Erscheint in: KI-93, 17. Fachtagung für Künstliche Intelligenz,
13.-16. September 1993, Humboldt-Universität zu Berlin

Neuere Entwicklungen der deklarativen KI-Programmierung

Harold Boley, François Bry, Ulrich Geske

DFKI Kaiserslautern, ECRC München, GMD-FIRST Berlin
boley@informatik.uni-kl.de, Francois.Bry@ecrc.de, geske@first.gmd.de

Deklarative Programme repräsentieren ihr Problemlösungswissen auf einer hohen Sprachebene, unabhängig vom Ausführungsmechanismus. KI-Sprachen verdichten die für Anwendungen der KI notwendigen Repräsentationsmethoden zu Programmierkonstrukten. Deklarative KI-Programmiersysteme haben einen wesentlichen softwaretechnischen Aspekt: Änderungsfreundliche Programme sollen ermöglicht und Freiheitsgrade für optimierende Compiler eröffnet werden. Geeignete Konzepte basieren auf funktionalen und logischen Formalismen und schließen Constraint- bzw. Taxonomiesysteme ein. Aktuelle Schwerpunkte bilden die Integration dieser Sprachen sowie ihre Kombination mit prozeduralen und objektorientierten Ausdrucksmitteln. Deklarative KI-Sprachen ermöglichen eine umfassendere Anwendung von Techniken der Programmtransformation und Metaprogrammierung.
Der Workshop wendet sich an Theoretiker und Praktiker, die Konzepte der Deklarativen KI-Programmierung weiterentwickeln und sie für KI-Anwendungen nutzen. Das Themenangebot erlaubt, einige neuere Entwicklungen dieses zukunftsweisenden Gebiets in konzentrierter Form kennenzulernen und zu bewerten.
Im technischen Programm des Workshops stehen eingereichte Beiträge und Tutorials zu aktuellen Teilgebieten im Mittelpunkt. Aus den eingereichten Beiträgen entstand nach einem Begutachtungs- und Überarbeitungsverfahren eine Zusammenstellung [3], die hier summarisch wiedergegeben wird: Vier Sessions beinhalten neben *Wissensrevision/Programmtransformation* die Teilgebiete *Typen* und *Constraints* zunächst einzeln und führen sie dann zusammen (Abschnitt 1). Drei Tutorials beinhalten *Deduktive Datenbanksysteme*, die *Programmiersprache Gödel* und die *Evolution von Wissensbasen* (Abschnitte 2 bis 4).

1 Wissensrevision/Transformation & Typen/Constraints

Die Session *Wissensrevision und Programmtransformation* beginnt mit G. Wagners Aktualisierungsoperationen (z.B. um schwach negierte Inputs: Kontraktion) auf Mengen von (z.T. stark negierten) Fakten, Disjunktiven Faktenbasen und Deduktiven Datenbanken [12]. Dann realisiert K. Hinkelmann die Ableitung von Konsequenz-Fakten aus logischen Programmen durch eine Erweiterung der "Magic Templates"-Transformation um Rückwärts- und Vorwärtspropagierung von Variablenbindungen initialer Fakten [6]. W. Goerigk und F. Simon behandeln schließlich die systemunterstützte Sourcecode-Transformation (Migrati-

on) von COMMON-LISP-Programmen in eine vollständig (in Objektfiles oder C-Programme) compilierbare Teilsprache [5].
In der Session *Typisierte Merkmalsstrukturen* präsentieren H.-U. Krieger und U. Schäfer eine (hierarchisch) typisierte merkmalsbasierte Sprache mit booleschen Verknüpfungen sowie zugehörige Typfolgerungsmechanismen [7]. J. Bedersdorfer et al. zeigen Ersetzungsregeln und komplexere Transformationen (z.B. mit boolesche Verknüpfungen und Sequenzen/Mengen) auf typisierten Merkmalsstrukturen [2]. G. Meyer und S. Weigel erläutern die statische Analyse streng (parametrisch polymorph) typisierter Merkmalsstrukturen und grenzen ihre (monotone) Typeinschränkung von der objektorientierten (nichtmonotonen) Typredefinition ab [9].
In der Session *Constraint-Systeme* diskutiert M. Meyer das Dilemma "Deklarativität vs. Effizenz" und seine Behandlung durch Constraints über endlichen (forward checking, (weak) looking-ahead) bzw. hierarchisch strukturierten (erweiterter HAC-Algorithmus) Domänen [10]. H.-J. Goltz und U. Geske verallgemeinern CLP, indem sie den Constraint-Solver durch ein Constraint-Handling-System ersetzen, das die Einflußnahme auf die Behandlung der Constraints ermöglicht und, außer Erfüllbarkeitstests, Operationen wie vorläufiges Akzeptieren der Variablenbelegung, Überprüfung notwendiger Bedingungen und Umformungen ausführen kann [4].
In der Session *Typen-Constraints-Kombinationen* schlägt H.C.R. Lock eine (für deterministische Programme) effizientere Alternative zur PROLOG-Operationalisierung der SLD-Resolution vor, bei der Ziele mit freien Variablen verzögert (Residuierung) und Termmengen für solche Variablen eingeschränkt werden (Typen-Constraints) [8]. A. Abecker und P. Hanschke argumentieren für die Effizienz und Modularität einer hybriden Architektur, die (beliebig viele, austauschbare) konkrete Domänen (spezielle Constraint-Solver) in ein terminologisches System einbettet, und dieses wiederum in DATALOG [1]. H. Wache und P. Tsarchopoulos integrieren Constraints über reellen und endlichen Domänen in eine terminologische Sprache, die (unter Zuhilfenahme eines erweiterten CLP-Schemas) in die Hornlogik eingebettet wird [11].

2 Deduktive Datenbanksysteme

F. Bry; ECRC, München

Seit mehr als einem Jahrzehnt beschäftigt sich die Datenbank-Forschung mit dem Gebiet der deduktiven Datenbanksysteme. Die Schwerpunkte liegen dabei sowohl auf der Untersuchung theoretischer Aspekte (für einen Überblick siehe [31, 32, 33, 34, 56, 27, 18, 44, 55, 23, 24, 41, 42]) als auch auf der Realisierung experimenteller Systeme (z.B. [49, 19, 28, 30, 35, 37, 46, 51, 57, 36, 59, 39, 47, 26]). Darüberhinaus werden derzeit, basierend auf Forschungsprototypen, industrielle Produkte entwickelt (z.B. [60]). In diesem Vortrag möchten wir Sie mit der Zielsetzung und den wesentlichen Techniken von deduktiven Datenbanksystemen vertraut machen.

Im Gegensatz zu herkömmlichen Datenbanksystemen, in denen Anwendungsdaten *extensional* beschrieben werden, erlauben deduktive Datenbanksysteme auch eine *intensionale* Definition. Der erste Teil des Vortrags stellt zwei sich ergänzende Konzepte vor, die in deduktiven Datenbanksystemen zur deklarativen Spezifikation einer Anwendung benutzt werden können. *Ableitungsregeln* auf der einen Seite erlauben *konstruktive* Definitionen, während *Integritätsbedingungen* auf der anderen Seite *normative* Bedingungen ausdrücken.
Anhand eines Beispiel, dem Flugplan einer Fluggesellschaft, werden wir zunächst zeigen wie eine Anwendung mittels Ableitungsregeln und Integritätsbedingungen intensional beschrieben werden kann. Dieses Beispiel macht die Vorteile einer intensionalen Beschreibung gegenüber einer konventionellen, rein extensionalen Beschreibung deutlich. Eine intensionale Beschreibung führt zu einer genaueren und natürlicheren Repräsentation der Anwendung. Sie ermöglicht eine kompaktere, platzsparende Spezifikation. Und sie ist einfacher zu warten.
Der zweite Teil des Vortrags beschäftigt sich dann mit der Auswertung von Ableitungsregeln bei der Anfragebeantwortung (z.B. [23, 24, 13, 14, 16, 17, 50, 52, 53, 54, 58, 21, 20]). Die Anfragebeantwortung in deduktiven Datenbanksystemen stellt Anforderungen an die Ableitungsprozeduren, die von klassischen Deduktionsmethoden wie etwa der SLD-Resolution nicht erfüllt werden. Daher wurden neue Methoden entwickelt, die sich wesentlich von den konventionellen Verfahren unterscheiden. Wir werden die Prinzipien dieser neuen Methoden vorstellen und ihre Vorteile bei der Anfragebeantwortung diskutieren.
Die effiziente Überprüfung von Integritätsbedingungen bei Änderungsoperationen bildet den dritten Teil des Vortrags. Wir werden die Prinzipien von verschiedenen Methoden zur Überprüfung von Integritätsbedingungen vorstellen (z.B. [23, 24, 22, 29, 38, 40, 43, 45, 48, 25]). Diese Methoden basieren entweder auf der Feststellung einer möglichen Verletzung der Integritätsbedingungen bezüglich Änderungsoperationen oder auf dem Propagieren von Änderungen.
Im letzten Teil des Vortrags argumentieren wir, daß es oft wünschenswert ist, nicht nur die Anwendung sondern auch Teile des Datenbanksystems selbst intensional zu beschreiben. Wir werden einige Beispiele für solche intensionalen Spezifikationen vorstellen.

3 Die Programmiersprache Gödel

U. Geske, J. Busse; GMD-FIRST, Berlin

3.1 Übersicht

Die Programmiersprache Gödel befindet sich an der Universität Bristol in Entwicklung ([61, 62]). Ihre Funktionalität und Ausdruckskraft orientiert sich an Prolog, ihre deklarative Semantik geht dagegen weit über die Entsprechung in Prolog hinaus. Die folgende Darstellung von Gödel beginnt nach der Diskussion der Motivation mit einer Übersicht über die wesentlichen Konzepte. Auf die

maßgebenden Eigenschaften und Begriffe (im Text hervorgehoben) wird im Anschluß etwas genauer eingegangen.

3.2 Motivation

Die Programmierung von Problemen mit Mitteln der Logik ist seit etwa 20 Jahren Gegenstand der Untersuchungen in der Logischen Programmierung. Die Vermeidung von Steuerelementen zur deklarativen Problembeschreibung und die Ausführbarkeit der Spezifikationen führten zu einem neuen Modell der Spezifikation, Testung, Ausführung und Wartung von Programmen. Das immer noch bedeutendste Programmiersystem der Logischen Programmierung ist Prolog. Daneben wurden eine Vielzahl weiterer logischer Programmiersprachen entwickelt, die aus Untersuchungen zu speziellen Richtungen resultieren, insbesondere Commited-Choice-Sprachen, Constraint-Sprachen, Spezifikationssprachen auf der Basis mehrsortiger Logiken und über die Horn-Klausel-Logik hinausgehender Logiken.
In Prolog erschweren die flache Programmstruktur, sequentielle Verarbeitung, Prozeduren mit Seiteneffekten, explizite Steuerelemente (!/0), die Realisierung der Negation und die Ununterscheidbarkeit von Objekt- und Metaprogrammierniveau eine deklarative Interpretation von Programmen. Die Konsequenzen sind die Einschränkung paralleler Verarbeitungsmöglichkeiten, die Schwierigkeiten in der Metaprogrammierung, die praktische Abweichung von theoretischen Aussagen über das Programmverhalten und die verminderte Verständlichkeit der Programme.

3.3 Konzepte

Durch die Programmiersprache Gödel sollen diese Nachteile von Prolog überwunden und die verschiedenen Entwicklungsrichtungen wieder zusammengeführt werden. Der Kerngedanke der Programmiersprache Gödel ist das von GÖDEL eingeführte Repräsentationskonzept - der Grund für die Wahl des Namens der Sprache, der aber auch als Akronym für *God's Own DEclarative Language* verstanden wird.
In der Programmiersprache Gödel soll ein Programm - so wie es in der Logischen Programmierung angestrebt wird - tatsächlich als eine Theorie und eine Programmabarbeitung als eine Deduktion aufgefaßt werden. Für die Darstellung der Theorie ist es erforderlich, die Problembeschreibungs-Niveaus der Objekt- und der *Metaprogrammierung* unterscheidbar zu halten, um eine deklarative Semantik zu sichern. Zusätzliche Mittel zur Problemspezifikation sind der Übergang zur mehrsortigen Logik (*Typen*) und die Problemstrukturierung durch *Modularisierung*. Die deduktive Programmabarbeitung (ohne außerlogische Effekte) basiert auf *flexibler Steuerung*, *Constraint-Lösen* und einer *verallgemeinerten Schnittoperation*. Eine Ausnahme bilden die *Ein-/Ausgabe*-Operationen, die weiterhin außerlogisch wirken, aber durch geeignete Verwendung der Modularisierung vom

deklarativen Teil des Programms weitestgehend separariert werden können.

3.4 Eigenschaften

Metaprogrammierung Ein Metaprogramm ist ein Programm (z.B. Interpreter, Programmtransformation), das ein anderes Programm als Daten benutzt. Gödel liefert Sprachkonstrukte und Repräsentationsformen, um zwischen Objektniveau- und Metaniveau-Ausdrücken zu unterscheiden. Dadurch können Metaprogramme als Theorien mit klarer deklarativer Semantik verstanden werden. Die Sprachkonstrukte betreffen die Prädikate var/1, nonvar/1, assert/1 und retract/1. Bei der Repräsentation werden Grund-Repräsentation (für Objektprogramme) und Nicht-Grund-Repräsentation (für Metaprogramme) unterschieden. Objektniveau-Variable werden durch Grund-Metaniveau-Terme dargestellt. Dadurch ist die System-Unifikation auf diese Ausdrücke nicht anwendbar und es müssen statt dessen entsprechende Anwenderprozeduren definiert werden. Der Vorteil liegt in der größeren Deklarativität und der leichteren Parallelisierbarkeit, der Nachteil in der ineffizienteren Verarbeitung durch von-Neumann-Rechner.

Typen Typisierung ist ein Mittel für exaktere Wissensrepräsentation und erlaubt darüber hinaus dem Compiler, effizienteren Code zu erzeugen. Der programmiertechnische Vorteil der Typisierung liegt in der Vermeidung von Programmierfehlern bzw. in der Entdeckung von Fehlern während der Syntaxanalyse und nicht erst bei der Abarbeitung auf Grund unerwarteter Abarbeitungsresultate.

Die Programmiersprache Gödel hat ein strenges Typkonzept. Die Typen basieren auf der mehrsortigen Logik 1.Stufe. Die Erweiterung besteht darin, daß Typvariablen vorhanden sind, die alle Typen zum Wert haben können. Durch diesen parametrischen Polymorphismus wird vermieden, daß für Argumente eines Terms, z.B. Elemente einer Liste, ein bestimmter Typ festgeschrieben werden muß.

Modularisierung Gödel verwendet das Konzept der abstrakten Datentypen, das neben dem Typsystem durch ein Modulsystem implementiert ist. Durch Module werden Namenskonflikte vermieden und Implementationsdetails verborgen. Die Systemmodule von Gödel stellen eine Reihe abstrakter Datentypen wie `List`, `String`, `Set`, `OProgram` (Objektprogramm) mit jeweils einer Menge von Operationen zur Verfügung.

Flexible Steuerung In Gödel erfolgt die Abarbeitung nicht sequentiell von links nach rechts, sondern ist durch Verwendung der DELAY-Steuer-Deklaration im generellen Abarbeitungsalgorithmus flexibel. Durch DELAY kann die Abarbeitung von Aufrufen bei Bedarf zurückgestellt werden. Dieser Mechanismus

ist die Grundlage für die bessere Behandlung der Negation, für Constraint-Abarbeitung, für eine effiziente Gestaltung und Steuerung der Abarbeitung und für die Sicherung der Korrektheit der Programmabarbeitung.

Constraint-Lösen Gödel erlaubt die Behandlung linearer und nichtlinearer Constraints im Bereich der ganzen und rationalen Zahlen.

Verallgemeinerte Schnittoperation Die Schnittoperation von Gödel baut auf dem Commitoperator ('|') in den Committed-Choice-Sprachen auf. Der Commit-Operator wirkt „vorwärts" und „rückwärts" auf die anderen Klauseln einer Prozedur, indem er deren Abarbeitung verhindert. Anders als in Prolog kann durch die Verwendung des Commit-Operators kein Test „eingespart" werden, so daß die logische Komponente eines Programms vollständig spezifiziert ist. Die logische Komponente eines Gödel-Programms kann durch Entfernen aller Schnittoperationen erhalten werden, während sie in Prolog nicht durch Weglassen der !/0-Aufrufe oder durch Einsetzung von Tests zu erhalten ist. Die Erweiterung des Commit-Operators zur Form $\{\texttt{Calls\_before\_Commit}\}_{Label}$ unterstützt die Ausführung von Folding/Unfolding, Partial-Evaluation und Programmtransformationen über Programmen, die die Schnittoperation enthalten. Wenn die Aufrufe `Calls_before_Commit` (die *Guards* in den Commited-Choice-Sprachen) erfolgreich abgearbeitet worden sind, wird die Abarbeitung aller anderen Klauseln der Prozedur, die einen Commit-Operator $\{\ldots\}_{Label}$ mit der gleichen Marke `Label` haben, verhindert. Klauseln einer Prozedur können Commit-Operatoren mit unterschiedlichen Marken besitzen. Die Marke kennzeichnet den Wirkungsbereich eines Commit-Operators. Aufrufe in `Calls_before_Commit` können selbst wieder Commit-Operatoren mit Marken sein.

Ein-/Ausgabe Durch die Verwendung des Modul-Systems mit der Beschreibung des Ein-/Ausgabe-Verhaltens eines Programms in Modulen, die weit oben in der Modulhierarchie angeordnet sind, kann die deklarative Darstellung des restlichen Programms gesichert werden.

4 Evolution von Wissensbasen

H. Boley, P. Hanschke, K. Hinkelmann, M. Meyer; DFKI, Kaiserslautern

Allgemein umfaßt die Evolution von Wissensbasen, kurz *Wissensevolution* oder *Evolution*, Techniken zur Steigerung der Güte formal repräsentierten Wissens. Sie ist 'unter' der *Wissensakquisition* und 'über' der *Wissenscompilation* angesiedelt, wobei es durchaus (fruchtbare) Grenzbereiche gibt: Während die *Akquisition* vorformales Wissen strukturiert und in eine formale Repäsentation abbildet, verbessert die Evolution eine bereits formale Repäsentation; und während die *Compilation* Wissen in Richtung auf die Maschine transformiert, verändert die Evolution es im Hinblick auf den Menschen.

Präziser definieren wir die **Evolution** als die **Validierung** und **Exploration** von Wissensbasen unter Verwendung von (weitgehend gemeinsamen) Analyse-Algorithmen:

Die *Validierung* prüft eine Wissensbasis in Bezug auf Redundanzen, Lücken, Widersprüche etc., z.B. mit Methoden der strukturellen/funktionalen Verifikation, Integritätsbedingungen, (Sub)Sorten-Prüfung und Anforderungsabschwächung/Verstärkung.

Die *Exploration* sucht nach interessanten Mustern und Zusammenhängen in einer Wissensbasis, um Wissenseinheiten zu abstrahieren, vervollständigen, induzieren etc., wobei diejenigen Methoden der Term-Abstraktion, Konzept-Formation, induktiven Inferenz, Abduktion und des entdeckenden Lernens usw. kombiniert werden, die von kleineren Beispielmengen auf größere Wissensbasen übertragbar sind.

Beide Teilbereiche der Evolution können u.a. in einem Wechselspiel zusammenwirken, bei dem explorierte Muster validiert werden, bevor sie (unter Benutzerkontrolle!) in die Wissensbasis zurückgespeist werden ("closed-loop learning" [74]).

Die *Repräsentationssprache* der Wissensbasis beeinflußt natürlich die Evolutionsalgorithmen. Zunächst unterstützt eine deklarative Formulierung des Wissens seine Evolution, da keine maschinenorientierten Artefakte die Analyse behindern und die gefundenen Ergebnisse das Wissen selbst, nicht seinen Zugriff, zum Inhalt haben. Dann gilt es, zwischen verschieden ausdrucksmächtigen deklarativen Repräsentationen abzuwägen, da auf schwächeren Sprachen i.a. mehr Eigenschaften durch Analysealgorithmen gefunden werden können, aber die gefundenen Muster evtl. nicht mehr in ihnen repräsentierbar sind. Somit lassen sich sprachliche und algorithmische Stufenfolgen zueinander in Entsprechung bringen [63].

Die *Analysealgorithmen* beinhalten eine abstrakte Interpretation [64], wodurch je nach Wahl der abstrakten Domäne zum Beispiel Informationen über redundante Wissenselemente oder Unvollständigkeiten in der Wissensbasis (fehlende Werte, Regeln usw.) gewonnen werden können. Die Ergebnisse der abstrakten Interpretation können dabei sowohl der Validierung (z.B. Aufdeckung von redundanten Regeln oder nichterfüllbaren Pämissen) als auch zur Exploration (z.B. Entdeckung von Aufrufstrukturen oder deterministischen Prädikaten) dienen.

Terminologische Wissensrepräsentationssysteme in der Tradition von KL-ONE [65, 70] tragen bereits in unveränderter Form zur Wissensbasisevolution bei: *Terminologische Inferenzdienste* [66] ermöglichen, ausgehend von einer Wissensbasis als einer Menge intensionaler Konzeptdefinitionen, z.B. die gleichzeitige Aufdeckung von Inkonsistenzen (Validierung) und Entdeckung von Vererbungsbeziehungen (Exploration) [68]. Varianten solcher Inferenzdienste können von KL-ONE-Sprachklassen auf weitere Repräsentationssprachen übertragen werden.

Die *induktive logische Programmierung* [69] ist eine Synthese der logischen Programmierung mit induktiven Verfahren. Bei der Theorierevision [73] werden

meist Lernverfahren eingesetzt, die eine gegebene Theorie aufgrund positiver und negativer Trainingsbeispiele verändern. Eine Charakterisierung der Verfahren ist je nach Grad der Verwendung von Hintergrundwissen möglich [72, 71]. Die Beispiele können entweder durch den Benutzer vorgegeben oder unter Verwendung von Hintergrundwissen auch automatisch generiert (bzw. in der Wissensbasis fokussiert) werden. Die Revision kann sowohl Aspekte der Validierung als auch der Exploration enthalten, je nachdem ob eine Modifikation der vorhandenen Theorie oder ihre Erweiterung um neue Klauseln notwendig ist. Weitere Methoden der induktiven logischen Programmierung sind die Anti-Unifikation und die inverse Resolution [67], welche die aus deduktiven Systemen bekannten Verfahren umkehren.

Literatur

1. Andreas Abecker, Philipp Hanschke. TaxLog: A Flexible Architecture for Logic Programming with Structured Types and Constraints. In Boley et al. [3].
2. Jochen Bedersdorfer, Karsten Konrad, Ingo Neis, Oliver Scherf, Jörg Steffen, Michael Wein. Eine Spezifikationssprache für Transformationen auf getypten Merkmalsstrukturen. In Boley et al. [3].
3. H. Boley, F. Bry, U. Geske (Hrsg.). *Proc. Workshop "Neuere Entwicklungen der deklarativen KI-Programmierung" auf der KI-93, Humboldt-Univ. zu Berlin*, Research Report RR-93-35, September 1993. DFKI Kaiserslautern.
4. Ulrich Geske, Hans-Joachim Goltz. Verallgemeinerte Behandlung von Constraints in einem CLP-System. In Boley et al. [3].
5. Wolfgang Goerigk, Friedemann Simon. Migration und Kompilation in Lisp: Ein Weg von Prototypen zu Anwendungen. In Boley et al. [3].
6. Knut Hinkelmann. Consequence Finding and Logic Programming. In Boley et al. [3].
7. Hans-Ulrich Krieger, Ulrich Schäfer. TDL – A Type Description Language for Unification-Based Grammars. In Boley et al. [3].
8. Hendrik C.R. Lock. Residuation and Type Constraints. In Boley et al. [3].
9. Gregor Meyer, Sybilla Weigel. Polymorphe Featuretypen - Typinferenz und Typüberprüfung. In Boley et al. [3].
10. Manfred Meyer. Finite Domain Constraints: eine deklarative Wissensrepräsentationsform mit effizienten Verarbeitungsverfahren. In Boley et al. [3].
11. Holger Wache, Panagiotis Tsarchopoulos. Ein erweitertes CLP-Schema für eine hybride Wissensverarbeitung. In Boley et al. [3].
12. Gerd Wagner. Update, Contraction and Revision in Knowledge Representation Systems. In Boley et al. [3].
13. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic Sets and Other Stange Ways to Implement Logic Programs. Proc. 5th ACM SIGMOD-SIGART Symp. on Principles of Database Systems (1986)
14. Bancilhon, F., Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing. Proc. ACM SIGMOD Conf. on the Management of Data (1986)
15. Beierle, C.: Knowledge Based PPS Applications in PROTOS-L. Proc. 2nd Logic Programming Summer School (1992)
16. Beeri, C.: Recursive Query Processing. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1989) (tutorial)

17. Beeri, C., Ramakrishnan, R.: On the Power of Magic. Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1987)
18. Bidoit, N.: Bases de Données Déductives. Armand Colin (1992) (in French)
19. Bocca, J.: On the Evaluation Strategy of Educe. Proc. ACM SIGMOD Conf. on the Management of Data (1986)
20. Bry, F.: Logic Programming as Constructivism: A Formalization and its Application to Databases. Proc. 8th ACM-SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1989)
21. Bry, F.: Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. Data & Knowledge Engineering 5 (1990) (Invited paper. A preliminary version of this article appeared in the proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases)
22. Bry, F., Decker, H., Manthey, R.: A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. Proc. 1st Int. Conf. on Extending Database Technology (1988)
23. Bry, F., Manthey, R.: Deductive Databases – Tutorial Notes. 6th Int. Conf. on Logic Programming (1989)
24. Bry, F., Manthey, R.: Deductive Databases – Tutorial Notes. 1st Int. Logic Programming Summer School (1992)
25. Bry, F., Manthey, R., Martens, B.: Integrity Verification in Knowledge Bases. Proc. 2nd Russian Conf. on Logic Programming (1991) (invited paper)
26. Cacace, F., Ceri, S., Crespi-Reghizzi, S., Tanca, L., Zicari, R.: Integrating Object-Oriented Data Modelling With a Rule-based Programming Paradigm. Proc. ACM SIGMOD Conf. on the Management of Data (1990)
27. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Surveys in Computer Science, Springer-Verlag (1990)
28. Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S., Tsur, S., Zaniolo, C.: The LDL System Prototype. IEEE Trans. on Knowledge and Data Engineering 2(1) (1990) 76–90
29. Decker, H.: Integrity Enforcement on Deductive Databases. Proc. 1st Int. Conf. Expert Database Systems (1986)
30. Freitag, B., Schütz, H., Specht, G.: LOLA – A Logic Language for Deductive Databases and its Implementation. Proc. 2nd Int. Symp. on Database System for Advanced Applications (1991)
31. Gallaire, H., Minker, J. (eds): Logic and Databases. Plenum Press (1978)
32. Gallaire, H., Minker, J., Nicolas, J.-M. (eds): Advances in Database Theory. Vol. 1. Plenum Press (1981)
33. Gallaire, H., Minker, J., Nicolas, J.-M. (eds): Advances in Database Theory. Vol. 2. Plenum Press (1984)
34. Gallaire, H., Minker, J., Nicolas, J.-M. (eds): Logic and Databases: A Deductive Approach. ACM Computing Surveys 16:2 (1984)
35. Haas, L. M., Chang, W., Lohman, G. M., McPherson, J., Wilms, P. F., Lpis, G., Lindsay, B., Pirahesh, H., Carey, M., Shekita, E.: Starburst Mid-Flight: As the Dust Clears. IEEE Trans. on Knowledge and Data Engineering (1990) 143–160
36. Jarke, M., Jeusfeld, M., Rose, T.: Software Process Modelling as a Strategy for KBMS Implementation. Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (1989)
37. Kiernan, G., de Maindreville, C., Simon, E.: Making Deductive Databases a Practical Technology: A Step Forward. Proc. ACM SIGMOD Conf. on the Management of Data (1990)

38. Kowalski, R. Sadri, F., Soper, P.: Integrity Checking in Deductive Databases. Proc. 13th Int. Conf. on Very Large Databases (1987)
39. Lefebvre, A., Vieille, L.: On Query Evaluation in the DedGin* System. Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (1989)
40. Lloyd, J. W., Sonenberg, E. A., Topor, R. W.: Integrity Constraint Checking in Stratified Databases. Jour. of Logic Programming 1(3) (1984)
41. Lloyd, J. W., Topor, R. W.: A Basis for Deductive Database Systems. Jour. of Logic Programming 2(2) (1985)
42. Lloyd, J. W., Topor, R. W.: A Basis for Deductive Database Systems II. Jour. of Logic Programming 3(1) (1986)
43. Martens, B., Bruynooghe, M.: Integrity Constraint Checking in Deductive Databases Using a Rule/Goal Graph. Proc. 2nd Int. Conf. Expert Database Systems (1988)
44. Minker, J. (ed.): Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1988)
45. Moerkotte, Karl, S.: Efficient Consistency Control in Deductive Databases. Proc. 2nd Int. Conf. on Database Theory (1988)
46. Morris, K., Ullman, J. D., Van Gelder, A.: Design Overview of the NAIL! System. Proc. 3rd Int. Conf. on Logic Programming (1986)
47. Naqvi, S., Tsur, S.: A Logical Language for Data and Knowledge Bases. Computer Science Press (1989)
48. Nicolas, J.-M.: Logic for Improving Integrity Checking in Relational Databases. Acta Informatica 18(3) (1982)
49. Nicolas, J.-M., Yazdanian, K.: Implantation d'un Système Déductif sur une Base de Données Relationnelle. Research Report, ONERA-CERT, Toulouse, France (1982) (in French)
50. Ramakrishnan, R.: Magic Templates: A Spellbinding Approach to Logic Programming. Proc. 5th Int. Conf. and Symp. on Logic Programming (1988)
51. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: CORAL: Control, Relation and Logic. Proc. Int. Conf. on Very Large Databases (1992)
52. Rohmer, J., Lescœur, R., Kerisit, J.-M.: The Alexander Method. A Technique for the Processing of Recursive Axioms in Deductive Databases. New Generation Computing 4(3) (1986)
53. Schmidt, H., Kiessling, W., Günther, H., Bayer, R.: Compiling Exploratory and Goal-Directed Deduction Into Sloopy Delta-Iteration. Proc. Symp. on Logic Programming (1987)
54. Seki, H.: On the Power of Alexander Templates. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1989)
55. Tsur, S.: A (Gentle) Introduction to Deductive Databases. Proc. 2nd Int. Logic Programming Summer School (1992)
56. Ullman, J. D.: Principles of Database and Knowledge-Base Systems. Vol. 1 and 2. Computer Science Press. (1988, 1989)
57. Vaghani, J., Ramamohanarao, K., Kemp, D., Somogyi, Z., Stuckey, P.: The Aditi Deductive Database System. Proc. NACLP Workshop on Deductive Database Systems (1990)
58. Vieille, L.: Recursive Query Processing: The Power of Logic. Theoretical Computer Science 69(1) (1989)
59. Vieille, L., Bayer, P., Küchenhoff, V., Lefebvre, A.: EKS-V1: A Short Overview. Proc. AAAI-90 Workshop on Knowledge Base Management Systems (1990)

60. Vieille, L.: A Deductive and Object-Oriented Database System: Why and How? Proc. ACM SIGMOD Conf. on the Management of Data (1993)
61. Hill, P.M., Lloyd,J.W.: The Gödel Programming Language. Report CSTR-92-27. Dept. CS, University of Bristol, Bristol BS8 1TR (1992)
62. Hill, P.M., Lloyd,J.W.: The Gödel Programming Language. The MIT Press. To appear (1993)
63. Harold Boley. Towards evolvable knowledge representation for industrial applications. To appear in: K. Hinkelmann, A. Laux (Eds.), Proc. DFKI-Workshop "Wissensrepräsentations-Techniken", DFKI Document, July 1993.
64. Andrew Bowles. Trends in applying abstract interpretation. *The Knowledge Engineering Review*, 7(2):157–171, 1992.
65. J. Brachman, R., G. Schmolze, J. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–216, 1985.
66. B. Hollunder. Hybrid Inferences in KL-ONE-Based Knowledge Representation Systems. In GWAI-90*; 14th German Workshop on Artificial Intelligence*, Band 251 von *Informatik-Fachberichte*, S. 38–47. Springer, 1990.
67. Peter Idestam-Almquist. Learning Missing Clauses by Inverse Resolution. S. 610–617.
68. Robert M. Mac Gregor. Using a Description Classifier to Enhance Deductive Inference. In *7th Conf. on AI Applications*, Band 1. IEEE, 1991.
69. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8:295–318, 1991.
70. Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, Band 422 von *LNAI*. Springer, 1990.
71. M. Pazzani, D. Kibler. the Utility of Knowledge in Inductive Learning. *Machine Learning*, 5:57–94, 1992.
72. J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266, 1990.
73. Bradley Richards, Raymond J. Mooney. First-order Theory Revision. Technischer Bericht AI 91-155, The University of Texas at Austin, Artificial Intelligence Laboratory, March 1991.
74. Stefan Wrobel. Demand-driven Concept Formation. In K. Morik (Hrsg.), *Knowledge Representation in Machine Learning*. Springer, 1989.

This article was processed using the LaTeX macro package with LLNCS style.