An Introduction to Separation Logic

Stephan van Staden
(slides by Cristiano Calcagno and Matthew Parkinson)

Overview

Introduction

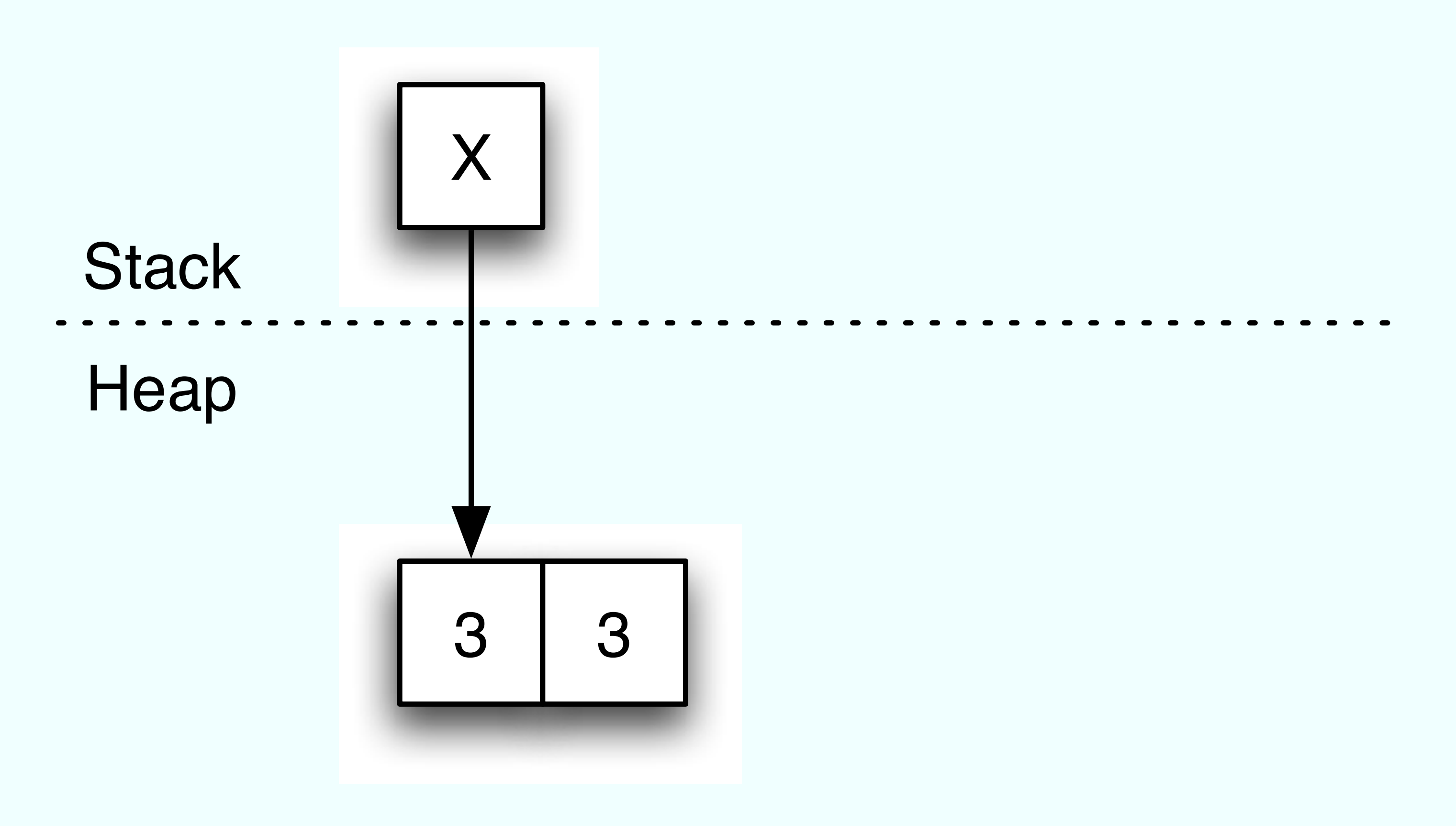
- Motivation
- The Logic
- Some examples

Motivation

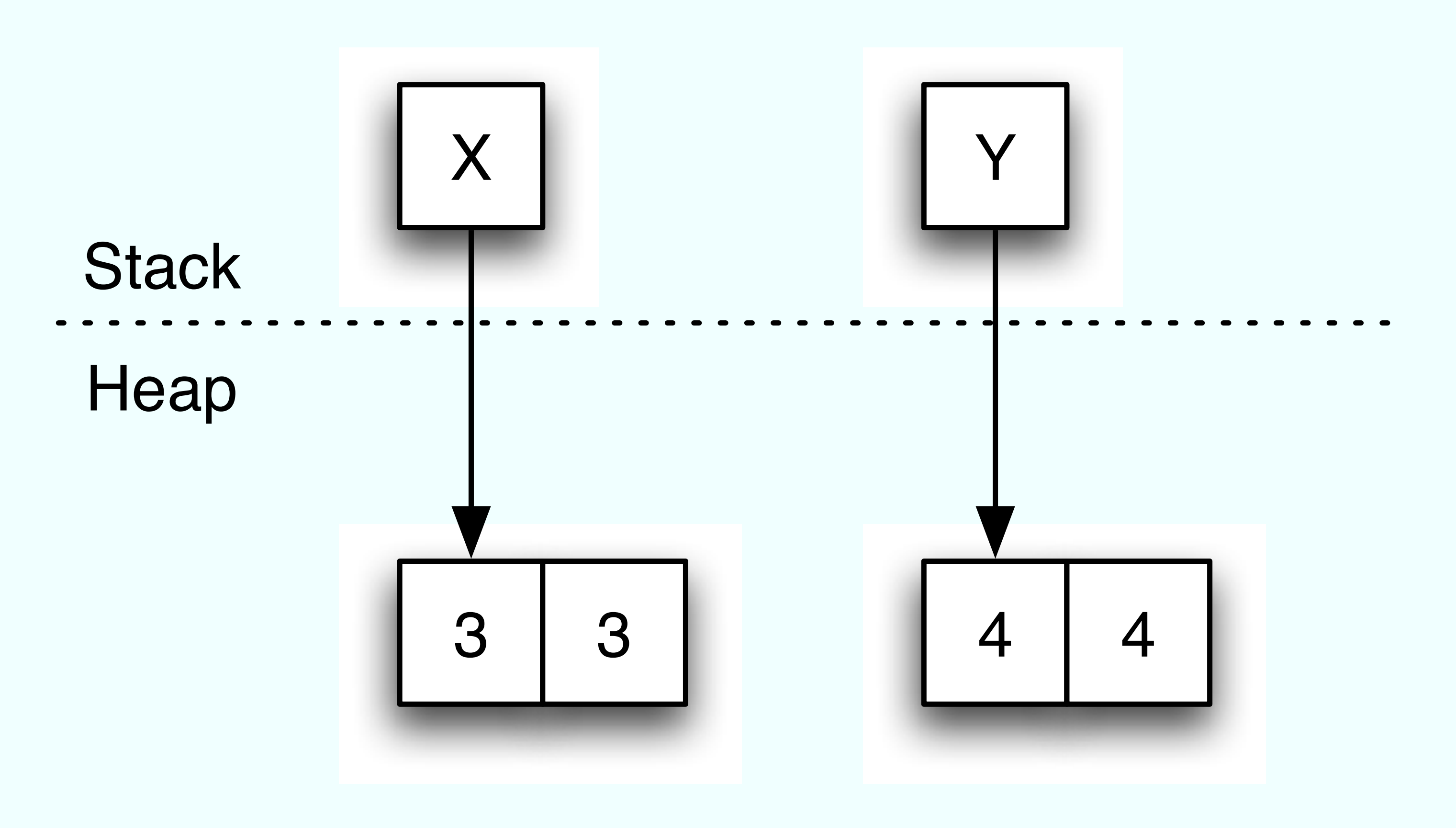
Example program

```
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];
```

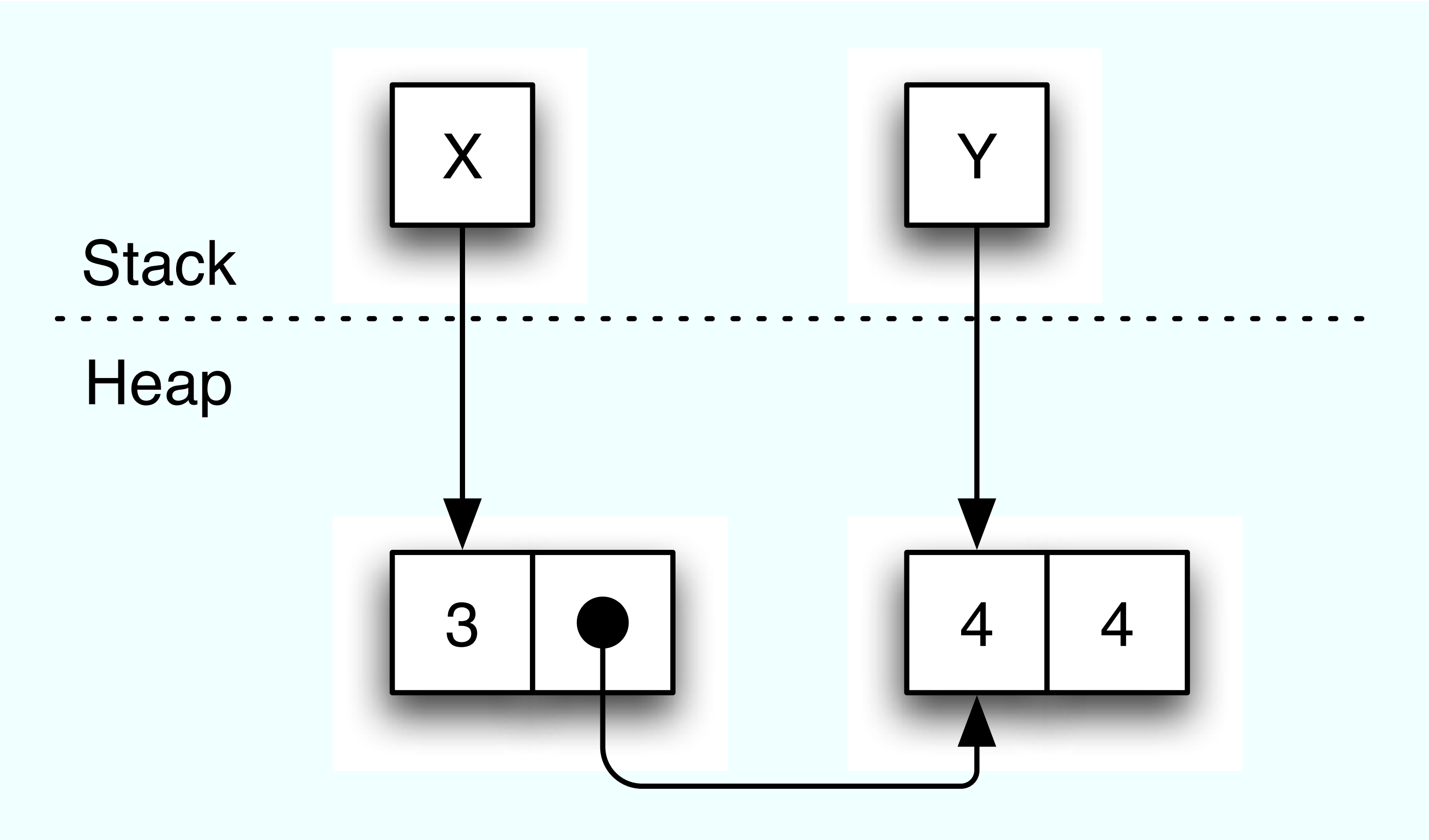


```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```

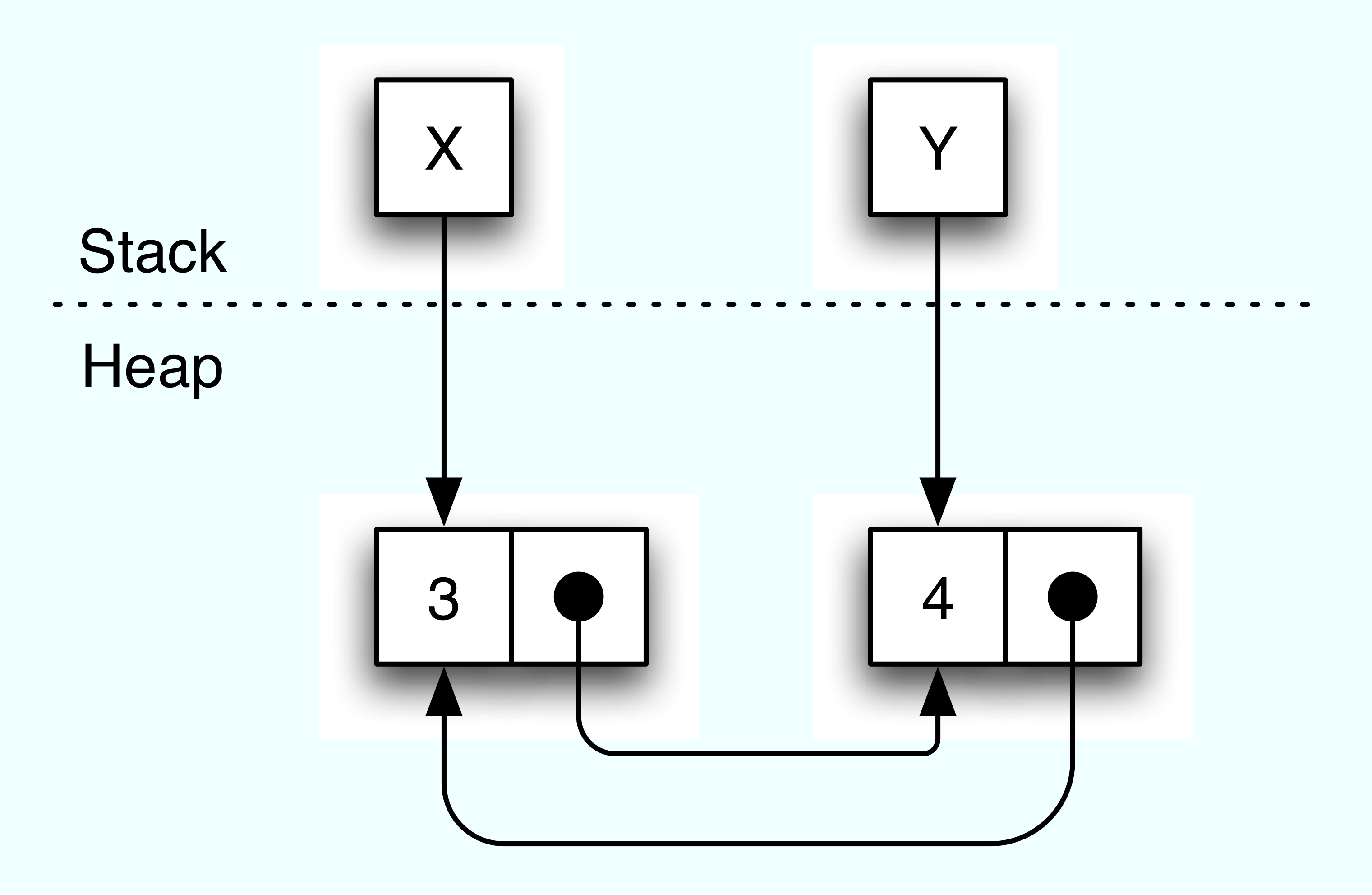


Example program

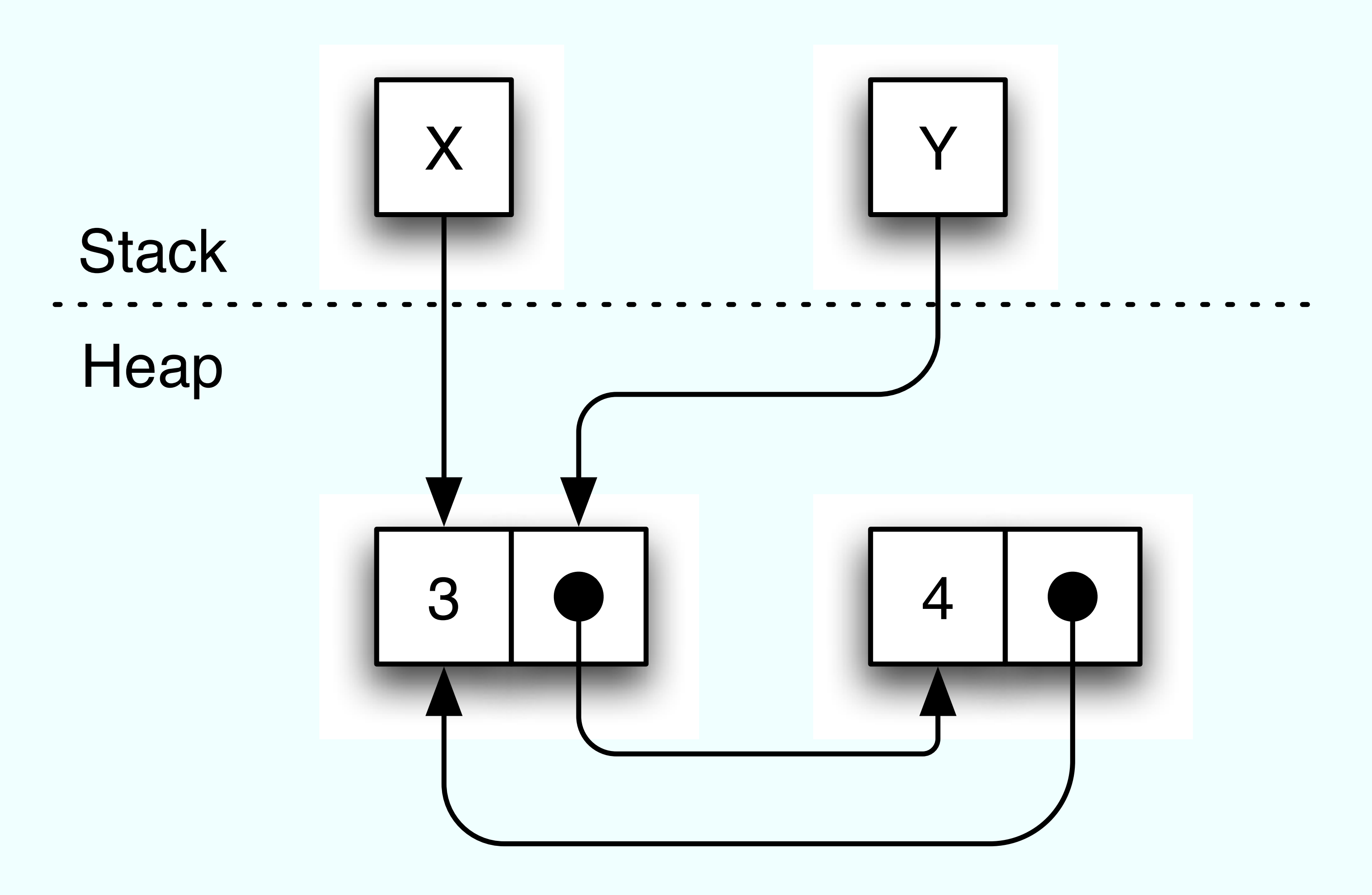
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



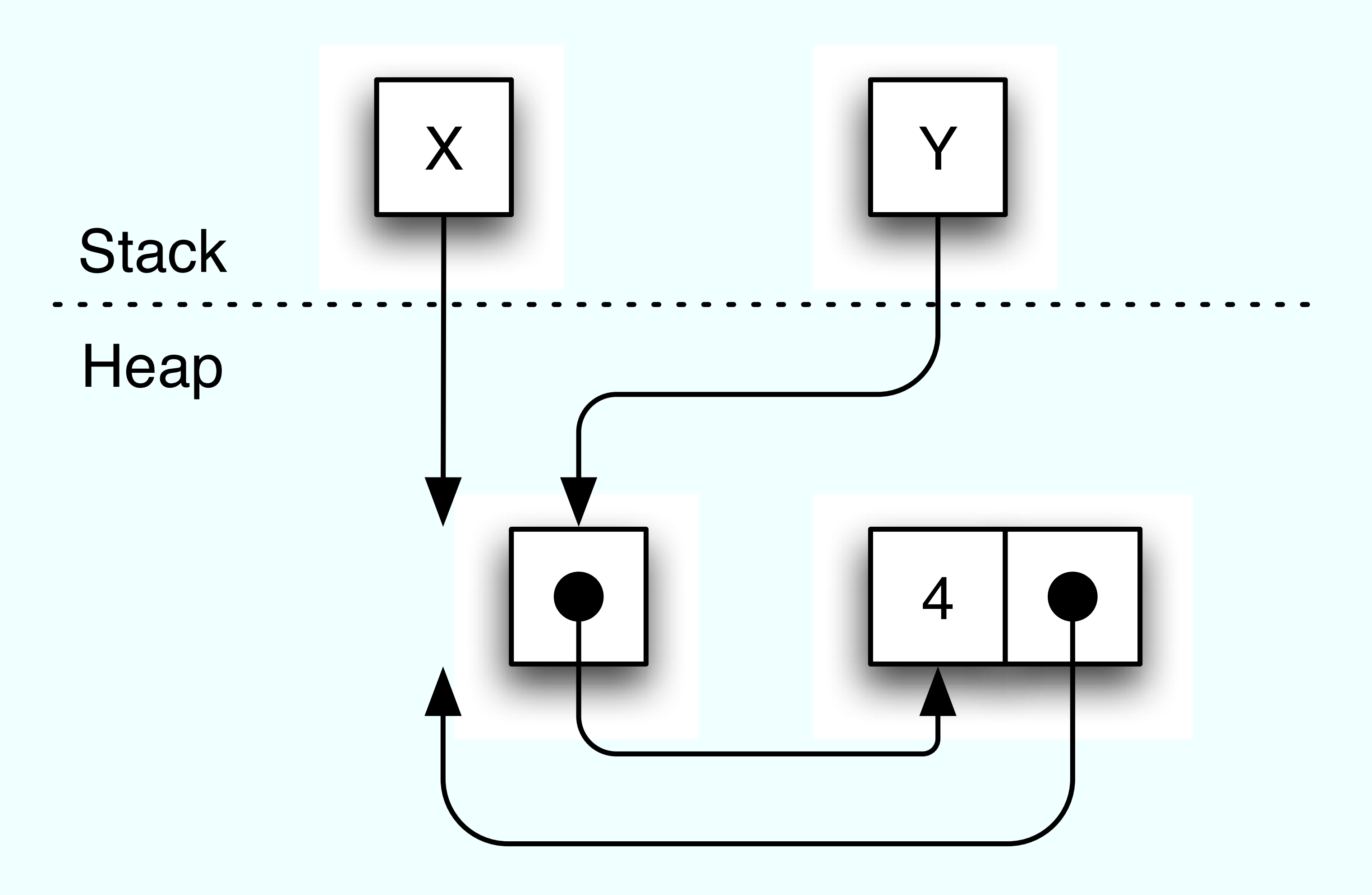
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



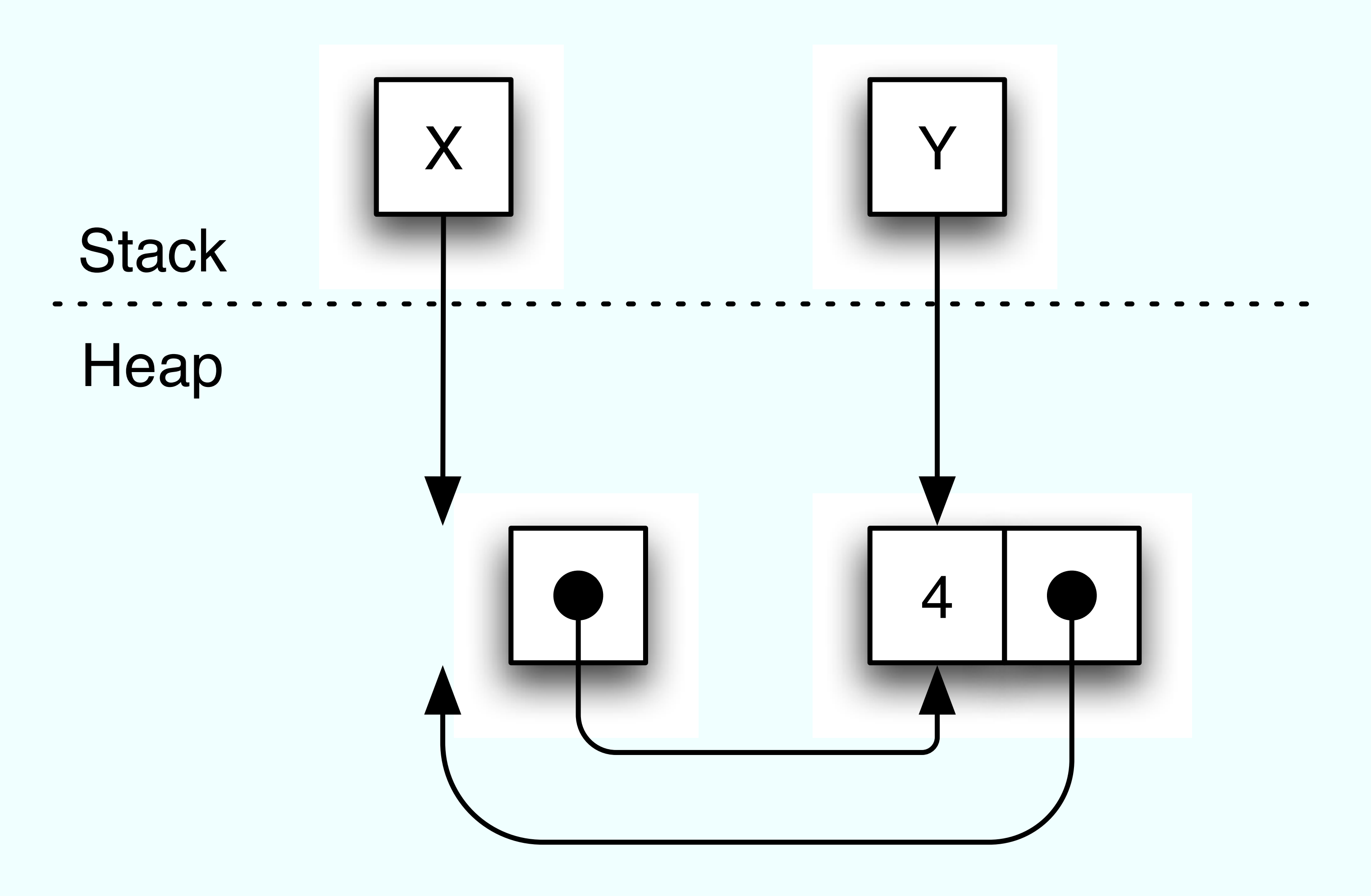
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];



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];



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 locations are different.

Why Separation Logic?

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 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 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 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.

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

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.

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

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.

Separation Logic

Syntax

P,Q	$::=$	$false$	Logical false
	$\mid$	$P \wedge Q$	Classical conjunction
	$\mid$	$P \vee Q$	Classical disjunction
	$\mid$	$P \Rightarrow Q$	Classical implication
	$\mid$	$P * Q$	Separating conjunction
	$\mid$	$P \mathrel{-\!\!*} Q$	Separating implication
	$\mid$	$E = E$	Expression value equality
	$\mid$	$E \mapsto E$	points to
	$\mid$	$empty$	empty heap
	$\mid$	$\exists x.P$	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 : Var $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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.

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

$S, H \models false$	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 $\rightarrow$ 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 $\exists H_1 H_2 . (H_1 \bot H_2) \wedge (H_1 \circ H_2 = H) \wedge (S, H_1 \models P) \wedge (S, H_2 \models Q)$

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

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

Semantics 2/2

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 ;)

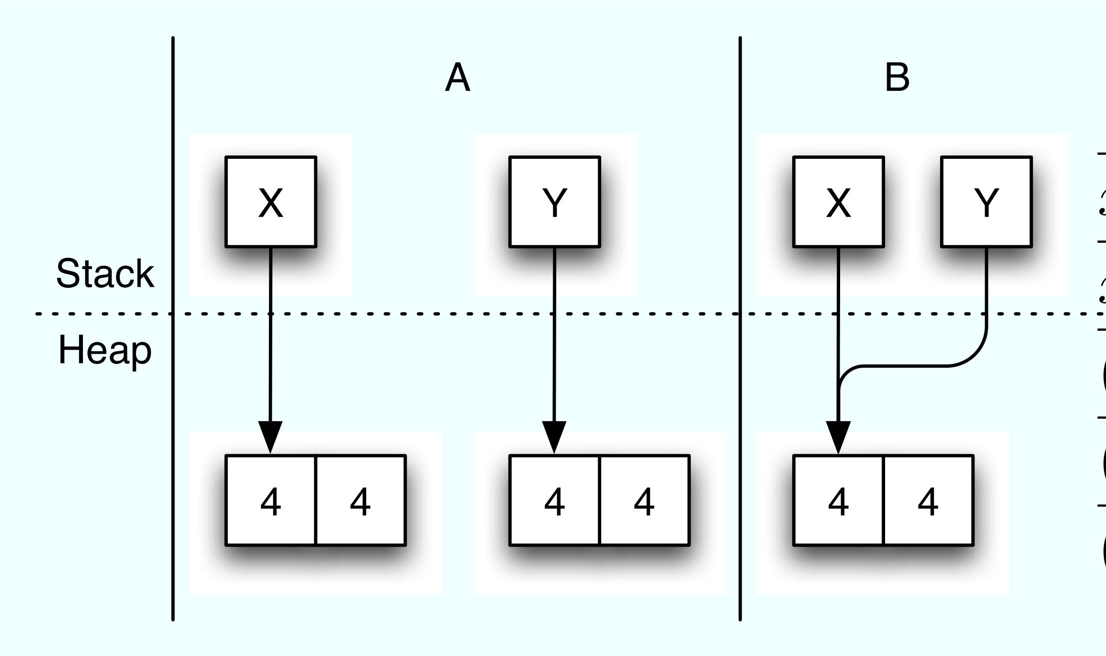
$S, H \models E \mapsto E'$
 iff $dom(H) = \{[\![E]\!]_S\} \wedge H([\![E]\!]_S) = [\![E']\!]_S$

$S, H \models P * Q$
 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.

Example heaps



	A	B
$x \mapsto 4,4$	X	✓
$x \hookrightarrow 4,4$	✓	✓
$(x \mapsto 4,4) \ast (y \mapsto 4,4)$	✓	X
$(x \mapsto 4,4) \wedge (y \mapsto 4,4)$	X	✓
$(x \hookrightarrow 4,4) \wedge (y \hookrightarrow 4,4)$	✓	✓

where

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

and $E \hookrightarrow E' \stackrel{\textbf{def}}{=} (E \mapsto E') \ast \mathrm{true}$

Similarities

$$\begin{array}{rcl} P \wedge Q & \Leftrightarrow & Q \wedge P \\ P \wedge true & \Leftrightarrow & P \\ P \wedge (P \Rightarrow Q) & \Rightarrow & Q \end{array} \qquad \begin{array}{rcl} P * Q & \Leftrightarrow & Q * P \\ P * empty & \Leftrightarrow & P \\ P * (P \mathrel{-\!\!*} Q) & \Rightarrow & Q \end{array}$$

$\wedge$ versus $*$

Similarities

$$\begin{array}{rcl} P \wedge Q & \Leftrightarrow & Q \wedge P \\ P \wedge true & \Leftrightarrow & P \\ P \wedge (P \Rightarrow Q) & \Rightarrow & Q \end{array} \qquad \begin{array}{rcl} P * Q & \Leftrightarrow & Q * P \\ P * empty & \Leftrightarrow & P \\ P * (P \mathrel{-\!\!*} Q) & \Rightarrow & Q \end{array}$$

Differences

$$\begin{array}{rcl} P & \Rightarrow & P \wedge P \\ P \wedge P & \Rightarrow & P \end{array} \qquad \begin{array}{rcl} one & \not\Rightarrow & one * one \\ one * one & \not\Rightarrow & one \end{array}$$

where $one \stackrel{\textbf{def}}{=} \exists x, y.(x \mapsto y)$.

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

$$(S, H, [E] := E') \Downarrow error \quad \text{if} \quad [\![E]\!]_S \notin dom(H)$$
$$(S, H, [E] := E') \Downarrow (S, (H \mid l \rightarrow [\![E']\!]_S)) \quad \text{if} \quad [\![E]\!]_S = l \in dom(H)$$

$$(S, H, x := [E]) \Downarrow error \quad \text{if} \quad [\![E]\!]_S \notin dom(H)$$
$$(S, H, x := [E]) \Downarrow ((S \mid x \rightarrow H(l)), H) \quad \text{if} \quad [\![E]\!]_S = l \in dom(H)$$

$$(S, H, dispose(E)) \Downarrow error \quad \text{if} \quad [\![E]\!]_S \notin dom(H)$$
$$(S, H, dispose(E)) \Downarrow (S, H - l) \quad \text{if} \quad [\![E]\!]_S = l \in dom(H)$$

$$(S, H, x := cons(E_0, \ldots, E_n)) \Downarrow$$
$$((S \mid x \rightarrow l), (H \mid l \rightarrow [\![E_0]\!]_S \cdots l + n \rightarrow [\![E_n]\!]_S)) \quad \text{if} \quad l, \ldots, l + n \notin dom(H)$$

We say that (S, H, C) is safe if $(S, H, C) \not\Downarrow error$.

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

$$\{X = x \wedge (E \mapsto Y)\} \;\; x := [E] \;\; \{(E[X/x] \mapsto Y) \wedge Y = x\}$$

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

$$\{empty\} \;\; x := cons(E_0, \ldots, E_n) \;\; \{(x \mapsto E_0, \ldots, E_n)\}$$

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

The most important rule

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

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

Why $\mathit{modifies}$?

$$\begin{array}{rcl} \mathit{modifies}([E] := E') & = & \mathit{modifies}(\mathit{dispose}(E)) = \emptyset \\ \mathit{modifies}(x := [E]) & = & \mathit{modifies}(x := \mathit{cons}(E_0, \ldots, E_n)) = \{x\} \end{array}$$

Otherwise

$$\frac{\{(x \mapsto 4)\} \quad y := [x] \quad \{(x \mapsto 4) \wedge y = 4\}}{\{(x \mapsto 4) * (y = 3)\} \quad y := [x] \quad \{((x \mapsto 4) \wedge y = 4) * (y = 3)\}}$$

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 $S', H' \models Q$

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!

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 $S', H' \models Q$

Tight interpretation!

Why safe? Otherwise

$$\frac{\{\mathit{true}\}\ \ [x] := 7\ \ \{\mathit{true}\}}{\{\mathit{true} * (x \mapsto 4)\}\ \ [x] := 7\ \ \{\mathit{true} * (x \mapsto 4)\}}$$

Consider binary trees

$$\tau \stackrel{\textbf{def}}{=} \epsilon \mid (\tau_1, a, \tau_2)$$

Consider binary trees

$$\tau \stackrel{\textbf{def}}{=} \epsilon \mid (\tau_1, a, \tau_2)$$

We can give the definition of a binary tree predicate as

$$\begin{array}{rcl} tree\, \epsilon\, i & \equiv & empty \;\; \wedge \;\; i = nil \\ tree\, (\tau_1, a, \tau_2)\, i & \equiv & \exists j, k.\;\; (i \mapsto j, a, k) \;\; * \;\; (tree\, \tau_1\; j) \;\; * \;\; (tree\, \tau_2\; k) \end{array}$$

Consider binary trees

$$\tau \overset{\textbf{def}}{=} \epsilon \mid (\tau_1, a, \tau_2)$$

We can give the definition of a binary tree predicate as

$$\begin{aligned} tree\ \epsilon\ i \quad &\equiv \quad empty \quad \wedge \quad i = nil \\ tree\ (\tau_1, a, \tau_2)\ i \quad &\equiv \quad \exists j, k.\ (i \mapsto j, a, k) \quad * \quad (tree\ \tau_1\ j) \quad * \quad (tree\ \tau_2\ k) \end{aligned}$$

Properties

$$(33 \mapsto 41, a, nil) * (41 \mapsto nil, b, nil) \quad \Longrightarrow \quad tree\ ((\epsilon, b, \epsilon), a, \epsilon)\ 33$$
$$tree\ \tau\ i \quad \Longrightarrow \quad (\tau = \epsilon) \Leftrightarrow empty \Leftrightarrow (i = nil)$$
$$(tree\ \_\ i) \wedge (i \neq nil) \quad \Longrightarrow \quad \exists j, k.\ (i \mapsto j, \_, k) * (tree\ \_\ j) * (tree\ \_\ k)$$

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
```

$\{empty\}$

$\{(tree\_p)\}$

```
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);
```

$\{empty\}$

$\{(tree\ \_\ p)\ \wedge p \neq nil\}$

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

$\{empty\}$

Tree dispose

$\{(tree\ \_\ p)\ \wedge p \neq nil\}$
$\{\exists i, j.\ \ (p \mapsto i, \_, j)\ \ * (tree\ \_\ i)\ \ \ *\ \ \ (tree\ \_\ j)\}$

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

$\{empty\}$

Tree dispose

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

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

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

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

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

$\{empty\}$

$\{(tree\ \_\ p)\ \wedge p \neq nil\}$

$\{\exists i, j.\ \ (p \mapsto i, \_, j)\ * (tree\ \_\ i)\ \ *\ \ (tree\ \_\ j)\}$

$\{(p \mapsto i')\ \ *\ \ \exists j.\ \ (p + 1 \mapsto \_, j)\ * (tree\ \_\ i')\ \ *\ \ (tree\ \_\ j)\}$

$\{i = i \wedge (p \mapsto i')\}$

i := [p];

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

$\{(p \mapsto i') \wedge i = i'\}$

j := [p+2];

dispTree(i);

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

$\{(tree\ \_\ p)\ \wedge p \neq nil\}$
$\{\exists i, j.\ \ (p \mapsto i, \_, j)\ * (tree\ \_\ i)\ \ *\ \ (tree\ \_\ j)\}$
$\{(p \mapsto i')\ \ *\ \ \exists j.\ \ (p + 1 \mapsto \_, j)\ * (tree\ \_\ i')\ \ *\ \ (tree\ \_\ j)\}$

$\{i = i \wedge (p \mapsto i')\}$

i := [p];

$\{(p \mapsto i') \wedge i = i'\}$

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

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];

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);

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

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

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

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

$\{empty\}$

dispTree(j);

dispose(p+2);

dispose(p+1);

dispose(p);

$\{empty\}$

$\{(tree\ \_\ p)\ \wedge p \neq nil\}$

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);

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

$\{empty\}$

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

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];

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

$\{(tree\_i)\}$

dispTree(i);

$\{empty\}$

$\{(p \mapsto i, \_, j) \ * \ empty \ * \ (tree\_j)\}$

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

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);

$\{empty\}$

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

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);

$\{(p \mapsto i, \_, j)\}$

dispose(p+2);

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

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);

$\{(p \mapsto i, \_, j)\}$

dispose(p+2);

$\{(p \mapsto i, \_)\}$

dispose(p+1); $\boxed{\{E \mapsto \_\} \;\; dispose(E) \;\; \{empty\}}$

dispose(p);

$\{empty\}$

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

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);

$\{(p \mapsto i, \_, j)\}$

dispose(p+2);

$\{(p \mapsto i, \_)\}$

dispose(p+1);

$\{(p \mapsto i)\}$

dispose(p);

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

$\{empty\}$

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);

$\{(p \mapsto i, \_, j)\}$

dispose(p+2);

$\{(p \mapsto i, \_)\}$

dispose(p+1);

$\{(p \mapsto i)\}$

dispose(p);

$\{empty\}$

$\checkmark$

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

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);

$\{(p \mapsto i, \_, j)\}$

dispose(p+2);

$\{(p \mapsto i, \_)\}$

dispose(p+1);

$\{(p \mapsto i)\}$

dispose(p);

$\{empty\}$

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

Consider sequences of integers $\sigma \stackrel{\text{def}}{=} [\,] \mid a :: \sigma$ and the recursive formula

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

This formula defines a linked list.

Consider sequences of integers $\sigma \stackrel{\textbf{def}}{=} [] \mid a :: \sigma$ and the recursive formula

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

This formula defines a linked list. We will also need list segments

$$
\begin{aligned}
lseg\ []\ i\ k &\equiv empty \;\wedge\; i = k \\
lseg\ (a\!::\!\sigma)\ i\ k &\equiv \exists j.\ (i \mapsto a, j) * (list\ \sigma\ j\ k)
\end{aligned}
$$

Consider sequences of integers $\sigma \stackrel{\text{def}}{=} [] \mid a :: \sigma$ and the recursive formula

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

This formula defines a linked list. We will also need list segments

$$\begin{aligned} \mathit{lseg}\ []\ i\ k &\equiv \mathit{empty} \ \wedge\ i = k \\ \mathit{lseg}\ (a :: \sigma)\ i\ k &\equiv \exists j.\ (i \mapsto a, j) * (\mathit{list}\ \sigma\ j\ k) \end{aligned}$$

Properties

$$(33 \mapsto a, 41) * (41 \mapsto b, 11) \iff \mathit{lseg}\ [a :: b]\ 33\ 11$$

$$\mathit{list}\ \sigma\ i \iff \mathit{lseg}\ \sigma\ i\ \mathit{nil}$$

$$(\mathit{lseg}\ \sigma_1\ i\ j) * (\mathit{lseg}\ \sigma_2\ j\ k) \Rightarrow \mathit{lseg}\ (\sigma_1 :: \sigma_2)\ i\ k$$

List append

$\{(list\ \sigma_1\ x) * (list\ \sigma_2\ y)\}$

```
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
```

$\{list\ (\sigma_1 :: \sigma_2)\ ret\}$

List append

$\{(list\ \sigma_1\ x) * (list\ \sigma_2\ y)\}$

```
proc append(x,y)
   newvar h,c,n;
   if x=nil then return y;
```

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

```
   return h;
end proc
```

$\{list\ (\sigma_1 :: \sigma_2)\ ret\}$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$
$\{(\exists i.\ \ (c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * list\ \sigma_2\ y\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$
$\{(\exists i.\ \ (c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * list\ \sigma_2\ y\}$
$\{((c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

$$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$$
$$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$$

h:= x;

c:= x;

$$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$
$$\{(\exists i.\ \ (c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * list\ \sigma_2\ y\}$$
$$\{((c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$
$$\{c + 1 \mapsto i\}$$

n:= [c+1];

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

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

$$\{list\ (\sigma_1 :: \sigma_2)\ h\}$$

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$
$\{(\exists i.\ \ (c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * list\ \sigma_2\ y\}$
$\{((c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$
$\{c + 1 \mapsto i\}$

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

$\{(c + 1 \mapsto i) \wedge i = n\}$
$\{((c \mapsto a, i) \wedge i = n) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

List append

$$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$$
$$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$$

h:= x;

c:= x;

$$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$
$$\{(\exists i.\ \ (c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * list\ \sigma_2\ y\}$$
$$\{((c \mapsto a, i) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$
$$\{c + 1 \mapsto i\}$$

n:= [c+1];

$$\{(c + 1 \mapsto i) \wedge i = n\}$$
$$\{((c \mapsto a, i) \wedge i = n) * (list\ \sigma_1'\ i) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$
$$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$

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

$$\{list\ (\sigma_1 :: \sigma_2)\ h\}$$

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

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

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

$$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$$

$$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$$

h:= x;

c:= x;

$$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$

n:= [c+1];

$$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$

c:= n;

n:= [c+1];

[c+1] := y;

$$\{list\ (\sigma_1 :: \sigma_2)\ h\}$$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$

$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

h:= x;

c:= x;

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

n:= [c+1];

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

$$\left\{ \begin{pmatrix} (empty \wedge h = c) * (c \mapsto a, n) * (list\ \sigma_1'\ n) \\ \wedge\ \sigma_1 = a :: \sigma_1' \end{pmatrix} * (list\ \sigma_2\ y) \right\}$$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$

c:= n;

n:= [c+1];

[c+1] := y;

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

h:= x;

c:= x;

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

n:= [c+1];

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

$$\left\{ \left(\begin{array}{c} (lseg\ []\ h\ c \wedge h = c) * (c \mapsto a, n) * (list\ \sigma_1'\ n) \\ \wedge\ \sigma_1 = a :: \sigma_1' \end{array} \right) * (list\ \sigma_2\ y) \right\}$$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \left(\begin{array}{c} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{array} \right) \right\}$

c:= n;

n:= [c+1];

[c+1] := y;

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$

$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

h:= x;

c:= x;

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

n:= [c+1];

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

$$\left\{\left(\begin{array}{c}(lseg\ []\ h\ c) * (c \mapsto a, n) * (list\ \sigma_1'\ n) \\ \wedge\ \sigma_1 = [] :: a :: \sigma_1'\end{array}\right) * (list\ \sigma_2\ y)\right\}$$

while(n!=nil) $\left\{\exists \sigma', a', \sigma''. \left(\begin{array}{c}(lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'')\end{array}\right)\right\}$

c:= n;

n:= [c+1];

[c+1] := y;

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

List append

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

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

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

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

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$

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

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

h:= x;

c:= x;

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

n:= [c+1];

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$

c:= n;

n:= [c+1];

$\left\{ \Big((lseg\ \sigma'\ h\ c) * (c \mapsto a', nil) \wedge (\sigma_1 = \sigma' :: a') \Big) * (list\ \sigma_2\ y) \right\}$

[c+1] := y;

$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

$$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$$
$$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$$

h:= x;

c:= x;

$$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$

n:= [c+1];

$$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$

c:= n;

n:= [c+1];

$$\left\{ \left((lseg\ \sigma'\ h\ c) * (c \mapsto a', nil) \wedge (\sigma_1 = \sigma' :: a') \right) * (list\ \sigma_2\ y) \right\}$$

[c+1] := y;

$$\left\{ \left((lseg\ \sigma'\ h\ c) * (c \mapsto a', y) \wedge (\sigma_1 = \sigma' :: a') \right) * (list\ \sigma_2\ y) \right\}$$
$$\{list\ (\sigma_1 :: \sigma_2)\ h\}$$

$\{((list\ \sigma_1\ x) \wedge x! = nil) * (list\ \sigma_2\ y)\}$
$\{((list\ (a :: \sigma_1')\ x) \wedge \sigma_1 = a :: \sigma_1') * (list\ \sigma_2\ y)\}$

h:= x;

c:= x;

$\{((list\ (a :: \sigma_1')\ c) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

n:= [c+1];

$\{((c \mapsto a, n) * (list\ \sigma_1'\ n) \wedge \sigma_1 = a :: \sigma_1' \wedge h = c) * (list\ \sigma_2\ y)\}$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$

c:= n;

n:= [c+1];

$\left\{ \Big((lseg\ \sigma'\ h\ c) * (c \mapsto a', nil) \wedge (\sigma_1 = \sigma' :: a') \Big) * (list\ \sigma_2\ y) \right\}$

[c+1] := y;

$\left\{ \Big((lseg\ \sigma'\ h\ c) * (c \mapsto a', y) \wedge (\sigma_1 = \sigma' :: a') \Big) * (list\ \sigma_2\ y) \right\}$
$\{(lseg\ \sigma_1\ h\ y) * (list\ \sigma_2\ y)\}$
$\{list\ (\sigma_1 :: \sigma_2)\ h\}$

while(n!=nil) $\left\{ \exists \sigma', a', \sigma''. \left(\begin{array}{c} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{array} \right) \right\}$

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

List append

$$\left\{ n \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

c:= n;

n:= [c+1];

$$\left\{ \exists \sigma', a', \sigma''. \ \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

c:= n;

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c') * (c' \mapsto a', c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

n:= [c+1];

$$\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

c:= n;

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c') * (c' \mapsto a', c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: \sigma'') \end{pmatrix} \right\}$$

n:= [c+1];

$$\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ n \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

c:= n;

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c') * (c' \mapsto a', c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n') * (list\ \sigma''\ n') \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

n:= [c+1];

$$\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

List append

$$\left\{ n \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

c:= n;

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c') * (c' \mapsto a', c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ c \neq nil \ \wedge \ \exists \sigma', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (list\ \sigma''\ c) \\ \wedge\ (\sigma_1 = \sigma' :: \sigma'') \end{pmatrix} \right\}$$

$$\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n') * (list\ \sigma''\ n') \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

n:= [c+1];

$$\left\{ \exists \sigma', a', \sigma''. \begin{pmatrix} (lseg\ \sigma'\ h\ c) * (c \mapsto a', n) * (list\ \sigma''\ n) \\ \wedge\ (\sigma_1 = \sigma' :: a' :: \sigma'') \end{pmatrix} \right\}$$

✓

Are small axioms and frame rule enough?

From small axiom

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

apply frame $(E \mapsto E') \mathrel{-\!\!*} Q$ to obtain

$$\{(E \mapsto \_) * ((E \mapsto E') \mathrel{-\!\!*} Q)\} \;\; [E] := E' \;\; \{(E \mapsto E') * ((E \mapsto E') \mathrel{-\!\!*} Q)\}$$

then consequence, to get the weakest precondition

$$\{(E \mapsto \_) * ((E \mapsto E') \mathrel{-\!\!*} Q)\} \;\; [E] := E' \;\; \{Q\}$$

Weakest precondition: if $\{P\} \;\; [E] := E' \;\; \{Q\}$ holds,
then $P \Rightarrow (E \mapsto \_) * ((E \mapsto E') \mathrel{-\!\!*} Q)$.

Conclusions

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