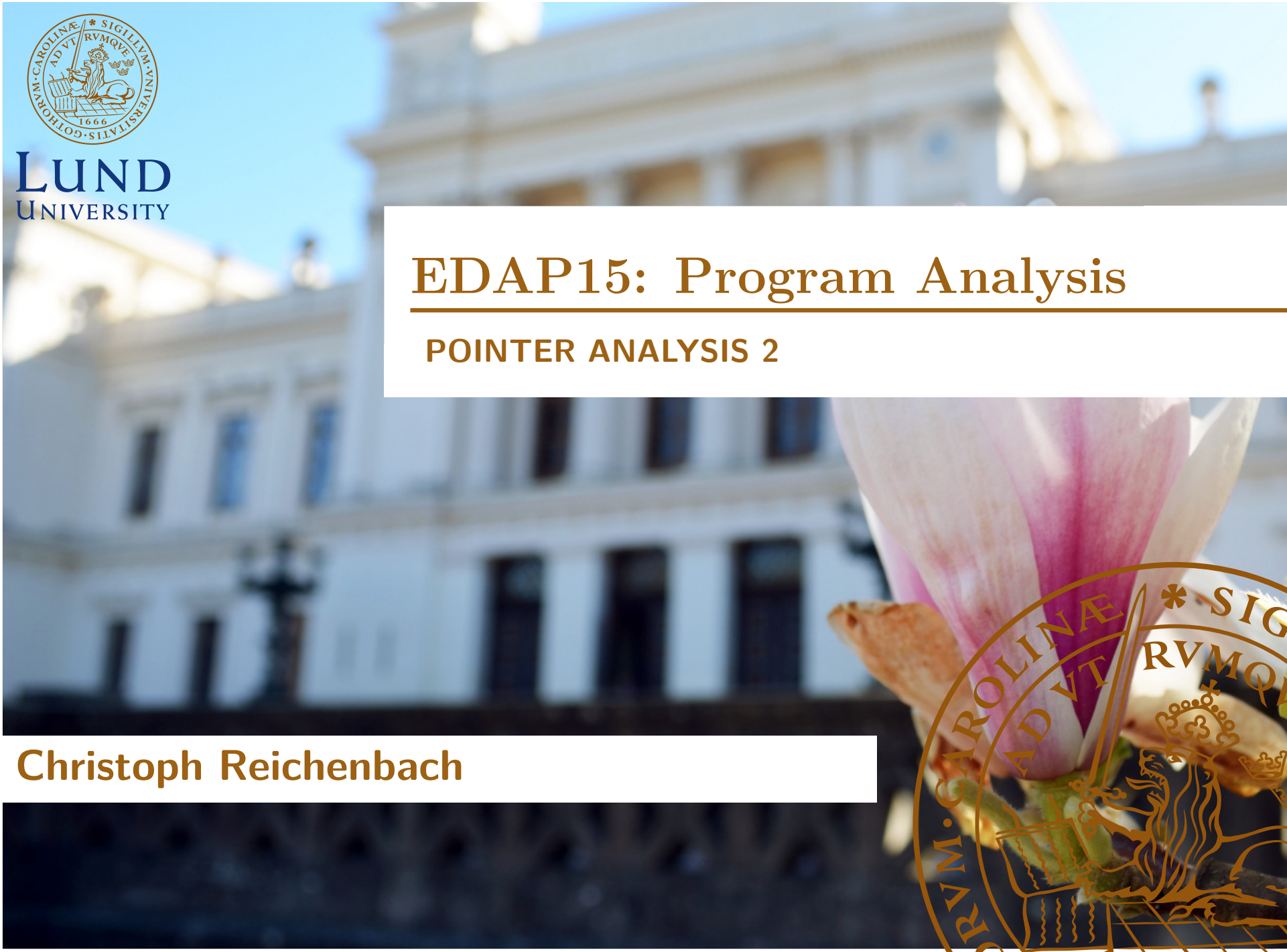


EDAP15: Program Analysis

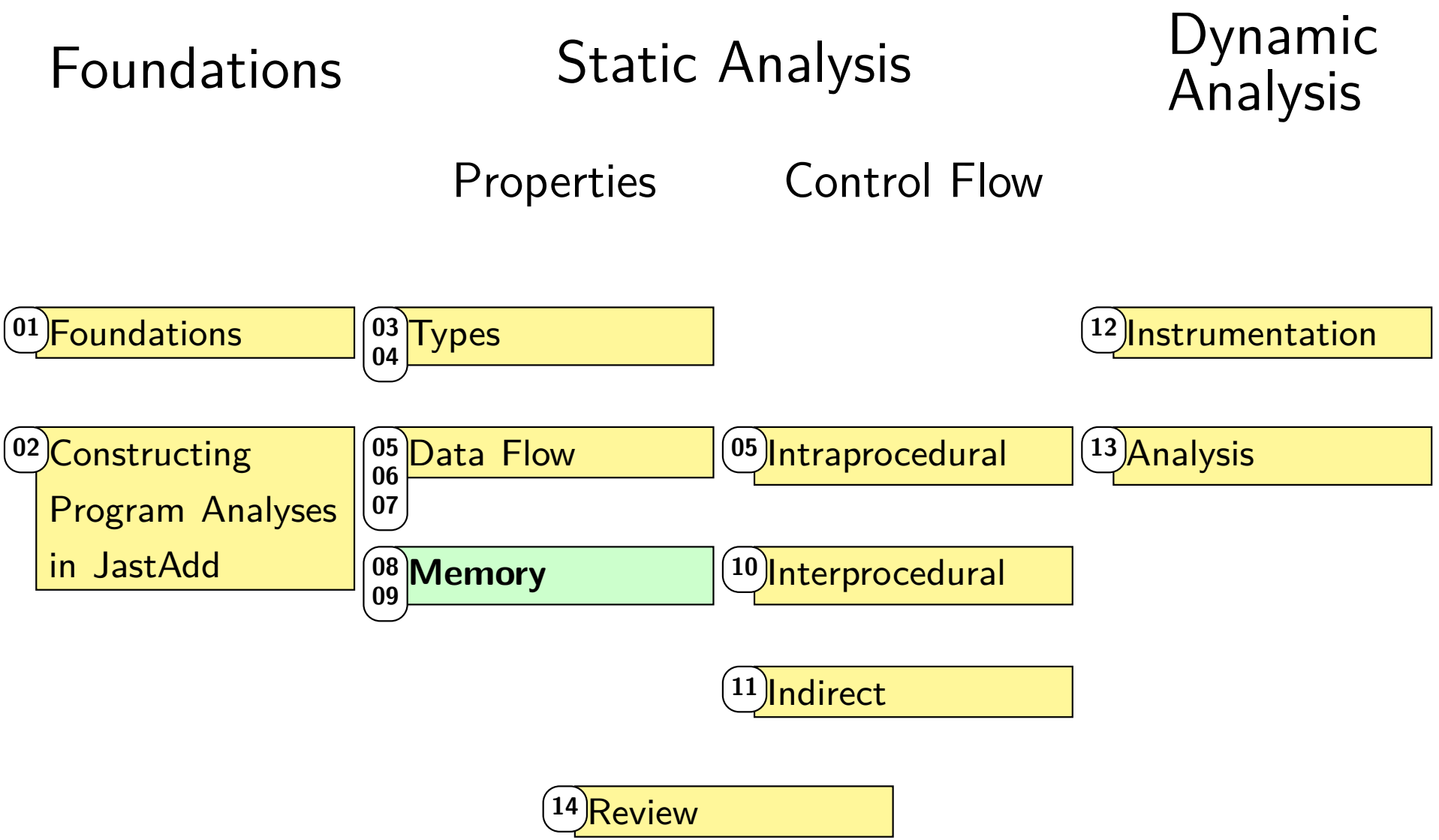
POINTER ANALYSIS 2

Christoph Reichenbach

Welcome back!

Questions?

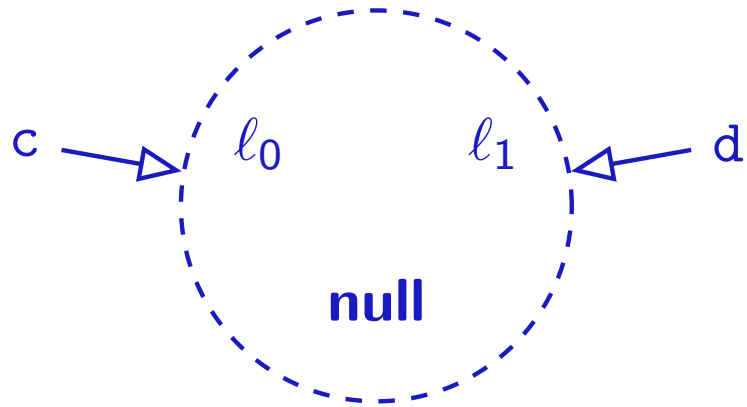
Lecture Overview



Alias Analysis in Practice (1/2)

Teal

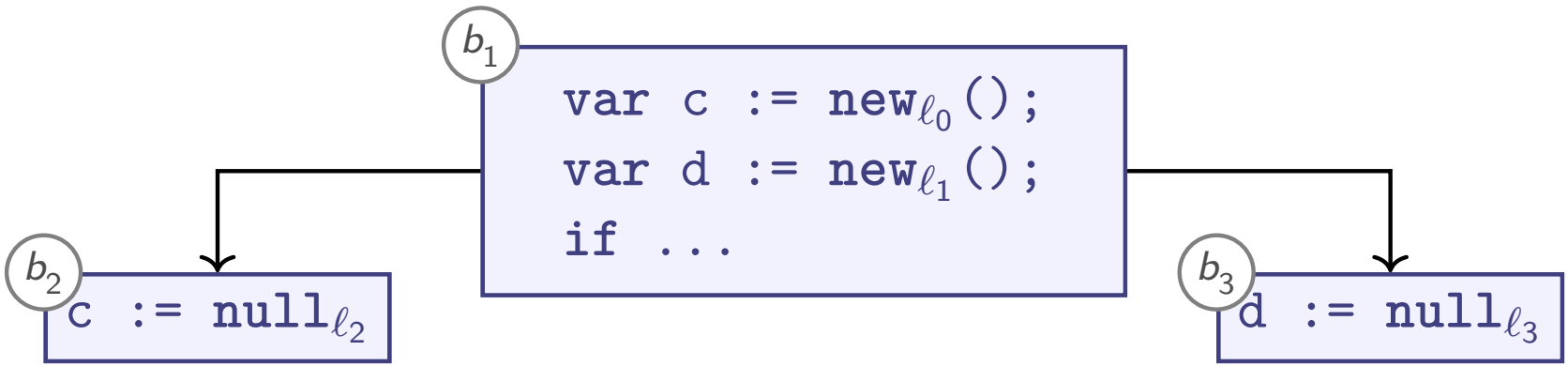
```
var c := new_{ℓ0}();
var d := new_{ℓ1}();
if ...  {
   c := null;
} else {
   d := null;
}
```



$c \overset{alias}{=\!=} d$

null as unique memory location: Imprecision!

Representing Null Pointers



1. One unique **null**

2. Many **null**s (More precise, takes up extra memory)

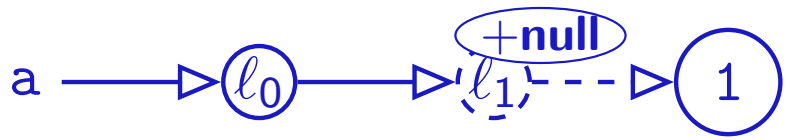
3. Nullness flags (Also more precise, minimal extra memory, but more complex analysis code)

Alias Analysis in Practice (2/2)

Teal

```
var a := new_ℓ0 XY();
a.x := new_ℓ1 XY();
a.x.x := 1;
a.y := null;

print(a.x.x);
// null dereference?
```



In this example: ignore field names (as we did with Steensgaard)

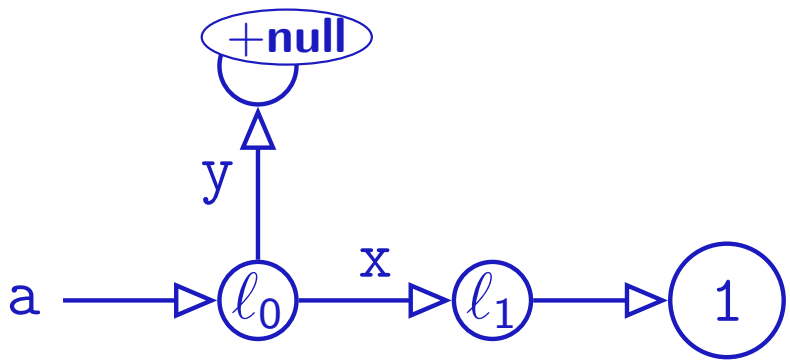
$$\texttt{a.x} \overset{\textit{alias}}{=\!=} \textbf{null} \overset{\textit{alias}}{=\!=} \texttt{a.}\square \overset{\textit{alias}}{=\!=} \texttt{a.y}$$

Field Sensitivity

- For Steensgaard, we merged all fields:

$$\texttt{a.x} \overset{\textit{alias}}{=\!=} \texttt{a.}\square \overset{\textit{alias}}{=\!=} \texttt{a.y}$$

- Analysis that merge field names: *field insensitive*
- A *field-sensitive* analysis would distinguish:



Summary

- Points-to analyses often want to represent **null**
  - Single global **null** may reduce precision (in unification-based analysis)
- Some program analyses are **field insensitive**:

$$\texttt{a.x} \overset{\textit{alias}}{=\!=} \texttt{a.}\square \overset{\textit{alias}}{=\!=} \texttt{a.y} \quad \text{(always)}$$

- **Field-sensitive** analyses improve precision by distinguishing fields along points-to edges:

$$\texttt{a.x} \overset{\cancel{\textit{alias}}}{=\!=} \texttt{a.y} \quad \text{(unless explicit evidence)}$$

- Analogous: **index-sensitive** analyses:

$$\texttt{a[0]} \overset{\textit{alias}}{=\!=} \texttt{a[1]} \overset{\textit{alias}}{=\!=} \ldots \overset{\textit{alias}}{=\!=} \texttt{a[i]}$$

Dataflow-Based Points-To Analysis

$$\begin{array}{lcl} G_{\mathsf{AHG}} & = & \langle \overline{L}, \rightarrow\!\!\triangleright \rangle \\ (\rightarrow\!\!\triangleright) & \subseteq & \overline{L} \times \overline{L} \end{array}$$

- Points-To via Dataflow:
- Lattice over set of edges between memory locations $\overline{L}$
- $\sqcup = \cup$
- $\sqsubseteq = \subseteq$

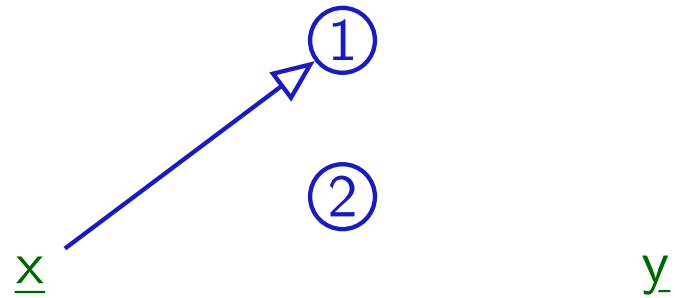
Example: Allocation and Update

Teal

```
    var x := new1();
⇒   var y := new2();

    if ... {
      y := new5();
    }

    x := new7();
    y := x;
```



Case	**Transfer Function**	
$\texttt{x := new}_\ell\texttt{()}$	$trans_1(\rightarrow) = (\rightarrow)$	$\cup\ \{\ x \rightarrow \ell\ \}$

Slight abuse of notation: writing $x \rightarrow \ell$ for $\langle x, \ell \rangle$

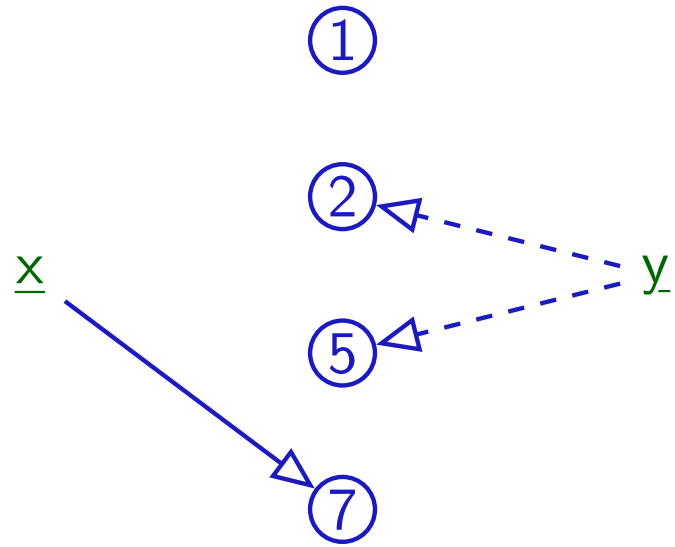
Example: Allocation and Update

Teal

```
    var x := new1();
    var y := new2();

    if ... {
      y := new5();
    }

⇒ x := new7();
    y := x;
```



Remove all $\underline{x}\rightarrow\ell$ for any $\ell \in \overline{L}$

Case	**Transfer Function**
$\underline{x}$:= new$_\ell$()	$trans_1(\rightarrow) = (\rightarrow) \setminus (\{\underline{x}\} \times \overline{L}) \cup \{\, \underline{x}\rightarrow\ell \,\}$

Slight abuse of notation: writing $x\rightarrow\ell$ for $\langle x, \ell \rangle$

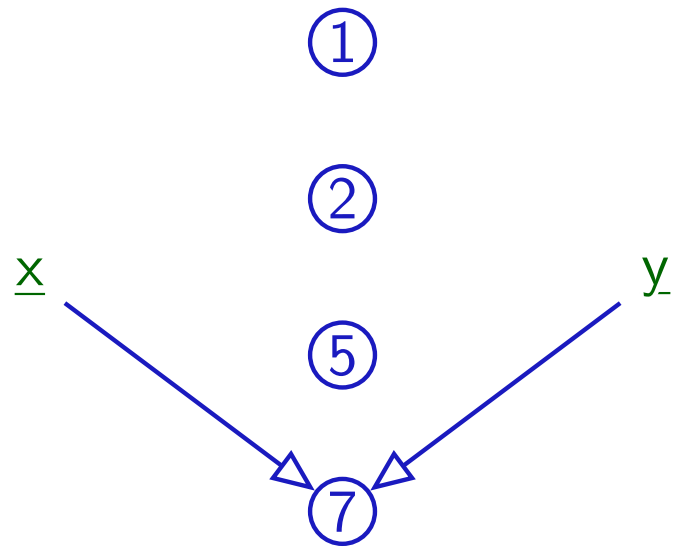
Example: Allocation and Update

Teal

```
    var x := new1();
    var y := new2();

    if ... {
      y := new5();
    }

    x := new7();
⇒   y := x;
```

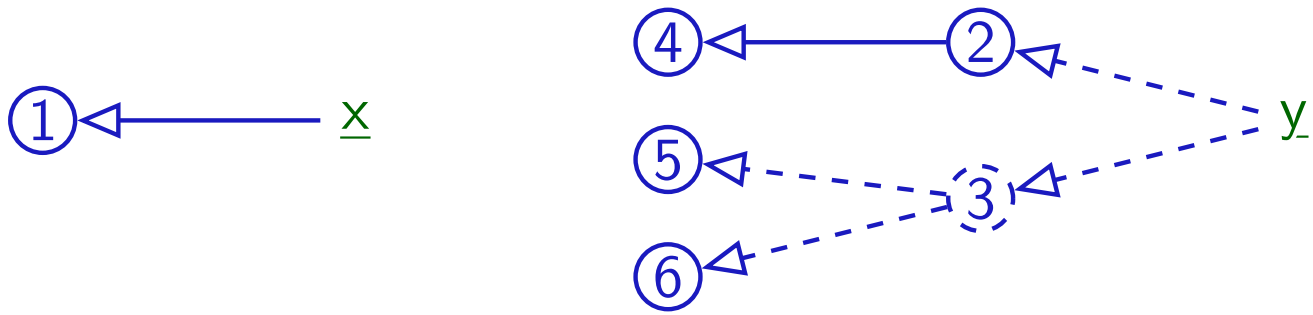


Case	**Transfer Function**
$\mathtt{x} := \mathtt{new}_\ell()$	$trans_1(\rightarrow) = (\rightarrow) \setminus (\{\mathtt{x}\} \times \overline{L}) \cup \{\ \mathtt{x} \rightarrow \ell\ \}$
$\mathtt{y} := \mathtt{x};$	$trans_2(\rightarrow) = (\rightarrow) \setminus (\{\mathtt{y}\} \times \overline{L}) \cup \{\ \mathtt{y} \rightarrow \ell \mid \mathtt{x} \rightarrow \ell\}$

Slight abuse of notation: writing $x \rightarrow \ell$ for $\langle x, \ell \rangle$

Dereferencing Read

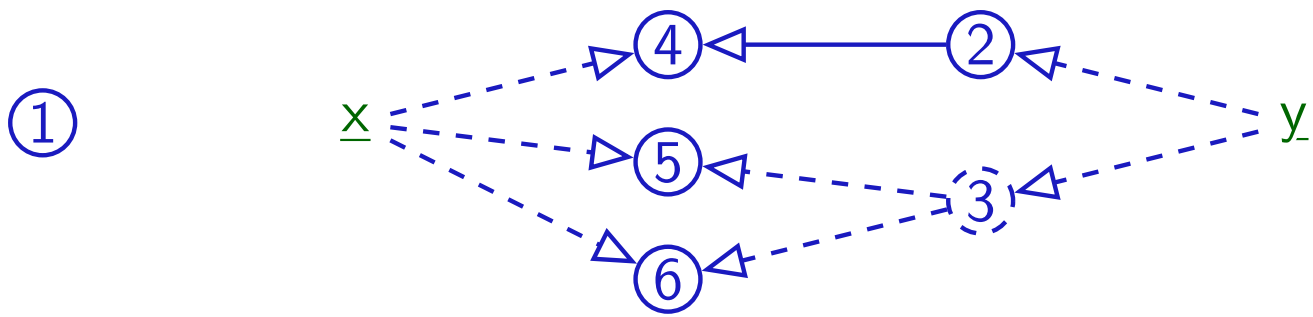
Before:



Teal

```
x := y.f;
```

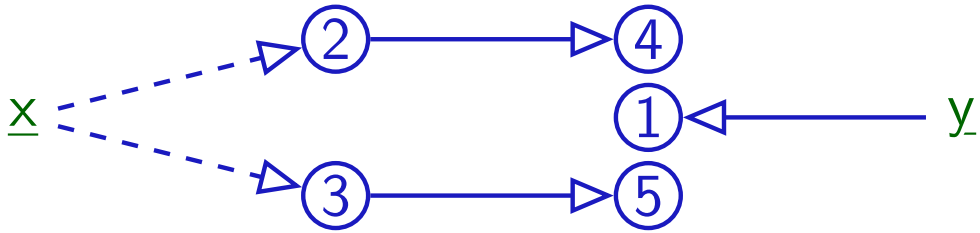
After:



Case	**Transfer Function**
$\underline{x} := \underline{y}.\square;$	$trans_3(\rightarrow) = (\rightarrow) \setminus (\{\underline{x}\} \times \overline{L}) \cup \{ \underline{x} \rightarrow \ell' \mid \exists \ell. \underline{y} \rightarrow \ell \rightarrow \ell' \}$

Dereferencing Write

Before:

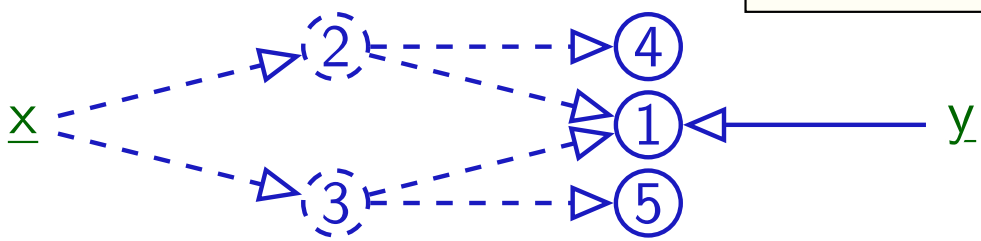


Teal

```
x.f := y;
```

Cannot remove edges: We don't know if it was 2 or 3 that was changed!

After:



Case	**Transfer Function**
$\underline{x}.\square := \underline{y};$	$trans_4(\rightarrow) = (\rightarrow) \setminus (\{\underline{x}\} \times \overline{L}) \cup \{\, \ell \rightarrow \ell' \mid \underline{x} \rightarrow \ell, \underline{y} \rightarrow \ell' \}$

Weak vs Strong Update?

- **Strong update**:
  - Can remove "obsolete" information
- **Weak update**:
  - Cannot remove "obsolete" information
  - Observed with dereferencing write
- Dereferencing write *can* be strong if x can point to only one location

Teal

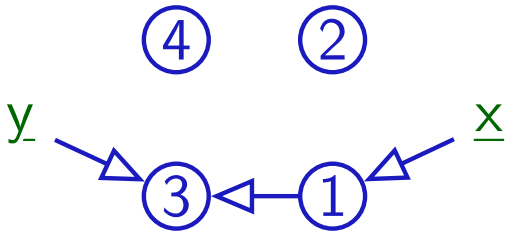
```
x.f := y;
```

Dataflow-Based Points-To Analysis

$$
\begin{array}{lcl}
G_{\mathsf{AHG}} & = & \langle \overline{L}, \rightarrow \rangle \\
(\rightarrow) & \subseteq & \overline{L} \times \overline{L}
\end{array}
$$

$\underline{\texttt{x}}$ `:= new`$_\ell$`()`	$trans_1(\rightarrow) = (\rightarrow) \setminus (\{\underline{\texttt{x}}\} \times \overline{L}) \cup \{\ \underline{\texttt{x}} \rightarrow \ell\ \}$
$\underline{\texttt{x}}$ `:=` $\underline{\texttt{y}}$`;`	$trans_2(\rightarrow) = (\rightarrow) \setminus (\{\underline{\texttt{x}}\} \times \overline{L}) \cup \{\ \underline{\texttt{x}} \rightarrow \ell \mid \underline{\texttt{y}} \rightarrow \ell\}$
$\underline{\texttt{x}}$ `:=` $\underline{\texttt{y}}$`.□;`	$trans_3(\rightarrow) = (\rightarrow) \setminus (\{\underline{\texttt{x}}\} \times \overline{L}) \cup \{\ \underline{\texttt{x}} \rightarrow \ell' \mid \exists \ell.\ \underline{\texttt{y}} \rightarrow \ell \rightarrow \ell'$
$\underline{\texttt{x}}$`.□ :=` $\underline{\texttt{y}}$`;`	$trans_{4_{weak}}(\rightarrow) = (\rightarrow) \cup \{\ \ell \rightarrow \ell' \mid \underline{\texttt{x}} \rightarrow \ell, \underline{\texttt{y}} \rightarrow \ell'\}$
$\underline{\texttt{x}}$`.□ :=` $\underline{\texttt{y}}$`;`	$trans_{4_{strong}}(\rightarrow) = (\rightarrow) \setminus (\{\underline{\texttt{x}}\} \times \overline{L}) \cup \{\ \ell \rightarrow \ell' \mid \underline{\texttt{x}} \rightarrow \ell, \underline{\texttt{y}} \rightarrow \ell'\}$

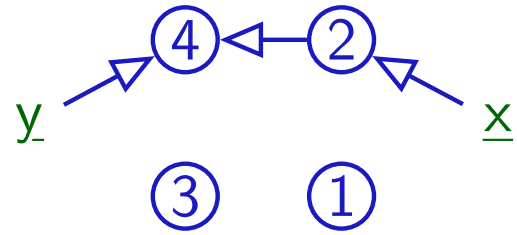
Distributivity?



Teal

```
x.f := y;
```

trans

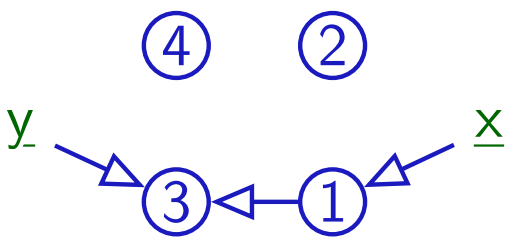


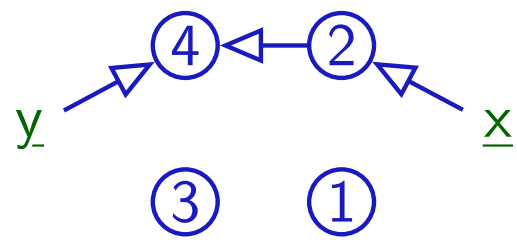
Teal

```
x.f := y;
```

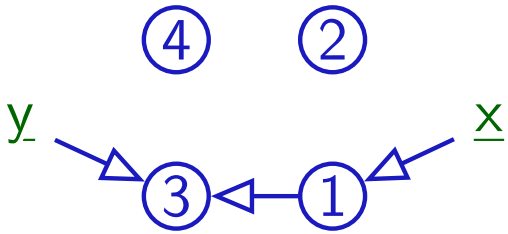
trans

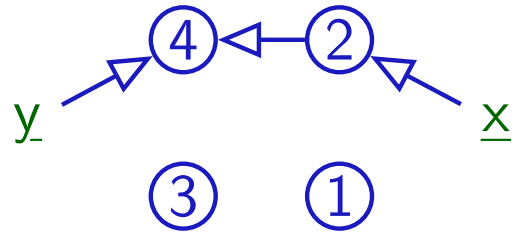
Transfer function *trans* does not change either graph in this case





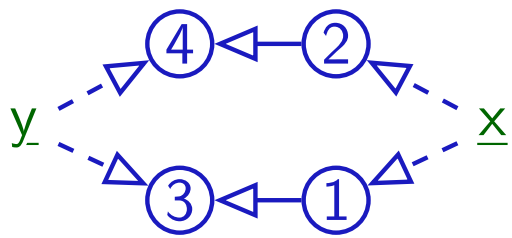
Distributivity?





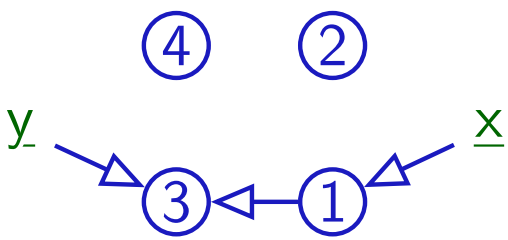
Teal

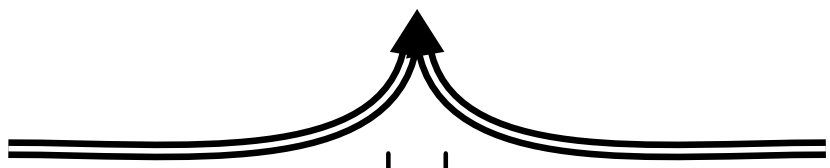
```
x.f := y;
```



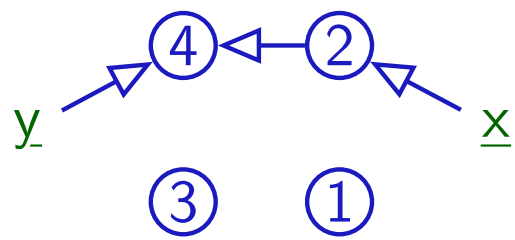
Teal

```
x.f := y;
```

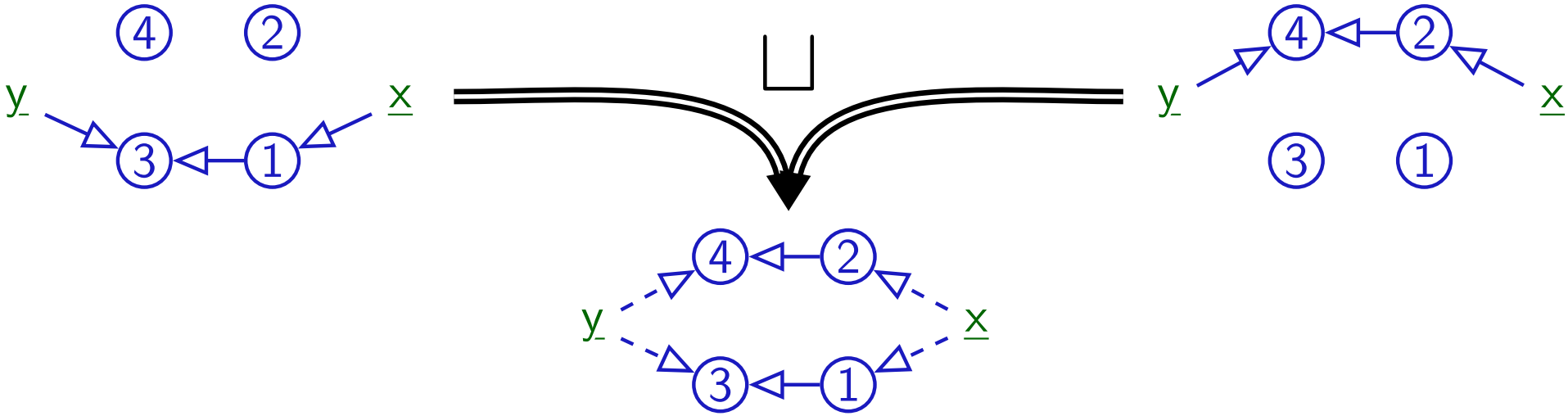




Result for join *after* transfer



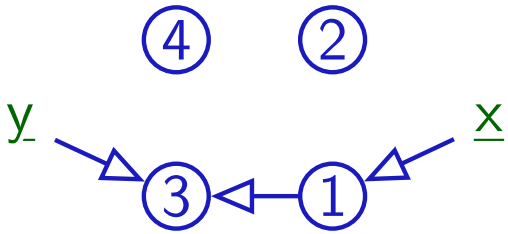
Distributivity?

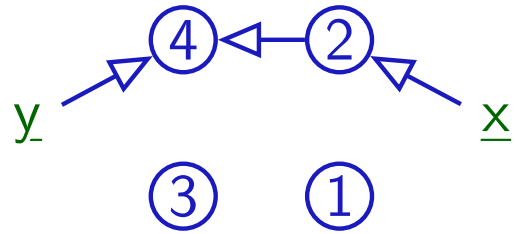


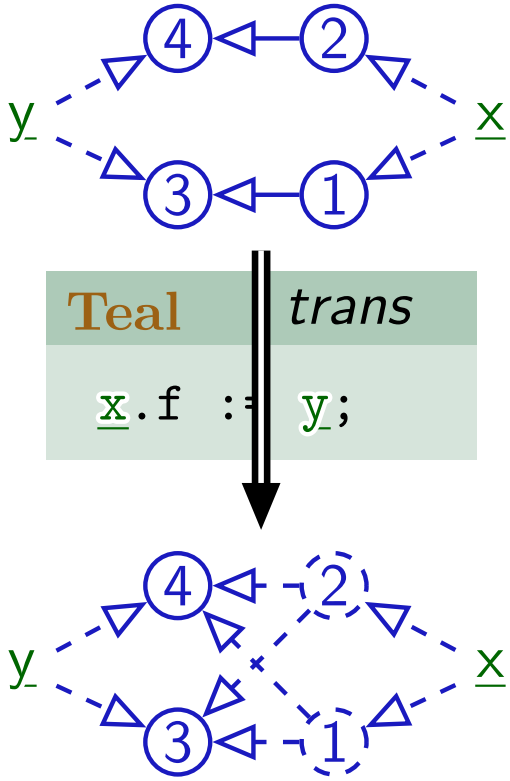
Teal

```
x.f := y;
```

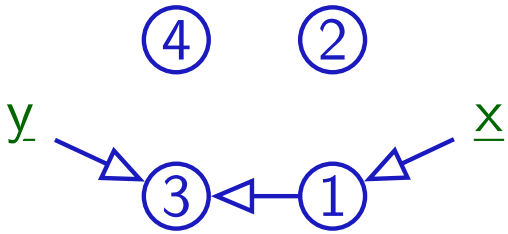
Distributivity?

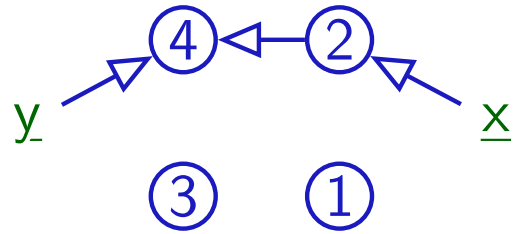


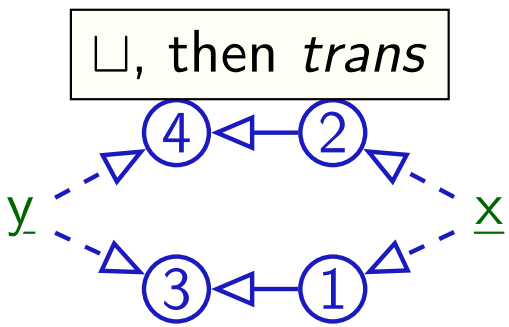


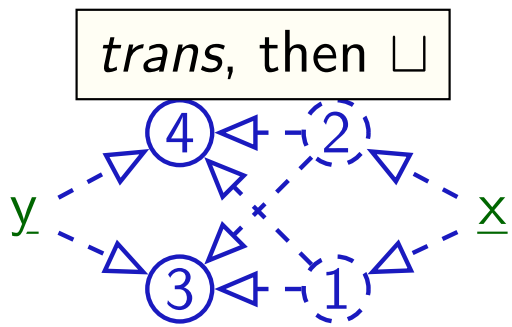


Distributivity?









Different results $\Longrightarrow$ not distributive!

Summary

- Flow-sensitive points-to analysis is possible but expensive
- **Weak updates** add new points-to relationship options
  - Don't remove existing options
- **Strong updates** add but also remove points-to relationship options
  - More precise than weak updates
  - Only possible if updated pointer is unambiguous
- *Not Distributive*

Andersen's Points-To Analysis

- Asymptotic performance is $O(n^3)$
- More precise than Steensgaard's analysis
- *Subset-based* (a.k.a. *inclusion-based*)
- $\Longrightarrow$ Flow-insensitive but *directed*
- Popular as basis for current points-to analyses

L. Andersen, "Program Analysis and Specialization for the C Programming Language", PhD. thesis, DIKU report 94/19, 1994

Collecting Constraints

- Collect constraints, resolve as needed
- For each statement in program, we record:
  - If **Referencing** ($\texttt{x := new}_{\ell_i}\ \texttt{A()}$):

$$\ell_i \in pt(\texttt{x}) \qquad (x \rightarrowtriangle \ell_i)$$

  - If **Aliasing** ($\texttt{x := y}$):

$$pt(\texttt{x}) \supseteq pt(\texttt{y})$$

  - If **Dereferencing read** ($\texttt{x := y.}\square$):

$$pt(\texttt{x}) \supseteq pt(\texttt{y.}\square)$$

  - If **Dereferencing write** ($\texttt{x.}\square\ \texttt{:= y}$):

$$pt(\texttt{x.}\square) \supseteq pt(\texttt{y})$$

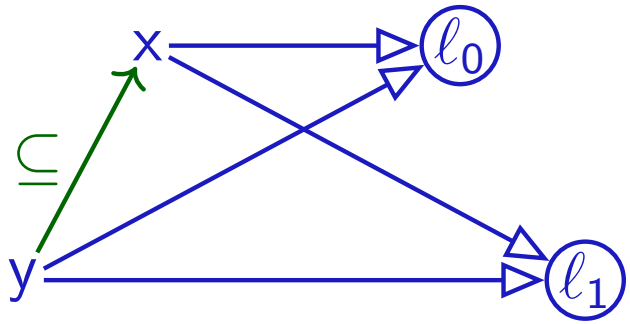
Andersen's Analysis

1. Fact extraction:
   - Initial points-to sets: $\ell \in pt(x)$, meaning $\ell \leftarrow x$
   - Constraints:
     - $pt(x) \supseteq pt(y)$
     - $pt(x) \supseteq pt(y.\square)$
     - $pt(x.\square) \supseteq pt(y)$

Subset Constraints (1/2)

- Solving $pt(x) \supseteq pt(y)$

```
y := new_ℓ0();
x := y;
y := new_ℓ1();
```

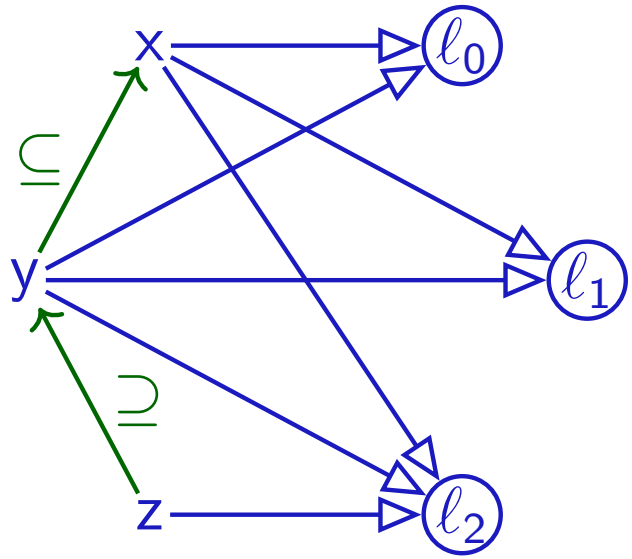


- $\ell \leftarrow y$ and $pt(x) \supseteq pt(y)$:
  $\Longrightarrow$ $\ell \leftarrow x$
- *Flow insensitive*: can't distinguish before/after

Subset Constraints (1/2)

- Solving $pt(x) \supseteq pt(y)$

```
y := new_ℓ0();
x := y;
y := new_ℓ1();
z := new_ℓ2();
if ... {
  y := z;
}
```



- $\ell \leftarrow y$ and $pt(x) \supseteq pt(y)$:
  $\Longrightarrow$ $\ell \leftarrow x$
- *Flow insensitive*: can't distinguish before/after

Solving one ($\supseteq$) can depend on all ($\leftarrow$) and ($\supseteq$) in program

Composite Statements

- Decompose more complex patterns:

$$\texttt{y.n := new}_{\ell_1}\texttt{();}$$

- No direct pattern for this code
- Can model as:

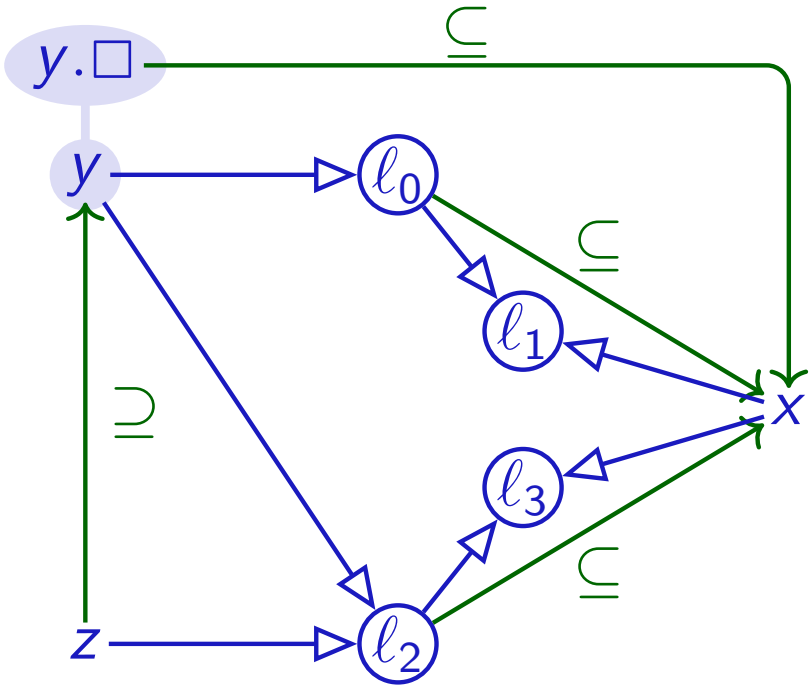
```
var tmp := new_ℓ1();
y.n := tmp;
```

Subset Constraints (2/2)

- Solving $pt(x) \supseteq pt(y.\square)$

```
y := new_ℓ0();
y.n := new_ℓ1();
z := new_ℓ2();
z.n := new_ℓ3();
y := z;
x := y.n;
```

Simplified presentation (omitting ($\supseteq$) constraints)



- Recall:
  $\ell \leftarrow z$ and $pt(y) \supseteq pt(z)$:
  $\Longrightarrow \ell \leftarrow y$
- $\ell \leftarrow y$ and $pt(x) \supseteq pt(y.\square)$:
  $\Longrightarrow pt(x) \supseteq pt(\ell)$

Andersen's Analysis

1. Fact extraction:
   - Initial points-to sets: $\ell \in pt(x)$, meaning $\ell \leftarrow x$
   - Constraints:
     - $pt(x) \supseteq pt(y)$
     - $pt(x) \supseteq pt(y.\square)$
     - $pt(x.\square) \supseteq pt(y)$
2. Build directed *inclusion graph* $G_I = \langle MemLoc, E \rangle$
   - $x \leftarrow y$ represents $pt(x) \supseteq pt(y)$ ("$x := y$")
3. Expand and propagate along inclusion graph:
   - Propagate points-to sets along E:
     - $\ell \leftarrow y$ and $x \leftarrow y$:
       $\Longrightarrow \ell \leftarrow x$
     - $\ell \leftarrow y$ and $x \leftarrow y.\square$:
       $\Longrightarrow x \leftarrow \ell$
     - $\ell \leftarrow x$ and $x.\square \leftarrow y$:
       $\Longrightarrow \ell \leftarrow y$

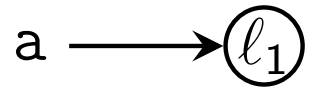
Example

```
⇒ x := new_ℓz     x⇾ℓz
  x := y          x⇽y
  x := y.□        x⇽y.□
  x.□ := y        x.□⇽y
```

Teal

```
var a := new_ℓ1(); //⇐
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



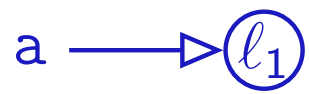
p

q

b

r

- **Andersen**:



p

q

b

r

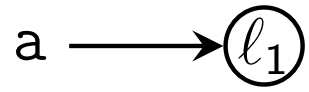
Example

```
⇒ x := new_ℓz     x⟼ℓz
  x := y          x⟵y
  x := y.□        x⟵y.□
  x.□ := y        x.□⟵y
```

Teal

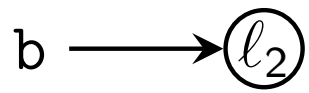
```
var a := new_ℓ1();
var b := new_ℓ2(); //⇐
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



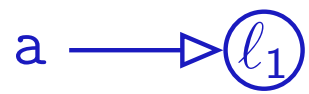
p

q



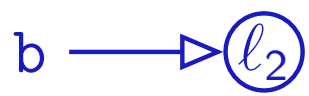
r

- **Andersen**:



p

q



r

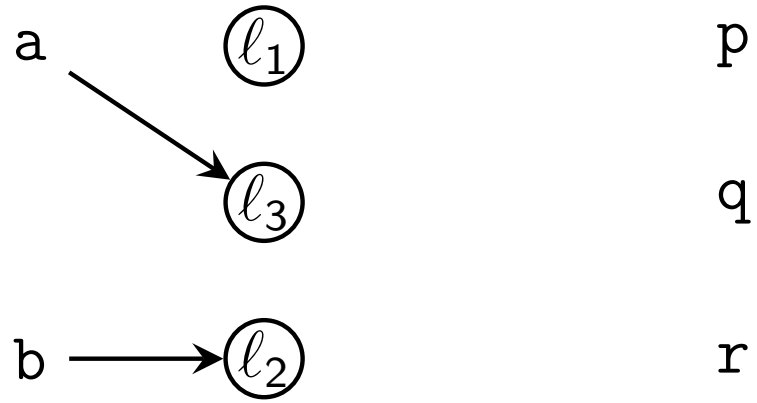
Example

```
⇒ x := new_ℓz     x⟼ℓz
  x := y          x⟵y
  x := y.□        x⟵y.□
  x.□ := y        x.□⟵y
```

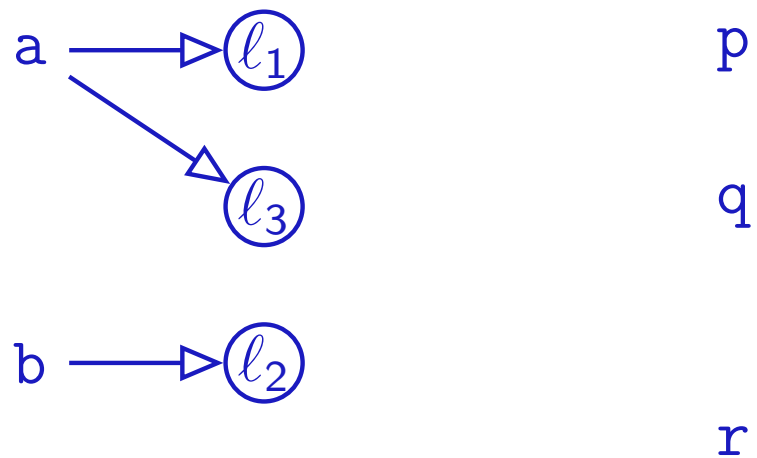
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();        //⇐
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



- **Andersen**:



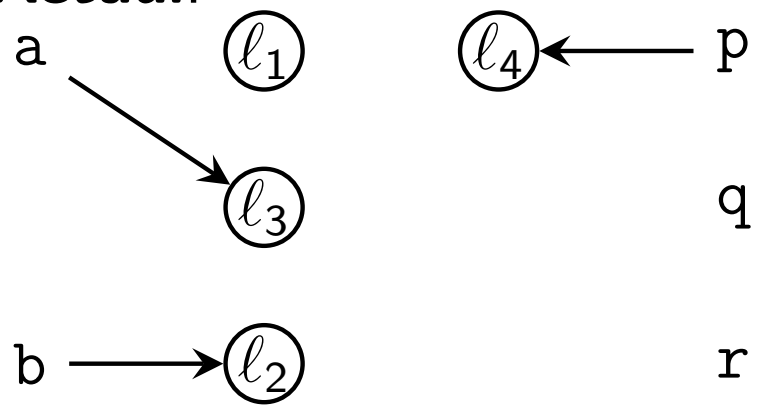
Example

```
⇒ x := new_ℓz     x⟿ℓz
  x := y          x⟵y
  x := y.□        x⟵y.□
  x.□ := y        x.□⟵y
```

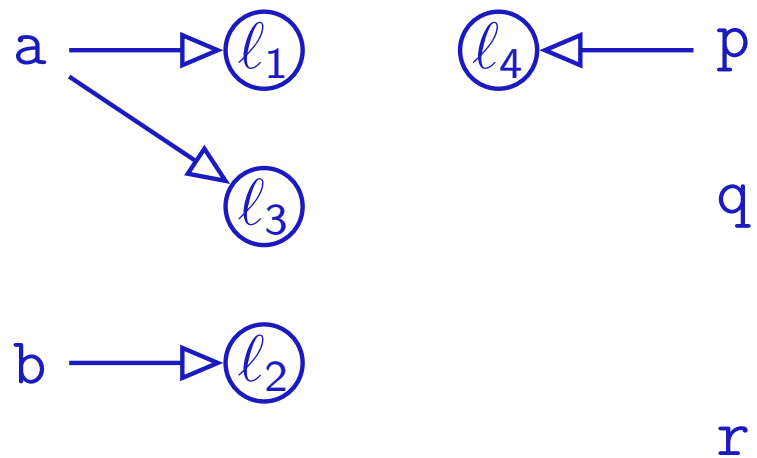
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4(); //⇐
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



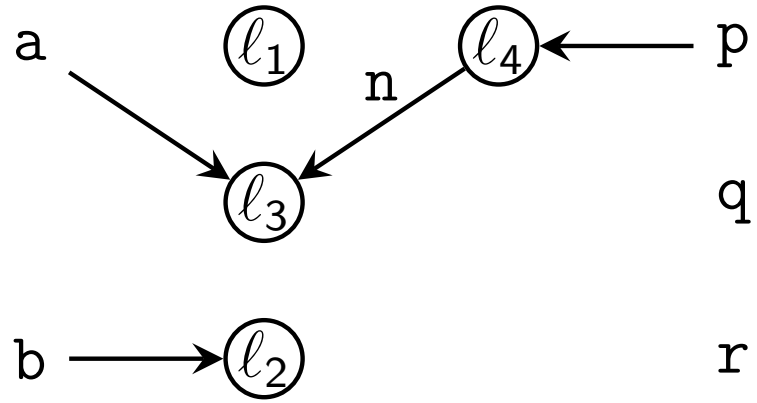
- **Andersen**:



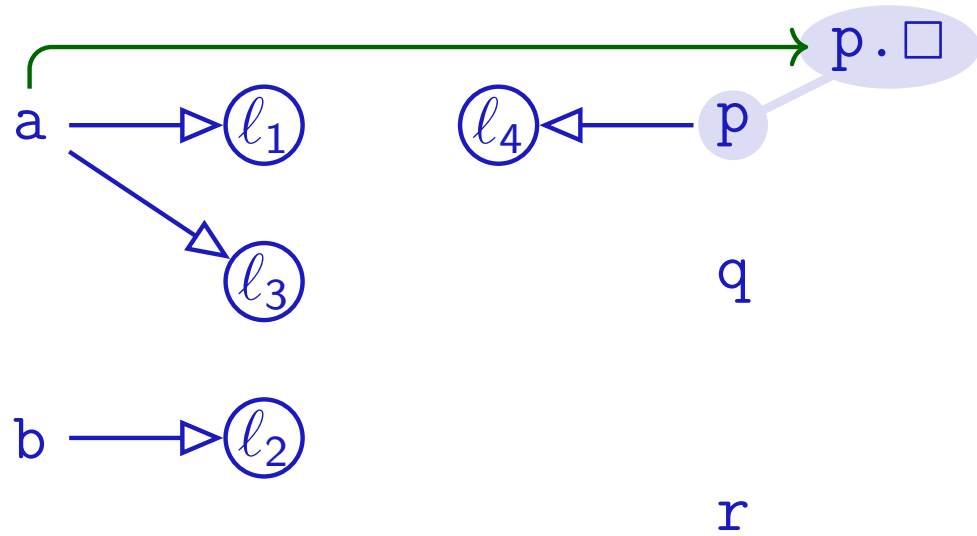
Example

`x := new`ℓ_z	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
$\Rightarrow$ `x.`$\square$ `:= y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



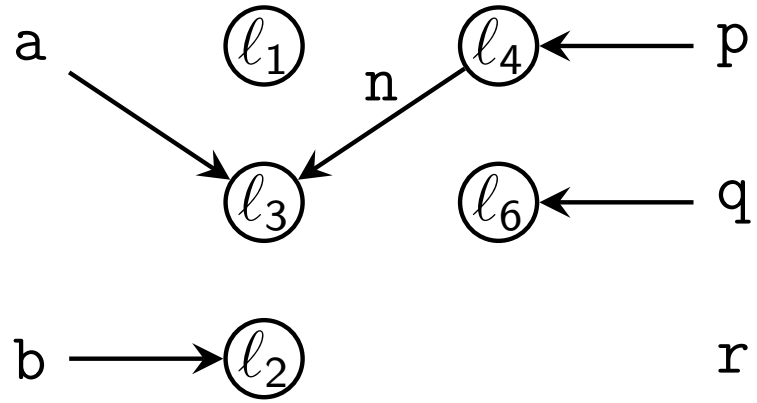
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;              //⇐
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

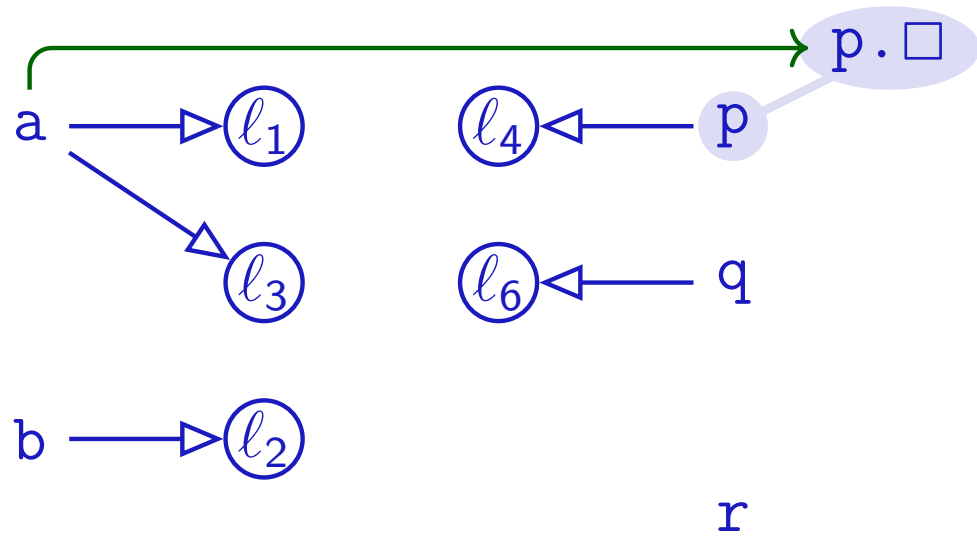
Example

⇒	`x := new`ℓ_z	$x \rightarrowtriangle \ell_z$
	`x := y`	$x \leftarrow y$
	`x := y.`$\square$	$x \leftarrow y.\square$
	`x.`$\square$ `:= y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



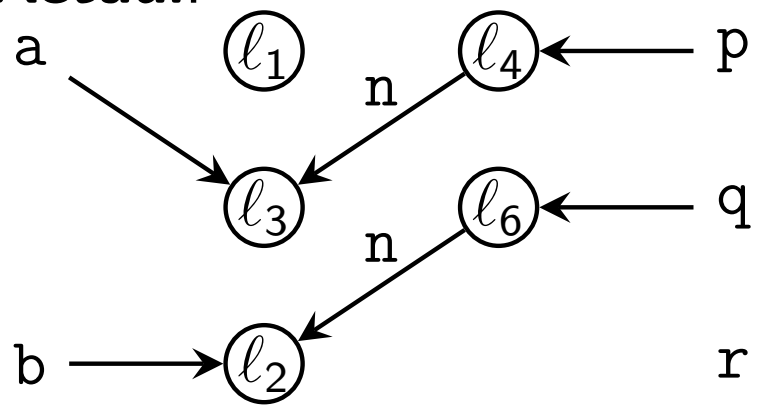
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6(); //⇐
q.n := b;
p := q;
var r := q.n;
```

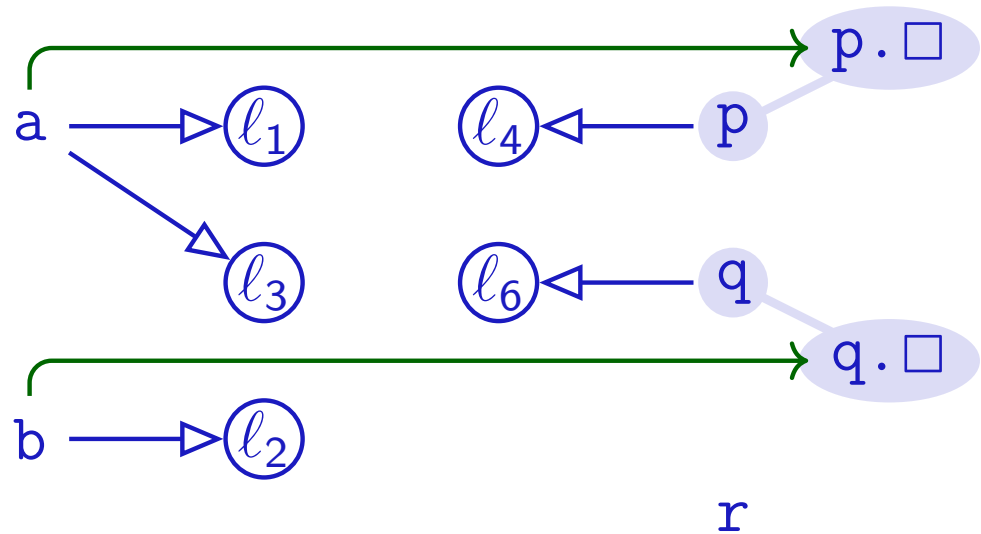
Example

`x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
$\Rightarrow$ `x.`$\square$ `:= y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



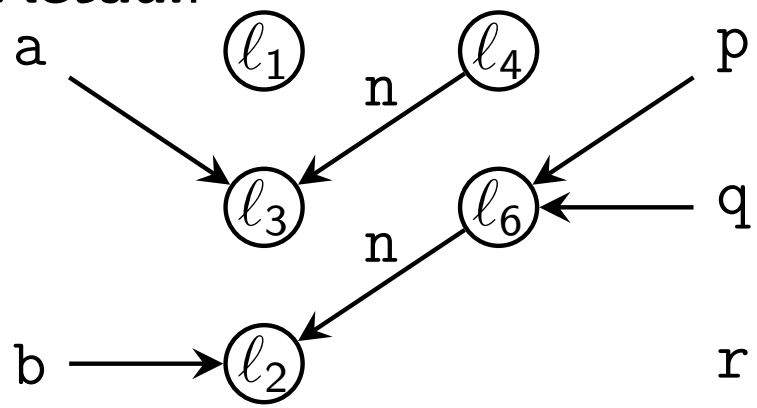
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;          //⇐
p := q;
var r := q.n;
```

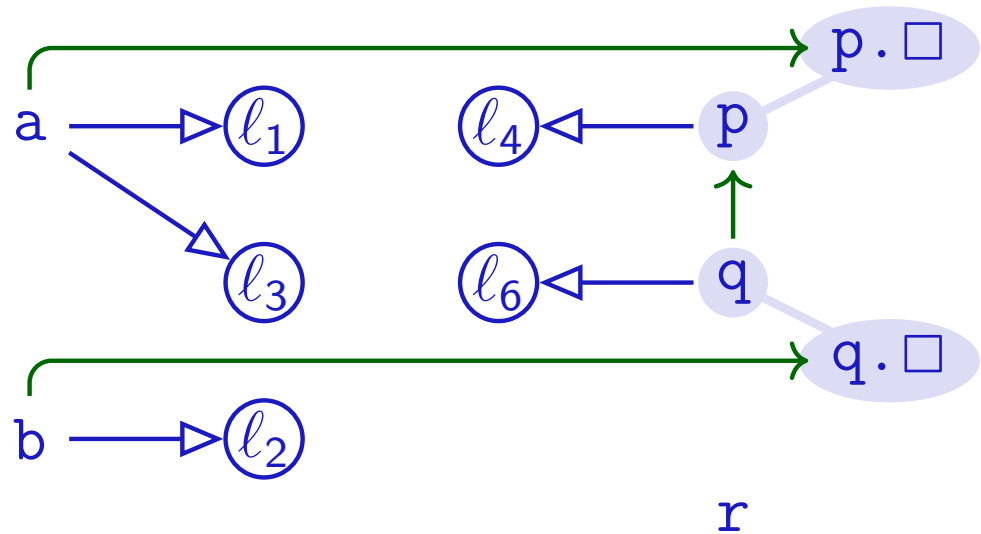
Example

	x := new$_{\ell_z}$	x$\rightarrowtriangle\ell_z$
$\Rightarrow$	x := y	x$\leftarrow$y
	x := y.$\square$	x$\leftarrow$y.$\square$
	x.$\square$:= y	x.$\square\leftarrow$y

- **Actual**:



- **Andersen**:



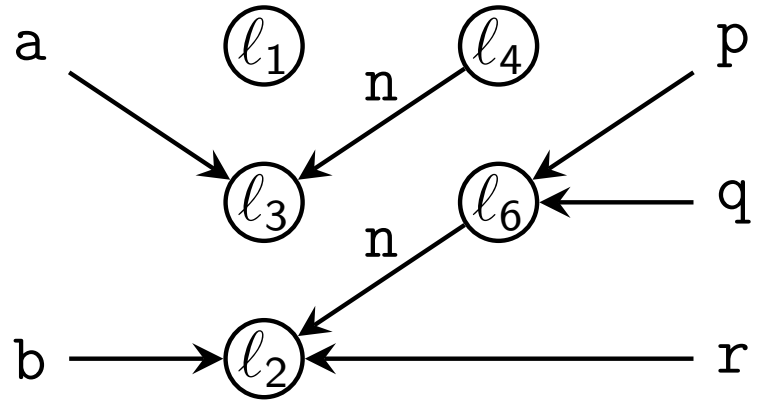
Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;              //⇐
var r := q.n;
```

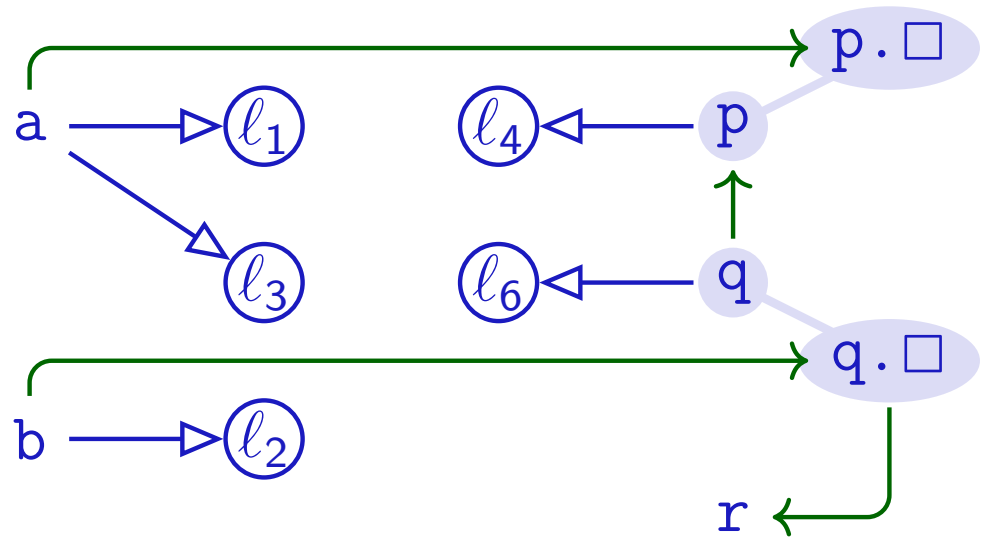
Example

`x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
`x.`$\square$ `:= y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

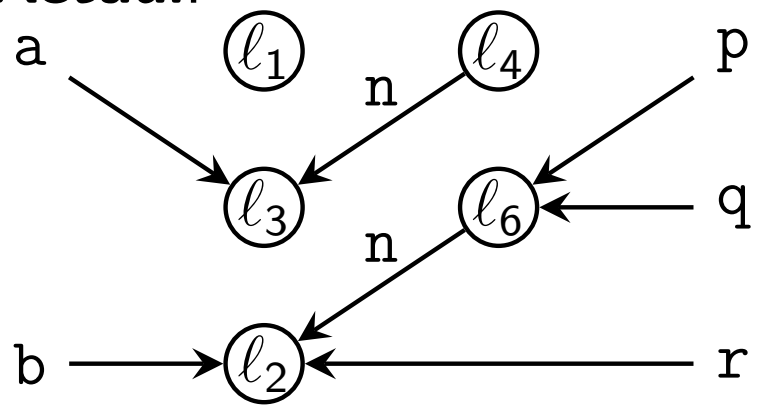
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;     //⇐
```

Example

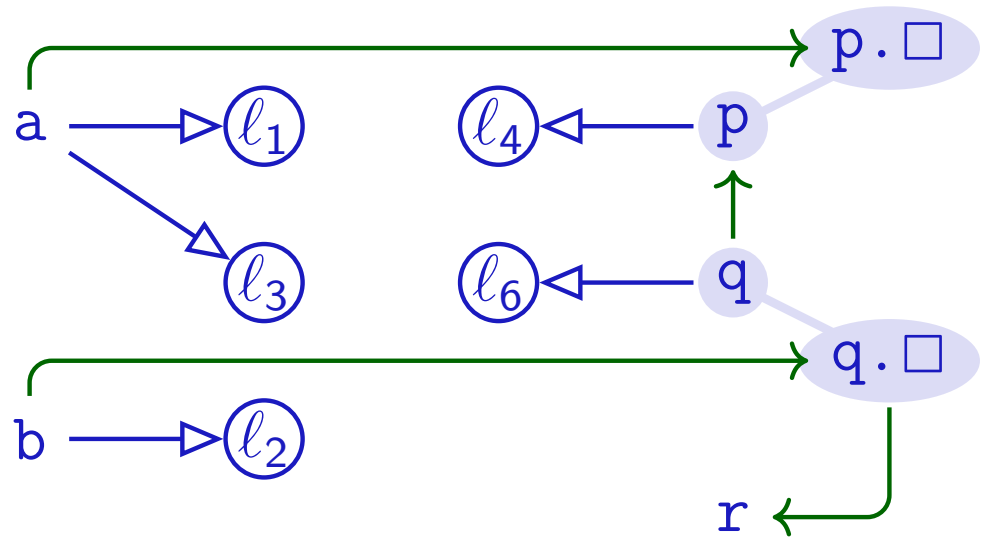
`x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
`x.`$\square$ `:= y`	$x.\square \leftarrow y$

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

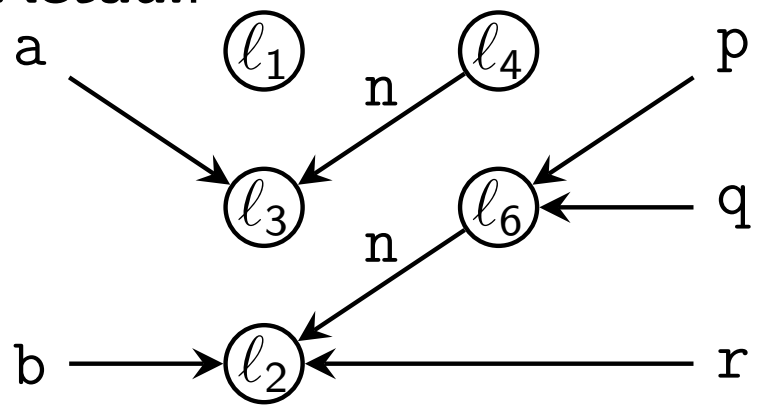
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

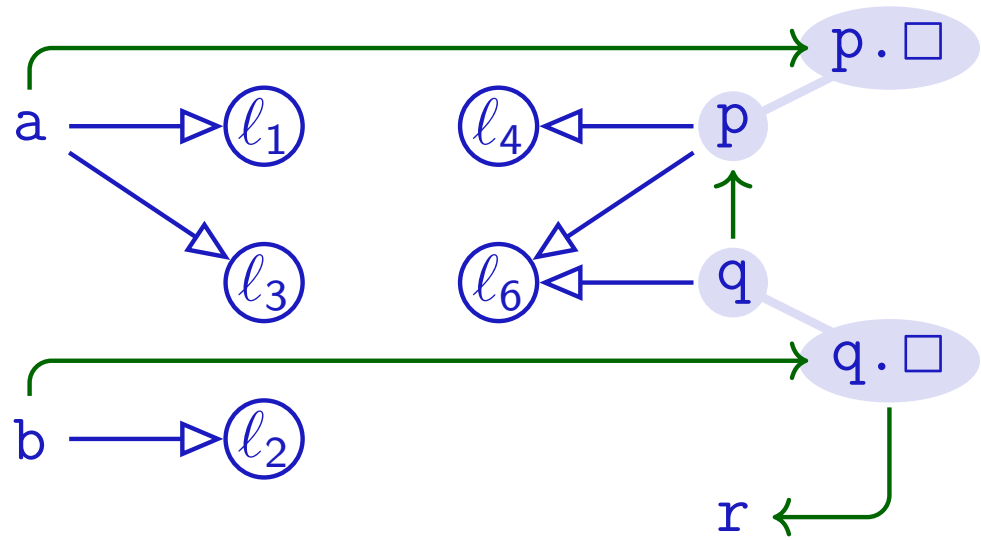
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

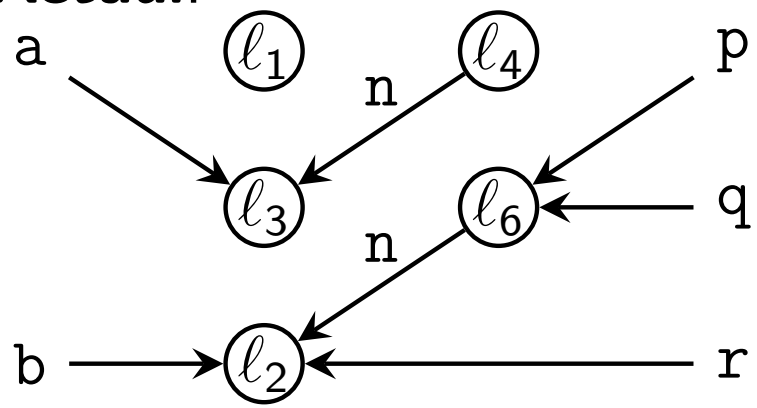
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

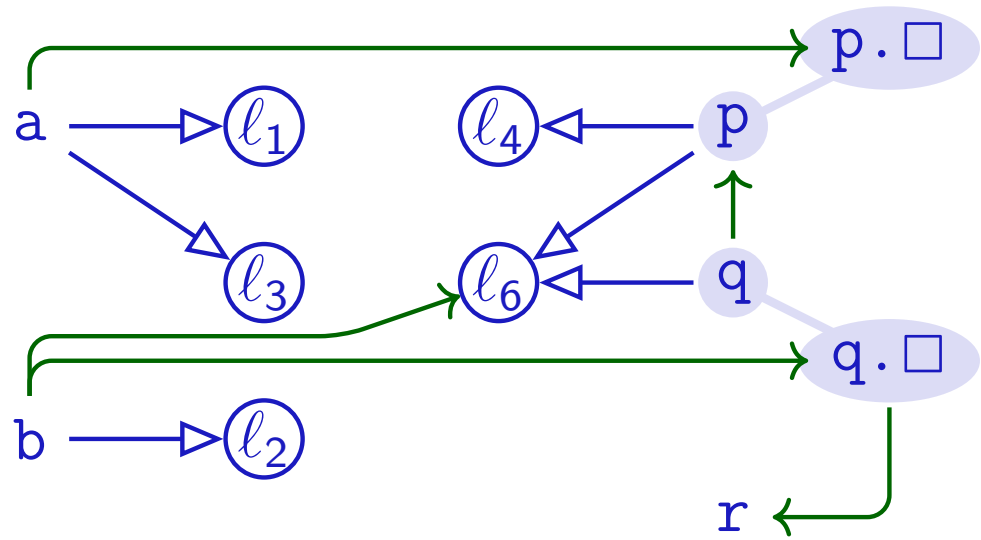
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

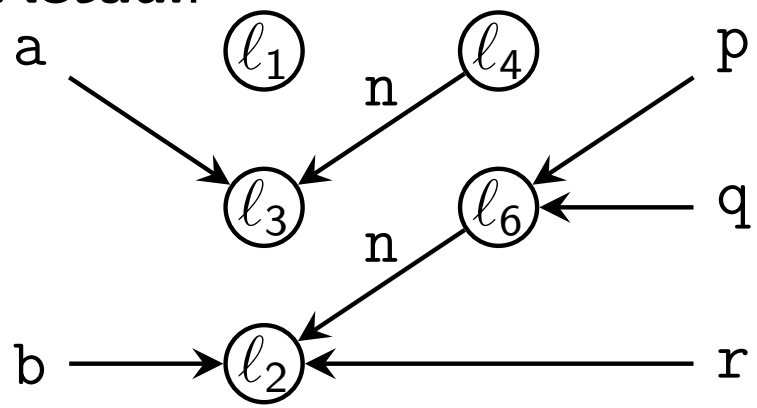
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

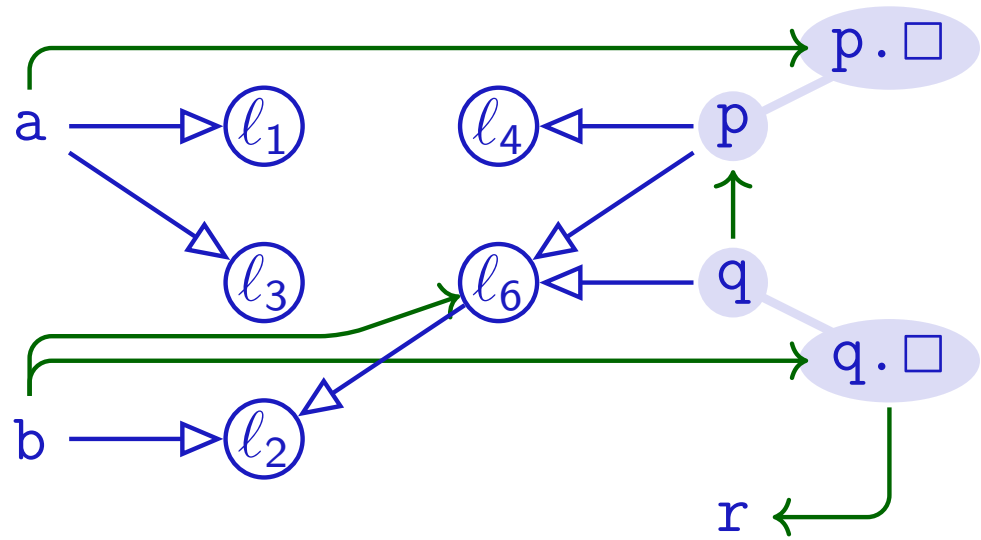
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

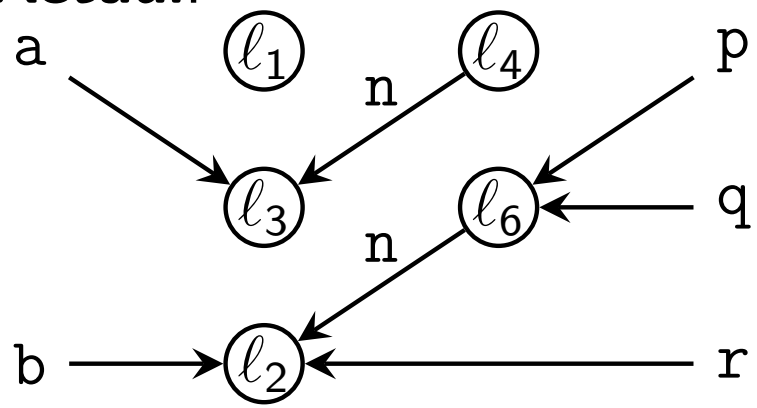
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

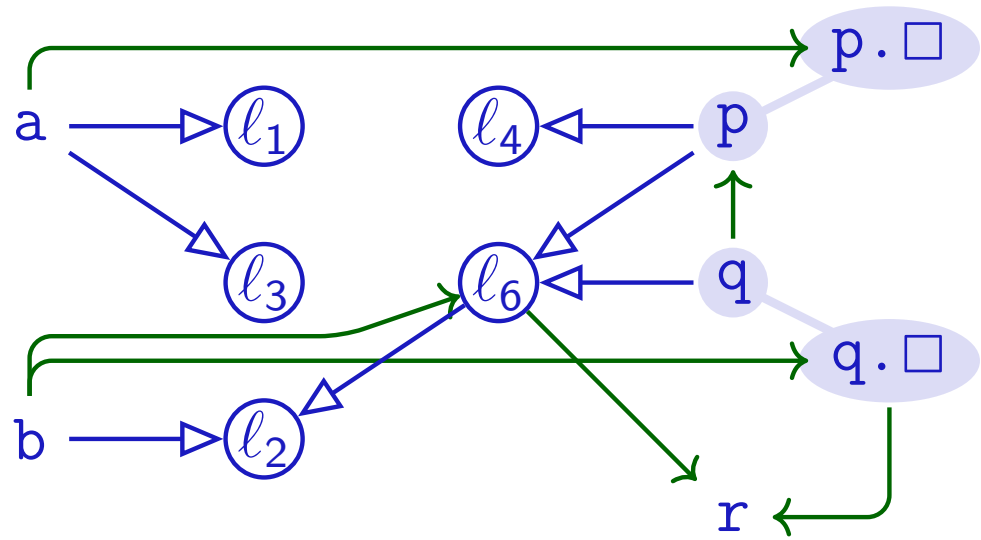
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

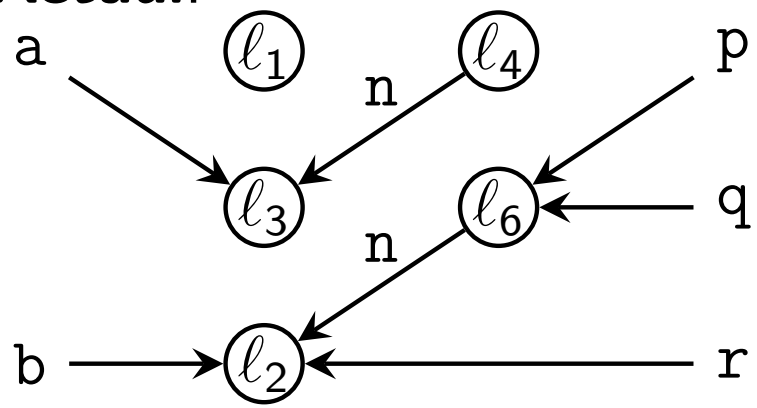
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

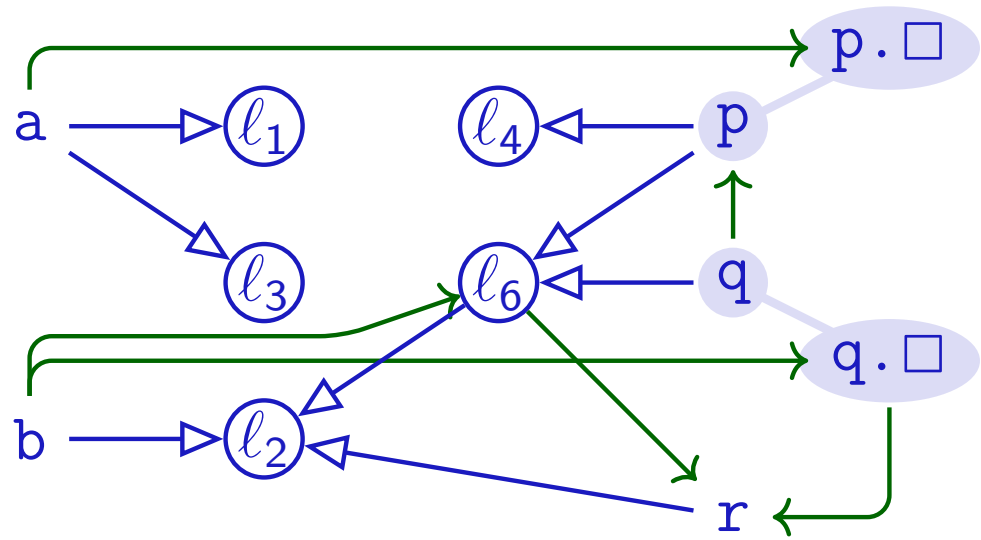
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

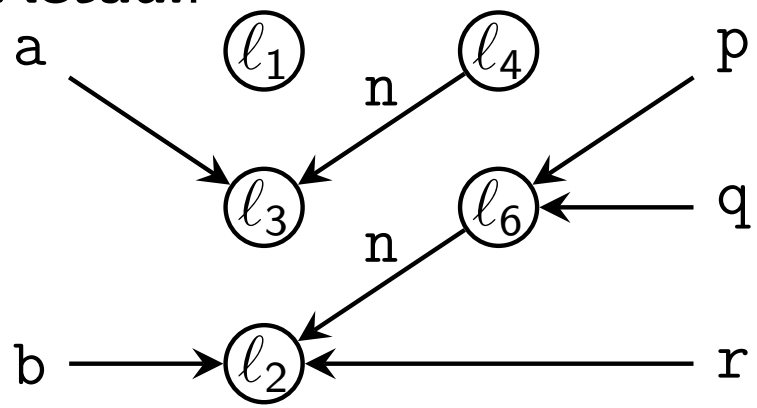
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

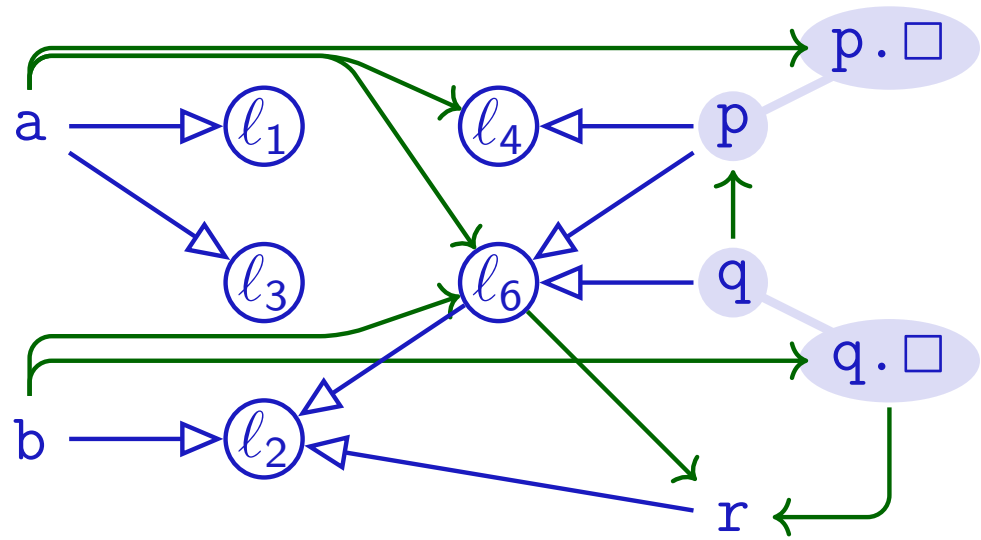
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

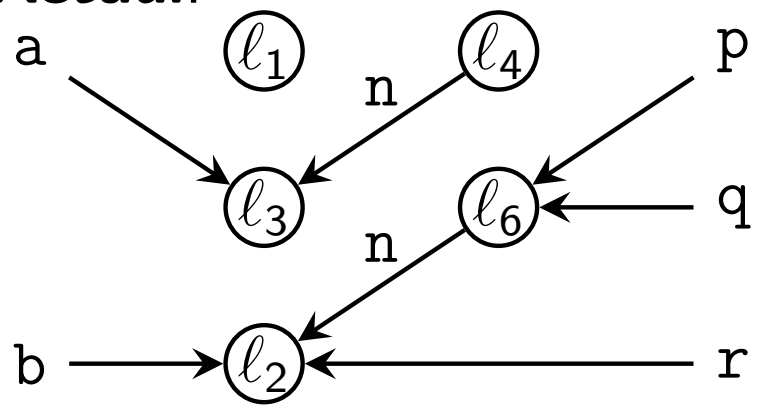
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

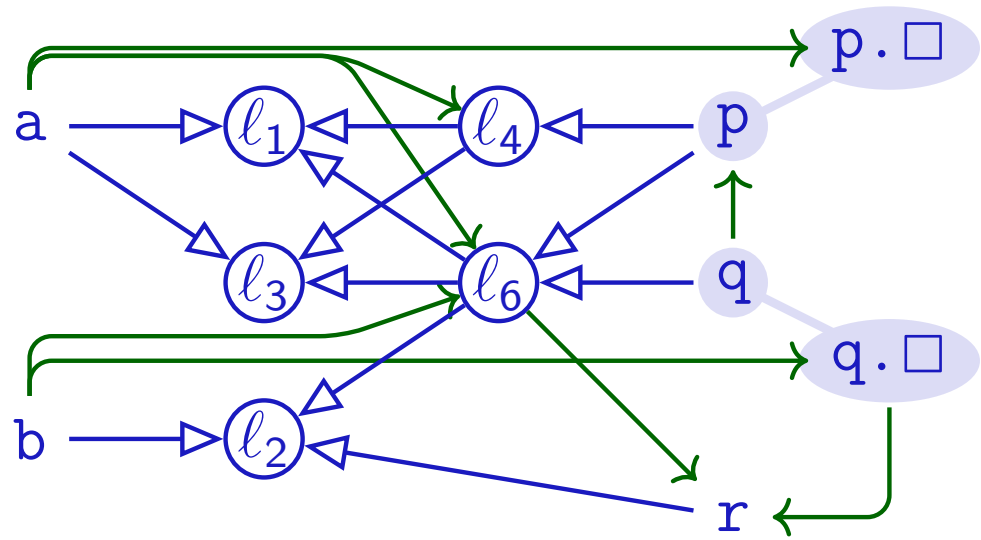
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrow x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

Implementation

- Graph structure
- Three types of edges: $\leftarrow\!\!\triangleleft$, $\leftarrow$, and indirect $\leftarrow$ (with a $\square$)
- Connection between x and $x.\square$
- Worklist:
  - Track all *new* edges (at start: *all* extracted edges)
  - Process one edge at a time:
    - Remove from worklist, add to "completed edges"
    - Check our thee rules: does current edge + completed edges allow producing new edge that is neither in worklist nor completed?
    - If so: add all such edges to worklist (may be several!)

$$\ell \leftarrow\!\!\triangleleft y \text{ and } x \leftarrow y \implies \ell \leftarrow\!\!\triangleleft x$$
$$v \leftarrow\!\!\triangleleft y \text{ and } x \leftarrow y.\square \implies x \leftarrow v$$
$$v \leftarrow\!\!\triangleleft x \text{ and } x.\square \leftarrow y \implies v \leftarrow y$$

Complexity

- Complexity of graph closure: $O(n^3)$
- Traditional assumption about Andersen's analysis
- Close to $O(n^2)$ if:
  1. Few statements dereference each variable
  2. Control flow graphs not too complex

  Both conditions are common in practical programs

  Manu Sridharan, Stephen J. Fink, "The Complexity of Andersen's Analysis in Practice", in SAS 2009

Summary

- Andersen's analysis:
  - Subset-based
  - Builds inclusion graph for propagating memory locations along subset constraints
  - $O(n^3)$ worst-case behaviour
  - Closer to $O(n^2)$ in practice
  - More precise than Steensgaard's analysis
  - Less scalable than Steensgaard's analysis

Lecture Overview