Technische Universität Dresden

Modern Datalog

Concepts, Methods, Applications

Markus Krötzsch

TU Dresden
Knowledge-Based Systems

Reasoning Web Summer School 2025
Istanbul, Turkey

These slides are accompanied by a tutorial chapter in the lecture notes.
Full text is available at
https://iccl.inf.tu-dresden.de/web/Inproceedings3435/en

Introduction to Datalog

The Simplest Rule Language

$$\text{hasUncle}(x, z) \leftarrow \text{hasParent}(x, y), \text{hasBrother}(y, z)$$

Some terminology:

- terms can be variables (e.g., x, y, z) or constants
- predicates denote relations; they have an arity (e.g., hasUncle has arity 2)
- atoms are constructed from predicates and terms (e.g., $\text{hasUncle}(x, z)$)

Definition 1: A Datalog rule is an expression of the form:

$$H \leftarrow B_1, \ldots, B_m$$

where H and $B_1, \ldots, B_m$ are atoms. H is called the head or conclusion; $B_1, \ldots, B_m$ is called the body or premise. A rule with empty body ($m = 0$) is called a fact. A ground rule is one without variables (i.e., all terms are constants).

Datlog Syntax in the Wild

```
          hasUncle(X, Z) :- hasFather(X, Y), hasBrother(Y, Z) .

          hasUncle(x, z) :- hasFather(x, y), hasBrother(y, z) .

        hasUncle(?x, ?z) :- hasFather(?x, ?y), hasBrother(?y, ?z) .

     [?x, :hasUncle, ?z] :- [?x, :hasFather, ?y], [?y, :hasBrother, ?z] .

 hasUncle(x, z) distinct :- hasFather(x, y), hasBrother(y, z) ;

      [(has-uncle ?x ?z)    [?x :has-father ?y] [?y :hasBrother ?z]]

hasUncle(p: x, uncle: z) :- hasFather(child: x, father: y),
                            hasBrother(p: y, brother: z) .

{ ?X :hasFather ?Y . ?Y :hasBrother ?Z .} => { ?X :hasUncle ?Z } .
```

Top to bottom: Prolog/ASP, Soufflé, Nemo, RDFox, Logica, Datomic, Percial, N3

From Rules to Programs

Example:

$$\begin{aligned}
\text{father}(\text{alice}, \text{bob}) \\
\text{mother}(\text{alice}, \text{cho}) \\
\text{father}(\text{cho}, \text{daniel}) \\
\text{mother}(\text{cho}, \text{eiko}) \\
\text{mother}(\text{finley}, \text{eiko}) \\
\text{parent}(x, y) &\leftarrow \text{father}(x, y) \\
\text{parent}(x, y) &\leftarrow \text{mother}(x, y) \\
\text{ancestor}(x, y) &\leftarrow \text{parent}(x, y) \\
\text{ancestor}(x, z) &\leftarrow \text{ancestor}(x, y), \text{parent}(y, z) \\
\text{commonAnc}(x) &\leftarrow \text{ancestor}(\text{alice}, x), \text{ancestor}(\text{finley}, x)
\end{aligned}$$

Trying it out

We will use Nemo for hands-on exercises.

Web app: `https://tools.iccl.inf.tu-dresden.de/nemo/next/`

Open previous example online: `https://tud.link/41v68f`

Task: Define an auncle (aunt or uncle) relation.
Hint: You might need inequality `!=`.

Nemo is free and open source. Issue reports and feature requests are welcome.

Datalog Programs as Functions

Idea:
Datalog programs represent functions from sets of input facts and to sets of output facts.

Definition 2: A Datalog program is a triple $\langle P, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$, where

- P is a finite set of rules,
- $\mathbf{P}_{\text{in}}$ is a non-empty set of input predicates that do not occur in rule heads of P, and
- $\mathbf{P}_{\text{out}}$ is a non-empty set of output predicates.

We require all predicates in $\mathbf{P}_{\text{in}}$ and $\mathbf{P}_{\text{out}}$ to occur in P.
Input predicates are also called EDB predicates, and all other (non-input) predicates are also called IDB predicates.

Note 1: Normally we want input facts to change, not to be a static part of programs.
Note 2: Finite sets of facts are often called databases, so Datalog maps input databases to output databases.

A Datalog Program

$$\begin{aligned}
\text{father}(\text{alice}, \text{bob}) \\
\text{mother}(\text{alice}, \text{cho}) \\
\text{father}(\text{cho}, \text{daniel}) \\
\text{mother}(\text{cho}, \text{eiko}) \\
\text{mother}(\text{finley}, \text{eiko}) \\
\text{parent}(x, y) &\leftarrow \text{father}(x, y) \\
\text{parent}(x, y) &\leftarrow \text{mother}(x, y) \\
\text{ancestor}(x, y) &\leftarrow \text{parent}(x, y) \\
\text{ancestor}(x, z) &\leftarrow \text{ancestor}(x, y), \text{parent}(y, z) \\
\text{commonAnc}(x) &\leftarrow \text{ancestor}(\text{alice}, x), \text{ancestor}(\text{finley}, x)
\end{aligned}$$

Input (=EDB) predicates: father, mother

Output predicates: commonAnc

IDB predicates: parent, ancestor, commonAnc

An Equivalent Datalog Program

$$\text{father}(\text{alice}, \text{bob})$$
$$\text{mother}(\text{alice}, \text{cho})$$
$$\text{father}(\text{cho}, \text{daniel})$$
$$\text{mother}(\text{cho}, \text{eiko})$$
$$\text{mother}(\text{finley}, \text{eiko})$$
$$\text{ancestor}(x, y) \leftarrow \text{father}(x, y)$$
$$\text{ancestor}(x, y) \leftarrow \text{mother}(x, y)$$

$$\text{ancestor}(x, z) \leftarrow \text{ancestor}(x, y), \text{ancestor}(y, z)$$
$$\text{commonAnc}(x) \leftarrow \text{ancestor}(\text{alice}, x), \text{ancestor}(\text{finley}, x)$$

Input (=EDB) predicates: father, mother

Output predicates: commonAnc

IDB predicates: ancestor, commonAnc

Why Datalog?

Why Datalog?

The semantics of Datalog is defined in mathematical logic. What are the potential advantages?

- Declarativity: Programs describe high-level logical relationships, not detailed computational process
- System and Platform Independence: Portable to new computing environments
- Optimisability and Performance: Reasoners can optimise computation
- Explainability: Results are justifiable and traceable
- Verifiability and Certifiability: Correctness can be proven independently
- Intuitive Understandability: Logic capturing natural
- Conciseness and Fast Development: Programs can focus on essentials of task at hand; program logic independent of underlying data structures

What is it good for? (1)

Application Area 1: Rule-Based KRR

Datalog-like rules occur in many applications of formal logic:

- Direct use of rules in many reasoning approaches (e.g., qualitative spatial reasoning)
- Many logics admit Datalog-based reasoning mechanisms (e.g., reasoning for OWL EL ontologies)
- Sometimes reasoning subtasks can be outsourced to Datalog (e.g., grounding in ASP, unit propagation in theorem proving)

Typical system requirements:

- Fast, reactive systems; typically main-memory based
- Pure Datalog, limited need for extensions

What is it good for? (2)

Application Area 2: Analysis of Structured Data

Datalog is great for analysing structured data, especially with nested/recursive structures:

- Program analysis (e.g., CodeQuest)
- Data flow and control flow analysis (e.g., DOOP/Soufflé)
- HTML analsyis and data extraction (e.g., DIADEM)
- Analystical processing of structured data bases (e.g., legal conformance checks in health care)

Typical system requirements:

- Interactive or batch processing, depending on use case
- Data-related extensions (datatypes, built-ins, aggregation)
- Possibly domain-specific, structured datatypes

What is it good for? (3)

Application Area 3: Data Extraction and Transformation

Datalog allows us to define recursive views over large data collections:

- Rule-based relational DB data access (e.g., Yedalog, Logica)
- Rule-based graph DB data access (e.g., RDFox, Nemo)

Typical system requirements:

- Mostly interactive query answering
- Database bindings and matching datatypes
- Efficient update handling

What is it good for? (4)

Application Area 4: Graph and Network Analysis

Graph-like structures suggest iterative, declarative processing:

- Network analysis, e.g., PageRank centrality (e.g., EmptyHeaded, SocialLite)
- Graph algorithms, e.g., shortest path (e.g., SocialLite, Dynalog)

Typical system requirements:

- Mostly main-memory based, may or may not be batch processed
- Custom control for termination of approxiation algorithms
- Support for recursive use of aggretation (at least min and max)

Semantics of Datalog

Intuition:
Given an input database, a Datalog program derives the output database that consists of all facts that are necessarily entailed by the given input facts and rules.

- Variables in rules represent arbitrary constants

  Definition 3: A substitution σ is a partial mapping from variables to terms. A substitution is ground if it maps to constants only.

- A rule can be applied under specific substitutions of its variables

  Definition 4: Consider a database $\mathcal{D}$ and a Datalog rule ρ of the form $H \leftarrow B_1, \ldots, B_n$. A ground substitution σ is a match for ρ on $\mathcal{D}$ if (1) σ is defined exactly on the variables in ρ, and (2) $B_1\sigma, \ldots, B_n\sigma \in \mathcal{D}$.
  In this case, applying ρ to $\mathcal{D}$ under σ yields the inference $H\sigma$.

- The output of a program are the facts that follow by applying rules exhaustively.

The Consequence Operator

Definition 5: The immediate consequence operator T_P maps sets of ground facts $\mathcal{D}$ to sets of ground facts $T_P(\mathcal{D})$:

$$T_P(\mathcal{D}) = \{H\sigma \mid \text{there is some } H \leftarrow B_1, \ldots, B_n \in P \text{ with match } \sigma \text{ on } \mathcal{D}\}$$

Given a database $\mathcal{D}$, we can define a sequence of databases $\mathcal{D}_P^i$ as follows:

$$\mathcal{D}_P^0 = \mathcal{D} \qquad \mathcal{D}_P^{i+1} = \mathcal{D} \cup T_P(\mathcal{D}_P^i) \qquad \mathcal{D}_P^\infty = \bigcup_{i \geq 0} \mathcal{D}_P^i$$

Observations:

- We obtain an increasing sequence $\mathcal{D}_P^0 \subseteq \mathcal{D}_P^1 \subseteq \mathcal{D}_P^2 \subseteq \ldots \subseteq \mathcal{D}_P^\infty$ (why?)
- Only a finite number of ground facts can ever be derived from $\mathcal{D} \cup P$ (why?).
- Hence the sequence $\mathcal{D}_P^0, \mathcal{D}_P^1, \ldots$ is finite and there is some $k \geq 1$ with $\mathcal{D}_P^k = \mathcal{D}_P^\infty$.

Definition 6: The output database of P over $\mathcal{D}$ is the restriction of $\mathcal{D}_P^\infty$ to output predicates, i.e., the set $\{p(c_1, \ldots, c_n) \mid p(c_1, \ldots, c_n) \in \mathcal{D}_P^\infty, p \in \mathbf{P}_{\text{out}}\}$.

The consequence operator: Example

Datalog rules P:

$$\begin{aligned}
\text{parent}(x, y) &\leftarrow \text{father}(x, y) \\
\text{parent}(x, y) &\leftarrow \text{mother}(x, y) \\
\text{ancestor}(x, y) &\leftarrow \text{parent}(x, y) \\
\text{ancestor}(x, z) &\leftarrow \text{ancestor}(x, y), \text{parent}(y, z) \\
\text{commonAnc}(x) &\leftarrow \text{ancestor}(\text{alice}, x), \text{ancestor}(\text{finley}, x)
\end{aligned}$$

Input database $\mathcal{D}$:

$$\begin{aligned}
&\text{father}(\text{alice}, \text{bob}) \\
&\text{mother}(\text{alice}, \text{cho}) \\
&\text{mother}(\text{cho}, \text{eiko}) \\
&\text{mother}(\text{finley}, \text{eiko})
\end{aligned}$$

$\mathcal{D}_P^0 = \{\text{father}(\text{alice}, \text{bob}), \text{mother}(\text{alice}, \text{cho}), \text{mother}(\text{cho}, \text{eiko}), \text{mother}(\text{finley}, \text{eiko})\}$

$\mathcal{D}_P^1 = \mathcal{D}_P^0 \cup \{\text{parent}(\text{alice}, \text{bob}), \text{parent}(\text{alice}, \text{cho}), \text{parent}(\text{cho}, \text{eiko}), \text{parent}(\text{finley}, \text{eiko})\}$

$\mathcal{D}_P^2 = \mathcal{D}_P^1 \cup \{\text{ancestor}(\text{alice}, \text{bob}), \text{ancestor}(\text{alice}, \text{cho}), \text{ancestor}(\text{cho}, \text{eiko}), \text{ancestor}(\text{finley}, \text{eiko})\}$

$\mathcal{D}_P^3 = \mathcal{D}_P^2 \cup \{\text{ancestor}(\text{alice}, \text{eiko})\}$

$\mathcal{D}_P^4 = \mathcal{D}_P^3 \cup \{\text{commonAnc}(\text{eiko})\}$

$\mathcal{D}_P^5 = \mathcal{D}_P^4 = \mathcal{D}_P^\infty$

Models of Datalog

Definition 7: An Herbrand model of P and $\mathcal{D}$ is a database $\mathcal{H}$ such that

1. $\mathcal{D} \subseteq \mathcal{H}$, and
2. for every rule $\rho \in P$ of the form $H \leftarrow B_1, \ldots, B_n$, and every match σ for ρ over $\mathcal{H}$, we also have $H\sigma \in \mathcal{H}$.

Notes:

- Herbrand models interpret constants "as themselves", hence can be defined as sets of facts
- Among all Herbrand models of P and $\mathcal{D}$, there is actually a least one (w.r.t. $\subseteq$), which coincides with $\mathcal{D}_P^\infty$.

Theorem 8: The output database of P over $\mathcal{D}$ is equal to:

- the set of all output facts that are true in $\mathcal{D}_P^\infty$,
- the set of all output facts that are true in all Herbrand models of P and $\mathcal{D}$,
- the set of all output facts that are true in all first-order models of P and $\mathcal{D}$.

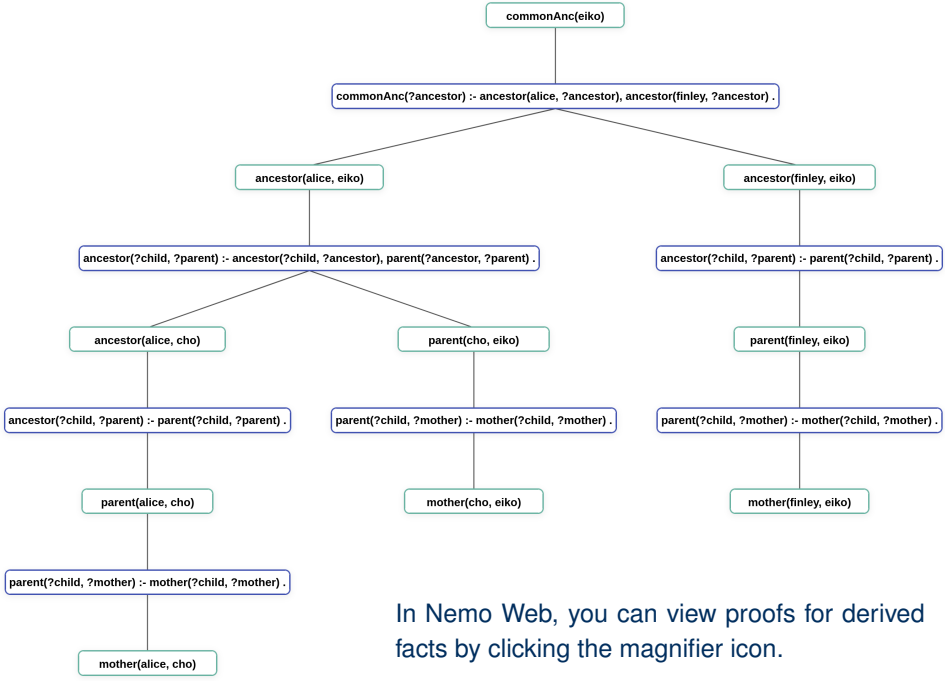
Proof trees

Definition 9: Consider a Datalog program P with input database $\mathcal{D}$. A proof tree with respect to P and $\mathcal{D}$ is a tree structure T that satisfies the following conditions:

1. every node n of T is labelled by a fact $\lambda(n)$,
2. if n is a leaf node, then $\lambda(n) \in \mathcal{D}$,
3. if n is an inner node, then n is additionally labelled by a rule $H \leftarrow B_1, \ldots, B_k \in P$ and a substitution σ, such that (1) $\lambda(n) = H\sigma$ and (2) n has exactly k child nodes $c_1, \ldots, c_k$ with $\lambda(c_i) = B_i\sigma$ for all $1 \leq i \leq k$.

If a proof tree T has root node r, we say that T is a proof for $\lambda(r)$ with respect to P and $\mathcal{D}$.

Theorem 10: The output database of P over $\mathcal{D}$ is equal to the set of all ground facts for which there is a proof with respect to P and $\mathcal{D}$.



In Nemo Web, you can view proofs for derived facts by clicking the magnifier icon.

Datalog as Second-Order Logic

Example 11: The following two programs are equivalent:

$$\begin{aligned} \text{reach}(x,y) &\leftarrow \text{edge}(x,y) \\ \text{reach}(x,z) &\leftarrow \text{reach}(x,y), \text{reach}(y,z) \\ \text{output}(z) &\leftarrow \text{reach}(c,z) \end{aligned} \qquad \begin{aligned} \text{output}(y) &\leftarrow \text{edge}(c,y) \\ \text{output}(z) &\leftarrow \text{output}(y), \text{edge}(y,z) \end{aligned}$$

Yet they do not have the same T_P operator, Herbrand models, or proof trees.

The following second-order logic formula captures the semantics more acurately:

$$\forall\ \text{Reach}, \text{Output}. \left(\left(\begin{array}{rl} (& \forall x,y. \quad \text{edge}(x,y)) \rightarrow \text{Reach}(x,y)) \wedge \\ (\forall x,y,z. & \text{Reach}(x,y) \wedge \text{Reach}(y,z) \rightarrow \text{Reach}(x,z)) \wedge \\ (& \forall z. \quad \text{Reach}(c,z) \rightarrow \text{Output}(z)) \end{array} \right) \rightarrow \text{Output}(v) \right)$$

Datalog Semantics: Summary

There are four equivalent ways of defining Datalog semantics:

- Operational semantics: least fixed point of consequence operator T_P
- Model-theoretic semantics: entailments of all/least Herbrand/FO model(s)
- Proof-theoretic semantics: every conclusion of some proof tree
- Second-order axiomatisation: Satisfying assignments in SO model checking

$\rightsquigarrow$ pleasing and reassuring agreement of various ideas, witnessing the underlying mathematical elegance

Note: Datalog is generally considered a fragment of second-order logic, but for most uses, we don't need to worry.

Datalog Complexity

How hard is reasoning with Datalog programs?

Various kinds of complexity are relevant:

- Combined complexity: based on program and database
- Data complexity: based on database; program fixed
- Program complexity: based on program; database fixed

Theorem 12: Deciding if a given fact is in the output of a Datalog program is:

- ExpTime-complete for combined complexity and program complexity
- P-complete for data complexity

Working with Real Data

Where can input data come from?

We often want to use input databases that are not given as facts in the program:

- Scalability: other formats are more suitable for large datasets
- Practicality: We don't want to edit or programs to change data
- Access: Some data sources cannot be converted to facts (size, access restrictions, legal constraints)

$\rightsquigarrow$ many systems support data imports

Commonly supported data sources include:

- CSV/TSV/DSV files: simple relational text format
- RDF: knowledge graphs in triples and quads
- SQL bindings: loading directly from relational DBMS
- SPARQL bindings: loading directly from RDF database

Excursion: RDF and SPARQL

RDF is a standard format for Web data and knowledge graphs:

- graph edges are encoded as triples $\langle \text{subject}, \text{predicate}, \text{object} \rangle$
- entities used as predicates are called properties
- abstract identifiers represented as IRIs
- extensive datatype system (strings, numbers, dates, and much more)

$\rightsquigarrow$ from a Datalog viewpoint, a 3-ary predicate + support for IRIs & data values

SPARQL is a query language standard for RDF:

- familiar SELECT ... WHERE ... structure
- Basic query patterns: RDF graph with variables as placeholders
- Many advanced features (filters, aggregates, path queries, ...)
- Results are relational tables (not RDF)

$\rightsquigarrow$ several large databases provide online SPARQL services (DBLP, Wikidata, ...)

Excursion: Wikidata

Wikidata is the knowledge base of Wikipedia

- Multi-lingual encyclopaedic knowledge graph
- Rich data model internally, exported as RDF, exposed through SPARQL
- Numeric identifiers, such as Q406 (item “Istanbul”) and P25 (property “mother”)
  $\rightsquigarrow$ find IDs at `wikidata.org` (search) or via Wikipedia article (Tools $\rightarrow$ Wikidata item)

Example: A SPARQL query for all child-mother pairs stored in Wikidata:

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?child ?mother
WHERE { ?child wdt:P25 ?mother . }
```

A Worked Example: https://tud.link/up8pdc

```
% Prefixes help to abbreviate long identifiers or URLs:
@prefix wdqs: <https://query.wikidata.org/> .
@prefix wd: <http://www.wikidata.org/entity/> .
% Parameters can be used for fixed terms throughout the program:
@parameter $personId1 = wd:Q7259 . % Ada Lovelace
@parameter $personId2 = wd:Q14045 . % Moby

% Import predicate "wdParent" (mother or father) through SPARQL:
@import wdParent :- sparql{
  endpoint=wdqs:sparql,
  query="""PREFIX wdt: <http://www.wikidata.org/prop/direct/>
    SELECT ?child ?parent WHERE { ?child (wdt:P22|wdt:P25) ?parent }"""
} .
% Import predicate "wdLabel" (English label) through SPARQL:
@import wdLabel :- sparql{
  endpoint=wdqs:sparql,
  query="""PREFIX wikibase: <http://wikiba.se/ontology#>
    SELECT ?qid ?qidLabel WHERE {
      SERVICE wikibase:label {
        <http://www.bigdata.com/rdf#serviceParam> wikibase:language "mul,en" } }"""
} .

% Find relevant ancestors, starting from selected persons:
ancestor($personId1, ?parent) :- wdParent($personId1, ?parent) .
ancestor($personId2, ?parent) :- wdParent($personId2, ?parent) .
 ancestor(?person, ?ancestor) :- ancestor(?person, ?x), wdParent(?x, ?ancestor) .
% Find common ancestors, and determine their names:
commonAnc(?qid, ?name) :- ancestor($personId1, ?qid), ancestor($personId2, ?qid),
                          wdLabel(?qid,?name) .
% Select one output predicate:
@output commonAnc .
```

Stratification: Computation in Layers

Negation

Negation enables us to ask for the absence of some data or inference.

Example 13: Using negation, a query for living people born in Istanbul might be expressed as follows:

$$\text{hasDied}(x) \leftarrow \text{dateOfDeath}(x, y)$$
$$\text{result}(x) \leftarrow \text{bornIn}(x, \text{istanbul}), \neg\text{hasDied}(x)$$

A negated ground atom $\neg A$ is true over a database $\mathcal{D}$ if $A \notin \mathcal{D}$. So we can define:

$$T_P(\mathcal{D}) = \{H\sigma \mid H \leftarrow B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P,$$
$$B_1\sigma, \ldots, B_n\sigma \in \mathcal{D}, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin \mathcal{D}$$
$$\text{for some ground substitution } \sigma\}$$

We could then use this step-wise consequence operator to compute conclusions as before . . . but there are some problems with that.

Semantics of negation: Unsafe variables

What is the meaning of the following rule?

$$\text{result}(x) \leftarrow \text{bornIn}(x, \text{istanbul}), \neg\text{dateOfDeath}(x, y)$$

According to our definition of T_P: "Find all x, such that x is born in Instanbul and there is a date y, such that x did not die on y." (All variables, including y, are universally quantified over the whole rule.)

Many systems do not allow this at all: they require that all variables in negated atoms are safe, i.e., occur in non-negated atoms as well.

Some systems allow unsafe variables but assume that their universal quantifier is below the negation: "Find all x, such that x is born in Istanbul and for all dates y, we find that x did not die on y." (example systems: Clingo, Nemo)

Definition 14: A rule is safe if all of its variables occur in non-negated atoms in its body.

Requiring all rules to be safe does not restrict expressivity (why?).

Semantics of Negation: Recursion

Even if rules are safe, the unrestricted use of negation in recursive queries leads to semantic problems:

Example 15: Consider the following facts and query:

$$\text{human(alice)}$$
$$\text{underage}(x) \leftarrow \text{human}(x), \neg\text{adult}(x)$$
$$\text{adult}(x) \leftarrow \text{human}(x), \neg\text{underage}(x)$$

What should be the result if adult and underage were output predicates?

If we define the sequence $\mathcal{D}_P^i$ as before, we obtain:

- $\mathcal{D}_P^0 = \mathcal{D} = \{\text{human}(alice))\}$
- $\mathcal{D}_P^1 = \mathcal{D} \cup T_P(\mathcal{D}_P^0) = \mathcal{D} \cup \{\text{underage(alice)}, \text{adult(alice)}\}$
- $\mathcal{D}_P^2 = \mathcal{D} \cup T_P(\mathcal{D}_P^1) = \mathcal{D}_P^0$
- $\mathcal{D}_P^3 = \mathcal{D} \cup T_P(\mathcal{D}_P^2) = \mathcal{D}_P^1 = \mathcal{D}_P^\infty$

$\rightsquigarrow$ non-monotonic behaviour leads to unfounded conclusions

Stratified Negation

Observation: Iterative evaluation of rules fails if negation is freely used in recursion

- Initially, when no facts were derived, many negated atoms are true
- However, these initially true atoms can become false when more inferences are computed

To avoid recursion through negation, one can try to organise rules in "layers" or "strata":

Definition 16: Let P be a set of rules with negation. A function ℓ that assigns a natural number $\ell(p)$ to every predicate p is a stratification of P if the following are true for every rule $h(\mathbf{t}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P$:

1. $\ell(h) \geq \ell(p_i)$ for all $i \in \{1, \ldots, n\}$
2. $\ell(h) > \ell(q_i)$ for all $i \in \{1, \ldots, m\}$

Intuition: The function ℓ defines the "level" of the rule. By applying rules exhaustively level-by-level, we can avoid non-monotonic behaviour.

Evaluating Stratified Rules

Evaluation of stratified programs: Let D be a database and let P be a program with stratification ℓ, with values of ℓ ranging from 1 to L (without loss of generality).

- For $i \in \{1, \ldots, L\}$, we define sub-programs for each stratum:
  $P_i = \{h(\mathbf{t}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}), \neg q_1(\mathbf{r_1}), \ldots, \neg q_m(\mathbf{r_m}) \in P \mid \ell(h) = i\}$
- Define $\mathcal{D}_0^\infty = \mathcal{D}$, and define for each $i = 1, \ldots, L$:
  - $\mathcal{D}_i^0 = \mathcal{D}_{i-1}^\infty$
  - $\mathcal{D}_i^{j+1} = \mathcal{D}_{i-1}^\infty \cup T_{P_i}(\mathcal{D}_i^j)$
  - $\mathcal{D}_i^\infty = \bigcup_{j \geq 1} \mathcal{D}_i^j$ is the limit of this process
- The evaluation of P over $\mathcal{D}$ is $\mathcal{D}_L^\infty$.

Observations:

- For every i, the sequence $\mathcal{D}_i^1 \subseteq \mathcal{D}_i^2 \subseteq \ldots$ is increasing, since facts relevant for negated body literals are not produced in any $\mathcal{D}_i^j$ (due to stratification)
- Such increasing sequences must be finite (since the set of all possible facts is finite)

$\rightsquigarrow$ The limits $\mathcal{D}_i^\infty$ are computed after finitely many steps

The Perfect Model

Summary: The stratified evaluation of rules terminates after finitely many steps (bounded by the number of possible facts)

What is the set of facts that we obtain from this procedure?

Fact 17: For a database $\mathcal{D}$ and stratified program P, the set of facts $\mathcal{M}$ that is obtained by the stratified evaluation procedure is the least set of facts with the property that

$$\mathcal{M} = \mathcal{D} \cup T_P(\mathcal{M}).$$

In particular, $\mathcal{M}$ does not depend on the stratification that was chosen.

M is called perfect model or unique stable model in logic programming.

Intuition: The stratified evaluation is the smallest set of self-supporting true facts that can be derived

- This is not the set of inferences under classical logical semantics! (why?)
- But it is a good extension of negation in queries to the recursive setting.

Obtaining a Stratification

To find a stratification, the following algorithm can be used:

Input: program P

- Construct a directed graph with two types of edges, $\overset{+}{\rightarrow}$ and $\overset{-}{\rightarrow}$:
  - The vertices are the predicate symbols in P
  - $p \overset{+}{\rightarrow} q$ if there is a rule with p in its non-negated body and q in the head
  - $p \overset{-}{\rightarrow} q$ if there is a rule with p in its negated body and q in the head
- Then P is stratified if and only if the graph contains no directed cycle that involves an edge $\overset{-}{\rightarrow}$
- In this case, we can obtain a stratification as follows:
  (1) produce a topological order of the strongly connected components of this directed graph (without distinguishing edge types), e.g., using Tarjan's algorithm
  (2) assign numerical strata bottom-up to all predicates in each component

Discussion: Declarativity of Stratified Negation

How far have we moved away from the sematic elegance of pure Datalog?

- Operational semantics: modified consequence operator based on stratification
- Model-theoretic semantics: entailments based on perfect model
- Proof-theoretic semantics: unclear/unsatisfactory
- Second-order axiomatisation: conceivable, but (even) more technical

Simpler special cases:

- Input negation: Only atoms with EDB predicates may be negated
- Inequality: Support a predicate $\neq$ that expresses non-identity

$\rightsquigarrow$ proof trees are more compelling for these cases

Hands-on: Finding last common ancestors

We work with the Wikidata-based common ancestor example as before:
`https://tud.link/up8pdc`

Task: Moby and Ada have a lot of common ancestors. Modify the program to find only the latest (most recent) among them.

Datalog Expressivity

How expressive is Datalog?

Definition 18: The expressive power of a Datalog-like language is given by the set of functions from input databases to output databases that one can express with such programs.

Recall: Datalog reasoning is

- ExpTime-complete for combined complexity and program complexity
- P-complete for data complexity

$\rightsquigarrow$ every Datalog-expressible function is computable in polynomial time!

Are all polytime computable functions between databases expressible in Datalog?
No!

Expressive Limits of Datalog

Without negation, Datalog is monotonic, but many polytime functions are not.

Examples:

- Any program that requires input negation
- Parity: derive “yes()” if there is an even number of constants
  $\leadsto$ Datalog cannot count (not even with stratified negation)

Capturing PTime: To capture all polytime functions, two features must be added:

- input negation (stratified negation only on EDB atoms)
- successor order: predicates first, succ, and last that define an (arbitrary) total order on the domain

However, programs that rely on succ are usually not practical, and not really “logical” at all.

Datatypes

Putting the Data into Datalog

Obviously, we want datatypes . . . and related functions/predicates.

Typical datatypes and functions:

- Strings and string functions (concatenation, substring, toUpper, . . .)
- Integers/floats and arithmetic functions
- RDF-specific terms (IRIs, language literals) and related functions
- . . . (calendar dates, geographic coordinates, . . .)
- . . . conversion functions (toString, toInt, round, . . .) and hash functions

$\rightsquigarrow$ available as built-in functions and predicates in Datalog systems

Strong Typing vs. Weak Typing

Two alternative ways to handle typed data in computer languages.

Strong typing

- Declare a type for all places where data may occur (typed schema)
- Ensure schema compliance statically (“at compile time”)
- Advantages: type safety, easier to implement
- Typical in: RDBMS, description logics, sorted logics
- In Datalog: Soufflé, Logica, Yedalog, Datomic (?), . . .

Weak typing

- Do not assign types to places (schema remains untyped)
- Handle type mismatches dynamically (“at runtime”)
- Advantages: flexibility, less declaration effort
- Typical in: RDF, JSON, CSV, logic programming (Prolog, ASP)
- In Datalog: Clingo, Nemo, RDFox, N3, . . .

Compromises exists, e.g., OWL ontologies distinguish “data properties” from “object properties” (strong typing), but allow mixed data values for data properties (weak typing)

Datatypes: What could go wrong?

Problem 1: Finding rule matches might be much harder with unrestricted datatype built-ins.

Possible solutions:

- Require safety: variables in built-ins must also occur in normal body atoms
- Implement datatype-related reasoning: constraint logic programming, SAT modulo theories, theorem proving, . . . (but not Datalog)

Problem 2: Inferences may contain unbounded amounts of new (computed) terms, and $\mathcal{D}_P^\infty$ may be infinite.

Possible solutions:

- Accept: let users worry about this (most systems do that)
- Enforce: add extra-logical termination mechanisms
- Analyse: restrict to cases where termination is certain (though undecidable in general)

Complex Value Types

Rule engines may also support structured types (a.k.a. complex values):

- Abstract function terms (such as $f(a, g(b))$)
- Lists, records, and tuples
- Maps and sets
- Graphs
- ...

$\rightsquigarrow$ uneven support in current systems, mainly lists/tuples/records

Datatype Support in Nemo

Nemo implements weak typing with an RDF-inspired type system and syntax

Datatype support as of Nemo v0.9:

- Integer types and functions (up to signed int 64)
- Floating point types and functions (f32, f64)
- Strings and language-tagged strings and functions, but no lexicographic orderings
- All other datatypes (RDF standard and custom types) faithfully handled, only few functions supported
- Complex value types under development (`https://github.com/knowsys/nemo/issues/699`)

Full documentation: `https://knowsys.github.io/nemo-doc/reference/builtins/`

Aggregates

Aggregation in Datalog

Aggregation functions are functions from a set of tuples (a relation) to a single tuple

Examples:

- `count` computes the cardinality of the input relation
- `sum` computes the sum of the first column in the input relation
- `max` computes the largest value of the first column in the input relation
- `min` computes the least value of the first column in the input relation

$\rightsquigarrow$ useful and possible in Datalog

Aggregation Step by Step

Essential inputs needed for aggregation:

- let R be a relation (set of tuples) of arity n,
- let G ("group by") and A ("aggregate") be disjoint lists of positions from $\{1, \ldots, n\}$,
- let f be an aggregation function.

Notation: For a tuple $\boldsymbol{t} = \langle t_1, \ldots, t_n \rangle \in R$, let $G(\boldsymbol{t}) := \langle t_{G[1]}, \ldots, t_{G[\text{len}(G)]} \rangle$, and likewise for A.

Definition 19: The aggregation of R w.r.t. G, A, and f is defined through the following steps:

1. Grouping: Let $R_G := \{G(\boldsymbol{t}) \mid \boldsymbol{t} \in R\}$ and $R_A := \{A(\boldsymbol{t}) \mid \boldsymbol{t} \in R\}$.

   The function $g : R_G \to 2^{R_A}$ is defined by setting $g(\boldsymbol{s}) := \{A(\boldsymbol{t}) \mid \boldsymbol{t} \in R, G(\boldsymbol{t}) = \boldsymbol{s}\}$.
2. Apply aggregation: The aggregated relation is $\{\langle \boldsymbol{s}, f(g(\boldsymbol{s})) \rangle \mid \boldsymbol{s} \in R_G\}$.

Datalog Syntax for Aggregation

Users should be able to answer to some questions:
Which relation R is aggregated? What G do we group by? What A do we aggregate?

Syntax in Nemo:

```
total(?org, #sum(?amount,?year)) :- emission(?org,?country,?year,?amount) .
```

- The aggregate function f is marked by #
- R is the (virtual) relation of all matching instances of the rule head
  (if two distinct body matches coincide on the head, they are treated as one)
- G are the head variables that occur outside of aggregation functions
- A are the variables inside the aggregation function

Datalog Syntax for Aggregation (2)

Syntax in Nemo:

```
orgsPerYear(?year,#count(?country)) :- emission(?org,?country,?year,?amount).
```

Syntax in Clingo:

```
orgsPerYear(YEAR, C) :- emission(_,_,_,YEAR),
                        C = #count{ ORG : emission(_,ORG,YEAR,_)} .
```

⇝ nested notation for aggregation

Syntax in Soufflé:

```
auxYearOrg(year, country) :- emission(_, country, year, _) .
     orgsPerYear(year, c) :- auxYearCountry(year, _),
                             c = count : { auxYearOrg(year, _) } .
```

⇝ helper rule needed to eliminate duplicates (cannot just count distinct organisations)

Stratified Aggregation

Aggregation is typically non-monotonic
(e.g., if we add more facts, previous counts will no longer be inferred)

Possible solution: Extend stratification to aggregates

- Same conditions for stratification of normal rules (with negation)
- For rules with aggregation, require that all predicates that we aggregate over are in a strictly lower stratum than the head predicate

$\rightsquigarrow$ aggregtion only applied to “final” sets of facts that will not change later on

Unstratified Aggregation

Stratified negation is most common in systems (e.g., Nemo, RDFox, Souffl'e), but it prevents some useful applications

Example: In a directed graph that has a cost for every edge, we might want to compute minimal-cost paths as follows:

```
          path(?s,?t,?c) :- edge(?s,?t,?c), ?c>0 .
path(?s,?t,#min(?cp+?c)) :- path(?s,?m,?cp), edge(?m,?t,?c), ?c>0 .
```

But this is not stratified, and therefore not supported. Indeed, it requires somewhat different processing.

Systems that support unstratified aggregation come in three forms:

- "Better safe than sorry" only special program shapes supported; well-defined semantics; guaranteed to be meaningful (example: SocialLite)
- "No strings attached" whatever users write down is "executed"; result may not be well-defined; complete freedom (example: Dynalog)
- "Stable models" ASP semantics; many models (example: Clingo, DLV)

Hands-on: Data integration and analysis

A new, somewhat more complex example: `https://tud.link/ps752s`

This example shows:

- How to load data from a Web-based CSV file
- How to match CSV records with Wikidata (by ID or name)
- How to use negation to implement a fallback handling (match ID before name)
- How to aggregate over partially matched data without overlooking unmatched records

Conclusions

Summary and Outlook

What we learned:

- Pure Datalog is elegant and simple, but rarely enough
- Main extensions: (stratified) negation, datatypes (weak or strong), aggregation (stratified or not)
- Datalog (and logic overall!) smoothly works with diverse data formats
- Logic has important practical benefits
- Nemo is a robust, user-friendly system

What was left out:

- Datalog can be generalised to other truth values (probabilistic, temporal, etc.)
- Other extensions, such as existential rules (= Datalog + value invention)
- Reasoning methods, optimisations, and implementation techniques

$\rightsquigarrow$ see lecture notes for pointers

Acknolwedgements and further reading

The practical hands-on exercises in these lecture notes rely on the Nemo rule engine. Thanks are due to all past and present contributors, but especially to Lukas Gerlach, Alex Ivliev, and Maximilian Marx for getting key features to work in time for this lecture.

This presentation is accompanied by an invited tutorial in the summer school lecture notes.

Reference:

- Markus Krötzsch: **Modern Datalog: Concepts, Methods, Applications.** In Alessandro Artale, Meghyn Bienvenu, Yazmín Ibáñez García, Filip Murlak, eds., Joint Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025), volume 138 of OASIcs, Dagstuhl Publishing, 2025. https://iccl.inf.tu-dresden.de/web/Inproceedings3435/en

Bonus Material: Datalog Complexity

Complexity of Datalog

How hard is answering Datalog queries?

Various kinds of complexity are relevant:

- Combined complexity: based on program and database
- Data complexity: based on database; program fixed
- Program complexity: based on program; database fixed

Plan:

- First show upper bounds (outline efficient algorithm)
- Then establish matching lower bounds (reduce hard problems)

A Simpler Problem: Ground Programs

Let's start with Datalog without variables

$\rightsquigarrow$ sets of ground rules a.k.a. propositional Horn logic program

Naive computation of $\mathcal{D}_P^\infty$:

```
01  D_P^0 := D
02  i := 0
03  repeat :
04      D_P^{i+1} := D
05      for H ← B_1, ..., B_ℓ ∈ P :
06          if {B_1, ..., B_ℓ} ⊆ D_P^i :
07              D_P^{i+1} := D_P^{i+1} ∪ {H}
08      i := i + 1
09  until D_P^{i-1} = D_P^i
10  return D_P^i
```

How long does this take?

- At most $|P|$ additional facts can be derived
- Algorithm terminates with $i \leq |P| + 1$
- In each iteration, we check each rule once (linear), and compare its body to $\mathcal{D}_P^i$ (quadratic)

$\rightsquigarrow$ polynomial runtime

Complexity of Propositional Horn Logic

Much better algorithms exist:

Theorem 20 (Dowling & Gallier, 1984): For a propositional Horn logic program P, the set $\mathcal{D}_P^\infty$ can be computed in linear time.

Nevertheless, the problem is not trivial:

Theorem 21: For a propositional Horn logic program P and a proposition (or ground atom) A, deciding if $A \in \mathcal{D}_P^\infty$ is a P-complete problem.

Remark:
all P problems can be reduced to propositional Horn logic entailment
yet not all problems in P (or even in NL) can be solved in linear time!

Datalog Complexity: Upper Bounds

A straightforward approach:

(1) Compute the grounding ground(P) of P w.r.t. the database $\mathcal{D}$

(2) Compute $\mathcal{D}^{\infty}_{\text{ground}(P)}$

Complexity estimation:

- The number of constants N for grounding is linear in P and $\mathcal{D}$
- A rule with m distinct variables has N^m ground instances
- Step (1) creates at most $|P| \cdot N^M$ ground rules, where M is the maximal number of variables in any rule in P
  - ground(P) is polynomial in the size of $\mathcal{D}$
  - ground(P) is exponential in P
- Step (2) can be executed in linear time in the size of ground(P)

Summing up: the algorithm runs in P data complexity and in ExpTime query and combined complexity

Datalog Complexity

These upper bounds are tight:

Theorem 22: Deciding if a given fact is in the output of a Datalog program is:

- ExpTime-complete for combined complexity
- ExpTime-complete for program complexity
- P-complete for data complexity

It remains to show the lower bounds.

P-Hardness of Data Complexity

We need to reduce a P-hard problem to Datalog query answering
$\rightsquigarrow$ propositional Horn logic programming

We restrict to a simple form of propositional Horn logic:

- facts have the usual form H
- all other rules have the form $H \leftarrow B_1, B_2$

Deciding fact entailment is still P-hard (why?)

We can store such programs in a database:

- For each fact H, the database has a tuple fact(H)
- For each rule $H \leftarrow B_1, B_2$,
  the database has a tuple rule(H, B_1, B_2)

P-Hardness of Data Complexity (2)

The following Datalog program acts as an interpreter for propositional Horn logic programs:

$$\text{True}(x) \leftarrow \text{fact}(x)$$
$$\text{True}(x) \leftarrow \text{rule}(x, y, z), \text{True}(y), \text{True}(z)$$

Easy observations:

- $\text{True}(A)$ is derived if and only if A is a consequence of the original propositional program
- The encoding of propositional programs as databases can be computed in logarithmic space
- The Datalog program is the same for all propositional programs

$\rightsquigarrow$ Datalog reasoning is P-hard for data complexity

ExpTime-Hardness of Program Complexity

A direct proof:
Encode the computation of a deterministic Turing machine for up to exponentially many steps

Recall that $\text{ExpTime} = \bigcup_{k \geq 1} \text{Time}(2^{n^k})$

- in our case, $n = N$ is the number of database constants
- k is some constant

$\rightsquigarrow$ we need to simulate up to 2^{N^k} steps (and tape cells)

Main ingredients of the encoding:

- $\text{state}_q(X)$: the TM is in state q after X steps
- $\text{head}(X, Y)$: the TM head is at tape position Y after X steps
- $\text{symbol}_\sigma(X, Y)$: the tape cell at position Y holds symbol σ after X steps

$\rightsquigarrow$ How to encode 2^{N^k} time points X and tape positions Y?

Preparing for a Long Computation

We need to encode 2^{N^k} time points and tape positions
$\rightsquigarrow$ use binary numbers with N^k digits

So X and Y in atoms like head(X, Y) are really lists of variables $X = x_1, \ldots, x_{N^k}$ and $Y = y_1, \ldots, y_{N^k}$, and the arity of head is $2 \cdot N^k$.

TODO: define predicates that capture the order of N^k-digit binary numbers

For each number $i \in \{1, \ldots, N^k\}$, we use predicates:

- succ$^i(X, Y)$: $X + 1 = Y$, where X and Y are i-digit numbers
- first$^i(X)$: X is the i-digit encoding of 0
- last$^i(X)$: X is the i-digit encoding of $2^i - 1$

Finally, we can define the actual order for $i = N^k$

- $\leq^i (X, Y)$: $X \leq Y$, where X and Y are i-digit numbers

Defining a Long Chain

We can define $\text{succ}^i(X, Y)$, $\text{first}^i(X)$, and $\text{last}^i(X)$ as follows:

$$\begin{aligned}
&\text{succ}^1(0,1) \quad \text{first}^1(0) \quad \text{last}^1(1) \\
&\left.\begin{aligned}
\text{succ}^{i+1}(0,X,0,Y) &\leftarrow \text{succ}^i(X,Y) \\
\text{succ}^{i+1}(1,X,1,Y) &\leftarrow \text{succ}^i(X,Y) \\
\text{succ}^{i+1}(0,X,1,Y) &\leftarrow \text{last}^i(X), \text{first}^i(Y) \\
\text{first}^{i+1}(0,X) &\leftarrow \text{first}^i(X) \\
\text{last}^{i+1}(1,X) &\leftarrow \text{last}^i(X)
\end{aligned}\right\} \begin{array}{l}\text{for } X = x_1, \ldots, x_i \\ \text{and } Y = y_1, \ldots, y_i \\ \text{lists of } i \text{ variables}\end{array}
\end{aligned}$$

Now for $M = N^k$, we define $\leq^M(X, Y)$ as the reflexive, transitive closure of $\text{succ}^M(X, Y)$:

$$\begin{aligned}
&\leq^M(X,X) \leftarrow \\
&\leq^M(X,Z) \leftarrow \leq^M(X,Y), \text{succ}^M(Y,Z)
\end{aligned}$$

Initialising the Computation

We can now encode the initial configuration of the Turing Machine for an input word $\sigma_1 \cdots \sigma_n \in (\Sigma \setminus \{\sqcup\})^*$.

We write B_i for the binary encoding of a number i with $M = N^k$ digits.

$$\mathsf{state}_{q_0}(B_0) \qquad \text{where } q_0 \text{ is the TM's initial state}$$

$$\mathsf{head}(B_0, B_0)$$

$$\mathsf{symbol}_{\sigma_i}(B_0, B_i) \qquad \text{for all } i \in \{1, \ldots, n\}$$

$$\mathsf{symbol}_{\sqcup}(B_0, Y) \leftarrow \leq^M(B_{n+1}, Y) \qquad \text{where } Y = y_1, \ldots, y_M$$

TM Transition and Acceptance Rules

For each transition $\langle q, \sigma, q', \sigma', d \rangle \in \Delta$, we add rules:

$$\text{symbol}_{\sigma'}(X', Y) \leftarrow \text{succ}^M(X, X'), \text{head}(X, Y), \text{symbol}_{\sigma}(X, Y), \text{state}_q(X)$$
$$\text{state}_{q'}(X') \leftarrow \text{succ}^M(X, X'), \text{head}(X, Y), \text{symbol}_{\sigma}(X, Y), \text{state}_q(X)$$

Similar rules are used for inferring the new head position (depending on d)

Further rules ensure the preservation of unaltered tape cells:

$$\begin{aligned}\text{symbol}_{\sigma}(X', Y) \leftarrow\ & \text{succ}^M(X, X'), \text{symbol}_{\sigma}(X, Y), \\ & \text{head}(X, Z), \text{succ}^M(Z, Z'), \leq^M(Z', Y) \\ \text{symbol}_{\sigma}(X', Y) \leftarrow\ & \text{succ}^M(X, X'), \text{symbol}_{\sigma}(X, Y), \\ & \text{head}(X, Z), \text{succ}^M(Z', Z), \leq^M(Y, Z')\end{aligned}$$

The TM accepts if it ever reaches the accepting state q_{acc}:

$$\text{accept}() \leftarrow \text{state}_{q_{\text{acc}}}(X)$$

Hardness Results

Lemma 23: A deterministic TM accepts an input in $\textsc{Time}(2^{n^k})$ if and only if the Datalog program defined above entails the fact accept().

We obtain ExpTime-hardness of Datalog query answering:

- The decision problem of any language in ExpTime can be solved by a deterministic TM in $\textsc{Time}(2^{n^k})$ for some constant k
- For any input word w, we can reduce acceptance of w by $\mathcal{M}$ in $\textsc{Time}(2^{n^k})$ to entailment of accept() by a Datalog program $P(w, \mathcal{M}, k)$
- $P(w, \mathcal{M}, k)$ is polynomial in the size of $\mathcal{M}$ and w (in fact: in logarithmic space), whereas k can be considered constant in this context

ExpTime-Hardness: Notes

Some further remarks on our construction:

- The constructed program does not use EDB predicates
  $\leadsto$ database can be empty
- Therefore, hardness extends to query complexity
- Using a fixed (very small) database, we could have avoided the use of constants
- We used IDB predicates of unbounded arity
  $\leadsto$ they are essential for the claimed hardness