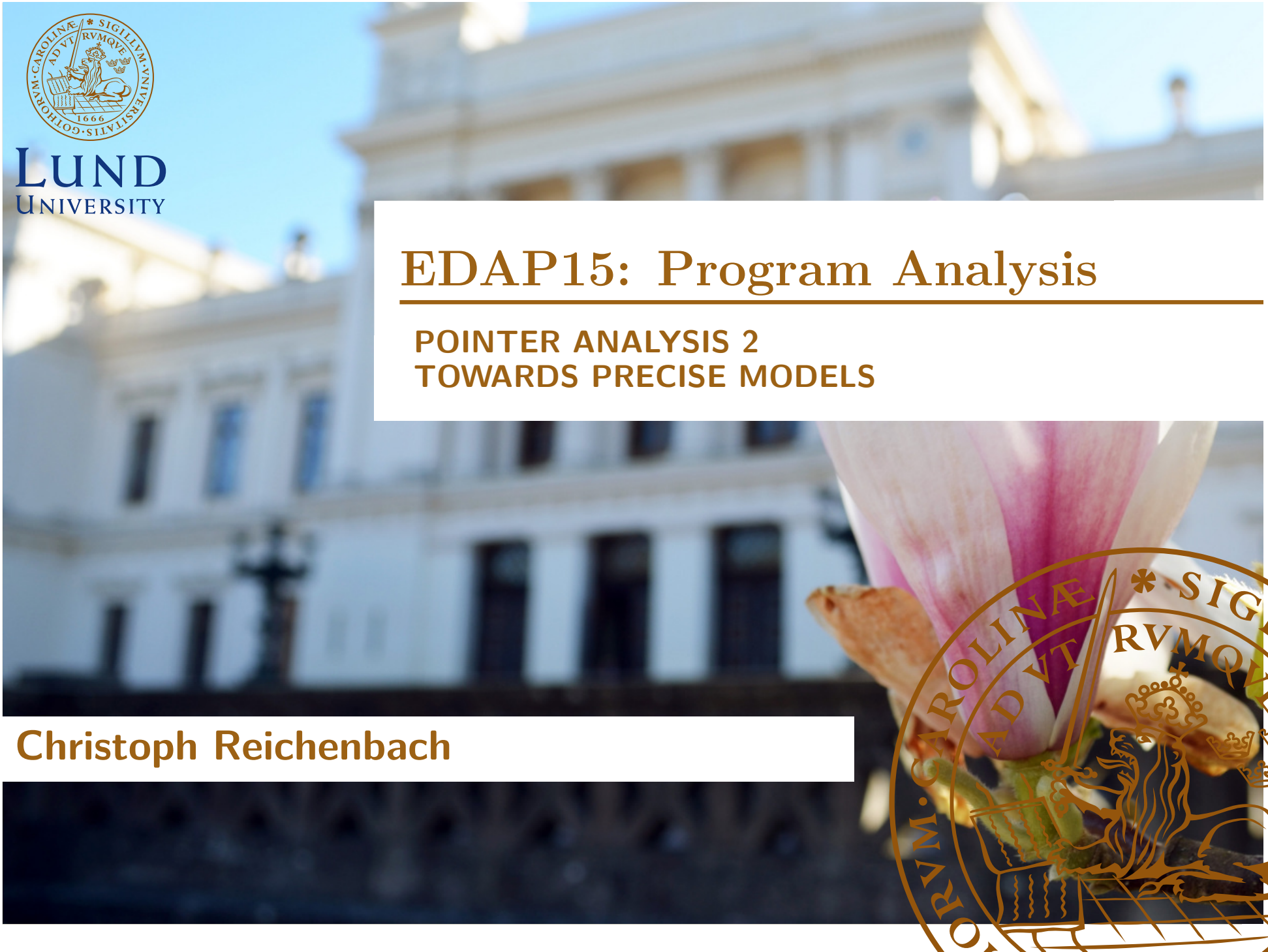


LUND UNIVERSITY

EDAP15: Program Analysis

POINTER ANALYSIS 2
TOWARDS PRECISE MODELS

Christoph Reichenbach

Announcements

- Relaxed homework deadlines
  - Still limited slots: guaranteed one slot per week
- HW2 out now
- Office hours tomorrow as usual

Pointer Operations

Referencing

Create, point to location:

Teal-2

```
a = new A();
a = [...];
```

Pointer Operations

Referencing

Create, point to location:

Teal-2

```
a = new A();
a = [...];
```

Dereferencing

Access location:

Teal-2

```
     - read -
...   := a.f;
...   := a[i];
     - write -
a.f := ...;
a[i] := ...;
```

Pointer Operations

Referencing

Create, point to location:

Teal-2

```
a = new A();
a = [...];
```

Dereferencing

Access location:

Teal-2

```
    - read -
...  := a.f;
...  := a[i];
    - write -
a.f := ...;
a[i] := ...;
```

Aliasing

Copy pointer:

Teal-2

```
a := b;
```

Principal Pointer Operations

- **Referencing**:
  - $v :=$ *memory-location-of*...
    - Fresh ℓ (**new**, **malloc** etc.)
    - In C/C++: location of variable (`&w`)
  - Introduce $v \rightarrow \ell$
- **Dereferencing**:
  - $x := v.f$
  - Access existing location ℓ
- **Aliasing**:
  - Pointer/reference variables v_1, v_2:
  - $v_2 := v_1$
  - $v_1 \rightarrow \ell \implies v_2 \rightarrow \ell$

Summary

- Points-to anaysis: *approximate* 'v points to location ℓ'

$$v \rightarrow \ell$$

- Analysis must consider:
  - **Referencing**: taking (fresh) location
    - In languages like C/C++, code can also reference locations of stack/global variables
  - **Dereferencing**: accessing object at location
  - **Aliasing**: copying location
- Locations ℓ may model different parts of memory:
  - Static variables: uniquely defined
  - Stack-dynamic variables: zero or more copies (recursion!)
  - Heap-dynamic variables: zero or more copies without variable names attached

Steensgaard's Points-To Analysis

- Fast: $O(n\alpha(n,n))$ over variables in program
- Developed to deal with large code bases at AT&T
- Sacrifices Precision
- *Equality-based*
- Intuition:
  Whenever two variables could point to the same memory location, treat them as globally equal

B. Steensgaard. 'Points-to analysis in almost linear time.' In Proceecings of POPL '96, pages 32–41. ACM Press, 1996.

Steensgard: Pointer Operations

Steensgard's analysis considers four cases:

	C	Java	Teal
Referencing	`a = &b`	`a = new A()`	`a := new A()`
Aliasing	`a = b`	`a = b`	`a := b`
Dereferencing read	`a = *b`	`a = b.f` `...`	`a := b.f` `b := a[i]`
Dereferencing write	`*a = b`	`a.f = b` `...`	`a.f := b` `a[i] := b`

- Teal:
  `a := [..., b, ...]`

Steensgard: Pointer Operations

Steensgard's analysis considers four cases:

	C	Java	Teal
Referencing	`a = &b`	`a = new A()`	`a := new A()`
Aliasing	`a = b`	`a = b`	`a := b`
Dereferencing read	`a = *b`	`a = b.f` `...`	`a := b.f` `b := a[i]`
Dereferencing write	`*a = b`	`a.f = b` `...`	`a.f := b` `a[i] := b`

- Teal:
  `a := [..., b, ...]`
- **Referencing** and also **Dereferencing Write**
  `a := new array[any](`*n*`);`
  `a[`*i*`] := b;`

Distinguishing Field Names?

- For simplicity, don't distinguish field names:
- `a.`$\square$ instead of `a.f` or `a.g`

Constraint Collection

- 'Points-to-set': $pts(v)$ approximates $\{\ell | v \rightarrow \ell\}$
  - Corresponds to $\{\ell | v \rightarrow \ell\}$
- For each statement in program:
  - If **Referencing** (`a := new` $\ldots \ell_b$):

$$\ell_{\mathtt{b}} \in pts(\mathtt{a})$$

Constraint Collection

- 'Points-to-set': $pts(v)$ approximates $\{\ell | v \rightarrow \ell\}$
  - Corresponds to $\{\ell | v \rightarrow \ell\}$
- For each statement in program:
  - If **Referencing** (`a := new` $\ldots \ell_b$):

$$\ell_{\mathtt{b}} \in pts(\mathtt{a})$$

  - If **Aliasing** (`a := b`):

$$pts(\mathtt{a}) = pts(\mathtt{b})$$

Constraint Collection

- 'Points-to-set': $pts(v)$ approximates $\{\ell | v \rightarrow \ell\}$
  - Corresponds to $\{\ell | v \rightarrow \ell\}$
- For each statement in program:
  - If **Referencing** (`a := new` $\ldots \ell_b$):

$$\ell_{\mathtt{b}} \in pts(\mathtt{a})$$

  - If **Aliasing** (`a := b`):

$$pts(\mathtt{a}) = pts(\mathtt{b})$$

  - If **Dereferencing read** (`a := b.`$\square$):

$$\text{for each } \ell \in pts(\mathtt{b}) \implies pts(\mathtt{a}) = pts(\ell)$$

Constraint Collection

- 'Points-to-set': $pts(v)$ approximates $\{\ell | v \rightarrow \ell\}$
  - Corresponds to $\{\ell | v \rightarrow \ell\}$
- For each statement in program:
  - If **Referencing** (`a := new` $\ldots \ell_b$):

$$\ell_{\mathtt{b}} \in pts(\mathtt{a})$$

  - If **Aliasing** (`a := b`):

$$pts(\mathtt{a}) = pts(\mathtt{b})$$

  - If **Dereferencing read** (`a := b.`$\square$):

$$\text{for each } \ell \in pts(\mathtt{b}) \implies pts(\mathtt{a}) = pts(\ell)$$

  - If **Dereferencing write** (`a.`$\square$ `:= b`):

$$\text{for each } \ell \in pts(\mathtt{a}) \implies pts(\mathtt{b}) = pts(\ell)$$

Example

`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

- **Actual**:

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;
```

- **Steensgaard**:

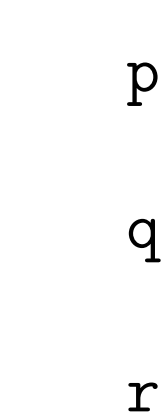
Example

`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

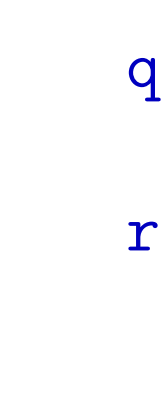
Teal

```
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**:



- **Steensgaard**:



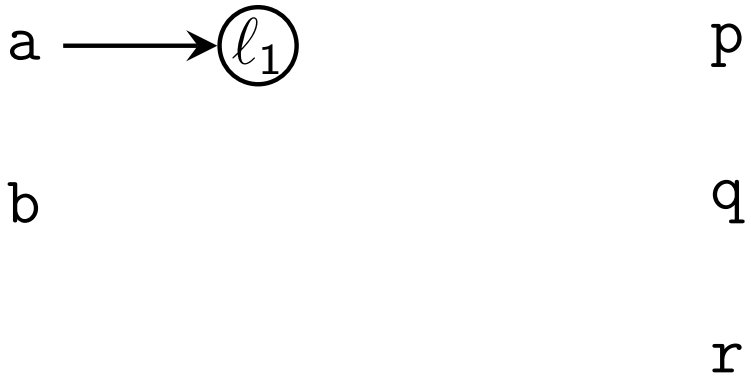
Example

$\Rightarrow$ `x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

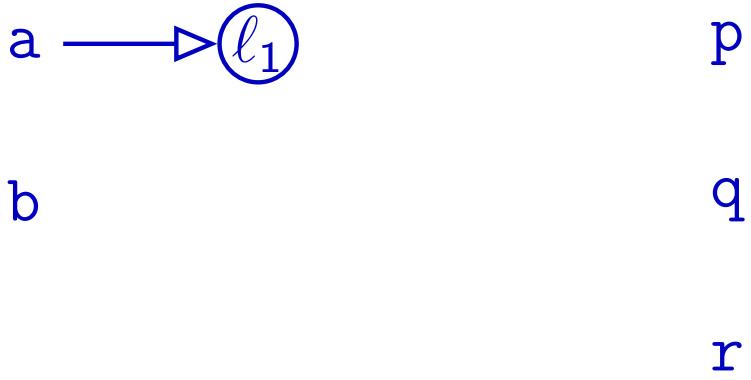
Teal

```
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**:



- **Steensgaard**:



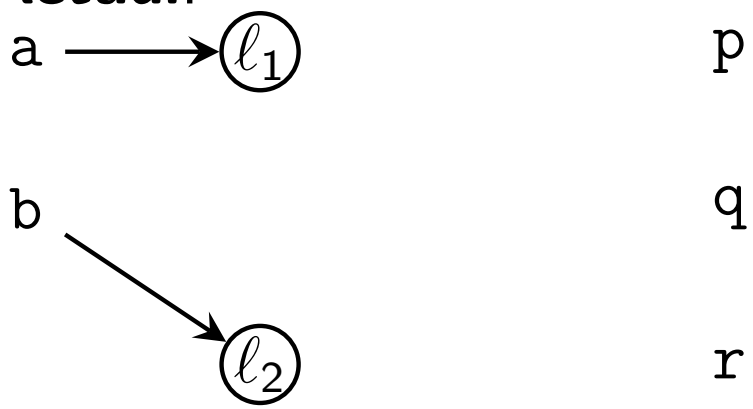
Example

$\Rightarrow$ `x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

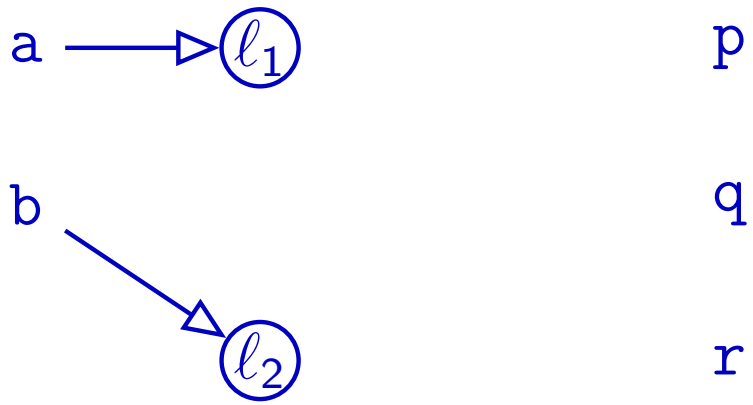
Teal

```
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**:



- **Steensgaard**:



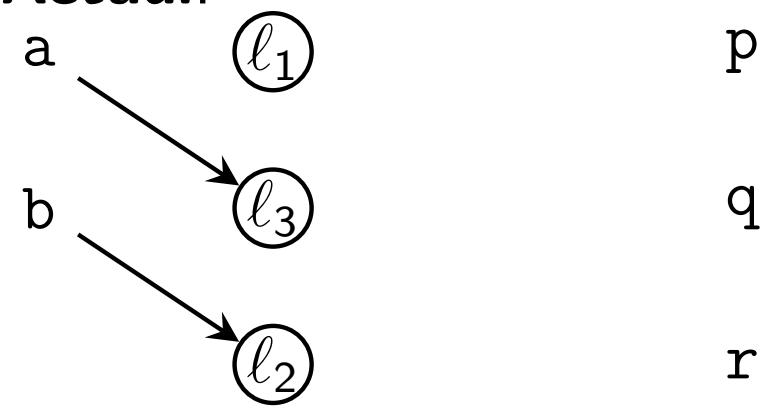
Example

$\Rightarrow$ `x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

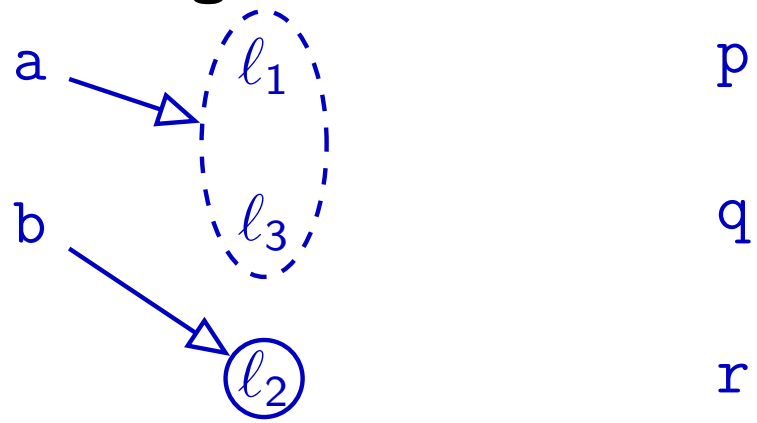
Teal

```
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**:



- **Steensgaard**:



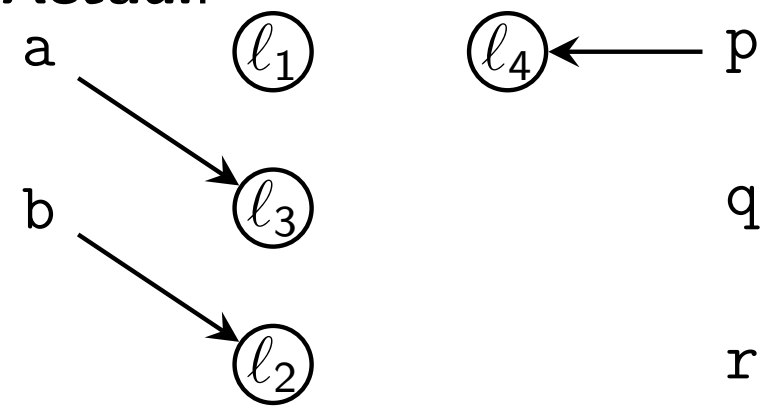
Example

⇒ `x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

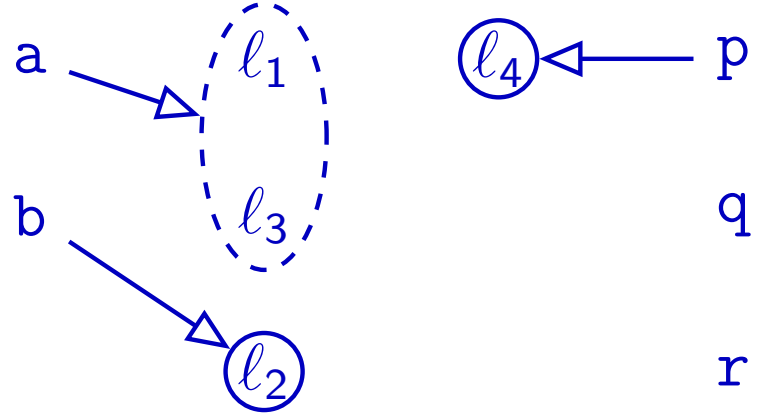
Teal

```
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**:



- **Steensgaard**:



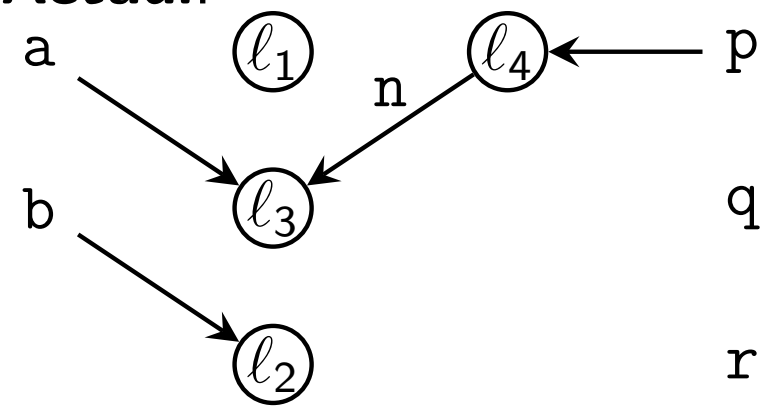
Example

`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
$\Rightarrow$ `x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

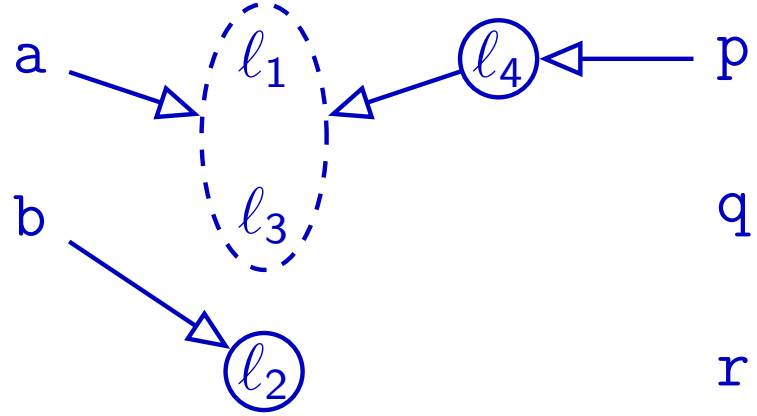
Teal

```
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**:



- **Steensgaard**:



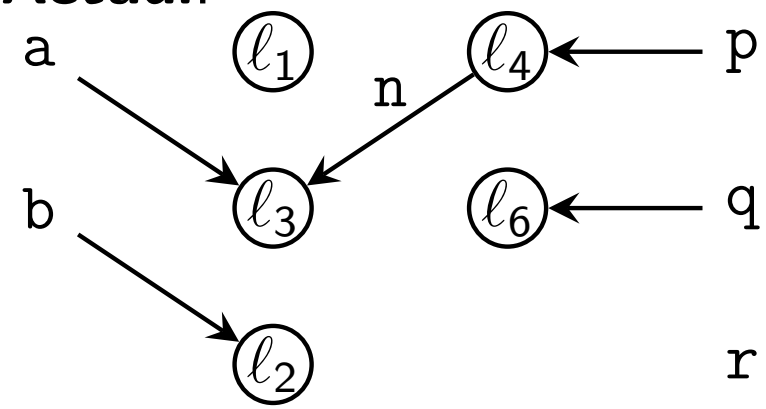
Example

$\Rightarrow$ `x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

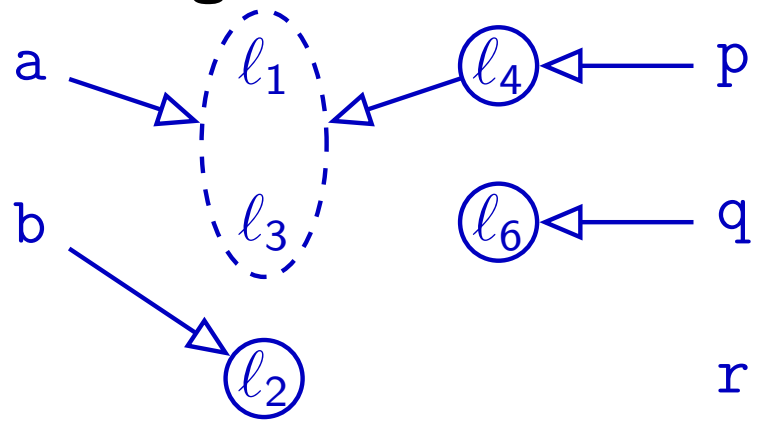
Teal

```
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**:



- **Steensgaard**:



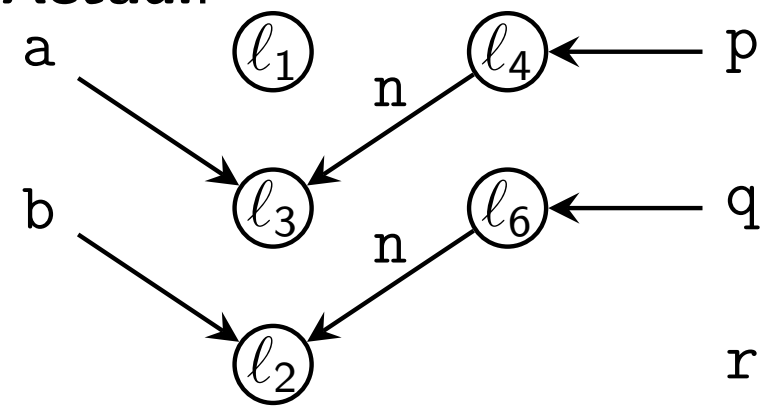
Example

`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
$\Rightarrow$ `x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

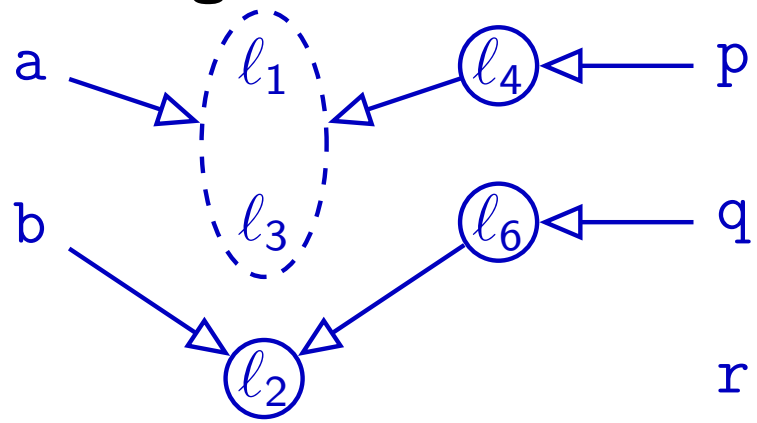
Teal

```
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**:



- **Steensgaard**:



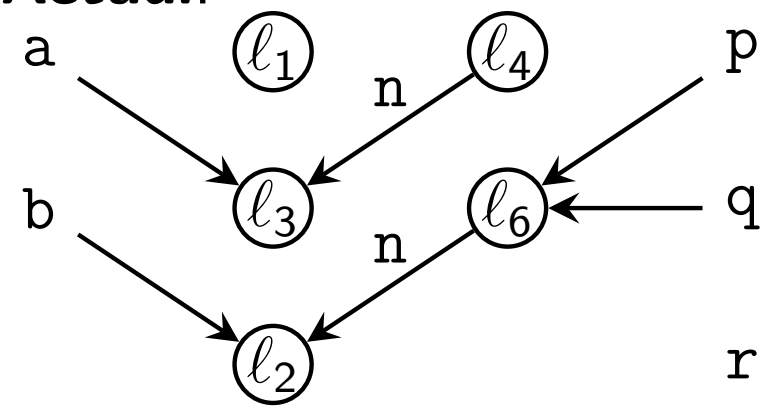
Example

	`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
$\Rightarrow$	`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
	`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
	`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

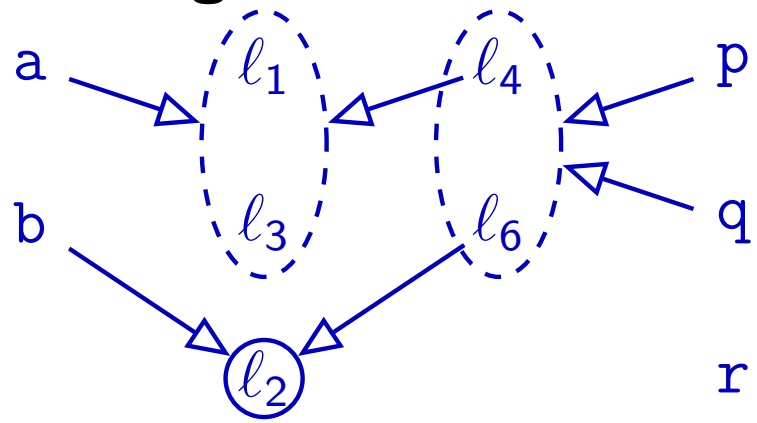
Teal

```
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**:



- **Steensgaard**:



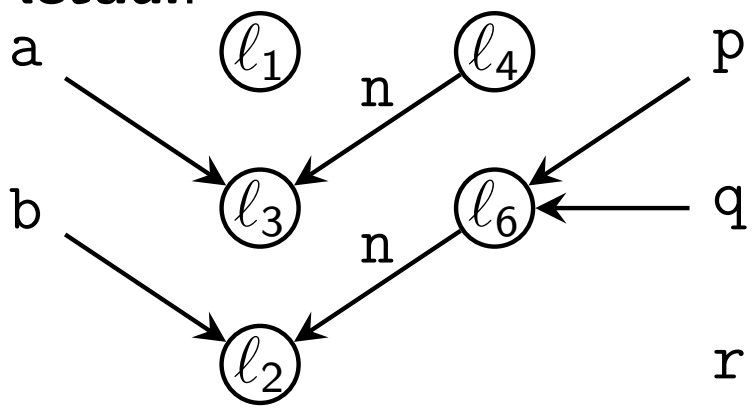
Example

	`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
$\Rightarrow$	`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
	`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
	`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

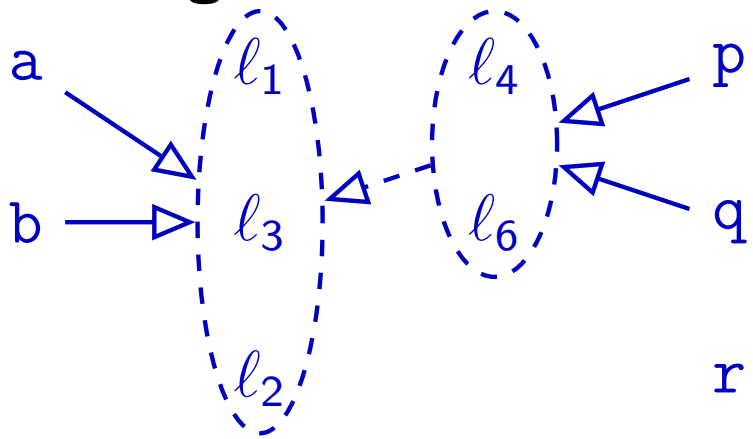
Teal

```
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**:



- **Steensgaard**:



When merging: 'collapse' children (merge recursively)

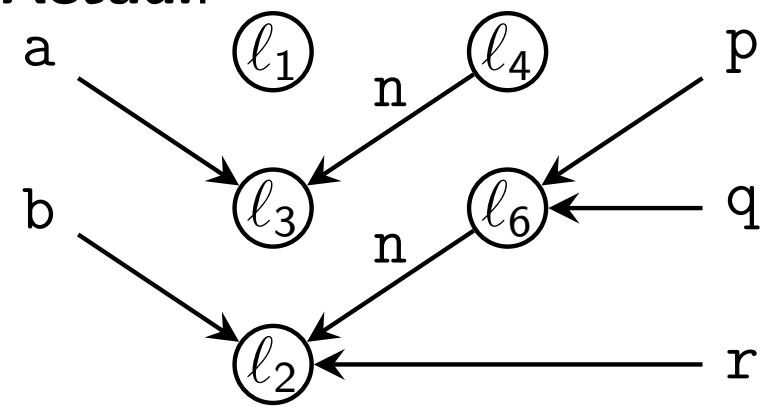
Example

	`x := new`$_{\ell_z}$	$\ell_z \in pts(\mathtt{x})$
	`x := y`	$pts(\mathtt{x}) = pts(\mathtt{y})$
$\Rightarrow$	`x := y.f`	for each $\ell \in pts(\mathtt{y})$ $\implies pts(\mathtt{x}) = pts(\ell)$
	`x.f := y`	for each $\ell \in pts(\mathtt{x})$ $\implies pts(\mathtt{y}) = pts(\ell)$

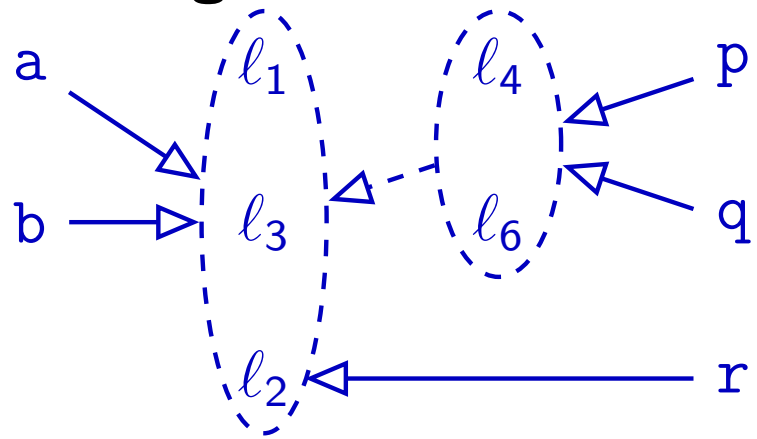
Teal

```
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**:



- **Steensgaard**:



When merging: 'collapse' children (merge recursively)

Summary

- Points-to sets $pts(v)$ serve as abstraction over addresses that v can point to
- Steensgaard's points-to analysis:

- Steensgaard's analysis in practice:
  - Highly efficient when implemented with UNION-FIND
  - Relatively imprecise

Summary

- Points-to sets $pts(v)$ serve as abstraction over addresses that v can point to
- Steensgaard's points-to analysis:
  - special case of *type analysis*
  - Needs some tweaking to distinguish e.g. reference field names
- Steensgaard's analysis in practice:
  - Highly efficient when implemented with UNION-FIND
  - Relatively imprecise

Alias Analysis in Practice (1/2)

Teal

```
var c := new_{ℓ0}();
var d := new_{ℓ1}();
```

Alias Analysis in Practice (1/2)

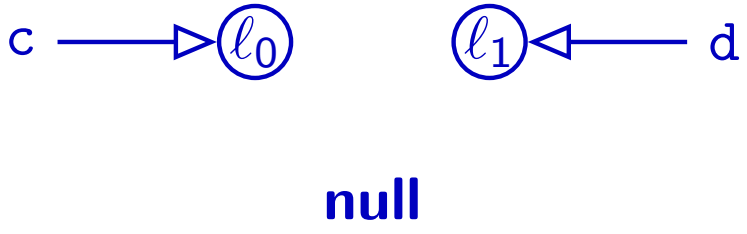
Teal

```
var c := new_{ℓ0}();
var d := new_{ℓ1}();
```

Alias Analysis in Practice (1/2)

Teal

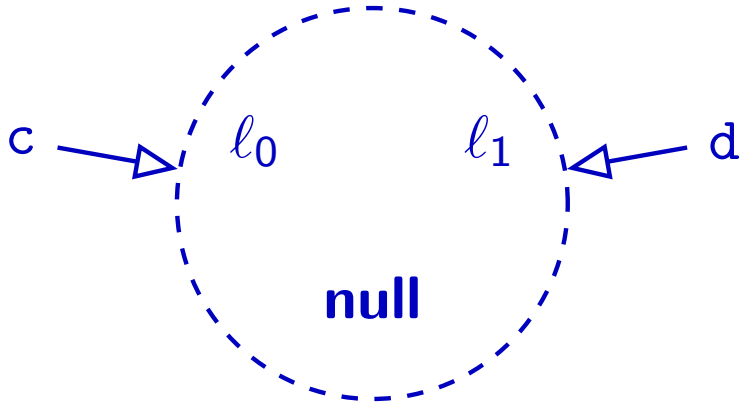
```
var c := new_{ℓ0}();
var d := new_{ℓ1}();
if ...  {
   c := null;
} else {
   d := null;
}
```



Alias Analysis in Practice (1/2)

Teal

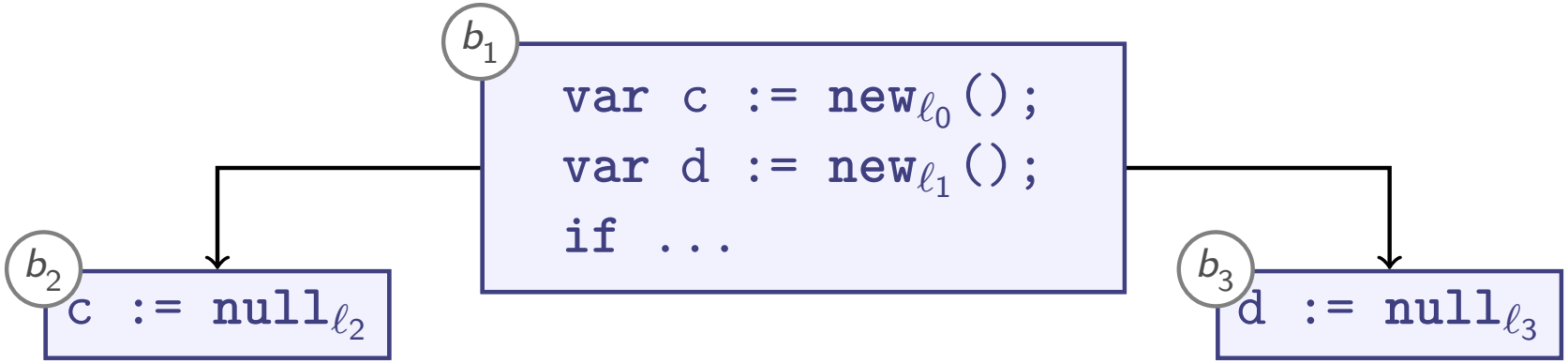
```
var c := new_{ℓ0}();
var d := new_{ℓ1}();
if ...  {
   c := null;
} else {
   d := null;
}
```



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

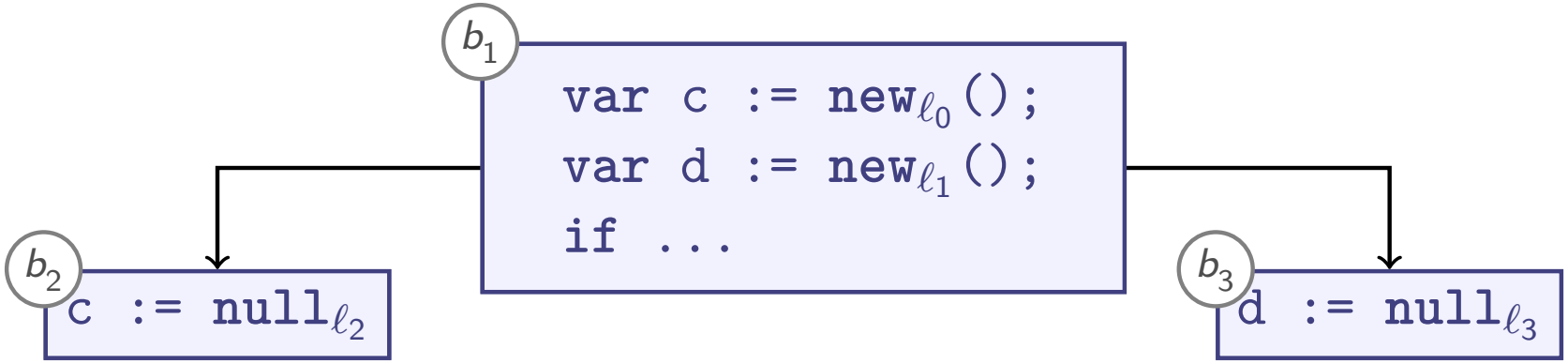
null as unique memory location: Imprecision!

Representing Null Pointers



1. One unique **null**

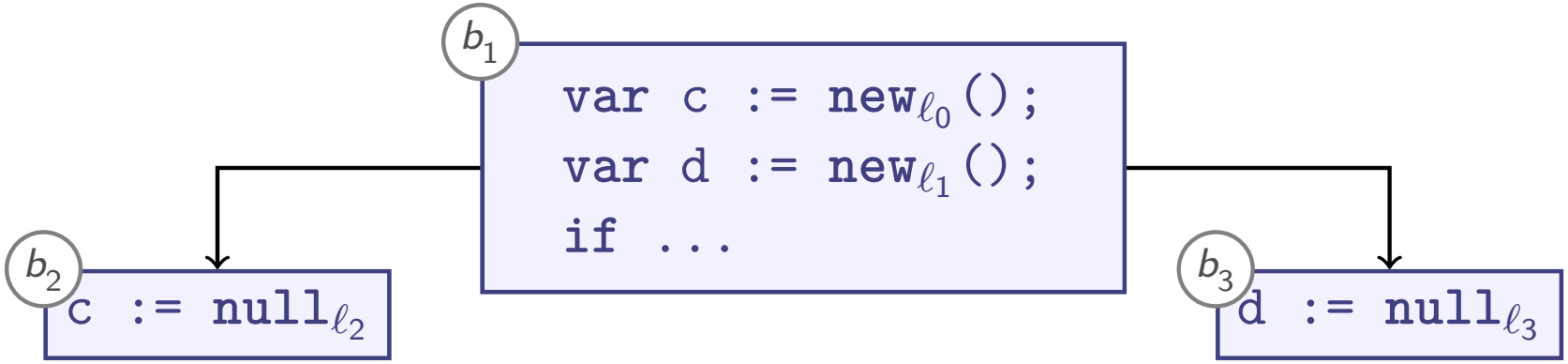
Representing Null Pointers



1. One unique **null**

2. Many **null**s

Representing Null Pointers



1. One unique **null**

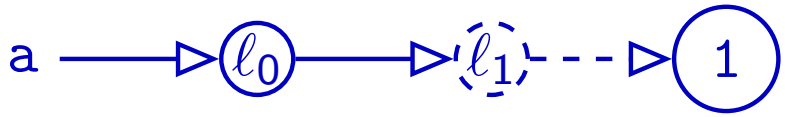
2. Many **nulls**

3. Nullness flags

Alias Analysis in Practice (2/2)

Teal

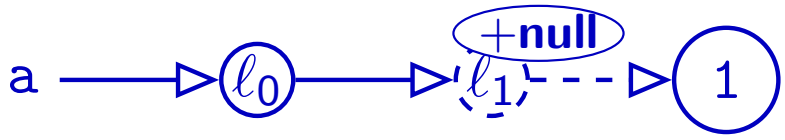
```
var a := new_{ℓ0} XY();
a.x := new_{ℓ1} XY();
a.x.x := 1;
```



Alias Analysis in Practice (2/2)

Teal

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

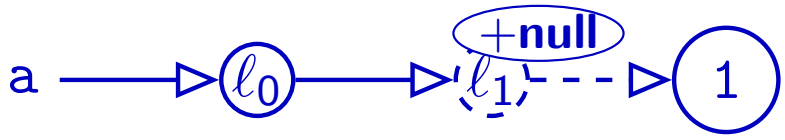


Alias Analysis in Practice (2/2)

Teal

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

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



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

Field Sensitivity

- By default, merge all fields:

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

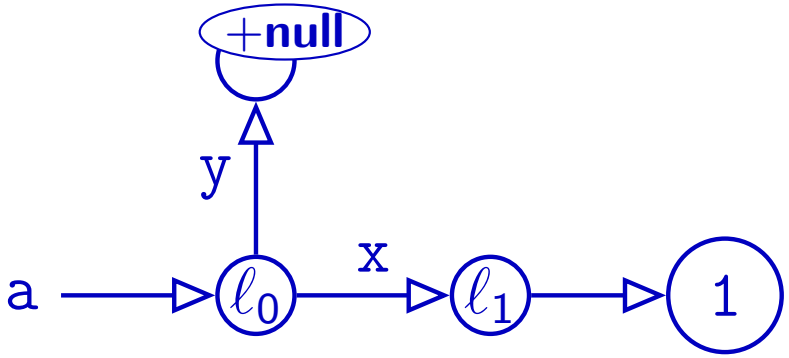
- Points-to analysis so far *field insensitive*
- Analogous for array indices

Field Sensitivity

- By default, merge all fields:

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

- 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**:

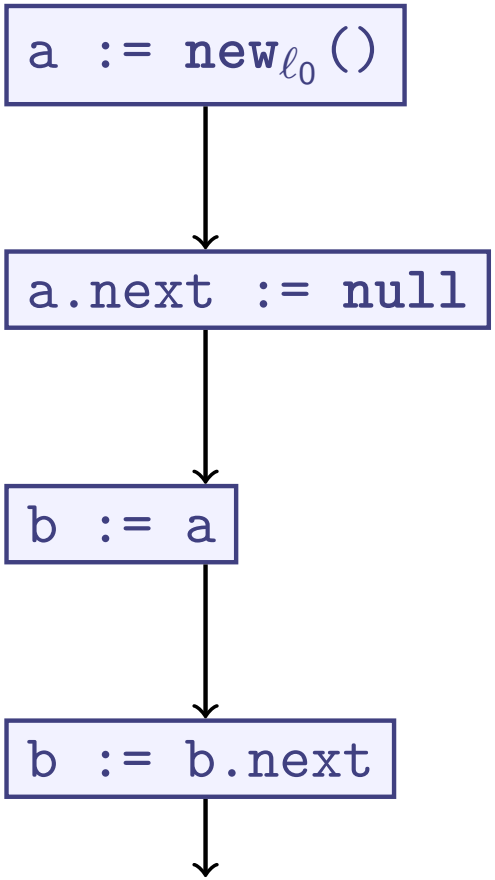
$$\texttt{a.x} \overset{alias}{=\!=} \texttt{a.}\square \overset{alias}{=\!=} \texttt{a.y}$$

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

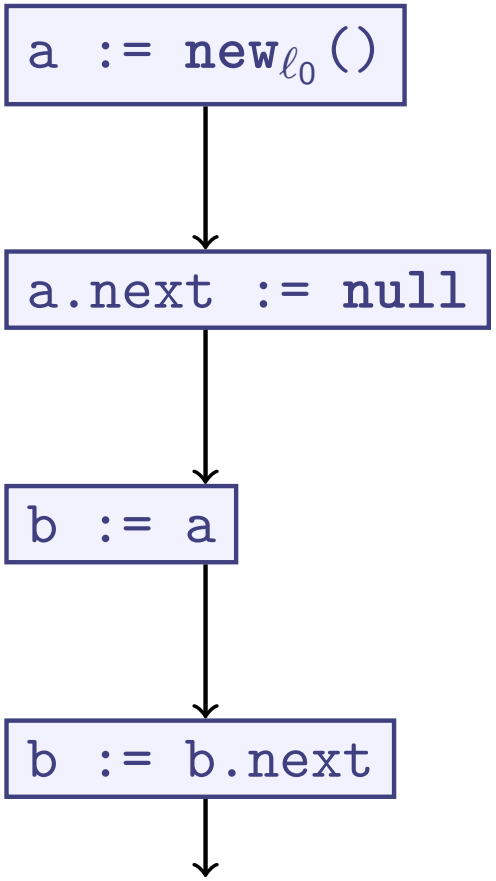
$$\texttt{a.x} \overset{alias}{\not=\!=} \texttt{a.y}$$

- Analogously for array indices

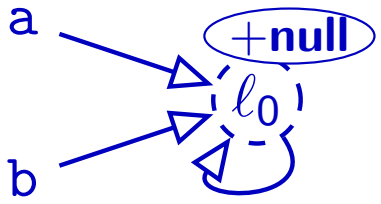
Flow-(In)Sensitive Points-To Analysis



Flow-(In)Sensitive Points-To Analysis

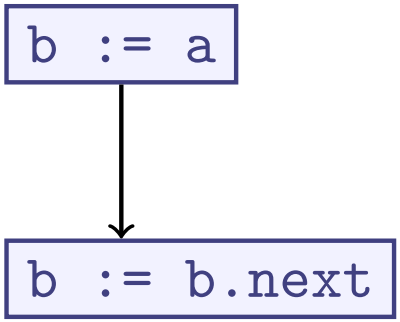


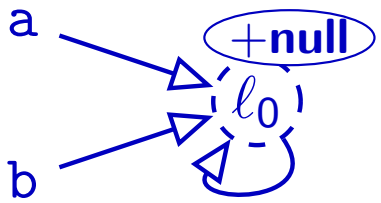
Flow Insensitive



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

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:

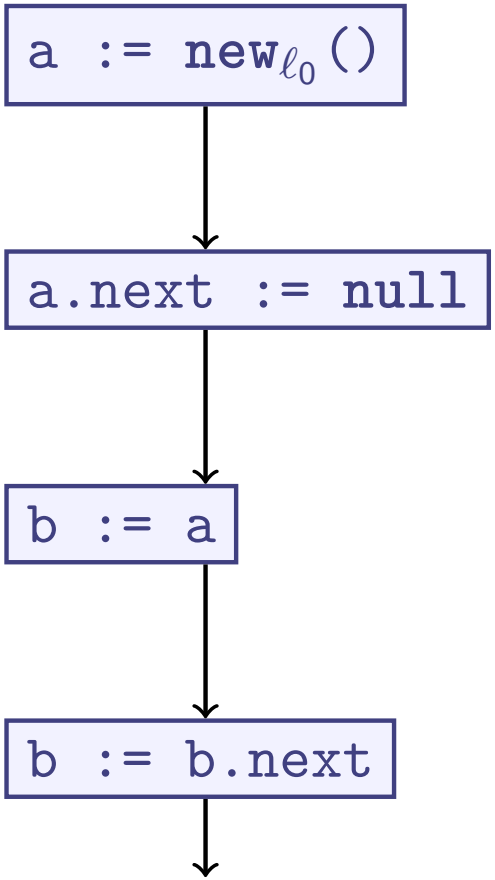
$$\boxed{\texttt{b := a}} \qquad \left[\ pts(\texttt{b}) \ \mapsto \ pts(\texttt{a}) \cup pts(\texttt{b})\ \right]$$

- *Weak Update*

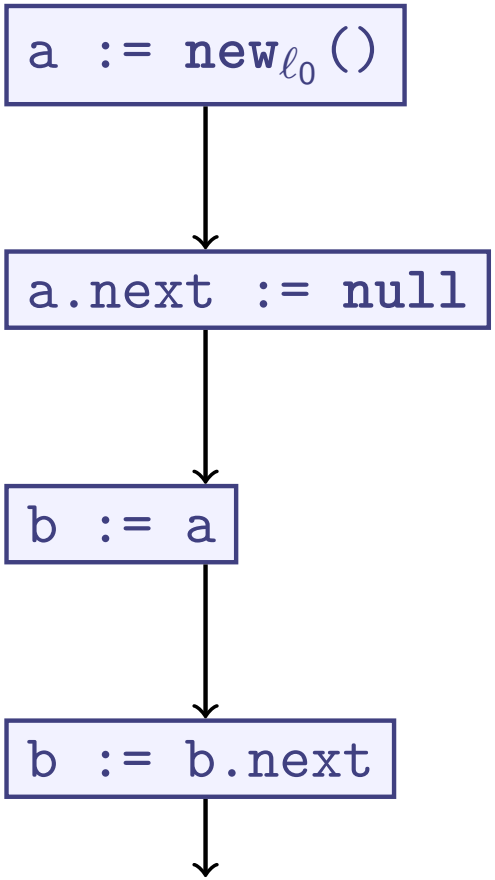
Points-To from Dataflow Analysis

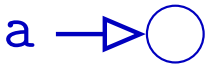
- Most (scalable) points-to analyses are flow insensitive
  $\Longrightarrow$ One global ($\overset{alias}{=\!=}$) relation
- *Flow-sensitive points-to analysis*:
  - Allows different Abstract Heap Graphs per basic block
  - Analogously ($\overset{alias}{=\!=}$) per basic block
  - Higher precision

Flow-(In)Sensitive Points-To Analysis

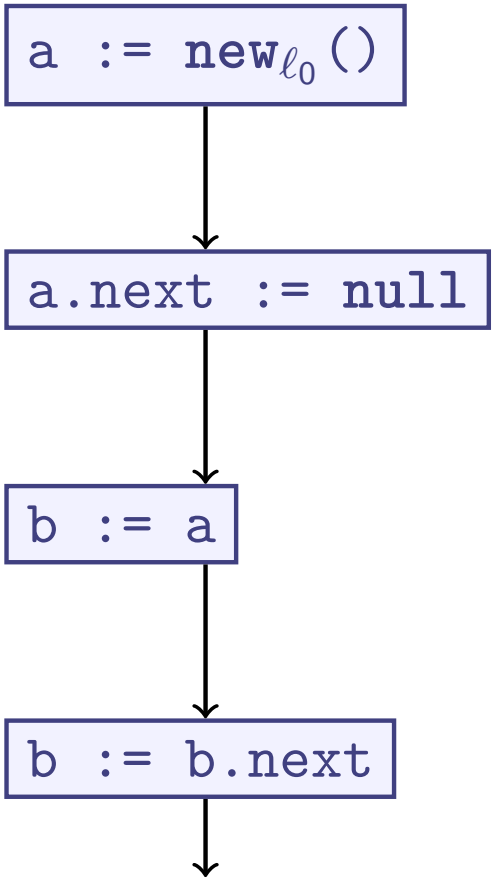


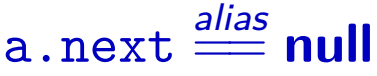
Flow-(In)Sensitive Points-To Analysis

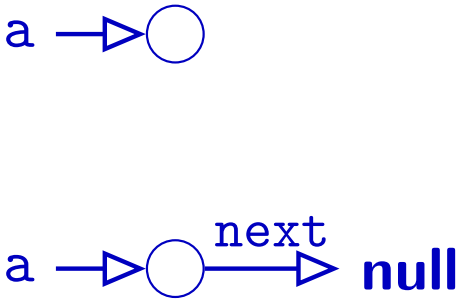




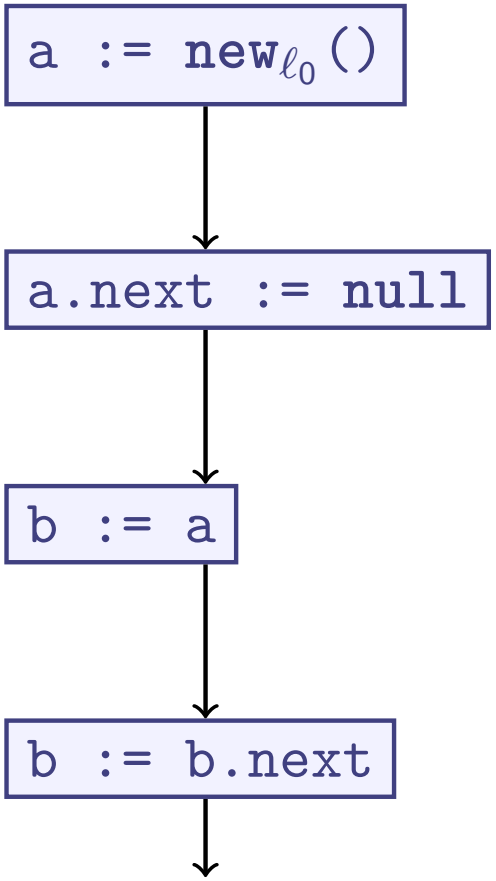
Flow-(In)Sensitive Points-To Analysis







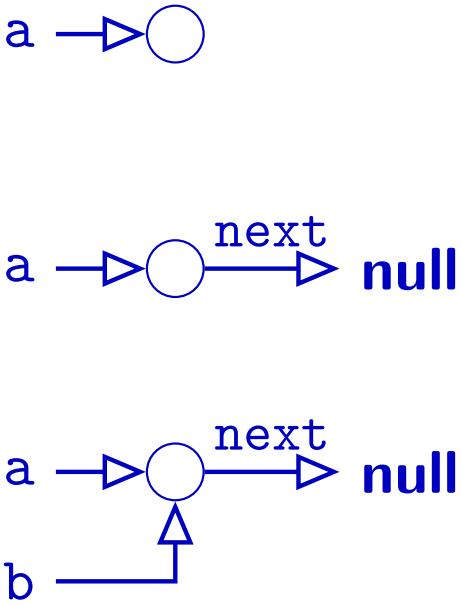
Flow-(In)Sensitive Points-To Analysis



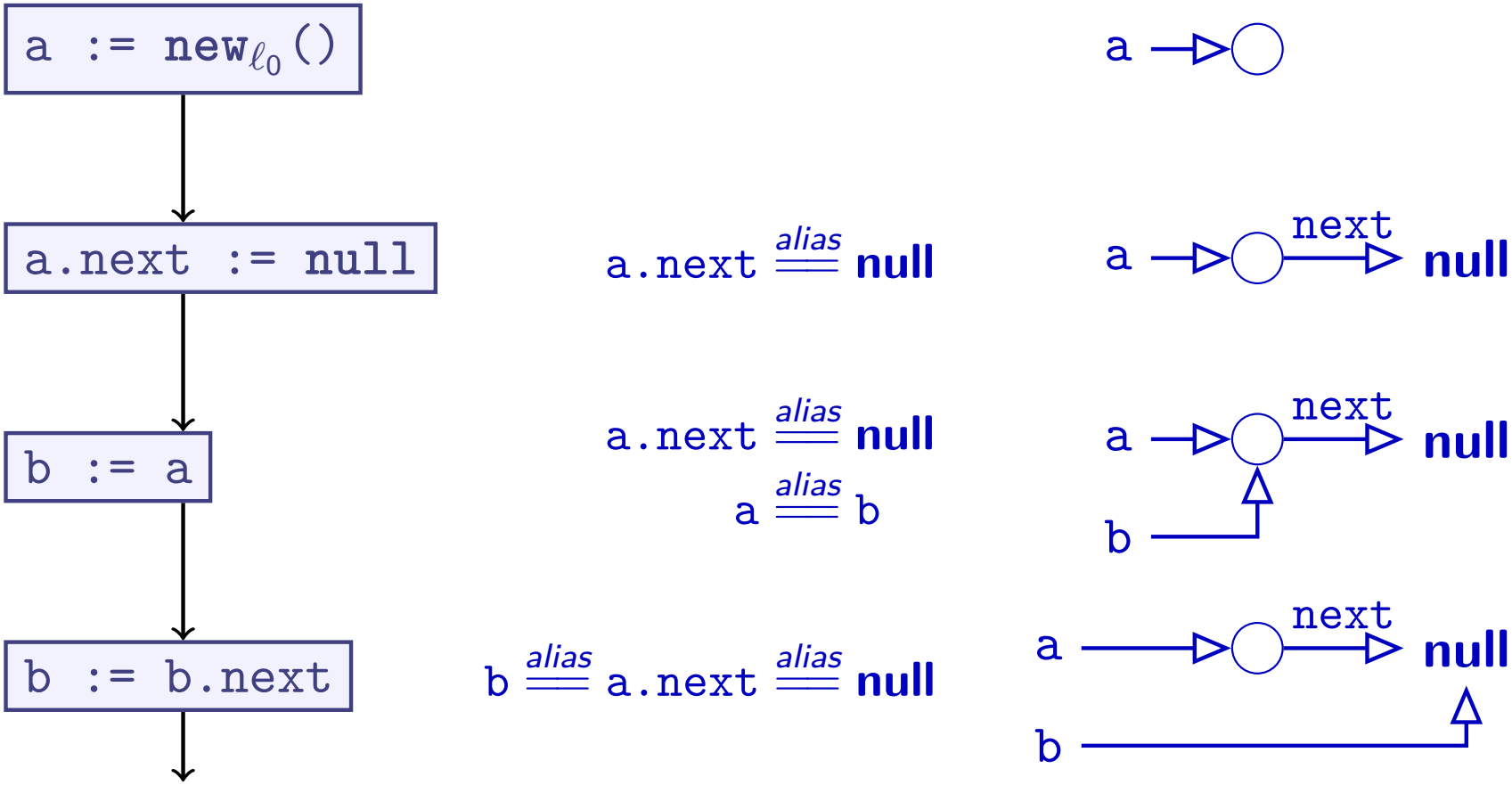
$\texttt{a.next} \overset{alias}{=\!=} \textbf{null}$

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

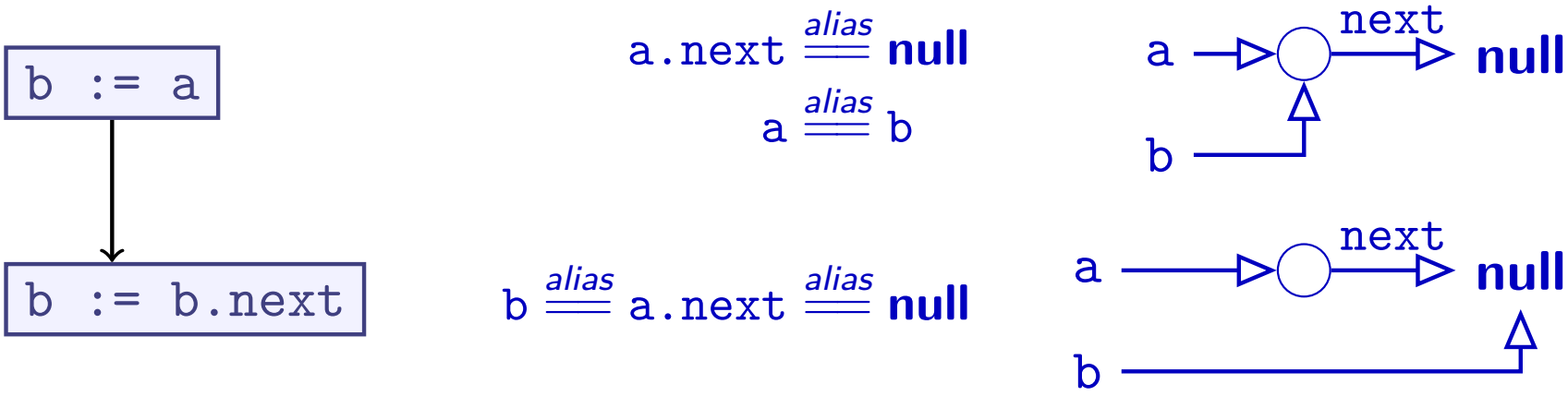
$\texttt{a} \overset{alias}{=\!=} \texttt{b}$



Flow-(In)Sensitive Points-To Analysis

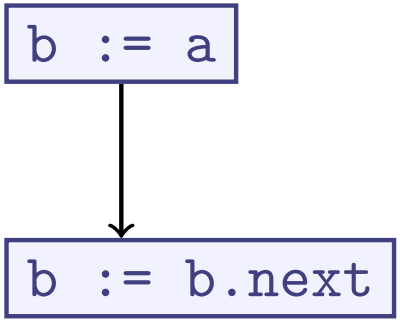


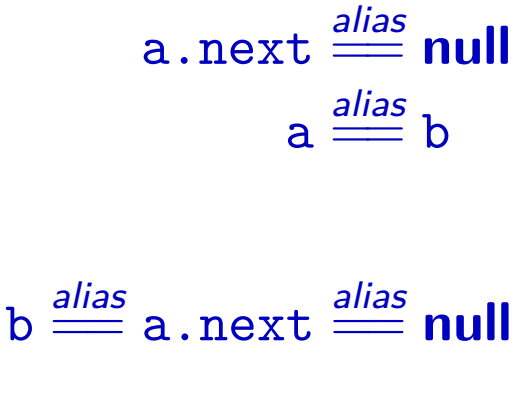
Strong and Weak Updates

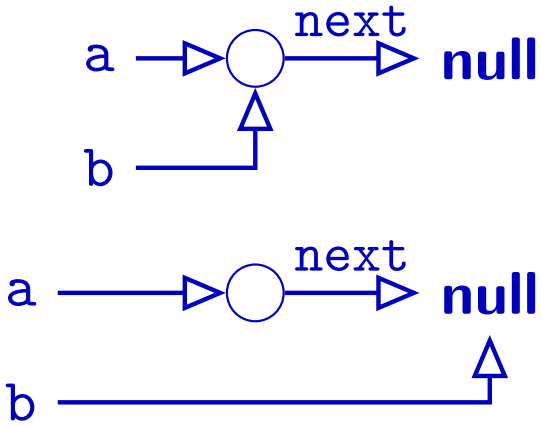


- Flow-sensitive points-to analysis enables *strong updates*:
  - Remove information that is overwritten by update

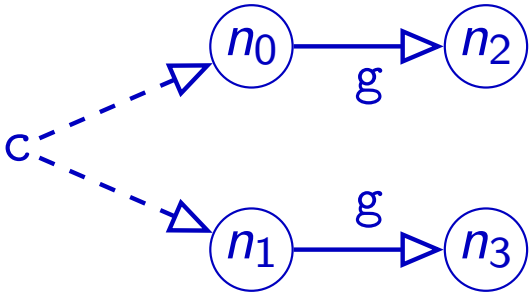
Strong and Weak Updates





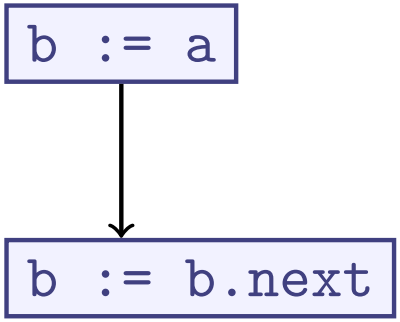


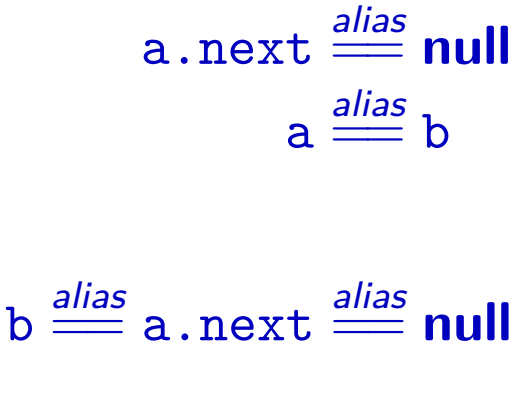
- Flow-sensitive points-to analysis enables *strong updates*:
  - Remove information that is overwritten by update

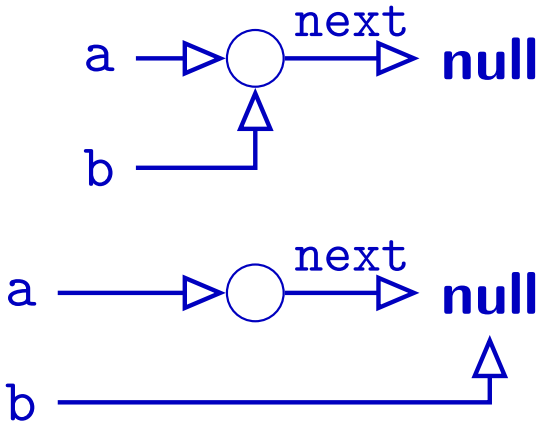


  - Imprecision still arises (conditionals, . . .)

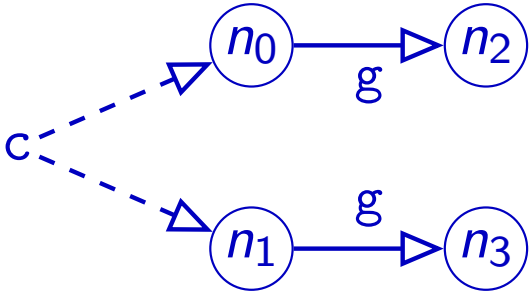
Strong and Weak Updates





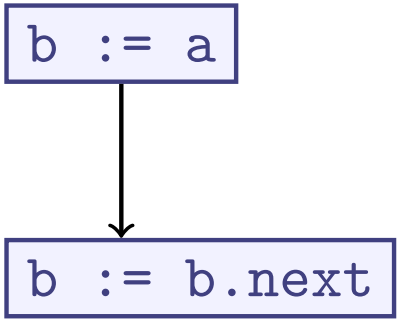


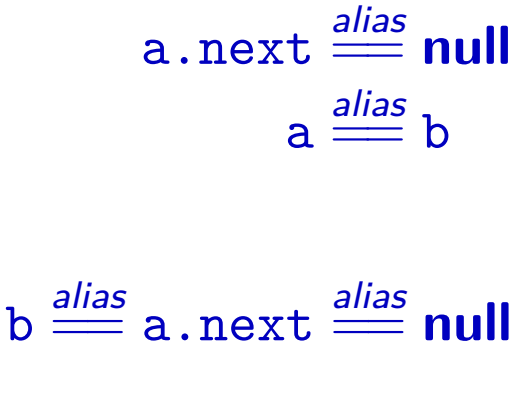
- Flow-sensitive points-to analysis enables *strong updates*:
  - Remove information that is overwritten by update

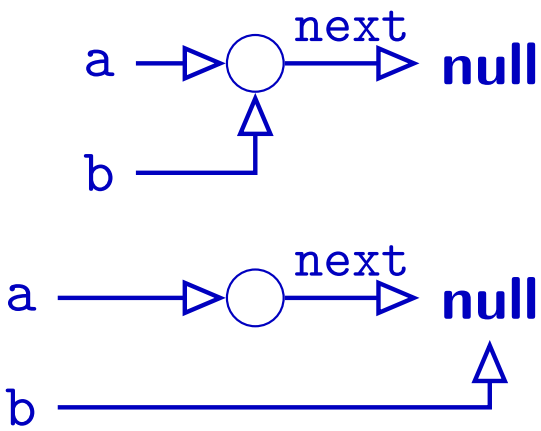


- Imprecision still arises (conditionals, . . .)
- Consider `c.g := null`

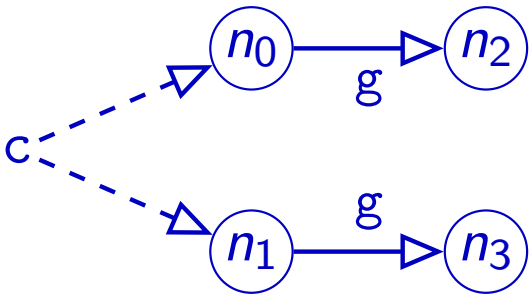
Strong and Weak Updates







- 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