Lund University

EDAP15: Program Analysis

POINTER ANALYSIS 2

Christoph Reichenbach

Welcome back!

- Questions?
- Lab-1 presentations coming up
- Lab-2 release tomorrow
- No office hours today

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

    $$\ell_i \in pts(\texttt{x}) \qquad (x \rightarrow \ell_i)$$

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

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

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

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

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

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

Solving Constraints

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

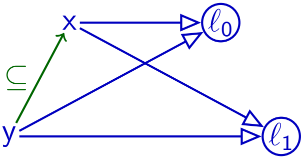
Subset Constraints (1/2)

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

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



}
```

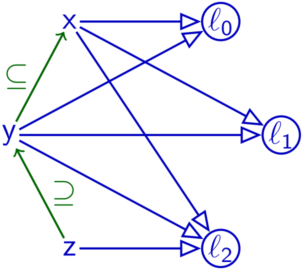


- $\ell \leftarrow y$ and $pts(x) \supseteq pts(y)$:
  $\Longrightarrow \ell \leftarrow x$
- *Flow insensitive*: can't distinguish before/after

Subset Constraints (1/2)

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

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



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

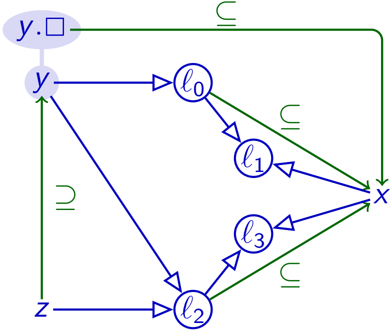
Solving one ($\supseteq$) can depend on all ($\leftarrow$) and ($\supseteq$) in program

Subset Constraints (2/2)

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

```
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:

  $\ell \leftarrow z$ and $pts(y) \supseteq pts(z)$:

  $\Longrightarrow \ \ell \leftarrow y$

- $\ell \leftarrow y$ and $pts(x) \supseteq pts(y.\square)$:

  $\Longrightarrow \ pts(()x) \supseteq pts(()\ell)$

Fresh Assignments to Fields

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

  var `tmp :=` $\mathbf{new}_{\ell_1}$`();`

  `y.n := tmp;`

Solving Constraints

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

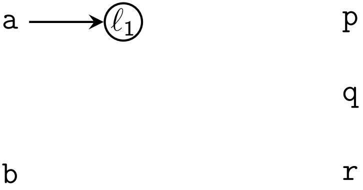
Example

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

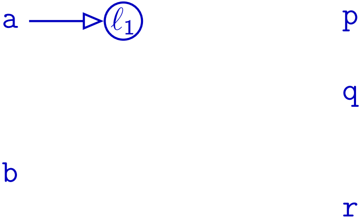
Teal

```
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**:



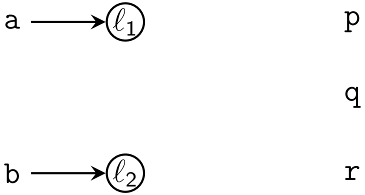
- **Andersen**:



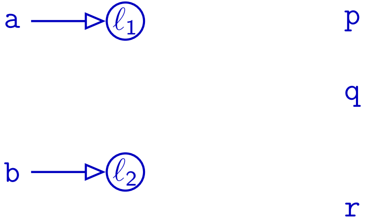
Example

⇒	x := new$_{\ell_z}$	x$\rightarrow\!\triangleright\,\ell_z$
	x := y	x$\leftarrow$y
	x := y.□	x$\leftarrow$y.□
	x.□ := y	x.□$\leftarrow$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;
```

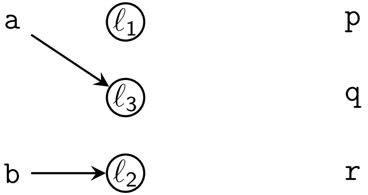
Example

⇒	$\texttt{x := new}_{\ell_z}$	$\texttt{x} \multimap \ell_z$
	`x := y`	$\texttt{x} \leftarrow \texttt{y}$
	`x := y.□`	$\texttt{x} \leftarrow \texttt{y.□}$
	`x.□ := y`	$\texttt{x.□} \leftarrow \texttt{y}$

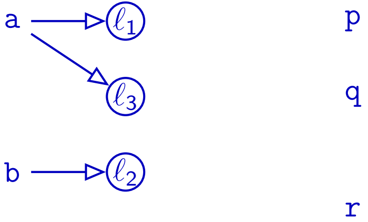
Teal

```
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

$\Rightarrow$ $\mathtt{x := new}_{\ell_z}$ $\quad \mathtt{x} \rightarrowtriangle \ell_z$

$\mathtt{x := y}$ $\quad \mathtt{x} \leftarrow \mathtt{y}$

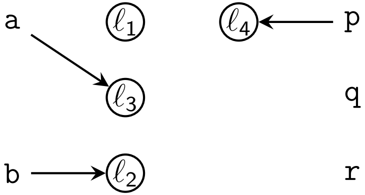
$\mathtt{x := y.}\square$ $\quad \mathtt{x} \leftarrow \mathtt{y.}\square$

$\mathtt{x.}\square\ \mathtt{:= y}$ $\quad \mathtt{x.}\square \leftarrow \mathtt{y}$

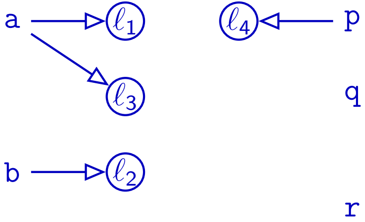
Teal

```
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**:



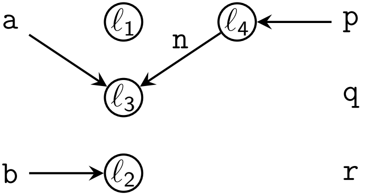
Example

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

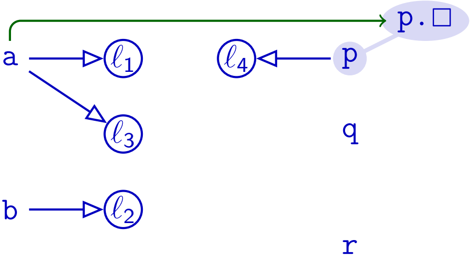
Teal

```
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**:



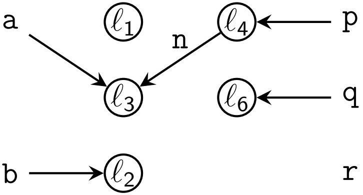
Example

⇒	x := new$_{\ell_z}$	x$\rightarrow\!\!\triangleright$$\ell_z$
	x := y	x←y
	x := y.□	x←y.□
	x.□ := y	x.□←y

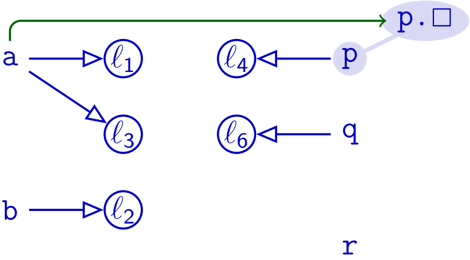
Teal

```
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**:



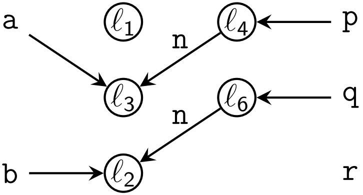
- **Andersen**:



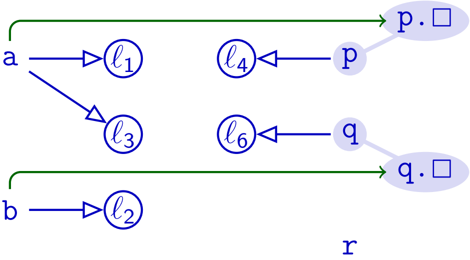
Example

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

- **Actual**:



- **Andersen**:



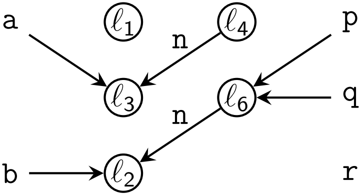
Teal

```
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;
```

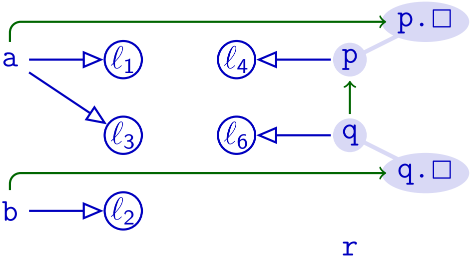
Example

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

- **Actual**:



- **Andersen**:



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;
```

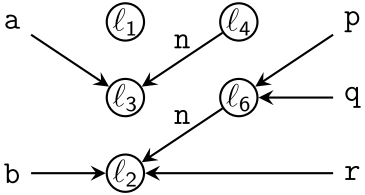
Example

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

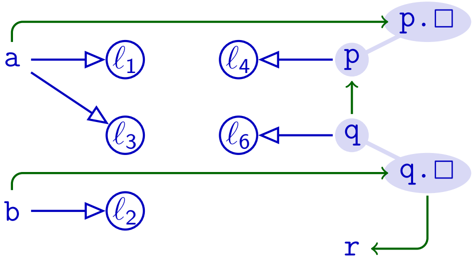
Teal

```
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`$_{\ell_z}$	$x \multimap \ell_z$
`x := y`	$x \leftarrow y$
`x := y.□`	$x \leftarrow y.\square$
`x.□ := y`	$x.\square \leftarrow y$

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

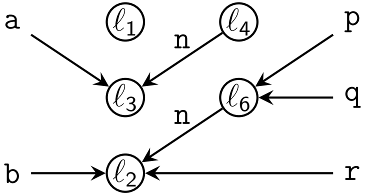
$\ell \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow \ell$

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

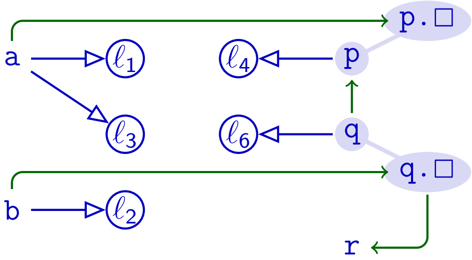
Teal

```
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

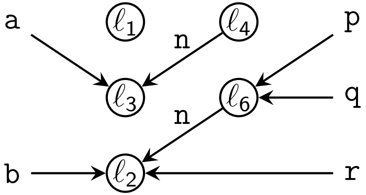
Example

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

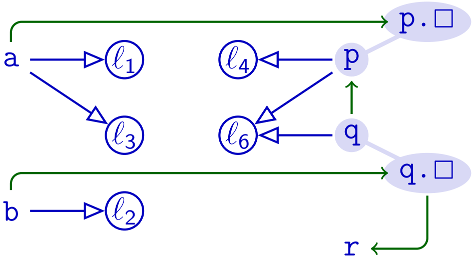
Teal

```
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

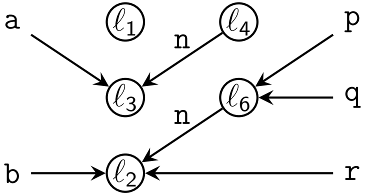
Example

$\ell \triangleleft\!\!- y$ and $x \leftarrow y \implies \ell \triangleleft\!\!- x$
$\ell \triangleleft\!\!- y$ and $x \leftarrow y.\Box \implies x \leftarrow \ell$
$\ell \triangleleft\!\!- x$ and $x.\Box \leftarrow y \implies \ell \leftarrow y$

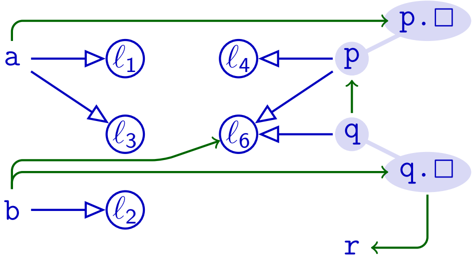
Teal

```
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

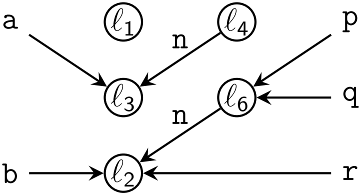
Example

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

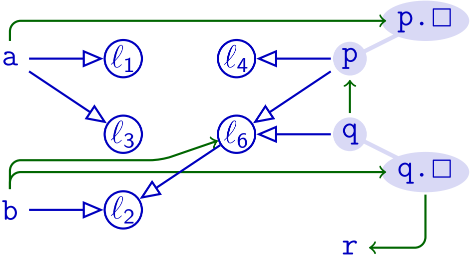
Teal

```
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

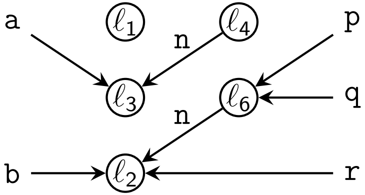
Example

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

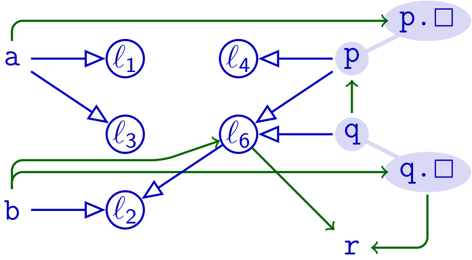
Teal

```
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

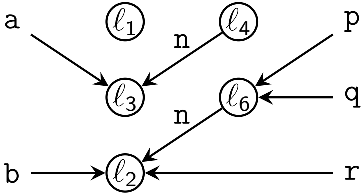
Example

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

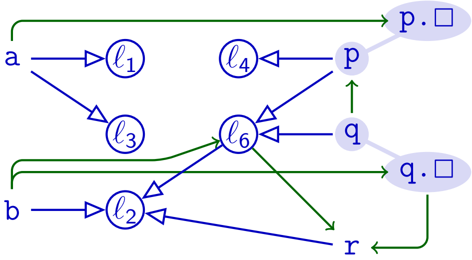
Teal

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

- **Actual**:



- **Andersen**:



Andersen's algorithm must propagate along inclusion graph

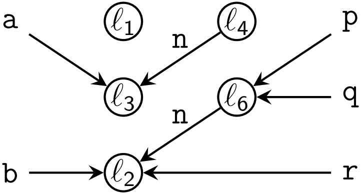
Example

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

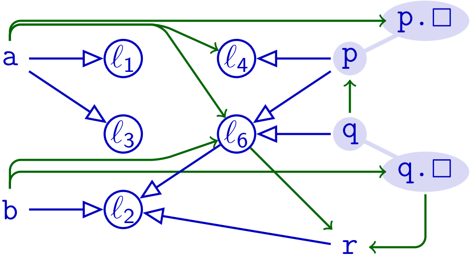
Teal

```
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**:



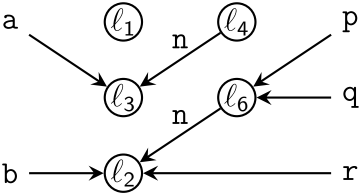
- **Andersen**:



Andersen's algorithm must propagate along inclusion graph

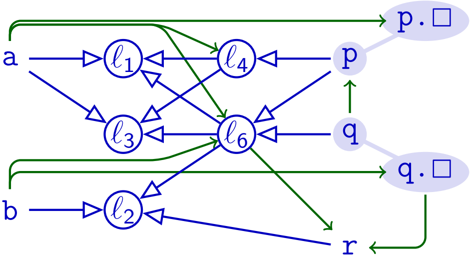
Example

- **Actual**:



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

- **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
- Two types of edges
- Connection between x and $x.\Box$
- 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!)

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

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

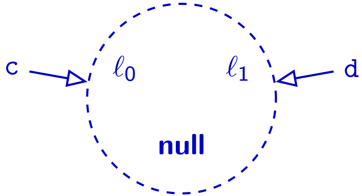
Summary

- 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

Alias Analysis in Practice (1/2)

Teal

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



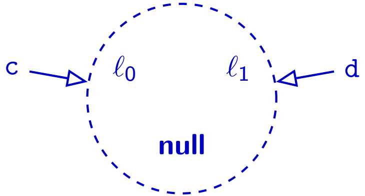
$c \overset{alias}{=\!=} d$

null as unique memory location: Imprecision!

Alias Analysis in Practice (1/2)

Teal

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



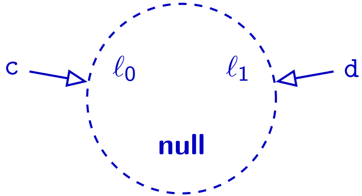
$c \overset{alias}{=\!=} d$

null as unique memory location: Imprecision!

Alias Analysis in Practice (1/2)

Teal

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



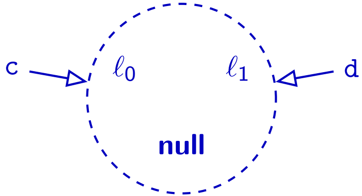
$\texttt{c} \overset{alias}{=\!=} \texttt{d}$

null as unique memory location: Imprecision!

Alias Analysis in Practice (1/2)

Teal

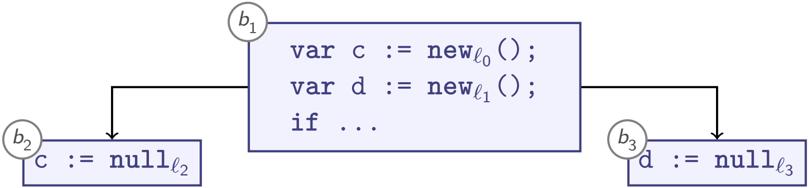
```
var c := new_ℓ0();
var d := new_ℓ1();
if ...  {
   c := null;
} else {
   d := null;
}
```



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

null as unique memory location: Imprecision!

Representing Null Pointers



1. One unique **null**

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

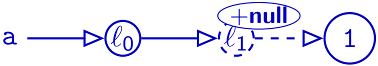
3. Nullness flags (Also more precise, minimal extra memory, but more complex analysis code)

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?
```



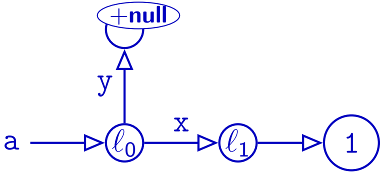
$$\texttt{a.x} \overset{\textit{alias}}{=\!=} \textbf{null} \overset{\textit{alias}}{=\!=} \texttt{a.y}$$

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 must represent **null**
  - Single global **null** may reduce precision
- Simple program analyses are **field insensitive**:

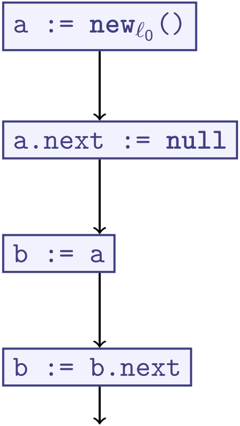
$$\texttt{a.x} \overset{\textit{alias}}{=\!=} \texttt{a.}\Box \overset{\textit{alias}}{=\!=} \texttt{a.y}$$

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

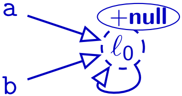
$$\texttt{a.x} \;\not\overset{\textit{alias}}{=\!=}\; \texttt{a.y}$$

- Analogously for array indices

Flow-(In)Sensitive Points-To Analysis

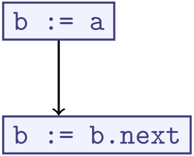


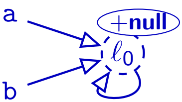
Flow Insensitive



$$\mathtt{a} \overset{\textit{alias}}{=\!=} \mathtt{a.next} \overset{\textit{alias}}{=\!=} \mathtt{b} \overset{\textit{alias}}{=\!=} \mathtt{b.next} \overset{\textit{alias}}{=\!=} \textbf{null}$$

Weak Updates





$\texttt{a} \overset{alias}{=\!=} \texttt{a.next} \overset{alias}{=\!=} \texttt{b} \overset{alias}{=\!=} \texttt{b.next} \overset{alias}{=\!=} \textbf{null}$

- Interpretation of updates in this analysis only adds, never removes:

`b := a` $\quad \left[\ pts(\texttt{b}) \mapsto pts(\texttt{a}) \cup pts(\texttt{b})\ \right]$

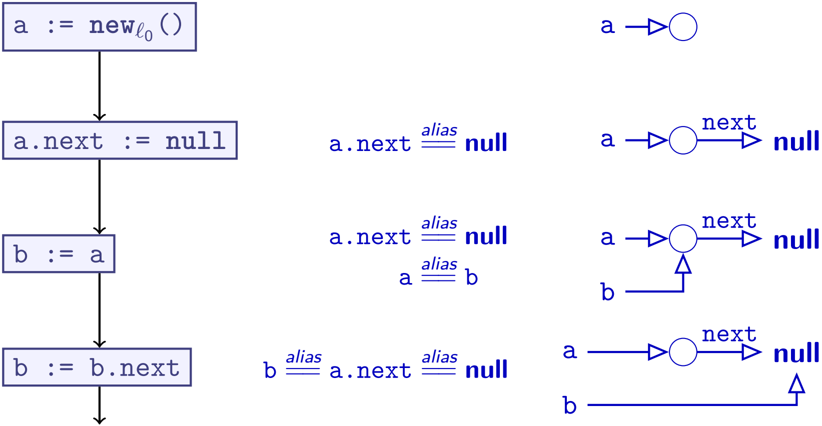
- *Weak Update*
- Flow-insensitive points-to analyses only use weak updates

Points-To from Dataflow Analysis

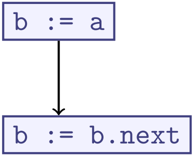
- Most (scalable) points-to analyses are flow insensitive

  $\Longrightarrow$ One global ($\stackrel{alias}{=\!=}$) relation
- *Flow-sensitive points-to analysis*:
  - Allows different Abstract Heap Graphs per basic block
  - Analogously ($\stackrel{alias}{=\!=}$) per basic block
  - Higher precision
- Feasible on small modules, commonly computed via Data Flow analysis

Flow-(In)Sensitive Points-To Analysis



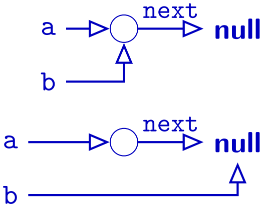
Strong and Weak Updates



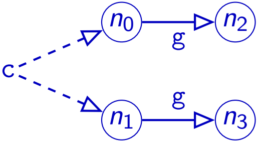
$\texttt{a.next} \stackrel{alias}{=\!=} \textbf{null}$

$\texttt{a} \stackrel{alias}{=\!=} \texttt{b}$

$\texttt{b} \stackrel{alias}{=\!=} \texttt{a.next} \stackrel{alias}{=\!=} \textbf{null}$



- Flow-sensitive points-to analysis enables *strong updates*:
  - Remove information that is overwritten by update



- Imprecision still arises (conditionals, . . .)
- Consider `c.g := null`
- No strong update possible here (which fact to delete?)
- Need weak updates even when flow-sensitive

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