CFL Reachability & Datalog

Candidate

Introduction (Approx. 2 Minutes)

Good morning. Today I will be presenting two frameworks: **Context-Free Language (CFL) Reachability** and **Datalog**. While I will connect them in the context of static analysis, it is important to note that they are distinct tools. CFL Reachability is a formal language framework used to filter graph paths, whereas Datalog is a general-purpose declarative logic language used in databases, networking, and complexity theory.

My agenda is as follows:

- First, I will explain **CFL Reachability**, starting with a general example and moving to the specific "Dyck" formulation used in analysis.
- I will discuss the "Interleaved Dyck" problem and the undecidability issues that arise when combining different sensitivities.
- Second, I will present **Datalog** as a powerful logic solver, focusing on its semantics, negation paradoxes, and how it captures polynomial-time algorithms.

Whiteboard Action

- **Title:** CFL Reachability & Datalog
- **Agenda:**
  1. CFL Reachability (General → Dyck → Interleaved)
  2. Datalog (Semantics, Negation, PTime)

Part 1: CFL Reachability

General CFL Reachability

Standard graph reachability checks if any path exists from node A to B. In program analysis, this is too imprecise because many paths in the graph represent invalid executions. We constrain these paths using a Context-Free Language.

Before we look at code analysis, let's look at a "normal" CFL example: **Palindromes**. Imagine a graph where edges are labeled with letters a and b. We want to find a path that reads the same forwards and backwards (e.g., $a \to b \to b \to a$). We can describe this with a grammar:

$$S \to aSa \mid bSb \mid \epsilon$$

Whiteboard Action

Draw Palindrome Example:

- Edges: $1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 4 \xrightarrow{a} 5$
- Path string: $abba$
- Reduction: $a(b(\epsilon)b)a \to a(b)a \to \epsilon$ (Valid)

If the path string can be reduced to ϵ using the grammar, the nodes are reachable.

Dyck Reachability (Context Sensitivity)

In program analysis, we don't usually care about palindromes. We care about **matched pairs**, like open and close parentheses. This is called **Dyck Reachability**. Function calls are "open" parentheses $(_i$ and returns are "close" parentheses $)_i$. A valid execution path must have balanced parentheses:

$$S \to (_i S)_i \mid SS \mid \epsilon$$

If method A calls B, B must return to A, not C. This filters out "impossible" paths in the control flow graph.

Field Sensitivity via Dyck

We can apply this exact same logic to heap analysis, known as **Field Sensitivity**. Standard pointer analysis might confuse data flowing through different fields of an object. We model a write to a field 'x.f = y' as an open bracket $[_f$ and a read 'z = x.f' as a close bracket $]_f$.

Whiteboard Action

Field Sensitivity Example:

- Flow: $y \xrightarrow{[_f} x \xrightarrow{]_f} z$
- Meaning: y is written to field f of x, then read back into z.
- Matches $[_f \dots]_f$.
- Mismatch: $y \xrightarrow{[_f} x \xrightarrow{]_g} z$ (Writing field f but reading field g is invalid).

The Problem of Interleaved Dyck Reachability

In a perfect world, we would track **both** function calls (Context Sensitivity) and heap accesses (Field Sensitivity) perfectly. This requires intersecting two Dyck languages:

- L_1: Matched calls/returns $(_i \dots)_i$
- L_2: Matched writes/reads $[_f \dots]_f$

We need to find a **single path** P such that the string of labels on P satisfies both grammars simultaneously ($w \in L_1 \cap L_2$). This problem, **Interleaved Dyck Reachability**, is undecidable.

Whiteboard Action

The Approximation Problem: Why can't we just solve them separately?

- If we say "Reachable if Path A satisfies L_1 AND Path B satisfies L_2"...
- We might find that function flow allows a path through 'foo()', but data flow requires a path through 'bar()'.
- Since the execution cannot take two different paths at once, this is an over-approximation.

Part 2: Datalog

Datalog as a General Framework

Now I will move to **Datalog**. While we use it to solve reachability, it is actually a general declarative logic language used for databases, networking, and data integration. In Datalog, we focus on the **what**, not the **how**. It is powerful enough to express **any polynomial-time algorithm** (The complexity class PTime).

A program consists of:

- **Facts:** The input data (Extensional Database).
- **Rules:** Logical inferences (Intensional Database).

Negation and Paradoxes

Simple Datalog is purely positive (monotonic). If we add **Negation** ('not'), we run into logical paradoxes where no minimal model exists. Consider this voting example from the slides:

Whiteboard Action

The Voting Paradox:

```
voteTrump(X)  :- vote(X), not voteHarris(X).
voteHarris(X) :- vote(X), not voteTrump(X).
vote(John).
```

If John votes, the first rule says he votes Trump if he didn't vote Harris. The second says he votes Harris if he didn't vote Trump. Depending on where you start, you get contradictory results. There is no defined "Least Fixed Point" here.

Stratified Datalog

To solve this, we use **Stratification**. We interpret the rules in layers (strata). We construct a Precedence Graph of predicates:

- If rule $A \leftarrow B$ exists, draw edge $B \rightarrow A$.
- If rule $A \leftarrow \text{not } B$ exists, draw a "negative edge" $B \overset{\neg}{\rightarrow} A$.

A program is valid (stratified) only if there are **no cycles containing a negative edge**. This ensures we fully compute the "negated" facts (lower layer) before using them in the upper layer.

Conclusion

To summarize:

1. **CFL Reachability** gives us a way to define valid paths using grammars. We looked at the Palindrome example, generalized it to Dyck for function/field matching, and saw that combining two Dyck languages (Interleaved) leads to undecidability.

2. **Datalog** is the solver engine. It captures all PTime algorithms but requires care with Negation (solved via Stratification) to maintain well-defined semantics.

Using Datalog to implement CFL reachability allows us to write concise, correct static analysis algorithms that are efficient to solve.

Thank you.