Constraint-based Analysis

Introduction

1. Constraint-based analysis
    - Follows a declarative program
        - Express what the analysis computes instead of how it computes it
        - Concerned with the specification of the analysis instead of the implementation
    - Specifiation involves constraints over program facts
    - Implementation involves solving these constraints using an off-the-shelf constraint solver
2. Advantages
    - Simplifies design and understanding of analysis
    - Allows rapid prototyping
    - Continuous performance improvement in constraints solvers
    - Datalog: Constraint programming language

Constraint-based Analysis Motivation

1. Motivation
    - Designing an efficient program analysis is challenging:
        - Program analysis = Specification + Implementation
        - Specification is what, implementation is how
    - Null pointer analysis
        - Specification: No null pointer is dereferenced along any path in the program
        - Implementation: Many design choices
            * Forward vs backward traversal
            * Symbolic vs explicit representation

What is Constraint-based Analysis?

1. Constraint-based analysis: Analysis designer defines the specification of the program analysis using a constraint language
    - Constraint solver automates the implementation of the analysis

Benefits of Constraint-based Analysis

1. Benefits
    - Separates analysis specification from implementation
        - Analysis writer can focus on "what" rather than "how"
    - Yields natural program specifications
        - Constraints are usually local, whose conjuntions capture global properties
    - Enables sophisticated analysis implementations
        - Leverage powerful, off-the-shelf solvers

Specification and Implementation Quiz

1. Consider a dataflow analysis such as live variables analysis. If one expresses it as a constraint-based analysis, one must still decide:
    - The order in which statements should be processed (false)
    - What the gen and kill sets for each kind of statement are (true)
    - In what language to implement the chaotic iteration algorithm (false)
    - Whether to take intersection or union at merge points (true)

Outline of the Lesson

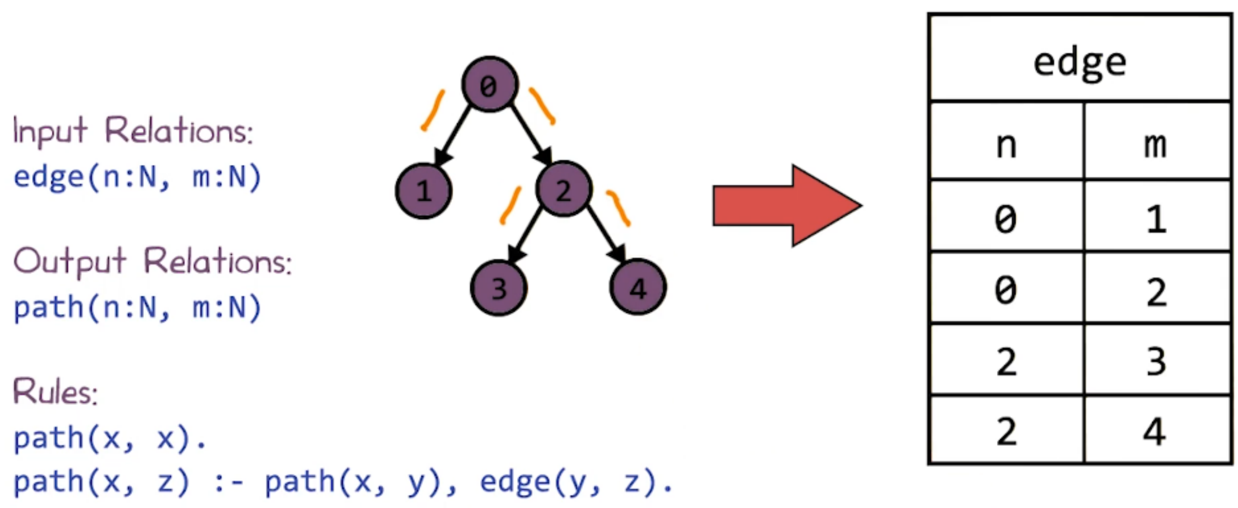
1. A constraint language: Datalog
   - Two static analyses in Datalog:
     - Intra-procedural analysis: computing reaching definitions
     - Inter-procedural analysis: computing points-to information

A Constraint Language: Datalog

1. 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 deductive databases
   - Later applied in many other domains: software analysis, data mining, networking, security, knowledge representation, cloud-computing, ...
   - Many implementations: Logicblox, bddbddb, IRIS, Paddle, ...

Syntax of Datalog: Example

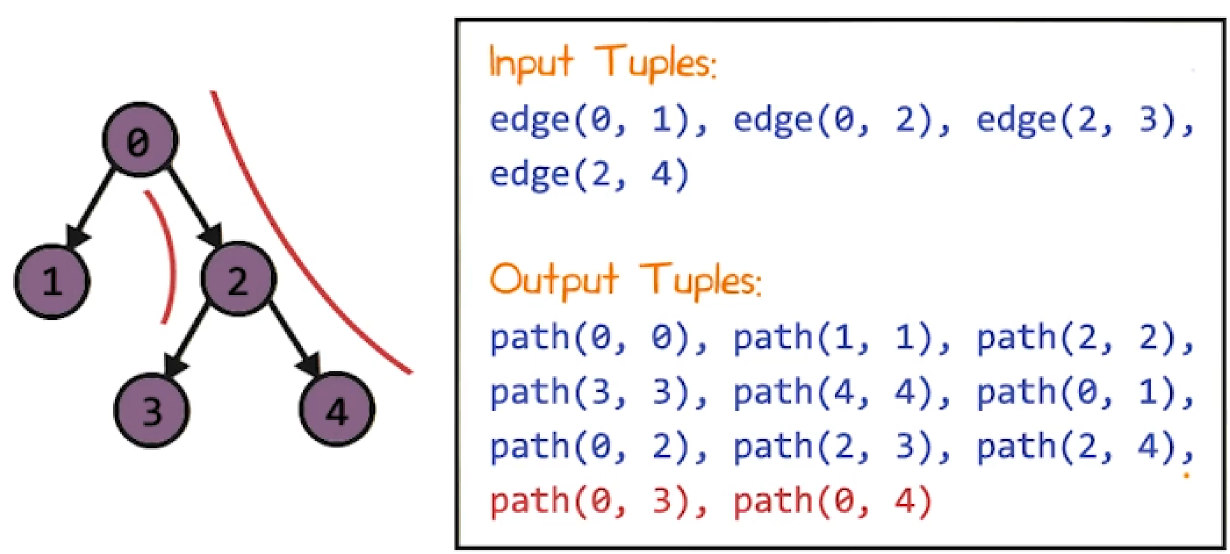
1. Example: Graph Reachability
   - Find all pairs of nodes that are connected in a graph
   - Need to define form of input, form of output, and rules of inference
   - A relation is similar to a table in a database
   - A tuple in a relation is similar to a row in a table
   - Rules of inference: Deductive rules that hold universally (i.e., variables like x, y, z can be replaced by any constant)
     - Specify "if ... then ..." logic
2. Input Relations:
   - edge(n:N, m:N)
   - n and m are nodes in the set of all nodes N
3. Output Relations:
   - path(n:N, m:N)
4. Rules:
   - path(x, x).
     - (if TRUE,) there is a path from each node to itself
   - path(x, z) :- path(x, y), edge(y, z).
     - If there is a path from node x to y, and there is an edge from y to z, then there is a path from x to z.
     - Hypothesis on the left, conclusion on the right
     - Rules separated by commas are logically ANDed together



Datalog Syntax

Semantics of Datalog: Example

1. Example: Graph Reachability
   - Start with empty path relation
   - Apply each rule, growing the path relation with each application
   - Stop when the path relation stops growing
   - If there are multiple rules, the order in which the rules are applied does not matter
   - The result is the “least” solution; the smallest solution that satisfies all the rules



Datalog Semantics

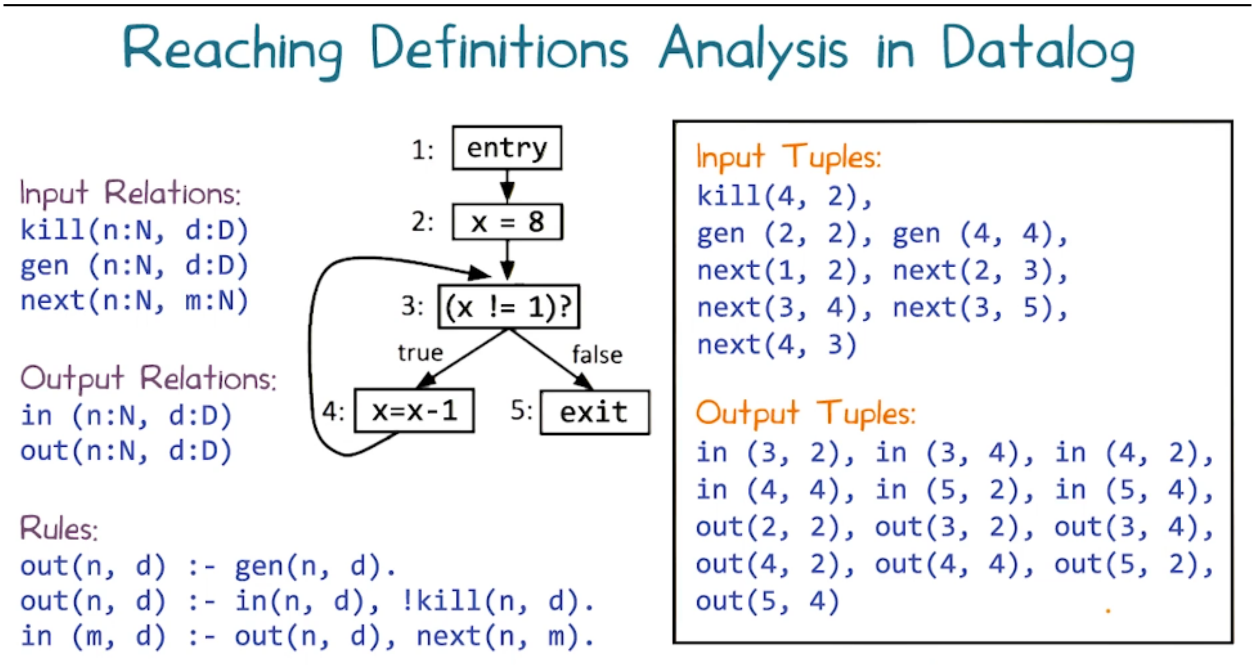
Computation Using Datalog

1. Check each of the below Datalog rules which correctly computes the relation scc: scc(n1, n2) if and only if n2 is reachable from n1 AND n1 is reachable from n2
   - scc(n1, n2) :- edge(n1, n2), edge(n2, n1). (false)
   - scc(n1, n2) :- path(n1, n2), path(n2, n1). (true)
   - scc(n1, n2) :- path(n1, n3), path(n3, n2), path(n2, n4), path(n4, n1). (true)
   - scc(n1, n2) :- edge(n1, n3), edge(n2, n3). (false)
   - First rule is not incorrect, but won't return non-adjacent nodes

Reaching Definitions Analysis

1. Specification of reaching definitions analysis:
   - OUT[n] = (IN[n] - KILL[n]) U GEN[n]
     - Input relations:
       * kill(n:N, d:D)
       * gen(n:N, d:D)
       * next(n:N, m:N)
   - IN[n] = U OUT[n'] where n' is predecessors(n)
     - Output relations:
       * in(n:N, d:D)
       * out(n:N, d:D)
   - Rules:
     - out(n, d) :- gen(n, d).
     - out(n, d) :- in(n, d), !kill(n, d).
     - in(m, d) :- out(n, d), next(n, m).
   - n are statements
   - d are definitions

Reaching Definitions Analysis 2



Reaching Definitions in Datalog

Live Variables Analysis

1. Complete the rules for live variables analysis:
    - Input relations:
        - kill(n:N, v:V)
        - gen(n:N, v:V)
        - next(n:N, m:N)
    - Output relations:
        - in(n:N, v:V)
        - out(n:N, v:V)
    - Rules:
        - in(n, v) :- gen(n, v).
        - in(n, v) :- out(n, v), !kill(n, v).
        - out(n, v) :- in(m, v), next(n, m).

Pointer Analysis in Datalog

1. Consider a flow-insensitive may-alias analysis for a simple language:
    - (function body)
        - f(v) {s1, ..., sn }
    - (statement)
        - s ::= v = new h | v = u | return u | v = f(u)
    - (pointer variable)
        - u, v
    - (allocation site label)
        - h
    - (function name)
        - f
    - Performs weak updates, not strong updates

Intra-procedural Pointer Analysis

1. Input Relations:
    - new(v:V, h:H)
    - assign(h:H, u:V)
2. Output Relations:
    - points(v:V, h:H)
3. Rules:
    - points(v, h) :- new(v, h).
    - points(v, h) :- assign(v, u), points(u, h).

Querying Pointer Analysis

1. Assume we allow function calls now
    - Parameter passing and return statements can be treated as copy assignments

```
x = new h1;
y = f(x); // v = x; u = v; y = u;

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

2. Need to add input relationsand rules for function calls and return statements
    - Input Relations:

- new(v:V, h:H)
- assign(h:H, u:V)
- arg(f:F, v:V) where F is set of all function calls
- ret(f:F, u:V)
- call(y:V, f:F, x:V)

- Output Relations:
  - points(v:V, h:H)
- Rules:
  - points(v, h) :- new(v, h).
  - points(v, h) :- assign(v, u), points(u, h).
  - points(v, h) :- call(_, f, x), arg(f, v), points(x, h).
    * Underscore is a wildcard
  - points(y, h) :- call(y, f, _), ret(f, u), points(u, h).

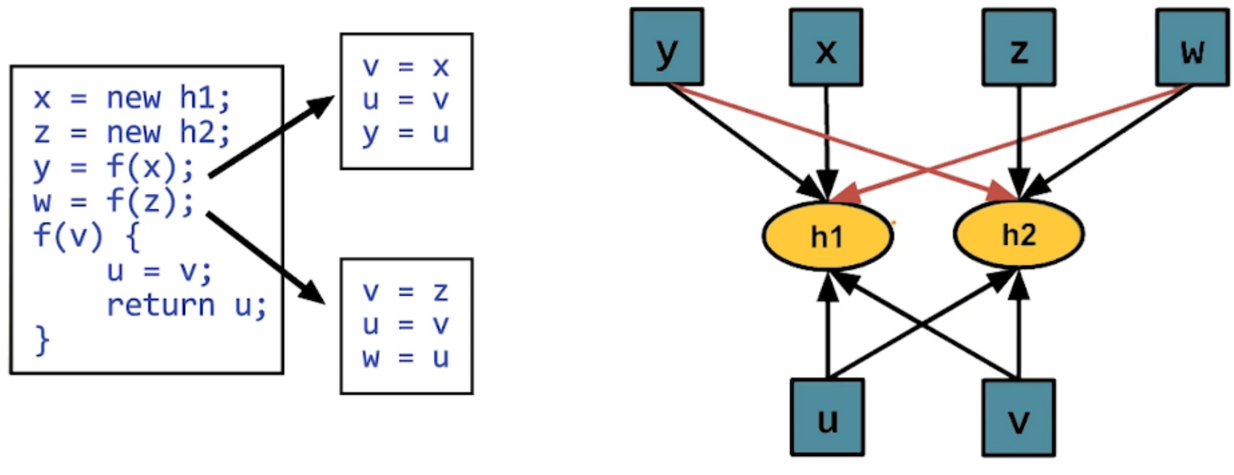
Context Sensitivity

1. Check each of the below Datalog programs which correctly computes the relation mustNotAlias: mustNotAlias(u,v) if and only if u and v do no alias in any run of the program.
   - Rule 1 (false; need to check every allocation site)
     - mustNotAlias(u, v) :- points(u, h1), points(v, h2), h1 != h2.
   - Rule 2 (true)
     - mayAlias(u, v) :- points(u, h), points(v, h).
     - mustNotAlias(u, v) :- !mayAlias(u, v).
   - Rule 3 (false; wildcard means they may be pointing to different sites)
     - mayAlias(u, v) :- points(u, _), points(v, _)
     - mustNotAlias(u, v) :- mayAlias(u, v).
   - Rule 4 (true)
     - common(u, v, h) :- points(u, h), points(v, h).
     - mayAlias(u, v) :- common(u, v, _).
     - mustNotAlias(u, v) :- !mayAlias(u, v).

Context Sensitivity

1. Current analysis targets pointer aliasing across function calls
   - Results in a loss of precision

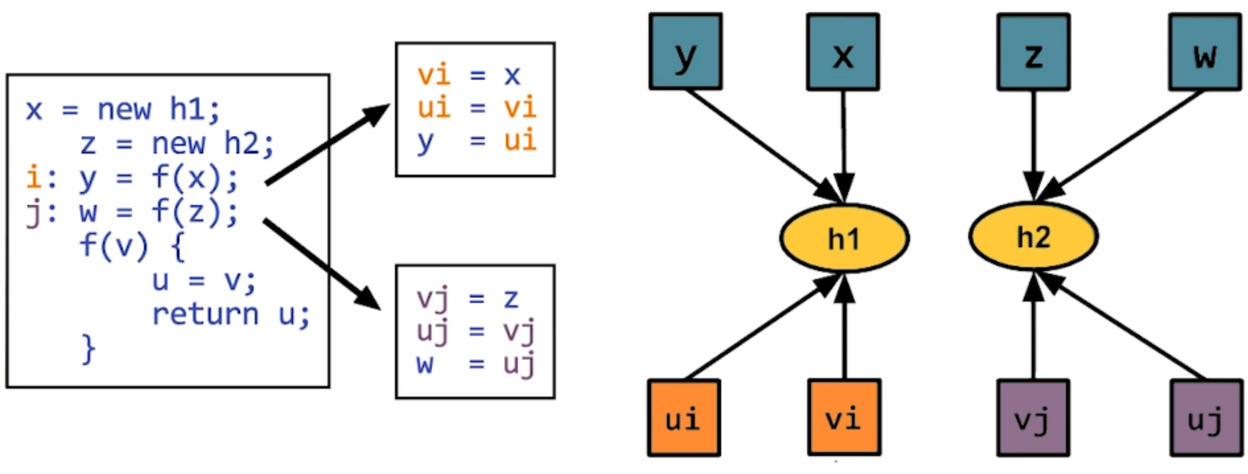
```
x = new h1;
y = new h2;
y = f(x);
w = f(z);
f(v) {
    u = v;
    return u;
}
```



Context Insensitivity

Cloning-based Inter-procedural Analysis

1. Cloning achieves context sensitivity by reproducing the bodies of the procedure inline with distinguished variable names
   - Cloning depth improves precision at the cost of scalability
     - Cost becomes exponential with depth of call stack



Context Sensitivity

What About Recursion?

1. Need infinite cloning depth to differentiate the points-to sets of x,y and w,z

```
x = new h1;
y = new h2;
y = f(x);
w = f(z);

f(v) {
```

```
    if (*)
        v = f(v);
    return v;
}
```

Summary-based Inter-procedural Analysis

1. Summary-based Approach
    - Use the incoming program states to differentiate calls to the same procedure
        - Same incoming program states yield same outgoing program states for a give procedure
    - As precise as cloning-based analysis with infinite cloning depth

Other Constraint Languages

Constraint Language	Problem Expressed	Example Solvers
Datalog	Least solution of deductive inference rules	LogixBlox, bddbddb
SAT	Boolean satisfiability problem	MiniSat, Glucose
MaxSAT	SAT extended with optimization	open-wbo, SAT4j
SMT	Satisfiability modulo theories problem	Z3, Yices
MaxSMT	SMT extended with optimization	Z3

Conclusion

1. Constraint-based analysis and its benefits
2. The Datalog constraint language
3. How to express static analyses in Datalog
    - Analysis logic == constraints in Datalog
    - Analysis inputs and outputs == relations of tuples
4. Context-insensitive and context-sensitive inter-procedural analysis