Knowledge Representation and Semantic Technologies
Datalog and Answer Set Programming

Claudia Medaglia

Chapter 1

Positive Datalog

1.1 Syntax

Alphabets We start from the following alphabets (sets of symbols):

- Alphabet of predicate symbols **Pred**→ every predicate symbol is associated with an arity (non-negative integer) (e.g. p/2, r/1, s/0,...)
- Alphabet of constant symbols **Const**→Syntactic convention: we will use symbols starting with lowercase letters (e.g. a, b, c,...)
- Alphabet of variable symbols **Var**→Syntactic convention: we will use symbols starting with uppercase letters (e.g. X, Y, Z,...)

The three alphabets are pairwise disjoint (every symbol belongs at most to one alphabet).
We also define the (derived) set of terms **Term** = Const U Var

Atoms
atom = expression of the form $p(t_1,...,t_n)$
where:
• n is a non-negative integer (we can have zero atom in the body)
• p is a predicate symbol of arity n
• every t_i is a term
(An atom is constituted by a pradicate followed by a sequence of constants or variables and the number of the arguments is the arity of the predicate)

Positive Rules
Positive rule = expression of the form α :- β_1,..., β_n .
where:
• :- is the "if" symbol. The body is the condition, the head is the implication (I want all the atoms in the body to be true, in order to derive that also α is true).
• n is a non-negative integer
• α is an atom
• every β_i
is an atom
• (safeness condition) every variable symbol occurring in α must appear in β_1,..., β_n

α is called the rule **head**
β_1,..., β_n is called the rule **body**

Pred = $\{p/1, q/0, r/2, s/2, t/1, u/3\}$
Const = $\{a, b, c\}$
Var = $\{X, Y, Z, W\}$

Correct positive rules:

- $p(X) :- r(X, Y), s(Y, Z).$
- $q :- r(X, Y), s(Y, Z).$
- $r(X, Z) :- r(X, Y), s(Y, Z).$
- $u(X, Y, Z) :- r(X, Y), s(Y, Z).$
- $u(X, a, b) :- r(X, Y).$
- $u(c, a, b) :- .$

Pred = $\{p/1, q/0, r/2, s/2, t/1, u/3\}$
Const = $\{a, b, c\}$
Var = $\{X, Y, Z, W\}$

Incorrect positive rules:

- $p(W) :- r(X, Y), s(Y, Z).$
  - variable W is not safe
- $q :- r(X, Y), s(Y, Z)$
  - missing "." at end of rule
- $t(X, Y, r) :- r(X, Y), s(Y, Z).$
  - uses a predicate in argument position
- $Y(X, a, b) :- r(X, Y).$
  - uses a variable in predicate position

Facts
A fact is a positve rule with an empty body E.g.:
• u(c, a, b) :– .
• p(a) :– .
• q :– .
When representing facts we can omit the ”if” symbol: u(c, a, b).
Notice that variable symbols cannot appear in a fact (due to the safeness condition).

Positive Datalog program
It is a set of Positive datalog rules (and facts).

EDB and IDB predicates
Predicates in a Datalog program are actually partitioned into EDB (Extensional Database) predicates and IDB (Intensional Database) predicates:
• EDB predicates can only be used in facts and in the body of rules
• IDB predicates cannot be used in facts

E.g.: EDB= $\{p/2, q/1\}$, IDB= $\{r/2, s/2\}$

$p(a, b).$
$p(b, c).$
$q(b).$
$r(X, Y) :- p(X, Y).$
$r(X, Z) :- p(X, Y), r(Y, Z).$
$s(X, Z) :- r(X, Y), q(Y).$

(In the 5th rule we have an example of **recursive** rule: same predicate appears both in the head and in the body)
(Later we will be able to forget about these rules because, even if they are not satisfied, we will be able to re-write the program into one that satisfies these condition).

Ground Rules and Programs
A Datalog rule is **ground** if no variable occurs in it Examples of ground rules:
r(a, b) :– p (b, c) .
s (c, d) :– t, r (a, a) .
t :– r (a, c) , s (b, c) , q(a).
p(a, b).
q(b).
(of course, every fact is a ground rule)

A Datalog program is **ground** if all its rules are ground

1.2 Semantics

Now we give a formal definition of semantics, in order to then have an informal understanding of the meaning.

Herbrand Base
Given a Datalog program P, the **Herbrand Universe** of P (denoted by HU(P)) is the set of constant symbols occurring in P (ground or non-ground).

Given a ground Datalog program P, the **Herbrand Base** of P (denoted by HB(P)) is the set of ground atoms occurring in P (each atom appears just once independently from the number of occurrences in the program).
E.g., let P1 be the following ground program:
r(a, b) :– p (b, a) .
s (c, d) :– t, r (a, a) .
t :– r (a, c) , s (b, c) , q(a).
p(b, a).
q (b) .

Then, HB(P1) = r (a, b) , p (b, a) , s (c, d) ,t, r (a, a) , r (a, c) , s (b, c) , q (a) , q(b)

Interpretations of ground programs
Given a ground Datalog program P, an **interpretation** for P is a subset of HB(P).

E.g., the following are possible interpretations for the ground program P1 of the previous example:
I1 = {r (a, a) , p (b, a)}
I2 = {q (b) }
I3 = {r (a, b) , p(b, a) , s (c, d) ,t, r (a, a) , r (a, c) , s (b, c) , q (a) , q(b)}
I4 ={}
I5 = {p (b, a) , q (b) , r(a, b)}.
When we compute an interpretation we are saying how to interpret the HB, true or false: when we put an atom of the HB inside the interpretation it means that we interpret that atom as a true statement, and all the other in the HB are false.
⇒ But not all the interpretations satisfy the program

Models for ground programs
A ground positive rule r is **satisfied** in an interpretation I if either some atom in the body of r does not belong to I (if the body is not true it is ok, because the rule does not say anything about the current interpretation because in this interpr the body is not true: non-violation means satisfaction) or the atom in the head of r belongs to I.
(If all the atoms in the body belong to I and the atom in the head does not belong to I, the program is not satisfied in I because in these cases the rule is violated) Examples:
• the rule r(a, b) :– p (b, a) . is satisfied in the interpretations I2, I3, I4, I5, but not in I1.
• the rule (fact) q(b). is satisfied in the interpretations I2, I3, I5 but not in I1, I4

⇒ An interpretation I is a **model** for a ground Datalog program P, if all the rules in P are satisfied in I. But differently from DL, not all models play the same role, there are some models that are more important than others.

Example: let P be the program:
r(a) :– p (a) .
r(b) :– q (b) .
p(a).
HB(P) = {r(a), p(a), r(b), q(b) }

then, the following interpretations are models for P:
{p (a) , r (a) }
{p (a) , r (a) , q (b) , r (b)}
while the following are NOT models for P:
{ } (the fact p(a))
{p (a) } (1st rule: the body is true, but the head is not true)
{p (a) , r (a) , q(b)} (2nd rule not satisfied)

Herbrand Base of non-ground programs
Herbrand Base of a **non-ground** program P = HB(P) = set of all the ground atoms that can be built with the predicates and the constants occurring in P.
E.g., if P is the program
r(X, Y) :– p (X, Y) .
s (X, Y) :– r (X, Z) , s(Z, Y).
p(b, a).
q (b) .
We have 4 predicates and 2 constants (a and b) . HB(P) = {p (a, a) , p (a, b), p (b, a) , p (b, b), q (a) , q (b) , r (a, a) , r (a, b) , r (b, a) , r (b, b) ,s (a, a) , s (a, b) , s(b, a) , s(b, b)}
The HB can be exponentially larger than the original program because of the fact that you can combine all constants as argument of the predicates.

Interpretations and models
We know how to interpret a ground rule (given a P and an I we can check whether a rule is satisfied in I)... How do we interpret rules with variables? We will instantiate the variables to all the possible ground instantiations of this rule w.r.t. the constants appearing in the program. For each rule with variables we will create all its ground versions where we repleace every variable with a constant (we muove from a rule with variables to a set of rules without variables) (Obviously the same variables in a rule must be repleaced with the same constant)

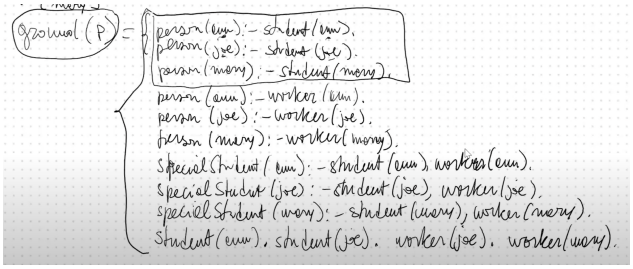
Given a non-ground Datalog program P, an **interpretation** for P is a subset of HB(P).

The **grounding** of a Datalog **rule** r with respect to a set of constants C, denoted as ground(r, C), is the set of all ground rules that can be obtained from r by replacing, for every variable x occurring in r, every occurrence of x with a constant from C.
The **grounding** of a non-ground Datalog **program** P, ground(P), is the ground Datalog program obtained by the union of all the sets ground (r, HU(P)) such that r $\in$ P.

An interpretation I of a non-ground Datalog program P is a **model** for P if I is a model for ground(P) (so we always interpret ground program).

Example:
person(X) :-student(X).
person(X) :- worker(X).
specialstudent(X) :- student(X), worker(X).
studnet(ann). student(joe). worker(j). worker(m).
- The safeness condition is satisfied
- The EDB and IDB division is satisfied

HU(P) = { a, j, m}
ground(P) $\rightarrow$ compute all the possible instantiations for the first, second and third rules, and add all the facts.



Minimal Models
An interpretation I is a minimal model for a (non-ground) Datalog program P, if I is a model for P and there exists no model I' for P such that I' is a strict subset of I.
Property: every positive Datalog program P has exactly one minimal model (we denote such a model by MM(P)).

Example: let P be the program

$r(a) :- p(a).$
$r(b) :- q(b).$
$p(a).$

the following interpretations are the models for P:

$\{p(a), r(a)\}$
$\{p(a), r(a), q(b), r(b)\}$

So, the minimal model for P is:

$\{p(a), r(a)\}$

Example
Let's try to find a model of the program of the previous example: We have 4 facts, it means that an interpretation in order to be a model must contain these 4 facts because they must be true. $I_1 = \{$ student(a), student(j), worker(j), worker(m)$\}$. It is not a model of ground(P)... see the first rule: we have student(a) and student(j) so its body is true, so according to the semantics also its head must be true, but person(a) and person(j) are not in I.
So we need to add person(a), person(j) in order to not violate rule1 and rule2. Rule3: we do not have student(m) so it is not violated. Rule4-5: satisfied if I add person(a) and person(j). Rule6: add person(m) otherwise it is not satisfied... Go on in this way...
$I_2 = \{$ student(a), student(j), worker(j), worker(m), person(a), person(j), person(m), specialstudent(j)$\}$.
I_2 is a model for ground(P) and so also for P.
Is it a minimal model? Can we find a model that is a subset of I_2?
The four facts must be true so they must be in the interpretation. But at the same time what we have done is to add things that are necessarely in order to satisfy all the rules of P, so I cannot delete any other atom because the interpr will be no longer a model.
So I_2 is the minimal model of P.

1.3 Reasoning in Positive Datalog

- Basic reasoning task: construction of MM(P)
- Derived reasoning task: ground atom entailment → Given a Datalog program P and a ground atom α, we say that P entails α if $\alpha \in$ MM(P)

Immediate Consequence Operator
Given a **ground** positive Datalog program P, the **immediate consequence operator** for P, denoted as T_P, is the function over the domain of interpretations for P defined as follows:
$T_P(I) = \{\alpha |$ there exists a rule α :- $\beta_1, ..., \beta_n$ in P s.t $\{\beta_1, ..., \beta_n\} \subseteq I \}$
The **least fixed point**of the function T_P is the minimal interpretation I such that $T_P(I) = I$.
Also it always exists and is unique, and coincides with MM(P) (→ it will be the basis to construct a new algo to compute the MM(P)).

Example: let P be the following ground program:

$r(a) :- p(a).$
$r(b) :- q(b).$
$q(b) :- r(b).$
$p(a).$

Then:

$T_P(\{\}) = \{ p(a) \}$
$T_P(\{ p(a) \}) = \{ p(a), r(a) \}$
$T_P(\{ p(a), r(a) \}) = \{ p(a), r(a) \}$ (least fixed point)
$T_P(\{ q(b) \}) = \{ p(a), r(b) \}$
$T_P(\{ p(a), r(b) \}) = \{ p(a), r(a), q(b) \}$
$T_P(\{ p(a), r(a), q(b) \}) = \{ p(a), r(a), r(b) \}$
$T_P(\{ p(a), r(a), r(b) \}) = \{ p(a), r(a), q(b) \}$
$T_P(\{ p(a), q(b), r(a), r(b) \}) = \{ p(a), q(b), r(a), r(b) \}$ (fixed point)

1) Empty interpretation: the body of a fact is empty so the fact is contained to the T_P of the empty interpretation, not the heads of the other rules.
2) The fact p(a) yes because it is in the interpr, then also r(a) because it is the head of the first rule, which body is contatined in the interpr.
3) Like the previous case. r(a) does not appear in any rule body. → Least Fixed Point
4) In T_P we have the fact always for the same reason. Then also r(b) because it is the head of the second rule, which body is contatined in the interpr.
8) I take all the atoms in the HB(P) and T_P returns all the atoms → Fixed Point The way of reasoning is the same for all other examples (In a T_P can only appear atoms that are in the head of some rules, not atoms that are only in the body).

Naive Evaluation

```
Algorithm naive-evaluation
Input: ground positive Datalog program P
Output: MM(P)
begin
  let I'={};
  repeat
    let I=I';
    compute I'=T_P(I)
  until I'==I;
  return I
end
```

This algorithm executes at most k+1 iterations of the repeat-until loop,
where k is the number of rules of P

Every time that I execute the algo either I do not add any ground atom, and then I'=I so the algo stops, or if I stay in the repeat loop I will add at least one ground atom, that was not before in I'. So I can add every time new atoms, but I can add at most k atoms (+1 because it is the last iteration in which I' and I are the same and I realize to have find a LFP).
In general the cost of the algo is linear w.r.t the number of rules.

1.3.1 Reasoning over non-ground programs

The naive evaluation algorithm could be used also for non-ground programs:
1. First, compute the ground program ground(P);
2. Then, execute the naive evaluation algorithm on ground(P).
The interpretation returned by the algorithm is the minimal model of ground(P) and therefore the minimal model of P.

In general we know that computing a ground of a program may be exponentially larger than the original P, and obviously it will take exponential time.
In order to solve this problem we now present the semi-naive evaluation algorithm, which optimizes the naive evaluation.

Delta Predicates
Idea of the optimization: reformulate the program P in a way such that the immediate consequence operator T_P can derive a ground atom only if its derivation depends on at least one ground atom derived in the previous application of T_P in the algorithm.
This minimizes redundant derivations, i.e., repeated derivations of the same ground atoms in different applications of T_P.
This idea is realized by introducing Delta versions of the IDB predicates of the program, and rewriting the program through the usage of such Delta predicates.

In the algorithm, the extension of a Δ-predicate ΔP represents the ground atoms relative to predicate p derived by the **previous** application of the immediate consequence operator.
A second set of Δ'-predicates is used in rule heads: in this way, each Δ'p represents the ground atoms relative to predicate p derived by the **current** application of the immediate consequence operator.
Basically we are avoiding to compute all the consequences by annotating with a different predicate name the new ground atom derived in the previous and in the current application of the T_P.
Let's see how we can do it in practise...

Δ-Transformation of a rule
In the following, we denote a rule r of a positive program P as follows:
α :- $\beta_1, ..., \beta_k, \lambda_1, ..., \lambda_h$
where every β_i is an IDB-atom (i.e. an atom in which an IDB predicate of P occurs), and every λ_i is an EDB-atom.

Given a rule r of the above form, Δr is the following set of k rules:
$\Delta'\alpha$:– $\Delta\beta_1, ..., \beta_k, \lambda_1, ..., \lambda_h$
$\Delta'\alpha$:– $\beta_1, \Delta\beta_2, ..., \beta_k, \lambda_1, ..., \lambda_h$
....
$\Delta'\alpha$:– $\beta_1, ..., \Delta\beta_k, \lambda_1, ..., \lambda_h$
where:
• if β_i = r(t_1,...,t_n), then $\Delta\beta_i$ denotes the atom Δr(t_1,...,t_n)
• if α = r(t_1,...,t_n), then $\Delta'\alpha$ denotes the atom Δ'r(t_1,...,t_n)
Therefore, if the body of r contains k atoms with IDB predicates, Δr contains k rules (for every such atom there is a rule where the delta predicate is used in such an atom instead of the standard predicate)

Example: let IDB = $\{r/1, s/2\}$, EDB = $\{p/2\}$, and let P be as follows:

$r(X)$:– $s(X,Y), p(X,Y)$. [r1]
$s(X,Y)$:– $p(X,Y), p(Y,Z)$. [r2]
$s(X,Y)$:– $s(X,Z), r(Y), p(Y,X)$. [r3]
$p(a,b)$. [r4]

Then:

Δr1 = { $\Delta' r(X)$:– $\Delta s(X,Y), p(X,Y)$. }
Δr2 = { }
Δr3 = { $\Delta' s(X,Y)$:– $\Delta s(X,Z), r(Y), p(Y,X)$.
$\Delta' s(X,Y)$:– $s(X,Z), \Delta r(Y), p(Y,X)$. }
Δr4 = { }

First rule: in r(X), with s(X,Y) I consider all the consequences that I derived so far, instead with Δs(X,Y) I only consider the consequences (the ground atoms) derived from the previous iteration (if in the previous iteration I derived just one ground atom, all the 100 atoms that I derived in the previous iterations are not considered for this rule). Consequently set of the immediate consequences of rule Δ'r(X) will be smaller than the set of the immediate consequences of rule r(X).
This Delta rules are used in order to reduce the amount of work to compute the immediate consequence (by reducing the numer of immediate consequences of the various rules).
In some cases like r2, the rule disappears: we start from the EDB so after the first iteration we have all the immediate consequences of this rule and in the next iteration (that only adds intentional atoms to the

interpretation) its atoms will not appear, since all the immediate consequences that can be produced of this rule are computed in the first iteration.

Δ-Transformation of a program

Given a positive Datalog program P, the Δ-transformation of P, denoted as ΔP, is the program obtained as the union of all the sets Δr of every rule r in P.

⇒ The algorithm then computes the immediate consequence operator $T_{\Delta P}$ of ΔP, treating both Δ-predicates and Δ'-predicates like all other standard predicates.

Semi-naive Evaluation

```
Algorithm semi-naive-evaluation
Input: positive Datalog program P
Output: MM(P)
begin
  let I=EDB(P); //(EDB(P) is the set of facts in P)
  compute I'=T_P(I);
  if I==I' then return I;
  Δ'I = { Δ'α | α ∈ I' − I };
  repeat
    let I = I ∪ { α | Δ'α ∈ Δ'I };
    let ΔI = { Δα | Δ'α ∈ Δ'I };
    let Δ'I = T_ΔP(I ∪ ΔI)
  until Δ'I== {};
  return I
end
```

Here we start with I = EDB(P), that is the set of facts in P, basically in this way we reduce of one the number of interations, since in the Naive Evaluation the first iteration produces an interpretation that is the set of facts in P.
There is a distinction between new stuff and old stuff: the old stuff is stored in I, because it stores everything. At every iteration ΔI contains all new consequences added in the previous iteration. At every iteration Δ'I contains all new consequences added in the current iteration.
In general, in this way computing the immediate consequence of ΔP is faster than computing the immediate consequence of P (body constituted only by extentional atoms).

Example

Let P be the following positive Datalog program:

$$r(X,Y) :- p(X,Y).$$
$$r(X,Z) :- p(X,Y), r(Y,Z).$$
$$s(X,Y) :- r(Y,X).$$
$$p(a,b).$$
$$p(b,c).$$
$$p(c,d).$$

- The initial value of I is the set of facts of P, i.e. $\{ p(a,b), p(b,c), p(c,d) \}$
- Then, I'=T_P(I)= I ∪ $\{ r(a,b), r(b,c), r(c,d) \}$ (applying the first rule)
- So, Δ'I = $\{\Delta' r(a,b),\ \Delta' r(b,c), \Delta' r(c,d) \}$
- Now notice that ΔP is the following program:

$$\Delta' r(X,Z) :- p(X,Y), \Delta r(Y,Z).$$ [r1]
$$\Delta' s(X,Y) :- \Delta r(Y,X).$$ [r2]

- Then, the algorithm executes the repeat-until loop for the first time, obtaining:

  $\mathtt{I} = \mathtt{I} \cup \{ r(a,b), r(b,c), r(c,d) \}$
  $\Delta\mathtt{I} = \{ \Delta r(a,b), \Delta r(b,c), \Delta r(c,d) \}$
  $\Delta'\mathtt{I} = \mathtt{T}_{\Delta P}(\mathtt{I} \cup \Delta\mathtt{I}) =$
  $= \{ \Delta' r(a,c), \Delta' r(b,d), \Delta' s(b,a), \Delta' s(c,b), \Delta' s(d,c) \}$
  (applying rules r1 and r2 of ΔP)

- Then, the algorithm executes the repeat-until loop for the second time, obtaining:

  $\mathtt{I} = \mathtt{I} \cup \{ r(a,c), r(b,d), s(b,a), s(c,b), s(d,c) \}$
  $\Delta\mathtt{I} = \{ \Delta r(a,c), \Delta r(b,d), \Delta s(b,a), \Delta s(c,b), \Delta s(d,c) \}$
  $\Delta'\mathtt{I} = \mathtt{T}_{\Delta P}(\mathtt{I} \cup \Delta\mathtt{I}) =$
  $= \{ \Delta' r(a,d),\ \Delta' s(c,a), \Delta' s(d,b) \}$ (applying r1 and r2)

- Then, the algorithm executes the repeat-until loop for the third time, obtaining:

  $\mathtt{I} = \mathtt{I} \cup \{ r(a,d), s(c,a), s(d,b) \}$
  $\Delta\mathtt{I} = \{ \Delta r(a,d), \Delta s(c,a), \Delta s(d,b) \}$
  $\Delta'\mathtt{I} = \mathtt{T}_{\Delta P}(\mathtt{I} \cup \Delta\mathtt{I}) = \{ \Delta' s(d,a) \}$ (applying r2)

- Then, the algorithm executes the repeat-until loop for the fourth time, obtaining:

  $\mathtt{I} = \mathtt{I} \cup \{ s(d,a) \}$
  $\Delta\mathtt{I} = \{ \Delta s(d,a) \}$
  $\Delta'\mathtt{I} = \mathtt{T}_{\Delta P}(\mathtt{I} \cup \Delta\mathtt{I}) = \{\}$

- Since $\Delta'\mathtt{I}$ is empty, the algorithm exits the repeat-until loop and terminates returning the interpretation I, that is, the set

  $\{ p(a,b), p(b,c), p(c,d), r(a,b), r(b,c), r(c,d), r(a,c), r(b,d),$
  $s(b,a), s(c,b), s(d,c), r(a,d), s(c,a), s(d,b), s(d,a) \}$

  Such an interpretation I is the minimal model of P.

1.3.2 Datalog Rules vs DL axioms

There is some relationship between the two languages?
Let's consider again this example:
EDB = student/1, worker/1, livesin/2, city/2 (appears only in the body or in facts)
IDB = person/1, specialstudent/1, citystudent/1
|person(X) :-student(X).
|person(X) :- worker(X).
|specialstudent(X) :- student(X), worker(X).
|citystudent(X) :- student(X), livesin(X,Y), city(Y) |student(ann). worker(joe). livesin(joe, rome). city(rome)
(the facts are easly translated because are all concept/role assertion and are already in a correct form)
(If an axiom has arity 1 can be a concept in DL, if it has arity 2 can be a role in DL)
Can we write this program in DL generating a DL knowledge base?
DL TBOX **T**: Student $\subseteq$ Person (In Datalog we have "if student(X), then person(X)", so student is a subconset of person)
Worker$\subseteq$Person
Student $\cap$ Worker$\subseteq$SpecialStudent
Student $\cap\exists$livesin.City$\subseteq$CityStudent
livesin(X,Y), city(Y)$\rightarrow \exists$livesin.City: the set of elements of the domain that partecipate in the livesin predicate s.t. the other component of the predicate belongs to the concept City.

In general, a lot of Datalog rules can be translated into DL, but it is not always simple like this exam-

ple.
It cannot be done for example when we have an axiom with arity higher than two (we cannot represent it neither as a concept nor as a role).
In DL we do not have the distinction between EDB and IDB because, as we said before, even if the distinction is not satisfied, we will be able to re-write the Datalog program into one that satisfies this condition.
⇒ It is not always possible to translate a Datalog program in DL, and viceversa.

Recursion
EDB = { flight/2 } flight(a,b)-¿ There exists a direct flight from airport a to airport b
IDB = { connected/2 } connected(a,b)-¿ There exists a flight connection (either direct or undirect flight) from a to b

connected(X,Y) :- flight (X,Y) (if there exists a flight from X to Y, then they are connected)
connected(X,Y) :- flight(X,Z), connected(Z,Y) (if there exists a direct from X to Z and Z and Y are connected in some way, then X and Y are connected)
flight(London,Rome).
flight(Milan, London)
flight(London, Moscow)
flight(London, Rome)
flight(Rome, Toronto)

First Iteration: The MM(P) of corse contains all the facts.
But since all the facts are of type "flight" that is the body of the first rule, the MM(P) must also contains the head of the first rule.. so we add all the connected(X,Y) axioms to MM(P).
MM(P) = flight(London,Rome), flight(Milan, London), flight(London, Moscow), flight(London, Rome), flight(Rome, Toronto), connected(London,Rome), connected(Milan, London), connected(London, Moscow), connected(London, Rome), connected(Rome, Toronto)...
Second Iteration: But now since we have some "connected" facts in the MM(P) and the flight facts that were already there, then the second rule has immediate consequences:
For example we have flight(London, Rome) and connect(Rome, Toronto), in this case both the axioms in the body of the second rule are true, so it produces as immediate consequence of rule 2 connected(London, Toronto), and so on.
MM(P) = flight(London,Rome), flight(Milan, London), flight(London, Moscow), flight(London, Rome), flight(Rome, Toronto), connected(London,Rome), connected(Milan, London), connected(London, Moscow), connected(London, Rome), connected(Rome, Toronto), connected(Rome, London), connected(Milan, Moscow), connected(Milan, Rome), connected(London, Toronto), connected(London, Milan)
Third Iteration: now we can generate the "two stops" connections, like connected(Rome, Moscow), connected(Milan, Toronto), etc

The power of recursion stays in the facts that a recursive rule generates with the immediate consequence operator new facts that can be used to re-activate that rule in the next iteration and generate other facts. So it can generate a huge number of iterations (that obviously depends on the data because after at most k+1 we must reach the LFP).

Recursion like this cannot be simulated in DL.
The first rule becames: flight$\subseteq$connected (Role Inclusion)
The second rule is not translatable in DL (due to this type of recursion)

Chapter 2

Positive Datalog with Constraints

Constraints
A constraint is a new kind of rule of the form: $:- \beta_1, ..., \beta_n$
where:
• n is a positive integer
• every β_i is an atom
That is, a constraint is a rule with an empty head.

Examples of constraints:
:– p (X, Y) , r(Y, Z).
:– (p a, b) , q(b).
:– s (Y, X) .
A constraint is **ground** if it does not contain occurrences of variables.

Datalog Programs with Constraints
A Datalog program with constraints is a set of positive rules and constraints.

2.1 Semantics of Constraints

A ground constraint $:- \beta_1, ..., \beta_n$ is **satisfied** in an interpretation I if at least one of its atoms β_i does not belong to I.

We naturally extend the notion of grounding of a Datalog program P to the presence of non-ground constraints in P:
The **grounding** of a **constraint** c with respect to a set of constants C, denoted as ground(c, C), is the set of all ground constraints that can be obtained from c by replacing, for every variable x occurring in c, every occurrence of x with a constant from C.
The **grounding** of a (non-ground) positive Datalog **program** with constraints P, denoted as ground(P), is the ground Datalog program obtained by the union of all the sets ground(r, HU(P)) such that r is a rule in P, and all the sets ground(c, HU(P)) such that c is a constraint in P.

An interpretation I is a **model** for a Datalog program with constraints P if every ground rule and every ground constraint in ground(P) is satisfied in I.

(analogous definition of minimal model as in the case of positive Datalog)

Property: every positive Datalog program with constraints P has either no models (and hence no minimal models) or exactly one minimal model (and in the latter case we denote such a model by MM(P)).

2.2 Reasoning over programs with constraints

The techniques for reasoning with positive Datalog programs (naive or semi-naive evaluation) can be easily extended to the presence of contraints in the program.
Let P be a Datalog program with constraints. Then:
- Let P' be the program obtained from P eliminating all the constraints;
- Compute (using e.g. the naive or semi-naive evaluation) MM(P'), the minimal model of P'; - Check whether every constraint in P is satisfied in MM(P'): if this is the case, then MM(P') is the minimal model of P; otherwise P has no models (and hence no minimal models).
(Of course, every constraint in P is satisfied in MM(P') iff every ground constraint in ground(P) is satisfied in MM(P'))

Example

Let P be the following program with constraints:

$r(X,Y) :- p(X,Y).$
$r(X,Z) :- p(X,Y), r(Y,Z).$
$s(X,Y) :- r(Y,X).$
$:- p(X,X).$ [c1]
$:- r(X,Y), r(Y,X).$ [c2]
$p(a,b).$
$p(b,c).$
$p(c,d).$

First, we notice that the program P' obtained from P eliminating the two constraints c1 and c2 corresponds to the previous program for which we have computed the minimal model through the semi-naive computation.

So, the program P' has the following minimal model MM(P'):

$\{ p(a,b), p(b,c), p(c,d), r(a,b), r(b,c), r(c,d), r(a,c), r(b,d),$
$s(b,a), s(c,b), s(d,c), r(a,d), s(c,a), s(d,b), s(d,a) \}$

Now we have to check whether the constraints c1 and c2 are satisfied in MM(P'). We have:

$ground(c1, HU(P))$ =
$:- p(a,a).$
$:- p(b,b).$
$:- p(c,c).$
$:- p(d,d).$

$ground(c2, HU(P))$ =
$:- r(a,a), r(a,a).$
$:- r(a,b), r(b,a).$
$:- r(a,c), r(c,a).$
$:- r(a,d), r(d,a).$
$:- r(b,a), r(a,b).$
$:- r(b,b), r(b,b).$
$:- r(b,c), r(c,b).$
$:- r(b,d), r(d,b).$
$:- r(c,a), r(a,c).$
$:- r(c,b), r(b,c).$
$:- r(c,c), r(c,c).$
$:- r(c,d), r(d,c).$
$:- r(d,a), r(a,d).$
$:- r(d,b), r(b,d).$
$:- r(d,c), r(c,d).$
$:- r(d,d), r(d,d).$

It is easy to verify that every ground constraint in $ground(c1, HU(P))$ is satisfied in MM(P'), and that every ground constraint in $ground(c2, HU(P))$ is satisfied in MM(P').

Consequently, MM(P') is the minimal model of P.

Now let P'' be the program obtained from P adding the following constraint:

$$:- r(X,Y), r(Y,W), r(W,Z), s(Z,X). \quad \text{[c3]}$$

Among others, the following ground constraint belongs to $ground\big(c3, HU(P)\big)$:

$$:- r(a,b), r(b,c), r(c,d), s(d,a).$$

This ground constraint in NOT satisfied in MM(P') (because all the atoms in the body of the constraints belong to MM(P')), therefore c3 is non satisfied in MM(P'). Consequently, P'' has no models (and hence no minimal models).

Chapter 3

Datalog with Negation

3.1 Syntax

Datalog rule with negation = expression of the form α :- $\beta_1, ..., \beta_n, not\lambda_1, ..., not\lambda_m$
where:
• n,m are non-negative integers
• α is an atom
• every β_i is an atom
• every λ_i is an atom (and is called negated atom)
• (safeness condition) every variable symbol occurring in the rule must appear in at least one of the positive atoms of the rule body $\beta_1, ..., \beta_n$

A Datalog program with negation is a set of Datalog rules with negation.

EDB = $\{p/1\}$, IDB = $\{q/0, r/2, s/2, t/1, u/3\}$
Const = $\{a, b, c\}$
Var = $\{X, Y, Z, W\}$

Correct rules with negation:

- $r(X,Y) :- r(Y,Z), s(Y,X), not\ r(X,Z).$
- $r(a,b) :- not\ p(b,c).$
- $t(X) :- r(X,Y), s(Y,Z), not\ r(X,Z), not\ s(Z,Y).$
- $t(a) :- not\ r(b,c).$
- $s(b,a) :- r(a,b), not\ q.$
- $q :- not\ p(a), not\ s(b,c).$

Pred = $\{p/1, q/0, r/2, s/2, t/1, u/3\}$
Const = $\{a, b, c\}$
Var = $\{X, Y, Z, W\}$

Incorrect rules with negation:

- $t(X) :- not\ p(X).$
  - variable X is not safe
- $t(Y) :- p(Y), not\ p(X).$
  - variable X is not safe
- $t(W) :- r(X,Y), s(Y,Z), not\ s(Y,W).$
  - variable W is not safe
- $not\ t(X) :- p(X).$
  - cannot use not in the head atom

3.2 Semantics

A **ground** rule r α :- $\beta_1, ..., \beta_n, not\lambda_1, ..., not\lambda_m$ is satisfied in an interpretation I if at least one of the following conditions holds:
• at least one of the atoms $\beta_1, ..., \beta_n$ does not belong to I
• at least one of the atoms $\lambda_1, ..., \lambda_m$ belongs to I
• the atom α belongs to I

Examples:

1) the rule $r(a,b) :- not\ p(b,c)$.

- is satisfied in the interpretations $\{p(b,c)\}$, $\{r(a,b)\}$, $\{p(b,c), r(a,b)\}$
- is not satisfied in the interpretation $\{\ \}$

2) the rule $r(a,b) :- q(a), not\ p(b,c)$.

- is satisfied in the interpretations $\{\ \}$, $\{p(b,c)\}$, $\{r(a,b)\}$, $\{p(b,c), r(a,b)\}$, $\{p(b,c), q(a)\}$, $\{q(a), r(a,b)\}$, $\{p(b,c), q(a), r(a,b)\}$
- is not satisfied in the interpretation $\{q(a)\}$.

An interpretation I is a **model** for a ground Datalog program with negation P if all the rules of P are satisfied in I.

To define models for non-ground programs with negation, we extend the notion of grounding to rules with negation in the obvious way:
The **grounding** of a Datalog **rule** with negation r with respect to a set of constants C, denoted as ground(r, C), is the set of all ground rules with negation that can be obtained from r by replacing, for every variable x occurring in r, every occurrence of x with a constant from C.
The **grounding** of a non-ground Datalog **program** with negation P, ground(P), is the ground Datalog program with negation obtained by the union of all the sets ground(r, HU (P)) such that r∈P.

An interpretation I is a **model** for a non-ground Datalog program with negation P if I is a model for ground(P).

Following what is done for positive programs, the next step would be to consider only minimal models of a program with negation.
However, this step is problematic in the presence of negation:
1) First, differently from the positive case, **multiple** minimal models for a program with negation may exist: e.g. (see previous examples), the program consisting of the single rule r(a, b) :– not p(b, c). has two minimal models:
$I_1 = \{p(b,c)\}$
$I_2 = \{r(a,b)\}$
2) Moreover, some minimal models are **not "intended"** ones:
e.g. the above minimal model $I_1 = \{p(b,c)\}$ appears quite strange, since in the program there is no rule that allows for deriving $I_1 = \{p(b,c)\}$ (i.e., that has p(b, c) in the head of the rule)(there are no facts or rule that says "if something happens then p(b,c) is true).
We cannot say that something is postive just because it makes false the body of a rule, we should say that something is positive because there is a rule that forces the positiveness of this atom, otherwise it is false.
I_2 is ok.

So, the declarative semantics based on a "classical" notion of model (and interpretation of negation) does not seem satisfactory.
Also, extending the operational semantics of positive programs to the presence of negation in rules seems problematic:

Example: let P be the program

$q(a) :\!-\ not\ q(a).$

Let I be our starting empty interpretation.

The immediate consequence operator T_P applied to I would produce the interpretation I'={q(a)}, because q(a) is false in I, therefore not q(a) is true in I, and so q(a) is derived.

But if now we apply T_P to I', we obtain the empty interpretation: ideed, since not q(q) is true in I', not q(a) is now false in I', therefore there are no immediate consequences. So, we have obtained again the initial interpretation I.

It is immediate to conclude that **the iterated application of the T_P operator will never converge to a least fixpoint**.

3.3 Answer Set Semantics

We thus introduce a new declarative semantics for Datalog programs with negation, called **Answer Set Semantics**.

Let P be a program with negation and let I be an interpretation. The **reduct** of P with respect to I, denoted as P/I, is the positive ground program obtained as follows:

- Delete from ground(P) every rule R such that an atom $not\beta$ occurs in the body of R and β belongs to I;
- Delete from every rule R in ground(P) every atom $not\beta$ occurring in the body of R such that β does not belong to I.

In this way we delete all the occurrences of the negated atoms: for example for the atom $not\beta$ either we delete the rule in which it appears or we delete the atom from the rule.

Let P be a Datalog program with negation. An interpretation I is an **answer set** of P if I is the minimal model of the positive program P/I.
→ The notion of answer set repleaces the notion of minimal model.

Reduct of a program

Example: let P be the program

$r(a) :\!-\ p(a), not\ q(a).$
$s(a) :\!-\ not\ t(a).$
$t(a) :\!-\ r(a), not\ p(a).$
$p(a).$

1) Let I be the interpretation $\{p(a), s(a)\}$. Then, P/I is the positive program

$r(a) :\!-\ p(a).$
$s(a).$
$p(a).$

2) Let I' be the interpretation $\{p(a), t(a)\}$. Then, P/I' is the positive program

$r(a) :\!-\ p(a).$
$p(a).$

Example (contd.): let P be the program

$r(a) :\!-\ p(a), not\ q(a).$
$s(a) :\!-\ not\ t(a).$
$t(a) :\!-\ r(a), not\ p(a).$
$p(a).$

3) Finally, let I'' be the interpretation $\{p(a), r(a), s(a)\}$. Then, P/I'' is the positive program

$r(a) :\!-\ p(a).$
$s(a).$
$p(a).$

Answer Sets: Example
Let P be a ground Datalog program with negation. An interpretation I is an **answer set** of P if I is the minimal model of the positive program P/I.

Example (contd.):

- the minimal model of program P/I is $\{p(a), r(a), s(a)\}$, therefore I is **not** an answer set of P
- the minimal model of program P/I' is $\{p(a), r(a)\}$, therefore I' is **not** an answer set of P
- the minimal model of program P/I'' is $\{p(a), r(a), s(a)\}$, therefore I'' is an answer set of P

Properties of Answer Sets

- Every answer set is a minimal model of P, but not viceversa.

Example:

Let P be the program consisting of the single rule $r(a, b) :\!-\ not\ p(b, c)$.

P has two minimal models:
$I_1 = \{p(b, c)\}$
$I_2 = \{r(a, b)\}$

However, only I_2 is an answer set of P, since:

- P/I_1 is the empty program, whose minimal model is the empty set
- P/I_2 is the program $r(a, b)$. whose minimal model is I_2

- A program with negation may have 0, 1 or multiple answer sets.

Examples:

1) Let P be the program consisting of the single rule $p(a) :\!-\ not\ p(a)$.
Then, P has no answer sets.

2) Let P be the program
$p(a) :\!-\ not\ q(a)$.
$q(a) :\!-\ not\ p(a)$.
Then, P has two answer sets:
$I_1 = \{p(a)\}$
$I_2 = \{q(a)\}$

3.4 Reasoning over Programs with Negation

- Basic reasoning tasks:
  - decide whether P has at least one answer set
  - compute all the answer sets of P
- Derived reasoning tasks:
  - Skeptical entailment: given a program P and a ground atom α, establish whether α belongs to all the answer sets of P.
  - Credulous entailment: given a program P and a ground atom α, establish whether α belongs to at least one answer set of P.

Basic reasoning technique that computes all the answer sets of P:

```
AS= {};
for every interpretation I over HB(P) do
  if I == MM(P/I)
  then add I to AS;
return AS;
```

(Checking whether I==MM(P/I) can be done through the naive or semi-naive evaluation of positive programs)

The computational cost of the above algorithm is **exponential**, even if we start from a ground program P.

(The reason is that there is an exponential number of subsets of HB(P) with respect to the size of HB(P), which is proportional to the size of ground(P))

Chapter 4

Datalog with Stratified Negation

As previously illustrated, the addition of negation to Datalog creates both semantic and computational issues.

The problematic examples presented above suggest that the use of **negation** creates problems when it is used in combination with **recursion**.

To overcome such problems, we define a subclass of programs with a non-recursive form of negation, called programs with s**tratified negation**.
What we are going to see is if we do not allow negation in combination with recursion, then we will be able to have a class of programs with just one answer set which coincides with the minimal model and the computation of this model can be done by modifying the semi-naive evaluation, having the same complexity of the Datalog programs.

4.1 Precedence Graph

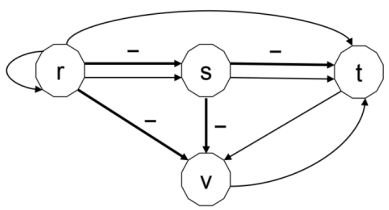
Let P be a Datalog program with negation. The precedence graph of P is a graph G=(V,E,L) where V is the set of vertices, E is the set of edges, and L is a labeling of the edges in E, defined as follows:

- There is one vertex v in V for every IDB predicate in P;
- There is an edge (s,t) in E (without label) if s,t are IDB predicates and P contains a rule R such that t appears in the head of R and s appears in a positive atom in the body of R;
- There is a negated edge, i.e. an edge with label "-", (s,t) in E if s,t are predicates and P contains a rule R such that t appears in the head of R and s appears in a negated atom in the body of R.

Example: let P be the following Datalog program with negation:

$r(X,Y) :- p(X,Y),\ not\ p(Y,X).$
$r(X,Y) :- p(X,Y),\ \ r(Y,Z).$
$s(X,Y) :- r(Z,X),\ r(Z,Y),\ not\ r(X,Y).$
$t(X) :- r(Y,X),\ s(X,X),\ not\ s(Y,Y).$
$t(X) :- v(X,Y).$
$v(X,Y) :- t(X),\ t(Y),\ not\ r(X,Y),\ not\ s(X,Y).$
$p(a,b).\ \ p(b,c).$

Precedence graph G of P:



4.2 Program with Stratified Negation

We say that a Datalog program with negation P is **stratified** if no cycle of the precedence graph of P contains a negated edge.

Example (contd.):
The precedence graph G has no cycle that contains a negated edge: the only cycles are: 1) the single edge (r,r) which is a positive edge; 2) the path (t,v), (v,t) which is only made of positive edges.
→ Consequently, P is a stratified program (or, equivalently, a program with stratified negation).
For example if we added a negated atom in the fifth rule "not t(Y)", we create a negative dependence from t in the body and t in the head, so it creates a negative loop.

4.3 Operational Semantics

For programs with stratified negation, the problems with the operational semantics (i.e. the iterated application of the immediate consequence operator) previously described do not arise.

Property: every stratified program P has a **unique** answer set, which coincides with the unique minimal model of P, and is denoted by MM(P).

It is possible to identify an algorithm that **deterministically** converges to the unique minimal model of the program with stratified negation.
But before let's define the notion of stratification.

Stratification of a program
A vertex v of G has a **negative dependence** if there exists a vertex v' in G such that there is a path from v' to v passing through a negated edge.
The **stratification** of a stratified program P is a sequence S1,..., Sk of sets of IDB predicates of P (called **strata**) obtained as follows:

G = predecence graph of P;
i=0;
while G is not empty do begin

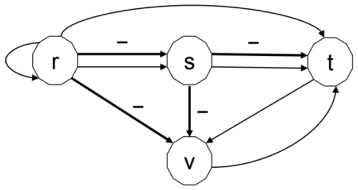
- i=i+1;
- Si = the set of vertices of G that do not have negative dependencies (given the fact that the program is stratified, it must be at least one negative dependence);
- G = the precedence graph of P obtained considering (in addition to the initial EDB predicates) the predicates in S1,..., Si as EDB (delete them and all the edges involving them from the graph)

end;
return S1,..., Si;

The maxium number of iteration (and consequently the number of stata) is equal to the number of IDB predicates k.

Example (contd.):

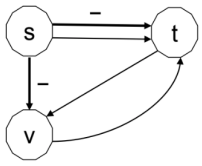
The algorithm first considers the initial precedence graph G of P:



The only vertex that does not have negative dependencies is r, therefore

$S_1 = \{ r \}$

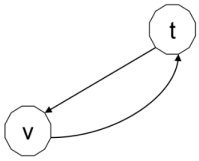
We now analyze the precedence graph obtained considering r an EDB predicate, i.e. we eliminate the vertex r and all its outcoming edges:



The only vertex that does not have negative dependencies is s, therefore

$S_2 = \{ s \}$

We now analyze the precedence graph obtained considering both r and s as EDB predicates, i.e. we eliminate from the previous graph the vertex s and all its outcoming edges:



Since there are no more negated edges, both t and v have no negative dependencies, therefore

$S_3 = \{ t, v \}$

And since, after deleting t and v, the graph is empty, the algorithm ends returning the stratification S_1, S_2, S_3.

Minimal model of a stratified program

The algorithm is a generalization of the semi-naive evaluation for positive datalog programs.

Let $S_1,\ldots, S_k$ be the stratification of P;
$MM_0 = EDB(P)$;
For i=1 to k do begin
  $P(S_i)$ = the program obtained from P by considering only the rules having a predicate from S_i in their head;
  MM_i = minimal model of $P(S_i) \cup MM_{i-1}$
end;
return MM_i;

In $P(S_i)$ we still can have negation in the body, so how can we compute a minimal model of a program

with negation?
We have to extend the computation of the minimal model in presence of negation, but however in this case it is simplified because the negation is always in front of existential predicates becuase when we take the stratum we are guaranteeing that the only predicates that can have negation in stratum S_i are the predicates of lower strata or extentional predicates; so the predicates appearing in the head cannot be negated in the body, only predicates that we have already computed in the previous strata can be negated in the stratum $S_i \rightarrow$ that is why stratification is so important.
The idea is: we are able to compute negated atoms about predicates for which we have already computed their extension in the minimal model.
So when we arrive at a negated atom, we already know everything about that predicate and so we can also evaluate the negation of the predicate.
Stratification implies that we will compute the extension of the predicates in the minimal model of the program stratum by stratum: in MM_0 we have the extensions of the extentional predicates, in MM_1 we have the extensions of the predicates of S_1, and so on...$\rightarrow$ At every iteration we increase the minimal model with the facts about the predicates in stratum S_i.
The important point is that when we compute the predicates of stratum S_i, we know how to evaluate the negation of the predicates of S_{i-1}, because we have already computed all the positive consequences of the predicates of the lower strata and we are also able to evaluate negation now.
Since I have not derived a fact about r (predicate of stratum S_{i-1}) in the previous iteration, now I know that this fact is negative in the minimal model, because if it were positive I would have derived it in the previous iteration.
That's the trick!

Let's consider the program of the previous example: the algorithm ends returning the stratification S_1 = r, S_2 = s and S_3 = t, v.
It means that we are going to consider the first two rules as $P(S_1)$, the third rule is $P(S_2)$, and the last 3 rules are considered as $P(S_3)$.

Example (contd.):

If we execute the previous algorithm on program P, we obtain:

MM_0 = EDB(P) = $\{ p(a,b), p(b,c) \}$

$P(S_1)$ =

$r(X,Y) :- p(X,Y),\ not\ p(Y,X).$
$r(X,Y) :- p(X,Y),\ r(Y,Z).$

$MM_1 = MM_0 \cup \{ r(a,b), r(b,c), r(a,c) \}$

$P(S_2)$ =

$s(X,Y) :- r(Z,X),\ r(Z,Y),\ not\ r(X,Y).$

$MM_2 = MM_1 \cup \{ s(b,b), s(c,c) \}$

$P(S_3)$ =

$t(X) :- r(Y,X),\ s(X,X),\ not\ s(Y,Y).$
$t(X) :- v(X,Y).$
$v(X,Y) :- t(X),\ t(Y),\ not\ r(X,Y),\ not\ s(X,Y).$

$MM_3 = MM_2 \cup \{ t(b), t(c), v(c,b) \}$

Therefore, the algorithm returns the model MM_3, i.e.:
$\{ p(a,b), p(b,c), r(a,b), r(b,c), r(a,c), s(b,b), s(c,c), t(b), t(c), v(c,b) \}$

In the first iteration we have to compute the minimal model of $P(S_1)$, but its first rule contains a negation, not p(Y,X). But it is not a problem, because we know everything about p, because it is extentional, so all the facts about p are already written in the program. In MM_0 we know what is true and false about p.
When we compute the immediate consequence w.r.t the facts in MM_0, in order to make true r(X,Y), p(X,Y)

must be true and also, it must be false that p(Y,X) is true (not p(Y,X) is true if the atom without the negation p(Y,X) is false).
After computing MM_1, we can consider r as an extentional predicate in the next iteration (in the next iteration we won't find r predicates, because they are all computed and established by MM_1, so all the other atoms for r are false -¿ Closed World Assumption).
In $P(S_2)$ and $P(S_3)$, negation is always in front of predicates of the previous strata.

⇒ The algo can use whatever procedure to compute the minimal model that has negation only in front of extentional predicates. We started from a program that has negation in front of both extentional and intentional predicates, and we end up with an algo that, thanks to stratification, iterates a construction of the model that uses negation only in front of extentional predicates (predicates of which we know exaclty what is positive and what is negative, so we are able deterministically to evaluate correctly negated atoms).

Negation and Recursion
Now we can have negation and recursion, but it cannot be in the definition of the predicate itself.

Example:
r(X,Y) :- p(X,Y), r(X,Z) .
s(X,Y) :- r(Z,X), r(Z,Y), not r(X,Y) .
This is ok!

Example:
r(X,Y) :- p(X,Y), not r(X, Z)
NO! Because this creates a negative edge from r to r and it makes the program not stratified.

Question
Are we loosing something with stratification obtaining just one answer set/ minimal model w.r.t compute all answer sets? We will see it later.

Chapter 5

Adding disjunction to Datalog

Disjunctive rules
A disjunctive rule is an expression of the form:
$\alpha_1 V ... V \alpha_k$:- $\beta_1, ..., \beta_n, not \lambda_1, ..., not \lambda_m$
where:
• n,m,k are non-negative integers
• every $\alpha_i, \beta_i, \lambda_i$ is an atom
• (safeness condition) every variable symbol occurring in the rule must appear in at least one of the positive atoms of the rule body $\beta_1, ..., \beta_n$.

The rule is called
textbfpositive if m=0, is called **non-disjunctive** if k=1, and is called constraint**constraint** if k=0.

A **disjunctive program** (also called **Answer Set Program**) is a set of disjunctive rules.

Now we have the or operators in the head of a rule: the informal meaning of this is that if the body of the rule is true, then the head of the rule is true; but now the head of the rule is not just one atom, it is a disjunction of atoms, so the conclusion is now non-deterministic (something that can be either this or that, we do not know what α_i atoms are true).

5.1 Semantics of disjunctive rules

A ground disjunctive rule $\alpha_1 V ... V \alpha_k$:- $\beta_1, ..., \beta_n, not \lambda_1, ..., not \lambda_m$. is **satisfied** in an interpretation I if at least one of the following conditions holds:
• at least one of the atoms $\beta_1, ..., \beta_n$ does not belong to I
• at least one of the atoms $\lambda_1, ..., \lambda_m$ belongs to I
• at least one of the atoms $\alpha_1 V ... V \alpha_k$ belongs to I

Example:
the rule q(a) V q(b) :- p (b, a) .
• is satisfied in the interpretations {}, {q(a)}, {q(b)}, {q(a) , q(b)}, {p (b, a) , q(a)}, {p (b, a) , q(b)}, {p (b, a) , q (a), q(b) }
• is not satisfied in the interpretation {p(b, a)}

5.2 Answer Set Semantics

(same notion of reduct and answer set as the ones given for non-disjunctive programs)

Let P be a disjunctive program and let I be an interpretation. The **reduct** of P with respect to Iis the

positive disjunctive ground program obtained as follows:
- Delete from ground(P) every rule R such that an atom $not\beta$ occurs in the body of R and β belongs to I;
- Delete from every rule R in ground(P) every atom $not\beta$ occurring in the body of R such that β does not belong to I.

Let P be a disjunctive program. An interpretation I is an **answer set** of P if I is a minimal model of the positive disjunctive program P/I.

Reduct of a disjunctive program: Example

Example: let P be the program
$r(a) \vee s(a) :- p(a), not\ q(a).$
$s(a) \vee t(a) :- p(a), not\ r(a).$
$p(a).$

1) Let I_1 be the interpretation $\{p(a)\}$. Then, P/I_1 is the positive disjunctive program
$r(a) \vee s(a) :- p(a).$
$s(a) \vee t(a) :- p(a).$
$p(a).$

It is easy to verify that the first two rules of P/I_1 are not satisfied in I_1, therefore I_1 is not a model (and consequently not a minimal model) for P/I_1, so I_1 is not an answer set of P.

Example (contd.): let P be the program
$r(a) \vee s(a) :- p(a), not\ q(a).$
$s(a) \vee t(a) :- p(a), not\ r(a).$
$p(a).$

2) Let I_2 be the interpretation $\{p(a), r(a)\}$. Then, P/I_2 is the positive program
$r(a) \vee s(a) :- p(a).$
$p(a).$

It is easy to verify that I_2 is a minimal model of P/I_2, therefore I_2 is an answer set of P.

Example (contd.): let P be the program
$r(a) \vee s(a) :- p(a), not\ q(a).$
$s(a) \vee t(a) :- p(a), not\ r(a).$
$p(a).$

3) Let I_3 be the interpretation $\{p(a), s(a)\}$. Then, P/I_3 is the positive disjunctive program
$r(a) \vee s(a) :- p(a).$
$s(a) \vee t(a) :- p(a).$
$p(a).$

It is easy to verify that I_3 is a minimal model of P/I_3, therefore I_3 is an answer set of P.

Example (contd.): let P be the program
$r(a) \vee s(a) :- p(a), not\ q(a).$
$s(a) \vee t(a) :- p(a), not\ r(a).$
$p(a).$

4) Let I_4 be the interpretation $\{p(a), r(a), s(a)\}$. Then, P/I_4 is the positive disjunctive program
$r(a) \vee s(a) :- p(a).$
$p(a).$

It is easy to verify that I_4 is a model of P/I_4, but is **not** a **minimal** model of P/I_4: the minimal models of P/I_4 are $\{p(a), r(a)\}$ and $\{p(a), s(a)\}$. Therefore, I_4 is not an answer set of P.

Chapter 6

SQL, DLs, Datalog and ASP: comparison

6.1 CWA vs OWA

- DLs are based on the semantics of classic logic, OWA:
  - The knowledge expressed by the theory is not complete (We are not assuming that the KB that represents the domain of interest is a complete specification of the domain, there may be something missing and not all the aspects of this domain are known)(if you cannot derive that A is a student, it does not means that A is not a student)
  - since the knowledge is not complete, there can be many possible worlds (models)
- Databases are based on CWA:
  - The knowledge expressed by the database is complete (if you cannot derive that A is a student, it means that A is not a student)
  - Only one world is possible (the one completely described by the database)
- Datalog and ASP inherit CWA from databases: CWA can be seen in minimality that is its formal realization.
- But ASP is able to deal with incomplete knowledge (disjuncton or recursive negation) (infact the result of this is that we can find multiple answer sets)

6.2 Complete vs Incomplete Knowledge

- DL knowledge bases are incomplete specification of the domain of interest
- The KB has multiple models (possible worlds in which the KB is satisfied)
- The TBox and the ABox can be used to infer implicit facts about the world:
  - Example:
    ABox = { student(Bob)} and TBox = { subClassOf(student, person)}
    We can conclude that person(Bob) is true (which is not stated in the box but is a logical consequence) because the TBox says that every instance of student must be an instance of person
    → The fact that we can infer something that is not written explicitly in our dataset is an important aspect of incomplete knowledge
- Conversely, the knowledge expressed by a relational database is assumed to be complete. In databases what happended in the previous example cannot happen: if we have a table Person and a table Student, we will have a complete list of people and students, we cannot have some person in the database that is not written in the person table and infer that it is a person.

- Only one world is possible (the one completely described by the database)
- In relational DBs, the database instance is a complete specification:
  - Schema constraints in databases are integrity constraints: they MUST be satisfied by the database instance (otherwise the database is in an illegal state)(if we have Bob in the student table and we don't have it in the person table and we have a foreign key that says that every student must be person, then we do not generate a consistent database becuase there is a violation of the integrity constraints)
  - For this reason, such constraints cannot derive new facts (not appearing in the database instance)

6.3 Recursion

- DLs are able to deal with recursive statements - E.g., cyclic definitions of concepts in the TBox (es. every person has a parent who is a person).
  DLs do not allow for recursive queries. - PARTIALLY SUPPORTED
- SQL does not allow for recursive queries - only a limited form of recursion in views is allowed (limited from both the theoretical and pratical point of views) - NOT SUPPORTED
- Datalog allows for expressing recursive queries over databases - SUPPORTED
- ASP allows for expressing recursive queries over databases - SUPPORTED

6.4 Negation

- Several DLs allow for expressing negation in the TBox (and in the ABox) and in queries.
  But, DLs do not allow for recursive negation in queries (we will learn how to write queries for DL later)
- SQL allows for negation in queries, but does not allow for recursive negation
- Datalog does not allow for expressing negation (since are positive)
- ASP allows for expressing negation, and even recursive negation

6.5 Value Invention (skolemization)

- DLs allow for expressing so-called value invention (or mandatory participation to roles)
  Example: inclusion axiom Person$\subseteq \exists$hasMother.T (there is some element of the domain that partecipates together with the element of person in the role hasMother and we cannot identify it. We are forcing some other element of the domain to partecipate to the role, the "mother" in this case. Also, the mother can be a different mother in every model)
- The above axiom cannot be represented in Datalog
  Example: the rule hasMother(X,Y) :- Person(X) is not allowed in Datalog (is not range-restricted) because the variable Y is not safe since it appears in the head but not in the body... There is not a way to represent the first rule in Datalog.
  Datalog does not allow for talking about elements outside the Herbrand Universe of the program (namely, the constants appearing in the program)
- Same restriction holds for ASP (and SQL too)
- Value invention is a form of incomplete knowledge

6.6 Existential rules

To overcome the above limitation, an extension of Datalog called existential rules, or Datalog +/-, has been proposed.
Existential rules do not have the range restriction on variables: a variable can appear in the head of the rule. Such variables are interpreted as existentially quantified variables
• Es: hasMother(x,y) :- Person(x). is an existential rule, whose meaning is: if x is a person, then there exists y such that x has mother y. In this way, value invention can be expressed by rules.
Extending Datalog with existential variables in the head makes the language more expressive, but on the other hand, such an extension makes reasoning harder: in general, reasoning with existential rules is **undecidable** (only some classes are decidable).
Recent studies have defined restricted classes of existential rules in which reasoning is decidable (and even tractable in data complexity).
Interesting relationship between some of these classes and Description Logics: in this way they are very close.

6.7 Comparison: expressiveness

	Semantics	Recursion	NOT (stratif.)	NOT (recursive)	OR	Value invention
SQL	CWA	no	yes	no	yes	no
DL	OWA	yes/no	yes	no	yes	yes
Positive Datalog	CWA	yes	no	no	no	no
Datalog + stratified negation	CWA	yes	yes	no	no	no
Datalog + negation	CWA	yes	yes	yes	no	no
ASP	CWA	yes	yes	yes	yes	no

6.8 Complexity of SQL

- Complexity of evaluating queries in SQL is NP-complete
- A query may have an exponential number of answers
- Example:

Table T:

A
0
1

Query:

```
select R1.A, R2.A, R3.A,…,Rn.A
from T AS R1, T AS R2, T AS R3, …, T AS Rn
```

Answers: (2^n tuples)

R1.A	R2.A	R3.A	…	Rn-1.A	Rn.A
0	0	0	…	0	0
0	0	0	…	0	1
…	…	…	…	…	…
1	1	1	…	1	0
1	1	1	…	1	1

Can we write the same query in Datalog?

Previous SQL query:

```
select R1.A, R2.A, R3.A,…,Rn.A
from T AS R1, T AS R2, T AS R3, …, T AS Rn
```

Same query expressed in Datalog:

```
Q(X1, X2, …,Xn) :- T(X1), T(X2), …, T(Xn).
```

(which shows that Datalog can build IDB predicates with an exponential number of tuples in the minimal model)

head → projection (select) in SQL
body → cartesian product

6.9 Comparison: complexity

	Reasoning task	Complexity
SQL	Query evaluation	NP-complete
ALC, unfoldable TBoxes	Concept consistency, KB satisfiability, instance checking	PSPACE-complete
ALC, GCIs	Concept consistency, KB satisfiability, instance checking	EXPTIME-complete

	Reasoning task	Complexity
Ground positive Datalog	Building the minimal model	PTIME-complete
Ground Datalog with stratified negation	Building the minimal model	PTIME-complete
Ground ASP (no disjunction)	Consistency, brave reasoning	NP-complete
Ground ASP	Consistency, brave reasoning	NP^{NP}-complete

	Reasoning task	Complexity
Positive Datalog	Building the minimal model	EXPTIME-complete
Datalog with stratified negation	Building the minimal model	EXPTIME-complete
ASP (no disjunction)	Consistency, brave reasoning	NEXPTIME-complete
ASP	Consistency, brave reasoning	$NEXPTIME^{NP}$-complete

Now there are tools that have very good performances, even if from the theoretical point of view the worst case complexity is very bad, but there are method to avoid exponential running time.