Specialized Static Analysis Framework: Datalog Analysis

Woosuk Lee

CSE 6049 Program Analysis

Hanyang University, Korea

Some slides are borrowed from http://rightingcode.org/slides/7.pdf
Goal of This Lecture

- Learn practical alternatives to the aforementioned general, abstract interpretation framework
- For simple languages and properties, there are frameworks that are simple yet powerful enough
- But with several limitations
Static Analysis by Monotonic Closure

- Static analysis = setting up initial facts then collecting new facts by a kind of chain reaction
  - has rules for collecting initial facts
  - has rules for generating new facts from existing facts
- the initial facts immediate from the program text
- the chain reaction steps simulate the program semantics
- the universe of facts are finite for each program
- analysis accumulates facts until no more possible
Representative Example: Pointer Analysis

Reasoning about any real programs needs pointer reasoning: e.g.,

```c
x = 1;
y = 2;
*p = 3;
*q = 4;
```

What is the value of \( x + y \) after the last statement?

- \( p = &x \) and \( q = &y \): 
- \( p = &x \) and \( q \neq &y \): 
- \( p \neq &x \) and \( q = &y \): 
- \( p \neq &x \) and \( q \neq &y \):
Pointer Analysis

• Static program analysis that computes the set of memory locations (objects) that a pointer variable may point to at runtime.

• One of the most important static analyses: all interesting questions on program reasoning eventually need pointer analysis.

• E.g., control-flows, data-flows, types, information-flows, etc
Example: (Flow-insensitive) Pointer Analysis

\[
P ::= C \quad \text{program}
\]
\[
C ::= L := R \quad \text{assignment}
\]
\[
| \quad L := R \quad \text{assignment}
\]
\[
| \quad C ; C \quad \text{sequence}
\]
\[
L ::= x \mid *x \quad \text{target to assign to}
\]
\[
R ::= n \mid x \mid *x \mid &x \quad \text{value to assign}
\]

- Goal: estimate all “points-to” relations between variables that can occur during executions
- \(a \rightarrow b\): variable \(a\) can point to (can have the address of) variable \(b\)
Rules

• The analysis globally collects the set of possible points-to facts that can happen during the program execution.

• Starting from the empty set, we apply rules of the following form to add new facts to the global set. This collection (hence the analysis) terminates when no more addition is possible.

• This collection terminates when no more addition is possible.
Rules for Pointer Analysis

The initial facts that are obvious from the program text are collected by this rule:

\[
\frac{x := \&y}{x \rightarrow y}
\]

The chain-reaction rules are as follows for other cases of assignments:

\[
\frac{x := y \quad y \rightarrow z}{x \rightarrow z}
\]

\[
\frac{x := \ast y \quad y \rightarrow z \quad z \rightarrow w}{x \rightarrow w}
\]

\[
\frac{\ast x := y \quad x \rightarrow w \quad y \rightarrow z}{w \rightarrow z}
\]

\[
\frac{\ast x := \ast y \quad x \rightarrow w \quad y \rightarrow z \quad z \rightarrow v}{w \rightarrow v}
\]

\[
\frac{\ast x := \&y \quad x \rightarrow w}{w \rightarrow y}
\]
Rules for Pointer Analysis

The initial facts that are obvious from the program text are collected by this rule:

\[
x := \&y \\
x \rightarrow y
\]

The chain-reaction rules are as follows for other cases of assignments:

\[
x := y \quad y \rightarrow z \\
x \rightarrow z
\]

\[
x := \*y \quad y \rightarrow z \quad z \rightarrow w \\
x \rightarrow w
\]

\[
\*x := y \quad x \rightarrow w \quad y \rightarrow z \\
w \rightarrow z
\]

\[
\*x := \*y \quad x \rightarrow w \quad y \rightarrow z \quad z \rightarrow v \\
w \rightarrow v
\]

\[
\*x := \&y \quad x \rightarrow w \\
w \rightarrow y
\]

\[
\*x := \*y \quad \text{Syntactic sugar:}
\]

Can be transformed to

t := \&y; \*x := t for a new temp var t
Example

Example (Pointer analysis steps)

```c
x := &a; y := &x;
while B
    *y := &b;
    *x := *y
```

- Initial facts are from the first two assignments:
  ```
  x \rightarrow a, \ y \rightarrow x
  ```

- From \( y \rightarrow x \) and the while-loop body, add
  ```
  x \rightarrow b
  ```

- From the last assignment:
  ```
  - from \( x \rightarrow a \) and \( y \rightarrow x \), add \( a \rightarrow a \)
  - from \( x \rightarrow b \) and \( y \rightarrow x \), add \( b \rightarrow b \)
  - from \( x \rightarrow a, y \rightarrow x, \) and \( x \rightarrow b \), add \( a \rightarrow b \)
  - from \( x \rightarrow b, y \rightarrow x, \) and \( x \rightarrow a \), add \( b \rightarrow a \)
  ```
General Algorithm

- let \( R \) be the set of the chain-reaction rules
- let \( X_0 \) be the initial fact set
- let \( \text{Facts} \) be the set of all possible facts

Then, the analysis result is

\[
\bigcup_{i \geq 0} Y_i,
\]

where

\[
Y_0 = X_0,
\]
\[
Y_{i+1} = Y \text{ such that } Y_i \vdash_R Y.
\]

Or, equivalently, the analysis result is the least fixpoint

\[
\bigcup_{i \geq 0} \phi^i(\emptyset)
\]

of monotonic function \( \phi : \wp(\text{Facts}) \to \wp(\text{Facts}) : X \mapsto X_0 \cup (Y \text{ such that } X \vdash_R Y). \)
Static Analysis by Monotonic Closure as Datalog

• We can express the rules in **Datalog**.

• Datalog: a declarative logic programming language

• Not Turing-complete: Subset of Prolog, or SQL with recursion => efficient algorithms to evaluate Datalog programs

• Originated as query language for databases

• Later applied in many other domains: program analysis, data mining, network, security, …
Benefits of Using Datalog

• Separates analysis design from implementation
  
  • Analysis designer can focus on “what” rather than “how”

• By leveraging powerful, off-the-shelf solver engines
  
  • many implementations: Souffle, Bddbddb, Paddle, Logicblox, …
Syntax of Datalog

- A Datalog program is a sequence of constraints:

\[ P ::= \overline{c} \]

- A constraint consists of a head of a literal and a body of a list of literals:

\[ c ::= l : - \overline{l} \]

A constraint represents a horn clause (a disjunction of literals with at most one positive, unnegated, literal):

\[ l \lor \neg l_1 \lor \neg l_2 \lor \cdots \lor \neg l_n \iff l \leftarrow l_1 \land l_2 \land \cdots \land l_n \]

- A literal is a relation with arguments:

\[ l ::= r(\overline{a}) \]

where an argument is either a variable or constant.
Syntax of Datalog: Example

Input Relations:
edge(n:N, m:N)

Output Relations:
path(n:N, m:N)

Rules:
path(x, x).
path(x, z) :- path(x, y), edge(y, z).
Syntax of Datalog: Example

<table>
<thead>
<tr>
<th>Input Relations:</th>
<th>Output Relations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge(n:N, m:N)</td>
<td>path(n:N, m:N)</td>
</tr>
</tbody>
</table>

A relation is similar to a table in a database. A tuple in a relation is similar to a row in a table.

Rules:

- path(x, x).
- path(x, z) :- path(x, y), edge(y, z).
Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).
**Syntax of Datalog: Example**

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

Deductive rules that hold universally (i.e., variables like x, y, z can be replaced by any constant). Specify “if ... then ...” logic.
Syntax of Datalog: Example

Input Relations:
edge(n:N, m:N)

Output Relations:
path(n:N, m:N)

Rules:
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

(If TRUE,) there is a path from each node to itself.

If there is path from node x to y, and there is an edge from y to z, then there is path from x to z.
Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

---

<table>
<thead>
<tr>
<th>path((x, x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>path := { (x, x) \mid x \in N }</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>path := path \cup { (x, z) \mid \exists y \in N: (x, y) \in path \text{ and } (y, z) \in edge }</td>
</tr>
<tr>
<td>until path relation stops changing</td>
</tr>
</tbody>
</table>
Syntax of Datalog: Example

Input Relations:
edge(n:N, m:N)

Output Relations:
path(n:N, m:N)

Rules:
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

Input Tuples:
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

Output Tuples:
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
Syntax of Datalog: Example

**Input Relations:**
`edge(n:N, m:N)`

**Output Relations:**
`path(n:N, m:N)`

**Rules:**
`path(x, x).`
`path(x, z) :- path(x, y), edge(y, z).`

<table>
<thead>
<tr>
<th>Input Tuples:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>edge(0, 1)</code>, <code>edge(0, 2)</code>, <code>edge(2, 3)</code>, <code>edge(2, 4)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Tuples:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>path(0, 0)</code>, <code>path(1, 1)</code>, <code>path(2, 2)</code>, <code>path(3, 3)</code>, <code>path(4, 4)</code></td>
</tr>
<tr>
<td><code>path(0, 1)</code>, <code>path(0, 2)</code>, <code>path(2, 3)</code>, <code>path(2, 4)</code>, <code>path(0, 3)</code>, <code>path(0, 4)</code></td>
</tr>
</tbody>
</table>
Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

**Input Tuples:**
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

**Output Tuples:**
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

**Input Tuples:**
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

**Output Tuples:**
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
Formal Semantics of Datalog

- A Datalog program denotes a set of ground literals:
  \[[P]\] \in \wp(G)

  where \(G\) is the set of ground literals (literals without variables).

- A Datalog rule \(l : l_1, \ldots, l_n\) denotes the function:
  \[f_l : l_1, \ldots, l_n(X) = \{\sigma(l_0) \mid \sigma(l_k) \in X \text{ for } 1 \leq k \leq n\}\]

  where \(\sigma\) is a variable substitution.

- The semantics of \(P\) is defined as the least fixed point of \(F_P\):
  \[[P]\] = \text{lfp } F_P \text{ where } F_P(X) = X \cup \bigcup_{c \in P} f_c(X)

- The semantics is monotone:
  \[P_1 \subseteq P_2 \implies [P_1] \subseteq [P_2]\]
Program as Relations

- A program can be represented by a set of input relations:

  - \( x := &y \) — new \((x: X, y: X)\)
  
  - \( x := y \) — assign \((x: X, y: X)\)
  
  - \( x := *y \) — load \((x: X, y: X)\)
  
  - \( *x := y \) — store \((x: X, y: X)\)

where \( X \) is the set of variables
Target Properties as Relations

- Points-to facts can be represented as output relations
  - \( x \rightarrow y \ - \text{points}(x:X, \ y:X) \)
Datalog Rules

- Datalog rule for
  \[ \frac{x := \& y}{x \rightarrow y} \]

  \bullet \quad \text{points}(x, y) :- \text{new}(x, y).

- Datalog rule for
  \[ \frac{x := y \quad y \rightarrow z}{x \rightarrow z} \]

  \bullet \quad \text{points}(x, z) :- \text{assign}(x, y), \text{points}(y, z).
Datalog Rules

- Datalog rule for

\[ x := ^*y \quad y \to z \quad z \to w \]
\[ x \to w \]

- \( \text{points}(x, w) :\text{load}(x, y), \text{points}(y, z), \text{points}(z, w). \)

- Datalog rule for

\[ ^*x := y \quad x \to w \quad y \to z \]
\[ w \to z \]

- \( \text{points}(w, z) :\text{store}(x, y), \text{points}(x, w), \text{points}(y, z). \)
Extended Language for Functions

\[ \text{Statement} \quad C ::= \quad \cdots \quad | \quad y := f(x) \quad \text{function call} \quad | \quad \text{return } x \quad \text{return from call} \]

\[ \text{Function} \quad F ::= \quad f(x) = C \quad \text{function definition} \]

\[ \text{Program} \quad P ::= \quad F^+ C \]
Inter-procedural Pointer Analysis

\[ f(v) = \begin{cases} 
  u &= v; \\
  \text{return } u; 
\end{cases} \]

\[ x = &h; \]

\[ y = f(x) \]

Parameter passing and return can be treated as assignments.
Inter-procedural Pointer Analysis

\[
f(v) = \{
    u = v;
    \text{return } u;
\};
\]

\[
x = \&h;
\]

\[
y = f(x)
\]

Input Relations:
- new(x:X, y:X)
- assign(x:X, y:X)
- load(x:X, y:X)
- store(x:X, y:X)
- arg(f:F, v:X)
- ret(f:F, u:X)
- call(y:X, f:F, x:V)

Output Relations:
- points(x:X, y:X)
Inter-procedural Pointer Analysis

Rules:

- points(x, y) :- new(x, y).
- points(w, z) :- store(x, y), points(x, w), points(y, z).
- points(x, w) :- load(x, y), points(y, z), points(z, w).
- points(w, z) :- store(x, y), points(x, w), points(y, z).
- points(v, h) :- call(_, f, x), arg(f, v), points(x, h).
- points(y, h) :- call(y, f, _), ret(f, u), points(u, h).

Wild card, “don’t care”
Let's build the points to graph that would correspond to the pointer analysis as we've defined it so far.
Context Sensitivity

```c
x = & h1;
z = & h2;
y = f(x);
w = f(z);
f(v) {
    u = v;
    return u;
}
```

Imprecision!
Context Sensitivity

One way to add context sensitivity to the analysis is through what is called "cloning". It achieves context sensitivity by reproducing the bodies of the procedure in line with distinguished variable names.

For example, in this program, instead of replacing $y = f(x)$; $u = v; y = u; w = f(z)$; by $v = z; u = v; w = u;$ we could introduce different copies of the variables $v$ and $u$ (say, $v_i$ and $u_i$ versus $v_j$ and $u_j$) for each call to $f$. In this way, we avoid imprecisely claiming that $w$ may point to $h_1$ or that $y$ may point to $h_2$. Instead, we would have an equivalent program for which pointer analysis would generate a precise points-to graph.

We can achieve greater precision by allowing cloning to be used for more levels in the call stack. However, the tradeoff for precision via cloning is scalability. The deeper we allow function calls to be cloned, the more space and time we need to allow for the resulting analysis. If each function calls just two other functions, the resources needed for a precise analysis becomes exponential in the depth of the stack of nested function calls.

Achieves context sensitivity by **inlining** procedure calls
Varying the Context-Sensitivity

- Context-sensitivity can be achieved by *inline* function calls.
- However, we cannot inline recursive function calls.
- *Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams*, PLDI’04
Limitation

Not powerful enough for arbitrary language

- sound rules?
  - error prone for complicated features of modern languages
  - e.g. function call/return, function as a data, dynamic method dispatch, exception, pointer manipulation, dynamic memory allocation, ...

- accuracy problem
  - consider program a set of statements, with no order between them
  - rules do not consider the control flow
  - the analysis blindly collects every possible facts when rules hold
  - accuracy improvement by more elaborate rules, but no systematic way for soundness proof