All lectures and the exercise on 19 June 2018 will be in room APB 1004
Introduction to Datalog
Introduction to Datalog

Datalog introduces **recursion** into database queries

- Use deterministic rules to derive new information from given facts
- Inspired by logic programming (Prolog)
- However, no function symbols and no negation
- Studied in AI (knowledge representation) and in databases (query language)

**Example 11.1:** Transitive closure $C$ of a binary relation $r$

$$
C(x, y) \leftarrow r(x, y)
$$

$$
C(x, z) \leftarrow C(x, y) \land r(y, z)
$$

**Intuition:**

- some facts of the form $r(x, y)$ are given as input, and the rules derive new conclusions $C(x, y)$
- variables range over all possible values (implicit universal quantifier)
Recall: A term is a constant or a variable. An atom is a formula of the form $R(t_1, \ldots, t_n)$ with $R$ a predicate symbol (or relation) of arity $n$, and $t_1, \ldots, t_n$ terms.

Definition 11.2: A Datalog rule is an expression of the form:

$$H \leftarrow B_1 \land \ldots \land B_m$$

where $H$ and $B_1, \ldots, B_m$ are atoms. $H$ is called the head or conclusion; $B_1 \land \ldots \land B_m$ 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).

A set of Datalog rules is a Datalog program.
Datalog: Example

father(alice, bob)
mother(alice, carla)
mother(evan, carla)
father(carla, david)

\begin{align*}
\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{Parent}(x, y) \land \text{Ancestor}(y, z) \\
\text{SameGeneration}(x, x) \\
\text{SameGeneration}(x, y) &\leftarrow \text{Parent}(x, v) \land \text{Parent}(y, w) \land \text{SameGeneration}(v, w)
\end{align*}
What does a Datalog program express?
Usually we are interested in entailed ground atoms

What can be entailed? Informally:

- Restrict to set of constants that occur in program (finite)
  \[ \sim \text{universe } U \]
- Variables can represent arbitrary constants from this set
  \[ \sim \text{ground substitutions map variables to constants} \]
- A rule can be applied if its body is satisfied for some ground substitution

**Example 11.3:** The rule `Parent(x, y) ← mother(x, y)` can be applied to `mother(alice, carla)` under substitution \( \{x \mapsto alice, y \mapsto carla\} \).

- If a rule is applicable under some ground substitution, then the according instance of the rule head is entailed.
An inductive definition of what can be derived:

**Definition 11.4:** Consider a Datalog program $P$. The set of ground atoms that can be derived from $P$ is the smallest set of atoms $A$ for which there is a rule $H \leftarrow B_1 \land \ldots \land B_n$ and a ground substitution $\theta$ such that

- $A = H\theta$, and
- for each $i \in \{1, \ldots, n\}$, $B_i\theta$ can be derived from $P$.

**Notes:**

- $n = 0$ for ground facts, so they can always be derived (induction base)
- if variables in the head do not occur in the body, they can be any constant from the universe
We can think of deductions as tree structures:

\[
\begin{align*}
\text{Ancestor}(\text{alice}, \text{david}) & \leftarrow \text{Parent}(\text{alice}, \text{carla}) \\
& \quad \leftarrow \text{mother}(\text{alice}, \text{carla}) \\
\text{Parent}(\text{alice}, \text{carla}) & \quad \leftarrow \text{father}(\text{alice}, \text{bob}) \\
& \quad \leftarrow \text{mother}(\text{alice}, \text{carla}) \\
& \quad \leftarrow \text{mother}(\text{evan}, \text{carla}) \\
\text{father}(\text{carla}, \text{david}) & \quad \leftarrow \text{Parent}(\text{carla}, \text{david}) \\
\text{mother}(\text{alice}, \text{carla}) & \quad \leftarrow \text{mother}(\text{alice}, \text{carla}) \\
\text{Parent}(\text{carla}, \text{david}) & \quad \leftarrow \text{father}(\text{carla}, \text{david}) \\
\text{Parent}(\text{carla}, \text{david}) & \quad \leftarrow \text{mother}(\text{carla}, \text{david}) \\
\text{Ancestor}(\text{carla}, \text{david}) & \leftarrow \text{Parent}(\text{carla}, \text{david}) \wedge \text{Ancestor}(\text{y}, \text{z})
\end{align*}
\]
Instead of using substitutions, we can also ground programs:

**Definition 11.5:** The **grounding** ground($P$) of a Datalog program $P$ is the set of all ground rules that can be obtained from rules in $P$ by uniformly replacing variables with constants from the universe.

Derivations are described by the **immediate consequence operator** $T_P$ that maps sets of ground facts $I$ to sets of ground facts $T_P(I)$:

- $T_P(I) = \{ H \mid H \leftarrow B_1 \land \ldots \land B_n \in \text{ground}(P) \text{ and } B_1, \ldots, B_n \in I \}$
- Least fixed point of $T_P$: smallest set $L$ such that $T_P(L) = L$
- Bottom-up computation: $T_P^0 = \emptyset$ and $T_P^{i+1} = T_P(T_P^i)$
- The least fixed point of $T_P$ is $T_P^\infty = \bigcup_{i \geq 0} T_P^i$ (exercise)

**Observation:** Ground atom $A$ is derived from $P$ if and only if $A \in T_P^\infty$
We can also read Datalog rules as universally quantified implications.

**Example 11.6:** The rule

\[
\text{Ancestor}(x, z) \leftarrow \text{Parent}(x, y) \land \text{Ancestor}(y, z)
\]

corresponds to the implication

\[
\forall x, y, z. \text{Parent}(x, y) \land \text{Ancestor}(y, z) \rightarrow \text{Ancestor}(x, z).
\]

A set of FO implications may have many models.

\(\sim\) consider **least model** over the domain defined by the universe.

**Theorem 11.7:** A fact is entailed by the least model of a Datalog program if and only if it can be derived from the Datalog program.
Datalog Semantics: Overview

There are three equivalent ways of defining Datalog semantics:

- **Proof-theoretic:** What can be proven deductively?
- **Operational:** What can be computed bottom up?
- **Model-theoretic:** What is true in the least model?

In each case, we restrict to the universe of given constants.

~ similar to active domain semantics in databases
How can we use Datalog to query databases?

\( \rightarrow \) View database as set of ground facts

\( \rightarrow \) Specify which predicate yields the query result

**Definition 11.8:** A Datalog query is a pair \( \langle R, P \rangle \), where \( P \) is a Datalog program and \( R \) is the answer predicate.

The result of the query is the set of \( R \)-facts entailed by \( P \).
Datalog as a Query Language

How can we use Datalog to query databases?

~ View database as set of ground facts
~ Specify which predicate yields the query result

**Definition 11.8**: A Datalog query is a pair \(\langle R, P \rangle\), where \(P\) is a Datalog program and \(R\) is the answer predicate. The result of the query is the set of \(R\)-facts entailed by \(P\).

Datalog queries distinguish “given” relations from “derived” ones:

- predicates that occur in a head of \(P\) are intensional database (IDB) predicates
- predicates that only occur in bodies are extensional database (EDB) predicates

**Requirement**: database relations used as EDB predicates only
A conjunctive query $\exists y_1, \ldots, y_m.A_1 \land \ldots \land A_\ell$ with answer variables $x_1, \ldots, x_n$ can be expressed as a Datalog query $\langle \text{Ans}, P \rangle$ where $P$ has the single rule:

$$\text{Ans}(x_1, \ldots, x_n) \leftarrow A_1 \land \ldots \land A_\ell$$

Unions of CQs can also be expressed (how?)

**Intuition:** Datalog generalises UCQs by adding recursion.
We can make the relationship of Datalog and UCQs more precise:

**Definition 11.9:** For a Datalog program $P$:
- An IDB predicate $R$ depends on an IDB predicate $S$ if $P$ contains a rule with $R$ in the head and $S$ in the body.
- $P$ is non-recursive if there is no cyclic dependency.

**Theorem 11.10:** UCQs have the same expressivity as non-recursive Datalog.

That is: a query mapping can be expressed by some UCQ if and only if it can be expressed by a non-recursive Datalog program.

However, Datalog can be exponentially more succinct (shorter), as illustrated in an exercise.
Theorem 11.10: UCQs have the same expressivity as non-recursive Datalog.

Proof: “Non-recursive Datalog can express UCQs”: Just discussed.

“UCQs can express non-recursive Datalog”
Theorem 11.10: UCQs have the same expressivity as non-recursive Datalog.

Proof: “Non-recursive Datalog can express UCQs”: Just discussed.

“UCQs can express non-recursive Datalog”: Obtained by resolution:

- Given rules $\rho_1 : R(s_1, \ldots, s_n) \leftarrow C_1 \land \ldots \land C_\ell$ and
  
  $\rho_2 : H \leftarrow B_1 \land \ldots \land R(t_1, \ldots, t_n) \land \ldots \land B_m$

  (w.l.o.g. having no variables in common with $\rho_1$)

- such that $R(t_1, \ldots, t_n)$ and $R(s_1, \ldots, s_n)$ unify with most general unifier $\sigma$,

- the resolvent of $\rho_1$ and $\rho_2$ with respect to $\sigma$ is

  $H\sigma \leftarrow B_1\sigma \land \ldots \land C_1\sigma \land \ldots \land C_\ell\sigma \land \ldots \land B_m\sigma$. 
Proof

**Theorem 11.10:** UCQs have the same expressivity as non-recursive Datalog.

**Proof:** "Non-recursive Datalog can express UCQs": Just discussed.

"UCQs can express non-recursive Datalog": Obtained by **resolution**:

- Given rules $\rho_1 : R(s_1, \ldots, s_n) \leftarrow C_1 \wedge \ldots \wedge C_\ell$ and $\rho_2 : H \leftarrow B_1 \wedge \ldots \wedge R(t_1, \ldots, t_n) \wedge \ldots \wedge B_m$
  (w.l.o.g. having no variables in common with $\rho_1$)
- such that $R(t_1, \ldots, t_n)$ and $R(s_1, \ldots, s_n)$ unify with most general unifier $\sigma$,
- the resolvent of $\rho_1$ and $\rho_2$ with respect to $\sigma$ is $H\sigma \leftarrow B_1\sigma \wedge \ldots \wedge C_1\sigma \wedge \ldots \wedge C_\ell\sigma \wedge \ldots \wedge B_m\sigma$.

Unfolding of $R$ means to simultaneously resolve all occurrences of $R$ in bodies of any rule, in all possible ways. After adding all these resolvents, we can delete all rules that contain $R$ in body or head (assuming that $R$ is not the answer predicate).
**Proof**

**Theorem 11.10:** UCQs have the same expressivity as non-recursive Datalog.

**Proof:** “Non-recursive Datalog can express UCQs”: Just discussed.

“UCQs can express non-recursive Datalog”: Obtained by resolution:

- Given rules \( \rho_1 : R(s_1, \ldots, s_n) \leftarrow C_1 \land \ldots \land C_\ell \) and
  \( \rho_2 : H \leftarrow B_1 \land \ldots \land R(t_1, \ldots, t_n) \land \ldots \land B_m \)
  (w.l.o.g. having no variables in common with \( \rho_1 \))
- such that \( R(t_1, \ldots, t_n) \) and \( R(s_1, \ldots, s_n) \) unify with most general unifier \( \sigma \),
- the resolvent of \( \rho_1 \) and \( \rho_2 \) with respect to \( \sigma \) is
  \( H\sigma \leftarrow B_1\sigma \land \ldots \land C_1\sigma \land \ldots \land C_\ell\sigma \land \ldots \land B_m\sigma \).

**Unfolding of** \( R \) **means to simultaneously resolve all occurrences of** \( R \) **in bodies of any rule, in all possible ways.** After adding all these resolvents, we can delete all rules that contain \( R \) in body or head (assuming that \( R \) is not the answer predicate).

Now given a non-recursive Datalog program, unfold each non-answer predicate (in any order). \( \leadsto \) program with only the answer predicate in heads (requires non-recursiveness). This is easy to express as UCQ (using equality to handle constants in heads). □
Domain independence was considered useful for FO queries
\[ \rightarrow \text{results should not change if domain changes} \]

Several solutions:

- **Active domain semantics:** restrict to elements mentioned in database or query
- **Domain-independent queries:** restrict to query where domain does not matter
- **Safe-range queries:** decidable special case of domain independence

Our definition of Datalog uses the active domain (=Herbrand universe) to ensure domain independence
Similar to safe-range FO queries, there are also simple syntactic conditions that ensure domain independence for Datalog:

**Definition 11.11:** A Datalog rule is safe if all variables in its head also occur in its body. A Datalog program/query is safe if all of its rules are.

Simple observations:

- safe Datalog queries are domain independent
- every Datalog query can be expressed as a safe Datalog query . . .
- . . . and un-safe queries are not much more succinct either (exercise)

Some texts require Datalog queries to be safe in general but in most contexts there is no real need for this.
Complexity
Complexity of Datalog

How hard is answering Datalog queries?

Recall:

- **Combined complexity**: based on query and database
- **Data complexity**: based on database; query fixed
- **Query complexity**: based on query; 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
\( \sim \) sets of ground rules a.k.a. propositional Horn logic program

Naive computation of \( T_P^\infty \):

```plaintext
01  \( T_P^0 := \emptyset \)
02  \( i := 0 \)
03  repeat :
04      \( T_P^{i+1} := \emptyset \)
05      for \( H \leftarrow B_1 \land \ldots \land B_\ell \in P : \)
06          if \( \{ B_1, \ldots, B_\ell \} \subseteq T_P^i : \)
07              \( T_P^{i+1} := T_P^{i+1} \cup \{ H \} \)
08            \( i := i + 1 \)
09  until \( T_P^{i-1} = T_P^i \)
10  return \( T_P^i \)
```
Let's start with Datalog without variables
\[ \rightarrow \text{sets of ground rules a.k.a. propositional Horn logic program} \]

Naive computation of \( T_P^\infty \):

```
01   T_P^0 := \emptyset
02   i := 0
03   repeat:
04       T_P^{i+1} := \emptyset
05       for H ← B_1 \land \ldots \land B_\ell \in P :
06           if \{B_1, \ldots, B_\ell\} \subseteq T_P^i :
07               T_P^{i+1} := T_P^{i+1} \cup \{H\}
08           i := i + 1
09       until T_P^{i-1} = T_P^i
10   return T_P^i
```

How long does this take?
- At most \(|P|\) 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 \( T_P^i \) (quadratic)

\[ \rightarrow \text{polynomial runtime} \]
Complexity of Propositional Horn Logic

Much better algorithms exist:

**Theorem 11.12 (Dowling & Gallier, 1984):** For a propositional Horn logic program $P$, the set $T^\infty_P$ can be computed in linear time.
Much better algorithms exist:

**Theorem 11.12 (Dowling & Gallier, 1984):** For a propositional Horn logic program $P$, the set $T^\infty_P$ can be computed in linear time.

Nevertheless, the problem is not trivial:

**Theorem 11.13:** For a propositional Horn logic program $P$ and a proposition (or ground atom) $A$, deciding if $A \in T^\infty_P$ 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!
A straightforward approach:

1. Compute the grounding $\text{ground}(P)$ of $P$ w.r.t. the database $I$
2. Compute $T^\infty_{\text{ground}(P)}$

Complexity estimation:

- The number of constants $N$ for grounding is linear in $P$ and $I$
- 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 $P$ - $\text{ground}(P)$ is polynomial in the size of $I$ - $\text{ground}(P)$ is exponential in $P$

Summing up: the algorithm runs in P data complexity and in ExpTime query and combined complexity
Datalog Complexity: Upper Bounds

A straightforward approach:

1. Compute the grounding $\text{ground}(P)$ of $P$ w.r.t. the database $\mathcal{I}$
2. Compute $T_{\text{ground}(P)}^\infty$

Complexity estimation:

- The number of constants $N$ for grounding is linear in $P$ and $\mathcal{I}$
- 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 $P$
  - $\text{ground}(P)$ is polynomial in the size of $\mathcal{I}$
  - $\text{ground}(P)$ is exponential in $P$
- Step (2) can be executed in linear time in the size of $\text{ground}(P)$

Summing up: the algorithm runs in \text{P data complexity} and in \text{ExpTime query and combined complexity}
These upper bounds are tight:

**Theorem 11.14:** Datalog query answering is:

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

It remains to show the lower bounds.
We need to reduce a P-hard problem to Datalog query answering
$\leadsto$ propositional Horn logic programming
We need to reduce a P-hard problem to Datalog query answering

\[ \rightarrow \text{propositional Horn logic programming} \]

We restrict to a simple form of propositional Horn logic:

- facts have the usual form \( H \leftarrow \)
- all other rules have the form \( H \leftarrow B_1 \land B_2 \)

Deciding fact entailment is still P-hard (exercise)
We need to reduce a P-hard problem to Datalog query answering
\[ \rightarrow \text{propositional Horn logic programming} \]

We restrict to a simple form of propositional Horn logic:
- facts have the usual form \( H \leftarrow \)
- all other rules have the form \( H \leftarrow B_1 \land B_2 \)

Deciding fact entailment is still P-hard (exercise)

We can store such programs in a database:
- For each fact \( H \leftarrow \), the database has a tuple \( \text{Fact}(H) \)
- For each rule \( H \leftarrow B_1 \land B_2 \),
  the database has a tuple \( \text{Rule}(H, B_1, B_2) \)
The following Datalog program acts as an interpreter for propositional Horn logic programs:

\[
\begin{align*}
\text{True}(x) & \leftarrow \text{Fact}(x) \\
\text{True}(x) & \leftarrow \text{Rule}(x, y, z) \land \text{True}(y) \land \text{True}(z)
\end{align*}
\]

**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 query answering is P-hard for data complexity
A direct proof:
Encode the computation of a deterministic Turing machine for up to exponentially many steps.
ExpTime-Hardness of Query Complexity

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

Recall that \( \text{ExpTime} = \bigcup_{k \geq 1} \text{Time}(2^{nk}) \)
- in our case, \( n = N \) is the number of database constants
- \( k \) is some constant

\( \sim \) we need to simulate up to \( 2^{nk} \) 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 \( \sigma \) after \( X \) steps

\( \sim \) How to encode \( 2^{nk} \) time points \( X \) and tape positions \( Y \)?
Preparing for a Long Computation

We need to encode $2^{N^k}$ time points and tape positions

\[ \sim \text{ use binary numbers with } N^k \text{ digits} \]

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
Preparing for a Long Computation

We need to encode $2^{N^k}$ time points and tape positions

$\sim$ 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:

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

\[
\begin{align*}
\text{succ}^1(0, 1) & \quad \text{first}^1(0) \quad \text{last}^1(1) \\
\text{succ}^{i+1}(0, X, 0, Y) & \leftarrow \text{succ}^i(X, Y) \\
\text{succ}^{i+1}(1, X, 1, Y) & \leftarrow \text{succ}^i(X, Y) \\
\text{succ}^{i+1}(0, X, 1, Y) & \leftarrow \text{last}^i(X) \land \text{first}^i(Y) \\
\text{first}^{i+1}(0, X) & \leftarrow \text{first}^i(X) \\
\text{last}^{i+1}(1, X) & \leftarrow \text{last}^i(X)
\end{align*}
\]

for \( X = x_1, \ldots, x_i \)
and \( Y = y_1, \ldots, y_i \)
lists of \( i \) variables

Markus Krötzsch, 12th June 2018
Defining a Long Chain

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

\[
\begin{align*}
\text{succ}^1(0, 1) & \quad \text{first}^1(0) \quad \text{last}^1(1) \\
\text{succ}^{i+1}(0, X, 0, Y) & \leftarrow \text{succ}^i(X, Y) \\
\text{succ}^{i+1}(1, X, 1, Y) & \leftarrow \text{succ}^i(X, Y) \\
\text{succ}^{i+1}(0, X, 1, Y) & \leftarrow \text{last}^i(X) \land \text{first}^i(Y) \\
\text{first}^{i+1}(0, X) & \leftarrow \text{first}^i(X) \\
\text{last}^{i+1}(1, X) & \leftarrow \text{last}^i(X)
\end{align*}
\]

for $X = x_1, \ldots, x_i$ and $Y = y_1, \ldots, y_i$

lists of $i$ variables

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

\[
\begin{align*}
\leq^M(X, X) & \leftarrow \\
\leq^M(X, Z) & \leftarrow \leq^M(X, Y) \land \text{succ}^M(Y, Z)
\end{align*}
\]
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.

\[
\begin{align*}
\text{state}_{q_0}(B_0) & \quad \text{where } q_0 \text{ is the TM's initial state} \\
\text{head}(B_0, B_0) & \\
\text{symbol}_{\sigma_i}(B_0, B_i) & \quad \text{for all } i \in \{1, \ldots, n\} \\
\text{symbol}_{\sqcup}(B_0, Y) & \leftarrow \leq^M (B_{n+1}, Y) & \text{where } Y = y_1, \ldots, y_M
\end{align*}
\]
TM Transition and Acceptance Rules

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

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

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

Similar rules are used for inferring the new head position (depending on $d$)
TM Transition and Acceptance Rules

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

\[
\begin{align*}
\text{symbol}_{\sigma'}(X', Y) & \leftarrow \text{succ}^M(X, X') \land \text{head}(X, Y) \land \text{symbol}_{\sigma}(X, Y) \land \text{state}_q(X) \\
\text{state}_{q'}(X') & \leftarrow \text{succ}^M(X, X') \land \text{head}(X, Y) \land \text{symbol}_{\sigma}(X, Y) \land \text{state}_q(X)
\end{align*}
\]

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

Further rules ensure the preservation of unaltered tape cells:

\[
\begin{align*}
\text{symbol}_{\sigma}(X', Y) & \leftarrow \text{succ}^M(X, X') \land \text{symbol}_{\sigma}(X, Y) \land \\
& \quad \text{head}(X, Z) \land \text{succ}^M(Z, Z') \land \leq^M(Z', Y) \\
\text{symbol}_{\sigma}(X', Y) & \leftarrow \text{succ}^M(X, X') \land \text{symbol}_{\sigma}(X, Y) \land \\
& \quad \text{head}(X, Z) \land \text{succ}^M(Z', Z) \land \leq^M(Y, Z')
\end{align*}
\]
TM Transition and Acceptance Rules

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

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

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

Further rules ensure the preservation of unaltered tape cells:

\[
\text{symbol}_\sigma(X', Y) \leftarrow \text{succ}^M(X, X') \land \text{symbol}_\sigma(X, Y) \land \\
\text{head}(X, Z) \land \text{succ}^M(Z, Z') \land \leq^M(Z', Y)
\]
\[
\text{symbol}_\sigma(X', Y) \leftarrow \text{succ}^M(X, X') \land \text{symbol}_\sigma(X, Y) \land \\
\text{head}(X, Z) \land \text{succ}^M(Z', Z) \land \leq^M(Y, Z')
\]

The TM accepts if it ever reaches the accepting state \( q_{\text{acc}} \):

\[
\text{accept()} \leftarrow \text{state}_{q_{\text{acc}}}(X)
\]
Lemma 11.15: A deterministic TM accepts an input in \( \text{Time}(2^{nk}) \) if and only if the Datalog program defined above entails the fact \( \text{accept}() \).

We obtain ExpTime-hardness of Datalog query answering:

- The decision problem of any language in ExpTime can be solved by a deterministic TM in \( \text{Time}(2^{nk}) \) for some constant \( k \)
- In particular, there are ExpTime-hard languages \( \mathcal{L} \) with suitable deterministic TM \( \mathcal{M} \) and constant \( k \)
- For any input word \( w \), we can reduce acceptance of \( w \) by \( \mathcal{M} \) in \( \text{Time}(2^{nk}) \) to entailment of \( \text{accept}() \) by a Datalog program \( P(w, \mathcal{M}, k) \)
- \( P(w, \mathcal{M}, k) \) is polynomial in \( k \) and the size of \( \mathcal{M} \) and \( w \) (in fact, it can be constructed in logarithmic space)
Some further remarks on our construction:

- The constructed program does not use EDB predicates
  \[\leadsto\text{database can be empty}\]
- 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
  \[\leadsto\text{they are essential for the claimed hardness}\]
Summary and Outlook

Datalog can overcome some of the limitations of first-order queries

Non-recursive Datalog can express UCQs

Datalog is more complex than FO query answering:
  - ExpTime-complete for query and combined complexity
  - P-complete for data complexity

Open questions:
  - Expressivity of Datalog
  - Query containment for Datalog
  - Implementation techniques for Datalog