EDAP15: Program Analysis

POINTER ANALYSIS 2

Christoph Reichenbach

LUND UNIVERSITY

Welcome back!

Questions?

Lecture Overview

Foundations

Static Analysis

Dynamic Analysis

Properties

Control Flow

01 Foundations

02 Constructing Program Analyses in JastAdd

03 04 Types

05 06 07 Data Flow

08 09 **Memory**

05 Intraprocedural

10 Interprocedural

11 Indirect

12 Instrumentation

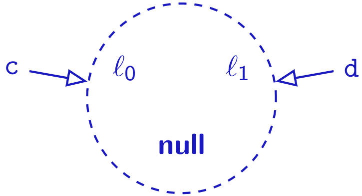
13 Analysis

14 Review

Alias Analysis in Practice (1/2)

Teal

```
var c := new_ℓ0();
var d := new_ℓ1();
if ... {
   c := null;
} else {
   d := null;
}
```



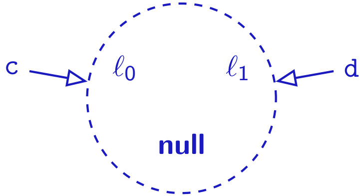
$\mathtt{c} \overset{\textit{alias}}{=\!=} \mathtt{d}$

null as unique memory location: Imprecision!

Alias Analysis in Practice (1/2)

Teal

```
var c := new_{ℓ_0}();
var d := new_{ℓ_1}();
if ... {
   c := null;
} else {
   d := null;
}
```



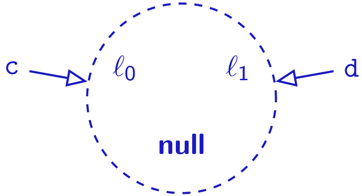
$$\mathtt{c} \overset{\textit{alias}}{=\!=} \mathtt{d}$$

null as unique memory location: Imprecision!

Alias Analysis in Practice (1/2)

Teal

```
var c := new_ℓ0();
var d := new_ℓ1();
if ...  {
   c := null;
} else {
   d := null;
}
```



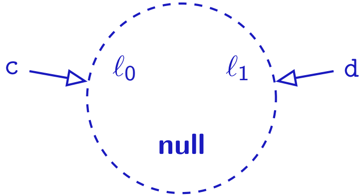
$$\mathtt{c} \overset{\textit{alias}}{=\!=} \mathtt{d}$$

null as unique memory location: Imprecision!

Alias Analysis in Practice (1/2)

Teal

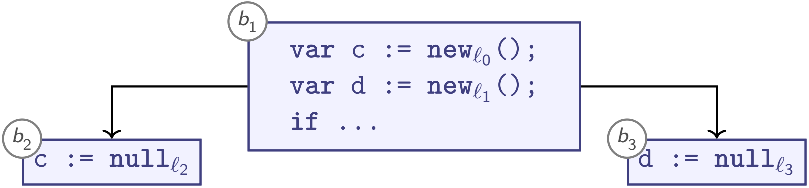
```
var c := new_ℓ0();
var d := new_ℓ1();
if ...  {
   c := null;
} else {
   d := null;
}
```



$\mathtt{c} \stackrel{\mathit{alias}}{=\!=} \mathtt{d}$

null as unique memory location: Imprecision!

Representing Null Pointers



1. One unique **null**

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

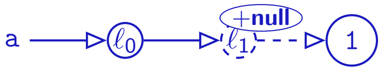
3. Nullness flags (Also more precise, minimal extra memory, but more complex analysis code)

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?
```



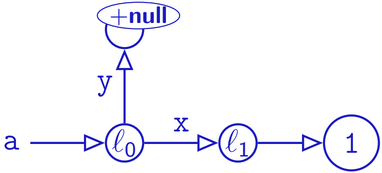
$$\texttt{a.x} \overset{\textit{alias}}{=\!=} \textbf{null} \overset{\textit{alias}}{=\!=} \texttt{a.y}$$

Field Sensitivity

- So far, we have merged all fields:

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

- Points-to analysis so far *field insensitive*
- Analogous for array indices
- A *field-sensitive* analysis would distinguish:



Summary

- Practical points to analysis usually wants to represent **null**
  - Single global **null** may reduce precision (unification-based analysis)
- Simple program analyses are **field insensitive**:

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

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

$$\texttt{a.x} \stackrel{alias}{\not=\!=} \texttt{a.y}$$

- Analogously for **Index-sensitive** analyses (for array indices)

Dataflow-Based Points-To Analysis

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

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

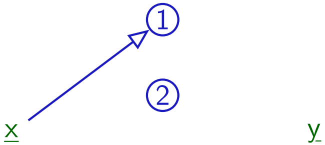
Example: Allocation and Update

Teal

 `var x :=` $\texttt{new}_1$`();`
$\Rightarrow$ `var y :=` $\texttt{new}_2$`();`

`if ... {`
  `y :=` $\texttt{new}_5$`();`
`}`

`x :=` $\texttt{new}_7$`();`
`y := x;`



Case	Transfer Function
`x :=` $\texttt{new}_\ell$`()`	$trans_1(\rightarrowtriangle) = (\rightarrowtriangle) \quad \cup \; \{\, x \rightarrowtriangle \ell \,\}$

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

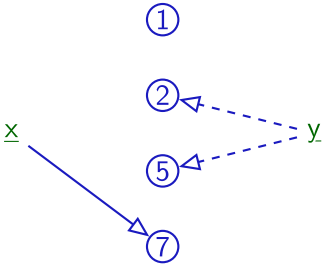
Example: Allocation and Update

Teal

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

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

⇒ x := new7();
   y := x;
```



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

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

Slight abuse of notation: writing $x {\rightarrow\!\triangleright} \ell$ for $\langle x, \ell \rangle$

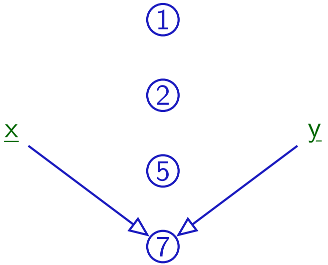
Example: Allocation and Update

Teal

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

    if ... {
      y := new₅();
    }

    x := new₇();
 ⇒  y := x;
```

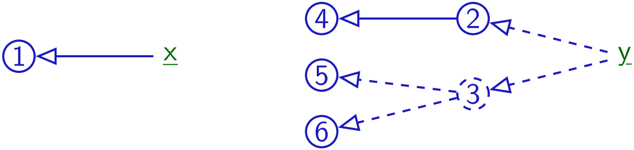


Case	**Transfer Function**
`x := new`$_\ell$`()`	$trans_1(\rightarrowtriangle) = (\rightarrowtriangle) \setminus (\{x\} \times \overline{L}) \ \cup \ \{ x \rightarrowtriangle \ell \}$
`y := x;`	$trans_2(\rightarrowtriangle) = (\rightarrowtriangle) \setminus (\{y\} \times \overline{L}) \ \cup \ \{ y \rightarrowtriangle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

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

Dereferencing Read

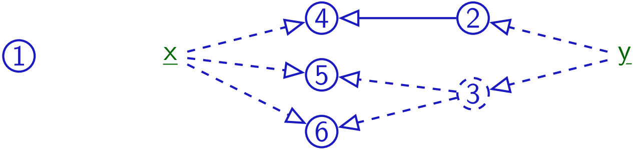
Before:



Teal

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

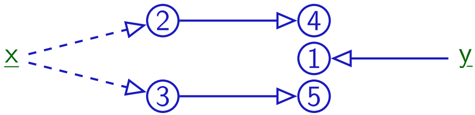
After:



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

Dereferencing Write

Before:

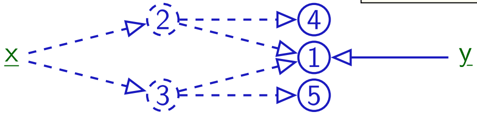


Teal

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

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

After:



Case	**Transfer Function**
`x.□ := y;`	$\mathit{trans}_4(\rightarrow) = (\rightarrow) \setminus (\{\underline{x}\} \times \overline{L}) \cup \{\, \ell \rightarrow \ell' \mid \underline{x} \rightarrow \ell,\ \underline{y} \rightarrow \ell' \}$

Weak vs Strong Update?

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

Teal

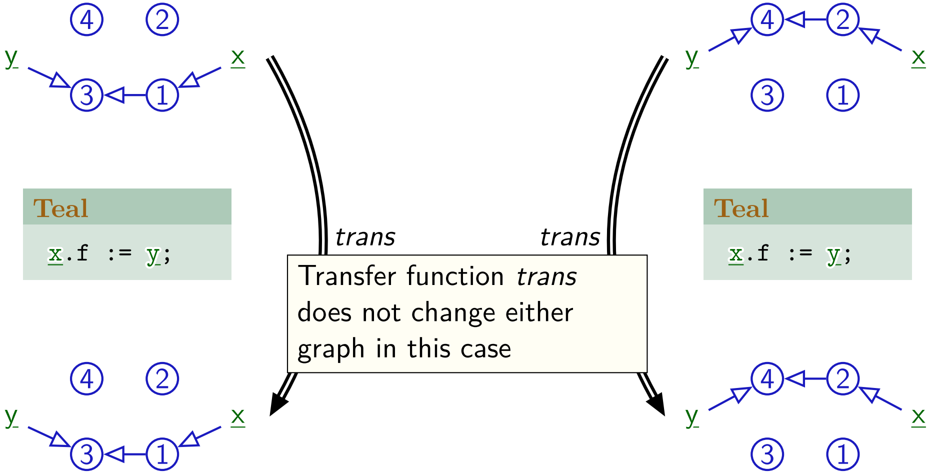
```
x.f := y;
```

Dataflow-Based Points-To Analysis

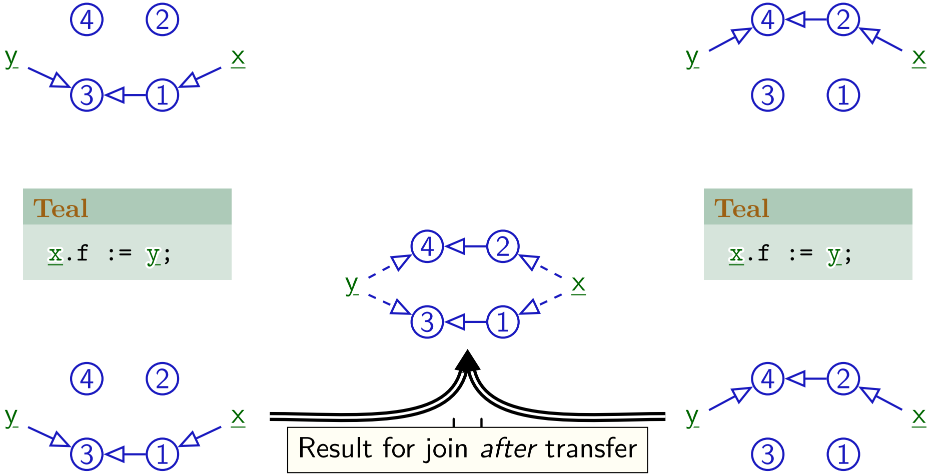
$$
\begin{array}{rcl}
G_{\mathsf{AHG}} & = & \langle \overline{L}, \rightarrowtriangle \rangle \\
(\rightarrowtriangle) & \subseteq & \overline{L} \times \overline{L}
\end{array}
$$

$$
\begin{array}{ll}
\underline{\mathtt{x}} \ \mathtt{:=}\ \mathbf{new}_{\ell}\mathtt{()} & \mathit{trans}_1(\rightarrowtriangle) = (\rightarrowtriangle) \setminus (\{\underline{\mathsf{x}}\} \times \overline{L}) \cup \{\ \underline{\mathsf{x}} \rightarrowtriangle \ell\ \} \\
\underline{\mathtt{x}} \ \mathtt{:=}\ \underline{\mathtt{y}}\mathtt{;} & \mathit{trans}_2(\rightarrowtriangle) = (\rightarrowtriangle) \setminus (\{\underline{\mathsf{x}}\} \times \overline{L}) \cup \{\ \underline{\mathsf{x}} \rightarrowtriangle \ell \mid \underline{\mathsf{y}} \rightarrowtriangle \ell \} \\
\underline{\mathtt{x}} \ \mathtt{:=}\ \underline{\mathtt{y}}\mathtt{.}\square\mathtt{;} & \mathit{trans}_3(\rightarrowtriangle) = (\rightarrowtriangle) \setminus (\{\underline{\mathsf{x}}\} \times \overline{L}) \cup \{\ \underline{\mathsf{x}} \rightarrowtriangle \ell' \mid \exists \ell.\ \underline{\mathsf{y}} \rightarrowtriangle \ell \rightarrowtriangle \ell' \} \\
\underline{\mathtt{x}}\mathtt{.}\square \ \mathtt{:=}\ \underline{\mathtt{y}}\mathtt{;} & \mathit{trans}_4(\rightarrowtriangle) = (\rightarrowtriangle) \cup \{\ \ell \rightarrowtriangle \ell' \mid \underline{\mathsf{x}} \rightarrowtriangle \ell,\ \underline{\mathsf{y}} \rightarrowtriangle \ell' \}
\end{array}
$$

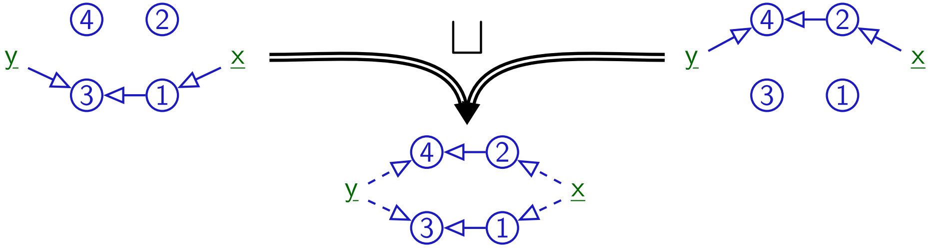
Distributivity?



Distributivity?



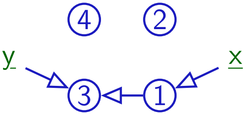
Distributivity?

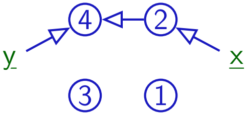


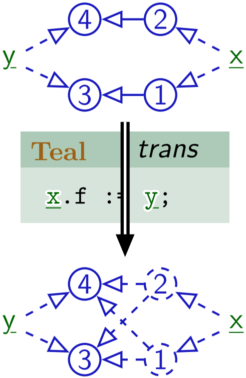
Teal

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

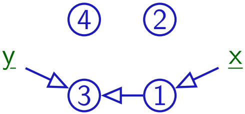
Distributivity?

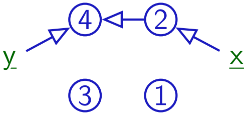


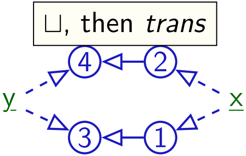


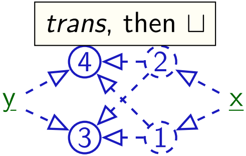


Distributivity?









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*)
- $\implies$ 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** ($\texttt{x := new}_{\ell_i}\ \texttt{A()}$):

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

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

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

Solving Constraints

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

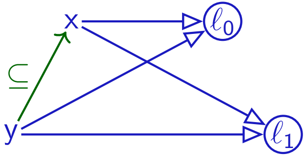
Subset Constraints (1/2)

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

```
y := new_ℓ0();
while ... {
  x := y;
  y := new_ℓ1();



}
```

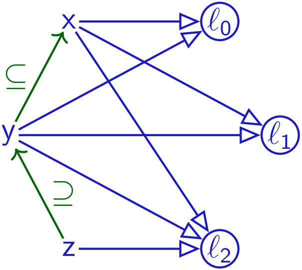


- $\ell \leftarrow y$ and $pt(x) \supseteq pt(y)$:
  $\implies \ell \leftarrow x$
- *Flow insensitive*: can't distinguish before/after

Subset Constraints (1/2)

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

```
y := new_ℓ0();
while ... {
  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

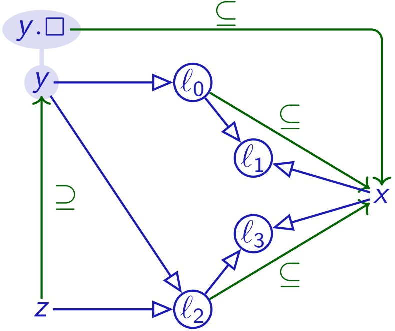
Solving one ($\supseteq$) can depend on all ($\leftarrow$) and ($\supseteq$) in program

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();
if ... {
  y := z;
}
x := y.n;
```

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



- Recall:
  $\ell \mathrel{\triangleleft\!\!-} z$ and $pt(y) \supseteq pt(z)$:
  $\implies \ell \mathrel{\triangleleft\!\!-} y$
- $\ell \mathrel{\triangleleft\!\!-} y$ and $pt(x) \supseteq pt(y.\square)$:
  $\implies pt(x) \supseteq pt(\ell)$

Fresh Assignments to Fields

- Recall:
  ```
  y.n := new_ℓ1();
  ```
  $\texttt{y.n := }\mathbf{new}_{\ell_1}\texttt{();}$
- No direct pattern for this code
- Can model as:
  $\mathbf{var}\ \texttt{tmp := }\mathbf{new}_{\ell_1}\texttt{();}$
  $\texttt{y.n := tmp;}$

Solving Constraints

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

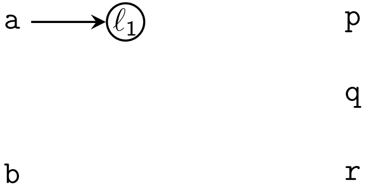
Example

$\Rightarrow$	`x := new`$_{\ell_z}$	$x \rightarrow\!\!\triangleright\, \ell_z$
	`x := y`	$x \leftarrow y$
	`x := y.`$\square$	$x \leftarrow y.\square$
	`x.`$\square$ ` := y`	$x.\square \leftarrow y$

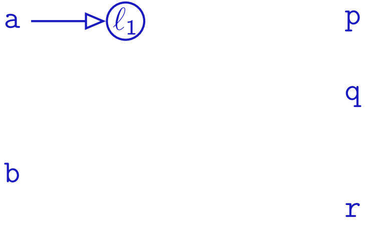
Teal

```
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**:



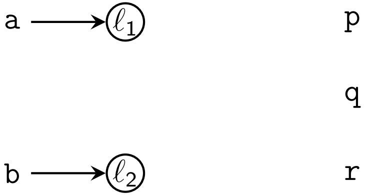
- **Andersen**:



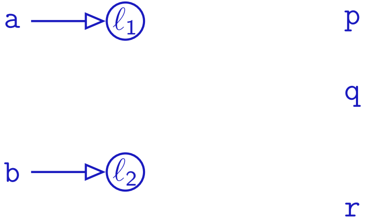
Example

$\Rightarrow$	$\texttt{x := new}_{\ell_z}$	$\texttt{x} \rightarrow\!\triangleright\, \ell_z$
	$\texttt{x := y}$	$\texttt{x} \leftarrow \texttt{y}$
	$\texttt{x := y.}\square$	$\texttt{x} \leftarrow \texttt{y.}\square$
	$\texttt{x.}\square\texttt{ := y}$	$\texttt{x.}\square \leftarrow \texttt{y}$

- **Actual**:



- **Andersen**:



Teal

$\texttt{var a := new}_{\ell_1}\texttt{();}$
$\texttt{var b := new}_{\ell_2}\texttt{();}$ $\texttt{//}\Leftarrow$
$\texttt{a := new}_{\ell_3}\texttt{();}$
$\texttt{var p := new}_{\ell_4}\texttt{();}$
$\texttt{p.n := a;}$
$\texttt{var q := new}_{\ell_6}\texttt{();}$
$\texttt{q.n := b;}$
$\texttt{p := q;}$
$\texttt{var r := q.n;}$

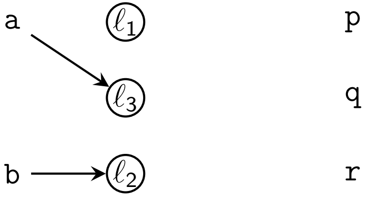
Example

$\Rightarrow$	`x := new`$_{\ell_z}$	$x \rightarrow\!\!\triangleright\, \ell_z$
	`x := y`	$x \leftarrow y$
	`x := y.`$\square$	$x \leftarrow y.\square$
	`x.`$\square$ ` := y`	$x.\square \leftarrow y$

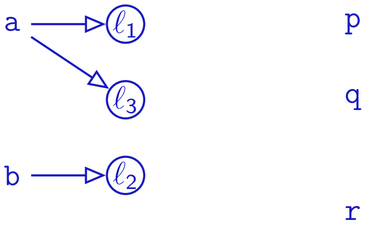
Teal

```
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**:



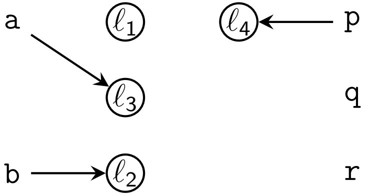
Example

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

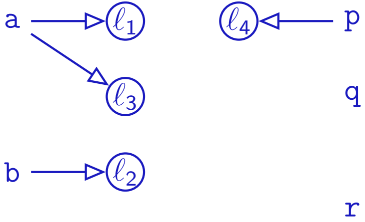
Teal

```
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**:



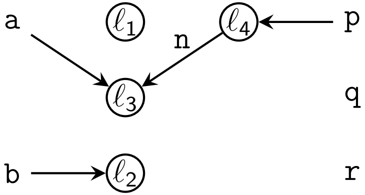
Example

	`x := new`$_{\ell_z}$	$\mathtt{x} \rightarrow\!\triangleright \ell_z$
	`x := y`	$\mathtt{x} \leftarrow \mathtt{y}$
	`x := y.□`	$\mathtt{x} \leftarrow \mathtt{y.□}$
$\Rightarrow$	`x.□ := y`	$\mathtt{x.□} \leftarrow \mathtt{y}$

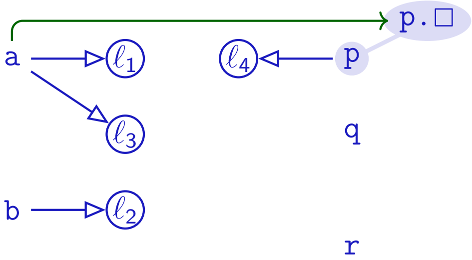
Teal

```
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

$\Rightarrow$ `x := new`$_{\ell_z}$ $\quad$ $x \rightarrow\!\triangleright\, \ell_z$

`x := y` $\quad$ $x \leftarrow y$

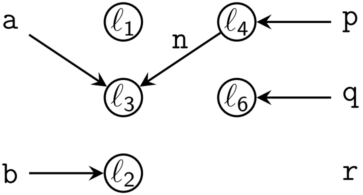
`x := y.`$\square$ $\quad$ $x \leftarrow y.\square$

`x.`$\square$ `:= y` $\quad$ $x.\square \leftarrow y$

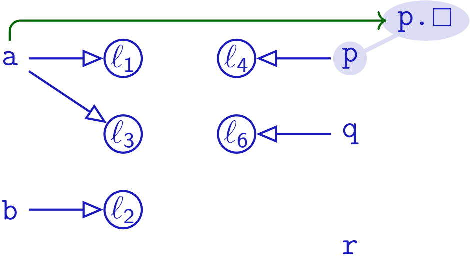
Teal

```
var a := newℓ1();
var b := newℓ2();
a := newℓ3();
var p := newℓ4();
p.n := a;
var q := newℓ6(); //⇐
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



- **Andersen**:



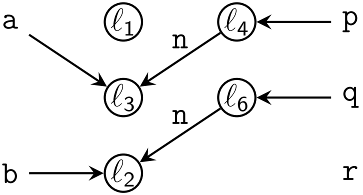
Example

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

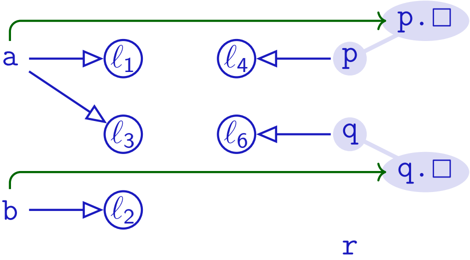
Teal

```
var a := newℓ1();
var b := newℓ2();
a := newℓ3();
var p := newℓ4();
p.n := a;
var q := newℓ6();
q.n := b;           //⇐
p := q;
var r := q.n;
```

- **Actual**:



- **Andersen**:



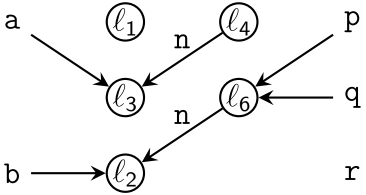
Example

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

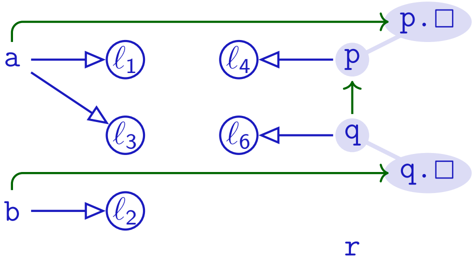
Teal

```
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**:



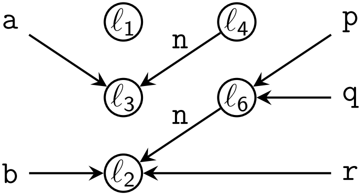
Example

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

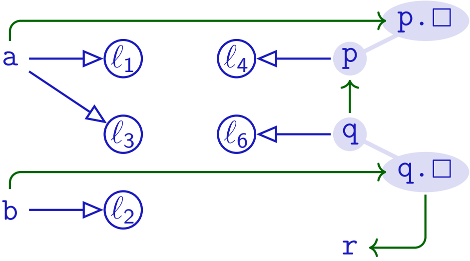
Teal

```
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

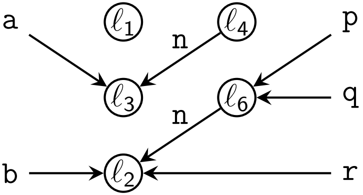
`x := new`$_{\ell_z}$	`x`$\mathrel{-\!\!\triangleright}\ell_z$
`x := y`	`x`$\leftarrow$`y`
`x := y.□`	`x`$\leftarrow$`y.□`
`x.□ := y`	`x.□`$\leftarrow$`y`

$\ell \mathrel{\triangleleft\!\!-} y$ and $x \leftarrow y \implies \ell \mathrel{\triangleleft\!\!-} x$
$\ell \mathrel{\triangleleft\!\!-} y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \mathrel{\triangleleft\!\!-} x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

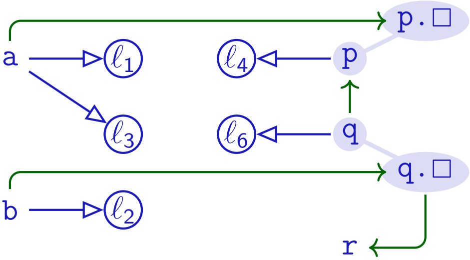
Teal

```
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**:



Andersen's algorithm must propagate along inclusion graph

Example

$\ell \vartriangleleft\!\!- y$ and $x \leftarrow y$ $\Longrightarrow$ $\ell \vartriangleleft\!\!- x$

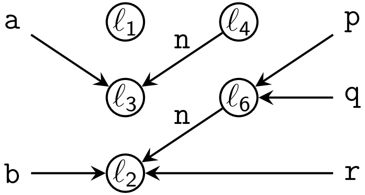
$\ell \vartriangleleft\!\!- y$ and $x \leftarrow y.\square$ $\Longrightarrow$ $x \leftarrow \ell$

$\ell \vartriangleleft\!\!- x$ and $x.\square \leftarrow y$ $\Longrightarrow$ $\ell \leftarrow y$

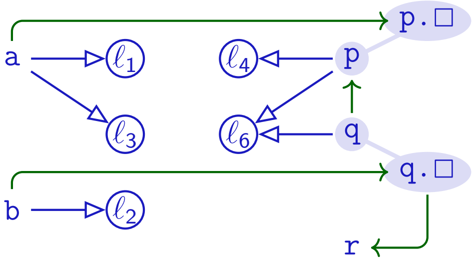
Teal

```
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**:



Andersen's algorithm must propagate along inclusion graph

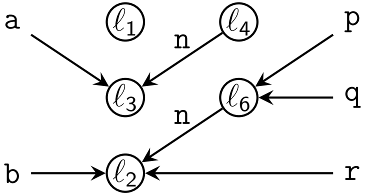
Example

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

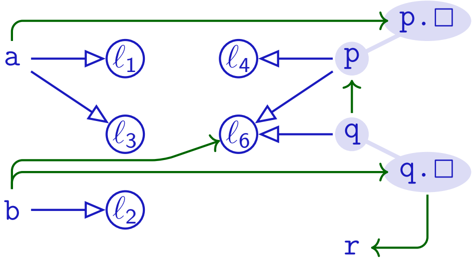
Teal

```
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**:



Andersen's algorithm must propagate along inclusion graph

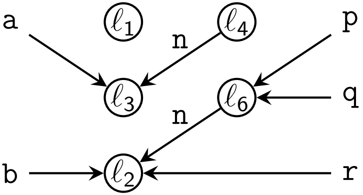
Example

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

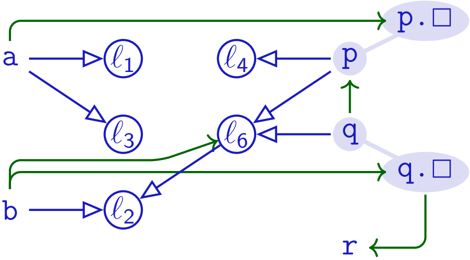
Teal

```
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**:



Andersen's algorithm must propagate along inclusion graph

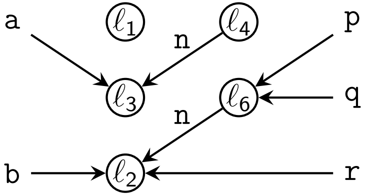
Example

$\ell \mathrel{\triangleleft\!\!-} y$ and $x \leftarrow y \implies \ell \mathrel{\triangleleft\!\!-} x$
$\ell \mathrel{\triangleleft\!\!-} y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \mathrel{\triangleleft\!\!-} x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

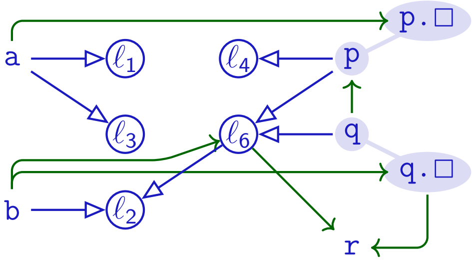
Teal

```
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**:



Andersen's algorithm must propagate along inclusion graph

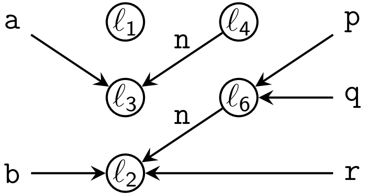
Example

$\ell \mathrel{\lhd\!\!-} y$ and $x \leftarrow y \implies \ell \mathrel{\lhd\!\!-} x$
$\ell \mathrel{\lhd\!\!-} y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \mathrel{\lhd\!\!-} x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

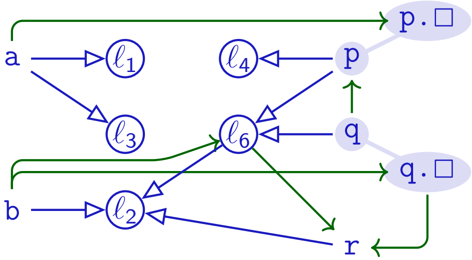
Teal

```
var a := newℓ1();
var b := newℓ2();
a := newℓ3();
var p := newℓ4();
p.n := a;
var q := newℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



- **Andersen**:



Andersen's algorithm must propagate along inclusion graph

Example

$\ell \mathrel{\triangleleft\!\!-} y$ and $x \leftarrow y \implies \ell \mathrel{\triangleleft\!\!-} x$

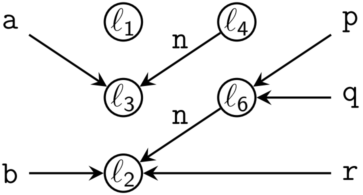
$\ell \mathrel{\triangleleft\!\!-} y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$

$\ell \mathrel{\triangleleft\!\!-} x$ and $x.\square \leftarrow y \implies \ell \leftarrow y$

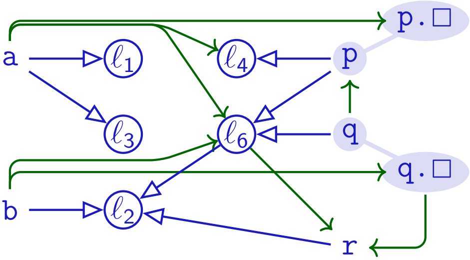
Teal

```
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**:

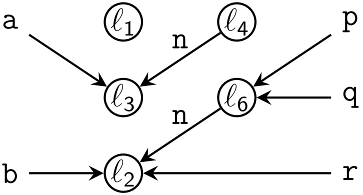


Andersen's algorithm must propagate along inclusion graph

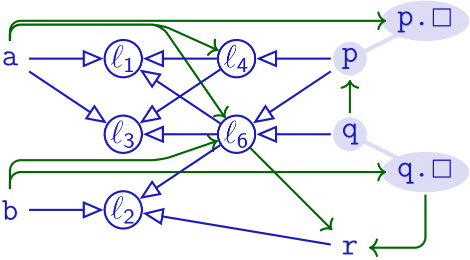
Example

$\ell \leftarrowtriangle y$ and $x \leftarrow y \implies \ell \leftarrowtriangle x$
$\ell \leftarrowtriangle y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$
$\ell \leftarrowtriangle x$ 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: $\leftarrowtriangle$, $\leftarrow$, 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!)

$$
\begin{aligned}
\ell \leftarrowtriangle y \text{ and } x \leftarrow y \quad &\Longrightarrow \ell \leftarrowtriangle x \\
v \leftarrowtriangle y \text{ and } x \leftarrow y.\square &\Longrightarrow x \leftarrow v \\
v \leftarrowtriangle x \text{ and } x.\square \leftarrow y &\Longrightarrow v \leftarrow y
\end{aligned}
$$

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

Foundations

- 01 Foundations
- 02 Constructing Program Analyses in JastAdd

Static Analysis

Properties

- 03 04 Types
- 05 06 07 Data Flow
- 08 09 **Memory**

Control Flow

- 05 Intraprocedural
- 10 Interprocedural
- 11 Indirect

Dynamic Analysis

- 12 Instrumentation
- 13 Analysis

- 14 Review