6. Negation

1. Motivation, Beispiele
2. Unterschiede zur logischen Negation
3. Syntax/Bereichsbeschränkung
4. Stratifikation
5. Perfektes Modell
6. Bottom-Up Berechnung
7. Ausblick: Nicht-Stratifizierte Negation, Aggregationen

Beispiel (Kleine Bibliothek)

bestand		
Kürzel	Autor	Titel
Ull89	Ullman	Princ. of DBS and KBS
Llo87	Lloyd	Found. of Logic Progr.

ausgeliehen
Kürzel
Ull89

$$\text{vorhanden(Autor, Titel)} \leftarrow \text{bestand(Kürzel, Autor, Titel)} \wedge \mathbf{not}\ \text{ausgeliehen(Kürzel)}.$$

vorhanden	
Lloyd	Found. of Logic Progr.

Motivation

- Anfragen/Sichtdefinitionen wie im obigen Beispiel sind in SQL kein Problem, können aber in dem bisherigen Datalog (Hornklauseln) nicht ausgedrücky werden.
  Dagegen können alle Anfragen der Relationenalgebra nach Datalog mit Negation übersetzt werden.
- Prolog hat auch einen **not**-Operator.
- Mengendifferenz ist eine natürliche Operation — hätte man sie nicht in der Anfragesprache, würde der Benutzer mehrere Anfragen stellen, und sie auf dem Papier ausrechnen.
- Wir haben schon definiert, welche Fakten (Grundatome) im minimalen Modell falsch sind. Bisher können wir dieses Wissen aber nicht im Programm selbst weiterverarbeiten.

Unterschied zur logischen Negation (1)

- Man beachte, daß **not** nicht die logische Negation $\neg$ ist.
- Ein klassischer Logiker würde aus den Formeln

  bestand('Ull89', 'Ullman', 'Princ. of DBS and KBS').

  bestand('Llo87', 'Lloyd', 'Found. of Logic Progr.').

  ausgeliehen('Ull89').

  $$\text{vorhanden(Autor, Titel)} \leftarrow \text{bestand(Kürzel, Autor, Titel)} \wedge \neg\text{ausgeliehen(Kürzel).}$$

  nicht schließen können, daß 'Llo87' nicht ausgeliehen ist — darüber sagen diese Formeln nichts aus.

Unterschied zur logischen Negation (2)

- Daher ist auch

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

  keine logische Folgerung.

- Das ist ein Unterschied zu früher: Bei Hornklauseln sind alle answer-Fakten des minimalen Modells logische Folgerungen.

- Die Formel folgt aus Clark's Vervollständigung.

- Negationen müssen als „Default" angenommen werden, wenn das Gegenteil nicht herleitbar ist („Negation as Failure").

  Um **not** A zu beweisen, versucht Prolog zuerst A zu beweisen. Wenn dies scheitert, gilt **not** A als „bewiesen".

Unterschied zur logischen Negation (3)

- In der klassischen Logik gilt: Wenn ψ aus Φ folgt, so auch aus $\Phi \cup \{\varphi'\}$. Monotonie: Hat man mehr Voraussetzungen, so kann man auch mehr (oder zumindest nicht weniger) folgern.
- Diese ganz wichtige Eigenschaft gilt hier nicht mehr: Fügt man ausgeliehen('Llo87') ein, so kann man

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

  nicht mehr folgern.
- Wir beschäftigen uns hier also mit einer „nichtmonotonen Logik“.

Unterschied zur logischen Negation (4)

- Es gelten auch nicht mehr alle klassischen Äquivalenzen. Zum Beispiel wäre obige Formel logisch äquivalent zu:

$$\begin{aligned}\text{vorhanden(Autor, Titel)} \lor {} & \neg\text{bestand(Kürzel, Autor, Titel)} \lor \\ & \neg\neg\text{ausgeliehen(Kürzel).}\end{aligned}$$

  und damit zu

$$\begin{aligned}\text{ausgeliehen(Kürzel)} \leftarrow {} & \text{bestand(Kürzel, Autor, Titel)} \land \\ & \neg\text{vorhanden(Autor, Titel).}\end{aligned}$$

Unterschied zur logischen Negation (5)

- Dagegen unterscheiden sich die beiden Regeln mit **not** sehr:

  $$\text{ausgeliehen(Kürzel)} \leftarrow \text{bestand(Kürzel, Autor, Titel)} \wedge \textbf{not}\ \text{vorhanden(Autor, Titel)}.$$

  Da hier keine vorhanden-Fakten herleitbar sind, folgt jetzt, daß auch 'Llo87' ausgeliehen ist.

- In der logischen Programmierung können die Regeln nur in einer Richtung verwendet werden, die Unterscheidung von Kopf und Rumpf ist wichtig.
  Die Kontraposition einer Regel wird nicht benutzt.

Unterschied zur logischen Negation (6)

- Man versteht **not** am besten, wenn man sich vorstellt, daß es für jedes Prädikat p noch ein neues, systemdefiniertes Prädikat **not**_p gibt.
- Dann ist es nicht verwunderlich, daß die Kontraposition hier nicht gilt, weil ja **not**_p zunächst nichts mit $\neg$p zu tun hat.
- Dann ist es nicht einmal verwunderlich, daß einige Negations-Semantiken bei schwierigen Programmen wie

$$\mathsf{p} \leftarrow \textbf{not}\ \mathsf{p}$$

weder p noch **not** p wahr machen.

Syntax (1)

Erweiterte Regeln:

- Rumpfliterale dürfen jetzt auch die Form

  $$\mathbf{not}\ \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

  haben (Default-Negation einer atomaren Formel).

- Solche Rumpfliterale werden dann auch als negative Rumpfliterale bezeichnet (die übrigen heißen entsprechend „positive Rumpfliterale“).

- Natürlich darf der Default-Negations-Operator **not** nicht im Kopf einer Regel verwendet werden — „**not**_p“ ist ja ein systemdefiniertes Prädikat.

Syntax (2)

Bereichsbeschränkung:

- **not** _p darf nur mit dem Bindungsmuster bb...b aufgerufen werden.
- D.h. jede Variable der Regel muß in einem positiven Rumpfliteral vorkommen.
- Viele Prolog-Systeme würden auch negative Literale mit Variablen auswerten. Dabei kehrt sich aber die Bedeutung des Quantors um:
  - **not** p(X) ist erfolgreich, wenn p(X) für alle X fehlschlägt.
  - Dagegen ist p(X) erfolgreich, wenn es nur ein X gibt mit p(X).

Syntax (3)

Beispiel:

$$p(X) \leftarrow \textbf{not}\ r(X) \wedge q(X).$$

$$q(a).$$

$$r(b).$$

- Viele Prolog-Systeme würden die Anfrage $p(X)$ mit „no“ beantworten, weil schon das Beweisziel **not** $r(X)$ scheitert.
  Das erste Vorkommen der Variable X hat hier nichts mit dem zweiten zu tun, sie ist implizit innerhalb des **not** $\exists$-quantifiziert.
- Vertauscht man die beiden Rumpfliterale aber, so würde jedes Prolog-System die gleiche Anfrage mit $X = a$ beantworten.

Syntax (4)

Bemerkung:

- Anonyme Variablen in negativen Rumpfliteralen können allerdings manchmal nützlich sein.
- Zum Beispiel sei die Tabelle ausgeliehen erweitert:

ausgeliehen	
Kürzel	Entleiher
Ull89	Brass

Syntax (5)

Bemerkung, Forts.:

- Die folgende Regel wäre nicht bereichsbeschränkt:

$$\text{vorhanden}(\text{Autor}, \text{Titel}) \leftarrow \text{bestand}(\text{Kürzel}, \text{Autor}, \text{Titel}) \wedge \neg\text{ausgeliehen}(\text{Kürzel}, \_).$$

- Prolog würde hier aber wie beabsichtigt funktionieren.
- Viele deduktive DBS erlauben negative Literale mit anonymen Variablen.
- Allerdings sind diese Variablen dann im Rumpf allquantifiziert, während andere Variablen, die nur im Rumpf vorkommen, dort existenzquantifiziert sind.

Syntax (6)

Aufgabe:

Gegeben seien die EDB-Relationen:

hörsaal(Name, Kapazität).

belegung(Hörsaal, Tag, Von, Bis, Vorlesung).

- Welche Hörsäle sind mittwochs, 8^{30}–10^{00} frei?
- Was ist die größte Kapazität?
- Gibt es eine Zeit, zu der alle Säle belegt sind?
  Wenn so eine Zeit existiert, tut es auch eine der Zeiten „Von“ aus der Relation „belegung“. Man braucht also nicht alle möglichen Zeitpunkte durchzuprobieren.

Stratifizierung (1)

Motivation:

- Es gibt Programme, deren Semantik nicht intuitiv klar ist:

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

  „p ist beweisbar, wenn p nicht beweisbar ist."

  - Das ist ein Widerspruch (wir interpretieren $\leftarrow$ ja als $\leftrightarrow$).
    Der T_P-Operator hat hier auch keinen Fixpunkt.
  - Wir können über p nichts aussagen:
    Weder p noch **not** p sind wahr.
  - p ist wahr nach dem Prinzip des indirekten Beweis:
    Nehmen wir **not** p an, so erhalten wir p (Widerspruch).

Stratifizierung (2)

Motivation, Forts.:

- Man kann negative Rumpfliterale **not** $\mathrm{p}(\mathrm{t}_1, \ldots, \mathrm{t}_n)$ leicht auswerten, wenn man die ganze Extension von p schon vorher berechnet hat.
- Dann braucht man nur nachzuschauen, ob die jeweilige Grundinstanz dieses Literals in der Extension enthalten ist.
  Eventuelle Variablen unter den t_i sind ja aufgrund der Bereichsbeschränkung bereits gebunden.
- Dann darf p aber nicht von einem Prädikat abhängen, das von **not** p abhängt.
  Kurz: Rekursion durch Negation hindurch ist verboten.

Stratifizierung (3)

Stufeneinteilung der Prädikate:

- Eine Stufeneinteilung der Prädikate ist eine Abbildung

$$\ell: \mathcal{PRED} \rightarrow \mathbb{N}_0.$$

- Wir erweitern diese Abbildung auf Literale (atomare Formeln) durch

$$\ell\big(\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)\big) := \ell(\mathsf{p}).$$

- Der Sinn einer Stufeneinteilung ist, daß sie eine Berechnungsreihenfolge vorschreibt: Wir beginnen mit den Prädikaten der Stufe 0, anschließend berechnen wir die der Stufe 1, u.s.w.

Stratifizierung (4)

Definition:

Ein Programm P heißt stratifiziert gdw. es eine Stufeneinteilung ℓ gibt, so daß für jede Regel

$$\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m \wedge \textbf{not } \mathsf{C}_1 \wedge \cdots \wedge \textbf{not } \mathsf{C}_n$$

gilt:

- $\ell(\mathsf{B}_i) \leq \ell(\mathsf{A})$ für $i = 1, \ldots, m$, und
- $\ell(\mathsf{C}_j) < \ell(\mathsf{A})$ für $i = 1, \ldots, n$.

Eine solche Stufeneinteilung heißt Stratifizierung von P.

Stratifizierung (5)

Bemerkung:

- Innerhalb einer Stufe ist also Rekursion erlaubt, aber keine Negation: Negationen müssen sich immer auf Prädikate niedrigerer Stufe beziehen.
- Man kann so eine Stufeneinteilung z.B. berechnen, indem man zuerst jedem Prädikat die Stufe 0 zuordnet, und dann immer die Stufe des Prädikates im Kopf um 1 erhöht, wenn die Regel die Bedingung verletzt.
- Sei n die Anzahl unterschiedlicher Prädikate, die in P vorkommen. Bekommt man bei obigen Verfahren eine Stufe $\geq n$, so ist das Programm nicht stratifiziert (Abbruch).

Stratifizierung (6)

Prädikat-Abhängigkeitsgraph:

- Man kann den Prädikat-Abhängigkeitsgraph auch für Programme mit Negation definieren:
  - Knoten sind wieder die vorkommenden Prädikate (wobei **not** p nicht als eigenes Prädikat zählt).
  - Eine Regel $p(\ldots) \leftarrow \ldots \wedge q(\ldots) \wedge \ldots$ führt zu einer positiven Kante von q nach p.
  - Eine Regel $p(\ldots) \leftarrow \ldots \wedge \textbf{not}\ q(\ldots) \wedge \ldots$ führt zu einer negativen Kante von q nach p.
- Ein Programm ist genau dann stratifiziert, wenn kein Zyklus eine negative Kante enthält.

Perfektes Modell (1)

Modell-Begriff:

- Bei stratifizierten Programmen kann man für die Definition von „Modell“ $\mathbf{not}$ doch wie $\neg$ in der klassischen Logik behandeln.
- Es sei $\mathsf{truth}\big(\mathcal{I}, \alpha, \mathbf{not}\ \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)\big) := \mathsf{T}$ gdw.

$$\big(\mathsf{eval}(\mathcal{I}, \alpha, \mathsf{t}_1)\ , \ldots ,\ \mathsf{eval}(\mathcal{I}, \alpha, \mathsf{t}_n)\big) \notin \mathcal{I}_{\mathsf{pred}}(\mathsf{p}).$$

  Ansonsten sei $\mathsf{truth}\big(\mathcal{I}, \alpha, \mathbf{not}\ \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)\big) := \mathsf{F}$.

- Bzw. für ein Herbrand-Modell $\mathcal{I}$ sei ein Grundliteral $\mathbf{not}\ \mathsf{p}(\mathsf{c}_1, \ldots, \mathsf{c}_n)$ genau dann wahr, wenn $\mathsf{p}(\mathsf{c}_1, \ldots, \mathsf{c}_n) \notin \mathcal{I}$.

Perfektes Modell (2)

Problem:

- Das minimale Herbrand-Modell ist nicht eindeutig bestimmt:

  $$\mathsf{p} \leftarrow \mathbf{not}\ \mathsf{q}.$$

  Dieses Programm hat zwei minimale Modelle:

  - $\mathcal{I}_1 = \{\mathsf{p}\}$.
  - $\mathcal{I}_2 = \{\mathsf{q}\}$.

- Von diesen beiden Modellen ist aber nur $\mathcal{I}_1$ intuitiv richtig (intendiertes Modell): Weil es keine Regel über q gibt, kann man q nicht herleiten. Es sollte daher falsch sein.

Perfektes Modell (3)

Idee:

- Priorisiert minimales Modell
- Weil man die Prädikat-Extensionen in der Reihenfolge der Stufeneinteilung berechnet, sollten die Prädikate niedrigerer Stufe mit höherer Priorität minimiert werden.
- Sie können es sich ja gewissermaßen zuerst aussuchen und möchten gerne eine möglichst kleine Extension haben.
- In obigem Beispiel (z.B. $\ell(\mathsf{q}) = 0$, $\ell(\mathsf{p}) = 1$) würde q mit höherer Priorität minimiert werden. Deswegen entscheidet man sich für $\mathcal{I}_1$, in dem p wahr und q falsch ist.

Perfektes Modell (4)

Priorisiert minimales (perfektes) Modell:

- Sei eine Stufeneinteilung ℓ für ein Programm P gegeben.
- Ein Herbrandmodell $\mathcal{I}_1$ von P ist kleiner („besser“) als ein Herbrand-Modell $\mathcal{I}_2$ ($\mathcal{I}_1 \prec_\ell \mathcal{I}_2$) gdw. es ein $i \in \mathbb{N}_0$ gibt, so daß
  - Für alle Prädikate p mit $\ell(\mathsf{p}) < i$ gilt: $\mathcal{I}_1(\mathsf{p}) = \mathcal{I}_2(\mathsf{p})$.
  - Für alle Prädikate p mit $\ell(\mathsf{p}) = i$ gilt: $\mathcal{I}_1(\mathsf{p}) \subseteq \mathcal{I}_2(\mathsf{p})$.
  - Für mindestens ein p mit $\ell(\mathsf{p}) = i$ gilt: $\mathcal{I}_1(\mathsf{p}) \neq \mathcal{I}_2(\mathsf{p})$.
- $\mathcal{I}_1 \preceq_\ell \mathcal{I}_2$ gelte gdw. $\mathcal{I}_1 \prec_\ell \mathcal{I}_2$ oder $\mathcal{I}_1 = \mathcal{I}_2$.

Perfektes Modell (5)

Satz/Definition:

- Jedes stratifizierte Programm P hat genau ein bezüglich $\preceq_\ell$ minimales Modell $\mathcal{I}_0$.
- Wir nennen $\mathcal{I}_0$ das perfekte Modell von P.
- Das perfekte Modell hängt nicht von der genauen Stratifizierung ab. Sind ℓ und ℓ' zwei verschiedene Stratifizierungen für P, so ist das bzgl. $\preceq_\ell$ minimale Modell auch bzgl. $\preceq_{\ell'}$ minimal.
  Tatsächlich benutzt die originale Definition von perfektem Modell keine Stufeneinteilung, sondern nur eine durch die Regeln gegebene Prioritätsrelation.

Bottom-Up Auswertung (1)

Idee:

- Man wendet zunächst die Regeln über Prädikate der 0-ten Stufe an. Diese enthalten noch keine Negation.
- Dann wendet man die Regeln über Prädikate der 1-sten Stufe an. Diese beziehen sich nur negativ auf Prädikate der 0-ten Stufe. Deren Extensionen haben wir aber schon berechnet. u.s.w.
- Diese Berechnungsreihenfolge ergibt sich automatisch, wenn wir den reduzierten Prädikat-Abhängigkeitsgraphen topologisch sortieren (wie oben).
- Eine solche iterierte Berechnung ist also eigentlich nicht neu.

Bottom-Up Auswertung (2)

Verallgemeinerter $\mathsf{T_P}$-Operator:

$$
\mathsf{T}_{\mathsf{P},\mathcal{J}}(\mathcal{I}) := \Big\{ \mathsf{F} \in \mathcal{H}_{\mathcal{SIG}} \;\Big|\; \begin{array}{l}
\text{Es gibt eine Regel} \\
\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m \\
\qquad \wedge\ \mathbf{not}\ \mathsf{C}_1 \wedge \cdots \wedge \mathbf{not}\ \mathsf{C}_n \\
\text{in P und eine Grundsubstitution } \theta, \\
\text{so daß} \\
\quad \theta(\mathsf{A}) = \mathsf{F}, \\
\quad \theta(\mathsf{B}_i) \in \mathcal{I} \cup \mathcal{J} \text{ für } i = 1, \ldots, m, \\
\quad \theta(\mathsf{C}_i) \notin \mathcal{J} \text{ für } i = 1, \ldots, n \Big\}.
\end{array}
$$

Bottom-Up Auswertung (3)

Iterierte Fixpunkt-Bildung:

- Sei ℓ eine Stratifizierung von P mit höchster vorkommenden Stufe n.
- Seien P_i die Regeln über Prädikate der Stufe i, also

$$\mathsf{P}_i := \{\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m \wedge \mathbf{not}\ \mathsf{C}_1 \wedge \cdots \wedge \mathbf{not}\ \mathsf{C}_n \ \in\ \mathsf{P} \mid \ell(\mathsf{A}) = i\}.$$

- Sei $\mathcal{I}_0 := \emptyset$ und $\mathcal{I}_{i+1} := \mathcal{I}_i \cup \mathsf{lfp}(\mathsf{T}_{\mathsf{P}_i, \mathcal{I}_i})$.
- Dann ist $\mathcal{I}_{n+1}$ das perfekte Modell von P.

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:

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

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 modernen 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).
- Semantiken für noch allgemeinere Programme (inklusive p $\leftarrow$ **not** p) sind etwa die wohlfundierte Semantik (WFS) und die stabile Modelle Semantik.

Ausblick (3)

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: $\texttt{count}(\ldots) = 0$.
- Auch hier verlangt man häufig eine Stratifizierung, verbietet also Rekursionen durch Aggregationen hindurch.
- Allerdings ist schon das bekannte „Bill of Materials"-Problem nur dynamisch stratifiziert.