Automatic Verification with the Infer Static Analyser

Dino Distefano

Facebook

and

Queen Mary University of London

Infer

A tool to detect bugs in Android and iOS apps before they ship

Goal of the course

Study some theoretical foundations of Infer

or

Look at Separation Logic having automatic verification in mind

Today's plan

- Motivation for Separation Logic
- Logic & semantics
- Programming language & semantics
- Small axioms & Frame Rule
- Symbolic Execution
- Abstraction techniques for the heap

Simple Imperative Language

- Safe commands:
    - S::= skip | x:=E | x:=new(E1,...,En)
- Heap accessing commands:
    - A(E) ::= dispose(E) | x:=[E] | [E]:=F

where E is and expression x, y, nil, etc.

- Command:
    - C::= S | A | C1;C2 | if B { C1 } else {C2} | while B do { C }

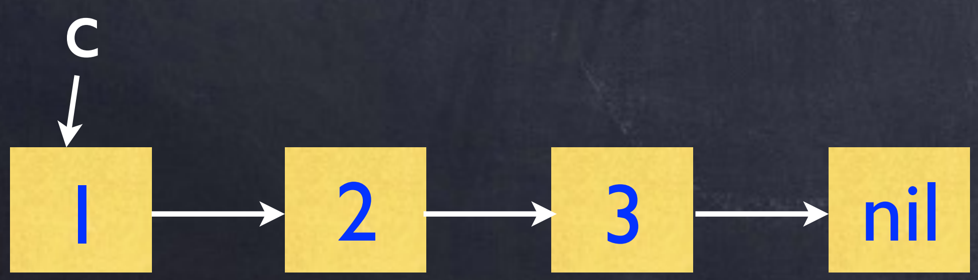
where B boolean guard E=E, E!=E, etc.

Example Program: List Reversal

```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

Example Program: List Reversal

```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```



Example Program: List Reversal

```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

Example Program: List Reversal

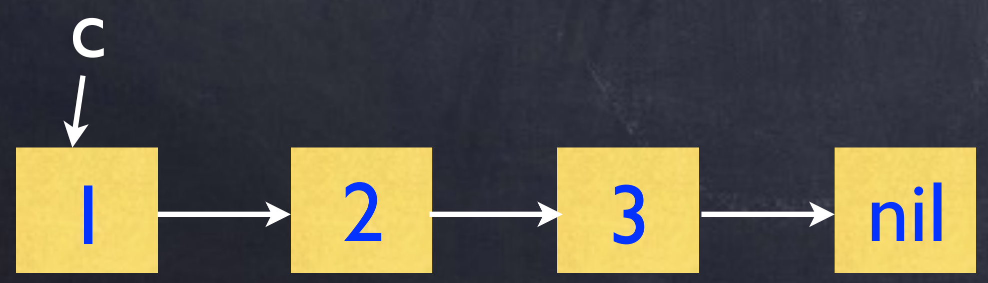
```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

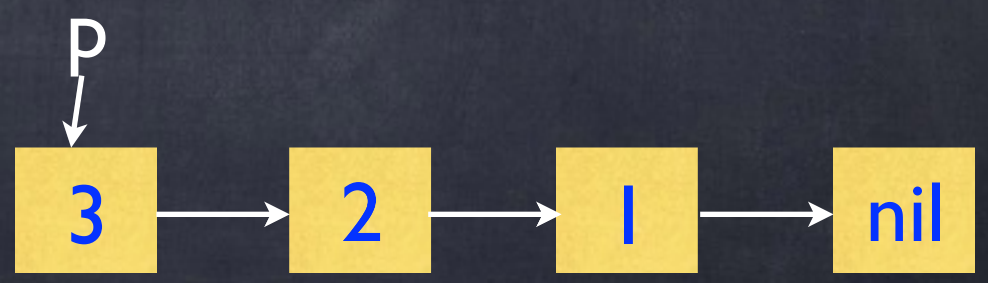
Some properties we would like to prove:

Does the program preserve acyclicity/cyclicity?

Does it core-dump?

Does it create garbage?

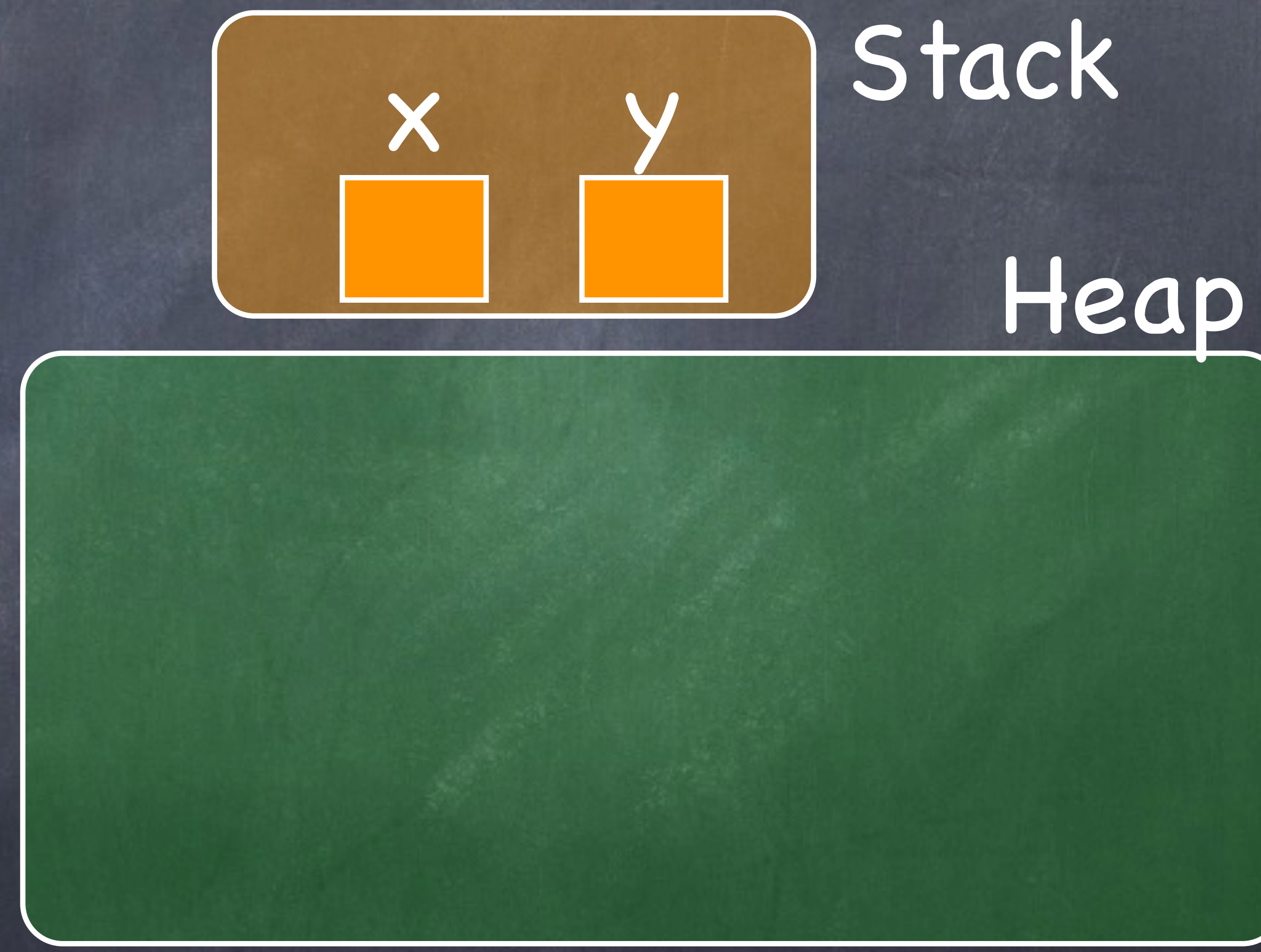




Example Program

We are interested in pointer manipulating programs

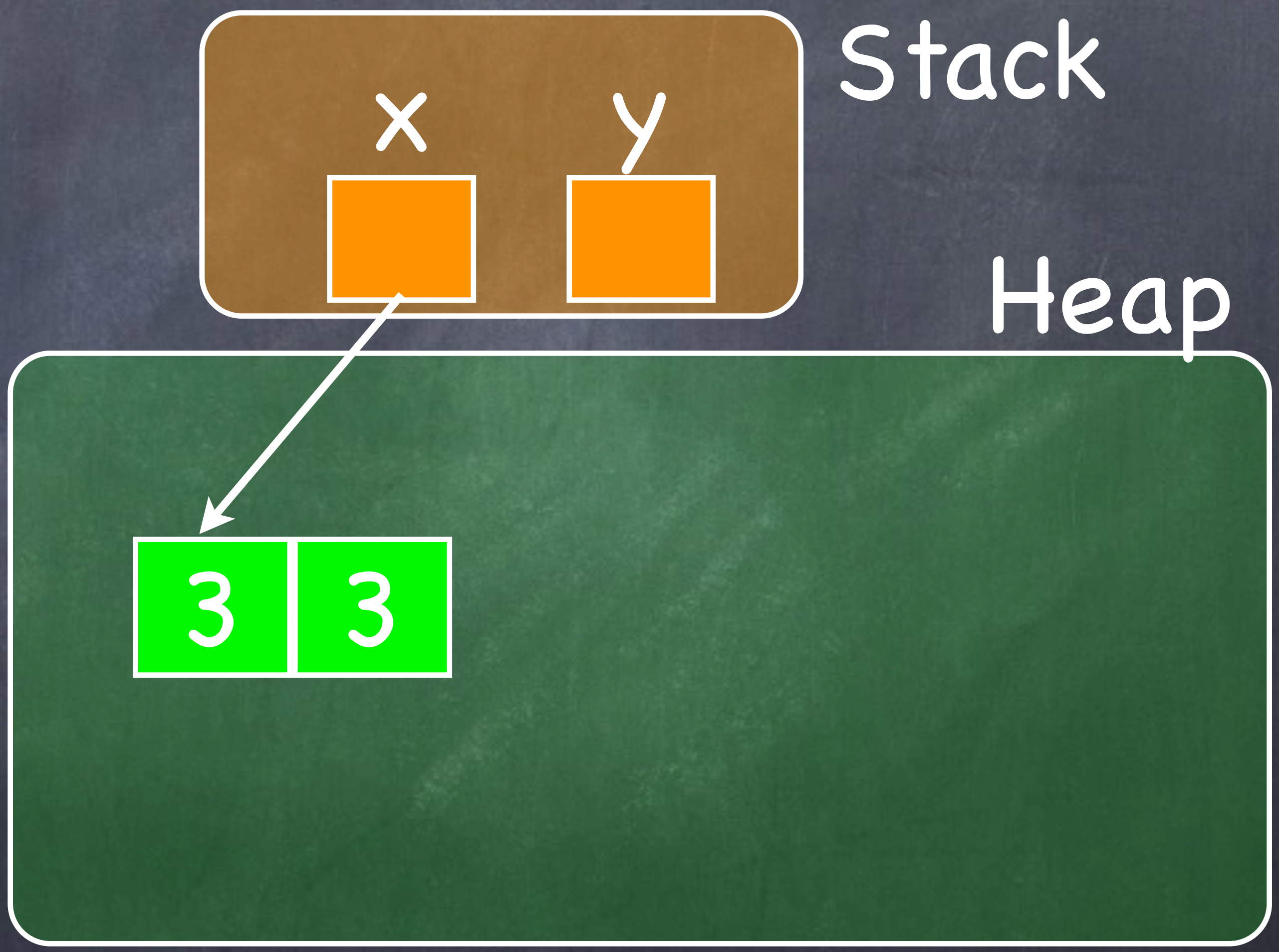
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

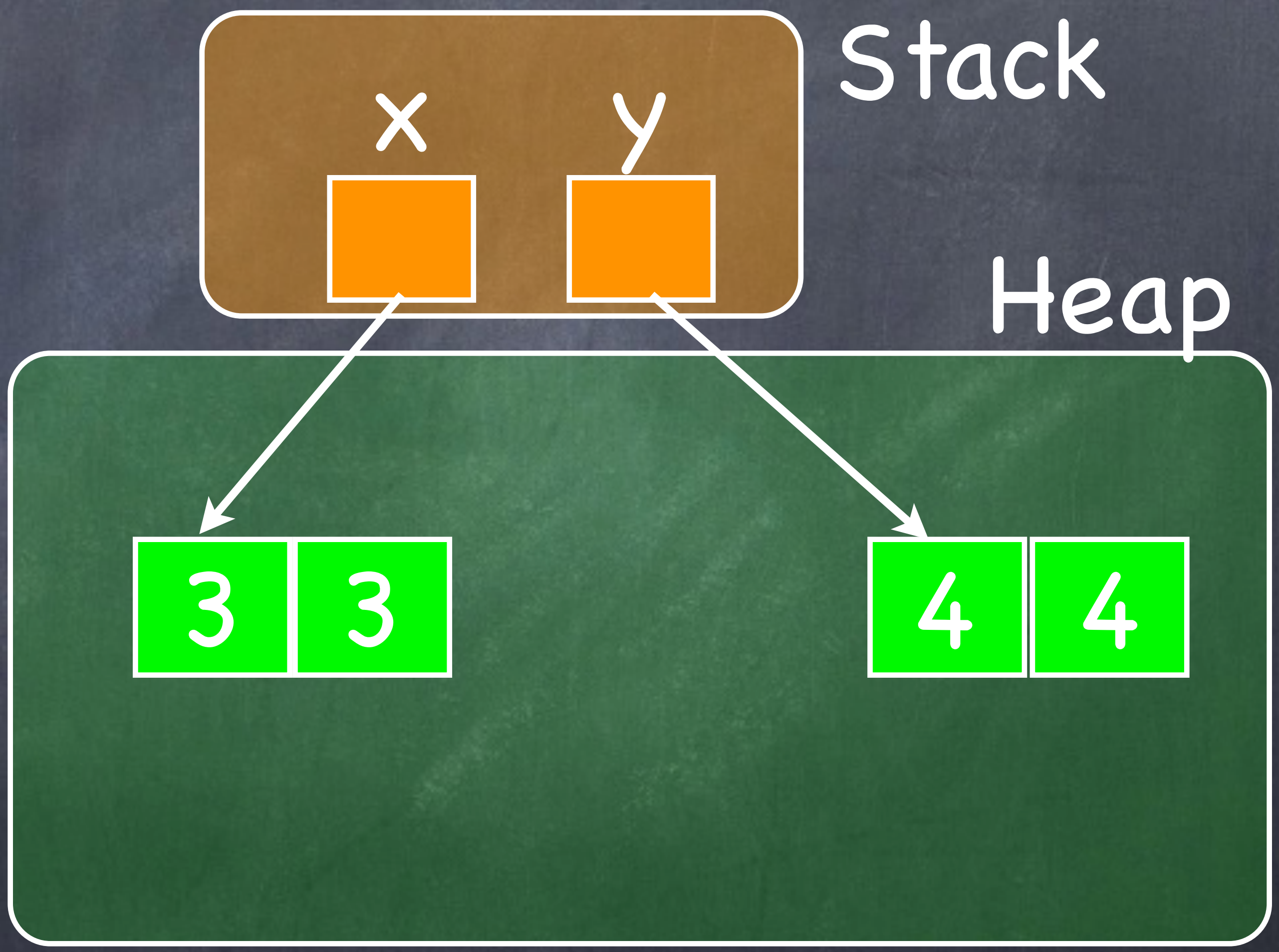
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

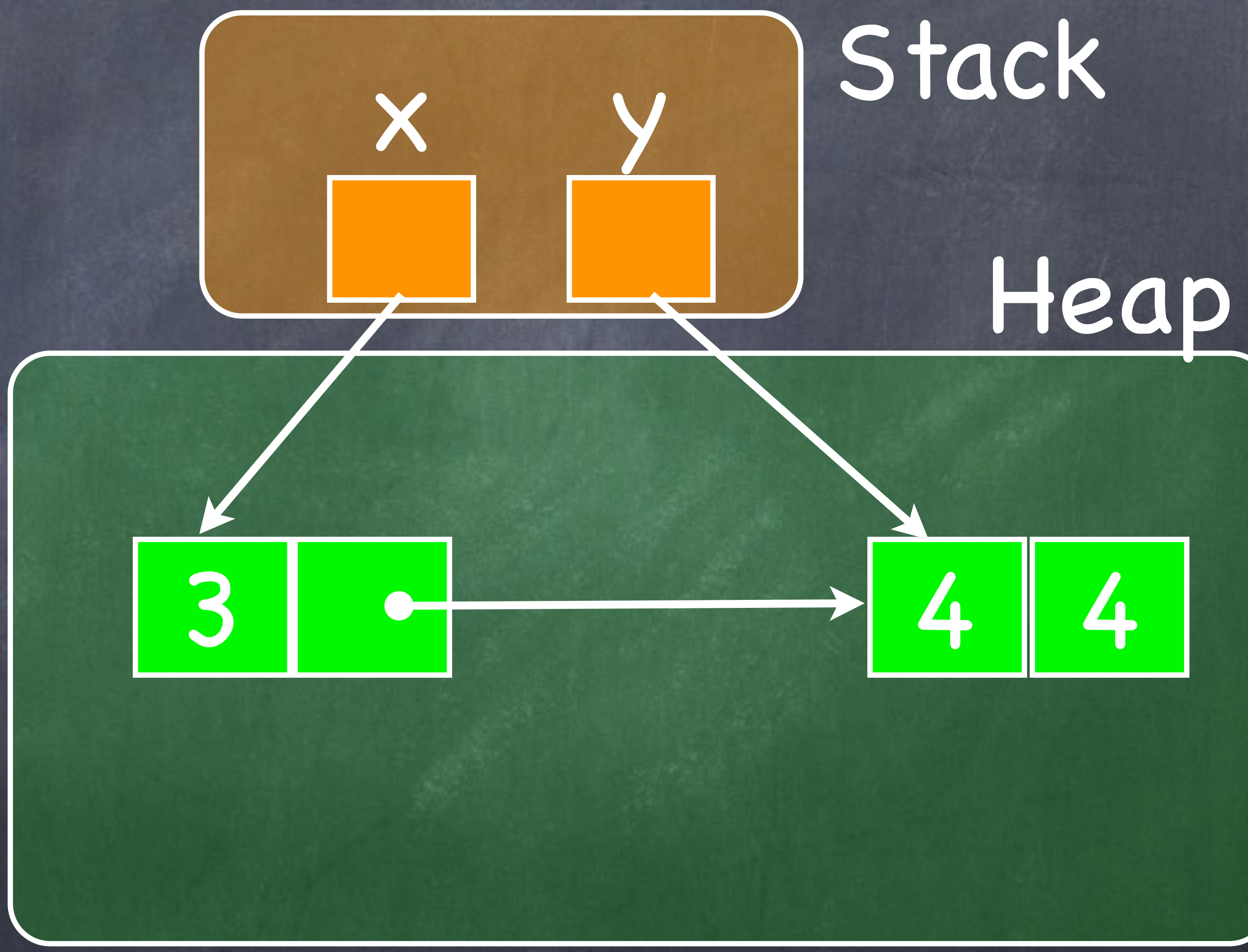
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

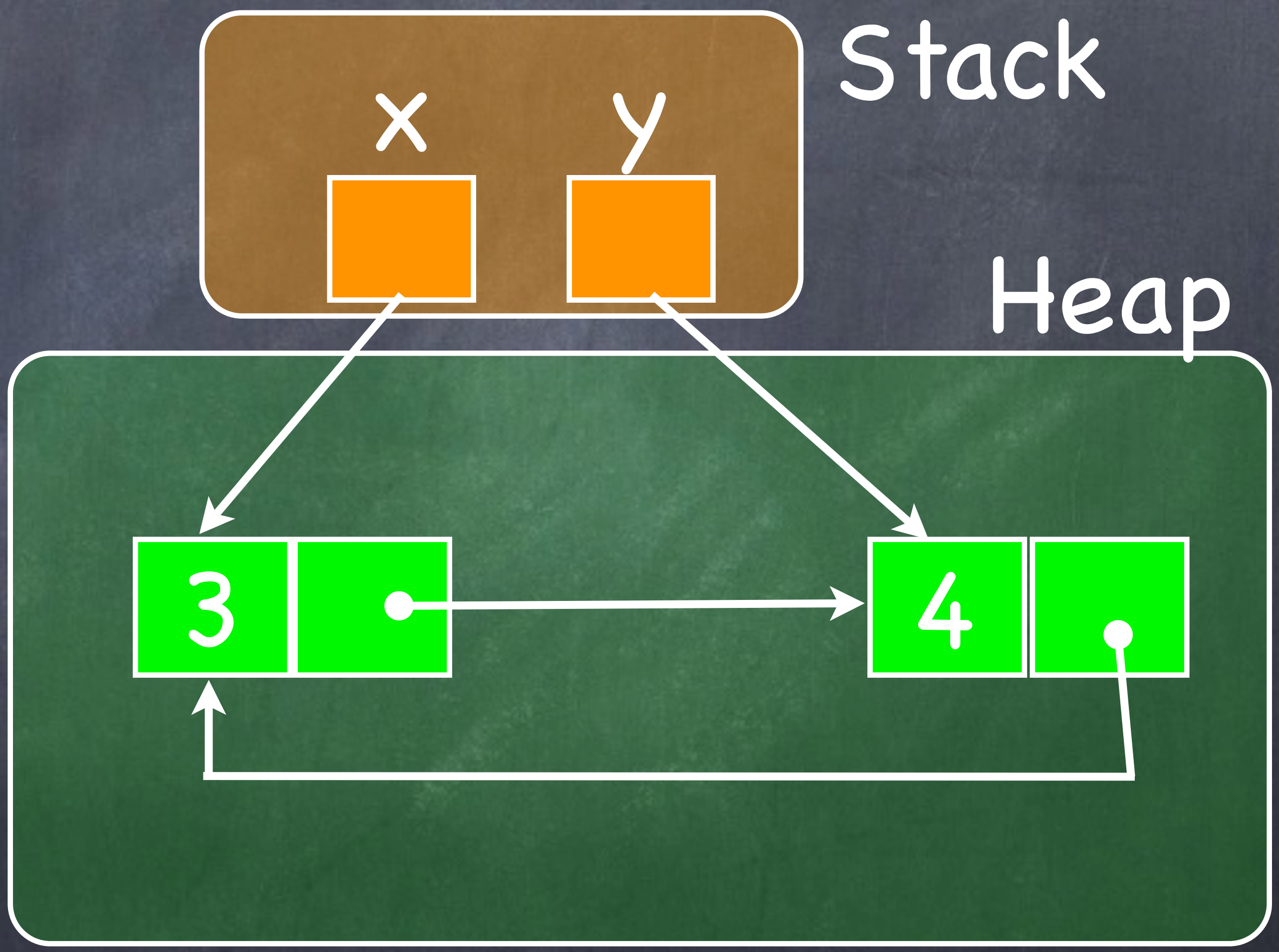
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

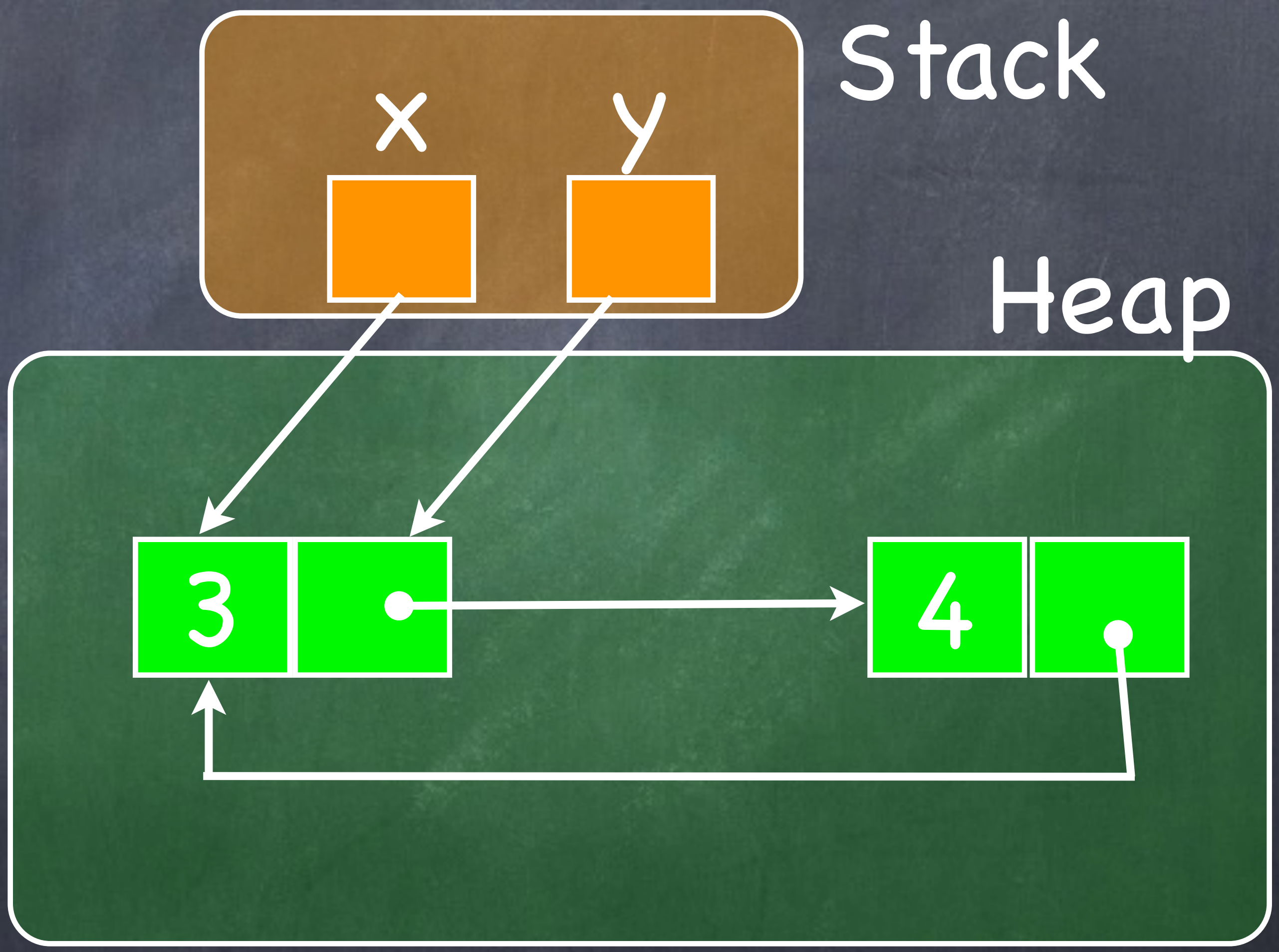
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

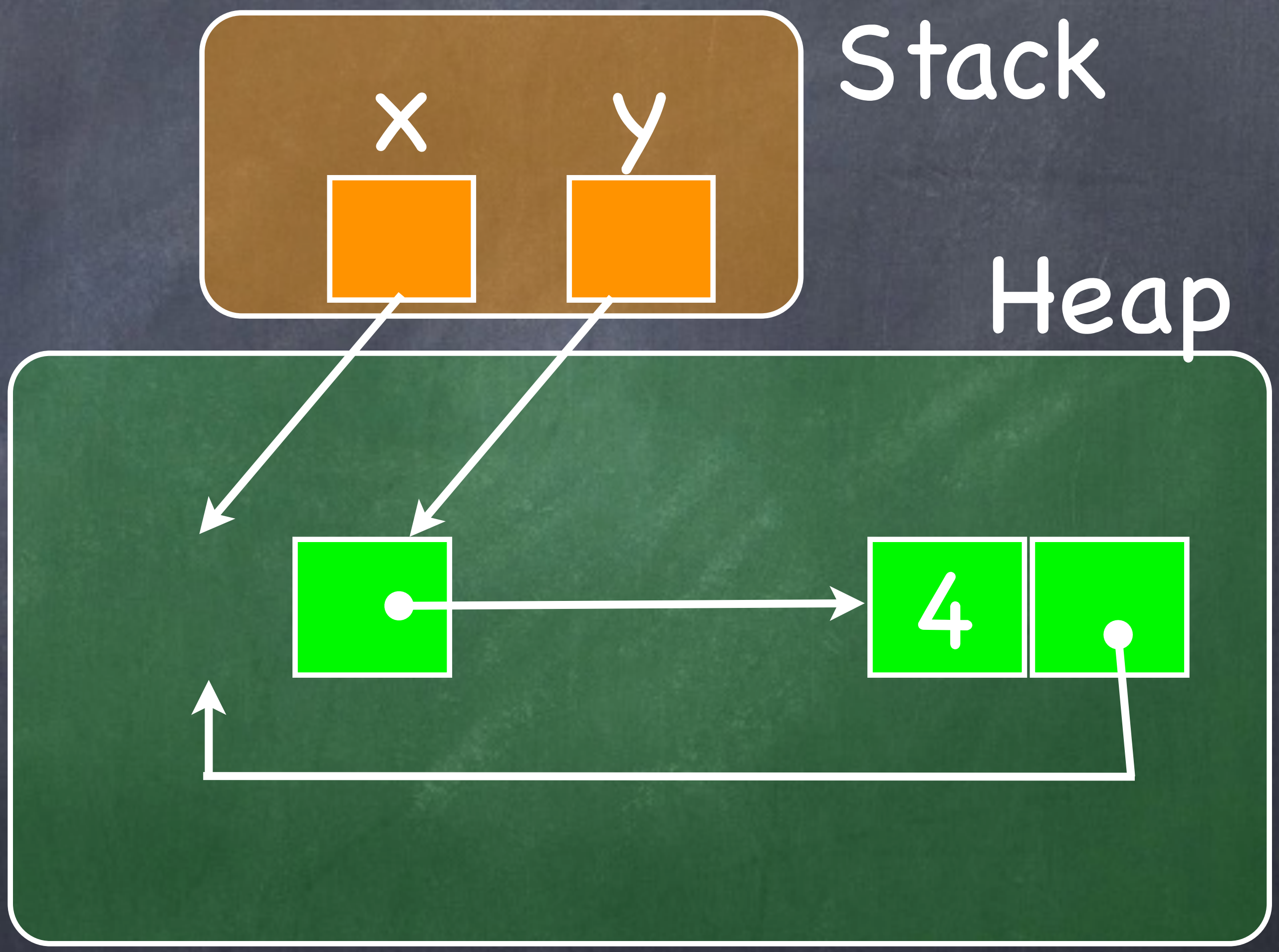
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

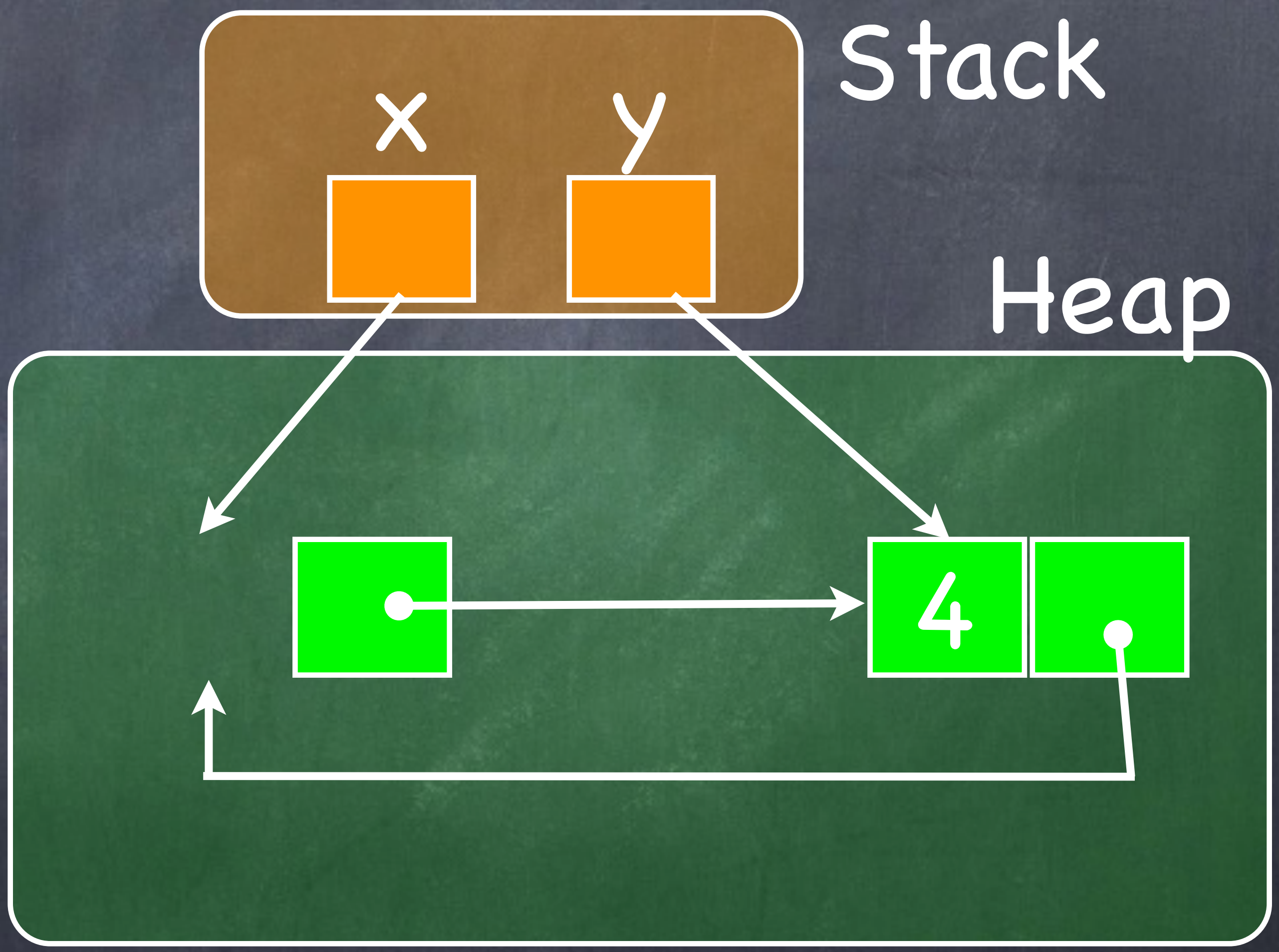
```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Example Program

We are interested in pointer manipulating programs

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



Why Separation Logic?

Consider this code:

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

We need to know that things are different. How?

Why Separation Logic?

Consider this code:

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

Add assertion?

We need to know that things are different. How?

Why Separation Logic?

Consider this code:

Add assertion?

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

We need to know that things are different. How?

We need to know that things stay the same. How?

Why Separation Logic?

Consider this code:

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

Add assertion?

We need to know that things are different. How?

We need to know that things stay the same. How?

Why Separation Logic?

Consider this code:

Assume([x] = 3 && x!=y && x!=z) Add assertion?
Assume(y != z) Add assertion?

```
[y] = 4;
[z] = 5;
```

Guarantee([y] != [z])
Guarantee([x] = 3)

We need to know that things are different. How?
We need to know that things stay the same. How?

Framing

We want a general concept of things not being affected.

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

What are the conditions on C and R?

Hard to define if reasoning about a heap and aliasing

Framing

We want a general concept of things not being affected.

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

What are the conditions on C and R?

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.

The Logic

Storage Model

$$\mathsf{Vars} \stackrel{def}{=} \{x, y, z, \ldots\}$$

$$\mathsf{Locs} \stackrel{def}{=} \{1, 2, 3, 4, \ldots\} \qquad \mathsf{Vals} \supseteq \mathsf{Locs}$$

$$\begin{aligned}
\mathsf{Heaps} &\stackrel{def}{=} \mathsf{Locs} \rightarrow_{\mathsf{fin}} \mathsf{Vals} \\
\mathsf{Stacks} &\stackrel{def}{=} \mathsf{Vars} \rightarrow \mathsf{Vals} \\
\mathsf{States} &\stackrel{def}{=} \mathsf{Stacks} \times \mathsf{Heaps}
\end{aligned}$$

Storage Model

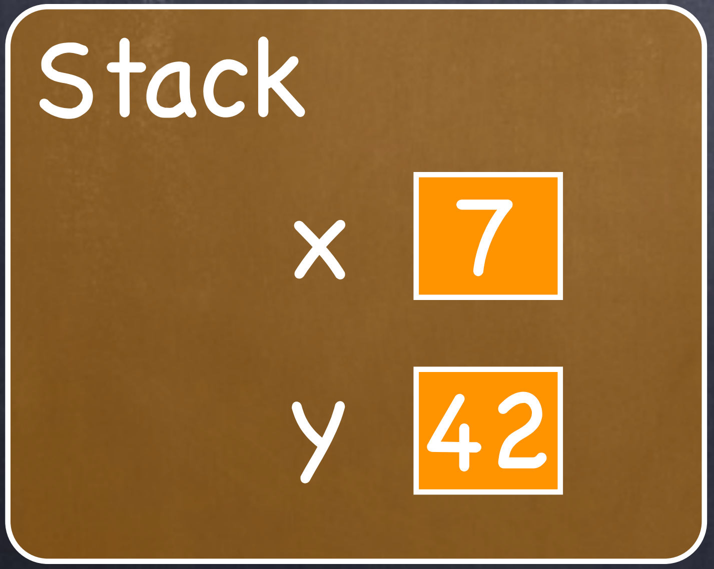
$$\mathsf{Vars} \stackrel{def}{=} \{x, y, z, \ldots\}$$

$$\mathsf{Locs} \stackrel{def}{=} \{1, 2, 3, 4, \ldots\} \qquad \mathsf{Vals} \supseteq \mathsf{Locs}$$

$$\mathsf{Heaps} \stackrel{def}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals}$$

$$\mathsf{Stacks} \stackrel{def}{=} \mathsf{Vars} \to \mathsf{Vals}$$

$$\mathsf{States} \stackrel{def}{=} \mathsf{Stacks} \times \mathsf{Heaps}$$



Storage Model

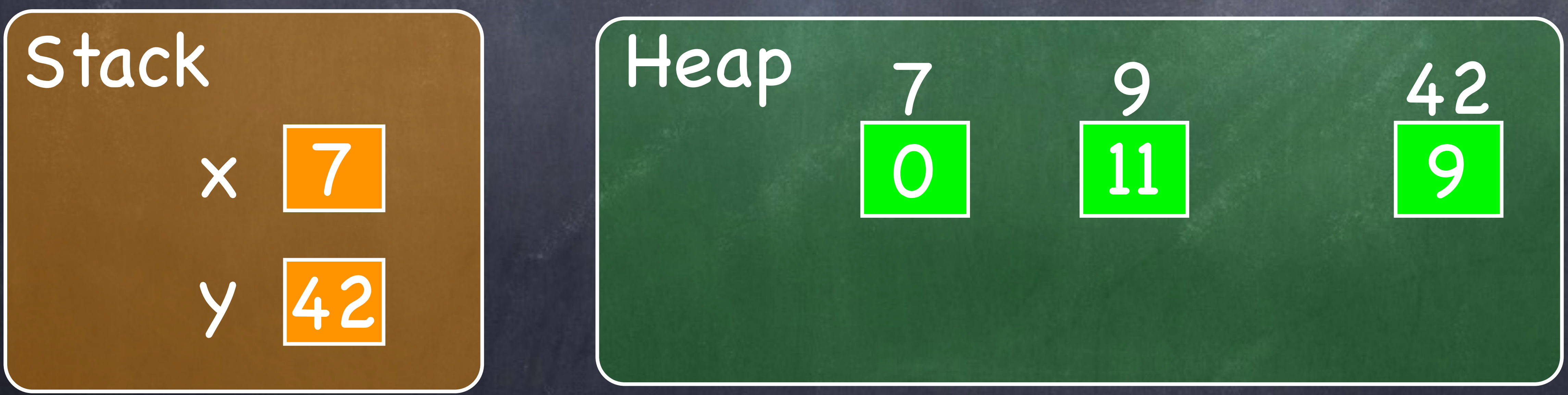
$$\mathsf{Vars} \stackrel{\textit{def}}{=} \{x, y, z, \ldots\}$$

$$\mathsf{Locs} \stackrel{\textit{def}}{=} \{1, 2, 3, 4, \ldots\} \qquad \mathsf{Vals} \supseteq \mathsf{Locs}$$

$$\mathsf{Heaps} \stackrel{\textit{def}}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals}$$

$$\mathsf{Stacks} \stackrel{\textit{def}}{=} \mathsf{Vars} \rightarrow \mathsf{Vals}$$

$$\mathsf{States} \stackrel{\textit{def}}{=} \mathsf{Stacks} \times \mathsf{Heaps}$$



Storage Model

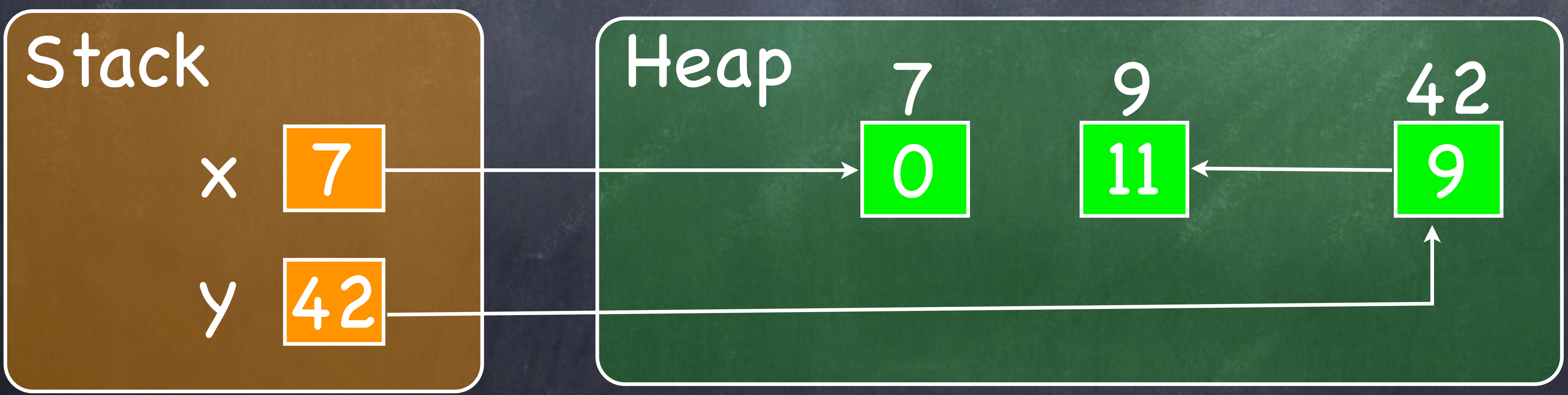
$$\mathsf{Vars} \stackrel{\mathit{def}}{=} \{x, y, z, \ldots\}$$

$$\mathsf{Locs} \stackrel{\mathit{def}}{=} \{1, 2, 3, 4, \ldots\} \qquad \mathsf{Vals} \supseteq \mathsf{Locs}$$

$$\mathsf{Heaps} \stackrel{\mathit{def}}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals}$$

$$\mathsf{Stacks} \stackrel{\mathit{def}}{=} \mathsf{Vars} \to \mathsf{Vals}$$

$$\mathsf{States} \stackrel{\mathit{def}}{=} \mathsf{Stacks} \times \mathsf{Heaps}$$



Mathematical Structure of Heap

$$\mathsf{Heaps} \stackrel{def}{=} \mathsf{Locs} \rightarrow_{\mathsf{fin}} \mathsf{Vals}$$

$$h_1 \# h_2 \quad \stackrel{def}{\Longleftrightarrow} \quad \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$$

$$h_1 * h_2 \quad \stackrel{def}{=} \quad \begin{cases} h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Mathematical Structure of Heap

$$\text{Heaps} \stackrel{def}{=} \text{Locs} \rightarrow_{\text{fin}} \text{Vals}$$

$$h_1 \# h_2 \stackrel{def}{\Longleftrightarrow} \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$$

$$h_1 * h_2 \stackrel{def}{=} \begin{cases} h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

1) * has a unit
2) * is associative and commutative

Assertions

$$
\begin{array}{lcll}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid P * Q & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P & \text{Classical Logic}
\end{array}
$$

Informal Meaning

Assertions

$$
\begin{array}{llll}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid P * Q & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P & \text{Classical Logic}
\end{array}
$$

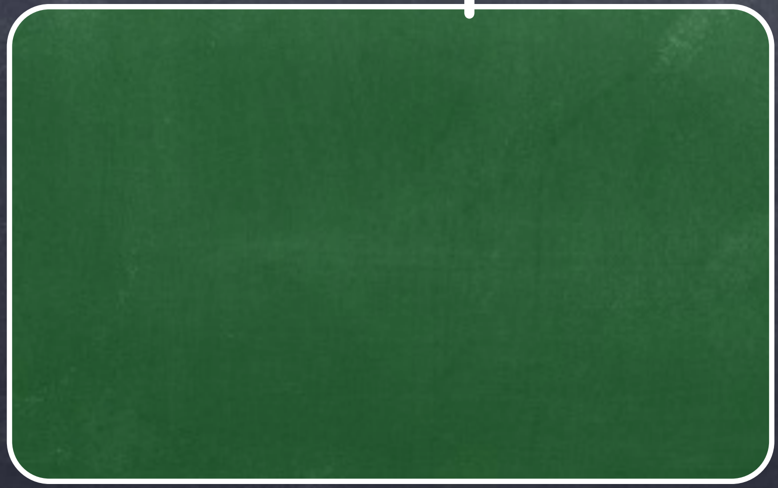
Informal Meaning

Assertions

$$
\begin{array}{lcll}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid P * Q & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P & \text{Classical Logic}
\end{array}
$$

Informal Meaning

Heap



Assertions

$$
\begin{array}{llll}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid P * Q & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P & \text{Classical Logic}
\end{array}
$$

Informal Meaning

Heap

Assertions

E, F	$::=$	$x \mid n \mid E{+}F \mid -E \mid \ldots$	Heap-independent Exprs
P, Q	$::=$	$E = F \mid E \geq F \mid E \mapsto F$	Atomic Predicates
	$\mid$	$\mathsf{emp} \mid P * Q$	Separating Connectives
	$\mid$	$\mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P$	Classical Logic

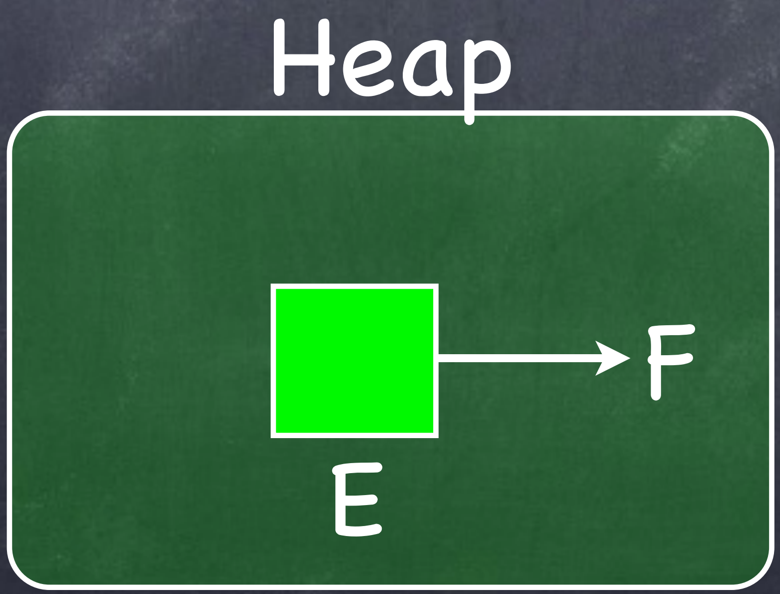
Informal Meaning

Heap

Assertions

$$
\begin{array}{llll}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid P * Q & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P & \text{Classical Logic}
\end{array}
$$

Informal Meaning

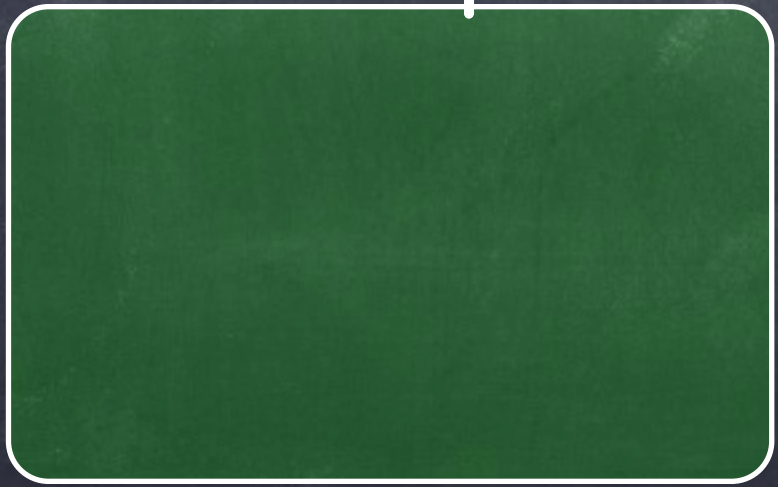


Assertions

E, F	$::=$	$x \mid n \mid E{+}F \mid -E \mid \ldots$	Heap-independent Exprs
P, Q	$::=$	$E = F \mid E \geq F \mid E \mapsto F$	Atomic Predicates
	$\mid$	$\mathsf{emp} \mid P * Q$	Separating Connectives
	$\mid$	$\mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P$	Classical Logic

Informal Meaning

Heap



Assertions

E, F	$::=$	$x \mid n \mid E{+}F \mid -E \mid \ldots$	Heap-independent Exprs
P, Q	$::=$	$E = F \mid E \geq F \mid E \mapsto F$	Atomic Predicates
	$\mid$	$\mathsf{emp} \mid P * Q$	Separating Connectives
	$\mid$	$\mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P$	Classical Logic

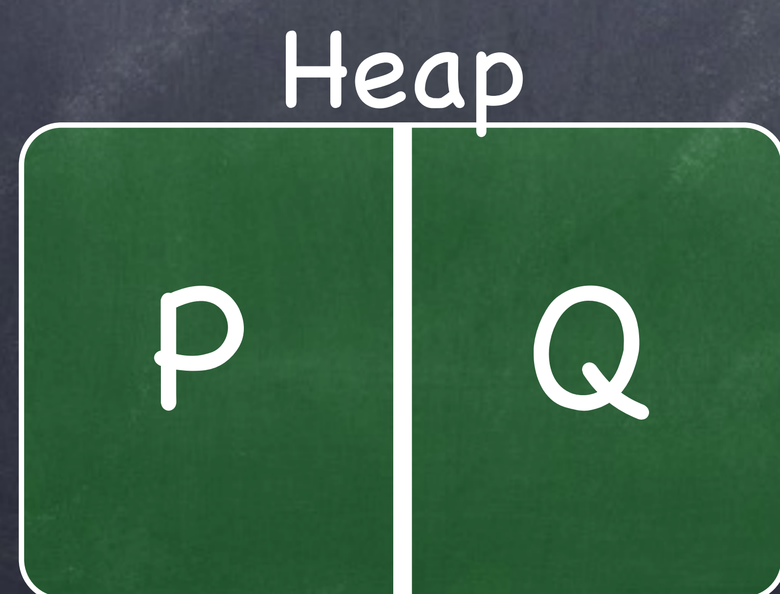
Informal Meaning

Heap

Assertions

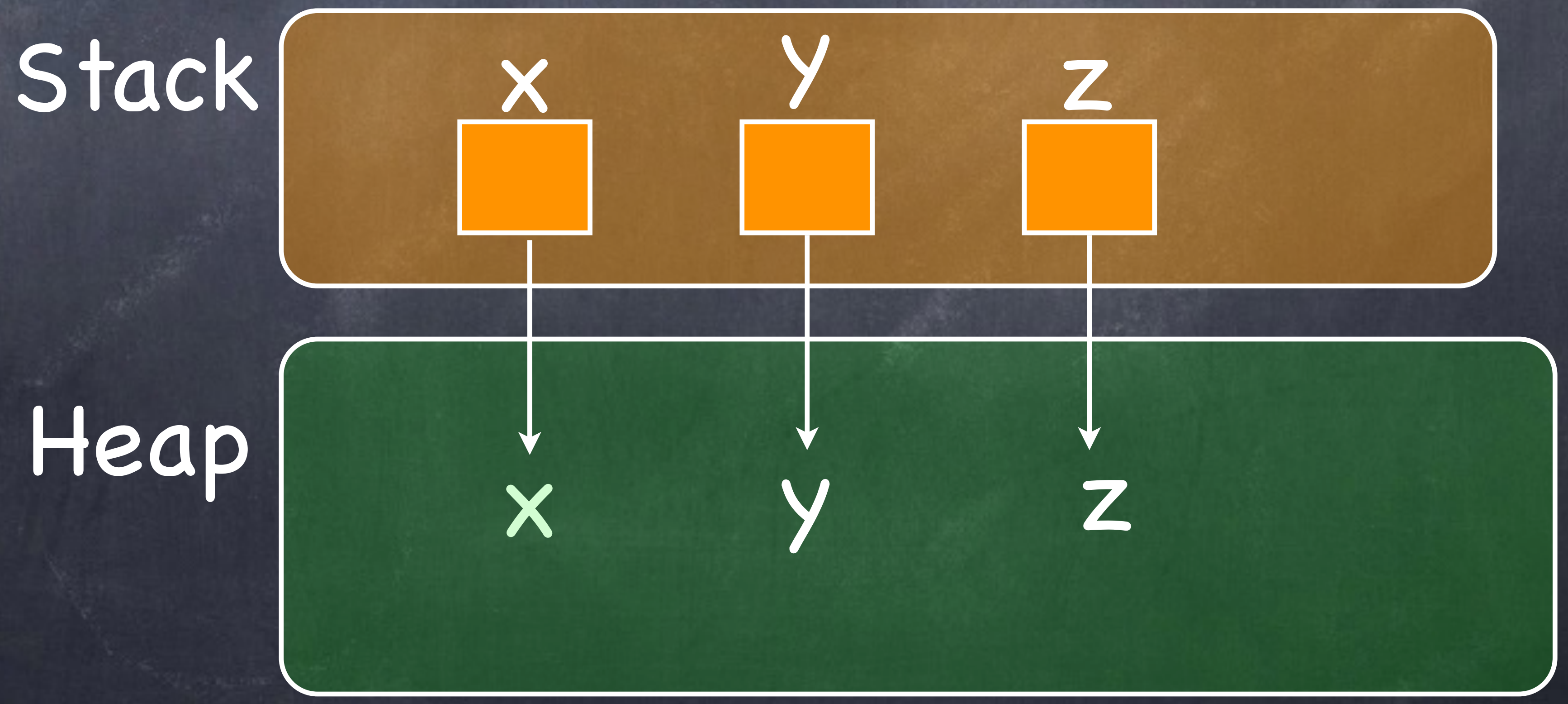
$$
\begin{array}{rcll}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid P * Q & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P & \text{Classical Logic}
\end{array}
$$

Informal Meaning



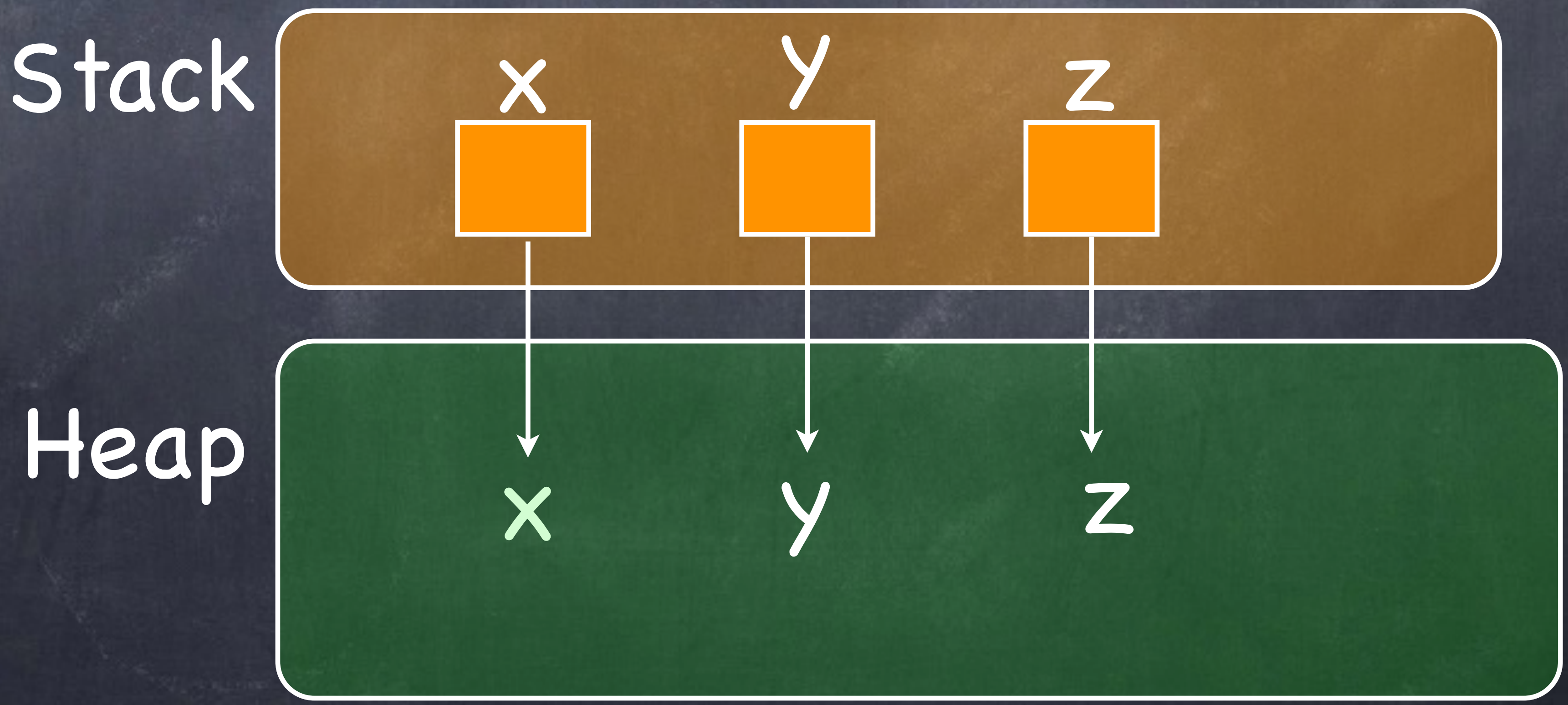
Examples

Formula: emp



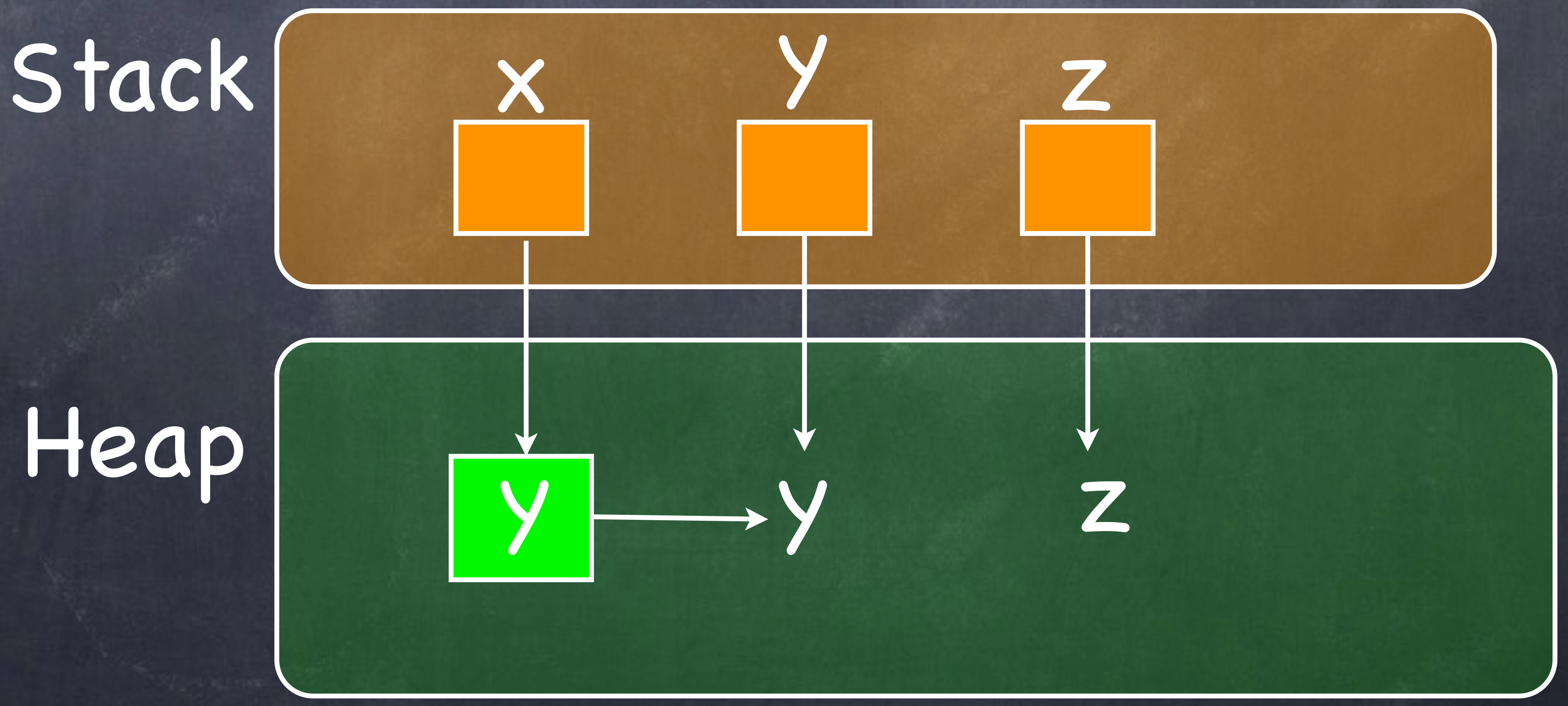
Examples

Formula: emp*x|->y



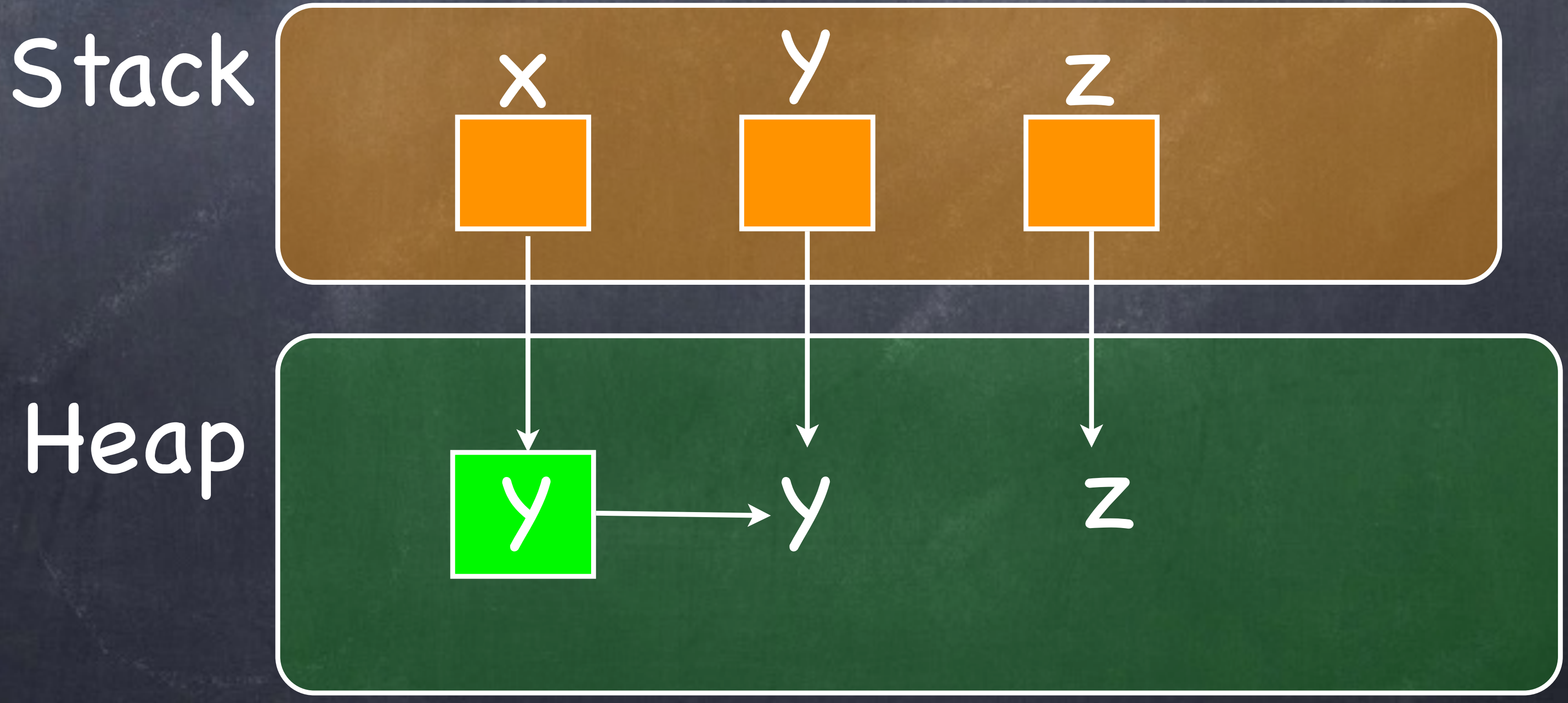
Examples

Formula: emp*x|->y



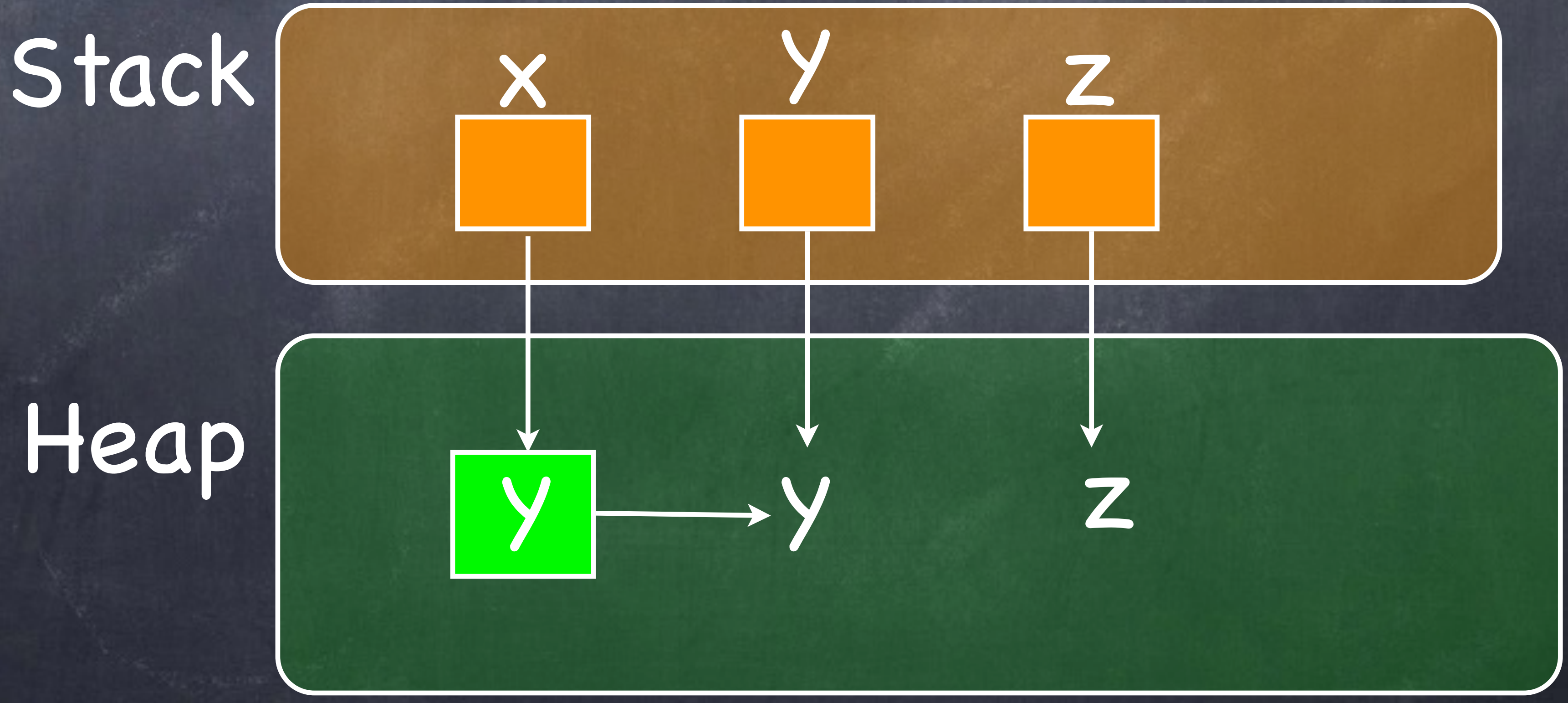
Examples

Formula: x|->y



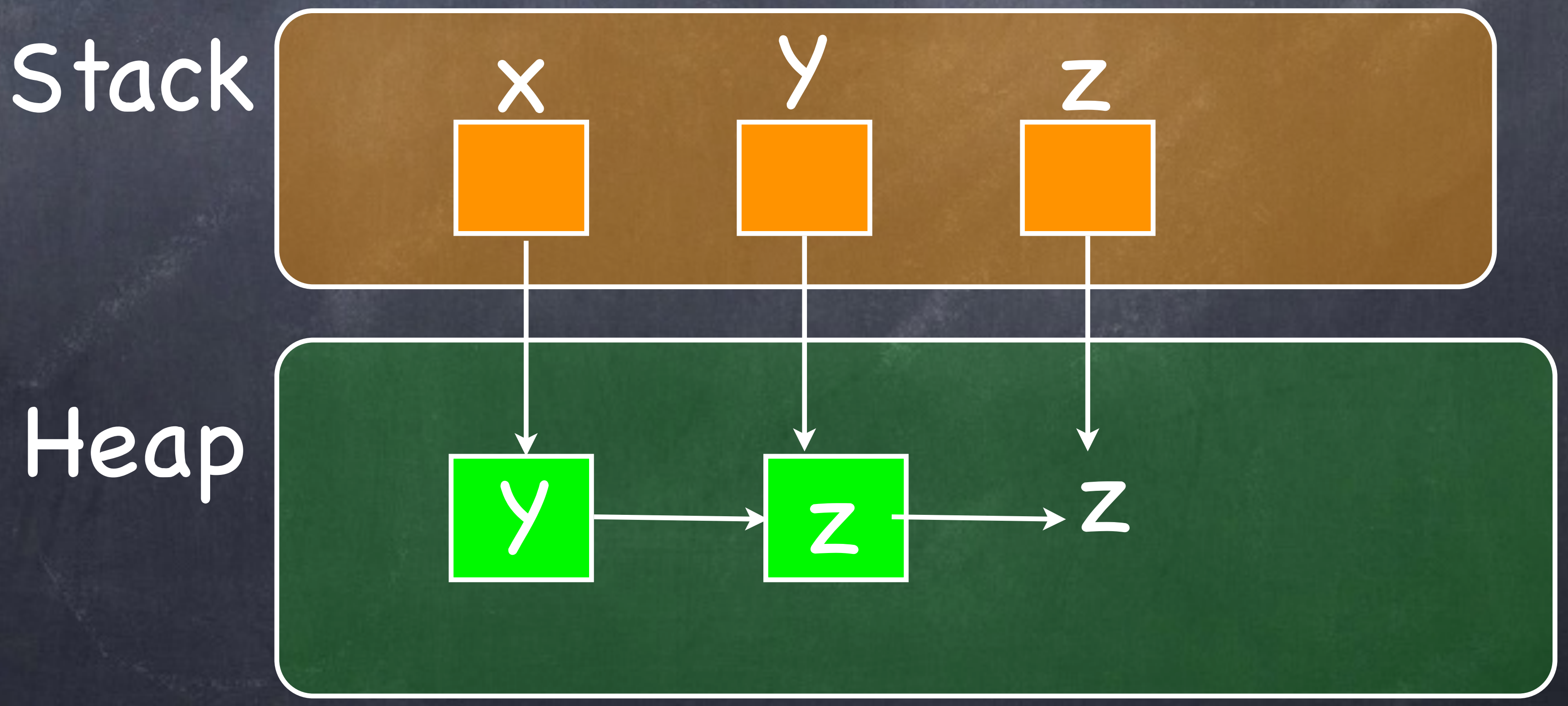
Examples

Formula: x|->y * y|->z



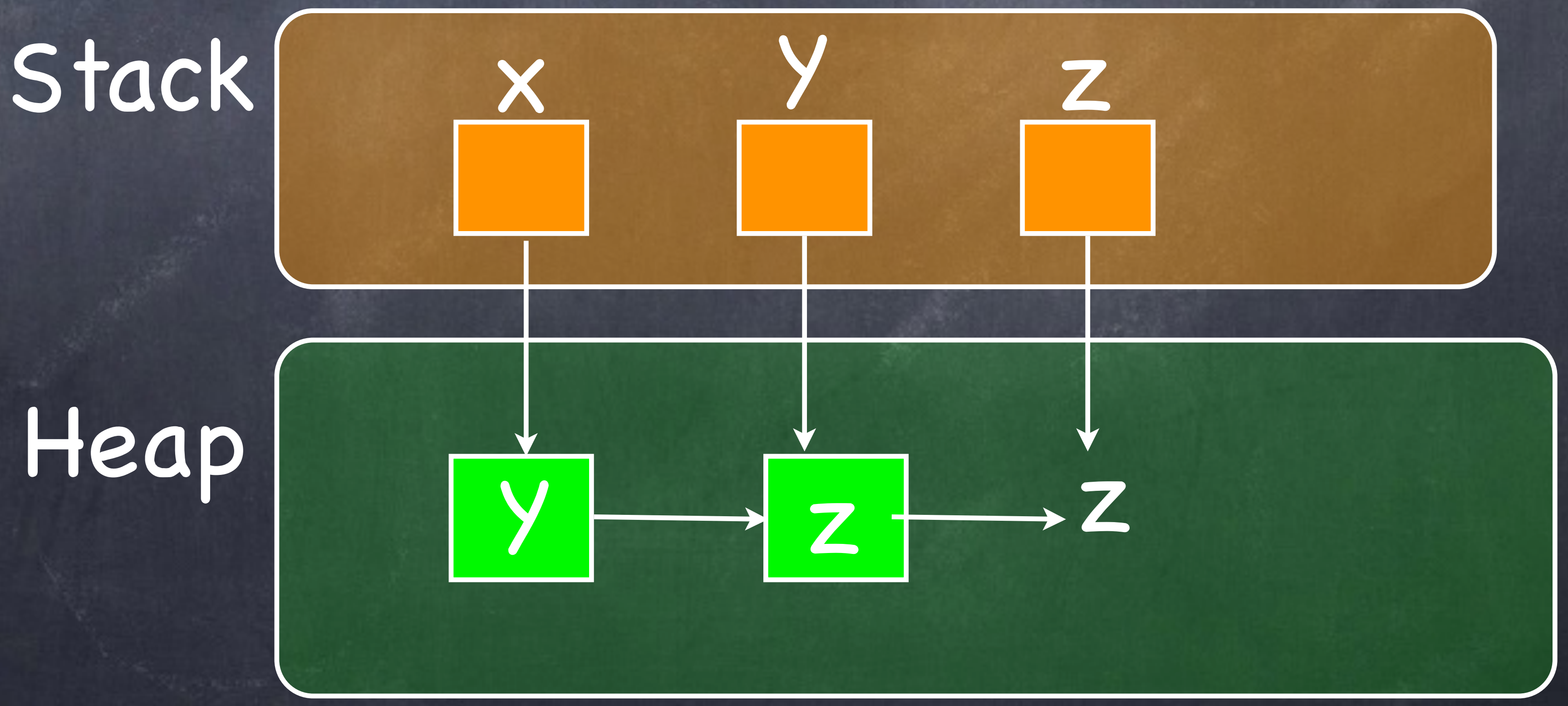
Examples

Formula: x|->y * y|->z



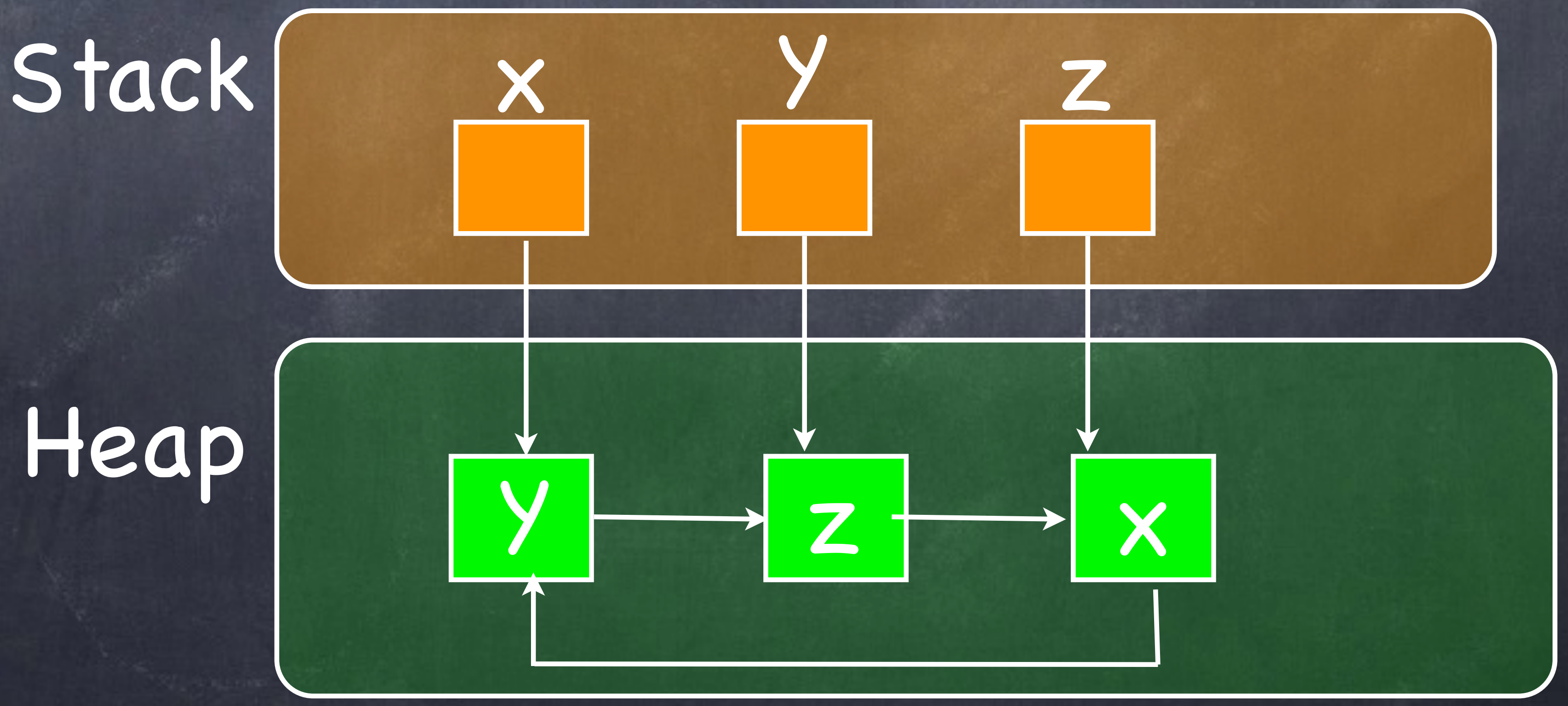
Examples

Formula: x|->y * y|->z* z|->x



Examples

Formula: x|->y * y|->z* z|->x



Semantics of Assertions

- Expressions mean maps from stacks to Vals (integers).

$$[\![E]\!] \quad : \quad \mathsf{Stacks} \longrightarrow \mathsf{Vals}$$

- Semantics of assertions given by satisfaction relation between states and assertions.

$$(s, h) \quad \models \quad P$$

Semantics of Assertions

- Expressions mean maps from stacks to Vals (integers).

$$[\![E]\!] \quad : \quad \mathsf{Stacks} \longrightarrow \mathsf{Vals}$$

- Semantics of assertions given by satisfaction relation between states and assertions.

$$(s, h) \quad \models \quad P$$

Stack

Semantics of Assertions

- Expressions mean maps from stacks to Vals (integers).

$$[\![E]\!] \quad : \quad \mathsf{Stacks} \longrightarrow \mathsf{Vals}$$

- Semantics of assertions given by satisfaction relation between states and assertions.

$$(s, h) \quad \models \quad P$$

Stack

Heap

Semantics of Assertions

$$
\begin{array}{lll}
(s,h) \models E \geq F & \text{iff} & [\![E]\!]s, [\![F]\!]s \in \mathsf{Integers} \text{ and } [\![E]\!]s \geq [\![F]\!]s \\
(s,h) \models E \mapsto F & \text{iff} & \mathsf{dom}(h) = \{[\![E]\!]s\} \;\text{ and }\; h([\![E]\!]s) = [\![F]\!]s \\
(s,h) \models \mathsf{emp} & \text{iff} & h = [] \;\text{ (i.e., } \mathsf{dom}(h) = \emptyset) \\
(s,h) \models P * Q & \text{iff} & \exists h_0 h_1.\; h_0 * h_1 = h,\; (s,h_0) \models P \;\text{ and }\; (s,h_1) \models Q \\
(s,h) \models \mathsf{true} & & \text{always} \\
(s,h) \models P \wedge Q & \text{iff} & (s,h) \models P \;\text{ and }\; (s,h) \models Q \\
(s,h) \models \neg P & \text{iff} & \text{not }\; ((s,h) \models P) \\
(s,h) \models \forall x.\, P & \text{iff} & \forall v \in \mathsf{Vals}.\; (s[x{\mapsto}v], h) \models P)
\end{array}
$$

Abbreviations

The address E is active:

$$E \mapsto - \triangleq \exists x'.E \mapsto x'$$

where x' not free in E

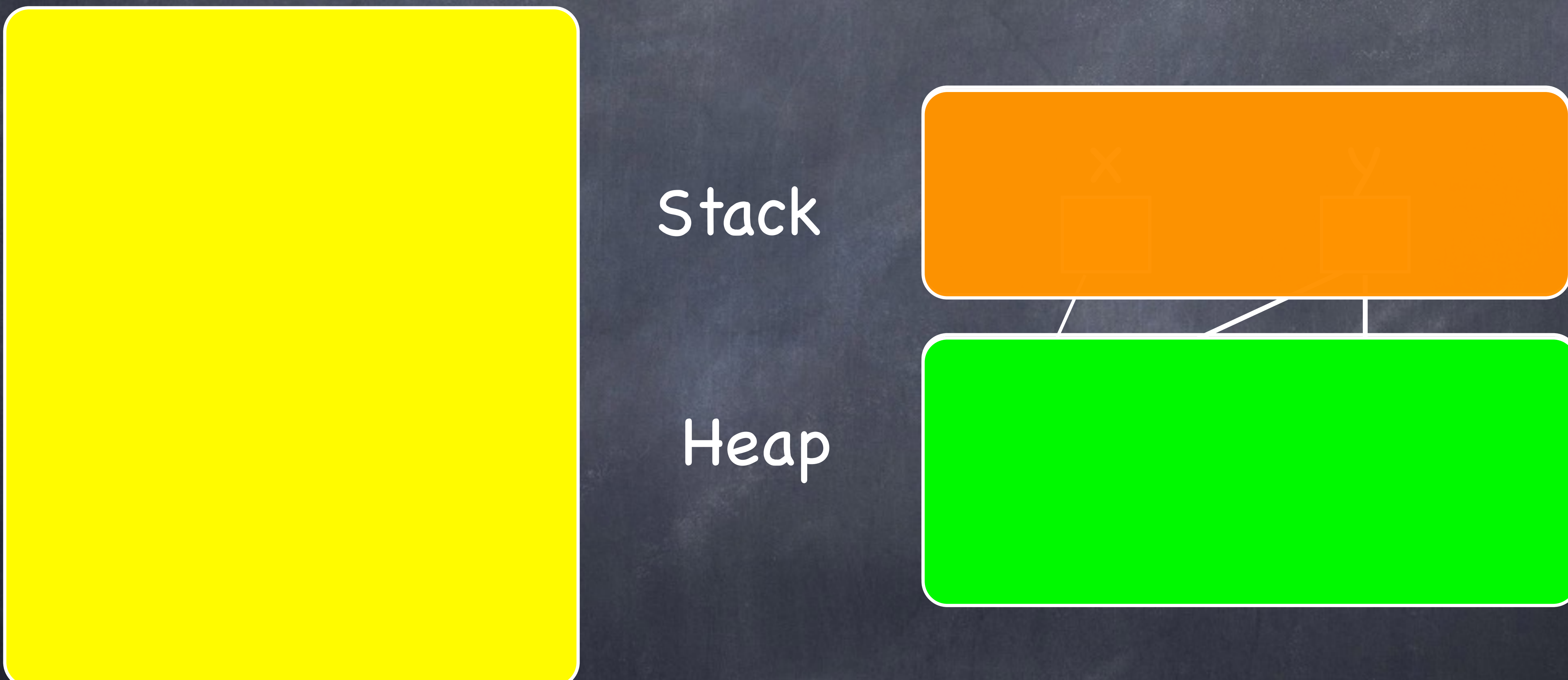
E points to F somewhere in the heap:

$$E \hookrightarrow F \triangleq E \mapsto F * \mathsf{true}$$

E points to a record of several fields:

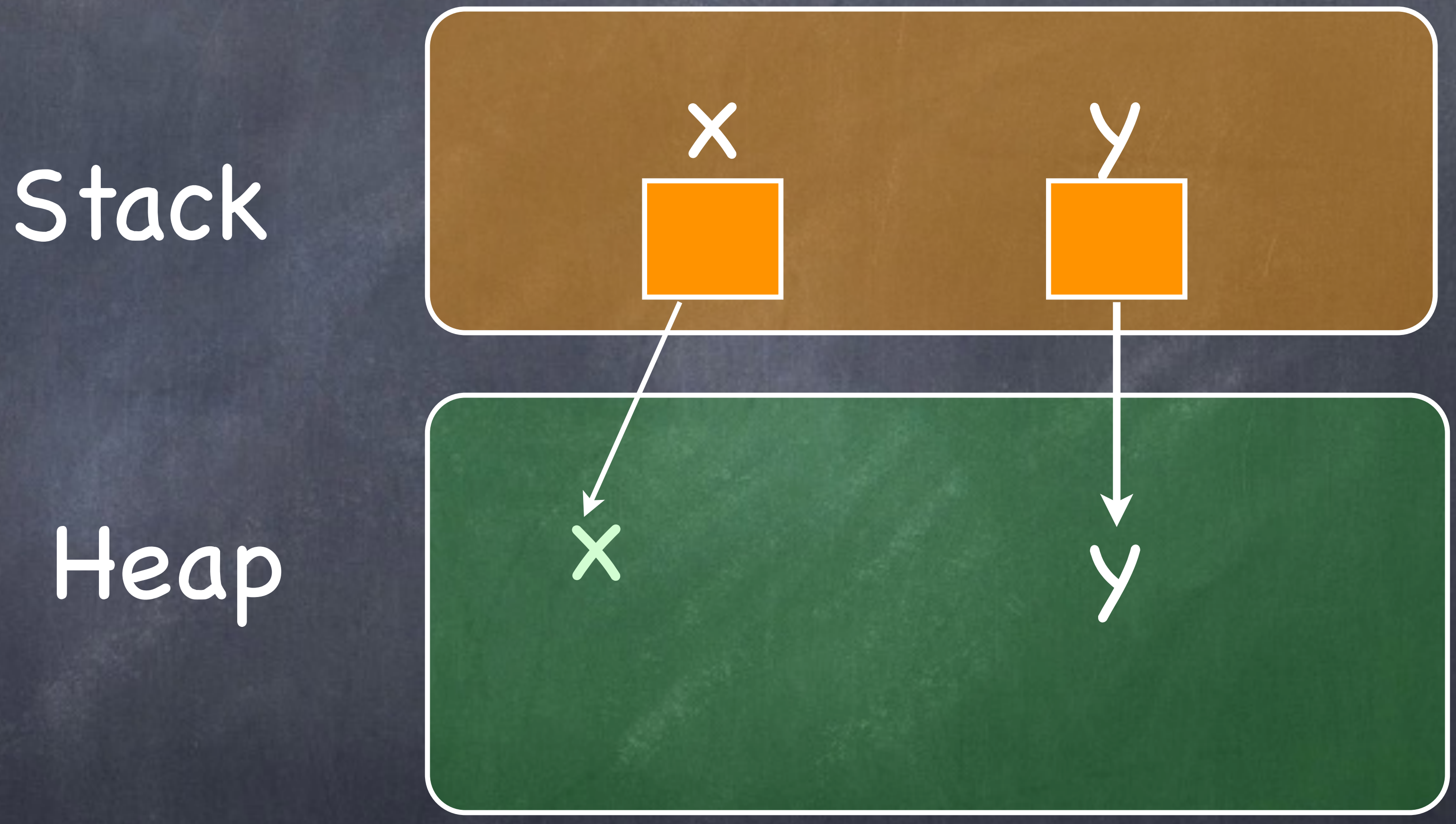
$$E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$$

Example



Example

$x \mapsto 3, y$

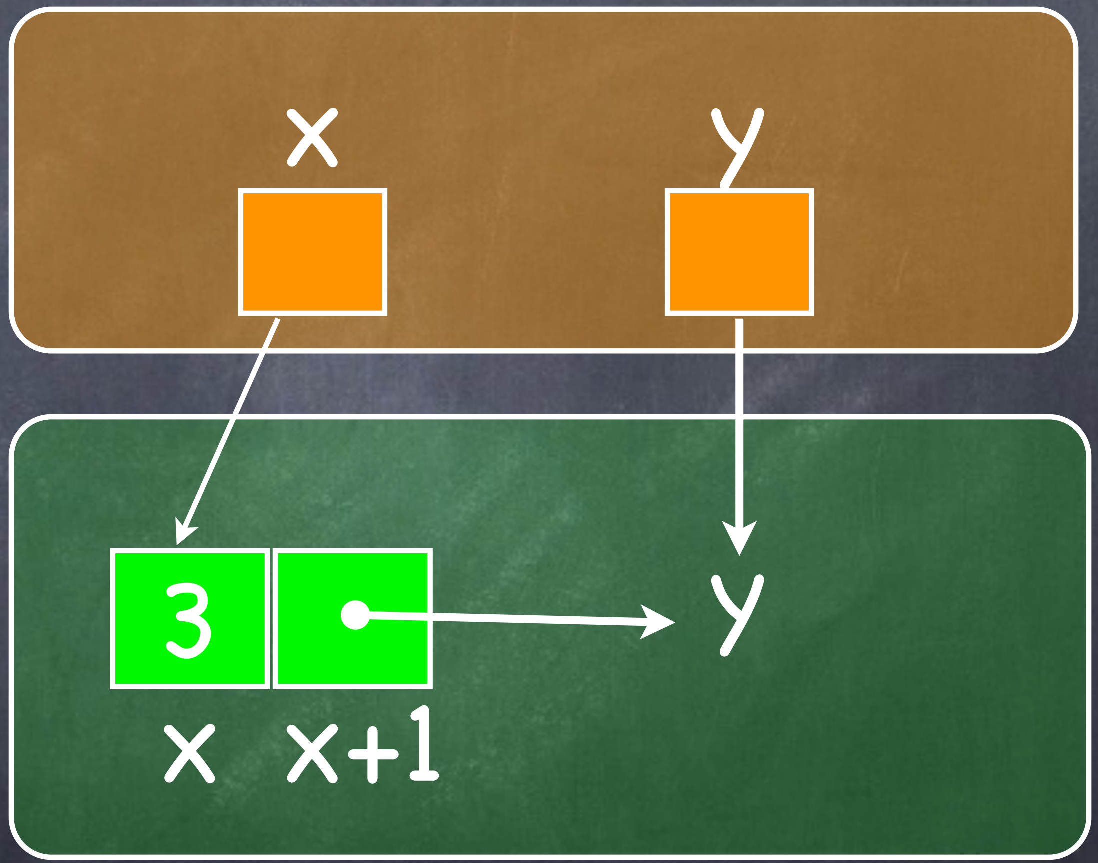


Example

$x \mapsto 3, y$

Stack

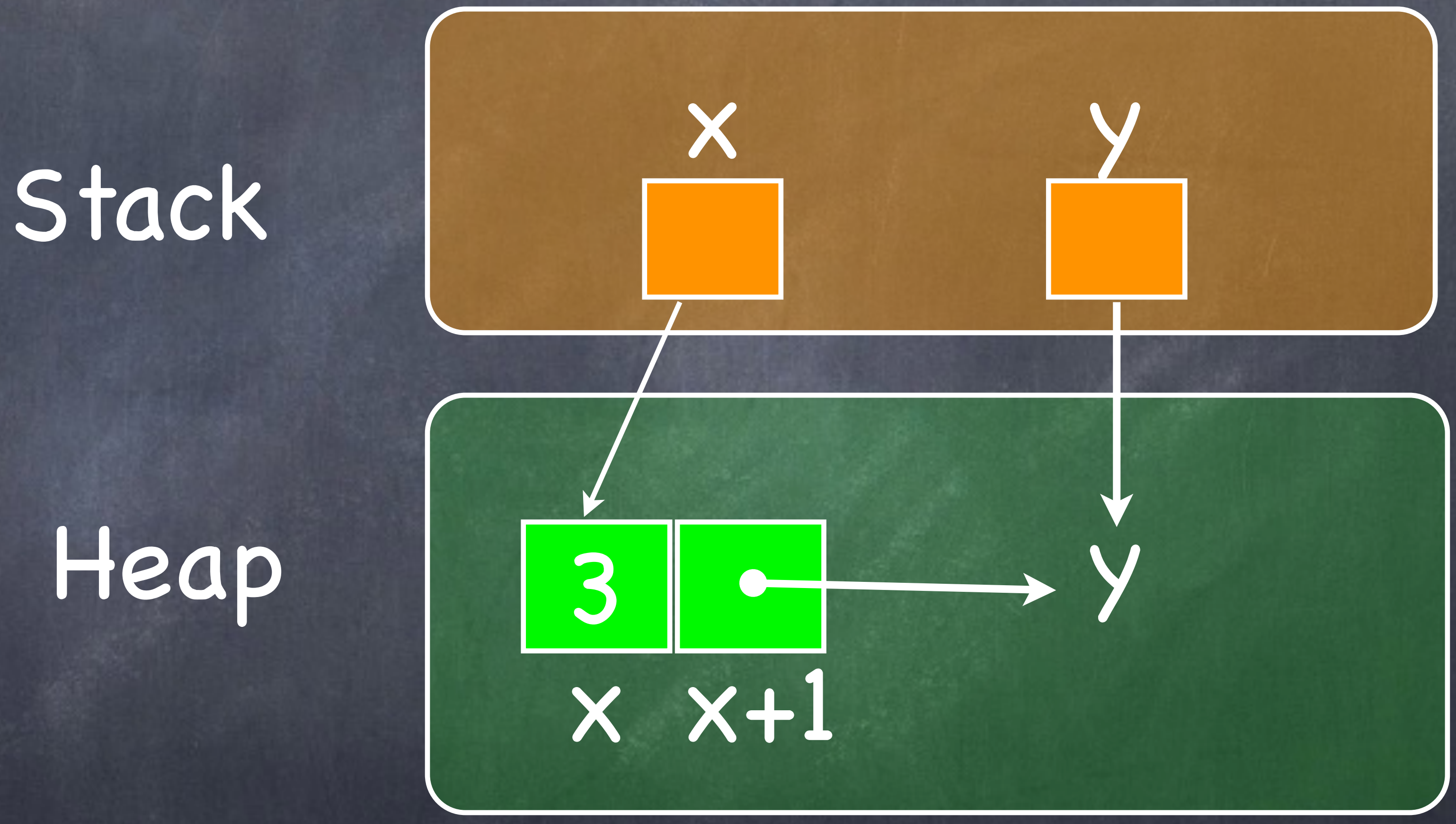
Heap



Example

$x \mapsto 3, y$

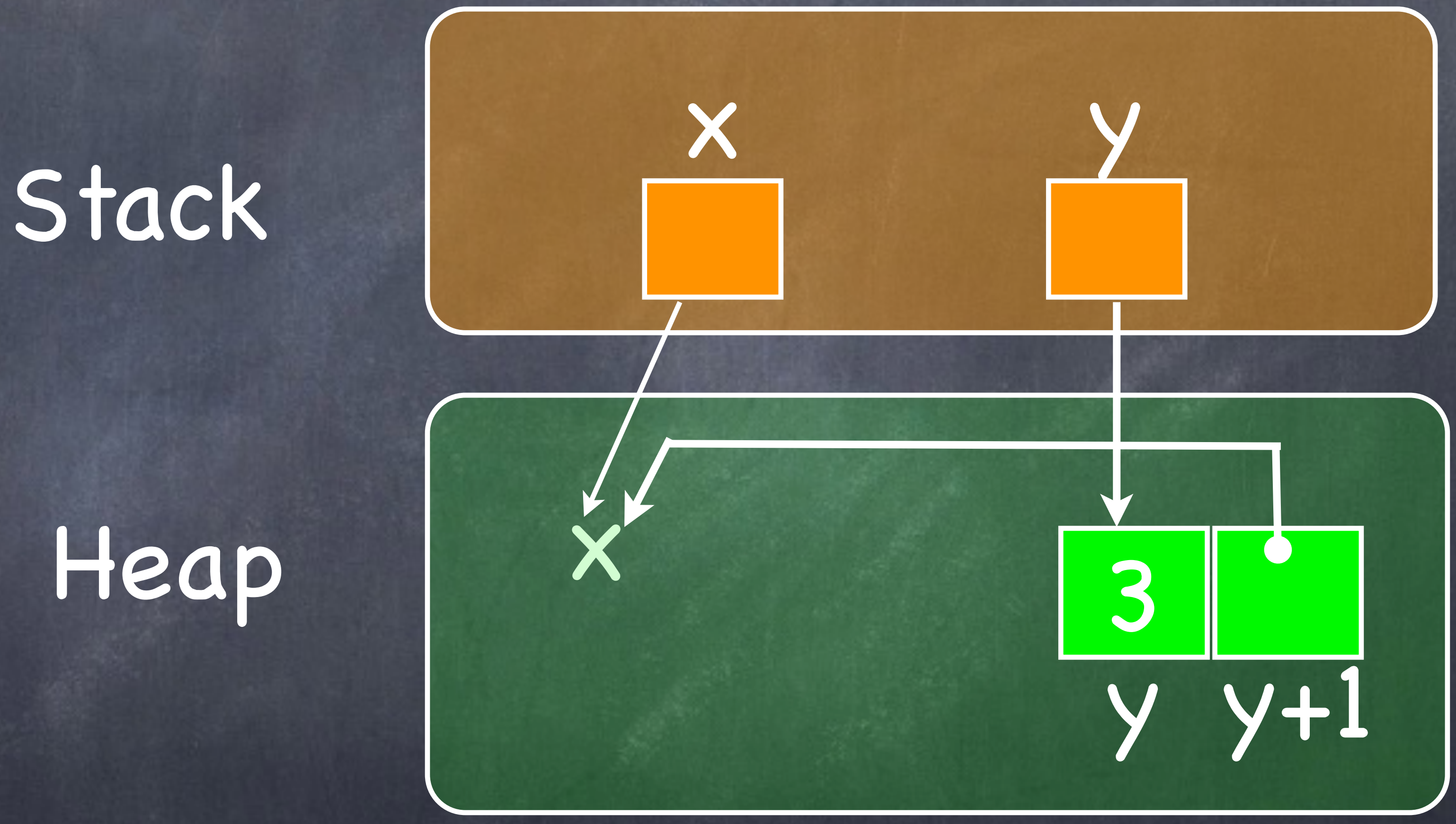
$y \mapsto 3, x$



Example

$x \mapsto 3, y$

$y \mapsto 3, x$

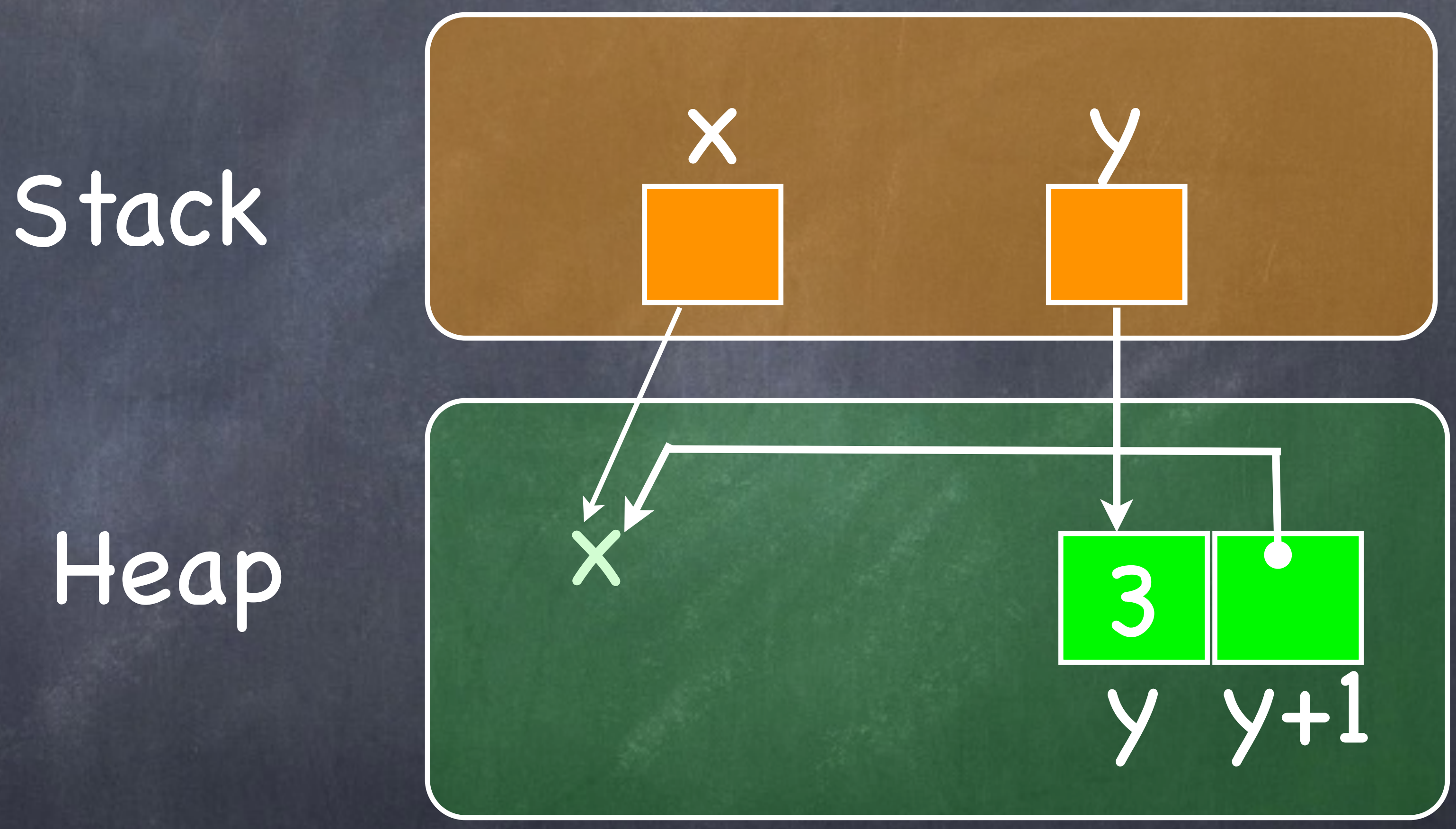


Example

$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

$$x \mapsto 3, y * y \mapsto 3, x$$

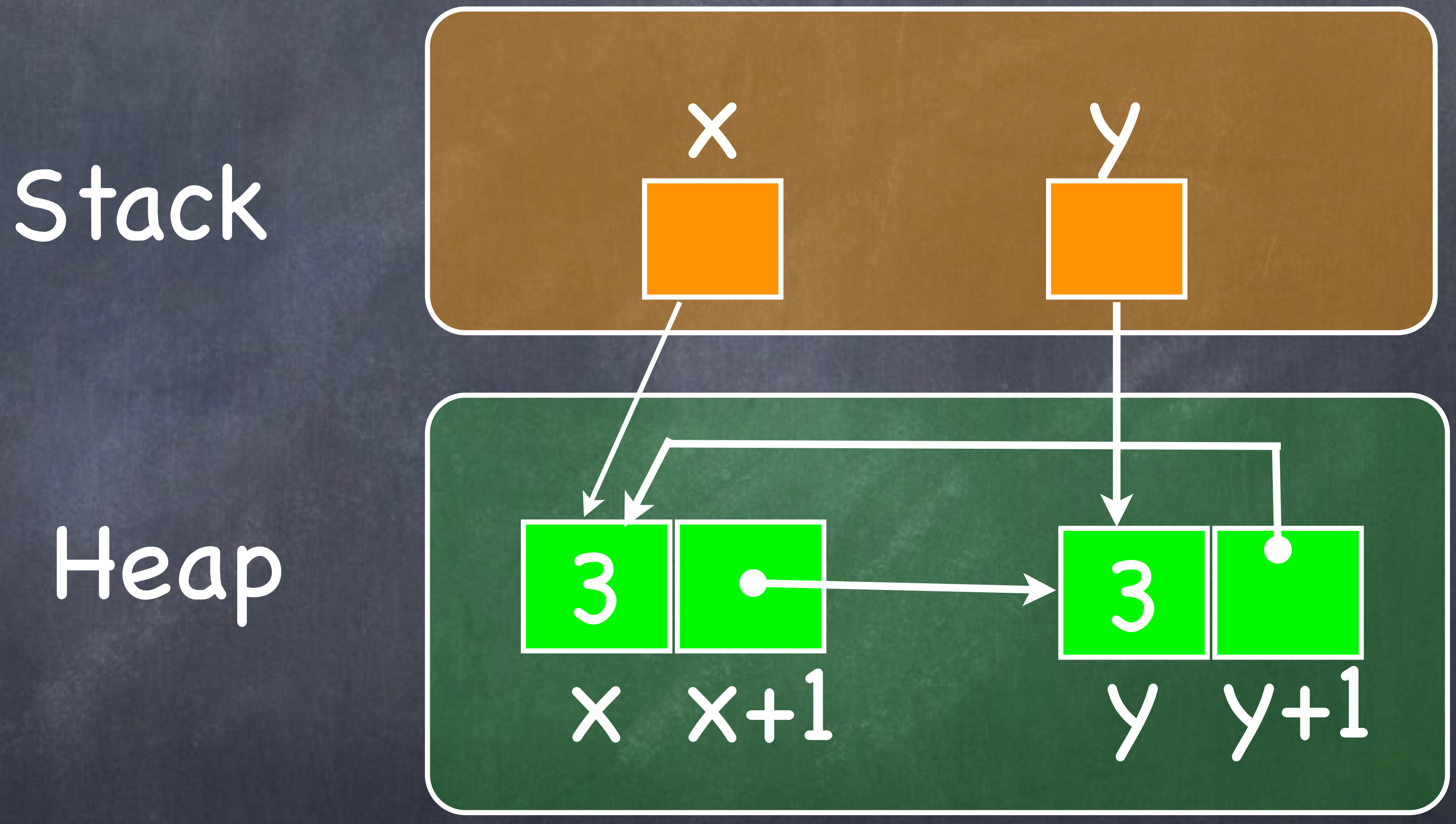


Example

$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

$$x \mapsto 3, y * y \mapsto 3, x$$



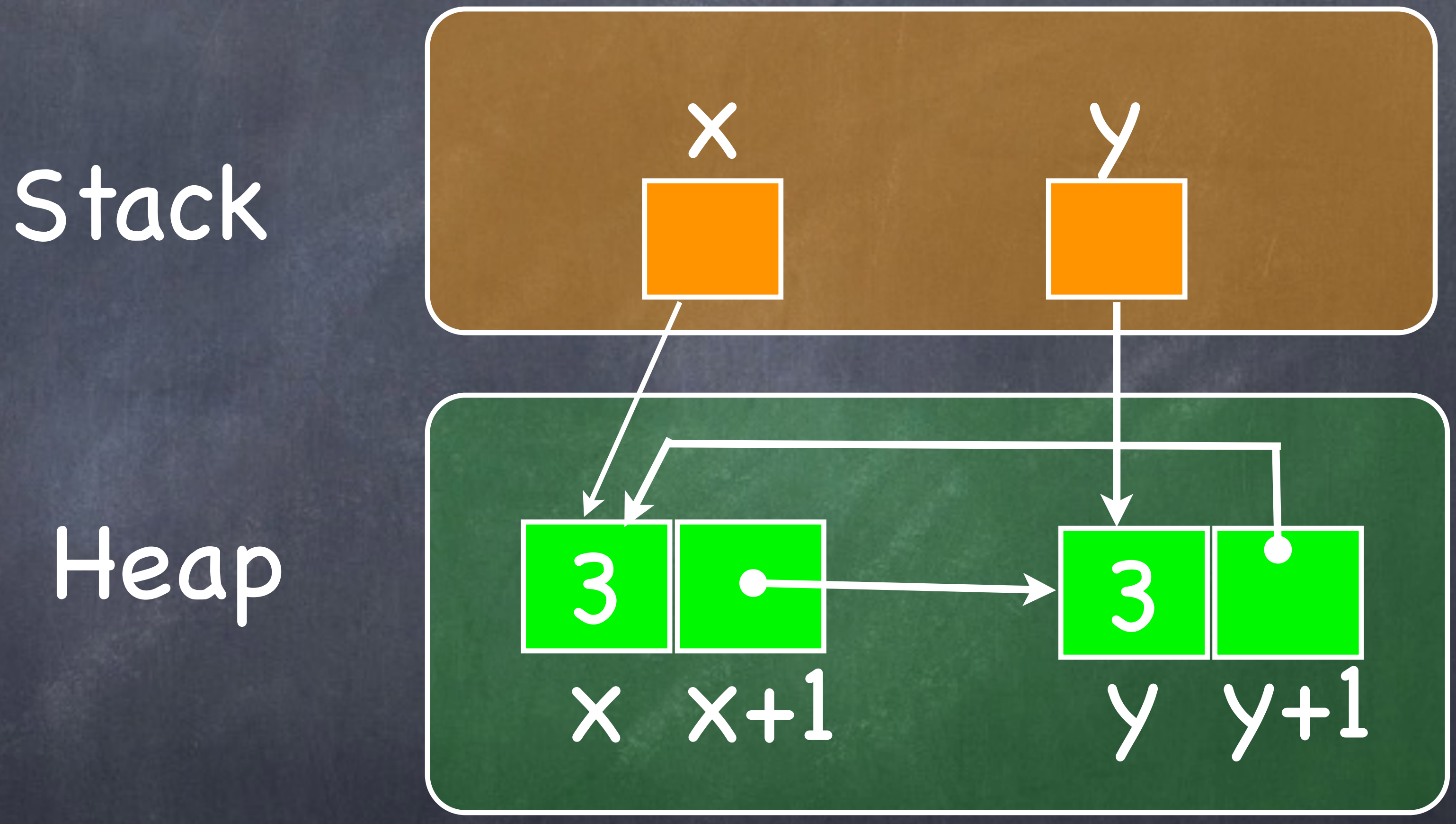
Example

$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

$$x \mapsto 3, y * y \mapsto 3, x$$

$$x \mapsto 3, y \wedge y \mapsto 3, x$$



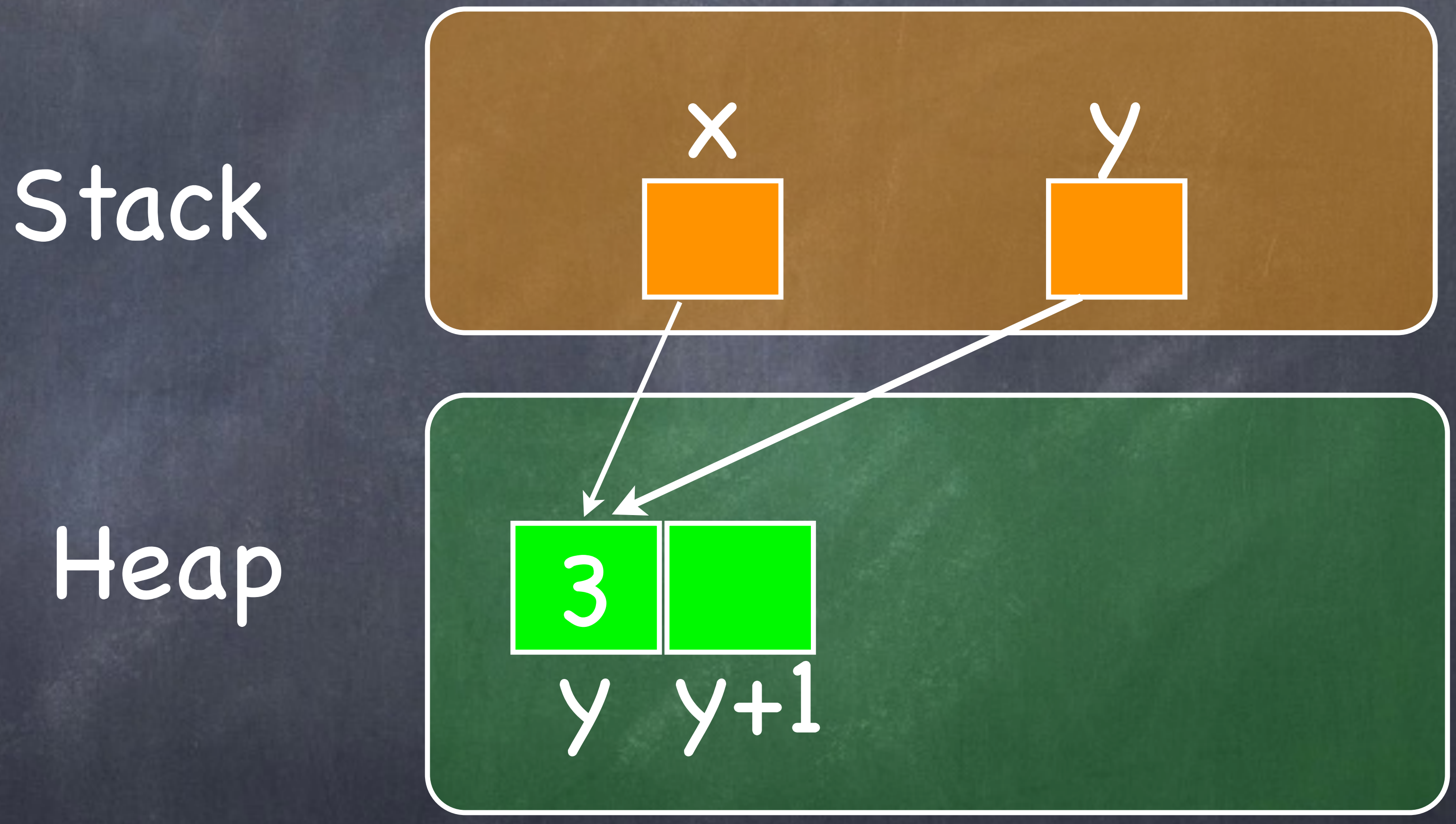
Example

$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

$$x \mapsto 3, y * y \mapsto 3, x$$

$$x \mapsto 3, y \wedge y \mapsto 3, x$$



Example

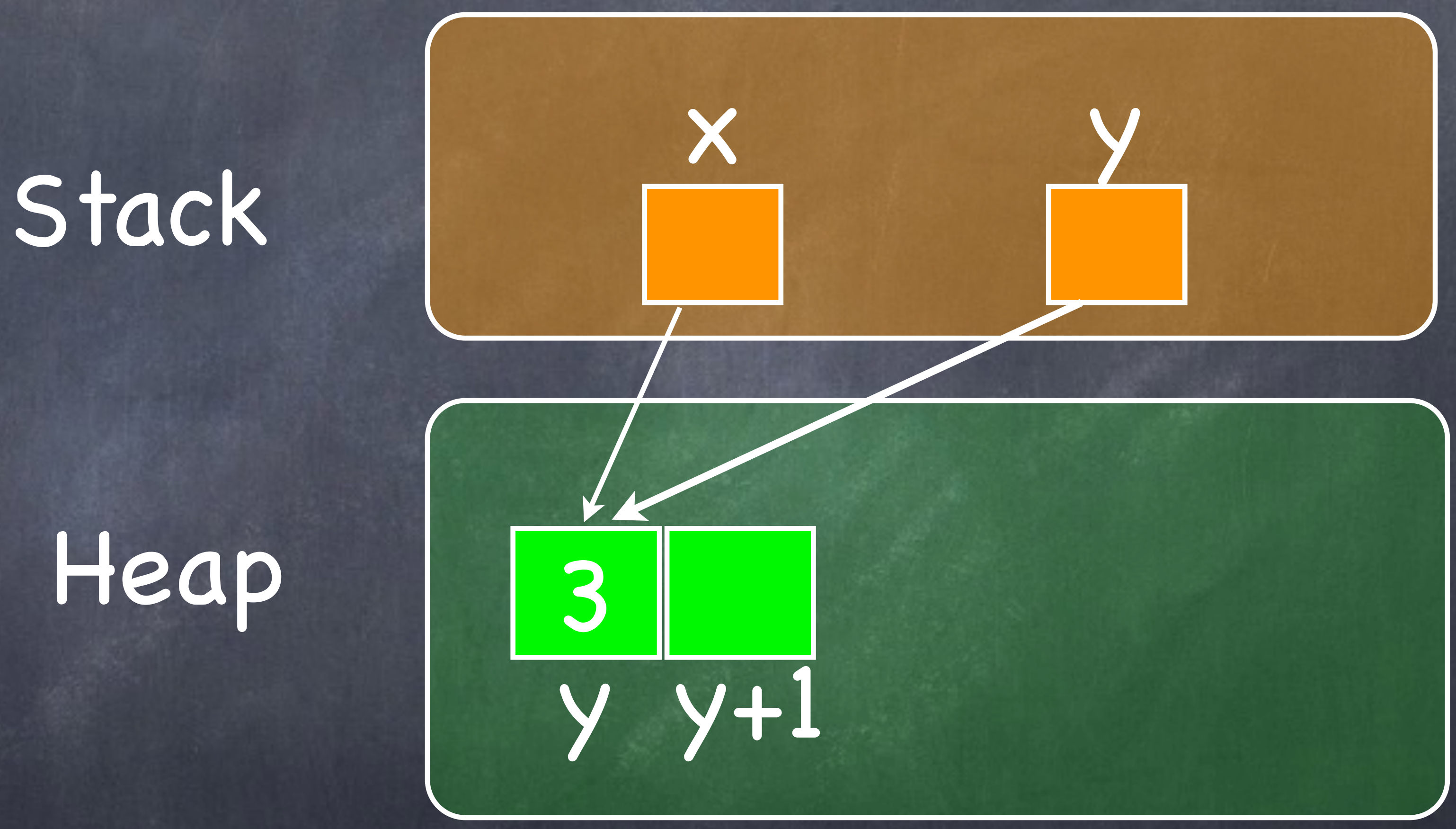
$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

$$x \mapsto 3, y * y \mapsto 3, x$$

$$x \mapsto 3, y \wedge y \mapsto 3, x$$

$$x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$$



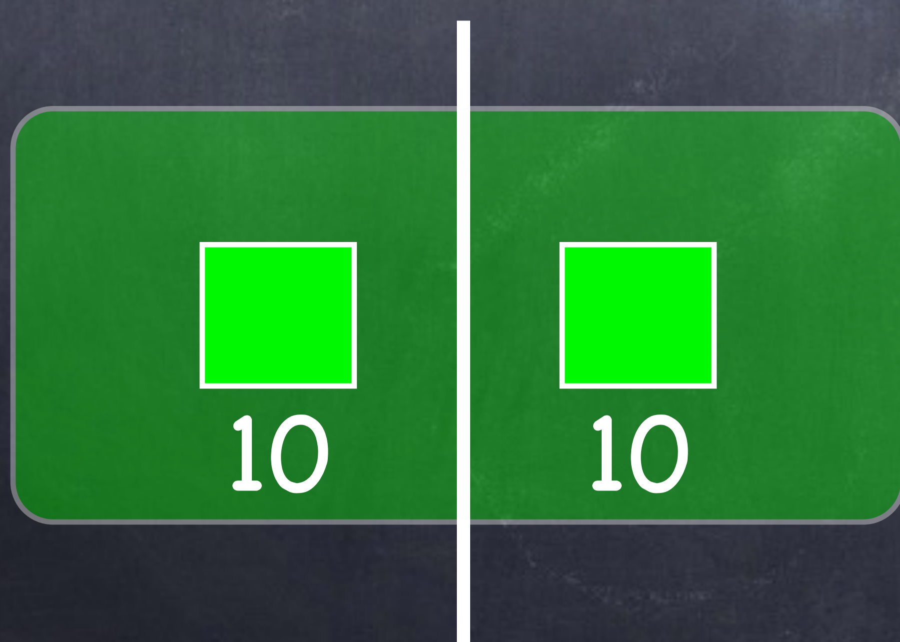
Exercise: what's the last formula asserting?

An inconsistency

- What's wrong with the following formula?
  - 10|->3 * 10|->3

An inconsistency

- What's wrong with the following formula?
  - 10|->3 * 10|->3



Try to be in two places at the same time

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0
  - E|-> - * E|-> -
  - E|-> - * F |-> - => E != F
  - E |-> 3 /\ F |-> 3 => E=F
  - E|->3 * F |->3 => E|->3 /\ F |->3

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0 Valid!
  - E|-> - * E|-> -
  - E|-> - * F |-> - => E != F
  - E |-> 3 /\ F |-> 3 => E=F
  - E|->3 * F |->3 => E|->3 /\ F |->3

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0 Valid!
  - E|-> - * E|-> - Invalid!
  - E|-> - * F |-> - => E != F
  - E |-> 3 /\ F |-> 3 => E=F
  - E|->3 * F |->3 => E|->3 /\ F |->3

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0 Valid!
  - E|-> - * E|-> - Invalid!
  - E|-> - * F |-> - => E != F Valid!
  - E |-> 3 /\ F |-> 3 => E=F
  - E|->3 * F |->3 => E|->3 /\ F |->3

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0 Valid!
  - E|-> - * E|-> - Invalid!
  - E|-> - * F |-> - => E != F Valid!
  - E |-> 3 /\ F |-> 3 => E=F Valid!
  - E|->3 * F |->3 => E|->3 /\ F |->3

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0   Valid!
  - E|-> - * E|-> -   Invalid!
  - E|-> - * F |-> - => E != F   Valid!
  - E |-> 3 /\ F |-> 3 => E=F   Valid!
  - E|->3 * F |->3 => E|->3 /\ F |->3   Invalid!

Substructural logic

- Separation logic is a substructural logic:

No Contraction $A \nvdash A * A$

No Weakening $A * B \nvdash A$

Examples:

$$10 \mapsto 3 \nvdash 10 \mapsto 3 * 10 \mapsto 3$$

$$10 \mapsto 3 * 42 \mapsto 7 \nvdash 42 \mapsto 7$$

Lists

A non circular list can be defined with the following inductive predicate:

list [] i = emp /\ i=nil
list (s::S) i = exists j. i|->s,j * list S j

List segment

Possibly empty list segment

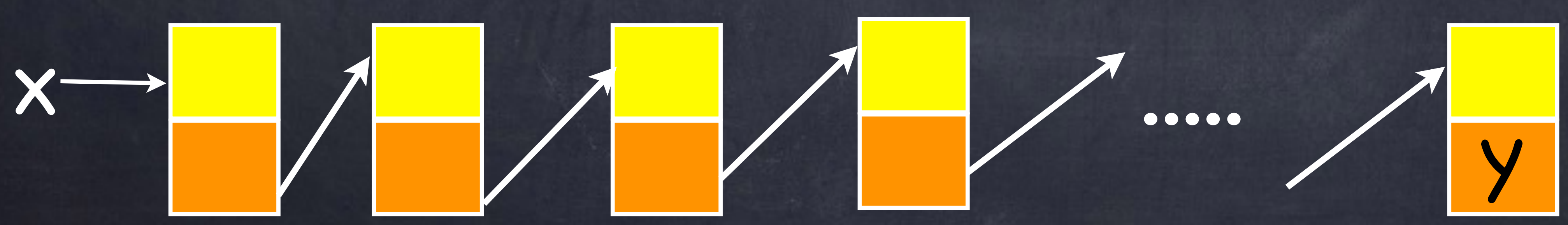
lseg(x,y) = (emp /\ x=y) OR

exists j. x|->j * lseg(j,y)

Non-empty non-circular list segment

lseg(x,y) = x!=y /\

((x|->y) OR exists j. x|->j * lseg(j,y))



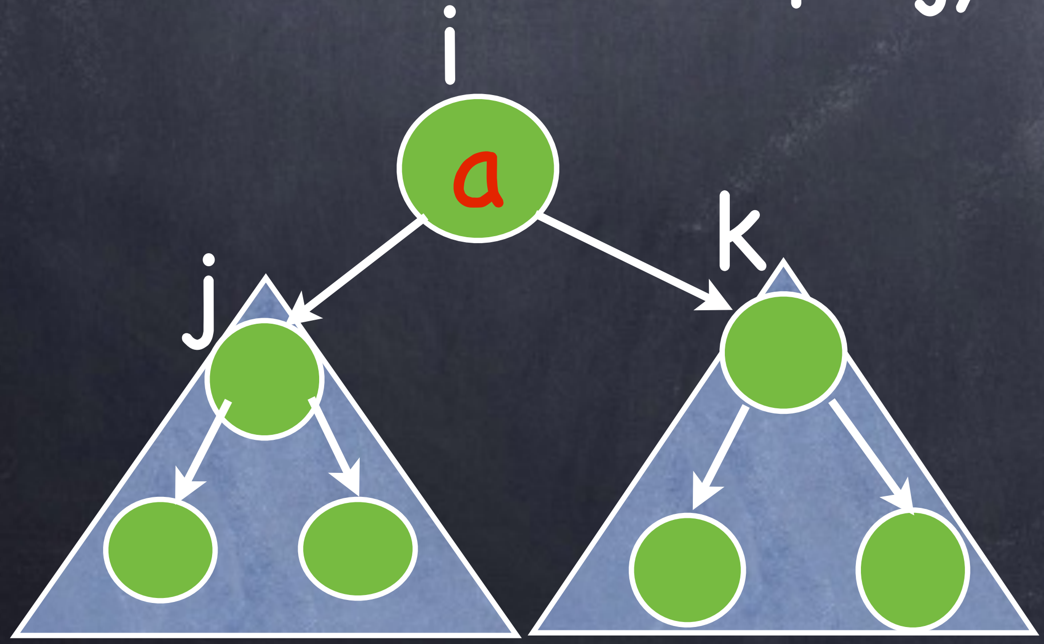
Trees

A tree can be defined with this inductive definition:

tree [] i = emp /\ i=nil

tree (t1,a,t2) i = exists j,k.

i|->j,a,k * (tree t1 j) * (tree t2 k)



Simple Imperative Language

- Safe commands:
    - S::= skip | x:=E | x:=new()
- Heap accessing commands:
    - A(E) ::= dispose(E) | x:=[E] | [E]:=F

where E is and expression x, y, nil, etc.

- Command:
    - C::= S | A | C1;C2 | if B { C1 } else {C2} | while B do { C }

where B boolean guard E=E, E!=E, etc.

Concrete semantics

$$\frac{\mathcal{C}[\![E]\!]s = n}{s, h,\ x := E \implies (s|x \mapsto n), h}$$

$$\frac{\ell \notin dom(h)}{s, h,\ \mathsf{new}(x) \implies (s|x \mapsto \ell), (h|\ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad h(\ell) = n}{s, h,\ x := [E] \implies (s|x \mapsto n), h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell}{s, h * [\ell \mapsto n],\ \mathsf{dispose}(E) \implies s, h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad \mathcal{C}[\![F]\!]s = n \quad \ell \in dom(h)}{s, h,\ [E] := F \implies s, (h|\ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s \notin dom(h)}{s, h,\ A(E) \implies \top}$$

Semantics of Hoare triples

- Partial correctness: {P} C {Q} is valid iff starting from a state s,h |= P, whenever the execution of C terminates in a state (s',h') then s',h'|= Q
- Total correctness: [P] C [Q] is valid iff starting from a state s,h|= P,
  - Every execution terminates
  - when an execution terminates in a state (s',h') then s',h'|= Q.

Semantics of Hoare triples

- Partial correctness: {P} C {Q} is valid iff starting from a state s,h |= P, whenever the execution of C terminates in a state (s',h') then s',h'|= Q
- Total correctness: [P] C [Q] is valid iff starting from a state s,h|= P,
  - Every execution terminates
  - when an execution terminates in a state (s',h') then s',h'|= Q.

Sequential Composition Rule

$$\frac{\{P\}\ C1\ \{P'\} \qquad \{P'\}\ C2\ \{Q\}}{\{P\}\ C1;C2\ \{Q\}}$$

Example:

Sequential Composition Rule

$$\frac{\{P\}\ C1\ \{P'\} \qquad \{P'\}\ C2\ \{Q\}}{\{P\}\ C1;C2\ \{Q\}}$$

Example:

{ y+z>4 } y:=y+z-1; x:=y+2 { x>5 }

Sequential Composition Rule

$$\frac{\{P\}\ C1\ \{P'\} \qquad \{P'\}\ C2\ \{Q\}}{\{P\}\ C1;C2\ \{Q\}}$$

Example:

$$\frac{\{\ y+z>4\ \}\ y:=y+z-1\ \{y > 3\}}{\{\ y+z>4\ \}\ y:=y+z-1;\ x:=y+2\ \{\ x>5\ \}}$$

Sequential Composition Rule

$$\frac{\{P\}\ C1\ \{P'\} \qquad \{P'\}\ C2\ \{Q\}}{\{P\}\ C1;C2\ \{Q\}}$$

Example:

$$\frac{\{\ y+z>4\ \}\ y:=y+z-1\ \{y > 3\} \qquad \{\ y>3\ \}\ x:=y+2\ \{x > 5\}}{\{\ y+z>4\ \}\ y:=y+z-1;\ x:=y+2\ \{\ x>5\ \}}$$

Small Axioms

- { x=m /\ emp } x:=E { x=(E[m/x]) /\ emp}
- { E|->- } [E]:=F { E|->F }
- { x=m /\ E|->n } x:=[E] { x=n /\ E[m/x] |->n }
- { E|->- } dispose(E) {emp}
- { x=m /\ emp} x:=new(E1,...,Ek) { x|->E1[m/x],...,Ek[m/x] }

where x,m,n are assumed to be distinct variables

These axioms mention only the local state which is touched, called footprint

Observation

- A Hoare triple only describes the effect an action has on the portion of program store it explicitly mentions.
- It does not say what cells among those not mentioned remain unchanged.

Observation

- A Hoare triple only describes the effect an action has on the portion of program store it explicitly mentions.
- It does not say what cells among those not mentioned remain unchanged.

We want instead to say:

any state alteration not explicitly required by the specification is excluded

Idea: focus on footprint

- Change the interpretation of the Hoare triple {P} C {Q}, so that C must only dereference cells guaranteed to exists by P or allocated by C itself
- Add an inference rule to obtain bigger specifications from small ones.

Idea: focus on footprint

The portion of memory touched by a command

- Change the interpretation of the Hoare triple {P} C {Q}, so that C must only dereference cells guaranteed to exists by P or allocated by C itself
- Add an inference rule to obtain bigger specifications from small ones.

Memory faults

- Some commands can "go wrong" for example:
  - dispose(x) or [x]:=y or x:=[y]
- Examples:

```
x=new();
y:=x;
dispose(x);
[y]:=nil;
```

Memory faults

- Some commands can "go wrong" for example:
  - dispose(x) or [x]:=y or x:=[y]
- Examples:

```
x:=new();
y:=x;
dispose(x);
[y]:=nil;
```

Tight Interpretation of Triples

- The interpretation of the triples in separation logic ensures that a program does not fault!

$$\{P\}\,C\,\{Q\} \text{ holds} \quad \text{iff} \quad \forall s,h.\ \text{if } s,h \models P \text{ then}$$
$$\neg C,s,h \rightarrow^* \mathsf{err}$$
$$\text{and, if } C,s,h \rightarrow^* s',h' \text{ then } s',h' \models Q$$

This ensure that a well-specified programs access only the cells guaranteed to exist in the precondition or created by C

Aliasing and Soundness

- In traditional Floyd-Hoare logic, the rule of constancy:

$$\frac{\{P\}\,C\,\{Q\}}{\{P \wedge R\}\,C\,\{Q \wedge R\}}\ \mathsf{Modify(C)} \cap \mathsf{Free(R)} = \emptyset$$

allows modular reasoning for sequential as well as parallel programs.

Aliasing and Soundness

- In traditional Floyd-Hoare logic, the rule of constancy:

$$\frac{\{P\}\, C\, \{Q\}}{\{P \wedge R\}\, C\, \{Q \wedge R\}}\ \mathsf{Modify(C)} \cap \mathsf{Free(R)} = \emptyset$$

allows modular reasoning for sequential as well as parallel programs.

This rule is unsound in presence of pointers

Aliasing and Soundness

- In traditional Floyd-Hoare logic, the rule of constancy:

$$\frac{\{P\}\,C\,\{Q\}}{\{P \wedge R\}\,C\,\{Q \wedge R\}}\ \mathsf{Modify(C)} \cap \mathsf{Free(R)} = \emptyset$$

allows modular reasoning for sequential as well as parallel programs.

This rule is unsound in presence of pointers

$$\frac{\{\ [x]=3\ \}\ [x]:=7\ \{\ [x]=7\ \}}{\{\ [x]=3\ /\backslash\ [y]=3\ \}\ [x]:=7\ \{\ [x]=7\ /\backslash\ [y]=3\}}$$

Frame Rule

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} \; \mathsf{Modifies}(C) \cap \mathsf{FV}(R) = \emptyset$$

R is the frame (it can be added as invariant)

* and err-avoiding triple take care of the heap access of C

The side condition takes care of the stack access

Note:
Modify(x:=E)=Modify(x:=[E])=Modify(x:=new(E1,..,Ek))={x} and
Modify([E]:=F)=Modify(dispose(E))={}

Example using the Frame Rule

{x|->-} [x]:=z {x|->z}

{ y|->c * x|->-} [x]:=3 {x|->z * y|->c }

Example

Let's assume:

{ x|->1,2 } C { z|-> 3,2 }

and C modifies only the heap.

Example

Let's assume:

{ x|->1,2 } C { z|-> 3,2 }

and C modifies only the heap.

If we give C more heap

{ x|->1,2 * y|->17,42} C { z|-> 3,2* ?????? }

Example

Let's assume:

{ x|->1,2 } C { z|-> 3,2 }

and C modifies only the heap.

If we give C more heap

{ x|->1,2 * y|->17,42} C { z|-> 3,2* y|->17,42 }

Example

Let's assume:

{ x|->1,2 } C { z|-> 3,2 }

and C modifies only the heap.

If we give C more heap

{ x|->1,2 * y|->17,42} C { z|-> 3,2* y|->17,42 }

We are sure that cell y cannot change otherwise we would have a fault and it would contradict the initial assumption where y is dangling

Proving a program

```
x = new(3,3);

y = new(4,4);

[x+1] = y;

[y+1] = x;

dispose x;
```

Proving a program

{exists n,m. x=n /\ y=m /\ emp}

```
x = new(3,3);

y = new(4,4);

[x+1] = y;

[y+1] = x;

dispose x;
```

Proving a program

```
{exists n,m. x=n /\ y=m /\ emp}
    x = new(3,3);
   {x|->3,3}
    y = new(4,4);

    [x+1] = y;

    [y+1] = x;

    dispose x;
```

Proving a program

```
{exists n,m. x=n /\ y=m /\ emp}
      x = new(3,3);
    {x|->3,3}
      y = new(4,4);
  {x|->3,3* y|->4,4}
      [x+1] = y;

      [y+1] = x;

      dispose x;
```

Proving a program

{exists n,m. x=n /\ y=m /\ emp}

```
x = new(3,3);
```

{x|->3,3}

```
y = new(4,4);
```

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

```
[x+1] = y;
```

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

```
[y+1] = x;


dispose x;
```

Proving a program

```
{exists n,m. x=n /\ y=m /\ emp}
      x = new(3,3);
   {x|->3,3}
      y = new(4,4);
  {x|->3,3* y|->4,4}
      [x+1] = y;
  {x|->3,y* y|->4,4}
      [y+1] = x;
  {x|->3,y* y|->4,x}
      dispose x;
```

Proving a program

{exists n,m. x=n /\ y=m /\ emp}

```
x = new(3,3);
```

{x|->3,3}

```
y = new(4,4);
```

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

```
[x+1] = y;
```

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

```
[y+1] = x;
```

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

```
dispose x;
```

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

Symbolic Execution

Symbolic Heaps

Symbolic Heaps $\Pi \wedge \Sigma$

Expressions $E \quad ::= \quad x \mid x' \mid \mathsf{nil}$

Pure Formulae

$$\Pi \quad ::= \quad \mathsf{true} \mid E{=}E \mid E \neq E \mid \Pi \wedge \Pi$$

Spatial Formulae

$$\Sigma \quad ::= \quad \mathsf{emp} \mid E \mapsto F \mid \mathsf{junk} \mid \mathsf{ls}\,(E, F) \mid \Sigma * \Sigma$$

Note: primed variable are existentially quantified

What can we express?

We can express:

```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

Shape properties: e.g.

Does a program preserve acyclicity/ciclicity?

Does is core dump?

Does is create garbage?

but not: the order of the element has been reversed

What can we express?

We can express:

```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

Shape properties: e.g.

Does a program preserve acyclicity/ciclicity?

Does is core dump?

Does is create garbage?

but not: the order of the element has been reversed

What can we express?

We can express:

```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

Shape properties: e.g.

Does a program preserve acyclicity/ciclicity?

Does is core dump?

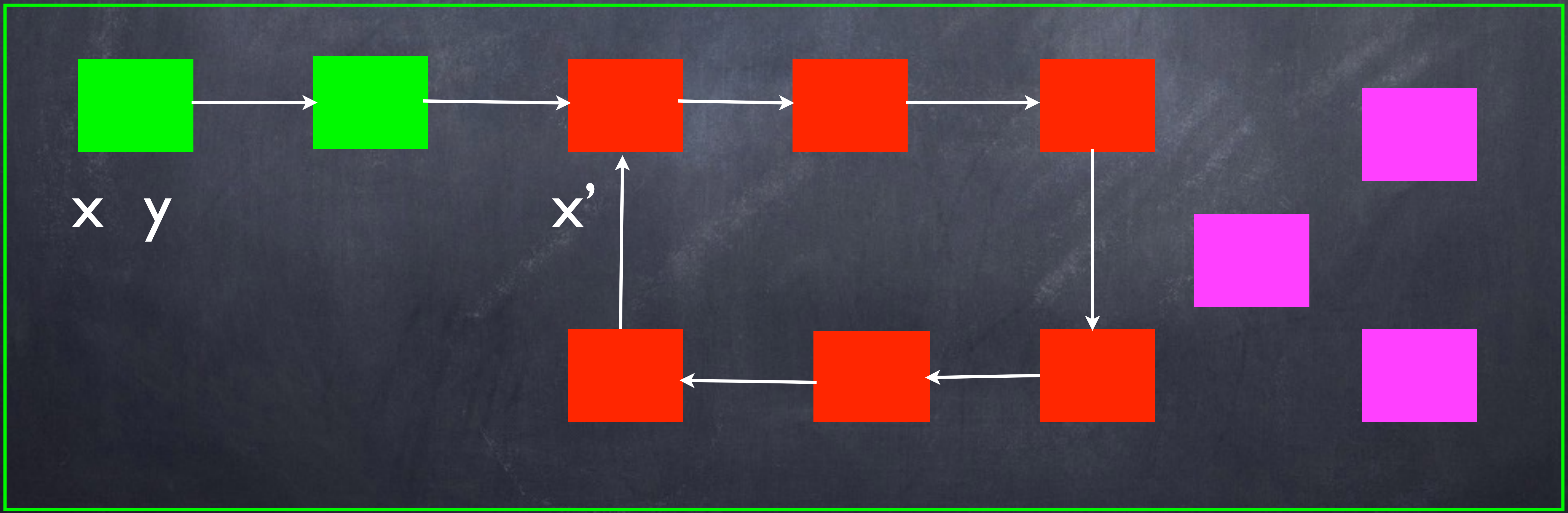
Does is create garbage?

but not: the order of the element has been reversed

Examples

x and y are aliases and they point to a pan-handle list and there is garbage

$$x = y \wedge \mathsf{ls}(x, x') * \mathsf{ls}(x', x') * \mathsf{junk}$$



Examples

$$z = \mathsf{nil} \wedge \mathsf{ls}(x, x') * \mathsf{ls}(y, x') * \mathsf{ls}(\mathsf{x}', \mathsf{nil})$$

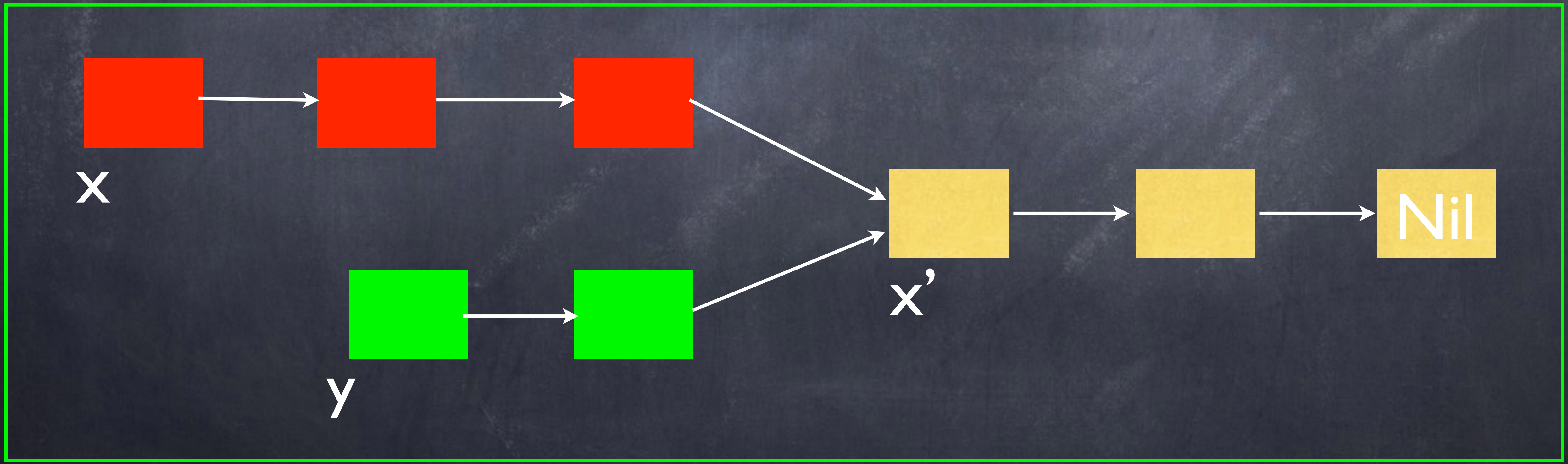
Which kind of heap does it describe?

Examples

$$z = \mathsf{nil} \wedge \mathsf{ls}(x, x') * \mathsf{ls}(y, x') * \mathsf{ls}(\mathsf{x}', \mathsf{nil})$$

Which kind of heap does it describe?

z is nil whereas x and y point to disjoint lists sharing the tail



Symbolic Execution

- Symbolic execution executes the effect of a statement on a symbolic heap
- The result of the modification is another heap or the error state (or T).
- Defined with a relation:

$$\Pi|\Sigma,\ C\ \Longrightarrow \Pi'|\Sigma'$$

Rule of Symbolic Execution

$$
\begin{array}{llll}
\Pi|\Sigma, & x := E & \Longrightarrow & x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Longrightarrow & x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Longrightarrow & \Pi|\Sigma * E \mapsto G \\
\Pi;\Sigma, & \mathsf{new}(x) & \Longrightarrow & (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Longrightarrow & \Pi|\Sigma
\end{array}
$$

$$
\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \ \Longrightarrow \top}
$$

x',y' fresh existentially quantified variables

Soundness

- Is this symbolic semantics sound?
- Is which sense it is sounds?
- We need to show that the symbolic semantics describe an superset of all possible computation of the program (i.e., it is an over-approximation)

Concrete semantics

$$\frac{\mathcal{C}[\![E]\!]s = n}{s, h,\ x := E \implies (s|x \mapsto n), h}$$

$$\frac{\ell \notin dom(h)}{s, h,\ \mathsf{new}(x) \implies (s|x \mapsto \ell), (h|\ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad h(\ell) = n}{s, h,\ x := [E] \implies (s|x \mapsto n), h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell}{s, h * [\ell \mapsto n],\ \mathsf{dispose}(E) \implies s, h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad \mathcal{C}[\![F]\!]s = n \quad \ell \in dom(h)}{s, h,\ [E] := F \implies s, (h|\ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s \notin dom(h)}{s, h,\ A(E) \implies \top}$$

Theorem

The symbolic semantics is a sound over-approximation of the concrete semantics.

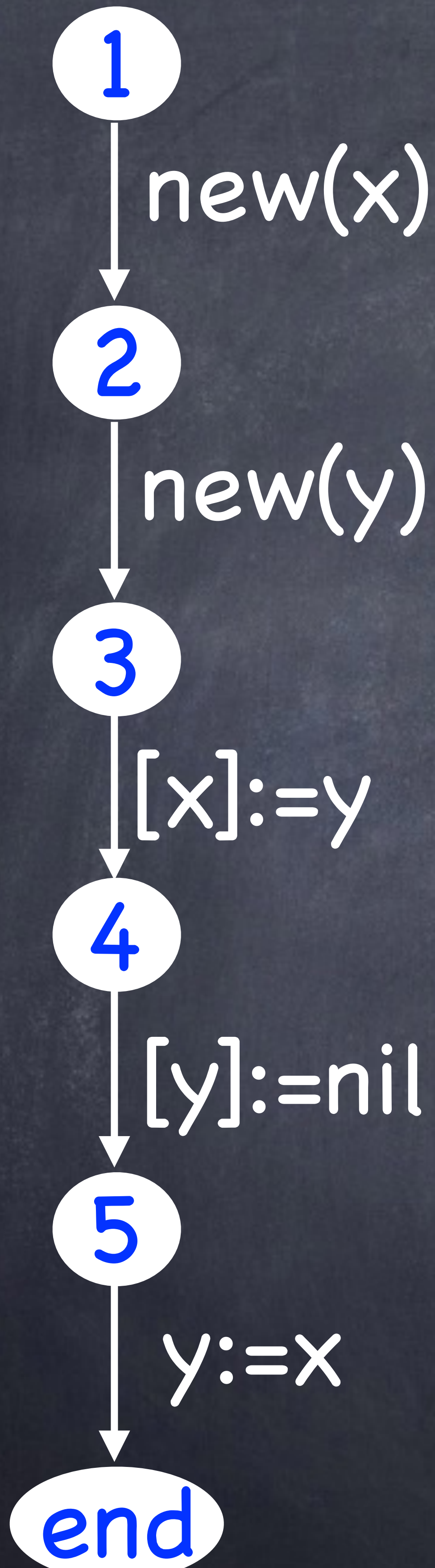
Example 1

$$
\begin{array}{llcl}
\Pi|\Sigma, & x := E & \Longrightarrow & x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Longrightarrow & x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Longrightarrow & \Pi|\Sigma * E \mapsto G \\
\Pi; \Sigma, & \mathsf{new}(x) & \Longrightarrow & (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Longrightarrow & \Pi|\Sigma
\end{array}
$$

$$
\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E)\ \Longrightarrow \top}
$$

new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;

Example 1

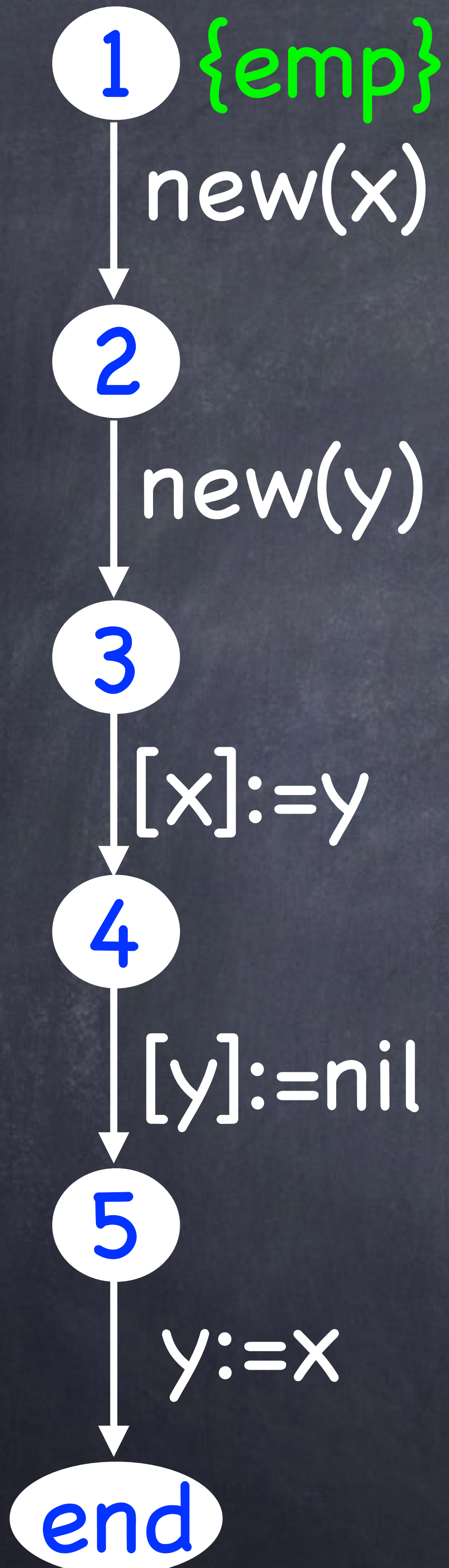


$$
\begin{array}{llcl}
\Pi|\Sigma, & x := E & \Longrightarrow & x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Longrightarrow & x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Longrightarrow & \Pi|\Sigma * E \mapsto G \\
\Pi; \Sigma, & \mathsf{new}(x) & \Longrightarrow & (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Longrightarrow & \Pi|\Sigma
\end{array}
$$

$$
\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \ \Longrightarrow \top}
$$

```
new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;
```

Example 1

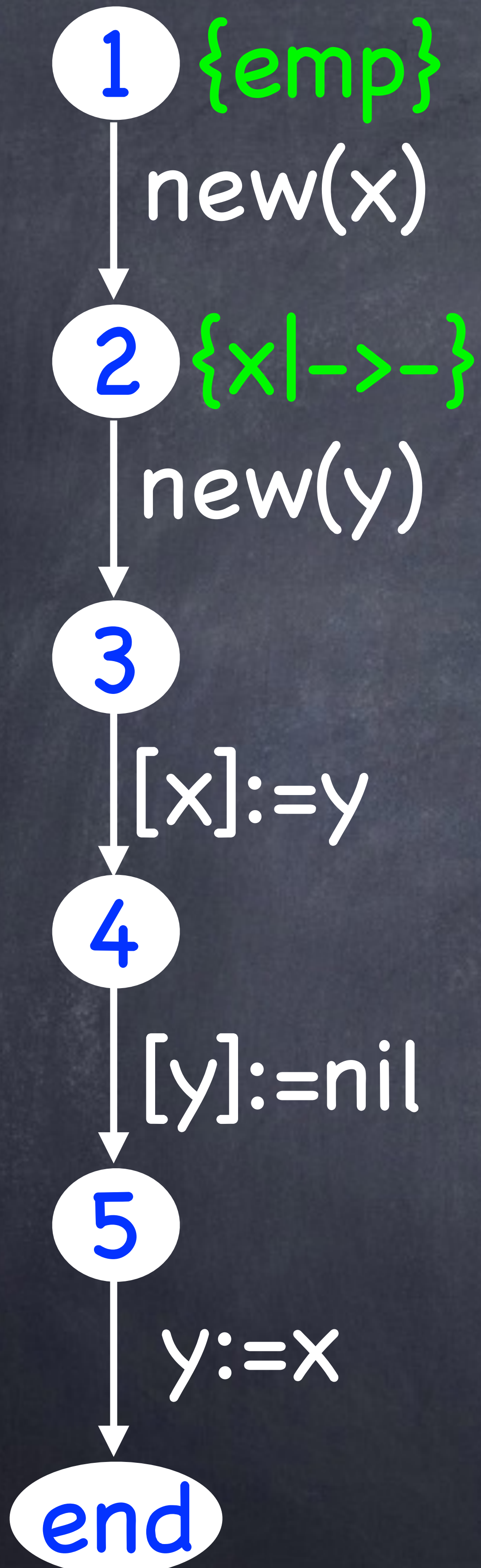


$\Pi \vert \Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi \vert \Sigma)[x'/x]$
$\Pi \vert \Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi \vert \Sigma * E \mapsto F)[x'/x]$
$\Pi \vert \Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi \vert \Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi \vert \Sigma)[x'/x] * x \mapsto y'$
$\Pi \vert \Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi \vert \Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \Longrightarrow \top}$$

new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;

Example 1



$$\Pi|\Sigma, \quad x := E \quad \Longrightarrow \quad x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x]$$

$$\Pi|\Sigma * E \mapsto F, \quad x := [E] \quad \Longrightarrow \quad x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x]$$

$$\Pi|\Sigma * E \mapsto F, \quad [E] := G \quad \Longrightarrow \quad \Pi|\Sigma * E \mapsto G$$

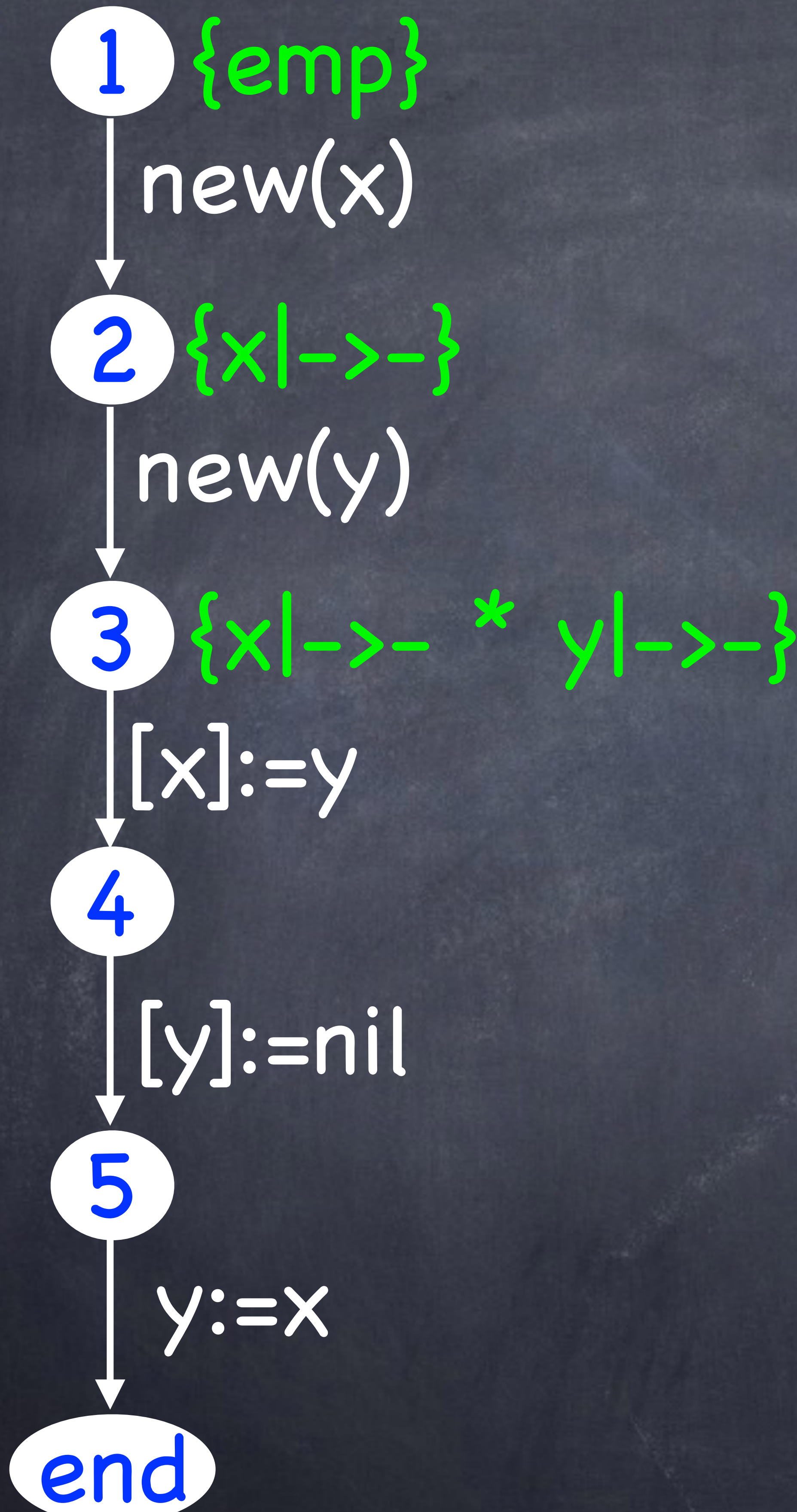
$$\Pi;\Sigma, \quad \mathsf{new}(x) \quad \Longrightarrow \quad (\Pi|\Sigma)[x'/x] * x \mapsto y'$$

$$\Pi|\Sigma * E \mapsto F, \quad \mathsf{dispose}(E) \quad \Longrightarrow \quad \Pi|\Sigma$$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma, \ A(E) \ \Longrightarrow \top}$$

```
new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;
```

Example 1

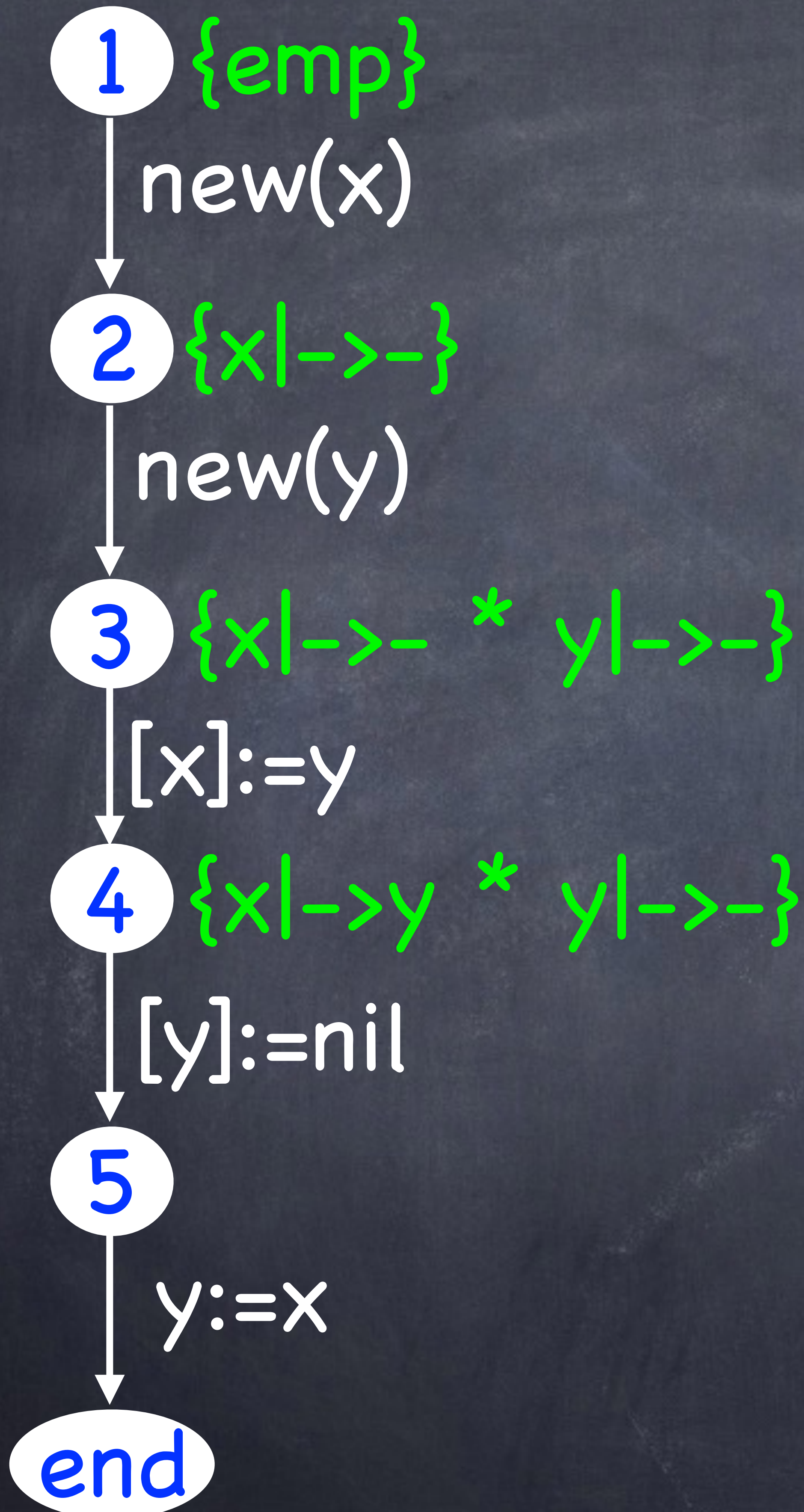


$\Pi\vert\Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi\vert\Sigma)[x'/x]$
$\Pi\vert\Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi\vert\Sigma * E \mapsto F)[x'/x]$
$\Pi\vert\Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi\vert\Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi\vert\Sigma)[x'/x] * x \mapsto y'$
$\Pi\vert\Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi\vert\Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \ \Longrightarrow \top}$$

```
new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;
```

Example 1

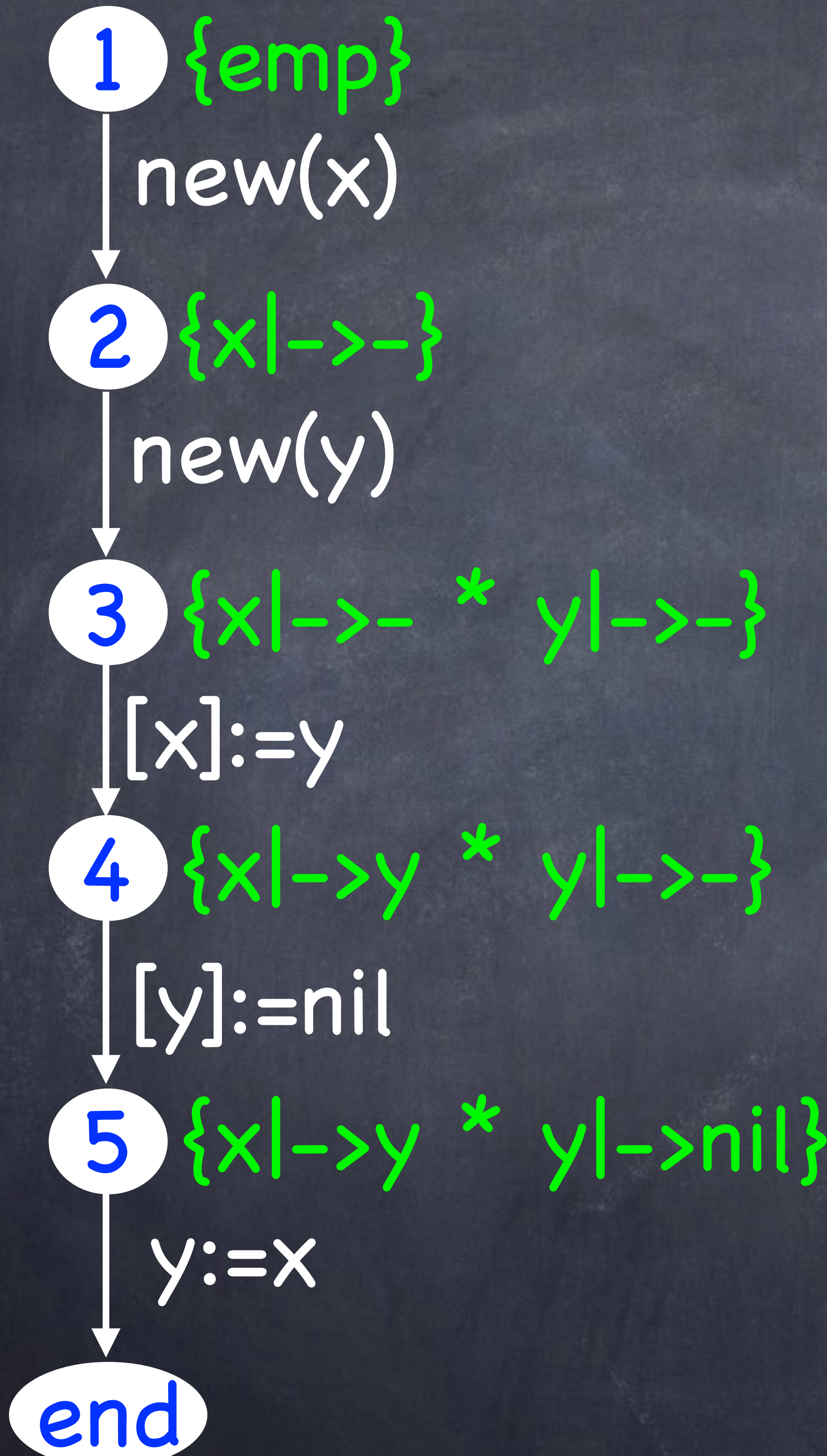


$\Pi\vert\Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi\vert\Sigma)[x'/x]$
$\Pi\vert\Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi\vert\Sigma * E \mapsto F)[x'/x]$
$\Pi\vert\Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi\vert\Sigma * E \mapsto G$
$\Pi;\Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi\vert\Sigma)[x'/x] * x \mapsto y'$
$\Pi\vert\Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi\vert\Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \implies \top}$$

new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;

Example 1



$$\Pi|\Sigma, \quad x := E \quad \Longrightarrow \quad x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x]$$

$$\Pi|\Sigma * E \mapsto F, \quad x := [E] \quad \Longrightarrow \quad x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x]$$

$$\Pi|\Sigma * E \mapsto F, \quad [E] := G \quad \Longrightarrow \quad \Pi|\Sigma * E \mapsto G$$

$$\Pi;\Sigma, \quad \mathsf{new}(x) \quad \Longrightarrow \quad (\Pi|\Sigma)[x'/x] * x \mapsto y'$$

$$\Pi|\Sigma * E \mapsto F, \quad \mathsf{dispose}(E) \quad \Longrightarrow \quad \Pi|\Sigma$$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \ \Longrightarrow \top}$$

```
new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;
```

Example 1

1 {emp}

new(x)

2 {x|->-}

new(y)

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

[x]:=y

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

[y]:=nil

5 {x|->y * y|->nil}

y:=x

end {x=y /\ x|->y' * y'|->nil}

$\Pi\|\Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi\|\Sigma)[x'/x]$
$\Pi\|\Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi\|\Sigma * E \mapsto F)[x'/x]$
$\Pi\|\Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi\|\Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi\|\Sigma)[x'/x] * x \mapsto y'$
$\Pi\|\Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi\|\Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \Longrightarrow \top}$$

new(x);
new(y);
[x]:=y;
[y]:=nil;
y:=x;

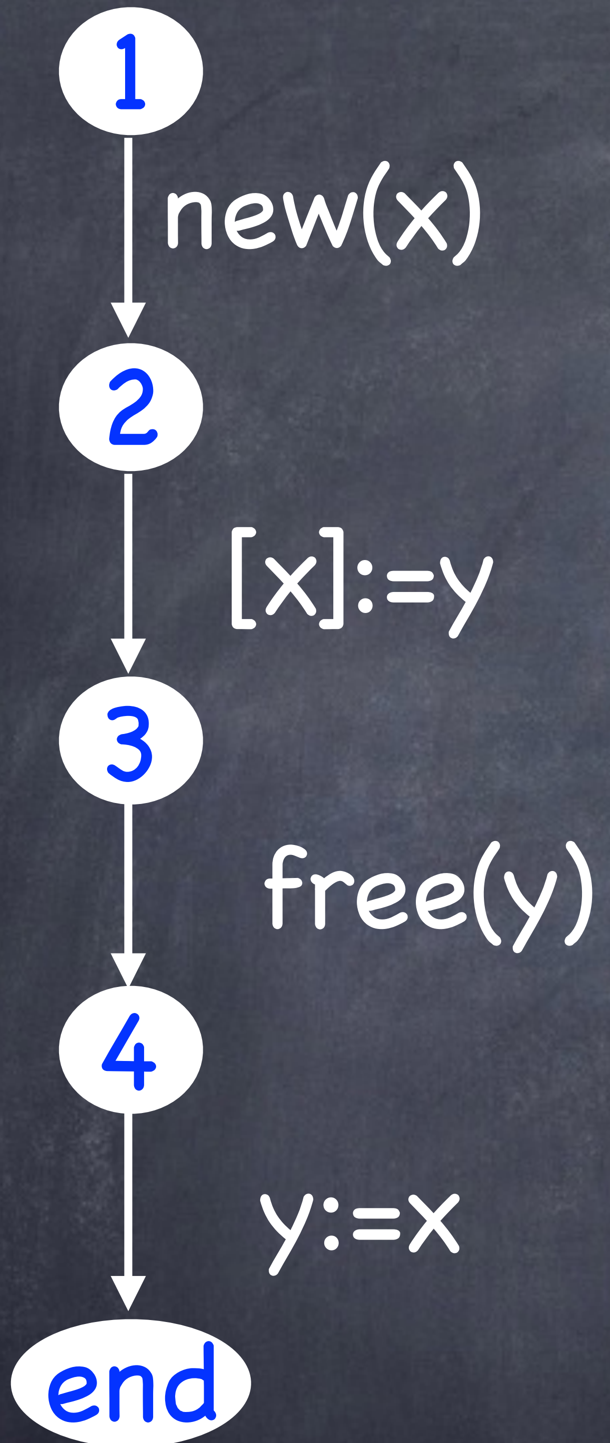
Example 2

$\Pi \vert \Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi \vert \Sigma)[x'/x]$
$\Pi \vert \Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi \vert \Sigma * E \mapsto F)[x'/x]$
$\Pi \vert \Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi \vert \Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi \vert \Sigma)[x'/x] * x \mapsto y'$
$\Pi \vert \Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi \vert \Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \implies \top}$$

new(x);
[x]:=y;
free(y);
y:=x;

Example 2



$$\Pi|\Sigma, \quad x := E \quad \Longrightarrow \quad x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x]$$

$$\Pi|\Sigma * E \mapsto F, \quad x := [E] \quad \Longrightarrow \quad x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x]$$

$$\Pi|\Sigma * E \mapsto F, \quad [E] := G \quad \Longrightarrow \quad \Pi|\Sigma * E \mapsto G$$

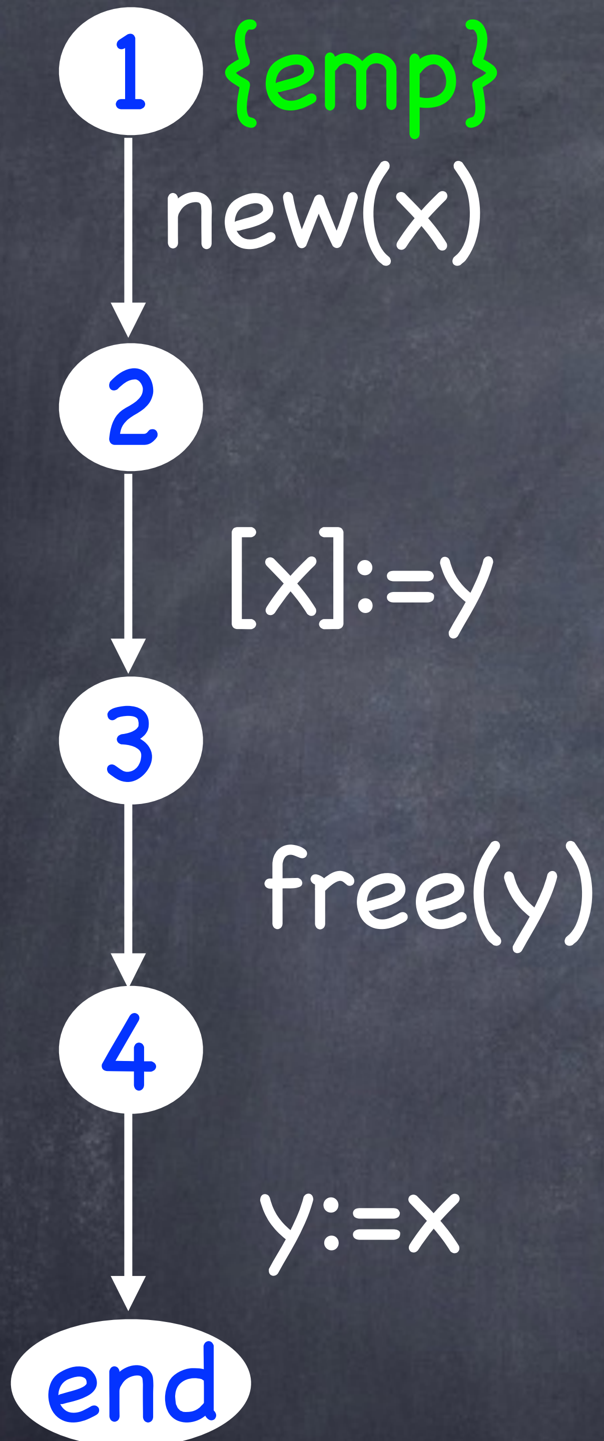
$$\Pi; \Sigma, \quad \mathsf{new}(x) \quad \Longrightarrow \quad (\Pi|\Sigma)[x'/x] * x \mapsto y'$$

$$\Pi|\Sigma * E \mapsto F, \quad \mathsf{dispose}(E) \quad \Longrightarrow \quad \Pi|\Sigma$$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma, \ A(E) \ \Longrightarrow \top}$$

```
new(x);
[x]:=y;
free(y);
y:=x;
```

Example 2

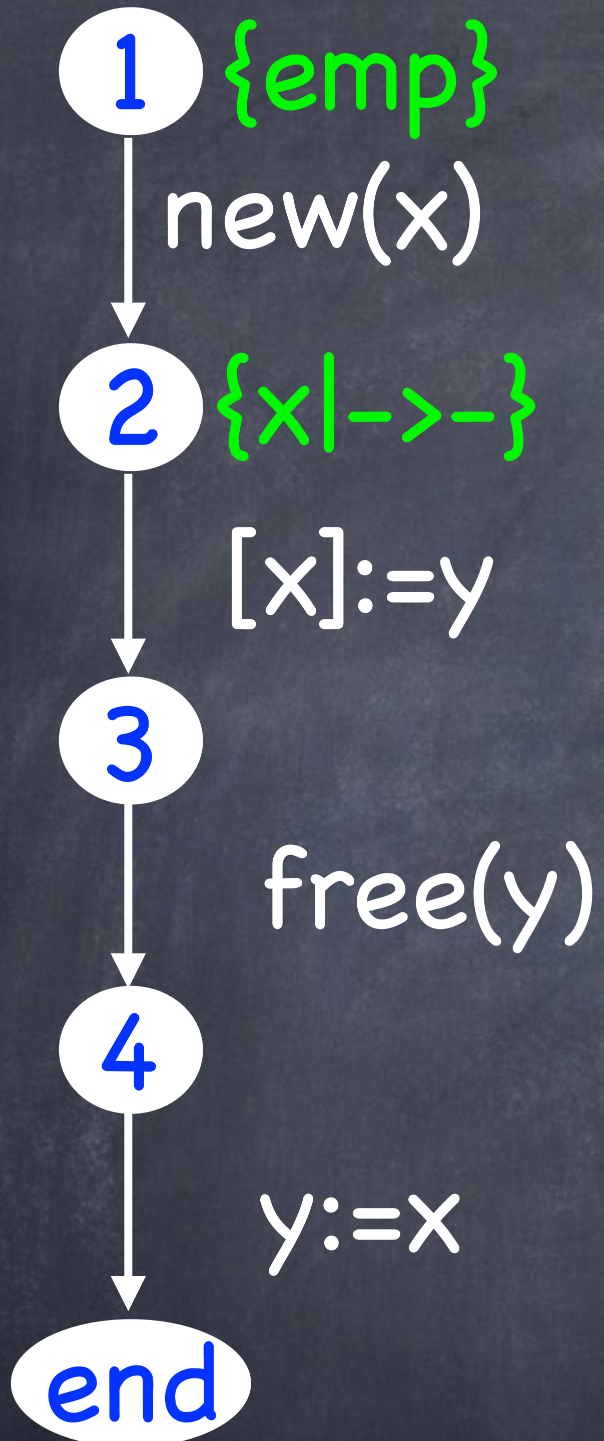


$\Pi\|\Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi\|\Sigma)[x'/x]$
$\Pi\|\Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi\|\Sigma * E \mapsto F)[x'/x]$
$\Pi\|\Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi\|\Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi\|\Sigma)[x'/x] * x \mapsto y'$
$\Pi\|\Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi\|\Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E)\ \Longrightarrow \top}$$

```
new(x);
[x]:=y;
free(y);
y:=x;
```

Example 2

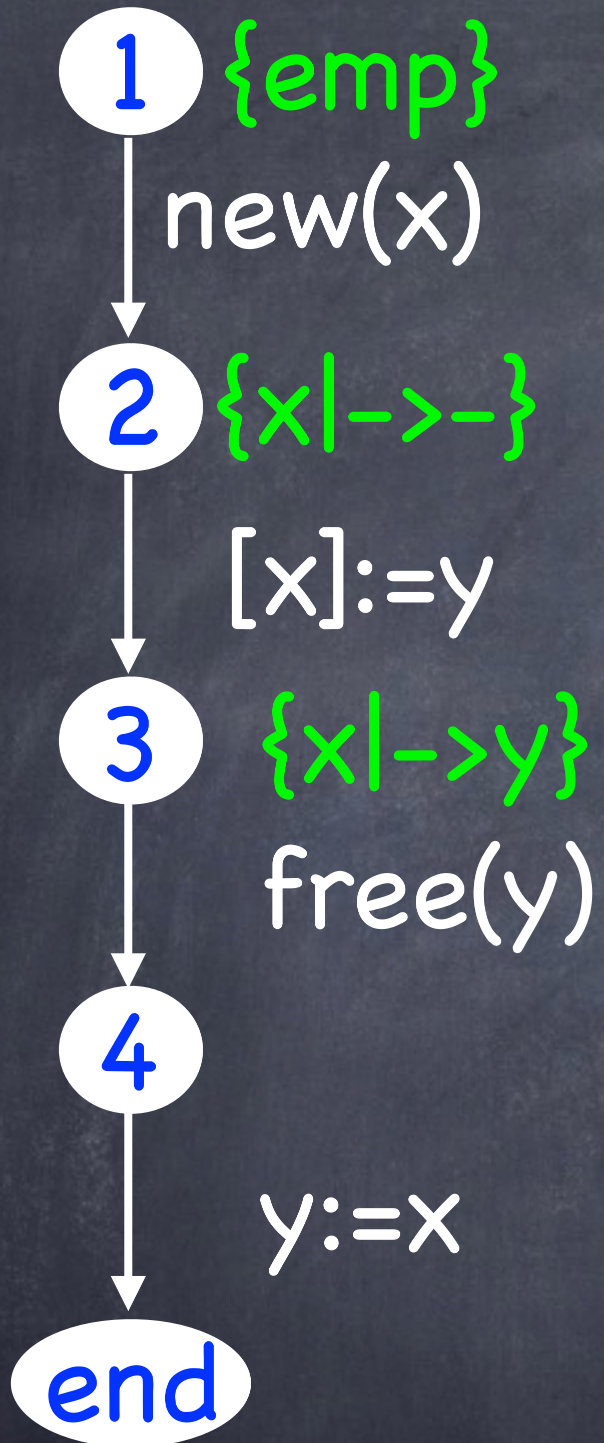


$\Pi\vert\Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi\vert\Sigma)[x'/x]$
$\Pi\vert\Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi\vert\Sigma * E \mapsto F)[x'/x]$
$\Pi\vert\Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi\vert\Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi\vert\Sigma)[x'/x] * x \mapsto y'$
$\Pi\vert\Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi\vert\Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ \ A(E) \ \Longrightarrow \top}$$

```
new(x);
[x]:=y;
free(y);
y:=x;
```

Example 2

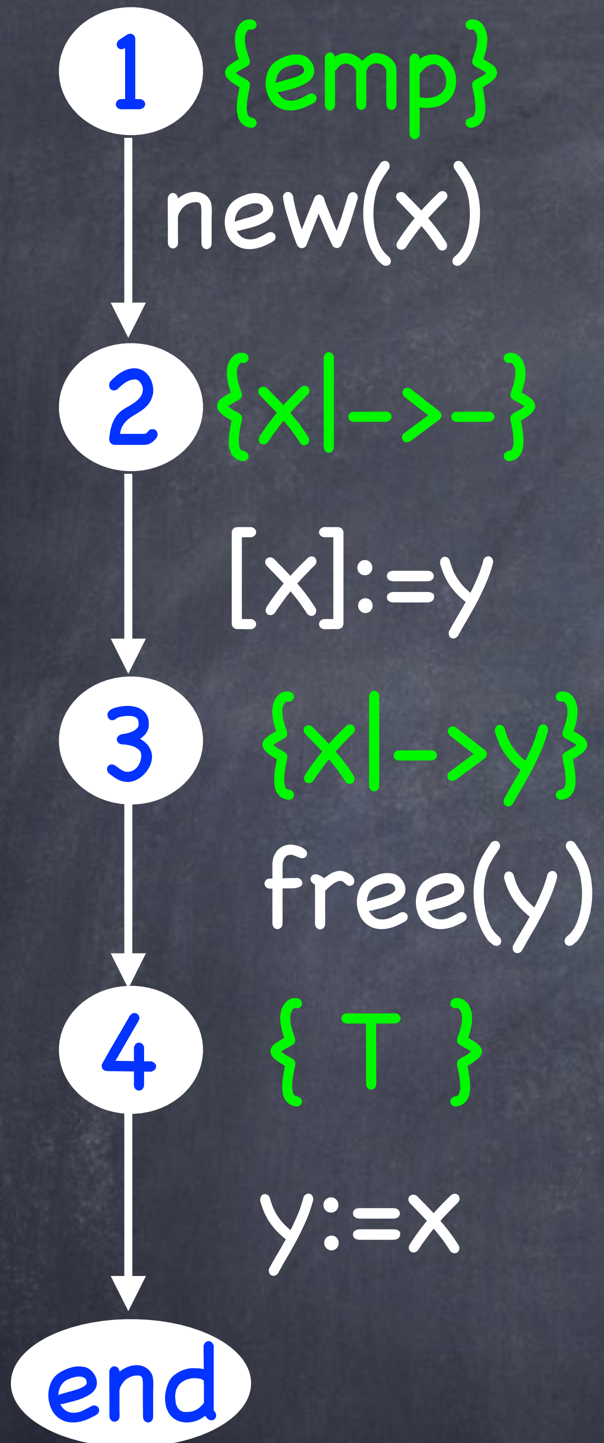


$$
\begin{array}{lll}
\Pi|\Sigma, & x := E & \Longrightarrow \quad x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Longrightarrow \quad x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Longrightarrow \quad \Pi|\Sigma * E \mapsto G \\
\Pi; \Sigma, & \mathsf{new}(x) & \Longrightarrow \quad (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Longrightarrow \quad \Pi|\Sigma
\end{array}
$$

$$
\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \ \Longrightarrow \top}
$$

```
new(x);
[x]:=y;
free(y);
y:=x;
```

Example 2



$\Pi \vert \Sigma,$	$x := E$	$\Longrightarrow$	$x = E[x'/x] \wedge (\Pi \vert \Sigma)[x'/x]$
$\Pi \vert \Sigma * E \mapsto F,$	$x := [E]$	$\Longrightarrow$	$x = F[x'/x] \wedge (\Pi \vert \Sigma * E \mapsto F)[x'/x]$
$\Pi \vert \Sigma * E \mapsto F,$	$[E] := G$	$\Longrightarrow$	$\Pi \vert \Sigma * E \mapsto G$
$\Pi; \Sigma,$	$\mathsf{new}(x)$	$\Longrightarrow$	$(\Pi \vert \Sigma)[x'/x] * x \mapsto y'$
$\Pi \vert \Sigma * E \mapsto F,$	$\mathsf{dispose}(E)$	$\Longrightarrow$	$\Pi \vert \Sigma$

$$\frac{\Pi|\Sigma \nvdash \mathsf{Allocated}(E)}{\Pi|\Sigma,\ A(E) \ \Longrightarrow \top}$$

```
new(x);
[x]:=y;
free(y);
y:=x;
```

Entailment

- During symbolic execution we need to compute entailments P |- Q
  - e.g. P |- E=F ???
- In a tool we need to compute them automatically.

Automating proofs

Specification $\{tree(p)\}$ DisposeTree(p) $\{emp\}$

$\{tree(i)*tree(j)\}$
DisposeTree(i);

DisposeTree(j);

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

Automating proofs

Specification {tree(p)} DisposeTree(p) {emp}

{tree(i)*tree(j)}
DisposeTree(i);

DisposeTree(j);

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

Automating proofs

Specification {tree(p)} DisposeTree(p) {emp}

{tree(i)*tree(j)}
DisposeTree(i);
{emp*tree(j)}
DisposeTree(j);

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

Automating proofs

Specification {tree(p)} DisposeTree(p) {emp}

{tree(i)*tree(j)}
DisposeTree(i);
{emp*tree(j)}
DisposeTree(j);

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

Automating proofs

Specification {tree(p)} DisposeTree(p) {emp}

{tree(i)*tree(j)}
DisposeTree(i);
{emp*tree(j)}
DisposeTree(j);
{emp*emp}

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

Automating proofs

Specification {tree(p)} DisposeTree(p) {emp}

{tree(i)*tree(j)}
DisposeTree(i);
{emp*tree(j)}
DisposeTree(j);
{emp*emp}
{emp}

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

Bi-Abduction

Synthesising both missing resources (anti-frame) and unneeded resources (frame) gives rise to a new notion

Bi-Abduction:

given A and B compute ?antiframe and ?frame such that

$$A * ?antiframe \vdash B * ?frame$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)

void foo(list_item *x,list_item *y)

Post: list(x)

emp

```
node* p(list_item *y) {
 node *x, *z;
   x=malloc(sizeof(list_item)); x->tail =  0;
   z=malloc(sizeof(list_item)); z->tail =  0;
   foo(x,y);
   foo(x,z);
   return x;
}
```

emp

Bi-abductive prover

Bi-Abductive symbolic execution

Pre: list(x) * list(y)

void foo(list_item *x,list_item *y)

Post: list(x)

emp

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

$x \mapsto 0$

Bi-abductive prover

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

emp

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
    void foo(list_item *x,list_item *y)
Post: list(x)

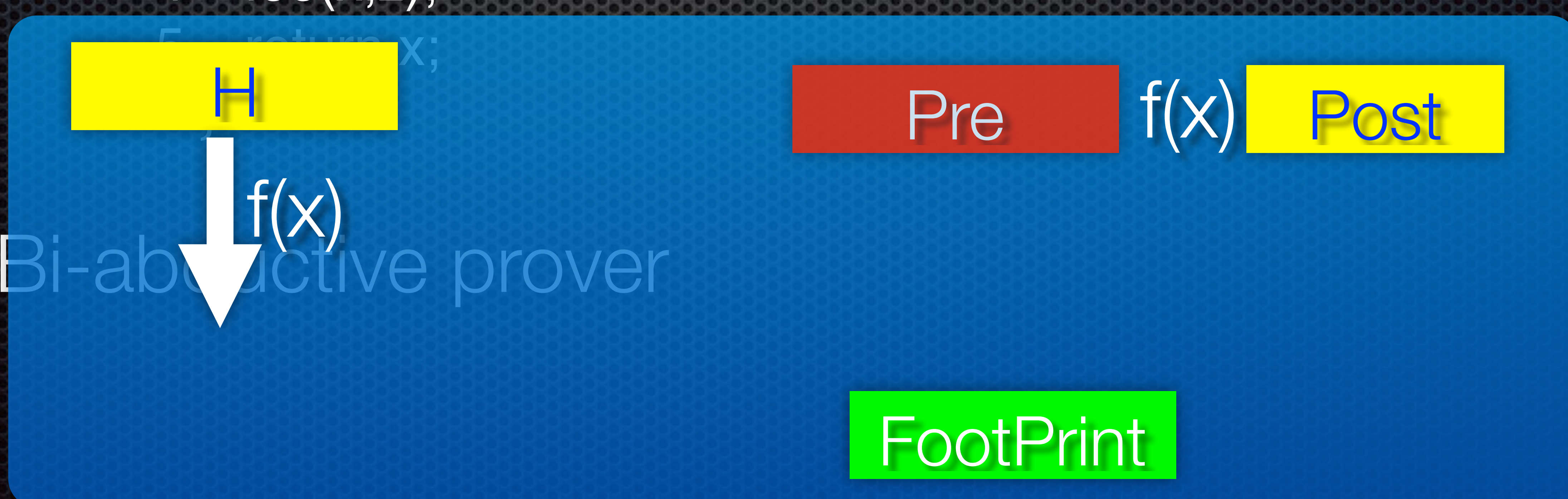
emp

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

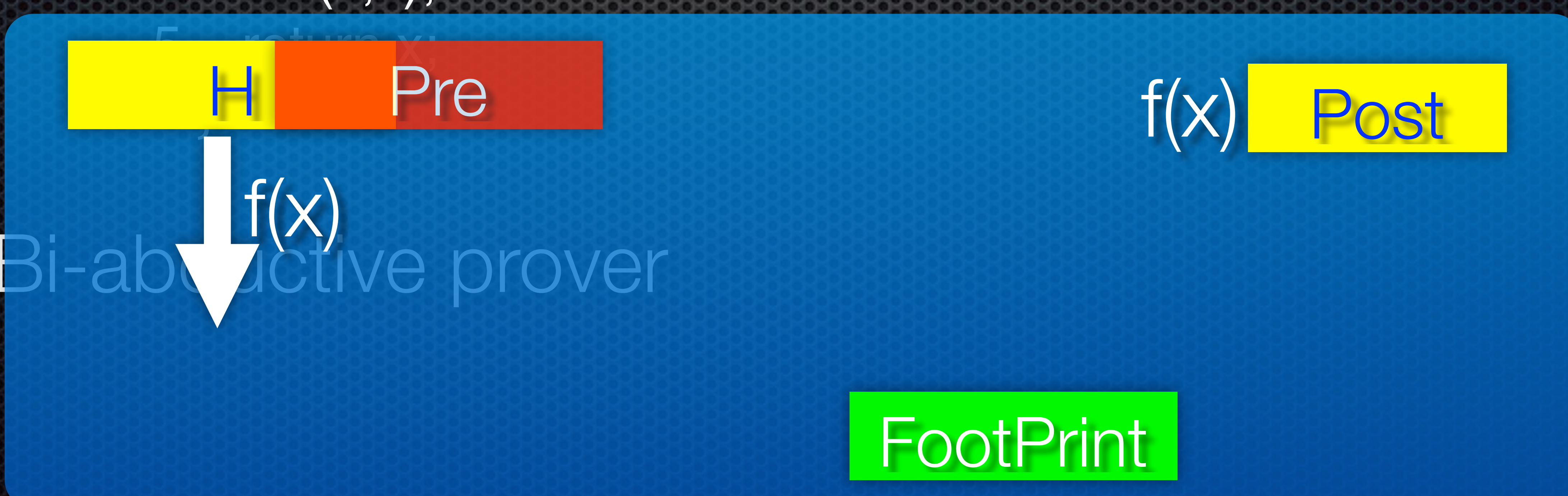
```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

```
node* p(list_item *y) {
 node *x, *z;
   x=malloc(sizeof(list_item)); x->tail =  0;
   z=malloc(sizeof(list_item)); z->tail =  0;
   foo(x,y);
   foo(x,z);
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

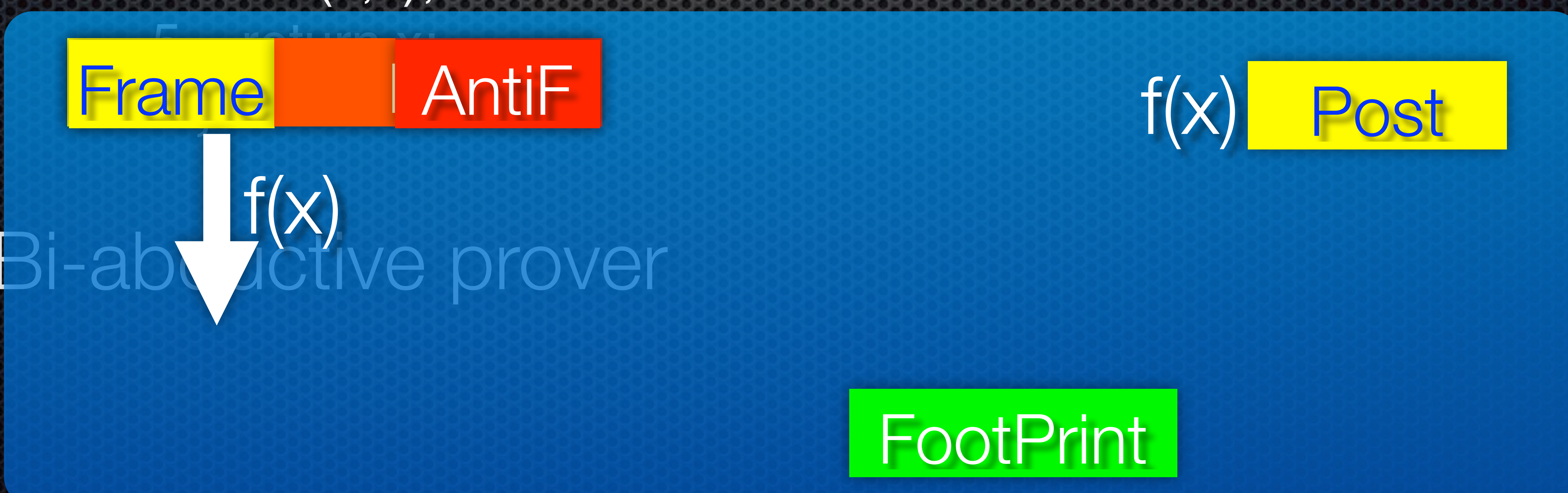
```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

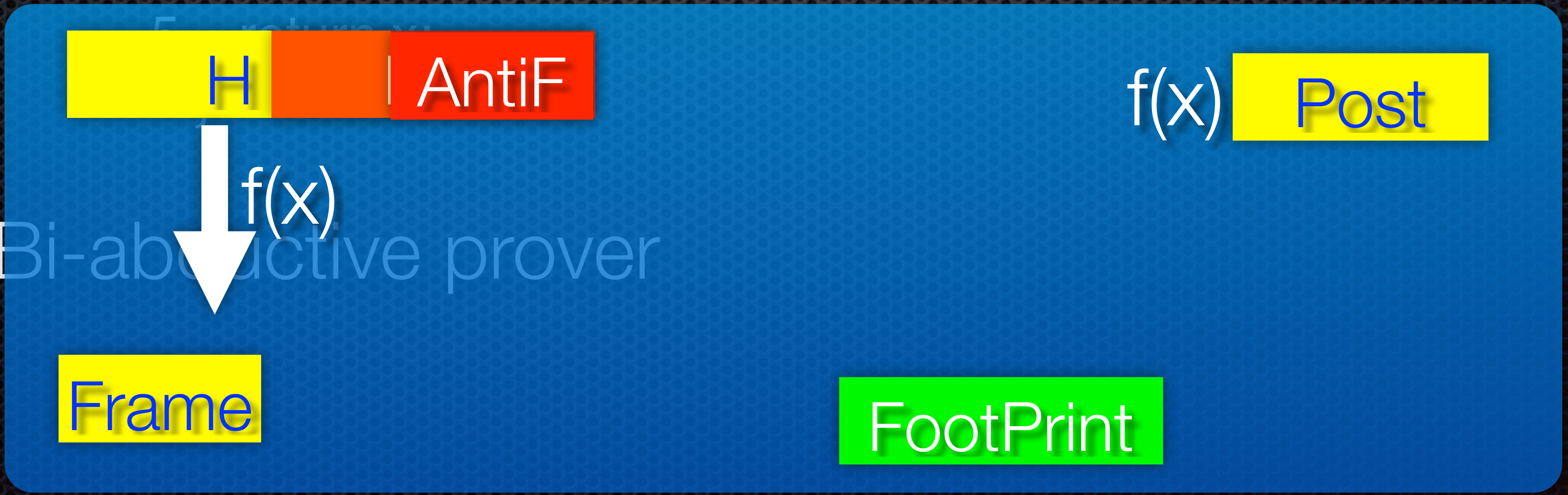
```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive symbolic execution

Pre: list(x) * list(y)

void foo(list_item *x,list_item *y)

Post: list(x)

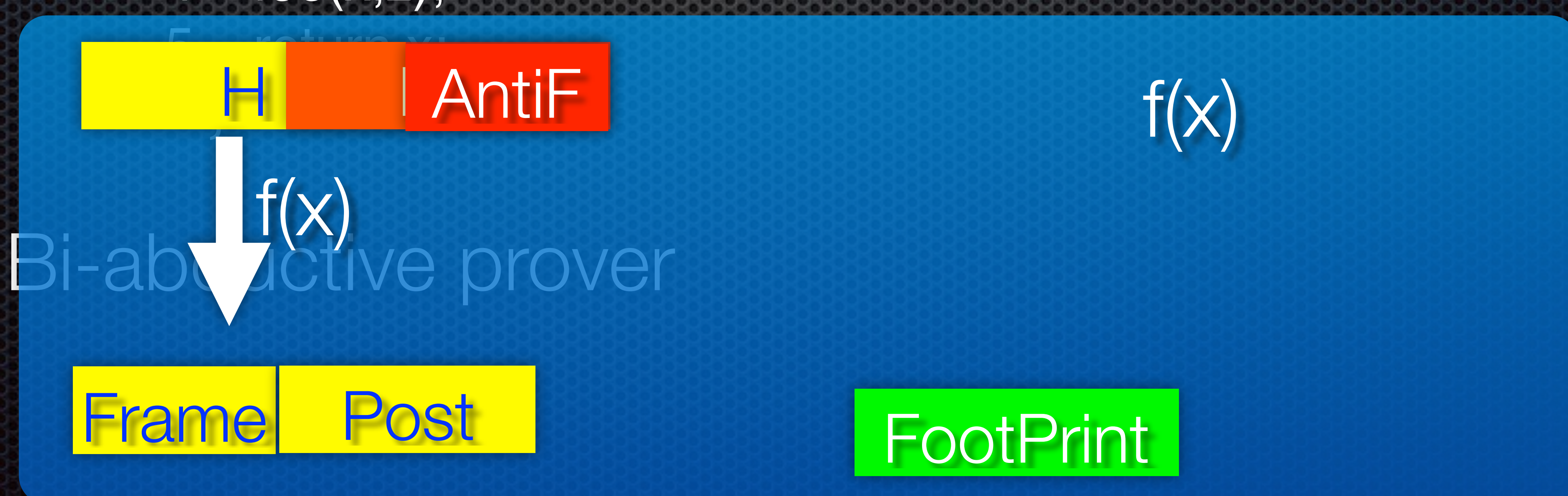
```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

```
node* p(list_item *y) {
 node *x, *z;
1   x=malloc(sizeof(list_item)); x->tail =  0;
2   z=malloc(sizeof(list_item)); z->tail =  0;
3   foo(x,y);
4   foo(x,z);
5   return x;
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

emp

```
node* p(list_item *y) {
  node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

$$x \mapsto 0 * z \mapsto 0 * ?antiframe \vdash \mathsf{list}(x) * \mathsf{list}(y) * ?frame$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

$$x \mapsto 0 * z \mapsto 0 * \text{list}(y) \vdash \text{list}(x) * \text{list}(y) * z \mapsto 0$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

$\mathsf{list}(y)$

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

Bi-abductive prover

$$x \mapsto 0 * z \mapsto 0 * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y) * z \mapsto 0$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

$\mathsf{list}(y)$

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

Bi-abductive prover

$$\mathsf{list}(x) * z \mapsto 0 * ?antiframe \vdash \mathsf{list}(x) * \mathsf{list}(z) * ?frame$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

$\text{list}(y)$

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\text{list}(x) * z \mapsto 0$

Bi-abductive prover

$$\text{list}(x) * z \mapsto 0 * \text{emp} \vdash \text{list}(x) * \text{list}(z) * \text{emp}$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

$\mathsf{list}(y)$

```
node* p(list_item *y) {
  node *x, *z;
   x=malloc(sizeof(list_item)); x->tail =  0;
   z=malloc(sizeof(list_item)); z->tail =  0;
   foo(x,y);
   foo(x,z);
   return x;
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

$\mathsf{list}(x)$

Bi-abductive prover

$$\mathsf{list}(x) * z \mapsto 0 * \mathsf{emp} \vdash \mathsf{list}(x) * \mathsf{list}(z) * \mathsf{emp}$$

Bi-Abductive symbolic execution

Pre: list(x) * list(y)
void foo(list_item *x,list_item *y)
Post: list(x)

$\mathsf{list}(y)$

```
node* p(list_item *y) {
 node *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
  return x;
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

$\mathsf{list}(x)$

$\mathsf{list}(\mathtt{ret})$

Bi-abductive prover

$$\mathsf{list}(x) * z \mapsto 0 * \mathsf{emp} \vdash \mathsf{list}(x) * \mathsf{list}(z) * \mathsf{emp}$$

Abstraction

Fixed-point computation

- For each node of CFG compute the set of all symbolic heaps it can have in any computations
- What happen in case of loops?

Example

```
head=nil;
while (true) {
    new(n);
    [n]=head;
    head=n;
}
....
```

Example

```
head=nil;
while (true) { head=n /\ n|->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Example

```
head=nil;
while (true) { head=n /\ n|->n'  * n' |-> nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Example

```
head=nil;
while (true) { head=n /\ n |->n″ * n″ |-> n′ * n′ |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Example

```
head=nil;
while (true) { he                        l->nil
    new(n)
    [n]=head;
    head=n;
}
....
```

Diverges!

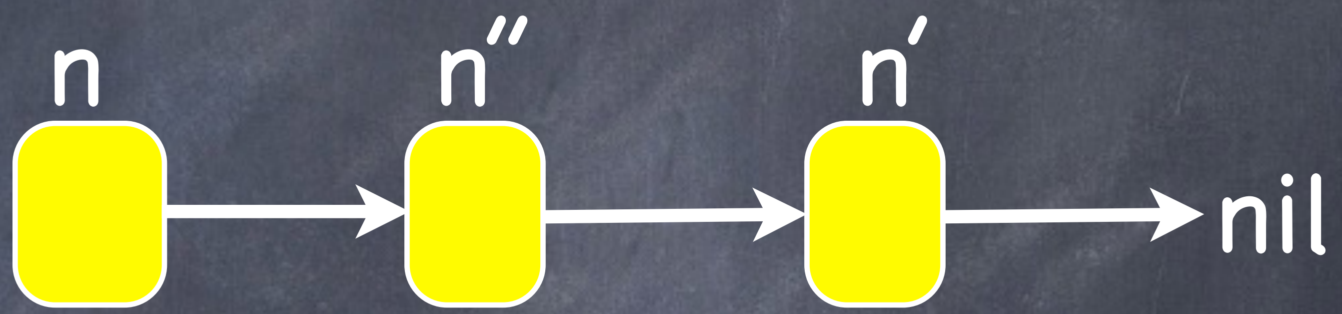
Observation

- The set of Symbolic Heaps is unbounded
- In general no guarantee of termination in the fixed-point computation.

Abstraction

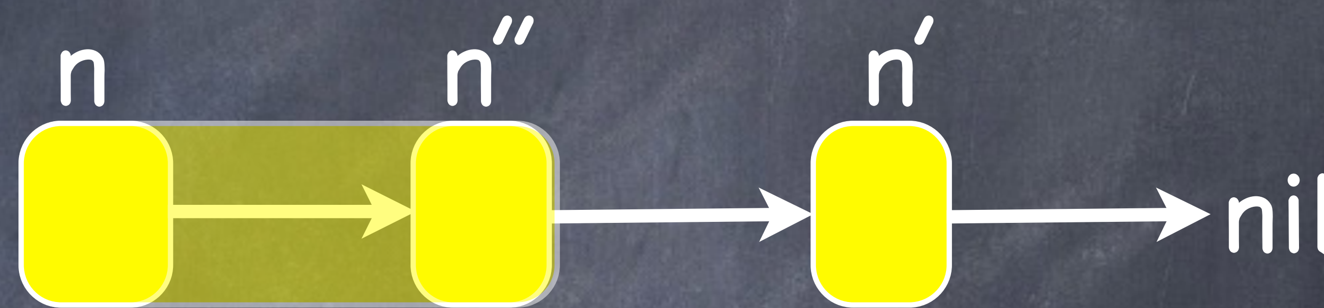
```
head=nil;
while (true) { head=n /\ n |->n" * n" |->n' * n' |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Abstraction



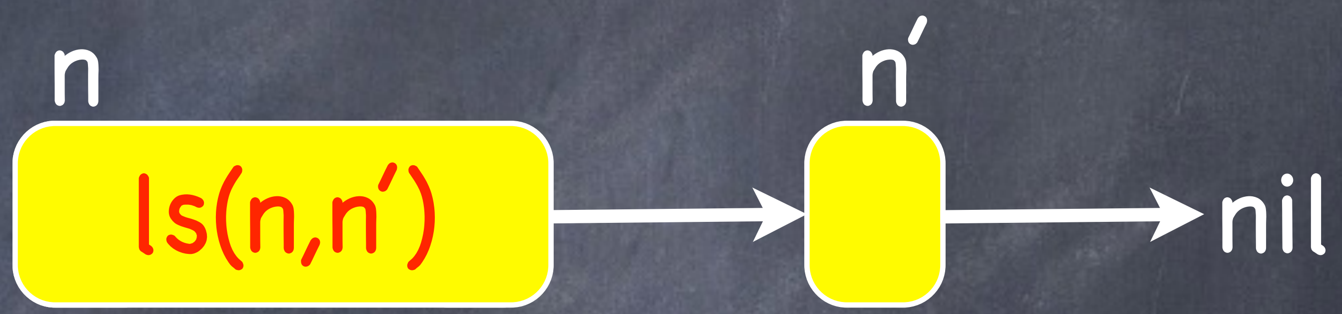

```
head=nil;
while (true) { head=n /\ n |->n″ * n″ |->n′ * n′ |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Abstraction



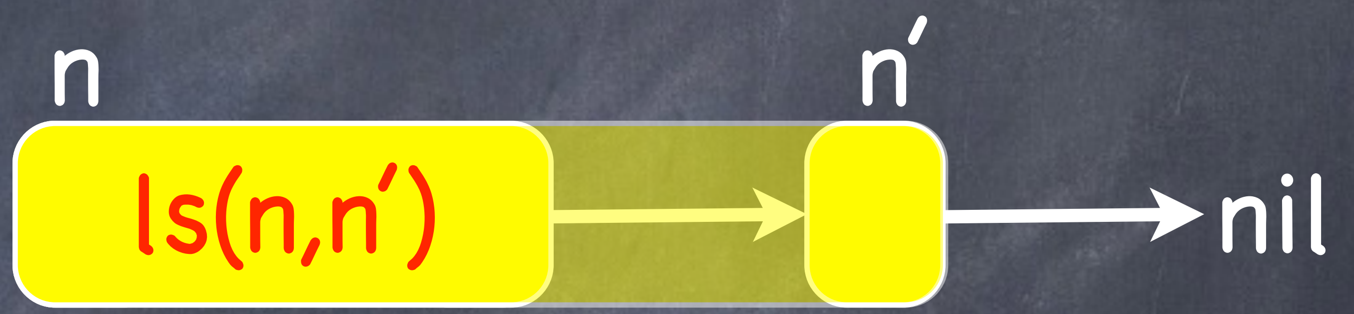

```
head=nil;
while (true) { head=n /\ n |->n″ * n″ |->n′ * n′ |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Abstraction



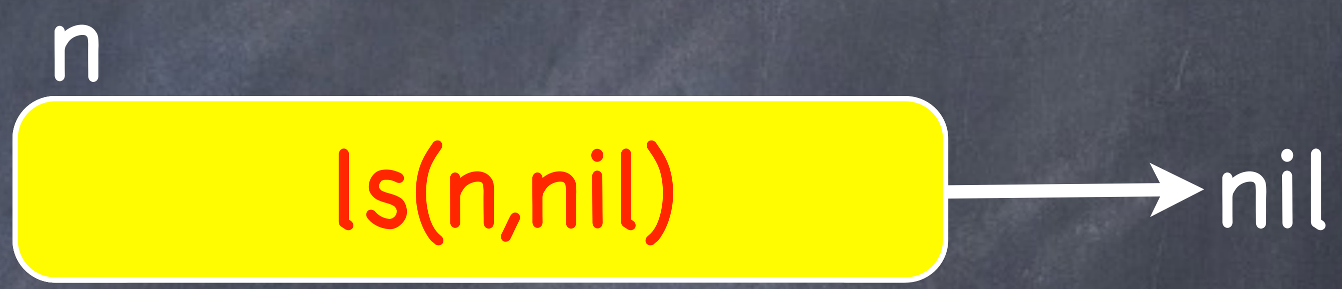

```
head=nil;
while (true) { head=n /\ n |->n″ * n″ |->n′ * n′ |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Abstraction




```
head=nil;
while (true) { head=n /\ n |->n″ * n″ |->n′ * n′ |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Abstraction




```
head=nil;
while (true) { head=n /\ n |->n" * n" |->n' * n' |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

Abstraction

Note: we are forgetting (abstracting) the information about the length of the list


```
head=nil;
while (true) { head=n /\ n |->n'' * n'' |->n' * n' |->nil
    new(n);
    [n]=head;
    head=n;
}
....
```

here we don't know

here we know it's a list of three elements

Canonical Symbolic Heaps

- We want to define a smaller set of symbolic heaps that is finite called Canonical Symbolic Heaps (CSH)
- We use a canonicalization function
    - can: SH ---> CSH

to obtain canonical heaps from non-canonical ones.

The canonicalization function is an abstraction function:

abs: SH ----> SH

Canonical Form

$\Pi|\Sigma$ is in canonical form if and only if

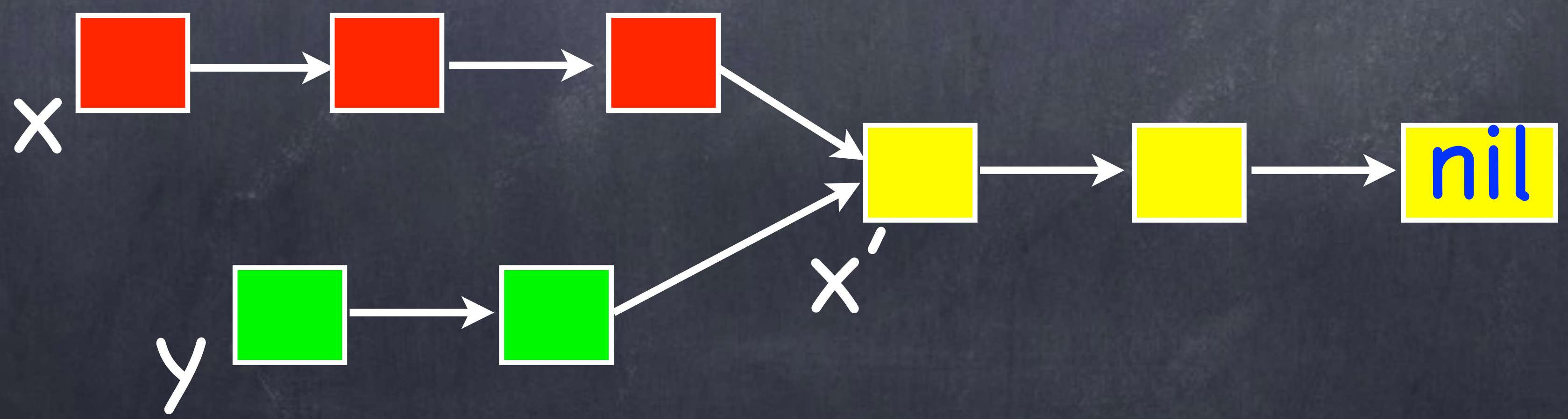
- Π does not contain primed variables
- if $x' \in \Sigma$ then is reachable and
  - x' is shared or
  - x' points to a possible dangling variable or
  - x' is possibly dangling
  - x' is the internal point of a cycle of length 2

Proposition: CSH is finite.

Examples of Canonical Heaps

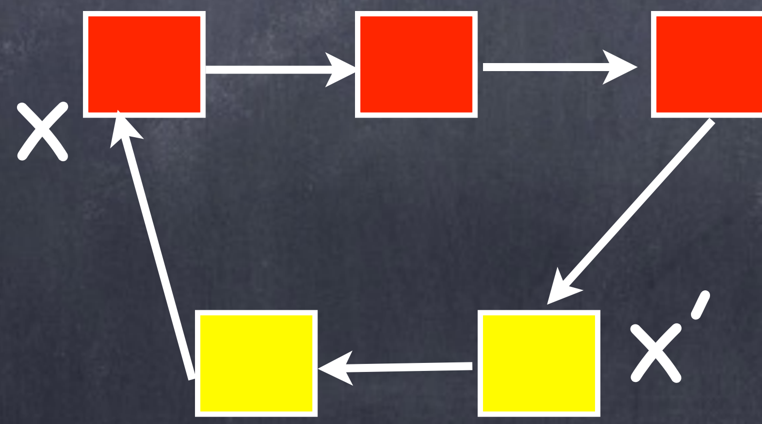
Examples of Canonical Heaps

z=nil | ls(x,x')*ls(y,x')*ls(x',nil)



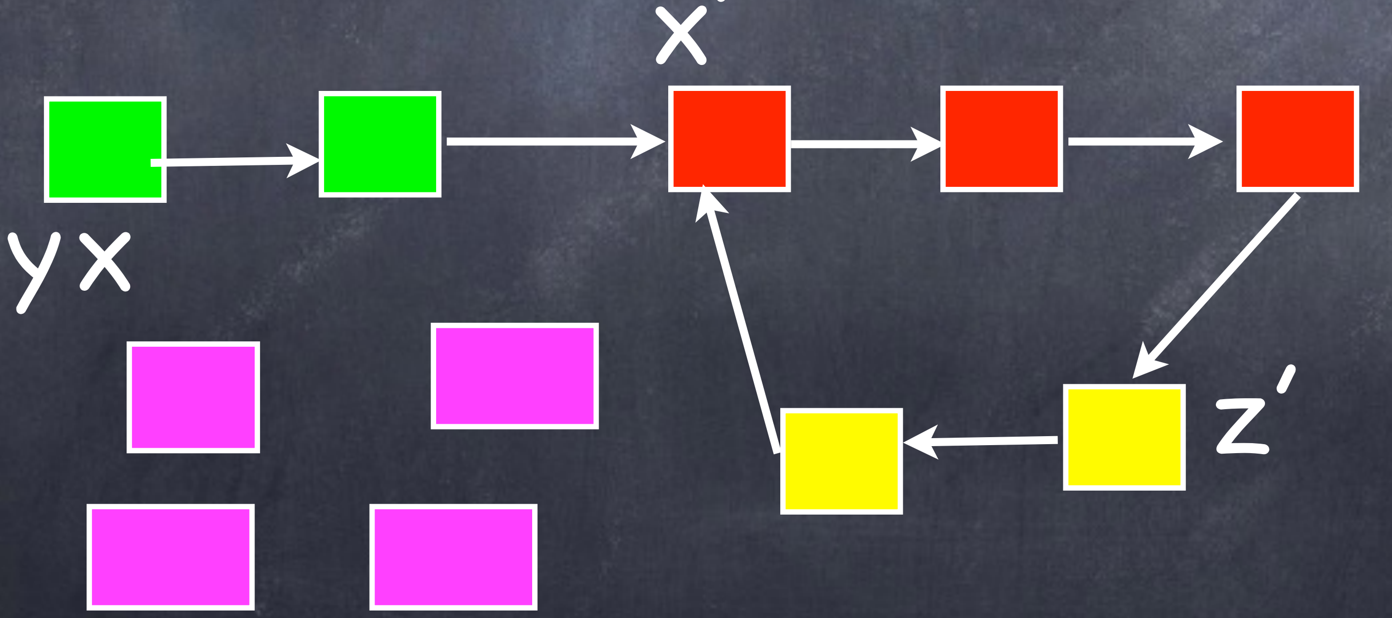
Examples of Canonical Heaps

true | ls(x,x′) * ls (x′,x)



Examples of Canonical Heaps

y=x | ls(x,x')* ls(x',z') * ls(z',x') * junk



Abstraction Function

abs: Heaps -->Heaps

Abstraction Function

abs: Heaps -->Heaps

Defined in terms of "abstraction rules"

$$\frac{\text{condition}}{H*H' \text{ ---> } H*H''}$$

Abstraction Function

abs: Heaps -->Heaps

Defined in terms of "abstraction rules"

$$\frac{\text{condition}}{H*H' \dashrightarrow H*H''}$$

Intuitive algorithm: Rules are applied as much as possible until they cannot be applied anymore

Abstraction Rules

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}} \text{ Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}} \text{ Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \mathsf{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \mathsf{nil})} \text{ Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)} \text{ Abs2}$$

where predicates P1, P2 stand for ls or |->

Example

x=nil | x'|->nil * ls(y,x')

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}} \text{ Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}} \text{ Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \mathsf{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \mathsf{nil})} \text{ Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)} \text{ Abs2}$$

Example

x=nil | x'|->nil * ls(y,x')

↓ Abs 1

x=nil | ls(y,nil)

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']}\ \text{St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']}\ \text{St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}}\ \text{Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}}\ \text{Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \mathsf{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \mathsf{nil})}\ \text{Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)}\ \text{Abs2}$$

Example

x=nil | x'|->nil * ls(y,x')

↓ Abs 1

x=nil | ls(y,nil) ✓

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}} \text{ Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \mathsf{junk}} \text{ Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \mathsf{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \mathsf{nil})} \text{ Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)} \text{ Abs2}$$

Example

x=nil | x'|->nil * ls(y,x')

↓ Abs 1

x=nil | ls(y,nil) ✓

x=x' | ls(y,x'')* x''|->x' *
ls(x',nil)

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}} \text{ Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}} \text{ Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \text{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \text{nil})} \text{ Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)} \text{ Abs2}$$

Example

x=nil | x'|->nil * ls(y,x')

↓ Abs 1

x=nil | ls(y,nil) ✓

x=x' | ls(y,x'')* x''|->x' * ls(x',nil)

↓ St 1

true | ls(y,x'')* x''|->x * ls(x,nil)

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']}\ \text{St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']}\ \text{St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}}\ \text{Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}}\ \text{Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \text{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \text{nil})}\ \text{Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)}\ \text{Abs2}$$

Example

x=nil | x'|->nil * ls(y,x')

↓ Abs 1

x=nil | ls(y,nil) ✓

x=x' | ls(y,x'')* x''|->x' * ls(x',nil)

↓ St 1

true | ls(y,x'')* x''|->x * ls(x,nil)

→ Abs 2

true | ls(y,x) * ls(x,nil)

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']} \text{ St2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P(x', E) \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}} \text{ Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi, \Sigma)}{\Pi \,|\, \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}} \text{ Garbage2}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \text{nil}}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi|\Sigma * (E, \text{nil})} \text{ Abs1}$$

$$\frac{x' \notin Vars'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi|\Sigma * (E, F) * P_3(G, H)} \text{ Abs2}$$

Example

x=nil | x'|->nil * ls(y,x')

↓ Abs 1

x=nil | ls(y,nil) ✓

x=x' | ls(y,x")* x"|->x' * ls(x',nil)

↓ St 1

true | ls(y,x")* x"|->x * ls(x,nil)

→ Abs 2

true | ls(y,x) * ls(x,nil) ✓

$$\frac{}{E = x' \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']}\ \text{St1}$$

$$\frac{}{x' = E \wedge \Pi|\Sigma \rightsquigarrow (\Pi|\Sigma)[E/x']}\ \text{St2}$$

$$\frac{x' \notin Vars'(\Pi,\Sigma)}{\Pi \,|\, \Sigma * P(x',E) \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}}\ \text{Garbage1}$$

$$\frac{x', y' \notin Vars'(\Pi,\Sigma)}{\Pi \,|\, \Sigma * P_1(x',y') * P_2(y',x') \rightsquigarrow \Pi \,|\, \Sigma \cup \text{junk}}\ \text{Garbage2}$$

$$\frac{x' \notin Vars'(\Pi,\Sigma,E,F) \qquad \Pi \vdash F = \text{nil}}{\Pi|\Sigma * P_1(E,x') * P_2(x',F) \rightsquigarrow \Pi|\Sigma * (E,\text{nil})}\ \text{Abs1}$$

$$\frac{x' \notin Vars'(\Pi,\Sigma,E,F,G,H) \quad \Pi \vdash F = G}{\Pi|\Sigma * P_1(E,x') * P_2(x',F) * P_3(G,H) \rightsquigarrow \Pi|\Sigma * (E,F) * P_3(G,H)}\ \text{Abs2}$$

Some results

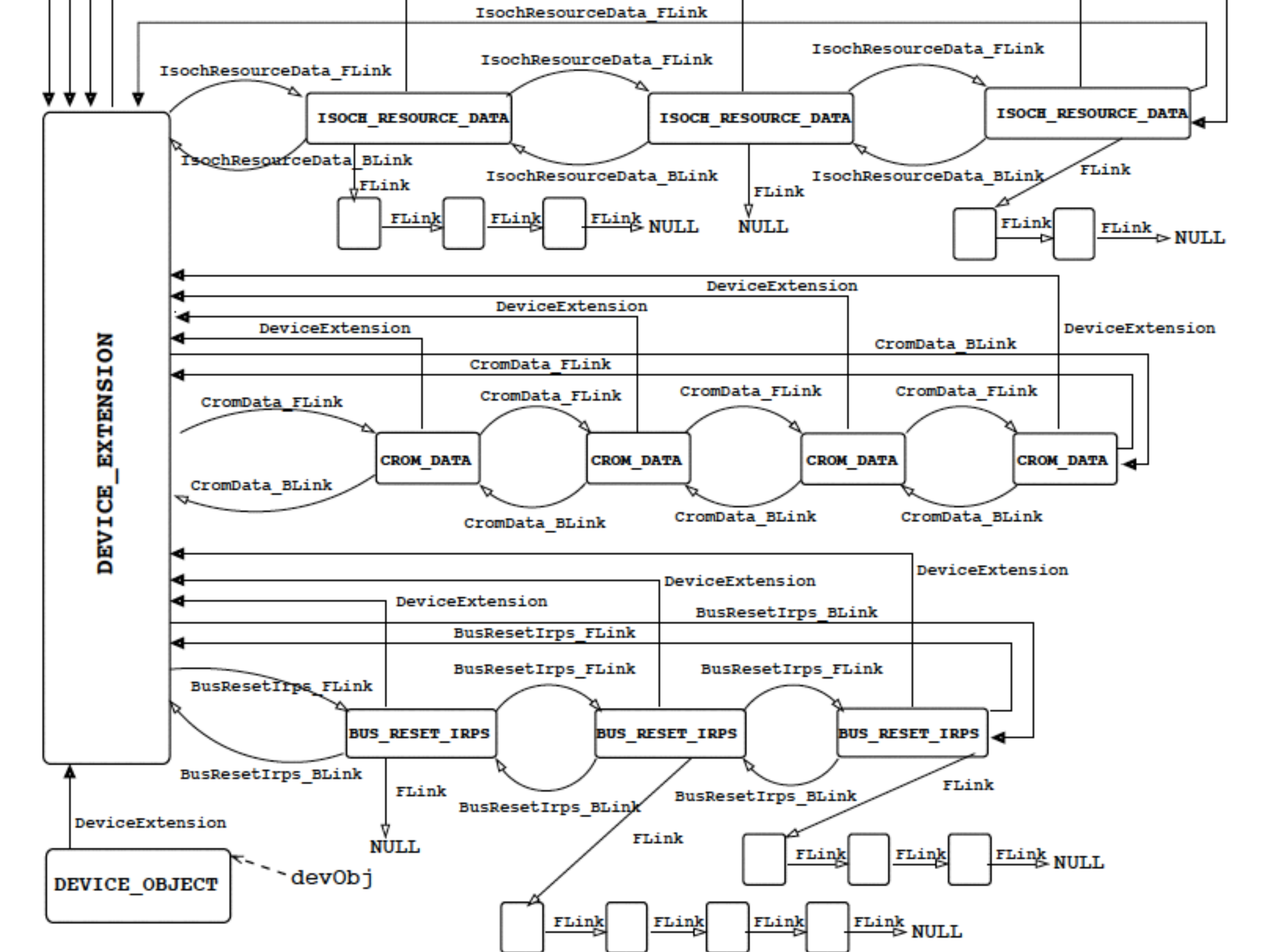
Lemma (Soundness): if H -->H' then H implies H'

Proposition: H is in canonical form iff no abstraction rule fires.

Proposition: Abstraction rules don't have infinite reduction sequences.

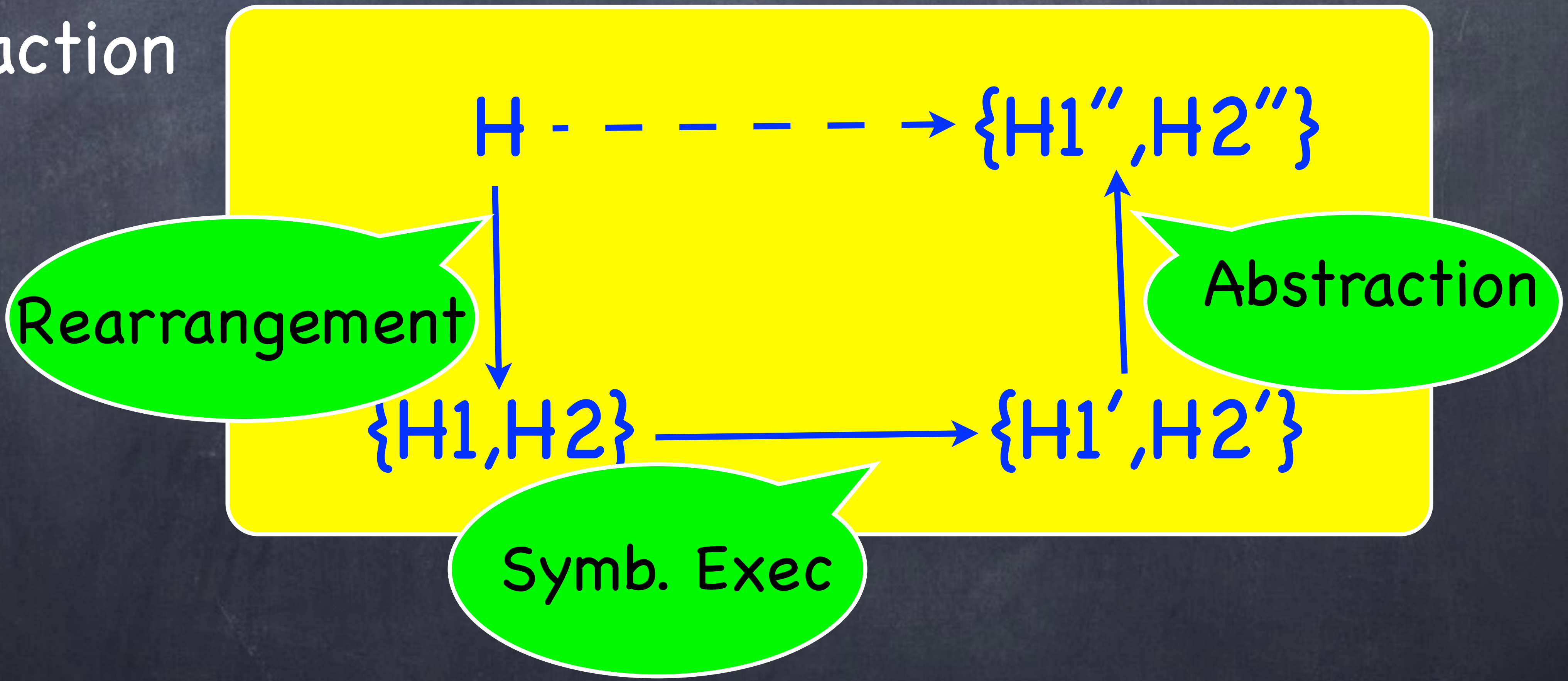
Framework for abstraction

- Abstraction rules give us a framework to define abstract domains on the heap
- By changing the set of abstraction rules we can encode different (e.g., richer) domains.
- Warning: defining good rules is not easy.


IsochResourceData_FLink
IsochResourceData_FLink
IsochResourceData_FLink
IsochResourceData_FLink
ISOCH_RESOURCE_DATA
ISOCH_RESOURCE_DATA
ISOCH_RESOURCE_DATA
IsochResourceData_BLink
IsochResourceData_BLink
IsochResourceData_BLink
FLink
NULL
DeviceExtension
CromData_BLink
CromData_FLink
CROM_DATA
CROM_DATA
CROM_DATA
CROM_DATA
DEVICE_EXTENSION
BusResetIrps_BLink
BusResetIrps_FLink
BUS_RESET_IRPS
BUS_RESET_IRPS
BUS_RESET_IRPS
DEVICE_OBJECT
devObj

Abstract Transformer

- It's computed in 3 phases:
  - Rearrangement
  - Symbolic Execution
  - Abstraction



Rearrangement Phase

- Try to make explicit a memory cell
- Explicit means: it appears in a points-to predicate
- Rearrangement allows application of symbolic execution rules.
- When the needed memory cell is already explicit, then rearrangement is the identity function.

Rearrangement Rules

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \ \text{Unroll}$$

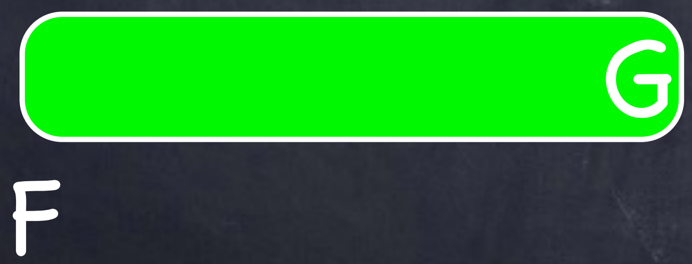
$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x', G)} \ \text{UnrollN}$$

Rearrangement Rules

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\; \longrightarrow_r \;\; \Pi|\Sigma * E \mapsto G} \text{ Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \;\; \longrightarrow_r \;\; \Pi|\Sigma * E \mapsto G} \text{ Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \;\; \longrightarrow_r \;\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x', G)} \text{ UnrollN}$$



Rearrangement Rules

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x', G)} \ \text{UnrollN}$$

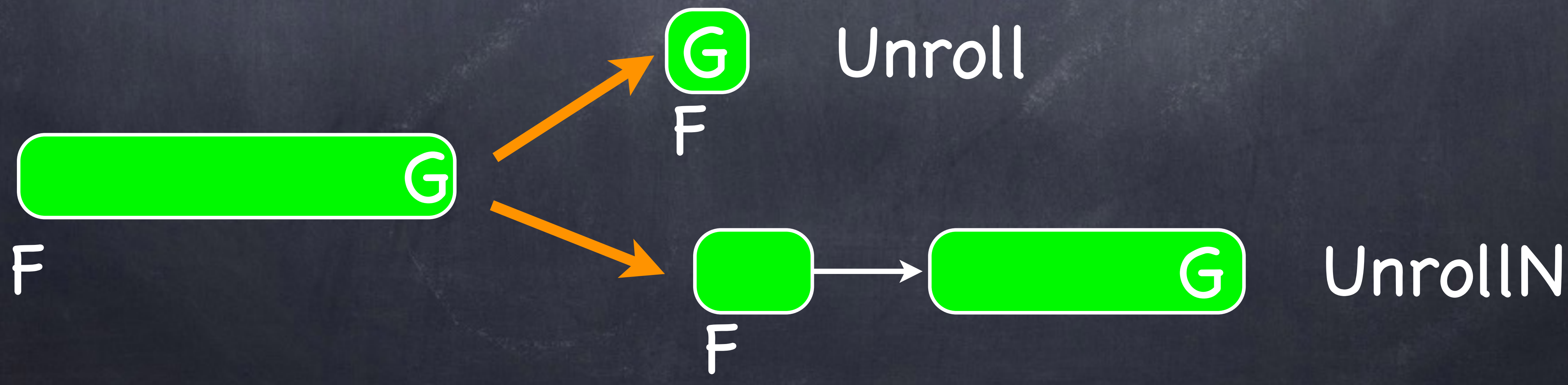
Unroll

Rearrangement Rules

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G} \text{ Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G} \text{ Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)} \text{ UnrollN}$$



Example

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)}\ \text{UnrollN}$$

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \text{ Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \text{ Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)} \text{ UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G} \text{ Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G} \text{ Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F, G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x', G)} \text{ UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll

Example

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \text{ Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \text{ Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)} \text{ UnrollN}$$

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)}\ \text{UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll

UnrollN

true | ls(y,x)*x|->nil

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \text{ Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto G} \text{ Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \ \longrightarrow_r \ \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)} \text{ UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll

UnrollN

true | ls(y,x)*x|->nil

true | ls(y,x)*x|->x'*ls(x',nil)

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)}\ \text{UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll

UnrollN

true | ls(y,x)*x|->nil

x:=[x]

true | ls(y,x)*x|->x'*ls(x',nil)

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)}\ \text{UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll ↓

true | ls(y,x)*x|->nil

↓ x:=[x]

x=nil | ls(y,x')*x'|->nil

UnrollN →

true | ls(y,x)*x|->x'*ls(x',nil)

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)}\ \text{UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

x:=[x]

x=nil | ls(y,x')*x'|->nil

UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

x:=[x]

Example

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * F \mapsto G \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Switch}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto G}\ \text{Unroll}$$

$$\frac{\Pi|\Sigma \vdash E = F}{\Pi|\Sigma * \mathsf{ls}(F,G) \;\longrightarrow_r\; \Pi|\Sigma * E \mapsto x' * \mathsf{ls}(x',G)}\ \text{UnrollN}$$

Consider the command

x:=[x]

true | ls(y,x)*ls(x,nil)

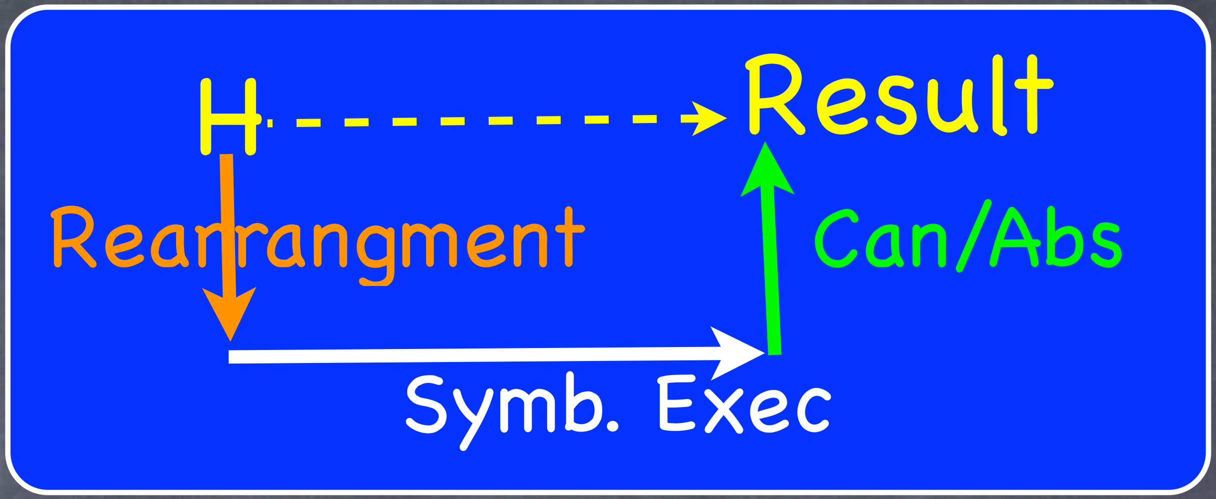
Unroll: true | ls(y,x)*x|->nil

x:=[x]: x=nil | ls(y,x')*x'|->nil

UnrollN: true | ls(y,x)*x|->x'*ls(x',nil)

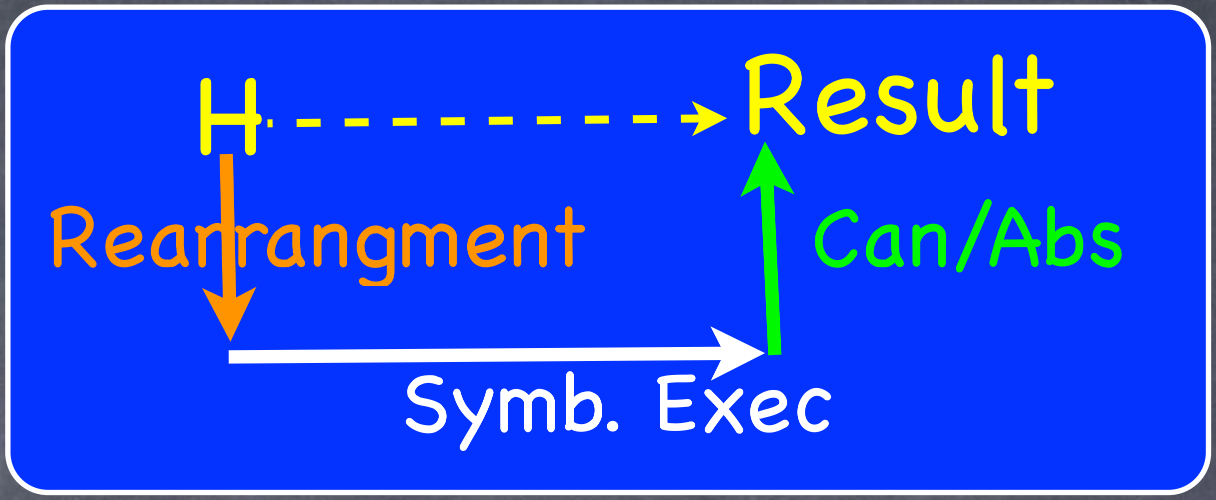
x:=[x]: x=x' | ls(y,x'')*x''|->x'*ls(x',nil)

Complete example



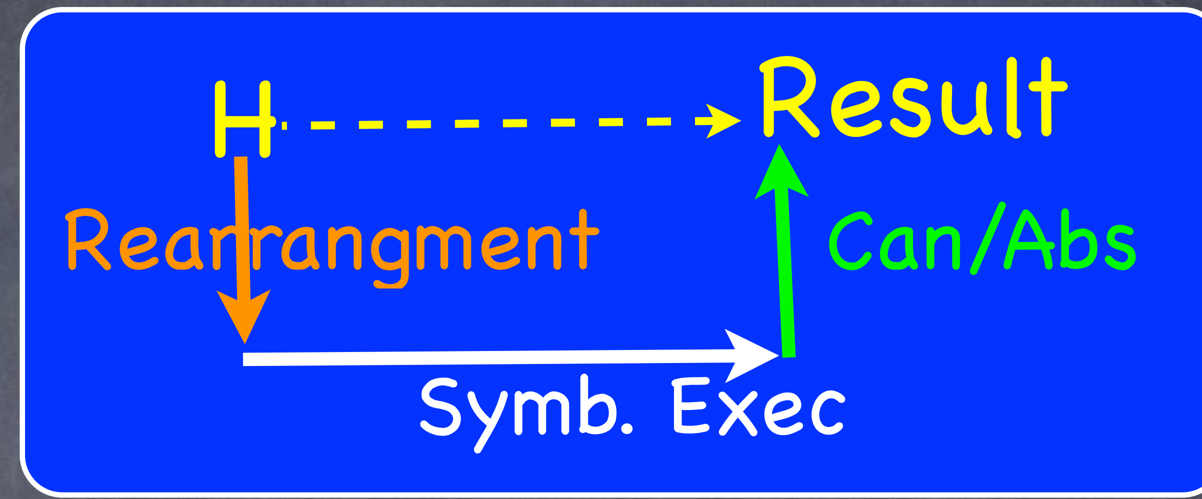
true | ls(y,x)*ls(x,nil)

Complete example



true | ls(y,x)*ls(x,nil)

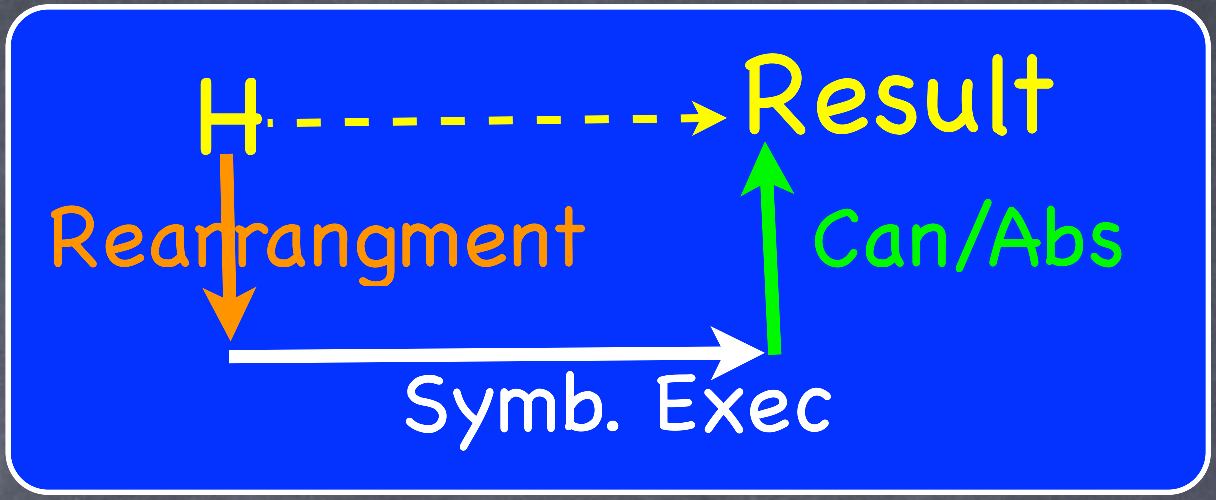
Complete example



true | ls(y,x)*ls(x,nil)

Unroll

Complete example

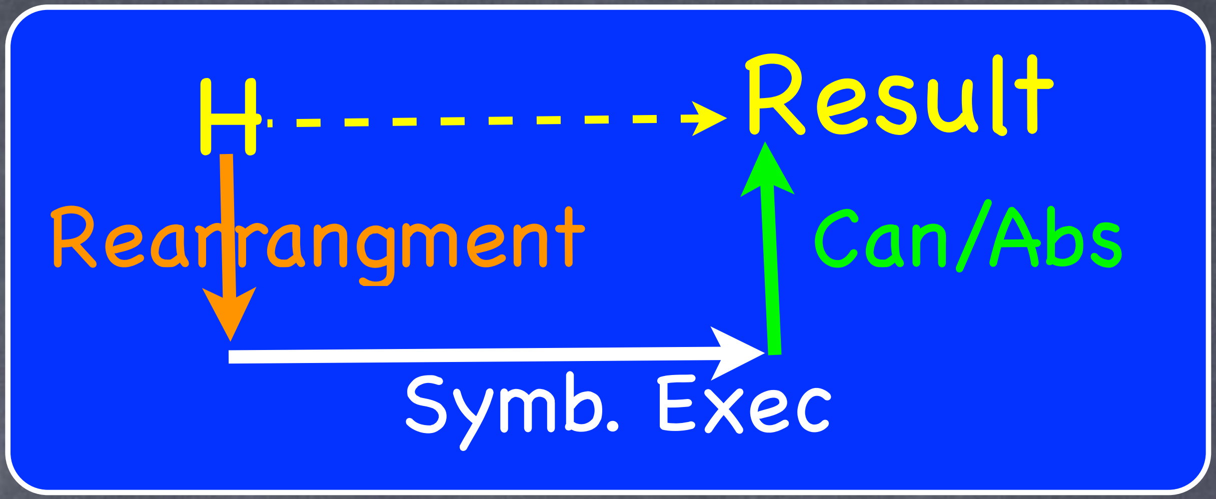


true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

Complete example



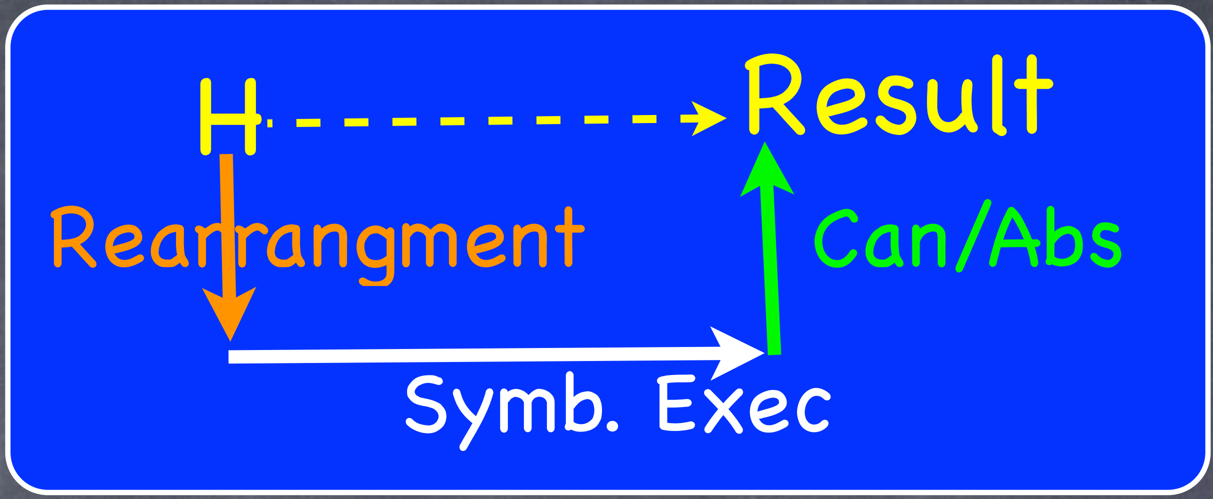
true | ls(y,x)*ls(x,nil)

Unroll

UnrollN

true | ls(y,x)*x|->nil

Complete example



true | ls(y,x)*ls(x,nil)

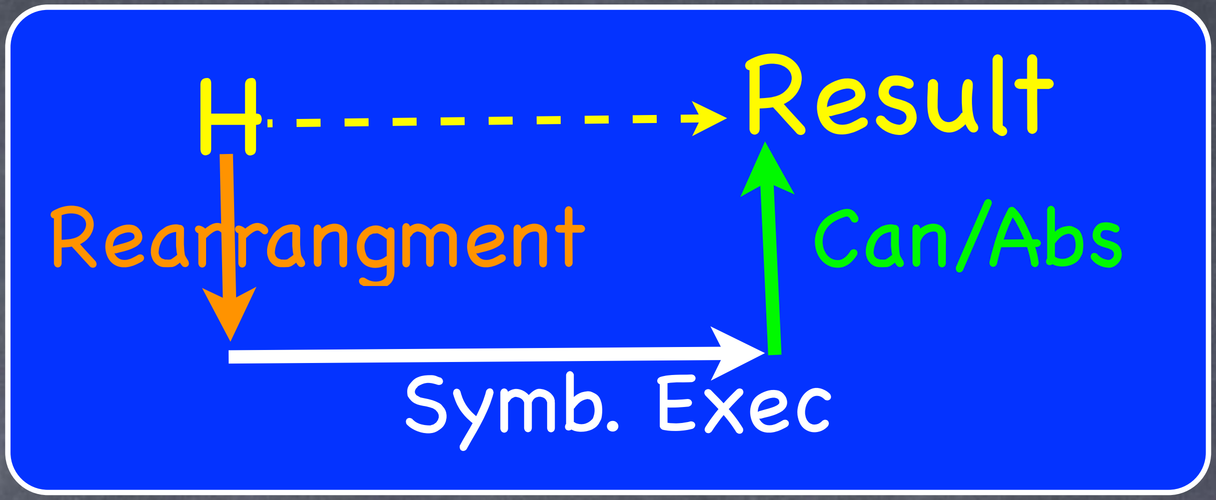
Unroll

true | ls(y,x)*x|->nil

UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

Complete example



true | ls(y,x)*ls(x,nil)

Unroll

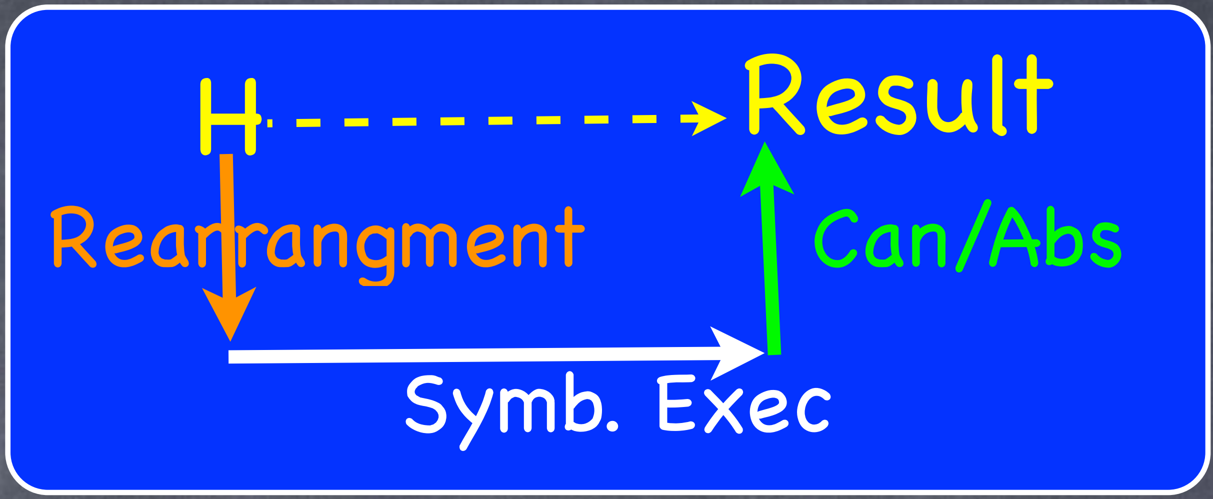
true | ls(y,x)*x|->nil

x:=[x]

UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

Complete example



true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

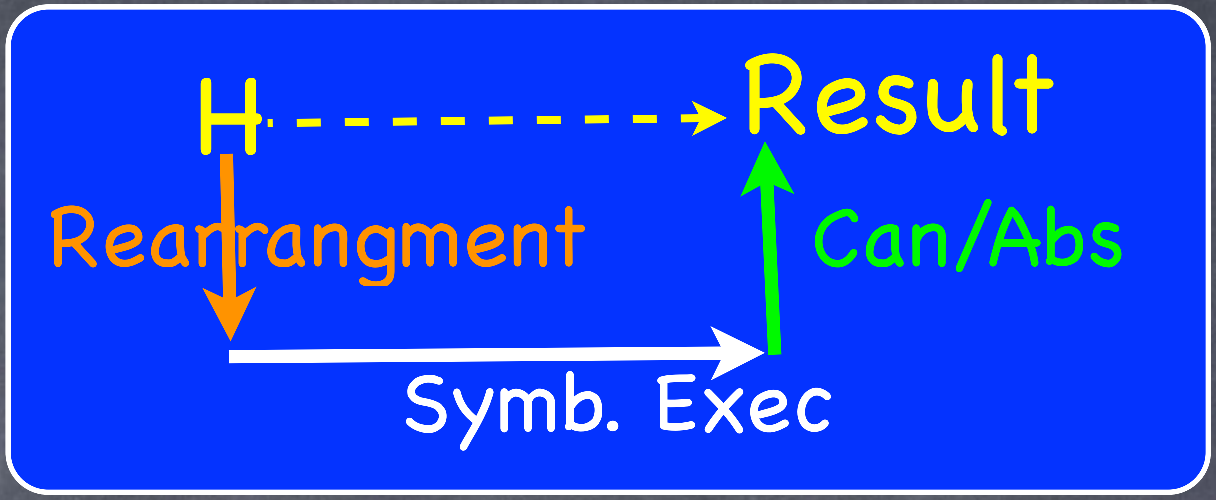
x:=[x]

x=nil | ls(y,x')*x'|->nil

UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

Complete example



true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

x:=[x]

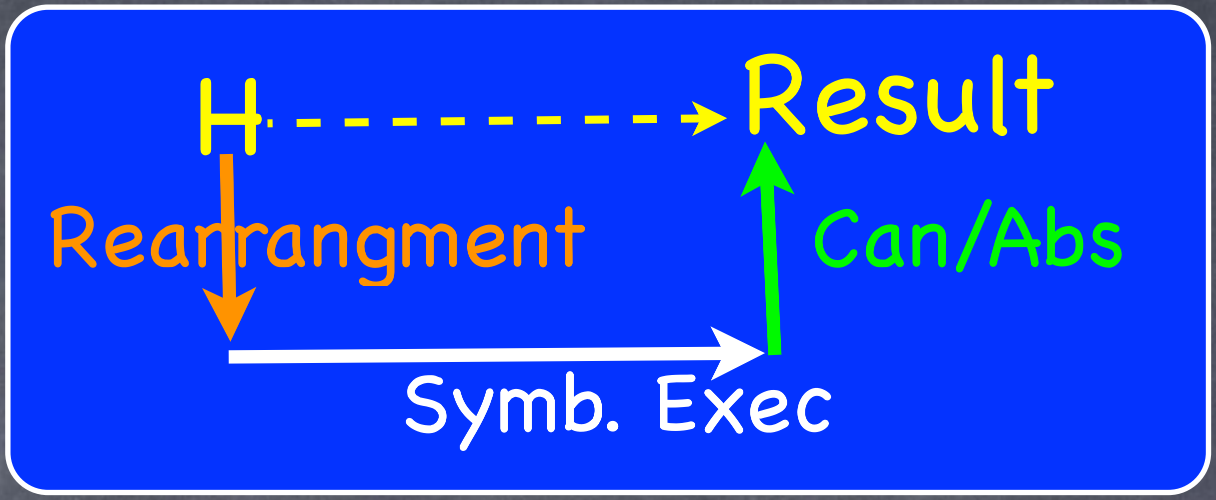
x=nil | ls(y,x')*x'|->nil

UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

x:=[x]

Complete example



true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

x:=[x]

x=nil | ls(y,x')*x'|->nil

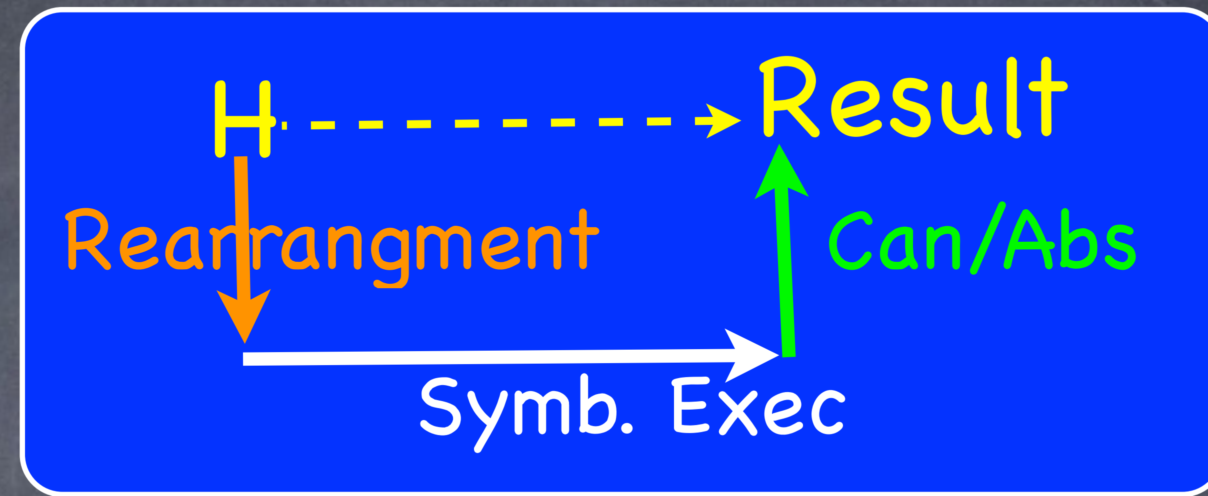
UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

x:=[x]

x=x' | ls(y,x'')*x''|->x'*ls(x',nil)

Complete example



true | ls(y,x)*ls(x,nil)

Unroll

true | ls(y,x)*x|->nil

x:=[x]

x=nil | ls(y,x')*x'|->nil

Abs1

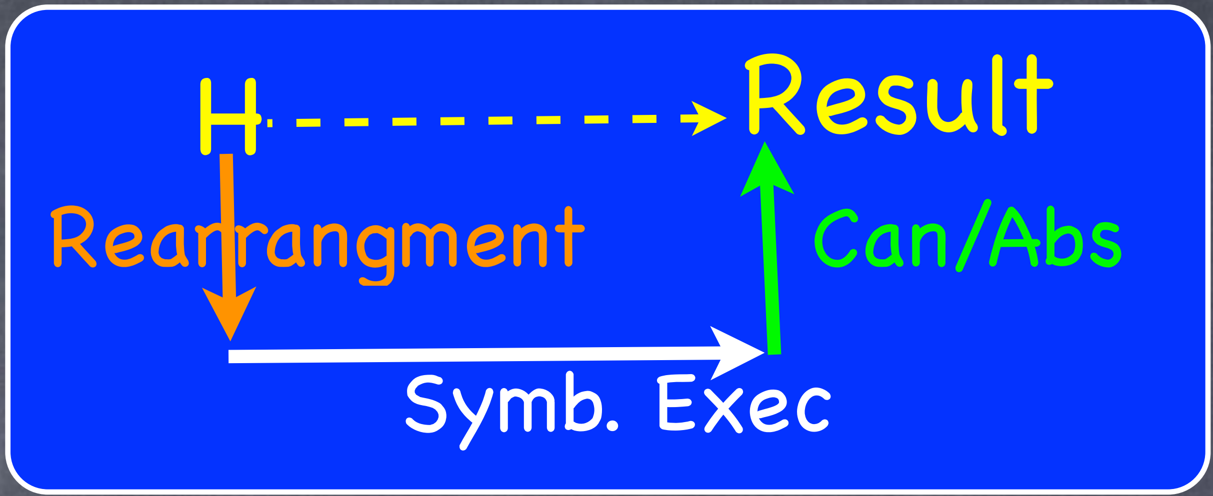
UnrollN

true | ls(y,x)*x|->x'*ls(x',nil)

x:=[x]

x=x' | ls(y,x'')*x''|->x'*ls(x',nil)

Complete example



true | ls(y,x)*ls(x,nil)

Unroll:

true | ls(y,x)*x|->nil

x:=[x]

x=nil | ls(y,x')*x'|->nil

Abs1

x=nil | ls(y,nil)

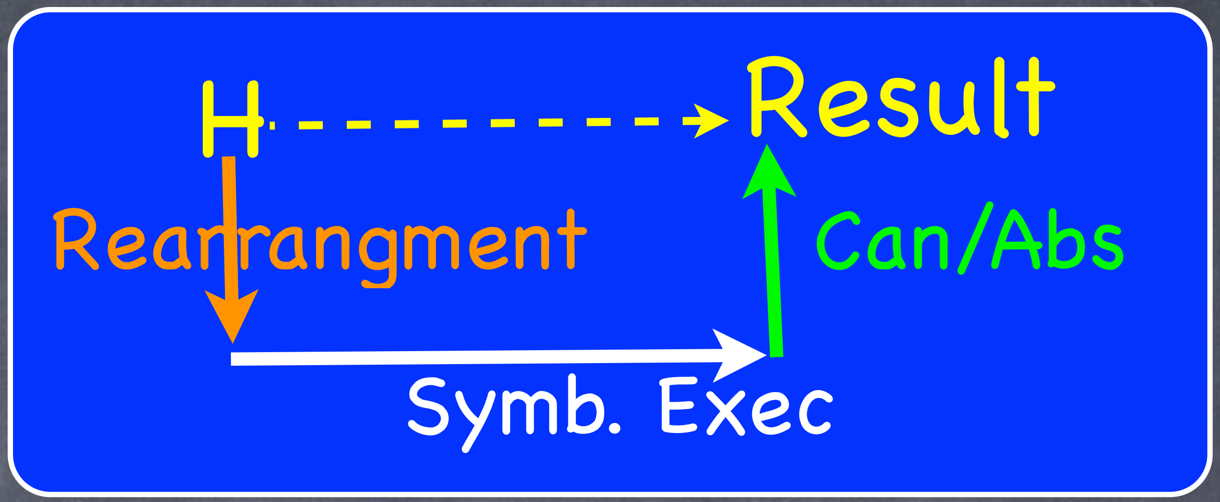
UnrollN:

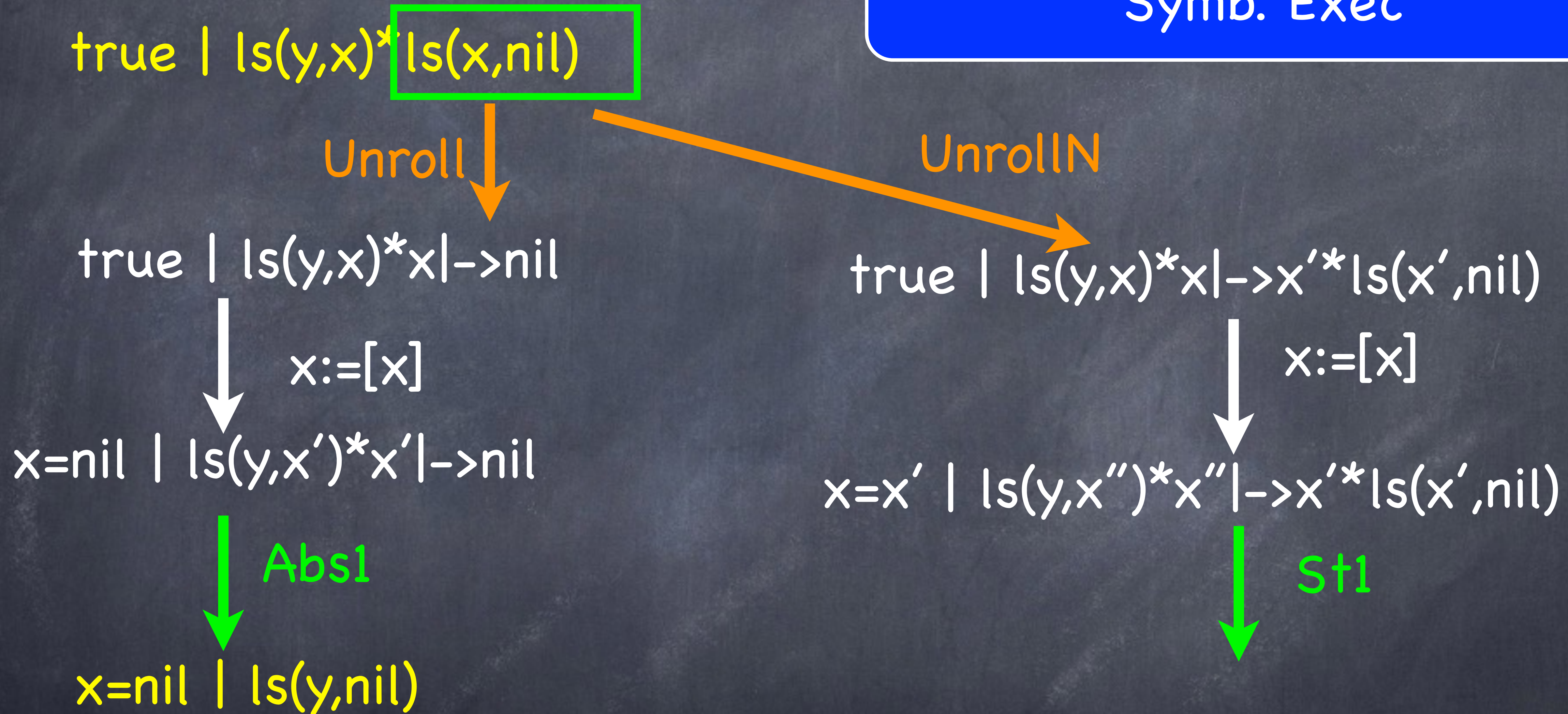
true | ls(y,x)*x|->x'*ls(x',nil)

x:=[x]

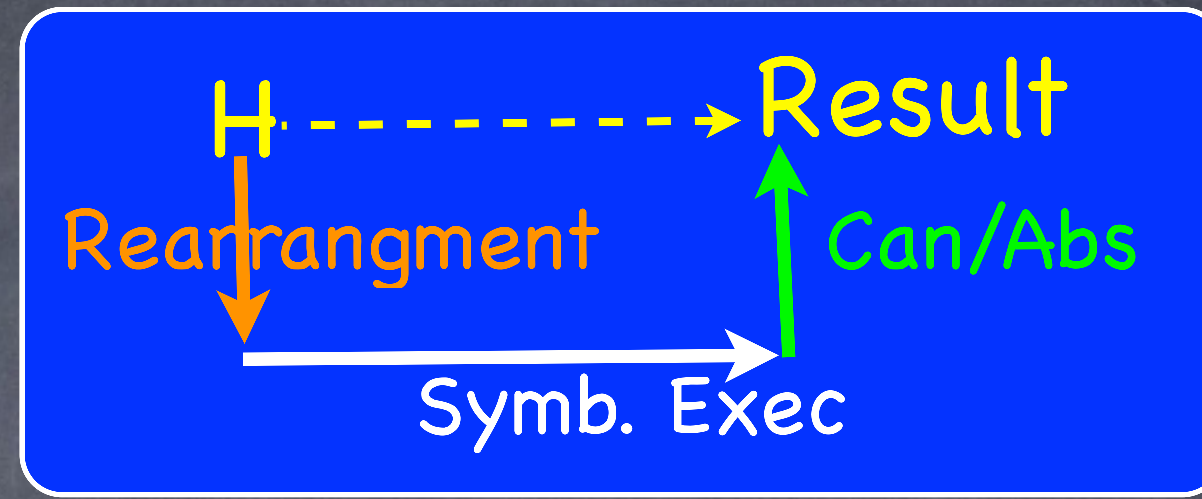
x=x' | ls(y,x")*x"|->x'*ls(x',nil)

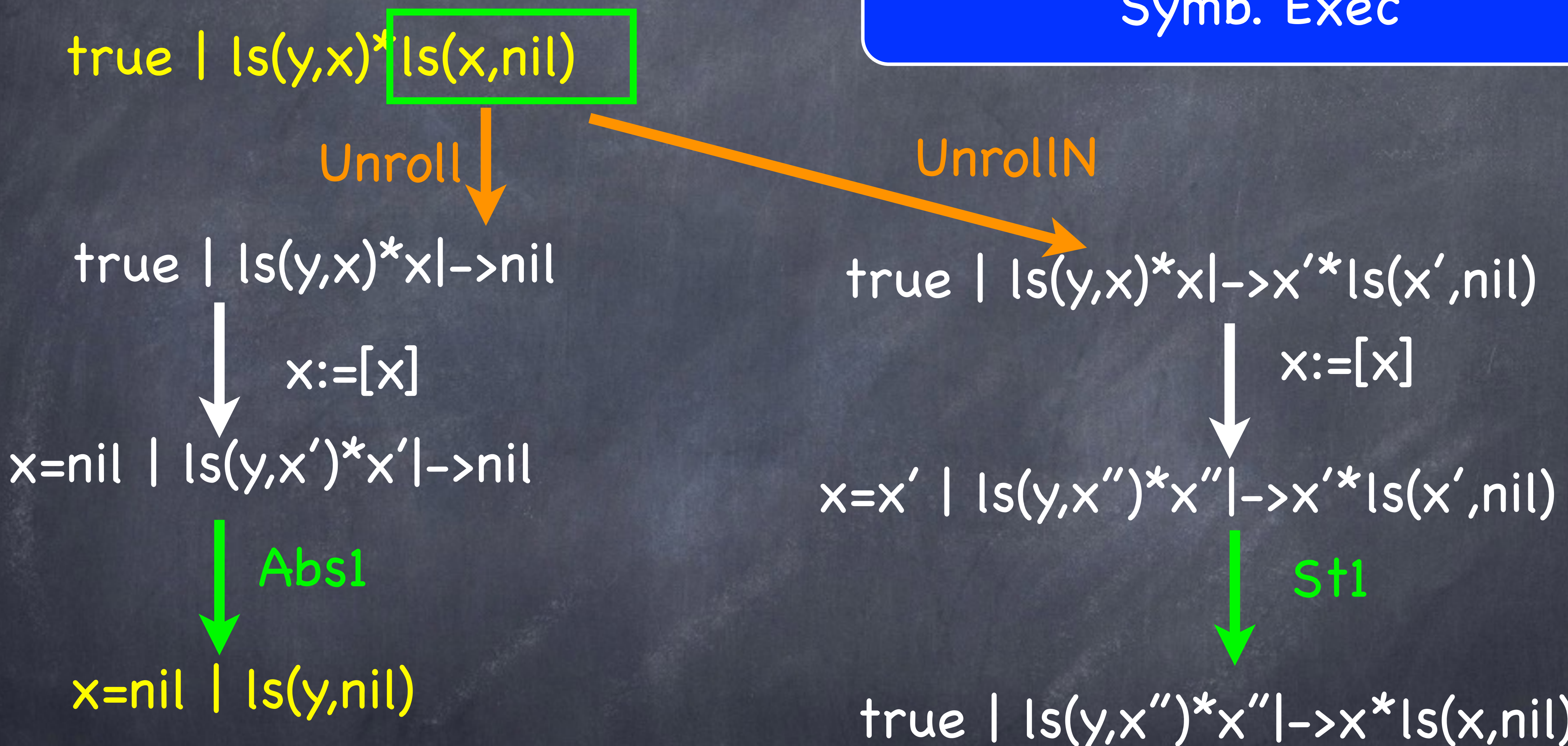
Complete example



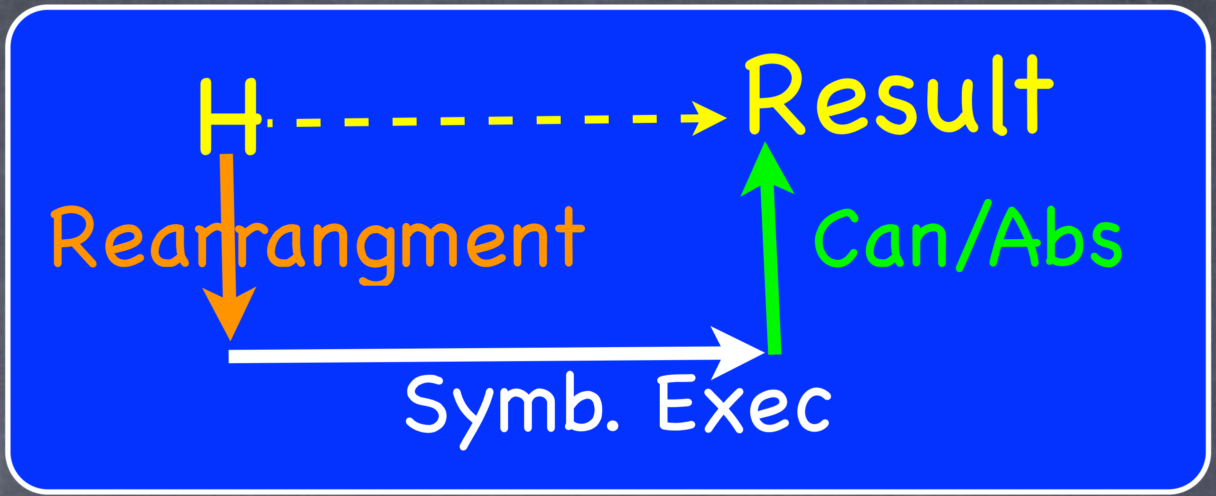


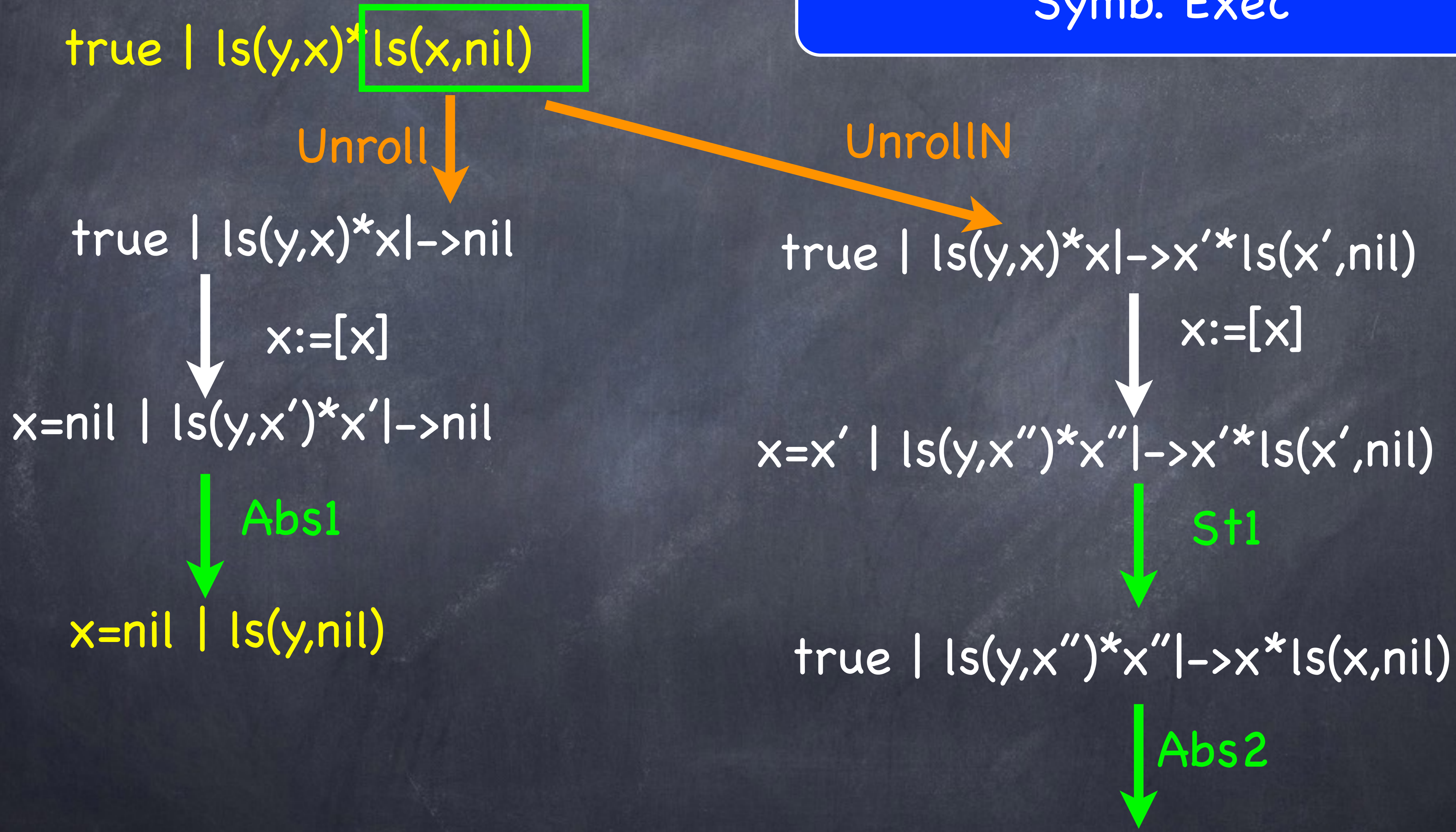
Complete example



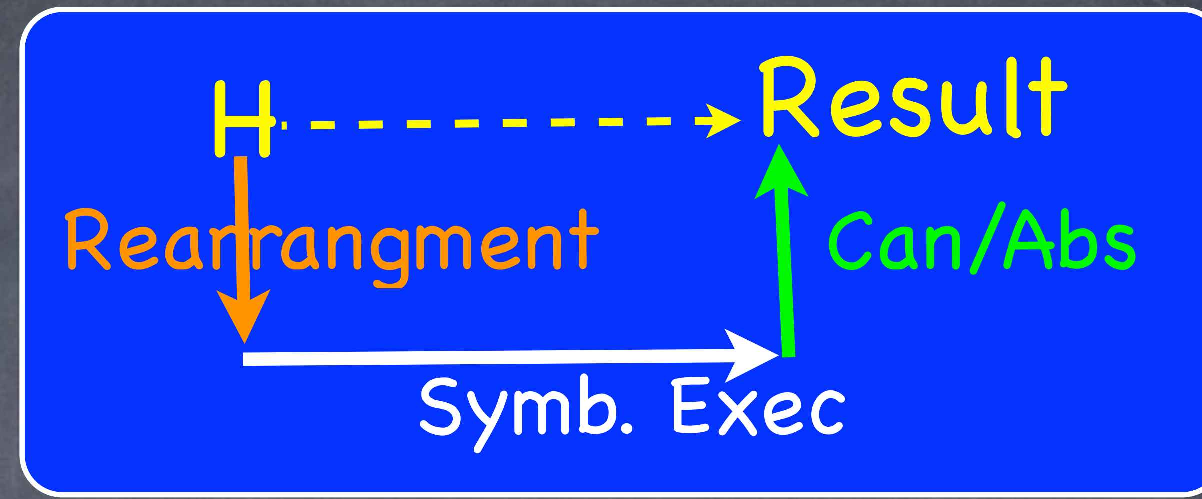


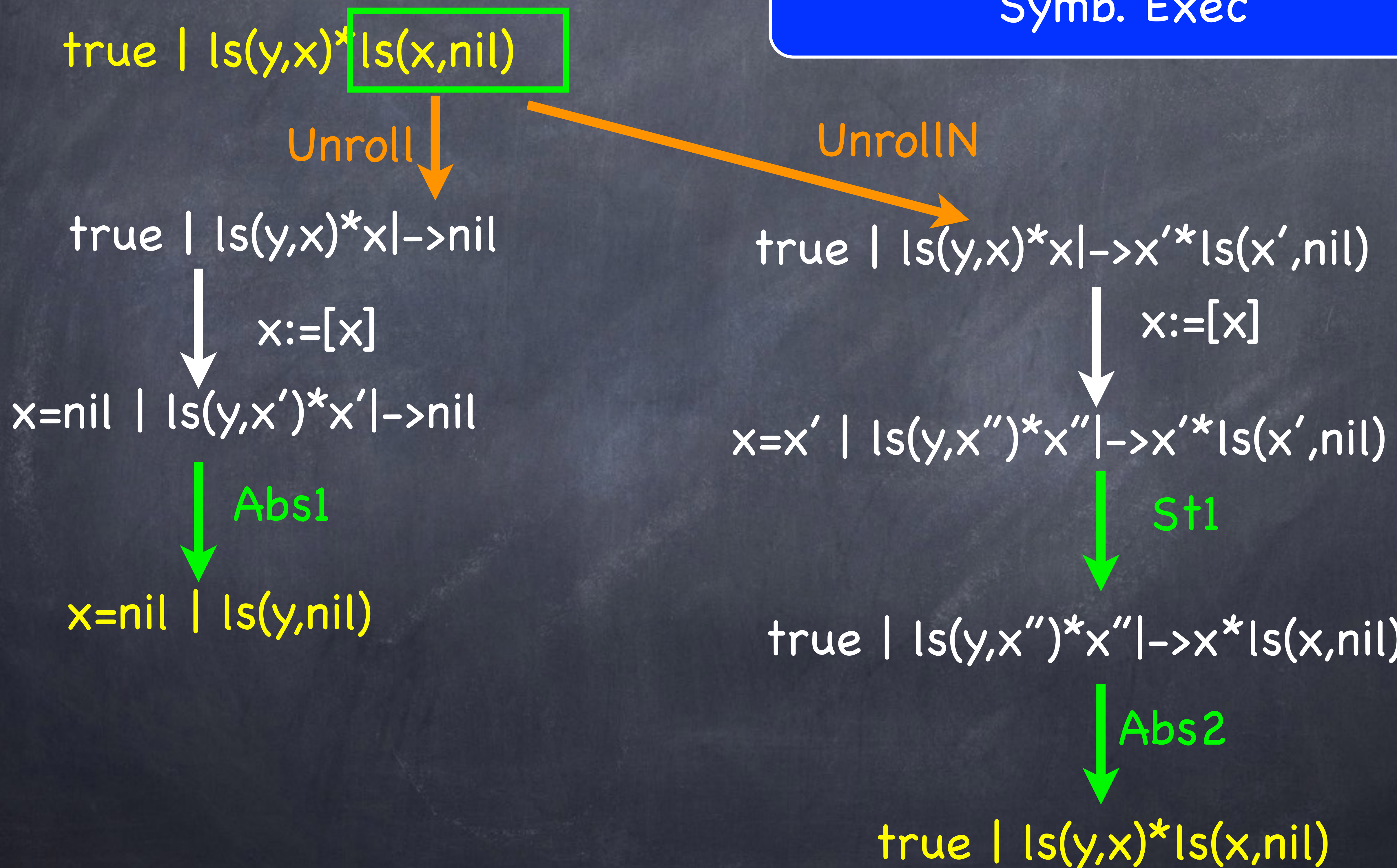
Complete example





Complete example





Properties of the analysis

- The abstract semantics is a sound approximation of the concrete semantics
- The algorithm computing the abstract semantics always terminate

Some references

- D. Distefano, P. O'Hearn, H. Yang: A Local Shape Analysis Based on Separation Logic. TACAS 2006.
- J.Berdine, C. Calcago, P. O'Hearn: Symbolic Execution with Separation Logic. APLAS 2005
- J.C. Reynolds. Separation Logic: A logic for shared mutable data structures. LICS 2002
- S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. POPL 2001.
- C. Calcagno, D. Distefano, P. O'Hearn: Compositional Shape Analysis by means of Bi-Abduction. POPL 2009