# Lecture 9 – Constraint-based Analysis

## SWS128: Software Analysis

Jihyeok Park




2026 Fall

# Recall

- **Dataflow analysis**: transfer functions, solved by a worklist.
- **Pointer analysis**: constraints such as $\mathrm{pt}(y) \subseteq \mathrm{pt}(x)$, solved by a loop.
- **Inter-procedural analysis**: the same analyses, across calls.
- Each analysis came with its **own solver**.

# Recall

- **Dataflow analysis**: transfer functions, solved by a worklist.
- **Pointer analysis**: constraints such as $\mathsf{pt}(y) \subseteq \mathsf{pt}(x)$, solved by a loop.
- **Inter-procedural analysis**: the same analyses, across calls.
- Each analysis came with its **own solver**.

- Today, **constraint-based analysis**:
  the same constraints, as **Datalog** rules, and one engine solves them all.

# Contents

1. Datalog

2. Pointer Analysis as Rules

3. More Analyses as Rules

4. In Practice

# Contents

1. Datalog

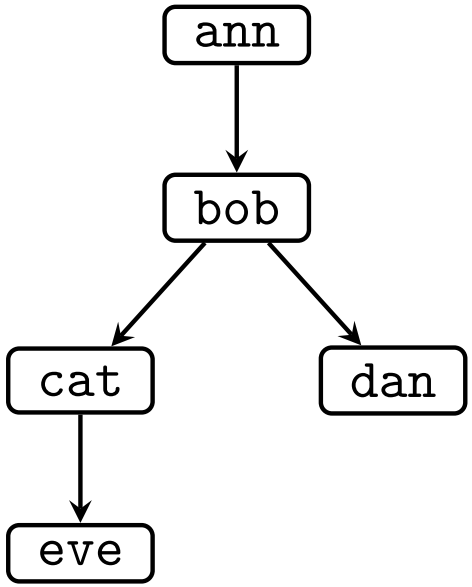
2. Pointer Analysis as Rules

3. More Analyses as Rules

4. In Practice

# Datalog – Facts

**Datalog**: a small language of **facts**, and **rules** that derive more facts.



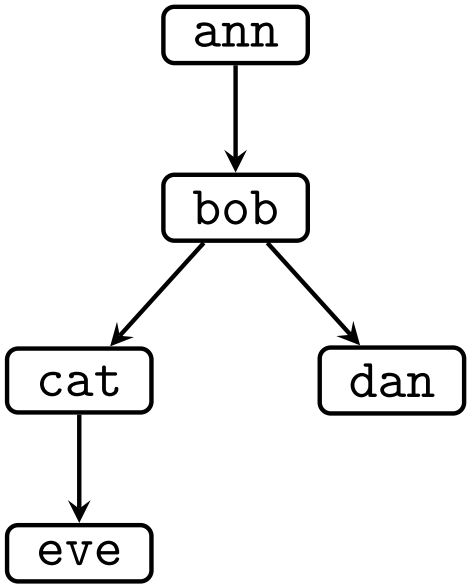

```
parent(ann, bob).
parent(bob, cat).
parent(bob, dan).
parent(cat, eve).
```

A **fact** is a name and its arguments: one arrow, or one row of a table.

Who are the **ancestors** of `eve`? No fact says so.

# Datalog – Rules

A **rule** is `head :- body.` If the body holds, the head holds.

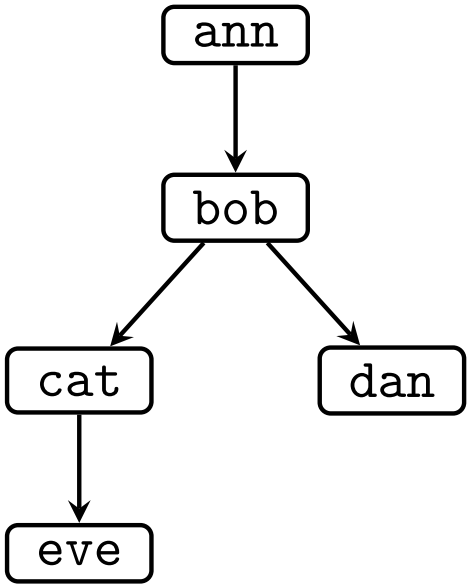



```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

A capital letter is a **variable**. A comma reads *and*.

# Datalog – Rules

A **rule** is `head :- body.` If the body holds, the head holds.



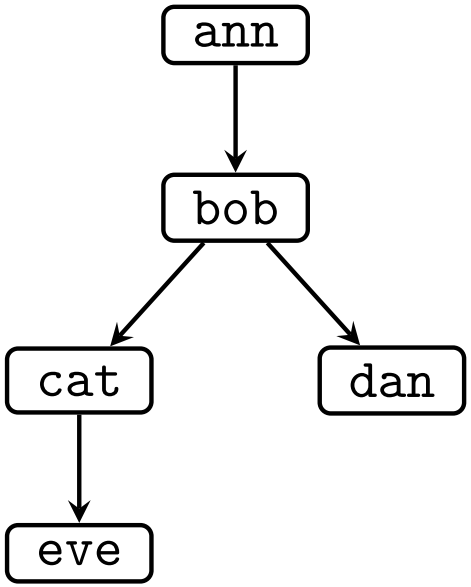

```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

- A parent is an ancestor.

A capital letter is a **variable**. A comma reads *and*.

# Datalog – Rules

A **rule** is `head :- body.` If the body holds, the head holds.




```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

- A parent is an ancestor.
- A parent of an ancestor is an ancestor.

A capital letter is a **variable**. A comma reads *and*.

# Applying a Rule

```
anc(X, Z)      :- parent(X, Y),      anc(Y, Z).
```

# Applying a Rule

```
anc(X, Z)      :- parent(X, Y),      anc(Y, Z).
```

Put `X = ann`, `Y = bob`, `Z = cat`:

```
anc(ann, cat) :- parent(ann, bob), anc(bob, cat).
```

# Applying a Rule

```
anc(X, Z)     :- parent(X, Y),     anc(Y, Z).
```

Put `X = ann`, `Y = bob`, `Z = cat`:

```
anc(ann, cat) :- parent(ann, bob), anc(bob, cat).
```

- We know `parent(ann, bob)` and `anc(bob, cat)`.
  So `anc(ann, cat)` is a **new fact**.

# Applying a Rule

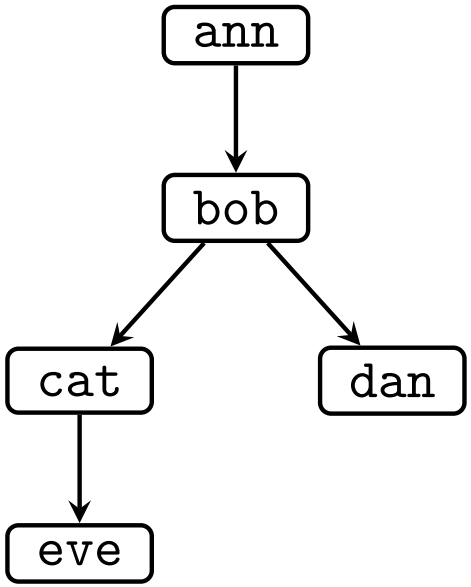
```
anc(X, Z)     :- parent(X, Y),     anc(Y, Z).
```

Put `X = ann`, `Y = bob`, `Z = cat`:

```
anc(ann, cat) :- parent(ann, bob), anc(bob, cat).
```

- We know `parent(ann, bob)` and `anc(bob, cat)`.
  So `anc(ann, cat)` is a **new fact**.
- A rule **fires** when some values for its variables make every part of the body a known fact.

# Facts Accumulate

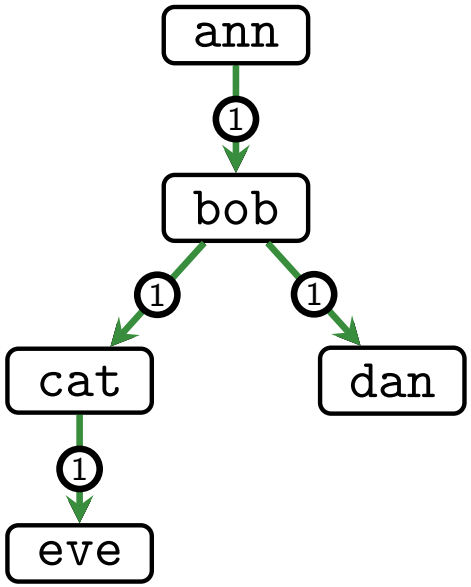



```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

| round | new facts |
|---|---|
| | |
| | |
| | |
| | |

Start with the four `parent` facts. In each **round**, fire every rule that can fire.

# Facts Accumulate

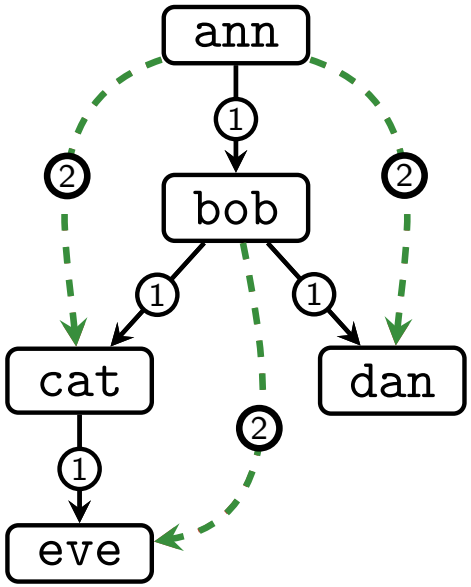



```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

| round | new facts |
|---|---|
| 1 | anc(ann, bob) anc(bob, cat)<br>anc(bob, dan) anc(cat, eve) |
| | |
| | |
| | |

Round 1. Only the first rule can fire: every parent is an ancestor.

# Facts Accumulate

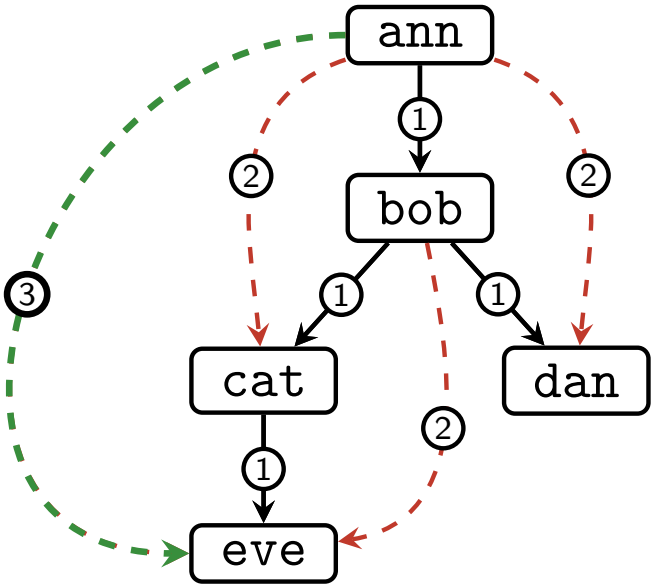



```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

| round | new facts |
|---|---|
| 1 | `anc(ann, bob)` `anc(bob, cat)` `anc(bob, dan)` `anc(cat, eve)` |
| 2 | `anc(ann, cat)` `anc(ann, dan)` `anc(bob, eve)` |
| | |
| | |

Round 2. `parent(ann, bob)` and `anc(bob, cat)` give `anc(ann, cat)`.
Two more alike.

# Facts Accumulate

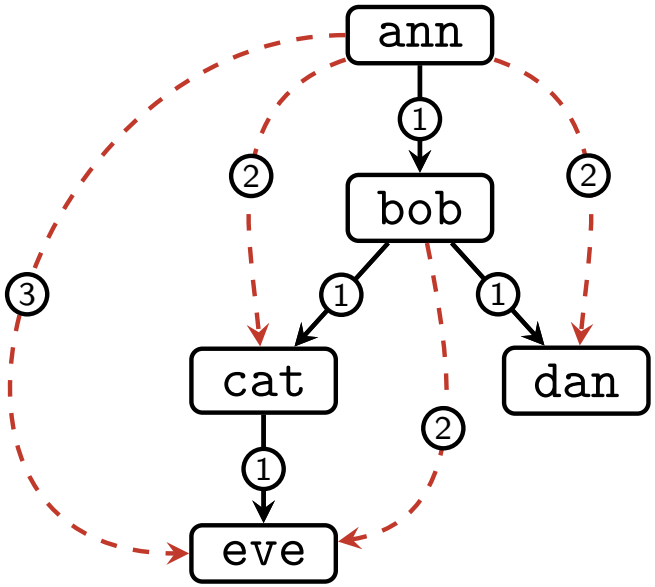



```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

| round | new facts |
|---|---|
| 1 | `anc(ann, bob)` `anc(bob, cat)` `anc(bob, dan)` `anc(cat, eve)` |
| 2 | `anc(ann, cat)` `anc(ann, dan)` `anc(bob, eve)` |
| 3 | `anc(ann, eve)` |
| | |

Round 3. `parent(ann, bob)` and `anc(bob, eve)`, new in round 2, give `anc(ann, eve)`.

# Facts Accumulate




```
anc(X, Y) :- parent(X, Y).
anc(X, Z) :- parent(X, Y), anc(Y, Z).
```

| round | new facts |
|---|---|
| 1 | `anc(ann, bob)` `anc(bob, cat)` `anc(bob, dan)` `anc(cat, eve)` |
| 2 | `anc(ann, cat)` `anc(ann, dan)` `anc(bob, eve)` |
| 3 | `anc(ann, eve)` |
| 4 | nothing new |

Round 4 gives nothing new: **stop**. The number on an arrow is its round.
The ancestors of `eve`: `cat`, `bob`, and `ann`.

# Why It Stops

- Rules only **add** facts. Nothing is ever taken back.

# Why It Stops

- Rules only **add** facts. Nothing is ever taken back.
- Rules cannot invent new names: every variable in the head comes from the body. So only **finitely many** facts can exist.

| | **possible facts** | **at most** | **derived** |
|---|---|---|---|
| `anc(X, Y)` | 5 names $\times$ 5 names | 25 | 8 |

# Why It Stops

- Rules only **add** facts. Nothing is ever taken back.
- Rules cannot invent new names: every variable in the head comes from the body. So only **finitely many** facts can exist.

| | **possible facts** | **at most** | **derived** |
|---|---|---|---|
| `anc(X, Y)` | $5$ names $\times$ $5$ names | $25$ | $8$ |

- So the rounds must stop, and they stop at the **same** set of facts whatever the order of the rules.

# Why It Stops

- Rules only **add** facts. Nothing is ever taken back.
- Rules cannot invent new names: every variable in the head comes from the body. So only **finitely many** facts can exist.

| | **possible facts** | **at most** | **derived** |
|---|---|---|---|
| `anc(X, Y)` | 5 names $\times$ 5 names | 25 | 8 |

- So the rounds must stop, and they stop at the **same** set of facts whatever the order of the rules.
- It is the **smallest** set that satisfies every rule: the **least solution**.

# Contents

1. Datalog

2. Pointer Analysis as Rules

3. More Analyses as Rules

4. In Practice

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Write the program down as **facts**, as we did for the family.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

A parameter points to whatever the caller passed: in_uid.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Allocation: `rows = [...]` at site $o_0$.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Allocation at site $o_1$, and `[uid]` stores `uid` into element `0`.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Copy: `view = rows`.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Store: `view[0] = row`.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Load: `r = rows[0]`.

# The Program as Facts

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

```
alloc(uid, in_uid).
alloc(rows, o0).
alloc(row, o1).
store(row, 0, uid).
copy(view, rows).
store(view, 0, row).
load(r, rows, 0).
```

Write the program down as **facts**, as we did for the family.

Four names, for the four kinds of statement: `alloc`, `copy`, `load`, `store`.

No analysis yet. This is only the program, as a table.

# The Constraints as Rules

| | | |
|---|---|---|
| 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 $o \in \mathsf{pt}(x)$ |

```
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

`pt(X, O)`: $O \in \mathsf{pt}(X)$ `hpt(O, K, P)`: $P \in \mathsf{pt}(O[K])$, **h**eap **p**oints-**t**o

The four constraints of Andersen's analysis, and the same four as rules.

# The Constraints as Rules

| 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 $o \in \mathsf{pt}(x)$ |

```
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

`pt(X, O)`: $O \in \mathsf{pt}(X)$ `hpt(O, K, P)`: $P \in \mathsf{pt}(O[K])$, **h**eap **p**oints-**t**o

If `X = [...]` allocates `O`, then `X` points to `O`.

# The Constraints as Rules

| | | |
|---|---|---|
| allocation | `x = [...]` | $o_i \in \text{pt}(x)$ |
| copy | `x = y` | $\text{pt}(y) \subseteq \text{pt}(x)$ |
| load | `x = y[k]` | $\text{pt}(o[k]) \subseteq \text{pt}(x)$ for each $o \in \text{pt}(y)$ |
| store | `x[k] = y` | $\text{pt}(y) \subseteq \text{pt}(o[k])$ for each $o \in \text{pt}(x)$ |

```
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

`pt(X, O)`: $O \in \text{pt}(X)$ `hpt(O, K, P)`: $P \in \text{pt}(O[K])$, **h**eap **p**oints-**t**o

If `X = Y`, and `Y` points to `O`, then `X` points to `O`.

# The Constraints as Rules

| | | |
|---|---|---|
| allocation | `x = [...]` | $o_i \in \text{pt}(x)$ |
| copy | `x = y` | $\text{pt}(y) \subseteq \text{pt}(x)$ |
| load | `x = y[k]` | $\text{pt}(o[k]) \subseteq \text{pt}(x)$ for each $o \in \text{pt}(y)$ |
| store | `x[k] = y` | $\text{pt}(y) \subseteq \text{pt}(o[k])$ for each $o \in \text{pt}(x)$ |

```
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

`pt(X, O)`: $O \in \text{pt}(X)$ `hpt(O, K, P)`: $P \in \text{pt}(O[K])$, **h**eap **p**oints-**t**o

If `X = Y[K]`, and `Y` points to `O`, and `O[K]` points to `P`,
then `X` points to `P`.

# The Constraints as Rules

| | | |
|---|---|---|
| allocation | `x = [...]` | $o_i \in \text{pt}(x)$ |
| copy | `x = y` | $\text{pt}(y) \subseteq \text{pt}(x)$ |
| load | `x = y[k]` | $\text{pt}(o[k]) \subseteq \text{pt}(x)$ for each $o \in \text{pt}(y)$ |
| store | `x[k] = y` | $\text{pt}(y) \subseteq \text{pt}(o[k])$ for each $o \in \text{pt}(x)$ |

```
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

`pt(X, O)`: $O \in \text{pt}(X)$ `hpt(O, K, P)`: $P \in \text{pt}(O[K])$, **h**eap **p**oints-**t**o

If `X[K] = Y`, and `X` points to `O`, and `Y` points to `P`,
then `O[K]` points to `P`.

# The Constraints as Rules

| | | |
|---|---|---|
| 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 $o \in \mathsf{pt}(x)$ |

```
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

`pt(X, O)`: $O \in \mathsf{pt}(X)$ `hpt(O, K, P)`: $P \in \mathsf{pt}(O[K])$, **h**eap **p**oints-**t**o

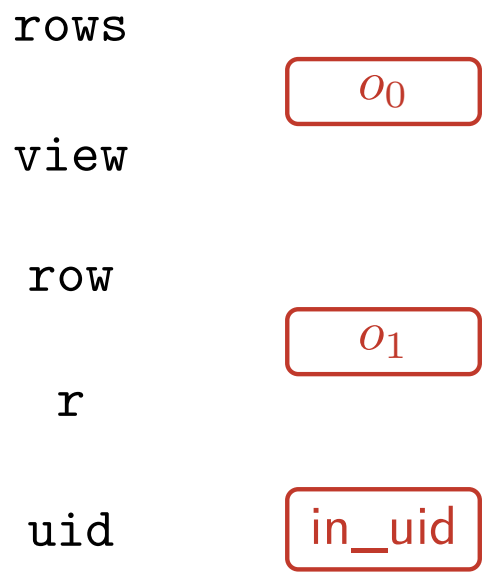
The four constraints are **four rules**.

`pt` and `hpt` are what we want to know, as `anc` was.

# Points-to, Step by Step

```
alloc(uid, in_uid).  alloc(rows, o0).
alloc(row, o1).      copy(view, rows).
store(row, 0, uid).  store(view, 0, row).
load(r, rows, 0).

pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

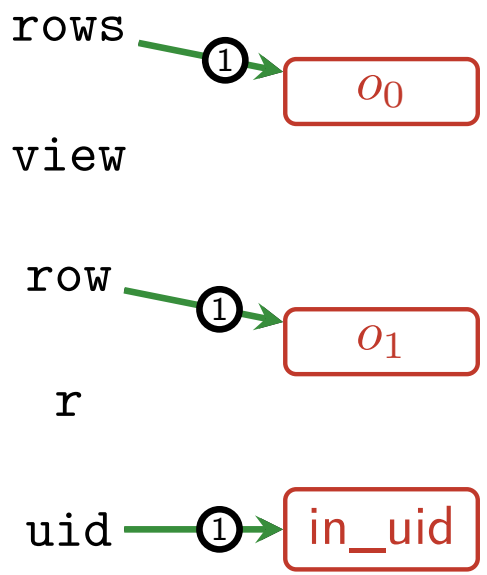



Start with the facts of the program, and fire the rules round by round, as before.

# Points-to, Step by Step

```
alloc(uid, in_uid).  alloc(rows, o0).
alloc(row, o1).      copy(view, rows).
store(row, 0, uid).  store(view, 0, row).
load(r, rows, 0).

pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

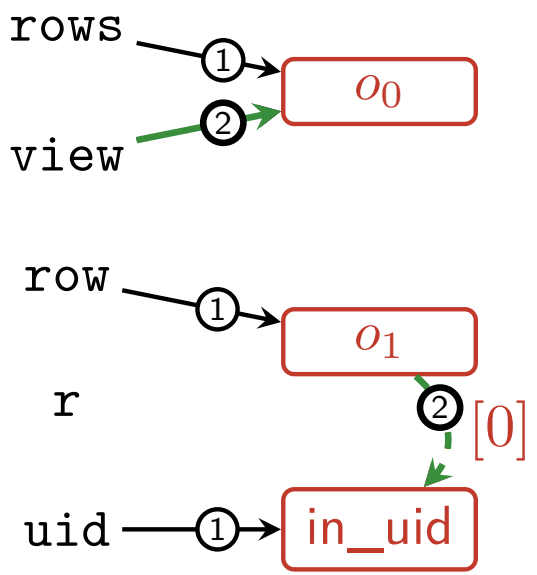



Round 1. Only the first rule can fire: `pt(uid, in_uid)`, `pt(rows, o0)`, `pt(row, o1)`.

# Points-to, Step by Step

```
alloc(uid, in_uid).  alloc(rows, o0).
alloc(row, o1).      copy(view, rows).
store(row, 0, uid).  store(view, 0, row).
load(r, rows, 0).

pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

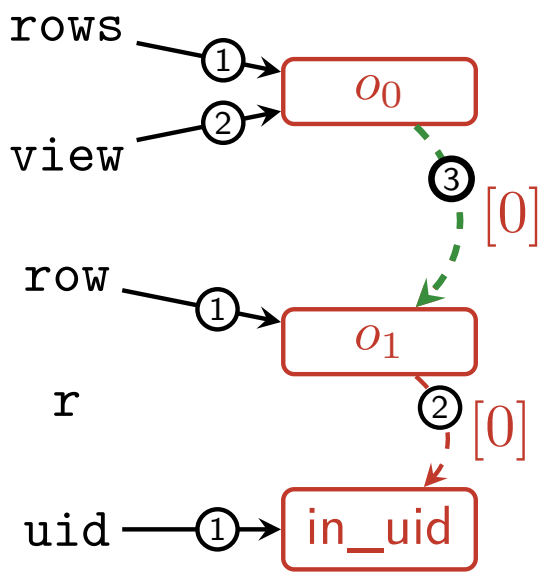



Round 2. `copy(view, rows)` and `pt(rows, o0)` give `pt(view, o0)`. `store(row, 0, uid)` gives `hpt(o1, 0, in_uid)`.

# Points-to, Step by Step

```
alloc(uid, in_uid).  alloc(rows, o0).
alloc(row, o1).      copy(view, rows).
store(row, 0, uid).  store(view, 0, row).
load(r, rows, 0).

pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

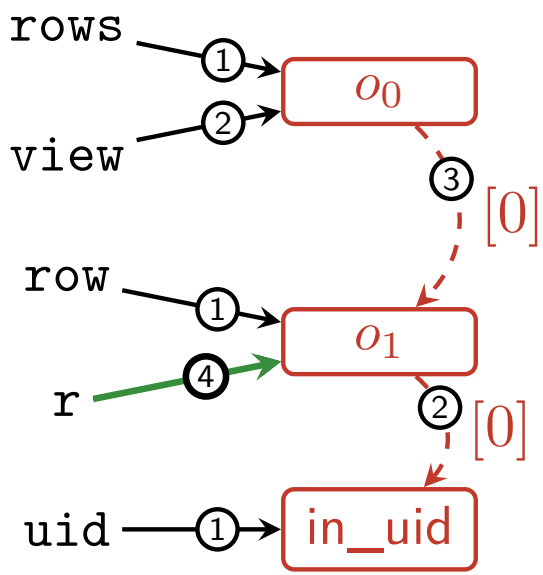



Round 3. `store(view, 0, row)` was waiting for `pt(view, o0)`: now it gives `hpt(o0, 0, o1)`.

# Points-to, Step by Step

```
alloc(uid, in_uid).  alloc(rows, o0).
alloc(row, o1).      copy(view, rows).
store(row, 0, uid).  store(view, 0, row).
load(r, rows, 0).

pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

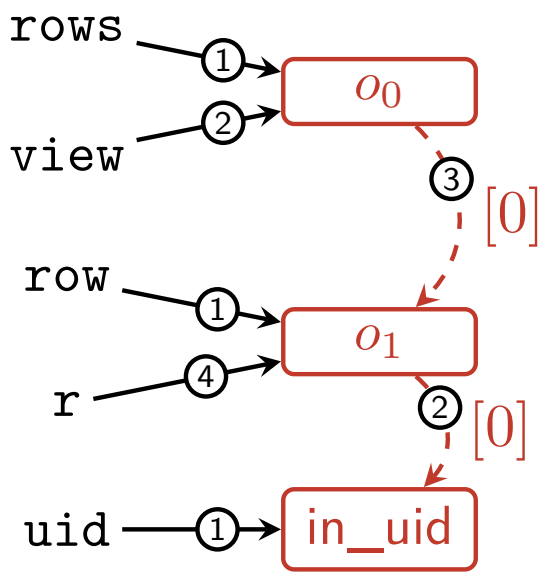



Round 4. `load(r, rows, 0)`, `pt(rows, o0)`, and `hpt(o0, 0, o1)` give `pt(r, o1)`.

# Points-to, Step by Step

```
alloc(uid, in_uid).  alloc(rows, o0).
alloc(row, o1).      copy(view, rows).
store(row, 0, uid).  store(view, 0, row).
load(r, rows, 0).

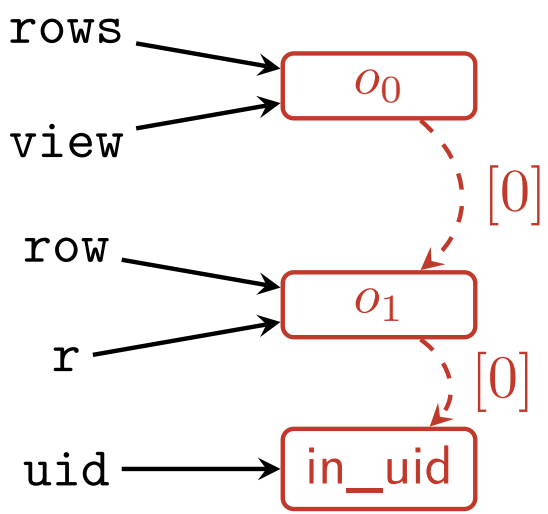
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```




Round 5 gives nothing new: **stop**. The number on an arrow is its round.
The same graph as Andersen's loop, and we wrote **no loop**.

# A New Analysis in One Line

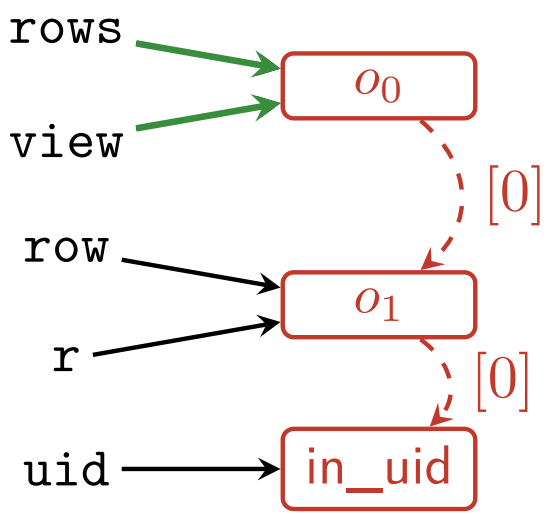
```
alias(X, Y) :- pt(X, O), pt(Y, O).
```




- **May alias**: `X` and `Y` may point to one object.

# A New Analysis in One Line

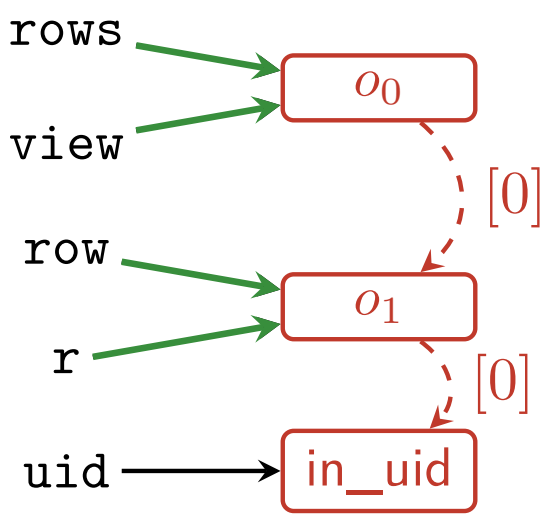
```
alias(X, Y) :- pt(X, O), pt(Y, O).
```




- **May alias**: `X` and `Y` may point to one object.
- `rows` and `view` point to $o_0$: `alias(rows, view)`.

# A New Analysis in One Line

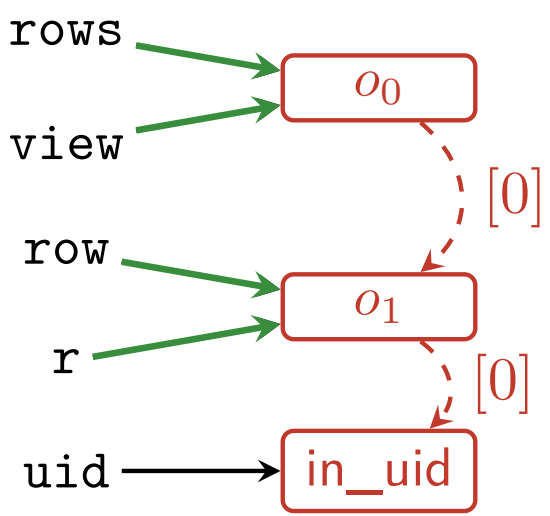
```
alias(X, Y) :- pt(X, O), pt(Y, O).
```




- **May alias**: `X` and `Y` may point to one object.
- `rows` and `view` point to $o_0$: `alias(rows, view)`.
- `row` and `r` point to $o_1$: `alias(row, r)`.

# A New Analysis in One Line

```
alias(X, Y) :- pt(X, O), pt(Y, O).
```




- **May alias**: `X` and `Y` may point to one object.
- `rows` and `view` point to $o_0$: `alias(rows, view)`.
- `row` and `r` point to $o_1$: `alias(row, r)`.
- One more rule, on top of `pt`. The engine does not change.

# Contents

1. Datalog

2. Pointer Analysis as Rules

3. More Analyses as Rules

4. In Practice

# Taint as Rules

```
tainted(X)      :- source(X).
tainted(X)      :- copy(X, Y), tainted(Y).
tainted(X)      :- binop(X, Y), tainted(Y).
tainted(X)      :- load(X, Y, K), pt(Y, O), htainted(O, K).
htainted(O, K) :- store(X, K, Y), pt(X, O), tainted(Y).
alarm(X)        :- sink(X), tainted(X).
```

A **source** is tainted. In MiniPy, the sources are the parameters.

# Taint as Rules

```
tainted(X)      :- source(X).
tainted(X)      :- copy(X, Y), tainted(Y).
tainted(X)      :- binop(X, Y), tainted(Y).
tainted(X)      :- load(X, Y, K), pt(Y, O), htainted(O, K).
htainted(O, K) :- store(X, K, Y), pt(X, O), tainted(Y).
alarm(X)        :- sink(X), tainted(X).
```

Taint moves from variable to variable. `binop(X, Y)`: `X` is computed from `Y`.

# Taint as Rules

```
tainted(X)      :- source(X).
tainted(X)      :- copy(X, Y), tainted(Y).
tainted(X)      :- binop(X, Y), tainted(Y).
tainted(X)      :- load(X, Y, K), pt(Y, O), htainted(O, K).
htainted(O, K) :- store(X, K, Y), pt(X, O), tainted(Y).
alarm(X)        :- sink(X), tainted(X).
```

Through a list, taint sits on the **object**: `htainted(O, K)` is element `K` of `O`.

Which object? `pt` says, and `pt` comes from the four rules we have.

# Taint as Rules

```
tainted(X)      :- source(X).
tainted(X)      :- copy(X, Y), tainted(Y).
tainted(X)      :- binop(X, Y), tainted(Y).
tainted(X)      :- load(X, Y, K), pt(Y, O), htainted(O, K).
htainted(O, K)  :- store(X, K, Y), pt(X, O), tainted(Y).
alarm(X)        :- sink(X), tainted(X).
```

A tainted **sink** is an alarm. In MiniPy, the sink is the returned value.

# Taint as Rules

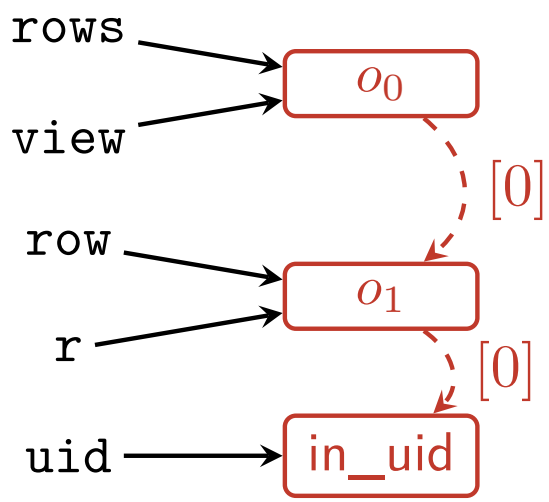
```
tainted(X)      :- source(X).
tainted(X)      :- copy(X, Y), tainted(Y).
tainted(X)      :- binop(X, Y), tainted(Y).
tainted(X)      :- load(X, Y, K), pt(Y, O), htainted(O, K).
htainted(O, K) :- store(X, K, Y), pt(X, O), tainted(Y).
alarm(X)        :- sink(X), tainted(X).
```

Six more rules, on top of the four for `pt`.

Through a list, taint sits on the **object**: `htainted(O, K)` is element `K` of `O`. Every name that points to `O` sees it. Rules **build on** rules.

# Taint, Step by Step

```
def f(uid):          # source
    rows = [0]       # o0
    row = [uid]      # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]      # sink: t0 = r[0]
```

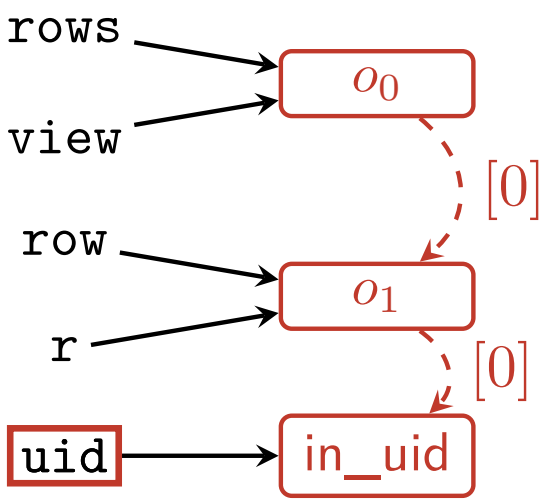



| new fact | because |
|---|---|
| | |
| | |
| | |
| | |

New facts: `source(uid)`, and `load(t0, r, 0)`, `sink(t0)` for the returned `r[0]`. It has no name, so we give it one: `t0`.

# Taint, Step by Step

```
def f(uid):          # source
    rows = [0]       # o0
    row = [uid]      # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]      # sink: t0 = r[0]
```

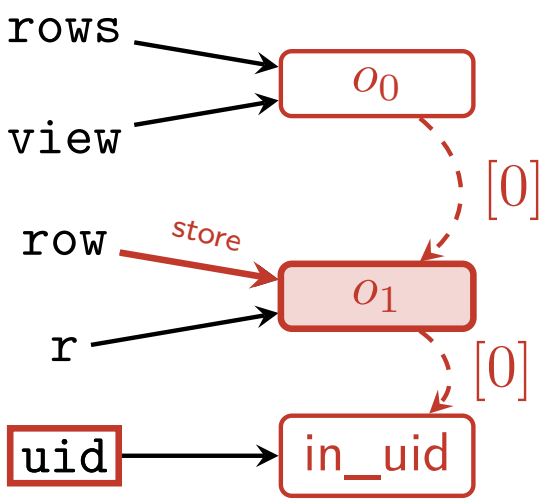



| new fact | because |
|---|---|
| `tainted(uid)` | `source(uid)` |
| | |
| | |
| | |

The parameter is a source.

# Taint, Step by Step

```
def f(uid):          # source
    rows = [0]       # o0
    row = [uid]      # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]      # sink: t0 = r[0]
```

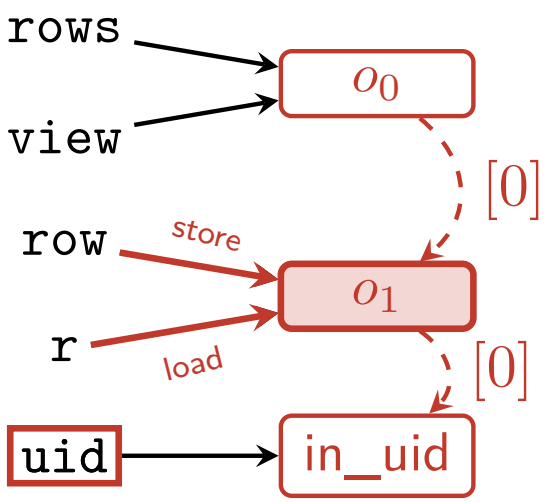



| new fact | because |
|---|---|
| `tainted(uid)` | `source(uid)` |
| `htainted(o1, 0)` | `store(row, 0, uid)`, `pt(row, o1)`, `tainted(uid)` |
| | |
| | |

`[uid]` stores `uid` through `row`, which points to $o_1$: element 0 of $o_1$ is tainted.

# Taint, Step by Step

```
def f(uid):          # source
    rows = [0]       # o0
    row = [uid]      # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]      # sink: t0 = r[0]
```

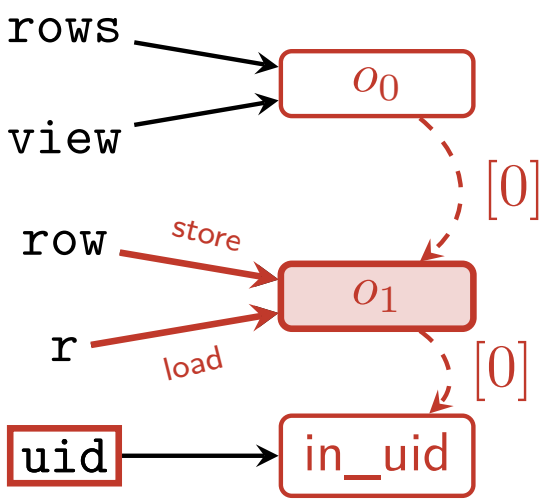



| new fact | because |
|---|---|
| `tainted(uid)` | `source(uid)` |
| `htainted(o1, 0)` | `store(row, 0, uid)`, `pt(row, o1)`, `tainted(uid)` |
| `tainted(t0)` | `load(t0, r, 0)`, `pt(r, o1)`, `htainted(o1, 0)` |
| | |

`r[0]` loads through `r`, which points to $o_1$ too: `t0` is tainted.

# Taint, Step by Step

```
def f(uid):          # source
    rows = [0]       # o0
    row = [uid]      # o1
    view = rows
    view[0] = row
    r = rows[0]
    return r[0]      # sink: t0 = r[0]
```

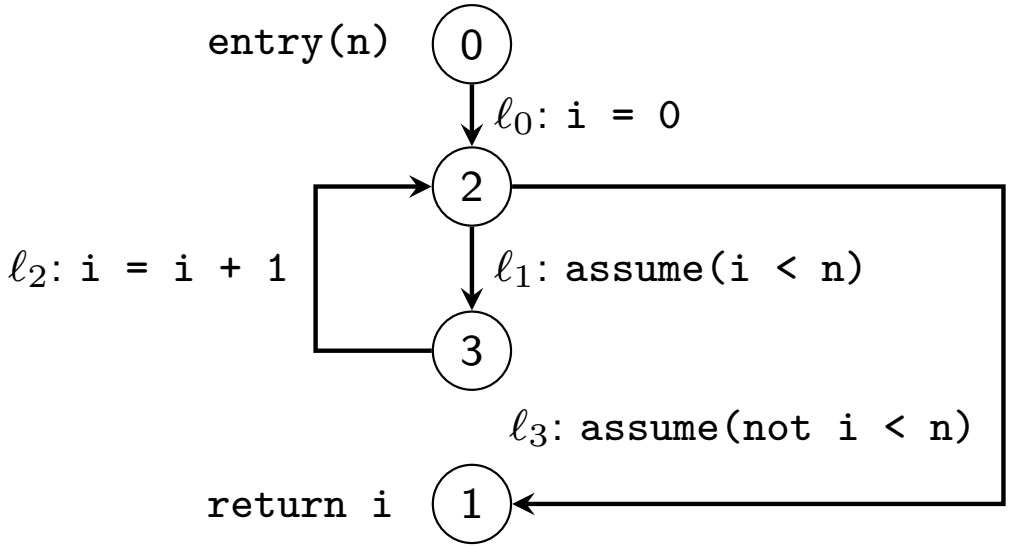



| new fact | because |
|---|---|
| `tainted(uid)` | `source(uid)` |
| `htainted(o1, 0)` | `store(row, 0, uid)`, `pt(row, o1)`, `tainted(uid)` |
| `tainted(t0)` | `load(t0, r, 0)`, `pt(r, o1)`, `htainted(o1, 0)` |
| `alarm(t0)` | `sink(t0)`, `tainted(t0)` |

Red: taint goes into $o_1$ through `row`, and comes out through `r`.
The earlier taint analysis has no `pt`: it misses this leak.

# The CFG as Facts

**Reaching definitions**: which assignments may reach each point?

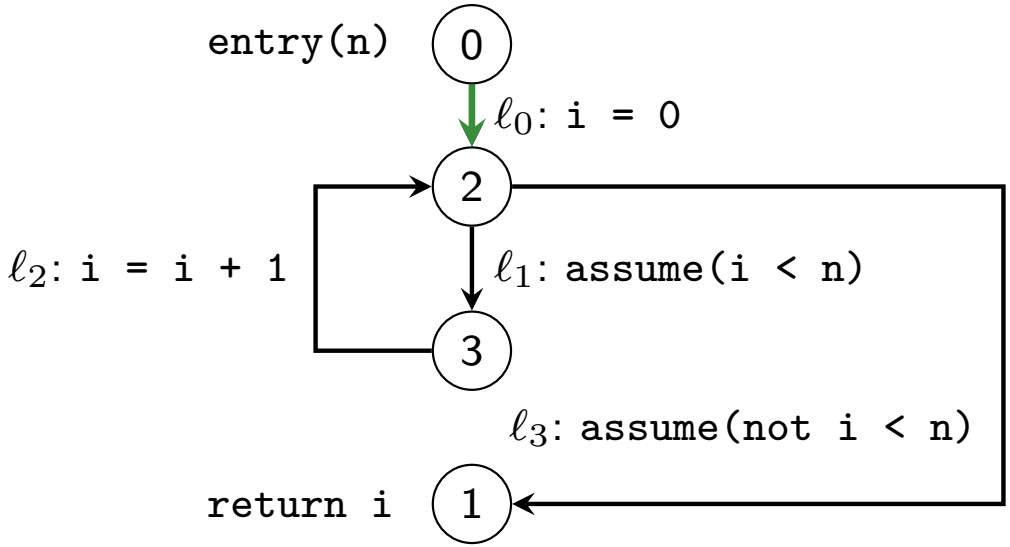



```
edge(0, 0, 2).  def(0, i).
edge(1, 2, 3).
edge(2, 3, 2).  def(2, i).
edge(3, 2, 1).
reach(0, n, -1).
```

Before, a worklist over the CFG. Now, the CFG as facts.

# The CFG as Facts

**Reaching definitions**: which assignments may reach each point?

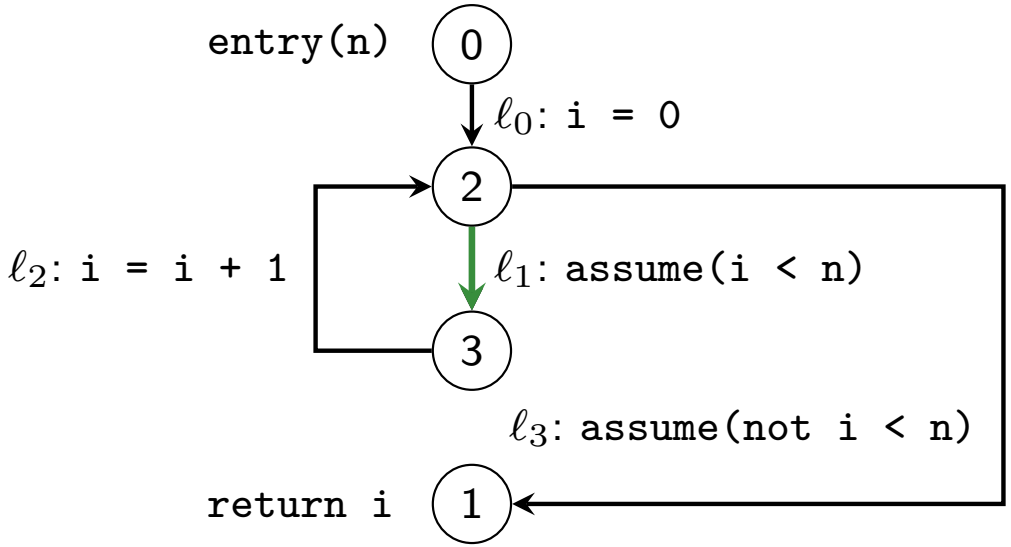



```
edge(0, 0, 2).  def(0, i).
edge(1, 2, 3).
edge(2, 3, 2).  def(2, i).
edge(3, 2, 1).
reach(0, n, -1).
```

Edge $0$ goes from point $0$ to point $2$, and assigns `i`.

# The CFG as Facts

**Reaching definitions**: which assignments may reach each point?

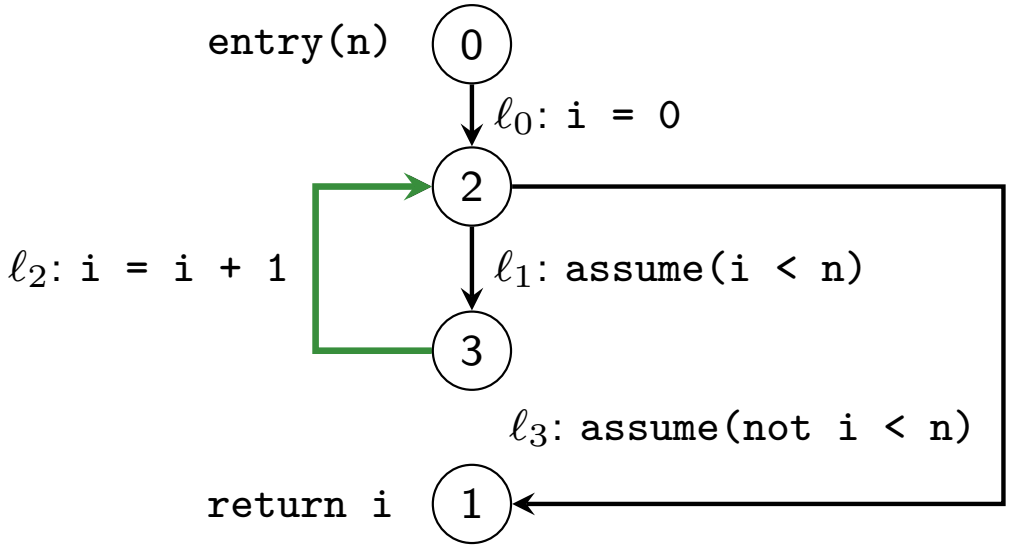



```
edge(0, 0, 2).  def(0, i).
edge(1, 2, 3).
edge(2, 3, 2).  def(2, i).
edge(3, 2, 1).
reach(0, n, -1).
```

Edge $1$ goes from point $2$ to point $3$. It assigns nothing.

# The CFG as Facts

**Reaching definitions**: which assignments may reach each point?

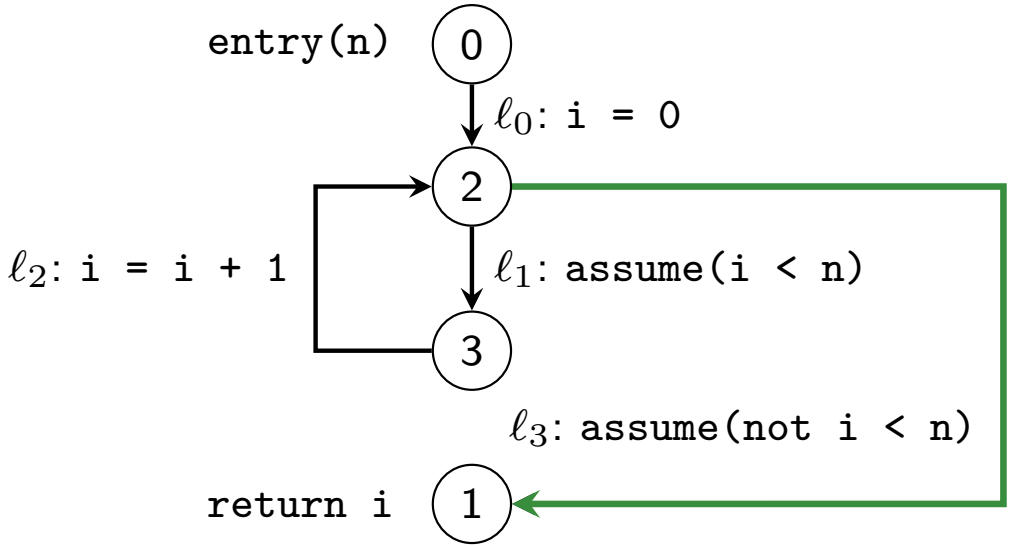



```
edge(0, 0, 2).  def(0, i).
edge(1, 2, 3).
edge(2, 3, 2).  def(2, i).
edge(3, 2, 1).
reach(0, n, -1).
```

Edge $2$ goes from point $3$ back to point $2$, and assigns `i`.

# The CFG as Facts

**Reaching definitions**: which assignments may reach each point?

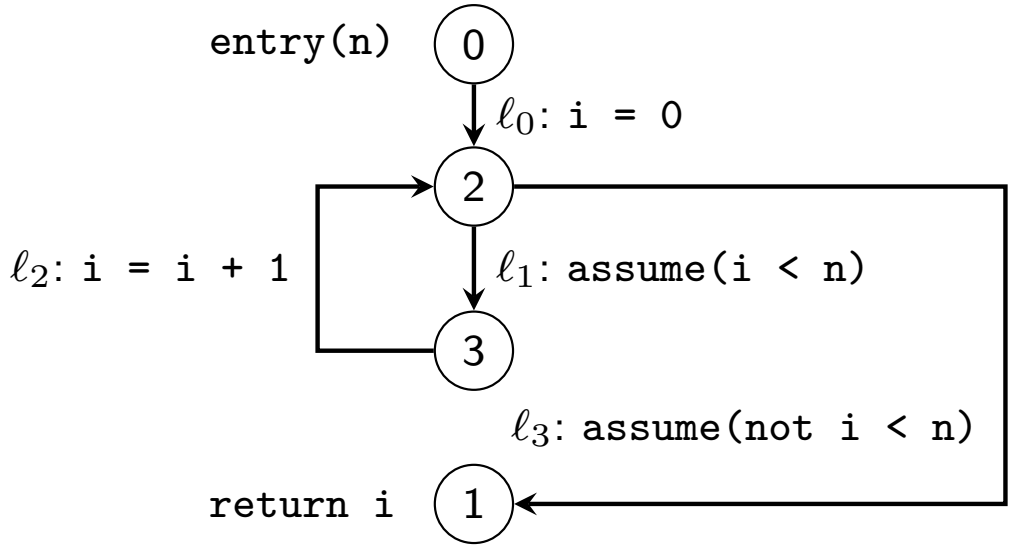



```
edge(0, 0, 2).  def(0, i).
edge(1, 2, 3).
edge(2, 3, 2).  def(2, i).
edge(3, 2, 1).
reach(0, n, -1).
```

Edge $3$ goes from point $2$ to point $1$. It assigns nothing.

# The CFG as Facts

**Reaching definitions**: which assignments may reach each point?




```
edge(0, 0, 2).  def(0, i).
edge(1, 2, 3).
edge(2, 3, 2).  def(2, i).
edge(3, 2, 1).
reach(0, n, -1).
```

`edge(E, P, Q)`: edge `E` goes from `P` to `Q`. `def(E, X)`: edge `E` assigns `X`.
`reach(0, n, -1)`: at the entry, `n` is a parameter, assigned by no edge.

# Reaching Definitions as Rules

```
reach(Q, X, E) :- edge(E, P, Q), def(E, X).
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

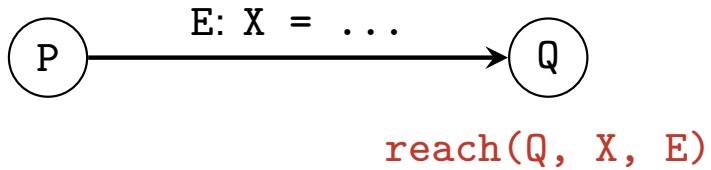
`reach(Q, X, D)`: the assignment to `X` on edge `D` reaches point `Q`.

# Reaching Definitions as Rules

```
reach(Q, X, E) :- edge(E, P, Q), def(E, X).
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

`reach(Q, X, D)`: the assignment to `X` on edge `D` reaches point `Q`.

- **Gen**: the edge assigns `X`.
  Edge `0` gives `reach(2, i, 0)`.

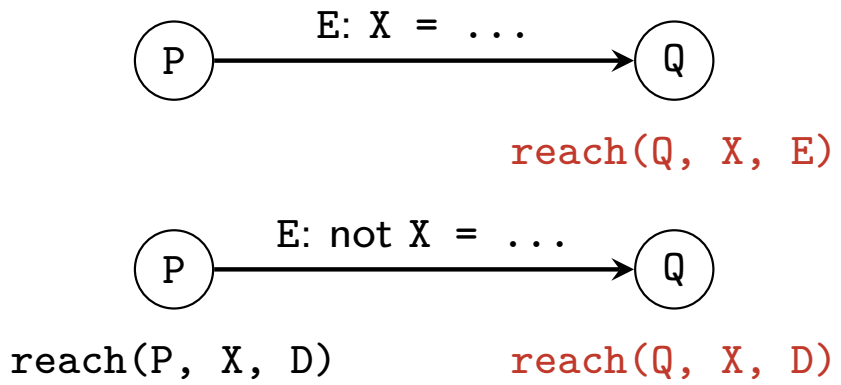


# Reaching Definitions as Rules

```
reach(Q, X, E) :- edge(E, P, Q), def(E, X).
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

`reach(Q, X, D)`: the assignment to `X` on edge `D` reaches point `Q`.

- **Gen**: the edge assigns `X`.
  Edge $0$ gives `reach(2, i, 0)`.
- **Pass**: the edge does *not* assign `X`.
  Edge $1$ gives `reach(3, i, 0)`.

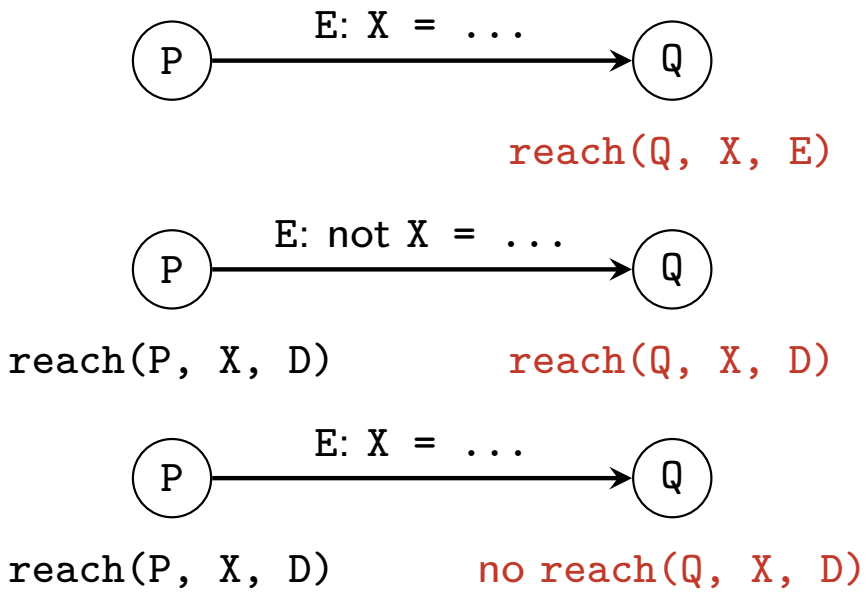


# Reaching Definitions as Rules

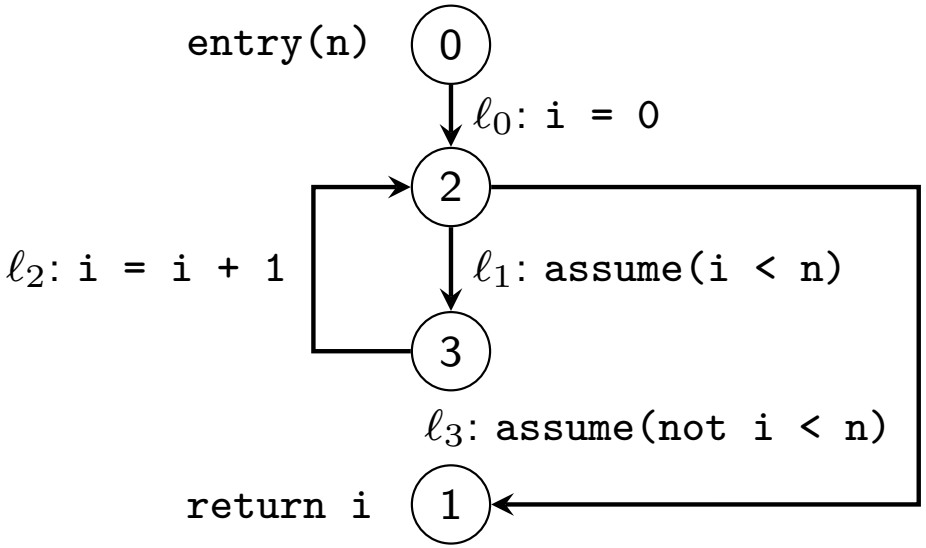
```
reach(Q, X, E) :- edge(E, P, Q), def(E, X).
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

`reach(Q, X, D)`: the assignment to `X` on edge `D` reaches point `Q`.

- **Gen**: the edge assigns `X`.
  Edge $0$ gives `reach(2, i, 0)`.
- **Pass**: the edge does *not* assign `X`.
  Edge $1$ gives `reach(3, i, 0)`.
- **Kill**: the `!`, which reads *not*.
  Edge $2$ stops `reach(3, i, 0)`.



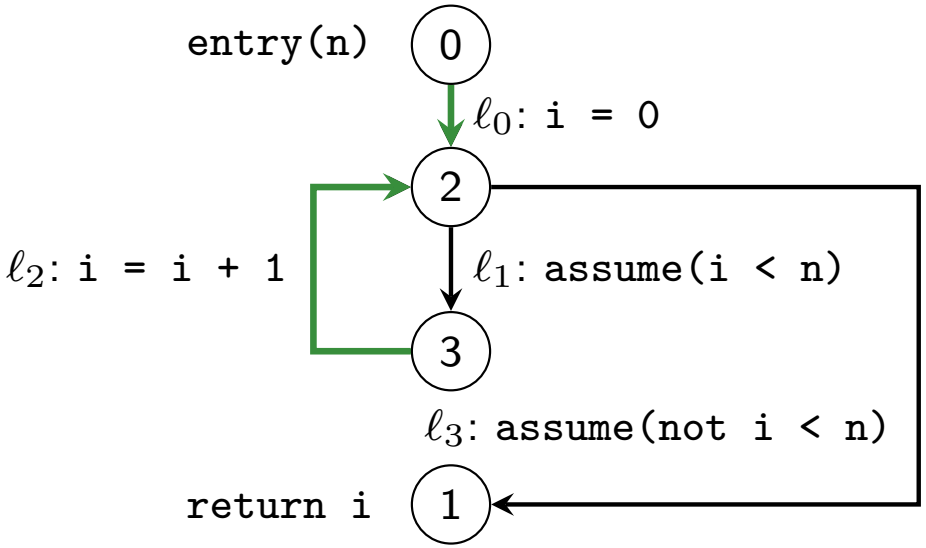
# Reaching Definitions, Step by Step




| round | new facts |
|---|---|
| | |
| | |
| | |

Start with the facts of the CFG, and fire the two rules round by round.

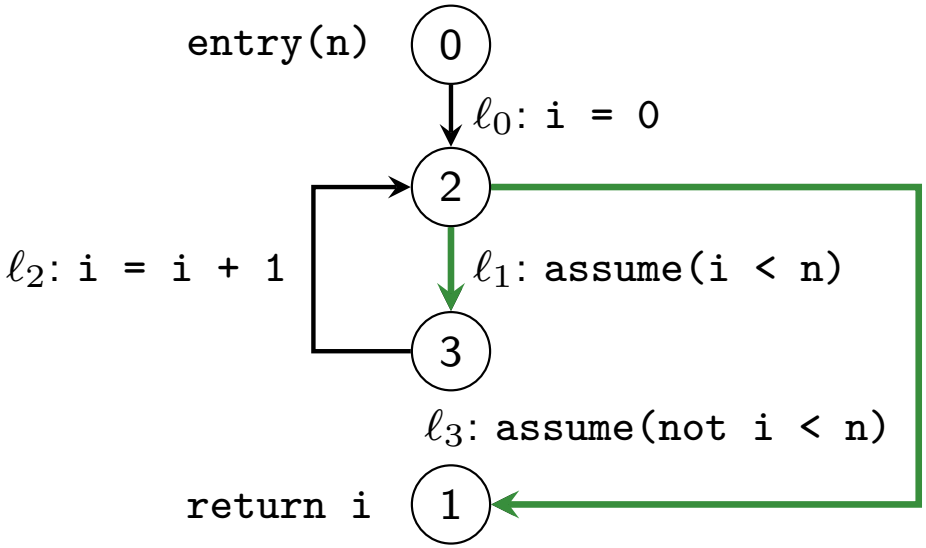
# Reaching Definitions, Step by Step




| round | new facts |
|---|---|
| 1 | reach(2, i, 0) reach(2, i, 2) reach(2, n, -1) |
| | |
| | |

Round 1. Gen: edges $0$ and $2$ assign `i`. Pass: `n` crosses edge $0$.

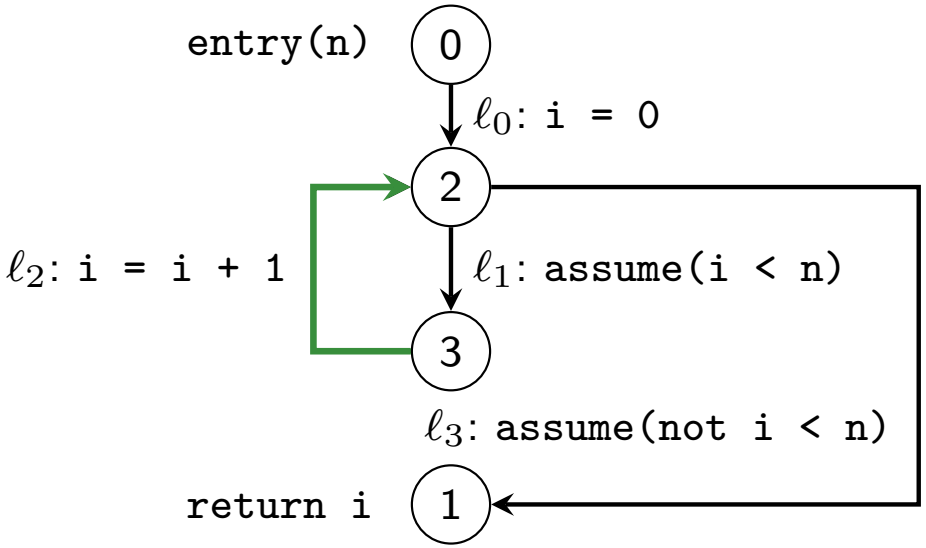
# Reaching Definitions, Step by Step




| round | new facts |
|---|---|
| 1 | reach(2, i, 0) reach(2, i, 2) reach(2, n, -1) |
| 2 | reach(3, i, 0) reach(1, i, 0) reach(3, i, 2) reach(1, i, 2) reach(3, n, -1) reach(1, n, -1) |
| | |

Round 2. Edges $1$ and $3$ assign nothing: everything at point $2$ crosses them.

# Reaching Definitions, Step by Step




| round | new facts |
|---|---|
| 1 | reach(2, i, 0) reach(2, i, 2) reach(2, n, -1) |
| 2 | reach(3, i, 0) reach(1, i, 0) reach(3, i, 2) reach(1, i, 2) reach(3, n, -1) reach(1, n, -1) |
| 3 | nothing new |

Round 3: nothing new. Edge $2$ assigns `i`, so no `i` crosses it.
At point $2$: `n` from the entry, `i` from edges $0$ and $2$, as the worklist gave.

# Negation

```
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

- `!def(2, n)` holds because there is **no** fact `def(2, n)`.

# Negation

```
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

- `!def(2, n)` holds because there is **no** fact `def(2, n)`.
- Safe here: `def` comes straight from the program. All of it is known before the first round.

# Negation

```
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

- `!def(2, n)` holds because there is **no** fact `def(2, n)`.
- Safe here: `def` comes straight from the program. All of it is known before the first round.
- Not safe on a relation still being derived: the missing fact may appear in a later round, and rules never take a fact back.

# Negation

```
reach(Q, X, D) :- edge(E, P, Q), reach(P, X, D), !def(E, X).
```

- `!def(2, n)` holds because there is **no** fact `def(2, n)`.
- Safe here: `def` comes straight from the program. All of it is known before the first round.
- Not safe on a relation still being derived: the missing fact may appear in a later round, and rules never take a fact back.
- So negate only what is already **finished**. Real engines order the relations in layers for this: **stratified negation**.

# Sensitivity Is a Column

Each kind of sensitivity adds one argument to a relation:

| **sensitivity** | **without** | **with** |
|---|---|---|
| field | `hpt(O, P)` | `hpt(O, K, P)` |
| | | |
| | | |

# Sensitivity Is a Column

Each kind of sensitivity adds one argument to a relation:

| **sensitivity** | **without** | **with** |
|---|---|---|
| field | `hpt(O, P)` | `hpt(O, K, P)` |
| flow | `pt(X, O)` | `pt(Q, X, O)` at point `Q` |
| | | |

# Sensitivity Is a Column

Each kind of sensitivity adds one argument to a relation:

| **sensitivity** | **without** | **with** |
|---|---|---|
| field | `hpt(O, P)` | `hpt(O, K, P)` |
| flow | `pt(X, O)` | `pt(Q, X, O)` at point `Q` |
| context | `pt(X, O)` | `pt(C, X, O)` in context `C` |

# Sensitivity Is a Column

Each kind of sensitivity adds one argument to a relation:

| **sensitivity** | **without** | **with** |
|---|---|---|
| field | `hpt(O, P)` | `hpt(O, K, P)` |
| flow | `pt(X, O)` | `pt(Q, X, O)` at point `Q` |
| context | `pt(X, O)` | `pt(C, X, O)` in context `C` |

- More columns, more facts: the cost of precision is the **size of the table**.

# Sensitivity Is a Column

Each kind of sensitivity adds one argument to a relation:

| **sensitivity** | **without** | **with** |
|---|---|---|
| field | `hpt(O, P)` | `hpt(O, K, P)` |
| flow | `pt(X, O)` | `pt(Q, X, O)` at point `Q` |
| context | `pt(X, O)` | `pt(C, X, O)` in context `C` |

- More columns, more facts: the cost of precision is the **size of the table**.
- **Doop** is built this way: pointer analyses for Java, each a set of Datalog rules. Its `-a 2-object-sensitive+heap` picks one set.

# Contents

1. Datalog

2. Pointer Analysis as Rules

3. More Analyses as Rules

4. In Practice

# Real Engines

The same four rules in **Soufflé**:

```
.decl alloc(x: symbol, o: symbol)
.decl copy(x: symbol, y: symbol)
.decl load(x: symbol, y: symbol, k: symbol)
.decl store(x: symbol, k: symbol, y: symbol)
.decl pt(x: symbol, o: symbol)
.decl hpt(o: symbol, k: symbol, p: symbol)
.input alloc
.input copy
.input load
.input store
.output pt
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

# Real Engines

PLRG

The same four rules in **Soufflé**:

```
.decl alloc(x: symbol, o: symbol)
.decl copy(x: symbol, y: symbol)
.decl load(x: symbol, y: symbol, k: symbol)
.decl store(x: symbol, k: symbol, y: symbol)
.decl pt(x: symbol, o: symbol)
.decl hpt(o: symbol, k: symbol, p: symbol)
.input alloc
.input copy
.input load
.input store
.output pt
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

- Soufflé compiles rules to parallel C++. Doop runs on it.

# Real Engines

The same four rules in **Soufflé**:

```
.decl alloc(x: symbol, o: symbol)
.decl copy(x: symbol, y: symbol)
.decl load(x: symbol, y: symbol, k: symbol)
.decl store(x: symbol, k: symbol, y: symbol)
.decl pt(x: symbol, o: symbol)
.decl hpt(o: symbol, k: symbol, p: symbol)
.input alloc
.input copy
.input load
.input store
.output pt
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

- Soufflé compiles rules to parallel C++. Doop runs on it.
- **CodeQL**, behind GitHub code scanning: Datalog with classes.

# Real Engines

PLRG

The same four rules in **Soufflé**:

```
.decl alloc(x: symbol, o: symbol)
.decl copy(x: symbol, y: symbol)
.decl load(x: symbol, y: symbol, k: symbol)
.decl store(x: symbol, k: symbol, y: symbol)
.decl pt(x: symbol, o: symbol)
.decl hpt(o: symbol, k: symbol, p: symbol)
.input alloc
.input copy
.input load
.input store
.output pt
pt(X, O)      :- alloc(X, O).
pt(X, O)      :- copy(X, Y), pt(Y, O).
pt(X, P)      :- load(X, Y, K), pt(Y, O), hpt(O, K, P).
hpt(O, K, P) :- store(X, K, Y), pt(X, O), pt(Y, P).
```

- Soufflé compiles rules to parallel C++. Doop runs on it.
- **CodeQL**, behind GitHub code scanning: Datalog with classes.
- Engines fire rules only on the **new** facts of a round: a worklist.

# A CodeQL Query

PLRG

Does user input reach a file path? For real Python, in **CodeQL**:

```
import python
import semmle.python.dataflow.new.TaintTracking    // and three more

module RemoteToFileConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}
module RemoteToFileFlow = TaintTracking::Global<RemoteToFileConfiguration>;

from DataFlow::Node input, DataFlow::Node fileAccess
where RemoteToFileFlow::flow(input, fileAccess)
select fileAccess, "This file access uses data from $@.", input, "user input."
```

An example from the CodeQL documentation. We write three things.

# A CodeQL Query

Does user input reach a file path? For real Python, in **CodeQL**:

```
import python
import semmle.python.dataflow.new.TaintTracking    // and three more

module RemoteToFileConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}
module RemoteToFileFlow = TaintTracking::Global<RemoteToFileConfiguration>;

from DataFlow::Node input, DataFlow::Node fileAccess
where RemoteToFileFlow::flow(input, fileAccess)
select fileAccess, "This file access uses data from $@.", input, "user input."
```

The **sources**: every input from the network.

# A CodeQL Query

PLRG

Does user input reach a file path? For real Python, in **CodeQL**:

```
import python
import semmle.python.dataflow.new.TaintTracking    // and three more

module RemoteToFileConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}
module RemoteToFileFlow = TaintTracking::Global<RemoteToFileConfiguration>;

from DataFlow::Node input, DataFlow::Node fileAccess
where RemoteToFileFlow::flow(input, fileAccess)
select fileAccess, "This file access uses data from $@.", input, "user input."
```

The **sinks**: every argument that is used as a file path.

# A CodeQL Query

PLRG

Does user input reach a file path? For real Python, in **CodeQL**:

```
import python
import semmle.python.dataflow.new.TaintTracking    // and three more

module RemoteToFileConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}
module RemoteToFileFlow = TaintTracking::Global<RemoteToFileConfiguration>;

from DataFlow::Node input, DataFlow::Node fileAccess
where RemoteToFileFlow::flow(input, fileAccess)
select fileAccess, "This file access uses data from $@.", input, "user input."
```

The **rules** of taint, for all of Python, come from the library.

# A CodeQL Query

PLRG

Does user input reach a file path? For real Python, in **CodeQL**:

```
import python
import semmle.python.dataflow.new.TaintTracking    // and three more

module RemoteToFileConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}
module RemoteToFileFlow = TaintTracking::Global<RemoteToFileConfiguration>;

from DataFlow::Node input, DataFlow::Node fileAccess
where RemoteToFileFlow::flow(input, fileAccess)
select fileAccess, "This file access uses data from $@.", input, "user input."
```

The **query**: every source that flows to a sink. It reads like SQL.

# A CodeQL Query

PLRG

Does user input reach a file path? For real Python, in **CodeQL**:

```
import python
import semmle.python.dataflow.new.TaintTracking    // and three more

module RemoteToFileConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}
module RemoteToFileFlow = TaintTracking::Global<RemoteToFileConfiguration>;

from DataFlow::Node input, DataFlow::Node fileAccess
where RemoteToFileFlow::flow(input, fileAccess)
select fileAccess, "This file access uses data from $@.", input, "user input."
```

We write the **sources**, the **sinks**, and the **query**. The rules of taint, with pointers and calls, for all of Python, come from the library.

# Summary

1. Datalog
2. Pointer Analysis as Rules
3. More Analyses as Rules
4. In Practice

# Next Lecture

- Symbolic Execution

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