Static Program Analysis

Datalog-Based Program Analysis

Nanjing University

Tian Tan

2020
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)

```java
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)

```java
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)

```sql
SELECT * FROM Persons WHERE Age >= 18;
```
# How to Implement Program Analyses?

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new } T() )</td>
<td>( o_i \in pt(x) )</td>
<td></td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
<td>( o_i \in pt(y) ) ( o_i \in pt(x) )</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
<td>( o_i \in pt(x) ) ( o_j \in pt(y) ) ( o_j \in pt(o_i.f) )</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
<td>( o_i \in pt(x) ) ( o_j \in pt(o_i.f) ) ( o_j \in pt(y) )</td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td>( L: r = x.k(a_1, \ldots, a_n) )</td>
<td>( 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_{\text{ret}}) ) ( o_i \in pt(m_{\text{this}}) ) ( o_u \in pt(m_{p_j}), 1 \leq j \leq n ) ( o_v \in pt(r) )</td>
<td></td>
</tr>
</tbody>
</table>
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($o_i.f, y$)
   ProcessCall($x, o_i$)

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

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($m$)
   foreach parameter $p_i$ of $m$ do
      AddEdge($a_i, p_i$)
      AddEdge($m_{ret}, r$)
   if $ps$ is not empty then
      $pt(n) \cup= pts$
   foreach $n \rightarrow s \in PFG$ do
      add $\langle s, pts \rangle$ to $WL$
Solve\((m_{\text{entry}})\)
\[
W L = [], P F G = \{\}, S = \{\}, R M = \{\}, C G = \{}
\]

AddReachable\((m_{\text{entry}})\)

while \(W L\) is not empty do
\[
\text{remove } \langle n, p t s \rangle \text{ from } W L
\]
\[
\Delta = p t s - p t(n)
\]

Propagate\((n, \Delta)\)

if \(n\) represents a variable \(x\) then
\[
\text{foreach } o_i \in \Delta \text{ do}
\]
\[
\text{foreach } x.f = y \in S \text{ do}
\]
\[
\text{AddEdge}(y, o_i.f)
\]
\[
\text{foreach } y = x.f \in S \text{ do}
\]
\[
\text{AddEdge}(o_i.f, y)
\]

AddEdge\((s, t)\)

if \(s \to t \notin P F G\) then
\[
\text{add } s \to t \text{ to } P F G
\]

if \(p t(s)\) is not empty then
\[
\text{add } \langle t, p t(s) \rangle \text{ to } W L
\]

AddReachable\((m)\)

if \(m \notin R M\) then
\[
\text{add } m \text{ to } R M
\]

\(S \cup \{m\} = S_m\)

foreach \(i: x = \text{new} T()\) \(\in S_m\) do
\[
\text{add } x, \{o_i\} \text{ to } W L
\]

foreach \(x = y \in S_m\) do
\[
\text{AddEdge}(y, x)
\]

AddEdge\((m, r)\)

if \(l \to m \notin C G\) then
\[
\text{add } l \to m \text{ to } C G
\]

foreach \(m_{\text{reachable}}\) do
\[
\text{AddEdge}(a_i, p_i)
\]

AddEdge\((m_{\text{ret}}, r)\)

---

How to implement \textit{worklist}?
- Array list or linked list?
- Which worklist entry should be processed first?
**Pointer Analysis, Imperative Implementation**

```
Solve(m_{entry})
    WL=[], PFG={}, S={}, RM={}, CG={}
AddReachable(m_{entry})
    while WL is not empty do
        remove ⟨n, pts⟩ from WL
        Δ = pts − pt(n)
        Propagate(n, Δ)
        if n represents a variable x then
            foreach o_i ∈ Δ do
                foreach x.f = y ∈ S do
                    AddEdge(y, o_i.f)
                foreach y = x.f ∈ S do
                    AddEdge(o_i.f, y)
        ProcessCall(x, o_i)

AddReachable(m)
    if m ∉ RM then
        add m to RM
        S ∪ = S_m
        foreach i: x = new T() ∈ S_m do
            add x, {o_i} to WL
        foreach x = y ∈ S_m do
            AddEdge(y, x)

Propagate(n, pts)
    if pts is not empty then
        pt(n) U= pts
        foreach n → s ∈ PFG do
            add ⟨s, pts⟩ to WL
    if s → t ∉ PFG then
        add s → t to PFG
        if pt(s) is not empty then
            add ⟨t, pt(s)⟩ to WL
```

- 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?
**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(o_i.f, y)
      ProcessCall(x, o_i)
  AddReachable(m)
    if m \notin RM then
      add m to RM
      \( S \cup M = S_m \)
      foreach i: x = new T() \in S_m do
        add \langle x, \{o_i\} \rangle to WL
      foreach x = y \in S_m do
        AddEdge(y, x)
      AddEdge(s, t)
        if s \rightarrow t \notin PFG then
          add s \rightarrow t to PFG
        if pt(s) is not empty then
          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
          \)
```

- 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?
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($o_i.f, y$)

ProcessCall($x, o_i$)

AddReachable($m$)

if $m \notin RM$ then

add $m$ to $RM$

$S \cup \{m\} = S_m$

foreach $i$ : $x = \text{new } T() \in S_m$ do

add $\langle x, \{o_i\} \rangle$ to $WL$

foreach $x = y \in S_m$ do

AddEdge($y, x$)

if $s \rightarrow t \notin PFG$ then

add $s \rightarrow t$ to $PFG$

foreach $n \rightarrow s \in PFG$ do

if $pt(s)$ is not empty then

AddEdge($t, pt(s)$)
Solve($m^{entry}$)

$WL = [], PFG = \{\}, S = \{\}, RM = \{\}, CG = \{\}$

Add reachable($m^{entry}$)

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
        Add Edge($y, o_i.f$)
      foreach $y = x.f \in S$ do
        Add Edge($o_i.f, y$)
  Process Call($x, o_i$)

Add reachable($m$)

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

Add Edge($s, t$)

if $s \rightarrow t \notin PFG$ then
  if $pts$ is not empty then
    $pt(n)$
    add $s \rightarrow t$ to $PFG$

Add Edge($m_{ret}, r$)

- 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)

\[
\text{VarPointsTo}(x, o) \leftarrow \\
\quad \text{Reachable}(m), \\
\quad \text{New}(x, o, m).
\]

\[
\text{VarPointsTo}(x, o) \leftarrow \\
\quad \text{Assign}(x, y), \\
\quad \text{VarPointsTo}(y, o).
\]

\[
\text{FieldPointsTo}(oi, f, oj) \leftarrow \\
\quad \text{Store}(x, f, y), \\
\quad \text{VarPointsTo}(x, oi), \\
\quad \text{VarPointsTo}(y, oj).
\]

\[
\text{VarPointsTo}(y, oj) \leftarrow \\
\quad \text{Load}(y, x, f), \\
\quad \text{VarPointsTo}(x, oi), \\
\quad \text{FieldPointsTo}(oi, f, oj).
\]

\[
\text{VarPointsTo}(this, o), \\
\quad \text{Reachable}(m), \\
\quad \text{CallGraph}(l, m) \leftarrow \\
\quad \quad \text{VCall}(l, x, k), \\
\quad \quad \text{VarPointsTo}(x, o), \\
\quad \quad \text{Dispatch}(o, k, m), \\
\quad \quad \text{ThisVar}(m, this).
\]

\[
\text{VarPointsTo}(pi, o) \leftarrow \\
\quad \text{CallGraph}(l, m), \\
\quad \text{Argument}(l, i, ai), \\
\quad \text{Parameter}(m, i, pi), \\
\quad \text{VarPointsTo}(ai, o).
\]

\[
\text{VarPointsTo}(r, o) \leftarrow \\
\quad \text{CallGraph}(l, m), \\
\quad \text{MethodReturn}(m, ret), \\
\quad \text{VarPointsTo}(ret, o), \\
\quad \text{CallReturn}(l, r),
\]
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

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
  • ...

Datalog

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

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

Datalog = \textbf{Data} + \textbf{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 \textbf{table} of data

<table>
<thead>
<tr>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

\textbf{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

<table>
<thead>
<tr>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

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

\[ P(X_1, X_2, \ldots, X_n) \]

- Terms
  - Variables: stand for any values
  - Constants
Atoms

• Atoms are basic elements of Datalog, which represent predicates of the form $P(X_1, X_2, \ldots, X_n)$

• Terms
  • Variables: stand for any values
  • Constants

• Examples
  • Age(person, age)
  • Age("Xiaoming", 18)
Atoms (Cont.)

• $P(X_1, X_2, \ldots, X_n)$ is called relation atom

• $P(X_1, X_2, \ldots, X_n)$ evaluates to true when predicate $P$ contains the tuple described by $X_1, X_2, \ldots, X_n$
Atoms (Cont.)

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

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>
Atoms (Cont.)

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

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Xiaohong</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Alan</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Abao</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>
Atoms (Cont.)

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

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>
Atoms (Cont.)

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

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

<table>
<thead>
<tr>
<th>Person</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>
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

\[ H \leftarrow B_1, B_2, \ldots, B_n. \]
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

\[ H \leftarrow B_1, B_2, \ldots, B_n. \]

- **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 B₁, B₂, ..., Bₙ is true if all subgoals B₁, B₂, ..., and Bₙ are true.

For example, we can deduce adults via Datalog rule:

\[
\text{Adult(person) ← \text{Age(person, age)}, \quad \text{age} \geq 18.}
\]
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**

\[ H(X_1, X_2) \leftarrow B_1(X_1, X_3), B_2(X_2, X_4), \ldots, B_n(X_m). \]

- 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.

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{Adult(person) } \leftarrow \text{Age(person,age), age } \geq 18.
\]

\[
\text{Adult("Xiaoming") } \leftarrow \text{Age("Xiaoming",18),18}\geq18.
\]

\[
\text{Adult("Xiaohong") } \leftarrow \text{Age("Xiaohong",23),23}\geq18.
\]
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

<table>
<thead>
<tr>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{Adult(person) } \leftarrow \text{Age(person,age), age } \geq 18.
\]

\[
\text{Adult(“Xiaoming”) } \leftarrow \text{Age(“Xiaoming”,18),18}\geq18.
\]

\[
\text{Adult(“Xiaohong”) } \leftarrow \text{Age(“Xiaohong”,23),23}\geq18.\text{Age(“Alan”,16),16}\geq18.
\]
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

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>XiaoHong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Adult</th>
<th>person</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
</tr>
<tr>
<td></td>
<td>XiaoHong</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
</tr>
</tbody>
</table>
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**

Where does initial data come from?

<table>
<thead>
<tr>
<th>Age</th>
<th>person</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xiaoming</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Xiaohong</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Alan</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Abao</td>
<td>31</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>person</th>
<th>hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiaoming</td>
<td>cooking</td>
</tr>
<tr>
<td>Xiaoming</td>
<td>singing</td>
</tr>
<tr>
<td>Xiaohong</td>
<td>jogging</td>
</tr>
<tr>
<td>Abao</td>
<td>sleeping</td>
</tr>
<tr>
<td>Alan</td>
<td>swimming</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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., \( H \leftarrow A, (B;C) \).
Negation

\[ H(X_1, X_2) \leftarrow B_1(X_1, X_3), !B_2(X_2, X_4), \ldots, B_n(X_m). \]

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

In Datalog rules, a subgoal can be a negated atom, which negates its meaning.

Negated subgoal is written as \( !B(\ldots) \), and read as not \( B(\ldots) \).

For example, to compute the students who need to take a make-up exam, we can write

\[
\text{MakeupExamStd}(\text{student}) \leftarrow \text{Student}(\text{student}), \neg \text{PassedStd}(\text{student}).
\]

Where \text{Student} stores all students, and \text{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 \texttt{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?

\[ A(x) \leftarrow B(y), \ x > y. \]
\[ A(x) \leftarrow B(y), \neg C(x, y). \]

For both rules, infinite values of \( x \) can satisfy the rule, which makes \( A \) an \textit{infinite relation}.

\[ A(x) \leftarrow B(y), \ x > y. \]
\[ A(x) \leftarrow B(y), \neg C(x, y). \]
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 \textit{infinite relation}.

- A rule is \textit{safe} if every variable appears in at least one non-negated relational atom
- Above two rules are \textit{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?  

\[ A(x) \leftarrow B(x), \lnot A(x) \]

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

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 analysis rules

New:  \( x = \text{new} \ T() \)
Assign:  \( x = y \)
Store:  \( x.f = y \)
Load:  \( y = x.f \)
Call:  \( r = x.k(a, \ldots) \)

First focus on these statements (suppose the program has just one method)
Then discuss method calls later
# Datalog Model for Pointer Analysis

## Kind Statement

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new } T() )</td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
</tr>
</tbody>
</table>

## 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 \( \text{VarPointsTo}(x, o_i) \) represents \( o_i \in pt(x) \)
- FieldPointsTo\( (o_i: O, f: F, o_j: O) \)
  e.g., fact \( \text{FieldPointsTo}(o_i, f, o_j) \) represents \( o_j \in pt(o_i.f) \)

---

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;

Variables: V
Fields: F
Objects: O
An Example

```java
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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>o₁</td>
</tr>
<tr>
<td>c</td>
<td>o₃</td>
</tr>
</tbody>
</table>
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;`

```
<table>
<thead>
<tr>
<th>New(x : V, o : O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>o_1</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>o_3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assign(x : V, y : V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>
```

Variables: V
Fields: F
Objects: O
An Example

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

Variables: \( V \)
Fields: \( F \)
Objects: \( O \)
An Example

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

Variables: \texttt{V}
Fields: \texttt{F}
Objects: \texttt{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 : V, o : O)

<table>
<thead>
<tr>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

Assign(x : V, y : V)

<table>
<thead>
<tr>
<th>Assign</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>d</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
</tr>
</tbody>
</table>
### Datalog Rules for Pointer Analysis

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new } T() )</td>
<td>( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
<td>( o_i \in pt(y) ) ( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
<td>( o_i \in pt(x) ) ( o_j \in pt(y) ) ( o_j \in pt(o_i.f) )</td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
<td>( o_i \in pt(x) ) ( o_j \in pt(o_i.f) ) ( o_j \in pt(y) )</td>
</tr>
</tbody>
</table>

\text{VarPointsTo}(x, o) \leftarrow \text{New}(x, o). \text{VarPointsTo}(x, o) \leftarrow \text{Assign}(x, y), \text{VarPointsTo}(y, o). \text{FieldPointsTo}(oi, f, oj) \leftarrow \text{Store}(x, f, y), \text{VarPointsTo}(x, oi), \text{VarPointsTo}(y, oj). \text{VarPointsTo}(y, oj) \leftarrow \text{Load}(y, x, f), \text{VarPointsTo}(x, oi), \text{FieldPointsTo}(oi, f, oj).
Datalog Rules for Pointer Analysis

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>i: x = new T()</td>
<td>( o_i \in pt(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(x, o) \leftarrow \text{New}(x, o) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(x, o) \leftarrow \text{Assign}(x, y) ),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(y, o) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{FieldPointsTo}(oi, f, oj) \leftarrow \text{Store}(x, f, y)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(x, oi) ),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(y, oj) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(y, oj) \leftarrow \text{Load}(y, x, f) ),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{VarPointsTo}(x, oi) ),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{FieldPointsTo}(oi, f, oj) ).</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
# Datalog Rules for Pointer Analysis

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>$i: x = \text{new } T()$</td>
<td>$o_i \in \text{pt}(x)$</td>
</tr>
</tbody>
</table>
| Assign | $x = y$   | $o_i \in \text{pt}(y)$  
$o_i \in \text{pt}(x)$ |
| Store  | $x.f = y$ | $o_i \in \text{pt}(x)$  
$o_j \in \text{pt}(y)$  
$o_j \in \text{pt}(o_i.f)$ |
| Load   | $y = x.f$ | $o_i \in \text{pt}(x)$  
$o_j \in \text{pt}(o_i.f)$  
$o_j \in \text{pt}(y)$ |

VarPointsTo($x, o$) $\leftarrow$ New($x, o$).

VarPointsTo($x, o$) $\leftarrow$ Assign($x, y$), VarPointsTo($y, o$).

FieldPointsTo($o_i, f, o_j$) $\leftarrow$ Store($x, f, y$), VarPointsTo($x, o_i$), VarPointsTo($y, o_j$).

VarPointsTo($y, o_j$) $\leftarrow$ Load($y, x, f$), VarPointsTo($x, o_i$), FieldPointsTo($o_i, f, o_j$).
## Datalog Rules for Pointer Analysis

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>$i: x = \text{new } T()$</td>
<td>$o_i \in pt(x)$</td>
</tr>
<tr>
<td>Assign</td>
<td>$x = y$</td>
<td>$o_i \in pt(y)$, $o_i \in pt(x)$</td>
</tr>
<tr>
<td>Store</td>
<td>$x.f = y$</td>
<td>$o_i \in pt(x)$, $o_j \in pt(y)$, $o_j \in pt(o_i.f)$</td>
</tr>
<tr>
<td>Load</td>
<td>$y = x.f$</td>
<td>$o_i \in pt(x)$, $o_j \in pt(o_i.f)$, $o_j \in pt(y)$</td>
</tr>
</tbody>
</table>

### Kind Statement Rule

- **New**
  
  $i: x = \text{new } T()$
  
  $o_i \in pt(x)$

- **Assign**
  
  $x = y$
  
  $o_i \in pt(y)$, $o_i \in pt(x)$

- **Store**
  
  $x.f = y$
  
  $o_i \in pt(x)$, $o_j \in pt(y)$, $o_j \in pt(o_i.f)$

- **Load**
  
  $y = x.f$
  
  $o_i \in pt(x)$, $o_j \in pt(o_i.f)$, $o_j \in pt(y)$

### Examples

- **VarPointsTo**
  
  $\text{VarPointsTo}(x, o) \leftarrow \text{New}(x, o)$

- **VarPointsTo**
  
  $\text{VarPointsTo}(x, o) \leftarrow \text{Assign}(x, y), \text{VarPointsTo}(y, o)$

- **FieldPointsTo**
  
  $\text{FieldPointsTo}(o_i, f, o_j) \leftarrow \text{Store}(x, f, y), \text{VarPointsTo}(x, o_i), \text{VarPointsTo}(y, o_j)$

- **VarPointsTo**
  
  $\text{VarPointsTo}(y, o_j) \leftarrow \text{Load}(y, x, f), \text{VarPointsTo}(x, o_i), \text{FieldPointsTo}(o_i, f, o_j)$
## Datalog Rules for Pointer Analysis

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>i: x = new T()</td>
<td>oᵢ ∈ pt(x)</td>
</tr>
<tr>
<td>Assign</td>
<td>x = y</td>
<td>oᵢ ∈ pt(y)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oᵢ ∈ pt(x)</td>
</tr>
<tr>
<td>Store</td>
<td>x.f = y</td>
<td>oᵢ ∈ pt(x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oⱼ ∈ pt(y)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oⱼ ∈ pt(oᵢ.f)</td>
</tr>
<tr>
<td>Load</td>
<td>y = x.f</td>
<td>oᵢ ∈ pt(x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oⱼ ∈ pt(oᵢ.f)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oⱼ ∈ pt(y)</td>
</tr>
</tbody>
</table>

- **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).

VarPointsTo(v:V, o:O) <-

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

Assign(x:V, y:V)

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

VarPointsTo(v:V, o:O)

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

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

\textbf{VarPointsTo}(x, o) \leftarrow \textbf{New}(x, o).

\textbf{VarPointsTo}(x, o) \leftarrow \textbf{Assign}(x, y), \textbf{VarPointsTo}(y, o).

\textbf{FieldPointsTo}(oi, f, oj) \leftarrow \textbf{Store}(x, f, y), \textbf{VarPointsTo}(x, oi), \textbf{VarPointsTo}(y, oj).

\textbf{VarPointsTo}(y, oj) \leftarrow \textbf{Load}(y, x, f), \textbf{VarPointsTo}(x, oi), \textbf{FieldPointsTo}(oi, f, oj).
An Example

```java
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).
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).
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).
An Example

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

VarPointsTo\((x, o) \leftarrow \text{New}(x, o)\).

VarPointsTo\((x, o) \leftarrow \text{Assign}(x, y), \text{VarPointsTo}(y, o)\).

\[
\begin{array}{c|c}
\text{VarPointsTo} & \text{FieldPointsTo} \\
\hline
b & o_1 \\
\hline
c & o_3 \\
\hline
a & o_1 \\
\hline
d & o_3 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{FieldPointsTo} & \text{VarPointsTo} \\
\hline
o_3 & f & o_1 \\
\hline
o_3 & f & o_3 \\
\end{array}
\]
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).
An Example

```java
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;
```

<table>
<thead>
<tr>
<th>VarPointsTo(x, o) &lt;-</th>
</tr>
</thead>
<tbody>
<tr>
<td>New(x: V, o: O)</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FieldPointsTo(oi: O, f: F, oj: O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₃</td>
</tr>
<tr>
<td>o₃</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store(x: V, f: F, y: V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assign(x: V, y: V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load(x: V, y: V, f: F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
</tr>
</tbody>
</table>

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).
## Handle Method Calls

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call | \( l: r = x.k(a_1, \ldots, a_n) \) | \( o_i \in pt(x), m = Dispatch(o_i, k) \)  
\( o_u \in pt(a_j), 1 \leq j \leq n \)  
\( o_v \in pt(m_{ret}) \)  
\( o_i \in pt(m_{this}) \)  
\( o_u \in pt(m_{pj}), 1 \leq j \leq n \)  
\( o_v \in pt(r) \) |

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

### IDB
- \( \text{Reachable}(m:M) \)
- \( \text{CallGraph}(l:S, m:M) \)

### Statements
- (Labels): \( S \)
- Methods: \( M \)
# Handle Method Calls

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call     | $l: r = x.k(a_1, \ldots, a_n)$ | $o_i \in pt(x), m = Dispatch(o_i, k)$
|          |           | $o_u \in pt(a_j), 1 \leq j \leq n$ |
|          |           | $o_v \in pt(m_{ret})$ |
|          |           | $o_i \in pt(m_{this})$ |
|          |           | $o_u \in pt(m_{pj}), 1 \leq j \leq n$ |
|          |           | $o_v \in pt(r)$ |

**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

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call | \( l: r = x.k(a_1, \ldots, a_n) \) | \( o_i \in pt(x), m = \text{Dispatch}(o_i, k) \)  
\( o_u \in pt(a_j), 1 \leq j \leq n \)  
\( o_v \in pt(m_{ret}) \)  
\( o_i \in pt(m_{this}) \)  
\( o_u \in pt(m_{pj}), 1 \leq j \leq n \)  
\( o_v \in pt(r) \) |

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

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

VarPointsTo(\( pi, o \)) <- CallGraph(\( l, m \)), Argument(\( l, i, ai \)), Parameter(\( m, i, pi \)), VarPointsTo(\( ai, o \)).
Handle Method Calls

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call | \( l: r = x.k(a_1, \ldots, a_n) \) | \( o_i \in pt(x), \quad m = \text{Dispatch}(o_i, k) \)  
\( o_u \in pt(a_j), 1 \leq j \leq n \)  
\( o_v \in pt(m_{ret}) \)  
\( o_i \in pt(m_{this}) \)  
\( o_u \in pt(m_{pj}), 1 \leq j \leq n \)  
\( o_v \in pt(r) \) |

EDB
- MethodReturn(\( m: M \), \( \text{ret}: V \))
- 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

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call   | $l: r = x.k(a_1, ..., a_n)$ | $o_i \in \text{pt}(x), m = \text{Dispatch}(o_i, k)$  
$o_u \in \text{pt}(a_j), 1 \leq j \leq n$  
$\frac{o_v \in \text{pt}(m_{ret})}{o_i \in \text{pt}(m_{this})}$  
$\frac{o_u \in \text{pt}(m_{pj}), 1 \leq j \leq n}{o_v \in \text{pt}(r)}$ |

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

\[ \text{Reachable}(m) \leftarrow \\text{EntryMethod}(m). \]

\[ \text{VarPointsTo}(x, o) \leftarrow \text{Reachable}(m), \text{New}(x, o, m). \]

\[ \text{VarPointsTo}(x, o) \leftarrow \text{Assign}(x, y), \text{VarPointsTo}(y, o). \]

\[ \text{FieldPointsTo}(oi, f, oj) \leftarrow \text{Store}(x, f, y), \text{VarPointsTo}(x, oi), \text{VarPointsTo}(y, oj). \]

\[ \text{VarPointsTo}(y, oj) \leftarrow \text{Load}(y, x, f), \text{VarPointsTo}(x, oi), \text{FieldPointsTo}(oi, f, oj). \]

\[ \text{VarPointsTo}(\text{this}, o), \ \text{Reachable}(m), \ \text{CallGraph}(l, m) \leftarrow \text{VCall}(l, x, k), \text{VarPointsTo}(x, o), \text{Dispatch}(o, k, m), \text{ThisVar}(m, \text{this}). \]

\[ \text{VarPointsTo}(pi, o) \leftarrow \text{CallGraph}(l, m), \text{Argument}(l, i, ai), \text{Parameter}(m, i, pi), \text{VarPointsTo}(ai, o). \]

\[ \text{VarPointsTo}(r, o) \leftarrow \text{CallGraph}(l, m), \text{MethodReturn}(m, ret), \text{VarPointsTo}(ret, o), \text{CallReturn}(l, r). \]
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**
  - Source(m : M) // source methods
  - Sink(m : M) // sink methods
  - Taint(l : S, t : T) // associates each call site to the tainted data from the call site

- **IDB predicate**
  - TaintFlow(t : T, m : M) // detected taint flows, e.g., TaintFlow(t,m) denotes that tainted data t may flow to sink method m
Taint Analysis via Datalog

• Handles sources (generates tainted data)

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call | \( l: r = x.k(a_1,...,a_n) \) | \( l \rightarrow m \in CG \)
\( m \in \text{Sources} \)
\( t_i \in pt(r) \)

\[
\text{VarPointsTo}(r, t) \leftarrow \text{CallGraph}(l, m), \text{Source}(m), \text{CallReturn}(l, r), \text{Taint}(l, t).
\]

• Handles sinks (generates taint flow information)

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Call | \( l: r = x.k(a_1,...,a_n) \) | \( l \rightarrow m \in CG \)
\( m \in \text{Sinks} \)
\( \exists i, 1 \leq i \leq n: t_j \in pt(a_i) \)
\( \langle t_j, m \rangle \in \text{TaintFlows} \)

\[
\text{TaintFlow}(t, m) \leftarrow \text{CallGraph}(l, m), \text{Sink}(m), \text{Argument}(l, _, a_i), \text{VarPointsTo}(a_i, t), \text{Taint}(\_, t).
\]
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
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