Antwortmengenprogrammierung

Torsten Schaub

19. Juni 2002

Die Idee der Formalisierung logischer Schlußweisen lässt sich bis in die Antike zu den Arbeiten von Aristoteles († 322 v.Chr.) zurückverfolgen. Die damals entscheidende Entdeckung war, daß logische Schlüsse allein von der Form der beteiligten Aussagen abhängen und damit vom Inhalt der Aussagen unabhängig sind — eine für die Automatisierung durch Rechenmaschinen unabdingbare Voraussetzung! Es war dann Leibniz (1646–1716), der als erster den Traum einer voll automatisierten Denkmaschine formulierte. Der effektive Durchbruch bei der Abbildung logischer Schlußweisen auf ausführbare Rechenoperationen gelang allerdings erst Robinson im Jahre 1965 mit dem sogenannten Resolutionskalkül. Im Gegensatz zu den bis dato dem menschlichen Vorgehen nachempfundenen Schlußregeln, ist die Resolutionsregel bestens für die mechanische Abarbeitung durch Computer geeignet. Genauso wie eine Vielzahl heutiger Hochleistungsbeweiser basiert auch die Programmiersprache Prolog auf dem Resolutionskalkül.

Mit der Erfindung von Prolog Anfang der 70er Jahre haben Colmerauer und Kowalski gezeigt, wie Logik zum Programmieren verwendet werden kann. Damit gelang eine erfolgreiche Synthese deklarativer und prozeduraler Programmierung. Grob gesprochen kann man damit eine Prolog-Regel, wie

```
über(X,Y) :- auf(X,Z),über(Z,Y),
```

(gelesen: Wenn X auf Z liegt und sich Z über Y befindet, dann befindet sich auch X über Y) in deklarativer Weise als die logische Formel

$$\forall xy(\exists z(auf(x,z) \wedge über(z,y)) \rightarrow über(x,y))$$

verstehen. Prozedural kann die Regel auch als (Teil einer) Prozedur '`über`' mit zwei Parametern `X` und `Y`, einer lokalen Variable `Z` und zwei Prozeduraufrufen `auf(X,Z)` und `über(Z,Y)`, ähnlich wie in der Programmiersprache Pascal, interpretiert werden.

Prolog ist allerdings nicht rein deklarativ, was man daran erkennen kann, daß die beiden Programme

```
über(X,Y) :- auf(X,Y).
über(X,Y) :- auf(X,Z),über(Z,Y).
```

und

```
über(X,Y) :- über(Z,Y),auf(X,Z).
über(X,Y) :- auf(X,Y).
```

zwar bezüglich ihrer logischen Interpretation äquivalent sind, sich aber bezüglich ihres prozeduralen Verhaltens unterscheiden. Fügt man zum Beispiel die Aussage `auf(buch,tisch)` zu beiden Programmen hinzu und stellt dann die Anfrage

`über(buch,tisch)`, so wird dies durch das erste Programm bejaht, während sich das zweite in einer Endlosschleife verirrt und nicht terminiert. Dies liegt daran, daß in Prolog die Reihenfolge der Regeln deren Abarbeitung bestimmt. Doch auch wenn dies aus Sicht eines Programmierers durchaus legitim erscheint, so genügt es nicht den Ansprüchen einer rein deklarativen Sicht der Programmierung.

Bis zum Ende der 90er Jahre wurde deshalb eine eine Vielzahl von Alternativen untersucht, um diese Abweichung der deklarativen Bedeutung eines logischen Programms von seiner prozeduralen Ausführung zu beseitigen. Die Antwortmengenprogrammierung kann als ein erfolgreiches Resultat dieses Forschungsvorhabens gesehen werden; sie vereint Techniken aus der logischen Programmierung mit denen aus den Gebieten der Wissensrepräsentation, der deduktiven Datenbanken und des nichtmonotonen Schließens. Regeln der Form

$$L_0 \leftarrow L_1, \ldots, L_m, notL_{m+1}, \ldots, notL_n$$

(mit atomaren Aussagen L_k für $0 \leq k \leq n$ über endlichem Gegenstandbereich) werden hier jeweils als Randbedingungen (engl.: constraints) an zu formende Antwortmengen A aufgefasst: Wenn $L_i \in A$ für $1 \leq i \leq m$ und $L_j \notin A$ für $m < j \leq n$, dann ist $L_0 \in A$. Logische Programme bestehen dann aus einer Menge solcher Regeln und haben keine, eine oder gar mehrere Antwortmengen. Zum Beispiel hat das Programm

$$\begin{array}{lll} (1) & p & \leftarrow & p \\ (2) & q & \leftarrow & not\ p \end{array}$$

eine einzige Antwortmenge, nämlich $\{q\}$. Denn nur mittels der zirkulären Regel (1) kann p nicht zur Antwortmenge gehören, und nach Regel (2) gehört q zur Antwortmenge, falls p nicht dazugehört. Das Programm

$$\begin{array}{lll} p & \leftarrow & not\ q \\ q & \leftarrow & not\ p \end{array}$$

besitzt zwei Antwortmengen $\{p\}$ und $\{q\}$. Auch können Integritätsbedingungen durch „kopflose“ Regeln (dh. unter Weglassen von L_0) formuliert werden. Das Programm

$$\begin{array}{lll} p & \leftarrow & not\ q \\ q & \leftarrow & not\ p \\ & \leftarrow & not\ p \end{array}$$

hat dann nur eine Antwortmenge: $\{p\}$. Die Integritätsbedingung wird verletzt, wenn immer p nicht zur Antwortmenge gehört; dies führt zur Elimination der Antwortmenge $\{q\}$.

Formal wird die Semantik von Antwortmengen durch Fixpunkte nichtmonotoner Operatoren beschrieben. Man kann zeigen, daß alle Entscheidungs– und Suchprobleme in der Problemklasse NP (auch salopp als die Klasse der in einfach exponentieller Zeit lösbarer Probleme bezeichnet) durch die Berechnung von Antwortmengen entsprechender (aussagen)logischer Programme gelöst werden können. [1] Interessanterweise erhöht weder die Verwendung von Integritätsbedingungen noch die Erweiterung der Sprache um die explizite Negation $\neg$ die Komplexität und damit die eigentliche Ausdrucksstärke des Formalismus. Mit letzterem kann man etwa leicht Weltabgeschlossenheitsaxiome, wie

[1] Antwortmengenprogrammierung ist über unendlichen Gegenstandsbereichen Turing–mächtig.

$\neg p(x) \leftarrow not\ p(x)$ formulieren, frei nach dem Motto: „Ist $p(x)$ nicht beweisbar, dann ist $p(x)$ falsch'". Die Komplexität und damit einhergehend die Ausdrucksskraft wird erst durch die Hinzunahme von Regeln mit Disjunktion auf der linken Seite der Regel erhöht. Man befindet sich dann in der Problemklasse $\mathrm{NP}^{\mathrm{NP}}$. Daran ändert auch die Hinzunahme des Operators *not* auf der linken Seite und sogar die beliebige Schachtelung von Formeln mit *not*, $\neg$, sowie Disjunktion und Konjunktion nichts mehr. Zum Beispiel, kann die Frage nach einer Belegung von a und b mit wahr/falsch, die die Formel $(a \vee \neg b) \wedge (\neg a \vee b)$ wahr macht, wie folgt repräsentiert werden:

$$\begin{array}{rcl@{\qquad\qquad}rcl} a \vee not\ a & \leftarrow & & \leftarrow & not\ a, b \\ b \vee not\ b & \leftarrow & & \leftarrow & a, not\ b \end{array}$$

Während die beiden Regeln auf der linken Seite den (zweiwertigen) Wertebereich von a und b, also die Menge aller möglichen Belegungen beschreiben, eliminieren die beiden Integritätsbedingungen alle unzulässigen Kandidaten. Hieraus resultieren zwei Antwortmengen, $A_1 = \{a, b\}$ und $A_2 = \emptyset$, die den beiden Lösungen unseres Problems entsprechen.

Ein in der Praxis enorm hilfreiches und erfreulicherweise komplexitätsneutrales Sprachkonstrukt ist die Kardinalitätsbeschränkung. Grob gesprochen steht ein Kardinalitätsliteral, wie $m\ \{p(x)\}\ n$, für jede Menge mit mindestens m und höchstens n Instanzen von $p(x)$. Zur Illustration wollen wir das oft in der Informatik behandelte n-Damenproblem verwenden. Es geht darum, n Damen so auf einem $n \times n$-Schachbrett zu plazieren, so daß keine Dame eine andere bedroht.

$$\begin{array}{rcl} d(1..n) & \leftarrow & \\ 1\ \{q(x,y)\}\ 1 & \leftarrow & d(x) \\ 1\ \{q(x,y)\}\ 1 & \leftarrow & d(y) \\ & \leftarrow & q(x,y), q(x',y'), x \neq x', y \neq y', |x - x'| = |y - y'| \end{array}$$

Während das Prädikat d den Wertebereich fixiert — es gibt also n Damen, die von 1 bis n durchnumeriert sind—, beschreibt q die Positionen der Damen. Die zweite und dritte Regel generieren sozusagen alle Schachbretter, in deren Spalten und Zeilen genau eine Dame steht. Steht die Dame mit der Nummer x in der Spalte (bzw. Zeile) x, so ist Zahl der Felder $q(x, y)$, auf der eine Dame steht, mindestens und höchstens 1. Die Integritätsbedingung eliminiert dann noch solche, bei denen sich zwei Damen auf derselben Diagonale befinden. Jede Antwortmenge repräsentiert dann eine Lösung des n-Damenproblems.

Oft wird die Modellierung spezieller Problemklassen durch abstrakte Sprachkonstrukte unterstützt. Bei der Handlungsplanung setzt man beispielsweise zur Modellierung kausaler Abläufe Konstrukte wie

$$\begin{array}{lll} \textit{bewege(b,l)} & \textbf{\textit{causes}} & \textit{auf(b,l)} \\ & \textbf{\textit{inertial}} & \textit{auf(b,l)} \end{array}$$

ein, die dann direkt in logische Regeln der Form

$$\begin{array}{lcl} \textit{auf(b,l,t+1)} & \leftarrow & \textit{bewege(b,l,t).} \\ \textit{auf(b,l,t+1)} & \leftarrow & \textit{auf(b,l,t), not } \neg\textit{auf(b,l,t+1).} \end{array}$$

übersetzt werden können. Die erste Regel sagt, daß die Bewegung eines Blockes b zum Ort l zum Zeitpunkt t zur Folge hat, daß b zum Zeitpunkt $t + 1$ auf l

steht; während die zweite Regel sagt, daß wenn sich ein Block b zum Zeitpunkt t auf l befindet, er sich auch zum Zeitpunkt $t+1$ dort befindet, es sei denn man kann das Gegenteil beweisen.

Neben der Handlungsplanung ist die Antwortmengenprogrammierung für viele weitere Anwendungen geeignet, etwa Diagnose, Scheduling, Konfigurierung, Timetabling, Model Checking, Kryptographie uvm. Zu den interessantesten Anwendungen der Antwortmengenprogrammierung zählen momentan die Konfigurierung von (Debian) Linux und Schedulingaufgaben beim Space Shuttle der NASA. Die Vorteile des Ansatzes liegen sicherlich in seiner Einfachheit und der daraus resultierenden Transparenz. Mit der Verfügbarkeit effizienter Systeme dringt die Methodik auch immer mehr in Anwendungsgebiete vor, wo es gilt komplexe Systeme qualitativ zu modellieren.

Links

- `http://www.dbai.tuwien.ac.at/proj/dlv/`
- `http://www.tcs.hut.fi/Software/smodels/`
- `http://www.cs.uni-potsdam.de/~linke/nomore/`

Literatur

- M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing, 1991, pp. 365-385.
- V. Lifschitz, Foundations of logic programming, in Principles of Knowledge Representation, CSLI Publications, 1996, pp. 69-127.
- V. Lifschitz, Answer set planning, in Proceedings of the 1999 International Conference on Logic Programming, 1999, pp. 23-37.