Declarative Knowledge Processing
Lecture 9: Datalog and its extensions

Magdalena Ortiz
Knowledge Base Systems Group
Institute of Information Systems
ortiz@kr.tuwien.ac.at

WS 2012/2013
Today’s lecture

We have discussed in detail **Description Logics**

- very powerful formalisms for Knowledge Representation and Reasoning
- the most prominent language for describing conceptualizations of application domains by means of ontologies

Today we discuss:

- **Datalog**, another fundamental formalism for processing declarative knowledge
- Some of its most prominent extensions
- How these languages relate to DLs, their similarities and differences
Outline

1. Datalog
   1.1 Syntax and Semantics
   1.2 Complexity of Datalog

2. Datalog and DLs
   2.1 Adding Existential Quantification

3. Adding non-monotonic negation to Datalog
   3.1 Motivation
   3.2 Negation in Logic Programs

4. Answer Set Programming
   4.1 ASP Basics
   4.2 Disjunctive ASP
   4.3 Answer Set Solvers
   4.4 Beyond the core ASP: Extensions and frontends
Datalog

**Datalog**: logic programming language which allows recursive queries to a relational database

- It is similar to other logic programming languages (e.g., Prolog)
- Its syntax is based on FOL
- It has a purely declarative semantics: the order of clauses and goals is immaterial
- No function symbols are available.

We now introduce the logical basis of Datalog.
Syntax of Datalog

Alphabets

Three infinite alphabets:

- **Var**: a countably infinite set of variables
  By convention, finite alphanumeric strings beginning with an upper-case letter. E.g., $X$, $X1$, *Dog*, *CATCH33*.

- **Const**: a countably infinite set of constants
  By convention, finite alphanumeric strings which are either numeric or start with a lower-case letter. E.g. 10, a, *xylophone*.

- **Pred**: a countably infinite set of predicates
  By convention, finite alphanumeric strings beginning with a non-numeric lower-case character.
Syntax of Datalog

Terms

Important: There are no function symbols.

- A term is a constant or a variable

- A term \( t \) is ground iff it is a constant

- The set \( \textbf{Const} \) of all ground terms is called Herbrand Universe
Syntax of Datalog
Atoms and Literals

Each predicate symbol represents a **predicate**, i.e., a mapping

\[ D^n \rightarrow \{0, 1\} \]

from \(n\)-tuples over the domain \(D\) to 1 (=true) or 0 (=false).

- An **atom** is a formula \(p(t_1, \ldots, t_n)\), where \(p\) is a \(n\)-ary predicate symbol and \(t_1, \ldots, t_n\) are terms.

**Examples:**

\(fatherOf(\text{john}, \text{henry})\)  \(q(a, X, 12, Y)\)

- An atom is **ground** if it contains only constants as arguments
- A **literal** is an atom \(p(t_1, \ldots, t_n)\) or a negated atom \(\neg p(t_1, \ldots, t_n)\)
- in the former case, it is **positive**, and in the latter **negative**.

**Examples:**

\(fatherOf(\text{john}, \text{henry})\)  \(\neg fatherOf(\text{joe}, \text{mary})\)
Syntax of Datalog

Clauses

A clause is a finite set of literals.

- Each clause $C$ corresponds to a formula of first-order logic, given by the universal closure of the disjunction of the literals in $C$.

Example

The clause $C = \{ \neg p(X, a), \ p(Y, b) \}$ corresponds to the formula $\forall X \ \forall Y \ (\neg p(X, a) \lor p(Y, b))$.

- A Horn clause is a clause containing at most one positive literal.

Example

The above clause is a Horn clause.

The clause $\forall X \ (p(X) \lor q(X))$ is not a Horn clause.
Syntax of Datalog

Types of Clauses

We distinguish two main types of Horn clauses:

- **Facts** are positive unit clauses. They express unconditional knowledge:

  Example
  \[
  \{\text{father}(\text{john}, \text{harry})\}, \{\text{eats}(\text{joe}, X)\}
  \]
  In rule notation:
  \[
  \text{father}(\text{john}, \text{harry}) . \quad \text{eats}(\text{joe}, X).
  \]

- **Rules** are clauses with exactly one positive literal and with at least one negative literal. A rule represents conditional knowledge:

  Example
  \[
  R : \{\neg \text{grandparent}(X), \text{grandfather}(X), \neg \text{male}(X)\}. \quad \text{In rule notation:}
  \]
  \[
  \text{grandfather}(X) :- \neg \text{grandparent}(X), \text{male}(X).
  \]
Syntax of Datalog
Types of Clauses (2)

Sometimes, we may also distinguish a third type:

- **Goals** are negative unit clauses, i.e. consisting only of one negative literal.

**Example**

\[ \{ \neg \text{fatherof}(\text{john}, X) \} \].

In rule notation:

\[ ?- \text{fatherof}(\text{john}, X). \]

The name "goal clause" stems from resolution-based theorem proving.
Syntax of Datalog
Clauses and Variable Scope

- Clauses represent **sentences**, i.e., formulas without free variables
- Hence, *the scope of each variable is the clause where it appears.*

**Example**

\[ C : \text{grandfatherOf}(X, Y) : \neg \text{fatherOf}(X, Z), \text{fatherOf}(Z, Y). \]
\[ D : \text{grandfatherOf}(X, Y) : \neg \text{motherOf}(X, Z), \text{fatherOf}(Z, Y). \]

\( X, Y, Z \) in \( C \) have nothing to do with \( X, Y, Z \) of clause \( D \).

- If \( S = \{C_1, \ldots C_n\} \) is a set of clauses, then \( S \) represents the conjunction of the first-order formulas corresponding to \( C_1, \ldots C_n \).
Syntax of Datalog

Extensional and Intentional Predicates

The set $\text{Pred}$ is composed of two disjoint sets: $\text{EPred}$ and $\text{IPred}$.

$$\text{EPred} \cup \text{IPred} = \text{Pred} \quad \text{and} \quad \text{EPred} \cap \text{IPred} = \emptyset$$

- $\text{EPred}$: extensional predicates, i.e. relations.
- $\text{IPred}$: intensional predicates, i.e. virtual relations defined by Datalog programs.

The Herbrand base $\text{HB}$ is the set of all positive literals that can be formed using the symbols in $\text{Pred}$ and $\text{Const}$.

We denote the extensional part of the Herbrand base by $\text{EHB}$ and the intensional part by $\text{IHB}$. 
Syntax of Datalog

Extensional Databases

An extensional database (EDB) is a finite subset of $EHB$ i.e. a finite set of positive ground facts.

Example

$$EDB = \{ \text{par}(\text{john}, \text{mary}), \text{par}(\text{john}, \text{reuben}), \text{par}(\text{terry}, \text{john}), \text{par}(\text{terry}, \text{lisa}), \text{lives}(\text{john}, \text{wash}), \text{lives}(\text{mary}, \text{ny}), \text{lives}(\text{terry}, \text{paris}), \text{lives}(\text{reuben}, \text{ny}), \text{lives}(\text{lisa}, \text{ny}) \}.$$  

where $\text{par}, \text{lives}$ are in $E Pred$

An extensional database represents the data stored in a relational database.
Syntax of Datalog

Datalog Programs

A Datalog program $P$ is a finite set of Horn clauses such that for all $C \in P$, either

- $C \in EHB$, i.e., $C$ is an extensional fact, or
- or $C$ is a rule which satisfies:
  a. The predicate occurring in the head of $C$ belongs to $IPred$.
  b. All variables which occur in the head of $C$ also occur in its body.

Condition b. is a safety condition: it ensures that only a finite number of facts can be deduced from a Datalog program.
Example Program

Example

Program $P_1$:

\[
\begin{align*}
\text{anc}(X, Y) & : - \text{par}(X, Y). \\
\text{anc}(X, Y) & : - \text{par}(X, Z), \text{anc}(Z, Y). \\
\text{anc}(X, \text{adam}) & : - \text{person}(X). \\
\text{person}(X) & : - \text{lives}(X, Y).
\end{align*}
\]

$IPred = \{\text{person, anc}\}$, $EPred = \{\text{par, lives}\}$.

A goal is an additional specification we may add to a Datalog program.

Example: ?- $\text{anc}(\text{terry}, X)$

Result: $\text{anc}(\text{terry, john}), \text{anc}(\text{terry, lisa}), \text{anc}(\text{terry, mary}),$

$\text{anc}(\text{terry, reuben}) \text{anc}(\text{terry, adam})$. 
Semantics of Datalog

General Intuition

What does a Datalog program compute?

A Datalog program is a query which acts on extensional databases and produces a result that is an intentional database.

- The result of program $P$ applied to EDB $E$ is the set of ground facts $\alpha$ such that
  - $\alpha$ is a logical consequence of the clause set $P \cup E$, and
  - the predicate symbol of $\alpha$ belongs to $IPred$.

- The meaning (or semantics) of a Datalog program $P$ can seen as a mapping from EDB’s to IDB’s.

- In case of a goal: result consists of ground facts for the goal predicate only.
Logical Consequence in Datalog

Let $F$ denote a ground fact and $S$ a set of Datalog clauses

\[ F \text{ is a consequence of } S \ (S \models F) \]

\[ \iff \]

each interpretation satisfying all clause in $S$ also satisfies $F$

\[ \iff \]

every model of $S$ is also a model of $F$.

Example

\begin{align*}
C1 &: \ p(a, b). \\
C2 &: \ p(X, Y) : \neg p(Y, X) \quad F &: \ p(b, a).
\end{align*}

For each possible interpretation of our constant and predicate symbols, whenever $C1$ and $C2$ are satisfied then $F$ is also satisfied.

Hence $\ C1, C2 \models F$
Herbrand Interpretations

As usual, to test logical consequence $S \models F$, one must check in principle infinitely many interpretations.

We can consider only a particular type of interpretations:

**Herbrand interpretation** $H$:

- the universe (domain) of $H$ is $\text{Const}$
- $c^H = c$ for every $c \in \text{Const}$ (each constant is interpreted as itself)
- for each $n$-ary predicate symbol $p$, its interpretation $p^H$ is a function $\text{Const}^n \rightarrow \{\text{true, false}\}$.

Thus, different Herbrand interpretations $H, H'$ differ only in the interpretations of predicate symbols.

**Theorem**

Let $F$ denote a ground fact and $S$ a set of Datalog clauses. Then $S \models F$ iff every Herbrand interpretation that is a model of $S$ is also a model of $F$. 
More on Herbrand Interpretations

- If we assign a truth value to each ground fact, then we have determined a particular Herbrand interpretation.

- This gives a 1-1 correspondence between Herbrand interpretations $H$ and subsets of $HB$:

  Each Herbrand interpretation corresponds to one subset of $HB$

- Hence we can view a Herbrand interpretation $H$ as a subset of the Herbrand base $HB$:

  $$H = \{p(t) \mid p \text{ has arity } n, t \in \text{Const}^n, p^H(t) = 1 (= \text{true})\}$$
Truth values of Datalog clauses

A ground substitution for \( R \) is a mapping \( \theta : \{ \text{Vars in } R \} \longrightarrow \text{Const} \). The application of \( \theta \) on \( R \) (or atom \( A \)), denoted by \( R\theta \) (\( A\theta \)), replaces each variable \( X \) by \( \theta(X) \).

Given a Herbrand interpretation \( I \subseteq HB \),

- A ground fact \( G \) is true under \( I \) if \( G \in I \)
  Otherwise, \( G \) is false under \( I \).

- Let \( R \) be a Datalog rule of the form
  \[
  L_0 : -L_1, \ldots, L_n
  \]

  \( R \) is true under \( I \), if for each ground substitution \( \theta \) for \( R \),
  \[
  L_1\theta \in I, \ldots, L_n\theta \in I \text{ implies } L_0\theta \in I
  \]

  Otherwise, \( R \) is false under \( I \).
Herbrand models

**Herbrand model**

A **Herbrand model** of a set of clauses $S$ is a Herbrand interpretation which satisfies all the clauses in $S$.

**Example**

$I_1 = \{\{\text{loves}(\text{bill}, \text{mary})\}, \{\text{loves}(\text{mary}, \text{bill})\}\}$

$I_2 = \{\{\text{loves}(\text{bill}, \text{mary})\}, \{\text{loves}(\text{mary}, \text{tom})\}, \{\text{loves}(\text{tom}, \text{mary})\}\}$

$R : \text{loves}(X, Y) : \neg \text{loves}(Y, X)$

$R$ is true in $I_1$.
Hence $I_1$ is a Herbrand model for $R$

Consider the ground substitution $\theta = \{X/\text{mary}, Y/\text{bill}\}$.
We then have: $\text{loves}(Y, X)\theta \in I_2$ but $\text{loves}(X, Y)\theta \notin I_2$.
Thus, $R$ is false in $I_2$ and $I_2$ is not a Herbrand model for $R$. 
Formal Semantics of a Datalog Program

Recall that $S \models F$ iff every Herbrand model for $S$ is also a Herbrand model for $F$.

$\mathcal{P}$ denotes powerset

Semantics of a Datalog program $P$

$\mathcal{M}_P : \mathcal{P}(EHB) \rightarrow \mathcal{P}(IHB)$

$\forall EDB \subseteq EHB : \mathcal{M}_P(EDB) = \{ G \mid G \in IHB, (P \cup EDB) \models G \}$.

When a goal “?- $H$” is specified, the output is limited to all instances of this goal which are consequences of $P \cup EDB$.

$\mathcal{M}_{P,H} : \mathcal{P}(EHB) \rightarrow \mathcal{P}(HB)$

$\forall EDB \subseteq EHB :$

$\mathcal{M}_{P,H}(EDB) = \{ C \mid C \in HB, (P \cup EDB) \models C, \exists \theta : H\theta = C \}$. 
Problems with Herbrand Interpretations

Check $S \models F$ by inspecting all Herbrand interpretations has three different problems of infinity:

1. We would have to check an infinite number of Herbrand interpretations.
2. Many of the Herbrand interpretations have an infinite number of elements.
3. If $R$ is a rule of $S$ containing variables, then we must consider an infinite number of ground substitutions for $R$.

How to cope with this?
The Least Herbrand Model

Let $S$ be a set of Datalog clauses (ground facts + rules), then define

\[ cons(S) = \{ F \mid F \in HB, \ S \models F \}. \]

Thus: \[ M_P(EDB) = cons(P \cup EDB) \cap IHB. \]

**Theorem**

- \( cons(S) = \bigcap\{ I \mid I \text{ is a Herbrand model of } S \} \)
- \( cons(S) \) is a Herbrand model of \( S \)

Note that, by definition, \( cons(S) \) is a subset of each Herbrand model of \( S \).

\( cons(S) \) is called the **least Herbrand model** \( S \).
Computation Issues – EPP

How can $\text{cons}(S)$ be computed?

- Datalog rules can be used to produce new facts from given facts.
- Consider a Datalog rule

$$R : \quad L_0 : -L_1, \ldots, L_n, \quad n \geq 1$$

and ground facts $F_1, \ldots, F_n$.

If there exists a ground substitution $\theta$ such that

$$\forall 1 \leq i \leq n, \quad L_i \theta = F_i$$

then we can infer in one step the fact $L_0 \theta$.

We call this (meta)rule **Elementary Production Principle (EPP)**
EPP - An example

Example (Same Generation Program)

\[ r_1 : sgc(X, X) : - person(X). \]
\[ r_2 : sgc(X, Y) : - par(X, X_1), sgc(X_1, Y_1), par(Y, Y_1). \]

- From rule \( r_1 \) and from the fact \( person(\text{george}) \) we can infer in one step \( sgc(\text{george}, \text{george}) \).
  The substitution used here was \( \theta = \{ X \leftarrow \text{george} \} \).

- Consider the facts:
  \( par(\text{dorothy}, \text{george}), sgc(\text{george}, \text{george}), \text{and} \)
  \( par(\text{evelyn}, \text{george}) \).
  We can infer in one step \( sgc(\text{dorothy}, \text{evelyn}) \).
  **Question:** with which substitution?
The Proof Theory of Datalog

Let $S$ be a set of Datalog clauses. Given a fact $F$, we say that $F$ is provable from $S$, denoted $S \vdash F$, if one of the following holds.

(I) $S \vdash F$ if $F \in S$.

(II) $S \vdash F$ if a rule $R \in S$ and ground facts $F_1, \ldots, F_n$ exist such that $\forall 1 \leq i \leq n, S \vdash F_i$, and $F$ can be inferred in one step applying EPP on $R$ and $F_1, \ldots, F_n$.

A recursive sequence according to (I) and (II) which is used to infer a ground fact $F$ from $S$ is called a proof of $F$ from $S$.

Every proof can be represented as a proof tree with $F$ at the top.
Example (continued)

Consider the database:

\[
PERS\text{ON} = \{ \langle ann \rangle, \langle bertrand \rangle, \langle charles \rangle, \langle dorothy \rangle, \\
\langle evelyn \rangle, \langle fred \rangle, \langle george \rangle, \langle hilary \rangle \}\]
\[
\text{PAR} = \{ \langle dorothy, george \rangle, \langle evelyn, george \rangle, \\
\langle bertrand, dorothy \rangle, \\
\langle ann, dorothy \rangle, \langle ann, hilary \rangle, \langle charles, evelyn \rangle \}. \\
\]

These relations express the set of ground facts:

\[
\text{EDB} = \{ \text{person}(ann), \text{person}(bertrand), \ldots, \text{person}(hilary) \\
\text{par}(dorothy, george), \ldots, \text{par}(charles, evelyn) \}. \\
\]

It is easy to see that \( \{ \text{PAR}, \ PERS\text{ON}, \ r_1, \ r_2 \} \vdash sgc(ann, charles) \).
Proof Theory vs Model Theory

**Theorem**

Let $S$ be a set of **Datalog clauses** and let $F$ be a ground fact. Then,

- (soundness) if $S \vdash F$, then $S \models F$.
- (completeness) if $S \models F$, then $S \vdash F$.

Soundness means that no false derivation is possible, and completeness that every logical consequence is provable.
Fixpoint characterization of Cons(S)

- The set $\text{cons}(S)$ can be characterized as the least fixpoint of a mapping $T_S$ from $\mathcal{P}(HB)$ to $\mathcal{P}(HB)$.

- For any set $S$ of Datalog clauses, let $EPP_1(S)$ denote the set of ground facts which can be inferred in one step from $S$ through $EPP$.

- Let $S = FACTS(S) \cup RULES(S)$, and define:

  $T_S : \mathcal{P}(HB) \longrightarrow \mathcal{P}(HB)$

  $\forall W \subseteq HB : T_S(W) = W \cup FACTS(S) \cup EPP_1(S \cup W)$;

  Furthermore:

  $\hat{T}_S : \mathcal{P}(HB) \longrightarrow \mathcal{P}(HB)$

  $\forall W \subseteq HB : \hat{T}_S(W) = FACTS(S) \cup EPP_1(S \cup W)$. 
1. Datalog
1.1 Syntax and Semantics

Fixpoint Theorems of Knaster-Tarski and Kleene

Complete lattice

Partially ordered set \((V, \leq)\) such that each subset \(W \subseteq V\) has a least upper bound \(\text{sup}(W)\) and a greatest lower bound \(\text{inf}(W)\).

Operators on \((V, \leq)\)

\(T : V \rightarrow V\) is

- **monotone** if \(\forall x, y \in V \; x \leq y \Rightarrow T(x) \leq T(y) \; \forall x, y \in V\)
- **continuous** if \(T(\text{sup}(C)) = \text{sup}\{T(x) \mid x \in C\}\) for each chain \(C\) in \(V\). (Chain: possibly infinite sequence \(x_0 < x_1 < \ldots\) in \(V\)

- Continuous operators are monotone
- Monotone and continuous operators have nice fixpoint properties
Fixpoint Theorems of Knaster-Tarski and Kleene

**Theorem (Knaster-Tarski)**

Any monotone operator $T$ on a complete lattice $(V, \leq)$ has a least fixpoint $\text{lfp}(T)$, and $\text{lfp}(T) = \inf \left( \{ x \in V \mid T(x) \leq x \} \right)$

A stronger theorem holds for continuous operators.

**Theorem (Kleene)**

Any continuous operator $T$ on a complete lattice $(V, \leq)$ has a least fixpoint, and $\text{lfp}(T) = \sup \left( \{ T^i \mid i \geq 0 \} \right)$, where $T^0 = \inf(V)$ and $T^{i+1} = T(T^i)$, for all $i \geq 0$.

- **Notation:** $T^\infty = \sup \left( \{ T^i \mid i \geq 0 \} \right)$.
- **Finite convergence** $T^k = T^{k-1}$ for some $k \Rightarrow T^\infty = T^k$

**Remark:** a weaker form of Kleene’s theorem holds for all monotone operators (transfinite sequence $T^i$).
Least Fixpoints and Datalog Programs

\( V_{DAT} = (\mathcal{P}(HB), \subseteq) \) is a complete lattice. The following is easily shown:

- \( T_S \) is a continuous operator on \( V_{DAT} \)
- Any Herbrand interpretation \( I \subseteq HB \) is a Herbrand model of \( S \) iff \( I \) is a fixpoint of the one step operator \( T_S \) (that is, if \( T_S(I) = I \)).

In particular, for the least Herbrand model

\[
\text{cons}(S) = \text{lfp}(T_S) = T_S^\infty
\]

Thus \( \text{cons}(S) \) can be computed by least fixpoint iteration:

\[
T_S(\emptyset) \subseteq T_S(T_S(\emptyset)) \subseteq T_S(T_S(T_S(\emptyset))) \subseteq \ldots
\]

If \( S \) is finite and safe, then \( T_S^\infty = T_S^k \) for some integer \( k \).
This applies to Datalog programs.
Evaluation Strategies for Datalog Programs

Strategies to evaluate Datalog programs are of two kinds:

**Bottom-up** aka **forward chaining**
Starting from the facts, add new facts until goal is reached.
The order in which $T^\infty$ is derived corresponds to the
bottom-up order of proof-trees

**Method:** Fixed-point iteration
Sometimes *algebraic evaluation*: view $P$ as an equational
system in Relational Algebra and compute the least solution.

**Top-down** aka **backward chaining**
Starts from the goal and looks for a proof of it.
Proof trees are constructed from the top to the bottom.
More appropriate when a goal is specified.

**Method:** Resolution.
Algebraic Naive Evaluation

**Algebraic view:** a rule \( L_0 : -L_1, \ldots, L_n \) assigns the value of an algebraic expression (body) to a relation (head).

### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression in relational algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(X, Y) : -r(X, Z), q(Z, Y) )</td>
<td>( \pi_{1,4}(r \bowtie_{[2=1]} q) )</td>
</tr>
</tbody>
</table>

The algebraic method evaluates the fixpoint solution of a system of algebraic equations. Thus, it evaluates **algebraically** the program solution.

\[
\Sigma : \quad R_i = E_i(R_1, \ldots, R_n) \quad (i = 1, \ldots, n).
\]

Initially, \( R_i^{(0)} = \emptyset \quad i = 1, \ldots, n \)

Then, the computation \( R_i^{(j+1)} = E_i(R_1^{(j)}, \ldots, R_n^{(j)}) \), \( i = 1, \ldots, n \) is iterated until none of the \( R_i \) changes.
Complexity of Datalog

We now discuss the complexity of some reasoning tasks in (fragments of) Datalog

First we consider the case where there are no variables, aka propositional Datalog

- Consistency (aka satisfiability) is trivial: every Datalog program has a model
- We consider the problem of deciding $P \models F$, that is, whether a program $P$ entails a fact $F$

**Theorem**

Let $P$ be a propositional (or ground) Datalog program and $F$ a ground atom. Deciding $P \models F$ is P-complete
Complexity of Propositional Datalog

**Theorem**

Let $P$ be a propositional (or ground) Datalog program and $F$ a ground atom. Deciding $P \models F$ is P-complete.

**Proof:**

**Membership** Computing $T_P^\infty$ is feasible in polynomial time, and then we only need to check whether $A \in T_P^\infty$.

**Hardness** Encoding of a deterministic Turing Machine (DTM).

Given a DTM $T$, an input string $I$ and a number of steps $N$ (where $N$ is polynomial in $|I|$), construct in logspace a program $P = P(T, I, N)$ and an atom $A$ such that $P \models A$ iff $T$ accepts input $I$ within $N$ steps.
Deterministic Turing Machine (DTM)

A Deterministic Turing Machine (DTM) is a quadruple

$$\langle \Sigma, S, \delta, s_0 \rangle$$

where

- $\Sigma$ is a finite (tape) alphabet of symbols, containing a special symbol \( \square \) called blank
- $S$ is a finite set of states,
- $\delta$ is a transition function, and
- $s_0 \in S$ is the initial state.
DTM (2)

- The DTM has a **tape**, which has a leftmost cell but no rightmost: it is infinite
- There is always exactly one symbol on each cell
- The DTM has a read/write **head** that moves along the tape reading and writing symbols on its tape

The transition function $\delta$ is a mapping

$$S \times \Sigma \rightarrow (S \cup \{\text{accept}, \text{reject}\} \times \Sigma \times \{-1, 0, +1\})$$

where

- **accept**, **reject** denote two special states not in $S$
- $-1, 0, +1$ denote the **directions** in which the head can move: left, none, right
1. Datalog

1.2 Complexity of Datalog

DTM - transition function

- At the beginning, the head is on the leftmost position and the machine is in state $s_0$

- There is an input string $I \in \Sigma^*$ written on the tape cells $c, \ldots, c_{|I|-1}$, all other cells contain ⊥

- The computation halts when the machine reaches the special states accept, reject
  - if it stops on the accept state, then the machine accepts the input $I$
  - if it stops on the reject state, then the machine rejects the input $I$

- We can assume that the machine is well-behaved and never moves further to the left than the leftmost tape cell
Encoding a DTM in Datalog

Given a DTM $T$ and an input $I$, we define a propositional program $P_{T,I,N}$ and a ground fact $accept$ s.t.

$$P_{T,I,N} \models accept$$

iff

$T$ accepts $I$ within $N$ steps

- We simulate the run of $T$ on $I$ but only for at most $N$ steps, where $N$ is polynomial in $|I|$
Encoding a DTM in Datalog

We use the following groups of ground atoms:

- $symbol_{\sigma}[\tau, \pi]$ for $0 \leq \tau \leq N$, $0 \leq \pi \leq N$ and $\sigma \in \Sigma$
  
  intuitive meaning: at instant $\tau$ of the computation, cell number $\pi$ contains symbol $\sigma$

- $head[\tau, \pi]$ for $0 \leq \tau \leq N$ and $0 \leq \pi \leq N$
  
  intuitive meaning: at instant $\tau$, the head is on cell $\pi$

- $state_s[\tau]$ for $0 \leq \tau \leq N$ and $s \in S$
  
  intuitive meaning: at instant $\tau$, $T$ is in state $s$

- accept

  intuitive meaning: $T$ has reached state accept
We denote by $I_k$ the $k$-th symbol of the input string $k$.

The initial configuration of $T$ on input $I$ is ensured by the following facts:

- $symbol_{I_{\pi}}[0, \pi]$. for each $0 \leq \pi \leq |I|$.
- $symbol_{\sqcup}[0, \pi]$. for each $|I| \leq \pi \leq N$.
- $head[0, 0]$.
- $states_0[0]$. 

Initial Configuration
Transitions

- For each transition $\delta(s, \sigma) = (s', \sigma', d)$ where $d \in \{-1, 0, +1\}$, we have transition rules

  $$
  \begin{align*}
  &symbol_{\sigma'}[\tau + 1, \pi]: -state_s[\tau], symbol_\sigma[\tau, \pi], head[\tau, \pi] \\
  &head[\tau + 1, \pi + d]: -state_s[\tau], symbol_\sigma[\tau, \pi], head[\tau, \pi] \\
  &state_{s'}[\tau + 1]: -state_s[\tau], symbol_\sigma[\tau, \pi], head[\tau, \pi]
  \end{align*}
  $$

- The following inertia rules ensure that the rest of the tape remains unchanged. For each $0 \leq \tau < N$ and $0 \leq \pi < \pi' \leq N$, we have:

  $$
  \begin{align*}
  &symbol_\sigma[\tau + 1, \pi]: -symbol_\sigma[\tau, \pi], head[\tau, \pi'] \\
  &symbol_\sigma[\tau + 1, \pi']: -symbol_\sigma[\tau, \pi'], head[\tau, \pi]
  \end{align*}
  $$

- Finally, we have accept rules for each $0 \leq \pi \leq N$:

  $$
  accept: -state_{accept}[\tau]
  $$
Correctness of the reduction

- Starting from the empty set, the one-step $T_{P(T,I,N)}$ operator computes the initial configuration.
- Then it will keep computing the next configuration until it reaches a fixed point.
- This will happen when the machine halts after at most $N + 2$ steps.

**Theorem**

$\text{accept} \in \text{cons}(P(T, I, N))$ iff $T$ accepts input $I$ in at most $N$ steps.

This shows P-hardness of ground Datalog.
Complexity of Datalog

Theorem

Let $P$ be a Datalog program and $F$ a ground atom. Deciding $P \models F$ is ExpTime-complete.

Proof:

Membership  We use the grounding $\text{ground}(P)$ of $P$

- Contains all ground instances of all rules, for all substitutions
- Its size is bounded by $|P| \times (|\text{Const}(P)|^{v_{max}})$, where $v_{max}$ is the maximal number of variables in one rule
- It can be easily computed in exponential time
- Therefore, $P \models F$ can be decided in exponential time

Hardness  We modify the DTM encoding
Encoding an Exponential Time DTM

We modify $P(T, I, N)$ into a non-ground program $P_{Dat}(T, I, N)$:

- It also simulates $N$ computation steps, but now $N = 2^m$
- Instead of ground atoms $symbol_\sigma[\tau, \pi]$, $head[\tau, \pi]$ and $state_s[\tau]$, we use non-ground atoms $symbol_\sigma(x, y)$, where $x$ and $y$ have arity $m$
- Time points $\tau$ and cell positions $\pi$ from 0 to $N - 1$ are encoded in binary, by tuples $\langle c_1, \ldots c_m \rangle$ with $c_i \in \{0, 1\}$
- We encode an order $\leq^m$ over the counter values $\langle c_1, \ldots c_m \rangle$, and the rules use a successor function $succ^m$ of arity $2m$
Defining $\leq^m$ and $\text{succ}^m$

- We add facts $\text{first}^1(0)$, $\text{last}^1(1)$ and $\text{succ}^1(0, 1)$
- We define $\text{succ}^m$ inductively on $m$:

\[
\begin{align*}
\text{succ}^{i+1}(z, \vec{x}, z, \vec{y}) & : -\text{succ}^i(\vec{x}, \vec{y}) \\
\text{succ}^{i+1}(z, \vec{x}, z', \vec{y}) & : -\text{succ}^1(z, z'), \text{last}^i(\vec{x}), \text{first}^i(\vec{y}) \\
\text{first}^{i+1}(z, \vec{x}) & : -\text{first}^1(z), \text{first}^i(\vec{x}) \\
\text{last}^{i+1}(z, \vec{x}) & : -\text{last}^1(z), \text{last}^i(\vec{x})
\end{align*}
\]

where $\vec{x}$ and $\vec{y}$ have arity $i$

- The order $\leq^m$ is then easy to define:

\[
\begin{align*}
\leq^m (\vec{x}, \vec{x}) \\
\leq^m (\vec{x}, \vec{y}) : -\text{succ}^m(\vec{x}, \vec{z}), \leq^m (\vec{z}, \vec{y})
\end{align*}
\]
Modifying the Program $P(T, I, N)$

- The facts $symbol_{\sigma}[0, \pi]$, where $0 \leq \pi \leq |I|$, are replaced by rules
  \[
  symbol_{\sigma}(\vec{x}, \vec{t}) : -\text{first}^m(\vec{x})
  \]
  where $\vec{t}$ represents the value of $\pi$.

- The facts $symbol_{\sqcup}[0, \pi]$, where $|I| \leq \pi \leq N$ are replaced by rules
  \[
  symbol_{\sqcup}(\vec{x}, \vec{y}) : -\text{first}^m(\vec{x}), \leq^m (\vec{t}, \vec{y})
  \]
  where $\vec{t}$ represents the value of $|I|$.

- The facts $head[0, 0]$ and $state_{s0}[0]$ are replaced by rules
  \[
  head(\vec{x}, \vec{x}) : -\text{first}^m(\vec{x}) \quad state_{s0}(\vec{x}) : -\text{first}^m(\vec{x})
  \]
Modifying the Program $P(T, I, N)$ (2)

For realizing $\tau + 1$ and $\pi + d$, transition and inertia rules use atoms $\text{succ}^m(\vec{x}, \vec{x}')$

Example

\[
\text{symbol}_{\sigma'}[\tau + 1, \pi] : -\text{state}_s[\tau], \text{symbol}_\sigma[\tau, \pi], \text{head}[\tau, \pi]
\]

becomes

\[
\text{symbol}_{\sigma'}(\vec{x}', \vec{y}) : -\text{state}_s(\vec{x}), \text{symbol}_\sigma(\vec{x}, \vec{y}), \text{head}(\vec{x}, \vec{y}), \text{succ}^m(\vec{x}, \vec{x}')
\]

Instead of the accept rules, we have:

\[
\text{accept} : -\text{state}_{\text{accept}}(\vec{x})
\]
Outline

1. Datalog
   1.1 Syntax and Semantics
   1.2 Complexity of Datalog

2. Datalog and DLs
   2.1 Adding Existential Quantification

3. Adding non-monotonic negation to Datalog
   3.1 Motivation
   3.2 Negation in Logic Programs

4. Answer Set Programming
   4.1 ASP Basics
   4.2 Disjunctive ASP
   4.3 Answer Set Solvers
   4.4 Beyond the core ASP: Extensions and frontends
Datalog and DLs have some common features:

- The distinction between extensional and intentional predicates in Datalog resembles the separation between TBox and ABox in DLs
- They have similar complexity (although stemming from different sources)

But there are also many differences:

- Datalog supports arbitrary arities, while in DLs we only have unary and binary predicates
- We will now see that there are also significant differences in their expressive power
Expressing Ontologies in Datalog

Many things expressible in DLs are also expressible in Datalog

For example, the TBox:

\[
\begin{align*}
\text{Mortal} &\sqcap \exists \text{hasChild} \sqsubseteq \text{mortalMother} \\
\text{Deity} &\sqsubseteq \forall \text{hasAncestor}. \text{Deity} \\
\text{hasParent} &\sqsubseteq \text{hasAncestor} \\
\text{hasParent} &\sqsubseteq \text{hasChild}^- \\
\text{trans(} \text{hasAncestor} \text{)}
\end{align*}
\]

can be expressed as the Datalog program:

\[
\begin{align*}
\text{mortalMother}(X) :& \neg \text{mortal}(X), \text{hasChild}(X, Y) \\
\text{deity}(Y) :& \neg \text{deity}(X), \text{hasAncestor}(X, Y) \\
\text{hasAncestor}(X, Y) :& \neg \text{hasParent}(X, Y) \\
\text{hasChild}(Y, X) :& \neg \text{hasParent}(X, Y) \\
\text{hasAncestor}(X, Z) :& \neg \text{hasAncestor}(X, Y), \text{hasAncestor}(Y, Z)
\end{align*}
\]
Expressing Ontologies in Datalog (2)

However, many DL axioms are not expressible in Datalog:

\[
\text{Mortal} \sqsubseteq \text{Male} \uplus \text{Female} \\
\text{Mortal} \sqsubseteq \leq 2 \text{hasParent}. \\
\text{funct(hasFather)} \\
\text{Hero} \sqsubseteq \exists \text{hasAncestor}.\text{Deity}
\]

- To simulate DLs with disjunction, one can employ extensions of Datalog with disjunction.

- Some forms of functionality/counting can be expressed using known techniques to axiomatize equality in Datalog.

- However, **existential quantification** is a crucial feature of DLs not supported in Datalog.
To express ontological knowledge, Datalog needs to be extended with existential quantification in the rule heads.

\[ \exists Y. \text{hasFather}(X, Y) : \neg \text{person}(X) \]

This extension of Datalog is known as Datalog$^\exists$

- The syntax is essentially as for Datalog rules, but the head is of the form
  \[ \exists \vec{y} P(\vec{y}, \vec{x}) \]
  where all of the variables in $\vec{x}$ and none of the variables in $\vec{y}$ occur in the body
Pros and Cons of Datalog

Datalog is a very expressive language
- it is well suited for knowledge representation and can express many DLs
- it generalizes relational database query languages and similar formalisms
- its expressive power goes beyond FOL
- extensions like constraints, disjunction, and non-monotonic negation (more later) make it yet more powerful

However, it has one major draw-back:

**Theorem**

Datalog is undecidable

**Intuition:** One can simulate a DTM (with no bounds on the time or space used for computation), e.g., using an infinite partial order

\[ int(1) \land \exists Y \ succ(X, Y) : \neg int(X) \land int(Y) : \neg succ(X, Y) \]
Decidable Fragments of Datalog

- Recent research aims at decidable fragments of Datalog
- These fragments should be well suited for representing ontologies
- The main way to regain decidability is to impose guardedness

Guarded Datalog, aka Datalog\(^\pm\)

All variables in the body of the rule occur together in one atom

\[ \text{manager}(X) : -\text{employee}(X), \text{supervisorOf}(X, Y), \text{manager}(Y) \]

The blue atom is called guard. The notion is adopted from the guarded fragment of FOL

[Andreka, van Benthem & Németi, 98]

- one of the most prominent decidable fragments of FOL
- it encompasses most modal, description and program logics
Variants of Guarded Datalog\textsuperscript{3}

Guarded Datalog\textsuperscript{3} can express Horn DLs like $\mathcal{ELHI}$

- it is \textbf{P-complete} in data complexity
- it is \textbf{ExpTime-complete} in combined complexity, assuming predicate arities are bounded
- The complexity increases exponentially for unbounded arities

Other refinements and generalizations have been proposed:

- \textbf{Linear Datalog}\textsuperscript{±} and \textbf{sticky Datalog}\textsuperscript{±}
  - generalizations of \textbf{DL-Lite}, FO-rewritable
  - the linear fragment is guarded, the sticky one is orthogonal

- Generalizations of guardedness, like \textbf{weakly guarded Datalog}\textsuperscript{3} and \textbf{frontier guarded Datalog}\textsuperscript{3} yield even more expressive languages

- In the DB setting, related languages like weakly-acyclic Datalog\textsuperscript{3} and extensions, Shy Datalog\textsuperscript{3}
Outline

1. Datalog
   1.1 Syntax and Semantics
   1.2 Complexity of Datalog

2. Datalog and DLs
   2.1 Adding Existential Quantification

3. Adding non-monotonic negation to Datalog
   3.1 Motivation
   3.2 Negation in Logic Programs

4. Answer Set Programming
   4.1 ASP Basics
   4.2 Disjunctive ASP
   4.3 Answer Set Solvers
   4.4 Beyond the core ASP: Extensions and frontends
Monotonicity

- Datalog overcomes a significant limitation of FOL (and hence of DLs): the lack of recursion
- However, both DLs and Datalog (and all the extensions we have discussed) still have a major limitation in the context of KR&R: their monotonicity

Monotonic Logics

A logic (or more specifically, and inference relation) is monotonic if for every pair of theories $T$, $T'$ and formula $\varphi$,

$$T \vdash \varphi \quad \text{implies} \quad T \cup T' \vdash \varphi$$

that is, if something follows from a given set of knowledge, then it will still follow no matter what new knowledge we add.
Non-monotonic reasoning

- Monotonicity is, in general, a natural property of inference relations
- but in real life, we often draw non-monotonic inferences.

The most classical example: Tweety

- We know that Tweety is a bird
- hence it makes sense to assume that Tweety flies!
- Later we find out that Tweety is a penguin
- We can simply withdraw our previous conclusion and infer that Tweety does not fly
Non-monotonic reasoning (cont’d)

We constantly use many forms of default reasoning that are not monotonic:

- My car is currently parked where I left it
- When I open the tap I will get running water
- On my way home I will be able to take the subway, and it comes every few minutes
- ...

Capturing this kind of reasoning through formal logics was a challenge that kept the AI community busy for quite a while.

Datalog extended with non-monotonic negation is one of the simplest and most powerful ways to achieve this goal.
Adding Negation to Datalog

Clauses of the form:

\[ p(\vec{X}) \leftarrow q_1(\vec{X}_1), \ldots, q_k(\vec{X}_k), \text{not } r_1(\vec{Y}_1), \ldots, \text{not } r_l(\vec{Y}_l) \]

- "not (·)" means "Negation as Failure" or "default negation"
- not a holds unless a can be proved true
- Different from negation in classical logic!

Example:

\[ P = \text{canFly}(X) : \neg \text{bird}(X), \text{not flightless}(X). \]
\[ \text{flightless}(X) : \neg \text{penguin}(X). \]

From \( P \) and \( \text{bird(tweety)} \) we infer \( \text{canFly(tweety)} \), but from \( \text{bird(tweety)}, \text{penguin(tweety)} \) we can not draw the same conclusion.
The frame Problem

Another major limitation of monotonic logic arises when we reason about actions and change

- In FOL/DLs we can define, for example
  - Different kind of actions that an agent may carry out in some domain, e.g., load a gun, aim a target, shoot ...
  - Infinitely many time instances over which actions can take place
  - Effects of carrying out some action, e.g., if the agent loads the gun at instant $t$, then the gun is loaded in the next instant $t + 1$
  - ...
The frame Problem (cont’d)

- However, we also want our model to imply that if at instant $t$ the gun is loaded, and the agent aims it at a target, then at instant $t + 1$
  - the gun is still loaded
  - the agent $A$ that has the gun is still the same one
  - agent $A'$ still knows what the position of the door is, etc.
  - ...

but we do not want to explicitly say what happens to every fluent of the domain after every action!

In AI this is called the frame problem
Default Negation and the Frame Problem

- There is no natural way to solve this problem in FOL/DLs/plain Datalog.
- but default negation allows us to state compact *inertia rules* for the fluents in our domain:

**Example**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gunLoaded(T+1)</code></td>
<td><code>:-</code> gunLoaded(T), not gunUnloaded(T+1).</td>
</tr>
<tr>
<td><code>doorOpen(T+1)</code></td>
<td><code>:-</code> doorOpen(T), not doorNotOpen(T+1).</td>
</tr>
</tbody>
</table>

- This solves the frame problem!
Recall that the **single minimal model** captures the intended semantics of Datalog programs.

What happens when we add default negation?

**Example:**

```
adult("dilbert").
single(X) :- not married(X), adult(X).
marrried(X) :- not single(X), adult(X).
```

**Result ????**

**Problem:** not single minimal model!

Two alternatives:

- \( M_1 = \{ \text{adult("dilbert")}, \text{single("dilbert")} \} \),
- \( M_2 = \{ \text{adult("dilbert")}, \text{married("dilbert")} \} \).

Which one to choose?
Semantics of Logic Programs with Negation

Great Logic Programming Schism

**Single Intended Model Approach:**
- Select a single model of all classical models
- Agreement for so-called “stratified programs”: Perfect model

**Multiple Preferred Model Approach:**
- Select a subset of all classical models
- Different selection principles for non-stratified programs
Stratified Negation

**Intuition**: For evaluating the body of a rule containing $not \, r(t)$, the value of the “negative” predicates $r(t)$ should be known.

1. Evaluate first $r(t)$
2. if $r(t)$ is false, then $not \, r(t)$ is true,
3. if $r(t)$ is true, then $not \, r(t)$ is false and rule is not applicable.

**Example:**

```prolog
adult("dilbert").
single(X) :- not married(X), adult(X).
```

Computed model

$M = \{ adult("dilbert"), single("dilbert") \}$. 
Program Layers

- Evaluate predicates bottom up in layers
- Methods works if negation stratified, that is, layered

Example:

L0: adult("dilbert").
L0: adult("fredFlintstone"). hasWife("fredFlintstone","wilmaFlintstone").
L0: married(X) :- adult(X), hasWife(X,Y).
L1: single(X) :- not married(X), adult(X).

Unique model resulting by layered evaluation ("perfect model"):

Result

\[ M = \{ \text{adult("dilbert")}, \text{adult("fredFlintstone")}, \text{hasWife("fredFlintstone","wilmaFlintstone")}, \text{single("dilbert")}, \text{married("fredFlintstone")} \} \]
Multiple preferred models

Unstratified Negation makes this layered approach unusable:

L0: adult("dilbert").
L0: adult("fredFlintstone"). hasWife("fredFlintstone","wilmaFlintstone").
L?: married(X) :- not single(X), adult(X).
L?: single(X) :- not married(X), adult(X).

What can we do?

- Assign to a program (theory) not one but several intended models (hence the term answer sets)
- How to interpret these semantics? Answer set programming caters for the following views:
  1. skeptical reasoning: Only take entailed answers, i.e. true in all models
  2. brave reasoning: each model represents a different solution to the problem
  3. additionally: one can define to consider only a subset of preferred models

- Alternative: well-founded inference takes a more “agnostic” view: One model, leaving ambiguous literals unknown.
4. Answer Set Programming

4.1 ASP Basics

**Answer Set Programming Paradigm**

**General idea: Models are Solutions!**

Reduce solving a problem instance $I$ to computing models of a Datalog program with default negation

1. **Encode** $I$ as a (non-monotonic) logic program $P$, such that solutions of $I$ are represented by models of $P$
2. **Compute** some model $M$ of $P$, using an ASP solver
3. **Extract** a solution for $I$ from $M$.

Variant: Compute multiple/all models (for multiple / all solutions)
## Applications of ASP

ASP facilitates **declarative problem solving**

Problems in different domains (some with substantial amount of data), see


- information integration
- constraint satisfaction
- planning, routing
- semantic web
- diagnosis
- security analysis
- configuration
- computer-aided verification
- ...

**ASP Showcase:** [http://www.kr.tuwien.ac.at/projects/WASP/showcase.html](http://www.kr.tuwien.ac.at/projects/WASP/showcase.html)
Example: 3-Coloring

- Problem specification $PS$: 3-Colorability condition
  
  $$P_{PS} = \{ \text{col}(X, r) \lor \text{col}(X, g) \lor \text{col}(X, b) : - \text{node}(X).$$
  
  $$: - \text{node}(X), \text{col}(X, C), \text{col}(X, D), C \neq D.$$  
  
  $$: - \text{edge}(X, Y), \text{col}(X, C), \text{col}(Y, C). \}$$

- Data $D$: Graph $G = (V, E)$
  
  $$P_D = \{ \text{node}(v) \mid v \in V \} \cup \{ \text{edge}(u, v) \mid (u, v) \in E \}.$$  

- Correspondence 3-Colorings $\iff$ models:
  
  $v$ is colored with $c \in \{r, g, b\}$ iff $\text{col}(v, c)$ is in the model.
Example: Sudoku

Fill in the grid so that every row, every column, and every 3x3 box contains the digits 1 through 9
Example: Sudoku

**Problem specification** $PS$

$tab(i, j, n)$: cell $(i, j)$, $i, j \in \{0, ..., 8\}$ has digit $n$

From *sudoku.dlv*:

```
% Assign a value to each field
tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9) :-
    #int(X), 0 <= X, X <= 8, #int(Y), 0 <= Y, Y <= 8.

% Check rows and columns
:- tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
:- tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
:- tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).
:- tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1), div(Y1,3,W2), div(Y2,3,W2).

% Auxiliary: X divided by Y is Z
div(X,Y,Z) :- XminusDelta = Y*Z, X = XminusDelta + Delta, Delta < Y.
```
Sudoku (cont’d)

**Data $D$:**

% Table positions X=0..8, Y=0..8  
tab(0,1,6). tab(0,3,1). tab(0,5,4). tab(0,7,5).  
tab(1,2,8). tab(1,3,3). tab(1,5,5). tab(1,6,6).  
...

**Solution:**

![Sudoku Solution](image-url)
Social Dinner Example

- Imagine the organizers of a conference plan a fancy dinner for the participants.
- In order to make the attendees happy with this event and to make them familiar with ontologies, the organizers decide to ask them to declare their preferences about wines, in terms of a class description reusing the (in)famous Wine Ontology.
- The organizers realize that only one kind of wine would not achieve the goal of fulfilling all the attendees’ preferences.
- Thus, they aim at automatically finding the cheapest selection of bottles such that any attendee can have her preferred wine at the dinner.

The organizers quickly realize that several building blocks are needed to accomplish this task.
Wanted!

A general-purpose approach for modeling and solving these and many other problems

- Diverse domains
- Spatial and temporal reasoning
- Constraints
- Incomplete information
- Preferences and priority

Proposal:

Answer Set Programming (ASP) paradigm!

We will introduce ASP via examples, often using the syntax of DLV programs
Simple Social Dinner Example

A simple program simple.dlv:

- Wine bottles (brands) "a", ..., "e"
- preference by facts
- a simple (DL) ontology could be directly represented within the logic program (or even better, the program could interact with a DL ontology)

% A suite of wine bottles and their kinds
wineBottle("a"). isA("a","whiteWine"). isA("a","sweetWine").
wineBottle("b"). isA("b","whiteWine"). isA("b","dryWine").
wineBottle("c"). isA("c","whiteWine"). isA("c","dryWine").
wineBottle("d"). isA("d","redWine"). isA("d","dryWine").
wineBottle("e"). isA("e","redWine"). isA("e","sweetWine").

% Persons and their preferences
person("axel"). preferredWine("axel","whiteWine").
person("gibbi"). preferredWine("gibbi","redWine").
person("roman"). preferredWine("roman","dryWine").

% Available bottles a person likes
compliantBottle(X,Z) :- preferredWine(X,Y), isA(Z,Y).
Social Dinner Example II

Extend the Simple Social Dinner Example (simple.dlv) to simpleGuess.dlv:

% These rules generate multiple answer sets:
(1) bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
(2) bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).

- Rules (1) and (2) enforce that either bottleChosen(X) or bottleSkipped(X) is included in an answer set (but not both), if it contains compliantBottle(Y,X).
- Rule (3) computes which persons have a bottle.
Social Dinner Example II

Extend the Simple Social Dinner Example (simple.dlv) to simpleGuess.dlv:

% Alternatively we could use disjunction:

(4) bottleSkipped(X) v bottleChosen(X) :- compliantBottle(Y,X).

(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).

- Rules (1) and (2) enforce that either bottleChosen(X) or bottleSkipped(X) is included in an answer set (but not both), if it contains compliantBottle(Y,X).
- Rule (3) computes which persons have a bottle
- Rule (4) (disjunction!) can be used for replacing (1)-(2), more on that later!
Answer Set Semantics

- Variable-free, non-disjunctive programs first!

- Rules

\[ a :\neg b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n \]

where all \( a, b_i, c_j \) are atoms

- a normal logic program \( P \) is a (finite) set of such rules

- \( HB(P) \) is the set of all atoms with predicates and constants from \( P \).
Example

compliantBottle("axel","a"). wineBottle("a").
bottleSkipped("a") :- not bottleChosen("a"), compliantBottle("axel","a").
bottleChosen("a") :- not bottleSkipped("a"), compliantBottle("axel","a").
hasBottleChosen("axel") :- bottleChosen("a"), compliantBottle("axel","a").

\[ HB(P) = \{ \text{wineBottle("a"), wineBottle("axel"), bottleSkipped("a"), bottleSkipped("axel"), bottleChosen("a"
}
 我们,
Answer Sets /2

Let

- $P$ be a normal logic program
- $M \subseteq HB(P)$ be a set of atoms

**Gelfond-Lifschitz (GL) Reduct $P^M$**

The reduct $P^M$ is obtained as follows:

1. remove from $P$ each rule

   $$a :\ b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$$

   where some $c_i$ is in $M$

2. remove all literals of form $\text{not } p$ from all remaining rules
The reduct $P^M$ is a Horn program
- It has the least model $lm(P^M)$

**Definition**

$M \subseteq HB(P)$ is an answer set of $P$ if and only if $M = lm(P^M)$

**Intuition:**
- $M$ makes an **assumption** about what is true and what is false
- $P^M$ derives positive facts under the assumption of $not(\cdot)$ as by $M$
- If the result is $M$, then the assumption of $M$ is “stable”
Computation of $lm(P)$

Recall that the least model of a $not$-free program can be computed by fixpoint iteration.

**Algorithm Compute_LM($P$)**

**Input:** Horn program $P$;  
**Output:** $lm(P)$

```plaintext
new_M := ∅;
repeat
    M := new_M;
    new_M := \{ a | a :- b_1, ..., b_m \in P, \{b_1, ..., b_m\} \subseteq M \}
until new_M == M
return M
```
Examples

compliantBottle("axel","a"). wineBottle("a").
hasBottleChosen("axel") :- bottleChosen("a"),
    compliantBottle("axel","a").

- $P$ has no $not$ (i.e., is Horn)
- thus, $P^M = P$ for every $M$
- the single answer set of $P$ is $M = lm(P) =$
  
  \{ wineBottle("a"), compliantBottle("axel","a") \}.  

Examples II

(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
    compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
    compliantBottle("axel","a").
(4) hasBottleChosen("axel") :- bottleChosen("a"),
    compliantBottle("axel","a").

Take $M = \{ \text{wineBottle("a")}, \text{compliantBottle("axel","a")}, \text{bottleSkipped("a")} \}$

- Rule (2) “survives” the reduction (cancel not bottleChosen("a"))
- Rule (3) is dropped

$lm(P^M) = M$, and thus $M$ is an answer set
Examples III

(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
    compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
    compliantBottle("axel","a").
(4) hasBottleChosen("axel") :- bottleChosen("a"),
    compliantBottle("axel","a").

Take $M = \{ \text{wineBottle("a"), compliantBottle("axel","a"),}
\text{bottleChosen("a"), hasBottleChosen("axel")} \}$

- Rule (2) is dropped
- Rule (3) “survives” the reduction (cancel not bottleSkipped("a"))

$Im(P^M) = M$, and therefore $M$ is another answer set.
Examples IV

(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") :- not bottleChosen("a"),
    compliantBottle("axel","a").
(3) bottleChosen("a") :- not bottleSkipped("a"),
    compliantBottle("axel","a").
(4) hasBottleChosen("axel") :- bottleChosen("a"),
    compliantBottle("axel","a").

Take $M = \{ \text{wineBottle}("a"), \text{compliantBottle}("axel","a"),$
\text{bottleChosen}("a"), \text{bottleSkipped}("axel"), \text{hasBottleChosen}("axel") \}$

- Rules (2) and (3) are dropped

$lm(P^M) = \{ \text{wineBottle}("a"), \text{compliantBottle}("axel","a")) \neq M$

Thus, $M$ is not an answer set
Programs with Variables

- Like in Datalog, no function symbols
- We consider Herbrand models only,
- Each clause is a shorthand for all its ground substitutions, i.e., replacements of variables with constants

E.g., \( b(X) :- \neg s(X), c(Y,X) \).

is with constants "axel","a" short for:

\begin{align*}
  b("a") & :- \neg s("a"), c("a","a"). \\
  b("a") & :- \neg s("a"), c("axel","a"). \\
  b("axel") & :- \neg s("axel"), c("axel","axel"). \\
  b("axel") & :- \neg s("axel"), c("axel","a").
\end{align*}
Programs with Variables /2

- The Herbrand base of $P$, $HB(P)$, consists of all ground (variable-free) atoms with predicates and constant symbols from $P$.

- The grounding of a rule $r$, $Ground(r)$, consists of all rules obtained from $r$ if each variable in $r$ is replaced by some ground term (over $P$, unless specified otherwise).

- The grounding of program $P$, is $\cup_{r \in P} Ground(r)$.

Definition

$M \subseteq HB(P)$ is an answer set of $P$ if and only if $M$ is an answer set of $Ground(P)$.

In the (function free) ASP setting, stable models are always finite. In this sense, ASP does not have the expressiveness of some DLs.
Inconsistent Programs

Program

\[ p :\neg \ p. \]

- This program has NO answer sets
- Let \( P \) be a program and \( p \) be a new atom
- Adding
  \[ p :\neg \ p. \]
  to \( P \) “kills” all answer sets of \( P \)
Constraints

- Adding

\[ p :- q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n, \text{not } p. \]

Adding to \( P \) “kills” all answer sets of \( P \) that:

- contain \( q_1, \ldots, q_m \), and
- do not contain \( r_1, \ldots, r_n \)

- Abbreviation:

\[ :- q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n. \]

This is called a “constraint” (cf. integrity constraints in databases)
Social Dinner Example II

Task:

Add a constraint to simpleGuess.dlv in order to filter answer sets in which for some person no bottle is chosen. We obtain simpleConstraint.dlv

% This rule generates multiple answer sets:
(1) bottleSkipped(X) :- not bottleChosen(X),
    compliantBottle(Y,X).

(2) bottleChosen(X) :- not bottleSkipped(X),
    compliantBottle(Y,X).

% Ensure that each person gets a bottle.
(3) hasBottleChosen(X) :- bottleChosen(Z),
    compliantBottle(X,Z).

(4) :- person(X), not hasBottleChosen(X).
Main Reasoning Tasks

**Consistency**

Decide whether a given program $P$ has an answer set.

**Cautious (resp. Brave) Reasoning**

Given a program $P$ and ground literals $l_1, \ldots, l_n$, decide whether $l_1, \ldots, l_n$ simultaneously hold in every (resp., some) answer set of $P$.

**Query Answering**

Given a program $P$ and non-ground literals $l_1, \ldots, l_n$ on variables $X_1, \ldots, X_k$, list all assignments of values $\nu$ to $X_1, \ldots, X_k$ such that $l_1\nu, \ldots, l_n\nu$ is cautiously resp. bravely true.

- seamless integration of query language and rule language
- expressivity beyond traditional query languages, e.g. SQL

**Answer Set Computation**

Compute some / all answer sets of a given program $P$. 
Simple Social Dinner Example – Reasoning

- For our simple Social Dinner Example (simple.dlv), we have a single answer set.
- Therefore, cautious and brave reasoning coincide.
- compliantBottle("axel","a") is both a cautious and a brave consequence of the program.
- For the query person(X), we obtain the answers "axel", "gibbi", "roman".
Social Dinner Example II – Reasoning

For `simpleConstraint.dlv`:

- The program has 20 answer sets.
- They correspond to the possibilities for all bottles being chosen or skipped.
- The cautious query `bottleChosen("a")` fails.
- The brave query `bottleChosen("a")` succeeds.
- For the nonground query `bottleChosen(X)`, we obtain under cautious reasoning an empty answer.
Disjunctive ASP

- The use of disjunction in rule heads is natural

\[
\text{man}(X) \lor \text{woman}(X) : - \text{person}(X)
\]

- ASP has thus been extended with disjunction

\[
a_1 \lor a_2 \lor \cdots \lor a_k : - b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n
\]

- The interpretation of disjunction is “minimal” (in LP spirit)

- Disjunctive rules thus permit to encode choices
Social Dinner Example II – Disjunctive Version

Task:

Replace the choice rules in simpleConstraint.dlv to obtain simpleDisj.dlv.

\[
\begin{align*}
\text{bottleSkipped}(X) & :\; not \; \text{bottleChosen}(X), \; \text{compliantBottle}(Y,X). \\
\text{bottleChosen}(X) & :\; not \; \text{bottleSkipped}(X), \; \text{compliantBottle}(Y,X).
\end{align*}
\]

with an equivalent disjunctive rule

\[
\text{bottleSkipped}(X) \lor \text{bottleChosen}(X) :\; \text{compliantBottle}(Y,X).
\]

This form is more natural and intuitive!

- Very often, disjunction corresponds to such cyclic negation
- However, disjunction is more expressive in general, and cannot be efficiently eliminated
## Answer Sets of Disjunctive Programs

Define answer sets similar as for normal logic programs

### Gelfond-Lifschitz Reduct $P^M$

Extend $P^M$ to disjunctive programs:

1. remove each rule in $\text{Ground}(P)$ with some literal $\text{not } a$ in the body such that $a \in M$
2. remove all literals $\text{not } a$ from all remaining rules in $\text{Ground}(P)$

However, $\text{lm}(P^M)$ does not necessarily exist (multiple minimal models!)

### Definition

$M \subseteq HB(P)$ is an answer set of $P$ if and only if $M$ is a minimal (wrt. $\subseteq$) model of $P^M$
Example

(1) compliantBottle("axel","a"). wineBottle("a").
(2) bottleSkipped("a") v bottleChosen("a") :-
    compliantBottle("axel","a").
(3) hasBottleChosen("axel") :- bottleChosen("a"),
    compliantBottle("axel","a").

This program contains no \textit{not}, so $P^M = P$ for every $M$

Its answer sets are its minimal models:

\begin{itemize}
    \item $M_1 = \{ \text{wineBottle("a"), compliantBottle("axel","a"),}
                      \text{bottleSkipped("a")} \} $
    \item $M_2 = \{ \text{wineBottle("a"), compliantBottle("axel","a"),}
                      \text{bottleChosen("a"), hasBottleChosen("axel")} \} $
\end{itemize}

This is the same as in the non-disjunctive version!
Properties of Answer Sets

**Minimality:**
Each answer set $M$ of $P$ is a minimal Herbrand model (wrt $\subseteq$).

**Generalization of Stratified Semantics:**
If negation in $P$ is layered ("$P$ is stratified"), then $P$ has a unique answer set, which coincides with the perfect model.

**NP-Completeness:**
Deciding whether a normal propositional program $P$ has an answer set is NP-complete in general.
⇒ Answer Set Semantics is an expressive formalism;
Higher expressiveness through further language constructs (disjunction, weak/weight constraints)
## Answer Set Solvers on the Web

<table>
<thead>
<tr>
<th>Solver</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLV</td>
<td><a href="http://www.dbai.tuwien.ac.at/proj/dlv/">http://www.dbai.tuwien.ac.at/proj/dlv/</a></td>
</tr>
<tr>
<td>SModels</td>
<td><a href="http://www.tcs.hut.fi/Software/smodels/">http://www.tcs.hut.fi/Software/smodels/</a></td>
</tr>
<tr>
<td>GnT</td>
<td><a href="http://www.tcs.hut.fi/Software/gnt/">http://www.tcs.hut.fi/Software/gnt/</a></td>
</tr>
<tr>
<td>Cmodels</td>
<td><a href="http://www.cs.utexas.edu/users/tag/cmodels/">http://www.cs.utexas.edu/users/tag/cmodels/</a></td>
</tr>
<tr>
<td>ASSAT</td>
<td><a href="http://assat.cs.ust.hk/">http://assat.cs.ust.hk/</a></td>
</tr>
<tr>
<td>NoMore</td>
<td><a href="http://www.cs.uni-potsdam.de/~linke/nomore/">http://www.cs.uni-potsdam.de/~linke/nomore/</a></td>
</tr>
<tr>
<td>Platypus</td>
<td><a href="http://www.cs.uni-potsdam.de/platypus/">http://www.cs.uni-potsdam.de/platypus/</a></td>
</tr>
<tr>
<td>clasp</td>
<td><a href="http://www.cs.uni-potsdam.de/clasp/">http://www.cs.uni-potsdam.de/clasp/</a></td>
</tr>
<tr>
<td>XASP</td>
<td>distributed with XSB v2.6</td>
</tr>
<tr>
<td>aspps</td>
<td><a href="http://www.cs.engr.uky.edu/ai/aspps/">http://www.cs.engr.uky.edu/ai/aspps/</a></td>
</tr>
<tr>
<td>ccalc</td>
<td><a href="http://www.cs.utexas.edu/users/tag/cc/">http://www.cs.utexas.edu/users/tag/cc/</a></td>
</tr>
</tbody>
</table>
Architecture of ASP Solvers

Typically, a two level architecture

1. Grounding Step
Given a program $P$ with variables, generate its grounding (or a subset of its grounding which has the same models)

2. Model search
Find a model for a given ground program.

Some solvers use special-purpose search procedures, others use variations of techniques known from SAT and theorem provers, or even export the task to sat solvers (e.g. Cmodels, ASSAT)
ASP and infinite domains

- Like in Datalog, Herbrand interpretations are finite
  - no function symbols
  - no existentially quantified variables

- The discussed extensions of Datalog with existentially quantified variables in are not known to be decidable with default negation

- There are some decidable extensions of ASP with function symbols: \text{FDNC}, \text{BD}, finitary, and finitely recursive programs
  - They support default negation, in some cases also disjunction and constraints
  - They can generate infinite structures and simulate expressive DLs
  - In presence of default negation, the semantics is slightly different to their counterparts with existential quantifiers
  - This plays a role in their decidability
Other extensions of ASP

- There are many more extensions than the ones discussed here
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:
  
  See http://www.tcs.hut.fi/Research/Logic/wasp/wp3/

- For example, the extensions supported by DLV include:
  
  - weak constraints
  - aggregates

Other solvers feature similar constructs.
Weak Constraints in DLV

- Allow the formalization of optimization problems in an easy and natural way.

- Weak constraints express desiderata which should be satisfied, if possible.

- The answer sets of a program $P$ with a set $W$ of weak constraints are those answer sets of $P$ which minimize the number of violated constraints.

- Such answer sets are called \textit{optimal or best models of} $(P, W)$.

- In our \textit{social dinner} example, we could use weak constraints to select the least possible number of wines, or the least possible number of expensive wines.
Aggregates in DLV

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)

- In our example, we can use this to calculate and minimize the price of the chosen wines
  For example,

  \[
  \text{totalcost}(N) :- \#\text{int}(N),
  \#\sum \{ Y : \text{bottleChosen}(X), \text{prize}(X, Y) \} = N.
  \]

  estimates the cost of the chosen wines, and

  \[
  \text{ok\_price} :- \#\sum \{ Y : \text{bottleChosen}(X), \text{prize}(X, Y) \} < 500.
  \]

  says that the price is OK if the sum of the prices of the chosen wines is below 50
DLV Front Ends

DLV offers a range of *front-ends* for specific applications, like

- diagnosis
- planning
- inheritance reasoning
- SQL3