CFL Reachability & Datalog

Candidate

Introduction (Approx. 2 Minutes)

Good morning. Today I will be presenting two fundamental frameworks used in static program analysis: **Context-Free Language (CFL) Reachability** and **Datalog**.

To give you some context, static analysis is all about reasoning about program behavior without actually running the code. We often model programs as graphs—where nodes are program points and edges represent control flow or data flow.

My agenda for this presentation is three-fold:

- First, I will explain **CFL Reachability**, discussing why standard reachability isn't enough and how formal grammars filter invalid paths.
- Second, I will introduce **Datalog**, a declarative logic language that separates the specification of analysis from its implementation.
- Finally, I will discuss the complexity and expressiveness trade-offs.

Whiteboard Action

- **Title:** CFL Reachability & Datalog
- **Agenda:**
  1. CFL Reachability (Validity & Grammars)
  2. Datalog (Declarative Logic)
  3. Complexity

Part 1: CFL Reachability (Detailed Breakdown)

The Core Problem: Standard vs. Valid Reachability

In a simple directed graph, we ask: *Is there a path from node s to node t?*. Standard algorithms like BFS or DFS solve this in linear time $O(n + m)$.

However, in program analysis, a "path" in the graph might not represent a valid execution. For example, if function `main` calls `foo`, and `foo` calls `bar`, when `bar` returns, it **must** return to `foo`. It cannot magically return to a different function.

Standard reachability treats all edges equally and misses this constraint. To fix this, we label the edges with symbols from an alphabet Σ and define valid paths using a Context-Free Language $\mathcal{L}$.

Whiteboard Action

Draw Graph Example:

- Nodes: 1, 2, 3, 4
- Edges representing calls/returns:

$$1 \xrightarrow{(_1} 2 \xrightarrow{(_2} 3 \xrightarrow{)_2} 4 \xrightarrow{)_1} 1$$

Dyck Reachability

The most important application is **Dyck Reachability** (matched-parenthesis reachability). We define a grammar G that generates balanced parentheses. A function call is an "open parenthesis" $(_i$ and a return is a "close parenthesis" $)_i$.

Whiteboard Action

Write the Grammar:

$$\mathcal{S} \to (_i\mathcal{S})_i \mid \mathcal{S}\mathcal{S} \mid \epsilon$$

Let's trace the path on the graph I drew.

1. From 1 to 2, we "open" scope 1.
2. From 2 to 3, we "open" scope 2.
3. Traversing edge $)_2$ "closes" scope 2. This matches the inner parenthesis.
4. Finally, $)_1$ closes the outer scope.

The path string is $(_1(_2)_2)_1$, which reduces to ϵ. Therefore, node 1 reaches itself via a valid Dyck path.

Field Sensitivity & Complexity

We can apply this same logic to **Field Sensitivity** in pointer analysis. Writing to a field 'x.f = y' is an "open" parenthesis; reading 'z = x.f' is a "close" parenthesis.

Complexity: While standard reachability is linear, CFL reachability generally takes $O(n^3)$ time. This is essentially a dynamic programming approach, similar to the CYK parsing algorithm.

Undecidability: If we try to model **both** function calls (context sensitivity) AND heap accesses (field sensitivity) perfectly, we reach an undecidable problem. This requires intersecting two Dyck languages, which is known to be undecidable.

Part 2: Datalog (Approx. 6-7 Minutes)

Introduction to Datalog

CFL reachability is the *concept*; Datalog is the *tool* or language we use to express it. Datalog is a declarative logic programming language. Unlike imperative languages, we only describe *what* implies what, not *how* to compute it.

A program consists of two things:

1. **Facts:** Represent input data (the graph edges).

2. **Rules:** Allow us to infer new information.

Whiteboard Action

Write Datalog Syntax:

```
1. Facts:   edge(1, 2).
2. Rules:   path(X, Y) :- edge(X, Y).
            path(X, Y) :- path(X, Z), edge(Z, Y).
```

The first rule is the base case: "There is a path from X to Y **if** there is an edge". The second rule is the recursive step: "There is a path from X to Y **if** there is a path to Z and an edge from Z to Y".

Semantics: Least Fixed Point

How does the computer execute this? It uses **Least Fixed Point semantics**.

- It starts with the known facts.
- It applies rules to discover new facts.
- It repeats this process until no new facts can be generated.

Negation and Stratification

If we want to say reachability depends on something **NOT** existing, we use Negation.

Whiteboard Action

Write Negation Rule:

$$oneWay(X,Y) : -path(X,Y), \text{not } path(Y,X).$$

Negation is dangerous because it can lead to paradoxes. To handle this, we use **Stratified Datalog**. We organize predicates into layers (strata). If a rule depends negatively on a predicate, that predicate must be computed in a lower layer first.

Conclusion (<2 Minutes)

To wrap up, we have looked at the two sides of this analysis:

- **CFL Reachability** is the theoretical framework. It allows precision (Context/Field sensitivity) by filtering paths via grammars, but costs $O(n^3)$.
- **Datalog** is the implementation tool. It is declarative, captures all polynomial-time algorithms, and efficient for program analysis where graphs are large but rules are few.

This combination—using formal grammars to define validity and Datalog to compute it—is the backbone of modern static analysis.

Thank you. I am happy to take any questions.