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

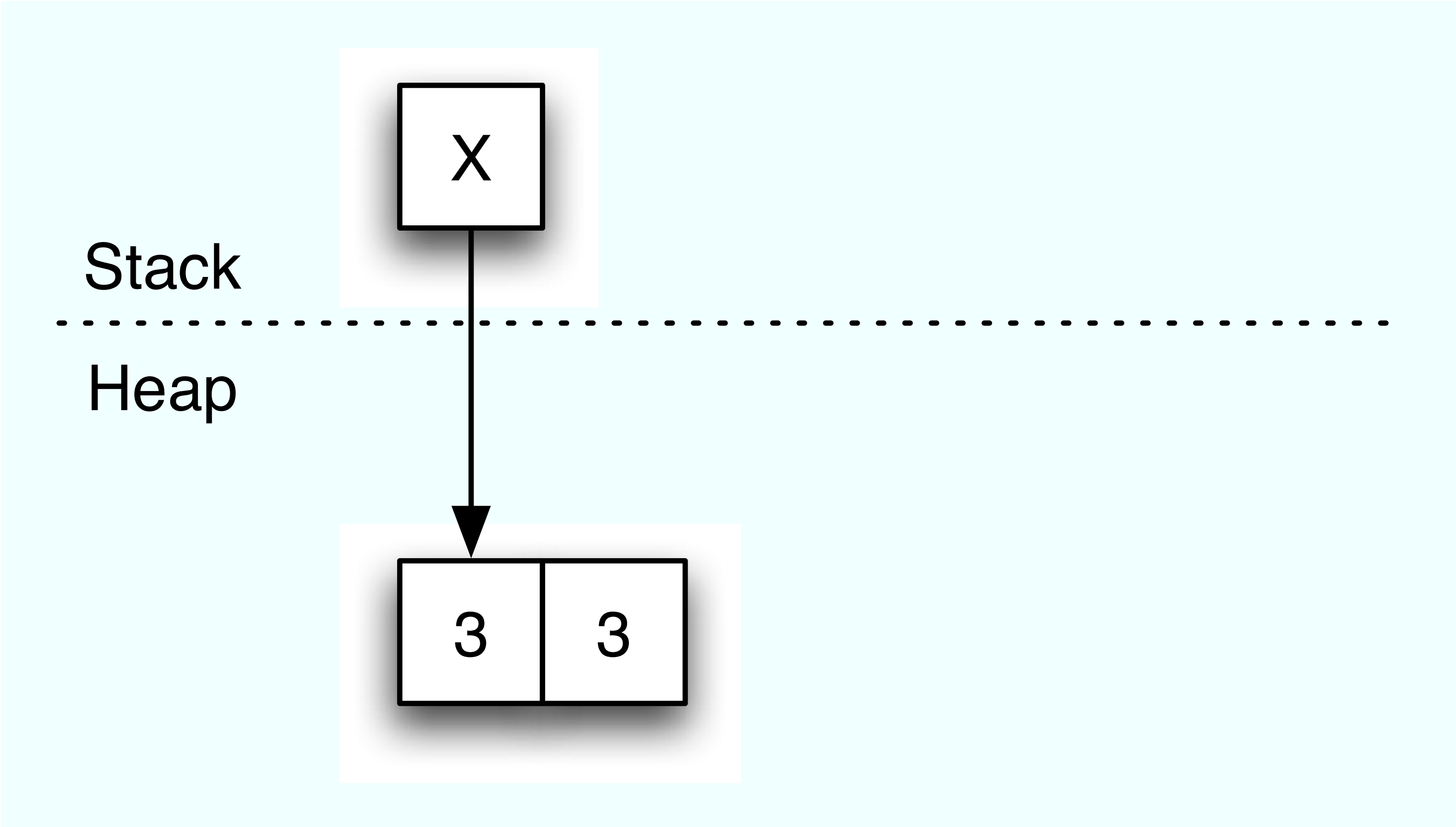
- Modularity
- Concurrency
- Decidability

Motivation

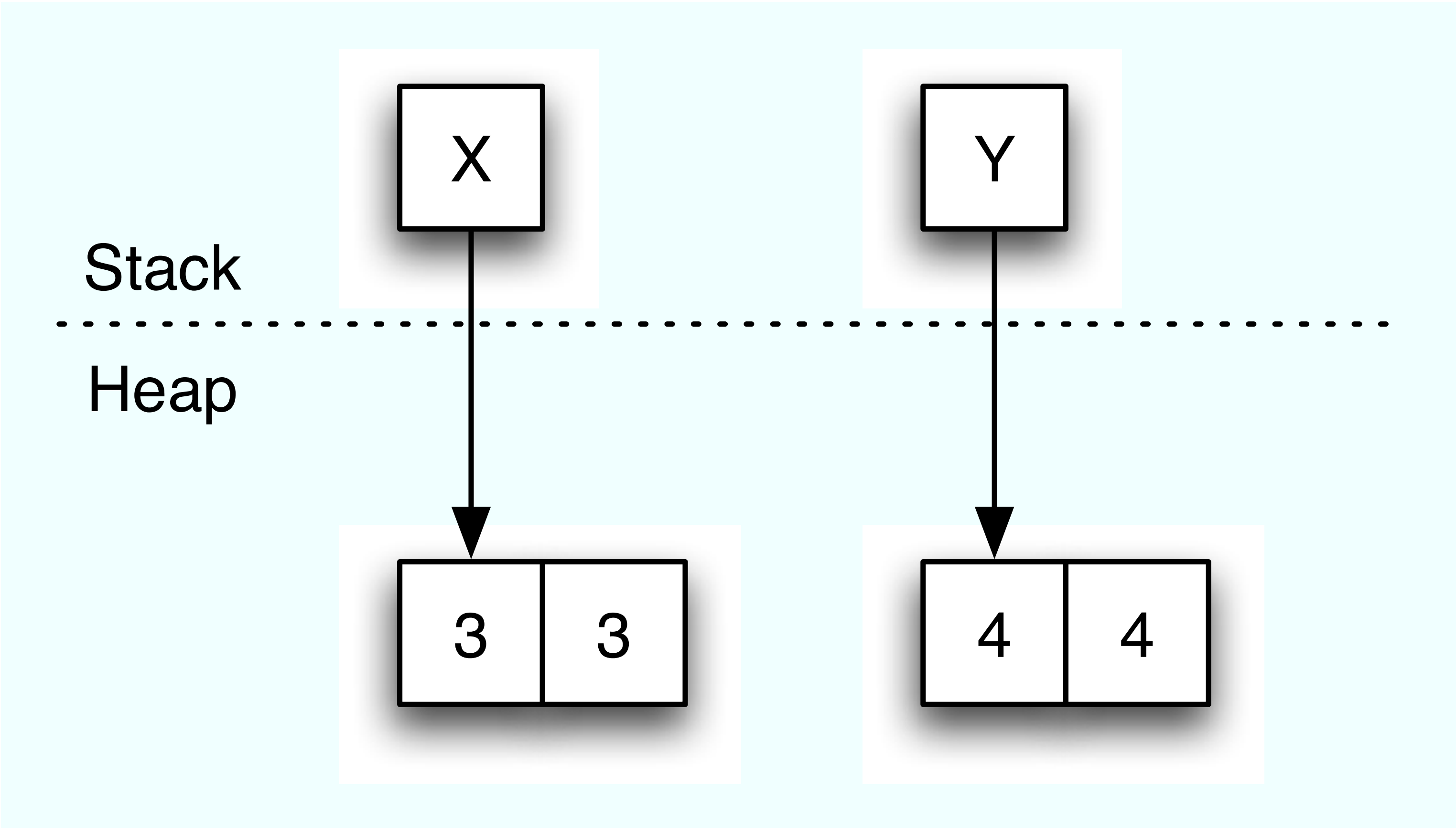
Example program

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

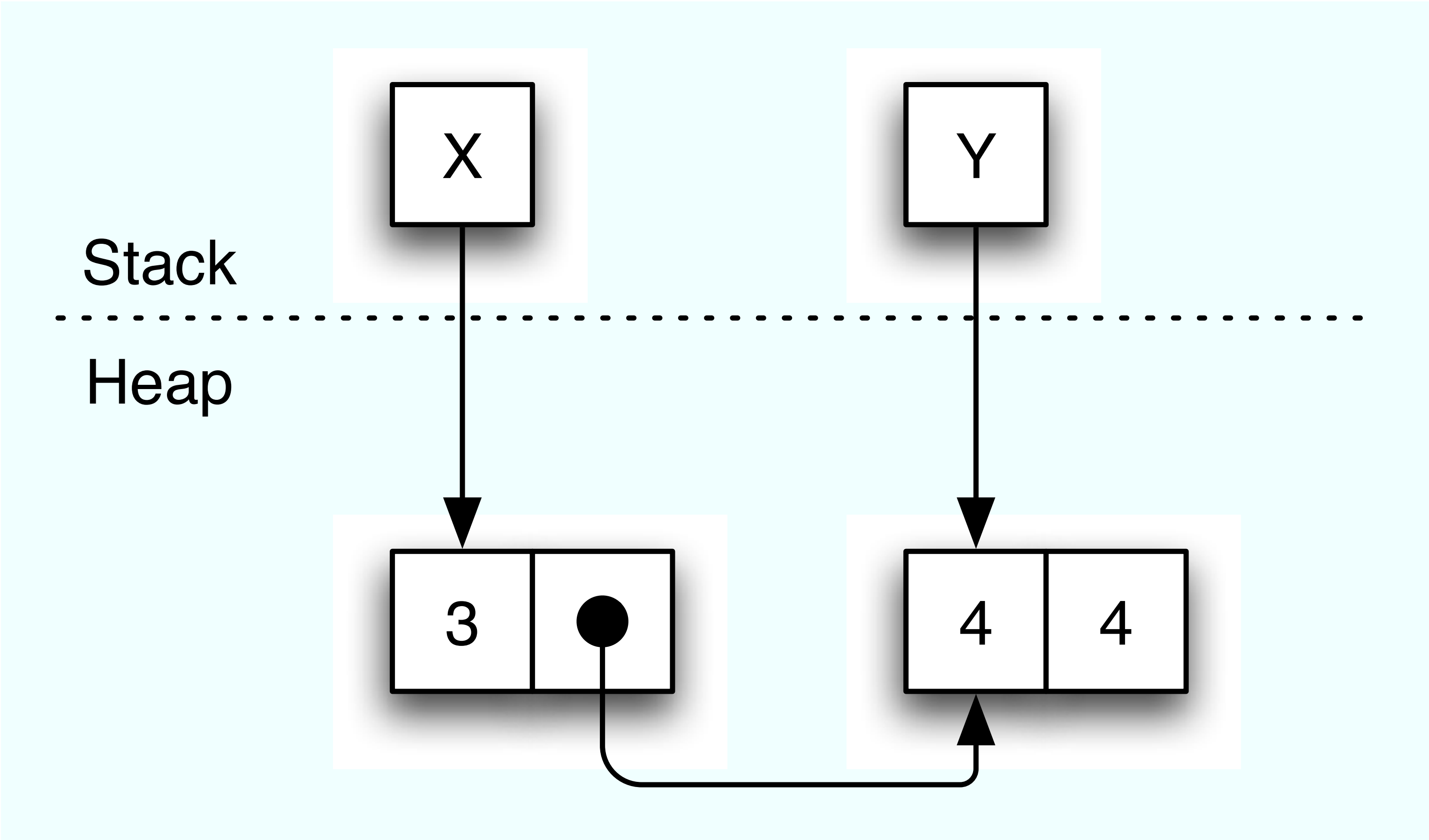
```
x = cons(3,3);
y = cons(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



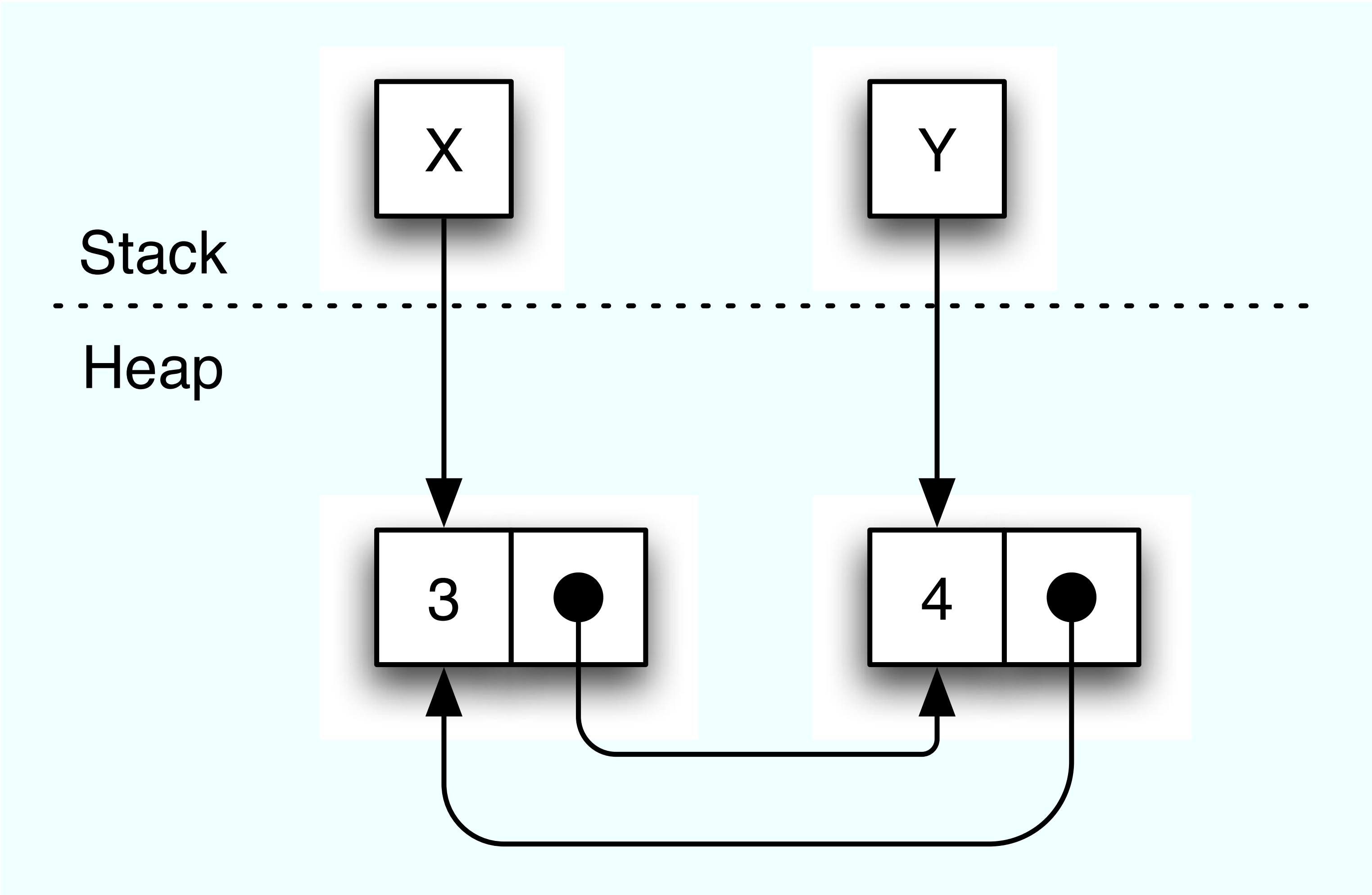
```
x = cons(3,3);
y = cons(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



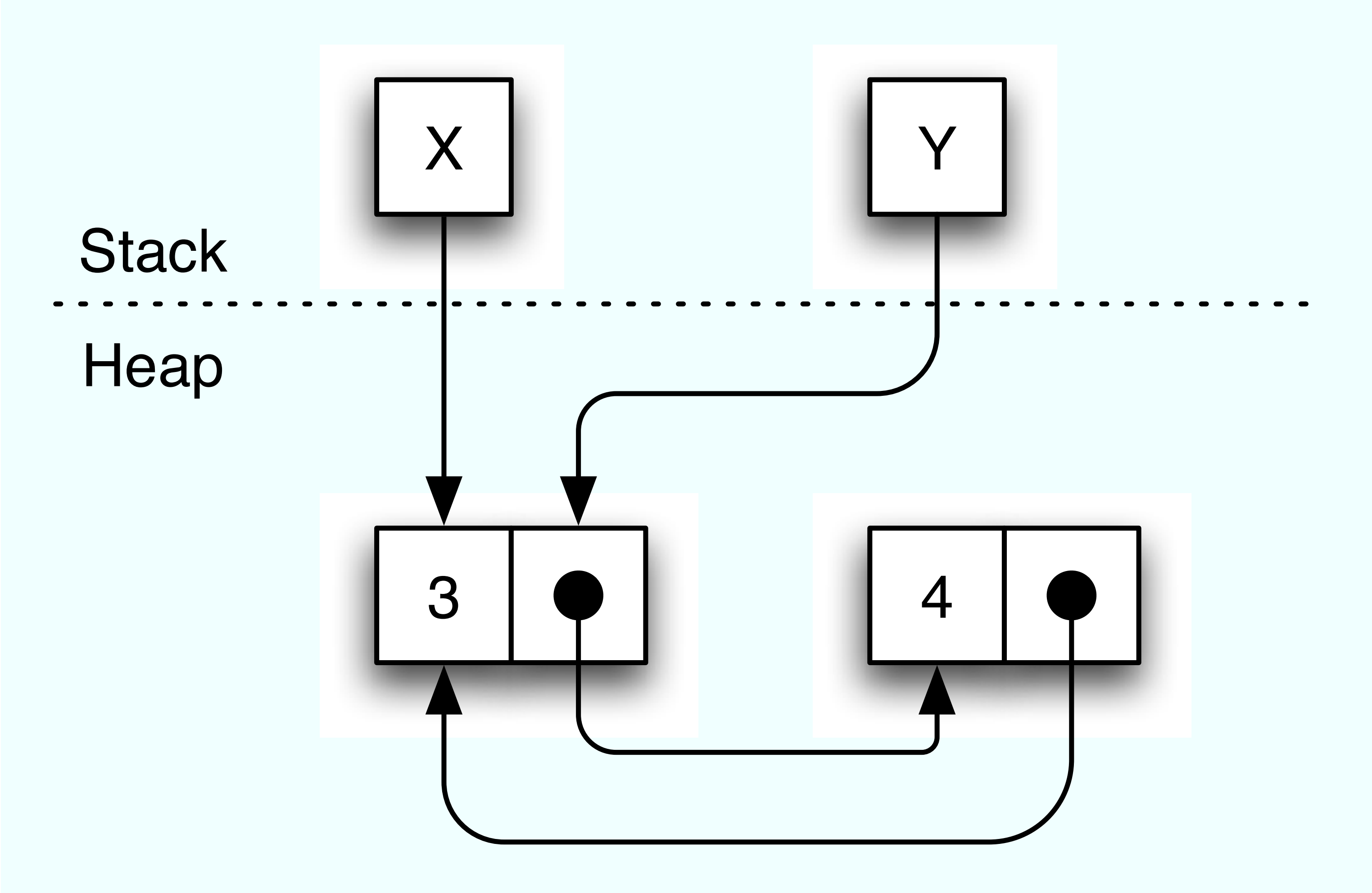
```
x = cons(3,3);
y = cons(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



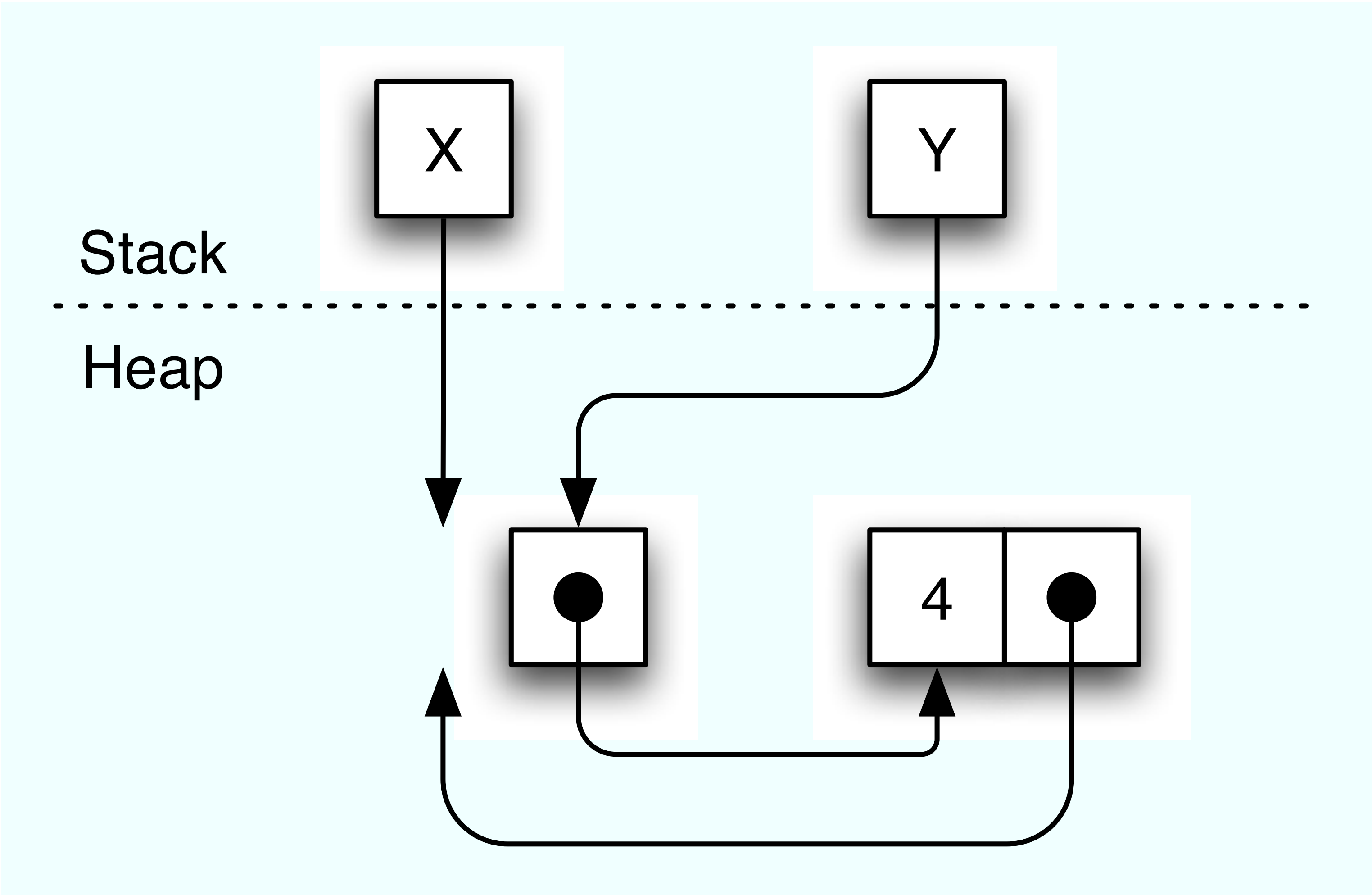
x = cons(3,3);
y = cons(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];



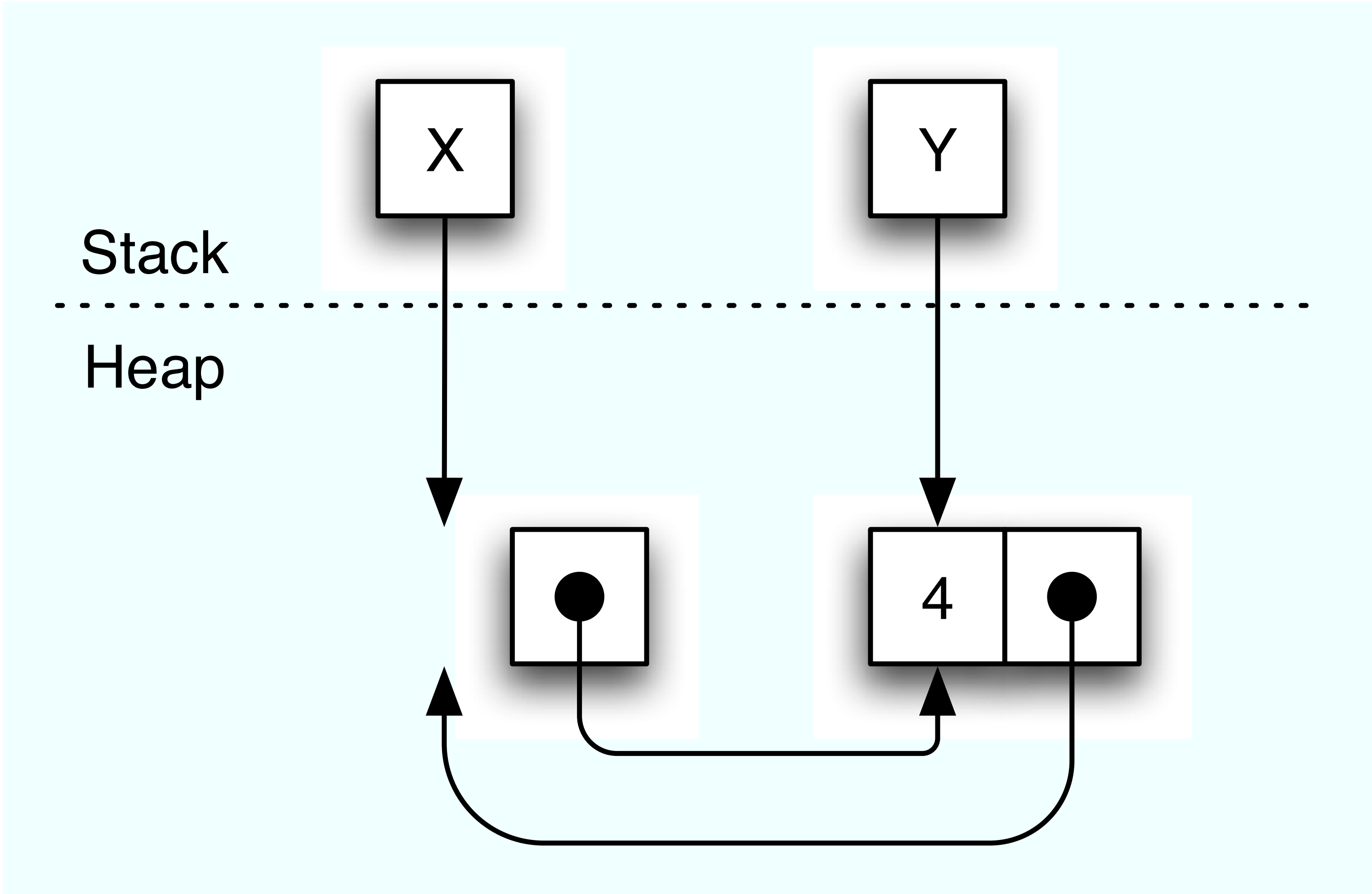
x = cons(3,3);
y = cons(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];



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



```
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] != [z])
```

Need to know things locations are different.

Consider the following piece of code
[Note: read [x] as indirect through x to the heap.]

```
Assume(y != z)
[y] = 4;
[z] = 5;
Guarantee([y] != [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 != z)
[y] = 4;
[z] = 5;
Guarantee([y] != [z])
Guarantee([x] = 3)
```

Need to know things locations are different.

- Add assertions?

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

Consider the following piece of code
[Note: read [x] as indirect through x to the heap.]

```
Assume([x] = 3 ∧ x != y ∧ x != z )
Assume(y != z)
[y] = 4;
[z] = 5;
Guarantee([y] != [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.

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

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.

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

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...

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge r \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \; weakl$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee l \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A} \; weakr$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow l \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow r \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \; contrl$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg l \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg r \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} \; contrr$$

$$\frac{}{A \vdash A} \; BS$$

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

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

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

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

$$\frac{}{A \vdash A}\ BS$$

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, . . .)

$\wedge$ 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

$$
\begin{array}{lcll}
\text{P,Q} & ::= & false & \text{Logical false} \\
& | & P \wedge Q & \text{Classical conjunction} \\
& | & P \vee Q & \text{Classical disjunction} \\
& | & P \Rightarrow Q & \text{Classical implication} \\
& | & P * Q & \text{Separating conjunction} \\
& | & P \mathrel{-\!\!*} Q & \text{Separating implication} \\
& | & E = E & \text{Expression value equality} \\
& | & E \mapsto E & \text{points to} \\
& | & empty & \text{empty heap} \\
& | & \exists x.P & \text{existential quantifier}
\end{array}
$$

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 : Var $\rightharpoonup$ Int   H : Loc $\rightharpoonup$ Int   where Loc $\subseteq$ Int

$S, H \models false$	never satisfied
$S, H \models P \wedge Q$	iff $S, H \models P \quad \wedge \quad S, H \models Q$
$S, H \models P \vee Q$	iff $S, H \models P \quad \vee \quad S, H \models Q$
$S, H \models P \Rightarrow Q$	iff $S, H \models P \quad \Rightarrow \quad S, H \models Q$
$S, H \models E = E'$	iff $[\![E]\!]_S = [\![E']\!]_S$
$S, H \models empty$	iff $H = \{\}$

We use $[\![E]\!]_S$ to mean evaluation with respect to the stack, S.

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

S : Var $\rightharpoonup$ Int H : Loc $\rightharpoonup$ Int where Loc $\subseteq$ Int

$S, H \models false$ never satisfied

$S, H \models P \wedge Q$ iff $S, H \models P \quad \wedge \quad S, H \models Q$

$S, H \models P \vee Q$ iff $S, H \models P \quad \vee \quad S, H \models Q$

$S, H \models P \Rightarrow Q$ iff $S, H \models P \quad \Rightarrow \quad S, H \models Q$

$S, H \models E = E'$ iff $[\![E]\!]_S = [\![E']\!]_S$

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

We use $[\![E]\!]_S$ to mean evaluation with respect to the stack, S.

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

S : Var $\rightharpoonup$ Int H : Loc $\rightharpoonup$ Int where Loc $\subseteq$ Int

$S, H \models false$	never satisfied
$S, H \models P \wedge Q$	iff $S, H \models P \quad \wedge \quad S, H \models Q$
$S, H \models P \vee Q$	iff $S, H \models P \quad \vee \quad S, H \models Q$
$S, H \models P \Rightarrow Q$	iff $S, H \models P \quad \Rightarrow \quad S, H \models Q$
$S, H \models E = E'$	iff $[\![E]\!]_S = [\![E']\!]_S$
$S, H \models empty$	iff $H = \{\}$

We use $[\![E]\!]_S$ to mean evaluation with respect to the stack, S.

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

S : Var $\rightharpoonup$ Int H : Loc $\rightharpoonup$ Int where Loc $\subseteq$ Int

$S, H \models false$	never satisfied
$S, H \models P \wedge Q$	iff $S, H \models P \quad \wedge \quad S, H \models Q$
$S, H \models P \vee Q$	iff $S, H \models P \quad \vee \quad S, H \models Q$
$S, H \models P \Rightarrow Q$	iff $S, H \models P \quad \Rightarrow \quad S, H \models Q$
$S, H \models E = E'$	iff $[\![E]\!]_S = [\![E']\!]_S$
$S, H \models empty$	iff $H = \{\}$

We use $[\![E]\!]_S$ to mean evaluation with respect to the stack, S.

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

S : Var $\rightharpoonup$ Int   H : Loc $\rightharpoonup$ Int   where Loc $\subseteq$ Int

$S, H \models false$	never satisfied
$S, H \models P \wedge Q$	iff $S, H \models P \quad \wedge \quad S, H \models Q$
$S, H \models P \vee Q$	iff $S, H \models P \quad \vee \quad S, H \models Q$
$S, H \models P \Rightarrow Q$	iff $S, H \models P \quad \Rightarrow \quad S, H \models Q$
$S, H \models E = E'$	iff $[\![E]\!]_S = [\![E']\!]_S$
$S, H \models empty$	iff $H = \{\}$

We use $[\![E]\!]_S$ to mean evaluation with respect to the stack, S.

Semantics 1/2

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

S : Var $\rightharpoonup$ Int   H : Loc $\rightharpoonup$ Int   where Loc $\subseteq$ Int

$S, H \models false$	never satisfied
$S, H \models P \wedge Q$	iff $S, H \models P \quad \wedge \quad S, H \models Q$
$S, H \models P \vee Q$	iff $S, H \models P \quad \vee \quad S, H \models Q$
$S, H \models P \Rightarrow Q$	iff $S, H \models P \quad \Rightarrow \quad S, H \models Q$
$S, H \models E = E'$	iff $[\![E]\!]_S = [\![E']\!]_S$
$S, H \models empty$	iff $H = \{\}$

We use $[\![E]\!]_S$ to mean evaluation with respect to the stack, S.

Now for more complicated semantics ;)

$S, H \models E \mapsto E'$
iff $dom(H) = \{[\![E]\!]_S\} \wedge H([\![E]\!]_S) = [\![E']\!]_S$

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

$S, H \models P \mathrel{-\!\!*} Q$
iff $\forall H' . (H \perp H') \wedge (S, H' \models P) \quad \Rightarrow \quad S, H \circ H' \models Q$

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

Now for more complicated semantics ;)

$S, H \models E \mapsto E'$
iff $dom(H) = \{[\![E]\!]_S\} \wedge H([\![E]\!]_S) = [\![E']\!]_S$

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

$S, H \models P \mathrel{-\!\!*} Q$
iff $\forall H'.(H \perp H') \wedge (S, H' \models P) \quad \Rightarrow \quad S, H \circ H' \models Q$

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

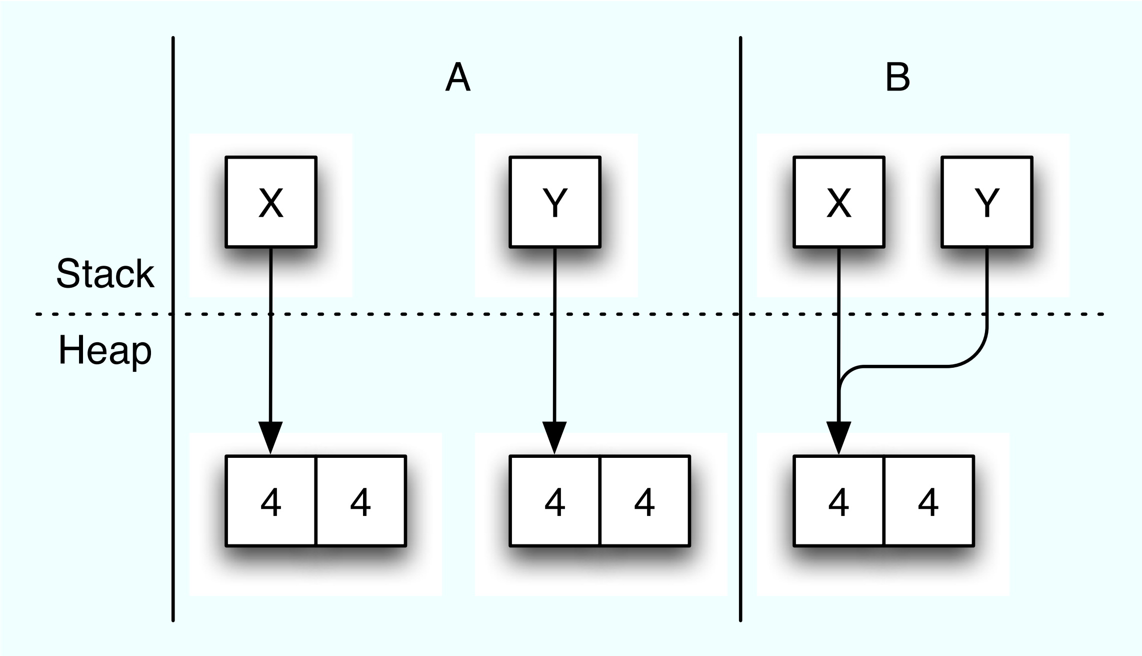
Now for more complicated semantics ;)

$S, H \models E \mapsto E'$
iff $dom(H) = \{[\![E]\!]_S\} \wedge H([\![E]\!]_S) = [\![E']\!]_S$

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

$S, H \models P \mathrel{-\!\!*} Q$
iff $\forall H' . (H \perp H') \wedge (S, H' \models P) \quad \Rightarrow \quad S, H \circ H' \models Q$

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



	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 $E \mapsto E_0, \ldots, E_n \stackrel{\textbf{def}}{=} E \mapsto E_0 \ * \ E+1 \mapsto E_1 \ * \ \ldots E+n \mapsto E_n$

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

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

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

This formula defines a non-cyclic list.

We can give the definition of a binary tree as

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

Small Axioms

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

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

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

$$\{empty\} \ \ x = cons(E_1, \ldots, E_n) \ \ \{x \mapsto E_1, \ldots, E_n)\}$$

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

The most important rule

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

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

Frame Rule

The most important rule

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

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

The semantics of a triple, $\models \{P\}\ \ C\ \ \{Q\}$, 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 $FV(R) \cap \mathit{modifies}(C) = \emptyset$

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
```

$\{tree\_\text{p} \wedge \text{p} \neq nil\}$

```
      i = [p];
      j = [p+2];
      dispTree(i);
      dispTree(j);
      dispose(p+2);
      dispose(p+1);
      dispose(p);
```

$\{empty\}$

```
   endif
endproc
```

$\{empty\}$

$\{tree\ \_\ \mathsf{p} \quad \wedge\ \mathsf{p} \neq nil\}$

```
    i = [p];
    j = [p+2];
    dispTree(i);
    dispTree(j);
    dispose(p+2);
    dispose(p+1);
    dispose(p);
```

$\{empty\}$

Tree dispose

$\{tree\ \_\ \mathsf{p} \quad \wedge\ \mathsf{p} \neq nil\}$

```
    i = [p];
    j = [p+2];
    dispTree(i);
    dispTree(j);
    dispose(p+2);
    dispose(p+1);
    dispose(p);
```

$\{empty\}$

Tree dispose

$\{tree\_\mathsf{p} \ \wedge \mathsf{p} \neq nil\}$

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

```
i = [p];
j = [p+2];
dispTree(i);
dispTree(j);
dispose(p+2);
dispose(p+1);
dispose(p);
```

$\{empty\}$

Tree dispose

$\{tree\_\mathsf{p} \wedge \mathsf{p} \neq nil\}$

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

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

```
i = [p];
j = [p+2];
dispTree(i);
dispTree(j);
dispose(p+2);
dispose(p+1);
dispose(p);
```

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

$\{empty\}$

$\{tree\_\mathsf{p} \;\; \wedge \mathsf{p} \neq nil\}$

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

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

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

i = [p];

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

$\{\mathsf{p} \mapsto \mathtt{i}\}$

j = [p+2];

dispTree(i);

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

$\{tree\_\mathtt{p} \;\wedge \mathtt{p} \neq nil\}$

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

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

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

i = [p];

$\{\mathtt{p} \mapsto \mathtt{i}\}$

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

j = [p+2];

dispTree(i);

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

Tree dispose

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

dispTree(i);

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

Tree dispose

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

dispTree(i);

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

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

$\{tree\_\mathtt{i}\}$

dispTree(i);

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

$\{empty\}$

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

Tree dispose

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

$\{tree\_\mathtt{i}\}$

dispTree(i);

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

$\{empty\}$

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j} \ * \ empty \ * \ tree\_\mathtt{j}\}$

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

$\{tree\_\mathtt{i}\}$

dispTree(i);

$\{empty\}$

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j} \ * \ empty \ * \ tree\_\mathtt{j}\}$

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

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

dispTree(i);

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

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

Tree dispose

$\{tree\_\mathtt{p} \;\wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j} \;*\; tree\_\mathtt{i} \;*\; tree\_\mathtt{j}\}$

dispTree(i);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j} \;*\; tree\_\mathtt{j}\}$

dispTree(j);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j}\}$

dispose(p+2);

$\boxed{\{E \mapsto \_\} \;\; dispose(E) \;\; \{empty\}}$

dispose(p+1);

dispose(p);

$\{empty\}$

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

dispTree(i);

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

dispTree(j);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j}\}$

dispose(p+2);

$\{\mathtt{p} \mapsto \mathtt{i}, \_\}$

dispose(p+1);

$\boxed{\{E \mapsto \_\} \ \ dispose(E) \ \ \{empty\}}$

dispose(p);

$\{empty\}$

Tree dispose

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

dispTree(i);

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

dispTree(j);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j}\}$

dispose(p+2);

$\{\mathtt{p} \mapsto \mathtt{i}, \_\}$

dispose(p+1);

$\{\mathtt{p} \mapsto \mathtt{i}\}$

dispose(p);

$\boxed{\{E \mapsto \_\} \ \ dispose(E) \ \ \{empty\}}$

$\{empty\}$

Tree dispose

$\{tree\_\mathtt{p} \ \wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

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

dispTree(i);

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

dispTree(j);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j}\}$

dispose(p+2);

$\{\mathtt{p} \mapsto \mathtt{i}, \_\}$

dispose(p+1);

$\{\mathtt{p} \mapsto \mathtt{i}\}$

dispose(p);

$\{empty\}$

✓

Tree dispose

$\{tree\_\mathtt{p} \;\wedge \mathtt{p} \neq nil\}$

i = [p];

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

j = [p+2];

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j} \; * \; tree\_\mathtt{i} \; * \; tree\_\mathtt{j}\}$

dispTree(i);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j} \; * \; tree\_\mathtt{j}\}$

dispTree(j);

$\{\mathtt{p} \mapsto \mathtt{i}, \_, \mathtt{j}\}$

dispose(p+2);

$\{\mathtt{p} \mapsto \mathtt{i}, \_\}$

dispose(p+1);

$\{\mathtt{p} \mapsto \mathtt{i}\}$

dispose(p);

$\{empty\}$

Frame rule is key to the proof! $\checkmark$

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

```
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;
```

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

```
   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

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

```
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;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

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

```
h= x;
c= x;
```

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$

$\{(\exists i.\ \ (c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * list\ \tau_2\ y\}$

```
n= [c+1];
while(n!=nil)
   c=n;
   n=[c+1];
[c+1]=y;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

List append

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

h= x;

c= x;

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$
$\{(\exists i.\ \ (c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * list\ \tau_2\ y\}$
$\{(c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * (list\ \tau_2\ y)\}$

```
n= [c+1];
while(n!=nil)
   c=n;
   n=[c+1];
[c+1]=y;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

List append

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

h= x;

c= x;

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$

$\{(\exists i.\ \ (c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * list\ \tau_2\ y\}$

$\{(c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * (list\ \tau_2\ y)\}$

$\{c + 1 \mapsto i\}$

n= [c+1];

$\{(c + 1 \mapsto i) \wedge i = n\}$

```
while(n!=nil)
   c=n;
   n=[c+1];
[c+1]=y;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

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

h= x;

c= x;

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$

$\{(\exists i.\ \ (c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * list\ \tau_2\ y\}$

$\{(c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * (list\ \tau_2\ y)\}$

$\{c + 1 \mapsto i\}$

n= [c+1];

$\{(c + 1 \mapsto i) \wedge i = n\}$

$\{((c \mapsto \alpha, i) \wedge i = n) * (list\ \tau_1'\ i) \wedge h = c) * (list\ \tau_2\ y)\}$

```
while(n!=nil)
    c=n;
    n=[c+1];
[c+1]=y;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

List append

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

h= x;

c= x;

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y))\}$

$\{(\exists i.\ \ (c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * list\ \tau_2\ y\}$

$\{(c \mapsto \alpha, i) * (list\ \tau_1'\ i) \wedge h = c) * (list\ \tau_2\ y)\}$

$\{c + 1 \mapsto i\}$

n= [c+1];

$\{(c + 1 \mapsto i) \wedge i = n\}$

$\{((c \mapsto \alpha, i) \wedge i = n) * (list\ \tau_1'\ i) \wedge h = c) * (list\ \tau_2\ y)\}$

$\{((c \mapsto \alpha, n) * (list\ \tau_1'\ n) \wedge h = c) * (list\ \tau_2\ y)\}$

```
while(n!=nil)
    c=n;
    n=[c+1];
[c+1]=y;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

List append

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

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)
   c=n;
   n=[c+1];
[c+1]=y;
```

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

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

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) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

[c+1]=y;

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

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

h= x;

c= x;

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$

n= [c+1];

$\{((c \mapsto \alpha, n) * (list\ \tau_1'\ n) \wedge h = c) * (list\ \tau_2\ y)\}$

$$\left\{ \begin{pmatrix} (P \mathrel{-\!\!*} P) \ \wedge \ h = c \\ \quad * (c \mapsto \alpha, n) * (list\ \tau_1'\ n) \end{pmatrix} * (list\ \tau_2\ y) \right\}$$

while(n!=nil) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

[c+1]=y;

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

List append

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

h= x;

c= x;

$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$

n= [c+1];

$\{((c \mapsto \alpha, n) * (list\ \tau_1'\ n) \wedge h = c) * (list\ \tau_2\ y)\}$

$$\left\{\left(\begin{array}{c}(list\ (\alpha :: \tau_1'@\tau_2)\ h) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h))\ \wedge\ h = c \\ * (c \mapsto \alpha, n) * (list\ \tau_1'\ n)\end{array}\right) * (list\ \tau_2\ y)\right\}$$

while(n!=nil) $\left\{\exists \tau', \alpha'. \left(\begin{array}{c}(list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n)\end{array}\right)\right\}$

c=n;

n=[c+1];

[c+1]=y;

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

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

h= x;

c= x;

$$\{((list\ (\alpha :: \tau_1')\ c) \wedge c \neq nil \wedge h = c) * (list\ \tau_2\ y)\}$$

n= [c+1];

$$\{((c \mapsto \alpha, n) * (list\ \tau_1'\ n) \wedge h = c) * (list\ \tau_2\ y)\}$$

$$\left\{ \begin{pmatrix} (list\ (\alpha :: \tau_1' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1' @ \tau_2)\ h)) \\ * (c \mapsto \alpha, n) * (list\ \tau_1'\ n) \end{pmatrix} * (list\ \tau_2\ y) \right\}$$

while(n!=nil) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1' @ \tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

[c+1]=y;

$$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$$

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

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) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1' @ \tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

[c+1]=y;

$$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$$

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

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) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

$\left\{ \left(\exists \alpha'.((list\ (\alpha' :: \tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * (c \mapsto \alpha', nil) \right) * (list\ \tau_2\ y) \right\}$

[c+1]=y;

$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$

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

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) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

$$\left\{ \left(\exists \alpha'.((list\ (\alpha' :: \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * \ (c \mapsto \alpha', nil) \right) * (list\ \tau_2\ y) \right\}$$

[c+1]=y;

$$\left\{ \left(\exists \alpha'.\ ((list\ (\alpha' :: \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * \ (c \mapsto \alpha', y) \right) * (list\ \tau_2\ y) \right\}$$

$$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$$

List append

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

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) $\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$

c=n;

n=[c+1];

$$\left\{ \left(\exists \alpha'.((list\ (\alpha' :: \tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * \ (c \mapsto \alpha', nil) \right) * (list\ \tau_2\ y) \right\}$$

[c+1]=y;

$$\left\{ \left(\exists \alpha'.\ ((list\ (\alpha' :: \tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * \ (c \mapsto \alpha', y) \right) * (list\ \tau_2\ y) \right\}$$

$$\{\exists \alpha'.\ ((list\ (\alpha' :: \tau_2)\ c) -\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * (list\ (\alpha' :: \tau_2)\ c)\}$$

$$\{list\ ((\alpha :: \tau_1')@\tau_2)\ h\}$$

List append

```
while(n!=nil)
```

$$\left\{ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1' @ \tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

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

List append

$$\left\{ n \neq nil \;\wedge\; \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1' @ \tau_2)\ h) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

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

$$\left\{ \exists \tau', \alpha'. \begin{pmatrix} ((list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1' @ \tau_2)\ h)) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \tau', \alpha'. \left(\begin{array}{c} (list\ (\alpha' :: \tau'@\tau_2)\ c) \twoheadrightarrow\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{array} \right) \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau', \alpha'. \left(\begin{array}{l} ((\exists i.(c \mapsto \alpha', i) * list\ (\tau'@\tau_2)\ i)) \twoheadrightarrow\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{array} \right) \right\}$$

c=n;

n=[c+1];

$$\left\{ \exists \tau', \alpha'. \left(\begin{array}{c} ((list\ (\alpha' :: \tau'@\tau_2)\ c) \twoheadrightarrow\!\!* (list\ (\alpha :: \tau_1'@\tau_2)\ h)) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{array} \right) \right\}$$

List append

$$\left\{ n \neq nil \;\wedge\; \exists \tau', \alpha'. \left(\begin{array}{c} (list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * \,(c \mapsto \alpha', n) * (list\ \tau'\ n) \end{array} \right) \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau', \alpha'. \left(\begin{array}{l} ((\exists i.(c \mapsto \alpha', i) * list\ (\tau'@\tau_2)\ i)) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * \,(c \mapsto \alpha', n) * (list\ \tau'\ n) \end{array} \right) \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau'. \left(((list\ (\tau'@\tau_2)\ n) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * (list\ \tau'\ n) \right) \right\}$$

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

$$\left\{ \exists \tau', \alpha'. \left(\begin{array}{c} ((list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) \\ * \,(c \mapsto \alpha', n) * (list\ \tau'\ n) \end{array} \right) \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau'_1 @ \tau_2)\ h) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau', \alpha'. \begin{pmatrix} ((\exists i.(c \mapsto \alpha', i) * list\ (\tau' @ \tau_2)\ i)) \mathrel{-\!\!*} (list\ (\alpha :: \tau'_1 @ \tau_2)\ h) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau'. \Big(((list\ (\tau' @ \tau_2)\ n) \mathrel{-\!\!*} (list\ (\alpha :: \tau'_1 @ \tau_2)\ h)) * (list\ \tau'\ n) \Big) \right\}$$

c=n;

$$\left\{ c \neq nil \ \wedge \ \exists \tau'. \begin{pmatrix} (list\ (\tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau'_1 @ \tau_2)\ h) \\ * \ list\ \tau'\ c \end{pmatrix} \right\}$$

n=[c+1];

$$\left\{ \exists \tau', \alpha'. \begin{pmatrix} ((list\ (\alpha' :: \tau' @ \tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau'_1 @ \tau_2)\ h)) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau', \alpha'. \begin{pmatrix} ((\exists i.(c \mapsto \alpha', i) * list\ (\tau'@\tau_2)\ i)) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau'. \Big(((list\ (\tau'@\tau_2)\ n) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * (list\ \tau'\ n) \Big) \right\}$$

c=n;

$$\left\{ \exists \tau'. \begin{pmatrix} (list\ (\tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * \exists \tau'', \alpha', j.\ (c \mapsto \alpha', j) * (list\ \tau''\ j) \wedge (\alpha' :: \tau'' = \tau') \end{pmatrix} \right\}$$

n=[c+1];

$$\left\{ \exists \tau', \alpha'. \begin{pmatrix} ((list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) \\ * (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \tau', \alpha'. \begin{pmatrix} (list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau', \alpha'. \begin{pmatrix} ((\exists i.(c \mapsto \alpha', i) * list\ (\tau'@\tau_2)\ i)) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \wedge \exists \tau'. \Big(((list\ (\tau'@\tau_2)\ n) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) * (list\ \tau'\ n) \Big) \right\}$$

c=n;

$$\left\{ \exists \tau'. \begin{pmatrix} (list\ (\tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h) \\ * \ \exists \tau'', \alpha', j.\ \ (c \mapsto \alpha', j) * (list\ \tau''\ j) \wedge (\alpha' :: \tau'' = \tau') \end{pmatrix} \right\}$$

n=[c+1];

$$\left\{ \exists \tau', \alpha'. \begin{pmatrix} ((list\ (\alpha' :: \tau'@\tau_2)\ c) \mathrel{-\!\!*} (list\ (\alpha :: \tau_1'@\tau_2)\ h)) \\ * \ (c \mapsto \alpha', n) * (list\ \tau'\ n) \end{pmatrix} \right\}$$

✓

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`