it 3/2004

Logik und Informatik

Logic and Computer Science

Stephan Kreutzer, Nicole Schweikardt, Humboldt-Universität zu Berlin

Zusammenfassung Als eine ihrer wichtigsten theoretischen Grundlagen hat die *Logik* Bedeutung für fast alle Bereiche der Informatik. Wir wollen diese hier anhand der Bereiche *Datenbanken*, *automatische Verifikation* und *Komplexitätstheorie* erläutern und dabei die Fragestellungen und Ergebnisse unserer beiden Dissertationen vorstellen.

►►► **Summary** As one of its most important foundations, *logic* has applications in many areas of computer science. In this article we present some of these applications in the areas *databases*, *verification*, and *complexity theory* and, at the same time, present results of our dissertations.

KEYWORDS F.4.1 [Mathematical Logic]; F.3.1 [Specifying and Verifying and Reasoning about Programs]; H.2 [Database Management]; F.2 [Analysis of Algorithm and Problem Complexity]

1 Einleitung

Als eine ihrer wichtigsten theoretischen Grundlagen hat die Logik Bedeutung für fast alle Bereiche der Informatik. Wir wollen dies hier anhand der Bereiche Datenbanken, Verifikation und Komplexitätstheorie erläutern und dabei einige Fragestellungen und Ergebnisse unserer Dissertationen vorstellen.

FO: Die Logik erster Stufe

Als erstes stellen wir die Logik erster Stufe (kurz: FO, nach der englischen Bezeichnung *first-order logic*) anhand eines Beispiels aus dem Datenbank-Kontext vor. In unserer Beispiel-Datenbank sind die Personaldaten eines Unternehmens gespeichert. Unter anderem gibt es eine Relation Angestellte mit den Attributen Name und Abteilung und eine Relation Unterabteilungen mit den Attributen Abt und Unterabt. In Bild 1 ist eine SQL-Anfrage gegeben, die die Namen aller Angestellten sucht, die höchstens eine Hierarchieebene unterhalb der Abteilung "Finanzmanagement" tätig sind.

In FO lässt sich diese Anfrage durch die in Bild 2 dargestellte Formel $\varphi(\mathsf{n})$ formulieren. Für einen Namen n besagt diese Formel, dass es Strings a und u gibt, sodass (n,u) ein Tupel der Angestellte-Relation ist und u der String "Finanzmanagement" oder aber a der String "Finanzmanagement" und (a,u) ein Tupel der Unterabteilungen-Relation ist. Das Symbol ∃ dient also als Existenzquantor, während die Symbole ∧ und ∨ für die logischen Verknüpfungen „und" und „oder" stehen.

Eine FO-Formel wird stets über einer Struktur – in diesem Fall der Datenbank – ausgewertet, die aus einem Universum und konkreten Belegungen (die Datenbankrelationen) der in der Formel benutzten Relationssymbole besteht. In FO hat man außer den bereits erwähnten noch das Symbol ¬ zur Negation logischer Aussagen zur Verfügung.

Oft betrachtet man FO an Stelle von SQL – einerseits, weil man mit FO viele „natürliche" SQL-Anfragen ausdrücken kann, und andererseits, weil FO im Gegensatz zu SQL eine sehr überschaubare und daher mathematischen Methoden gut zugängliche Syntax und Semantik hat. Außerdem wurde die Logik in der Mathematik und Informatik auch unabhängig vom Datenbank-Kontext bereits gut untersucht.

So wichtig FO auch ist, erweist sich doch ihre Ausdrucksstärke für viele Anwendungen als zu ge-

Die Dissertationen von Herrn Dr. Kreutzer und Frau Dr. Schweikardt wurden mit dem GI-Dissertationspreis 2002 ausgezeichnet. Die Gutachter in Dr. Kreutzers Promotionsverfahren, das im Jahre 2002 an der RWTH Aachen durchgeführt wurde, waren Prof. Erich Grädel (RWTH Aachen) und Prof. Wolfgang Thomas (RWTH Aachen). Die Gutachter in Dr. Schweikardts Promotionsverfahren, das im Jahr 2002 an der Universität Mainz durchgeführt wurde, waren Prof. Clemens Lautemann (Johannes Gutenberg-Universität Mainz), Prof. Thomas Schwentick (Philipps-Universität Marburg) und Prof. Jörg Flum (Albert-Ludwigs-Universität Freiburg).

Bild 1
SQL-Anfrage

```
SELECT A.Name
FROM Angestellte A, Unterabteilungen U
WHERE A.Abteilung = "Finanzmanagement"
    OR (A.Abteilung = U.Unterabt AND U.Abt = "Finanzmanagement")
```

Bild 2
FO-Formel zur Anfrage in Bild 1

$$\varphi(n) := \exists a\, \exists u \;\; \text{Angestellte}(n,u) \wedge \big(u = \text{"Finanzmanagement"} \vee (a = \text{"Finanzmanagement"} \wedge \text{Unterabteilungen}(a,u))\big)$$

ring. Daher behandeln beide hier vorgestellten Dissertationen Erweiterungen von FO. In [1] geht es vornehmlich um Erweiterungen durch „rekursive Definitionen", in [2] um Erweiterungen zum „Rechnen und Zählen".

Rechnen und Zählen

In SQL sind arithmetische Operationen wie + und × möglich. Wenn in unserer Beispiel-Datenbank eine zusätzliche Relation Monatsgehalt mit den Attributen Name und Gehalt vorkommt, könnte es von Interesse sein, das Jahresgehalt jedes Mitarbeiters zu ermitteln. Die zugehörige SQL-Anfrage findet sich in Bild 3, wo auch eine dazu äquivalente Formel angegeben ist. Für einen Namen n und eine Zahl j besagt die dortige Formel $\varphi(n, j)$, dass j das 12-fache des Monatsgehalts des Angestellten namens n ist.

Wir schreiben FO(+, ×) um anzudeuten, dass FO um die arithmetischen Operationen + und × erweitert wurde.

Eine andere wichtige SQL-Operation, die nicht ohne weiteres in FO umgesetzt werden kann, ist die Aggregatfunktion Count, mit der man zum Beispiel zählen kann, wie viele Angestellte ein Monatsgehalt von über 10000 Euro beziehen (siehe Bild 4). Um diese Anfrage mit logischen Formeln beschreiben zu können, erweitern wir FO um einstellige Zählquantoren der Form $\exists^{=\nu}$. Die Beispiel-Formel $\varphi(z)$ aus Bild 4 besagt, dass es genau z viele Namen n gibt, sodass das Monatsgehalt g des Angestellten namens n größer als 10000 ist.

Die Erweiterung von FO um einstellige Zählquantoren bezeichnen wir mit FOunC.

Rekursive Definitionen

Eine andere Art von SQL-Abfragen, die in FO nicht ausgedrückt werden können, sind solche, die auf Rekursion beruhen. Betrachten wir dazu noch einmal die SQL-Anfrage aus Bild 1. Hier wurde nach Mitarbeitern gefragt, die in einer Abteilung höchstens eine Hierarchiestufe unterhalb des Finanzmanagements arbeiten. Möchte man nun aber alle Mitarbeiter wissen, die dem Finanzmanagement irgendwie unterstehen, so stößt FO hier an Grenzen. Wenn man die genaue Struktur der Unterabteilungen kennt, insbesondere die Tiefe der Hierarchie innerhalb der Finanzabteilung, kann man die Anfrage aus Bild 1 natürlich entsprechend erweitern, um alle Unterabteilungen explizit einzubeziehen. Bei tiefen Hierarchien ist dies aber sehr mühsam und außerdem sehr anfällig gegenüber Änderungen der Strukur. Denkt man an Beispiele wie Abstammungs- oder XML-Bäume mit zunächst unbestimmter Tiefe, so scheitert FO hier. Aus diesem Grunde wurde im SQL-99 Standard ein Konstrukt eingeführt, das es erlaubt, Abfragen mittels Rekursion zu definieren. Bild 5 zeigt eine SQL-Anweisung, die alle Mitarbeiter auswählt, die in einer dem Finanzmanagement untergliederten Abteilung arbeiten. Dazu wird zunächst rekursiv die Relation SubAbt aufgebaut, indem beginnend mit der Abteilung Finanzmanagement schrittweise immer tiefere Unterabteilungen hinzugenommen werden. Anschließend werden alle Angestellten ausgewählt, die in einer Abteilung aus der Relation SubAbt arbeiten.

Rekursive Anfragen spielen vor allem auch im Bereich moderner XML-Datenbanken eine zentrale Rolle. Eine elegante Methode, solche Anfragen logisch zu abstrahieren, stellen Fixpunktlogiken dar. Sie erweitern FO um Konstrukte zur Bildung von Fixpunkten, mit denen rekursive Definitionen formuliert werden können. Unter den verschiedenen Arten von Fixpunkten, die dabei verwendet werden, spielen kleinste Fixpunkte monotoner Operatoren eine besondere Rolle. So entsteht zum Beispiel die kleinste Fixpunktlogik (LFP) aus der Erweiterung von FO um kleinste Fixpunktoperatoren. Als Beispiel sei hier die LFP-Formel in Bild 6 genannt, die zur SQL-Anfrage in Bild 5 äquivalent ist. Hierbei definiert der Teil „**lfp** SubAbt(Abt) ..." wie oben beschrieben die Relation SubAbt.

Bild 3
SQL-Anfrage und FO(×)-Formel

```
SELECT Name, Gehalt × 12
FROM Monatsgehalt
```

$$\varphi(n,j) := \exists g\, (j = g \times 12) \wedge \text{Monatsgehalt}(n,g)$$

Bild 4
SQL-Anfrage und FOunC(<)-Formel

```
SELECT Count(*)
FROM Monatsgehalt
WHERE Gehalt > 10000
```

$$\varphi(z) := \exists^{=z} n\, \exists g\, (g > 10000) \wedge \text{Monatsgehalt}(n,g)$$

2 Logiken mit Rekursion

Logiken wie LFP, die auf kleinsten Fixpunkten basieren, haben, vor allem im algorithmischen Bereich, viele Vorteile, jedoch auch eine Reihe von Schwächen. Insbesondere sind sie meist vergleichs-

Bild 5
Rekursive SQL-Anfrage.

```
WITH RECURSIVE SubAbt(Abt)
AS (
   SELECT A.Abteilung
   FROM Angestellte A
   WHERE A.Abteilung = "Finanzmanagement"
UNION ALL
   SELECT U.Unterabt
   FROM SubAbt S, Unterabteilungen U
   WHERE S.Abt = U.Abteilung
)
SELECT A.Name, S.Abt
FROM Angestellte A, SubAbt S
WHERE A.Abteilung = S.Abt
```

Bild 6
LFP-Formel zur Anfrage in Bild 5.

$$\varphi(\text{Name, Abt}) := \textbf{Angestellte}(\text{Name, Abt}) \wedge \textbf{lfp}\ \text{SubAbt(Abt): (Abt = "Finanzmanagement")} \vee (\exists \text{Abt2}\ \text{SubAbt(Abt2)} \wedge \text{Unterabteilungen(Abt2, Abt)})$$

weise starken syntaktischen Einschränkungen unterworfen, die das Schreiben solcher Formeln erschweren. Eine zweite, wichtige Art von Fixpunkten, die diese syntaktischen Einschränkungen jedoch vermeiden, sind die so genannten inflationären Fixpunkte, mit denen das rekursive, also stufenweise, Aufbauen der Lösungsrelation genau nachgebildet werden kann. So entsteht zum Beispiel die inflationäre Fixpunktlogik (IFP) aus der Erweiterung von FO um inflationäre Fixpunkte.

Aus der Definition der Logiken folgt sofort, dass LFP in IFP enthalten ist, dass also jede Formel aus LFP zu einer in IFP äquivalent ist. 1986 konnten Gurevich und Shelah zeigen, dass im Bereich endlicher Strukturen LFP und IFP die gleiche Ausdrucksstärke haben. Die Lösung nutzte jedoch spezielle Eigenschaften endlicher Strukturen und ließ sich auf den allgemeinen Fall beliebiger Strukturen nicht ohne weiteres übertragen. Und in der Tat zeigen sich wichtige Unterschiede im Verhalten von LFP und IFP zwischen endlichen und unendlichen Strukturen. Dennoch konnte in [1] gezeigt werden, dass auch im Fall beliebiger Strukturen die Logiken die gleiche Ausdrucksstärke haben. Damit wurde dieses offene Problem abschließend geklärt.

3 Datenbanken

In räumlichen Datenbanken werden Informationen über Objekte im Raum gespeichert, zum Beispiel ihre Position und Ausdehnung. Ein typisches Beispiel solcher Datenbanken sind geographische Informationssysteme (GIS), die unter anderem Informationen über Landkarten speichern. Flüsse oder Staaten auf der Landkarte kann man sich als Linien beziehungsweise Flächen in der Ebene vorstellen. Eine Landkarte ist also eine „Datenbank", die für jeden Staat und jeden Fluss eine Relation enthält, die aus denjenigen Punkten in der Ebene besteht, die auf der Karte zum jeweiligen Staat beziehungsweise Fluss gehören. Diese Sichtweise hat den Vorteil, dass man Anfragen auf natürliche Weise formulieren kann. Beispielsweise kann die Anfrage, ob der Fluss F durch den Staat S fließt, in FO elegant durch die Formel $\exists x\, \exists y\, \mathsf{S}(x, y) \wedge \mathsf{F}(x, y)$ ausgedrückt werden, die fragt, ob S und F einen gemeinsamen Punkt (x,y) haben. *Intern* müssen die – theoretisch unendlich großen – Relationen S und F natürlich auf endliche Weise repräsentiert sein. Im Bereich der Constraint Datenbanken wird dies dadurch realisiert, dass die Relationen durch sie definierende Funktionen beziehungsweise Formeln gespeichert werden.

Abfragesprachen

Im Bereich räumlicher Datenbanken spielen auf Rekursion beruhende Datenbankanfragen, zum Beispiel Erreichbarkeitsfragen, eine wichtige Rolle. Daher liegt es nahe, Logiken wie LFP als Basis von Abfragesprachen zu verwenden. Wie in [1] gezeigt wurde, führt der naive Ansatz allerdings schon für Logiken mit sehr einfachen Rekursionskonstrukten zu unentscheidbaren Sprachen, also Sprachen, deren Anfragen sich nicht mehr in endlicher Zeit auswerten lassen. Als Lösung dieses Problems werden daher auf Fixpunktlogiken basierende Sprachen vorgeschlagen, in denen die Fixpunktoperatoren so eingeschränkt werden, dass man vergleichsweise ausdrucksstarke Sprachen erhält, die aber trotzdem noch effizient ausgewertet werden können.

Kollaps-Resultate

Für eine Logik wie $\mathrm{FO}(<, +, \times)$ spricht man von einem *Kollaps-Resultat*, falls alle relevanten darin ausdrückbaren Anfragen bereits in $\mathrm{FO}(<)$ ausgedrückt werden können. Im Falle eines solchen Kollapses kann man dem Nutzer zur komfortableren Anfrage-Formulierung erlauben, Rechen-Operationen wie $+, \times$ zu verwenden, und dies dann intern in eine effizient auszuwertende $\mathrm{FO}(<)$-Anfrage umwandeln.

Von besonderem Interesse sind hier die so genannten Ordnungs-generischen Anfragen. In [2] wurde mit Hilfe einer neu entwickelten, Spiel-basierten Beweismethode, eine Reihe von Kollaps-Resultaten für solche Anfragen erzielt.

4 Komplexitätstheorie

In der Komplexitätstheorie wird die Schwierigkeit eines Problems durch den Zeit- oder Platzbedarf gemessen, der benötigt wird, um das Problem auf einer idealisierten Rechenmaschine zu lösen.

Charakterisierungen von Komplexitätsklassen

Fagins bahnbrechende Arbeit von 1974 hat diese Berechnungskomplexität mit der Beschreibungskomplexität in Verbindung gebracht: Er

Bild 7
Einige Klassen und Logiken.

$$\begin{array}{rcl} \mathsf{P} & \hat{=} & \mathsf{LFP} \\ \mathsf{AC}^0 & \hat{=} & \mathsf{FO}(+,\times) \\ \mathsf{TC}^0 & \hat{=} & \mathsf{FOunC}(+,\times) \end{array}$$

zeigte, dass ein Problem genau dann in der Klasse NP liegt, wenn es durch eine Formel der existentiellen Logik zweiter Stufe (Σ^1_1) beschrieben werden kann. Mittlerweile wurden die meisten Komplexitätsklassen auf solch deskriptive Art charakterisiert. Einige weitere Beispiele finden sich in Bild 7.

Viele Komplexitätsklassen lassen sich durch Fixpunktlogiken beschreiben. Da Σ^1_1 keine Fixpunktlogik ist, fällt sie aus diesem Rahmen etwas heraus. Daher wurde in [1] eine Fixpunktlogik eingeführt, deren Ausdrucksstärke genau NP, und damit Σ^1_1, entspricht.

Trennung von Klassen

Ein seit langem ungelöstes Problem ist es, Komplexitätsklassen (etwa P und NP) voneinander zu trennen. Ein prinzipieller Ansatz zum Beweis, dass ein Problem nicht in einer bestimmten Komplexitätsklasse liegt, besteht darin, zu zeigen, dass das Problem nicht durch eine Formel der zur Komplexitätsklasse gehörenden Logik beschrieben werden kann. Da dies notorisch schwierig ist, betrachtet man oft Teillogiken, die zwar ausdrucksstark genug sind, um interessante Probleme zu beschreiben, aber auch schwach genug sind, um Nicht-Ausdrückbarkeits-Beweise zuzulassen.

Ein Resultat aus [2] ist, dass die Logik FO in der *Presburger Arithmetik* unter einstelligen Zählquantoren abgeschlossen ist. Mit Hilfe dieses Resultats lässt sich dann relativ leicht zeigen, dass Graph-Probleme wie Zusammenhang oder Erreichbarkeit nicht durch FOunC(+)-Formeln beschrieben werden können, also nicht zur entsprechenden Teilklasse der Schaltkreis-Komplexitätsklasse TC^0 gehören.

Crane Beach-Vermutung

Die Crane Beach-Vermutung ist eng mit Kollaps-Resultaten aus der Datenbanktheorie verwandt. Sie ist nach dem Ort benannt, an dem sie erstmals formuliert wurde: Crane Beach, St. Philip, Barbados.

Die Vermutung bezieht sich auf Wortsprachen (d. h. Mengen von Strings) mit *neutralem Buchstaben*. Ein neutraler Buchstabe ist ein Buchstabe, der in einem String eingefügt oder gelöscht werden kann, ohne dass dies die Zugehörigkeit beziehungsweise Nichtzugehörigkeit des Strings zur Sprache ändert.

Für eine Liste A, zum Beispiel „$+, \times$“, von arithmetischen Operationen besagt die Crane Beach-Vermutung, dass Wortsprachen mit neutralem Buchstaben nur dann zur A-uniformen Version der Schaltkreis-Komplexitätsklasse AC^0 gehören, wenn sie bereits sternfrei-regulär sind. Übersetzt in die Terminologie von FO heißt das, dass Wortsprachen mit neutralem Buchstaben, die mit $\mathrm{FO}(<, A)$-Formeln beschrieben werden können, auch durch $\mathrm{FO}(<)$-Formeln beschreibbar sind.

In [2] wurden die Gültigkeit beziehungsweise Widerlegung der Crane Beach-Vermutung für unterschiedliche Wahlen von A und für verschiedene Erweiterungen sowie Teillogiken von FO nachgewiesen. Dadurch wird der Vergleich von Uniformitätsmaßen für Schaltkreis-Komplexitätsklassen möglich.

5 Verifikation

Ziel der computerunterstützten Verifikation ist die Entwicklung möglichst automatischer Verfahren zur Überprüfung, ob ein gegebener Prozess, zum Beispiel ein Hard- oder Softwaresystem, eine gewünschte Eigenschaft erfüllt. Ein besonders für die Hardwareverifikation erfolgreiches Verfahren ist das so genannte Model-Checking. Hierbei wird der betrachtete Prozess zunächst durch ein Transitionssystem modelliert, dessen Knoten die möglichen Zustände des Prozesses und dessen Kanten die durch Ereignisse ausgelösten Zustandsübergänge repräsentieren. Danach formuliert man die zu überprüfende Eigenschaft in einer Spezifikationssprache. Die Frage, ob die spezifizierte Eigenschaft in dem Prozess gilt, d. h. die Gültigkeit der erhaltenen Formel in dem Transitionssystem, wird dann automatisch überprüft.

Möchte man wissen, ob ein Prozess einen gewissen Fehlerzustand einnehmen kann, braucht man natürlich Spezifikationssprachen, die beliebig weit in die Zukunft sehen können. Daher werden auch in diesem Bereich Sprachen benutzt, die auf Logiken mit Fixpunktoperatoren basieren. Neben den wichtigen Sprachen LTL und CTL ist vor allem der modale μ-Kalkül, eine auf kleinsten Fixpunkten basierende Logik, im Verifikationsbereich sehr gut untersucht worden. Eine auf inflationären Fixpunkten basierende Sprache wurde allerdings bisher noch nicht betrachtet, obschon gerade in diesem Bereich die Möglichkeit, Aussagen leichter formulieren zu können, sehr interessant wäre. Eine solche Logik ist daher in [1] eingeführt und untersucht worden. Es zeigt sich, dass hier die inflationären Fixpunkte eine sehr viel größere Ausdrucksstärke bringen als kleinste Fixpunkte. Dies wird allerdings erkauft mit einer erheblich größeren Komplexität der algorithmischen Probleme.

Literatur

[1] S. Kreutzer. *Pure and Applied Fixed-Point Logics.* Dissertation, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, 2002. http://www.informatik.hu-berlin.de/~kreutzer.

[2] N. Schweikardt. *On the Expressive Power of First-Order Logic with Built-In Predicates.* Dissertation, Johannes Gutenberg-Universität Mainz, 2002. Logos Verlag Berlin, ISBN 3-8325-0017-0.

1

2

1 Dr. Stephan Kreutzer studierte Informatik an der Rheinisch-Westfälischen Technischen Hochschule Aachen. Seine Promotion schloss er im Jahr 2002 ab und erhielt dafür die Borchers-Plakette der RWTH Aachen sowie den GI-Dissertationspreis 2002. Von April bis September 2003 war er als Post-Doc an der University of Edinburgh, Schottland tätig. Seit Oktober 2003 ist er wissenschaftlicher Assistent am Lehrstuhl „Logik in der Informatik“ der Humboldt-Universität zu Berlin.
Adresse: Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Tel.: 030-2093-3075,
Fax: 030-2093-3081,
E-mail: kreutzer@informatik.hu-berlin.de

2 Dr. Nicole Schweikardt studierte Mathematik und Informatik an der Johannes Gutenberg-Universität Mainz. Ihre Promotion schloss sie im Jahr 2002 ab und erhielt dafür den Forschungsförderpreis der Freunde der Universität Mainz sowie den GI-Dissertationspreis 2002. Von Oktober 2002 bis August 2003 war sie als Post-Doc an der University of Edinburgh, Schottland tätig. Seit September 2003 ist sie wissenschaftliche Assistentin am Lehrstuhl „Logik in der Informatik“ der Humboldt-Universität zu Berlin.
Adresse: Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden, 6, 10099 Berlin, Tel.: 030-2093-3086,
Fax: 030-2093-3081,
E-mail: schweika@informatik.hu-berlin.de