Deductive Databases and Logic Programming

(Winter 2007/2008)

Chapter 8: Negation

- Motivation, Differences to Logical Negation
- Syntax, Supported Models, Clark's Completion
- Stratification, Perfect Model
- Bottom-Up Computation
- Conditional Facts, Well-Founded Model
- Stable Models

Objectives

After completing this chapter, you should be able to:

- explain the difference between negation in logic programming and negation in classical logic
- explain why stratification is helpful, check a given program for the stratification condition.
- compute supported, perfect, well-founded, and stable models of a given program.
- explain the well-founded semantics with conditional facts and elementary program transformations.

Overview

Example (Small Library)

book

BID	Author	Title
Ul189	Ullman	Princ. of DBS and KBS
Llo87	Lloyd	Found. of Logic Progr.

borrowed

BID
Ul189

$$\texttt{available(Author, Title)} \leftarrow \texttt{book(BID, Author, Title)} \land \texttt{not borrowed(BID)}.$$

available

Lloyd	Found. of Logic Progr.

Motivation (1)

- Queries or view definitions as in the above example are possible in SQL, but cannot be expressed with definite Horn clauses (classical Datalog).
  - ◇ A good query language should be relationally complete, i.e. it should be possible to translate every relational algebra expression into that language.
  - ◇ This goal is reached for Datalog with negation (Datalog^{neg}) (even without recursion).
- Prolog has an operator `not` (also written `\+`).

Motivation (2)

- Set difference is a natural operation. If it misses in a query language, the user will pose several queries and compute the set difference himself/herself.

  If a query language computes sets, it should be closed under the usual set operations. I.e. a set operation applied to the result of two queries should be expressable as a single query. (One could also require other simple operations, such as counting.) For relations, it should be closed under relational algebra operations. This is not quite the same as relational completeness, because this closure condition holds also e.g. for recursive queries (not expressible in relational algebra).
- It was defined above which facts are false in the minimal model. Up to now, knowledge about false facts cannot be used within in the program.

Not vs. Classical Negation (1)

- The negation operator `not` in Prolog / Datalog^{neg} is not logical negation ¬ known from classical logic.
- From the formulas

  book('Ull89', 'Ullman', 'Princ. of DBS and KBS').
  book('Llo87', 'Lloyd', 'Found. of Logic Progr.').
  borrowed('Ull89').
  available(Author, Title) ← book(BID, Author, Title) ∧ ¬borrowed(BID).

  one cannot conclude that 'Llo87' is not borrowed.

  The given formulas specify only the positive information. This was also the motivation for defining the minimal model.

Not vs. Classical Negation (2)

- Therefore, also this is not a logical consequence:

  available('Lloyd', 'Found. of Logic Progr.')
- This is a difference to Horn clause Datalog: There, all `answer`-facts in the minimal model are logical consequences of the given program.
- Negative facts must be assumed "by default" if the corresponding positive fact is not provable.
- In order to prove `not` A, Prolog first tries to prove A. If this fails, `not` A is considered "proven": "Negation as (Finite) Failure" / "Default Negation".

Not vs. Classical Negation (3)

- Classical logic is monotonic: If ψ follows from Φ, it also follows from $\Phi \cup \{\varphi'\}$ for any φ'.

  If one has more preconditions, one can derive more (or at least the same).

- This important property does not hold in logic programming: If one adds borrowed('Llo87') to the given formulas, one can no longer conclude

  available('Lloyd', 'Found. of Logic Progr.')

- Thus, "Nonmonotonic Logic" is used to explain negation in logic programming.

Not vs. Classical Negation (4)

- Not all classical equivalences hold in logic programming: For instance,

  $$\texttt{available(Author, Title)} \leftarrow \texttt{book(BID, Author, Title)} \land \neg\texttt{borrowed(BID)}.$$

  is logically equivalent to

  $$\texttt{available(Author, Title)} \lor \neg\texttt{book(BID, Author, Title)} \lor \neg\neg\texttt{borrowed(BID)}.$$

  and thus to

  $$\texttt{borrowed(BID)} \leftarrow \texttt{book(BID, Author, Title)} \land \neg\texttt{available(Author, Title)}.$$

Not vs. Classical Negation (5)

- In logic programming (with not instead of $\neg$), the two formulas have a completely different semantics:

  $$\texttt{borrowed(BID)} \leftarrow \texttt{book(BID, Author, Title)} \land \texttt{not available(Author, Title)}.$$

- Because no available-facts can be derived, Prolog now concludes that also 'Llo87' is borrowed.

- In logic programming, rules can be used in only one direction. The distinction between head and body is important.

  The contraposition of a rule is not used.

Not vs. Classical Negation (6)

- To understand not, it is helpful to assume that for every predicate p there is a new, system-defined predicate not_p.

  One can also use modal logic (with an operator "I know that").

- Then exchanging the head literal and a negated body literal is no longer logical contraposition, since not_p is not necessarily $\neg p$.

- It is not even astonishing that some negation semantics make neither p nor not_p true for difficult programs like $p \leftarrow \texttt{not}\ p$.

Overview

1. Motivation, Differences to classical logic)
2. Syntax, Supported Models
3. Stratification, Perfect Model
4. Conditional Facts, Well-Founded Model
5. Stable Models

Syntax

- Now two types of body literals are allowed:
  - ◇ Postive body literals (atomic formulas, as usual):
    $$p(t_1, \ldots, t_n)$$
  - ◇ Negative body literals (default negation of an atomic formula):
    $$\texttt{not}\ p(t_1, \ldots, t_n)$$
- The default negation operator `not` cannot be used in the head of a rule.

  This corresponds to the above view that "$\texttt{not\_}p$" is a system defined predicate. One cannot introduce rules that define this predicate.

SLDNF-Resolution (1)

- SLDNF-Resolution (SLD-resolution with negation as failure) is a generalization of SLD-resolution to programs with negative body literals.

  Some authors think that it is more precise to say "finite failure".
- As in SLD-resolution, a tree is constructed, where the nodes are marked with goals (conjunctions of positive and negative literals).

  Seen as a refutation proof, one can also view the goals as disjunction of the opposite literals.
- If the selected literal is a positive literal, child nodes are constructed as in SLD-resolution.

SLDNF-Resolution (2)

- If the selected literal is a negative literal, SLDNF-resolution calls itself recursively with the corresponding positive literal as query.

  I.e. a new tree is constructed, the root marked the the positive literal.
  - ◇ If this tree is finite and contains no success node (empty goal), the negative literal is considered proven, and the calling node gets a single child node with the negative literal removed.
  - ◇ If the tree contains a success node, the calling node is a failure node (without child nodes).

SLDNF-Resolution (3)

- Most authors require that a negative literal can only be selected if it is ground.
- The reason is the probably unexpected local quantification, see next slides.
- If the goal is not empty, but the selection function cannot select a literal (because only nonground negative literals are left), evaluation "flounders".

  This is an error condition.

- Most Prolog systems do not obey this restriction.

Range Restriction (1)

- $\texttt{not\_}p$ can only be called with the binding pattern $\texttt{bb...b}$.
- I.e. every variable of the rule must occur in a positive body literal.

  With further restrictions if built-in predicates are used.

- Many Prolog systems evaluate also negative literals with variables. But then the usual quantification is inverted:
  - ◇ $\texttt{not}\ p(\texttt{X})$ is successful if $p(\texttt{X})$ fails for all $\texttt{X}$.
  - ◇ $p(\texttt{X})$ is successful if it succeeds for at least one $\texttt{X}$.

Range Restriction (2)

- Consider the following example:

$$\begin{array}{l} p(\texttt{X}) \leftarrow \texttt{not}\ r(\texttt{X}) \wedge q(\texttt{X}). \\ q(a). \\ r(b). \end{array}$$

- Most Prolog systems will answer the query "$p(\texttt{X})$" with "no", since already $\texttt{not}\ r(\texttt{X})$ fails.

  In this case, there are actually two different variables named "$\texttt{X}$", since $\texttt{X}$ within $\texttt{not}$ is implicitly $\exists$-quantified.

- If one exchanges the two body literals, every Prolog systems answers withe the same query with $\texttt{X} = a$.

Range Restriction (3)

Remark (Anonymous Variables):

- Anonymous variables in negated body literals can be useful.
- Suppose that the table `borrowed` is extended:

borrowed	
BID	User
U1189	Brass

- Formally, the following rule would not be range-restricted, but Prolog would work as expected:

$$\begin{array}{ll} \texttt{available(Author, Title)} \leftarrow & \texttt{book(BID, Author, Title)} \wedge \\ & \texttt{not borrowed(BID, \_)}. \end{array}$$

Range Restriction (4)

Remark, continued:

- Probably, most deductive database systems permit negative body literals with anonymous variables.

  However, these variables are ∀-quantified in the body, whereas all other variables that occur only in the body are ∃-quantified.

  One can also say that anonymous variables are ∃-quantified immediately in front of the atomic formula, i.e. inside the negation.

- This is also consistent with the idea that anonymous variables project away unnecessary columns.
- Exercise: If anonymous variables were not allowed in negated literals, how would one define `available`?

Exercise

- Let the following EDB-relations be given:
  - ◇ `lecture_hall(RoomNo, Capacity)`.
  - ◇ `reservation(RoomNo, Day, From, To, Course)`.
- Which lecture halls are free on tuesdays, 8^{30}–10^{00}?
- What is the largest capacity of a lecture hall?
- Is there a time at which all lecture halls are used?

  If there is such a time at all, also one of the times in `From` satisfies this condition (Proof: Go back from the given time, when all lecture halls are used, to the nearest start of a reservation.). Thus, it is not necessary to check all possible times.

Clark's Completion (1)

- The first approach to define a semantics of negation in logic programming non-operationally was (probably) Clark's Completion (also called CDB: "completed database") [1978].
- Basically, the idea was to turn "←" into "↔.
- E.g., if the only rule about p is

$$p(X) \leftarrow q(X) \wedge r(X)$$

  the definition of p in the CDB is (equivalent to)

$$\forall X\colon\ p(X) \leftrightarrow q(X) \wedge r(X)$$

Clark's Completion (2)

- When variables occur only in the body, e.g. Y in

$$p(X) \leftarrow q(X, Y)$$

  it is normally not important whether
  - ◇ it is universally quantified over the entire rule:

$$\forall X, Y\colon\ p(X) \leftarrow q(X, Y)$$

  - ◇ or existentially over the body (equivalent):

$$\forall X\colon\ p(X) \leftarrow \exists Y\colon q(X, Y)$$

- But for the CDB only the second version works:

$$\forall X\colon\ p(X) \leftrightarrow \exists Y\colon q(X, Y)$$

Clark's Completion (3)

- If there are several rules about one predicate, the rule bodies must be connected disjunctively.
- E.g. consider

$$p(a, X) \leftarrow q(X).$$
$$p(b, X) \leftarrow r(X).$$

- The rule heads must be normalized: New variables are introduced for the arguments of the head, and equated with the original arguments in the body:

$$\forall Y_1{,}Y_2\colon\; p(Y_1, Y_2) \leftrightarrow \big(\exists X\colon Y_1 = a \wedge Y_2 = X \wedge q(X)\big) \vee \big(\exists X\colon Y_1 = b \wedge Y_2 = X \wedge r(X)\big)$$

Clark's Completion (4)

- E.g., consider this program:

$$p(X) \leftarrow q(X) \wedge \texttt{not}\ r(X).$$
$$q(a).$$
$$q(b).$$
$$r(b).$$

- Clark's completion (as follows) implies e.g. $p(a)$:

$$\forall X\ p(X) \leftrightarrow q(X) \wedge \neg r(X).$$
$$\forall X\ q(X) \leftrightarrow X = a \vee X = b.$$
$$\forall X\ r(X) \leftrightarrow X = b.$$

In addition it contains an equality theory that includes, e.g., $a \neq b$ (unique names assumption, UNA).

Clark's Completion (5)

- Consider the program

$$p \leftarrow p.$$

- The definition of p in Clark's Completion is

$$p \leftrightarrow p$$

i.e. p can be true or false.

- SLDNF resolution can prove neither p nor $\texttt{not}\ p$.

  It always gets into an infinite loop.

- However, if one uses a deductive database with bottom-up evaluation, it is clear that p is false.

  This motivates the search for stronger negation semantics, see below.

Clark's Completion (6)

- Consider the program

$$p \leftarrow \texttt{not}\ p.$$

- The definition of p in Clark's Completion is

$$p \leftrightarrow \neg p$$

which is inconsistent.

- SLDNF resolution can prove neither p nor $\texttt{not}\ p$.
- However, if the program contains other, unrelated predicates, SLDNF resolution would give reasonable positive and negative answers for them, while Clark's completion implies everything.

Clark's Completion (7)

- Of course, the rule $p \leftarrow \texttt{not}\ p$ is strange and contradictory: p is provable iff p is not provable.

  In classical logic $p \leftarrow \neg p$ is simply equivalent to p, i.e. p is true. However, as explained above, when negation is used, logic programming rules do not behave as the corresponding formulas in classical logic.

- However, such a case can be hidden in a large program, and totally unrelated to a given query.
- In order to support goal-directed query evaluation procedures, such cases must be excluded or the semantics must "localize" the consistency problem.

Overview

Stratification (1)

- In order to avoid the $p \leftarrow \texttt{not}\ p$ problem, the class of stratified programs is introduced.
- Note that negative body literals $\texttt{not}\ p(t_1, \ldots, t_n)$ can be easily evaluated if the complete extension of p was already computed previously.

  Variables among the arguments are already bound to a concrete value because of the range restriction. Thus, one only has to check whether the argument tuple is contained in the extension of p.

- This means that p must not depend on a predicate that depends on $\texttt{not}\ p$.
  In short: Recursion through negation is excluded.

Stratification (2)

Definition (Level Mapping of the Predicates):

- A level mapping of the predicates is a mapping
  $$\ell: \mathcal{P} \rightarrow \mathbb{N}_0.$$
- The domain of this mapping is extended to atomic formulas through
  $$\ell\big(\,p(t_1, \ldots, t_n)\,\big) := \ell(p).$$

Note:

- The purpose of this level mapping is to define an evaluation sequence: One starts with predicates of level 0, continues with level 1, and so on.

Stratification (3)

Definition (Stratified Program):

- A program P is stratified if and only if there is a level mapping ℓ such that for each rule

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \texttt{not}\ C_1 \wedge \cdots \wedge \texttt{not}\ C_n$$

  the following holds:
  - ◇ $\ell(B_i) \leq \ell(A)$ for $i = 1, \ldots, m$, and
  - ◇ $\ell(C_j) < \ell(A)$ for $i = 1, \ldots, n$.
- Such a level mapping is called a stratification of P.

Stratification (4)

- One can compute a stratification as follows:
  - ◇ Let k be the number of different predicates in the program.
  - ◇ Assign level 0 to every predicate.
  - ◇ Whenever the condition is violated for a rule, and the level of the predicate in the head is less than k, increment it.
  - ◇ If the level of a predicate reaches k, the program is not stratified. Otherwise, when a stable state is reached, this is a valid stratification.

Stratification (5)

- The predicate dependency graph for programs with negation is defined as follows:
  - ◇ Nodes: Predicates that occur in the program.

    Here $\texttt{not}\ p$ does *not* count as a predicate on its own.
  - ◇ There is a positive edge from q to p iff there is a rule of the form $p(\ldots) \leftarrow \ldots \wedge q(\ldots) \wedge \ldots$
  - ◇ There is a negative edge from q to p iff there is a rule of the form $p(\ldots) \leftarrow \ldots \wedge \texttt{not}\ q(\ldots) \wedge \ldots$.
- A program is stratified if and only if there is no cycle that contains at least one negative edge.

Perfect Model (1)

- For defining when an interpretation is a model of a program one treats $\texttt{not}$ like classical negation $\neg$.
- Thus, an interpretation $\mathcal{I}$ is a model of a program P iff for every rule

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \texttt{not}\ C_1 \wedge \cdots \wedge \texttt{not}\ C_n$$

  and every variable assignment $\mathcal{A}$ the following holds:
  - ◇ If $(\mathcal{I}, \mathcal{A}) \models B_i$ for $i = 1, \ldots, m$ and $(\mathcal{I}, \mathcal{A}) \not\models C_j$ for $j = 1, \ldots, n$, then $(\mathcal{I}, \mathcal{A}) \models A$.

    If one identifies again a Herbrand model $\mathcal{I}$ with its set of true facts, a ground literal $\texttt{not}\ p(c_1, \ldots, c_n)$ is true in $\mathcal{I}$ if and only if $p(c_1, \ldots, c_n) \notin \mathcal{I}$.

Perfect Model (2)

Problem:

- With negation, the minimal Herbrand model is no longer unique:

$$p \leftarrow \mathtt{not}\ q.$$

  This program has two minimal models (it is logically equivalent to $p \vee q$):

  ◇ $\mathcal{I}_1 = \{p\}$.

  ◇ $\mathcal{I}_2 = \{q\}$.

- Of these, only $\mathcal{I}_1$ is intuitively right (intended model): Since there are no rules about q, one cannot prove q. Thus, it should be false.

Perfect Model (3)

Idea:

- Predicate minimization with priorities:
  - ◇ It is natural to compute the predicate extensions in the sequence given by the level mapping.
  - ◇ Then predicates with lower level are minimized with higher priority.

    They can choose first and they want to have a minimal extension.
  - ◇ In the example $p \leftarrow \mathtt{not}\ q$ (e.g., $\ell(q) = 0$, $\ell(p) = 1$) q is minimized with higher priority, i.e. it is more important to make q false than to make p false. Therefore, one chooses $\mathcal{I}_1 = \{p\}$.

Perfect Model (4)

Definition (Prioritized Minimal Model):

- Let a level mapping ℓ for a program P be given.
- A Herbrand model $\mathcal{I}_1$ of P is preferable to a Herbrand model $\mathcal{I}_2$ ($\mathcal{I}_1 \prec_\ell \mathcal{I}_2$) iff there is $i \in \mathbb{N}_0$ with
  - ◇ $\mathcal{I}_1(p) = \mathcal{I}_2(p)$ for all predicates p with $\ell(p) < i$.
  - ◇ $\mathcal{I}_1(p) \subseteq \mathcal{I}_2(p)$ for all predicates p with $\ell(p) = i$.
  - ◇ $\mathcal{I}_1(p) \neq \mathcal{I}_2(p)$ for at least one p with $\ell(p) = i$.
- $\mathcal{I}_1 \preceq_\ell \mathcal{I}_2$ iff $\mathcal{I}_1 \prec_\ell \mathcal{I}_2$ or $\mathcal{I}_1 = \mathcal{I}_2$.

Perfect Model (5)

Theorem/Definition:

- Every stratified program P minimales Modell $\mathcal{I}_0$ with respect to $\preceq_\ell$.
- This model $\mathcal{I}_0$ is called the perfect model of P.
- The perfect model does not depend on the exact stratification. If ℓ und ℓ' are two stratifications for P, the minimal model with respect to $\preceq_\ell$ is also minimal with respect to $\preceq_{\ell'}$.

  Actually, the original definition of the perfect model does not use a level mapping but the priority relation between the predicates given by the rules of the program.

Bottom-Up Evaluation (1)

- One first applies rules about predicates of level 0. These do not contain negation.
- Then one applies the rules about predicates of level 1. These refer negatively only to predicates of level 0. But the extensions of these predicates are already known. And so on.
- When the rules are applied in a sequence obtained from topologically sorting the predicate dependency graph, one automatically gets this order compatible with the predicate levels.

Bottom-Up Evaluation (2)

Definition (Generalized T_P-Operator):

$$T_{P,\mathcal{J}}(\mathcal{I}) := \{F \in \mathcal{B}_\Sigma \mid \text{There is a rule } A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \texttt{not}\ C_1 \wedge \cdots \wedge \texttt{not}\ C_n \text{ in } P \text{ and ground substitution } \theta, \text{ such that } \theta(A) = F,\ \theta(B_i) \in \mathcal{I} \cup \mathcal{J} \text{ for } i = 1, \ldots, m,\ \theta(C_i) \notin \mathcal{J} \text{ for } i = 1, \ldots, n\}.$$

Bottom-Up Evaluation (3)

Iterated Fixpoint Computation:

- Let ℓ be a stratification of P with maximal level k.
- Let P_i be the rules about predicates of level i, i.e.
  $P_i := \{A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \texttt{not}\ C_1 \wedge \cdots \wedge \texttt{not}\ C_n \in P \mid \ell(A) = i\}$.
- Let $\mathcal{I}_0 := \emptyset$ and $\mathcal{I}_{i+1} := \mathcal{I}_i \cup \mathit{lfp}(T_{P_i,\mathcal{I}_i})$.
- Then $\mathcal{I}_{k+1}$ is the perfect model of P.

Overview

Ausblick (1)

Nicht-Stratifizierte Negation:

- Manche praktisch sinnvollen Programme sind nicht stratifiziert.
- Dies kommt z.B. immer dann vor, wenn Prädikate als ein Argument einen „Zustand“ haben, und man sich bei der Definition für den nächsten Zustand negativ auf den vorigen Zustand bezieht.
- Ein einfaches Beispiel wäre etwa:

  $even(0).$

  $even(\mathtt{X}) \leftarrow \mathtt{X} > 0 \wedge succ(\mathtt{Y}, \mathtt{X}) \wedge \mathtt{not}\ even(\mathtt{Y}).$

Ausblick (2)

- Da $succ$ azyklisch ist, hängt ein $even$-Fakt nicht wirklich negativ von sich selbst ab.
- Die Semantik der Negation ist noch relativ problemlos, wenn es so ein „dynamische Stratifikation“ der Fakten gibt.
- Die meisten deduktiven DBS können auch bei solchen Programmen die Negation auswerten.
- Allerdings hängt es nun von den Daten ab, ob ein Programm auswertbar ist oder nicht (nur falls $succ$ azyklisch).

Ausblick (3)

- Semantiken für noch allgemeinere Programme (inklusive $p \leftarrow \mathtt{not}\ p$) sind etwa die wohlfundierte Semantik (WFS) und die stabile Modelle Semantik.
- Die wohlfundierte Semantik würde im Beispiel

  $$p \leftarrow \mathtt{not}\ p$$

  weder p noch $\mathtt{not}\ p$ wahr machen (d.h. p einen dritten Wahrheitswert zuordnen).
- Für stratifizierte Programme stimmt sie (wie praktisch alle anderen vorgeschlagenen Semantiken) mit dem perfekten Modell überein.

Ausblick (4)

Aggregationen:

- Die gleichen Probleme ergeben sich im Zusammenhang mit Aggregationen wie `count`, `sum`, `min`, `max`.

  In Prolog verwendet man dazu `findall`, um die Menge der Lösungen zu einer Anfrage zu berechnen. Anschließend kann man diese Menge aggregieren (zu einem Wert zusammenfassen).
- Negation bedeutet ja auch nur: $\mathtt{count}(\ldots) = 0$.
- Auch hier verlangt man häufig eine Stratifizierung, verbietet also Rekursionen durch Aggregationen.
- Allerdings ist schon das bekannte „Bill of Materials“-Problem nur dynamisch stratifiziert.