Lecture 7 – Pointer Analysis

SWS128: Software Analysis

Jihyeok Park



2026 Fall

Recall – Taint

Run `f("' OR 1=1")`:

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]     # sink
```

uid `"' OR 1=1"`

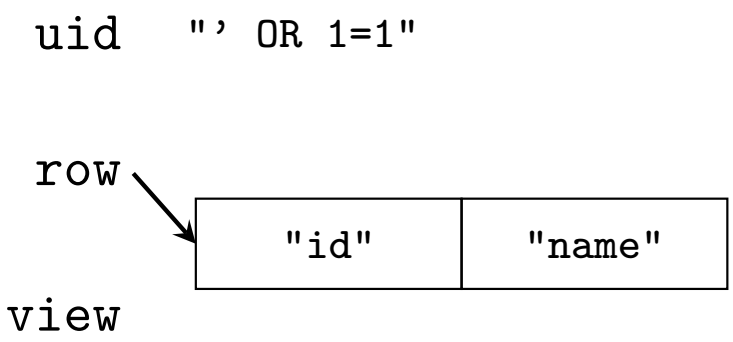
row

view

Recall – Taint

Run `f("' OR 1=1")`:

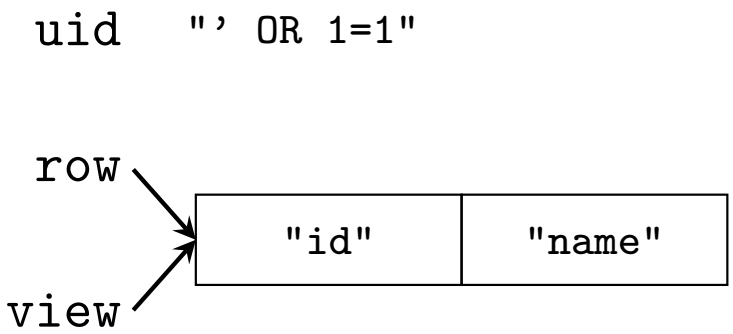
```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]     # sink
```



Recall – Taint

Run `f("' OR 1=1")`:

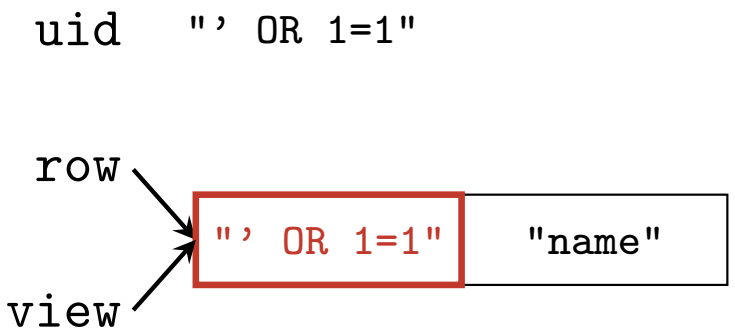
```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]     # sink
```



Recall – Taint

Run `f("' OR 1=1")`:

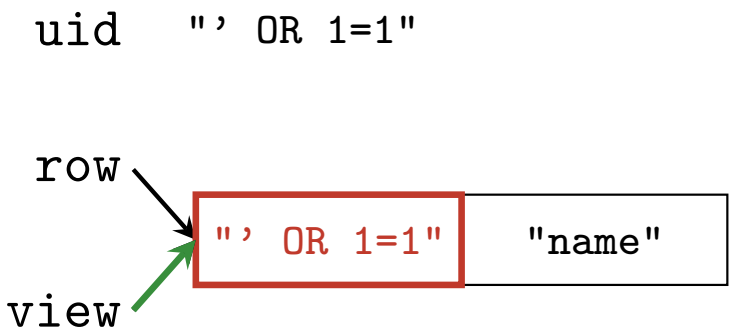
```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]      # sink
```



Recall – Taint

Run `f("' OR 1=1")`:

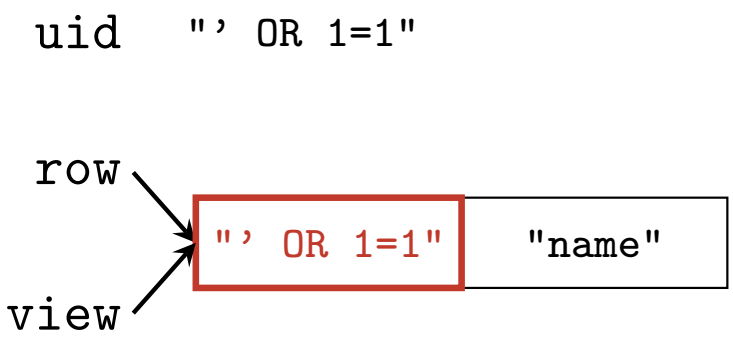
```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]     # sink
```



Recall – Taint

Run `f("' OR 1=1")`:

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]     # sink
```

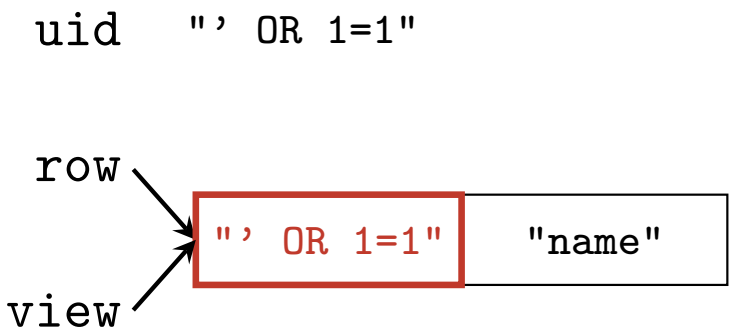


- `view[0]` is never written, so our taint analysis says the returned value is **clean**. The run returns the attacker's string.

Recall – Taint

Run `f("' OR 1=1")`:

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]     # sink
```



- `view[0]` is never written, so our taint analysis says the returned value is **clean**. The run returns the attacker's string.
- `row` and `view` are **two names for one list**. A **false negative**, and a silent one.

Recall – Available Expressions

Run `f(5)`:

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```

```
 v   5

xs

ys

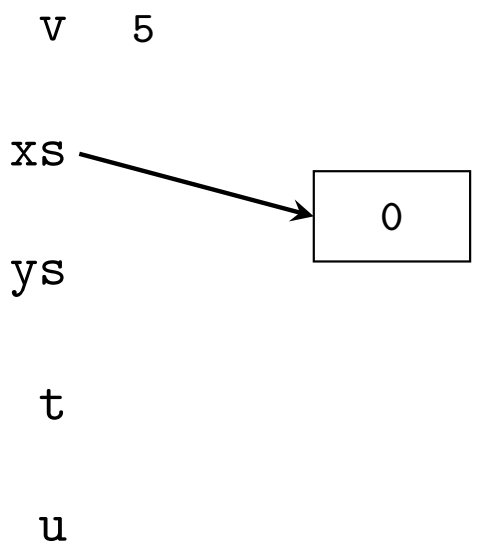
 t

 u
```

Recall – Available Expressions

Run `f(5)`:

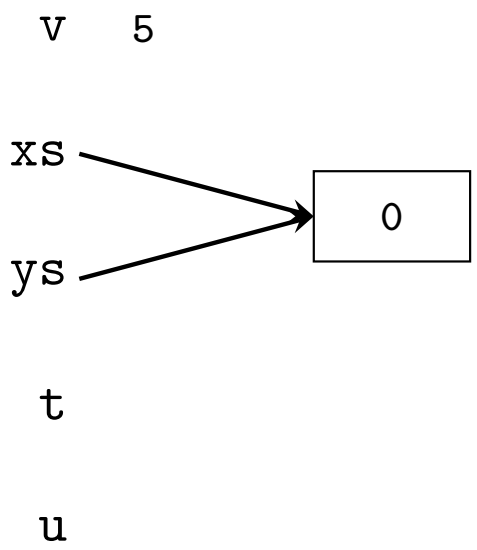
```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



Recall – Available Expressions

Run `f(5)`:

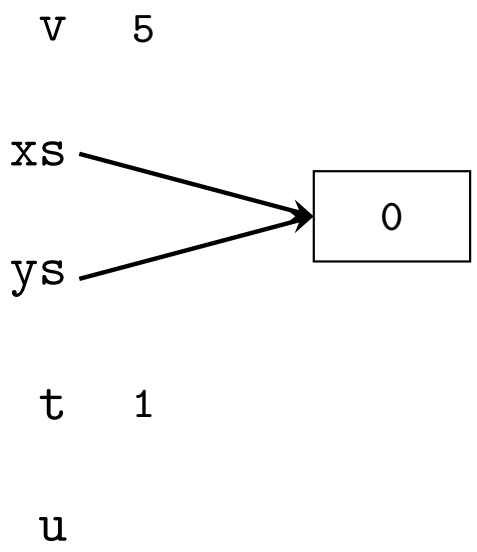
```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



Recall – Available Expressions

Run `f(5)`:

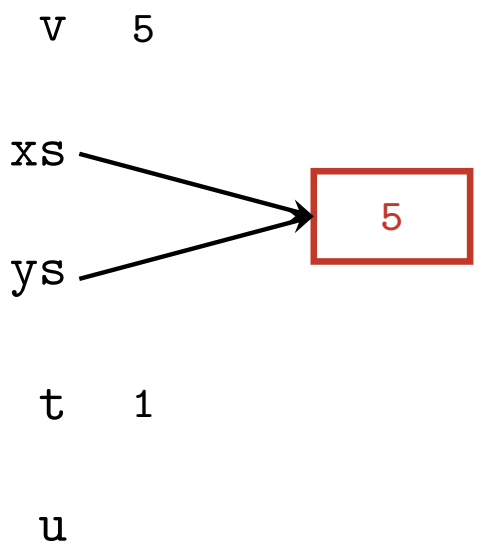
```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



Recall – Available Expressions

Run `f(5)`:

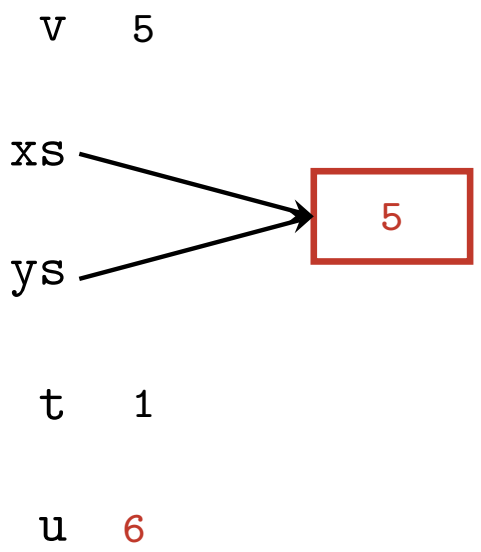
```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



Recall – Available Expressions

Run `f(5)`:

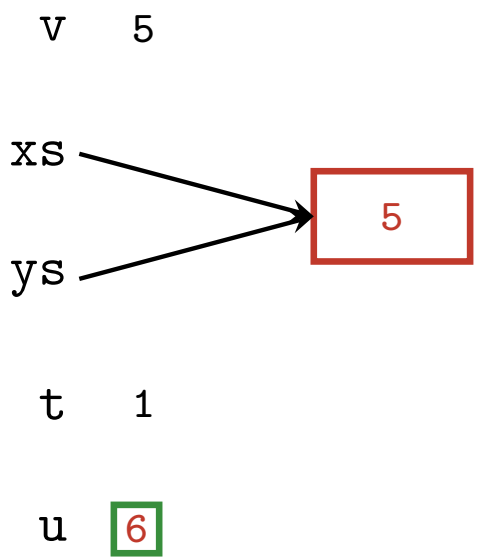
```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



Recall – Available Expressions

Run `f(5)`:

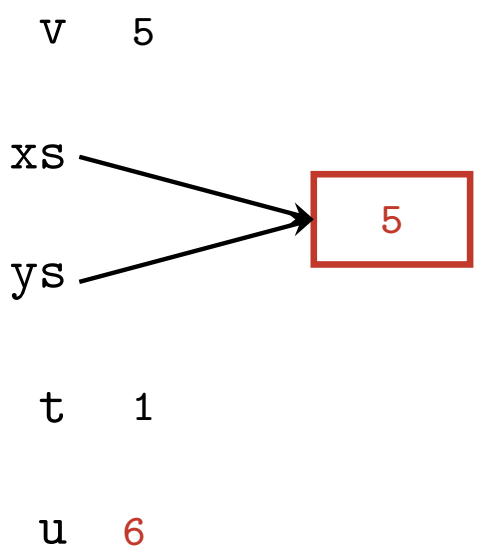
```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



Recall – Available Expressions

Run `f(5)`:

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```

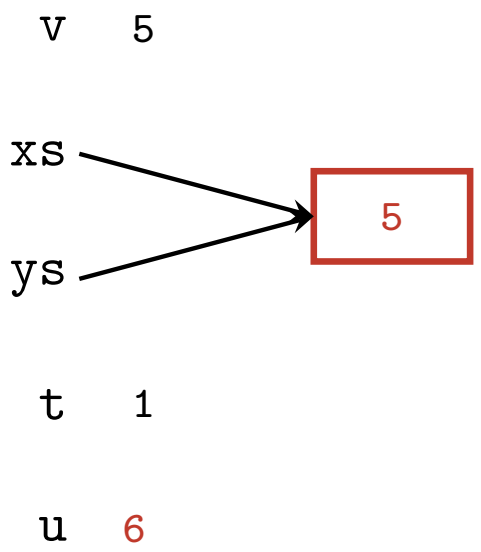


- Available expressions says `xs[0] + 1` is still **available**: `ys[0] = v` defines no variable, so it kills nothing. Reusing `t` returns 1, not 6.

Recall – Available Expressions

Run `f(5)`:

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1     # reuse t?
    return u
```



- Available expressions says `xs[0] + 1` is still **available**: `ys[0] = v` defines no variable, so it kills nothing. Reusing `t` returns 1, not 6.
- Killing what reads `ys[...]` is not enough either: `xs[0]` changed too. Both analyses need to know *which lists a name can refer to*.

Contents

1. Points-To

2. Andersen's Analysis

3. Precision

4. Graph Reachability

5. In Python

Contents

1. Points-To

2. Andersen's Analysis

3. Precision

4. Graph Reachability

5. In Python

Points-To Sets

Run `f("x")`:

```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```

```
 uid   "x"

rows

view

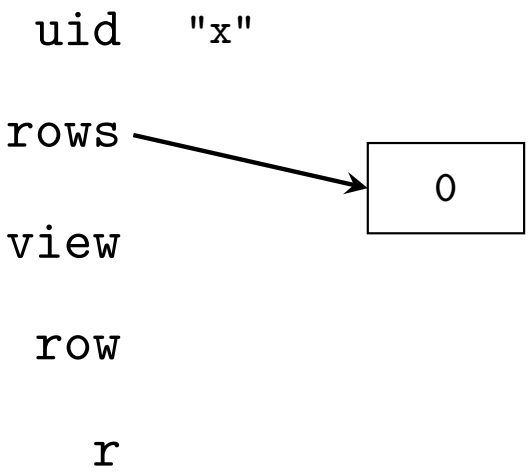
 row

   r
```

Points-To Sets

Run `f("x")`:

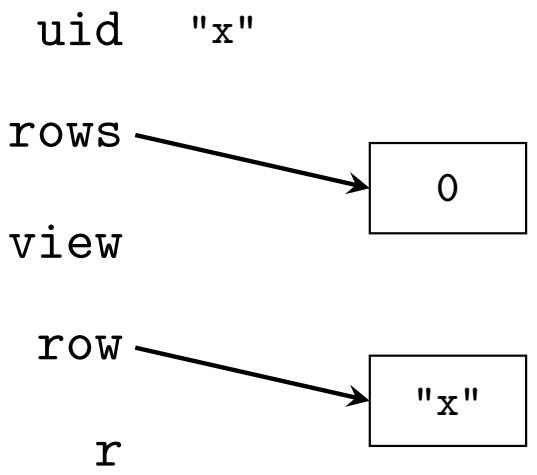
```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



Points-To Sets

Run `f("x")`:

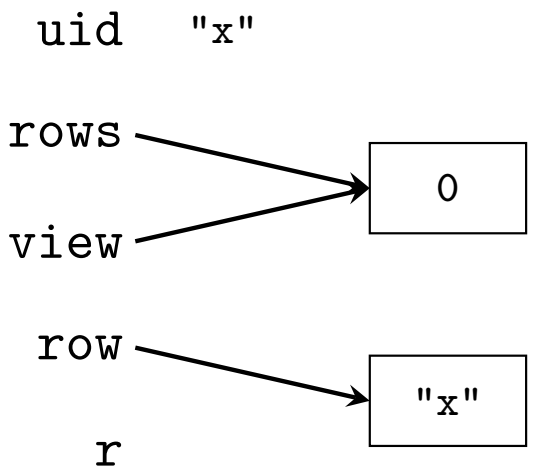
```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



Points-To Sets

Run `f("x")`:

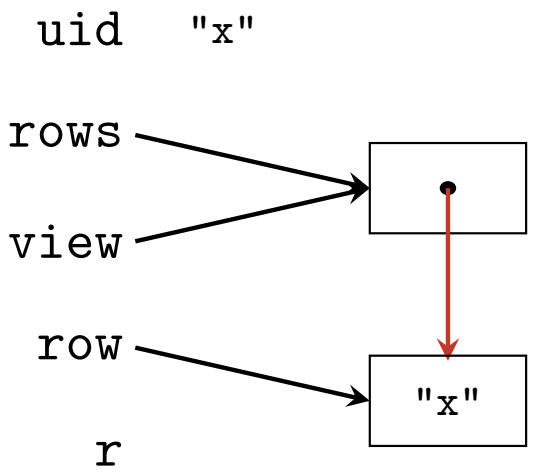
```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



Points-To Sets

Run `f("x")`:

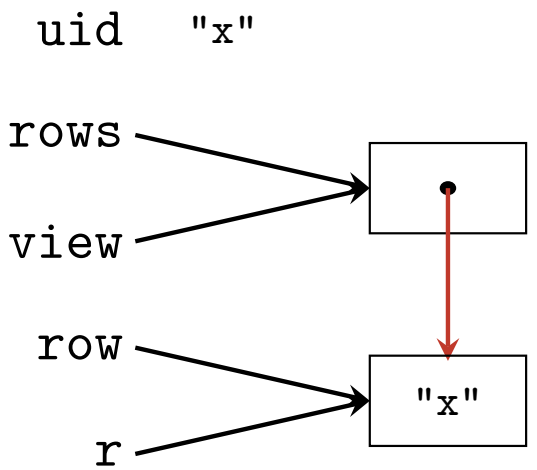
```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



Points-To Sets

Run `f("x")`:

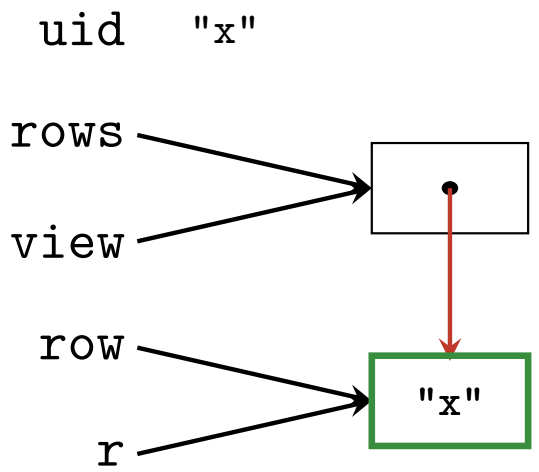
```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



Points-To Sets

Run `f("x")`:

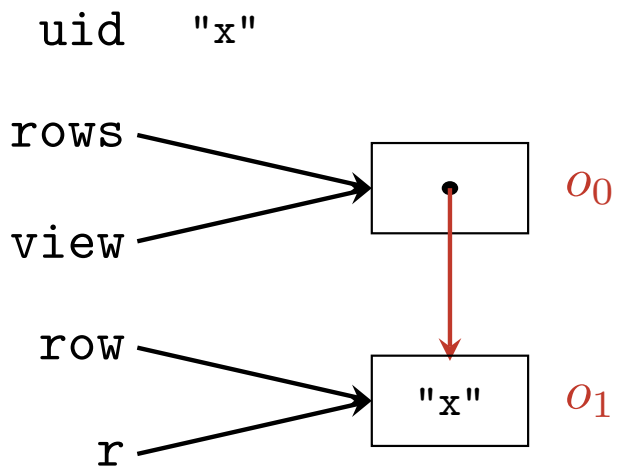
```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



Points-To Sets

Run `f("x")`:

```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



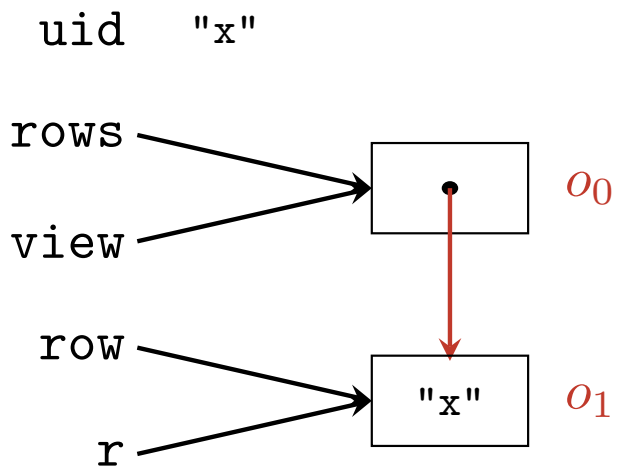
The analysis gets this *without running*, one object per `[...]`:

$$\text{pt}(\texttt{rows}) = \text{pt}(\texttt{view}) = \{o_0\} \qquad \text{pt}(o_0[0]) = \{o_1\}$$
$$\text{pt}(\texttt{row}) = \text{pt}(\texttt{r}) = \{o_1\} \qquad \text{pt}(o_1[0]) = \{\text{in_uid}\}$$

Points-To Sets

Run `f("x")`:

```
def f(uid):
    rows = [0]      # o0
    row = [uid]     # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]
```



The analysis gets this *without running*, one object per `[...]`:

$$\text{pt}(\texttt{rows}) = \text{pt}(\texttt{view}) = \{o_0\} \qquad \text{pt}(o_0[0]) = \{o_1\}$$
$$\text{pt}(\texttt{row}) = \text{pt}(\texttt{r}) = \{o_1\} \qquad \text{pt}(o_1[0]) = \{\text{in_uid}\}$$

- `view` and `rows` **may alias**: their sets meet. `r[0]` reads $o_1[0]$, which holds `uid`: **tainted**.

Allocation Sites

Run `f(3)`:

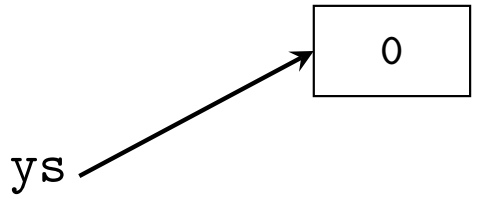
```
def f(n):
    i = 0
    while i < n:
        ys = [i]
        i = i + 1
    return ys
```

ys

Allocation Sites

Run `f(3)`:

```
def f(n):
    i = 0
    while i < n:
        ys = [i]
        i = i + 1
    return ys
```

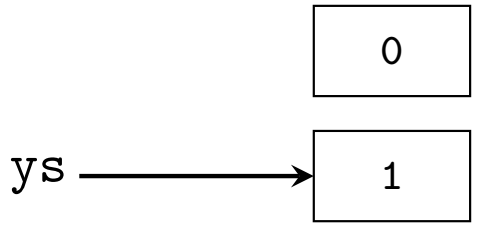


- One list per lap: as many as `n`, **unbounded**.

Allocation Sites

Run `f(3)`:

```
def f(n):
    i = 0
    while i < n:
        ys = [i]
        i = i + 1
    return ys
```

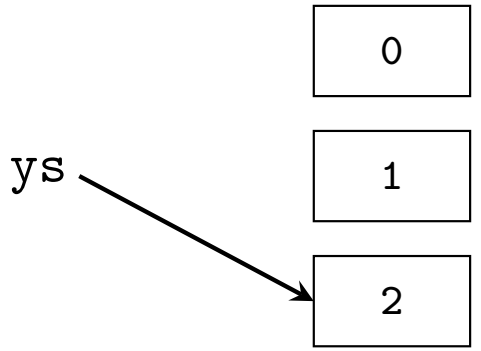


- One list per lap: as many as `n`, **unbounded**.

Allocation Sites

Run `f(3)`:

```
def f(n):
    i = 0
    while i < n:
        ys = [i]
        i = i + 1
    return ys
```

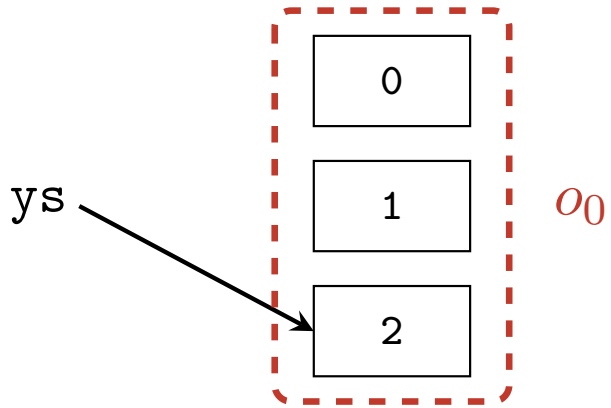


- One list per lap: as many as `n`, **unbounded**.

Allocation Sites

Run `f(3)`:

```
def f(n):
    i = 0
    while i < n:
        ys = [i]
        i = i + 1
    return ys
```



- One list per lap: as many as `n`, **unbounded**.
- **Allocation site**: one name per `[...]` in the code. All three lists are o_0.

Allocation Sites and Calls

Real Python, run `f("x")`:

```
def new():
    return [0]        # o0
def f(uid):
    a = new()
    b = new()
    a[0] = uid
    return b[0]
```

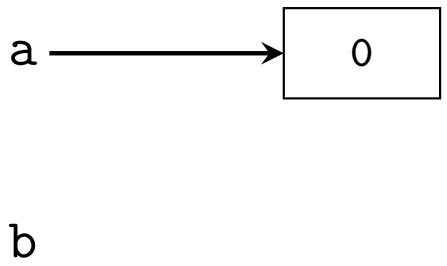
a

b

Allocation Sites and Calls

Real Python, run `f("x")`:

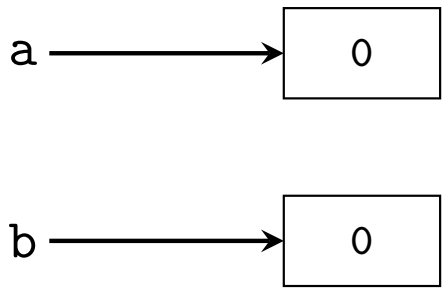
```
def new():
    return [0]        # o0
def f(uid):
    a = new()
    b = new()
    a[0] = uid
    return b[0]
```



Allocation Sites and Calls

Real Python, run `f("x")`:

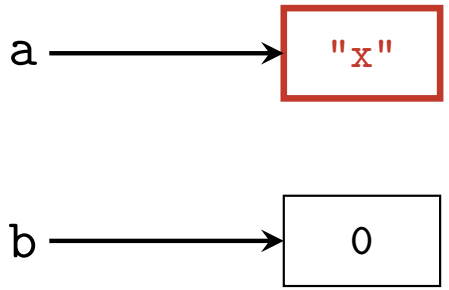
```
def new():
    return [0]        # o0
def f(uid):
    a = new()
    b = new()
    a[0] = uid
    return b[0]
```



Allocation Sites and Calls

Real Python, run `f("x")`:

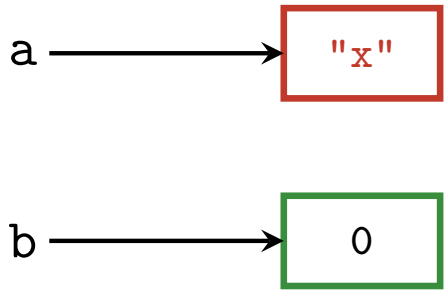
```
def new():
    return [0]        # o0
def f(uid):
    a = new()
    b = new()
    a[0] = uid
    return b[0]
```



Allocation Sites and Calls

Real Python, run `f("x")`:

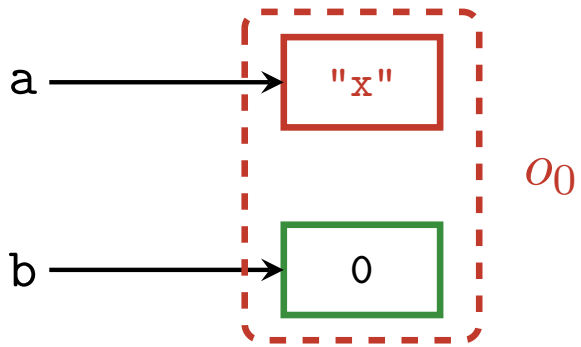
```
def new():
    return [0]        # o0
def f(uid):
    a = new()
    b = new()
    a[0] = uid
    return b[0]
```



Allocation Sites and Calls

Real Python, run `f("x")`:

```
def new():
    return [0]        # o0
def f(uid):
    a = new()
    b = new()
    a[0] = uid
    return b[0]
```



- At run time, two lists: `b[0]` is still `0`. But both come from the same `[0]`, so both are o_0: $\text{pt}(\text{a}) = \text{pt}(\text{b}) = \{o_0\}$, and `b[0]` looks tainted. A false alarm.

Contents

1. Points-To

2. Andersen's Analysis

3. Precision

4. Graph Reachability

5. In Python

Four Kinds of Statement

Each statement that touches a list gives one **constraint**:

Kind	MiniPy	**Constraint**
allocation	`x = [...]`	$o_i \in \mathsf{pt}(x)$
copy	`x = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(x)$
load	`x = y[k]`	$\mathsf{pt}(o[k]) \subseteq \mathsf{pt}(x)$ for each $o \in \mathsf{pt}(y)$
store	`x[k] = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

Four Kinds of Statement

Each statement that touches a list gives one **constraint**:

Kind	MiniPy	**Constraint**
allocation	`x = [...]`	$o_i \in \mathsf{pt}(x)$
copy	`x = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(x)$
load	`x = y[k]`	$\mathsf{pt}(o[k]) \subseteq \mathsf{pt}(x)$ for each $o \in \mathsf{pt}(y)$
store	`x[k] = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

- A literal index is its own field: `xs[0]` and `xs[1]` stay apart. A computed index `xs[i]` uses one field $*$.

Four Kinds of Statement

Each statement that touches a list gives one **constraint**:

Kind	MiniPy	**Constraint**
allocation	`x = [...]`	$o_i \in \mathsf{pt}(x)$
copy	`x = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(x)$
load	`x = y[k]`	$\mathsf{pt}(o[k]) \subseteq \mathsf{pt}(x)$ for each $o \in \mathsf{pt}(y)$
store	`x[k] = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

- A literal index is its own field: `xs[0]` and `xs[1]` stay apart. A computed index `xs[i]` uses one field $*$.
- **Copy** is $\subseteq$, not $=$: `x` points to *at least* what `y` does.

Four Kinds of Statement

Each statement that touches a list gives one **constraint**:

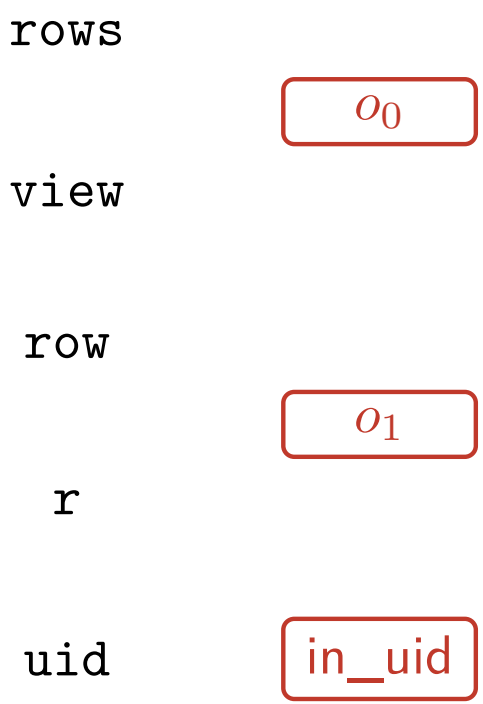
Kind	MiniPy	**Constraint**
allocation	`x = [...]`	$o_i \in \mathsf{pt}(x)$
copy	`x = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(x)$
load	`x = y[k]`	$\mathsf{pt}(o[k]) \subseteq \mathsf{pt}(x)$ for each $o \in \mathsf{pt}(y)$
store	`x[k] = y`	$\mathsf{pt}(y) \subseteq \mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

- A literal index is its own field: `xs[0]` and `xs[1]` stay apart. A computed index `xs[i]` uses one field $*$.
- **Copy** is $\subseteq$, not $=$: `x` points to *at least* what `y` does.
- This is **Andersen's analysis**: collect the constraints, then find the **least** solution.

Andersen, Step by Step

`def f(uid):`	$\mathsf{in_uid} \in \mathsf{pt}(\texttt{uid})$
`rows = [0]`	$o_0 \in \mathsf{pt}(\texttt{rows})$
`row = [uid]`	$o_1 \in \mathsf{pt}(\texttt{row})$ $\mathsf{pt}(\texttt{uid}) \subseteq \mathsf{pt}(\texttt{row}[0])$
`view = rows`	$\mathsf{pt}(\texttt{rows}) \subseteq \mathsf{pt}(\texttt{view})$
`view[0] = row`	$\mathsf{pt}(\texttt{row}) \subseteq \mathsf{pt}(\texttt{view}[0])$
`r = rows[0]`	$\mathsf{pt}(\texttt{rows}[0]) \subseteq \mathsf{pt}(\texttt{r})$

$\mathsf{pt}(x[k])$: $\mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

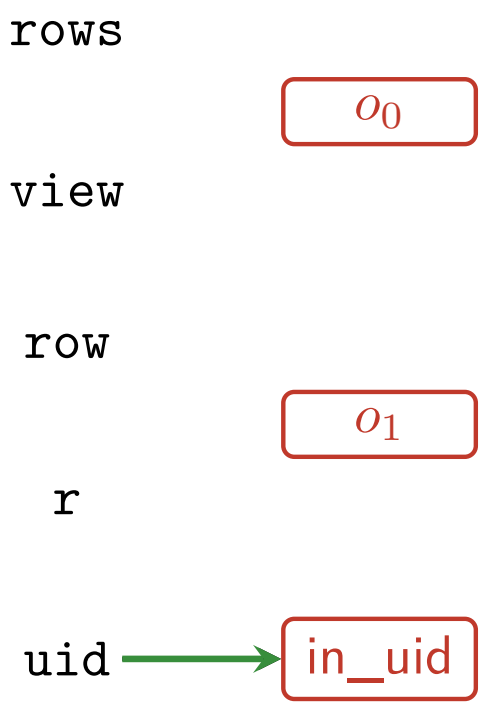


One constraint per line. Start with every set empty.

Andersen, Step by Step

`def f(uid):`	$\mathsf{in_uid} \in \mathsf{pt}(\texttt{uid})$
`rows = [0]`	$o_0 \in \mathsf{pt}(\texttt{rows})$
`row = [uid]`	$o_1 \in \mathsf{pt}(\texttt{row})$ $\mathsf{pt}(\texttt{uid}) \subseteq \mathsf{pt}(\texttt{row}[0])$
`view = rows`	$\mathsf{pt}(\texttt{rows}) \subseteq \mathsf{pt}(\texttt{view})$
`view[0] = row`	$\mathsf{pt}(\texttt{row}) \subseteq \mathsf{pt}(\texttt{view}[0])$
`r = rows[0]`	$\mathsf{pt}(\texttt{rows}[0]) \subseteq \mathsf{pt}(\texttt{r})$

$\mathsf{pt}(x[k])$: $\mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

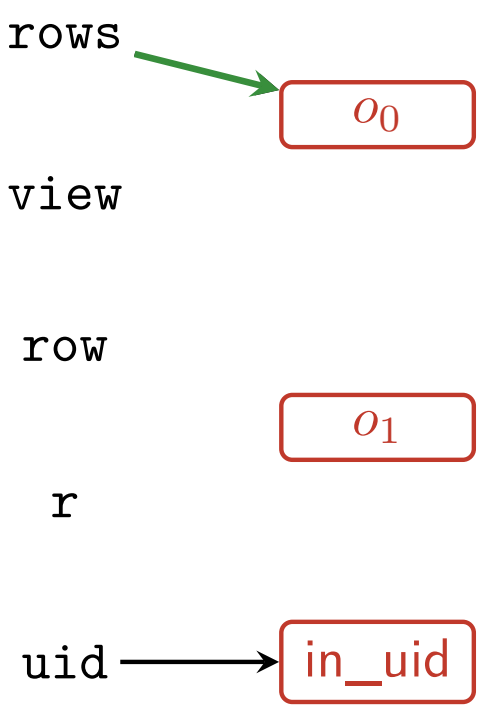


A parameter comes from the caller: one object for whatever it is, in_uid.

Andersen, Step by Step

`def f(uid):`	in_uid $\in$ pt(`uid`)
`rows = [0]`	$o_0 \in$ pt(`rows`)
`row = [uid]`	$o_1 \in$ pt(`row`) pt(`uid`) $\subseteq$ pt(`row`$[0]$)
`view = rows`	pt(`rows`) $\subseteq$ pt(`view`)
`view[0] = row`	pt(`row`) $\subseteq$ pt(`view`$[0]$)
`r = rows[0]`	pt(`rows`$[0]$) $\subseteq$ pt(`r`)

pt($x[k]$): pt($o[k]$) for each $o \in$ pt(x)

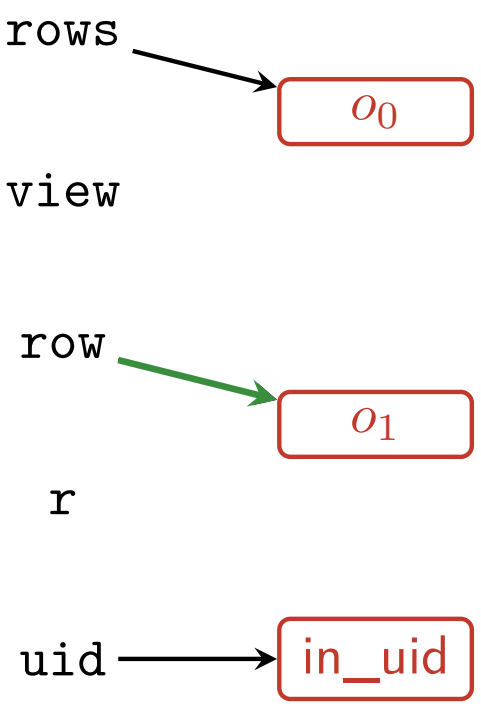


Allocation: `rows` points to o_0.

Andersen, Step by Step

`def f(uid):`	in_uid $\in$ pt(`uid`)
`rows = [0]`	$o_0 \in$ pt(`rows`)
`row = [uid]`	$o_1 \in$ pt(`row`) pt(`uid`) $\subseteq$ pt(`row`$[0]$)
`view = rows`	pt(`rows`) $\subseteq$ pt(`view`)
`view[0] = row`	pt(`row`) $\subseteq$ pt(`view`$[0]$)
`r = rows[0]`	pt(`rows`$[0]$) $\subseteq$ pt(`r`)

pt($x[k]$): pt($o[k]$) for each $o \in$ pt(x)

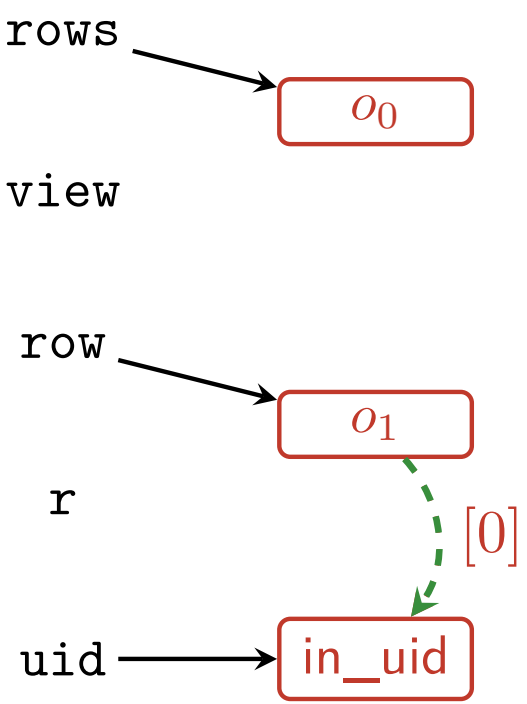


Allocation: `row` points to o_1.

Andersen, Step by Step

`def f(uid):`	$\mathsf{in_uid} \in \mathsf{pt}(\texttt{uid})$
`rows = [0]`	$o_0 \in \mathsf{pt}(\texttt{rows})$
`row = [uid]`	$o_1 \in \mathsf{pt}(\texttt{row})$ $\mathsf{pt}(\texttt{uid}) \subseteq \mathsf{pt}(\texttt{row}[0])$
`view = rows`	$\mathsf{pt}(\texttt{rows}) \subseteq \mathsf{pt}(\texttt{view})$
`view[0] = row`	$\mathsf{pt}(\texttt{row}) \subseteq \mathsf{pt}(\texttt{view}[0])$
`r = rows[0]`	$\mathsf{pt}(\texttt{rows}[0]) \subseteq \mathsf{pt}(\texttt{r})$

$\mathsf{pt}(x[k])$: $\mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

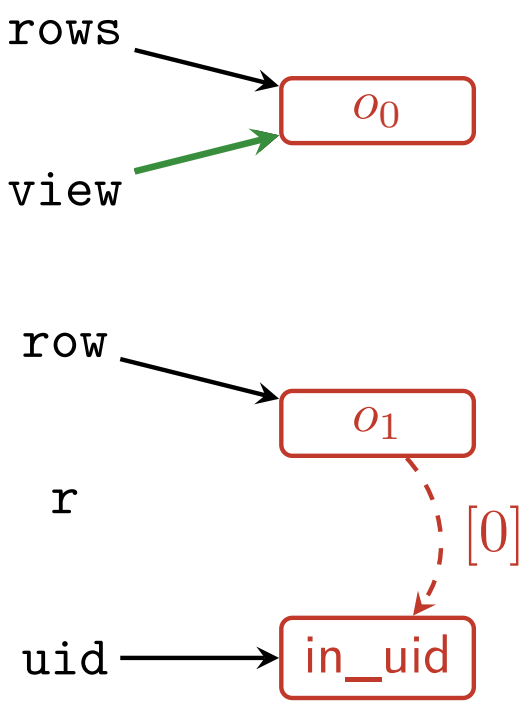


`[uid]` also stores `uid` into element 0: $\mathsf{pt}(o_1[0]) = \{\mathsf{in_uid}\}$.

Andersen, Step by Step

`def f(uid):`	in_uid $\in$ pt(`uid`)
`rows = [0]`	$o_0 \in$ pt(`rows`)
`row = [uid]`	$o_1 \in$ pt(`row`) pt(`uid`) $\subseteq$ pt(`row`$[0]$)
`view = rows`	pt(`rows`) $\subseteq$ pt(`view`)
`view[0] = row`	pt(`row`) $\subseteq$ pt(`view`$[0]$)
`r = rows[0]`	pt(`rows`$[0]$) $\subseteq$ pt(`r`)

pt($x[k]$): pt($o[k]$) for each $o \in$ pt(x)

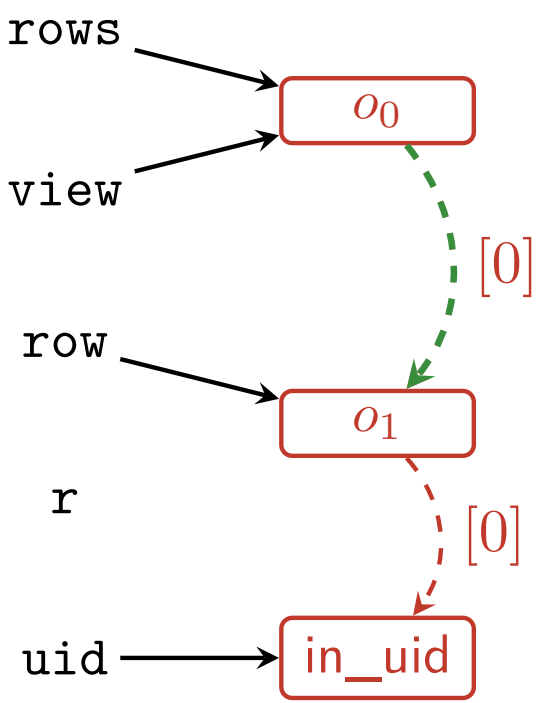


Copy: `view` gets what `rows` has, o_0.

Andersen, Step by Step

`def f(uid):`	$\mathsf{in_uid} \in \mathsf{pt}(\texttt{uid})$
`rows = [0]`	$o_0 \in \mathsf{pt}(\texttt{rows})$
`row = [uid]`	$o_1 \in \mathsf{pt}(\texttt{row})$ $\mathsf{pt}(\texttt{uid}) \subseteq \mathsf{pt}(\texttt{row}[0])$
`view = rows`	$\mathsf{pt}(\texttt{rows}) \subseteq \mathsf{pt}(\texttt{view})$
`view[0] = row`	$\mathsf{pt}(\texttt{row}) \subseteq \mathsf{pt}(\texttt{view}[0])$
`r = rows[0]`	$\mathsf{pt}(\texttt{rows}[0]) \subseteq \mathsf{pt}(\texttt{r})$

$\mathsf{pt}(x[k])$: $\mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

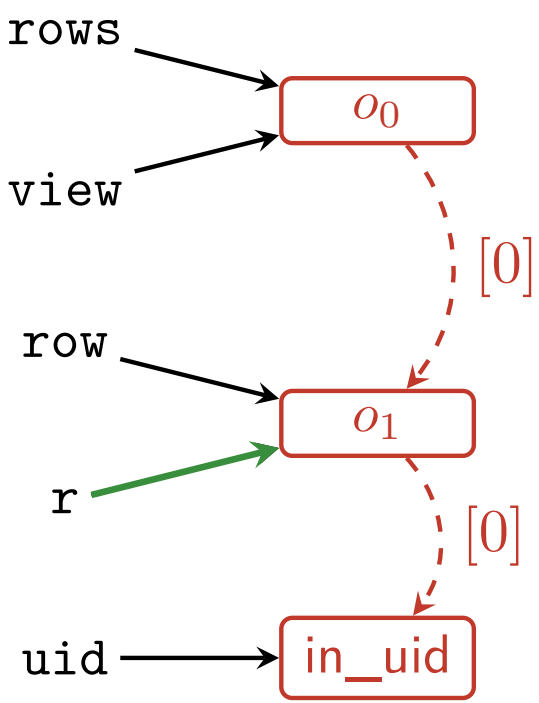


Store through `view`, which points to o_0: $\mathsf{pt}(o_0[0]) = \{o_1\}$.

Andersen, Step by Step

`def f(uid):`	$\mathsf{in_uid} \in \mathsf{pt}(\texttt{uid})$
`rows = [0]`	$o_0 \in \mathsf{pt}(\texttt{rows})$
`row = [uid]`	$o_1 \in \mathsf{pt}(\texttt{row})$ $\mathsf{pt}(\texttt{uid}) \subseteq \mathsf{pt}(\texttt{row}[0])$
`view = rows`	$\mathsf{pt}(\texttt{rows}) \subseteq \mathsf{pt}(\texttt{view})$
`view[0] = row`	$\mathsf{pt}(\texttt{row}) \subseteq \mathsf{pt}(\texttt{view}[0])$
`r = rows[0]`	$\mathsf{pt}(\texttt{rows}[0]) \subseteq \mathsf{pt}(\texttt{r})$

$\mathsf{pt}(x[k])$: $\mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$

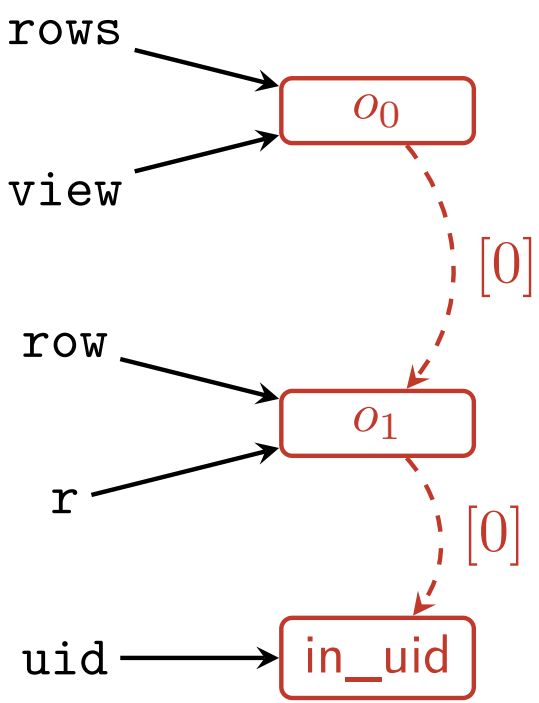


Load through `rows`, which points to o_0: `r` gets $\mathsf{pt}(o_0[0]) = \{o_1\}$.

Andersen, Step by Step

`def f(uid):`	$\mathsf{in_uid} \in \mathsf{pt}(\texttt{uid})$
`rows = [0]`	$o_0 \in \mathsf{pt}(\texttt{rows})$
`row = [uid]`	$o_1 \in \mathsf{pt}(\texttt{row})$ $\mathsf{pt}(\texttt{uid}) \subseteq \mathsf{pt}(\texttt{row}[0])$
`view = rows`	$\mathsf{pt}(\texttt{rows}) \subseteq \mathsf{pt}(\texttt{view})$
`view[0] = row`	$\mathsf{pt}(\texttt{row}) \subseteq \mathsf{pt}(\texttt{view}[0])$
`r = rows[0]`	$\mathsf{pt}(\texttt{rows}[0]) \subseteq \mathsf{pt}(\texttt{r})$

$\mathsf{pt}(x[k])$: $\mathsf{pt}(o[k])$ for each $o \in \mathsf{pt}(x)$



Go around again: **nothing changes**. Stop. The store went through `view` and the load through `rows`: they meet at o_0.

Order Does Not Matter

```
def f():
    a = [0]        # o0
    b = [1]        # o1
    x = a
    x = b
    return x
```

	x at the end	x may alias a
real runs	o_1	no

Order Does Not Matter

```
def f():
    a = [0]        # o0
    b = [1]        # o1
    x = a
    x = b
    return x
```

	x at the end	x may alias a
real runs	o_1	no
Andersen	$\{o_0, o_1\}$	yes

- No CFG: Andersen sees a **set** of statements, not their order. It is **flow-insensitive**.

Order Does Not Matter

```
def f():
    a = [0]        # o0
    b = [1]        # o1
    x = a
    x = b
    return x
```

	x at the end	x may alias a
real runs	o_1	no
Andersen	$\{o_0, o_1\}$	yes

- No CFG: Andersen sees a **set** of statements, not their order. It is **flow-insensitive**.
- Still **sound**: anything that happens in some order is included.
- Much cheaper, and on real code variables are rarely pointed somewhere else. One answer serves every point of the program.

Two Bugs, Fixed

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1
    return u
```

Two Bugs, Fixed

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1
    return u
```

$\mathsf{pt}(\texttt{row}) = \mathsf{pt}(\texttt{view}) = \{o_0\}$

`row[0] = uid` taints $o_0[0]$.

`view[0]` reads $o_0[0]$: **alarm**.

Two Bugs, Fixed

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```

$\mathrm{pt}(\texttt{row}) = \mathrm{pt}(\texttt{view}) = \{o_0\}$

`row[0] = uid` taints $o_0[0]$.

`view[0]` reads $o_0[0]$: **alarm**.

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1
    return u
```

$\mathrm{pt}(\texttt{xs}) = \mathrm{pt}(\texttt{ys}) = \{o_0\}$

`ys[0] = v` writes $o_0[0]$.

Kill `xs[0] + 1`: **recompute**.

Two Bugs, Fixed

```
def f(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```

$\mathrm{pt}(\texttt{row}) = \mathrm{pt}(\texttt{view}) = \{o_0\}$

`row[0] = uid` taints $o_0[0]$.

`view[0]` reads $o_0[0]$: **alarm**.

```
def f(v):
    xs = [0]
    ys = xs
    t = xs[0] + 1
    ys[0] = v
    u = xs[0] + 1
    return u
```

$\mathrm{pt}(\texttt{xs}) = \mathrm{pt}(\texttt{ys}) = \{o_0\}$

`ys[0] = v` writes $o_0[0]$.

Kill `xs[0] + 1`: **recompute**.

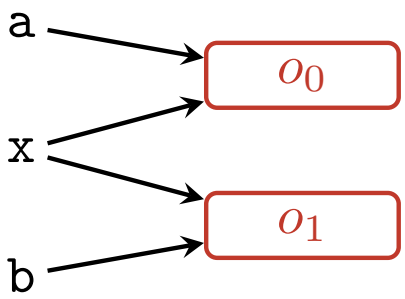
- Taint moves from *variables* to **objects**: one fact per $o[k]$, shared by every name that points to o.
- A store `ys[k] = e` kills what reads `xs[k]` for every `xs` that may alias `ys`.

Steensgaard's Analysis

Faster and coarser: treat `x = y` as **equality**, and merge.

```
a = [0]        # o0
b = [1]        # o1
x = a
x = b
```

Andersen ($\subseteq$)

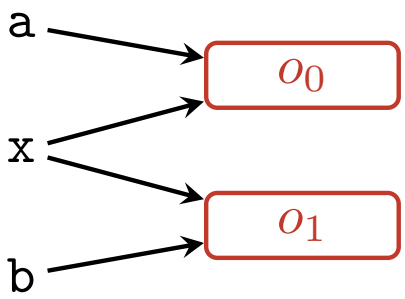


Steensgaard's Analysis

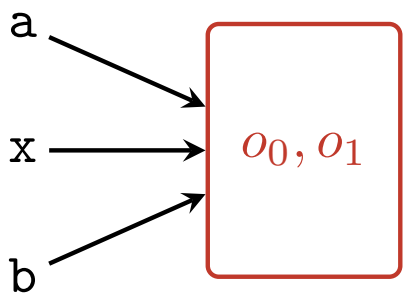
Faster and coarser: treat `x = y` as **equality**, and merge.

```
a = [0]        # o0
b = [1]        # o1
x = a
x = b
```

Andersen ($\subseteq$)



Steensgaard ($=$)

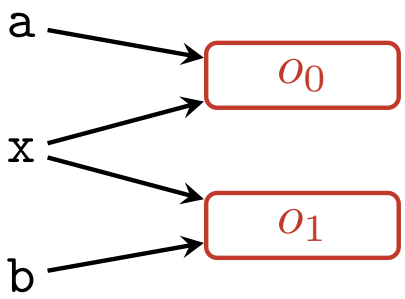


Steensgaard's Analysis

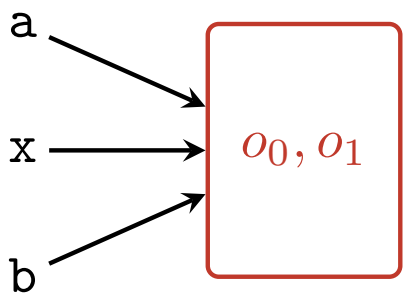
Faster and coarser: treat `x = y` as **equality**, and merge.

```
a = [0]        # o0
b = [1]        # o1
x = a
x = b
```

Andersen ($\subseteq$)



Steensgaard ($=$)



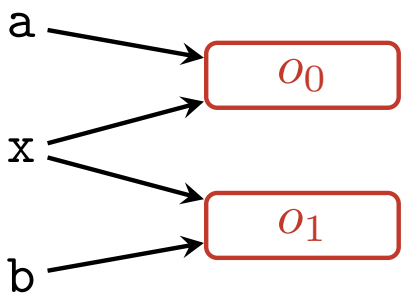
- `x = a` merges pt(`x`) with pt(`a`), then `x = b` merges it with pt(`b`). Now `a` and `b` alias, though they never meet.

Steensgaard's Analysis

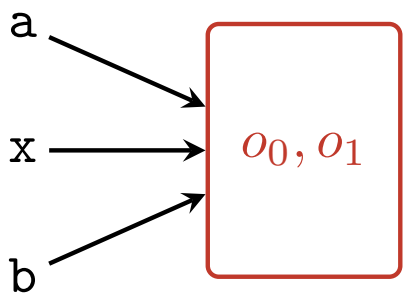
Faster and coarser: treat `x = y` as **equality**, and merge.

```
a = [0]        # o0
b = [1]        # o1
x = a
x = b
```

Andersen ($\subseteq$)



Steensgaard ($=$)



- `x = a` merges pt(`x`) with pt(`a`), then `x = b` merges it with pt(`b`). Now `a` and `b` alias, though they never meet.

	Andersen	Steensgaard
`x = y` means	$\text{pt}(y) \subseteq \text{pt}(x)$	$\text{pt}(y) = \text{pt}(x)$
time	$O(n^3)$	almost $O(n)$ (union-find)
precision	higher	lower

Contents

1. Points-To

2. Andersen's Analysis

3. Precision

4. Graph Reachability

5. In Python

Strong and Weak Updates

```
def f(uid):
    xs = [0]
    xs[0] = uid
    xs[0] = 0
    return xs[0]    # always 0
```

```
def g(uid, c):
    a = [0]
    b = [0]
    if c:
        xs = a
    else:
        xs = b
    a[0] = uid
    xs[0] = 0
    return a[0]     # uid if not c
```

Strong and Weak Updates

```
def f(uid):
    xs = [0]
    xs[0] = uid
    xs[0] = 0
    return xs[0]    # always 0
```

$\mathrm{pt}(\texttt{xs}) = \{o_0\}$, and o_0 is **one** list.

`xs[0] = 0` **overwrites** $o_0[0]$:
a **strong update**. Clean.

```
def g(uid, c):
    a = [0]
    b = [0]
    if c:
        xs = a
    else:
        xs = b
    a[0] = uid
    xs[0] = 0
    return a[0]     # uid if not c
```

Strong and Weak Updates

```
def f(uid):
    xs = [0]
    xs[0] = uid
    xs[0] = 0
    return xs[0]    # always 0
```

$\text{pt}(\texttt{xs}) = \{o_0\}$, and o_0 is **one** list.

`xs[0] = 0` **overwrites** $o_0[0]$:
a **strong update**. Clean.

```
def g(uid, c):
    a = [0]
    b = [0]
    if c:
        xs = a
    else:
        xs = b
    a[0] = uid
    xs[0] = 0
    return a[0]    # uid if not c
```

$\text{pt}(\texttt{xs}) = \{o_0, o_1\}$: which one was written?

We may only **add**: a **weak update**.

Strong and Weak Updates

```
def f(uid):
    xs = [0]
    xs[0] = uid
    xs[0] = 0
    return xs[0]    # always 0
```

$\text{pt}(\texttt{xs}) = \{o_0\}$, and o_0 is **one** list.

`xs[0] = 0` **overwrites** $o_0[0]$:
a **strong update**. Clean.

```
def g(uid, c):
    a = [0]
    b = [0]
    if c:
        xs = a
    else:
        xs = b
    a[0] = uid
    xs[0] = 0
    return a[0]    # uid if not c
```

$\text{pt}(\texttt{xs}) = \{o_0, o_1\}$: which one was written?

We may only **add**: a **weak update**.

- Strong update needs: **one** target, which is **one** list (not allocated in a loop), and the order of statements.
- Our analyses only do weak updates, so `f` gets a false alarm. Sound, but imprecise.

Field Sensitivity

Taint already split `xs[0]` and `xs[1]`. Points-to splits them the same way, on **objects**: $\text{pt}(o_0[0])$ and $\text{pt}(o_0[1])$.

Field Sensitivity

Taint already split `xs[0]` and `xs[1]`. Points-to splits them the same way, on **objects**: $\text{pt}(o_0[0])$ and $\text{pt}(o_0[1])$.

	one set per	`xs[0]` vs. `xs[1]`
field-insensitive	object o	same
field-sensitive	(o, k)	different
field-based	field k, for all objects	different, but all `[0]`s are one

Field Sensitivity

Taint already split `xs[0]` and `xs[1]`. Points-to splits them the same way, on **objects**: pt($o_0[0]$) and pt($o_0[1]$).

	one set per	`xs[0]` vs. `xs[1]`
field-insensitive	object o	same
field-sensitive	(o, k)	different
field-based	field k, for all objects	different, but all `[0]`s are one

In real Python, the fields are dictionary keys:

```
d = {"user": request.args["id"], "role": "guest"}
os.system("ls " + d["role"])      # field-insensitive: alarm
```

Field Sensitivity

Taint already split `xs[0]` and `xs[1]`. Points-to splits them the same way, on **objects**: $\text{pt}(o_0[0])$ and $\text{pt}(o_0[1])$.

	one set per	`xs[0]` vs. `xs[1]`
field-insensitive	object o	same
field-sensitive	(o, k)	different
field-based	field k, for all objects	different, but all `[0]`s are one

In real Python, the fields are dictionary keys:

```
d = {"user": request.args["id"], "role": "guest"}
os.system("ls " + d["role"])      # field-insensitive: alarm
```

- A computed index `xs[i]` is field $*$, which mixes with every other field. Interval analysis could tell which `i`.

Contents

1. Points-To

2. Andersen's Analysis

3. Precision

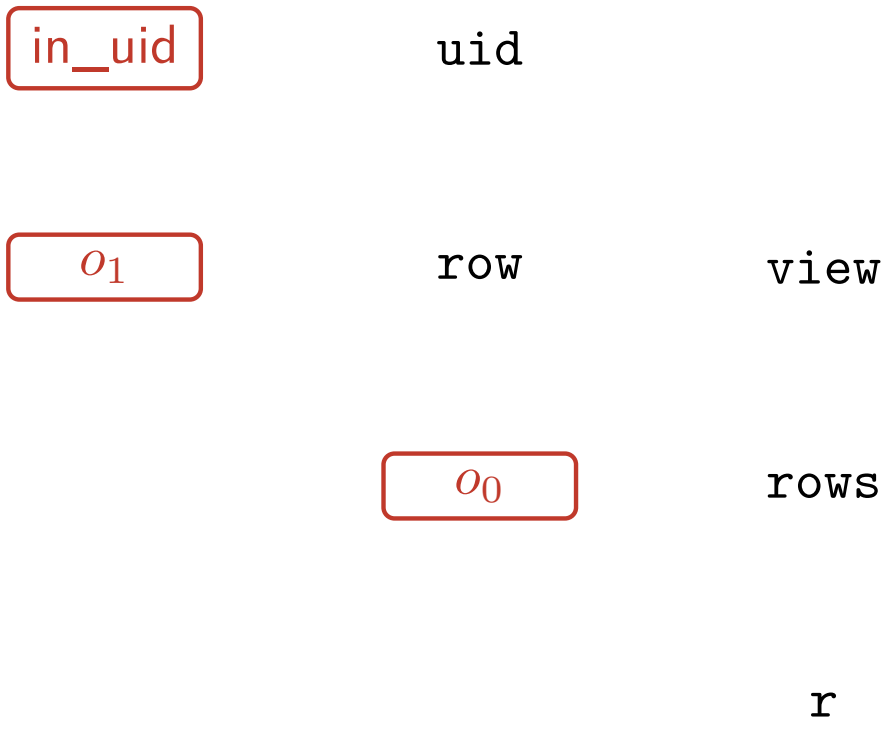
4. Graph Reachability

5. In Python

Points-To as a Graph

One node per variable and per object, one edge per statement:

`x = [...]`	$o_i \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assign}} x$
`x[k] = y`	$y \xrightarrow{\text{st}_k} x$
`x = y[k]`	$y \xrightarrow{\text{ld}_k} x$

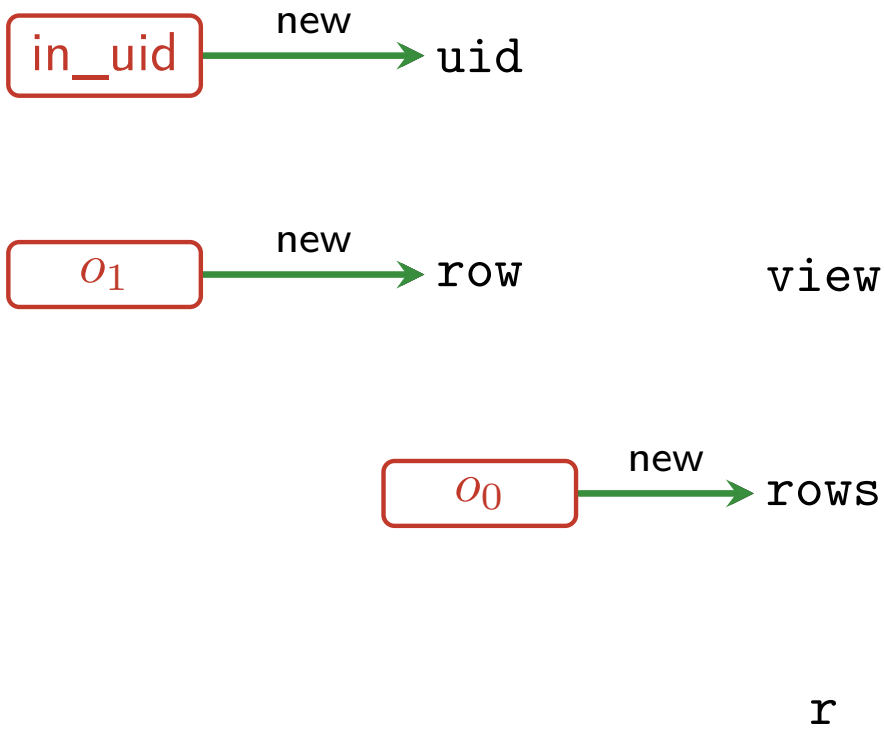


The running example: `rows = [0]; row = [uid]; view = rows; view[0] = row; r = rows[0].`

Points-To as a Graph

One node per variable and per object, one edge per statement:

`x = [...]`	$o_i \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assign}} x$
`x[k] = y`	$y \xrightarrow{\text{st}_k} x$
`x = y[k]`	$y \xrightarrow{\text{ld}_k} x$

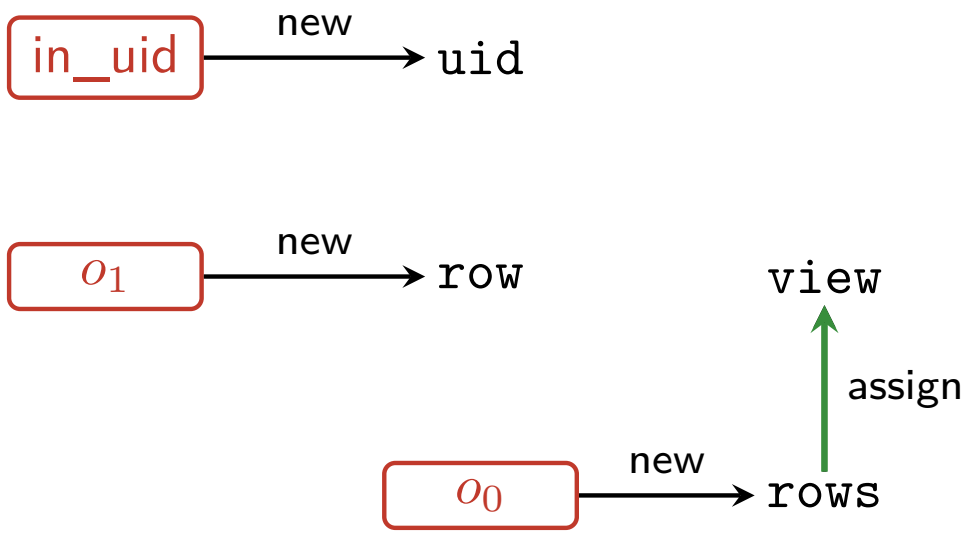


`rows = [0]`, `row = [uid]`, and the parameter `uid`.

Points-To as a Graph

One node per variable and per object, one edge per statement:

`x = [...]`	$o_i \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assign}} x$
`x[k] = y`	$y \xrightarrow{\text{st}_k} x$
`x = y[k]`	$y \xrightarrow{\text{ld}_k} x$

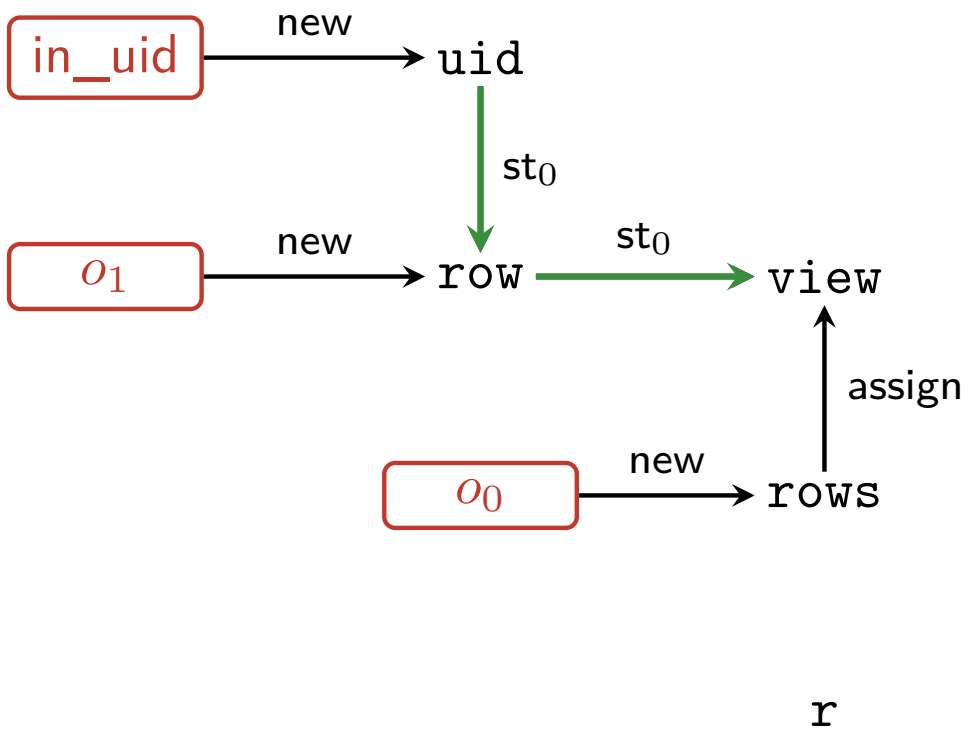


`view = rows.`

Points-To as a Graph

One node per variable and per object, one edge per statement:

`x = [...]`	$o_i \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assign}} x$
`x[k] = y`	$y \xrightarrow{\text{st}_k} x$
`x = y[k]`	$y \xrightarrow{\text{ld}_k} x$

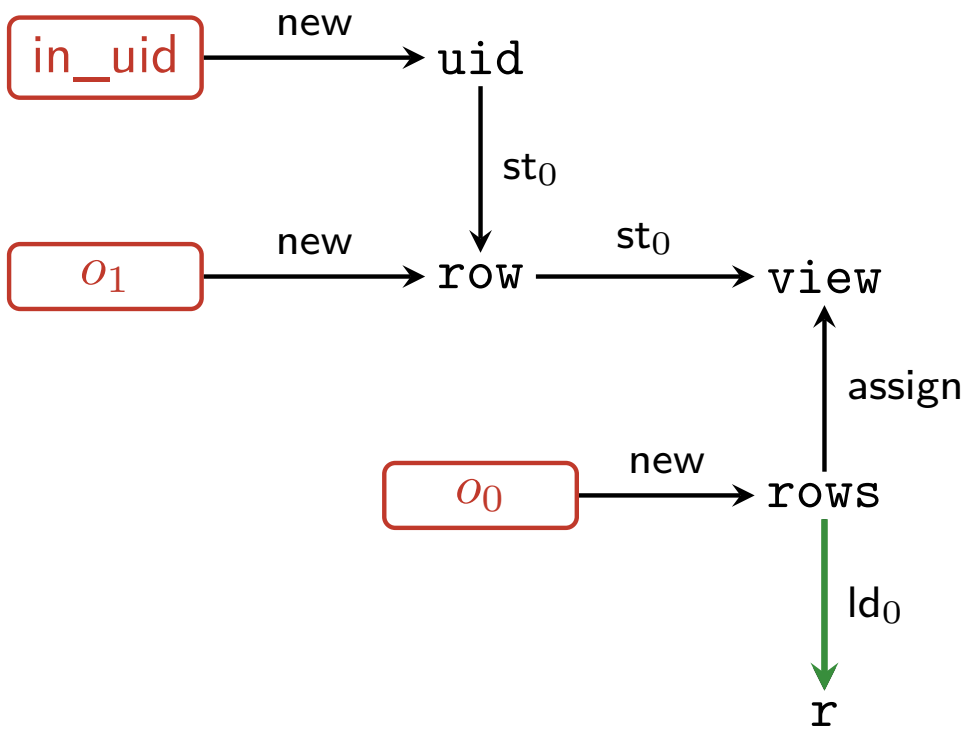


`view[0] = row`, and `[uid]` stores `uid` into element 0 of `row`.

Points-To as a Graph

One node per variable and per object, one edge per statement:

`x = [...]`	$o_i \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assign}} x$
`x[k] = y`	$y \xrightarrow{\text{st}_k} x$
`x = y[k]`	$y \xrightarrow{\text{ld}_k} x$

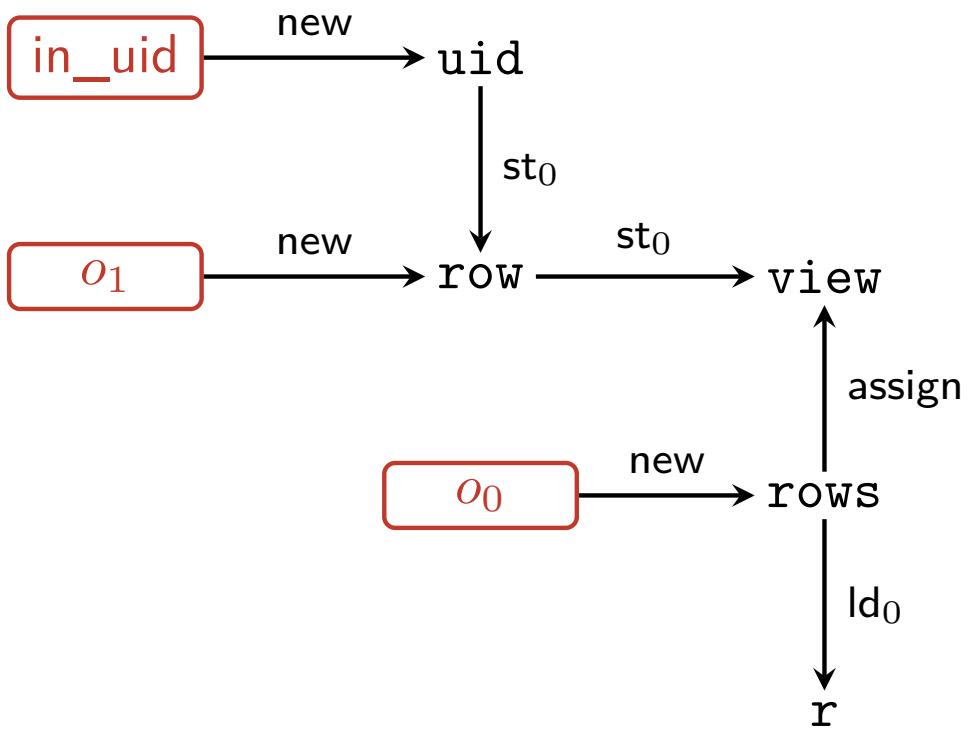



```
r = rows[0].
```

Points-To as a Graph

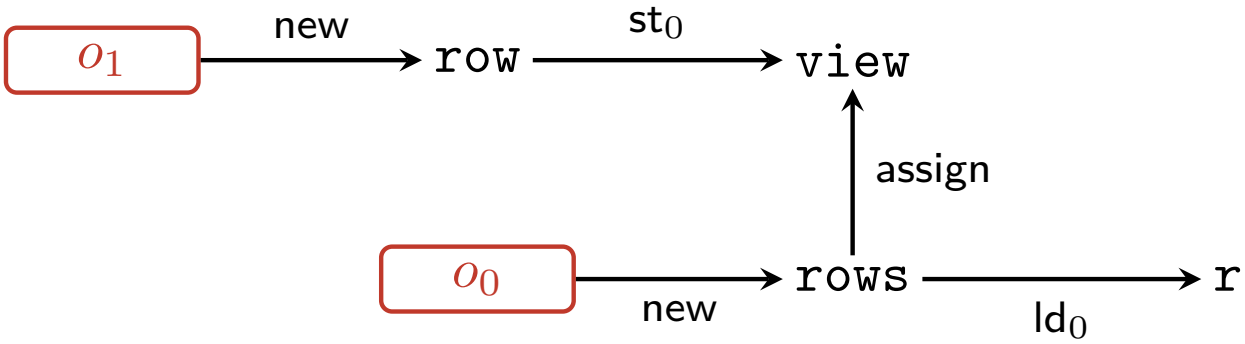
One node per variable and per object, one edge per statement:

`x = [...]`	$o_i \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assign}} x$
`x[k] = y`	$y \xrightarrow{\text{st}_k} x$
`x = y[k]`	$y \xrightarrow{\text{ld}_k} x$

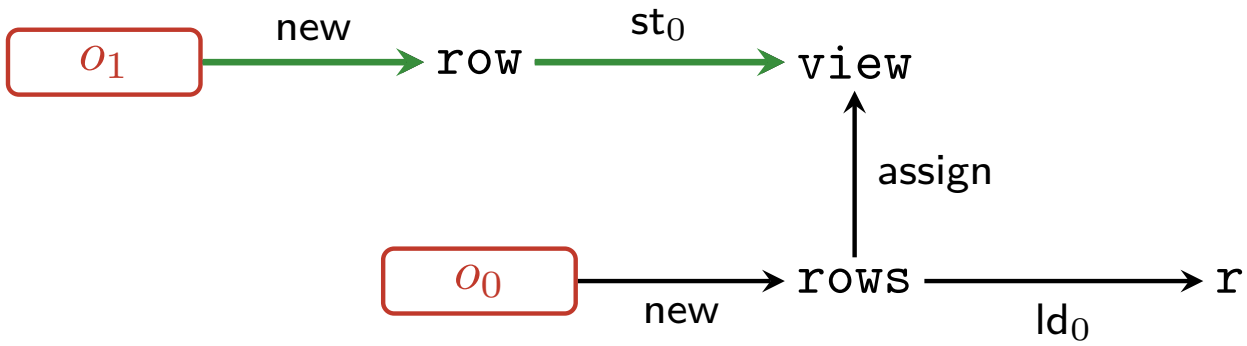


Is $o_1 \in \text{pt}(\texttt{r})$? There is no path $o_1 \rightsquigarrow \texttt{r}$ along the arrows.
Not every path counts, and some go **backwards**.

Balanced Parentheses



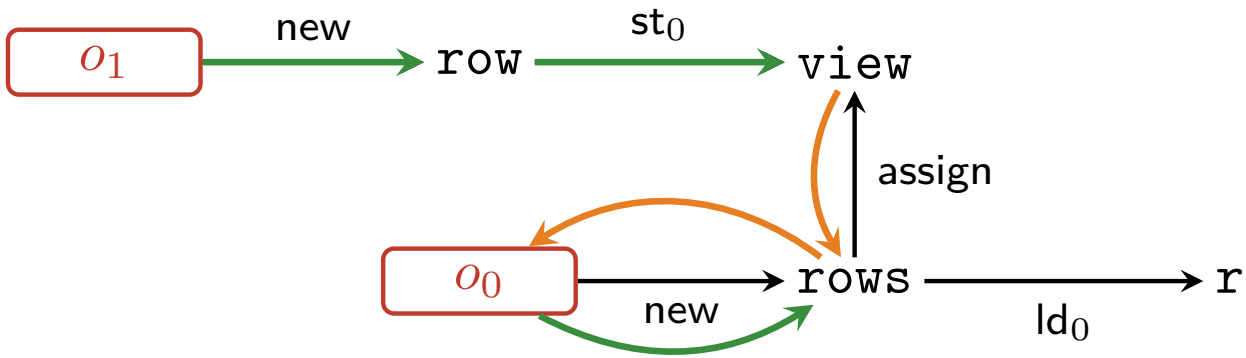
Balanced Parentheses



new st_0

- o_1 is stored into field 0 of `view`, ...

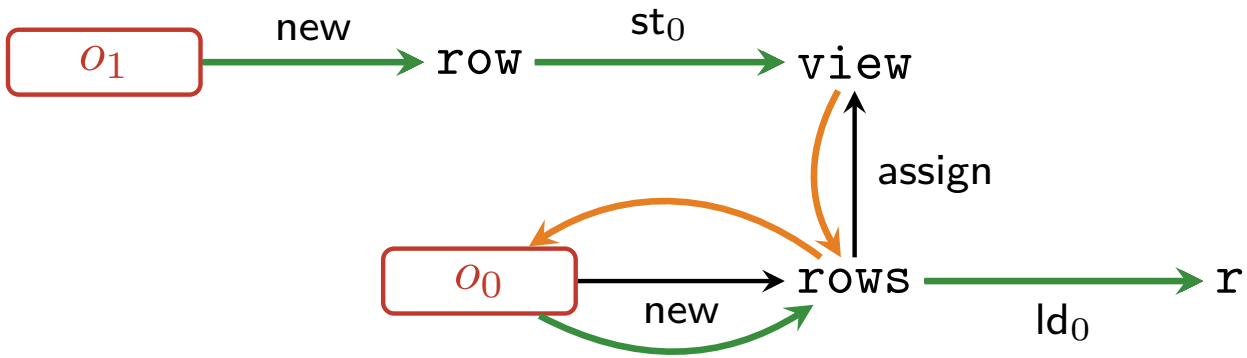
Balanced Parentheses



$$\text{new} \quad \text{st}_0 \quad \underbrace{\overline{\text{assign}}\ \overline{\text{new}} \quad \text{new}}_{\texttt{view} \text{ and } \texttt{rows} \text{ alias}}$$

- o_1 is stored into field 0 of `view`, ...
- ... go **back** from `view` to its object o_0, and forward again to `rows`, ...

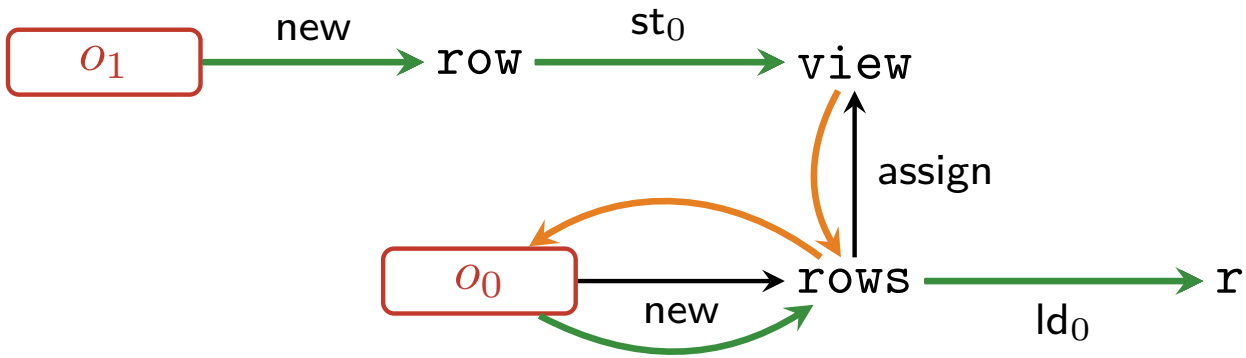
Balanced Parentheses



$$\text{new} \quad \text{st}_0 \quad \underbrace{\overline{\text{assign}}\ \overline{\text{new}} \quad \text{new}}_{\texttt{view and rows alias}} \quad \text{ld}_0$$

- o_1 is stored into field 0 of `view`, ...
- ... go **back** from `view` to its object o_0, and forward again to `rows`, ...
- ... and loaded from field 0 of `rows`. So $o_1 \in \text{pt}(\texttt{r})$.

Balanced Parentheses



$$\text{new}\ \ \text{st}_0\ \ \underbrace{\overline{\text{assign}}\ \overline{\text{new}}\ \ \text{new}}_{\texttt{view and rows alias}}\ \ \text{ld}_0$$

- o_1 is stored into field 0 of `view`, ...
- ... go **back** from `view` to its object o_0, and forward again to `rows`, ...
- ... and loaded from field 0 of `rows`. So $o_1 \in \text{pt}(\texttt{r})$.

$$\begin{aligned} \text{flowsTo} &::= \text{new}\ (\ \text{assign} \mid \text{st}_k\ \text{alias}\ \text{ld}_k\ )^* \\ \text{alias} &::= \overline{\text{flowsTo}}\ \text{flowsTo} \end{aligned}$$

CFL Reachability

Write st_k as $(_k$ and ld_k as $)_k$. Then $o \in \mathsf{pt}(x)$ iff the labels on $o \rightsquigarrow x$ are **balanced**:

labels	the program does	
$(_0 \ \ldots \ )_0$	store into `[0]`, load from `[0]`	
$(_0 \ \ldots \ )_1$	store into `[0]`, load from `[1]`	
$(_0 \ (_1 \ \ldots \ )_1 \ )_0$	a list inside a list	

CFL Reachability

Write st_k as $(_k$ and ld_k as $)_k$. Then $o \in \mathsf{pt}(x)$ iff the labels on $o \rightsquigarrow x$ are **balanced**:

labels	the program does	
$(_0 \ \ldots \ )_0$	store into `[0]`, load from `[0]`	✓
$(_0 \ \ldots \ )_1$	store into `[0]`, load from `[1]`	×
$(_0 \ (_1 \ \ldots \ )_1 \ )_0$	a list inside a list	✓

CFL Reachability

Write st_k as $(_k$ and ld_k as $)_k$. Then $o \in \mathsf{pt}(x)$ iff the labels on $o \rightsquigarrow x$ are **balanced**:

labels	the program does	
$(_0 \ \ldots \ )_0$	store into `[0]`, load from `[0]`	✓
$(_0 \ \ldots \ )_1$	store into `[0]`, load from `[1]`	×
$(_0 \ (_1 \ \ldots \ )_1 \ )_0$	a list inside a list	✓

- Checking this needs a **stack**, like parsing: **CFL reachability**. Cubic time.

CFL Reachability

Write st_k as $(_k$ and ld_k as $)_k$. Then $o \in \mathsf{pt}(x)$ iff the labels on $o \rightsquigarrow x$ are **balanced**:

labels	the program does	
$(_0 \ \ldots \ )_0$	store into `[0]`, load from `[0]`	✓
$(_0 \ \ldots \ )_1$	store into `[0]`, load from `[1]`	×
$(_0 \ (_1 \ \ldots \ )_1 \ )_0$	a list inside a list	✓

- Checking this needs a **stack**, like parsing: **CFL reachability**. Cubic time.
- Calls add another kind, $(_c \ldots)_c$: a call at site c must return to c. That is the next lecture.

CFL Reachability

Write st_k as $(_k$ and ld_k as $)_k$. Then $o \in \mathsf{pt}(x)$ iff the labels on $o \rightsquigarrow x$ are **balanced**:

labels	the program does	
$(_0 \ \ldots \ )_0$	store into `[0]`, load from `[0]`	✓
$(_0 \ \ldots \ )_1$	store into `[0]`, load from `[1]`	×
$(_0 \ (_1 \ \ldots \ )_1 \ )_0$	a list inside a list	✓

- Checking this needs a **stack**, like parsing: **CFL reachability**. Cubic time.
- Calls add another kind, $(_c \ldots)_c$: a call at site c must return to c. That is the next lecture.
- Fields and calls at once need **two** stacks: undecidable. Every real tool approximates fields, calls, or both.

Contents

1. Points-To

2. Andersen's Analysis

3. Precision

4. Graph Reachability

5. In Python

In Python – Constraints

```
def constraints(prog: Prog) -> list:
    ...
    for p in prog.params:                    # one object per parameter
        out.append(Alloc(p, Obj(f"in_{p}")))
    def stmt(st: Stmt) -> None:
        ...
        match st:
            case Assign(x, ListLit(items)):    # x = [..]
                out.append(Alloc(x, Obj(f"o{site}")))
                ...                            # and a Store per item
            case Assign(x, Var(y)):            # x = y
                out.append(Copy(x, y))
            case Assign(x, Index(Var(y), i)):  # x = y[i]
                out.append(Load(x, y, field(i)))
            case IndexAssign(x, i, e):         # x[i] = e
                if (y := source(e)) is not None:
                    out.append(Store(x, field(i), y))
            ...
```

In Python – Solving

```
def andersen(cs: list) -> dict:
    pt: dict = defaultdict(set)
    changed = True
    while changed:
        changed = False
        def grow(k, objs) -> None:
            nonlocal changed
            if not objs <= pt[k]:
                pt[k] |= objs
                changed = True
        for c in cs:
            match c:
                case Alloc(x, o):      grow(x, {o})
                case Copy(x, y):       grow(x, set(pt[y]))
                case Load(x, y, f):
                    for o in set(pt[y]):
                        grow(x, set(pt[(o, f)]))
                case Store(x, f, y):
                    for o in set(pt[x]):
                        grow((o, f), set(pt[y]))
    return dict(pt)
```

- Sets only grow, and there are finitely many objects: it stops.

Running It

```
>>> from pointer import points_to, may_alias, show
>>> src = """
... def f(uid):
...     rows = [0]
...     row = [uid]
...     view = rows
...     view[0] = row
...     r = rows[0]
...     return r[0]
... """
>>> pt = points_to(parse(src))
>>> print(show(pt))
  pt(o0[0]) = {o1}
  pt(o1[0]) = {in_uid}
  pt(r) = {o1}
  pt(row) = {o1}
  pt(rows) = {o0}
  pt(uid) = {in_uid}
  pt(view) = {o0}
>>> may_alias(pt, "view", "rows"), may_alias(pt, "r", "row")
(True, True)
>>> may_alias(pt, "rows", "row")
False
```

Summary

1. Points-To
2. Andersen's Analysis
3. Precision
4. Graph Reachability
5. In Python

Next Lecture

- Inter-procedural Analysis

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr