

# An Introduction to Separation Logic (1/2)

Matthew Parkinson

# + Overview

---

## First part (Today) – Introduction

- Motivation
- In the beginning...
- The Logic
- Some examples

## Second part (Thursday) – Harder stuff

- Modularity
- Concurrency
- Decidability

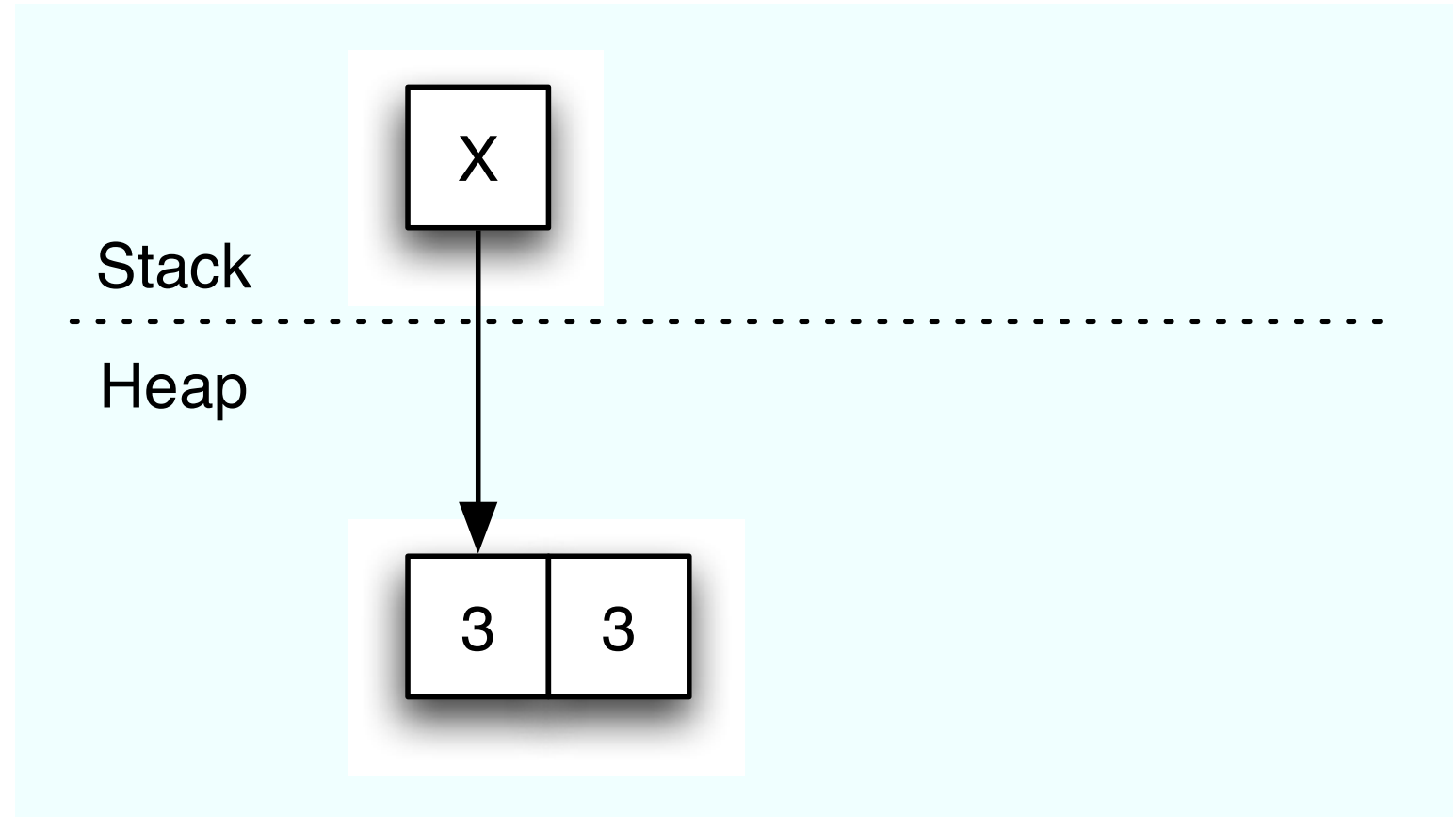
# Motivation

## + Example program

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

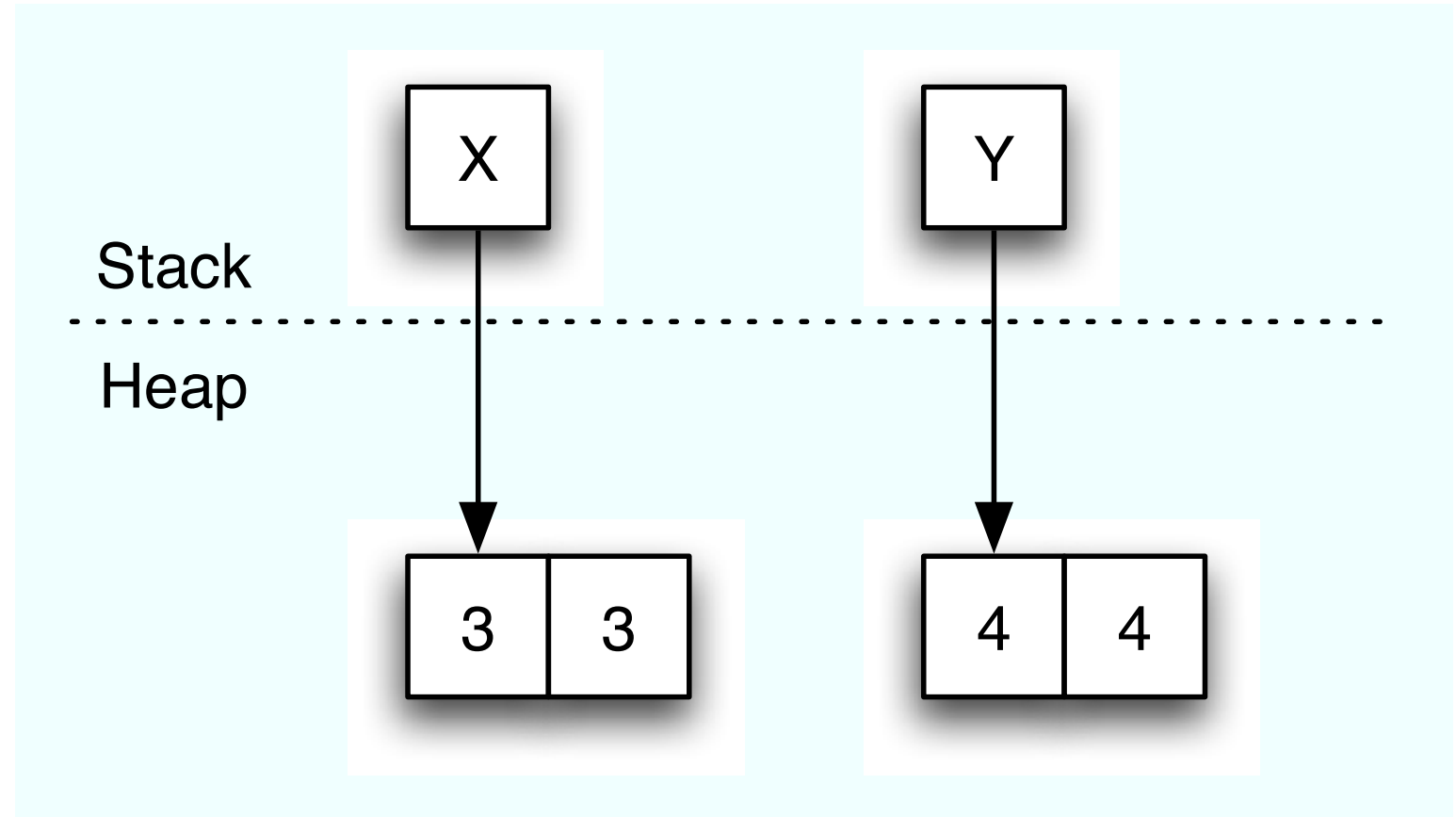
## + Example program

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



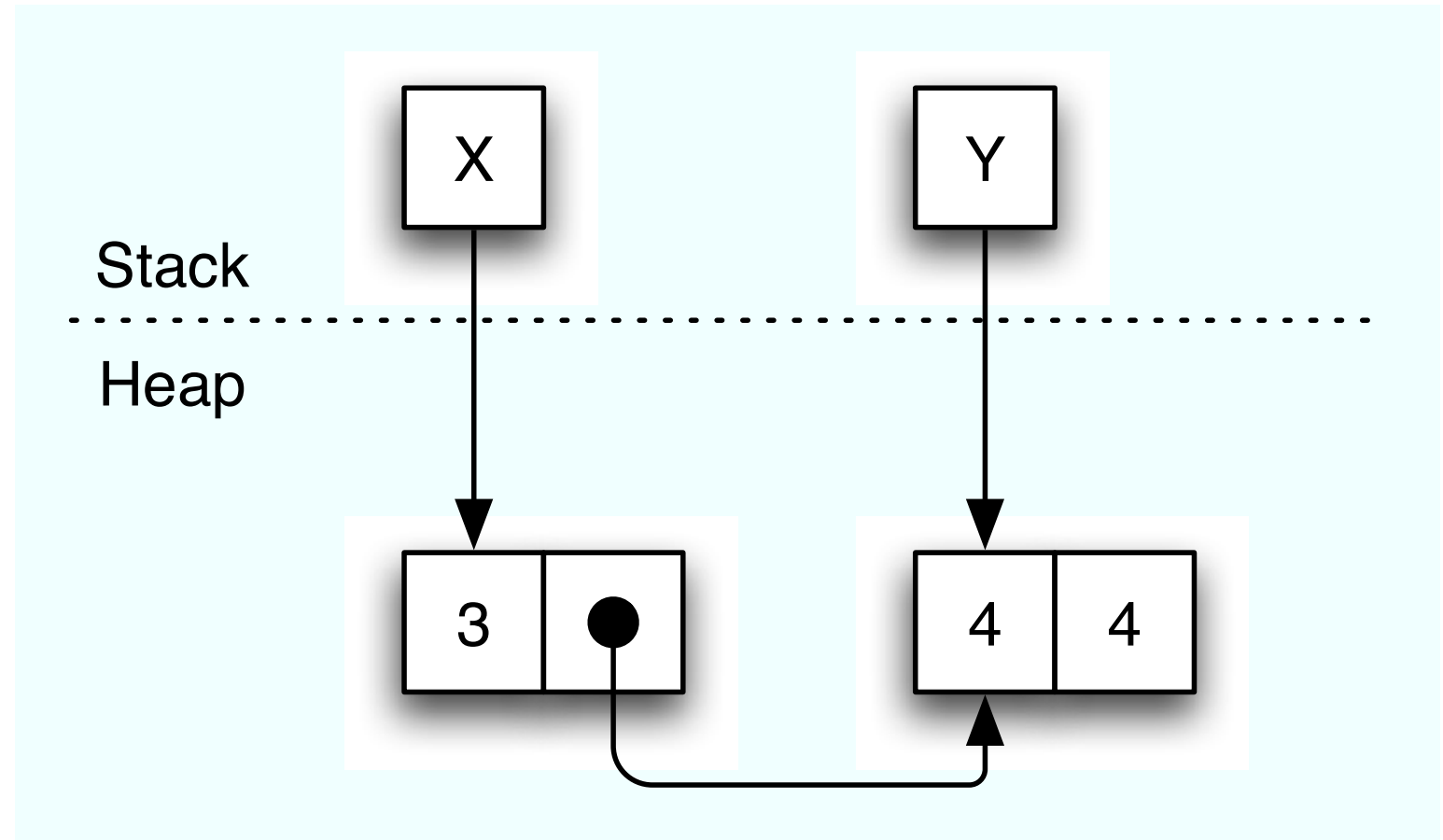
## + Example program

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



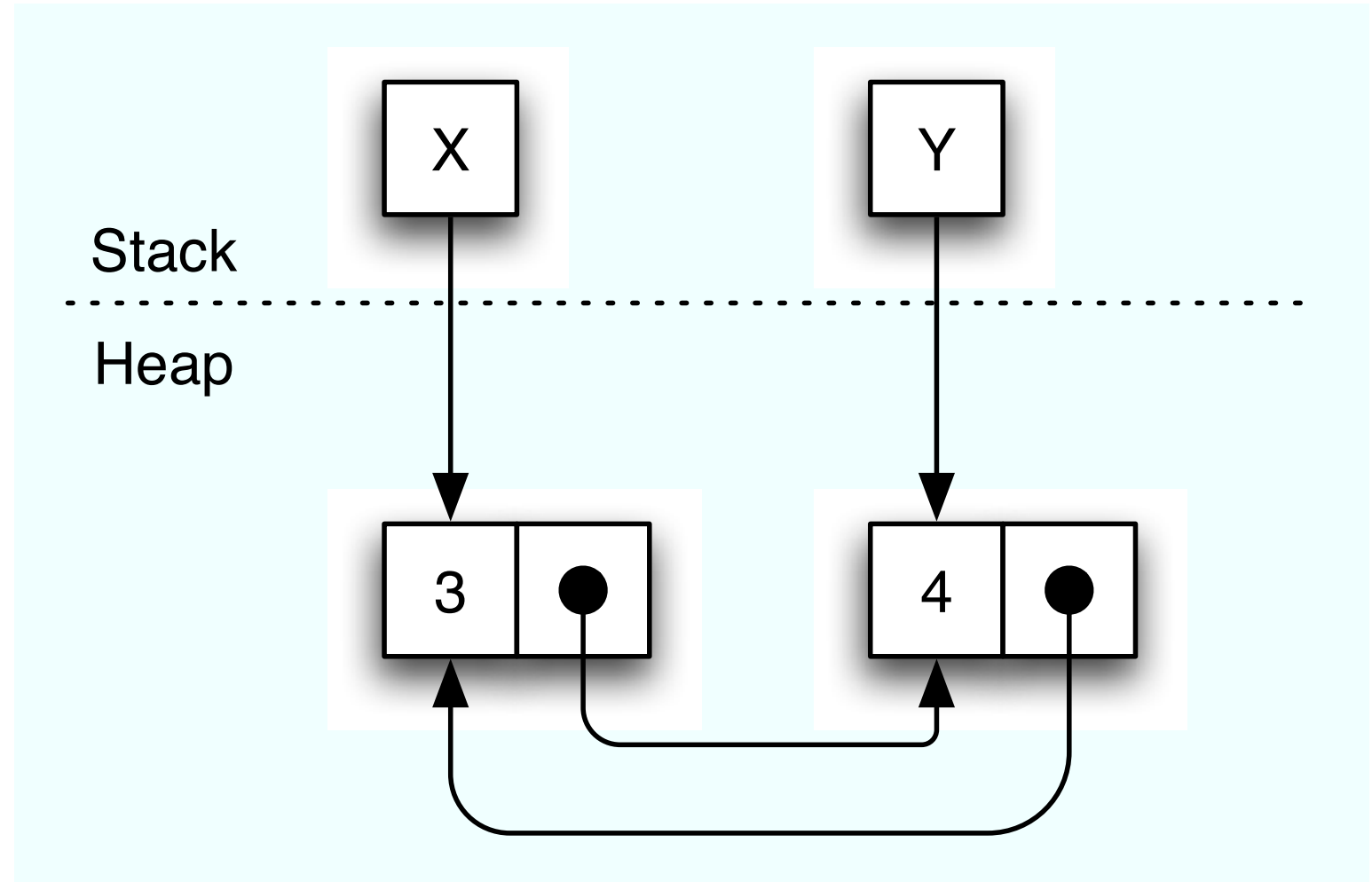
## + Example program

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



## + Example program

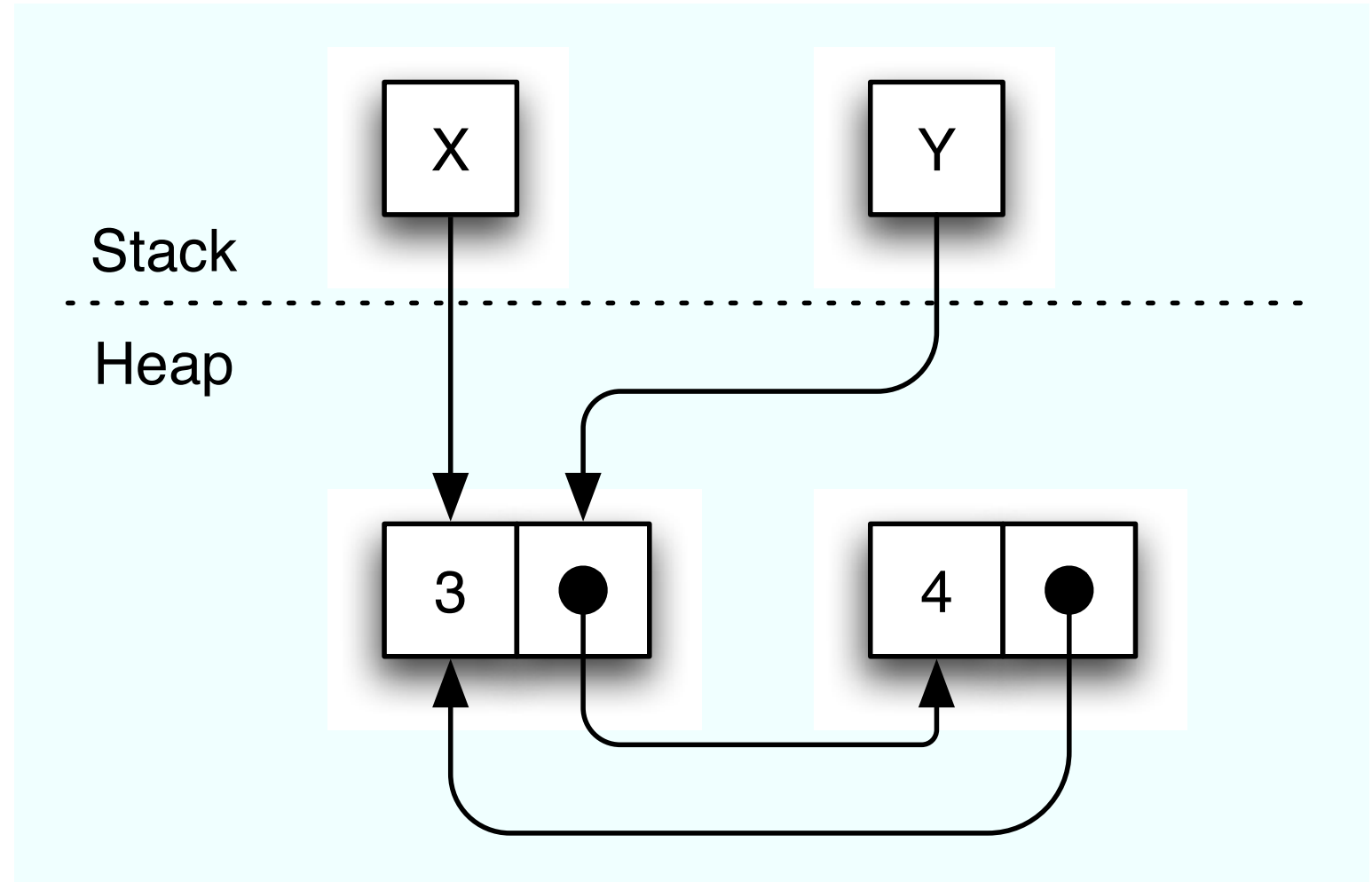
```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```





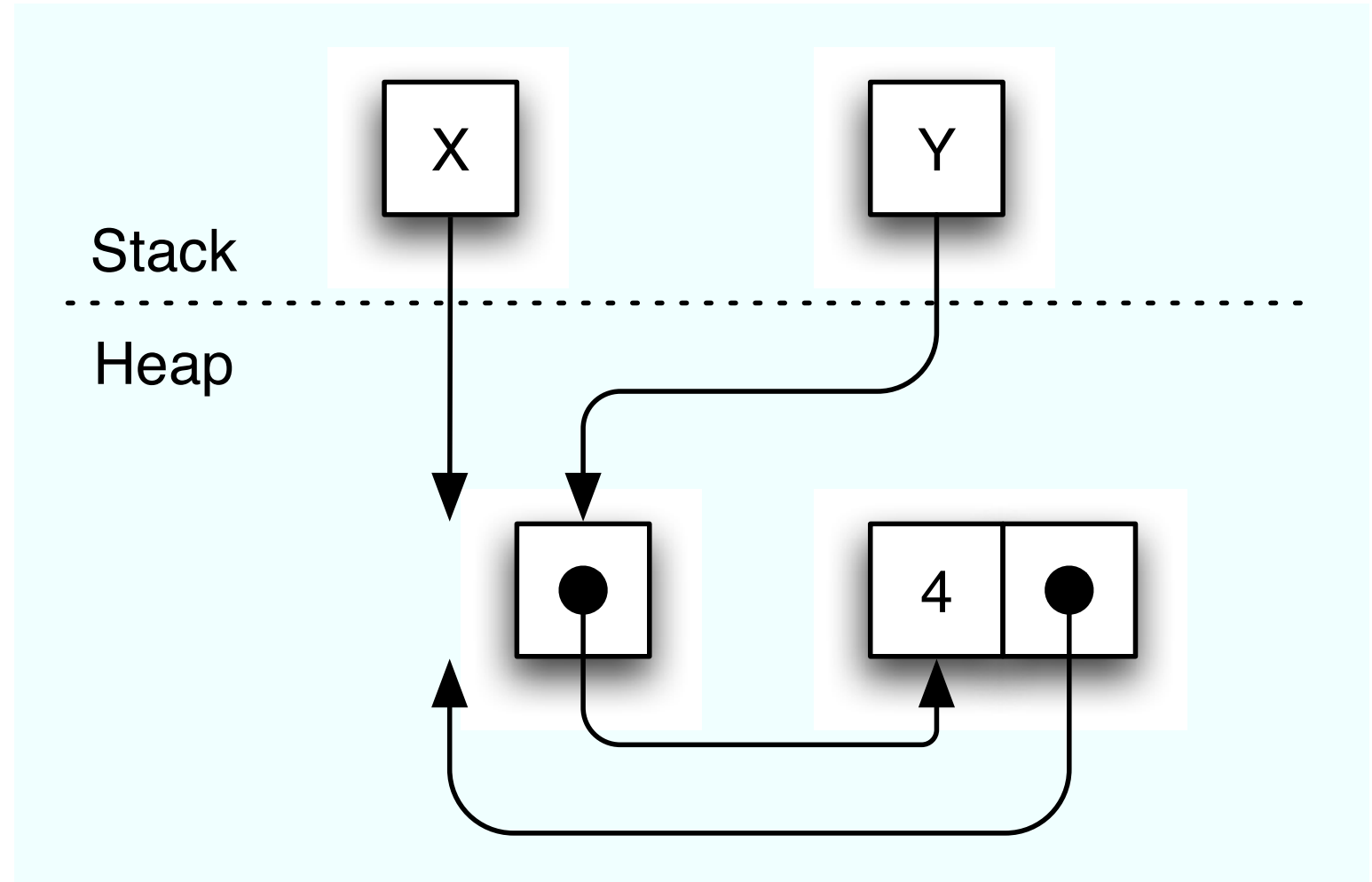
## + Example program

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



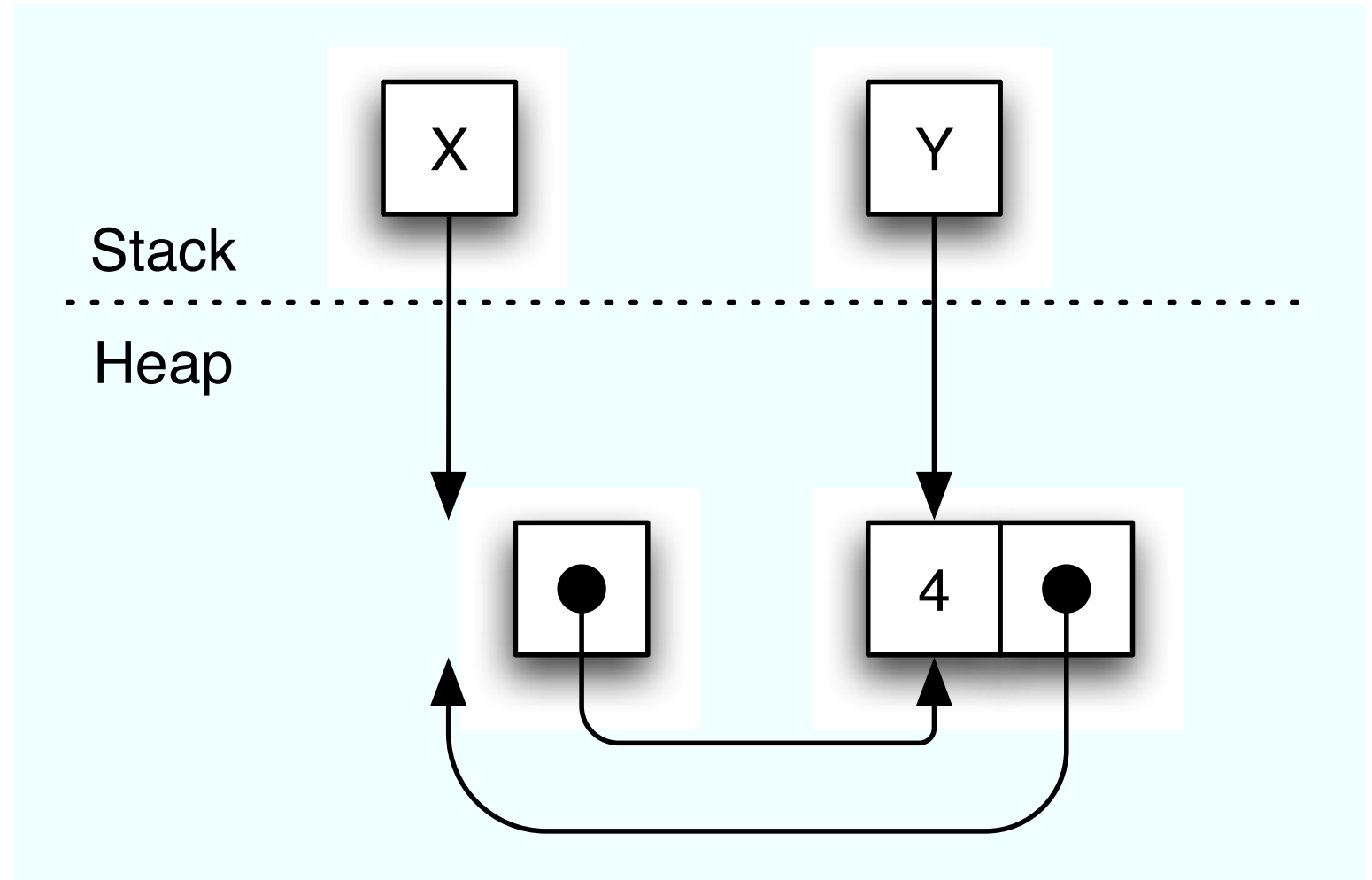
## + Example program

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



## + Example program

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



## + Why Separation Logic?

Consider the following piece of code

[Note: read  $[x]$  as indirect through  $x$  to the heap.]

$[y] = 4;$

$[z] = 5;$

Guarantee( $[y] \neq [z]$ )

Need to know things locations are different.

## + Why Separation Logic?

Consider the following piece of code

[Note: read  $[x]$  as indirect through  $x$  to the heap.]

Assume( $y \neq z$ )

$[y] = 4;$

$[z] = 5;$

Guarantee( $[y] \neq [z]$ )

Need to know things locations are different.

- Add assertions?

## + Why Separation Logic?

Consider the following piece of code

[Note: read  $[x]$  as indirect through  $x$  to the heap.]

Assume( $[x] = 3$ )

Assume( $y \neq z$ )

$[y] = 4;$

$[z] = 5;$

Guarantee( $[y] \neq [z]$ )

Guarantee( $[x] = 3$ )

Need to know things locations are different.

- Add assertions?

We need to know when things stay the same but how?

## + Why Separation Logic?

Consider the following piece of code

[Note: read  $[x]$  as indirect through  $x$  to the heap.]

Assume( $[x] = 3 \wedge x \neq y \wedge x \neq z$ )

Assume( $y \neq z$ )

$[y] = 4;$

$[z] = 5;$

Guarantee( $[y] \neq [z]$ )

Guarantee( $[x] = 3$ )

Need to know things locations are different.

- Add assertions?

We need to know when things stay the same but how?

- Add assertions?

We want a general concept of things not being affected.

$$\frac{\{P\}C\{Q\}}{\{[x] = 3 \wedge P\}C\{Q \wedge [x] = 3\}}$$



We want a general concept of things not being affected.

$$\frac{\{P\}C\{Q\}}{\{\textcolor{red}{R} \wedge P\}C\{Q \wedge \textcolor{red}{R}\}}$$

What are the conditions on  $C$  and  $R$ ?

- Very hard to define if reasoning about a heap and aliasing

We want a general concept of things not being affected.

$$\frac{\{P\}C\{Q\}}{\{\textcolor{red}{R} \wedge P\}C\{Q \wedge \textcolor{red}{R}\}}$$

What are the conditions on  $C$  and  $R$ ?

- Very hard to define if reasoning about a heap and aliasing

This is where separation logic comes in

$$\frac{\{P\}C\{Q\}}{\{R * P\}C\{Q * R\}}$$

Introduces new connective  $*$  used to separate state.

In the beginning...

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge r$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} weakl$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee l$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A} weakr$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow l$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow r$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} contrl$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg l$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg r$$

$$\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} contrr$$

$$\overline{A \vdash A} BS$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge r$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \text{ weakl}$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee l$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A} \text{ weakr}$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow l$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow r$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \text{ contrl}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg l$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg r$$

$$\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} \text{ contrr}$$

$$\overline{A \vdash A} \text{ BS}$$

# + Substructural Logic +

Substructural logics consider the removal of structural rules:

- Weakening
- Contraction
- Commutativity
- Associativity

## Examples

- Philosophy (Relevant Logics)
- Linguistics (Lambek Calculus)
- Computer Science (Linear Logic, Bunched Implications, ...)

## + Bunched Implications

$\wedge$  admits weakening and contraction, but  $*$  does not.

$$\frac{\Delta(\Gamma) \vdash \psi}{\Delta(\Gamma \wedge \Gamma') \vdash \psi} \qquad \frac{\Delta(\Gamma \wedge \Gamma) \vdash \psi}{\Delta(\Gamma) \vdash \psi}$$

However we don't have

$$\frac{\Delta(\Gamma) \vdash \psi}{\Delta(\Gamma * \Gamma') \vdash \psi} \qquad \frac{\Delta(\Gamma * \Gamma) \vdash \psi}{\Delta(\Gamma) \vdash \psi}$$

Key concept BI mixes substructural logic with classical/intuitionistic logic.

If this doesn't make sense, **Don't Panic!**

# Separation Logic



$P, Q$	$::=$	$false$	Logical false
		$P \wedge Q$	Classical conjunction
		$P \vee Q$	Classical disjunction
		$P \Rightarrow Q$	Classical implication
		$P * Q$	Separating conjunction
		$P \multimap Q$	Separating implication
		$E = E$	Expression value equality
		$E \mapsto E$	points to
		$empty$	empty heap
		$\exists x. P$	existential quantifier

We use  $E$  to range over integer expressions ( $E$  does not contain indirection through the heap),  $x$  over variables and  $C$  over commands.

Assertions are given with respect to a heap,  $H$ , and stack,  $S$ .

$$S : \text{Var} \rightarrow \text{Int} \quad H : \text{Loc} \rightarrow \text{Int} \quad \text{where } \text{Loc} \subseteq \text{Int}$$

$S, H \models \text{false}$       **never satisfied**

$$S, H \models P \wedge Q \quad \text{iff } S, H \models P \quad \wedge \quad S, H \models Q$$

$$S, H \models P \vee Q \quad \text{iff } S, H \models P \quad \vee \quad S, H \models Q$$

$$S, H \models P \Rightarrow Q \quad \text{iff } S, H \models P \quad \Rightarrow \quad S, H \models Q$$

$$S, H \models E = E' \quad \text{iff } \llbracket E \rrbracket_S = \llbracket E' \rrbracket_S$$

$$S, H \models \text{empty} \quad \text{iff } H = \{\}$$

We use  $\llbracket E \rrbracket_S$  to mean evaluation with respect to the stack,  $S$ .

Assertions are given with respect to a heap,  $H$ , and stack,  $S$ .

$$S : \text{Var} \rightarrow \text{Int} \quad H : \text{Loc} \rightarrow \text{Int} \quad \text{where } \text{Loc} \subseteq \text{Int}$$

$$S, H \models \text{false} \quad \text{never satisfied}$$

$$S, H \models P \wedge Q \quad \text{iff } S, H \models P \quad \wedge \quad S, H \models Q$$

$$S, H \models P \vee Q \quad \text{iff } S, H \models P \quad \vee \quad S, H \models Q$$

$$S, H \models P \Rightarrow Q \quad \text{iff } S, H \models P \quad \Rightarrow \quad S, H \models Q$$

$$S, H \models E = E' \quad \text{iff } \llbracket E \rrbracket_S = \llbracket E' \rrbracket_S$$

$$S, H \models \text{empty} \quad \text{iff } H = \{\}$$

We use  $\llbracket E \rrbracket_S$  to mean evaluation with respect to the stack,  $S$ .

Assertions are given with respect to a heap,  $H$ , and stack,  $S$ .

$$S : \text{Var} \rightarrow \text{Int} \quad H : \text{Loc} \rightarrow \text{Int} \quad \text{where } \text{Loc} \subseteq \text{Int}$$

$$S, H \models \text{false} \quad \text{never satisfied}$$

$$S, H \models P \wedge Q \quad \text{iff } S, H \models P \quad \wedge \quad S, H \models Q$$

$$S, H \models P \vee Q \quad \text{iff } S, H \models P \quad \vee \quad S, H \models Q$$

$$S, H \models P \Rightarrow Q \quad \text{iff } S, H \models P \quad \Rightarrow \quad S, H \models Q$$

$$S, H \models E = E' \quad \text{iff } \llbracket E \rrbracket_S = \llbracket E' \rrbracket_S$$

$$S, H \models \text{empty} \quad \text{iff } H = \{\}$$

We use  $\llbracket E \rrbracket_S$  to mean evaluation with respect to the stack,  $S$ .

Assertions are given with respect to a heap,  $H$ , and stack,  $S$ .

$$S : \text{Var} \rightarrow \text{Int} \quad H : \text{Loc} \rightarrow \text{Int} \quad \text{where } \text{Loc} \subseteq \text{Int}$$

$$S, H \models \textit{false} \quad \text{never satisfied}$$

$$S, H \models P \wedge Q \quad \text{iff } S, H \models P \quad \wedge \quad S, H \models Q$$

$$S, H \models P \vee Q \quad \text{iff } S, H \models P \quad \vee \quad S, H \models Q$$

$$S, H \models P \Rightarrow Q \quad \text{iff } S, H \models P \quad \Rightarrow \quad S, H \models Q$$

$$S, H \models E = E' \quad \text{iff } \llbracket E \rrbracket_S = \llbracket E' \rrbracket_S$$

$$S, H \models \textit{empty} \quad \text{iff } H = \{\}$$

We use  $\llbracket E \rrbracket_S$  to mean evaluation with respect to the stack,  $S$ .

Assertions are given with respect to a heap,  $H$ , and stack,  $S$ .

$$S : \text{Var} \rightarrow \text{Int} \quad H : \text{Loc} \rightarrow \text{Int} \quad \text{where } \text{Loc} \subseteq \text{Int}$$

$$S, H \models \textit{false} \quad \text{never satisfied}$$

$$S, H \models P \wedge Q \quad \text{iff } S, H \models P \quad \wedge \quad S, H \models Q$$

$$S, H \models P \vee Q \quad \text{iff } S, H \models P \quad \vee \quad S, H \models Q$$

$$S, H \models P \Rightarrow Q \quad \text{iff } S, H \models P \quad \Rightarrow \quad S, H \models Q$$

$$S, H \models E = E' \quad \text{iff } \llbracket E \rrbracket_S = \llbracket E' \rrbracket_S$$

$$S, H \models \textit{empty} \quad \text{iff } H = \{\}$$

We use  $\llbracket E \rrbracket_S$  to mean evaluation with respect to the stack,  $S$ .

Assertions are given with respect to a heap, H, and stack, S.

$$S : \text{Var} \rightarrow \text{Int} \quad H : \text{Loc} \rightarrow \text{Int} \quad \text{where } \text{Loc} \subseteq \text{Int}$$

$$S, H \models \textit{false} \quad \text{never satisfied}$$

$$S, H \models P \wedge Q \quad \text{iff } S, H \models P \quad \wedge \quad S, H \models Q$$

$$S, H \models P \vee Q \quad \text{iff } S, H \models P \quad \vee \quad S, H \models Q$$

$$S, H \models P \Rightarrow Q \quad \text{iff } S, H \models P \quad \Rightarrow \quad S, H \models Q$$

$$S, H \models E = E' \quad \text{iff } \llbracket E \rrbracket_S = \llbracket E' \rrbracket_S$$

$$S, H \models \textit{empty} \quad \text{iff } H = \{\}$$

We use  $\llbracket E \rrbracket_S$  to mean evaluation with respect to the stack, S.

Now for more complicated semantics ;)

$$\begin{aligned} S, H \models E \mapsto E' \\ \text{iff } \text{dom}(H) = \{ \llbracket E \rrbracket_S \} \wedge H(\llbracket E \rrbracket_S) = \llbracket E' \rrbracket_S \end{aligned}$$

$$\begin{aligned} S, H \models P * Q \\ \text{iff } \exists H_1 H_2. (H_1 \perp H_2) \wedge (H_1 \circ H_2 = H) \wedge (S, H_1 \models P) \wedge (S, H_2 \models Q) \end{aligned}$$

$$\begin{aligned} S, H \models P \multimap Q \\ \text{iff } \forall H'. (H \perp H') \wedge (S, H' \models P) \Rightarrow S, H \circ H' \models Q \end{aligned}$$

where  $H \perp H'$  means disjoint domains,  
and  $H \circ H'$  means disjoint function composition.



Now for more complicated semantics ;)

$$S, H \models E \mapsto E' \\ \text{iff } \text{dom}(H) = \{ \llbracket E \rrbracket_S \} \wedge H(\llbracket E \rrbracket_S) = \llbracket E' \rrbracket_S$$

$$S, H \models P * Q \\ \text{iff } \exists H_1 H_2. (H_1 \perp H_2) \wedge (H_1 \circ H_2 = H) \wedge (S, H_1 \models P) \wedge (S, H_2 \models Q)$$

$$S, H \models P \multimap Q \\ \text{iff } \forall H'. (H \perp H') \wedge (S, H' \models P) \Rightarrow S, H \circ H' \models Q$$

where  $H \perp H'$  means disjoint domains,  
and  $H \circ H'$  means disjoint function composition.

Now for more complicated semantics ;)

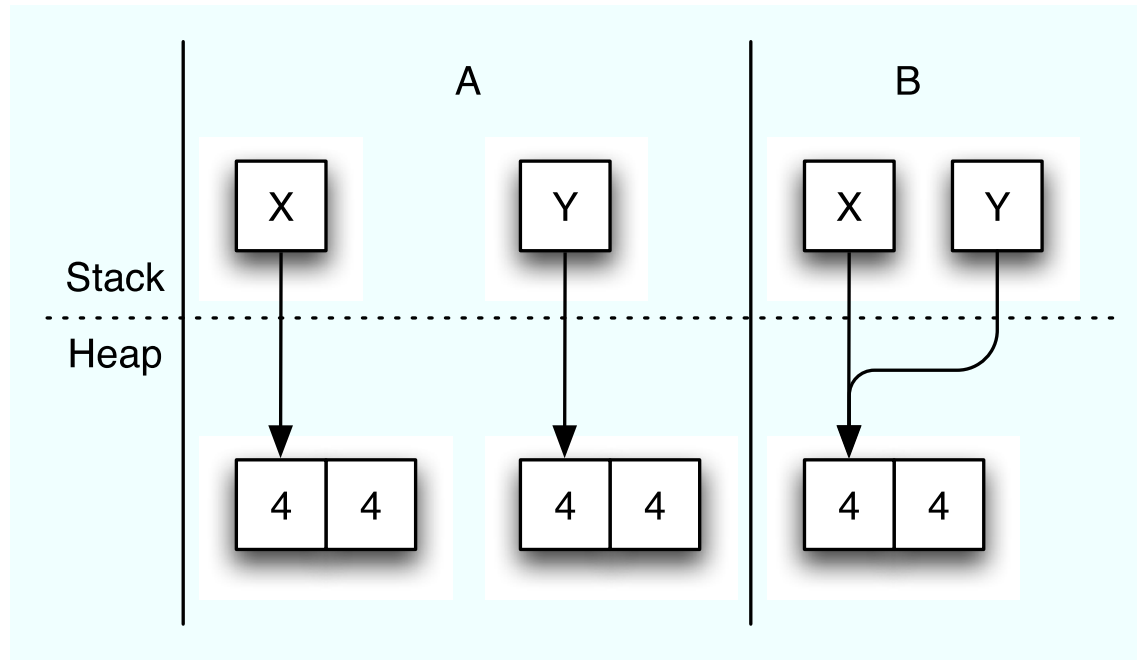
$$S, H \models E \mapsto E' \\ \text{iff } \text{dom}(H) = \{ \llbracket E \rrbracket_S \} \wedge H(\llbracket E \rrbracket_S) = \llbracket E' \rrbracket_S$$

$$S, H \models P * Q \\ \text{iff } \exists H_1 H_2. (H_1 \perp H_2) \wedge (H_1 \circ H_2 = H) \wedge (S, H_1 \models P) \wedge (S, H_2 \models Q)$$

$$S, H \models P \multimap Q \\ \text{iff } \forall H'. (H \perp H') \wedge (S, H' \models P) \Rightarrow S, H \circ H' \models Q$$

where  $H \perp H'$  means disjoint domains,  
and  $H \circ H'$  means disjoint function composition.

## + Example heaps



	A	B
$x \mapsto 4, 4$	✗	✓
$x \hookrightarrow 4, 4$	✓	✓
$x \mapsto 4, 4 * y \mapsto 4, 4$	✓	✗
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$	✗	✓
$x \hookrightarrow 4, 4 \wedge y \hookrightarrow 4, 4$	✓	✓

where  $E \mapsto E_0, \dots, E_n \stackrel{\text{def}}{=} E \mapsto E_0 * E + 1 \mapsto E_1 * \dots * E + n \mapsto E_n$

and  $E \hookrightarrow E' \stackrel{\text{def}}{=} E \mapsto E' * \text{true}$

Consider the following recursive formula

$$\begin{aligned}list [] i &\equiv empty \wedge i = nil \\list (\alpha :: \tau) i &\equiv \exists j. (i \mapsto \alpha, j) * (list \tau j)\end{aligned}$$

This formula defines a non-cyclic list.

Consider the following recursive formula

$$\begin{aligned} \text{list } [] \ i &\equiv \text{empty} \wedge i = \text{nil} \\ \text{list } (\alpha :: \tau) \ i &\equiv \exists j. (i \mapsto \alpha, j) * (\text{list } \tau \ j) \end{aligned}$$

This formula defines a non-cyclic list.

We can give the definition of a binary tree as

$$\begin{aligned} \text{tree } \epsilon \ i &\equiv \text{empty} \wedge i = \text{nil} \\ \text{tree } (\tau, a, \tau') \ i &\equiv \exists j, k. (i \mapsto j, a, k) * (\text{tree } \tau \ j) * (\text{tree } \tau' \ k) \end{aligned}$$

$$\{E \mapsto \_ \} \quad [E] = E' \quad \{E \mapsto E' \}$$

$$\{X = x \wedge E \mapsto Y \} \quad x = [E] \quad \{E[X/x] \mapsto Y \wedge Y = x \}$$

$$\{E \mapsto \_ \} \quad \text{dispose}(E) \quad \{\text{empty} \}$$

$$\{\text{empty} \} \quad x = \text{cons}(E_1, \dots, E_n) \quad \{x \mapsto E_1, \dots, E_n \}$$

We use  $E \mapsto \_$  as a shorthand for  $\exists x. E \mapsto x$ .

## + Frame Rule

The most important rule

$$\frac{\{P\} \quad C \quad \{Q\}}{\{P * R\} \quad C \quad \{Q * R\}}$$

where  $FV(R) \cap \text{modifies}(C) = \emptyset$

The most important rule

$$\frac{\{P\} \quad C \quad \{Q\}}{\{P * R\} \quad C \quad \{Q * R\}}$$

where  $FV(R) \cap \text{modifies}(C) = \emptyset$

The semantics of a triple,  $\models \{P\} \quad C \quad \{Q\}$ , is  $\forall S, H$  if  $(S, H \models P)$ , then  $(S, H, C)$  is safe and if  $(S, H, C) \Downarrow (S', H')$  then  $S', H' \models Q$



The most important rule

$$\frac{\{P\} \quad C \quad \{Q\}}{\{P * R\} \quad C \quad \{Q * R\}}$$

where  $FV(R) \cap \text{modifies}(C) = \emptyset$

The semantics of a triple,  $\models \{P\} \quad C \quad \{Q\}$ , is  $\forall S, H$  if  $(S, H \models P)$ , then  $(S, H, C)$  is **safe** and if  $(S, H, C) \Downarrow (S', H')$  then  $S', H' \models Q$

**Tight interpretation!**

## + Tree dispose

```
{tree _ p}  
proc dispTree(p)  
  newvar i,j  
  if p!=nil  
    i = [p];  
    j = [p+2];  
    dispTree(i);  
    dispTree(j);  
    dispose(p+2);  
    dispose(p+1);  
    dispose(p);  
  endif  
endproc  
{empty}
```

## + Tree dispose

```
{tree _ p}  
proc dispTree(p)  
  newvar i,j  
  if p!=nil  
    {tree _ p  $\wedge p \neq nil$ }  
    i = [p];  
    j = [p+2];  
    dispTree(i);  
    dispTree(j);  
    dispose(p+2);  
    dispose(p+1);  
    dispose(p);  
  {empty}  
  endif  
endproc  
{empty}
```

## + Tree dispose

$\{tree\_p \wedge p \neq nil\}$

i = [p];

j = [p+2];

dispTree(i);

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

## + Tree dispose

$\{ \textit{tree\_p} \wedge p \neq \textit{nil} \}$

i = [p];

j = [p+2];

dispTree(i);

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{ \textit{empty} \}$

## + Tree dispose

```
{tree _ p  ∧ p ≠ nil}  
{∃i, j.  p ↦ i, _, j * tree _ i * tree _ j}  
  i = [p];  
  j = [p+2];  
  dispTree(i);  
  dispTree(j);  
  dispose(p+2);  
  dispose(p+1);  
  dispose(p);  
{empty}
```

## + Tree dispose

$\{tree\_p \wedge p \neq nil\}$

$\{\exists i, j. p \mapsto i, \_, j * tree\_i * tree\_j\}$

$\{\exists i. p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\}$

$i = [p];$

$\boxed{\{X = x \wedge E \mapsto Y\} \quad x = [E] \quad \{E[X/x] \mapsto Y \wedge Y = x\}}$

$j = [p+2];$

$dispTree(i);$

$dispTree(j);$

$dispose(p+2);$

$dispose(p+1);$

$dispose(p);$

$\{empty\}$

## + Tree dispose

$\{tree\_p \wedge p \neq nil\}$

$\{\exists i, j. p \mapsto i, \_, j * tree\_i * tree\_j\}$

$\{\exists i. p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\}$

$\{\exists i. p \mapsto i\}$

$i = [p];$

$\{X = x \wedge E \mapsto Y\} \quad x = [E] \quad \{E[X/x] \mapsto Y \wedge Y = x\}$
--

$\{p \mapsto i\}$

$j = [p+2];$

$dispTree(i);$

$dispTree(j);$

$dispose(p+2);$

$dispose(p+1);$

$dispose(p);$

$\{empty\}$



## + Tree dispose

$$\begin{aligned} & \{tree\_p \wedge p \neq nil\} \\ & \{\exists i, j. p \mapsto i, \_, j * tree\_i * tree\_j\} \\ & \{\exists i. p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\} \\ & \quad \{\exists i. p \mapsto i\} \\ & \quad i = [p]; \\ & \quad \{p \mapsto i\} \\ & \{p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\} \\ & \quad j = [p+2]; \\ & \quad dispTree(i); \\ & \quad dispTree(j); \\ & \quad dispose(p+2); \\ & \quad dispose(p+1); \\ & \quad dispose(p); \\ & \{empty\} \end{aligned}$$

## + Tree dispose

$\{tree\_p \wedge p \neq nil\}$

$i = [p];$

$\{p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\}$

$j = [p+2];$

$dispTree(i);$

$dispTree(j);$

$dispose(p+2);$

$dispose(p+1);$

$dispose(p);$

$\{empty\}$

## + Tree dispose

$\{tree\_p \wedge p \neq nil\}$

$i = [p];$

$\{p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\}$

$j = [p+2];$

$\{p \mapsto i, \_, j * tree\_i * tree\_j\}$

$dispTree(i);$

$\{tree\_p\} \quad dispTree(p) \quad \{empty\}$

$dispTree(j);$

$dispose(p+2);$

$dispose(p+1);$

$dispose(p);$

$\{empty\}$

## + Tree dispose

$\{tree\_p \wedge p \neq nil\}$

$i = [p];$

$\{p \mapsto i * \exists j. p + 1 \mapsto \_, j * tree\_i * tree\_j\}$

$j = [p+2];$

$\{p \mapsto i, \_, j * tree\_i * tree\_j\}$

$\{tree\_i\}$

$dispTree(i);$

$\{empty\}$

$\{tree\_p\} \quad dispTree(p) \quad \{empty\}$
---

$dispTree(j);$

$dispose(p+2);$

$dispose(p+1);$

$dispose(p);$

$\{empty\}$

## + Tree dispose

```
{tree _ p ∧ p ≠ nil}
  i = [p];
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}
  j = [p+2];
{p ↦ i, _, j * tree _ i * tree _ j}
  {tree _ i}
  dispTree(i);
  {empty}
{p ↦ i, _, j * empty * tree _ j}
  dispTree(j);
  dispose(p+2);
  dispose(p+1);
  dispose(p);
{empty}
```

$\{tree\_p\} \text{ dispTree}(p) \{empty\}$
---

## + Tree dispose

```
{tree _ p ∧ p ≠ nil}  
  i = [p];  
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}  
  j = [p+2];  
{p ↦ i, _, j * tree _ i * tree _ j}  
  {tree _ i}  
  dispTree(i);  
  {empty}  
{p ↦ i, _, j * empty * tree _ j}  
{p ↦ i, _, j * tree _ j}  
  dispTree(j);  
  dispose(p+2);  
  dispose(p+1);  
  dispose(p);  
{empty}
```

## + Tree dispose

```
{tree _ p  $\wedge$   $p \neq nil$ }  
  i = [p];  
{ $p \mapsto i$  *  $\exists j. p + 1 \mapsto \_, j$  * tree _ i * tree _ j}  
  j = [p+2];  
{ $p \mapsto i, \_, j$  * tree _ i * tree _ j}  
  dispTree(i);  
{ $p \mapsto i, \_, j$  * tree _ j}  
  dispTree(j);  
  dispose(p+2);  
  dispose(p+1);  
  dispose(p);  
{empty}
```

## + Tree dispose

```
{tree _ p ∧ p ≠ nil}  
  i = [p];  
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}  
  j = [p+2];  
{p ↦ i, _, j * tree _ i * tree _ j}  
  dispTree(i);  
{p ↦ i, _, j * tree _ j}  
  dispTree(j);  
{p ↦ i, _, j}  
  dispose(p+2);  
  dispose(p+1);  
  dispose(p);  
{empty}
```

$\{E \mapsto \_ \} \text{dispose}(E) \{empty\}$
---



## + Tree dispose

```
{tree _ p ∧ p ≠ nil}  
  i = [p];  
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}  
  j = [p+2];  
{p ↦ i, _, j * tree _ i * tree _ j}  
  dispTree(i);  
{p ↦ i, _, j * tree _ j}  
  dispTree(j);  
{p ↦ i, _, j}  
  dispose(p+2);  
{p ↦ i, _}  
  dispose(p+1);  
  dispose(p);  
{empty}
```

$\{E \mapsto \_ \} \text{ dispose}(E) \{empty\}$
--

## + Tree dispose

```
{tree _ p ∧ p ≠ nil}  
  i = [p];  
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}  
  j = [p+2];  
{p ↦ i, _, j * tree _ i * tree _ j}  
  dispTree(i);  
{p ↦ i, _, j * tree _ j}  
  dispTree(j);  
{p ↦ i, _, j}  
  dispose(p+2);  
{p ↦ i, _}  
  dispose(p+1);  
{p ↦ i}  
  dispose(p);  
{empty}
```

$\{E \mapsto \_ \} \text{dispose}(E) \{empty\}$
---

## + Tree dispose

```
{tree _ p ∧ p ≠ nil}  
  i = [p];  
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}  
  j = [p+2];  
{p ↦ i, _, j * tree _ i * tree _ j}  
  dispTree(i);  
{p ↦ i, _, j * tree _ j}  
  dispTree(j);  
{p ↦ i, _, j}  
  dispose(p+2);  
{p ↦ i, _}  
  dispose(p+1);  
{p ↦ i}  
  dispose(p);  
{empty}
```



## + Tree dispose

```
{tree _ p ∧ p ≠ nil}  
  i = [p];  
{p ↦ i * ∃j. p + 1 ↦ _, j * tree _ i * tree _ j}  
  j = [p+2];  
{p ↦ i, _, j * tree _ i * tree _ j}  
  dispTree(i);  
{p ↦ i, _, j * tree _ j}  
  dispTree(j);  
{p ↦ i, _, j}  
  dispose(p+2);  
{p ↦ i, _}  
  dispose(p+1);  
{p ↦ i}  
  dispose(p);  
{empty}
```

Frame rule is key to the proof!



## + List append

$\{ (list\ \tau_1\ x) * (list\ \tau_2\ y) \}$

proc append(x,y)

  newvar h,c,n;

  if x=nil then return y;

  h= x;

  c= x;

  n= [c+1];

  while(n!=nil)

    c=n;

    n=[c+1];

  [c+1]=y;

  return h;

end proc

$\{ list\ (\tau_1 @ \tau_2)\ ret \}$

## + List append

$\{(list\ \tau_1\ x) * (list\ \tau_2\ y)\}$

proc append(x,y)

  newvar h,c,n;

  if x=nil then return y;

$\{((list\ \tau_1\ x) \wedge x \neq nil) * (list\ \tau_2\ y)\}$

  h= x;

  c= x;

  n= [c+1];

  while(n!=nil)

    c=n;

    n=[c+1];

  [c+1]=y;

$\{list\ (\tau_1 @ \tau_2)\ h\}$

  return h;

end proc

$\{list\ (\tau_1 @ \tau_2)\ ret\}$

## + List append

$\{((list\ \tau_1\ x) \wedge x! = nil) * (list\ \tau_2\ y)\}$

h = x;

c = x;

n = [c+1];

while(n != nil)

  c = n;

  n = [c+1];

  [c+1] = y;

$\{list\ (\tau_1 @ \tau_2)\ h\}$

## + List append

$\{((list (\alpha :: \tau'_1) x) \wedge x! = nil) * (list \tau_2 y)\}$

h= x;

c= x;

n= [c+1];

while(n!=nil)

  c=n;

  n=[c+1];

  [c+1]=y;

$\{list ((\alpha :: \tau'_1)@ \tau_2) h\}$



## + List append

$\{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\}$

h = x;

c = x;

$\{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\}$

n = [c+1];

while(n != nil)

  c = n;

  n = [c+1];

  [c+1] = y;

$\{list ((\alpha :: \tau'_1) @ \tau_2) h\}$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \{(\exists i. (c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c) * list \tau_2 y\} \\ & \quad n = [c+1]; \\ & \quad \text{while}(n \neq nil) \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \{(\exists i. (c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c) * list \tau_2 y\} \\ & \{(c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c\} * (list \tau_2 y) \\ & \quad n = [c+1]; \\ & \quad \text{while}(n \neq nil) \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \{(\exists i. (c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c) * list \tau_2 y\} \\ & \{(c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c\} * (list \tau_2 y) \\ & \quad \{c + 1 \mapsto i\} \\ & \quad n = [c+1]; \\ & \quad \{(c + 1 \mapsto i) \wedge i = n\} \\ \\ & \text{while}(n \neq nil) \\ & \quad c = n; \\ & \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \{(\exists i. (c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c) * list \tau_2 y\} \\ & \{(c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c\} * (list \tau_2 y) \\ & \quad \{c + 1 \mapsto i\} \\ & \quad n = [c+1]; \\ & \quad \{(c + 1 \mapsto i) \wedge i = n\} \\ & \{((c \mapsto \alpha, i) \wedge i = n) * (list \tau'_1 i) \wedge h = c\} * (list \tau_2 y) \\ & \quad \text{while}(n \neq nil) \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \{(\exists i. (c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c) * list \tau_2 y\} \\ & \{(c \mapsto \alpha, i) * (list \tau'_1 i) \wedge h = c\} * (list \tau_2 y) \} \\ & \quad \{c + 1 \mapsto i\} \\ & \quad n = [c+1]; \\ & \quad \{(c + 1 \mapsto i) \wedge i = n\} \\ & \{((c \mapsto \alpha, i) \wedge i = n) * (list \tau'_1 i) \wedge h = c\} * (list \tau_2 y) \} \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y) \} \\ & \quad \text{while}(n \neq nil) \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \quad \text{while}(n \neq nil) \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \quad \text{while}(n \neq nil) \quad \left\{ \exists \tau', \alpha'. \left( \begin{aligned} & (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (list \tau' n) \end{aligned} \right) \right\} \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$



## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \left\{ \left( \begin{array}{l} (P \multimap P) \wedge h = c \\ * (c \mapsto \alpha, n) * (list \tau'_1 n) \end{array} \right) * (list \tau_2 y) \right\} \\ & \quad \text{while}(n \neq nil) \left\{ \exists \tau', \alpha'. \left( \begin{array}{l} (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \\ * (c \mapsto \alpha', n) * (list \tau' n) \end{array} \right) \right\} \\ & \quad c = n; \\ & \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \left\{ \left( \begin{array}{l} (list (\alpha :: \tau'_1 @ \tau_2) h) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) \wedge h = c \\ * (c \mapsto \alpha, n) * (list \tau'_1 n) \end{array} \right) * (list \tau_2 y) \right\} \\ & \text{while}(n \neq nil) \left\{ \exists \tau', \alpha'. \left( \begin{array}{l} (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \\ * (c \mapsto \alpha', n) * (list \tau' n) \end{array} \right) \right\} \\ & \quad c = n; \\ & \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \left\{ \left( \begin{array}{l} (list (\alpha :: \tau'_1 @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \\ * (c \mapsto \alpha, n) * (list \tau'_1 n) \end{array} \right) * (list \tau_2 y) \right\} \\ & \text{while}(n \neq nil) \left\{ \exists \tau', \alpha'. \left( \begin{array}{l} (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \\ * (c \mapsto \alpha', n) * (list \tau' n) \end{array} \right) \right\} \\ & \quad c = n; \\ & \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \quad \text{while}(n \neq nil) \quad \left\{ \exists \tau', \alpha'. \left( \begin{aligned} & (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (list \tau' n) \end{aligned} \right) \right\} \\ & \quad \quad c = n; \\ & \quad \quad n = [c+1]; \\ & \quad [c+1] = y; \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\}$$

h = x;

c = x;

$$\{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\}$$

n = [c+1];

$$\{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\}$$

while(n != nil)  $\left\{ \exists \tau', \alpha'. \left( (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \right) * (c \mapsto \alpha', n) * (list \tau' n) \right\}$

c = n;

n = [c+1];

$$\left\{ \left( \exists \alpha'. ((list (\alpha' :: \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) * (c \mapsto \alpha', nil) \right) * (list \tau_2 y) \right\}$$

[c+1] = y;

$$\{list ((\alpha :: \tau'_1) @ \tau_2) h\}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \quad \text{while}(n \neq nil) \left\{ \exists \tau', \alpha'. \left( (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \right) \right. \\ & \quad \quad \left. * (c \mapsto \alpha', n) * (list \tau' n) \right\} \\ & \quad c = n; \\ & \quad n = [c+1]; \\ & \left\{ \left( \exists \alpha'. ((list (\alpha' :: \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) * (c \mapsto \alpha', nil) \right) * (list \tau_2 y) \right\} \\ & \quad [c+1] = y; \\ & \left\{ \left( \exists \alpha'. ((list (\alpha' :: \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) * (c \mapsto \alpha', y) \right) * (list \tau_2 y) \right\} \\ & \{list ((\alpha :: \tau'_1) @ \tau_2) h\} \end{aligned}$$

## + List append

$$\begin{aligned} & \{((list (\alpha :: \tau'_1) x) \wedge x \neq nil) * (list \tau_2 y)\} \\ & \quad h = x; \\ & \quad c = x; \\ & \{((list (\alpha :: \tau'_1) c) \wedge c \neq nil \wedge h = c) * (list \tau_2 y)\} \\ & \quad n = [c+1]; \\ & \{((c \mapsto \alpha, n) * (list \tau'_1 n) \wedge h = c) * (list \tau_2 y)\} \\ & \quad \text{while}(n \neq nil) \quad \left\{ \exists \tau', \alpha'. \left( (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \right) \right. \\ & \quad \quad \left. * (c \mapsto \alpha', n) * (list \tau' n) \right\} \\ & \quad c = n; \\ & \quad n = [c+1]; \\ & \left\{ \left( \exists \alpha'. ((list (\alpha' :: \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) * (c \mapsto \alpha', nil) \right) * (list \tau_2 y) \right\} \\ & \quad [c+1] = y; \\ & \left\{ \left( \exists \alpha'. ((list (\alpha' :: \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) * (c \mapsto \alpha', y) \right) * (list \tau_2 y) \right\} \\ & \{ \exists \alpha'. ((list (\alpha' :: \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) * (list (\alpha' :: \tau_2) c) \} \\ & \{ list ((\alpha :: \tau'_1) @ \tau_2) h \} \end{aligned}$$

$$\begin{array}{l} \text{while}(n \neq \text{nil}) \\ \quad c = n; \\ \quad n = [c+1]; \end{array} \quad \left\{ \exists \tau', \alpha'. \left( \begin{array}{l} (\text{list } (\alpha' :: \tau' @ \tau_2) \ c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) \ h) \\ * (c \mapsto \alpha', n) * (\text{list } \tau' \ n) \end{array} \right) \right\}$$



$$\left\{ n \neq nil \wedge \exists \tau', \alpha'. \left( (list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h) \right) \right. \\ \left. * (c \mapsto \alpha', n) * (list \tau' n) \right\}$$

c=n;

n=[c+1];

$$\left\{ \exists \tau', \alpha'. \left( ((list (\alpha' :: \tau' @ \tau_2) c) \multimap (list (\alpha :: \tau'_1 @ \tau_2) h)) \right) \right. \\ \left. * (c \mapsto \alpha', n) * (list \tau' n) \right\}$$

## + List append

$$\left\{ \begin{array}{l} n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{array}{l} (\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{array} \right) \\ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{array}{l} ((\exists i. (c \mapsto \alpha', i) * \text{list } (\tau' @ \tau_2) i)) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{array} \right) \end{array} \right\}$$

c=n;  
n=[c+1];

$$\left\{ \exists \tau', \alpha'. \left( \begin{array}{l} ((\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) \\ * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{array} \right) \right\}$$

$$\begin{aligned}
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & (\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & ((\exists i. (c \mapsto \alpha', i) * \text{list } (\tau' @ \tau_2) i)) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau'. \left( ((\text{list } (\tau' @ \tau_2) n) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) * (\text{list } \tau' n) \right) \right\} \\
& \quad \text{c=n;} \\
& \quad \text{n=[c+1];} \\
& \left\{ \exists \tau', \alpha'. \left( \begin{aligned} & ((\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\}
\end{aligned}$$

$$\begin{aligned}
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & (\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & ((\exists i. (c \mapsto \alpha', i) * \text{list } (\tau' @ \tau_2) i)) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau'. \left( ((\text{list } (\tau' @ \tau_2) n) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) * (\text{list } \tau' n) \right) \right\} \\
& \quad \mathbf{c} = \mathbf{n}; \\
& \left\{ c \neq \text{nil} \wedge \exists \tau'. \left( \begin{aligned} & (\text{list } (\tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * \text{list } \tau' c \end{aligned} \right) \right\} \\
& \quad \mathbf{n} = [\mathbf{c} + 1]; \\
& \left\{ \exists \tau', \alpha'. \left( \begin{aligned} & ((\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\}
\end{aligned}$$

$$\begin{aligned}
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & (\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & ((\exists i. (c \mapsto \alpha', i) * \text{list } (\tau' @ \tau_2) i)) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau'. \left( ((\text{list } (\tau' @ \tau_2) n) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) * (\text{list } \tau' n) \right) \right\} \\
& \quad \mathbf{c=n;} \\
& \left\{ \exists \tau'. \left( \begin{aligned} & (\text{list } (\tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * \exists \tau'', \alpha', j. (c \mapsto \alpha', j) * (\text{list } \tau'' j) \wedge (\alpha' :: \tau'' = \tau') \end{aligned} \right) \right\} \\
& \quad \mathbf{n=[c+1];} \\
& \left\{ \exists \tau', \alpha'. \left( \begin{aligned} & ((\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\}
\end{aligned}$$

$$\begin{aligned}
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & (\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau', \alpha'. \left( \begin{aligned} & ((\exists i. (c \mapsto \alpha', i) * \text{list } (\tau' @ \tau_2) i)) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\} \\
& \left\{ n \neq \text{nil} \wedge \exists \tau'. \left( ((\text{list } (\tau' @ \tau_2) n) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) * (\text{list } \tau' n) \right) \right\} \\
& \quad \mathbf{c=n;} \\
& \left\{ \exists \tau'. \left( \begin{aligned} & (\text{list } (\tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h) \\ & * \exists \tau'', \alpha', j. (c \mapsto \alpha', j) * (\text{list } \tau'' j) \wedge (\alpha' :: \tau'' = \tau') \end{aligned} \right) \right\} \\
& \quad \mathbf{n=[c+1];} \\
& \left\{ \exists \tau', \alpha'. \left( \begin{aligned} & ((\text{list } (\alpha' :: \tau' @ \tau_2) c) \multimap (\text{list } (\alpha :: \tau'_1 @ \tau_2) h)) \\ & * (c \mapsto \alpha', n) * (\text{list } \tau' n) \end{aligned} \right) \right\}
\end{aligned}$$



## + Conclusions +

- Tight specifications
- Dangling pointers
- Local surgeries
- Frame rule

## + References +

---

The references for this part of the course can all be found on:

<http://www.dcs.qmw.ac.uk/~ohearn/localreasoning.html>