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
李樾 谭添

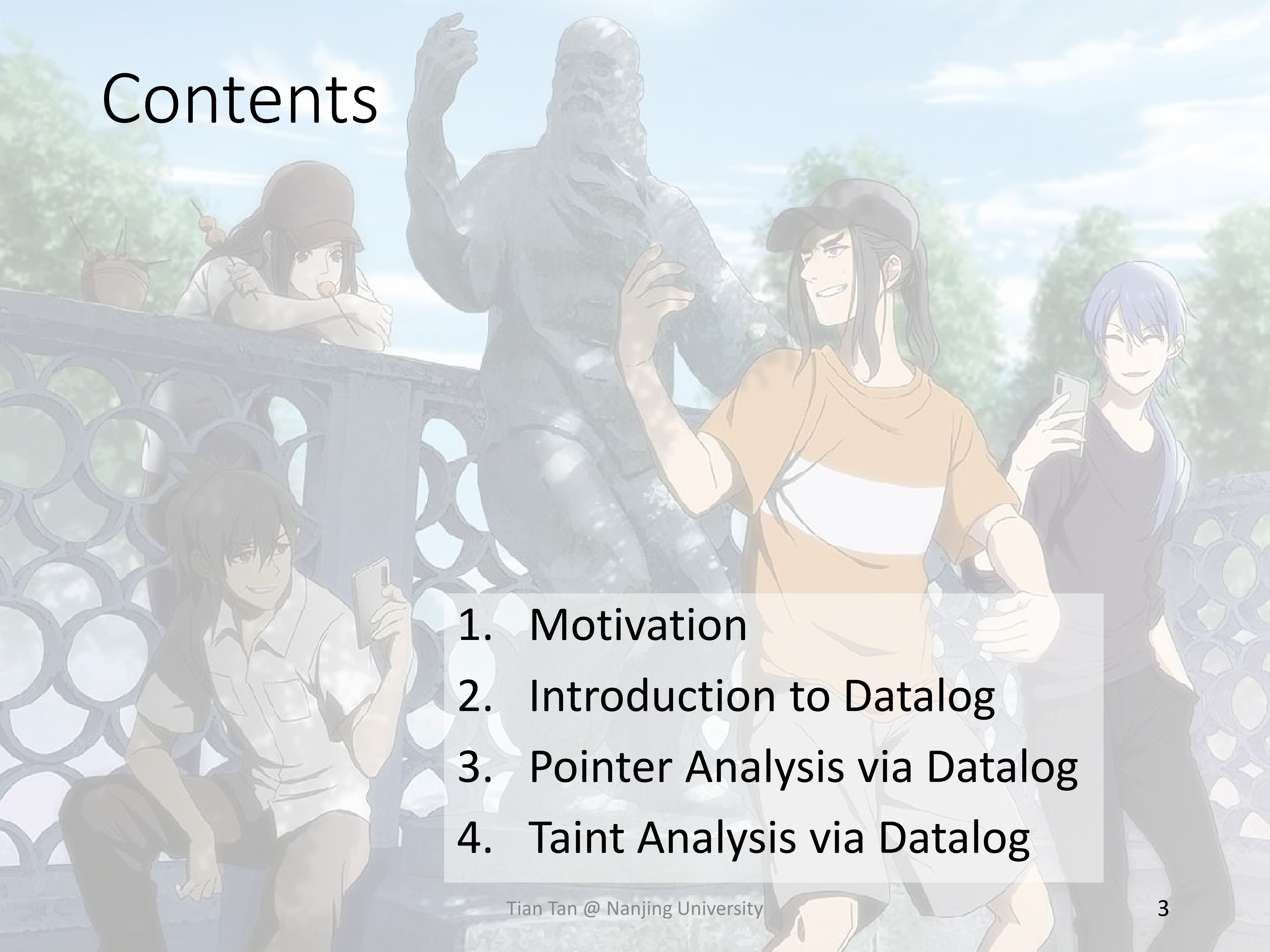
Static Program Analysis

Datalog-Based Program Analysis

Nanjing University

Tian Tan

2021



Contents

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

Contents

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 = new T()`	$\overline{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x),\ \ o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x),\ \ o_j \in pt(o_i.f)}{o_j \in pt(y)}$
Call	`l: r = x.k(a1,…,an)`	$\dfrac{\begin{array}{c} o_i \in pt(x),\ \ m = \text{Dispatch}(o_i, \text{k}) \\ o_u \in pt(aj), 1 \leq j \leq n \\ o_v \in pt(m_{ret}) \end{array}}{\begin{array}{c} o_i \in pt(m_{this}) \\ o_u \in pt(m_{pj}), 1 \leq j \leq n \\ o_v \in pt(r) \end{array}}$

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, Δ)
    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**

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

ProcessCall(x, o_i)
  foreach `l: r = x.k(a1,…,an)` $\in S$ **do**
    m = Dispatch(o_i, k)
    add $\langle m_{this}, \{o_i\} \rangle$ to WL
    $\notin CG$ **then**
    m to CG
    chable(m)
    parameter p_i of m **do**
    dge(a_i, p_i)
    e(m_{ret}, r)

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

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, Δ)
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**

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(a1,…,an)` $\in S$ **do**
m = Dispatch(o_i, k)
add $\langle m_{this}, \{o_i\} \rangle$ to WL
$\notin CG$ **then**
m to CG
chable(m)
parameter p_i of m **do**
dge(a_i, p_i)
e(m_{ret}, r)

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

Propagate(n, pts)
if pts is not empty **then**
$pt(n)$ ⋃= 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, Δ)
    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**

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
$\notin CG$ **then**
m to CG
chable(m)
parameter p_i of m **do**
dge(a_i, p_i)
e(m_{ret}, r)

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

Propagate(n, pts)
  if pts is not empty **then**
    $pt(n)$ ⋃= 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, Δ)
    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**

AddReachable(m)
  if $m \notin RM$ **then**
    add m to RM
  …
  $\notin CG$ **then**
  m to CG
  chable(m)
  parameter p_i of m **do**
  dge(a_i, p_i)
  e(m_{ret}, r)

- 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?

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

Propagate(n, pts)
  if pts is not empty **then**
    $pt(n)$ ⋃= 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, Δ)
        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**

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?

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

Propagate(
    if pts is
        $pt(n)$ ⋃= pts
        foreach $n \rightarrow s \in PFG$ **do**
            add $\langle s, pts \rangle$ to WL

chable(m)
    parameter p_i of m **do**
    dge(a_i, p_i)
    e(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, Δ)
    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**

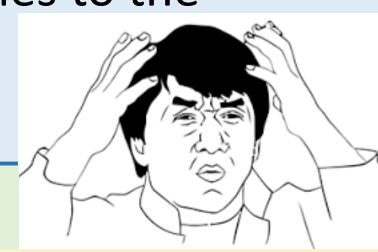
AddReachable(m)
  if $m \notin RM$ **then**
    add m to RM

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(
  if pts is
    $pt(n)$

AddEdge(m_{ret}, r)

- 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?
- …



So many implementation details

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

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

$$\underbrace{\texttt{P}}_{\text{Name of predicate}}(\underbrace{\texttt{X1,X2,…,Xn}}_{\text{Arguments (terms)}})$$

- Terms
  - Variables: stand for any values
  - Constants

Atoms

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

$$\underbrace{\texttt{P}}_{\text{Name of predicate}}\texttt{(}\underbrace{\texttt{X1,X2,…,Xn}}_{\text{Arguments (terms)}}\texttt{)}$$

- Terms
  - Variables: stand for any values
  - Constants
- Examples
  - `Age(person,age)`
  - `Age(“Xiaoming”,18)`

Atoms (Cont.)

- `P(X1,X2,…,Xn)` is called relational atom
- `P(X1,X2,…,Xn)` evaluates to true when predicate P contains the tuple described by `X1,X2,…,Xn`

Atoms (Cont.)

- `P(X1,X2,…,Xn)` is called relational atom
- `P(X1,X2,…,Xn)` evaluates to true when predicate P contains the tuple described by `X1,X2,…,Xn`
  - `Age("Xiaoming",18)` is ?

Age

person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

Atoms (Cont.)

- `P(X1,X2,…,Xn)` is called relational atom
- `P(X1,X2,…,Xn)` evaluates to true when predicate P contains the tuple described by `X1,X2,…,Xn`
  - `Age("Xiaoming",18)` is true
  - `Age("Alan",23)` is ?

Age	person	age
	Xiaoming	18
	Xiaohong	23
	Alan	16
	Abao	31

Atoms (Cont.)

- `P(X1,X2,…,Xn)` is called relational atom
- `P(X1,X2,…,Xn)` evaluates to true when predicate P contains the tuple described by `X1,X2,…,Xn`
  - `Age("Xiaoming",18)` is true
  - `Age("Alan",23)` is false

Age	person	age
	Xiaoming	18
	Xiaohong	23
	Alan	16
	Abao	31

Atoms (Cont.)

- `P(X1,X2,…,Xn)` is called relational atom
- `P(X1,X2,…,Xn)` evaluates to true when predicate P contains the tuple described by `X1,X2,…,Xn`
  - `Age("Xiaoming",18)` is true
  - `Age("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., `age >= 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 <- B1,B2,…,Bn.
```

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 <- B1,B2,…,Bn.
```

Head (consequent)
H is an atom

Body (antecedent)
`Bi` is a (possibly negated) atom
Each `Bi` 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 `B1,B2,…,Bn` is true if all subgoals B1, B2, …, and Bn 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 `B1,B2,…,Bn` is true if all subgoals B1, B2, …, and Bn 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(X1,X2) <- B1(X1,X3),B2(X2,X4),…,Bn(Xm).
```

- 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.
                     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

Age

person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

\+

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

→

Adult

person
Xiaoming
Xiaohong
Abao

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 <- B1,B2,…,Bn.
```

- H can only be IDB
- `Bi` 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 <- A,(B;C).`

Hobby

person	hobby
Xiaoming	cooking
Xiaoming	singing
Xiaohong	jogging
Abao	sleeping
Alan	swimming
…	…

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(…)`

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) <- B(y), x > y.
```

```
A(x) <- B(y), !C(x,y).
```

Rule Safety

Are these rules ok?

```
A(x) <- B(y), x > y.
```


```
A(x) <- 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) <- B(y), x > y.
```


```
A(x) <- B(y), !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) <- B(x), !A(x).
```

Recursion and Negation

Is this rule ok?

```
A(x) <- B(x), !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?

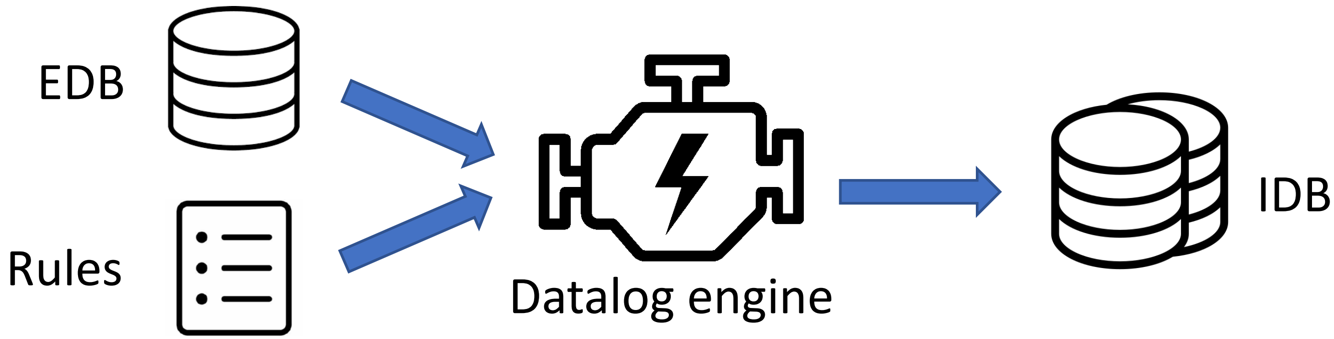
```
A(x) <- B(x), !A(x).
```

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.

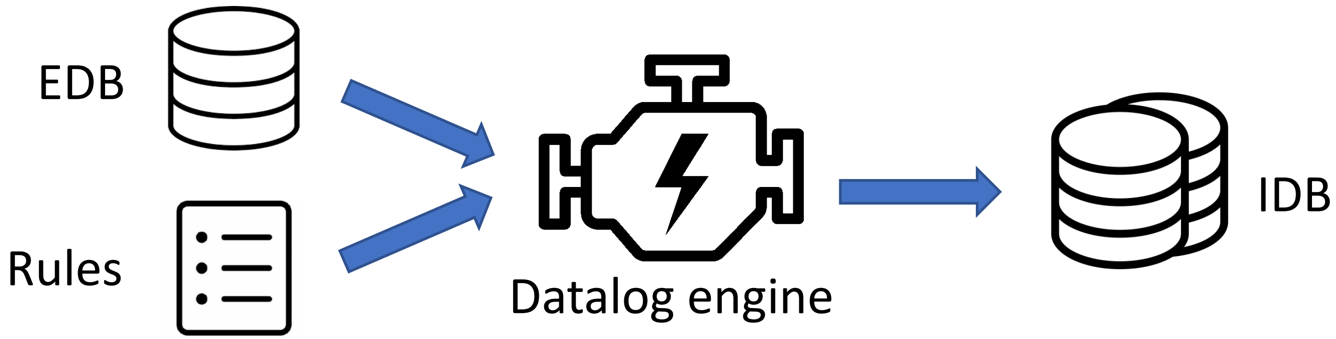
Execution of Datalog Programs



- 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

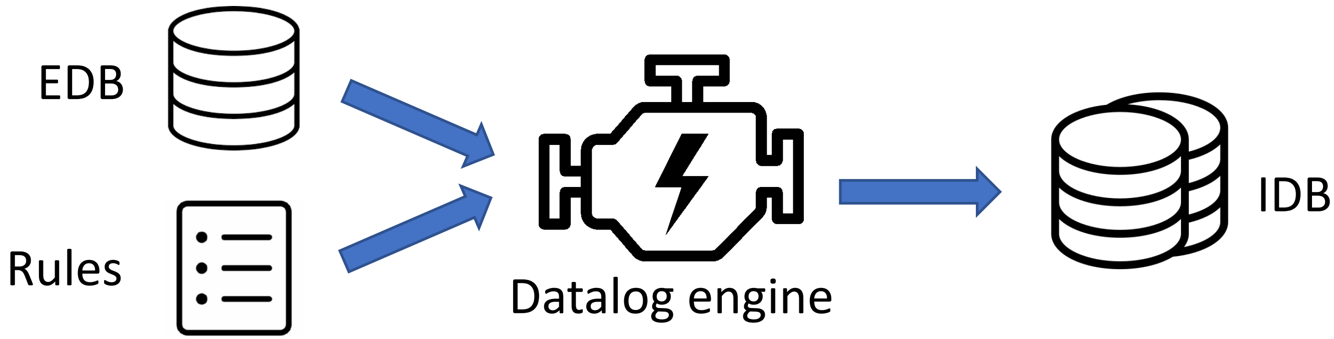


- Datalog engine deduces facts by given rules and EDB predicates until no new facts can be deduced. Some modern Datalog engines

  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

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

New	`x = new T()`	First focus on these statements (suppose the program has just one method)
Assign	`x = y`	
Store	`x.f = y`	
Load	`y = x.f`	
Call	`r = x.k(a, …)`	Then discuss method calls later

Datalog Model for Pointer Analysis

Kind	Statement
New	`i: x = 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 $\texttt{VarPointsTo(}x\texttt{,}o_i\texttt{)}$ represents $o_i \in pt(x)$

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

e.g., fact $\texttt{FieldsPointsTo(}o_i\texttt{,}f\texttt{,}o_j\texttt{)}$ represents $o_j \in pt(o_i.f)$

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
e = d.f;
```

Variables: V

Fields: F

Objects: O

An Example

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

New	
b	o_1
c	o_3

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
e = d.f;
```

Variables: V

Fields: F

Objects: O

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
e = d.f;
```

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

New	
b	o_1
c	o_3

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

Assign	
a	b
d	c

Variables:	V
Fields:	F
Objects:	O

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
e = d.f;
```

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

Variables:	V
Fields:	F
Objects:	O

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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 = new T()`	$\dfrac{}{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
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 = 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 = new T()`	$\overline{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
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 = new T()`	$\overline{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
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 = new T()`	$\dfrac{}{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
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

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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	
b	o_1
c	o_3

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

FieldPointsTo

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

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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	
b	o_1
c	o_3

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_1
c	o_3

FieldPointsTo

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

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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_1
c	o_3

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_1
c	o_3
a	o_1

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

FieldPointsTo

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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	
b	o_1
c	o_3

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_1
c	o_3
a	o_1
d	o_3

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	
b	o_1
c	o_3

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_1
c	o_3
a	o_1
d	o_3

FieldPointsTo		
o_3	f	o_1

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

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

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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	
b	o_1
c	o_3

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_1
c	o_3
a	o_1
d	o_3

FieldPointsTo		
o_3	f	o_1
o_3	f	o_3

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

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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	
b	o_1
c	o_3

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_1
c	o_3
a	o_1
d	o_3
e	o_1
e	o_3

FieldPointsTo		
o_3	f	o_1
o_3	f	o_3

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

An Example

```
b = new C();
a = b;
c = new C();
c.f = a;
d = c;
c.f = d;
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	
b	o_1
c	o_3

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_1
c	o_3
a	o_1
d	o_3
e	o_1
e	o_3

FieldPointsTo		
o_3	f	o_1
o_3	f	o_3

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

Handle Method Calls

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k}) \quad o_u \in pt(aj), 1 \le j \le n \quad o_v \in pt(m_{ret})}{o_i \in pt(m_{this}) \quad o_u \in pt\left(m_{pj}\right), 1 \le j \le n \quad o_v \in pt(r)}$

EDB

- VCall(l:**S**, x:**V**, k:**M**)
- Dispatch(o:**O**, k:**M**, m:**M**)
- ThisVar(m:**M**, this:**V**)

IDB

- Reachable(m:**M**)
- CallGraph(l:**S**, m:**M**)

Statements (Labels):	S
Methods:	M

Handle Method Calls

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k}) \quad o_u \in pt(aj), 1 \le j \le n \quad o_v \in pt(m_{ret})}{o_i \in pt(m_{this}) \quad o_u \in pt(m_{pj}), 1 \le j \le n \quad o_v \in pt(r)}$

EDB
- `VCall(l:`**S**`, x:`**V**`, k:`**M**`)`
- `Dispatch(o:`**O**`, k:`**M**`, m:`**M**`)`
- `ThisVar(m:`**M**`, this:`**V**`)`

IDB
- `Reachable(m:`**M**`)`
- `CallGraph(l:`**S**`, m:`**M**`)`

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

Statements (Labels):	S
Methods:	M

Handle Method Calls

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k}) \quad o_u \in pt(aj), 1 \le j \le n \quad o_v \in pt(m_{ret})}{o_i \in pt(m_{this}) \quad o_u \in pt(m_{pj}), 1 \le j \le n \quad o_v \in pt(r)}$

EDB
- Argument(l:**S**, i:**N**, ai:**V**)
- Parameter(m:**M**, i:**N**, pi:**V**)

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

Statements (Labels):	S
Methods:	M
Nature numbers (indexes)	N

Handle Method Calls

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$$\frac{o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k}) \quad o_u \in pt(aj), 1 \le j \le n \quad \Rightarrow o_v \in pt(m_{ret})}{o_i \in pt(m_{this}) \quad o_u \in pt(m_{pj}), 1 \le j \le n \quad \Rightarrow o_v \in pt(r)}$$

EDB
- `MethodReturn(m:`**M**`, ret:`**V**`)`
- `CallReturn(l:`**S**`, r:`**V**`)`

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

Statements (Labels):	S
Methods:	M

Handle Method Calls

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$$\frac{o_i \in pt(x),\ m = \text{Dispatch}(o_i, \mathrm{k}) \quad o_u \in pt(aj), 1 \le j \le n \quad o_v \in pt(m_{ret})}{o_i \in pt(m_{this}) \quad o_u \in pt(m_{pj}), 1 \le j \le n \quad o_v \in pt(r)}$$

```
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

Datalog Model for Taint Analysis

On top of pointer analysis

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

Taint Analysis via Datalog

- Handles sources (generates tainted data)

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{l \rightarrow m \in CG \quad m \in Sources}{t_l \in pt(r)}$

```
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(a1,…,an)`	$\dfrac{l \rightarrow m \in CG \quad \langle m, i \rangle \in Sinks \quad t_j \in pt(ai)}{\langle j, l, i \rangle \in TaintFlows}$

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