CS 4110 – Programming Languages and Logics
Lecture #34: Logic Programming

1 Introduction

Logic programming emerged in the late 1960s and early 1970s as an attempt to use mathematical logic (in particular, first-order logic), as a foundation for automated reasoning and computation. Prolog was an early language, developed by Colmerauer and Kowalski, and showed that a logic-based programming language could express a variety of computations. Datalog is a well-behaved subset of Prolog that avoids some of the complications that arise in the full language and has been used a foundational language in database, program analysis, and more.

A logic program consists of facts, rules, and queries. Facts assert built-in knowledge that is assumed, rules encode logical implications, and queries ask whether given statements follow from the facts and the rules. The key building blocks of languages like Prolog and Datalog are Horn clauses of the form:

$$p_1 \wedge \cdots \wedge p_k \rightarrow h$$

usually written as:

$$h :\!- \, p_1, \ldots, p_k.$$

Intuitively, a Horn clause can be understood as follows: if the predicates p_1 through p_k in the body hold, then the head predicate h also holds. Thus, a Horn clause can be seen as anlogous to an inference rule:

$$\frac{p_1 \qquad \ldots p_k}{h}$$

Prolog vs. Datalog

Prolog is a general-purpose logic programming language with data types (e.g., integers, strings, lists, etc.), functions (e.g., arithmetic), and control operators (e.g., an operator known as "cut", which halts the search for a proof of a given predicate). It answers queries via a depth-first, left-to-right proof search strategy. This makes Prolog expressive and powerful, but also sensitive the order of rules and subgoals. In practice, termination and performance depend on the way that programs are written, which can be complicated in practice.

Datalog was designed to be a disciplined subset of Prolog that recovers a clean, declarative semantics closer to pure mathematical logic. It eliminates function symbols, restricts the use of logical negation, and enforces extra syntactic conditions on rules. Together, these conditions guarantee that:

- All programs terminate,
- The semantics does not depend on the order of evaluation, and
- The set of facts that can be derived is finite.

Thus, Datalog occupies a "sweet spot" between an expressive language and a robust semantics. It has been widely used in databases, as a domain-specific language (DSL) for querying and transforming graph structures, and in programming languages where it powers program analysis tools.

2 Formalizing Datalog

3 Datalog Syntax

The syntax of Datalog can be formalized as follows.

$$\begin{array}{rcll} r & ::= & h \mathrel{:-} b. & \textit{rule} \\ & | & h. & \\ h & ::= & p & \textit{head} \\ b & ::= & p_1, p_2, \ldots p_n & \textit{body} \\ p & ::= & p(t_1, \ldots t_n) & \textit{predicate} \\ t & ::= & x & \textit{term} \\ & | & n & \end{array}$$

Each variable in the head of a rule must also appear in the body—a condition that is also known as the *range restriction*.

4 Example

Suppose we have built-in facts `edge(u,v)` for the edges in a graph. The rules:

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

define the set of reachable nodes. The first rule says that x and y are reachable if they directly connected by an edge; the second rule says they are reachable if there is an intermediate node z that has an edge from x such that y is reachable from z.

We can detect cycles in the graph using the following rule:

```
cycle(x) :- reachable(x,x).
```

Note that because how we defined it, `reachable` is not necessarily reflexive—i.e., `reachable(x,x)` only holds if there is a cycle involving `x`.

5 Datalog Semantics

There are several ways to define the semantics of a Datalog program. In this lecture we'll focus on the simplest and most direct, a so-called model-theoretic semantics.

The constants appearing in a program P form the *Herbrand universe*:

$$HU(P) = \mathit{Const}(P).$$

In our setting, this is just the set of integers n appearing in the program.

An atomic predicate $p(n_1, \ldots, n_k)$ with no variables is called a *ground atom*. The set of ground atoms over predicates and constants form the *Herbrand base*:

$$HB(P) = \{\, p(n_1, \ldots, n_k) \mid n_i \in HU(P) \,\}.$$

Given a rule, suppose we substitute the variables with each constant in $HU(P)$. Let $Ground(P)$ denote the set of all such rules, called *ground instances*.

We can define the *immediate consequence* operator as follows:

$$T_P(I) = \{\, h \mid (h : -p_1, \ldots, p_k) \in Ground(P) \text{ and } p_1, \ldots, p_k \in I \,\}.$$

The T_P operator is monotone:

$$I \subseteq J \Rightarrow T_p(I) \subseteq T_P(J)$$

Intuitively, monotonicity means that if we add additional built-in facts or rules, then the meaning of a program can only grow. The same is *not* true of Prolog, which supports non-monotonic reasoning.

As the Herbrand base is finite, $T_P(I)$ reaches a fixed point in finitely many iterations. We can compute the meaning of a program by iterating the T_P operator to a fixed point, starting from the empty set:

$$\begin{aligned} I_0 &\triangleq \emptyset \\ I_{i+1} &\triangleq T_P(I_i) \end{aligned}$$

Definition (Meaning of Datalog Program)**.** $M(P) \triangleq \mathit{fix}(T_P)$

6 Extensions

Many extensions to Datalog have been studied, including with negation, datatypes, and more. One well-studied variant is Datalog with *stratified negation*, where negation is only used with previously-defined relations.

More formally, a Datalog program is *stratified* if its predicates can be partitioned into layers (strata) $S_0, S_1, \ldots, S_n$ such that:

- If P depends *positively* on Q, then $\mathrm{stratum}(P) \geq \mathrm{stratum}(Q)$.
- If P depends *negatively* on Q, then $\mathrm{stratum}(P) \rangle \mathrm{stratum}(Q)$.

The intuition is that lower strata computed first; higher strata may use negation involving lower strata. However, no predicate may depend *negatively* on itself, even indirectly. Other extensions of Datalog include:

- Aggregation (min, count, etc.)
- Arithmetic (plus, times, etc.)
- Datatypes (lists, trees, etc.)

For further details, see the excellent Foundations of Databases textbook.