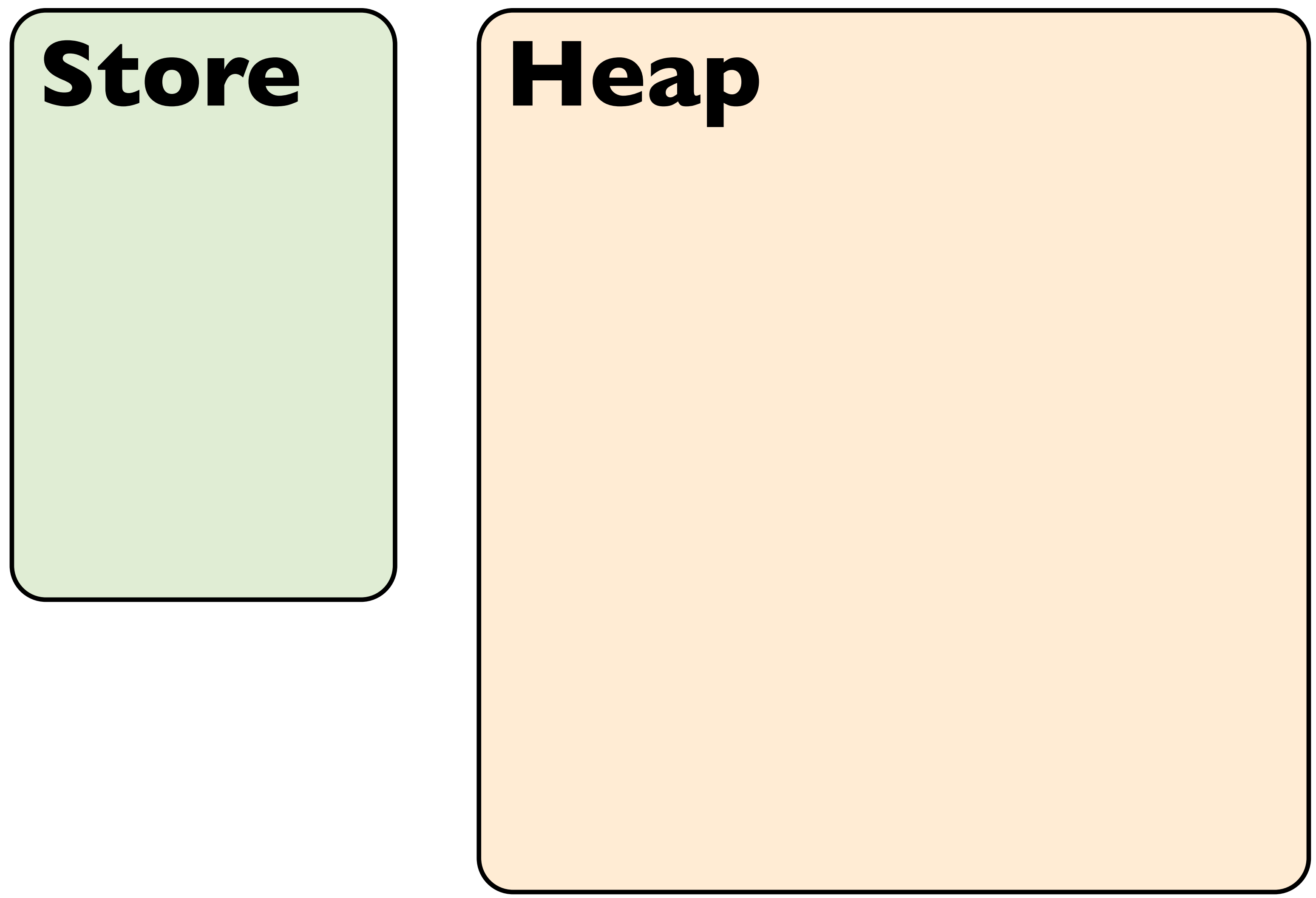
dispose(e) — pointer disposal

Some program constructs for pointers

$v := e$	variable assignment
$v := [e]$	fetch assignment
$[e] := f$	heap assignment
$v :=$ cons$(e1, \ldots, en)$	allocation assignment
dispose(e)	pointer disposal

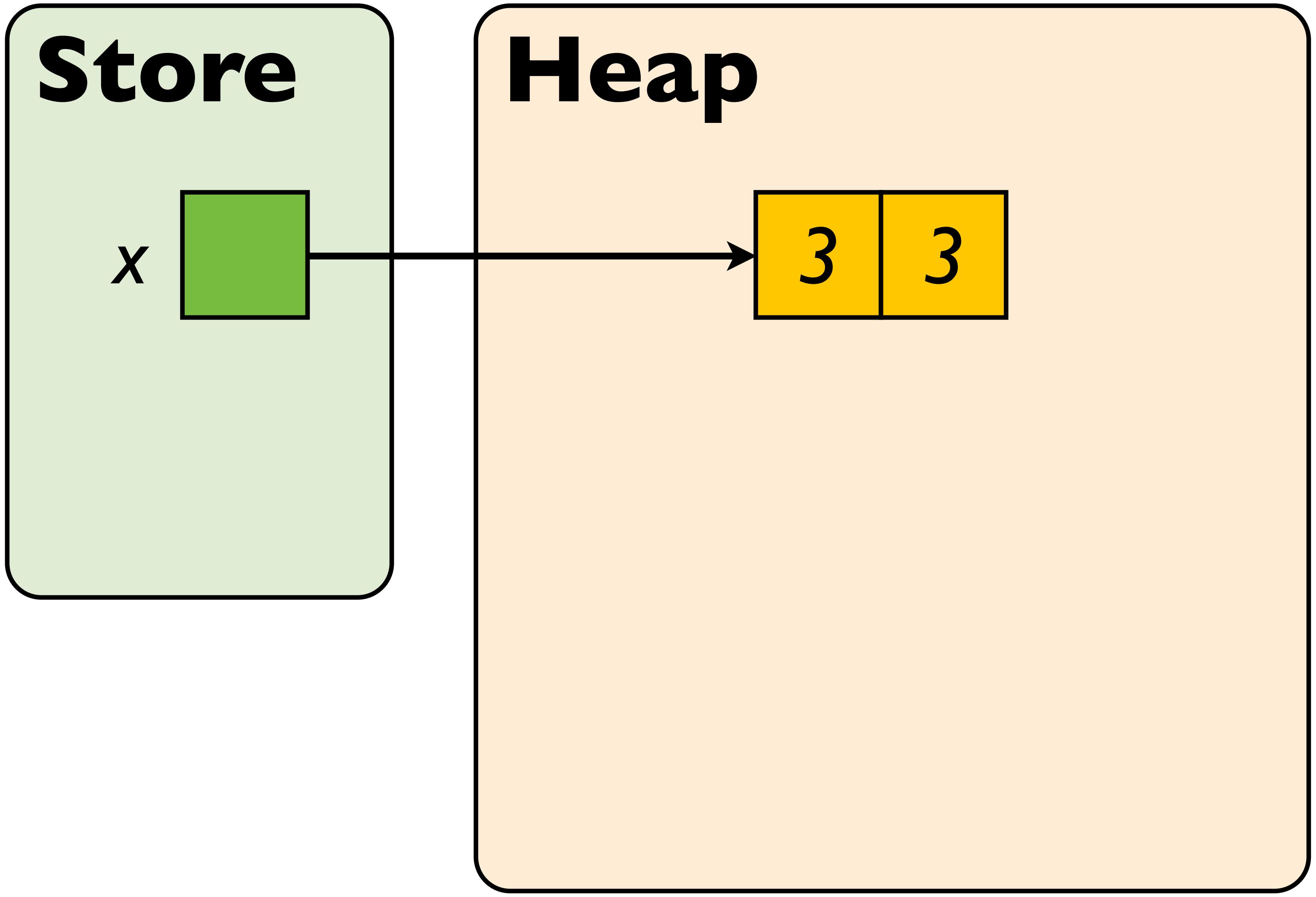
Example program
(from Parkinson)

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



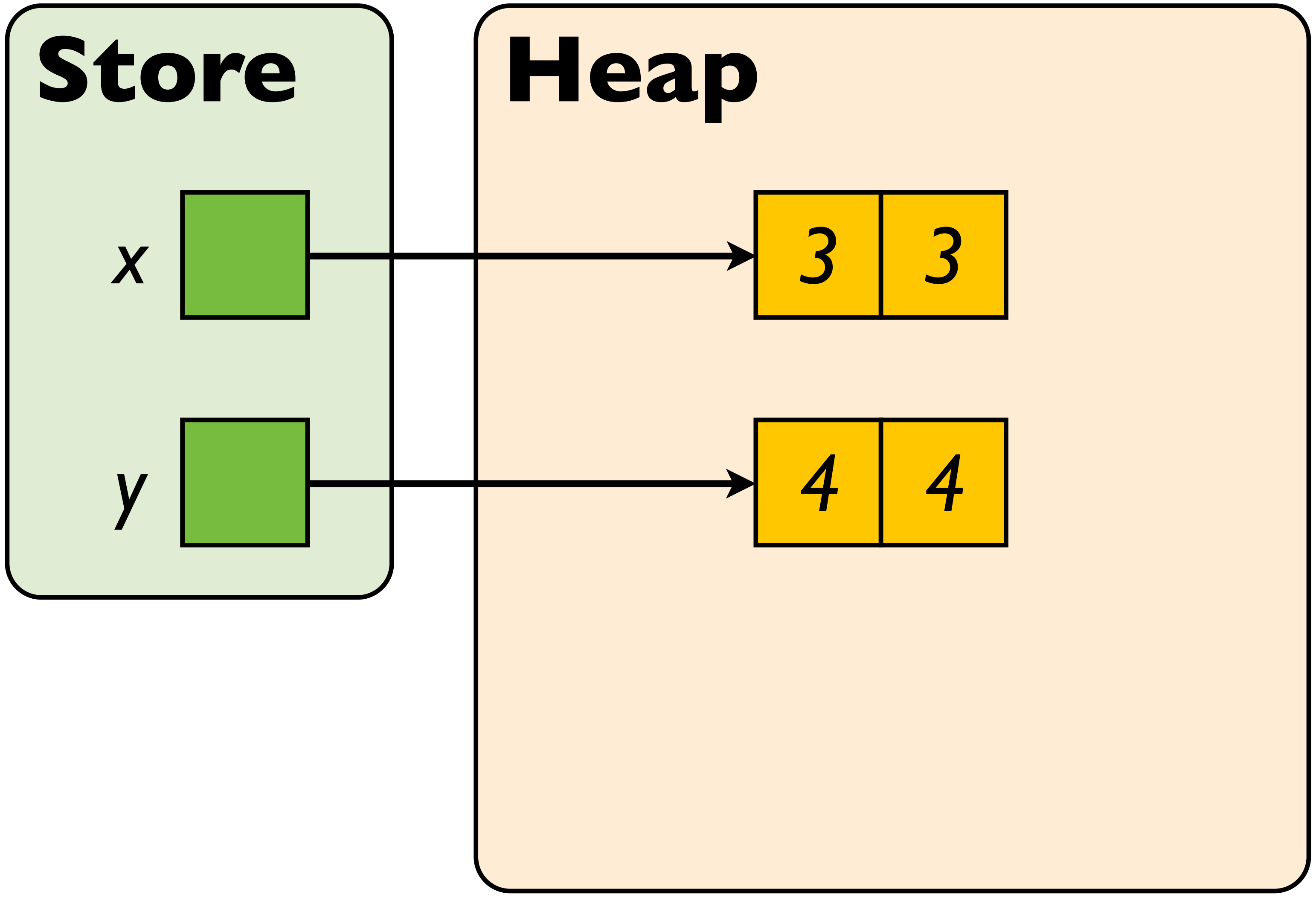
Example program
(from Parkinson)

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



Example program
(from Parkinson)

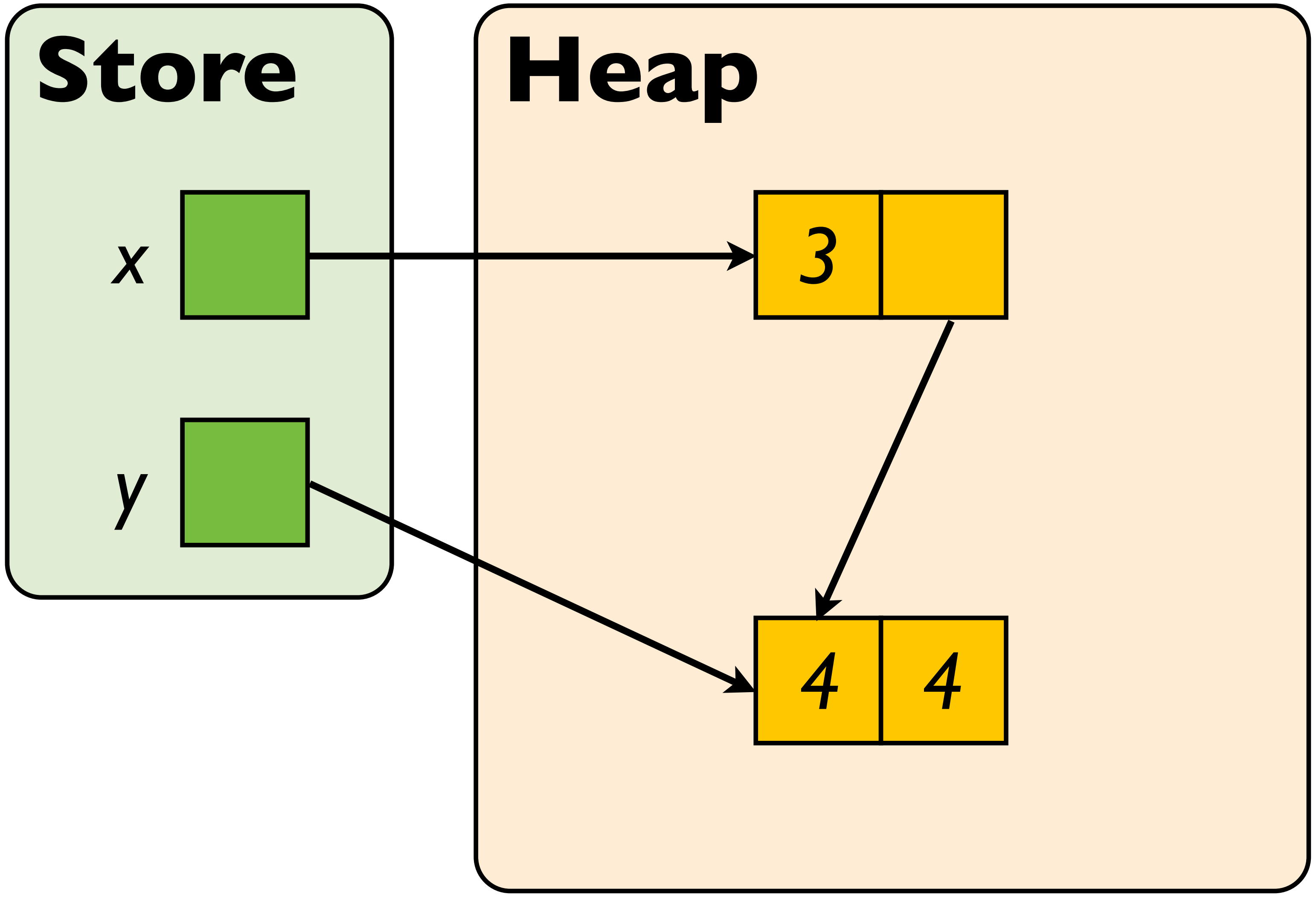
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example program

(from Parkinson)

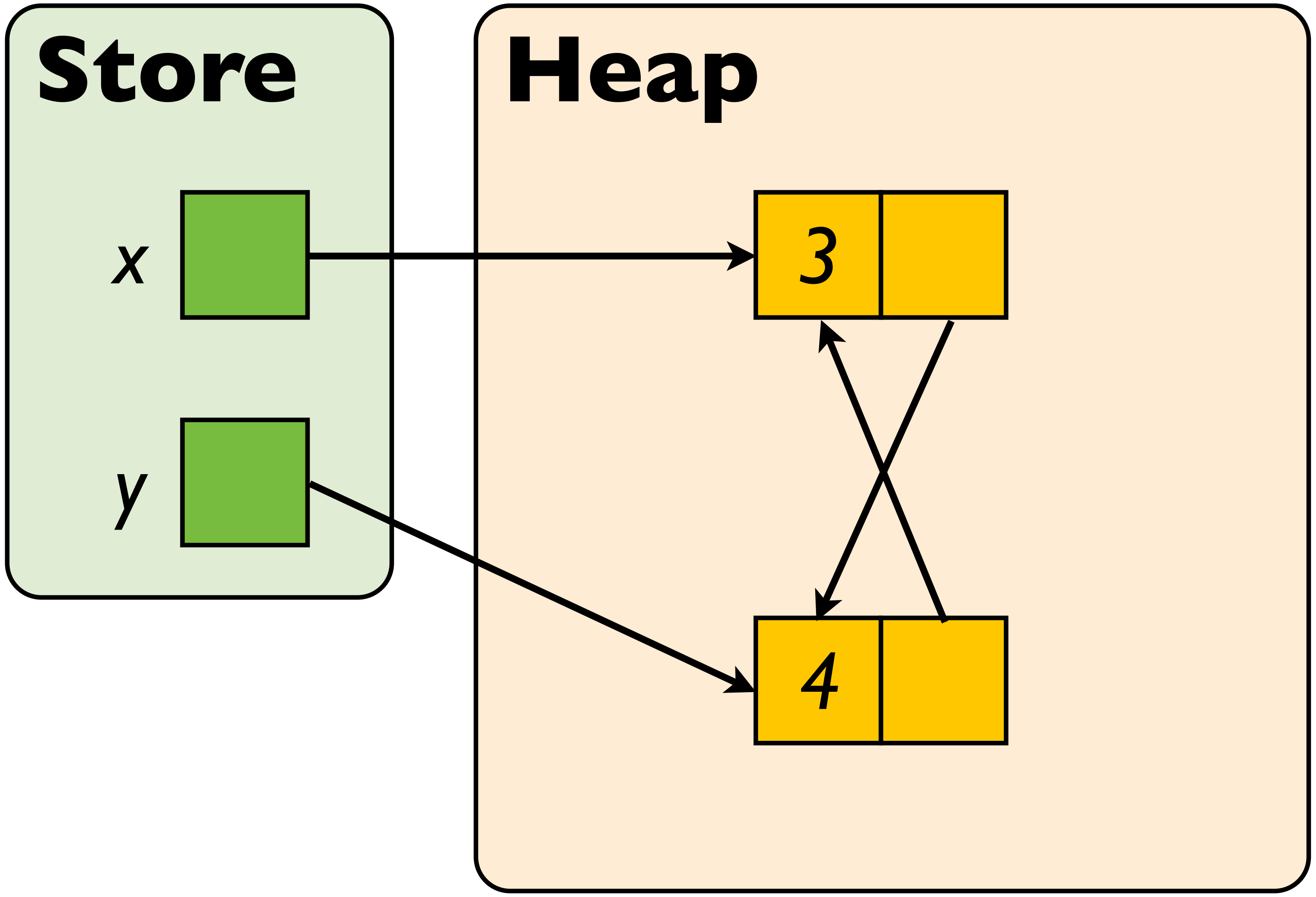
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example program

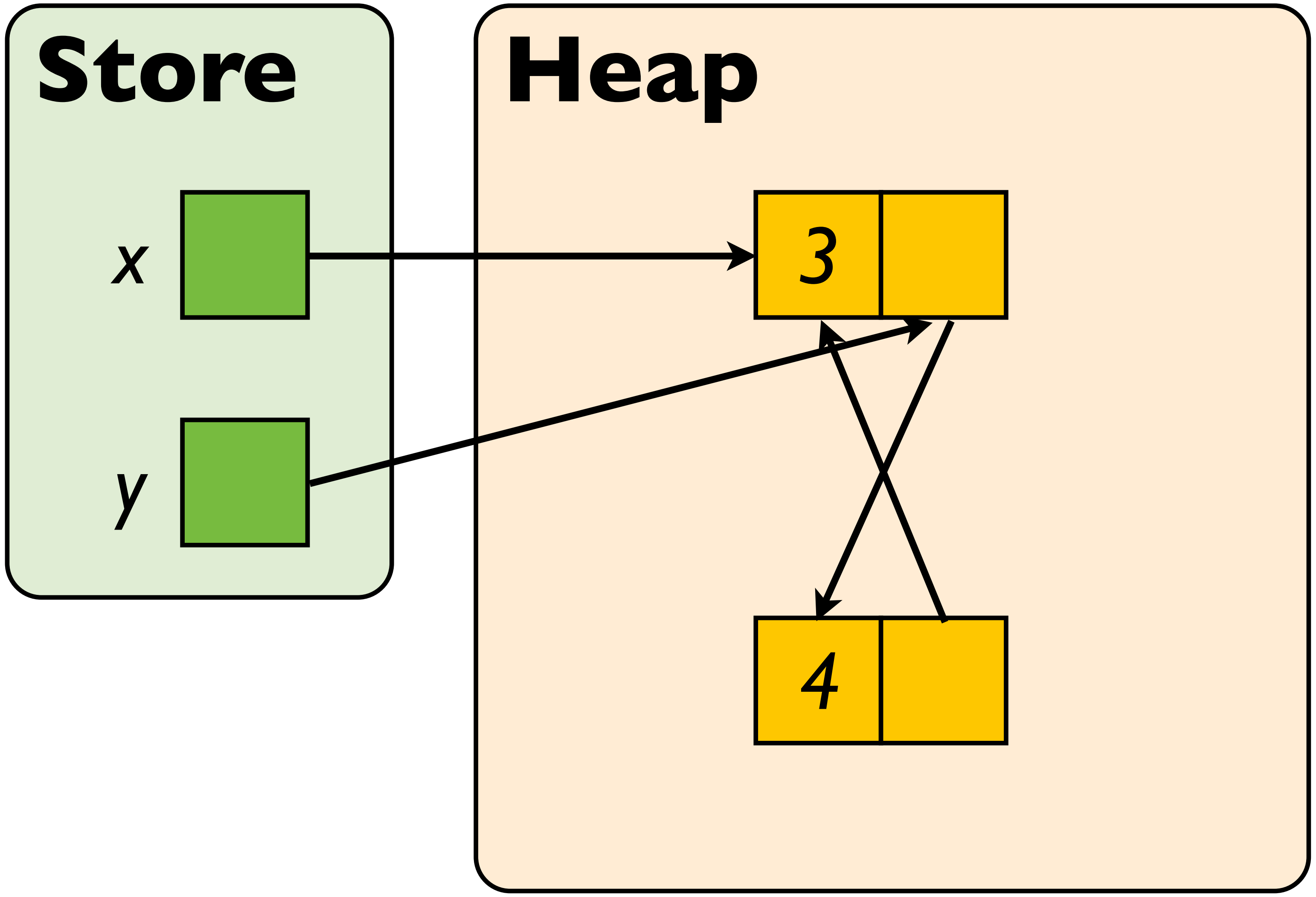
(from Parkinson)

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



Example program
(from Parkinson)

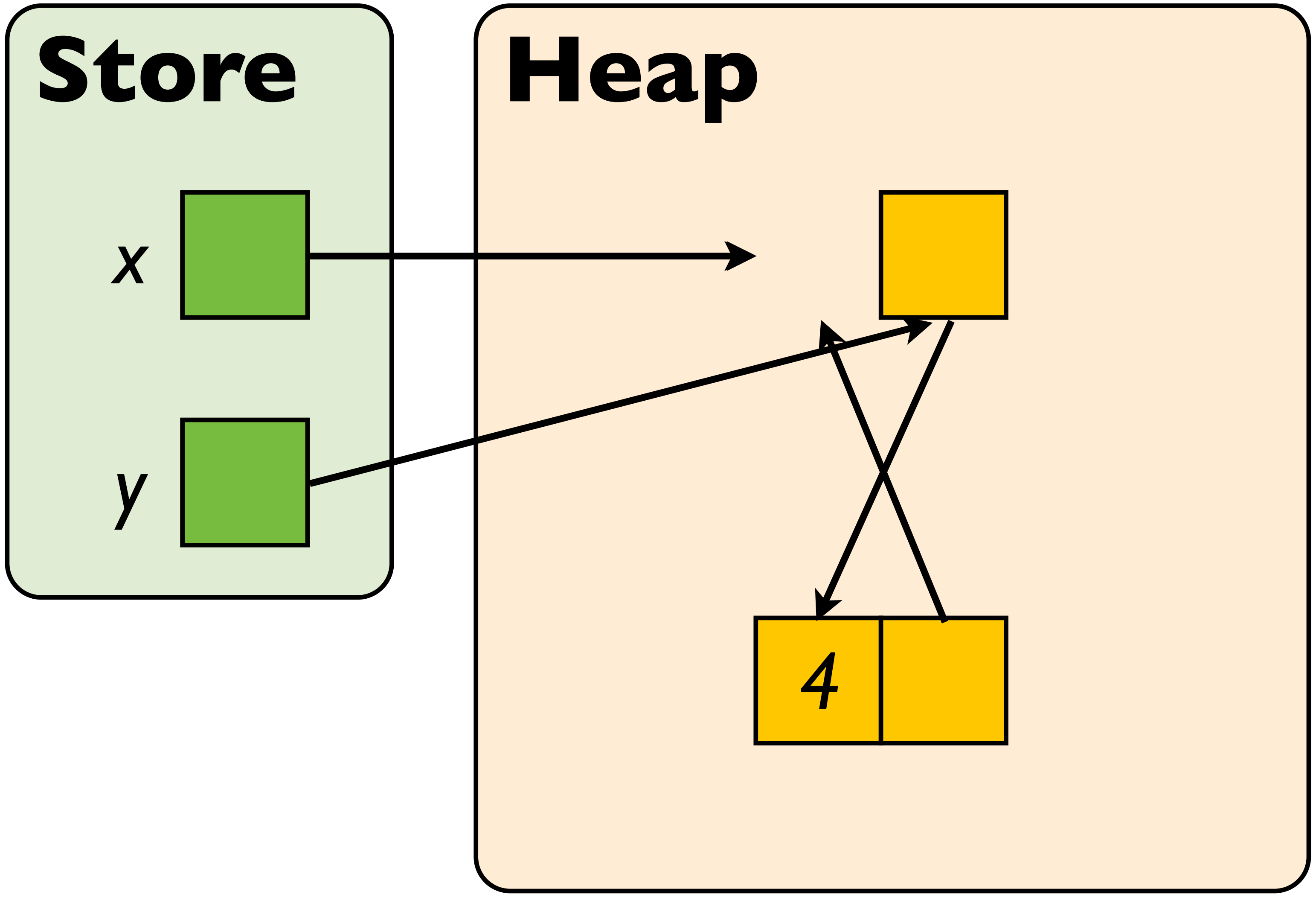
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example program

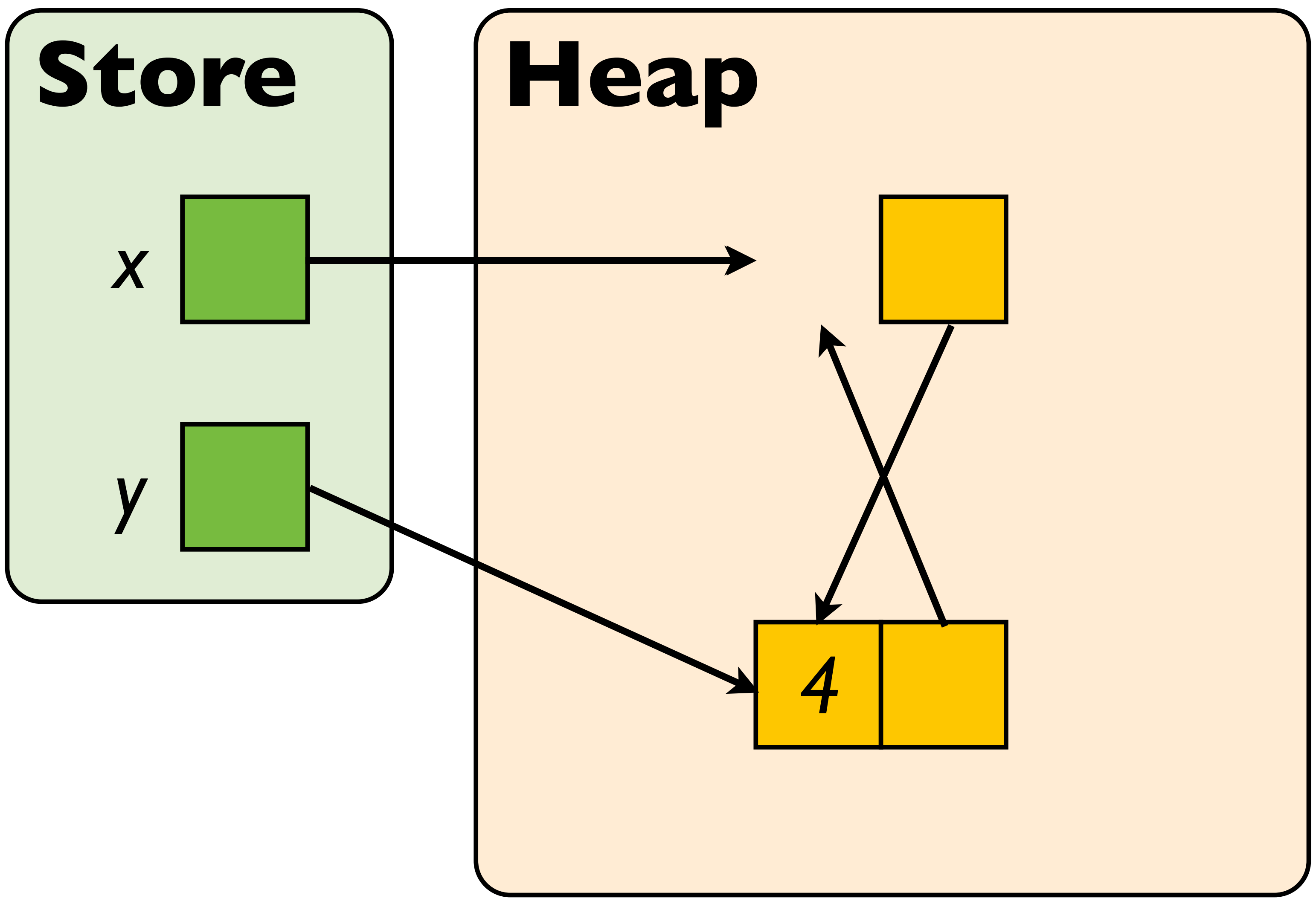
(from Parkinson)

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



Example program
(from Parkinson)

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



New axioms for separation logic

$$\{e \mapsto \_\}\ [e] := f\ \{e \mapsto f\}$$

$$\{e \mapsto \_\}\ \mathrm{dispose}(e)\ \{\mathrm{emp}\}$$

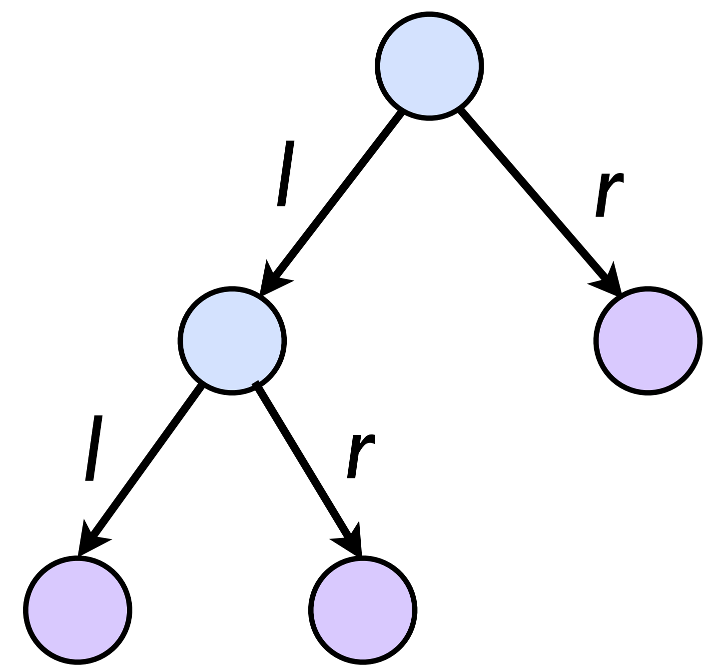
$$\{X = x \wedge e \mapsto Y\}\ x := [e]\ \{e[X/x] \mapsto Y \wedge Y = x\}$$

$$\{\mathrm{emp}\}\ x := \mathrm{cons}(e_0, \ldots, e_n)\ \{x \mapsto e_0, \ldots, e_n\}$$

(where e |->_ means "the location given by the evaluation of e points to *something*")

Recall the problem in verifying this program:

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



$\{\ tree(p) \wedge reach(p,n) \wedge \neg reach(p,m) \wedge allocated(m)$
$\wedge\ m.f = m' \wedge \neg allocated(q)\ \}$

DispTree(p)

$\{\ \neg allocated(n) \wedge \neg reach(p,m) \wedge allocated(m)$
$\wedge\ m.f = m' \wedge \neg allocated(q)\ \}$

Framing!

The frame rule

(the *most* important rule!)

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

- side condition: no variable modified by C appears free in r

- enables local reasoning: programs that execute correctly in a small state ($\models p$) also execute correctly in a bigger state ($\models p*r$)

Warning: interpretation of triples!

- interpretation of triples slightly stronger in separation logic than partial correctness

$$\models \{p\}\ \ C\ \ \{q\}$$

- "if *C* is executed on a state satisfying *p*, then it will not fault, and if it terminates, that state will satisfy *q*"

Why no faulting?

- if we don't insist that programs do not fault, then strange "proofs" like the following will be possible:

$$\frac{\{\mathrm{true}\}\ \ [x] := 7\ \ \{\mathrm{true}\}}{\{\mathrm{true} * x \mapsto 4\}\ \ [x] := 7\ \ \{\mathrm{true} * x \mapsto 4\}}$$

Next on the agenda

(1) model of program states for separation logic ✓

(2) assertions and spatial connectives ✓

(3) axioms and inference rules ✓

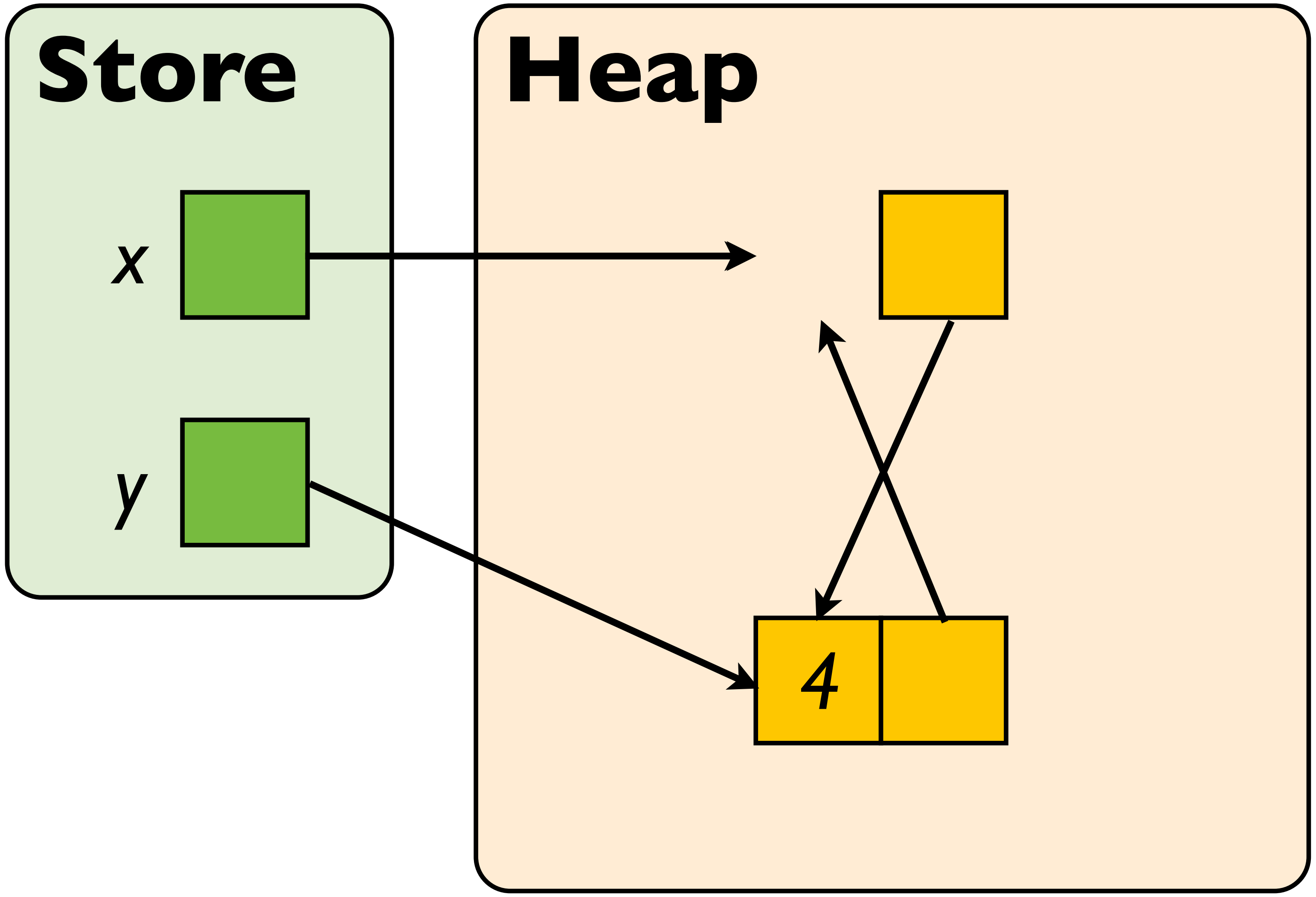
(4) program proofs

Exercise (for next time): 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}



Exercise (for next time): 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}

- the frame rule is crucial!
- reason forwards
  e.g. use the "forward" assignment axiom
- try a proof outline (proof trees too large)

Summary

- 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

Thank you! Questions?

Next lecture:

- writing proofs in separation logic
- inductive definitions in assertions