# Scalable Shape Analysis For Systems Code

Hongseok Yang (Queen Mary, Univ. of London)

Joint work with
Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook,
Dino Distefano, Peter O'Hearn

# Open Question, Summer 2006

Can we automatically prove the pointer safety of programs $\geq$ 10K in separation logic?

# Demo

# Result

## Designed and implemented a shape analyzer for C.

| Program | LOC | Sec | MB | Memory leaks | Dereference errors |
|---|---|---|---|---|---|
| `scull.c` | 1010 | 0.21 | 1.47 | 1 | 0 |
| `class.c` | 1983 | 6.68 | 8.36 | 2 | 1 |
| `pci-driver.c` | 2532 | 0.79 | 3.19 | 0 | 0 |
| `ll_rw_blk.c` | 5469 | 997.66 | 523.22 | 3 | 1 |
| `cdrom.c` | 6218 | 91.88 | 84.30 | 0 | 2 |
| `md.c` | 6635 | 1440.53 | 814.45 | 6 | 5 |
| `t1394Diag.c` | 10240 | 145.95 | 71.27 | 33 | 10 |

**Table 1.** Experimental Results. Performed on an Intel Core Duo 2GHz with 2GB.

# Resu



Designed and implemented a shape analyzer for C.

| Program | LOC | Sec | MB | Memory leaks | Dereference errors |
|---|---|---|---|---|---|
| `scull.c` | 1010 | 0.21 | 1.47 | 1 | 0 |
| `class.c` | 1983 | 6.68 | 8.36 | 2 | 1 |
| `pci-driver.c` | 2532 | 0.79 | 3.19 | 0 | 0 |
| `ll_rw_blk.c` | 5469 | 997.66 | 523.22 | 3 | 1 |
| `cdrom.c` | 6218 | 91.88 | 84.30 | 0 | 2 |
| `md.c` | 6635 | 1440.53 | 814.45 | 6 | 5 |
| `t1394Diag.c` | 10240 | 145.95 | 71.27 | 33 | 10 |

**Table 1.** Experimental Results. Performed on an Intel Core Duo 2GHz with 2GB.

# Resu

Automatic Pointer-Safety Prover in Sep. Logic

Designed and implemented a shape analyzer for C.

| Program | LOC | Sec | MB | Memory leaks | Dereference errors |
|---|---|---|---|---|---|
| scull.c | 1010 | 0.21 | 1.47 | 1 | 0 |
| class.c | 1983 | 6.68 | 8.36 | 2 | 1 |
| pci-driver.c | 2532 | 0.79 | 3.19 | 0 | 0 |
| ll_rw_blk.c | 5469 | 997.66 | 523.22 | 3 | 1 |
| cdrom.c | 6218 | 91.88 | 84.30 | 0 | 2 |
| md.c | 6635 | 1440.53 | 814.45 | 6 | 5 |
| t1394Diag.c | 10240 | 145.95 | 71.27 | 33 | 10 |

**Table 1.** Experimental Results. Performed on an Intel Core Duo 2GHz with 2GB.

- Programs of 1010 ~ 10240 LOC.
- Includes the full t1394Diag firewire driver.

# Resu

Automatic Pointer-Safety Prover in Sep. Logic

Designed and implemented a shape analyzer for C.

| Program | LOC | Sec | MB | Memory leaks | Dereference errors |
|---|---|---|---|---|---|
| scull.c | 1010 | 0.21 | 1.47 | 1 | 0 |
| class.c | 1983 | 6.68 | 8.36 | 2 | 1 |
| pci-driver.c | 2532 | 0.79 | 3.19 | 0 | 0 |
| ll_rw_blk.c | 5469 | 997.66 | 523.22 | 3 | 1 |
| cdrom.c | 6218 | 91.88 | 84.30 | 0 | 2 |
| md.c | 6635 | 1440.53 | 814.45 | 6 | 5 |
| t1394Diag.c | 10240 | 145.95 | 71.27 | 33 | 10 |

**Table 1.** Experimental Results. Performed on an Intel Core Duo 2GHz with 2GB.

- Found memory leaks and safety errors.
- Fixed those errors. Then, verified the integrity of pointer manipulation.

# Resu

Automatic Pointer-Safety Prover in Sep. Logic

{emp} t1394Diag_main() {emp}

| Program | LOC | Sec | MB | Memory leaks | Dereference errors |
|---|---|---|---|---|---|
| scull.c | 1010 | 0.21 | 1.47 | 1 | 0 |
| class.c | 1983 | 6.68 | 8.36 | 2 | 1 |
| pci-driver.c | 2532 | 0.79 | 3.19 | 0 | 0 |
| ll_rw_blk.c | 5469 | 997.66 | 523.22 | 3 | 1 |
| cdrom.c | 6218 | 91.88 | 84.30 | 0 | 2 |
| md.c | 6635 | 1440.53 | 814.45 | 6 | 5 |
| t1394Diag.c | 10240 | 145.95 | 71.27 | 33 | 10 |

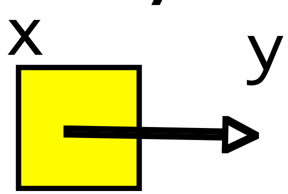
**Table 1.** Experimental Results. Performed on an Intel Core Duo 2GHz with 2GB.

- Found memory leaks and safety errors.
- Fixed those errors. Then, verified the integrity of pointer manipulation.

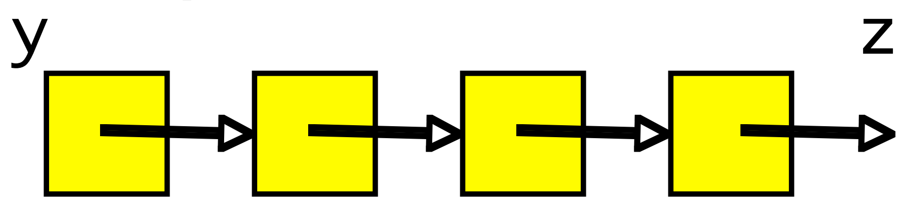
# Separation Logic

$x \mapsto y$

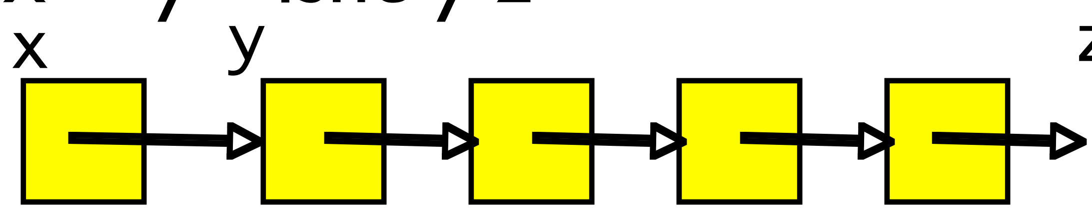
emp




lsne y z




$x \mapsto y$ * lsne y z




$\exists w',v'.\ \ (z=y \wedge w'!=v') \wedge (x \mapsto w' * \text{lsne } w'\ v' * \text{lsne } y\ v')$

# Symbolic Heaps

Separation logic formulas of the form:

$\exists$w’,v’. (y=z$\wedge$w’!=v’) $\wedge$ (x $\mapsto$ w’ * lsne w’ v’ * lsne y v’)

# Symbolic Heaps

Separation logic formulas of the form:

$$(y=z \wedge w'!=v') \wedge (x \mapsto w' * \text{lsne } w'\ v' * \text{lsne } y\ v')$$

# Function "Abs"

$$Abs : SH \rightarrow SH$$

- Forgets the length of lists. Removes unnecessary primed vars.
- Ensures the termination of the analysis.
- E.g. $z!=x' \wedge (x \mapsto x' * x' \mapsto 0 * lsne\ y\ y' * lsne\ y'\ 0)$.

# Function "Abs"

$$Abs : SH \rightarrow SH$$

- Forgets the length of lists. Removes unnecessary primed vars.
- Ensures the termination of the analysis.
- E.g. z!=x' ∧ (lsne x 0 * lsne y y' * lsne y' 0).

# Function "Abs"

$$Abs : SH \to SH$$

- Forgets the length of lists. Removes unnecessary primed vars.
- Ensures the termination of the analysis.
- E.g. z!=x' ∧ (lsne x 0 * lsne y 0).

# Function "Abs"

$$\text{Abs} : \text{SH} \rightarrow \text{SH}$$

- Forgets the length of lists. Removes unnecessary primed vars.
- Ensures the termination of the analysis.
- E.g. (lsne x 0 * lsne y 0).

# Shape Analysis

- Run a program symbolically with symbolic heaps.
- Apply abstraction periodically to ensure termination.
- Repeat until the fixpoint is reached.

# Interprocedural Shape Analysis

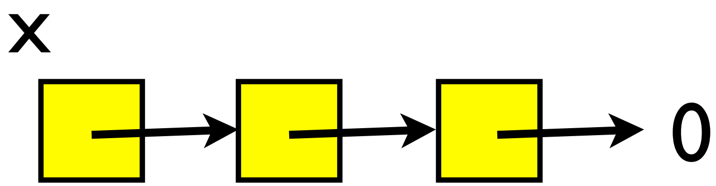
- For each procedure, create a table that records all the past analysis results.
- Table for create():

| Precond. | Postcondition |
|---|---|
| emp | $ret=0 \wedge emp$, $ret \mapsto 0$, ls ret 0 |
| ... | ... |

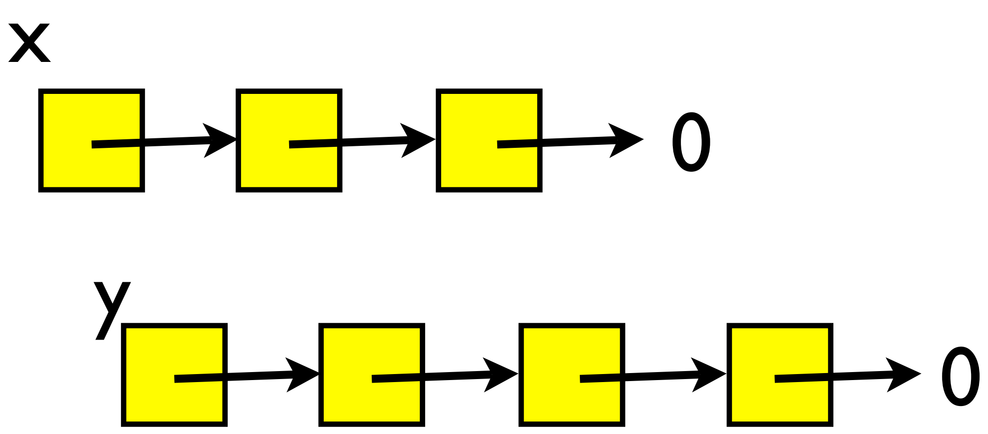
- Use the table whenever possible.
- The implementation is based on a more sophisticated algorithm (RHS).

```
L create() {...}



L append(L a,L b) {...}






void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

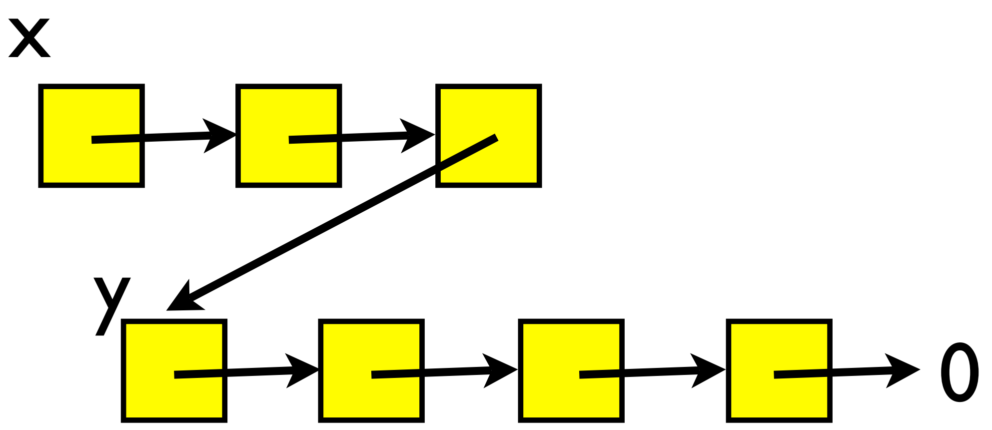
```
L create() {...}


L append(L a,L b) {...}


void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```



```
L create() {...}



L append(L a,L b) {...}






void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```



```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);
}
```



```
L create() {...}



L append(L a,L b) {...}





void main() {   emp
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

| emp | ret=0 ∧ emp, ret↦0, lsne ret 0 |
| --- | --- |

```
L create() {...}

L append(L a,L b) {...}

void main() {      emp
 x=create();
 y=create();
 x=append(x,y);

}
```

| emp | ret=0 ∧ emp, ret↦0, lsne ret 0 |
| --- | --- |

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

x=0 ∧ emp

x↦0

lsne x 0

| emp | ret=0 ∧ emp, ret↦0, lsne ret 0 |
|---|---|
| x=0 ∧ emp | |
| x↦0 | |
| lsne x 0 | |

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

x=0 ∧ emp    x↦0    lsne x 0

|  |  |  |  |
|---|---|---|---|
| emp | ret=0 ∧ emp, | ret↦0, | lsne ret 0 |
| x=0 ∧ emp | x=ret=0 ∧ emp, | x=0 ∧ ret↦0, | x=0 ∧ lsne ret 0 |
| x↦0 | ret=0 ∧ x↦0, | x↦0 * ret↦0, | x↦0 * lsne ret 0 |
| lsne x 0 | ret=0∧lsne x 0, | lsne x 0*ret↦0, | lsne x 0*lsne ret |

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

x=0 ∧ emp

x↦0

lsne x 0

| | |
|---|---|
| emp | ret=0 ∧ emp, ret↦0, lsne ret 0 |
| x=0 ∧ emp | x=ret=0 ∧ emp, x=0 ∧ ret↦0, x=0 ∧ lsne ret 0 |
| x↦0 | ret=0 ∧ x↦0, x↦0 * ret↦0, x↦0 * lsne ret 0 |
| lsne x 0 | ret=0∧lsne x 0, lsne x 0*ret↦0, lsne x 0*lsne ret |

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

x=0 ∧ emp    x↦0    lsne x 0

x=y=0 ∧ emp ... x↦0*y↦0    lsnex0*lsney0 ...    lsnex0*lsney0

| | | | |
|---|---|---|---|
| emp | ret=0 ∧ emp, | ret↦0, | lsne ret 0 |
| x=0 ∧ emp | x=ret=0 ∧ emp, | x=0 ∧ ret↦0, | x=0 ∧ lsne ret 0 |
| x↦0 | ret=0 ∧ x↦0, | x↦0 * ret↦0, | x↦0 * lsne ret 0 |
| lsne x 0 | ret=0∧lsne x 0, | lsne x 0*ret↦0, | lsne x 0*lsne ret |

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

x=0 ∧ emp    x↦0    lsne x 0

x=y=0 ∧ emp  ...  x↦0*y↦0    lsne x 0 * lsne y 0  ...  lsne x 0 * lsne y 0

| | |
|---|---|
| emp | ret=0 $\wedge$ emp, ret $\mapsto$ 0, lsne ret 0 |
| x=0 $\wedge$ emp | x=ret=0 $\wedge$ emp, x=0 $\wedge$ ret $\mapsto$ 0, x=0 $\wedge$ lsne ret 0 |
| x $\mapsto$ 0 | ret=0 $\wedge$ x $\mapsto$ 0, x $\mapsto$ 0 * ret $\mapsto$ 0, x $\mapsto$ 0 * lsne ret 0 |
| lsne x 0 | ret=0 $\wedge$ lsne x 0, lsne x 0*ret $\mapsto$ 0, lsne x 0*lsne ret |

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);
}
```

emp

x=0 $\wedge$ emp | x $\mapsto$ 0 | lsne x 0

x=y=0 $\wedge$ emp ... x $\mapsto$ 0*y $\mapsto$ 0 | lsne x 0 * lsne y 0 ... lsne x 0 * lsne y 0

x=y=0 $\wedge$ emp ... x $\mapsto$ y*y $\mapsto$ 0 | lsne x y * lsne y 0 ... lsne x y * lsne y 0

L create() {...}

| | |
|---|---|
| emp | ret=0 ∧ emp, ret↦0, lsne ret 0 |
| x=0 ∧ emp | x=ret=0 ∧ emp, x=0 ∧ ret↦0, x=0 ∧ lsne ret 0 |
| x↦0 | ret=0 ∧ x↦0, x↦0 * ret↦0, x↦0 * lsne ret 0 |
| lsne x 0 | ret=0∧lsne x 0, lsne x 0*ret↦0, lsne x 0*lsne ret |

**4 entries, 12 results**

L append(L a,L b) {...}

**9 entries, 12 results**

void main() {

emp

x=create();

x=0 ∧ emp | x↦0 | lsne x 0

y=create();

x=y=0 ∧ emp ... x↦0*y↦0 | lsnex0*lsney0 ... lsnex0*lsney0

x=append(x,y);

x=y=0 ∧ emp ... x↦y*y↦0 | lsnexy * lsney0 ... lsnexy*lsney0

}

| | |
|---|---|
| emp | $ret=0 \land emp$, $ret \mapsto 0$, lsne ret 0 |
| $x=0 \land emp$ | $x=ret=0 \land emp$, $x=0 \land ret \mapsto 0$, $x=0 \land$ lsne ret 0 |
| $x \mapsto 0$ | $ret=0 \land x \mapsto 0$, $x \mapsto 0 * ret \mapsto 0$, $x \mapsto 0 *$ lsne ret 0 |
| lsne x 0 | $ret=0 \land$ lsne x 0, lsne x $0 * ret \mapsto 0$, lsne x 0*lsne ret |

4 entries, 12 results

```
L create() {...}

L append(L a,L b) {...}
```

9 entries, 12 results

```
void main() {
 x=create();
 y=create();
 x=append(x,y);
}
```

emp

$x=0 \land emp$ | $x \mapsto 0$ | lsne x 0

$x=y=0 \land emp$ ... $x \mapsto 0 * y \mapsto 0$ | lsne x 0 * lsne y 0 ... lsne x 0 * lsne y 0

$x=y=0 \land emp$ ... $x \mapsto y * y \mapsto 0$ | lsne x y * lsne y 0 ... lsne x y * lsne y 0

| | | | |
|---|---|---|---|
| emp | ret=0 ∧ emp, | ret↦0, | lsne ret 0 |
| x=0 ∧ emp | x=ret=0 ∧ emp, | x=0 ∧ ret↦0, | x=0 ∧ lsne ret 0 |
| x↦0 | ret=0 ∧ x↦0, | x↦0 * ret↦0, | x↦0 * lsne ret 0 |
| lsne x 0 | ret=0∧lsne x 0, | lsne x 0*ret↦0, | lsne x 0*lsne ret |

4 entries, 12 results

```
L create() {...}

L append(L a, L b) {...}
```

9 entries, 12 results

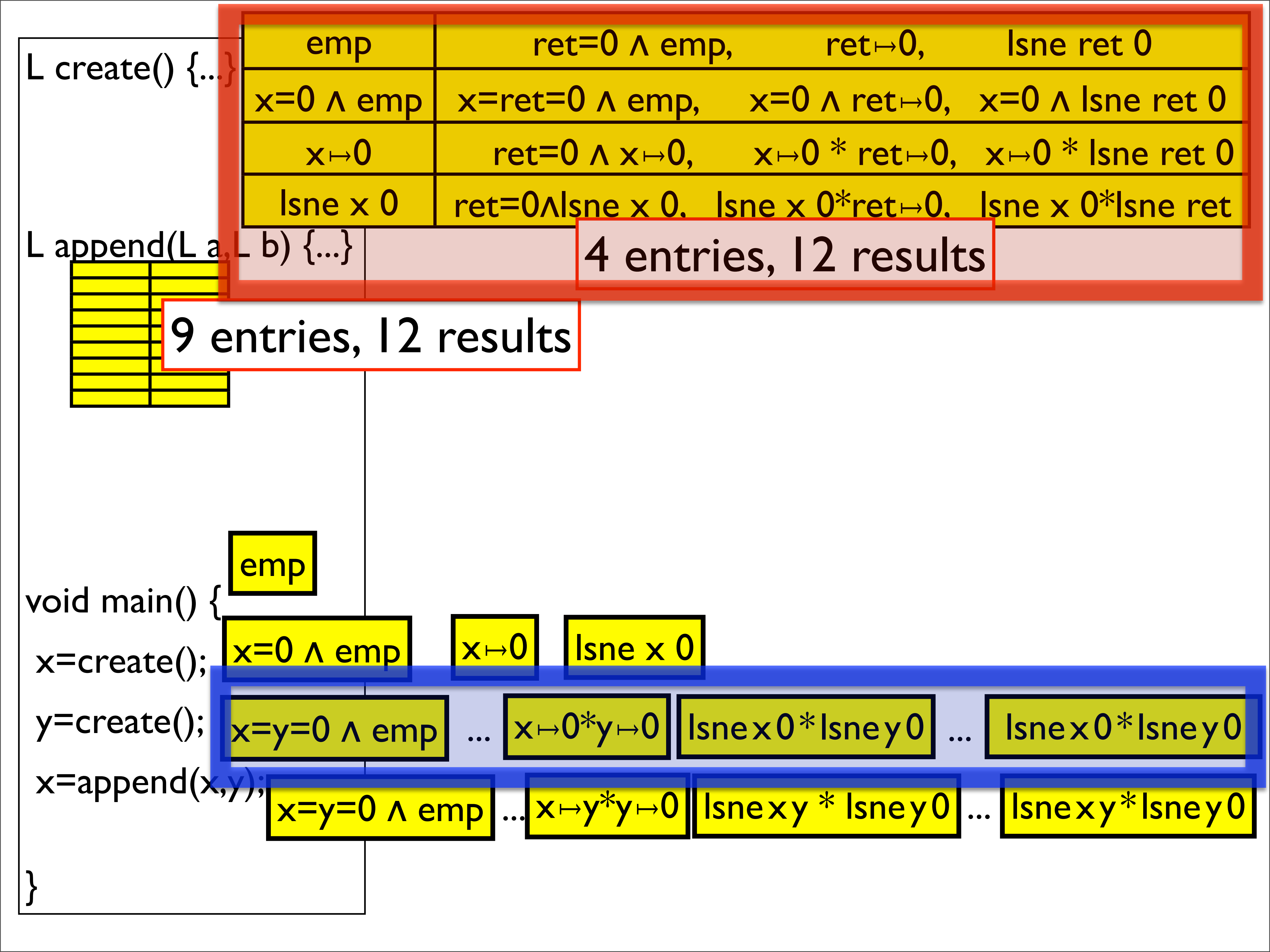
```
void main() {
  x=create();
  y=create();
  x=append(x,y);
}
```

emp

x=0 ∧ emp  x↦0  lsne x 0

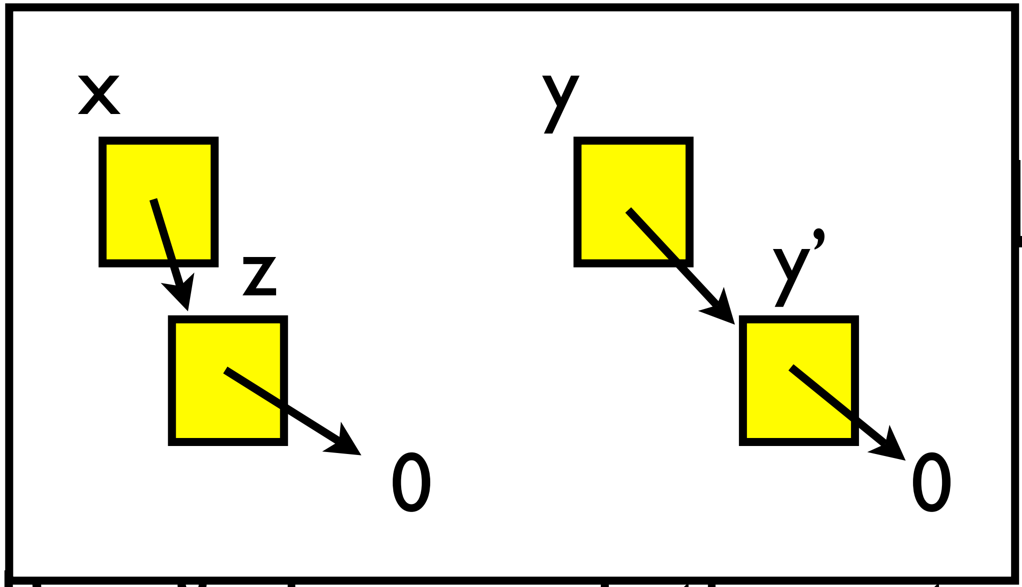
x=y=0 ∧ emp ... x↦0*y↦0  lsnex0*lsney0 ... lsnex0*lsney0

x=y=0 ∧ emp ... x↦y*y↦0  lsnexy * lsney0 ... lsnexy*lsney0


L create() {...}
emp | ret=0 ∧ emp, ret↦0, lsne ret 0
x=0 ∧ emp | x=ret=0 ∧ emp, x=0 ∧ ret↦0, x=0 ∧ lsne ret 0
x↦0 | ret=0 ∧ x↦0, x↦0 * ret↦0, x↦0 * lsne ret 0
lsne x 0 | ret=0∧lsne x 0, lsne x 0*ret↦0, lsne x 0*lsne ret
4 entries, 12 results
L append(L a, L b) {...}
9 entries, 12 results
void main() {
emp
x=create();
x=0 ∧ emp
x↦0
lsne x 0
y=create();
x=y=0 ∧ emp ... x↦0*y↦0 lsnex0*lsney0 ... lsnex0*lsney0
x=append(x,y);
x=y=0 ∧ emp ... x↦y*y↦0 lsnexy * lsney0 ... lsnexy*lsney0
}

# Optimization 1: Localization

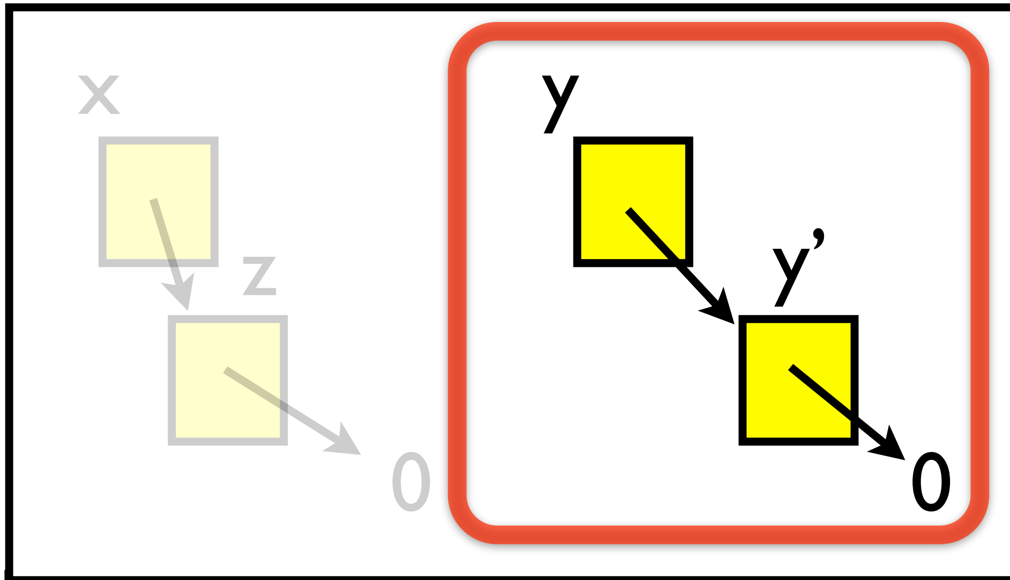
- Pass & change only the part of a symbolic heap, that is reachable from the parameters. [Rinetzky+POPL05, Gotsman+SAS06.]
- Based on frame rule in separation logic. [CSL01]
- E.g.
  - lsne x z * z $\mapsto$0 * lsne y y' * lsne y' 0,      dispose(y)



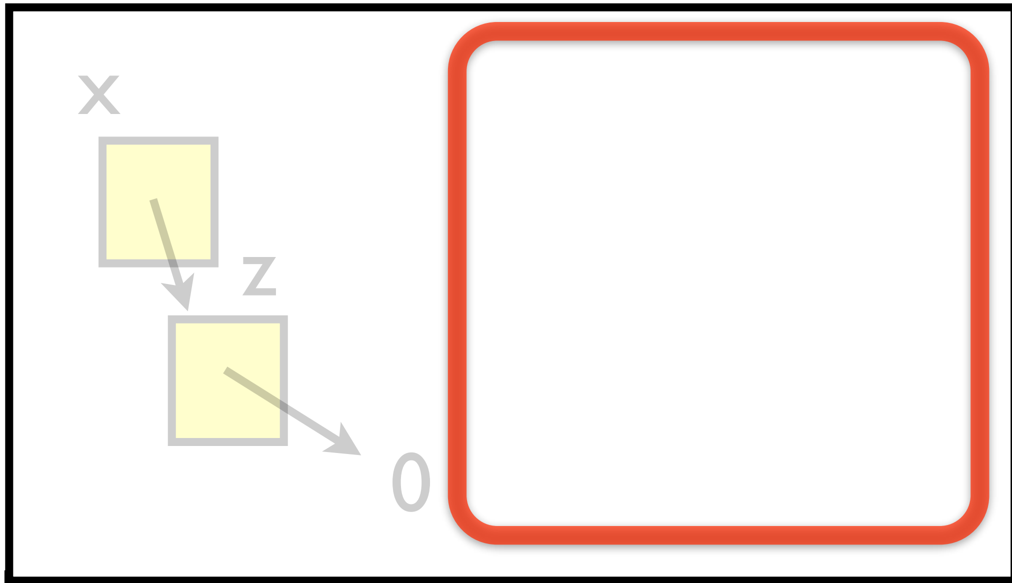

# Localization

- Pass & change only the part of a symbolic heap, that is reachable from the parameters. [Rinetzky+POPL05, Gotsman+SAS06]
- Based on frame rule in separation logic. [CSL01]
- E.g.
  - lsne x z * z ↦ 0 * lsne y y' * lsne y' 0,       dispose(y)

# Localization

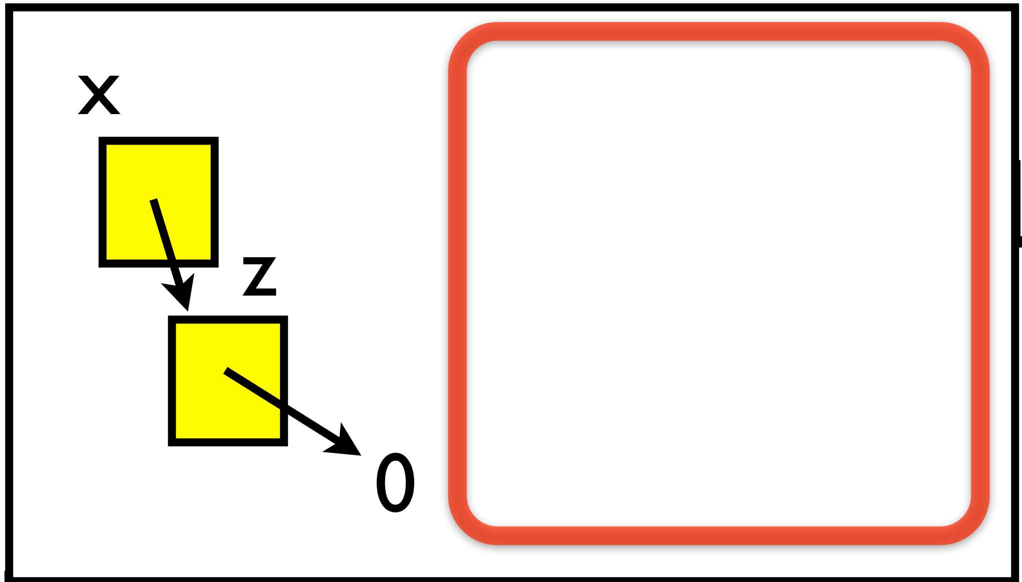



- Pass & change only the part of a symbolic heap, that is reachable from the parameters. [Rinetzky+POPL05, Gotsman+SAS06]
- Based on frame rule in separation logic. [CSL01]
- E.g.
  - lsne x z * z ↦ 0 * lsne y y' * lsne y' 0,     dispose(y)



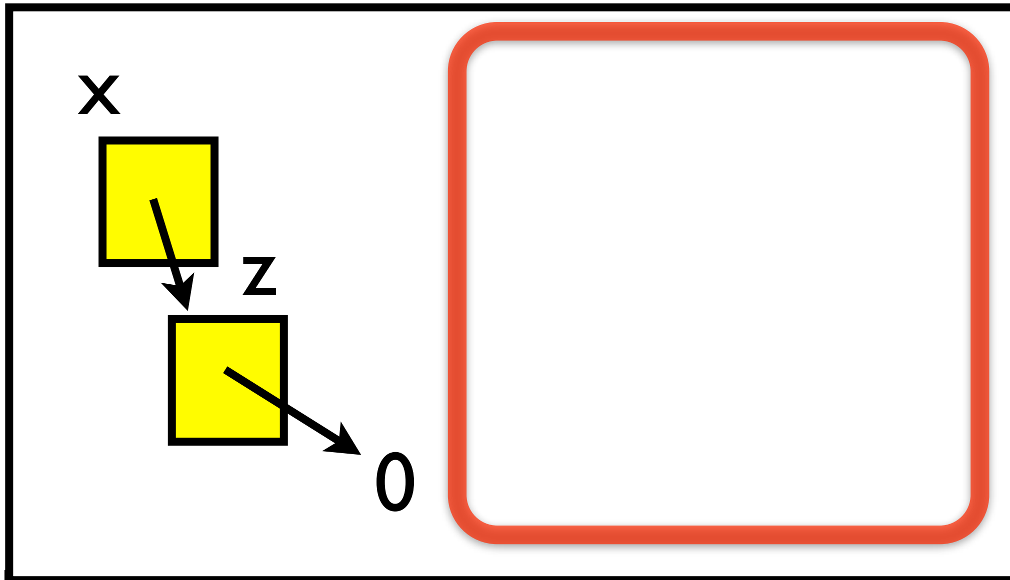

# Localization

- Pass & change only the part of a symbolic heap, that is reachable from the parameters. [Rinetzky+POPL05, Gotsman+SAS06]
- Based on frame rule in separation logic. [CSL01]
- E.g.
  - lsne x z * z↦0 * lsne y y’ * lsne y’ 0,     dispose(y)




# Localization

- Pass & change only the part of a symbolic heap, that is reachable from the parameters. [Rinetzky+POPL05, Gotsman+SAS06]
- Based on frame rule in separation logic. [CSL01]
- E.g.
  - lsne x z * z$\mapsto$0 * lsne y y' * lsne y' 0,       dispose(y)




| Pre | Post |
|---|---|
| lsne y y' * lsne y' 0 | emp |

- Pass & change only the part of a symbolic heap, that is reachable from the parameters. [Rinetzky+POPL05, Gotsman+SAS06]
- Based on frame rule in separation logic. [CSL01]
- E.g.
  - lsne x z * z ↦ 0 * lsne y y' * lsne y' 0,     dispose(y)

# Symbolic Heaps with Possibly Empty List-seg lspe

$$
\begin{aligned}
e &::= x \mid x' \mid \mathsf{nil} \\
\Pi &::= \Pi \wedge \Pi \mid e{=}e \mid e{\neq}e \mid \mathsf{true} \\
\Sigma &::= \Sigma * \Sigma \mid \mathsf{emp} \mid (e \mapsto e) \mid \mathsf{lsne}\ e\ e \mid \mathsf{lspe}\ e\ e \mid \mathsf{true} \\
q &::= \Pi \wedge \Sigma
\end{aligned}
$$

# Symbolic Heaps with Possibly Empty List-seg lspe

$$
\begin{aligned}
e &::= x \mid x' \mid \mathsf{nil} \\
\Pi &::= \Pi \wedge \Pi \mid e{=}e \mid e{\neq}e \mid \mathsf{true} \\
\Sigma &::= \Sigma * \Sigma \mid \mathsf{emp} \mid (e \mapsto e) \mid \mathsf{lsne}\ e\ e \mid \boxed{\mathsf{lspe}\ e\ e} \mid \mathsf{true} \\
q &::= \Pi \wedge \Sigma
\end{aligned}
$$

# Optimization 2: Partial Join Operator

- pjoin : SH x SH $\rightharpoonup$ SH
- Overapproximation :

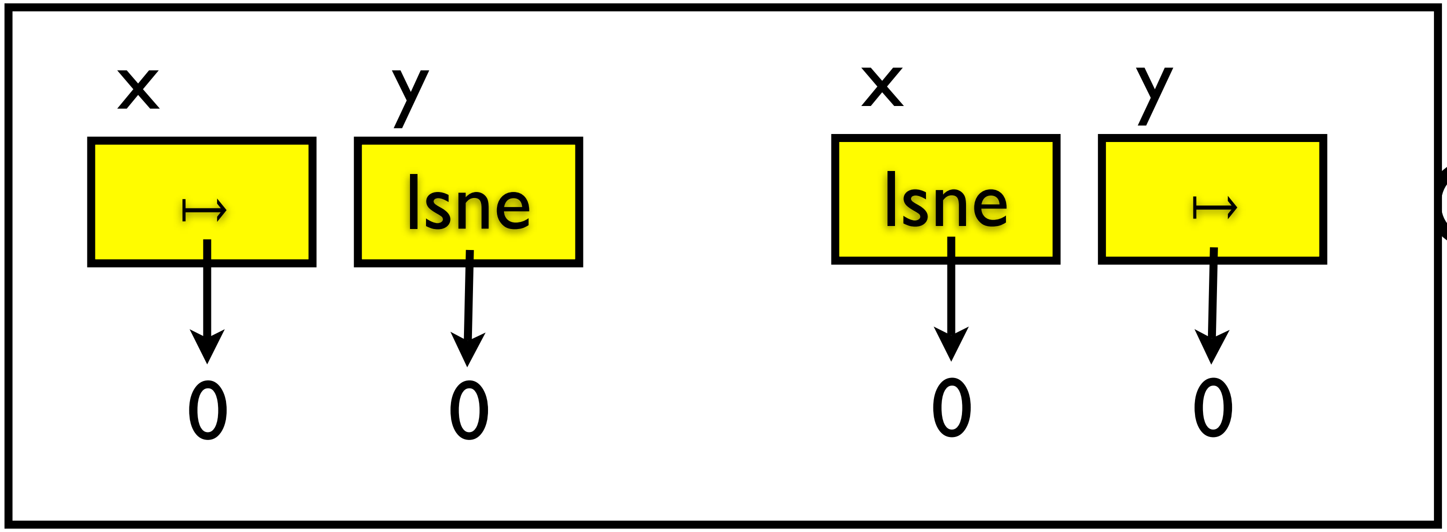
  If pjoin(q, q')=q'', then $(q \vee q') \Rightarrow q''$

- E.g.

  pjoin(x$\mapsto$0 * lsne y 0, lsne x 0 * y$\mapsto$0) = lsne x 0 * lsne y 0

  pjoin(x$\mapsto$y * lsne y 0, lsne x 0 * y$\mapsto$0) = undefined.

  pjoin(y=0 $\wedge$ lsne x y, x$\mapsto$y * lsne y 0) = lsne x y * lspe y 0




# Operator

- Overapproximation :

  If pjoin(q, q')=q'', then $(q \vee q') \Rightarrow q''$

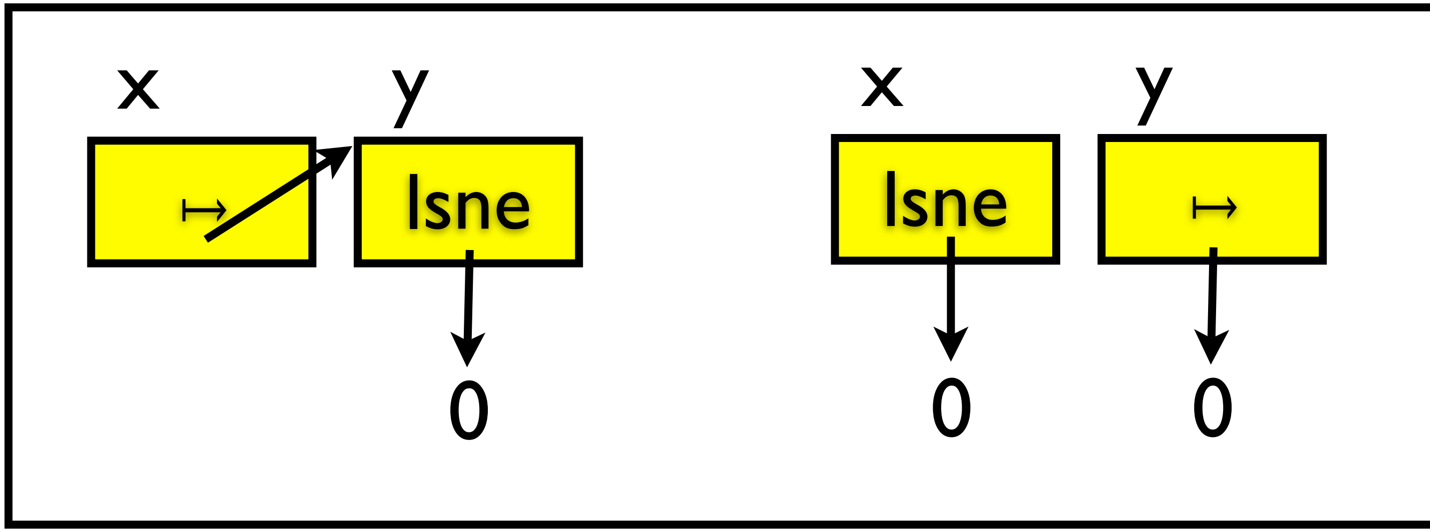
- E.g.

  pjoin(x↦0 * lsne y 0, lsne x 0 * y↦0) = lsne x 0 * lsne y 0

  pjoin(x↦y * lsne y 0, lsne x 0 * y↦0) = undefined.

  pjoin(y=0 $\wedge$ lsne x y, x↦y * lsne y 0) = lsne x y * lspe y 0

# Operator




- Overapproximation :

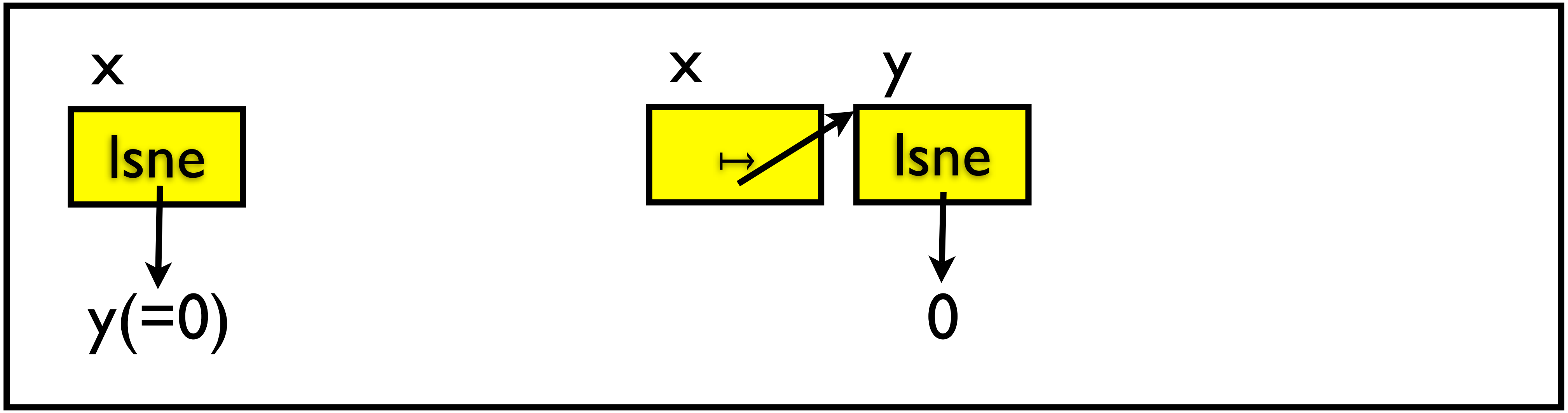
  If pjoin(q, q')=q'', then $(q \vee q') \Rightarrow q''$

- E.g.

  pjoin($x \mapsto 0$ * lsne y 0, lsne x 0 * $y \mapsto 0$) = lsne x 0 * lsne y 0

  pjoin($x \mapsto y$ * lsne y 0, lsne x 0 * $y \mapsto 0$) = undefined.

  pjoin($y=0 \wedge$ lsne x y, $x \mapsto y$ * lsne y 0) = lsne x y * lspe y 0

- 


- Overapproximation :

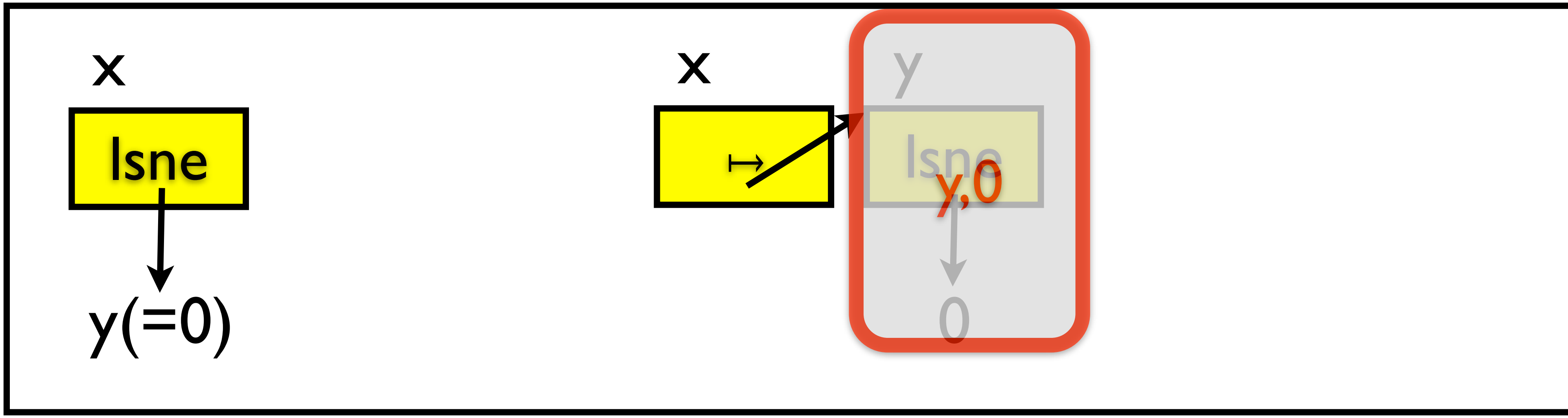
  If pjoin(q, q')=q'', then $(q \lor q') \Rightarrow q''$

- E.g.

pjoin(x↦0 * lsne y 0, lsne x 0 * y↦0) = lsne x 0 * lsne y 0

pjoin(x↦y * lsne y 0, lsne x 0 * y↦0) = undefined.

pjoin(y=0 ∧ lsne x y, x↦y * lsne y 0) = lsne x y * lspe y 0

- 


- Overapproximation :

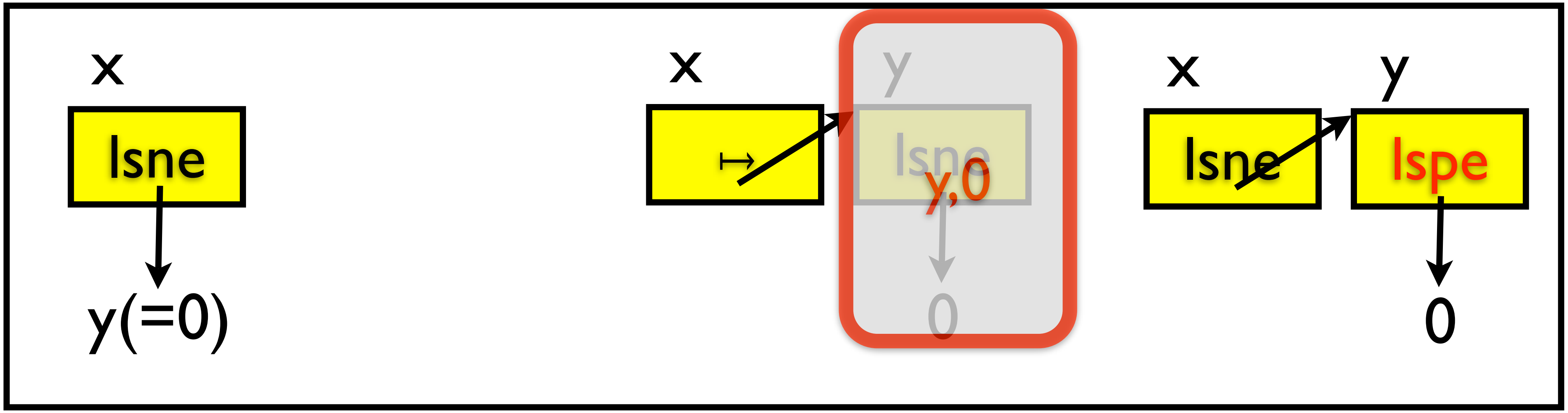
  If pjoin(q, q')=q'', then $(q \vee q') \Rightarrow q''$

- E.g.

  pjoin(x↦0 * lsne y 0, lsne x 0 * y↦0) = lsne x 0 * lsne y 0

  pjoin(x↦y * lsne y 0, lsne x 0 * y↦0) = undefined.

  pjoin(y=0 ∧ lsne x y, x↦y * lsne y 0) = lsne x y * lspe y 0

- 


- Overapproximation :

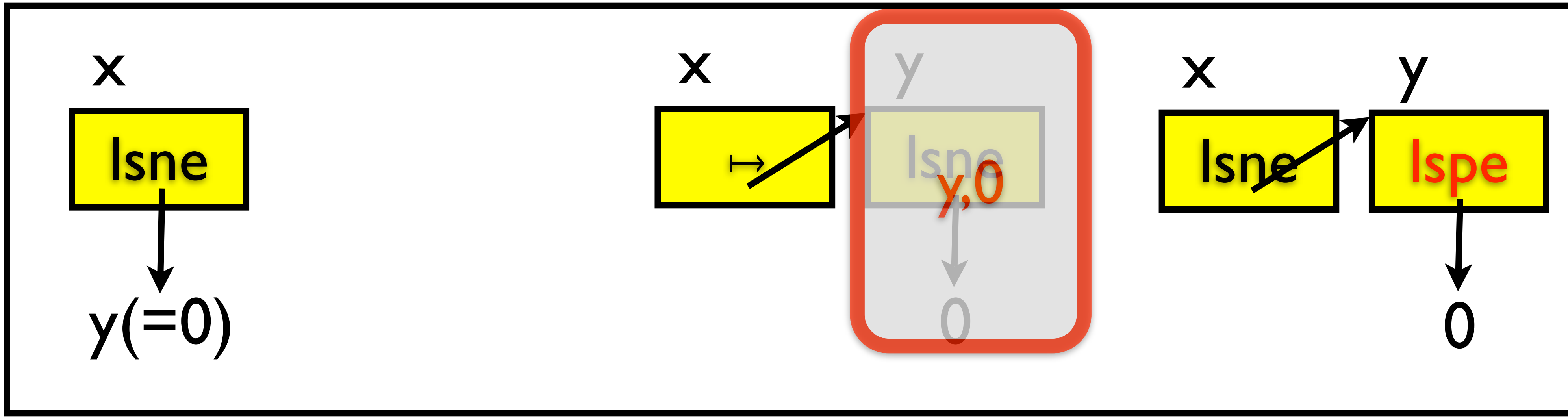
  If pjoin(q, q')=q'', then $(q \vee q') \Rightarrow q''$

- E.g.

  pjoin(x↦0 * lsne y 0, lsne x 0 * y↦0) = lsne x 0 * lsne y 0

  pjoin(x↦y * lsne y 0, lsne x 0 * y↦0) = undefined.

  pjoin(y=0 ∧ lsne x y, x↦y * lsne y 0) = lsne x y * lspe y 0



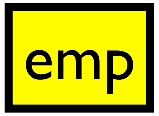

1. Most tricky part.
2. Discovers, on the fly, which nodes to forget.

pjoin(x↦0 * lsne y 0,  lsne x 0 * y↦0) =  lsne x 0 * lsne y 0

pjoin(x↦y * lsne y 0,  lsne x 0 * y↦0) = undefined.

pjoin(y=0 ∧ lsne x y,  x↦y * lsne y 0) = lsne x y * lspe y 0


```
L create() {...}

L append(L a,L b) {...}
```




```
void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

| emp | ret=0 ∧ emp, ret↦0, lsne ret 0 |
| --- | --- |

L create() {...}

L append(L a,L b) {...}

```
void main() {    emp
 x=create();
 y=create();
 x=append(x,y);

}
```

| emp | lspe ret 0 |
| --- | --- |

```
L create() {...}

L append(L a,L b) {...}

void main() {   [emp]
 x=create();
 y=create();
 x=append(x,y);

}
```

emp | lspe ret 0

```
L create() {...}

L append(L a,L b) {...}

void main() {          emp
 x=create();           lspe x 0
 y=create();
 x=append(x,y);

}
```

emp

lspe ret 0

```
L create() {...}

L append(L a,L b) {...}

void main() {
 x=create();
 y=create();
 x=append(x,y);

}
```

emp

lspe x 0

emp | lspe ret 0

```
L create() {...}

L append(L a,L b) {...}

void main() {        emp
 x=create();         lspe x 0
 y=create();         lspe x 0 * lspe y 0
 x=append(x,y);

}
```

emp | lspe ret 0

```
L create() {...}

L append(L a,L b) {...}

void main() {          emp
 x=create();           lspe x 0
 y=create();           lspe x 0 * lspe y 0
 x=append(x,y);        lspe x y * lspe y 0

}
```

emp

lspe ret 0

~~4 entries, 12 results~~

L create() {...}

L append(L a,L b) {...}

~~9 entries, 12 re~~

All tables have
one entry, one result

emp

```
void main() {
 x=create();      lspe x 0
 y=create();      lspe x 0 * lspe y 0
 x=append(x,y);   lspe x y * lspe y 0
}
```

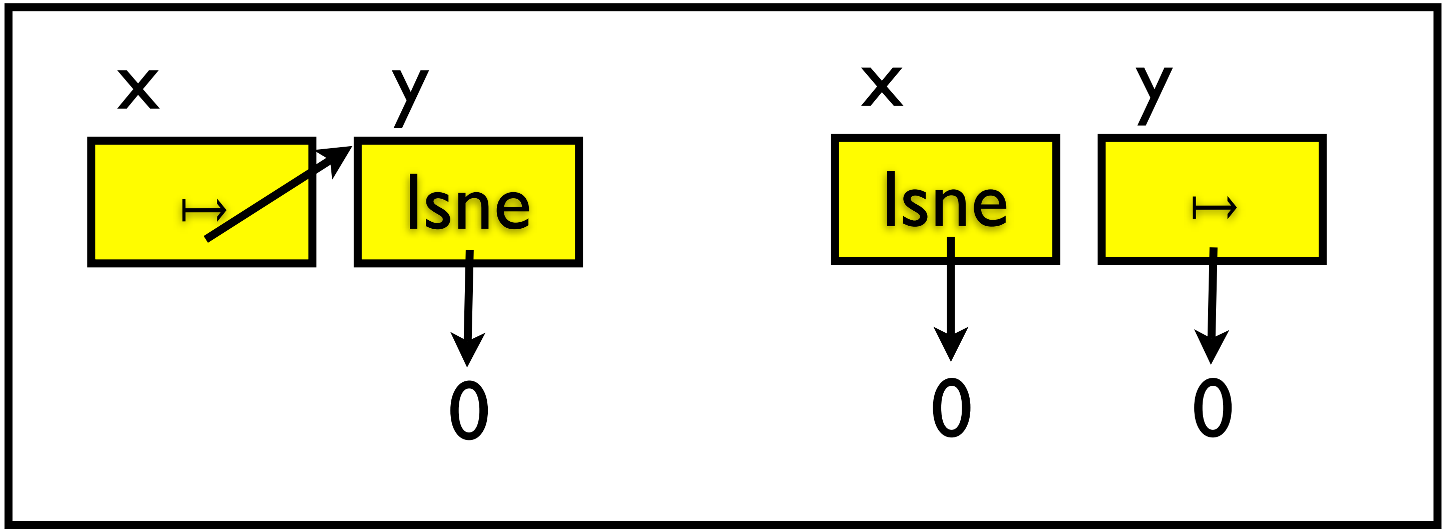
# Why Partial Join?

Prevent the analysis from misusing existential variables and losing crucial shape information.

$\{ \; x \mapsto y * \mathrm{lsne}\; y\; 0 \;\vee\; \mathrm{lsne}\; x\; 0 * y \mapsto 0 \; \}$

```
free_list(x)
```

$\{ \; \mathrm{emp} \;\vee\; y \mapsto 0 \; \}$

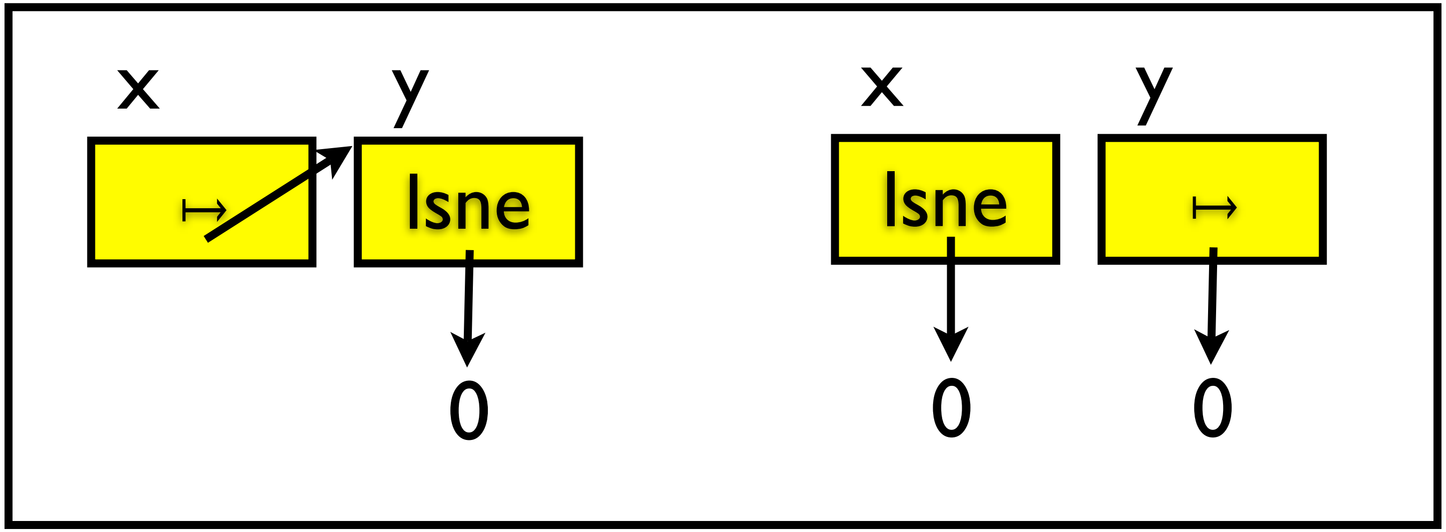



Prevent the analysis from misusing existential variables and losing crucial shape information.

$$\{\ x \mapsto y * \text{lsne}\ y\ 0\ \ \vee\ \ \text{lsne}\ x\ 0 * y \mapsto 0\ \}$$

```
free_list(x)
```

$$\{\ \text{emp}\ \ \vee\ \ y \mapsto 0\ \}$$

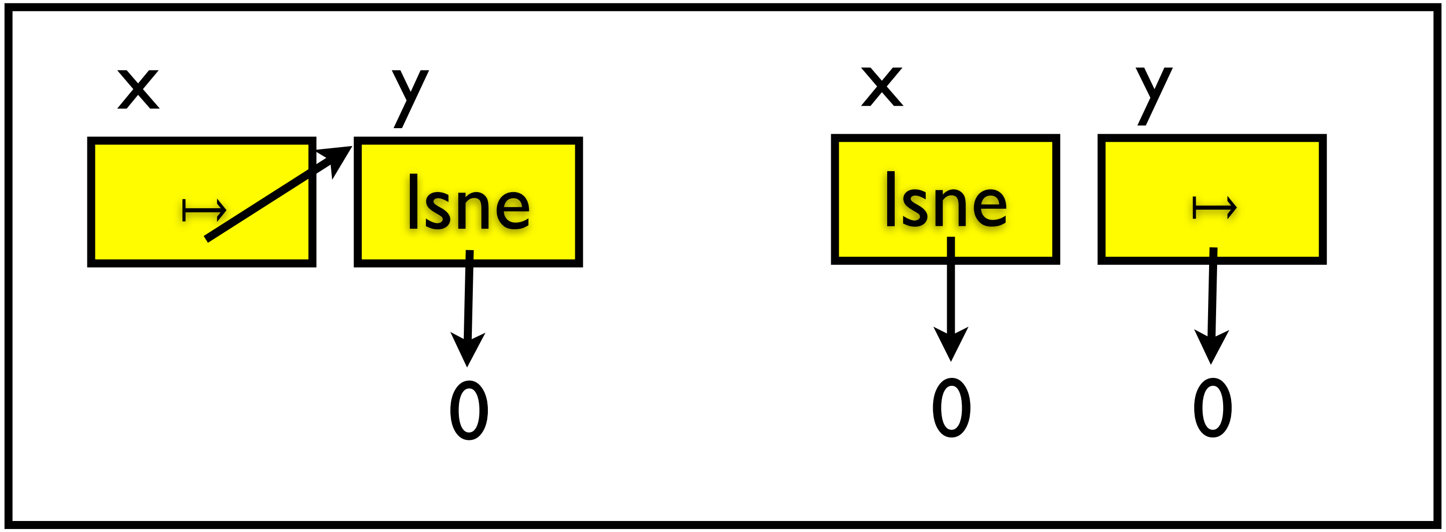



Prevent the analysis from misusing existential variables and losing crucial shape information.

$$\{\ x \mapsto y * \mathrm{lsne}\ y\ 0 \ \vee\ \mathrm{lsne}\ x\ 0 * y \mapsto 0\ \}$$

```
free_list(x)
```

$$\{\ \mathrm{emp} \ \vee\ y \mapsto 0\ \}$$

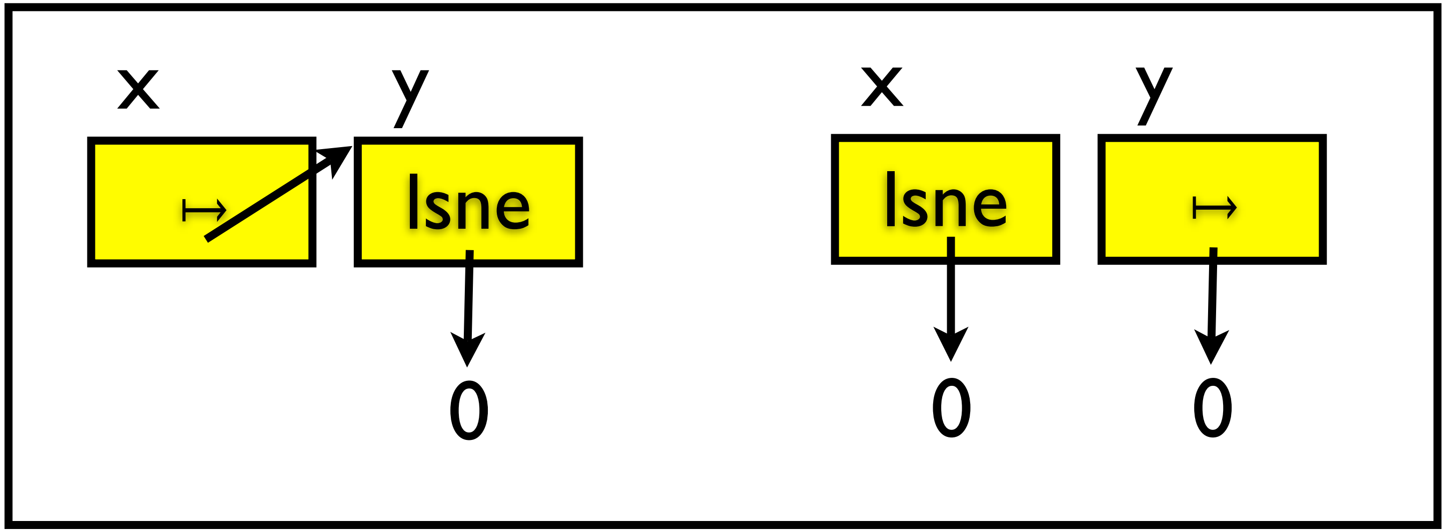



Prevent the analysis from misusing existential variables and losing crucial shape information.

{ lsne x x' * lsne y 0 }

```
free_list(x)
```

{ }




Prevent the analysis from misusing existential variables and losing crucial shape information.

{ lsne x x' * lsne y 0 }

```
free_list(x)
```

{ ???? }

# Why Partial Join?

Keep important co-relation between value and shape.

$\{ x \mapsto 0 \}$

```
y=t1394Diag_IoControl(x);
```

$\{ y=0 * emp \lor y!=0 * x \mapsto 0 \}$

```
if (y!=0) free(x);
```

$\{ emp \}$

# Why Partial Join?

Keep important co-relation between value and shape.

{ $x \mapsto 0$ }

```
y=t1394Diag_IoControl(x);
```

{ $y=0 * emp \lor y!=0 * x \mapsto 0$ }

```
if (y!=0) free(x);
```

{ emp }

# Why Partial Join?

Keep important co-relation between value and shape.

{ $x \mapsto 0$ }

```
y=t1394Diag_IoControl(x);
```

{ lspe x 0 }

```
if (y!=0) free(x);
```

{ }

# Why Partial Join?

Keep important co-relation between value and shape.

{ $x \mapsto 0$ }

```
y=t1394Diag_IoControl(x);
```

{ lspe x 0 }

```
if (y!=0) free(x);
```

{ ???? }

# Ingredients in Solution

- Partial join for separation domain (CAV08).
- Composite adaptive analysis (CAV07).
- Localization (CSL01, Rinetzky+POPL05, Gotsman+SAS06).
- Partial concretization (Sagiv+TOPLAS98).
- RHS (Reps+POPL95).
- Separation logic.

# Open Question, Summer 2006

Can we automatically prove the pointer safety of programs ≥ 10K in separation logic?