

# Static Program Analysis

Yue Li and Tian Tan



2020 Spring

# Static Program Analysis

## Datalog-Based Program Analysis

Nanjing University

Tian Tan

2020

# Contents

The background features a light blue sky with soft clouds. In the center, a large, dark, muscular statue of a man with a beard and a raised right hand stands behind a decorative metal railing. To the left, a girl with long dark hair, wearing a brown cap and a white shirt, leans on the railing eating a red lollipop. In the foreground, a boy with dark hair and a white shirt sits on the railing, holding a smartphone. To the right, a boy with long brown hair, wearing a brown t-shirt with a white stripe and a dark cap, stands with his hands raised in a gesture. Further right, a girl with long blue hair and a black top stands holding a smartphone. The overall scene is a peaceful outdoor setting, likely a park or a university campus.

1. Motivation
2. Introduction to Datalog
3. Pointer Analysis via Datalog
4. Taint Analysis via Datalog

# Contents

The background features a faded illustration of several anime-style characters. In the upper left, a character with dark hair and a brown cap is eating. In the center, a large, muscular, grey statue of a man with a beard stands on a pedestal. In the foreground, a character with long brown hair and a white shirt is sitting on a bench, holding a smartphone. To the right, a character with long blue hair and a black shirt is standing and holding a smartphone. The scene is set outdoors with a blue sky and green trees.

- 1. Motivation**
2. Introduction to Datalog
3. Pointer Analysis via Datalog
4. Taint Analysis via Datalog

# Imperative vs Declarative

Goal: select adults from a set of persons

# Imperative vs Declarative

Goal: select adults from a set of persons

- Imperative: how to do (~implementation)

```
Set<Person> selectAdults(Set<Person> persons) {  
    Set<Person> result = new HashSet<>();  
    for (Person person : persons)  
        if (person.getAge() >= 18)  
            result.add(person);  
    return result;  
}
```

# Imperative vs Declarative

Goal: select adults from a set of persons

- Imperative: how to do (~implementation)

```
Set<Person> selectAdults(Set<Person> persons) {  
    Set<Person> result = new HashSet<>();  
    for (Person person : persons)  
        if (person.getAge() >= 18)  
            result.add(person);  
    return result;  
}
```

- Declarative: what to do (~specification)

```
SELECT * FROM Persons WHERE Age >= 18;
```

# How to Implement Program Analyses?

Kind	Statement	Rule	Specification
New	$i: x = \text{new } T()$		$\overline{o_i \in pt(x)}$
Assign	$x = y$		$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$		$\frac{o_i \in pt(x), o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$		$\frac{o_i \in pt(x), o_j \in pt(o_i.f)}{o_j \in pt(y)}$
Call	$l: r = x.k(a_1, \dots, a_n)$		$  \begin{array}{c}  o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\  o_u \in pt(a_j), 1 \leq j \leq n \\  o_v \in pt(m_{ret}) \\  \hline  o_i \in pt(m_{this}) \\  o_u \in pt(m_{pj}), 1 \leq j \leq n \\  o_v \in pt(r)  \end{array}  $



# Pointer Analysis, Imperative Implementation

```

Solve( $m^{entry}$ )
   $WL = [], PFG = \{\}, S = \{\}, RM = \{\}, CG = \{\}$ 
  AddReachable( $m^{entry}$ )
  while  $WL$  is not empty do
    remove  $\langle n, pts \rangle$  from  $WL$ 
     $\Delta = pts - pt(n)$ 
    Propagate( $n, \Delta$ )
    if  $n$  represents a variable  $x$  then
      foreach  $o_i \in \Delta$  do
        foreach  $x.f = y \in S$  do
          AddEdge( $y, o_i.f$ )
        foreach  $y = x.f \in S$  do
          AddEdge( $s, t$ )

```

```

AddEdge( $s, t$ )
  if  $s \rightarrow t \notin PFG$  then
    add  $s \rightarrow t$  to  $PFG$ 
  if  $pt(s)$  is not empty
    add  $\langle t, pt(s) \rangle$  to  $WL$ 

```

```

Propagate( $n, pts$ )
  if  $pts$  is not empty then
     $pt(n) \cup = pts$ 
  foreach  $n \rightarrow s \in PFG$  do
    add  $\langle s, pts \rangle$  to  $WL$ 

```

```

AddReachable( $m$ )
  if  $m \notin RM$  then
    add  $m$  to  $RM$ 
     $S \cup = S_m$ 
  foreach  $i: x = \text{new } T() \in S_m$  do
    add  $\langle x, \{o_i\} \rangle$  to  $WL$ 

```

```

ProcessCall( $x, o_i$ )
  foreach  $l: r = x.k(a_1, \dots, a_n) \in S$  do
     $m = \text{Dispatch}(o_i, k)$ 
    add  $\langle m_{this}, \{o_i\} \rangle$  to  $WL$ 
  if  $m \notin CG$  then
    add  $m$  to  $CG$ 
  foreach parameter  $p_i$  of  $m$  do
    AddReachable( $m$ )
    AddEdge( $a_i, p_i$ )
  AddEdge( $m_{ret}, r$ )

```

# Pointer Analysis, Imperative Implementation

```
Solve( $m^{entry}$ )
   $WL = []$ ,  $PFG = \{\}$ ,  $S = \{\}$ ,  $RM = \{\}$ ,  $CG = \{\}$ 
```

```
  AddReachable( $m^{entry}$ )
```

```
  while  $WL$  is not empty do
```

```
    remove  $\langle n, pts \rangle$  from  $WL$ 
```

```
     $\Delta = pts - pt(n)$ 
```

```
    Propagate( $n, \Delta$ )
```

```
    if  $n$  represents a variable  $x$  then
```

```
      foreach  $o_i \in \Delta$  do
```

```
        foreach  $x.f = y \in S$  do
```

```
          AddEdge( $y, o_i.f$ )
```

```
        foreach  $y = x.f \in S$  do
```

```
          AddEdge( $y, f$ )
```

```
AddEdge( $s, t$ )
```

```
  if  $s \rightarrow t \notin PFG$  then
```

```
    add  $s \rightarrow t$  to  $PFG$ 
```

```
  if  $pt(s)$  is not empty
```

```
    add  $\langle t, pt(s) \rangle$  to  $WL$ 
```

```
Propagate( $n, pts$ )
```

```
  if  $pts$  is not empty then
```

```
     $pt(n) \cup = pts$ 
```

```
    foreach  $n \rightarrow s \in PFG$  do
```

```
      add  $\langle s, pts \rangle$  to  $WL$ 
```

```
AddReachable( $m$ )
```

```
  if  $m \notin RM$  then
```

```
    add  $m$  to  $RM$ 
```

- How to implement **worklist**?

- Array list or linked list?

- Which worklist entry should be processed first?

```
  foreach  $l: r = x.k(a_1, \dots, a_n) \in S$  do
```

```
     $m = Dispatch(o_i, k)$ 
```

```
    add  $\langle m_{this}, \{o_i\} \rangle$  to  $WL$ 
```

```
    if  $m \notin CG$  then
```

```
      add  $m$  to  $CG$ 
```

```
      AddReachable( $m$ )
```

```
      foreach parameter  $p_i$  of  $m$  do
```

```
        AddEdge( $a_i, p_i$ )
```

```
        AddEdge( $m_{ret}, r$ )
```

# Pointer Analysis, Imperative Implementation

```
Solve( $m^{entry}$ )
   $WL = []$ ,  $PFG = \{\}$ ,  $S = \{\}$ ,  $RM = \{\}$ ,  $CG = \{\}$ 
```

```
  AddReachable( $m^{entry}$ )
```

```
  while  $WL$  is not empty do
```

```
    remove  $\langle n, pts \rangle$  from  $WL$ 
```

```
     $\Delta = pts - pt(n)$ 
```

```
    Propagate( $n, \Delta$ )
```

```
    if  $n$  represents a variable  $x$  then
```

```
      foreach  $o_i \in \Delta$  do
```

```
        foreach  $x.f = y \in S$  do
```

```
          AddEdge( $y, o_i.f$ )
```

```
        foreach  $y = x.f \in S$  do
```

```
          AddEdge( $y, x.f$ )
```

```
AddEdge( $s, t$ )
```

```
  if  $s \rightarrow t \notin PFG$  then
```

```
    add  $s \rightarrow t$  to  $PFG$ 
```

```
  if  $pt(s)$  is not empty
```

```
    add  $\langle t, pt(s) \rangle$  to  $WL$ 
```

```
AddReachable( $m$ )
```

```
  if  $m \notin RM$  then
```

```
    add  $m$  to  $RM$ 
```

- How to implement **worklist**?
  - Array list or linked list?
  - Which worklist entry should be processed first?
- How to implement **points-to set** ( $pt$ )?
  - Hash set or bit vector?

```
    add  $\langle m_{this}, \{o_i\} \rangle$  to  $WL$ 
```

```
  if  $m \notin CG$  then
```

```
    add  $m$  to  $CG$ 
```

```
  AddReachable( $m$ )
```

```
  foreach parameter  $p_i$  of  $m$  do
```

```
    AddEdge( $a_i, p_i$ )
```

```
  AddEdge( $m_{ret}, r$ )
```

```
Propagate( $n, pts$ )
```

```
  if  $pts$  is not empty then
```

```
     $pt(n) \cup = pts$ 
```

```
    foreach  $n \rightarrow s \in PFG$  do
```

```
      add  $\langle s, pts \rangle$  to  $WL$ 
```

# Pointer Analysis, Imperative Implementation

```
Solve( $m^{entry}$ )
   $WL = []$ ,  $PFG = \{\}$ ,  $S = \{\}$ ,  $RM = \{\}$ ,  $CG = \{\}$ 
```

```
  AddReachable( $m^{entry}$ )
```

```
  while  $WL$  is not empty do
```

```
    remove  $\langle n, pts \rangle$  from  $WL$ 
```

```
     $\Delta = pts - pt(n)$ 
```

```
    Propagate( $n, \Delta$ )
```

```
    if  $n$  represents a variable  $x$  then
```

```
      foreach  $o_i \in \Delta$  do
```

```
        foreach  $x.f = y \in S$  do
```

```
          AddEdge( $y, o_i.f$ )
```

```
        foreach  $y = x.f \in S$  do
```

```
          AddEdge( $y, x.f$ )
```

```
AddEdge( $s, t$ )
```

```
  if  $s \rightarrow t \notin PFG$  then
```

```
    add  $s \rightarrow t$  to  $PFG$ 
```

```
  if  $pt(s)$  is not empty
```

```
    add  $\langle t, pt(s) \rangle$  to  $WL$ 
```

```
Propagate( $n, pts$ )
```

```
  if  $pts$  is not empty then
```

```
     $pt(n) \cup = pts$ 
```

```
    foreach  $n \rightarrow s \in PFG$  do
```

```
      add  $\langle s, pts \rangle$  to  $WL$ 
```

```
AddReachable( $m$ )
```

```
  if  $m \notin RM$  then
```

```
    add  $m$  to  $RM$ 
```

- How to implement **worklist**?
  - Array list or linked list?
  - Which worklist entry should be processed first?
- How to implement **points-to set** ( $pt$ )?
  - Hash set or bit vector?
- How to connect **PFG nodes** and pointers?

```
  foreach  $m \in CG$  then
```

```
    add  $m$  to  $CG$ 
```

```
  AddReachable( $m$ )
```

```
  foreach parameter  $p_i$  of  $m$  do
```

```
    AddEdge( $a_i, p_i$ )
```

```
  AddEdge( $m_{ret}, r$ )
```

# Pointer Analysis, Imperative Implementation

```
Solve( $m^{entry}$ )
   $WL = []$ ,  $PFG = \{\}$ ,  $S = \{\}$ ,  $RM = \{\}$ ,  $CG = \{\}$ 
```

```
  AddReachable( $m^{entry}$ )
```

```
  while  $WL$  is not empty do
```

```
    remove  $\langle n, pts \rangle$  from  $WL$ 
```

```
     $\Delta = pts - pt(n)$ 
```

```
    Propagate( $n, \Delta$ )
```

```
    if  $n$  represents a variable  $x$  then
```

```
      foreach  $o_i \in \Delta$  do
```

```
        foreach  $x.f = y \in S$  do
```

```
          AddEdge( $y, o_i.f$ )
```

```
          foreach  $y = x.f \in S$  do
```

```
            AddEdge( $s, t$ )
```

```
  AddEdge( $s, t$ )
```

```
    if  $s \rightarrow t \notin PFG$  then
```

```
      add  $s \rightarrow t$  to  $PFG$ 
```

```
      if  $pt(s)$  is not empty
```

```
        add  $\langle t, pt(s) \rangle$  to  $WL$ 
```

```
  Propagate( $n, \Delta$ )
```

```
    if  $pts$  is
```

```
       $pt(n) \cup = pts$ 
```

```
      foreach  $n \rightarrow s \in PFG$  do
```

```
        add  $\langle s, pts \rangle$  to  $WL$ 
```

```
AddReachable( $m$ )
```

```
  if  $m \notin RM$  then
```

```
    add  $m$  to  $RM$ 
```

- How to implement **worklist**?
  - Array list or linked list?
  - Which worklist entry should be processed first?
- How to implement **points-to set** ( $pt$ )?
  - Hash set or bit vector?
- How to connect **PFG nodes** and pointers?
- How to associate variables to the **relevant statements**?

```
  foreach  $parameter\ p_i\ of\ m$  do
```

```
    AddEdge( $a_i, p_i$ )
```

```
    AddEdge( $m_{ret}, r$ )
```

# Pointer Analysis, Imperative Implementation

```
Solve( $m^{entry}$ )
   $WL = []$ ,  $PFG = \{\}$ ,  $S = \{\}$ ,  $RM = \{\}$ ,  $CG = \{\}$ 
```

```
  AddReachable( $m^{entry}$ )
```

```
  while  $WL$  is not empty do
```

```
    remove  $\langle n, pts \rangle$  from  $WL$ 
```

```
     $\Delta = pts - pt(n)$ 
```

```
    Propagate( $n, \Delta$ )
```

```
    if  $n$  represents a variable  $x$  then
```

```
      foreach  $o_i \in \Delta$  do
```

```
        foreach  $x.f = y \in S$  do
```

```
          AddEdge( $y, o_i.f$ )
```

```
          foreach  $y = x.f \in S$  do
```

```
            AddEdge( $s, t$ )
```

```
  AddEdge( $s, t$ )
```

```
    if  $s \rightarrow t \notin PFG$  then
```

```
      add  $s \rightarrow t$  to  $PFG$ 
```

```
    if  $pt(s)$  is not empty
```

```
      add  $\langle t, pt(s) \rangle$  to  $WL$ 
```

```
AddReachable( $m$ )
```

```
  if  $m \notin RM$  then
```

```
    add  $m$  to  $RM$ 
```

- How to implement **worklist**?
  - Array list or linked list?
  - Which worklist entry should be processed first?
- How to implement **points-to set** ( $pt$ )?
  - Hash set or bit vector?
- How to connect **PFG nodes** and pointers?
- How to associate variables to the **relevant statements**?
- ...

```
  Propagate( $n, \Delta$ )
```

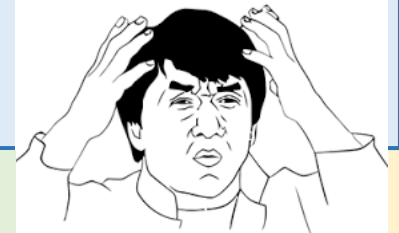
```
    if  $pts$  is
```

```
       $pt(n)$ 
```

**So many implementation details**

```
    add  $\langle s, pt(s) \rangle$  to  $WL$ 
```

```
  AddEdge( $m_{ret}, r$ )
```



# Pointer Analysis, Declarative Implementation (via Datalog)

```
VarPointsTo(x, o) <-  
  Reachable(m),  
  New(x, o, m).
```

```
VarPointsTo(x, o) <-  
  Assign(x, y),  
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-  
  Store(x, f, y),  
  VarPointsTo(x, oi),  
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-  
  Load(y, x, f),  
  VarPointsTo(x, oi),  
  FieldPointsTo(oi, f, oj).
```

```
VarPointsTo(this, o),  
Reachable(m),  
CallGraph(l, m) <-  
  VCall(l, x, k),  
  VarPointsTo(x, o),  
  Dispatch(o, k, m),  
  ThisVar(m, this).
```

```
VarPointsTo(pi, o) <-  
  CallGraph(l, m),  
  Argument(l, i, ai),  
  Parameter(m, i, pi),  
  VarPointsTo(ai, o).
```

```
VarPointsTo(r, o) <-  
  CallGraph(l, m),  
  MethodReturn(m, ret),  
  VarPointsTo(ret, o),  
  CallReturn(l, r),
```

# Pointer Analysis, Declarative Implementation (via Datalog)

```
VarPointsTo(x, o) <-  
  Reachable(m),  
  New(x, o, m).
```

```
VarPointsTo(x, o) <-  
  Assign(x, y),  
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-  
  Store(x, f, y),  
  VarPointsTo(x, oi),  
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-  
  Load(y, x, f),  
  VarPointsTo(x, oi),  
  FieldPointsTo(oi, f, oj).
```

```
VarPointsTo(this, o),  
Reachable(m),  
CallGraph(l, m) <-  
  VCall(l, x, k),  
  VarPointsTo(x, o),  
  Dispatch(o, k, m),  
  ThisVar(m, this).
```

```
VarPointsTo(pi, o) <-  
  CallGraph(l, m),  
  Argument(l, i, ai),  
  Parameter(m, i, pi),  
  VarPointsTo(ai, o).
```


```
VarPointsTo(r, o) <-  
  CallGraph(l, m),  
  MethodReturn(m, ret),  
  VarPointsTo(ret, o),  
  CallReturn(l, r),
```



- Succinct
- Readable (logic-based specification)
- Easy to implement



# Contents

- 
- The background features a light blue sky with soft clouds and green foliage. In the center, a large, dark, muscular statue of a man with a beard stands on a pedestal. To the left, a girl with long black hair and a brown cap leans on a decorative metal railing, eating a red lollipop. In the foreground, a boy with long black hair and a brown cap stands with his hand raised, wearing an orange t-shirt with a white horizontal stripe. To his right, a girl with long blue hair and a black top holds a smartphone. In the bottom left corner, a circular inset shows a close-up of a boy with long black hair and a white shirt, also holding a smartphone.
1. Motivation
  - 2. Introduction to Datalog**
  3. Pointer Analysis via Datalog
  4. Taint Analysis via Datalog

# Datalog

- Datalog is a **declarative logic** programming language that is a subset of **Prolog**.
- It emerged as a database language (mid-1980s)\*
- Now it has a variety of applications
  - Program analysis
  - Declarative networking
  - Big data
  - Cloud computing
  - ...

\*David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren, *“Datalog: Concepts, History, and Outlook”*. Chapter, 2018.

# Datalog

Datalog = Data + Logic

(and, or, not)

- No side-effects
- No control flows
- No functions
- Not Turing-complete

# Datalog

Datalog = **Data** + Logic  
(and, or, not)

- No side-effects
- No control flows
- No functions
- Not Turing-complete

# Predicates (Data)

- In Datalog, a predicate (relation) is a set of statements
- Essentially, a predicate is a **table** of data

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

Age is a predicate, which states the age of some persons.

# Predicates (Data)

- In Datalog, a predicate (relation) is a set of statements
- Essentially, a predicate is a **table** of data
- A **fact** asserts that a particular tuple (a row) belongs to a relation (a table), i.e., it represents a predicate being true for a particular combination of values

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

**Age** is a predicate, which states the age of some persons. For **Age**:

- (“Xiaoming”, 18) means “Xiaoming is 18”, which is a fact
- (“Abao”, 23) means “Abao is 23”, which is not a fact

# Atoms

- Atoms are basic elements of Datalog, which represent predicates of the form

$P(X_1, X_2, \dots, X_n)$

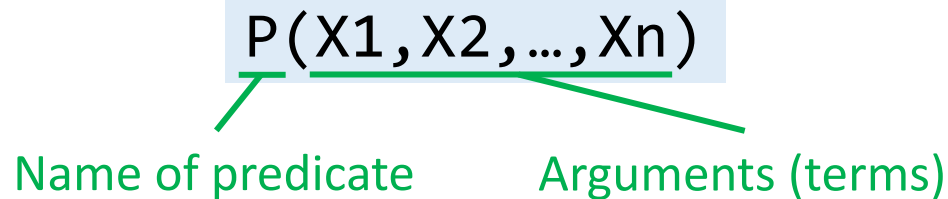
Name of predicate

Arguments (terms)

- Terms
  - Variables: stand for any values
  - Constants

# Atoms

- Atoms are basic elements of Datalog, which represent predicates of the form



- Terms
  - Variables: stand for any values
  - Constants
- Examples
  - $\text{Age}(\text{person}, \text{age})$
  - $\text{Age}(\text{"Xiaoming"}, 18)$



# Atoms (Cont.)

- $P(X_1, X_2, \dots, X_n)$  is called **relational atom**
- $P(X_1, X_2, \dots, X_n)$  evaluates to true when predicate  $P$  contains the tuple described by  $X_1, X_2, \dots, X_n$

# Atoms (Cont.)

- $P(X_1, X_2, \dots, X_n)$  is called **relational atom**
- $P(X_1, X_2, \dots, X_n)$  evaluates to true when predicate  $P$  contains the tuple described by  $X_1, X_2, \dots, X_n$ 
  - $\text{Age}(\text{“Xiaoming”}, 18)$  is ?

Age	person	age
	Xiaoming	18
	Xiaohong	23
	Alan	16
	Abao	31

# Atoms (Cont.)

- $P(X_1, X_2, \dots, X_n)$  is called **relational atom**
- $P(X_1, X_2, \dots, X_n)$  evaluates to true when predicate  $P$  contains the tuple described by  $X_1, X_2, \dots, X_n$ 
  - $\text{Age}(\text{“Xiaoming”}, 18)$  is true
  - $\text{Age}(\text{“Alan”}, 23)$  is ?

Age	person	age
	Xiaoming	18
	Xiaohong	23
	Alan	16
	Abao	31

# Atoms (Cont.)

- $P(X_1, X_2, \dots, X_n)$  is called **relational atom**
- $P(X_1, X_2, \dots, X_n)$  evaluates to true when predicate  $P$  contains the tuple described by  $X_1, X_2, \dots, X_n$ 
  - $\text{Age}(\text{“Xiaoming”}, 18)$  is true
  - $\text{Age}(\text{“Alan”}, 23)$  is false

Age	person	age
	Xiaoming	18
	Xiaohong	23
	Alan	16
	Abao	31

# Atoms (Cont.)

- $P(X_1, X_2, \dots, X_n)$  is called **relational atom**
- $P(X_1, X_2, \dots, X_n)$  evaluates to true when predicate  $P$  contains the tuple described by  $X_1, X_2, \dots, X_n$

- $\text{Age}(\text{"Xiaoming"}, 18)$  is true
- $\text{Age}(\text{"Alan"}, 23)$  is false

Age	person	age
	Xiaoming	18
	Xiaohong	23
	Alan	16
	Abao	31

- In addition to relational atoms, Datalog also has **arithmetic atoms**
  - E.g.,  $\text{age} \geq 18$

# Datalog Rules (Logic)

- Rule is a way of expressing logical inferences
- Rules also serve to specify how facts are deduced
- The form of a rule is

$$H \leftarrow B_1, B_2, \dots, B_n.$$

# Datalog Rules (Logic)

- Rule is a way of expressing logical inferences
- Rules also serve to specify how facts are deduced
- The form of a rule is

$H \leftarrow B_1, B_2, \dots, B_n.$

**Head** (consequent)

H is an atom

**Body** (antecedent)

$B_i$  is a (possibly negated) atom

Each  $B_i$  is called a **subgoal**

The meaning of a rule is “**head is true if body is true**”

# Datalog Rules (Cont.)

```
H <- B1, B2, ..., Bn.
```

“,” can be read as (logical) **and**, i.e., body  $B_1, B_2, \dots, B_n$  is true if **all subgoals**  $B_1, B_2, \dots$ , and  $B_n$  are **true**

For example, we can deduce adults via Datalog rule:

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```



# Datalog Rules (Cont.)

```
H <- B1, B2, ..., Bn.
```

“,” can be read as (logical) **and**, i.e., body  $B_1, B_2, \dots, B_n$  is true if **all subgoals**  $B_1, B_2, \dots$ , and  $B_n$  are **true**

For example, we can deduce adults via Datalog rule:

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

How to interpret the rules?

# Interpretation of Datalog Rules

$H(X_1, X_2) \leftarrow B_1(X_1, X_3), B_2(X_2, X_4), \dots, B_n(X_m).$

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

```
Adult("Xiaoming") <- Age("Xiaoming",18),18>=18.
```

# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

```
Adult("Xiaoming") <- Age("Xiaoming",18),18>=18.  
Adult("Xiaohong") <- Age("Xiaohong",23),23>=18.
```

# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

```
Adult("Xiaoming") <- Age("Xiaoming",18),18>=18.  
Adult("Xiaohong") <- Age("Xiaohong",23),23>=18.  
Age("Alan",16),16>=18.
```

# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

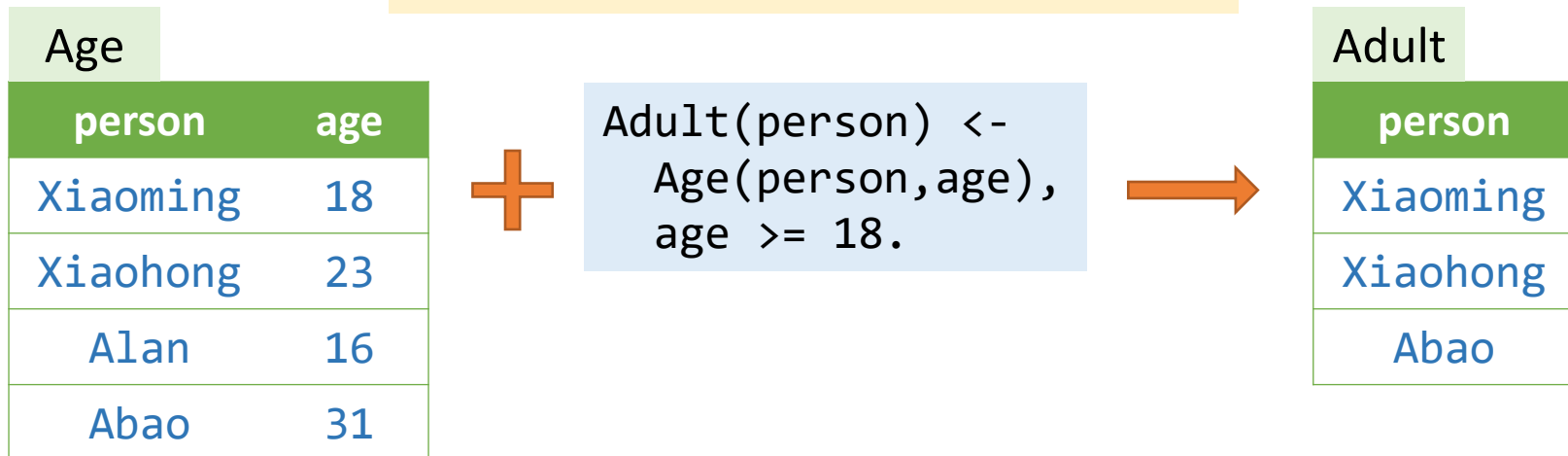
```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

```
Adult("Xiaoming") <- Age("Xiaoming", 18), 18 >= 18.  
Adult("Xiaohong") <- Age("Xiaohong", 23), 23 >= 18.  
Adult("Alan") <- Age("Alan", 16), 16 >= 18.  
Adult("Abao") <- Age("Abao", 31), 31 >= 18.
```

# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Datalog program = Facts + Rules





# Rule Interpretation: An Example

- Consider **all** possible **combinations** of values of the variables in the subgoals
- If a combination makes **all subgoals true**, then the head atom (with corresponding values) is also true
- The head **predicate** consists of all **true atoms**

Datalog program = **Facts** + Rules

Age	
person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

+

```
Adult(person) <-  
  Age(person, age),  
  age >= 18.
```

→

Adult	
person	
Xiaoming	
Xiaohong	
Abao	

Where does initial data come from?

# EDB and IDB Predicates

Conventionally, predicates in Datalog are divided into two kinds:

## 1. EDB (extensional database)

- The predicates that are defined in a priori
- Relations are immutable
- Can be seen as input relations

## 2. IDB (intensional database)

- The predicates that are established only by rules
- Relations are inferred by rules
- Can be seen as output relations

# EDB and IDB Predicates

Conventionally, predicates in Datalog are divided into two kinds:

## 1. EDB (extensional database)

- The predicates that are defined in a priori
- Relations are immutable
- Can be seen as input relations

## 2. IDB (intensional database)

- The predicates that are established only by rules
- Relations are inferred by rules
- Can be seen as output relations

$H \leftarrow B_1, B_2, \dots, B_n.$

- H can only be IDB
- $B_i$  can be EDB or IDB

# Logical Or

There are two ways to express logical or in Datalog

## 1. Write multiple rules with the same head

```
SportFan(person) <- Hobby(person, "jogging").  
SportFan(person) <- Hobby(person, "swimming").
```

## 2. Use logical or operator “;”

```
SportFan(person) <-  
  Hobby(person, "jogging");  
  Hobby(person, "swimming").
```

Hobby	
person	hobby
Xiaoming	cooking
Xiaoming	singing
Xiaohong	jogging
Abao	sleeping
Alan	swimming
...	...

# Logical Or

There are two ways to express logical or in Datalog

## 1. Write multiple rules with the same head

```
SportFan(person) <- Hobby(person, "jogging").  
SportFan(person) <- Hobby(person, "swimming").
```

## 2. Use logical or operator “;”

```
SportFan(person) <-  
  Hobby(person, "jogging");  
  Hobby(person, "swimming").
```

The precedence of “;” (or) is **lower** than “,” (and), so disjunctions may be enclosed by parentheses, e.g.,  $H \leftarrow A, (B;C)$ .

Hobby	
person	hobby
Xiaoming	cooking
Xiaoming	singing
Xiaohong	jogging
Abao	sleeping
Alan	swimming
...	...

# Negation

$H(X1, X2) \leftarrow B1(X1, X3), !B2(X2, X4), \dots, Bn(Xm).$

- In Datalog rules, a subgoal can be a **negated** atom, which negates its meaning
- Negated subgoal is written as **!B(...)**, and read as not **B(...)**

# Negation

```
H(X1,X2) <- B1(X1,X3), !B2(X2,X4),..., Bn(Xm).
```

- In Datalog rules, a subgoal can be a **negated** atom, which negates its meaning
- Negated subgoal is written as **!B(...)**, and read as not **B(...)**
- For example, to compute the students who need to take a make-up exam, we can write

```
MakeupExamStd(student) <-  
  Student(student),  
  !PassedStd(student).
```

Where **Student** stores all students, and **PassedStd** stores the students who passed the exam.

# Recursion

- Datalog supports **recursive rules**, which allows that an IDB predicate can be deduced (directly/indirectly) from itself



# Recursion

- Datalog supports **recursive rules**, which allows that an IDB predicate can be deduced (directly/indirectly) from itself
- For example, we can compute the reachability information (i.e., transitive closure) of a graph with recursive rules:

```
Reach(from, to) <-  
  Edge(from, to).
```

```
Reach(from, to) <-  
  Reach(from, node),  
  Edge(node, to).
```

Where **Edge(a, b)** means that the graph has an edge from node **a** to node **b**, and **Reach(a, b)** means that **b** is reachable from **a**.

# Recursion (Cont.)

- Without recursion, Datalog can only express the queries of basic relational algebra
  - Basically a SQL with **SELECT - FROM - WHERE**
- With recursion, Datalog becomes much more powerful, and is able to express sophisticated program analyses, such as pointer analysis

# Rule Safety

Are these rules ok?

$A(x) \leftarrow B(y), x > y.$

$A(x) \leftarrow B(y), !C(x,y).$

# Rule Safety

Are these rules ok?

$A(x) \leftarrow B(y), x > y.$



$A(x) \leftarrow B(y), !C(x,y).$

For both rules, infinite values of  $x$  can satisfy the rule, which makes  $A$  an *infinite relation*.

# Rule Safety

Are these rules ok?

$A(x) \leftarrow B(y), x > y.$



$A(x) \leftarrow B(y), \neg C(x,y).$

For both rules, infinite values of  $x$  can satisfy the rule,  
which makes  $A$  an *infinite relation*.

- A rule is **safe** if every variable appears in at least one **non-negated relational** atom
- Above two rules are **unsafe**
- In Datalog, **only** safe rules are allowed

# Recursion and Negation

Is this rule ok?

$A(x) \leftarrow B(x), \neg A(x)$

# Recursion and Negation

Is this rule ok?

$A(x) \leftarrow B(x), \neg A(x)$

Suppose  $B(1)$  is true.  
If  $A(1)$  is false, then  $A(1)$  is true.  
If  $A(1)$  is true,  $A(1)$  should not be true.  
...



# Recursion and Negation

Is this rule ok?

$A(x) \leftarrow B(x), \neg A(x)$

Suppose  $B(1)$  is true.  
If  $A(1)$  is false, then  $A(1)$  is true.  
If  $A(1)$  is true,  $A(1)$  should not be true.  
...

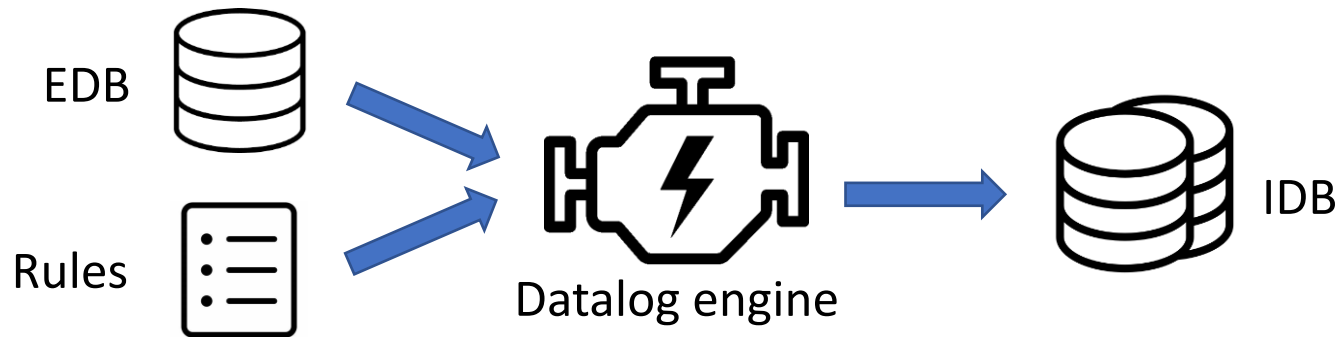


The rule is ***contradictory*** and makes no sense

In Datalog, **recursion** and **negation** of an atom must be **separated**. Otherwise, the rules may contain contradiction and the inference fails to converge.



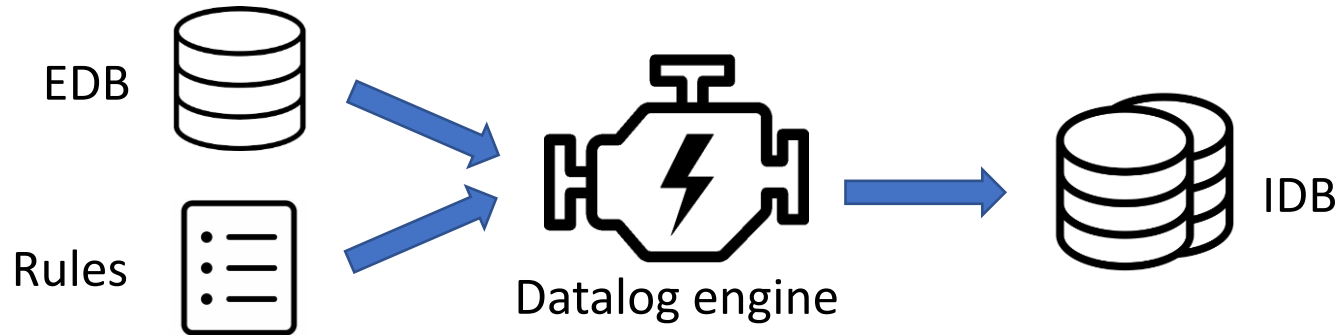
# Execution of Datalog Programs



- Datalog engine deduces facts by given rules and EDB predicates until no new facts can be deduced. Some modern Datalog engines

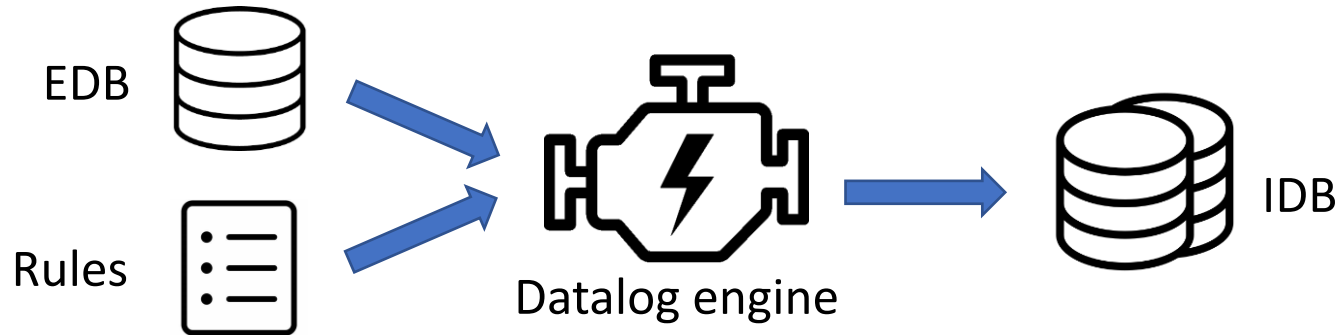
LogicBlox, Soufflé, XSB, Datomic, Flora-2, ...

# Execution of Datalog Programs



- Datalog engine deduces facts by given rules and EDB predicates until no new facts can be deduced. Some modern Datalog engines  
*LogicBlox, Soufflé, XSB, Datomic, Flora-2, ...*
- Monotonicity: Datalog is **monotone** as facts cannot be deleted

# Execution of Datalog Programs



- Datalog engine deduces facts by given rules and EDB predicates until no new facts can be deduced. Some modern Datalog engines

*LogicBlox, Soufflé, XSB, Datomic, Flora-2, ...*

- **Monotonicity:** Datalog is **monotone** as facts cannot be deleted
- **Termination:** A Datalog program **always terminates** as
  - 1) Datalog is monotone
  - 2) Possible values of IDB predicates are finite (rule safety)

# Contents

The background features a faded illustration of several anime-style characters in a park setting. A large, muscular, grey stone statue of a man with a beard and a raised hand stands in the center. To the left, a girl with a brown cap leans on a decorative metal railing, eating a lollipop. In the foreground, a boy with dark hair sits on the railing, looking at a smartphone. To the right, a girl with long purple hair stands holding a smartphone. In the center, a boy with long black hair and a black cap is shown from the chest up, wearing a brown t-shirt with a white horizontal stripe, gesturing with his hands as if speaking.

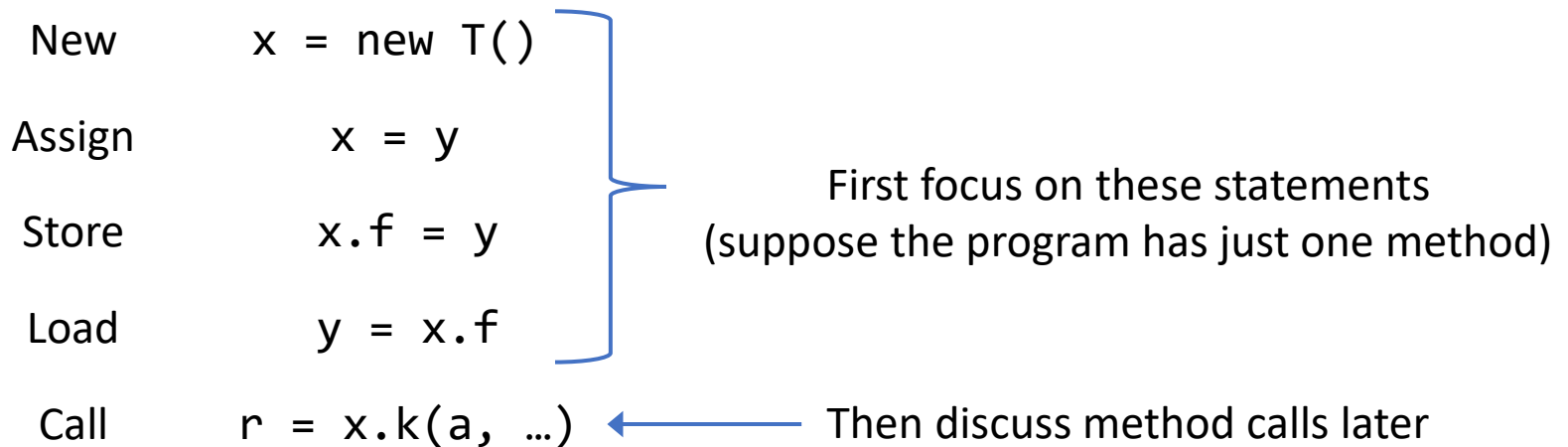
1. Motivation
2. Introduction to Datalog
- 3. Pointer Analysis via Datalog**
4. Taint Analysis via Datalog

# Pointer Analysis via Datalog

- EDB: pointer-relevant information that can be extracted from program syntactically
- IDB: pointer analysis results
- Rules: pointer analysis rules

# Pointer Analysis via Datalog

- EDB: pointer-relevant information that can be extracted from program syntactically
- IDB: pointer analysis results
- Rules: pointer analysis rules



# Datalog Model for Pointer Analysis

Kind	Statement
New	$i: x = \text{new } T()$
Assign	$x = y$
Store	$x.f = y$
Load	$y = x.f$

Variables: **V**

Fields: **F**

Objects: **O**

EDB

`New(x : V, o : O)`

`Assign(x : V, y : V)`

`Store(x : V, f : F, y : V)`

`Load(y : V, x : V, f : F)`

IDB

`VarPointsTo(v : V, o : O)`

e.g., fact `VarPointsTo(x, oi)` represents  $o_i \in pt(x)$

`FieldPointsTo(oi : O, f : F, oj : O)`

e.g., fact `FieldPointsTo(oi, f, oj)` represents  $o_j \in pt(o_i.f)$

# An Example

```
1 b = new C();  
2 a = b;  
3 c = new C();  
4 c.f = a;  
5 d = c;  
6 c.f = d;  
7 e = d.f;
```

Variables: V

Fields: F

Objects: O



# An Example

New( $x : \mathbf{V}$ ,  $o : \mathbf{O}$ )

New	
$b$	$o_1$
$c$	$o_3$

```
1 b = new C();  
2 a = b;  
3 c = new C();  
4 c.f = a;  
5 d = c;  
6 c.f = d;  
7 e = d.f;
```

Variables: V

Fields: F

Objects: O

# An Example

```
1 b = new C();  
2 a = b;  
3 c = new C();  
4 c.f = a;  
5 d = c;  
6 c.f = d;  
7 e = d.f;
```

New( $x : \mathbf{V}$ ,  $o : \mathbf{O}$ )

New	
$b$	$o_1$
$c$	$o_3$

Assign( $x : \mathbf{V}$ ,  $y : \mathbf{V}$ )

Assign	
$a$	$b$
$d$	$c$

Variables:  $\mathbf{V}$   
Fields:  $\mathbf{F}$   
Objects:  $\mathbf{O}$

# An Example

```
1 b = new C();  
2 a = b;  
3 c = new C();  
4 c.f = a;  
5 d = c;  
6 c.f = d;  
7 e = d.f;
```

Variables: V  
Fields: F  
Objects: O

New( $x : \mathbf{V}$ ,  $o : \mathbf{O}$ )

New	
$b$	$o_1$
$c$	$o_3$

Assign( $x : \mathbf{V}$ ,  $y : \mathbf{V}$ )

Assign	
$a$	$b$
$d$	$c$

Store( $x : \mathbf{V}$ ,  $f : \mathbf{F}$ ,  $y : \mathbf{V}$ )

Store		
$c$	$f$	$a$
$c$	$f$	$d$

# An Example

```
1 b = new C();  
2 a = b;  
3 c = new C();  
4 c.f = a;  
5 d = c;  
6 c.f = d;  
7 e = d.f;
```

Variables: V

Fields: F

Objects: O

New( $x : \mathbf{V}$ ,  $o : \mathbf{O}$ )

New	
$b$	$o_1$
$c$	$o_3$

Assign( $x : \mathbf{V}$ ,  $y : \mathbf{V}$ )

Assign	
$a$	$b$
$d$	$c$

Store( $x : \mathbf{V}$ ,  $f : \mathbf{F}$ ,  $y : \mathbf{V}$ )

Store		
$c$	$f$	$a$
$c$	$f$	$d$

Load( $x : \mathbf{V}$ ,  $y : \mathbf{V}$ ,  $f : \mathbf{F}$ )

Load		
$e$	$d$	$f$

# An Example

```
1 b = new C();  
2 a = b;  
3 c = new C();  
4 c.f = a;  
5 d = c;  
6 c.f = d;  
7 e = d.f;
```

Variables: V

Fields: F

Objects: O

New( $x : V, o : O$ )

New	
$b$	$o_1$
$c$	$o_3$

Assign( $x : V, y : V$ )

Assign	
$a$	$b$
$d$	$c$

Store( $x : V, f : F, y : V$ )

Store		
$c$	$f$	$a$
$c$	$f$	$d$

Load( $x : V, y : V, f : F$ )

Load		
$e$	$d$	$f$

# Datalog Rules for Pointer Analysis

Kind	Statement	Rule
New	$i: x = \text{new } T()$	$\frac{}{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
VarPointsTo(x, o) <-
  New(x, o).
```

```
VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

# Datalog Rules for Pointer Analysis

Kind	Statement	Rule
New	$i: x = \text{new } T()$	$\frac{}{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
VarPointsTo(x, o) <-
  New(x, o).
```

```
VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

# Datalog Rules for Pointer Analysis

Kind	Statement	Rule
New	$i: x = \text{new } T()$	$\frac{}{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
VarPointsTo(x, o) <-
  New(x, o).
```

```
VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```



# Datalog Rules for Pointer Analysis

Kind	Statement	Rule
New	$i: x = \text{new } T()$	$\frac{}{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
VarPointsTo(x, o) <-
  New(x, o).
```

```
VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

# Datalog Rules for Pointer Analysis

Kind	Statement	Rule
New	$i: x = \text{new } T()$	$\frac{}{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
VarPointsTo(x, o) <-
  New(x, o).
```

```
VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

# An Example

```
1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;
```

```
VarPointsTo(x, o) <-
  New(x, o).
```

```
VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

New(x:V, o:O)

New	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

VarPointsTo(v:V, o:O)

VarPointsTo

FieldPointsTo

FieldPointsTo(oi:O, f:F, oj:O)

# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(b, o1) <-
  New(b, o1).
VarPointsTo(c, o3) <-
  New(c, o3).

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

VarPointsTo(v:V, o:O)

VarPointsTo	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

FieldPointsTo

FieldPointsTo(o<sub>i</sub>:O, f:F, o<sub>j</sub>:O)

# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(a, o1) <-
  Assign(a, b),
  VarPointsTo(b, o1).

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
b	o <sub>1</sub>
c	o <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
c	f	a
c	f	d

Assign(x:V, y:V)

Assign	
a	b
d	c

Load(x:V, y:V, f:F)

Load		
e	d	f

VarPointsTo(v:V, o:O)

VarPointsTo	
b	o <sub>1</sub>
c	o <sub>3</sub>
a	o <sub>1</sub>

FieldPointsTo

FieldPointsTo(oi:O, f:F, oj:O)

# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

VarPointsTo(v:V, o:O)

VarPointsTo	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>
<i>a</i>	<i>o</i> <sub>1</sub>
<i>d</i>	<i>o</i> <sub>3</sub>

FieldPointsTo

FieldPointsTo(oi:O, f:F, oj:O)

# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

VarPointsTo(v:V, o:O)

VarPointsTo	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>
<i>a</i>	<i>o</i> <sub>1</sub>
<i>d</i>	<i>o</i> <sub>3</sub>

FieldPointsTo

<i>o</i> <sub>3</sub>	<i>f</i>	<i>o</i> <sub>1</sub>
-----------------------	----------	-----------------------

FieldPointsTo(oi:O, f:F, oj:O)

```

FieldPointsTo(o1, f, o3) <-
  Store(c, f, a),
  VarPointsTo(c, o3),
  VarPointsTo(a, o1).

```

# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

VarPointsTo(v:V, o:O)

VarPointsTo	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>
<i>a</i>	<i>o</i> <sub>1</sub>
<i>d</i>	<i>o</i> <sub>3</sub>

FieldPointsTo

FieldPointsTo		
<i>o</i> <sub>3</sub>	<i>f</i>	<i>o</i> <sub>1</sub>
<i>o</i> <sub>3</sub>	<i>f</i>	<i>o</i> <sub>3</sub>

FieldPointsTo(oi:O, f:F, oj:O)



# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

VarPointsTo(v:V, o:O)

VarPointsTo	
<i>b</i>	<i>o</i> <sub>1</sub>
<i>c</i>	<i>o</i> <sub>3</sub>
<i>a</i>	<i>o</i> <sub>1</sub>
<i>d</i>	<i>o</i> <sub>3</sub>
<i>e</i>	<i>o</i> <sub>1</sub>
<i>e</i>	<i>o</i> <sub>3</sub>

FieldPointsTo

FieldPointsTo		
<i>o</i> <sub>3</sub>	<i>f</i>	<i>o</i> <sub>1</sub>
<i>o</i> <sub>3</sub>	<i>f</i>	<i>o</i> <sub>3</sub>

FieldPointsTo(oi:O, f:F, oj:O)

# An Example

```

1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;

```

```

VarPointsTo(x, o) <-
  New(x, o).

```

```

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

```

```

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

```

```

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).

```

New(x:V, o:O)

New	
<i>b</i>	$o_1$
<i>c</i>	$o_3$

Store(x:V, f:F, y:V)

Store		
<i>c</i>	<i>f</i>	<i>a</i>
<i>c</i>	<i>f</i>	<i>d</i>

Assign(x:V, y:V)

Assign	
<i>a</i>	<i>b</i>
<i>d</i>	<i>c</i>

Load(x:V, y:V, f:F)

Load		
<i>e</i>	<i>d</i>	<i>f</i>

VarPointsTo(v:V, o:O)

VarPointsTo	
<i>b</i>	$o_1$
<i>c</i>	$o_3$
<i>a</i>	$o_1$
<i>d</i>	$o_3$
<i>e</i>	$o_1$
<i>e</i>	$o_3$

FieldPointsTo

FieldPointsTo		
$o_3$	<i>f</i>	$o_1$
$o_3$	<i>f</i>	$o_3$

FieldPointsTo(oi:O, f:F, oj:O)

# Handle Method Calls

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$\frac{\begin{array}{l} o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\ o_u \in pt(a_j), 1 \leq j \leq n \\ o_v \in pt(m_{ret}) \end{array}}{\begin{array}{l} o_i \in pt(m_{this}) \\ o_u \in pt(m_{pj}), 1 \leq j \leq n \\ o_v \in pt(r) \end{array}}$

## EDB

- $V\text{Call}(l:\mathbf{S}, x:\mathbf{V}, k:\mathbf{M})$
- $\text{Dispatch}(o:\mathbf{O}, k:\mathbf{M}, m:\mathbf{M})$
- $\text{ThisVar}(m:\mathbf{M}, \text{this}:\mathbf{V})$

## IDB

- $\text{Reachable}(m:\mathbf{M})$
- $\text{CallGraph}(l:\mathbf{S}, m:\mathbf{M})$

Statements      S  
 (Labels):  
 Methods:      M

# Handle Method Calls

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$\frac{\begin{array}{l} \rightarrow o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\ o_u \in pt(a_j), 1 \leq j \leq n \\ o_v \in pt(m_{ret}) \end{array}}{\begin{array}{l} \rightarrow o_i \in pt(m_{this}) \\ o_u \in pt(m_{pj}), 1 \leq j \leq n \\ o_v \in pt(r) \end{array}}$

EDB

- $\text{VCall}(l:\mathbf{S}, x:\mathbf{V}, k:\mathbf{M})$
- $\text{Dispatch}(o:\mathbf{O}, k:\mathbf{M}, m:\mathbf{M})$
- $\text{ThisVar}(m:\mathbf{M}, \text{this}:\mathbf{V})$

IDB

- $\text{Reachable}(m:\mathbf{M})$
- $\text{CallGraph}(l:\mathbf{S}, m:\mathbf{M})$

```

VarPointsTo(this, o),
Reachable(m),
CallGraph(l, m) <-
  VCall(l, x, k),
  VarPointsTo(x, o),
  Dispatch(o, k, m),
  ThisVar(m, this).
    
```

Statements      S  
 (Labels):  
 Methods:        M

# Handle Method Calls

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$  \begin{array}{c}  o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\  \longrightarrow o_u \in pt(a_j), 1 \leq j \leq n \\  \frac{o_v \in pt(m_{ret})}{o_i \in pt(m_{this})} \\  \longrightarrow o_u \in pt(m_{pj}), 1 \leq j \leq n \\  o_v \in pt(r)  \end{array}  $

EDB

- $\text{Argument}(l:\mathbf{S}, i:\mathbf{N}, ai:\mathbf{V})$
- $\text{Parameter}(m:\mathbf{M}, i:\mathbf{N}, pi:\mathbf{V})$

```

VarPointsTo(pi, o) <-
  CallGraph(l, m),
  Argument(l, i, ai),
  Parameter(m, i, pi),
  VarPointsTo(ai, o).
    
```

Statements  $\mathbf{S}$   
 (Labels):  
 Methods:  $\mathbf{M}$   
 Nature numbers  $\mathbf{N}$   
 (indexes)

# Handle Method Calls

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$  \begin{array}{l}  o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\  o_u \in pt(a_j), 1 \leq j \leq n \\  \longrightarrow o_v \in pt(m_{ret}) \\  \hline  o_i \in pt(m_{this}) \\  o_u \in pt(m_{pj}), 1 \leq j \leq n \\  \longrightarrow o_v \in pt(r)  \end{array}  $

EDB

- MethodReturn( $m:\mathbf{M}$ ,  $ret:\mathbf{V}$ )
- CallReturn( $l:\mathbf{S}$ ,  $r:\mathbf{V}$ )

```

VarPointsTo(r, o) <-
  CallGraph(l, m),
  MethodReturn(m, ret),
  VarPointsTo(ret, o),
  CallReturn(l, r).
    
```

Statements      S  
 (Labels):  
 Methods:        M

# Handle Method Calls

Kind	Statement	Rule
Call	$l: r = x.k(a_1, \dots, a_n)$	$\frac{\begin{array}{l} o_i \in pt(x), m = \text{Dispatch}(o_i, k) \\ o_u \in pt(a_j), 1 \leq j \leq n \\ o_v \in pt(m_{ret}) \end{array}}{\begin{array}{l} o_i \in pt(m_{this}) \\ o_u \in pt(m_{pj}), 1 \leq j \leq n \\ o_v \in pt(r) \end{array}}$

```

VarPointsTo(this, o),
Reachable(m),
CallGraph(l, m) <-
  VCall(l, x, k),
  VarPointsTo(x, o),
  Dispatch(o, k, m),
  ThisVar(m, this).
    
```

```

VarPointsTo(pi, o) <-
  CallGraph(l, m),
  Argument(l, i, ai),
  Parameter(m, i, pi),
  VarPointsTo(ai, o).
    
```

```

VarPointsTo(r, o) <-
  CallGraph(l, m),
  MethodReturn(m, ret),
  VarPointsTo(ret, o),
  CallReturn(l, r).
    
```

# Whole-Program Pointer Analysis

```
Reachable(m) <-  
  EntryMethod(m).
```

```
VarPointsTo(x, o) <-  
  Reachable(m),  
  New(x, o, m).
```

```
VarPointsTo(x, o) <-  
  Assign(x, y),  
  VarPointsTo(y, o).
```

```
FieldPointsTo(oi, f, oj) <-  
  Store(x, f, y),  
  VarPointsTo(x, oi),  
  VarPointsTo(y, oj).
```

```
VarPointsTo(y, oj) <-  
  Load(y, x, f),  
  VarPointsTo(x, oi),  
  FieldPointsTo(oi, f, oj).
```

```
VarPointsTo(this, o),  
Reachable(m),  
CallGraph(l, m) <-  
  VCall(l, x, k),  
  VarPointsTo(x, o),  
  Dispatch(o, k, m),  
  ThisVar(m, this).
```

```
VarPointsTo(pi, o) <-  
  CallGraph(l, m),  
  Argument(l, i, ai),  
  Parameter(m, i, pi),  
  VarPointsTo(ai, o).
```

```
VarPointsTo(r, o) <-  
  CallGraph(l, m),  
  MethodReturn(m, ret),  
  VarPointsTo(ret, o),  
  CallReturn(l, r).
```



# Contents

The background features a faded illustration of several anime-style characters in a park setting. A large, dark, muscular statue stands in the center. To the left, a girl with a brown cap leans on a decorative metal railing, eating a lollipop. In the foreground, a boy with dark hair sits on the railing, holding a smartphone. To the right, a boy with long hair and a brown t-shirt stands with his hand raised, and a girl with blue hair stands behind him, also holding a smartphone. The scene is set against a backdrop of green trees and a blue sky with light clouds.

1. Motivation
2. Introduction to Datalog
3. Pointer Analysis via Datalog
- 4. Taint Analysis via Datalog**

# Datalog Model for Taint Analysis

On top of pointer analysis

- EDB predicates
  - $\text{Source}(m : \mathbf{M})$  // source methods
  - $\text{Sink}(m : \mathbf{M})$  // sink methods
  - $\text{Taint}(l : \mathbf{S}, t : \mathbf{T})$  // associates each call site to the tainted data from the call site
- IDB predicate
  - $\text{TaintFlow}(t : \mathbf{T}, m : \mathbf{M})$  // detected taint flows, e.g.,  $\text{TaintFlow}(t, m)$  denotes that tainted data  $t$  may flow to sink method  $m$

# Taint Analysis via Datalog

- Handles sources (generates tainted data)

Kind	Statement	Rule
Call	$L: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG \quad m \in \text{Sources}}{t_l \in pt(r)}$

```
VarPointsTo(r, t) <-
  CallGraph(l, m),
  Source(m),
  CallReturn(l, r),
  Taint(l, t).
```

- Handles sinks (generates taint flow information)

Kind	Statement	Rule
Call	$L: r = x.k(a_1, \dots, a_n)$	$\frac{l \rightarrow m \in CG \quad m \in \text{Sinks} \quad \exists i, 1 \leq i \leq n: t_j \in pt(a_i)}{\langle t_j, m \rangle \in \text{TaintFlows}}$

```
TaintFlow(t, m) <-
  CallGraph(l, m),
  Sink(m),
  Argument(l, _, ai),
  VarPointsTo(ai, t),
  Taint(_, t).
```

# Datalog-Based Program Analysis

- Pros
  - Succinct and readable
  - Easy to implement
  - Benefit from off-the-shelf optimized Datalog engines
- Cons
  - Restricted expressiveness, i.e., it is impossible or inconvenient to express some logics
  - Cannot fully control performance

# The X You Need To Understand in This Lecture

- Datalog language
- How to implement pointer analysis via Datalog
- How to implement taint analysis via Datalog

注意注意!  
划重点了!



# 软件分析

南京大学

计算机科学与技术系

程序设计语言与

静态分析研究组

李棣  
谭添