Knowledge-Based Systems and Deductive Databases

Wolf-Tilo Balke
Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
5.1 Complexity of Logic
5.2 The Datalog Language
5.3 Datalog Programs

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y) . \\
\text{ancestor}(X, Y) \leftarrow \text{ancestor}(X, Z), \text{ancestor}(Z, Y) .
\]
5.1 Complexity of Logic

• The last lectures dealt with basics of first order logics
  – We showed how to write syntactically correct logical statements
  – We discussed interpretations and models
  – We showed how to deduce and to prove statements
  – We heard some stuff about history of logics

• So, we get closer to building a deductive DB
  – Essentially, a deductive DB will later check if a given statement can be followed given a set of facts and rules
    • $\mathcal{W} \models \mathcal{W}$, i.e. $\mathcal{W} \cup \{\mathcal{W}\}$ satisfiable
• Now, it’s time to have a look at the computational complexity of logics
  – The check for validity and the check for satisfiability is especially important
  – A database is about performance
  – If it turns out that the anticipated complexity is prohibitive, we are in deep trouble
    • Will some restrictions save the day?
• First, let’s have a look on plain **Boolean logic**
  – i.e. no predicates, no quantifiers, universe is limited to \{true, false\}
    • e.g. \( W_1 \equiv x \lor \neg x \), \( W_2 \equiv (x \lor \neg y) \land \neg x \)
  – Like first order logic Boolean statements can also be valid, satisfiable, or unsatisfiable
So, how do you test whether some **Boolean** statement $W$ is satisfiable, valid, or unsatisfiable

- This is commonly known as the **SAT** problem

- **Unsatisfiable:**
  - Check if $W$ is satisfiable; if not, it is unsatisfiable

- **Valid:**
  - Check if $\neg W$ is unsatisfiable; if not, it is valid

- **Satisfiable:**
  - Generate a **substitution** for all variables in $W$
  - **Evaluate** the substituted expression
• Unfortunately, SAT is in \textbf{NP}
  
  \textbf{Deterministic decidable algorithm}
  
  • Generate all $2^n$ substitutions
  • Evaluate substituted expression for each substitution
    \begin{itemize}
    \item In $\mathcal{O}(n^2)$ each
    \end{itemize}
  • Overall, in $\mathcal{O}(n^2 \ 2^n)$

  \textbf{Non-Deterministic semi-decidable algorithm}
  
  • Guess any substitution
  • Evaluate substituted expression in $\mathcal{O}(n^2)$
  • Continue until you find a working substitution

  \textbf{NP} is a pretty bad property for an algorithm…
5.1 Complexity of Logic

• **Example:** Is \( W \) satisfiable?

  – \( W_1 = (x \lor \neg y) \land \neg x \)
    
    • Yes, for \( x = y = \text{false} \)

  – \( W_2 =
    
    ((x \lor y \lor z) \land
    (x \lor \neg y) \land (y \lor \neg z) \land (z \lor \neg x) \land
    (\neg x \lor \neg y \lor \neg z))
    
    • Nope, unsatisfiable
• **Example:** Is $W$ satisfiable?
  
  – You could also try to construct the substitution which satisfies the expression (or show that there is none)
    
    • For general formulas, this is very difficult to be done automatically

  – $W_2 =$
    
    $$((x \lor y \lor z) \land (x \lor \neg y) \land (y \lor \neg z) \land (z \lor \neg x) \land (\neg x \lor \neg y \lor \neg z))$$

    • One false, one true, all need to be the same? Not possible
      
      → unsatisfiable
• The default algorithm for solving the SAT problem is the **Davis-Putnam algorithm**
  – Solves the problem of satisfiability for a Boolean formula in **conjunctive normal form**
  – Complexity is somewhere around $\mathcal{O}(1.8^n)$...
  – Basic idea: Build a pruned tree of possible substitutions

No Funny Logician Hat?
There are several variants of the SAT problem:

- **3-SAT**: This is the problem for Boolean satisfiability in conjunctive normal form, where each clause contains at most 3 literals
  - 3-SAT was the *first problem* that was ever shown to be *NP-complete*
  - Normal SAT *reduces polynomial* to 3-SAT
- **Horn-SAT**: This restricts SAT to formulas in conjunctive normal form, where each clause is a Horn clause
Horn-SAT is very important as it is in P-complete

Example Horn-SAT problem:

\[ \mathcal{W} := \{\neg x_2 \lor \neg x_1 \lor x_3, \neg x_1 \lor \neg x_2 \lor \neg x_3 \lor \neg x_4, (x_1), (x_2)\} \]

This results in (implicative form):

- Facts: true→ x_1, true→ x_2
- Definites: (x_2 ∧ x_1 → x_3)
- Goals: (x_1 ∧ x_2 ∧ x_3 ∧ x_4 → false)

Whole set is satisfiable, if conjunction of implications is true

Idea: find all those variables which have to be true and look for any contradiction!
• Find all those variables $T$ which have to be true!

  – **Init** $T:=\emptyset$ (i.e. all variables are false)

  – **Pick** any unsatisfied implication $H_i$ (facts or definites)
    
    - $H_i \equiv (x_1 \land \ldots \land x_n) \rightarrow y$
    - unsatisfied implication: all $x_i$ are true, $y$ is false
    - Add $y$ to $T$ (thus $H_i$ is satisfied now)
    - Repeat until there are no unsatisfied implications

  – $\mathcal{W}$ is satisfiable, iff $T$ satisfies all clauses
    
    - Furthermore, $T$ is the **minimal set** of variables which satisfies $\mathcal{W}$, i.e. for each satisfying substitution $T'$ holds, $T \subseteq T'$

### Table

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \rightarrow x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
• Example **Horn-SAT** problem:
  - $H_1 \equiv (\text{true} \rightarrow x_1)$, $H_2 \equiv (\text{true} \rightarrow x_2)$,
    $H_3 \equiv (x_1 \land x_2 \land x_3 \land x_4 \rightarrow \text{false})$, $H_4 \equiv (x_2 \land x_1 \rightarrow x_3)$

• **Algorithm**
  - $(\text{true} \rightarrow x_1) \quad \Rightarrow \quad T:=\{x_1\}$
  - $(\text{true} \rightarrow x_2) \quad \Rightarrow \quad T:=\{x_1, x_2\}$
  - $x_2 \land x_1 \rightarrow x_3 \quad \Rightarrow \quad T:=\{x_1, x_2, x_3\}$
  - Does $T$ satisfy all clauses?
    - It obviously satisfies $H_1$, $H_2$, and $H_4$
    - It also satisfies $H_3$
    - $T$ satisfies $\mathcal{W}$!
  - If there was also an $H_5 \equiv (x_2 \land x_3 \rightarrow \text{false})$, $\mathcal{W}$ would be unsatisfiable
• So, let's switch to general first order logic. What changes with respect to complexity?
  – Universe of potentially unlimited size
  – Quantifiers
    • A given sub-formula has to be true for all / some elements of the universe

• How does this affect our complexity?
  – As an example, we will use the popular axiomatization of the number theory
• Number theory
  – \( \mathcal{L}_{\text{NT}} = (\Gamma, \Omega, \Pi, X) = (\{0\}, \{\sigma, +, \times, \uparrow\}, \{=, <\}, \{x, y, z\}) \)
  – There is just the constant 0
  – The \( \sigma \) function represents the successor function
    • i.e. \( \sigma(\sigma(\sigma(0))) = 3 \)
    • As using the successor function is very unhandy, we employ a shortcut notation for all natural numbers
      – We may, e.g. use \( 3451 \) instead of \( \sigma(\sigma(\sigma(\sigma(\sigma(\ldots \sigma(0) \ldots)))))) \)
  – The functions \( +, \times, \text{ and } \uparrow \) represent addition, multiplication, and exponentiation
  – The predicates \( =, \text{ and } < \) represent equality and the less-than-predicate
So, how can we (naively) evaluate first order logic?

– Generate all substitutions for quantified sub-formulas and evaluate the main formula

– \( \forall x \ ( x > 5) \)
  • \( x=0 \); Bang. Untrue.

– \( \exists x \ ( x > 5) \)
  • \( x=0; x=1; x=2; \ldots; x=6 \); Ok. True.

– \( \forall x, y \ (3>x \ V 3>y \ V (x \times y > x + y)) \)
  • \( x=0, y=0 ; x=1, y=0; x=0, y=1; x=1, y=1; x=2, y=0, \ldots \)
  So, this seems to be true? Where do we stop?
• Testing all substitutions for universes with unlimited size is kind of tricky

• **Alternative Idea:**
  
  – Use *deductive systems* to construct a *proof* from a set of valid axioms to the questionable statement
  
  – **Number theory** has been axiomized several times on different styles
  
  – Most popular: *Peano arithmetic*
    
    • Commonly 15 axiom types inducing countable unlimited number of axioms
    
    • Introduced by Italian mathematician Giuseppe Peano in 1889
But consider this:

- \( \neg \exists n, x, y, z ((a^n + b^n = c^n) \land n \geq 2) \)
- This is Fermat's Last Theorem
- It took 357 years to show that this statement is provable for the natural numbers
- 1637-1995
- Proof did some really nasty tricks...
- So, we seem to be in severe trouble here...
5.1 Complexity of Logic

• So, where is the problem?

  – Remember Gödel’s Incompleteness Theorem
    • “Any effectively generated theory capable of expressing elementary arithmetic cannot be both consistent and complete. In particular, for any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory.”

  – Thus, for any non-trivial deductive system NT, there are statements which cannot be proofed within that system
    • Unprooﬁeable statements are called undecidable
    • NT ⊢ W : W can be proven in the system NT
      \[ \not\vdash W \]
      \[ \text{decidable} \]
      \[ \text{NT} \vdash W \]
      \[ W \]
      \[ \neg W \]
      \[ \text{undecidable} \]
      \[ \text{decidable} \]
      \[ \text{NT} \vdash \neg W \]
      \[ \not\vdash \neg W \]
Example: The **Goodstein Theorem** and the **Paris-Kirby Theorem**

- **Goodstein’s Theorem**: “Imagine *Hercules* fighting the *Hydra*, chopping off one if its head after the other. But every time a head is chopped off, the Hydra **regrows** a finite number of heads (according to the **Goodstein sequence**). Still, Hercules will eventually **defeat** the Hydra as long as he does not give up.”
…the actual theorem is not important for us. But note that the theorem is indeed **expressible** and **true** within the Peano arithmetic

- i.e. all Goodstein sequences are finite regardless of their start value.
BUT: Paris-Kirby Theorem:
“The Goodstein Theorem is not decidable.”

- i.e. there is no way to prove Goodstein within Peano arithmetic
  - Actually, there is no way to prove it at all using first order logic
- Proof Sketch: “Show that the consistency of Peano arithmetic directly follows from the Goodstein theorem. If Goodstein was provable within Peano, the consistency of Peano was shown within itself. This is not possible according to Gödel’s incompleteness theorem.”

- This is pretty bad. Obviously, we need some restrictions…
  - …but first, we move to some algorithms
5.1 Herbrand Theorem

• Jacques Herbrand
  – Born 1908 in Paris, finished his Doctorate degree 1929 at the Sorbonne in Paris
  – In early 1931, he got a fellowship at the university of Berlin, worked there with John von Neumann
  – Later moved to Göttingen to work with Emmy Noether
  – There, started his signature work “On the consistency of arithmetic”
    • Before finishing, died during a hiking trip in the Alps in July 1931 at age 23
5.1 Herbrand Theorem

• The **Herbrand theorem** (1928)
  – Informal: “A is a closed formula in *universal prenex form*. Then A is *unsatisfiable* if and only if there is a *finite subset* of its *Herbrand expansion* which is *Boolean unsatisfiable*”
    • Note that Herbrand himself messed the proof up, the flaw was discovered in 1960 by Dreben…
    • Today, proving is quite easy if the Compactness theorem is used
  • Universal prenex form:
    \[ A ≡ ∀ y_1, \ldots, y_n \ F(y_1, \ldots, y_n) \] and F being quantifier-free.
  – Very important: The problem of **first order logics unsatisfiability** is transformed to a **Boolean unsatisfiability problem**
5.1 Herbrand Theorem

• **Herbrand Theorem** (more formally)
  
  – Let be \( A \) is in universal prenex form \( A \equiv \forall y_1, \ldots, y_n F(y_1, \ldots, y_n) \). Then \( A \) is **unsatisfiable** if and only if there is a finite set \( TA \) of **ground terms** \( t_{ij} \) with \( 1 \leq i \leq k \) and \( 1 \leq i \leq n \) such that \( TA := \{ F(t_{11}, \ldots, t_{1n}), \ldots, F(t_{k1}, \ldots, t_{kn}) \} \) is unsatisfiable

  • \( F(t_{i1}, \ldots, t_{in}) \) are called **ground instances** of \( A \)
  
  • Set of possible ground instances is potentially of unlimited size (e.g. Herbrand base)
  
  • The set of all possible ground instances is called **Herbrand expansion** \( E(A) \)

  • i.e. for a set of terms \( TA \) holds \( TA \subseteq E(A) \) and \( TA \) finite

  – The Herbrand theorem can **equivalently** be stated for existentially quantified or mixed closed formulas

  • Transformation rules for \( \forall \) and \( \exists \) (Herbrandization)!
• Some considerations
  – TA can be checked for unsatisfiability in finite time
    • e.g. Davis-Putnam algorithm, etc
  – However, it is not known which TA will show the unsatisfiability
    • There is potentially an unlimited number of TA ⊆ E(A)
  – If you did not find an unsatisfiable TA yet, this either means
    a) There are none and thus A is satisfiable
    b) You have not looked long enough
    • You cannot know which of both are true (reduces to the Halting problem…)

Knowledge-Based Systems and Deductive Databases – Wolf-Tilo Balke – IfIS – TU Braunschweig
This lead’s to a simple meta-algorithm for checking unsatisfiability (Gilmore algorithm,)

**Preparation:**
- Take any first-order-logic formula \( A \)
- Transform \( A \) into universal prenex form \( A' := \forall y_1, \ldots, y_n F(y_1, \ldots, y_n) \)
  - i.e. pull all quantifiers to the front and transform to universal quantifiers
- Be able to generate a the **Herbrand expansion** \( E(A') = \{A_1, A_2, \ldots\} \)

**Gilmore Algorithm**
- \( k := 1 \)
- **While** \( \bigwedge_{i=1}^{k} A_i \) is satisfiable (or: “while not unsatisfiable”)
  - \( k++ \)
- **Return** ”\( A \) is unsatisfiable”
• Thus, the **Gilmore algorithm** is semi-decidable
  
  – Answers only if A is unsatisfiable, else is caught in endless loop
• **Restriction 1:** Allow only a **decidable subset** of first order formulas
  
  – One such subset of first order logics are the so-called **Schönfinkel-Bernays** expressions:

  • Given a language **without functional** symbols and **without the equality predicate**
  
  • Given expressions in **prenex form**
  
  \[ W \equiv \exists x_1, \ldots, x_n \forall y_1, \ldots, y_n \quad W_2 \]  
  
  with \( W_2 \) is quantifier free
  
  • Then it is **decidable** if \( W \) has a model or not
    
    – SB-SAT problem
5.1 Complexity of Logic

• Easy proof:
  – Without functional symbols, the Herbrand base is finite
  – If the Herbrand base is finite, the Herbrand expansion is finite
  – If the Herbrand expansion is finite, you can generate all subsets of the expansion in finite time (and which are also finite)
  – Each check for unsatisfiability for a finite set of ground instances is in finite time
• However, there is a catch: \( SB\text{-SAT} \in NEXP \)
  – NEXP: The class of all non-deterministic exponential algorithms
  – What does that mean:
    • You can only guess the solution AND then you need an exponential amount of time to check if your guess was correct….
    • \( O(2^{p(n)}) \) using a non-deterministic Turing machine and unlimited space.
    • Or, you could **unfold** the problem to an deterministic machine which takes even longer….
  – This is obviously a very very bad complexity class…
Additionally, Schönfinkel-Bernays severely restricts the expressiveness of logics

– No functions!

• This is even bad in the case where you actually can avoid functions as many predicates could be implemented more efficiently as functions
• So… how do we solve all this?
  – We need a subset of first order logic which has guaranteed finite Herbrand expansions
  – We should try to find subset which is in a better complexity class than NEXP
  – We should find a subset which does not limit the expressiveness too much

• Approach: Restrict to first order logics allowing only for Horn clauses and non-recursive typed functions
  – Ground instances are thus Horn clauses
  – Check for unsatisfiability of finite subsets of Herbrand expansion is in P
  – Herbrand expansions is finite as the Herbrand universe is finite
5.2 Logic as Data Model

• Relational databases distinguish between DDL and DML
  – **DDL** is for creating schemas or views
  – **DML** is for maintaining data and queries
    • Evaluation follows the (tuple) relational calculus

• In deductive databases both data and queries are specified by formulae
Every predicate is written by **Horn clauses** of the form

- \( \forall (L_1 \lor L_2 \lor ... \lor L_n), \ L_i \in L_L \)
- With atomic formulae \( L_i \) and at most one positive literal \( L_j \)

**Logic programming** introduced a slightly different notation of Horn clauses for simplicity

- \( L_j \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n \).
- That means ‘\( \leftarrow \)’ is understood as **implication**, ‘,’ as **conjunction**, and ‘.’ denotes the **end** of a clause
5.2 Datalog

• A **deductive database** consists of facts and rules
  
  – The set of facts is called **extensional database** (EDB)
    • If no functions are used in the facts, it can be stored as a simple relational database table
  
  – The set of rules is called **intentional database** (IDB)
    • The reflects the idea of views in relational databases, but allows for recursion
5.2 Datalog

- **Datalog** is a query and rule language specifically defined for deductive databases
  - Syntactically similar to Prolog
  - Introduced around 1978 for academic database research by Hervé Gallaire and Jack Minker
  - Used as the main foundation for expert systems theory during the 1980ies
5.2 Datalog Syntax

• A database clause (DB-clause) is defined as
  - \( A \leftarrow L_1, \ldots, L_n \) with an atomic formula \( A \in A_\mathcal{L} \) and literals \( L_i \in L_\mathcal{L} \)
    • \( A \) is referred to as head and \( L_1, \ldots, L_n \) as body (or body literals) of the DB-clause
    • often written as \( A :\text{-} L_1, \ldots, L_n \)
  - DB-clauses with \( n > 0 \) are called rules
  - DB-clauses with \( n = 0 \) and an atomic ground formula \( A \) are called facts
  - A DB-clause with only atomic body literals is called definite
5.2 Datalog Syntax

- **Example facts**
  - parent(John, Mary).
  - parent(John, Thomas).
  - parent(Thomas, George).
  - ...

- **Example rules**
  - grandparent(X,Y) ← parent(X,Z), parent(Z,Y).
  - mary’s_love(X) ← parent(Y, Mary), parent(Y,X).
  - ...

Knowledge-Based Systems and Deductive Databases – Wolf-Tilo Balke – IfIS – TU Braunschweig
• The most important feature of Datalog is the possibility to use **recursion**

- edge(3,2).
  edge(2,6).
  edge(2,5).
  edge(5,3).
- path(X,Y) ← edge(X,Y).
  path(X,Y) ← edge(X,Z), path(Z,Y).

- Alternative ways for writing the last rule are:
  path(X,Y) ← path(X,Z), edge(Z,Y).
  or  path(X,Y) ← path(X,Z), path(Z,Y).
5.2 Datalog Syntax

- The **definition** `def(p)` of a predicate symbol `p` is the set of facts/rules in the Datalog program, where `p` occurs in the **head**
  
  - `grandmother(X,Y) ← parent(X,Z), parent(Z,Y), female(Y).`
  
  - `path(X,Y) ← edge(X,Y).`
    `path(X,Y) ← edge(X,Z), path(Z,Y).`

- If a definition does not at all depend on some variable in a body literal, it is often written as ‘_’ (don’t care)
  
  - `p(X, Y) ← r(X, Z), q(Y, Z, _).`
Problem of variables in heads of rules

Consider a rule $p(X) \leftarrow r(Y)$.  
What does it mean?

- If there is a substitution for $Y$ making $r(Y)$ true, then $p(X)$ is true for all possible substitutions for $X$?
- ... if $r(Y)$ is true for all possible substitutions of $Y$, then $p(X)$ is true??
- ... or only for $Y=X$??

Restriction: In Datalog all variables used in a head predicate *always* have to occur in some body literal, too

- Similar problem arises, if a constant in the head would depend on variables in body literals $p(a) \leftarrow r(X)$.
### 5.2 Datalog Syntax

- **A database query** is defined as
  - \(? L_1, ..., L_n\). with literals \(L_i \in L_L\), \(n > 0\)
  - Alternative notation \(\leftarrow L_1, ..., L_n\). or \(:- L_1, ..., L_n\).
  - A query with only **atomic literals** is called **definite**
  - A definite query with \(n = 1\) is called a **Datalog query**

- **Why is this a query?**
  - A set of DB-clauses \(\mathcal{W}\) and a query \(Q \equiv L_1, ..., L_n\) are unsatisfiable, iff \(\mathcal{W} \models \neg Q\) with \(\neg Q \equiv \exists (L_1 \land ... \land L_n)\)
5.2 Datalog Syntax

- **Example database**
  - parent(John, Mary). parent(John, Thomas). ...
  - female(Mary). female(Sonja). ...
  - grandmother(X,Y) ← parent(X,Z), parent(Z,Y), female(Y).

- **Example datalog query**
  - Who is John’s grandmother?
  - ?grandmother(John,X).
    - grandmother(John, Sonja).
    - grandmother(John, Karen).
For simplicity often some often used arithmetic predicates like \{<, \leq, \geq, >, =, \neq\} are predefined for the use in body literals of rules.

- Example:
  \[
  \text{digit}(X) \leftarrow \text{naturalnumber}(X), X \leq 9.
  \]
  \[
  \text{smaller}(X,Y) \leftarrow \text{number}(X), \text{number} (Y), X < Y.
  \]

The same holds for simple arithmetic functions like \{+, -, *, /\}

- \[
  \text{sum}(X,Y, Z) \leftarrow Z=X+Y
  \]
• If predicate symbols defining facts never occur in the head of any rule, a set of DB-clauses is called a Datalog\textsuperscript{f,\textneg}-program
  – This name follows the idea of logic programming
  – There are different kinds of programs…
Depending on the use of **functions** and **negation** several Datalog language classes can be distinguished

- **Datalog$^{\text{neg}}$** programs do not contain function symbols
- **Datalog$^{\text{f}}$** programs (or definite programs) do not contain negative literals
- **Datalog** programs contain neither negative literals nor function symbols
5.3 Datalog Language Classes

- Expressiveness

Datalog$^f$ \( \subseteq \) Datalog$^{f, \text{neg}}$ \( \subseteq \) Datalog$^{\text{neg}}$ \( \subseteq \) Datalog

Knowledge-Based Systems and Deductive Databases – Wolf-Tilo Balke – IfIS – TU Braunschweig
• Datalog programs can also be distinguished by their dependencies between predicates
  – We have seen already that negation in literals may sometimes lead to strange results…
    • Remember: closed world assumption
  – Idea: find out about the relation between different predicates by examining their respective definitions
5.3 Program Classes

• The program connection graph (PCG) of some program $P$ consists of
  – **Nodes** for each predicate symbol $p$ in $P$
  – **Directed edges** from node $p$ to node $q$, if $q$ is in the definition of $p$
  – An edge is **negative**, if $q$ occurs in a negated literal, otherwise the edge is **positive**

• A **recursive clique** is a maximum subset of the predicates in $P$, such that between each two predicate symbols there is a path in the PCG
5.3 Program Classes

- A program is called **hierarchic**, if the PCG does not contain cycles
  - If there are cycles the program is called **recursive**

  - `bachelor(X) ← male(X), ¬ married(X)`. is **hierarchic**
5.3 Program Classes

- \( \text{path}(X, Z) \leftarrow \text{edge}(X, Y), \text{path}(Y, Z) \). is recursive

- \( \text{p}(X,Y) \leftarrow \text{q}(Y, Z), \text{s}(Z) \).
  \( \text{q}(X,Y) \leftarrow \text{r}(Y), \text{s}(X) \).
  \( \text{r}(X) \leftarrow \text{p}(X, X) \). is also recursive

Knowledge-Based Systems and Deductive Databases – Wolf-Tilo Balke – IfIS – TU Braunschweig
A program is called **stratified**, if cycles in the PCG only consist of **positive edges**

- goodpath(X, Y) ← path(X, Y), ¬ toll(X).
- goodpath(X, Z) ← goodpath(X, Y), goodpath(Y, Z).

is a **stratified** and **recursive** program.
– even(X) ← number(X), ¬ odd(X).
odd(X) ← number(X), ¬ even(X).

is a not stratified and recursive program
5.3 Stratification

- A *stratification* of some program $P$ is a disjoint partitioning $P = P_1 \cup \ldots \cup P_n$ of $P$ into program parts (strata) such that
  
  - The definition of each predicate symbol is a subset of some stratum
  
  - The definition of a predicate symbol in a *positive* body literal of a DB-clause in $P_i$ is part of a $P_j$ with $j \leq i$
  
  - The definition of a predicate symbol in a *negative* body literal of a DB-clause in $P_i$ is part of a $P_j$ with $j < i$
5.3 Stratification

• Basic idea: **layer the program** such that definitions of negatively used predicates are always already given in previous layers
  
  – This effectively excludes the use of negation within recursion

• It can be proved that a program is **stratified**, if and only if it has a **stratification**
5.3 Stratification

• **Stratification Algorithm**
  – Takes a Datalog\(^{f,neg}\) program as input and outputs either the stratification or ‘not stratified’
  – Thus, the problem of stratification is **syntactically decidable**

• **Initialization:**
  – For each predicate symbol \(p\) do \(\text{stratum}[p] := 1\)
  – \(\text{maxstratum} := 1\)
5.3 Stratification

• Main loop:
  – Repeat
    for each **DB-clause** with head predicate p do
      for each **negative body literal** with predicate q do
        \[ \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q] + 1) \]
      for each **positive body literal** with predicate q do
        \[ \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q]) \]
    \[ \text{maxstratum} := \max(\{\text{stratum}[p] | p \text{ is predicate}\}) \]
  until \[ \text{maxstratum} > \# \text{predicates} \]
  or the **stratum function becomes stable**
5.3 Stratification

• Output:

  – If \( \text{maxstratum} > \# \text{predicates} \)
    then return ‘not stratified’
  else for \( i := 1 \) to \( \text{maxstratum} \) do
    \[
P_i := \bigcup_{p \in \text{i-th stratum}} \text{def}(p)
    \]
5.3 Stratification

• Example

- \text{goodpath}(X, Y) \leftarrow \text{path}(X, Y), \neg \text{toll}(X).
- \text{goodpath}(X, Z) \leftarrow \text{goodpath}(X, Y), \text{goodpath}(Y, Z).

- Initialization:
  \text{stratum}[\text{goodpath}] = \text{stratum}[\text{path}] = \text{stratum}[\text{toll}]
  := 1
  \text{maxstratum} := 1
5.3 Stratification

• Example

- goodpath(X, Y) ← path(X, Y), ¬ toll(X).
  goodpath(X, Z) ← goodpath(X, Y), goodpath(Y, Z).

- **First loop** (maxstratum = 1):
  first rule: stratum[goodpath] :=
  \[\max(\text{stratum}[\text{goodpath}], \text{stratum}[\text{path}]) = 1\]
  stratum[goodpath] :=
  \[\max(\text{stratum}[\text{goodpath}], \text{stratum}[	ext{toll}] + 1) = 2\]
  second rule: stratum[goodpath] :=
  \[\max(\text{stratum}[\text{goodpath}], \text{stratum}[\text{goodpath}]) = 2\]
  maxstratum := stratum[goodpath] = 2
5.3 Stratification

• Example

– goodpath(X, Y) ← path(X, Y), ¬ toll(X).
  goodpath(X, Z) ← goodpath(X, Y), goodpath(Y, Z).

– **Second loop** (maxstratum = 2):
  results in no more changes to the strata and the algorithm terminates with maxstratum < 3

– Hence the program is **stratified** and

  P1 := \{def(path), def(toll)\}
  P2 := \{def(goodpath)\}
• How about a **not stratified** program?
  
  – `even(X) ← number(X), ¬ odd(X).`
  
  `odd(X) ← number(X), ¬ even(X).`
  
• The loop will increase the strata of `even` and `odd` until `maxstratum > # predicates` (**=3**)

  – **First loop:**
    
    `stratum[even] :=`
    
    `max(stratum[even], stratum[odd]+1) (**=2**)`
    
    `stratum[odd] :=`
    
    `max(stratum[even], stratum[odd]+1) (**=3**)`

  – **Second loop:**
    
    `stratum[even] :=`
    
    `max(stratum[even], stratum[odd]+1) (**=4**)`
5.3 DES

• In this detour, we will show an example implementation of Datalog
  – DES: Datalog Educational System
    • http://des.sourceforge.net
• Write some Datalog program in a text file editor

father(herrick, jenny).
mother(dorthy, jenny).
father(brant, herrick).
mother(angela, herrick).
father(louis, dorthy).
mother(jud, dorthy).
father(hubert, thekla).
mother(jenny, thekla).

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
ancestor(X, Y, D) :- parent(X, Y), D is 1.
ancestor(X, Y, D) :- ancestor(X, Z, D1), ancestor(Z, Y, D2), D is D1 + D2.
• Load (consult the program) the program
  – consult command
  – Filename relative to DES installation directory
5.3 DES

- Review the program
  - listing command

```
ancestor(X,Y,D) :-
  ancestor(X,Z,D1),
  ancestor(Z,Y,D2),
  D is D1 + D2.
ancestor(X,Y,D) :-
  parent(X,Y),
  D is 1.
father(brant,herrick).
father(herrick,jenny).
father(hubert,thekla).
father(louis,dorthy).
mother(angela,herrick).
mother(dorthy,jenny).
mother(jenny,thekla).
mother(jud,dorthy).
parent(X,Y) :-
  mother(X,Y).
parent(X,Y) :-
  father(X,Y).
```

Knowledge-Based Systems and Deductive Databases – Wolf-Tilo Balke – IfIS – TU Braunschweig 68
5.3 DES

• Run some queries

List all ancestors of Jenny with a distance of 2 (grandparents)

Is Thekla an ancestor of Jenny?
• Add some new rules from the shell

New rule with assert command

Who is Jenny’s grandmother?
• Semantics of Datalog
• Evaluation