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.,

```
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

$$
\begin{array}{lcll}
\mathtt{P} & ::= & \mathtt{C} & \text{program} \\
\mathtt{C} & ::= & & \text{statement} \\
 & | & L \ := \ R & \text{assignment} \\
 & | & \mathtt{C} \ \mathtt{;} \ \mathtt{C} & \text{sequence} \\
L & ::= & x \mid *x & \text{target to assign to} \\
R & ::= & n \mid x \mid *x \mid \&x & \text{value to assign}
\end{array}
$$

- 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.

If program text has component C

$$\frac{C \quad i_1 \cdots i_k}{j}$$

and the current solution set has i_1 … i_k,

Then, add j to the solution set

- 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:

$$\frac{x\ :=\ \&y}{x \to y}$$

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

$$\frac{x\ :=\ y \quad y \to z}{x \to z} \qquad \frac{x\ :=\ *y \quad y \to z \quad z \to w}{x \to w}$$

$$\frac{*x\ :=\ y \quad x \to w \quad y \to z}{w \to z} \qquad \frac{*x\ :=\ *y \quad x \to w \quad y \to z \quad z \to v}{w \to v}$$

$$\frac{*x\ :=\ \&y \quad x \to w}{w \to y}$$

Rules for Pointer Analysis

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

$$\frac{x \;:=\; \&y}{x \to y}$$

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

$$\frac{x \;:=\; y \quad y \to z}{x \to z} \qquad \frac{x \;:=\; *y \quad y \to z \quad z \to w}{x \to w}$$

$$\frac{*x \;:=\; y \quad x \to w \quad y \to z}{w \to z} \qquad \boxed{\frac{*x \;:=\; *y \quad x \to w \quad y \to z \quad z \to v}{w \to v}}$$

$$\boxed{\frac{*x \;:=\; \&y \quad x \to w}{w \to y}}$$

*x := &y — Syntactic sugar:
Can be transformed to
t := &y; *x := t for a new temp var t

*x := *y — Syntactic sugar:
Can be transformed to
t := *y; *x := t for a new temp var t

Example

Example (Pointer analysis steps)

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

- Initial facts are from the first two assignments:

$$\mathtt{x} \rightarrow \mathtt{a}, \quad \mathtt{y} \rightarrow \mathtt{x}$$

- From $\mathtt{y} \rightarrow \mathtt{x}$ and the while-loop body, add

$$\mathtt{x} \rightarrow \mathtt{b}$$

- From the last assignment:
  - from $\mathtt{x} \rightarrow \mathtt{a}$ and $\mathtt{y} \rightarrow \mathtt{x}$, add $\mathtt{a} \rightarrow \mathtt{a}$
  - from $\mathtt{x} \rightarrow \mathtt{b}$ and $\mathtt{y} \rightarrow \mathtt{x}$, add $\mathtt{b} \rightarrow \mathtt{b}$
  - from $\mathtt{x} \rightarrow \mathtt{a}$, $\mathtt{y} \rightarrow \mathtt{x}$, and $\mathtt{x} \rightarrow \mathtt{b}$, add $\mathtt{a} \rightarrow \mathtt{b}$
  - froom $\mathtt{x} \rightarrow \mathtt{b}$, $\mathtt{y} \rightarrow \mathtt{x}$, and $\mathtt{x} \rightarrow \mathtt{a}$, add $\mathtt{b} \rightarrow \mathtt{a}$

General Algorithm

- let R be the set of the chain-reaction rules
- let X_0 be the initial fact set
- let *Facts* be the set of all possible facts

Then, the analysis result is

$$\bigcup_{i \geq 0} Y_i,$$

where

$$\begin{array}{rcl} Y_0 & = & X_0, \\ Y_{i+1} & = & Y \text{ such that } Y_i \vdash_R Y. \end{array}$$

Or, equivalently, the analysis result is the least fixpoint

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

of monotonic function $\phi : \wp(\textit{Facts}) \to \wp(\textit{Facts})$:

$$\phi(X) = X_0 \cup (Y \text{ such that } X \vdash_R Y).$$

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:

$$P ::= \bar{c}$$

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

$$c ::= l \text{ :- } \bar{l}$$

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

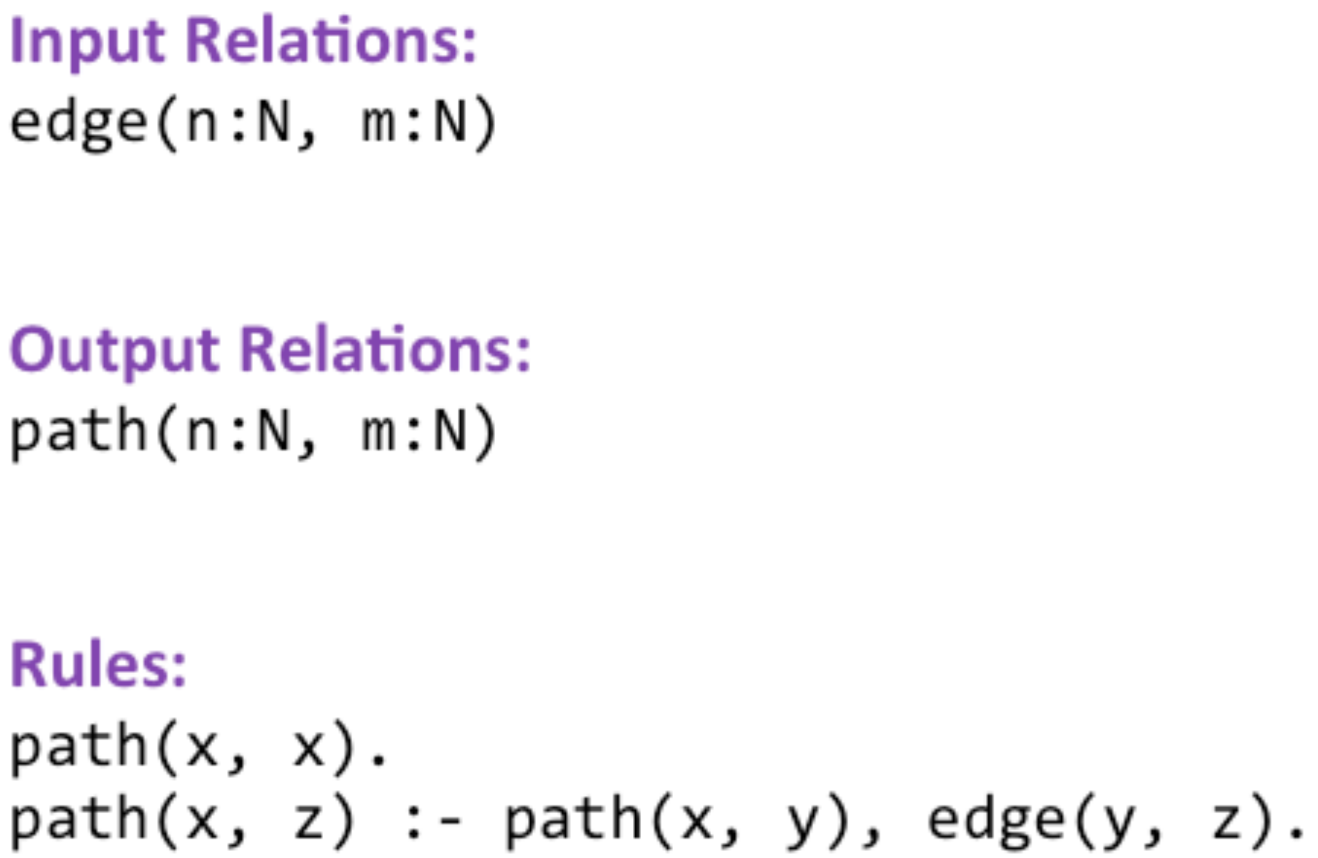
$$l \vee \neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_n \iff l \leftarrow l_1 \wedge l_2 \wedge \cdots \wedge l_n$$

- A literal is a relation with arguments:

$$l ::= r(\bar{a})$$

  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

Input Relations:

```
edge(n:N, m:N)
```

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

Output Relations:

```
path(n:N, m:N)
```

Rules:

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

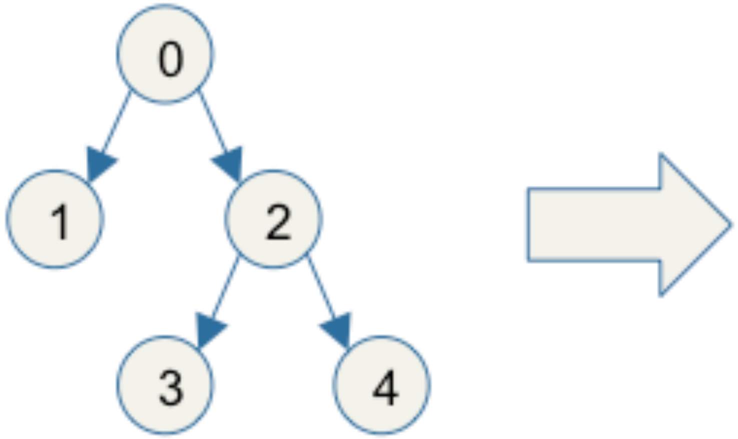
Syntax of Datalog: Example

Input Relations:

```
edge(n:N, m:N)
```

Output Relations:

```
path(n:N, m:N)
```



edge	
n	m
0	1
0	2
2	3
2	4

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:

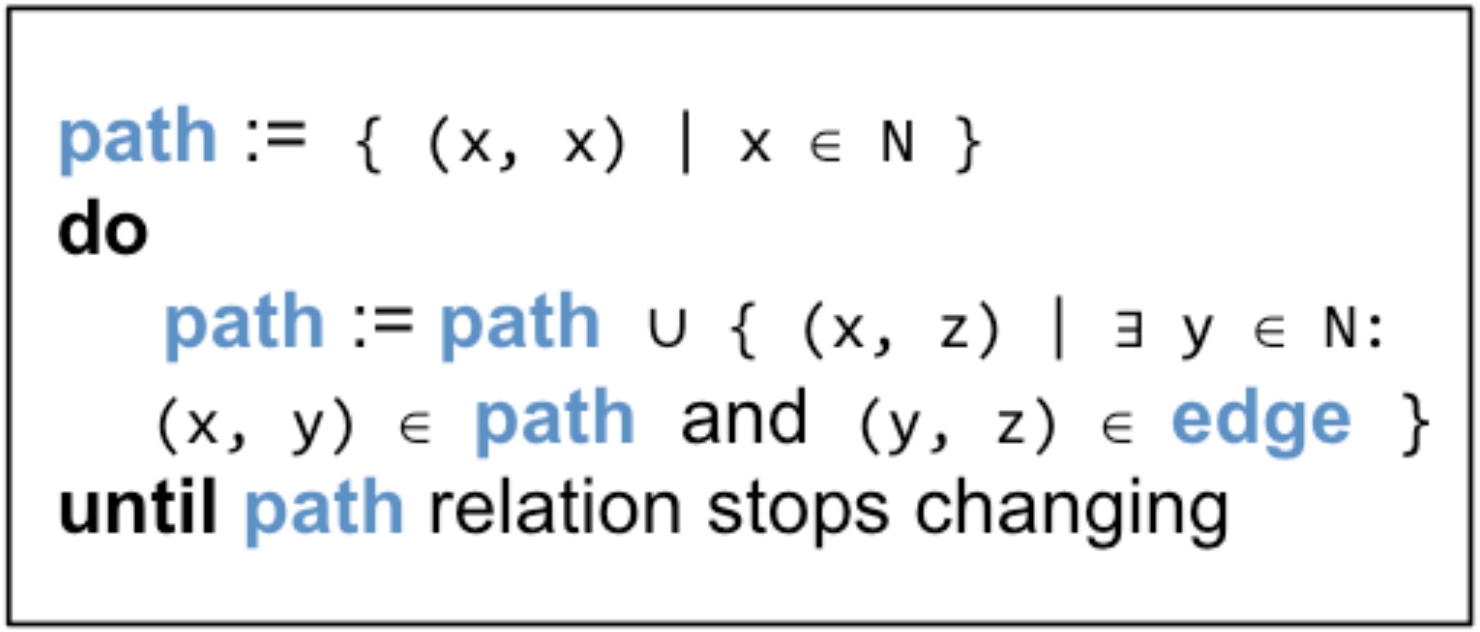
```
edge(n:N, m:N)
```

Output Relations:

```
path(n:N, m:N)
```

Rules:

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

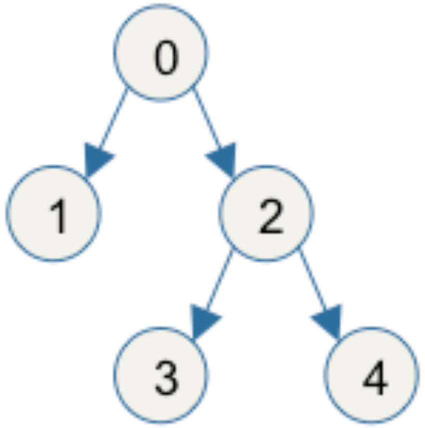


```
path := { (x, x) | x ∈ N }
do
    path := path ∪ { (x, z) | ∃ y ∈ N:
    (x, y) ∈ path and (y, z) ∈ edge }
until path relation stops changing
```

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)
```

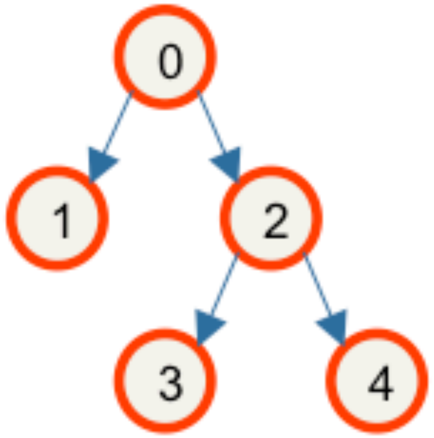
Output Tuples:

```
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)
```

Syntax of Datalog: Example

Input Relations:

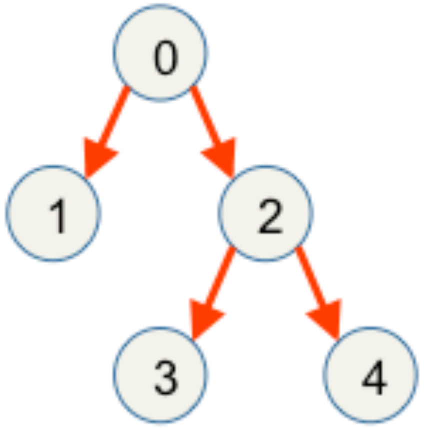
```
edge(n:N, m:N)
```

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)
```

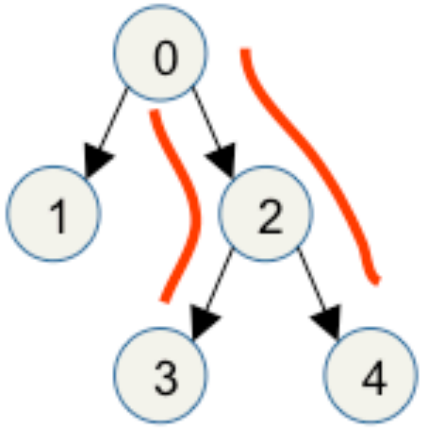
Output Tuples:

```
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:

  $$\llbracket P \rrbracket \in \wp(G)$$

  where G is the set of ground literals (literals without variables).
- A Datalog rule l :- $l_1, \ldots, l_n$ denotes the function:

  $$f_{l \text{ :- } l_1,\ldots,l_n}(X) = \{\sigma(l_0) \mid \sigma(l_k) \in X \text{ for } 1 \leq k \leq n\}$$

  where σ is a variable substitution.
- The semantics of P is defined as the least fixed point of F_P:

  $$\llbracket P \rrbracket = lfp F_P \;\text{ where } F_P(X) = X \cup \bigcup_{c \in P} f_c(X)$$

- The semantics is monotone:

  $$P_1 \subseteq P_2 \implies \llbracket P_1 \rrbracket \subseteq \llbracket P_2 \rrbracket$$

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 → y — points(x:X, y:X)`

Datalog Rules

- Datalog rule for $\dfrac{x \; := \; \&y}{x \rightarrow y}$
  - `points(x, y) :- new(x, y).`
- Datalog rule for $\dfrac{x := y \quad y \rightarrow z}{x \rightarrow z}$
  - `points(x, z) :- assign(x, y), points(y, z).`

Datalog Rules

- Datalog rule for $\dfrac{x := *y \quad y \to z \quad z \to w}{x \to w}$
  - `points(x, w) :- load(x, y), points(y, z),`
    `points(z, w).`
- Datalog rule for $\dfrac{*x := y \quad x \to w \quad y \to z}{w \to z}$
  - `points(w, z) :- store(x, y), points(x, w),`
    `points(y, z).`

Extended Language for Functions

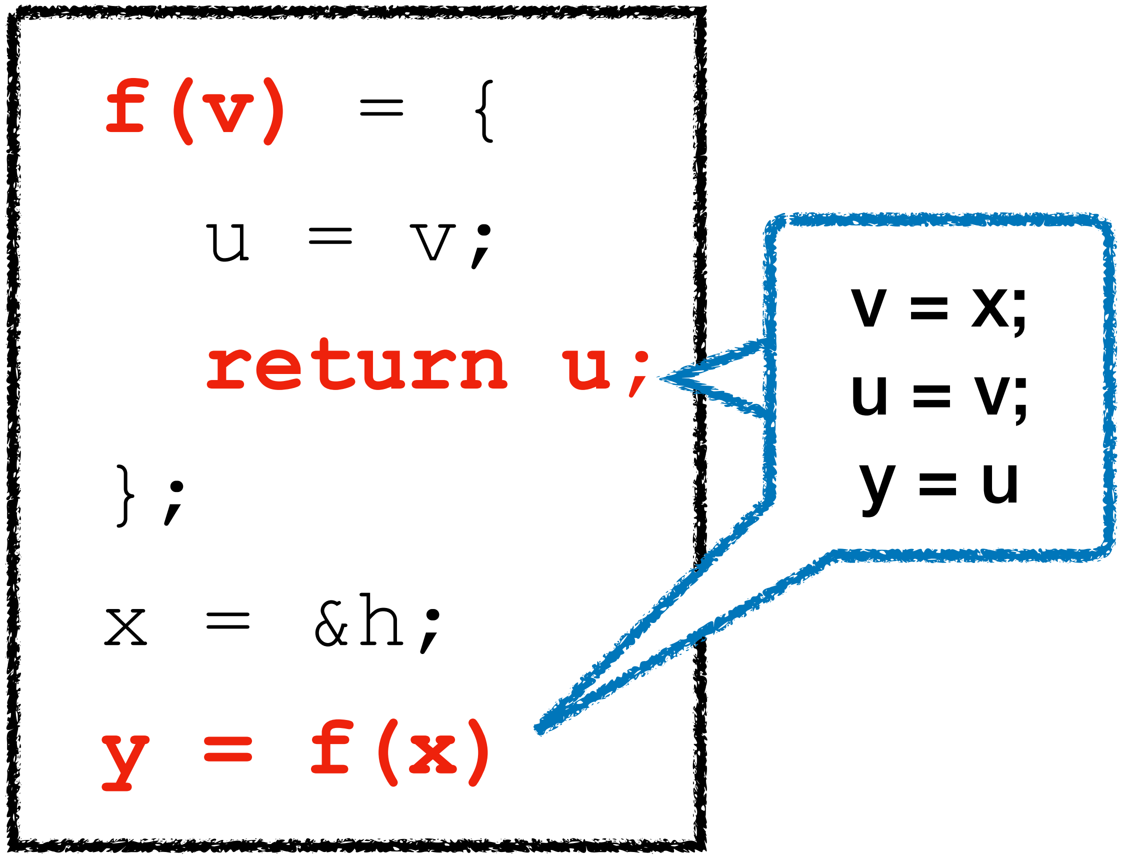
$$\begin{array}{llcll}
\text{Statement} & \mathtt{C} & ::= & \cdots & \\
 & & \mid & \mathtt{y} := \mathtt{f}(\mathtt{x}) & \text{function call} \\
 & & \mid & \mathbf{\mathtt{return}}\ \mathtt{x} & \text{return from call} \\
\text{Function} & \mathtt{F} & ::= & \mathtt{f}(\mathtt{x}) = \mathtt{C} & \text{function definition} \\
\text{Program} & \mathtt{P} & ::= & \mathtt{F}^{+}\mathtt{C} &
\end{array}$$

Inter-procedural Pointer Analysis

```
f(v) = {
  u = v;
  return u;
};
x = &h;
y = f(x)
```

Parameter passing and return can be treated as assignments.

Inter-procedural Pointer Analysis



```
f(v) = {
   u = v;
   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

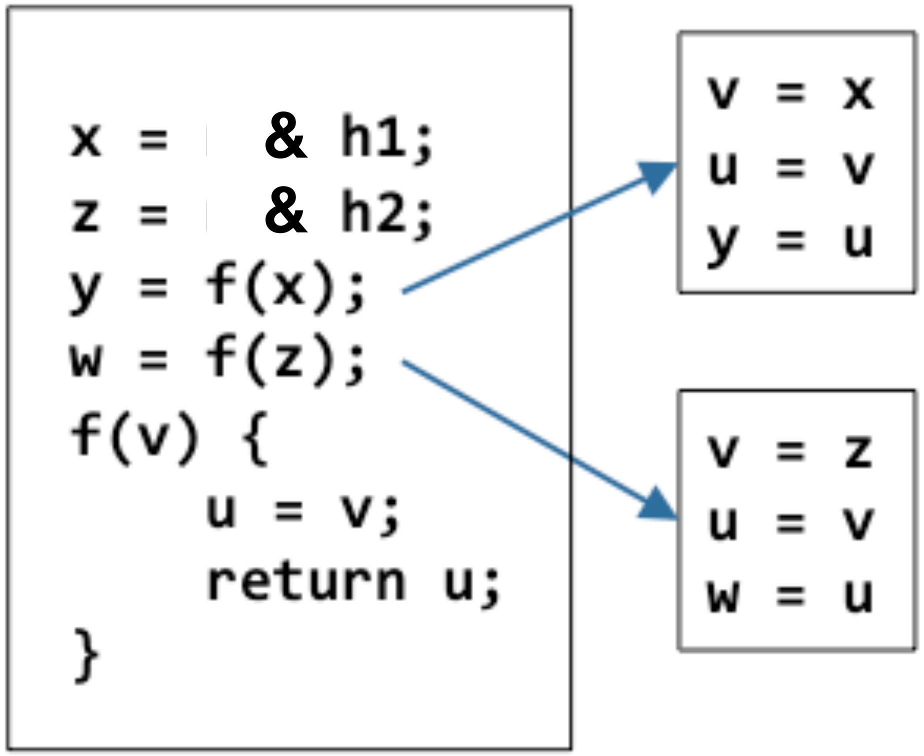
```
f(v) = {
  u = v;
  return u;
};
x = &h;
y = f(x)
```

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"

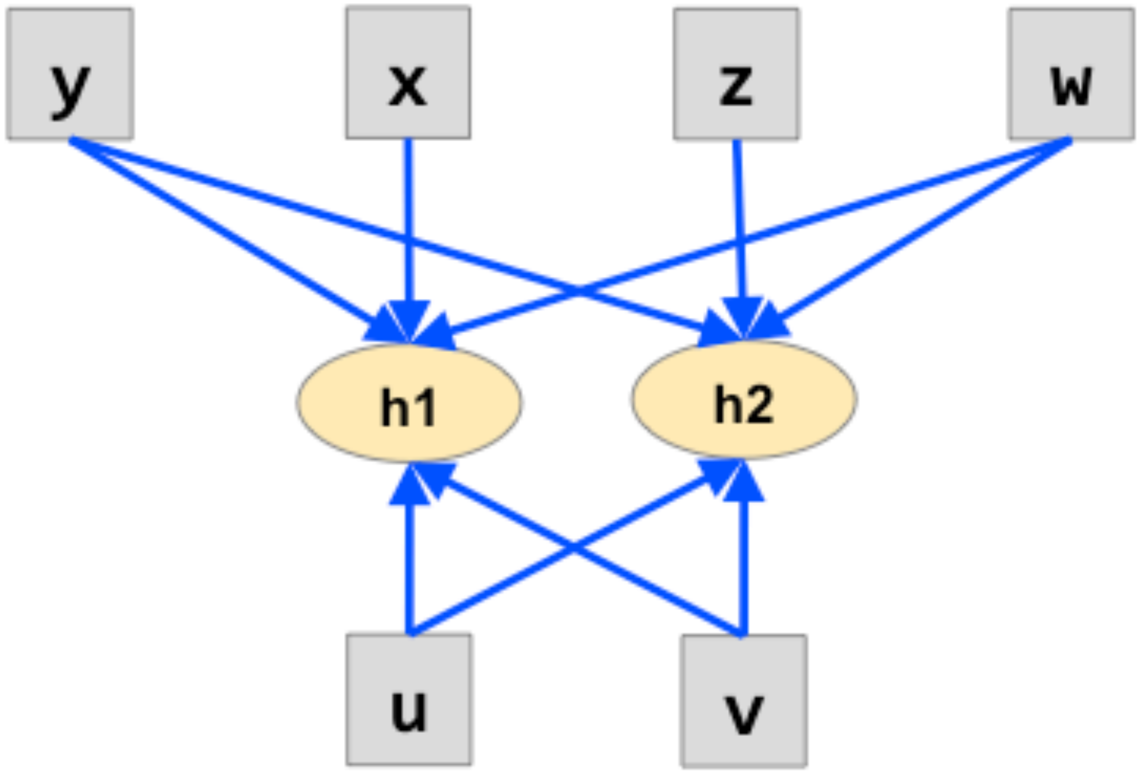
Context Sensitivity



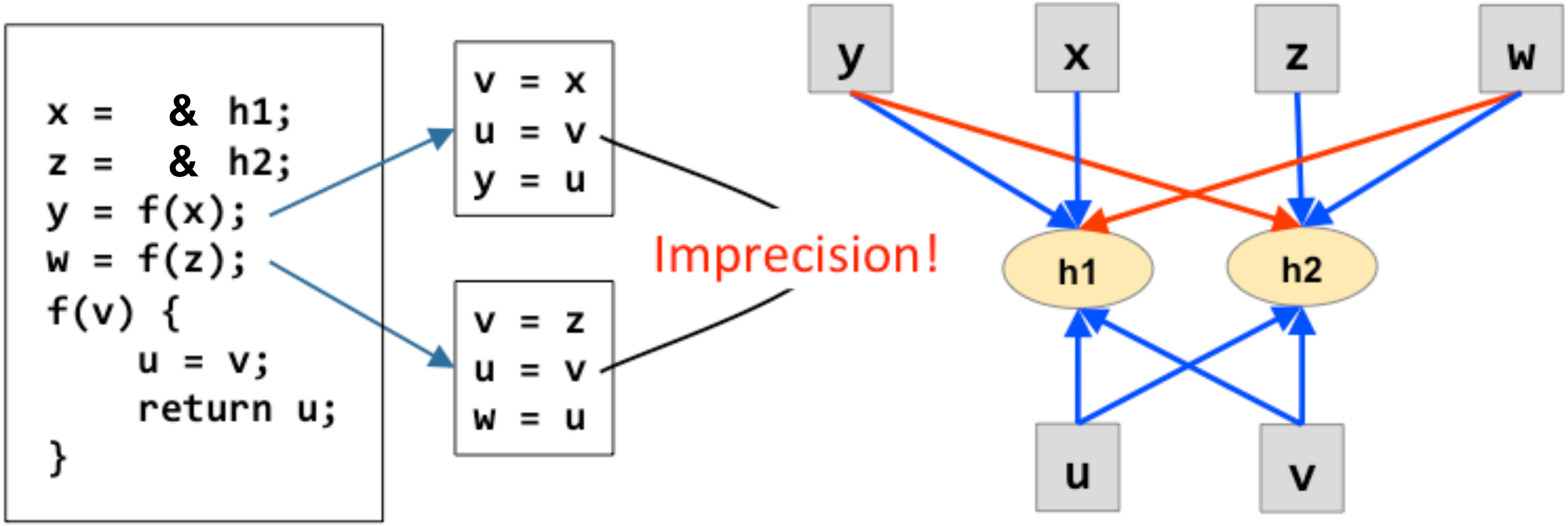
```
x =   & h1;
z =   & h2;
y = f(x);
w = f(z);
f(v) {
    u = v;
    return u;
}
```

```
v = x
u = v
y = u
```

```
v = z
u = v
w = u
```



Context Sensitivity



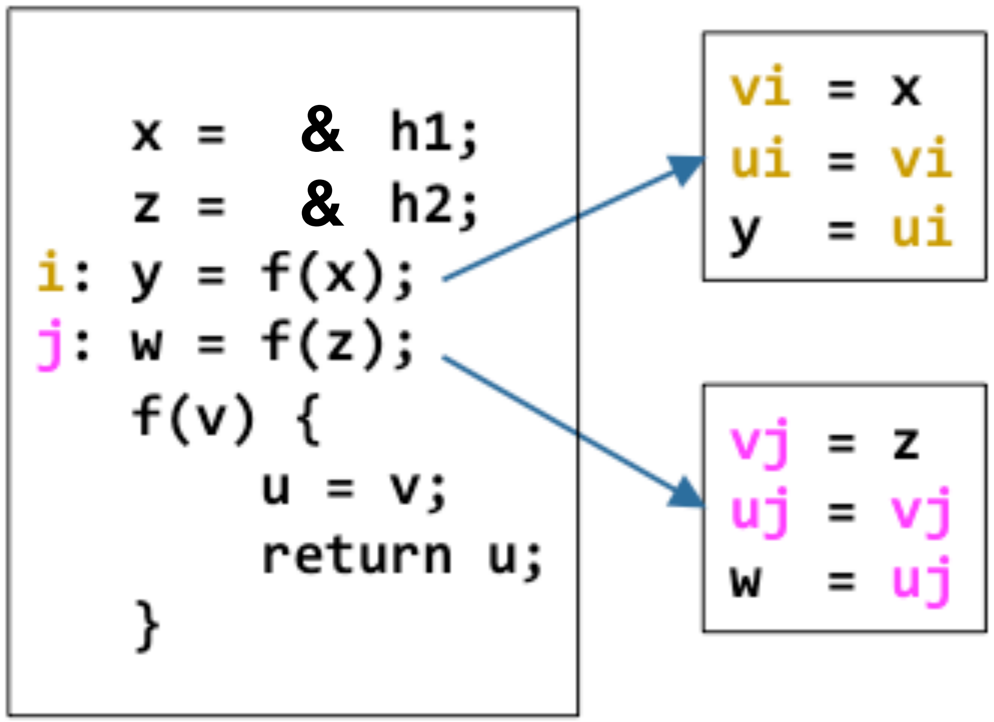
```
x =  & h1;
z =  & h2;
y = f(x);
w = f(z);
f(v) {
    u = v;
    return u;
}
```

```
v = x
u = v
y = u
```

```
v = z
u = v
w = u
```

Imprecision!

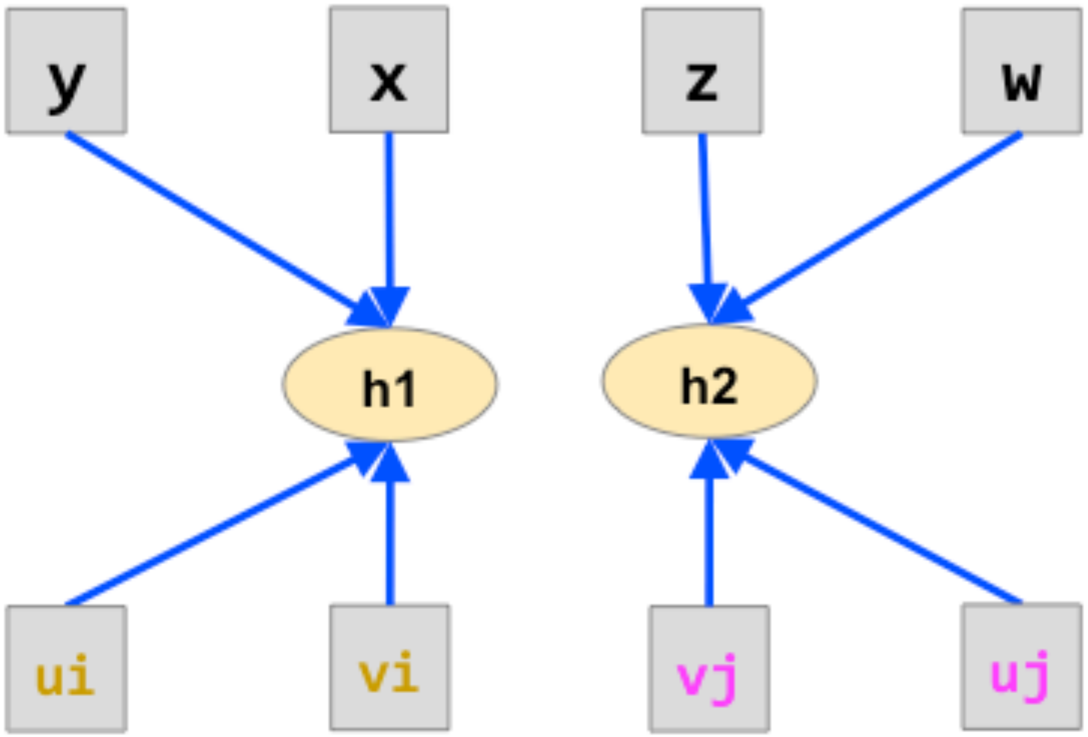
Context Sensitivity



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

```
vi = x
ui = vi
y  = ui
```

```
vj = z
uj = vj
w  = uj
```



Achieves context sensitivity by **inlining** procedure calls

Varying the Context-Sensitivity

- Context-sensitivity can be achieved by *inlining* 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