Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-93-35

Neuere Entwicklungen der deklarativen KI-Programmierung

Proceedings

Harold Boley, François Bry, Ulrich Geske (Eds.)

September 1993

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum
für
Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ❑ Intelligent Engineering Systems
- ❑ Intelligent User Interfaces
- ❑ Computer Linguistics
- ❑ Programming Systems
- ❑ Deduction and Multiagent Systems
- ❑ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Neuere Entwicklungen
der deklarativen KI-Programmierung — *Proceedings*

Harold Boley, François Bry, Ulrich Geske (Eds.)

DFKI-RR-93-35

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ 413-5839-ITW9304/3).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1993

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

Neuere Entwicklungen der deklarativen KI-Programmierung — Proceedings

Harold Boley, François Bry, Ulrich Geske (Eds.)

Abstract

The field of declarative AI programming is briefly characterized. Its recent developments in Germany are reflected by a workshop as part of the scientific congress KI-93 at the Berlin Humboldt University. Three tutorials introduce to the state of the art in deductive databases, the programming language Gödel, and the evolution of knowledge bases. Eleven contributed papers treat knowledge revision/program transformation, types, constraints, and type-constraint combinations.

Contents

Page

Neuere Entwicklung der deklarativen KI-Programmierung (H. Boley, F. Bry, U. Geske) — i

TaxLog: A Flexible Architecture for Logic Programming with Structured Types and Constraints (A. Abecker, P. Hanschke) — 1

Eine Spezifikationssprache für Transformationen auf getypten Merkmalsstrukturen (J. Bedersdorfer, K. Konrad, I. Neis, O. Scherf, J. Steffen, M. Wein) — 17

Migration und Kompilation in Lisp: Ein Weg von Prototypen zu Anwendungen (W. Goerigk, F. Simon) — 31

Verallgemeinerte Behandlung von Constraints in einem CLP-System (H.-J. Goltz, U. Geske) — 53

Consequence Finding and Logic Programming 61
(K. Hinkelmann)

TDL -- A Type Description Language for Unification-Based Grammars 67
(H.-U. Krieger, U. Schaefer)

Residuation and Type Constraints 83
(H.C.R. Lock)

Polymorphe Featuretypen - Typinferenz und Typüberprüfung 95
(G. Meyer, S. Weigel)

Finite Domain Constraints: eine deklarative Wissens-repräsentationsform mit effizienten Verarbeitungsverfahren 105
(M. Meyer)

Ein erweitertes CLP-Schema für eine hybride Wissensverarbeitung 112
(H. Wache, P. Tsarchopoulos)

Update, Contraction and Revision in Knowledge Representation Systems 121
(G. Wagner)

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

5. August 1993

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 (Migration) 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 Trans-

formationen (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) Typ**einschränkung** von der objektorientierten (nicht-monotonen) Typ**redefinition** 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

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.4.5 Constraint-Lösen

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

3.4.6 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.

3.4.7 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 Wisseneinheiten 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.

TaxLog: A Flexible Architecture for Logic Programming with Structured Types and Constraints

Andreas Abecker and Philipp Hanschke
DFKI
Postfach 2080, 67608 Kaiserslautern (Germany)
E-mail: {aabecker,hanschke}@dfki.uni-kl.de

Abstract

Terminological systems in the tradition of KL-ONE are handling declarative logic-based descriptions of conceptual knowledge. Most of these systems restrict their expressiveness and focus on (efficient) reasoning algorithms for certain services such as subsumption of concepts.

We present such a system, called **Taxon**, with sound and complete, terminating inference algorithms that extends the abstract concept language by *concrete domains*, such as predicates over rational numbers. The extension by concrete domains is similar in spirit to the extension of logic programming by constraints.

The decidability of the reasoning procedures imposes principal restrictions on the expressivity of such a formalism. To get maximal benefit of the terminological reasoning services, while being able to tackle representation problems in realistic applications, **Taxon** has been integrated with logic programming by applying a CLP scheme to its assertional formalism. The resultant language **TaxLog** replaces syntactical unification by semantical terminological inferences considering both abstract type information and concrete constraints. Thus, we have a logic programming language with a very powerful typing facility.

We discuss some main features of **TaxLog** focusing upon its three-layered architecture: 1) rules on the basis of a tuned vocabulary which is formulated in 2) a concept language that is grounded by 3) concrete domains.

1 Introduction

Terminological systems in the tradition of KL-ONE [9] are handling declarative logic-based descriptions of conceptual knowledge. Most of these systems restrict expressiveness of representation and power of inferences in order to get (efficiently) decidable reasoning services. In chapter 2 we present such a system, called **Taxon**, that extends the abstract logical formalism by integrating a special kind of constraint solvers (called *concrete domains*).

But because of the restricted language architecture (that was the price for decidability and efficiency), we are forced to embed the terminological formalism into another formalism to get a knowledge representation and reasoning system powerful enough for application programs.

This embedding is done in chapter 3. The resultant language **TaxLog** uses the terminological formalism as the basis for a simple rule language (definite function-free horn clauses, i.e. DATALOG). It replaces syntactical unification by semantical terminological inferences considering both abstract type information and concrete constraints. Thus, we have a logic programming (LP) language with a very powerful typing facility for variables. A small example will illustrate syntax and semantics of **TaxLog**.

Chapter 4 discusses some main features of **TaxLog**. Some stress will be laid upon the three-layered language architecture. This (easily extensible) architecture may be a good basis for the design of highly-integrated LP languages supporting expressive types and several constraint solvers. It promises a number of advantages under both efficiency and knowledge engineering aspects.

Chapter 5 sketches some related work. In chapter 6, we give a short summary, enumerate some impacts of the **TaxLog** idea, and sketch the future work.

2 Taxon: Terminological Reasoning with Concrete Domains

Traditionally, terminological systems provide two subsystems distinguished by the kind of knowledge they are representing:

- the purely terminological component (called *TBox*) allows to model the conceptual knowledge of a domain; it handles so-called *concepts* (that shall semantically denote sets of individuals of the application domain) und *roles* (binary relations between individuals)
- the assertional component (*ABox*) allows to reason about individuals and their relationships by instantiating the abstract conceptual structures defined in the TBox

We will start with **Taxon** [5, 7, 17], a terminological system that is especially characterized by two features:

1. it uses concrete domains
2. all reasoning services presented in the following are done by sound and *complete* algorithms

Concept Descriptions with Concrete Domains

In the TBox, the user compositionally defines concepts out of primitive concepts, previously defined concepts, and roles/attributes.[1] Concept forming operators are usually

[1] An attribute is a special form of role that should be functional, i.e. single-valued.

set-theoretic operators and value restrictions for roles and attributes. Example:

(PRIM male human)
(CONC man = male $\sqcap$ human)
(CONC female = $\neg$ male)
(CONC woman = human $\sqcap$ female)
(ROLE child)
(CONC mother = woman $\sqcap$ $\exists$ (child).human)
(CONC father = man $\sqcap$ $\exists$ (child).human)
(CONC foso = father $\sqcap$ $\forall$ (child).man)

Here, e.g., we introduce two primitive concepts and form a new concept as their conjunction. Then we declare a role and describe a mother as a woman with a human filler for the child role; a father of only sons is a man with only men as possible fillers for this role.

A drawback of this kind of concept descriptions is that all the terminological knowledge has to be expressed on an abstract logical level. In order to overcome this disadvantage, [5] proposed the integration of concrete domains into the terminological formalism:

(ATTR age)
(CONC foyso = foso $\sqcap$
$\forall$ (child age).<= 18)

Here, we assume a constraint solver processing inequations over real numbers to be installed. In general, a constraint solver suitable to be a concrete domain is characterized by:

- a set of individuals of this domain
- a set of concrete predicates over these individuals (this set must be closed under logical negation)
- a(n efficient) decision procedure for conjunctions of predicate instantiations (possibly instantiated by variables)

In the above example, our concrete domain provides a predicate that tests its argument for being less or equal to 18. We define a father of only young sons as a father of only sons with: the age of all his children has to be less or equal to 18.

This integration of concrete domains has several obvious advantages:

- use well-known user-convenient notions (e.g. inequations over real numbers) directly without a need for encoding them in first-order logic
- provide notions for domains that are hard to encode in small sets of logical axioms (especially under the language restrictions of a terminological system)
- integrate procedural knowledge into the general logical deduction process (concrete domains are black boxes for efficient implementations of special-purpose decision algorithms)

Thus, we have an extension of an abstract logical formalism by special-purpose reasoners that is rather similar to the evolution step from LP to *constraint logic programming* (CLP) [29].

Definition 2.1 (interpretations and models) *An interpretation $\mathcal{I}$ consists of*

- *a non-empty abstract domain $dom(\mathcal{I})$ that is disjoint to the concrete domain $dom(\mathcal{D})$*
- *an interpretation function $.^{\mathcal{I}}$ that associates*
  - *with each concept name A a subset $A^{\mathcal{I}}$ of $dom(\mathcal{I})$*
  - *with each attribute name f a partial function $f^{\mathcal{I}} : dom(\mathcal{I}) \rightarrow dom(\mathcal{I}) \cup dom(\mathcal{D})$*
  - *with each role name a binary relation $r^{\mathcal{I}} \subseteq dom(\mathcal{I}) \times (dom(\mathcal{I}) \cup dom(\mathcal{D}))$*

An abstract predicate ρ of arity n is interpreted as $\rho^{\mathcal{I}} \subseteq dom(\mathcal{I})^n$ and $(\neg\rho)^{\mathcal{I}}$ as $dom(\mathcal{I})^n \backslash \rho^{\mathcal{I}}$, and a concrete predicate ρ is interpreted as $\rho^{\mathcal{I}} = \rho^{\mathcal{D}}$. If $u = f_1 ... f_n$ is an attribute chain, then $u^{\mathcal{I}}$ denotes the composition $f_1^{\mathcal{I}} \circ ... \circ f_n^{\mathcal{I}}$ of partial functions.[2]

The interpretation function can be extended to arbitrary concept terms as follows:

1. $(s \sqcap t)^{\mathcal{I}} = s^{\mathcal{I}} \cap t^{\mathcal{I}}$, $(s \sqcup t)^{\mathcal{I}} = s^{\mathcal{I}} \cup t^{\mathcal{I}}$, *and* $(\neg s)^{\mathcal{I}} = dom(\mathcal{I}) \setminus s^{\mathcal{I}}$,
2. $(\forall u_1, \cdots, u_n.\rho)^{\mathcal{I}} = \{x \in dom(\mathcal{I}) |$ *for all* $y_1, \cdots, y_n$ *with* $(x, y_1) \in u_1^{\mathcal{I}}, \cdots, (x, y_n) \in u_n^{\mathcal{I}}$ *we have* $(y_1, \cdots, y_n) \in \rho^{\mathcal{I}}\}$
3. $(\exists u_1, \cdots, u_n.\rho)^{\mathcal{I}} = \{x \in dom(\mathcal{I}) |$ *there exist* $y_1, \cdots, y_n$ *with* $(x, y_1) \in u_1^{\mathcal{I}}, \cdots, (x, y_n) \in u_n^{\mathcal{I}}$ *and* $(y_1, \cdots, y_n) \in \rho^{\mathcal{I}}\}$

An interpretation $\mathcal{I}$ is a model of the T-box $\mathcal{T}$ iff it satisfies $A^{\mathcal{I}} = t^{\mathcal{I}}$ for all terminological axioms $(CONC\ A = t)$ in $\mathcal{T}$.

TBox Reasoning

The most important inference service concerning the terminology is computing the subsumption relation between concepts.

Definition 2.2 *A concept s* subsumes *a concept t wrt. a terminology $\mathcal{T}$ iff for all models $\mathcal{I}$ of $\mathcal{T}$ $s^{\mathcal{I}}$ is a superset of $t^{\mathcal{I}}$.*

The *classification* service arranges all terminology concepts in a partially ordered structure wrt. subsumption. This structure can serve as a fast subsumption cache for other reasoning services. Another important inference service determines whether two concept descriptions denote *disjoint* sets in all models. Example:

(CONC fovyso = foso ⊓
∀ (child age).< 8)
(CONC foos = foso ⊓
∀ (child age).> 50)

Here, we define some new concepts, for fathers of very young sons and fathers of only old sons. Calling the concrete domain algorithms, **Taxon** will compute for example, that:

[2] Composition is from left to right: $(f \circ g)(x) = g(f(x))$

- `foyso` subsumes `fovyso`
- `foys` and `foos` are disjoint (and hence, `fovyso` and `foos`, too)
- `mother` and `foso` are disjoint

For space reasons, we could only present a single very simple concrete domain. But, please note, that the extension scheme [5] allows several concrete domains to be installed at the same time. Some useful examples for a practical application could be:

- the above mentioned domain of real numbers with comparison operators, but plus arithmetic,
- a finite domain constraint system like e.g. CONTAX [7] (such a use of finite domain constraint systems as a declarative knowledge representation tool could be particularly worthful e.g. in solving configuration tasks),
- a relational database system (this is rather important when trying to integrate declarative programming into existing industrial information systems), or
- some notions of space and time (that some people try to support directly by the (termino)logical formalism [26]).

ABox Reasoning

ABox reasoning moves the level of abstraction from the conceptual to the individual knowledge. It allows instantiations (so-called *assertions*) of the conceptual and relational structures defined in the TBox.

Definition 2.3 (ABox syntax) *Let a, b, a_i be individual names, C a concept and R a role or attribute, then*

$a = b$	(equality),
$a \neq b$	(negated equality),
$a : C$	(membership assertion),
$(a, b) : R$	(role/attribute-filler assertion), *and*
$\rho(a_1, \cdots, a_n)$	(predicate assertion)

(where ρ can be an abstract, a negated abstract, or a concrete predicate of arity n)
are assertional axioms. *An* ABox *is a finite set of assertional axioms.*

An $\mathcal{I}$-variable assignment α wrt. an interpretation $\mathcal{I}$ is a partial mapping from individual names to $dom(\mathcal{I})$.

Definition 2.4 (interpretations and models) *An interpretation $\mathcal{I}$ and a variable assignment α* satisfy an equality (a negated equality) $a = b$ *($a \neq b$) if $a\alpha = b\alpha$ ($a\alpha \neq b\alpha$), they* satisfy a membership assertion $a : C$ *if $a\alpha \in C^{\mathcal{I}}$, they* satisfy a role/attribute-filler assertion $(a, b) : R$ *if $(a\alpha, b\alpha) \in R^{\mathcal{I}}$, and they* satisfy a predicate assertion $\rho(a_1, \cdots, a_n)$ *if $(a_1\alpha, \cdots, a_n\alpha) \in \rho^{\mathcal{I}}$.*

An interpretation $\mathcal{I}$ is called a model *of $\mathcal{A}$ wrt. a terminology $\mathcal{T}$ if it is a model of $\mathcal{T}$ and there is an assignment α such that all assertional axioms in $\mathcal{A}$ are satisfied.*

Definition 2.5 (ABox services) *Given a TBox $\mathcal{T}$, and an ABox $\mathcal{A}$, $\mathcal{A}$ is called* consistent *if it has a model wrt. $\mathcal{T}$.*

Theorem 2.6 *The consistency test for ABoxes is decidable.*

A more well-founded discussion of this result and other terminological reasoning services can be found e.g. in [5, 17]. Example:

This ABox is inconsistent for two independent reasons:

(fovys tom)
(tom child tina)
(tina age 13)
(mother tina)

- there is an 'abstract contradiction' because all children of tom have to be men, which are disjoint to mothers.
- there is a 'concrete problem' because tina's age is too high for a very young son.

Thus, we have a quite comfortable tool for representing a special kind of knowledge and reasoning about it. But it's a special-purpose system; if we want to implement real applications [19] we need a general-purpose reasoner with the computational power of a Turing machine.[3] This is the starting-point for **TaxLog**: take **Taxon** as a special-purpose reasoner and embed it into a horn clause interpreter, the simplest possible incarnation of the LP paradigm.[4] Or, from another point of view: take a horn logic interpreter as a general, but very 'uninformed' inferential machine and 'tune' it by a very smart, powerful typing system.[5] This will be done in the next chapter.

3 TaxLog: Embedding Taxon into Clauses

Definition 3.1 (program) *Let* P *be a set of new predicate names.* $r(X_1, \ldots, X_n)$ *is defined to be an* atom *iff* $r \in$ P *is an n-ary predicate name, and* $X_1, \ldots, X_n$ *are n* pairwise different *variables. A* goal *is a (possibly empty) sequence of atoms and assertional axioms; and a (program)* clause *is an expression* $B \leftarrow G$*, where G denotes a goal and B is an atom. A* program *consists of the terminology $\mathcal{T}$ and a set of* clauses.

This example shows that a **TaxLog** clause contains a relational and a constraint part (the rule ABox). These parts could be separated (for analysis purposes):

```
p(X) ← q(Y), X:foos, (X,Y):child
q(Y) ← Y:mother
q(Y) ← Y:man, (Y,A):age, <(A,19)
q(Y) ← Y:father
```

```
p(X) ← q(Y) & X : foos, (X,Y) : child
```

[3] This may be a point of discussion. An influential idea in the terminological system discussion was Vilain's *restricted language architecture* for hybrid reasoning [30]: take several restricted but efficient systems and get a reasonable inferential power from their coupling and cooperation.

[4] It should be noted that we will use definite function-free horn clauses (DATALOG) that are stand-alone not as powerful as a Turing machine. But they will be expressive enough in combination with **Taxon**, because **Taxon** can simulate arbitrary Herbrand terms.

[5] Indeed, we have in some sense the most powerful reasonable typing formalism, because terminological systems like **Taxon** move their language expressiveness as near as possible to the cliff of undecidability.

But, on the other side, it may be more intuitive to bring both parts closer together:

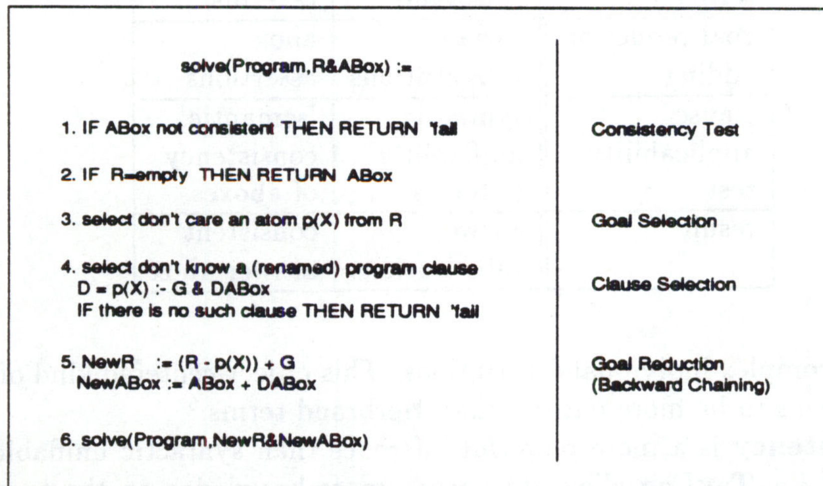
Definition 3.2 (ϕ-term) *Assume a terminology to be given. ϕ-terms can inductively be defined as follows: For a variable X and a concept t, X and $X : t$ are ϕ-terms. Each constant of a concrete domain is a ϕ-term. If $s_1, \ldots s_n$ are ϕ-terms, t is a concept term, and $R_1, \ldots, R_n$ are role/attribute names then the following expression is a ϕ-term:* $X : t\{R_1 \Rightarrow s_1, \ldots, R_n \Rightarrow s_n\}$.

There are two possible interpretations of ϕ-terms: (i) as a structured object-centered representation like records, or frames, or (ii) as a declarative way of describing a set of individuals constraining possible interpretations by conceptual and relational conditions.

Using ϕ-terms in the above example yields:

```
p(X:foos {child ⇒ Y}) ← q(Y)
q(Y:mother)
q(Y:man {age ⇒ A}) ← <(A,19)
q(Y:father)
```

TaxLog programs are executed by the following interpreter:



The example program above and a goal `p(X)` would cause the following computation:

1. select the first program clause, so get a new goal `q(Y)` and the ABox `{X:foos, (X,Y):child}`
2. the ABox is consistent; so select goal `q(Y)`; try the second program clause and get the ABox `{X:foos, (X,Y):child, Y:mother}`
3. this ABox is inconsistent, so do backtracking and try the third clause; new ABox `{X:foos, (X,Y):child, Y:man, (Y,A):age, <(A,19)}`
4. this ABox is inconsistent, too, so try the last clause; new ABox `{X:foos, (X,Y):child, Y:father}`
5. this ABox is consistent, the goal stack is empty, hence return the ABox as result

A declarative semantics for **TaxLog** can be easily obtained applying the Höhfeld-Smolka-Scheme for constraint logic programming [21] to the **Taxon** ABox formalism (as the underlying constraint language).

The notion of terminological interpretations can be extended in a straightforward manner to program clauses reading $B \leftarrow G$ as the universal closure $\forall(B \leftarrow G)$. Then, we can characterize the meaning of an answer ABox to a program $\mathcal{P}$ and a goal:

Soundness: every assignment α wrt. a model $\mathcal{I}$ of $\mathcal{P}$ that satisfies an answer ABox also satisfies the goal.

Completeness: if an assignment α wrt. a model $\mathcal{I}$ of $\mathcal{P}$ satisfies the goal, then there exists a computed answer ABox satisfied by $\mathcal{I}, \beta$, for some β extending α.

A more detailed discussion of this ***possible world semantics*** can be found in [1].

The step from a conventional LP language like Prolog to **TaxLog** is illustrated in the following table:

	Prolog	**TaxLog**
terms	Herbrand	ϕ-terms
goal reduction adding	variable substitutions	abox assertions
clause applicability test	syntactic unifiability of terms	'semantic' consistency of aboxes
result	answer substitutions	consistent answer abox

Discussion:

ϕ-terms are complex individual descriptions. This object-centered kind of knowledge representation seems to be more natural than Herbrand terms.[6]

ABox consistency is a more powerful inference than syntactic unifiability test for Herbrand terms,[7] i.e. **TaxLog** allows to encode more knowledge on the term (or, rule) level. Moreover, special algorithms for inheritance and concrete domains support the decision procedure.

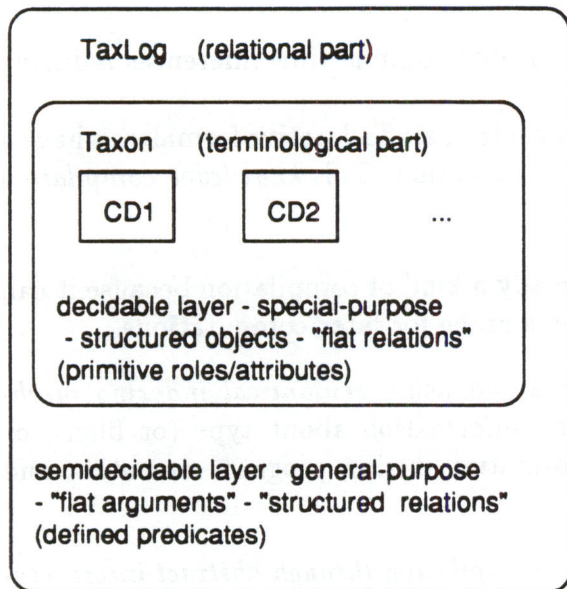
Consistent ABoxes as an answer formalism represent more information (concept membership, role/attribute-relations, concrete constraints, equalities) than answer substitutions (only equalities) and they represent less information (no explicit answer but only necessary conditions within minimal models because of our open world assumption). Similar to CLP, we have an implicit characterization of answers instead of explicit enumeration. An advantage over ordinary CLP languages may be, that this implicit characterization is done in terms of the user-defined vocabulary. Instead of constraint simplification mechanisms, we have **Taxon** ABox reasoning for sophisticated answer analysis.

---

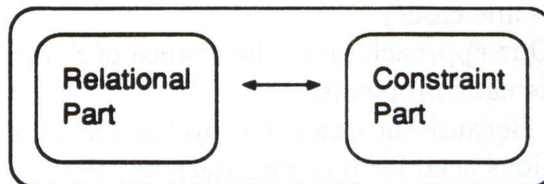
[6] Our notion comes from the notion of ψ-terms in LOGIN [3] that implement *feature-terms*, a kind of knowledge representation often used in natural language processing.

[7] An argument for this is that the first one has only PSPACE-complete decision algorithms whereas the latter can be decided with linear complexity. Another argument may be, that Herbrand terms can be simulated using concepts and attributes, whereas **Taxon** additionally provides roles and concrete predicates in assertions.

4 Some Features of the TaxLog Architecture



TaxLog *Architecture*



Standard CLP Architecture

A main characteristic of **TaxLog** is its layered language architecture with (arbitrary many) concrete domains integrated into the relational language (that is the user interface) via the concept language as an intermediate level.
In contrast to this, usual CLP languages have a flat architecture with mostly only one constraint solver and without types.

This seems to be a very small technical difference, but we will argue that this layered architecture provides a number of advantages in several fields. For space reasons, we will only sketch them briefly. Some further elaborations can be found in [1, 2]. But in parts, these are still first considerations that need further work.

Efficiency Aspects.

Principally, **TaxLog** has efficiency gains because it employs both types *and* concrete constraints solvers. Hence, there are *two* ways for the integration of procedural knowledge:

1. special algorithms for type inferences support the LP inference machine
2. special algorithms for concrete domains support the type inference machine

This gives good chances for an *early failure detection* because of type inconsistencies: special type inferences can detect early in a derivation branch problems with incompatible type constraints for individuals.

One possible problem should be mentioned: **TaxLog** with its possible world semantics considers consistent, i.e. *satisfiable* ABoxes in each derivation step. Therefore, an inconsistency cannot be shown by a single counter example[8] but forces perhaps investigation of many possible models. This may cause a loss of efficiency greater than the gains obtained by our special-purpose reasoners. Nevertheless, we think that **TaxLog** should be further examined, because:

[8] as in horn logic with its Herbrand interpretations

– Only practical experience with a good prototype can show how serious this problem is in real applications.
– This is the price of open world reasoning (that seems to be appropriate (and necessary) in some cases).
– Our approach gives the chance of doing sophisticated compile-time inferences reducing efficiency problems:

Because they do not allow the specification of control, all declarative formalisms have a serious need for precomputations prior to program execution. This *knowledge compilation* can be done on several levels:

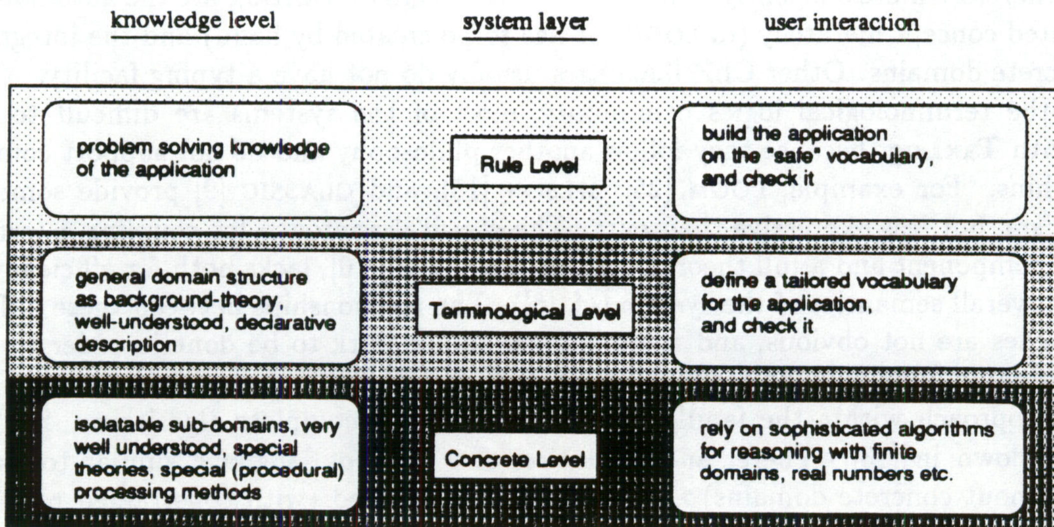
- The *classification* of **Taxon** concepts is already a kind of compilation because it can result in a fast subsumption and disjointness cache for later computations.
- Another efficiency improvement can be reached using *semantical indexing mechanisms* [28, 1]: sort program clauses using information about type (or filler-, or concrete) constraints, determine this information during program execution and access directly to the appropriate clause.
- The most promising technique is *knowledge compilation through abstract interpretation* [8]: the basic idea is to get information about program behaviour by program execution over an abstract domain; each value of this abstract domain represents a class of possible 'real' variable values. Such reasoning about classes decreases complexity.
  The obtained information may be e.g. knowledge about bindings of variables, generation of attribute fillers, or about deterministic behaviour of procedures. This information is then used for generating an optimized program code.
  Our hope is, that **TaxLog** improves such analysis possibilities providing a more fine-grained abstract domain, namely the concept hierarchy. Moreover, it allows to use precomputed conceptual knowledge (like subsumption relations) and to infer more detailed information (by using sophisticated terminological type inferences).
  A problem could be finding an appropriate granularity of the abstract domain because abstract interpretation over a very refined taxonomy leads back to the problem of a total precomputation.

Theoretical and Implementational Aspects.

TaxLog is a modular system with small, well-defined interfaces between its components. This allows clean implementation, maintenance and extension, e.g. by new concrete domains or new terminological language constructs. This modularity carries over to the theoretical work. Expanding the language, you have not to do full soundness and correctness proofs; it is enough to show features of the new component (for example that a new concrete domain is admissible).

Knowledge Engineering Aspects.

- The system architecture corresponds to a similar organization on the knowledge level:



- This shows two ways for building an appropriate representation language for a special application domain:
  - **Taxon** provides a rich concept language to define an application specific vocabulary for typing,
  - the concept language itself can be tailored for a certain application by installation of the appropriate concrete domains.
- This supports an interactive rapid prototyping of knowledge bases:
  - each level of the system allows interactive development and debugging by its own algorithms,
  - each level is clearer (because of its smaller interface) than a flat language with several components one beside the other, and
  - the knowledge of each level can be used to make debugging at next higher level simpler and more efficient.
- The level architecture supports reuse of knowledge: the terminology is usually application independent, concrete domains even domain independent. At least these parts are easily separatable and reusable for other application programs.
- **Knowledge Base Validation [24]**: large knowledge bases tend to be full of inconsistencies, redundancies etc. **TaxLog** allows type declarations for program clauses and sophisticated type inferences based on them. Already on the rule level, **TaxLog** facilitates knowledge base debugging, because our typed and constrained rules contain more information than usual. This constraint information can be isolated from the relational part, and used for decidable inferences.

5 Related Work

The most prominent related work in LP is LOGIN [3].[9] It implements a *feature logic* [27], i.e. its typing instrument is less expressive than ours; it remains at the syntactical level, but is therefore more efficiently to handle.[10] Advantages of **TaxLog** are the automatically computed concept hierarchy (in LOGIN, it has to be created by hand) and the integration of concrete domains. Other CLP languages usually do not have a typing facility.

In the terminological logics community, most of the systems are difficult to compare with **TaxLog**, because they follow another philosophy and do not support complete algorithms. For example, LOOM [25], MESON [15], and CLASSIC [9] provide some rule formalism, but less expressive than ours. The classical approach for coupling a terminological component and a full theorem-prover, KRYPTON [10], lacks both for efficiency and a clear overall semantics of the system (cf. [6]). The relationships between these different approaches are not obvious, and there is some future work to be done in order to get a reasonable unifying view on terminological systems (cf. [20]).

An approach within the family of systems based on complete algorithms has been written down in [13]. There, an integration of a concept language similar to **Taxon** (but without concrete domains) and DATALOG is proposed (with constrained resolution as inference procedure) as the basis for a deductive database. This approach is very similar to our idea. It confirms our feeling, that deductive databases may be a good field of application for **TaxLog**.

An integration of **Taxon** and forward-chaining rules and its use for abstraction processes is described in [19]. A general theoretical framework integrating rules and a terminological formalism is elaborated in [18].

Some more theoretical work can be found in [12, 4] where the authors present *constrained resolution* as a CLP-scheme based on the open world assumption, that even generalizes the Höhfeld-Smolka-Scheme. They also propose the integration of a terminological logic as constraint system. In [22, 23, 11], it is shown how terminological logics can serve as a basis for abductive reasoning in solving configuration tasks. See [16, 31], how hybrid formalisms close to the ideas of the presented paper may contribute to configuration.

6 Conclusions

We have presented a new CLP language that incorporates relational LP with object-centered terminological reasoning.[11] The architecture of the language promises both theoretical and pragmatic advantages. Therefore, it may be a starting point towards a methodology for the construction of next generation declarative programming languages.

[9] LOGIN was a starting point for the development of the Höhfeld-Smolka-Scheme. It shows that order-sorted types are some kind of constraint information.

[10] It may be a point of discussion how expressive a typing facility should be.

[11] Note, that this coupling resembles to the embedding of descriptive relational database query languages (SQL) into procedural languages (C) to have both the efficiency of the DB system and the computational power of C.

Some common patterns concerning the possible advantages of the architecture in several fields are:

- lift things from a syntactical to a semantical level,
- provide *two* points of application for analysis or improvement approaches where conventional systems have at most one, and
- use a modular architecture to support divide-and-conquer approaches to efficiency improvement, program analysis, user convenience, ...

In order to make logic programming, or, more general, declarative programming, more suitable in practice, we should start a discussion about construction principles for heterogeneous hybrid reasoning systems. To such a discussion, **TaxLog** could contribute the following ideas:

- Both order-sorted types and (flat) constraints should be integrated into the LP framework to enhance both efficiency and cognitive (epistemological) adequacy.
- This integration should be done in a declarative, theoretically clean way, and not only as an ad hoc solution. It should be flexible and extensible.
- Terminological systems provide automatic classification and sophisticated type inferences. We should investigate their usefulness for LP.
- We should try to use constraint solvers not only for enhancing interpretive runtime efficiency but also for supporting sophisticated program analyses prior to runtime, like compilation and validation.
- For this purpose, an architecture like **TaxLog** allows the separation of a large part of knowledge for decidable analysis algorithms.

A number of the presented ideas are still first considerations that need further work. **Taxon** has been implemented successfully but only with two very simple concrete domains. Some more interesting concrete domains could show the practical relevance of the scheme. A prototype of **TaxLog** is running, but *very* slow. New ideas for fast ABox reasoning (under work [14]) must be the starting point for more efficient implementations. Principal theoretical considerations concerning (i) the building of hybrid declarative architectures and their semantics (ii) compilation and validation methods could provide exciting work.

Acknowledgements

We would like to thank Manfred Meyer who encouraged us to write down the ideas presented here relating **TaxLog** with the field of constraint processing. We also thank Christof Sprenger for carefully reading a draft and Holger Wache for some helpful discussions. Part of the work described was supported by the German "Bundesministerium für Forschung und Technik" under grant ITW 8902 C4.

References

[1] A. Abecker. TaxLog: Taxonomische Wissensrepräsentation und logisches Programmieren. Diploma thesis, Universität Kaiserslautern, 1993. To appear as DFKI Document. In German.

[2] A. Abecker and Ph. Hanschke. TaxLog: A flexible architecture for logic programming with structured types and constraints. To appear as DFKI Technical Memo, 1993. A long version of this paper.

[3] H. Aït-Kaci and R. Nasr. Login: a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[4] F. Baader, H.-J. Bürckert, B. Hollunder, W. Nutt, and J.H. Siekmann. Concept logics. DFKI RR-90-10, 1990.

[5] F. Baader and Ph. Hanschke. A scheme for integrating concrete domains into concept languages. In *IJCAI-91*, 1991. Also as DFKI RR-91-10.

[6] C. Beierle, U. Hedtstück, U. Pletat, and J. Siekmann. An order-sorted logic for knowledge representation systems. IWBS-Report 113, IBM Deutschland, 1990.

[7] H. Boley, Ph. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: a hybrid compilation laboratory. In *3rd International Workshop on Data, Expert Knowledge and Decisions*, September 1991. To appear in a special issue of 'Annals of Operations Research'. Also as DFKI RR-93-03.

[8] A. Bowles. Trends in applying abstract interpretation. *The Knowledge Engineering Review*, 7(2), 1992.

[9] R.J. Brachman, D.L. McGuiness, P.F. Patel-Schneider, L. Alperin Resnick, and A. Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In *J. Sowa: Principles of Semantic Networks.* Morgan Kaufmann, 1991.

[10] R.J. Brachmann, R.E. Fikes, and H.J. Levesque. KRYPTON: Integrating terminology and assertion. In *AAAI-83*, pages 31–35, 1983.

[11] M. Buchheit, W. Nutt, and R. Klein. A model construction approach towards configuration. Draft. Unpublished, 1993.

[12] H.-J. Bürckert. A resolution principle for clauses with constraints. DFKI RR-90-02, 1990.

[13] F. Donini, M. Lenzerini, D. Nardi, W. Nutt, and A. Schaerf. A hybrid system with Datalog and concept languages. Draft, 1991.

[14] D. Drollinger. Intelligentes Backtracking in Inferenzsystemen am Beispiel terminologischer Logiken. Diploma thesis, Universität Kaiserslautern, 1993. Forthcoming. In German.

[15] J. Edelmann and B. Owsnicki. Data models in knowledge representation systems: A case study. In *GWAI-86 und 2. Österreichische AI Tagung*. Springer, 1986.

[16] Th. Frühwirth and Ph. Hanschke. Terminological reasoning with constraint handling rules. In *First Workshop on Principles and Practice of Constraint Programming*, 1993. Also as DFKI D-93-01.

[17] Ph. Hanschke. Specifying role interaction in concept languages. In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan Kaufmann Publishers, 1992. Also as DFKI RR-92-37.

[18] Ph. Hanschke. *A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning*. PhD thesis, Universität Kaiserslautern, 1993. Forthcoming.

[19] Ph. Hanschke and K. Hinkelmann. Combining terminological and rule-based reasoning for abstraction processes. In *GWAI-92*. Springer, 1992. Also as DFKI RR-92-40.

[20] J. Heinsohn, D. Kudenko, B. Nebel, and H.-J. Profitlich. An empirical analysis of terminological representation systems. DFKI D-92-16, 1992.

[21] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. Lilog-Report 53, IBM Deutschland, 1988.

[22] R. Klein. An approach to the integration of term description languages and clauses. In B. Nebel, Ch. Peltason, and K. von Luck, editors, *International Workshop on Terminological Logics*. DFKI D-91-13, 1991.

[23] R. Klein. Constructive problem solving: An abductive formalization. In *German Workshop on Inference Systems*, 1991. Also as Daimler-Benz Technical Report.

[24] B. Lopez, P. Meseguer, and E. Plaza. Knowledge based systems validation: A state of the art. *AICOM*, 3(2):58–72, June 1990.

[25] R. MacGregor and M.H. Burstein. Using a description classifier to enhance knowledge representation. *IEEE Expert*, June 1991.

[26] A. Schmiedel. A temporal terminological logic. In *8th AAAI*, volume 2, pages 640–645, 1990.

[27] Gert Smolka. A feature logic with subsorts. LILOG-Report 33, IBM Deutschland, 1988.

[28] W. Stein. Indexing principles for relational languages applied to PROLOG code generation. DFKI D-92-22, 1992.

[29] P. van Hentenryck. Constraint logic programming. *The Knowledge Engineering Review*, 6(3), 1991.

[30] M. Vilain. The restricted language architecture of a hybrid representation system. In *IJCAI-85*, 1985.

[31] H. Wache. Knowledge representation and processing in TOOCON. In K. Hinkelmann and A. Laux, editors, *DFKI-Workshop on Knowledge Representation Techniques*. DFKI D-93-11, 1993.

Eine Spezifikationssprache für Transformationen auf getypten Merkmalsstrukturen

Jochen Bedersdorfer
Karsten Konrad
Ingo Neis
Oliver Scherf
Jörg Steffen
Michael Wein*

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany
email: {beders|konrad|neis|scherf|steffen|wein}@dfki.uni-sb.de

Zusammenfassung

Termersetzungssysteme spielen heute eine wichtige Rolle bei der Beschreibung und Implementierung von Softwareschnittstellen und Compilern. Die Produktionsregeln in den hierbei verwendeten Spezifikationssprachen operieren meist auf der syntaktischen Struktur der zu manipulierenden Terme. Im Gegensatz dazu wurde unsere Spezifikationssprache speziell für die *abstrakte* Beschreibung von Transformationen entwickelt, um weitgehend unabhängig von den syntaktischen Details der zu transformierenden Terme zu sein. Transformationen werden in unserer Sprache als deklarative Regeln der Form $l \rightarrow r$ geschrieben, wobei l und r getypte Merkmalsstrukturen sind. Diese aus der Computerlinguistik stammende Datenstruktur eignet sich gleichermaßen gut zur Beschreibung von *Patterns* auf der linken Regelseite als auch zur Darstellung der Ersetzungsknoten auf der rechten. Die Mächtigkeit der aus den gängigen Merkmalsbeschreibungssprachen bekannten syntaktischen Konstrukte kann hierbei für unsere Anwendung durch merkmalslogikfremde Ansätze wie zum Beispiel reguläre Audrücke für Listen erweitert werden.

Schlüsselwörter: Transformation, abstrakte Syntax, Merkmalsstrukturen.

---

*Großer Dank gilt unseren Betreuern John Nerbonne und Abdel Kader Diagne für hilfreiche Diskussionen und konstruktive Kritik. Diese Arbeit wurde durch Mittel des Bundesministeriums für Forschung und Technologie gefördert (01 IV 101 K/1 für das Projekt ASL).

1 Einleitung und Motivation

KI–Systeme sind, bedingt durch ihre speziellen Aufgabenstellungen, im allgemeinen große Programmkomplexe aus mehreren, oft weitgehend autonomen Modulen. Unabhängig von der gewählten Architektur des Gesamtsystems stellt sich fast immer das Problem, bestehende Module mit festliegender Ein– und Ausgabestruktur einzubinden. In der Praxis kann diese Anbindung am leichtesten durch spezielle Schnittstellenmodule realisiert werden, die jeweils zwei Module dadurch verbinden, daß sie die Ausgabesprache eines Moduls in die Eingabesprache eines Zielmoduls übersetzen. Es handelt sich also um sehr spezialisierte Übersetzer, die neben der syntaktischen Transformation oft Aufgaben wie die Simplifikation von Eingabetermen oder das Aussortieren nicht relevanter Informationen übernehmen. An die Schnittstellenmodule und ihre Entwicklungswerkzeuge werden dabei insbesondere folgende Anforderungen gestellt:

Flexibilität: Während der Entwicklung eines KI–Systems kann sich das Ein– und Ausgabeverhalten einzelner Module stark verändern. Die Schnittstellenmodule müssen dem dadurch Rechnung tragen, daß sie mit möglichst wenig Aufwand an diese Veränderungen angepaßt werden können.

Performance: Die Laufzeit der Übersetzungen muß im Interesse einer guten Gesamtleistung möglichst klein bleiben. Im Gegensatz zu herkömmlichen Compilern, wo die Laufzeit einer Compilation weniger kritisch ist, geht die Übersetzungzeit voll in das Zeitverhalten des Gesamtsystems ein und muß daher genau beachtet werden.

Mächtigkeit: In einem KI–System müssen je nach Komplexität eine ganze Reihe von Schnittstellenmodulen spezifiziert werden. Da man möglichst früh die Zusammenarbeit der einzelnen Komponenten sicherstellen und testen will, ist die Fähigkeit des *rapid prototyping* eine wichtige Forderung an die Werkzeuge zur Entwicklung der Schnittstellen.

Die Idee der *Programmtransformation*, d.h. der Übersetzung einer Quell– in eine Zielsprache wird bereits im Compilerbau seit längerem verfolgt. Dabei beschreibt man mithilfe einer (meist) funktionalen Programmiersprache eine Transformation auf der Oberflächensyntax; der Compiler–Compiler erzeugt aus dieser Beschreibung ein Programm, das diese Transformationen durchführt. Beispiele für derartige Entwicklungswerkzeuge sind das Lisp–orientierte Refine und die funktionale Programmiersprache Trafola ([Refine 90] und [Trafola 88]).

Da bei unserer speziellen Aufgabe die Syntax sowohl von Quell– als auch von Zielsprache nicht immer endgültig festgelegt ist, ist es sinnvoll, für die Beschreibung der Transformationen einen möglichst abstrakten, syntaxfernen Ansatz zu wählen. Hier bietet es sich an, deklarative Transformationsregeln auf dem abstrakten Syntaxbaum als Basis der Beschreibung von Schnittstellenmodulen zu wählen. Abstrakte Syntax bietet gegenüber konkreter den Vorteil, weitgehend unabhängig von willkürlichen und nicht relevanten Details der Oberflächensyntax zu sein. So ist es zum Beispiel semantisch unerheblich, ob mathematische Ausdrücke in Post– oder Präfixnotation geschrieben werden.

Es stellt sich letztlich noch die Frage, wie eine Spezifikation für Transformationen auf abstrakten Syntaxbäumen aussehen soll. Wir haben uns für eine Sprache auf der Basis *getypter Merkmalsstrukturen* entschieden. Diese aus der Computer–Linguistik stammende Datenstruktur eignet sich zur Darstellung beliebiger dekorierter Bäume und Graphen und wird im Rahmen von *Merkmalsbeschreibungssprachen* (wie z.B. im Fall der Typdefinitionssprache $\mathcal{TDL}$, siehe [Krieger 93]) zur linguistischen Wissensrepräsentation verwendet. Da abstrakte Syntaxterme als dekorierte Bäume angesehen werden können, kann man sie leicht als Merkmalsstrukturen interpretieren. Die in den getypten Merkmalsstrukturen zusätzlich vorhandene Typinformation entspricht dabei der Zuordnung eines Knotens des abstrakten Syntaxbaums zu einer semantischen Kategorie. Die Idee der Abstraktion kann hier sogar noch weiter getrieben werden, da man mit Merkmalsstrukturen auch die *internen* Datenstrukturen eines Moduls abbilden kann; die Transformation durch ein Schnittstellenmodul kann dadurch direkt auf den internen Daten eines Moduls arbeiten, ohne daß ein zeitaufwendiges Generieren und Parsen stattfinden muß.

2 Getypte Merkmalsstrukturen

[Carpenter 92] definiert eine Getypte Merkmalsstruktur über einer endlichen Menge von Merkmalen oder Attributen **Feat** und einer endlichen Menge von Typen **Type** mit definierter Vererbungshierarchie $\sqsubseteq$ als ein Tupel $F = \langle Q, q, \theta, \delta \rangle$ mit

- Q : einer endlichen Menge von Knoten
- $q \in Q$: dem Wurzelknoten
- $\theta : Q \to \textbf{Type}$ einer totalen Typfunktion der Knoten
- $\delta : \textbf{Feat} \times Q \to Q$ einer partiellen Feature–Wert–Funktion

Üblicherweise konzeptualisiert man Merkmalsstrukturen als gerichtete Graphen mit einer Menge von Knoten Q und einer Wurzel q, wobei θ die Attributierung der Knoten und δ die an den Knoten hängenden Subgraphen bestimmt. Oft findet man auch Definitionen, bei denen als Wert eines Attributs auch atomare Werte, wie Symbole oder Zahlen, erlaubt sind. Die hier vorgestellte Definition erlaubt prinzipiell alle Arten von Graphen, insbesondere auch zyklische. Für bestimmte Anwendungen kann es sinnvoll sein, diese Definition auf azyklische Merkmalsstrukturen einzuschränken.

Wir wollen nun als einfaches Beispiel den Term $((3 + x) * 4)$ betrachten. Für seine abstrakte Darstellung wählen wir die Typen *Plus* mit den Attributen **SUMMAND1** und **SUMMAND2** und *Mult* mit den Attributen **FAKTOR1** und **FAKTOR2**. Der Term enthält die atomaren Werte 3, 4 und x. Die Standardnotation für Merkmalsstrukturen sind die mit dem *Framekonzept* ([Minsky 75]) verwandten Attribut–Wertmatrizen (AWM). In dieser Darstellung sieht der Term folgendermaßen aus:

$$ {}_{Mult}\left[\begin{array}{ll} \texttt{FAKTOR1} & {}_{Plus}\left[\begin{array}{ll}\texttt{SUMMAND1} & 3\\ \texttt{SUMMAND2} & x\end{array}\right] \\ \texttt{FAKTOR2} & 4\end{array}\right] $$

Diese abstrakte Darstellung geht von Addition und Multiplikation als binären Operatoren aus. Bei Prä– oder Postfixdarstellung können diese Operatoren auch n–stellig benutzt werden. Wir können nun den abstrakten Syntaxbaum von Termen mit binären Operatoren in n–stellige transformieren. Dazu verwenden wir Ersetzungsregeln, die durch Paare von getypten Merkmalsstrukturen kodiert werden.

3 Ersetzungsregeln

Termersetzungssysteme bestehen in aller Regel aus einer Menge von *gerichteten Gleichungen*, die dazu verwendet werden, Unterterme u in einem gegebenen Term t durch korrespondierende Terme zu ersetzen. In unserem Fall sind die Terme auf jeder Seite der Gleichung getypte Merkmalsstrukturen, wobei die getypten Merkmalsstrukturen der linken Seite als *Pattern* auf den Untertermen des zu bearbeitenden Syntaxbaums und die getypten Merkmalsstrukturen auf der rechten Seite als Ersetzungsknoten im Fall eines erfolgreichen *Matchings* anzusehen sind. Eine Regel der Form $l \rightarrow r$ wird also so interpretiert, daß im Fall einer Übereinstimmung des Musters l mit einem Unterbaum im abstrakten Syntaxbaum dieser Unterbaum durch die instanziierte Form von r ersetzt wird. Der Antezedenz ist hierbei ein partielles, dekompositionelles Muster, die Konsequenz dagegen eine kompositionelle, totale Beschreibung des Ersetzunsknotens.

Eine Regel, die den Typ *Mult* in den Typ *Mult–n* transformiert, könnte etwa so aussehen:

$$ {}_{Mult}\left[\begin{array}{ll} \texttt{FAKTOR1} & \boxed{1} \\ \texttt{FAKTOR2} & \boxed{2}\end{array}\right] \Longrightarrow {}_{Mult\text{-}n}\left[\texttt{FAKTOREN} \; < \boxed{1}, \boxed{2} > \right] $$

Die durch diese Regel nicht transformierten Unterterme von t werden auf der linken Seite an logische Variablen, die Koreferenzen[1] gebunden. Die Koreferenz $\boxed{1}$ erhält also den Unterterm am Attribut **FAKTOR1**, die Koreferenz $\boxed{2}$ den an **FAKTOR2** als Wert. Die gebundenen Unterterme werden auf der rechten Seite mit dem Ausdruck $<\boxed{1},\boxed{2}>$ zu einer Liste zusammengebaut und in einer zu erzeugenden Instanz des Typs *Mult–n* am Attribut **FAKTOREN** eingesetzt. *Mult–n* stellt dabei unsere abstrakte Form für n–stellige Multiplikation dar. Entsprechend sieht die Transformationsregel für die Addition aus:

$$ {}_{Plus}\left[\begin{array}{ll} \texttt{SUMMAND1} & \boxed{1} \\ \texttt{SUMMAND2} & \boxed{2}\end{array}\right] \Longrightarrow {}_{Plus\text{-}n}\left[\texttt{SUMMANDEN} \; < \boxed{1}, \boxed{2} > \right] $$

[1] Obwohl Koreferenzen hier die Aufgabe von logischen oder Metavariablen übernehmen, werden sie üblicherweise nicht als solche gesehen, sondern beschreiben in Merkmalsstrukturen das sogenannte *Structure Sharing*, also die Identität von Werten an verschiedenen Attributen.

Die Anwendung solcher Regeln erfolgt analog zu Termersetzungssystemen; jede Regel des Systems wird solange auf jeden Knoten des zu transformierenden Baums angewendet, bis keine weitere Regel mehr einen Ersetzungsknoten erzeugt, d.h. eine Normalform erreicht wurde. Im Fall eines terminierenden und konfluenten Ersetzungssystems ist die Reihenfolge der Regelanwendung beliebig. Da wir nur an eindeutigen Normalformen interessiert sind, gehen wir von Ersetzungssystemen mit den entsprechenden Eigenschaften aus.

4 Syntaktische Konstrukte

Da als Wert eines Attributs in einer getypten Merkmalsstruktur wiederum eine getypte Merkmalsstruktur zugelassen ist, können wir beliebig tief geschachtelte Bäume matchen und generieren. Damit ist es bereits möglich, einen großen Bereich einfacher Typtransformationen durch Regeln abzudecken. Wir erlauben aber weiterhin, mithilfe erweiternder Syntaxkonstrukte komplexere Transformationen durchzuführen. Die wichtigsten dieser Sprachelemente sind:

- Die üblichen logischen Verknüpfungen ‚∨', ‚∧' und ‚¬'
- Funktionen
- Verteilte bzw. indizierte Disjunktionen
- Negierte Koreferenzen
- Mengen- und Sequenzterme

Diese, im folgenden näher beschriebenen Konstrukte bilden den Kern aller Transformationsbeschreibungen mit getypten Merkmalsstrukturen.

4.1 Disjunktion

Während die Terme der Objektseite einfache konjunktive Merkmalsstrukturen ohne disjunktive Kanten sind, finden wir auf der Beschreibungsebene der Sprache Strukturen von höherer Komplexität. Im folgenden Beispiel steht an dem Attribut **FAKTOR1** eine Disjunktion, die entweder mit dem Integer *1* oder der reellen Zahl *1.0* matcht. In Anlehnung an die übliche Schreibweise für logische Ausdrücke verwenden wir ‚∨' als Symbol für die Disjunktion.

$$\begin{bmatrix} \texttt{FAKTOR1} & {}_{Int}\begin{bmatrix}\texttt{VAL} & \mathit{1}\end{bmatrix} \vee {}_{Real}\begin{bmatrix}\texttt{VAL} & \mathit{1.0}\end{bmatrix} \\ \texttt{FAKTOR2} & \boxed{\text{val}} \end{bmatrix}_{Mult} \Longrightarrow \boxed{\text{val}}$$

4.2 Verteilte Disjunktion

Verteilte Disjunktionen drücken Abhängigkeiten von Werten innerhalb einer Merkmalsstruktur und zwischen Regelseiten aus. Im folgenden Beispiel[2] wird mithilfe der Information an den Attributen **GENDER** und **NUM** für Genus und Numerus eines Nomens der dazugehörige bestimmte Artikel zurückgegeben.

$$Noun\begin{bmatrix}\texttt{GENDER} & \%gen\ ('mas \vee 'fem \vee 'neu) \\ \texttt{NUM} & \%num\ ('sing \vee 'plur)\end{bmatrix} \Longrightarrow$$

$$Article\begin{bmatrix}\texttt{PHON}\ \%num\ (\%gen\ (\text{„der“} \vee \text{„die“} \vee \text{„das“}) \vee \text{„die“})\end{bmatrix}$$

Die verteilte Disjunktion *%gen* bekommt hierbei durch das Matching die Genusinformation an **GENDER** und *%num* die Information über den Numerus an **NUM** zugewiesen. Auf der rechten Seite entscheidet der Zustand der beiden Disjunktionsvariablen, die durch das Matching einen eindeutigen Index[3] erhalten haben, ob auf der rechten Seite der an **PHON** zu bindende Wert „der“, „die“ oder „das“ sein soll.

4.3 Funktionen und Prädikate

Als eine Möglichkeit, besonders schwierig zu formulierende oder zeitkritische Konstrukte zu spezifizieren, erlauben wir die Verwendung von *foreign functions* aus der Lisp–Umgebung auf beiden Seiten der Spezifikationsregeln. Auch Transformationsregeln selbst können wieder als lokal einzusetzende Funktionen verwendet werden, um so rekursive oder kontextabhängige Transformationen auszudrücken.

Auf der linken Seite wird der Rückgabewert einer Funktion als boolescher Wert eines Matchings aufgefasst, und innerhalb der implizit konjunktiven Form der Merkmalsstruktur ausgewertet. Auf der rechten Seite liefern Funktionsaufrufe Werte zurück, die innerhalb der aufzubauenden Struktur oder auf dem Top–Level beliebig verwendet werden können. Die folgende Transformationsregel konkateniert die Werte an den Attributen **LIST1** und **LIST2** des Typs *Append* nur dann, wenn es sich dabei tatsächlich um (LISP–) Listen handelt, was mittels des Prädikats „listp“ festgestellt wird. Das Prädikat bekommt hierbei den an die Koreferenz gebundenen Wert als Parameter übergeben.

$$Append\begin{bmatrix}\texttt{LIST1} & \boxed{1}\ \texttt{listp}(\boxed{1}) \\ \texttt{LIST2} & \boxed{2}\ \texttt{listp}(\boxed{2})\end{bmatrix} \Longrightarrow \texttt{append}(\boxed{1},\boxed{2})$$

---

[2] Die folgenden Beispiele erheben keinerlei Anspruch auf linguistische Korrektheit, sondern dienen lediglich der Illustration.

[3] In der Merkmalslogik werden logische Formeln ohne Beachtung ihrer Reihenfolge behandelt. Dies gilt insbesondere auch für verteilte Disjunktionen. Für unsere Anwendung ist es allerdings notwendig, durch die Reihenfolge der Subterme einer Disjunktion eine Priorität des Matchings auszudrücken. Es ist in diesem Zusammenhang sinnvoll, für unsere Version der verteilten Disjunktionen den Begriff „indizierte Disjunktion“ zu verwenden.

4.4 Negation und negierte Koreferenzen

Die übliche Termnegation, bei der das boolesche Ergebnis eines Matchings umgekehrt wird, kann auf alle Arten von Ausdrücken angewendet werden. Das erste Beispiel zeigt die Verwendung der Negation, um eine Division durch 0 explizit auszuschließen. Der Negationsoperator ‚¬' kehrt hierbei den booleschen Wert des Matchings am Attribut `DIVISOR` mit dem Atom 0 um.

$$\mathit{Division}\begin{bmatrix}\texttt{DIVIDENT} & \boxed{1} \\ \texttt{DIVISOR} & \boxed{2}\neg 0\end{bmatrix} \Longrightarrow \texttt{divide}(\boxed{1},\boxed{2})$$

Eine der Stärken der Merkmalslogik ist die Möglichkeit, die Gleichheit von Werten durch Koreferenzen auszudrücken, ohne daß eine genaue Spezifikation dieser Werte erfolgen muß. Mithilfe negierter Koreferenzen kann dazu analog Ungleichheit ausgedrückt werden.

Beispielsweise ist eine Person üblicherweise weder ihr eigener Vater noch ihre eigene Mutter. Das folgende Beispiel überprüft diesen Sachverhalt und testet außerdem, ob Vater und Mutter verschiedene[4] Personen sind.

$$\boxed{1}\ \mathit{Entry}\begin{bmatrix}\texttt{FATHER} & \boxed{2}\ \neg\boxed{1} \\ \texttt{MOTHER} & \boxed{3}\ \neg\boxed{1} \wedge \neg\boxed{2}\end{bmatrix} \Longrightarrow \mathit{Person}\begin{bmatrix}\texttt{FATHER} & \boxed{2} \\ \texttt{MOTHER} & \boxed{3}\end{bmatrix}$$

4.5 Konjunktion

Der Typ einer Merkmalsstruktur und die Attribut-Wert-Paare drücken bereits implizit eine Konjunktion aus. Will man mehrere Bedingungen an den Wert eines Attributs stellen, kann man dies analog zur Disjunktion durch die Termkonjunktion ‚∧' ausdrücken.

Da man die logischen Operatoren sowohl auf dem Top-Level als auch in den Strukturen des Antezedenz frei miteinander kombinieren kann, lassen sich sehr mächtige und komplexe Matchings auf den Ausgangsstrukturen durchführen. Insbesondere die Kombination aus Merkmalslogik, funktionalen Ausdrücken und Aussagenlogik ergibt vielfältige Möglichkeiten, die beabsichtigten Aussagen adäquat zu formulieren.

4.6 Sequenzen

Unsere Sprache erlaubt sowohl *Matching* als auch Aufbau von Listen durch spezielle Syntaxkonstrukte, die sich an bekannten *Pattern-Matching*-Verfahren mit regulären

[4] Gleiche Koreferenzen auf der linken Seite prüfen die sogenannte „Tokenidentität" ab. Gleichheit bedeutet hierbei, daß komplexe Strukturen tatsächlich nicht nur dieselbe Form, sondern tatsächlich identisch sein müssen.

Ausdrücken orientiert. Wir verwenden mit Kleene–Operatoren ‚*' und ‚+' kombinierte Koreferenzen als *Sequenzvariablen*, um Teillisten mit bestimmten Eigenschaften zu extrahieren.

Die folgende Regel bildet die Lisp–Funktion **rest** nach:

$$< \boxed{1}, \boxed{2}* > \Longrightarrow \boxed{2}$$

Die Koreferenz $\boxed{1}$ bindet hierbei das erste Element der Liste, der Ausdruck $\boxed{2}*$ beliebig viele nachfolgende Elemente der Ausgangssequenz. Die Koreferenz $\boxed{2}$ enthält nach erfolgreichem Matching alle Elemente der Liste außer dem ersten, also den Rest der Liste.

Diese Regel kann auch etwas effizienter mithilfe des Rest–Operators ‚.' formuliert werden. Hierbei wird der Rest der Liste direkt an die Koreferenz $\boxed{2}$ gebunden. Da wir den Wert der Koreferenz $\boxed{1}$ nicht weiter benötigen, binden wir ihn an eine anonyme Koreferenz $\boxed{}$.

$$< \boxed{} . \boxed{2} > \Longrightarrow \boxed{2}$$

Analog zu normalen Koreferenzen können auch Kleene–Koreferenzen mit weiteren Konstrukten kombiniert werden. Sequenzmatcher und -konstruktoren dürfen natürlich für ihre Elemente alle geeigneten Sprachkonstrukte verwenden.

4.7 Mengen

Unsere Sprache unterstützt *Matching*–Verfahren, die auch mit ungeordneten Listen arbeiten können. Auch hier können, wie bei den Sequenzen, Koreferenzen mit Kleene–Operatoren kombiniert werden. Diese „Sequenzvariablen" von Mengen binden Teilmengen der Ausgangsmenge. Intern werden sowohl Sequenzen als auch Mengen als Listen repräsentiert. Es steht dem Benutzer also frei, je nach gewünschter Anwendung Werte als Mengen oder Sequenzen zu behandeln.

Die folgende Regel liefert alle geraden Zahlen aus einer Liste:

$$\{\boxed{1}* \; \boxed{2}\,\mathbf{evenp}(\boxed{2}), ...\} \Longrightarrow \boxed{1}$$

Im Gegensatz zu Kleene–Operatoren bei Sequenzen binden die Kleene–Operatoren der Mengen immer die größtmögliche Teilmenge der vorhandenen Werte. Der Unterschied zwischen ‚+' und ‚*' besteht darin, daß im Fall von ‚*' die gebundene Teilmenge auch leer sein darf.

4.8 Erweiterungen

Zusätzlich zu den oben beschriebenen Syntaxkonstrukten erproben wir zur Zeit weitere Ausdrucksmöglichkeiten, um die Formulierungsstärke der Sprache zu erhöhen, z.B. die nichtmonotone Überschreibung von Werten in den Objektstrukturen oder das Matchen mit Subtermen ohne exakte Angabe von Attribut-Pfaden (*functional uncertainty*). Während einige dieser Konstrukte lediglich syntaktischer Zucker sind, kann der Benutzer mit anderen sehr viel mächtigere, oft aber auch laufzeitintensivere Transformationen beschreiben.

5 Implementierungen

Zur Zeit existieren zwei Compiler der vorgestellten Spezifikationssprache, die beide in den natürlichsprachlichen Systemen der DISCO und ASL-Projekte des DFKI verwendet werden.

5.1 SL∀NT

SL∀NT steht für „Specification Language for all $\mathcal{NLL}$ Transformations" und ist gedacht als ein Transformationssystem für die abstrakte Syntax der linguistischen Semantikrepräsentationssprache $\mathcal{NLL}$ ([Laubsch 91] und [Nerbonne 92]). SL∀NT wurde in Common Lisp implementiert und erzeugt aus Transfomationsregeln ausführbare Lisp-Funktionen, die mithilfe einer Treiberfunktion auf den Syntaxbäumen der $\mathcal{NLL}$-Zielstrukturen angewendet werden. Insbesondere ist **SL∀NT** in der Lage, die internen Strukturen von $\mathcal{NLL}$ zu bearbeiten und Regeln auf die Konformität mit dem Typsystem von $\mathcal{NLL}$ zu überprüfen.

5.2 Zebra

Ein weiterer Compiler der vorgestellten Spezifikationssprache ist **Zebra**, der als Erweiterung des **Common Lisp**-Parsergenerators **Zebu** ([Laubsch 93]) konzipiert wurde. **Zebu** erzeugt aus einer formalen Beschreibung LALR(1)-Parser, die als Ergebnis des Parsens getypte Merkmalsstrukturen in Lisp erzeugen. Diese Merkmalsstrukturen werden durch von **Zebra** definierte Transformationsfunktionen in Zielstrukturen umgewandelt, die beim Ausdruck mit Hilfe von *pretty-printern* die gewünschte konkrete Syntax der Zielsprache zeigen. Besonderheiten von **Zebra** sind die inkrementelle Kompilation von Regeln und die typorientierte Optimierung von Regelgruppen.

5.3 Systemarchitektur

Abbildung 1 veranschaulicht die Architektur unserer Compiler.

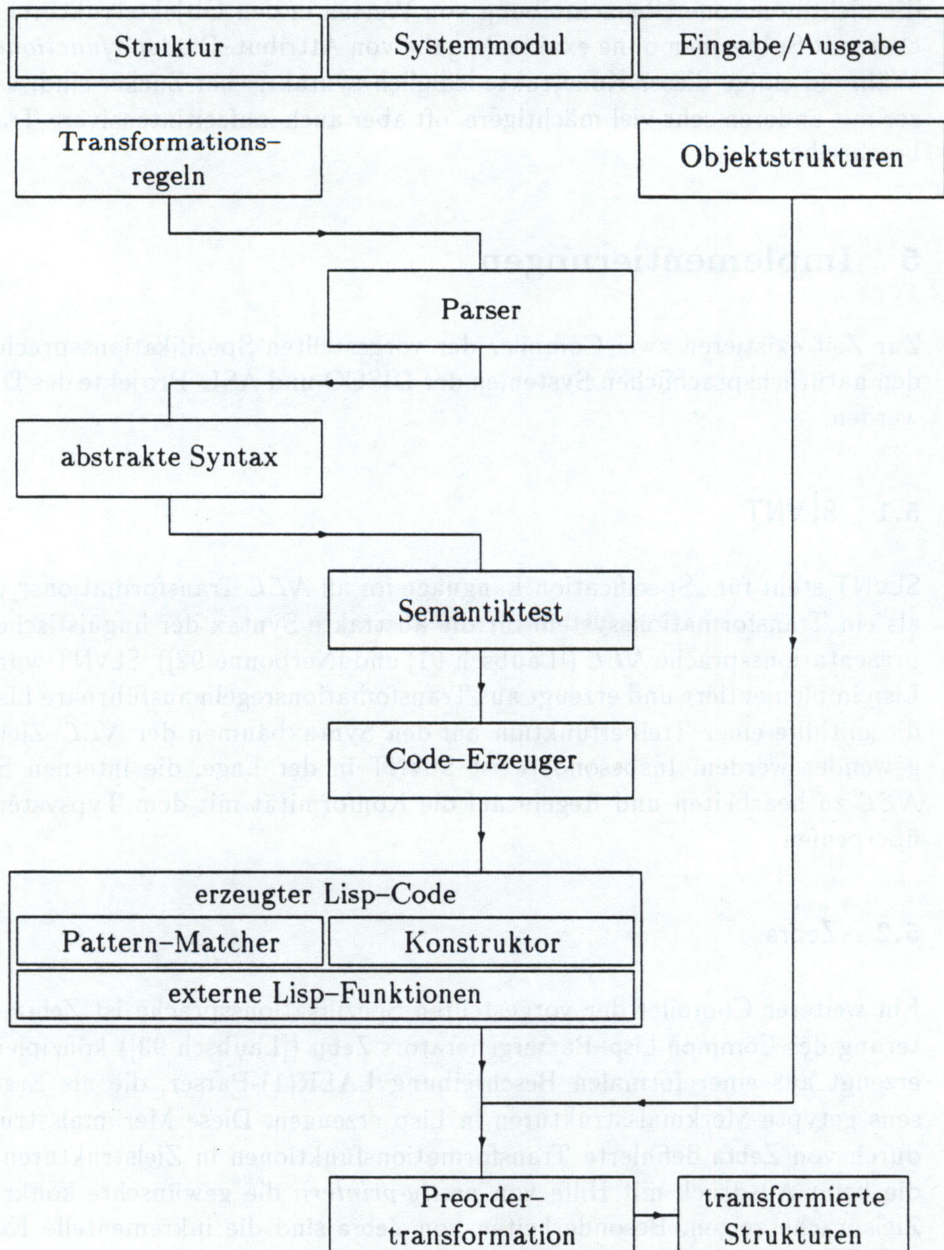


Abbildung 1: Systemarchitektur

6 Ausblick

Da die Integration einer zunehmend größeren Anzahl von Modulen eine der Hauptaufgaben des Designs von komplexen KI-Systemen ist, besteht ein erhöhter Bedarf an leicht wartbaren Softwareschnittstellen. Wir glauben, daß unser Ansatz der deklarativen und abstrakten Spezifikation dieser Schnittstellen eine Möglichkeit ist, dieses Ziel zu erreichen. Bestehende Testmodule zeigen, daß sich mit unseren Systemen bei relativ geringem Aufwand ausreichend leistungsfähige Übersetzer erstellen lassen. Neben dieser wichtigen Aufgabe können unsere Systeme aber auch die Standardaufgaben herkömmlicher Transformationssysteme übernehmen. So ist geplant, vorhandene linguistische Lexika, die für einen bestimmten Grammatikformalismus entwickelt wurden, mittels SL∀NT bzw. **Zebra** für neuere Formalismen verwendbar zu machen. Auch hier kann wieder das eigentliche Ziel der syntaktischen Transformation sehr elegant über abstrakte Transformationsregeln erreicht werden.

Anhang

Codeerzeugung — Ein Beispiel

`Nll2fs` ist ein vergleichsweise einfaches Transformationsmodul von $\mathcal{NLL}$ in die Eingabesprache des DISCO-Unifikators $\mathcal{UD}i\mathcal{N}e$. Wir zeigen nun anhand einer Regel aus diesem Modul die Wirkungsweise unserer Compiler.

Bei der Eingabe von Regeln verwenden wir eine ASCII-Notation, die sich an die gleichartige Syntax von $\mathcal{TDL}$ anlehnt. Obwohl auch geeignete graphische Editoren wie z.B. der Merkmalseditor Fegramed ([Kiefer 93]) zur Verfügung stehen, ist diese Form der Darstellung für die Eingabe von Programmen die am besten geeignete.

Die folgende Regel

$$\mathit{role\text{-}argument\text{-}pair}\left[\begin{array}{ll} \texttt{-role} & \boxed{1}\,\%\mathit{role} \left(\begin{array}{l} \mathit{role\text{-}complex}[\;] \vee \\ \mathit{individual\text{-}role\text{-}complex}[\;] \vee \\ \mathit{group\text{-}role\text{-}complex}[\;] \end{array}\right) \\ \texttt{-argument} & \boxed{2} \end{array}\right] \Longrightarrow$$

$$\mathit{label\text{-}value\text{-}pair}\left[\begin{array}{ll} \texttt{-label} & \%\mathit{role} \left(\begin{array}{l} \mathit{fs\text{-}role}[\texttt{-role make-label-from-role}(\boxed{1})] \vee \\ \mathit{fs\text{-}i\text{-}role}[\texttt{-role make-label-from-role}(\boxed{1})] \vee \\ \mathit{fs\text{-}g\text{-}role}[\texttt{-role make-label-from-role}(\boxed{1})] \end{array}\right) \\ \texttt{-value} & \boxed{2} \end{array}\right]$$

beschreibt die Übersetzung von Rollen-Argument-Paaren in atomaren $\mathcal{NLL}$-Formeln in die korrespondierenden Attribut-Wert-Paare der Merkmalsstrukturen. Die verteilte

Disjunktion entscheidet dabei die genaue Übersetzung der Rolle, während das Argument am Attribut -argument einfach von links nach rechts weitergereicht wird. Die ASCII-Schreibweise dieser Regel sieht folgendermaßen aus:

```
Rule nll-rap2fs-lvp :=
  role-argument-pair:[(-role #1 %role(role-complex:[] |
                                      individual-role-complex:[] |
                                      group-role-complex:[]))
                      (-argument #2)]
-->
  label-value-pair:[(-label
                     %role(fs-role:[(-role
                                     make-label-from-role(#1))] |
                           fs-i-role:[(-role
                                       make-label-from-role(#1))] |
                           fs-g-role:[(-role
                                       make-label-from-role(#1))]))
                    (-value #2)];
```

Die Aufgabe der Code-Erzeugung besteht darin, aus dieser Regel eine (totale) Funktion zu erzeugen, die im Fall einer gegebenen Subsumptionsbeziehung zwischen der in der linken Regelseite beschriebenen Struktur und der Objektstruktur die gewünschte Zielstruktur aufbaut und als Funktionsergebnis zurückliefert. Falls dagegen das Matching mißlingt, also keine Subsumption vorliegt, muß die Funktion den jeweils aktuellen Knoten unverändert zurückgeben.

Beide Compiler sind Ein-Pass-Compiler, die Code, d.h. S-Expressions in Lisp, durch rekursives Absteigen in die abstrakten Syntaxbäume der Regeln erzeugen. Das Ergebnis der Übersetzung der oben gezeigten Regel ist folgende Funktionsdefinition:

```
(defun nll-rap2fs-lvp (cnode)
  (let (crf-2 crf-1)
    (case (block pattern
            (and (role-argument-pair-p cnode)
                 (let ((cn-role-argument-pair cnode))
                   (setq crf-2
                         (role-argument-pair--argument
                          cn-role-argument-pair))
                   (let ((cnode
                          (role-argument-pair--role
                           cn-role-argument-pair)))
                     (progn (setq crf-1 cnode)
                            (or (and (role-complex-p cnode) 0)
                                (and (individual-role-complex-p
                                      cnode)
```

```
                                      1)
                                  (and (group-role-complex-p cnode)
                                       2)))))))
                (0
                 (make-label-value-pair
                     :-label (make-fs-role
                              :-role (make-label-from-role crf-1))
                     :-value crf-2))
                (1
                 (make-label-value-pair
                     :-label (make-fs-i-role
                              :-role (make-label-from-role crf-1))
                     :-value crf-2))
                (2
                 (make-label-value-pair
                     :-label (make-fs-g-role
                              :-role (make-label-from-role crf-1))
                     :-value crf-2))
                (otherwise cnode))))
```

Literatur

[Aho 86] Alfred Aho und Ravi Sethi und Jeffrey Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[Backofen 90] R. Backofen und L. Euler und G. Goerz. *Towards the Integration of Functions, Relations and Types in an AI Programming Language.* In: Proceedings of GWAI-90, Springer-Verlag, Berlin, 1990.

[Carpenter 92] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, Cambridge, 1993.

[Eisinger 92] Norbert Eisinger und Andreas Nonnengart und Axel Präcklein. *Termersetzungssysteme.* In: K. H. Bläsius and H. J. Bürckert (Hrsg.), Deduktionssysteme, Oldenbourg, München, 1992.

[Hoffmann 82] O'Donnell Hoffmann. *Pattern Matching in Trees.* In: JACM 29,1, 1992.

[Kiefer 93] Bernd Kiefer. *Fegramed — An Interactive Graphics Editor for Feature Structures.* To appear.

[Konrad 93] Karsten Konrad. **Zebra**: *Rewriting and Rule Compilation for Typed Feature Structures.* Unpublished.

[Krieger 93] Hans-Ulrich Krieger und Ulrich Schäfer. *$\mathcal{TDL}$ — A Type Definition Language for Unification-Based Grammars.* To appear.

[Laubsch 91] Joachim Laubsch und John Nerbonne. *An overview of $\mathcal{NLL}$.* Technical report, Hewlett-Packard Laboratories, Palo Alto, July 1991.

[Laubsch 93] Joachim Laubsch. **Zebu**: *A Tool for Specifying Reversible LALR(1) Parsers.* Technical report, Hewlett-Packard Laboratories, Palo Alto, Mai 1993.

[Minsky 75] Marvin Minsky. *A framework for representing knowledge.* In: P. H. Winston (Hrsg.), The Psychology of Computer Vision, McGraw-Hil, New York, 1975.

[Nerbonne 92] John Nerbonne und Stephan Oepen und Abdel Kader Diagne und Karsten Konrad und Ingo Neis. *$\mathcal{NLL}$—Tools for Meaning Representation.* In: Stephan Busemann and Karin Harbusch (Hrsg.), DFKI Workshop on Natural Language Systems, Saarbrücken, Oktober 1992.

[Refine 90] REFINE *User's Guide.* Reasoning Systems Inc., Palo Alto, 1990.

[Sperschneider 91] V. Sperschneider and G. Antoniou. *Logic, a Foundation for Computer Science.* Addison-Wesley, 1992.

[Slant 93] SL∀NT: *Specification Language for all $\mathcal{NLL}$ Transformations. User's guide.* To appear.

[Trafola 88] M. Alt, C. Fecht, C. Ferdinand und G. Sander. *A Prototype Of A Transformation Language.* In: PROSPECTRA S.1.6-R-16.0, 1988.

Migration und Kompilation in Lisp: Ein Weg von Prototypen zu Anwendungen*

Wolfgang Goerigk Friedemann Simon

Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel
Preußerstraße 1-9, D-24105 Kiel
wg@informatik.uni-kiel.de
fs@informatik.uni-kiel.de

Zusammenfassung

Lisp-basierte Anwendungen behalten oft den Charakter von Prototypen und weisen deshalb Defizite auf in den Bereichen: Wartbarkeit, klares Design und sparsamer Umgang mit Ressourcen. Der auf die Kompilation von Anwendungsprogrammen ausgerichtete Ansatz von APPLY bietet einen Weg zur Qualitätsverbesserung. Es entstehen von einem Lisp-System unabhängige Applikationen, d.h. übliche Objektfiles oder C-Programme, die mit Betriebssystemmitteln zu fertigen Programmen verbunden werden können. Dies ist eine wesentliche Voraussetzung für eine wirkliche Integration in ein DV-Umfeld. Zur Erzeugung von kompakten Anwendungen ist zudem ein Migrationsschritt erforderlich, der die dynamischen und hochflexiblen Sprachkonstrukte eliminiert. Praktisch bedeutet das eine gründliche manuelle Überarbeitung des Programms entsprechend einem hier ausgearbeiteten methodischen Vorgehen und allgemeiner Prinzipien des Software-Engineering. Nach vorliegenden Erfahrungen sind Migrationsschritte nur an wenigen Stellen erforderlich, wo man sich beim Prototypen mit einer ad-hoc-Lösung durch ein dynamisches, universell einsetzbares Sprachmittel beholfen hat. Durch statische Programmanalyse werden zudem Programmierfehler aufgedeckt.

1 Einleitung

Der breiten industriellen Anwendung Lisp-basierter Expertensystemtechnik stehen verschiedene Hindernisse im Wege [KG93] [Weu93] [MBG90]. Ein Problemkreis ist die mangelnde technische und organisatorische Integrationsfähigkeit in das industrielle DV-Umfeld. Ein weiteres Problem entsteht, weil die klassischen Vorgehensweisen bei einer Systementwicklung häufig unberücksichtigt bleiben. Die Anwendungen behalten den Charakter von Prototypen und weisen deshalb Defizite auf in den Bereichen: Wartbarkeit, klares Design und sparsamer Umgang mit Ressourcen.

Der auf die Kompilation von Anwendungsprogrammen ausgerichtete Ansatz von APPLY [BCF+91] [GS91] bietet einen Weg zur Qualitätsverbesserung unter besonderer Berücksichtigung der genannten Kriterien.

Grundannahme ist eine klare Trennungsmöglichkeit von Entwicklung und Anwendung eines Programms. Fertige Programme benötigen die Flexibilität und den Leistungsumfang von Lisp nicht

*Diese Arbeit ist im Rahmen des Verbundvorhabens APPLY entstanden, das vom Bundesminister für Forschung und Technologie unter dem Kennzeichen „01IW205B/O“ gefördert wird.

mehr in dem Maße, wie es während der Programmentwicklung notwendig ist. Anwendungsprogramme können durch die Elimination des nicht benötigten Leistungsumfangs nicht nur bedeutend kleiner, sondern auch effizienter werden. Es entstehen von einem Lisp-System unabhängige Applikationen, da die Kompilationsergebnisse übliche Objektfiles oder C-Programme sind, die mit Betriebssystemmitteln zu fertigen Programmen verbunden werden können. Das ist eine wesentliche Voraussetzung für die wirkliche, sowohl technische als auch organisatorische Integration in ein DV-Umfeld.

Da der volle Sprachumfang von Lisp unter dem Kriterium der sparsamen Nutzung von Ressourcen, insbesondere Speicher, nicht kompilierbar ist, müssen fertige Programme in der Regel in eine Teilsprache migriert werden, in der einige der hochflexiblen, dynamischen Sprachkonstrukte, die den *Entwicklungsprozeß* von Programmen beschleunigen, nicht mehr auftreten. Im Rahmen von APPLY wurde zu diesem Zweck die Teilsprache CommonLisp_0 [Knu92] definiert. Praktisch bedeutet das eine gründliche manuelle Überarbeitung des Programms entsprechend einem methodischen Vorgehen, das wir in Abschnitt 3 ausarbeiten, und entsprechend allgemeiner Prinzipien des Software-Engineering. Sie geschieht im Dialog mit dem Compiler, der gleichzeitig eine Analyse der statischen Semantik des Programms vornimmt und dabei Programmierfehler aufdecken kann. Dieser Migrationsschritt entspricht seinem Wesen nach einer manuellen Transformation, die durch das Entwicklungssystem und auch den Übersetzer unterstützt wird. Der Grund ist, daß CommonLisp_0 eine strikte Teilsprache von Common Lisp ist und somit die im Sinne einer Übersetzung möglichen automatischen Sourcecode-Transformationen per se entfallen. Der Programmieraufwand für die Migration ist im Vergleich zu einer Reimplementation z. B. in C gering, und das Vorgehen ist wesentlich weniger fehleranfällig.

Nach vorliegenden Erfahrungen sind Migrationsschritte nur an wenigen Stellen eines Programms erforderlich, wo man sich beim Prototypen mit einer ad hoc Lösung durch ein dynamisches, universell einsetzbares Sprachmittel beholfen hat. Die Erfahrungen stammen aus der Migration und erfolgreichen Kompilation von Anwendungen, die in Common Lisp [Ste84] [Ste90] entwickelt wurden, in die im Rahmen des APPLY-Projektes entstandene strikte Teilsprache CommonLisp_0 von Common Lisp.

Die vorliegende Arbeit stellt das methodische Vorgehen, Möglichkeiten und Grenzen des Ansatzes sowie Resultate über Migrationen und Kompilationen von Common-Lisp-Anwendungen externer Partner des APPLY-Projektes vor. Wir setzen die Kenntnis von Common Lisp [Ste84] [Ste90] voraus, die für das Verständnis des durchgängig benutzten Beispiels nötig ist.

Ziel der Migration ist ein Programm der Sprache CommonLisp_0, das über ein Hauptprogramm verfügt und sich somit als eigenständige Applikation mit dem im Rahmen des Projektes APPLY entwickelten Komplettcompiler entweder als Ganzes oder modulweise in ein C-Programm und dann weiter in ein vom Lisp-System unabhängiges Maschinenprogramm übersetzen läßt. Dabei ist die Transformation von Lisp nach C und die dann anschließende Wartung und Weiterentwicklung des entstandenen C-Programmes ausdrücklich nicht ein Hauptziel dieser Arbeit. Vielmehr entsteht durch die Übersetzung C-Code, der eher dem einer abstrakten C-Maschine gleicht und nicht im Hinblick auf Lesbarkeit und Änderbarkeit, sondern im Hinblick auf Effizienz und Portabilität generiert wurde.

2 Ansatz und Aufgabenstellung

Diese Arbeit stellt einen Ansatz vor, wie Anwendungsprogramme in Lisp unter Ausnutzung des vollen Leistungsumfanges interaktiver Lisp-Systeme entwickelt und danach effizient, portabel und integrationsfähig implementiert werden können. Grundannahme ist, daß die Entwicklung eines Programmes von dessen Anwendung klar zu trennen und in der Anwendung nicht mehr der volle Leistungsumfang des Entwicklungssystems nötig ist.

2.1 Programmentwicklung

Die Entwicklung eines Lisp-Programmes geschieht in der Regel in einem interaktiven Common-Lisp-System, häufig im Sinne von *rapid prototyping*, und unter Ausnutzung des vollen Leistungsumfangs von Common Lisp. Der schnelle Entwicklungszyklus wird dabei durch das interaktive System selbst, durch die in einem konkreten System zusätzlich bereitgestellte Funktionalität und auch durch die der Sprache Lisp eigenen flexiblen und dynamischen Sprachkonstrukte unterstützt. Insbesondere der in einem interpretierenden System mögliche Test unvollständiger Programme spielt dabei eine wichtige Rolle.

Wir halten dies für ein Leistungsmerkmal von Lisp und plädieren ausdrücklich nicht für eine Programmentwicklung in der bereits im letzten Abschnitt erwähnten Sprache CommonLisp_0. Viele Lisp-Anwendungen, aber auch viele in Lisp-Anwendungen benutzte Programmiertechniken sind durch „Experimente in Lisp" entstanden, und dies stellt eine eigene Qualität dar, die nicht aufgegeben werden sollte.

Wir hoffen andererseits, daß der aus softwaretechnischer Sicht nötige und sich an die Programmentwicklung anschließende Migrationsschritt letztlich auch *normativ* auf die Programmentwicklung auswirkt. Genauer: Dort, wo der Programmentwicklungsprozeß durch die rechtzeitige Verwendung übersetzbarer oder effizienter Datenstrukturen und Algorithmen nicht gehemmt würde, sollten diese auch verwendet werden.

2.2 Migration

Analysiert man vorhandene Lisp-Applikationen genauer, so stellt man fest, daß sie an verschiedenen Stellen unverwechselbar den Charakter von Prototypen aufweisen:

- Sie enthalten häufig kein Hauptprogramm, sondern bestehen nur aus einer Reihe von Definitionen, und ein Programmablauf besteht aus dem mehr oder weniger koordinierten und dokumentierten Aufrufen der zur Verfügung gestellten Funktionalität in einem interaktiven Lisp-System. Es sei an dieser Stelle darauf hingewiesen, daß manche in Lisp implementierten Systeme ihrem Wesen nach den Charakter von Spracherweiterungen von Lisp haben, die im Zusammenhang mit einem interaktiven Lisp-System zur Programmentwicklung genutzt werden sollen. Diese Systeme selbst sind keine Lisp-Applikationen in dem hier verwendeten Sinne. Erst ihre Anwendungen würden wir als Lisp-Applikationen bezeichnen.
- Sie enthalten eine Reihe von Programmteilen, die den Entwicklungsprozeß unterstützen, die für einen Programmablauf aber völlig irrelevant sind. Dies sind z.B. Funktionen zum automatischen Rekompilieren, zum automatischen Nachladen oder zur Konsistenzerhaltung des Programmes, falls nur Teile redefiniert oder nachgeladen werden.
- Sie enthalten eine Reihe von Mehrfachdefinitionen derselben Funktion oder Prozedur, die in der Entwicklung der Reihe nach entstanden sind und von denen in einer interaktiven Entwicklungsumgebung jeweils nur die als letzte in das System geladene die gültige ist.
- Sie enthalten eine Reihe von Programmpfaden und auch Funktionsdefinitionen, die nur zu Testzwecken entstanden sind und die in einem tatsächlichen Programmlauf nicht aufgerufen werden. Neben totem Code, den jeder gute Compiler eliminieren kann, sind dies aber auch Programmteile, die zwar formal aufgerufen, tatsächlich jedoch nie erreicht werden.
- Sie enthalten eine Reihe von Programmiertricks unter Ausnutzung sehr flexibler dynamischer, häufig reflektiver Lisp-Konstrukte, die im Entwicklungsprozeß zugunsten der schnellen Entwicklung durchaus ihre Berechtigung hatten und dann im Programm verblieben sind.
- Sie nutzen häufig Funktionalität, die nicht zur Sprache Lisp, wohl aber zu dem in einem konkreten Lisp-System bereitgestellten Funktionsumfang gehört.

Man stellt also fest, daß die Programme ohne Änderungen nicht den erforderlichen Einschränkungen genügen, die für Programme einer effizient kompilierbaren Teilsprache von Common Lisp verlangt werden müssen.

Vor der Kompilation ist daher ein weiterer Schritt nötig, den wir als *Migration* bezeichnen. Unter diesem Begriff verstehen wir eine lokale oder globale Sourcecode-Transformation von Programmen in einer Sprache in Programme in einer anderen (in unserem Fall einer Teilsprache der ersten) unter Erhaltung der Funktionalität des ursprünglichen Programms. Der Begriff läßt sich zwischen automatischer, semantikerhaltender Programmtransformation und Reimplementation einordnen. (vgl. Abbildung 1)

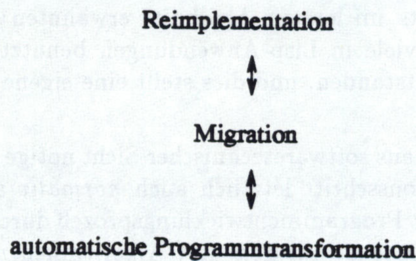


Abbildung 1: Migration

Dieser Migrationsschritt entspricht hier seinem Wesen nach einer manuellen Transformation, die durch das Entwicklungssystem und auch den Übersetzer unterstützt wird. Der Grund ist, daß CommonLisp$_0$ eine strikte Teilsprache von Common Lisp ist und somit die im Sinne einer Übersetzung möglichen automatischen Sourcecode-Transformationen per se entfallen, da sie syntaktisch und semantisch identischen Code erzeugen müßten. Die Striktheit der Teilsprachenbeziehung hat einen weiteren nutzbringenden Effekt: Die Migration kann inkrementell in einem Lisp-System durchgeführt werden, da jeder Migrationsschritt wieder zu einem gültigen Common-Lisp-Programm führt.

Nachdem der erste Migrationsschritt, die Montage des Programmes (siehe den folgenden Abschnitt) durchgeführt ist, läßt sich der Komplettcompiler als Werkzeug benutzen, um nötige Migrationsschritte aufzudecken und durch eine weitgehende statische Programmanalyse Programmierfehler zu finden.

Es wird in den folgenden Abschnitten deutlich, daß sich das Programm in seiner Struktur durch die Migration nicht wesentlich ändert, sondern höchstens um einige Teile ergänzt wird. Damit kann das migrierte Programm als Ausgangspunkt für die Wartung und Weiterentwicklung dienen, und Migrationsschritte sind nur noch für die dann neu entwickelten Teile nötig. (vgl. Abbildung 2)

2.3 Komplettkompilation

Nachdem ein gültiges CommonLisp$_0$-Programm entstanden ist, kann der Komplettcompiler daraus durch Übersetzung nach C eine eigenständig ablauffähige Applikation generieren. Diese zeichnet sich aus durch

- einen sparsamen Umgang mit Ressourcen, d.h. durch Speicherplatz- und Laufzeiteffizienz,
- Portabilität auf der Basis von C-Sourcecode und
- die Integrationsfähigkeit, zum einen mit Fremdsoftware, zum anderen als Programm oder Modul in Standard-DV-Umgebungen.

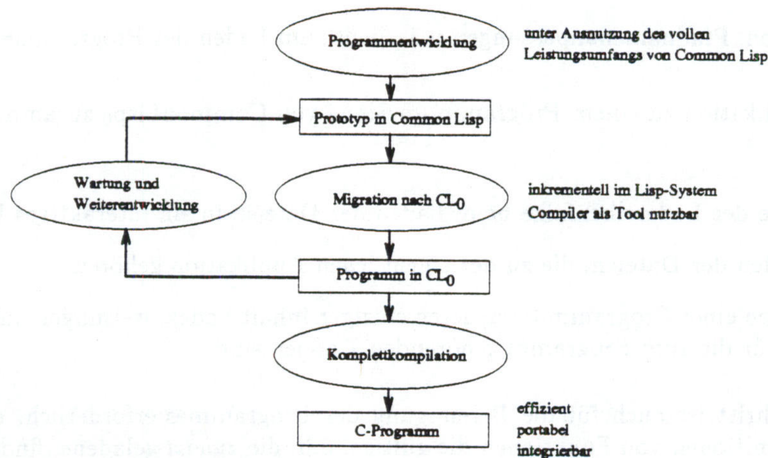


Abbildung 2: Migration und Kompilation

3 Methodisches Vorgehen

In diesem Abschnitt soll das methodische Vorgehen bei der Migration von Common-Lisp-Anwendungen nach CommonLisp_0 aufgezeigt werden. Zunächst gilt es, festzustellen, ob ein Programm tatsächlich als schlüsselfertige Applikation vorliegt, für die es prinzipiell Sinn macht, sie mit einem explizit ausprogrammierten Hauptprogramm zu versehen und als eigenständig ablauffähiges Programm auszuliefern.

Die nötigen Arbeiten, die wir im folgenden motivieren und genauer beschreiben wollen, lassen sich wie folgt klassifizieren:

1. Montage des Programmes
2. Bereinigung des Programmes
3. Erzeugen eines Hauptprogrammes
4. Elimination von Reflexion
5. Entkopplung vom Entwicklungssystem
6. Modularisierung

Die ersten vier Schritte werden in der Regel in der angegebenen Reihenfolge durchgeführt, die Montage des Programmes ist notwendige Voraussetzung für alle anderen Schritte. Die letzten beiden Schritte sind nur für spezielle Anwendungen erforderlich, insbesondere ist die Modularisierung nur dann zwingend, wenn die separate Kompilation einzelner Module des Programmes angestrebt wird. Dies ist typischerweise bei großen Anwendungen der Fall, oder wenn die Anwendung mit Fremdsprachenmodulen integriert werden soll.

3.1 Montage des Programmes

In der Regel sind Lisp-Anwendungen in verschiedenen Dateien abgelegt, und es ist nicht leicht, das eigentliche Programm zwischen ebenfalls vorhandenen Testroutinen und Hilfsmitteln zur Erleichterung des Entwicklungsprozesses auszumachen. Eine mehr oder weniger komplizierte Laderoutine, teilweise mit Unterstützung von *dynamischem Nachladen* (engl. *autoloading*) oder automatischer

Rekompilation, Pfadnamenanpassungen u.ä. dient zum Laden des Programmes in ein interaktives Lisp-System.

Um die Applikation zu einem Programm im Sinne von CommonLisp$_0$ zusammenzusetzen, ist folgendes nötig:

- Analyse der Ladereihenfolge beim Laden der Dateien in ein interaktives Lisp-System.
- Auffinden der Dateien, die zu der eigentlichen Applikation gehören.
- Erzeugen einer Programmdatei, deren einziger Inhalt Ladeanweisungen auf Hauptprogrammebene für die zum Programm gehörenden Dateien sind.

Der erste Schritt ist auch für die Bereinigung des Programmes erforderlich, um bei eventuellen Mehrfachdefinitionen von Funktionen die gültige, d.h. die *zuletzt* geladene, finden zu können.

Die Haupt-Programmdatei stellt nunmehr das eigentliche Programm dar, das allerdings noch nicht über ein Hauptprogram verfügen muß (vgl. Abschnitt 3.3).

3.2 Bereinigung des Programmes

Da ein interaktives Lisp-System Mehrfachdefinitionen von Funktionen zuläßt (die letzte geladene Funktion ist die aktuell verwendete), in CommonLisp$_0$ Mehrfachdefinitionen aber nicht zulässig sind, sind die nicht aktuellen Funktionsdefinitionen zu streichen. Ebenfalls sollten die Programmteile, die ausschließlich zu Testzwecken oder zur Erleichterung des Entwicklungsprozesses dienen, eliminiert werden, soweit dies nicht bereits bei der Montage des Programmes geschehen ist.

Sind nur noch die Bestandteile der eigentlichen Applikation übrig und ist eine Haupt-Programmdatei (vgl. den vorherigen Abschnitt) vorhanden, so kann nunmehr der Compiler als Werkzeug zur statischen Programmanalyse verwendet werden, um weitere Stellen zu finden, an denen Migrationsarbeiten, insbesondere auch zur Bereinigung des Programmes, vorgenommen werden müssen.

Häufig finden sich Aufrufe nicht definierter Funktionen. Diese sind entweder zu definieren oder sie befinden sich in Programmpfaden, die nie durchlaufen werden. so daß sie als *toter Code* gelöscht werden können. Diese Situation kann auftreten, wenn in einem Programm Funktionalität vorgesehen war, die aber nicht oder anders implementiert worden ist.

Funktionalität, die nicht Bestandteil der in [Ste90] angegebenen Sprache Common Lisp ist, sondern von dem speziellen interaktiven Common-Lisp-System *geborgt* wurde, muß reimplementiert werden. Dies hat natürlich immer zu geschehen, wenn eine Anwendung in nicht portablem Common Lisp entwickelt wurde und nun portiert werden soll.

3.3 Erzeugen eines Hauptprogrammes

Eine Lisp-Anwendung, die nur aus Definitionen besteht, kann im Zusammenhang mit einem interaktiven Lisp-System durchaus nutzbar sein, da dieses die Dialog-Umgebung der Anwendung darstellen kann. Zum Zwecke der Komplettkompilation muß allerdings ein *Hauptprogramm* explizit ausprogrammiert werden, das Bestandteil der Haupt-Programmdatei werden sollte.

Dieser Schritt ist in vielen Anwendungen leicht, da nur der Aufruf der Hauptfunktion in die Haupt-Programmdatei zusätzlich aufgenommen werden muß. Existiert allerdings keine Hauptfunktion, so ist in der Regel zusätzliche Design- und Implementationsarbeit zu leisten:

- Der Umfang der dem Programmbenutzer als *Dialogschnittstelle* zur Verfügung gestellten Funktionalität muß festgelegt werden.

- Die so festgelegte Dialogschnittstelle muß implementiert werden. Ihr Aufruf wird zum Hauptprogramm, das nunmehr nur noch antizipierte Dialoge bearbeiten kann. Der Aufruf beliebiger Funktionen der Anwendung wird damit unterbunden, was die Dynamik einschränkt, aber auch die Sicherheit gegen Fehlbedienungen erhöht.

3.4 Elimination von Reflexion

In [Smi82] [Smi84] taucht der Begriff *Reflexion* (engl. *reflection*) zuerst auf. Programme, die „Wissen über ihren eigenen Programmtext" haben, werden dort als reflektiv bezeichnet. Folgerichtig nennen wir Programmiersprachen, die die Programmierung reflektiver Programme gestatten, ebenfalls reflektiv, und die Sprachbestandteile, die das Ausdrücken von Reflexion gestatten, nennen wir reflektive Sprachkonstrukte der Sprache.

M. Wand und D.Friedman [WF86] unterscheiden die Begriffe *reflection* und *reification*. Der Übergang von Programm zu Datum, genauer von einem Abschnitt des abstrakten mathematischen Objektes *Programm* in dessen textuelle Repräsentation, wird dort als *reification* bezeichnet. Als *reflection* wird der eigentlich kritische Übergang von Datum zu Programm bezeichnet, also die Einbindung von Programmtext in das abstrakte Programm, die im ungünstigen Fall eben auch eine Modifikation des Programmes selbst hervorrufen kann.

Lisp, speziell auch Common Lisp, ist eine reflektive Programmiersprache, in der auch der ungünstige Fall der Selbstmodifikation eines Programmes ausdrückbar ist. Um es ganz deutlich zu machen: Es ist nicht der Umgang eines Programmes mit Funktionen oder Prozeduren als Daten erster Klasse (*first class citizens*), der die Reflexion ausmacht. Dieser beinhaltet keine Uminterpretation von Daten in Programm. Es ist vielmehr die in Lisp mögliche freie Uminterpretation von Lisp-Daten, speziell Listen, in Programmausdrücke, also die Ausführung von zum Programm gehörenden, aber im Programmtext i.a. nicht sichtbaren Lisp-Ausdrücken.

Der klassische Vertreter eines reflektiven Sprachbestandteils von Lisp ist die Funktion `(eval ... )`, die einen als Lisp-Liste übergebenen Ausdruck in der globalen Laufzeitumgebung des Programmes auswertet oder ausführt.

Dem Leser dürfte schnell klar sein, welche Konsequenzen z.B. das Vorhandensein einer Funktionsdefinition der Form

```
(defun f () (eval (read)))
```

in einem Programm hat. Zunächst kann der Übersetzer diese Funktion ohne die Möglichkeit der Kompilation zur Laufzeit des durch `(read)` eingegebenen Ausdrucks nicht implementieren. Zum zweiten können keine globalen Programmoptimierungen mehr durchgeführt werden, die von der vollständigen Kenntnis des Programmtextes abhängen, da dieser eben nicht bekannt ist. Zum dritten erlaubt beispielsweise die Eingabe des Ausdrucks

```
> (defun g (...) ...)
```

die Definition zusätzlicher oder die Modifikation vorhandener Funktionen des Programmes zur Laufzeit, speziell z.B. auch die Modifikation der Funktion `f` selbst.

Insgesamt werden Programme dadurch i.a. unnötig groß, unnötig langsam und aus softwaretechnischer Sicht fehleranfälig und nahezu unwartbar.

Trotzdem ist die durch `(eval... )` und verwandte Sprachkonstrukte gegebene hohe Flexibilität während des Programmentwicklungsprozesses durchaus sehr hilfreich.

Reflektive Bestandteile von Common Lisp

Wir beschränken uns im Rahmen dieser Arbeit auf den imperativen Kern der Sprache Common Lisp ohne die objektorientierte Erweiterung CLOS, die mit ihrem Meta-Objekt-Protokoll ebenfalls reflektive Sprachkonstrukte enthält.

Natürlich ist die Funktion `(eval... )` reflektiv. Ebenso klar ist, daß die Funktionen `(compile... )` und `(load... )` die gleiche Mächtigkeit haben. Diese Funktionen dürfen in der Sprache CommonLisp$_0$ nicht vorkommen. Eine Ausnahme bildet `(load... )`, das auf Hauptprogrammebene mit konstantem Dateinamen-Argument erlaubt ist und syntaktisch in den Inhalt der angegebenen Datei expandiert.

Die Funktionen

```
(funcall <function> <arg-1> ... <arg-n>)
(apply <function> <arg-1> ... <arg-k> <arglist>)
```

erlauben in Common Lisp auf der Position des funktionalen Arguments auch Symbole (interpretiert als Funktionsnamen) und Listen der Form **(lambda *parlist body*)** (als Funktionen interpretiert). Sie erlauben auf der Argumentposition Ausdrücke, die zu Funktionsnamen oder Lambda-Listen evaluieren. Damit läßt sich die Funktion `eval` natürlich leicht durch

```
(defun eval (form) (funcall (list 'LAMBDA () form)))
```

programmieren. Für diese Funktionen wie auch für `(mapcar... )`, `(mapc... )` usw. ist in CommonLisp$_0$ der Definitionsbereich eingeschränkt, indem nur Funktionen, nicht aber Funktionsnamen oder Lambda-Listen auf der Position des funktionalen Arguments zugelassen sind.

Die Funktion `( setf symbol-function)` erlaubt in Common Lisp durch

```
(setf (symbol-function 'f) ...)
```

die Redefinition globaler Funktionen. Diese Funktion ist in CommonLisp$_0$ nicht vorhanden.

Es gibt einige weitere Stellen, z.B. in Typdefinitionen, beim Einlesen konstanter Strukturen oder in der Funktion `(format... )`, die in CommonLisp$_0$ eingeschränkt sind, da sich auch hier die Möglichkeit der Interpretation eines beliebigen Ausdrucks zur Laufzeit des Programmes programmieren läßt. Im Rahmen der vorliegenden Arbeit soll auf eine vollständige und genaue Darstellung verzichtet werden.

Programmspezifische Interpretation

Wie bereits aus den Bemerkungen des vorherigen Abschnitts deutlich geworden ist, verbietet CommonLisp$_0$ alle reflektiven Sprachbestandteile von Common Lisp, teilweise durch Streichen, teilweise dadurch, daß der Definitionsbereich von Funktionen derart eingeschränkt ist, daß ein dynamischer Übergang von Datum zu Programm nicht mehr möglich ist. Reflektive Sprachbestandteile sind also zu eliminieren.

Die universelle Lisp-Funktion `(eval... )` zur Interpretation beliebiger Lisp-Ausdrücke stellt häufig sehr viel mehr Funktionalität zur Verfügung, als mit ihrer Verwendung intendiert ist. Ein Grund dafür ist die komplizierte Semantik von `(eval... )` [WF86]. In der Regel weiß der Programmierer jedoch sehr genau, welche Art von Ausdrücken durch die Verwendung von `(eval... )` interpretiert werden sollen. In den meisten Fällen handelt es sich um den Aufruf von Funktionen, die ihrerseits durch das Programm definiert werden. Hier liegt es nahe, statt des allgemeinen Interpretationsmechanismus' einen programmspezifischen Interpreter auszuprogrammieren, der genau die

gewünschten Ausdrücke interpretieren kann und alle weiteren, die beispielsweise durch Programmierfehler oder Fehlbedienungen entstehen, explizit abweist. Wir wollen dies in einem Beispiel tun, indem wir

```
(eval (read))
```

unter Ausnutzung der zusätzlichen Information, daß nur Ausdrücke der Form

$$e \ ::= \ number \ | \ (+\ e_1 \ldots e_n\ ) \ | \ (-\ e_1 \ldots e_n\ ) \ | \ (*\ e_1 \ldots e_n\ )$$

interpretiert werden sollen, in den Ausdruck

```
(my-eval (read))
```

ändern, wobei die Funktion **my-eval** definiert ist durch

```
(defun my-eval (form)
  (cond ((numberp form) form)
        ((eq (car form) '+)
           (apply #'+ (mapcar #'my-eval (cdr form))))
        ((eq (car form) '-)
           (apply #'- (mapcar #'my-eval (cdr form))))
        ((eq (car form) '*)
           (apply #'* (mapcar #'my-eval (cdr form))))
        (T (error "unknown form ~s" form))))
```

Es ist klar, daß es sich hierbei um eine Programmtransformation handelt, die nicht semantisch äquivalent ist. Das geänderte Programm verhält sich aber bzgl. der intendierten „äußeren" Semantik äquivalent und ist zudem sicherer und wartbarer geworden, da Fehlbedienungen ausgeschlossen wurden und mehr Informationen explizit niedergeschrieben sind.

Natürlich lassen sich bestimmte Anwendungen von **(eval...)**, **(funcall...)** oder **(apply...)** auch ohne Kenntnis der äußeren Semantik des Programmes semantisch äquivalent transformieren, beispielsweise gilt bei leerer lexikalischer Umgebung

```
(eval '(f arg-1 ... arg-n))          ==  (f arg-1 ... arg-n)
(eval (cons 'f arglist))             ==  (apply #'f arglist)
(apply 'f arglist)                   ==  (apply #'f arglist)
(apply '(lambda (...)...) arglist)
      ==  (apply #'(lambda (...)...) arglist)
```

Diese Fälle kann natürlich auch der Übersetzer selbst abdecken, so daß sie für die Migration eher uninteressant sind. Es wird an dieser Stelle noch einmal deutlich, daß die Migration ihrem Wesen nach eine Aufgabe ist, die der Programmierer eigenhändig und unter Ausnutzung seiner Kenntnis über die Bedeutung oder die intendierte äußere Semantik seines Programmes vornehmen sollte.

Ersetzen von Ausdrücken durch Closures

Sehr häufig finden sich in Lisp-Programmen globale konstante Datenstrukturen, die der Speicherung von zu interpretierenden Ausdrücken oder von in bestimmten Fällen aufzurufenden Funktionen dienen. Als Beispiel seien hier eine Assoziationsliste mit Zuordnungen von Symbolen zu Interpretationsfunktionen

```
(defvar *fun-alist* '((+ . eval-plus) (- . eval-minus) ...))
```

oder auch einfach eine Liste von Ausdrücken

```
(defvar *expr-list* '((+ 3 *x*) (* *y* *x*) ...))
```

genannt. Typischerweise tauchen dann in dem Programm Ausdrücke der Form

```
(apply (cdr (assoc '+ *fun-alist*)) args)
```

oder

```
(eval (second *expr-list*))
```

auf, die natürlich nicht im Sprachumfang von CommonLisp_0 enthalten sind, da z.B. der Ausdruck `(cdr (assoc '+ *fun-a-list*))` ein Symbol, nicht aber eine Funktion liefern wird.

In derartigen Fällen ermöglicht der Übergang von Ausdrücken zu *closures*, d.h. zu parameterlosen unbenannten Funktionen, bzw. der Übergang von Symbolen zu den entsprechenden Funktionen selbst den nötigen Migrationsschritt. Im ersten Fall ersetzen wir die konstante Assoziationsliste durch eine, die statt der Symbole **eval-plus** usw. die Funktionen **#'eval-plus** usw. enthält:

```
(defvar *fun-alist*
  (list (cons '+ #'eval-plus)
        (cons '- #'eval-minus)
        ...))
```

und können den Aufruf sogar unverändert lassen. Im zweiten Fall ersetzen wir die Liste von Ausdrücken durch eine Liste von *closures*

```
(defvar *expr-list*
  (list #'(lambda () (* 3 *x*))
        #'(lambda () (* *y* *x*))
        ...))
```

und ersetzen den Ausdruck **(eval (second *expr-list*))** durch den Funktionsaufruf

```
(funcall (second *expr-list*))
```

3.5 Ein Beispiel

An einem sehr kleinen Beispiel, einem Programm zum symbolischen Differenzieren von Polynomen, wollen wir eine typische Programmentwicklung, die entstehende Lisp-Anwendung und die wesentlichen Schritte bei der Migration zu einem CommonLisp_0-Programm aufzeigen.

Programmentwicklung

Die Aufgabe ist die Entwicklung eines symbolischen Differenzierers für Polynome:

Eingabe: $p(x) = a_n x^n + \cdots + a_1 x + a_0$

Ausgabe: $p'(x) = n a_n x^{n-1} + \cdots + a_1$

Wir gehen im Sinne des *rapid prototyping* vor und vereinfachen die Aufgabenstellung in einer für Lisp typischen Weise, indem wir die Programmierung der konkreten Text-Ein/Ausgabe verschieben und zunächst den Kern, nämlich die Differenzierung einer abstrakten Listendarstellung des Polynoms, entwickeln. Die zu Testzwecken nötige Eingabe stellt der Lisp-Reader zur Verfügung. Damit ist die Aufgabe nun sehr einfach geworden:

Wir stellen das Polynom in abstrakter Syntax dar, aufgebaut aus Ausdrücken der Form

(PLUS e_1 e_2**)**
(MINUS e_1 e_2**)**
(MULT e_1 e_2**)**
(EXPP e n**)** .

Das Polynom $p(x) = x^2 + 2x + 2$ beispielsweise wird dargestellt durch die folgende Lisp-Liste:

```
(PLUS (EXPP x 2) (PLUS (MULT 2 x) 2))
```

Mit den Ableitungsregeln

- $c' = y' = 0$ für Konstanten c und Variablen $y \not\equiv x$
- $x' = 1$
- $(f + g)' = f' + g'$
- $(f\,g)' = f'g + fg'$
- $(f^n)' = nf'f^{n-1}$ für $n \geq 2$

ist es nun einfach, die Funktion **diff** auszuprogrammieren:

```
(defun diff (e x)
  (cond ((numberp e) 0)
        ((eq e x) 1)
        ((symbolp e) 0)
        ((consp e)
          (case (first e)
            ((PLUS MINUS) ...)
            ((MULT) (list 'PLUS
                      (list 'MULT (diff (second e) x)
                            (third e))
                      (list 'MULT (second e)
                            (diff (third e) x))))
            ((EXPP) ...)
            (t e)))
        (t e)))
```

Die komplette Funktionsdefinition findet sich im Anhang. Ein erster Test ergibt nun, daß wir mit dem Ergebnis noch nicht zufrieden sind, da das differenzierte Polynom noch erheblich vereinfacht werden kann:

$$p(x) = x^2 + 2x + 2$$

```
(diff '(PLUS (EXPP x 2) (PLUS (MULT 2 x) 2)) 'x) =
   (PLUS (MULT 1 (MULT 2 (EXPP X (MINUS 2 1))))
         (PLUS (PLUS (MULT 0 X) (MULT 2 1)) 0))
```

Die Vereinfachung kann durch partielles Auswerten des Ergebnisses geschehen, indem wir die Funktionen **PLUS**, **MINUS**, **MULT**, **EXPP** schreiben und den entstandenen Ausdruck evaluieren. Als Beispiel seien die Funktionen **MULT** und **EXPP** mit den entsprechenden Auswertungsregeln angegeben:

$$(\texttt{MULT}\ e_1\ e_2) = \begin{cases} e_1 e_2 & \text{, falls } e_1, e_2 \text{ Zahlen sind} \\ 0 & \text{, falls } e_1 = 0 \text{ oder } e_2 = 0 \\ e_2 & \text{, falls } e_1 = 1 \\ e_1 & \text{, falls } e_2 = 1 \\ (\texttt{MULT}\ e_1\ e_2) & \text{, sonst} \end{cases}$$

```
(defun MULT (e1 e2)
  (cond ((and (numberp e1) (numberp e2)) (* e1 e2))
        ((or (eql 0 e1) (eql 0 e2)) 0)
        ((eql e1 1) e2)
        ((eql e2 1) e1)
        (t (list 'MULT e1 e2))))
```

$$(\texttt{EXPP}\ e\ n) = \begin{cases} e^n & \text{, falls } e, n \text{ Zahlen sind} \\ 1 & \text{, falls } n = 0 \\ e & \text{, falls } n = 1 \\ (\texttt{EXPP}\ e\ n) & \text{, sonst} \end{cases}$$

```
(defun EXPP (e n)
  (cond ((and (numberp e) (numberp n)) (^ e n))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))
```

Die Funktion zur partiellen Auswertung der entstehenden Ausdrücke, die wir **simplify** nennen wollen, kann dann einfach die Lisp-Funktion **eval** aufrufen. Damit ergibt sich eine erste Version des Programmes, vervollständigt durch Definitionen der Funktionen **parse** und **unparse**, die aufgrund der verabredeten Vereinfachung nur die als Argument übergebene abstrakte Darstellung des Polynoms identisch zurückgeben:

```
(defun diff ... )

(defun PLUS ... )
(defun MINUS ... )
(defun MULT ... )
(defun EXPP ... )

(defun simplify (p) (eval p))
(defun parse (p) p)
(defun unparse (p) p)
(defun test (p)
  (unparse (simplify (diff (parse p) 'x))))
```

Das komplette Programm findet sich im Anhang.

Migration

Zur Montage des Programmes ist nichts zu tun. Eine zusätzliche Haupt-Programmdatei ist nicht erforderlich. Wir können also den Übersetzer auf das Programm ansetzen, um Stellen zu finden, bei denen Migrationsschritte nötig sind. Die statisch semantische Analyse des Übersetzers zeigt uns, daß weder die Funktion `(eval... )`, deren Aufruf in der Funktion `simplify` auftritt, noch die Funktion `(^... )`, die in `EXPP` aufgerufen wird, im Sprachumfang von CommonLisp_0 vorhanden sind.

Die Funktion `(^... )` ist tatsächlich keine definierte Funktion in Common Lisp. Wir haben einen Programmierfehler entdeckt, der deshalb bislang nicht zutage getreten ist, weil der entsprechende Fall in der Funktion `EXPP` nie erreicht wurde und auch nie erreicht werden wird. Ausdrücke der Form (`EXPP` n_1 n_2) mit Zahlen n_1 und n_2 tauchen in den Ergebnissen von `diff` nicht auf, wenn wir sicherstellen, daß die Funktion `parse` reduzierte abstrakte Repräsentationen liefert. Damit kann der Aufruf von `(^... )` beispielsweise durch die Fehlermeldung

```
(error "in function EXPP")
```

ersetzt werden.

Um den Aufruf der Funktion `(eval... )` zu eliminieren, entscheiden wir uns für die Programmierung eines programmspezifischen Interpretierers `diff-eval`, den wir stattdessen in der Funktion `simplify` aufrufen. Dieser läßt sich nach demselben Schema programmieren, das wir bereits im Abschnitt 3.4 benutzt haben:

```
(defun diff-eval (e)
  (cond
    ((or (numberp e) (symbolp e)) e)
    ((eq (first e) 'PLUS)
      (apply #'PLUS (mapcar #'diff-eval (rest e))))
    ((eq (first e) 'MINUS)
      (apply #'MINUS (mapcar #'diff-eval (rest e))))
    ((eq (first e) 'MULT)
      (apply #'MULT (mapcar #'diff-eval (rest e))))
    ((eq (first e) 'EXPP)
      (apply #'EXPP (mapcar #'diff-eval (rest e))))
    (t (error "Unknown expression ~s." e))))
```

Ergänzen wir das entstandene Programm nun noch um ein Hauptprogramm, das aus einer Schleife besteht, die jeweils ein zu differenzierendes Polynom einliest und das differenzierte Polynom ausdruckt, dann entsteht letztlich als Ergebnis der Migration das folgende Programm, das im Anhang vollständig wiedergegeben ist:

```
(defun diff ... )

(defun PLUS ... )
(defun MINUS ... )
(defun MULT ... )

(defun EXPP (e n)
  (cond ((and (numberp e) (numberp n)) (error "in function EXPP"))
        ((eql 0 n) 1)
        ((eql n 1) e)
```

```
         (t (list 'EXPP e n))))

(defun diff-eval (e) ... )
(defun simplify (p) (diff-eval p))

(defun parse (p) p)
(defun unparse (p) p)

(defun test (p)
  (unparse (simplify (diff (parse p) 'x))))

(defun main ()
  (loop
    (princ "Enter expression : ")
    (let ((e (read)))
      (if (member e '(quit exit end stop halt))
        (return)
        (format t "~&~s~%" (test1 e))))))

(main)
```

Dieses Programm können wir nun komplett übersetzen. Die generierte eigenständig ablauffähige Applikation führt das Hauptprogramm, also den Aufruf der Funktion **(main)** aus. Im Anschluß würden die Funktionen **parse** und **unparse** entwickelt werden, die eine Textrepräsentation eines Polynoms lesen und in abstrakte Syntax umwandeln bzw. erzeugen.

4 Resultate

Mit dem in dieser Arbeit vorgestellten methodischen Vorgehen sind im Rahmen des APPLY-Projektes eine Reihe mittelgroßer bis großer Lisp-Applikationen auch externer Partner des Projektes erfolgreich migriert, übersetzt und auch auf kleine Anwendungsrechner portiert worden. Exemplarisch seien hier zwei Expertensystemanwendungen und ein Interpreter für eine KI-Sprache genannt, die im folgenden mit XPS1, XPS2 und KI-S bezeichnet werden sollen.

Die folgende Tabelle zeigt für diese Anwendungen einen Vergleich der Codegrößen. Dabei sind die Sourcecode-Größen, die Binärcode-Größe eines mit herkömmlicher Technik erzeugten Lisp-Images in Allegro-CL 4.1 und die Binärcode-Größen der mit der hier vorgestellten Technik übersetzten ausführbaren Programme zueinander in Beziehung gesetzt:

	XPS 1	XPS 2	KI-S
Lisp-Source-Code	10120 lines of code 485 KB	10671 lines of code 505 KB	3185 lines of code 92.4 KB
C-Source-Code	2073.5 KB	2212.1 KB	556 KB
Lisp-Image	8.6 MB	8.8 MB	7.8 MB
Programm (SPARC)	794.6 KB	942.0 KB	464.0 KB
Programm (AT-386)	620.9 KB	747.9 KB	178.0 KB

Alle hier angegebenen Programme sind auf PC-ATs ohne Einschränkungen ablauffähig, sogar auf Notebooks mit nur 2MByte Hauptspeicher.

Fassen wir die Vorteile des Vorgehens im Hinblick auf die entstandenen ablauffähigen Programme zusammen, so ergibt sich:

1. Sparsamer Umgang mit Ressourcen
   - Drastische Verringerung der Code-Größen
   - AblaufFähigkeit auf Kleinrechnern (Expertensystem auf einem 2MB Notebook)
2. Portabilität
   - maschinenunabhängiger C-Code
3. Integrierbarkeit
   - als Programme in Standard-DV-Umgebungen
   - mit C-basierter Standardsoftware, z.B. Bedienoberflächen, Datenbanken oder Kommunikationssoftware

Das in dieser Arbeit vorgeschlagene Vorgehen, um Lisp-Anwendungen zu entwickeln, zu migrieren und sie schließlich als eigenständig ablauffähige Applikationen zu implementieren, hat bei einer Reihe von nichttrivialen Anwendungen zum Erfolg geführt.

Dabei bleibt die Produktivität beim Programmieren in Lisp erhalten, wegweisende Entwicklungen in der symbolischen Datenverarbeitung und Wissensverarbeitung werden nicht gehemmt und das schnelle Entstehen von Prototypen wird begünstigt.

Wir haben ein methodisches Vorgehen vorgeschlagen und ausgearbeitet, wie die entstandenen Prototypen zu Anwendungen weiterentwickelt werden können, die den klassischen Anforderungen an Laufzeit- und Speicherplatzeffizienz genügen, und die insbesondere der Anforderung gerecht werden, sich in vorhandene Datenverarbeitungsumwelten integrieren zu lassen.

Sie können Industriestandards nutzen, wie beispielsweise Datenbanken, Bedienoberflächen oder Kommunikationssoftware. Dies ist nicht neu. Auch die bekannten kommerziell verfügbaren Common-Lisp-Systeme bieten durch ein Foreign-Language-Interface die technischen Voraussetzungen dazu.

Neu ist, daß sich Lisp-Programme selbst diesen Standards entsprechend verhalten, sich also als Programme oder Module wie solche jeder anderen Programmiersprache nutzen lassen. Anwender müssen nicht Lisp lernen, um Lisp-Programme nutzen zu können, und die Anwendungsumgebung muß nur noch die Rechenleistung für die Anwendung vorhalten, nicht aber die für die Nutzung eines kompletten Lisp-Systems nötige.

Der Preis, den man für diese Vorteile zu zahlen hat, ist der Arbeitsaufwand für die Migration. Wir meinen, daß dieser aus softwaretechnischer Sicht sowieso nötig ist, um von Prototypen zu Anwendungen zu gelangen.

5 Vergleichbare Arbeiten

Es gibt heute eine Reihe weiterer Übersetzerbauprojekte, in denen C erfolgreich als Zielsprache der Übersetzung höherer Sprachen verwendet wird [Hof93]. Dazu gehören Übersetzer von Fortran, Scheme und ML nach C. Aber auch die Übersetzung von Common Lisp nach C erfährt mehr und mehr Beachtung. Es seien hier z.B. KCL (Kyoto Common Lisp), WCL (Wade Common Lisp) und auch der kommerziell verfügbare Chestnut Lisp-to-C-Translator genannt.

KCL [TM86] ist ein in C implementiertes interaktives Lisp-System. Der inkrementelle Übersetzer erzeugt zwar C-Code, der aber nicht eigenständig ablauffähig ist, sondern durch dynamisches Linken als inkrementell übersetzter Programmbestandteil im Entwicklungssystem nutzbar wird.

WCL [Hen92] verfolgt einen eher mit dem von APPLY vergleichbaren Ansatz. In WCL werden Common-Lisp-Programme ebenfalls nach C übersetzt, die ablauffähigen Programme benötigen allerdings die vollständige Common-Lisp-Laufzeitbibliothek in Form einer sogenannten *shared library*.

Damit ist die Portabilität der Anwendungen im wesentlichen auf Unix-Umgebungen beschränkt, zudem besteht wegen der Reflektivität des vollen Common Lisp kaum eine nennenswerte Möglichkeit, globale Programmoptimierungen auszunutzen.

Am ehesten vergleichbar mit dem APPLY-Ansatz ist der kommerziell verfügbare Chestnut Lisp-to-C-Translator, der eigenständig ablauffähigen C-Code erzeugt, dies aber mit einem Hauptaugenmerk auf die Lesbarkeit und Wartbarkeit des entstehenden C-Codes, der im APPLY-Ansatz nur als portabler Zwischencode dient und im Hinblick auf Effizienz, nicht aber in erster Linie im Hinblick auf Lesbarkeit erzeugt wird.

Literatur

[BCF+91] Harry Bretthauer, Thomas Christaller, Horst Friedrich, Wolfgang Goerigk, Winfried Heicking, Dieter Hovekamp, Heinz Knutzen, Jürgen Kopp, E. Ulrich Kriegel, Ingo Mohr, Rainer Rosenmüller und Friedemann Simon. Spezifikation der Funktionalität des APPLY-Systems. APPLY-Arbeitspapier APPLY/IfKI/I.1/1, CAU/GMD/IfKI/VW-GEDAS, Sankt Augustin, Oktober 1991.

[BGK91] O. Burkart, W. Goerigk und H. Knutzen. CLICC: A New Approach to the Compilation from Common Lisp Programs to C. In *Workshop der Fachgruppe 2.1.4. "Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, erschienen als Bericht Nr. 8/91-I des Instituts für angewandte Mathematik und Informatik der Univ. Münster, 1991.

[GS91] W. Goerigk und F. Simon. Zielsetzungen im Verbundvorhaben APPLY. In *Workshop der Fachgruppe 2.1.4. "Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, erschienen als Bericht Nr. 8/91-I des Instituts für angewandte Mathematik und Informatik der Univ. Münster, 1991.

[Hen92] W. Hennessey. WCL: Delivering Efficient Common Lisp Applications under Unix. In *Proc. of the 1992 ACM Conference on Lisp and Functional Programming*, S. 260–269, San Francisco, CA, 1992.

[Hof93] Ulrich Hoffmann. Über C als Zielcode der Übersetzung hoher Programmiersprachen. APPLY-Arbeitspapier APPLY/CAU/IV.4/1, Christian-Albrechts-Universität, Kiel, Mai 1993.

[KG93] U. Knemeyer und J.-M. Graf von der Schulenburg. „Expertensysteme" - Welche Faktoren fördern und hemmen die Implementation und Diffusion der Technologie in der Versicherungswirtschaft. In F. Puppe und A. Günter, Hrsg., *Expertensysteme '93*, Berlin, Heidelberg, New York, 1993. Informatik aktuell, Springer Verlag.

[Knu92] Heinz Knutzen. COMMONLISP$_0$-Sprachdefinition. APPLY-Arbeitspapier APPLY/CAU/II.2/1, Christian-Albrechts-Universität, Kiel, April 1992.

[MBG90] P. Mertens, V. Barowski und W. Geis. *Betriebliche Expertensystem-Anwendungen*. 2. Auflage, Springer Verlag, Berlin, Heidelberg, New York, 1990.

[Smi82] B.C. Smith. Reflection and Semantics in a Procedural Language. Technischer Bericht MIT-LCS-TR-272, Massachusetts Institute of Technology, Cambridge, MA, 1982.

[Smi84] B.C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages POPL'84*, S. 23–35, 1984.

[Ste84] G.L. Steele. *Common Lisp: The Language*. Digital Press, Bedford, MA, 1984.

[Ste90] G.L. Steele. *Common Lisp: The Language. Second Edition.* Digital Press, Bedford, MA, 1990.

[TM86] Y. Taiichi und H. Masami. Kyoto Common Lisp Report. Technical report, Research Institute for Mathematical Science, Kyoto University, 1986.

[Weu93] H. Weule. Expertensysteme im industriellen Einsatz. In F. Puppe und A. Günter, Hrsg., *Expertensysteme '93*, Berlin, Heidelberg, New York, 1993. Informatik aktuell, Springer Verlag.

[WF86] M. Wand und D.P. Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 ACM-Conference on Lisp and Functional Programming*, S. 298–307, Cambridge, MA, 1986.

A Das Beispielprogramm

```
;;;---------------------------------------------------------------------
;;;
;;; Symbolisches Differenzieren von Polynomen
;;;
;;;---------------------------------------------------------------------

(defun diff (e x)
  (cond ((numberp e) 0)
        ((eq e x) 1)
        ((symbolp e) 0)
        ((consp e)
         (case (first e)
           ((PLUS MINUS) (list (first e)
                               (diff (second e) x)
                               (diff (third e) x)))
           ((MULT) (list 'PLUS
                         (list 'MULT (diff (second e) x) (third e))
                         (list 'MULT (second e) (diff (third e) x))))
           ((EXPP) (list 'MULT
                        (diff (second e) x)
                        (list 'MULT
                              (third e)
                              (list 'EXPP
                                    (second e)
                                    (list 'MINUS (third e) 1)))))
           (t e)))
        (t e)))

;;;---------------------------------------------------------------------

(defun PLUS (e1 e2)
  ;;;
  ;;;  (PLUS n1 n2) = n1 + n2
  ;;;  (PLUS 0 e2) = e2
  ;;;  (PLUS e1 0) = e1
  ;;;
  (cond ((and (numberp e1) (numberp e2)) (+ e1 e2))
        ((eql 0 e1) e2)
        ((eql 0 e2) e1)
        (t (list 'PLUS e1 e2))))

(defun MINUS (e1 e2)
  ;;;
  ;;;  (MINUS n1 n2) = n1 - n2
  ;;;  (MINUS e1 0) = e1
  ;;;
  (cond ((and (numberp e1) (numberp e2)) (- e1 e2))
        ((eql 0 e2) e1)
        (t (list 'MINUS e1 e2))))
```

```
(defun MULT (e1 e2)
   ;;;
   ;;;  (MULT n1 n2) = n1 * n2
   ;;;  (MULT 0 e2) = (MULT e1 0) = 0
   ;;;  (MULT 1 e2) = e2
   ;;;  (MULT e1 1) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (* e1 e2))
        ((or (eql 0 e1) (eql 0 e2)) 0)
        ((eql e1 1) e2)
        ((eql e2 1) e1)
        (t (list 'MULT e1 e2))))

(defun EXPP (e n)
   ;;;
   ;;;  (EXPP n1 n2) = n1 ^ n2
   ;;;  (EXPP e 0) = 1
   ;;;  (EXPP e 1) = e
   ;;;
  (cond ((and (numberp e) (numberp n)) (^ e n))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))

;;;---------------------------------------------------------------------

(defun parse (p) p)
(defun unparse (p) p)

(defun simplify (p) (eval p))

;;;---------------------------------------------------------------------

(defun test (p)
  (setq x 'x)
  (unparse (simplify (diff (parse p) 'x))) )

;;;---------------------------------------------------------------------
```

B Das Beispielprogramm nach der Migration

```
;;;-----------------------------------------------------------------------
;;;
;;; Symbolisches Differenzieren von Polynomen (nach der Migration)
;;;
;;;-----------------------------------------------------------------------

(defun diff (e x)
  (cond ((numberp e) 0)
        ((eq e x) 1)
        ((symbolp e) 0)
        ((consp e)
         (case (first e)
           ((PLUS MINUS) (list (first e)
                               (diff (second e) x)
                               (diff (third e) x)))
           ((MULT) (list 'PLUS
                         (list 'MULT (diff (second e) x) (third e))
                         (list 'MULT (second e) (diff (third e) x))))
           ((EXPP) (list 'MULT
                        (diff (second e) x)
                        (list 'MULT
                              (third e)
                              (list 'EXPP
                                    (second e)
                                    (list 'MINUS (third e) 1)))))
           (t e)))
        (t e)))

;;;-----------------------------------------------------------------------

(defun PLUS (e1 e2)
   ;;;
   ;;;  (PLUS n1 n2) = n1 + n2
   ;;;  (PLUS 0 e2) = e2
   ;;;  (PLUS e1 0) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (+ e1 e2))
        ((eql 0 e1) e2)
        ((eql 0 e2) e1)
        (t (list 'PLUS e1 e2))))

(defun MINUS (e1 e2)
   ;;;
   ;;;  (MINUS n1 n2) = n1 - n2
   ;;;  (MINUS e1 0) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (- e1 e2))
        ((eql 0 e2) e1)
        (t (list 'MINUS e1 e2))))
```

```
(defun MULT (e1 e2)
   ;;;
   ;;;  (MULT n1 n2) = n1 * n2
   ;;;  (MULT 0 e2) = (MULT e1 0) = 0
   ;;;  (MULT 1 e2) = e2
   ;;;  (MULT e1 1) = e1
   ;;;
  (cond ((and (numberp e1) (numberp e2)) (* e1 e2))
        ((or (eql 0 e1) (eql 0 e2)) 0)
        ((eql e1 1) e2)
        ((eql e2 1) e1)
        (t (list 'MULT e1 e2))))


(defun EXPP (e n)
   ;;;
   ;;;  (EXPP n1 n2) = n1 ^ n2
   ;;;  (EXPP e 0) = 1
   ;;;  (EXPP e 1) = e
   ;;;
  (cond ((and (numberp e) (numberp n)) (error "in function EXPP"))
        ((eql 0 n) 1)
        ((eql n 1) e)
        (t (list 'EXPP e n))))

;;;-----------------------------------------------------------------

(defun parse (p) p)
(defun unparse (p) p)

(defun simplify (p) (diff-eval p))

;;;-----------------------------------------------------------------

(defun test (p)
  (unparse (simplify (diff (parse p) 'x))) )

;;;-----------------------------------------------------------------

(defun diff-eval (e)
  (cond ((or (numberp e) (symbolp e)) e)
        ((eq (first e) 'PLUS)
          (apply #'PLUS (mapcar #'diff-eval (rest e))))
        ((eq (first e) 'MINUS)
          (apply #'MINUS (mapcar #'diff-eval (rest e))))
        ((eq (first e) 'MULT)
          (apply #'MULT (mapcar #'diff-eval (rest e))))
        ((eq (first e) 'EXPP)
          (apply #'EXPP (mapcar #'diff-eval (rest e))))
        (t (error "Unknown expression ~s." e))))

;;;-----------------------------------------------------------------
```

```
(defun main ()
  (loop
    (princ "Enter expression : ")
    (let ((e (read)))
      (if (member e '(quit exit end stop halt))
        (return)
        (format t "~&~s~%" (test e))))))

(main)
```

Verallgemeinerte Behandlung von Constraints in einem CLP-System

(Kurzfassung)

Hans-Joachim Goltz und Ulrich Geske
Gesellschaft für Mathematik und Datenverarbeitung mbH
GMD-FIRST, Rudower Chaussee 5, 12489 Berlin
{goltz,geske}@first.gmd.de

Zusammenfassung

Eine Verallgemeinerung der constraint-logischen Programmierung (GCLP), in der ein Ableitungsschritt keinen vollständigen Test auf Erfüllbarkeit erfordert und durch die die Behandlung der Constraints beeinflußt werden kann, wird beschrieben. Der Constraint-Solver der constraint-logischen Programmierung wird in GCLP durch ein Constraint-Handling-System ersetzt, das verschiedenartige Operationen ausführen kann. Eine endliche Ableitungsfolge kann nur dann erfolgreich sein, wenn die letzte Zielklausel leer ist oder nur Constraints enthält, die erfüllbar sind. Aus den Eigenschaften des Constraint-Handling-Systems ergibt sich dann, daß in einer erfolgreichen Ableitungsfolge alle vorkommenden Constraints erfüllbar sind. Es wird vorausgesetzt, daß eine Standardstrategie der Behandlung der Constraints vorgegeben ist. Zusätzlich existiert aber die Möglichkeit, mit metalogischen Methoden die Art der Behandlung der Constraints zu beeinflussen. In ersten Versuchen konnte der Vorteil der verallgemeinerten Behandlung von Constraints nachgewiesen werden.

1 Einleitung

Die constraint-logische Programmierung (CLP) ist eine elegante Verallgemeinerung der logischen Programmierung, bei der die syntaktische Unifikation durch das allgemeinere Konzept des Lösens von Constraints über einer gegebenen algebraischen Struktur (oder einer Theorie) $\mathcal{D}$ ersetzt wird. Durch Constraints kann deklarativ ausgedrückt werden, welche relationale Beziehungen gegebene Variablen zu erfüllen haben. Das Lösen von Constraints spielt in wissensbasierten Systemen und beim Problemlösen eine immer größere Rolle. Durch die constraint-logische Programmierung wird das bedeutungsvolle Konzept des Lösens von Constraints mit dem Deduktionsmechanismus der logischen Programmierung verbunden.

Der allgemeine Rahmen der constraint-logischen Programmierung $CLP(\mathcal{D})$ wurde zuerst in [7] beschrieben. Dieser Ansatz wurde inzwischen in verschiedenen Richtungen weiterentwickelt (siehe z.B. [6,10,9,3,11,1]) und es wurden verschiedenartige CLP-Systeme realisiert. Die ersten Programmiersprachen, die als constraint-logische Sprachen angesehen werden können (obwohl sie nicht in diesem Rahmen entwickelt wurden) sind PROLOG II und PROLOG III ([4,2]). Weitere Beispiele constraint-logischer Programmiersprachen sind $CLP(\mathcal{R})$ ([5,7]) und CHIP ([10]).

Der allgemeine Ansatz der constraint-logischen Programmierung geht davon aus, daß in jedem Ableitungsschritt ein Erfüllbarkeitstest erfolgt ([7,8]) und daß die syntaktische Unifikation innerhalb der Behandlung der Constraints betrachtet wird. Existierende CLP-Systeme modifizieren aber diesen allgemeinen Ansatz. So wird oft die Einbeziehung „harter“ Constraints verschoben, weil ein vollständiger Erfüllbarkeitstest sehr zeitaufwendig bzw. nicht realisierbar sein kann.

Wir beschreiben in dieser Arbeit eine Verallgemeinerung von CLP, GCLP genannt, in der ein Ableitungsschritt keinen vollständigen Test auf Erfüllbarkeit erfordert. Außerdem besteht in GCLP die Möglichkeit die Art der Behandlung der Constraints zu beeinflussen. Der Constraint-Solver von CLP wird durch ein Constraint-Handling-System in GCLP ersetzt. Es kann verschiedenartige Operationen ausführen, wie: vorläufiges Akzeptieren der Variablenbelegung, Überprüfung notwendiger Bedingungen, Umformen von Constraints, Erzeugen von Lösungen, Erfüllbarkeitstest. Die konkreten Operationen, die ein Constraint-Handling-System ausführt, ist abhängig von Parametern, die wir als Zustände bezeichnen. Ein Constraint-Solver ist ein Spezialfall eines Constraint-Handling-Systems (mit einem bestimmten festen Parameter). Es wird vorausgesetzt, daß eine Standardstrategie der Behandlung der Constraints vorgegeben ist. Zusätzlich existiert aber die Möglichkeit, die Art der Behandlung der Constraints zu beeinflussen. In ersten Versuchen konnte der Vorteil dieses Vorgehens nachgewiesen werden.

Im nächsten Abschnitt werden wir kurz das grundlegende Konzept der constraint-logischen Programmierung beschreiben und einige Modifikationen dieses allgemeinen Ansatzes diskutieren. Anschließend wird das verallgemeinerte Konzept der constraint-logischen Programmierung vorgestellt. Einfache Realisierungen dieser Verallgemeinerung werden im vierten Abschnitt diskutiert.

2 Constraint-logische Programmierung

Wir setzen voraus, daß die Relationszeichen der gegebenen Signatur in Relationszeichen der Constraints Π_c und in Relationszeichen der Atomformeln Π_p unterteilt ist ($\Pi_c \cap \Pi_p = \emptyset$), wobei das Zeichen $=$ der Gleichheitsrelation zu Π_c gehört. Weiterhin setzen wir voraus, daß eine geeignete algebraische Struktur (oder eine Theorie) $\mathcal{D}$ als Bereich der Constraints gegeben ist und daß die Erfüllbarkeit von Constraints bez. $\mathcal{D}$ überprüft werden kann.

Ein *Constraint* ist ein Ausdruck der Form $c(t_1,\ldots,t_n)$, wobei c ein n-stelliges Relationszeichen aus Π_c und $t_1,\ldots,t_n$ Terme sind. Eine *Atomformel* ist ein Ausdruck der Form $r(t_1,\ldots,t_n)$, wobei r ein n-stelliges Relationszeichen Π_p und $t_1,\ldots,t_n$ Terme sind. Mit $(\forall)F$ kennzeichnen wir den universalen Abschluß einer Formel F.

Ein *CLP($\mathcal{D}$)-Program* besteht aus einer endlichen Menge von Klauseln der Form:

$$H \leftarrow C \diamond B_1,\ldots,B_n$$

wobei H (der Kopf der Klausel), $B_1,\ldots,B_n$ Atomformeln und C ein Konjunktion von Constraints ist. Eine *Zielklausel* ist eine Klausel ohne Kopf.

Ein *Ableitungsschritt* eines constraint-logischen Programms über $\mathcal{D}$ wird wie folgt definiert: Wenn $\leftarrow C_A \diamond A_1,\ldots,A_i,\ldots,A_n$ eine Zielklausel, C_A in $\mathcal{D}$ erfüllbar und $H \leftarrow C_B \diamond B_1,\ldots,B_m$ eine Klausel aus dem gegebenen Programm ist, dann kann die Zielkausel

$$\leftarrow C_A \wedge C_B \wedge H = A_i \diamond A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n,$$

abgeleitet werden, falls $C_A \wedge C_B \wedge H = A_i$ in $\mathcal{D}$ erfüllbar ist. Um festzustellen, ob gegebene Constraints erfüllbar sind, existieren in Abhängigkeit von $\mathcal{D}$ spezielle algorithmische Verfahren, die auch als *Constraint-Solver* bezeichnet werden. Ein Constraint-Solver kann unabhängig von der logischen Komponente programmiertechnisch realisiert sein.

Wie bereits in der Einleitung erwähnt wurde, wird der allgemeine Ansatz der constraint-logischen Programmierung in den CLP-Systemen modifiziert. Im allgemeinen werden in einem Ableitungsschritt neben Überprüfungen für die Erfüllbarkeit auch Umformungen (Vereinfachungen) in bestimmte Grundformen (kanonische Formen oder Normalformen) durchgeführt. Dadurch soll vermieden werden, daß bei jeder Überprüfung immer wieder die gleichen Umformungen durchgeführt werden müssen.

Vollständige Erfüllbarkeitstests können teilweise nicht ausgeführt werden (z.B., wenn nichtlineare Gleichungen und Ungleichungen vorkommen) oder sie sind sehr zeitaufwendig. Deswegen erfolgt in einigen CLP-Systemen die Überprüfung der Erfüllbarkeit in einem Ableitungsschritt nur unvollständig. So wird z.B. in $CLP(\mathcal{R})$ (siehe [5]) innerhalb eines Inferenzschrittes nur die Lösbarkeit der linearen Gleichungen überprüft. Die nichtlinearen Gleichungen werden nur gesammelt. Erst wenn durch weitere Variablenbindungen eine gesammelte Gleichung linear wird, wird diese Gleichung in den Lösbarkeitstest einbezogen. In [11] wird eine constraint-logische Programmiersprache für endliche Constraintbereiche vorgestellt, bei der die Constraints in „basic“ und „nonbasic“ Constraints eingeteilt werden und ein vollständiger Erfüllbarkeitstest nur für die „basic“ Constraints durchgeführt wird.

Weiterhin existieren CLP-Systeme, die in einem Ableitungsschritt zuerst die syntaktische Unifikation (ausgewähltes Literal mit dem Kopf einer Klausel) ausführen. Erst wenn diese erfolgreich ist, werden die vorhandenen Constraints betrachtet. Etwas allgemeiner bedeutet dieser Ansatz, daß für die Auswahl einer anwendbaren Klausel zuerst die Erfüllbarkeit einer Teilmenge der entsprechenden Constraints überprüft wird.

3 Verallgemeinerte Constraint-logische Programmierung

In diesem Abschnitt wird eine Verallgemeinerung von CLP, die mit GCLP bezeichnet wird, beschrieben. Der Constraint-Solver von CLP wird durch ein Constraint-Handling-System in GCLP ersetzt.

Ein *Constraint-Handling-System* hinsichtlich eines gegebenen Constraintbereiches $\mathcal{D}$ ist eine Funktion $chs : \mathcal{S} \times \mathcal{C} \mapsto \mathcal{C} \cup \{false\}$, wobei $\mathcal{C}$ eine Menge endlicher Konjunktionen von Constraints ist, *"false"* der übliche Boolsche Wert, $\mathcal{S}$ ist eine endliche Menge (*Menge von Zuständen*) ist, und die folgenden Bedingungen für jedes $s \in \mathcal{S}$ und jedes $C_1 \in \mathcal{C}$ erfüllt sind:

- Wenn $chs(s, C_1) = false$, dann ist C_1 unerfüllbar in $\mathcal{D}$.
- Wenn $chs(s, C_1) = C_2$, dann $\mathcal{D} \models (\forall)(C_1 \leftrightarrow C_2)$.

Wir setzen dabei voraus, daß das Constraint-Handling-System Kombinationen von verschiedenartigen Grundoperationen ausführen kann. Der jeweils gewählte Parameter, den wir als Zustand bezeichnen, bestimmt, welche Kombination ausgeführt wird. Grundoperationen können sein:

(1) vorläufiges Akzeptieren der Variablenbelegung (d.h., die Constraints werden nur gesammelt),

(2) Überprüfung von notwendigen Bedingungen,

(3) Umformen oder Vereinfachen von Constraints,

(4) Überprüfung der Erfüllbarkeit für eine bestimmte Teilmenge von Constraints,

(5) vollständiger Erfüllbarkeitstest,

(6) Erzeugen von Lösungen.

In Abhängigkeit, auf welche Teilmengen der Constraints sich die Operationen beziehen, können die angegebenen Arten von Grundoperationen weiter unterteilt werden. Außerdem könnten in (2) verschiedene notwendige Bedingungen betrachtet werden. Welche Grundoperationen und Kombinationen von Grundoperationen möglich bzw. sinnvoll sind, ist abhängig von dem gegebenen Constraintbereich $\mathcal{D}$. Ein Constraint-Solver ist ein Spezialfall eines Constraint-Handling-Systems (mit einem bestimmten festen Parameter).

Ein *Ableitungsschritt* eines verallgemeinerten constraint-logischen Programms über $\mathcal{D}$ wird wie folgt definiert: Wenn $\leftarrow C_A \diamond A_1, \ldots, A_i, \ldots, A_n$ eine Zielklausel und die Klausel $H \leftarrow C_B \diamond B_1, \ldots, B_m$ aus dem gegebenen Programm ist, dann kann in Abhängigkeit vom aktuellen Zustandes s des Constraint-Handling-Systems die neue Zielklausel

$$\leftarrow C_{chs} \diamond A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n,$$

abgeleitet werden, falls $chs(s, C) = C_{chs}$ und $C_{chs} \neq false$, wobei C die Konjunktion $C_A \wedge C_B \wedge H = A_i$ ist.

Eine *Ableitungsfolge* ist eine Folge von Zielklauseln, die durch die angegebenen Ableitungsschritte verbunden sind. Eine Ableitungsfolge ist *erfolgreich*, wenn sie endlich ist, die letzte Zielklausel nur Constraints enthält und diese Constraints in $\mathcal{D}$ erfüllbar sind. Aus den Eigenschaften des Constraint-Handling-Systems ergibt sich: Wenn eine Ableitungsfolge erfolgreich ist, dann sind alle Konjunktionen von Constraints, die in den Zielklauseln dieser Folge vorkommen, erfüllbar in der Struktur $\mathcal{D}$.

Die Behandlung der Constraints in einem Ableitungsschritt ist abhängig vom jeweiligem Zustand des Constraint-Handling-Systems. Welche Operationen sollten aber in einem Ableitungschritt zur Constraint-Behandlung ausgeführt werden ? Dies führt zu einem generellen Problem: In welchem Verhältnis sollte der Aufwand für die Klauselauswahl in einem Ableitungsschritt zum Nachteil stehen, der sich aus einer falschen Klauselauswahl ergeben kann ? Es ist nicht zu erwarten, daß eine allgemein beste Strategie existiert. Deswegen gehen wir davon aus, daß eine Standardstrategie gegeben ist, die aber mit metalogischen Methoden beeinflußbar ist (d.h. der Parameter der Art der Behandlung der Constraints kann verändert werden), um spezielle Eigenschaften eines zu realisierenden Deduktionssystems berücksichtigen zu können. Für die gewählte Standardstrategie sollte der Aufwand zur Klauselauswahl in einem Ableitungsschritt nur so groß sein, daß durch die Operationen des Constraint-Handling-Systems der Suchraum hinreichend begrenzt wird. Eine solche Begrenzung des Suchraumes kann auch dadurch erreicht werden, daß nach einer vorgegebenen Anzahl von Ableitungsschritten eine umfangreiche Überprüfung oder ein Erfüllbarkeitstest erfolgt und in den Zwischenschritten nur einfache (schnelle) Operationen der Constraint-Behandlung durchgeführt werden.

Prinzipiell gibt es mehrere metalogische Methoden, um den Parameter für ein Constraint-Handling-System festzulegen bzw. zu verändern: am Beginn eines Programms spezifizieren, über Metaregeln definieren (Metainterpreter), explizit in den Klauseln angeben.

4 Versuchsweise Realisierung

Zum Nachweis, daß eine verallgemeinerte Behandlung von Constraints einen positiven Einfluß auf die Abarbeitungszeit constraint-logischer Programme hat, wurde prototypisch in Prolog ein einfacher Constraint-Solver cs/2 programmiert.

```
cs(X,Y) :- cs1(X,Y0),!,((\+ X==Y0) -> cs(Y0,Y) ; Y=Y0).
cs1(A,A) :- var(A), !.
cs1((A,B),Constr) :-(var(A),!,cs1(B,Constr));(var(B),!,cs1(A,Constr)).
cs1(((A,R),B),C) :- !,cs1((A,R),C1),cs1(B,C2),
   (var(C1)->C=C2;var(C2)->C=C1;C=(C1,C2)).
cs1((A,B), Constr) :-
   !,((A=(X > Y) ; A=(X < Y) ; A=(X=<Y) ; A=(X>=Y)) ->
      (cs1(B, C2),cs1(A, C1),
       ((var(C1),var(C2))->true;(nonvar(C2),var(C1)) -> cs1(C2,Constr)
         ;var(C2)->Constr=C1;Constr = (C1,C2) ))
      ;(cs1(A, C1),cs1(B, C2),
        ((var(C1),var(C2))->true;(nonvar(C1),var(C2)) -> cs1(C1,Constr)
      ;var(C1)->Constr = C2;Constr=(C1,C2) ))  ),
   !.
cs1(A, Constr) :- constraint(A, Constr).
```

Die erste Klausel von cs/2 ruft das „Arbeitspferd" cs1/2 auf, in dessen Verlauf eventl. Variablen gebunden und einzelne Constraints berechnet werden. Stimmen nach dem Abarbeiten von cs1/2 dessen Inputvariable X und Outputvariable Y0 überein, ist keine weitere Vereinfachung des Constraintsystems mehr möglich. Anderenfalls wird cs/2 erneut aufgerufen.

Die ersten vier Klauseln von cs1/2 dienen der Analyse des Constraintsystems, um die Reihenfolge der Berechnung der einzelnen Constraints zu bestimmen. Die Berechnung selbst wird von der fünften Klausel von cs1/2 veranlaßt.

Konflikte, die in nicht berechenbaren Constraintfolgen enthalten sind (z.B.: (X<3,X>3)), sind durch diesen einfachen Solver nicht erkennbar. Er ist deshalb für den abschließenden Erfüllbarkeitstest bei Anwendungen, die zu derartigen Konfliktmengen führen können, nicht ausreichend. Die hier verwendeten Testbeispiele vermeiden diese Konflikte.

Die Strategie des Constraint-Handling-Systems wurde mit

- Akzeptieren der Variablenbelegung durch Unifikation
- verzögerte Aktivierung des Constraint-Solvers

definiert. Die Verwendung des Constraint-Solvers erfolgt durch Aufruf im Metainterpreter, z.B. für das Verzögern bis nach der Abarbeitung aller Prozeduraufrufe im Klauselkörper:

```
metaint(Call,Constraint) :-
   metaint(Call,_ , Constr),
   cs(Constr,Constraint).
metaint((P , Q), ConstrColl,Constr) :-
   !,metaint(P, ConstrColl,ConstrP),
   metaint(Q, ConstrP, Constr).
metaint((P ; Q), ConstrColl,_/*Constr*/) :-
   !, (metaint(P,ConstrColl,_/*ConstrP*/)
      ; metaint(Q, ConstrColl, _/*ConstrQ*/).
metaint(P, C/*ConstrColl*/, Constr) :-
   constraint(P, Constrc),
   (var(C) ->  Constr=Constrc ; Constr=(C,Constrc)),
   !.
metaint(P,X,X/*Constr*/) :-
   sys(P),
   !,call(P).
metaint(P,ConstrColl, Constr) :-
   !,clause(P, Q),
   metaint(Q, ConstrColl, ConstrQ),
   ((Q=(_,_);Q=true) -> Constr=ConstrQ ; cs(ConstrQ, Constr)).
```

Durch die Prozedur `constraint/2`, die im Metainterpreter vor der Übergabe eines Prozeduraufrufs an `call/1` bzw. `clause/2` aufgerufen wird, werden einige vordefinierte (arithmetische) Constraints und beliebige problemspezifisch definierbare Constraints erkannt, wie z.B.:

```
/* Erkennung eines Aufrufs als Constraint */
constraint(Call, Constr) :-
  constraints(Call, Compute,Vars)
  (solvable(Compute,Vars)->call(Compute); Constr=Call).

 /* Definition von Praedikaten, die als Constraints behandelt werden:
   constraints/3
   - 1.Arg.: Aufruf im Quellprogramm,
   - 2.Arg.: auszufuehrender Aufruf,
   - 3.Arg.: max. Variablenanzahl im 2.Arg. fuer loesbaren Aufruf */
constraints('<'(X,Y), '<'(X,Y),0)
constraints('>'(X,Y), '>'(X,Y),0).
constraints('=<'(X,Y), '=<'(X,Y),0).
constraints('>='(X,Y), '>='(X,Y),0).
constraints(is(X,Y), ist(X,Y),1).
constraints('=='(X,Y), '=='(X,Y),1).
%constraints(is_sorted(X,Y),is_sorted(X,Y),1). /*problemspezifisch*/

ist(A, X+B):- nonvar(A), nonvar(B), !,X is A-B.
ist(A, B+X):- nonvar(A), nonvar(B), !,X is A-B.
ist(A, X-B):- nonvar(A), nonvar(B), !,X is A+B.
ist(A, B-X):- nonvar(A), nonvar(B), !,X is A+B.
ist(A, X*B):- nonvar(A),nonvar(B),  !,X is A//B.
```

```
ist(A,B*X):- nonvar(A), nonvar(B),  !,X is A//B.
ist(X,Y):- extractvars(Y,L),
         (L=[] -> X is Y).
solvable(Call,Vars):- extractvars(Call,VarList),length(VarList,V),Vars>=V.
```

Für die Anwendung des Programms kann die Prozedur zur Berechnung der Fakultät dienen:

```
fact(0, 1).
fact(N, F) :-
    N>0,
    ist(N1, N -1),
    fact(N1, M),
    ist(F, N*M).
```

Wegen des Aufrufs `N>0` in der zweiten Klausel ist ein Aufruf `?- fact(X,6)` nicht möglich, aber `?- metaint(fact(X,6),C)` liefert `X=3` (und `C= _Var` als Kennzeichen, daß alle Constraints berechnet wurden).

Der Vergleich unterschiedlicher Verzögerungen (keine bis maximale) zeigt, daß sofortiges Lösen der Constraints (auch lösbarer) nicht die beste Strategie ist. Problemabhängig können Verbesserungen des Zeitverhaltens von mehr als einer Größenordnung durch die Verzögerung der Aktivierung des Constraint-Solvers erreicht werden.

5 Schlußbemerkungen

Wir haben eine Verallgemeinerung der constraint-logischen Programmierung vorgestellt, in der der Constraint-Solver durch ein Constraint-Handling-System ersetzt wird, so daß die Überprüfung der Erfüllbarkeit verzögert und die Behandlung der Constraints innerhalb eines Ableitungsschrittes beeinflußt werden kann. Dadurch können Besonderheiten eines zu realisierenden Deduktionssystems berücksichtigt und Abweichungen von der Standardstrategie der Inferenz realisiert werden. Ein Schwerpunkt der weiteren Forschung soll die Entwicklung von allgemeinen Richtlinien zur Festlegung einer Standardstrategie und zu sinnvollen Möglichkeiten der Beeinflussung der Standardstrategie sein. Dazu wird prototypisch ein verallgemeinertes CLP-System mit sehr flexiblen Kontrollmöglichkeiten implementiert und Untersuchungen zu verschiedenen Strategien sowohl theoretisch als auch experimentell durchgeführt.

Literatur

[1] H.-J. Bürckert. *A Resolution Principle for a Logic with Restricted Quantifiers*, volume 568 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 1991.

[2] A. Colmerauer. Opening the PROLOG III universe. Byte Magazine, Special Issue on Logic Programming, August 1987.

[3] M. Gabbrielli and G. Levi. Modeling answer constraints in constraint logic programs. In Koichi Furukawa, editor, *Proc. 8th Int. Conf. Logic Programming*, pages 238–252. MIT Press, Cambridge (Mass.), London, 1991.

[4] F. Giannesini, H. Kanoi, R. Pasero, and M. van Caedhem. *PROLOG*. Addison-Wesley, Reading (Mass.), 1986.

[5] N. Heintze, J. Jaffar, S. Michaylov, P. Stucky, and R. Yap. *The CLP($\mathcal{R}$) Programmer's Manual, Version 1.1*, November 1991.

[6] M. Höhfeld and G. Smolka. Definite relations over constraint language. LILOG-Report 53, IBM-Deutschland,, Germany, 1988.

[7] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th Principles of Programming Languages*, pages 111–119, Munich, 1987.

[8] J. Jaffar and J.-L. Lassez. From unification to constraints. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '87*, Lecture Notes in Computer Science 315, pages 1–18. Springer-Verlag, Berlin, Heidelberg, 1987.

[9] P. Lim and P.J. Stuckey. Meta programming as constraint programming. In Saumya Debray and Manuel Hermenegildo, editors, *Proc. North American Conf. Logic Programming*, pages 417–430. MIT Press, Cambridge (Mass.), London, 1990.

[10] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge (Mass.), London, 1989.

[11] P. van Hentenryck and Y. Deville. Operational semantics of constraint logic programming over finite domains. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 395–406, Berlin, Heidelberg, 1991. Springer-Verlag.

Consequence Finding and Logic Programming

Knut Hinkelmann
DFKI, Postfach 2080, 67608 Kaiserslautern
hinkelma@dfki.uni-kl.de

Short Paper

Abstract

Using logic programs as a knowledge representation formalism, besides proof finding a number of other inferences may be necessary. Given a theory T, the consequence-finding problem is to find a statement B such that B follows from T. A useful specialization is to compute exactly the newly derivable consequences caused by new information added to the theory. This version of consequence finding is adapted here for logic programs. Applications in production planning and checking of integrity constraints are presented. Although the main direction of consequence finding is forward chaining, a straightforward solution would be to integrate it with a backward chaining component to prove premises of triggered rules. Alternatively, a new rewriting approach is presented, which extends the Magic-Templates rewriting. Magic-Templates rewriting has originally been invented for query answering in a bottom-up reasoning (forward chaining) system. To determine, for which queries the rewriting has to be done, the *down* propagation of Magic Templates is extended by an *up* propagation phase. The presented rewriting approach can also be regarded as a specialization of an upside-down meta-interpreter.

1 Introduction

The consequence-finding problem [Lee, 1967] may be formulated as:

> Given a set of statements $A_1, \ldots, A_n$, find a statement B such that B follows from $A_1, \ldots, A_n$ [Slagle *et al.*, 1969].

There are a number of specializations of this general formulation. They are characterized by restrictions of the statement B, which we are interested in. In [Lee, 1967] the consequence-finding problem is expressed in a more restricted form:

> Given a set of formulas T and a resolution procedure P, for any logical consequence D of T, can P derive a logical consequence C of T such that C subsumes D?

In [Slagle *et al.*, 1969] consequence finding is examined for prime (non-trivial) consequences. Also proof finding can be regarded as a special case of consequence finding, if B is the empty clause.

Inoue presents an extended ordered linear resolution strategy, which is complete for consequence finding in the sense that only clauses having a certain property (called characteristic clauses) should be found [Inoue, 1991]. A useful specialization is to compute exactly the newly derivable consequences caused by new information added to the theory. These new consequences are called new characteristic clauses.

Here we will adapt this latter version of the consequence-finding problem for logic programs. In logic programming, deductive databases, and rule-based reasoning we are not interested in general consequences but only in ground unit clauses (i.e. ground facts). Thus, our consequence-finding problem can be informally described as:

> Given a set of Horn clauses T and a set of ground unit clauses $F \subseteq T$ (F is called the set of initial facts), derive all the consequences B of T, such that B is a ground unit clause and at least one clause $C \in F$ has been used to derive B.

In Section 2 we will motivate, why this type of consequence finding is useful. We will see that the consequence-finding problem can be divided into a forward chaining and a backward chaining subtask. A straightforward solution for this problem would integrate a forward and a backward chaining system. In Section 3 we will present an alternative solution. This approach is based on a rewriting of the original logic program for evaluation in a forward chaining system. It extends the Magic-Templates rewriting to solve the consequence-finding problem.

2 Why are we interested in Consequence Finding?

Proof finding has been studied as the main inference in logic programming and deductive databases, while other inferences have been neglected. But by applying logic programs as a knowledge representation formalism in expert systems, besides proof finding a number of other inferences may be necessary.

Example 2.1 [Production Planning]

As described in [Klauck *et al.*, 1993] production planning can be regarded as an instance of the heuristic classification inference scheme. The objective of production planning is to derive a working plan describing how a given workpiece can be manufactured. The input to a production planning system is a very 'elementary' description of a workpiece as it comes from a CAD system. Geometrical descriptions of the workpiece's surfaces, topological neighbourhood relations, and technological data are the central parts of this product model representation. The first step in production planning is the generation of an abstract feature description from the elementary workpiece data. A feature thus associates a workpiece model with corresponding manufacturing methods [Klauck *et al.*, 1991].

In [Hanschke and Hinkelmann, 1992] we have suggested a hybrid declarative formalism for the abstraction phase of heuristic classification. A logical rule component is responsible for the feature aggregation. Since it is not known in advance, which kind of feature are present in the workpiece, a proof-finding approach would have to find possible instantiations for every known feature class. Instead, we used a consequence finding approach: the theory is the set of rules defining workpiece features and the initial facts are the facts describing the workpiece [Boley *et al.*, 1993].

Example 2.2 [Integrity Constraints]

Consequence finding can be used to detect whether database updates would lead to inconsistencies. Consider a logic program with integrity constraints denoting negative or disjunctive knowledge. These integrity constraints are represented as denials, i.e. clauses with empty head. Eshghi and Kowalski use these kind of integrity constraints for their abduction procedure [Eshghi and Kowalski, 1989]. We can also represent them as clauses with the special atom **inconsistent** as conclusion [Manthey and Bry, 1987]. Thus, the following rule demands that two connected solids must coincide at their contact point.

```
inconsistent <- connected(S1,S2), position(S1,[A1,E1]),
                position(S2,[A2,E2]), E1 =\= A2.
```

A real database will have many of these integrity constraints. Let's assume, that the facts `position(cylinder1,[1,5])` and `position(truncated_cone2,[6,9])` are in the database. Now we want to connect these elements. Using a proof-finding approach one has to assert the fact `connected(cylinder1,truncated_cone2)` and then ask the query `?- inconsistent`. This procedure would invoke all integrity constraints in backward direction even if they are independent from the new fact. Instead, it would be more efficient to derive only those facts, that are consequences of this new assertion. In [Eshghi and Kowalski, 1989] it is argued to do constraint checking by forward reasoning. Forward reasoning alone, however, is not sufficient. Consider the following program:

```
endpoint(X)   <- cone(X),radius2(X,0).
startpoint(X) <- cone(X),radius1(X,0).
cone(c1).
radius1(c1,0).
radius2(c1,20).
cylinder(cyl1).
```

and the following integrity constraints, which say that you cannot connect two elements, if there is no surface but only a point at the end of one element (see Fig. 1):

```
inconsistent <-  connected(I1,I2), endpoint(I1).
inconsistent <-  connected(I1,I2), startpoint(I1).
```

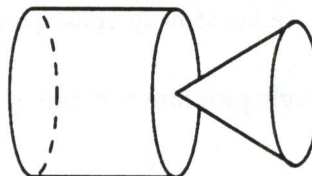


Figure 1: Forbidden Connection

Adding the new fact `connected(cyl1,c1)` would lead to an inconsistency, which will not be detected by forward chaining alone. Additionally we need to prove, whether the premise `startpoint(c1)` can be satisfied. In [Manthey and Bry, 1987] a model-generation approach has been applied for this problem. Alternatively, we can also combine consequence finding with some kind of proof-finding component.

3 How to implement Consequence Finding

From example 2.2 it became clear, that forward chaining and backward chaining may interchange to solve the consequence-finding problem. An obvious approach would be to use an integrated system with forward and backward chaining components.

The forward chaining process starts with the set F of initial facts. A rule

$$C \leftarrow P_1, \ldots, P_n$$

is triggered, if at least one $P_i, i \in \{1, \ldots, n\}$ is unifiable with a fact $f \in F$. Then, the remaining premises have to be proved. Besides changing the implementation, this can be achieved by a program transformation, declaring the premises as backward-provable and evaluating them by the backward chaining component. Since each of the premises can serve as a trigger premise, this approach is equivalent to the following transformation of the above rule:

$$\begin{array}{lcl} C & \leftarrow & P_1, prove(P_2), \ldots, prove(P_n) \\ C & \leftarrow & prove(P_1), P_2, \ldots, prove(P_n) \\ C & \leftarrow & prove(P_1), prove(P_2), \ldots, P_n \end{array}$$

The rules are evaluated by forward chaining. The predicate *prove* is a built-in operator activating the backward chaining component to prove its argument.

In the following we will present a rewriting approach for consequence finding, which avoids a call to the backward chaining system. A bottom-up, hyper-resolution strategy is a forward chaining fixpoint procedure deriving the minimal model of a logic program. Rewriting strategies like Magic Sets [Bancilhon *et al.*, 1986; Beeri and Ramakrishnan, 1991] or more recently Magic Templates [Ramakrishnan, 1988] specialize a logic program to prove a particular goal, if evaluated by a fixpoint procedure. Information about the query is passed to the rules at compile time by introducing additional predicates.

Instead of calling a backward chaining system to prove the remaining premises of triggered rules, we can apply this Magic-Templates rewriting approach. The rewritten program is evaluated by a bottom-up procedure, in this case the semi-naive evaluation strategy for logic programs [Bancilhon and Ramakrishnan, 1986]. To determine, for which queries the rewriting has to be done, the down propagation of the Magic-Templates rewriting is extended by an up propagation phase. In the following we will demonstrate bottom-up consequence finding with an example.

Example 3.1 Consider a logic program containing the following rules. The predicates b_1, b_2, b_3, b_4, and b_5 are base predicates:

$$\begin{array}{lcl} r_1 : p(X,Y) & \leftarrow & p_1(X,Z), b(Z,V), p_3(X,V,Y) \\ r_2 : q(U,V,W) & \leftarrow & q_1(U,Z,W), p(Z,V) \\ r_3 : p_1(X,Z) & \leftarrow & b_1(Z,Y), b_2(Y,X) \\ r_4 : p_3(X,V,Y) & \leftarrow & b_3(X,V,Z), b_4(Z,Y) \\ r_5 : q_1(U,Z,W) & \leftarrow & b_1(X,Z), b_5(U,X,W) \end{array}$$

Let's assume we want to derive the consequences of the fact $b(a,b)$. That is, the seed – specifying the bound arguments of the inital facts – for consequence finding is

$$upmagic_b(a,b)$$

To derive the consequences of a set of initial facts specified by a pattern, e.g. $b(a,X)$ the seed is given by a rule, e.g.

$$upmagic_b(a,X) \leftarrow b(a,X)$$

To derive the consequences of a given fact, a first step is to select all the rules, which can be triggered by this fact. In our example it is rule r_1. If we assume that the sideway information passing strategy is determined by a left-to-right evaluation of the remaining premises, we see that we have to prove p_1^{fb}, p_3^{bbf}. The argument values of the trigger fact, i.e. $upmagic_b(Z,V)$, are propagated up as initial values (seeds) for these queries:

$$\begin{array}{lll} magic_p_1^{fb}(Z) & \leftarrow & upmagic_b(Z,V) \\ magic_p_3^{bbf}(X,V) & \leftarrow & upmagic_b(Z,V), p_1^{fb}(X,Z) \end{array}$$

For the adorned predicates p_1^{fb}, p_3^{bbf} we apply the usual Magic-Templates rewriting, propagating the initial values down to the rules defining p_1 and p_3:

$$\begin{array}{lll} p_1^{fb}(X,Z) & \leftarrow & magic_p_1^{fb}(Z), b_1(Z,Y), b_2(Y,X) \\ p_3^{bbf}(X,V,Y) & \leftarrow & magic_p_3^{bbf}(X,V), b_3(X,V,Z), b_4(Z,Y) \end{array}$$

Thus, the rewritten rule r_1' is

$$p(X,Y) \leftarrow b(Z,V), p_1^{fb}(X,Z), p_3^{bbf}(X,V,Y)$$

The derived facts of rule r_1 can themselves trigger further rules, in our example rule r_2. Analogously as before, the values for $p(X,Y)$ are propagated up to rule r_2, resulting in the following rules:

$$\begin{array}{lll} q(U,V,W) & \leftarrow & p(Z,V), q_1^{fbf}(U,Z,W) \\ magic_q_1^{fbf}(Z) & \leftarrow & p(Z,V) \\ q_1^{fbf}(U,Z,W) & \leftarrow & magic_q_1^{fbf}(Z), b_1(X,Z), b_5(U,X,W) \end{array}$$

This kind of transformation has to be performed for each of the initial facts. Then, the evaluation of the rewritten program by a bottom-up reasoning system, e.g. the semi-naive strategy, solves the consequence-finding problem.

4 Conclusion

A particular consequence-finding problem for logic programs has been presented. The goal is to derive all consequences of a set of initial facts. The problem has been solved by bottom-up evaluation of the logic program after a new extension of the Magic-Templates rewriting. This rewriting approach avoids the necessity of implementing a specialized inference engine. A further extension can be to combine consequence finding and proof finding: only those consequences of the initial facts are derived, that are instances of a given query.

In [Hinkelmann, 1992] an alternative transformation of a logic program is presented, performing consequence finding in a top-down reasoning system. This transformation is equivalent to the partial evaluation of an upside-down meta-interpreter. The presented rewriting approach of Section 3 can also be regarded as a specialization of a meta-interpreter. This meta-interpreter is an extension of the one presented in [Bry, 1990], which is used to formalize the Magic-Set approach by upside-down meta-interpretation.

References

[Bancilhon and Ramakrishnan, 1986] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD Conference*, pages 16–52. ACM, 1986.

[Bancilhon *et al.*, 1986] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15. ACM, 1986.

[Beeri and Ramakrishnan, 1991] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–299, October 1991.

[Boley *et al.*, 1993] Harold Boley, Philipp Hanschke, Knut Hinkelmann, and Manfred Meyer. COLAB: A hybrid knowledge compilation laboratory. Research Report RR-93-08, DFKI, Kaiserslautern, Germany, January 1993. Also to appear in *Annals of Operations Research*.

[Bry, 1990] Francois Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[Eshghi and Kowalski, 1989] Kave Eshghi and Robert Kowalski. Abduction compared with negation by failure. In *6th International Conference on Logic Programming (ICLP'89)*, 1989.

[Hanschke and Hinkelmann, 1992] Philipp Hanschke and Knut Hinkelmann. Combining terminological and rule-based reasoning for abstraction processes. In *Proceedings of the 16th German Workshop on Artificial Intelligence (GWAI-92)*, number 671 in Lecture Notes on Artificial Intelligence. Springer-Verlag, September 1992. also available as DFKI Research Report RR-92-40.

[Hinkelmann, 1992] Knut Hinkelmann. Forward Logic Evaluation: Compiling a Partially Evaluated Meta-interpreter into the WAM. In *Proceedings of the 16th German Workshop on Artificial Intelligence (GWAI-92)*, number 671 in Lecture Notes on Artificial Intelligence. Springer-Verlag, September 1992.

[Inoue, 1991] Katsumi Inoue. Consequence-finding based on ordered linear resolution. In *Proc. of the 12 th IJCAI*, Sidney, Australia, 1991.

[Klauck *et al.*, 1991] Christoph Klauck, Angsar Bernardi, and Ralf Legleitner. FEAT-REP: Representing features in CAD/CAM. In *4th International Symposium on Artificial Intelligence: Applications in Informatics*, Cancun, Mexiko, 1991. An extended Version is also available as Research Report RR-91-20, DFKI GmbH.

[Klauck *et al.*, 1993] Christoph Klauck, Angsar Bernardi, and Ralf Legleitner. Heuristic classification for automated CAPP. In *Proceddings of the Eleventh Conference on Applications of Artificial Intelligence (AAI-XI) - Knowledge-Based Systems in Aerospace and Industry, SPIE*, 1993. An extended Version is also available as Research Report RR-92-49, DFKI GmbH.

[Lee, 1967] R. C. T. Lee. *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms.* PhD thesis, University of California, Berkeley, 1967.

[Manthey and Bry, 1987] Rainer Manthey and Francois Bry. SATCHMO: a theorem prover implemented in prolog. In *Conference on Automated Deduction, CADE*, 1987.

[Ramakrishnan, 1988] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programms. In R.A. Kowalski and K.B. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, 1988.

[Slagle *et al.*, 1969] J. R. Slagle, C. L. Chang, and R. C. T. Lee. Completeness theorems for semantic resolution in consequence-finding. In *IJCAI-69*, pages 281–285, 1969.

TDL–A Type Description Language for Unification-Based Grammars*

Hans-Ulrich Krieger, Ulrich Schäfer
{krieger, schaefer}@dfki.uni-sb.de

German Research Center for Artificial Intelligence (DFKI)
Universität des Saarlandes
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany

Abstract

Unification-based grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics. Their success stems from the fact that they can be seen as high-level declarative programming languages for linguists which allow them to express linguistic knowledge in a monotonic fashion. Moreover, such formalisms can be given a precise, set-theoretical semantics.
This paper presents *TDL*, a typed feature-based language which is specifically designed to support highly lexicalized grammar theories like HPSG, FUG, or CUG. *TDL* offers the possibility to define (possibly recursive) types, consisting of type constraints and feature constraints over the standard connectives $\wedge$, $\vee$, and $\neg$, where the types are arranged in a subsumption hierarchy. *TDL* distinguishes between *avm types* (open-world reasoning) and *sort types* (closed-world reasoning) and allows the declaration of partitions and incompatible types. Working with partially as well as with fully expanded types is possible, both at definition and at run time. *TDL* is incremental, i.e., it allows the redefinition of types and the use of undefined types. Efficient reasoning is accomplished through specialized modules.

Topic Areas: Type and Feature Constraints, Disjunction and Negation, Type Hierarchies.

*We are grateful to the other *FoPra Brothers*, Stephan Diehl and Karsten Konrad, for their support and numerous help. This work has been supported by a research grant (ITW 9002 0) from the German Bundesministerium für Forschung und Technologie to the DISCO project of the DFKI.

1 Introduction

Over the last few years, unification-based (or more general: constraint-based) grammar formalisms have become the predominant paradigm in natural language processing and computational linguistics.[1] Their success stems from the fact that they can be seen as a monotonic, high-level representation language for linguistic knowledge, where a dedicated parser/generator or a uniform type deduction mechanism acts as the inference engine. The main idea of representing as much linguistic knowledge as possible through a unique data type called *feature structures*, allows the integration of different description levels, starting with phonology and ending in pragmatics, therefore a feature structure directly serves as an *interface* between the different description stages, which can be accessed by a parser or a generator at the same time. In this context, *unification* is concerned with two different tasks: (i) *combining information* (unification is a structure-building operation), and (ii) *rejecting inconsistent knowledge* (unification determines the satisfiability).

While the first approaches relied on annotated phrase structure rules (for instance GPSG [Gazdar et al. 85] and PATR-II [Shieber et al. 83], as well as their successors CLE [Alshawi 92] and ELU [Russell et al. 92]), modern formalisms try to specify grammatical knowledge as well as lexicon entries merely through feature structures. In order to achieve this goal, one must enrich the expressive power of the first unification-based formalisms with different forms of *disjunctive descriptions* (atomic disjunctions, general disjunctions, distributed disjunctions etc.). Later, other operations came into play, viz., *(classical) negation* or *implication*. Full negation however can be seen as an input macro facility because it can be expressed through the use of disjunctions, negated coreferences, and negated atoms with the help of existential quantification as shown in [Smolka 88]. Other proposals considered the integration of *functional* and *relational dependencies* into the formalism which makes them Turing-complete in general.[2] However the most important extension to formalisms consists of the incorporation of *types*, for instance in modern systems like TFS [Zajac 92], CUF [Dörre & Eisele 91], or $\mathcal{TDL}$ [Krieger & Schäfer 93a]. Types are ordered *hierarchically* (via *subsumption*) as it is known from object-oriented programming languages. This leads to *multiple inheritance* in the description of linguistic entities (see [Daelemans et al. 92] for a comprehensive introduction). Finally, *recursive types* are necessary to describe at least phrase-structure recursion which is inherent in all grammar formalisms which are not provided with a *context-free backbone*.

Martin Kay was the first person who laid out a generalized linguistic framework, called *unification-based grammars*, by introducing the notions of *extension*, *unification*, and *generalization* into computational linguistics. Kays *Functional Grammar* [Kay 79] represents the first formalism in the unification paradigm and is the predecessor of strictly lexicalized approaches like FUG [Kay 85], HPSG [Pollard & Sag 87; Pollard & Sag 93] or UCG [Moens et al. 89]. Pereira and Shieber were the first to give a mathematical reconstruction of PATR-II in terms of a denotational semantics [Pereira & Shieber 84]. The work of Karttunen led to major extensions of PATR-II, concerning disjunction, atomic negation, and the use of cyclic structures [Karttunen 84]. Kasper and Rounds' seminal work is important in many respects: they clarified the connection between feature structures and finite automata, gave a logical characterization of the notion of disjunction, and presented for the first time complexity results (see [Kasper & Rounds 90] for a summary). Mark Johnson then enriched the descriptive apparatus with classical negation and showed that the feature calculus is a decidable subset of first-order predicate logic [Johnson 88]. Finally, Gert Smolka's work gave a fresh impetus to the whole field: his approach is distinguished from others in that he presents a sorted set-theoretical semantics for feature structures [Smolka 88; Smolka 89]. Moreover, Smolka gave solutions to problems concerning the complexity and decidability of feature structure descriptions. Paul King's work aims to reconstruct a special grammar theory, viz. HPSG, in mathematical terms [King 89], whereas the Backofen and Smolka's treatment is the

[1] [Shieber 86] and [Uszkoreit 88] give an excellent introduction to the field of unification-based grammar theories. [Pereira 87] makes the connection explicit between unification-based grammar formalisms and logic programming. [Knight 89] presents an overview to the different fields in computer science which make use of the notion of unification.

[2] For instance, Carpenter's ALE system [Carpenter 92a] gives a user the opportunity to define definite clauses, using disjunction, negation, and Prolog cut.

most general and complete one, bridging the gap between logic programming and unification-based grammar formalisms [Backofen & Smolka 92]. There exist only a few other proposals to feature structures nowadays which do not use standard first order logic directly, for instance Reape's approach, using a polymodal logic [Reape 91].

2 Motivation

Modern typed unification-based grammar formalisms (like TFS, CUF, or *TDL*) differ from early untyped systems like PATR-II in that they highlight the notion of a *feature type*. Types can be arranged hierarchically, where a subtype *inherits* monotonically all the information from its supertypes and unification plays the role of the primary information-combining operation. A *type definition* can be seen as an abbreviation for a complex expression, consisting of type constraints (concerning the sub-/supertype relationship) and feature constraints (stating the appropriate values of attributes) over the standard connectives $\wedge$, $\vee$, and $\neg$. Types can therefore lay foundations for a grammar development environment because they might serve as abbreviations for lexicon entries, ID rule schemata, and universal as well as language-specific principles as is familiar from HPSG. Besides using types as a referential mean as templates are, there are other advantages as well which however cannot be accomplished by templates:

- EFFICIENT PROCESSING
  Certain type constraints can be compiled into more efficient representations like bit vectors (see [Aït-Kaci et al. 89]), where a GLB (greatest lower bound), LUB (least upper bound), or a $\preceq$ (type subsumption) computation reduces to low-level bit manipulation (see section 3.2). Moreover, types release untyped unification from expensive computation, e.g., through the possibility of declaring them incompatible. In addition, working with type names only or with partially expanded types, minimizes the costs of copying structures during processing. This can only be accomplished if the system makes a mechanism for type expansion available (see section 3.4).
- TYPE CHECKING
  Type definitions allow a grammarian to declare which attributes are appropriate for a given type and which types are appropriate for a given attribute, therefore disallowing to write inconsistent feature structures. Again, type expansion is necessary to determine the global consistency of a given description.
- RECURSIVE TYPES
  Recursive types give a grammar writer the opportunity to formulate certain functions or relations as recursive type specifications. Working in the *Parsing as Deduction* [Pereira 83] paradigm enforces a grammar writer to replace the context-free backbone through recursive types. Here, parameterized delayed type expansion is the ticket to the world of controlled linguistic deduction [Uszkoreit 91] (see section 3.4).

3 *TDL*

TDL is a unification-based grammar development environment and run time system supporting HPSG-like grammars. Work on *TDL* has started at the end of 1988 in the DISCO project of the DFKI and led to *TDLExtraLight*, the predecessor of *TDL* [Krieger & Schäfer 93b]. The DISCO grammar currently consists of more than 700 type specifications written in *TDL* and is the largest HPSG grammar for German [Netter 93]. Grammars and lexicons written in *TDL* can be tested by using the parser of the DISCO system. The parser is a bidirectional bottom-up chart parser, providing a user with parameterized parsing strategies as well as giving him control over the processing of individual rules [Kiefer 93]. The core machinery of DISCO consists of *TDL* (see below) and the feature constraint solver *UDiNe* [Backofen & Weyers 93]. *UDiNe* itself is a powerful untyped unification machinery which allows the use of distributed disjunctions, general

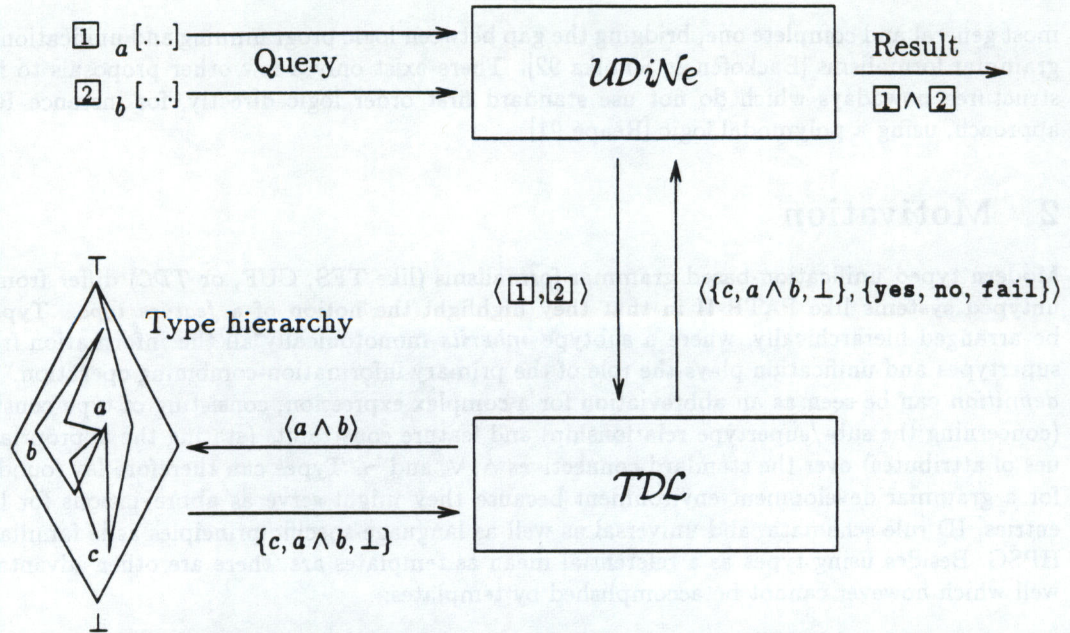


Figure 1: **Interface between *TDL* and *UDiNe*. *Depending on the type hierarchy and the type of* 1 *and* 2, *TDL either returns c (c is definitely the GLB of a and b) or a ∧ b (open-world reasoning) resp. ⊥ (closed-world reasoning) if there doesn't exist a single type which is equal to the GLB of a and b. In addition, TDL determines whether UDiNe must carry out feature term unification (*yes*) or not (*no*), i.e., the return type contains all the information one needs to work on properly (*fail* signals a global unification failure).***

negation, and functional dependencies. The modules communicate through an interface, and this communication mirrors exactly the way an abstract type unification algorithm works: two typed feature structures can only be unified if the attached types are definitely compatible. This is accomplished by the unifier in that *UDiNe* handles over two typed feature structures to *TDL* which gives back a simplified form (plus additional information; see Fig. 1). The motivation for separating type and feature constraints and processing them in dedicated modules (which again might consist of specialized components as is the case in *TDL*) is twofold: (i) it reduces the complexity of the whole system, thus making the architecture much clearer, and (ii) leads to a faster system performance because every dedicated module is designed to cover only a specialized task.

We will now turn our focus to the main ingredients, *TDL* consists of (see Fig. 2). We start with a general overview of the language and then have a closer look on certain modules of the system.

3.1 *TDL* Language

TDL supports type definitions consisting of type constraints and feature constraints over the standard operators ∧, ∨, ¬, and ⊕ (xor). The operators are generalized in that they can connect feature descriptions, coreference tags (logical variables) as well as types. *TDL* distinguishes between avm types (open-world semantics), sort types (closed-world semantics), and built-in types. In asking for the greatest lower bound of two avm types a and b which share no common subtype, *TDL* always returns $a \wedge b$ (open-world reasoning), and not ⊥. The opposite case holds for sort types. Furthermore, sort types differ in another point from avm types in that they are not further structured, like atoms are. Moreover, *TDL* offers the possibility to declare *exhaustive and disjoint partitions* of types, for example *sign* = *word* ⊕ *phrase* which expresses the fact that (i) there are no other subtypes of *sign* than *word* and *phrase*, (ii) the sets of objects denoted by these types are disjoint, and (iii) the disjunction of *word* and *phrase* can be rewritten (during processing) to

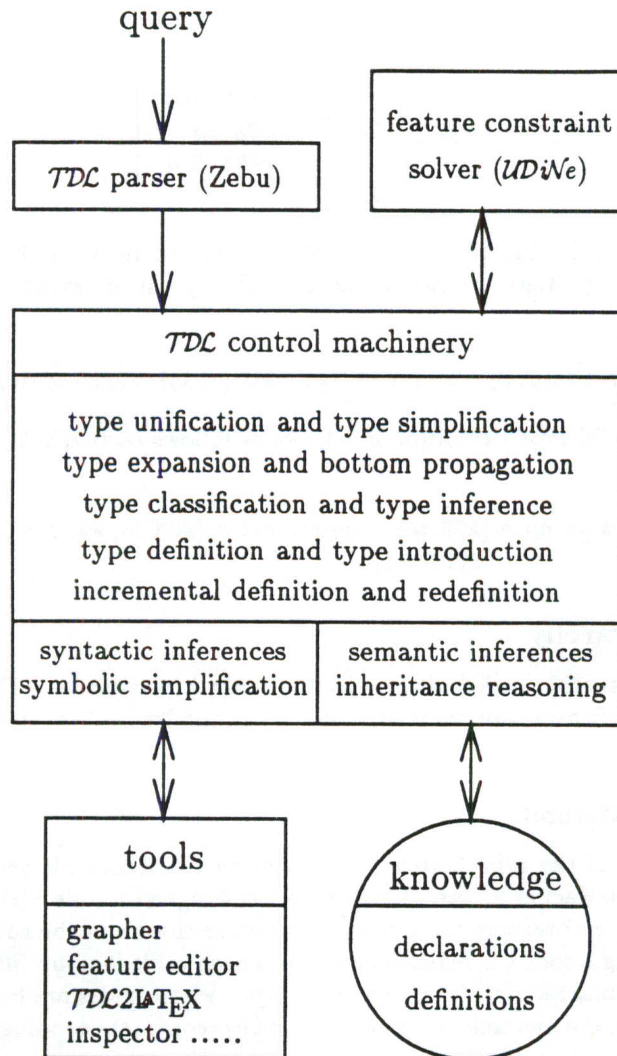


Figure 2: **Architecture of *TDL*.** *The control machinery of TDL is called either by UDiNe during run time or by a user at definition time.*

sign. In addition, one can declare sets of types as *incompatible*, meaning that the conjunction of them yields $\perp$.

TDL allows a grammarian to define and use parameterized templates (macros). There exists a special instance definition facility to ease the writing of lexicon entries which differ from normal types in that they are not entered into the type hierarchy. Strictly speaking, lexicon entries can be seen as the leaves in the type hierarchy which do not admit further subtypes (see also [Pollard & Sag 87], p. 198). This dichotomy is the analogue to the distinction between classes and instances in object-oriented programming languages. Input given to *TDL* is parsed by a Zebu-generated LALR(1) parser [Laubsch 93] to allow for an intuitive, high-level input syntax and to abstract from uninteresting details imposed by the unifier and the underlying LISP system.

The kernel of *TDL* (and of most other monotonic systems) can be given a set-theoretical semantics along the lines of [Smolka 88]. It is easy to translate *TDL* statements into denotation-preserving expressions of Smolka's feature logic, thus viewing *TDL* only as syntactic sugar for a restricted (decidable) subset of first-order logic. Take for instance the following feature description ϕ written as an attribute-value matrix:

$$\phi = \begin{bmatrix} np & \\ \texttt{AGR} & \boxed{\texttt{x}} \begin{bmatrix} agreement \\ \texttt{NUM}\ sg \\ \texttt{PERS}\ 3rd \end{bmatrix} \\ \texttt{SUBJ} & \boxed{\texttt{x}} \end{bmatrix}$$

It is not hard to rewrite this two-dimensional descriprtion to a flat first-order formula, where attributes/features (e.g., AGR) are interpreted as binary predicate symbols and sorts (e.g., *np*) as unary predicates:

$$\exists x \ .\ np(\phi) \wedge \texttt{AGR}(\phi, x) \wedge agreement(x) \wedge \texttt{NUM}(x, sg) \wedge \texttt{PERS}(x, 3rd) \wedge \texttt{SUBJ}(\phi, x)$$

The corresponding $\mathcal{TDL}$ type definition of ϕ looks as follows (actually & is used on the keyboard instead of $\wedge$):

$$\phi := np \wedge [\texttt{AGR}\ \#1 \wedge agreement \wedge [\texttt{NUM}\ sg,\ \texttt{PERS}\ 3rd],\ \texttt{SUBJ}\ \#1].$$

3.2 Type Hierarchy

The type hierarchy is either called directly by the control machinery of $\mathcal{TDL}$ during the definition of a type (type classification) or indirectly via the simplifier both at definition and at run time (type unification).

3.2.1 Encoding Method

The implementation of the type hierarchy is based on Aït-Kaci's bit vector encoding technique for partial orders [Aït-Kaci et al. 89]. Every type t is assigned a code $\gamma(t)$ (represented through a bit vector) such that $\gamma(t)$ reflects the reflexive transitive closure of the subsumption relation with respect to t. Decoding a code c is realized either by a hash table look-up (iff $\exists t_c \ .\ \gamma^{-1}(c) = t_c$) or by computing the 'maximal restriction' of the set of types whose codes are less than c. Depending on the encoding method, the hierarchy occupies $O(n \log n)$ (compact encoding) resp. $O(n^2)$ (transitive closure encoding) bits. Here, GLB/LUB operations directly corresponds to bit-or/and instructions. GLB, LUB and $\preceq$ computations have the nice property that they can be carried out in this framework in $O(n)$ (resp. $O(1)$ on an ideal machine), where n is the number of types.[3]
The method has been modified to open-world reasoning over avm types in that potential GLB/LUB candidates (calculated from their codes), must be verified by inspecting the type hierarchy through a sophisticated graph search. Why so? Take the following example to see why this is necessary:

$$x := y \wedge z$$
$$x' := y' \wedge z' \wedge [a\ 1]$$

During processing, one can definitely substitute $y \wedge z$ through x, but rewriting $y' \wedge z'$ to x' is not correct, because x' differ from $y' \wedge z'$—x' is more specific as a consequence of the feature constraint $a \doteq 1$. Therefore we made a distinction between the 'internal' greatest lower bound $\text{GLB}_{\preceq}$, concerning only the type subsumption relation by using Aït-Kaci's method alone (which is used for sort types) and the 'external' one $\text{GLB}_{\sqsubseteq}$ which takes the subsumption relation over feature structures into account. The same distinction is made for LUBs.
With $\text{GLB}_{\preceq}$ and $\text{GLB}_{\sqsubseteq}$ in mind, we can define a generalized GLB operation informally by the following table. This GLB operation is actually used during type unification.

[3] Actually, one can choose in $\mathcal{TDL}$ between the two encoding techniques and between bit vectors and bignums for the representation of the codes. Operations on bignums are a magnitude faster than the corresponding operations on bit vectors.

GLB	avm_1	$sort_1$	$atom_1$	$feat_constr_1$
avm_2	see 1.	$\bot$	$\bot$	see 2.
$sort_2$	$\bot$	see 3.	see 4.	$\bot$
$atom_2$	$\bot$	see 4.	see 5.	$\bot$
$feat_constr_2$	see 2.	$\bot$	$\bot$	see 6.

where

$$1. \begin{cases} avm_3 \iff avm_3 = \mathrm{GLB}_{\sqsubseteq}(avm_1, avm_2) \\ \bot \iff \bot = \mathrm{GLB}_{\preceq}(avm_1, avm_2) \ \text{(through an explicit incompatibility declaration)} \\ avm_1 \wedge avm_2, \ \text{otherwise (open-world semantics)} \end{cases}$$

$$2. \begin{cases} avm_{1,2} \iff \text{expand-type}(avm_{1,2}) \sqcap feat_constr_{2,1} \neq \bot \\ \bot, \ \text{otherwise} \end{cases}$$

$$3. \begin{cases} sort_3 \iff sort_3 = \mathrm{GLB}_{\preceq}(sort_1, sort_2) \\ sort_1 \iff sort_1 \doteq sort_2 \\ \bot, \ \text{otherwise (closed-world semantics)} \end{cases}$$

$$4. \begin{cases} atom_{1,2} \iff \text{type-of}(atom_{1,2}) \preceq sort_{2,1}, \ (sort_{2,1} \text{ is a built-in type}) \\ \bot, \ \text{otherwise} \end{cases}$$

$$5. \begin{cases} atom_1 \iff atom_1 \doteq atom_2 \\ \bot, \ \text{otherwise} \end{cases}$$

$$6. \begin{cases} \top \iff feat_constr_1 \sqcap feat_constr_2 \neq \bot \\ \bot, \ \text{otherwise} \end{cases}$$

The encoding algorithm is extended to cope with the redefinition of types and the use of undefined types, an essential part of an incremental grammar/lexicon development system. Redefining a type means not only to make changes local to this type. Instead, one has to redefine all dependents of this type—all subtypes, in case of a conjunctive type definition and all disjunction elements for a disjunctive type specification plus, in both cases, all types which mention these types in their definition. The dependent types of a type t can be characterized graph-theoretically via the *strongly connected components* (SCC) of t which respect to the dependency relation. It is important to redefine the dependents in the 'right' order to obtain a new consistent type hierarchy.[4]

3.2.2 Decomposing Type Definitions

Conjunctive, e.g., $x := y \wedge z$ and disjunctive type specifications, e.g., $x' := y' \vee z'$ are entered differently into the hierarchy: x inherits from its supertypes y and z, whereas x' defines itself through its elements y' and z'.[5] This distinction is represented through the use of different kinds of edges in the type graph (bold edges denote disjunction elements, see Fig. 4). But it is worth noting that both of them express subsumption ($x \preceq y$ and $x' \succeq y'$ in the above example) and that the GLB/LUB operations must work properly over 'conjunctive' as well as 'disjunctive' subsumption links.

TDL decomposes complex definitions consisting of $\wedge$, $\vee$, and $\neg$ by introducing *intermediate types*, so that the resulting expression is either a pure conjunction or a disjunction of type symbols. Intermediate type names are enclosed in vertical bars (cf. the intermediate types $|u \wedge v|$ and $|u \wedge v \wedge w|$ in Fig. 3).

[4] Enriching the type hierarchy with dependency links leads in general no longer to a cycle-free graph. So it is not obvious how to establish a topological order on the set of types. However, one can topologically sort the SCCs of the hierarchy without dependency links (which leads to a total order with respect to a certain SCC) and then implode the SCCs of the hierarchy into nodes which ultimately leads to a DAG which itself can be totally ordered.

[5] So one can see conjunctive types as *top-down specialization* of their supertypes and disjunctive ones as *bottom-up generalization* of their disjunction elements.

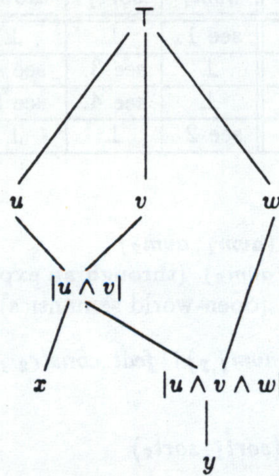


Figure 3: ***The intermediate types $|u \wedge v|$ and $|u \wedge v \wedge w|$ are introduced during the type definitions $x := u \wedge v \wedge [a\ 0]$ and $y := w \wedge v \wedge u \wedge [a\ 1]$.***

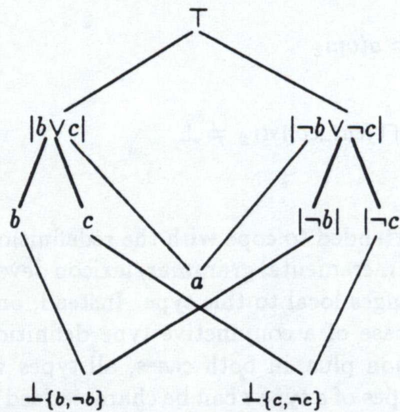


Figure 4: ***Decomposing $a := b \oplus c$, such that a inherits from the intermediates $|b \vee c|$ and $|\neg b \vee \neg c|$.***

The same technique is applied when using $\oplus$ (see Fig. 4). $\oplus$ will be decomposed into $\wedge$, $\vee$ and $\neg$, plus additional intermediates. For each negated type $\neg t$, *TDL* introduces a new intermediate type symbol $|\neg t|$ with the definition $\neg t$ and declares it incompatible with t (see section 3.2.3). In addition, if t is not already present, *TDL* will add t as a new type to the hierarchy (see types $|\neg b|$ and $|\neg c|$ in Fig. 4).

Let's consider the example $a := b \oplus c$. The decomposition performed by *TDL* can then be stated informally by the following rewrite steps (assuming that CNF is switched on):

$$\frac{\dfrac{\dfrac{a := b \oplus c}{a := (b \wedge \neg c) \vee (\neg b \wedge c)}}{a := (b \vee \neg b) \wedge (b \vee c) \wedge (\neg b \vee \neg c) \wedge (\neg c \vee c)}}{\dfrac{a := (b \vee c) \wedge (\neg b \vee \neg c)}{a := |b \vee c| \wedge |\neg b \vee \neg c|}}$$

where $|b \vee c| := b \vee c$, $|\neg b \vee \neg c| := |\neg b| \vee |\neg c|$, $|\neg b| := \neg b$, $|\neg c| := \neg c$, $\perp_{\{b,\neg b\}} := b \wedge |\neg b|$, and $\perp_{\{c,\neg c\}} := c \wedge |\neg c|$.

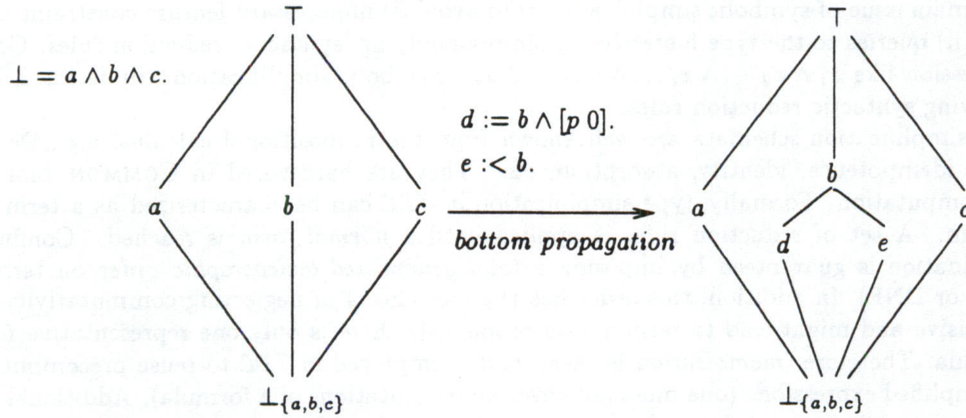


Figure 5: *Bottom propagation triggered through the subtypes d and e of b, so that $a \wedge d \wedge c$ as well as $a \wedge e \wedge c$ will simplify to $\bot$ during processing.*

If disjunctive normal form instead is enforced by the user, the decomposition of $a := b \oplus c$ leads of course to a different type hierarchy:

$$\frac{\dfrac{a := b \oplus c}{a := (b \wedge \neg c) \vee (\neg b \wedge c)}}{a := |b{\wedge}\neg c| \vee |\neg b{\wedge} c|}$$

where $|b{\wedge}\neg c| := b \wedge |\neg c|$, $|\neg b{\wedge} c| := |\neg b| \wedge c$, $|\neg c| := \neg c$, $|\neg b| := \neg b$, $\bot_{\{b,\neg b\}} := b \wedge |\neg b|$, *and* $\bot_{\{c,\neg c\}} := c \wedge |\neg c|$.

3.2.3 Incompatible Types and Bottom Propagation

Incompatible types lead to the introduction of specialized bottom symbols (see Fig. 4 and 5) which are however identified in the underlying logic (this is related to the construction of a *separated sum* in domain theory). These bottom symbols must be propagated downwards by a mechanism called *bottom propagation* which takes place at definition time (see Fig. 5). Note that it is important to take not only subtypes of incompatible types into account but also disjunction elements as the following example shows:

$$\left\{ \begin{array}{l} \bot = a \wedge b. \\ b := b_1 \vee b_2. \end{array} \right\} \xrightarrow{\text{bottom propagation}} a \wedge b_1 = \bot \text{ and } a \wedge b_2 = \bot$$

One might expect that incompatibility statements together with feature term unification lead no longer to a monotonic, set-theoretical semantics. But this is not the case. To preserve monotonicity, one must assume a *2-level interpretation* of typed feature structures, where feature constraints and type constraints can denote different sets of objects and the global interpretation is determined by the intersection of the two sets. Take for instance the type definitions $A := [a\ 1]$ and $B := [b\ 1]$, plus the user declaration $\bot = A \wedge B$ that A and B are incompatible. Then $A \wedge B$ will simplify to $\bot$ although the corresponding feature structures of A and B successfully unify to $[a\ 1, b\ 1]$.

3.3 Symbolic Simplifier

The simplifier operates on arbitrary $\mathcal{TDL}$ expressions. Simplification is done at definition time as well as at run time when typed unification takes place (cf. Fig. 1).

The main issue of symbolic simplification is to avoid (i) unnecessary feature constraint unification and (ii) queries to the type hierarchy by simply applying 'syntactic' reduction rules. Consider an expression like $x_1 \wedge x_2 \ldots \wedge x_i \ldots \wedge \neg x_i \ldots \wedge x_n$. Symbolic simplification will detect $\bot$ by simply applying syntactic reduction rules.
The simplification schemata are well known from the propositional calculus, e.g., De Morgan's laws, idempotence, identity, absorption, etc. They are hard-wired in COMMON LISP to speed up computation. Formally, type simplification in $\mathcal{TDL}$ can be characterized as a term rewriting system. A set of reduction rules is applied until a *normal form* is reached. Confluency and termination is guaranteed by imposing a *total generalized lexicographic order* on terms (either CNF or DNF). In addition, this order has the nice effects of neglecting commutativity (which is expensive and might lead to termination problems): there is only one representative for a given formula. Therefore, *memoization* is cheap and is employed in $\mathcal{TDL}$ to reuse precomputed results of simplified expressions (one must not cover all permutations of a formula). Additional reduction rules are applied at run time using 'semantic' information of the type hierarchy (GLB, LUB, and $\preceq$).

3.3.1 Type Expressions

$\mathcal{TDL}$ type expressions are recursively defined as follows:[6]

- any type symbol is a valid type expression,
- any atom (a quoted symbol, a string or a number) is a valid type expression,
- if $t_1, \ldots, t_n$ are valid type expressions, the *conjunction* $t_1 \wedge \cdots \wedge t_n$ is a valid type expression $(n \geq 0)$,
- if $t_1, \ldots, t_n$ are valid type expressions, the *disjunction* $t_1 \vee \cdots \vee t_n$ is a valid type expression $(n \geq 0)$,
- if t is a valid type expression, its negation $\neg t$ is a valid type expression,
- nothing else is a type expression.

Symbols and negated symbols are also called *literals*.

3.3.2 Normal Form

In order to reduce an arbitrary type expression to a simpler expression, simplification rules must be applied. So we have to define what it means for an expression to be 'simple'. We choose the conjunctive (or disjunctive) normal form. A type expression is in *conjunctive normal form* (CNF), if it is a literal, or a conjunction of literals, or a conjunction of disjunctions of literals. The definition of *disjunctive normal form* (DNF) is the dual counterpart. In $\mathcal{TDL}$, the user may choose between CNF and DNF. The advantages of CNF/DNF are:

- UNIQUENESS
  Type expressions in normal form are unique modulo commutativity. Sorting type expressions according to a total lexicographic order will lead to a total uniqueness of type expressions (see section 3.3.4).
- LINEARITY
  Type expressions in normal form are linear. Any arbitrarily nested expression may be transformed into a 'flat' expression. This may reduce the complexity of later simplifications, e.g., at run time.

[6] For the sake of simplicity, we do not distinguish between *sort* and *avm types* here, both are subsumed by the notion of *type* in this section.

- COMPARABILITY
  This property is a consequence of the two properties mentioned before. Unique and linear expressions make it easy to find or compare (sub)expressions. This is important for the memoization technique described in section 3.3.6.

3.3.3 Reduction Rules

The current implementation of the simplifier uses the following hard-wired reduction rules (only the 'conjunctive' schemata are depicted—the dual set of disjunctive rules are applied as well):

dbl_neg $$\frac{\neg\neg f}{f}$$

demorgan $$\frac{\neg(f_1 \wedge \cdots \wedge f_n)}{\neg f_1 \vee \cdots \vee \neg f_n}$$

distrib $$\frac{f_1 \wedge \cdots \wedge (g_1 \vee \cdots \vee g_m) \wedge \cdots \wedge f_n}{(g_1 \wedge f_1 \wedge \cdots \wedge f_n) \vee \cdots \vee (g_m \wedge f_1 \wedge \cdots \wedge f_n)}$$

flatten $$\frac{f_1 \wedge \cdots \wedge (g_1 \wedge \cdots \wedge g_m) \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge g_1 \wedge \cdots \wedge g_m \wedge \cdots \wedge f_n}$$

idempot $$\frac{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge g \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge f_n}$$

absorpt $$\frac{f_1 \wedge \cdots \wedge h \wedge \cdots \wedge (g_1 \vee \cdots \vee h \vee \cdots \vee g_m) \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge h \wedge \cdots \wedge f_n}$$

inverse1 $$\frac{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge \neg g \wedge \cdots \wedge f_n}{\bot}$$

inverse2 $$\frac{f_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge f_n}{\bot}$$

inverse3 $$\frac{\neg \top}{\bot}$$

neutr_el $$\frac{f_1 \wedge \cdots \wedge \top \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge f_n}$$

identity $$\frac{\bigwedge_{i=1}^{1} f_i}{f_1}$$

empty $$\frac{\bigwedge_{i=1}^{0} f_i}{\top}$$

Note that only one of the two distributivity rules is applied depending on the chosen normal form (CNF or DNF). Otherwise simplification might not terminate.
In order to reach a normal form, it would suffice to apply only the rules dbl_neg, demorgan and distrib. But in the worst case, the application of these three rules would blow up the length of the normal form to exponential size (compared with the number of literals in the original expression). To avoid this, the other rules are used intermediately. If they can be applied, they always reduce the length of the (sub)expressions.

3.3.4 Lexicographic Order

In order to avoid the application of the commutativity rule, we introduce a lexicographic order on type expressions. Together with DNF/CNF, we get a unique sorted normal form for an arbitrary type expression. This guarantees confluency and fast comparability of type expressions.
First of all, we define the order $x <_{NF} y$ on n-ary normal forms by the following table, with symbol $<_{NF}$ neg_symbol $<_{NF}$ conjunction $<_{NF}$ disjunction:

$\downarrow x \quad y \rightarrow$	symbol	neg_symbol	conjunction	disjunction
symbol	$x <_{lex} y$	*true*	*true*	*true*
neg_symbol	*false*	$x_1 <_{lex} y_1$	*true*	*true*
conjunction	*false*	*false*	$\forall i : x_i <_{NF} y_i$	*true*
disjunction	*false*	*false*	*false*	$\forall i : x_i <_{NF} y_i$

where $1 \leq i \leq \max(|x|, |y|)$ and $<_{lex}$ is a total lexicographic order on strings (resp. symbol names), e.g. the predicate STRING< in COMMON LISP, for example:

$$a <_{NF} b <_{NF} bb <_{NF} \neg a <_{NF} a \wedge b <_{NF} a \wedge \neg a <_{NF} a \vee b <_{NF} a \vee b \vee c$$

We then extend $<_{NF}$ for atomic values, such that disjunction $<_{NF}$ atomic_symbol $<_{NF}$ string $<_{NF}$ number. The following matrix is the continuation of the table above at its lower right corner ($1 \leq i \leq \max(|x|, |y|)$):

$\downarrow x \quad y \rightarrow$	disjunction	atomic_symbol	string	number
disjunction	$\forall i : x_i <_{NF} y_i$	*true*	*true*	*true*
atomic_symbol	*false*	$x <_{lex} y$	*true*	*true*
string	*false*	*false*	$x_1 <_{lex} y_1$	*true*
number	*false*	*false*	*false*	$x < y$

3.3.5 Using Information from the Type Hierarchy

Especially at run time, but also at definition time, it is useful to exploit information from the type hierarchy. Further simplifications are possible by using the following rules (it is possible to switch off the use of type hierarchy information at any time). Again, the two dual disjunctive schemata are used as well (extension/LUB).

subsumption $$\frac{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge h \wedge \cdots \wedge f_n}{f_1 \wedge \cdots \wedge g \wedge \cdots \wedge f_n}, \quad \textit{where } g \preceq h$$

GLB $$\frac{f_1 \wedge \cdots \wedge f_n}{g}, \quad \textit{where } g = \mathrm{GLB}(\mathrm{f}_1, \ldots, \mathrm{f}_n)$$

3.3.6 Memoization

The memoization technique described by [Norvig 91] has been adapted in order to reuse precomputed results of type simplification. There are four memoization hash tables for each *TDL* type domain: for CNF with/without hierarchy and DNF with/without hierarchy. The lexicographically sorted normal form described in section 3.3.4 guarantees fast access to precomputed type simplifications. Memoization results are also used by the recursive simplification algorithm to exploit precomputed results for subexpressions.
Some empirical results show the usefulness of memoization. The DISCO grammar for German consists of 750 types, 35 templates, and a toy lexicon of 170 instances/entries. After a full type

expansion of all instances, the memoization hashtables contain 4413 entries (literals are not memoized). 194 849 results have been reused at least once (some up to 1000 times) of which 84.6 % are proper simplifications (i.e., the simplified formulae are really shorter than the unsimplified formulae).

3.4 Type Expansion and Control

As we noted earlier, types allow us to refer to complex constraints through the use of a symbol name. In order to reconstruct the constraints which determine a type (represented as a feature structure), we require a complex operation called *type expansion*. This operation is comparable to Carpenter's *total well-typedness* [Carpenter 92b].

3.4.1 Motivation

In *TDL*, the motivation for type expansion is manifold:

- CONSISTENCY
  The global consistency/satisfiability of a type expression can in general only be decided through type expansion. At definition time, type expansion determines whether a type definition is consistent. At run time, type expansion is involved in checking the consistency of the unification of two typed feature structures.

- ECONOMY
  From the standpoint of efficiency, it does make sense to work only with small, partially expanded structures (if possible) to speed up feature term unification and to reduce the amount of copying. At last however, at the end of processing, one has to make the result (the constraints) explicit.

- RECURSION
  Recursive types are inherently present in modern constraint-based grammar formalisms like HPSG which are not provided with a context-free backbone. Moreover, if the formalism does not allow functional or relational constraints, one must specify certain functions/relations like *append* through recursive types. Take for instance Aït-Kaci's version of *append* [Aït-Kaci 86] which can be stated in *TDL* as follows:

```
append0 := [FRONT < >,
            BACK  #1 ∧ list,
            WHOLE #1 ].
append1 := [FRONT < #first . #rest1 >,
            BACK  #back ∧ list,
            WHOLE < #first . #rest2 >,
            PATCH append∧ [FRONT #rest1,
                           BACK  #back,
                           WHOLE #rest2 ] ].
append := append0 ∨ append1 .
```

- TYPE DEDUCTION
  Parsing and generation can be seen in the light of type deduction as a uniform process, where only the phonology (for parsing) or the semantics (for generation) must be given as

the following simplified example illustrates:

Parsing: [*phrase* PHON ⟨ *"John" "likes" "bagels"* ⟩]

Generation: [*phrase* SEM [RELN *like* ARG1|IND|RESTR|NAME *john* ARG2|IND|RESTR|RELN *bagel*]]

Type expansion (together with a sufficient specified grammar) then is responsible (in both cases) for constructing a fully specified feature structure which is maximal informative and compatible with the input structure. However, the system TFS has shown that type expansion without sophisticated control strategies is hopelessly inefficient and leads to termination problems.

3.4.2 Controlled Type Expansion

Uszkoreit introduced in [Uszkoreit 91] a new strategy for linguistic processing called *controlled linguistic deduction*. His approach permits the specification of linguistic performance models without giving up the declarative basis of linguistic competence, especially monotonicity and completeness. The evaluation of both conjunctive and disjunctive constraints can be *controlled* in this framework. For conjunctive constraints, the one with the highest failure probability should be evaluated first. For disjunctive ones, a success probability is used instead. The alternative with the highest success probability is used until a unification fails, in which case one has to backtrack to the next best alternative.

TDL will support this strategy in that every feature structure is associated with its success/failure potential such that type expansion can be sensitive to these settings. Moreover, one can make other decisions as well during type expansion.

- use the failure/success probabilities or not
- stick to breadth-first or depth-first type expansion
- only regard structures which are subsumed by a given type
- take into account only structures under certain paths
- set the depth of type expansion for a given type

Note that we are not restricted to apply only one of these settings—they can be used in combination and can be changed dynamically during processing. Some of these software switches have been realized in the current implementation of *TDL*'s type expansion mechanism. The next version will incorporate all of them and will be integrated into a declarative specification language which allows a linguists to define *control knowledge* that can be used during processing.

References

[Aït-Kaci et al. 89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. *Efficient Implementation of Lattice Operations.* ACM Transactions on Programming Languages and Systems, 11(1):115–146, January 1989.

[Aït-Kaci 86] Hassan Aït-Kaci. *An Algebraic Semantics Approach to the Effective Resolution of Type Equations.* Theoretical Computer Science, 45:293–351, 1986.

[Alshawi 92] Hiyan Alshawi (ed.). *The Core Language Engine.* ACL-MIT Press Series in Natural Language Processing. MIT Press, 1992.

[Backofen & Smolka 92] Rolf Backofen and Gert Smolka. *A Complete and Recursive Feature Theory.* Technical Report RR-92-30, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1992.

[Backofen & Weyers 93] Rolf Backofen and Christoph Weyers. *UDiNe—A Feature Constraint Solver with Distributed Disjunction and Classical Negation.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[Carpenter 92a] Bob Carpenter. *ALE—The Attribute Logic Engine User's Guide. Version β.* Technical report, Laboratory for Computational Linguistics. Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, December 1992.

[Carpenter 92b] Bob Carpenter. *The Logic of Typed Feature Structures.* Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press, 1992.

[Daelemans et al. 92] Walter Daelemans, Koenraad De Smedt, and Gerald Gazdar. *Inheritance in Natural Language Processing.* Computational Linguistics, 18(2):205–218, 1992.

[Dörre & Eisele 91] Jochen Dörre and Andreas Eisele. *A Comprehensive Unification-Based Grammar Formalism.* Technical Report Deliverable R3.1.B, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Gazdar et al. 85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Harvard University Press, 1985.

[Johnson 88] Mark Johnson. *Attribute Value Logic and the Theory of Grammar.* CSLI Lecture Notes, Number 16. Stanford: Center for the Study of Language and Information, 1988.

[Karttunen 84] Lauri Karttunen. *Features and Values.* In: Proceedings of the 10th International Conference on Computational Linguistics, COLING-84, pp. 28–33, 1984.

[Kasper & Rounds 90] Robert T. Kasper and William C. Rounds. *The Logic of Unification in Grammar.* Linguistics and Philosophy, 13:35–58, 1990.

[Kay 79] Martin Kay. *Functional Grammar.* In: C. Chiarello et al. (ed.), Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society, pp. 142–158, Berkeley, Cal, 1979.

[Kay 85] Martin Kay. *Parsing in Functional Unification Grammar.* In: David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky (eds.), Natural Language Parsing. Psychological, Computational, and Theoretical Perspectives, chapter 7, pp. 251–278. Cambridge: Cambridge University Press, 1985.

[Kiefer 93] Bernd Kiefer. *Gimmie more HQ Parsers.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[King 89] Paul J. King. *A Logical Formalism for Head-Driven Phrase Structure Grammar.* PhD thesis, University of Manchester, Department of Mathematics, 1989.

[Knight 89] Kevin Knight. *Unification: A Multidisciplinary Survey.* ACM Computing Surveys, 21(1):93–124, March 1989.

[Krieger & Schäfer 93a] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL—A Type Description Language for HPSG. Part 2: System Description.* Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993. Forthcoming.

[Krieger & Schäfer 93b] Hans-Ulrich Krieger and Ulrich Schäfer. *TDL ExtraLight User's Guide.* Technical Report D-93-09, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1993.

[Laubsch 93] Joachim Laubsch. *Zebu: A Tool for Specifying Reversible LALR(1) Parsers.* Technical report, Hewlett-Packard, 1993.

[Moens et al. 89] Marc Moens, Jo Calder, Ewan Klein, Mike Reape, and Henk Zeevat. *Expressing generalizations in unification-based grammar formalisms.* In: Proceedings of the 4th EACL, pp. 174–181, 1989.

[Netter 93] Klaus Netter. ***Architecture and Coverage of the DISCO Grammar.*** In: S. Busemann and Karin Harbusch (eds.), Proceedings of the DFKI Workshop on Natural Language Systems: Modularity and Re-Usability, 1993.

[Norvig 91] Peter Norvig. ***Techniques for Automatic Memoization with Applications to Context-Free Parsing.*** Computational Linguistics, 17(1):91–98, 1991.

[Pereira & Shieber 84] Fernando C.N. Pereira and Stuart M. Shieber. ***The Semantics of Grammar Formalisms Seen as Computer Languages.*** In: Proceedings of the 10th International Conference on Computational Linguistics, pp. 123–129, 1984.

[Pereira 83] Fernando C.N. Pereira. ***Parsing as Deduction.*** In: Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, pp. 137–144, 1983.

[Pereira 87] Fernando C.N. Pereira. ***Grammars and Logics of Partial Information.*** In: J.-L. Lassez (ed.), Proceedings of the 4th International Conference on Logic Programming, Vol. 2, pp. 989–1013, 1987.

[Pollard & Sag 87] Carl Pollard and Ivan Sag. ***Information-Based Syntax and Semantics. Vol. I: Fundamentals.*** CSLI Lecture Notes, Number 13. Stanford: Center for the Study of Language and Information, 1987.

[Pollard & Sag 93] Carl Pollard and Ivan Sag. ***Head-Driven Phrase Structure Grammar.*** CSLI Lecture Notes. Stanford: Center for the Study of Language and Information, 1993.

[Reape 91] Mike Reape. ***An Introduction to the Semantics of Unification-Based Grammar Formalisms.*** Technical Report Deliverable R3.2.A, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Russell et al. 92] Graham Russell, Afzal Ballim, John Carroll, and Susan Warwick-Armstrong. ***A Practical Approach to Multiple Default Inheritance for Unification-Based Lexicons.*** Computational Linguistics, 18(3):311–337, 1992.

[Shieber et al. 83] Stuart Shieber, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and Mabry Tyson. ***The Formalism and Implementation of PATR-II.*** In: Barbara J. Grosz and Mark E. Stickel (eds.), Research on Interactive Acquisition and Use of Knowledge, pp. 39–79. Menlo Park, Cal.: AI Center, SRI International, 1983.

[Shieber 86] Stuart M. Shieber. ***An Introduction to Unification-Based Approaches to Grammar.*** CSLI Lecture Notes, Number 4. Stanford: Center for the Study of Language and Information, 1986.

[Smolka 88] Gert Smolka. ***A Feature Logic with Subsorts.*** LILOG Report 33, WT LILOG–IBM Germany, Stuttgart, Mai 1988.

[Smolka 89] Gert Smolka. ***Feature Constraint Logic for Unification Grammars.*** IWBS Report 93, IWBS–IBM Germany, Stuttgart, November 1989.

[Uszkoreit 88] Hans Uszkoreit. ***From Feature Bundles to Abstract Data Types: New Directions in the Representation and Processing of Linguistic Knowledge.*** In: A. Blaser (ed.), Natural Language at the Computer—Contributions to Syntax and Semantics for Text Processing and Man-Machine Translation, pp. 31–64. Berlin: Springer, 1988.

[Uszkoreit 91] Hans Uszkoreit. ***Strategies for Adding Control Information to Declarative Grammars.*** In: Proceedings of the 29th Meeting of the ACL, pp. 237–245, 1991.

[Zajac 92] Rémi Zajac. ***Inheritance and Constraint-Based Grammar Formalisms.*** Computational Linguistics, 18(2):159–182, 1992.

Residuation with Type Constraints

Hendrik C.R. Lock*

Abstract

An alternative to the usual depth-first operational semantics of SLD-resolution is proposed. The aim is to avoid problems with termination and unnecessary recomputation during backtrack search.

The new semantics involves residuation and type constraints. Residuation is here understood as a particular resolution strategy delaying goals that require the unification of unbound input variables. Type constraints are used both to define and restrict the set of terms to which an input variable can be bound, and to generate bindings for the purpose of unification.

The key result is that the proposed operational semantics is correct and complete. Furthermore, residuation combined with type constraints reduces the search space of a program and increases the efficiency considerably.

1 Motivation

The frequent use of *generate and test* in logic programming leads in general to inefficient search. Our work tackles this problem.

Suppose a predicate $p(X)$ holds for any list which is bound to X; simply take:

$$p([]).$$
$$p([_ \mid L]) \quad :- \quad p(L).$$

Further, suppose the existence of a predicate $q(X)$ that holds for $X = [1]$ and $X = [2]$, and some predicate *bang*(..) being expensive to compute (*bang* has no effect on X). E.g.

$$q([1]) \quad :- \quad e_1.$$
$$q([2]) \quad :- \quad e_2.$$

Take a goal:

$$:- \quad p(X),\ bang(..),\ q(X).$$

Under the usual left-to-right depth-first execution strategy of Prolog, *bang* must be backtracked and unnecessarily recomputed in order to make $q(X)$ succeed. This overhead is clearly undesirable.

In the above example, the order of atoms can be statically rearranged (by executing $q(X)$ first) in order to avoid expensive recomputations. In general, however, this will not be feasible as we can construct examples where an optimal execution order depends on actual data. Thus, we seek for a means to rearrange the execution order at run-time.

*IWBS, WZH, IBM Informationssysteme, Vangerowstr. 18, POB 103068, D-69020 Heidelberg, FRG, Email: lock@dhdibm1.bitnet

2 Residuation

How can we avoid unnecessary recomputation? By residuation. In our example, residuation means to delay resolving $p(X)$ until X is bound. However, this works only if $q(X)$ itself entails a unique binding for X, otherwise $q(X)$ must be delayed as well.

A variable upon which an atom is delayed is said to be *demanded* by that atom.

Residuation alone is incomplete, because execution may stop with a set of delayed atoms. On the other hand, residuation can be taken to execute a subclass of logic programs deterministically, and therefore more efficiently.

3 Generating Instances

A complete operational semantics[1] must be able to continue execution on a set of delayed atoms. In such a case, it is necessary to instantiate demanded variables. If a demanded variable is bound, all atoms demanding that variable can be released and considered for further resolution.

The generation of instances can be driven by some of the demanding atoms, for instance by p (which not incidentally defines just the list type).

What have we gained so far? Searching for an instance of some demanded variable X does not entail recomputation of atoms that do not depend on X. For instance in our example, the recomputation of *bang* is entirely avoided, since *bang* has either already been resolved or delayed independently of X.

In summary, residuation combined with instance generation reduces the search space by detecting determinism and therefore avoids unnecessary recomputations.

However, the search space spanned by combining instances of demanded variables that satisfy all delayed atoms still has to be explored. Thereby, either atom may blindly generate bindings that are rejected by the others. For instance, in our example predicate p may generate lists of arbitrary length of which all except two are rejected by q.

4 Type Constraints

When several delayed atoms demand a common variable, a (local) search process is required that finds a binding on which all atoms succeed. Of course, the process of trying subsequent bindings can be driven by any of those delayed atoms. However, as in our example, the one atom may blindly generate infinitely many bindings that are rejected by the others.

Now, the essential idea is to introduce type constraints defining the set of possible terms that can be bound to a demanded variable. Most importantly, type constraints will be used to successively restrict the binding sets in the course of execution. The term *type* will be used in the following as a synonym for binding sets.

[1] Completeness means here the ability to compute any solution of the declarative semantics. Breadth-first derivation strategies are known to be complete but computationally extreme expensive. Depth-first strategies are necessarily incomplete but efficient.

4.1 Types Define Binding Sets

A type constraint attaches a type to any variable X. A type is any finite subset of the set of flat linear terms generated over the set of constructor symbols Σ and the enumerable set of variables $\mathcal{V}$. A term is linear iff each of its variables occurs only once in this term.

$$X : type \quad X \in \mathcal{V}$$
$$type \subseteq \mathcal{T}_1(\Sigma, \mathcal{V}) \quad \%\mathcal{T}_1 \ldots \text{flatterms}$$

We will write e.g. $X : \{./2\}$ instead of $X : \{[_ \mid _]\}$ since the constructor[2] Uniquely denotes the corresponding flat linear term.

The definition of predicate p on page 1 induces the constraint $A : \{[], ./2\}$ with respect to some atom $p(A)$, and q's definition induces $B : \{./2\}$ with respect to $q(B)$. Both type constraints define the set of (partial) terms that can be instantiated for A and B respectively. Note, that ./2 denotes a non-empty list.

A conjunction of atoms sharing a common variable, say X, induces an intersection of the respective types. In the example, this means that the type of X narrows to $X : \{./2\}$.

We now illustrate residuation under type constraints:

$$:- \quad p(X : \{[], ./2\}),\ bang(..),\ q(X : \{./2\}).$$

entails

$$:- \quad p(X : \{./2\}),\ \ldots,\ q(X).$$

$bang(..)$ has been executed (we are not interested in its delayed atoms), and the two other atoms have been delayed.

Since the type has become singular, it entails a unique binding, and thus we immediately obtain:

$$:- \quad p([_ \mid _]),\ \ldots,\ q([_ \mid _]).$$

This clearly shows that type constraints improve the outcome of residuation. Furthermore, at this point atom $p([_ \mid _])$ can be released for further deterministic evaluation as its argument is sufficiently instantiated.

However, this is not true for $q([_ \mid _])$ due to the inherent non-determinism in the definition of q. We will discuss now how q can be handled by means of a simple program transformation.

The definition of q entails that for an atom $q([_ \mid _])$ it cannot be decided which clause to select. Therefore, we can not yet benefit from the type constraint involved, and moreover, we are not able to propagate the type information that the list element must be either 1 or 2. However, a simple program transformation makes the actual source of non-determinism explicit, which in this case is the choice between 1 and 2. The transformation relies on a few ideas: first flatten pattern terms, then coerce overlapping patterns, such as $[_ \mid _]$, and thereby propagate choices to the inside. Transforming the definition of q yields:

$$\begin{array}{lll} q([Y|L]) & :- & L : \{[]\},\ L == [],\ Y : \{1,2\},\ q2(Y). \\ q2(1) & :- & e_1. \\ q2(2) & :- & e_2. \end{array}$$

In the transformed definition, $L : \{[]\}$ constrains the value of L which entails a unique binding, then a match of L follows , and finally the auxiliary predicate $q2$ is called which defines the choice for Y. The additional type constraints for Y reflects the possible bindings.

[2] ./2 is the list constructor

4.2 Terms in Types Are Flat

Most remarkably, flat terms in type constraints suffice entirely. The reason is that unification always proceeds from the outermost part of a term into its arguments. Whenever unification is delayed on the outermost part, no type constraints are required yet for the arguments. Since types contain flat terms, we call them *flat types.*

The following illustrates how type constraints for arguments are generated automatically as soon as the instantiation of the outermost part releases any atom demanding that part.

Below is the predicate definition of p enhanced by explicit type constraints:

$$p([]).$$
$$p([_ \mid L]) \quad :- \quad p(L : \{[], ./2\}).$$

Resolving the atom $p(X)$ by means of the second clause entails the binding $X = [_ \mid L]$ with a type constraint for argument L, which is $L : \{[], ./2\}$. This is a consequence of the recursive occurrence of atom $p(L)$ which is delayed upon the unbound variable L.

We now continue with our example. As we had seen, through the propagation of type constraints in the original goal it was inferred that $X = [_ \mid _]$. This execution sequence first delayed $p(X)$, executed $q(X)$ and subsequently found that $p(X)$ can be released immediately since q entailed the unique binding of X. The current goal is (we may neglect *bang*):

$$:- \quad p(X), \ldots, q(X) \quad \textit{where } X = [_ \mid _].$$

Now, we can resolve p by the second clause of its typed definition (see above):

$$:- \quad p(L : \{[], ./2\}), \ldots, q([Y \mid L]).$$

with the implicit, constrained binding $X = [Y \mid L : \{[], ./2\}]$. Note, that the recursive call $p(L)$ is delayed upon L.

When executing q with respect to its transformed definition introduced in section 4.1, a type constraint appears that restricts L to [], since q holds for [1] and [2] only. Thus, we may safely continue as follows:

$$:- \quad p([]), \ldots, q2(Y : \{1, 2\}).$$

whereby atom $q2(Y)$ suspends on the unbound variable Y. Of course, $p([])$ can be resolved immediately. Hence, the remaining search space is attached to the execution of $q2$ which may bind Y to either 1 or 2.

4.3 Reduced Search

Type constraints reduce the search space. As the above simple example already illustrated, the type constraint on the argument of p excludes $p(X)$ being resolved by means of p's first clause, essentially because X cannot bind to []. Later on, a similar restriction entailed that L cannot bind $[_ \mid _]$. Thus, sources of non-determinism have been eliminated and the search space reduced.

Furthermore, compared to Prolog, the search space in the example has become finite. In Prolog, after backtracking 2 times, p will generate lists of increasing length ($\textit{length} > 1$) on all of which q necessarily fails. Hence, execution will not terminate. Technically, the finite search space is reflected by the constrained binding $X = [Y : \{1, 2\}]$.

Finally, residuation executes *bang*(..) before $p(X)$ and $q(X)$ due to the delay mechanism. This has the desired effect that *bang*(..) needs to be backtracked only in the case when both $p(X)$ and $q(X)$ fail completely.

4.4 Generating Instances through Types

Now, instead of generating instances through delayed atoms, we will exploit the information contained in the type constraints. In our example, instead of letting $q2(Y)$ generate bindings of Y, Y's type will be used instead.

The following procedure is proposed:

1. select some demanded variable,
2. delete a (partial) term from the type and assign it to the variable,
3. release for further resolution all atoms demanding that variable.

When this leads to failure, further alternative bindings are successively enumerated from the selected type. An empty type, of course, entails failure.

With this,

$$:- \quad \ldots,\ q2(Y : \{1,2\}).$$

can be instantiated into

$$:- \quad \ldots,\ q2(1).$$

The remaining possibility $Y : \{2\}$ can be stored in a choice point in order to be considered during backtracking. The remaining atom $q2(1)$, which has been delayed so far, can now be released and resolved immediately.

5 Formal Semantics and Completeness Proof

Residuation with type constraints (TC-residuation) will be formalized as a particular computation rule of SLD-resolution and shown to be correct and complete.

The proof idea is as follows: we want to show that for each successful SLD-derivation in the original program a derivation by TC-residuation exists, and vice versa. First, we define type constraints as *redundant* goals that do not change the observable semantics of a program. A program enhanced by type constraints is then equivalent to the original program. Second, a transformation will be given that allows for a synchronization between evaluating type constraints and original goals. TC-residuation is then defined as a computation rule that *delays* goals whose type constraints are not yet resolved. Finally, we use Lloyd's "Independence of Computation Rule" [Llo84] in order to establish equivalence of SLD-resolution and TC-residuation.

5.1 Preliminaries

We use conventions and notions agreeing chiefly with those in [Llo84], some of them we re-introduce here explicitly. The experienced reader may skip this part.

Σ is an alphabet of predicate symbols enhanced by a function *arity* from Σ into the natural numbers, and $X, Y, \ldots \in \mathcal{V}$ denotes the alphabet of variables. Terms are defined as usual, they are our (uninterpreted) data structures. A *flat* term, e.g. $f(t_1, .., t_n)$ is one whose arguments t_i are all variables, and a *linear* term is one in which each variable occurs only once.

An atom $p(t_1, .., t_n)$ is the application of $p \in \Sigma$ to $n = arity(p)$ terms t_i. A goal is any set of atoms A_1 & .. & A_n, $n \geq 0$; the empty goal denoted by $\square$. We use $G = A \ \& \ G'$ to denote that atom A is contained in goal G. A goal over any Σ' contains only predicate symbols in Σ'. A clause is a pair of head atom and body goal, written *Head* :- *Body*. A program is a set of clauses. $\Sigma(P)$ denotes the predicate symbols occurring in P, and $\Sigma(\backslash P)$ the alphabet Σ excluding the predicate symbols defined by P. A predicate p is defined by P if p occurs in the head of a clause in P.

Substitutions $\sigma, \theta, \ldots$ and their composition are defined a usual. The subsumption ordering on substitutions is: $\sigma_1 \leq \sigma_2$ iff exists θ such that $\theta\sigma_1 = \sigma_2$, that is, σ_1 is less specific (more general) than σ_2. A *unifier* θ is a substitution that makes two terms s, t equal: $\theta s = \theta t$, and θ is a *most general unifier* (mgu) iff for all unifiers σ holds $\theta \leq \sigma$. A match σ of s, t is a uni-directional unifier such that: $s = \sigma t$.

A consistent renaming is an injective substitution α from $\mathcal{V}$ into $\mathcal{V}$. A *new variant* of a clause is a consistent renaming of the clause where all variables are mapped to new symbols never used before.

Given a non-empty goal $G = A \ \& \ G'$ and a program P, the SLD-resolution rule defines the following derivation step:

$A \ \& \ G' \xrightarrow{\theta} \theta(B \ \& \ G')$
if there exists a new variant of a clause in P, H :- B,
and exists an mgu θ such that: $\theta A = \theta H$.

$G \longrightarrow G'$ will also denote the reflexive and transitive closure of the SLD-derivation relation, and for any $n \geq 0$: $G_0 \xrightarrow{\theta_n .. \theta_2 \theta_1} G_n$ iff $G_0 \xrightarrow{\theta_1} G_1, .., G_{n-1} \xrightarrow{\theta_n} G_n$. A derivation $G \longrightarrow G'$ is *successful* if $G = \square$ and *failed* if not exists $G' \longrightarrow \square$.

A *computation rule* specifies the one atom in a goal that is selected next for the SLD-rule to be applied. *Leftmost selection* is a computation rule that assumes goals to be sequentially ordered sequences and which selects always the leftmost atom. The operational semantics of Prolog and related logic programming languages relies on leftmost selection.

5.2 Semantics

Equivalence of programs will be defined in terms of what we can observe, which is the answer substitution produced by any successful derivation. The order in which those substitutions are produced will not be considered here.

Definition 1

Two programs P_1 and P_2 are *(observable) equivalent* iff successful SLD-derivations on all goals G yield equivalent[3] answer substitutions in P_1 and P_2 respectively. Formally: for any goal G: $G \xrightarrow{\theta_1} \square$ in P_1 then there exists $G \xrightarrow{\theta_2} \square$ in P_2 such that $\theta_1 \equiv \theta_2$, and vice versa.

[3] identical up to consistent renaming

When we later enhance a goal G by a type constraint C, we want that, first, none of the successful derivations on G is disabled by C, and second, that C preserves observational equivalence. These two requirements are captured by the notion of redundancy.

Definition 2

Given any goal G in some program P, a goal A defined by a separate program P_A is *redundant* to G iff for any derivation $G \xrightarrow{\theta_1} \Box$ in P :

1. exists $\theta A \longrightarrow \Box$ (hence, $\theta(A \,\&\, G) \longrightarrow \Box$)
2. exists $\theta_1 A \xrightarrow{\theta_2} \Box$ implies $\theta_2 = \epsilon$[4] (not exists a more specific derivation)

The idea is to define type constraints as redundant goals by a separate program that shares no predicate symbols with the original program, and to join both. Thereby, type constraints act on variables of the original program.

Definition 3

C is a type constraint for atom G iff: $C = C_1 \,\&\, .. \,\&\, C_n$, $n \geq 0$, each C_i is a unary atom, and C is redundant to G.

The idea of using unary predicates for type constraints is due to Achim Goltz [Gol93].

Definition 4

Given programs P and R with $\Sigma(P) \cap \Sigma(R) = \emptyset$. The *redundant join* $P + R$ denotes the program $P \cup R$ where each atom G in a goal of P has been replaced by $C \,\&\, G$ where C is a goal defined by R and redundant to G.

Excluding goals defined by R, equivalence is given.

Lemma 5 Given two programs P and R such that $\Sigma(P) \cap \Sigma(R) = \emptyset$, their redundant join $P + R$ is equivalent to P on $\Sigma(\backslash R)$.

Proof By induction on the length of derivations, for any $G \xrightarrow{\theta} \Box$. Length 0 is trivial. Length $n+1, n > 0$: in P, $G \xrightarrow{\theta_1} G_1 \xrightarrow{\theta'} \Box$. In $P + R$, $A \,\&\, G \xrightarrow{\theta\sigma_1} \sigma_1 G \xrightarrow{\sigma_2} G_1$ by part (1) of definition 2. Part (2) implies $\sigma_2\sigma_1 = \theta_1$. By the induction hypothesis, $G_1 \xrightarrow{\theta'} \Box$ in $P + R$. The inverse direction is analogous. $\Box$

The following examples will illustrate the definitions. Given the program P_{app}:

```
append([] ,     L, L    ).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

The trivial type constraint being redundant to any goal is obviously defined as:

```
omega( _ ).
```

Thus, the redundant join contains

---

[4] ϵ is the identity substitution

```
append([X|L1],L2,[X|L3]) :- omega(L1) & omega(L2) & omega(L3),
                                append(L1,L2,L3).
```

which is for obvious reasons equivalent to the original program. A more specific type constraint is defined by P_{list}:

```
t_list( []    ).
t_list( [_|_] ).
```

and the redundant join $P_{\text{app}} + P_{\text{t_list}}$ now contains instead:

```
append([X|L1],L2,[X|L3]) :- t_list(L1) & append(L1,L2,L3).
```

The new type constraint is redundant to **append(L1,L2,L3)** since each use of **append** yields an answer substitution that is at least as specific as the one of **t_list**. Furthermore, the following violates redundancy as it produces more specific answer substitutions than the original:

```
t_list(L2) & append([],L2,L3).
```

As mentioned earlier, the essence of residuation is delaying the evaluation of atoms until they are sufficiently instantiated. An atom is sufficiently instantiated iff the head unification degenerates to a (deterministic) pattern matching process. The idea is that type constraints test whether an atom must be delayed or not. In order to realize this synchronization concept on the abstract level of derivation sequences we introduce a simple source code transformation. The transformation introduces type constraints and auxiliary predicates that are used to control the synchronization.

Definition 6

Let P be a program. We define an auxiliary predicate p' for each p defined by P. Each clause in P

```
p(t1,..,tn) :- Goal.
```

is transformed into

```
p'(t1,..,tn) :- Goal.
p(X1,..,Xn) :- tc -> p'(X1,..,Xn).
```

where **tc** is a type constraint to **p'(X1,..,Xn)** defined by some program T. The new connective **->** has the same declarative meaning as **&**, but its operational semantics will require left-to-right evaluation (see below). The transformation is called *type enhancement* of P and defines a particular redundant join.

After these preparation we are able to define the derivation relation. The first two rules resolve type constraints

Definition 7

Let $P + T$ be a type enhanced program and $tc \rightarrow G$ be any goal.
The rule *tc-match* is: $tc \xrightarrow{\epsilon} \square$ then $(tc \rightarrow G) \rightarrow G$.
The rule *tc-unify* is: $tc \xrightarrow{\theta} \square$ then $(tc \rightarrow G_1) \;\&\; G_2 \xrightarrow{\theta} \theta(G_1 \;\&\; G_2)$.

Definition 8

TC-residuation is the following computation rule. Let $P + T$ be a type enhanced program, G be any goal, and Σ_p the predicate signature of the original P.

deterministic evaluation **(R-rule): select any atom in G such that the leading symbol belongs to Σ_p. Apply the resolution rule once, that is:**

$$p(t_1,..,t_n) \;\&\; G \xrightarrow{\epsilon} (\sigma(tc \twoheadrightarrow p'(X_1,..,X_n))) \;\&\; G \quad \textit{where } \sigma = \{X_i \mapsto t_i \mid i = 1..n\}$$

type resolution **(M-rule):** apply a tc-match to G whenever possible.

instantiation **(I-rule):** apply tc-unify to G if neither rule 1) nor rule 2) apply.

A *TC-derivation* is any SLD-derivation $G \xrightarrow[tc]{\theta} G'$ that obeys solely the TC-residuation rule. If $G \xrightarrow[tc]{\theta} G'$ and TC-residuation does not apply to G', the derivation is said to be finitely failed; if $G' = \Box$ it is said to be successful.

Thus, TC-residuation exploits as much determinism as possible before it starts to search non-deterministically for instantiations of logical variables. The generation of instances is entirely controlled by type constraints since this enables the search to prune inconsistent instantiations as early as possible. The transformation in definition 6 guarantees that the M-rule produces the empty answer substitution ϵ.

5.3 Correctness and Completeness

The following theorem is essential for proving the correctness and completeness of TC-residuation.

Theorem 9 *Independence of Computation Rule* ([Llo84]): Let P be some program and γ be some computation rule. For any goal G, if $G \xrightarrow{\theta} \Box$ in P then exists a derivation $G \xrightarrow{\theta} \Box$ in P that obeys γ.

We are now able to state our main theorem:

Theorem 10 TC-residuation is correct and complete. Let P be a program, $P + T$ any type enhancement, and G a goal over $\Sigma(\backslash T)$. For any SLD-derivation $G \xrightarrow{\theta_1} \Box$ in P exists a TC-derivation $G \xrightarrow[tc]{\theta_2} \Box$ in $P + T$ such that $\theta_1 = \theta_2$ (completeness), and vice versa (correctness).

Proof $\circ$ By Lemma 5 any program P is equivalent to any type enhancement $P + T$.
$\circ\circ$ The R-rule and I-rule together simulate the full SLD-rule. The task of the additional M-rule is to resolve all cases with empty answer substitution, and to delay all others. In the cases where the M-rule is not applicable, it is covered by the I-rule. Therefore, TC-residuation is a particular computational rule for SLD-resolution. By the independence of computation rule, evaluating any goal over $\Sigma(\backslash T)$ in $P + T$ by SLD-resolution is equivalent to evaluating it by TC-residuation. The theorem follows by argument $\circ$ and $\circ\circ$. $\Box$

6 Related Work

Various *delay concepts* have been proposed for logic programming languages that provide an explicit means for controlling the execution order of goals, e.g. [Nai84, MDV88, Hen89, Sha89]. Common to all these approaches is that the execution of a goal can be delayed as long as it is not sufficiently instantiated. For instance, in [Hen89] we find a realization of the inequality operator by means of a delay concept called *forward declarations*: the inequality predicate acts as a constraint which is active only when both arguments are instantiated to ground terms. A particular direction of research uses such delay concepts for modelling concurrency, see [Sha89] for a comprehensive discussion.

The problem with delay concepts is that they do not preserve the semantics, in particular completeness is lost. When using them, a program must ensure that each consumer (delayed goal) has at least one producer that will not be delayed. If this requirement is not met, execution may deadlock, that is, a set of pending goals will remain in the end. It is known that this requirement is hard to meet, and it seems in general not decidable whether a program is deadlock free.

Residuation is an operational mechanism that applies the delay concept implicitly to all parameters of an expression that are not yet sufficiently instantiated. It has been proposed originally in [AKLN87] as a mechanism that guarantees deterministic evaluation of functional expressions in a logical context with unbound variables. Without residuation, evaluating an expression containing logical variables is highly non-deterministic as different instantiations of the logical variables can by guessed, and therefore different results computed. This non-determinism has been identified as the source of new, large search spaces. With residuation, evaluating an expression is delayed until (logic) computations elsewhere provide a sufficient instantiation in order to carry on deterministically. Again, residuation is incomplete. A sufficient condition for functional logic programs which are complete under residuation has been given recently in [Han92b]. Furthermore, the same author describes in [Han92a] how to improve control over logic programs in a declarative way by a translation (by hand) into functional logic programs. This approach solves some of the above-mentioned problems but is not general enough as the problem of finding the right control structure is entirely delegated to the programmer.

The *constraint satisfaction techniques* described in [Hen89] come closest to our own approach. This approach combines a delay concept with a finite domain concept. A finite domain describes a set of values that can be assigned to a variable. Particular constraint handling techniques are used to narrow domains during execution. The effect is an a priori pruning of search spaces. There are certain important differences to our approach: domain values are constants and the universe of values is interpreted (e.g. arithmetic over integer intervals), whereas we consider the universe of syntactic, uninterpreted term structures. The work of [Hen89] concentrates accordingly on specialized satisfaction techniques that exploit the laws of the interpreted universe, and provides a complete operational semantics.

7 Summary

In summary, the elements of our operational semantics, as we have introduced them informally, are:

1. residuation,
2. type constraints,
3. generation of instances from types.
4. backtracking (enumerate alternative bindings)

The key result is that residuation combined with type constraints reduces search spaces and avoids unnecessary recomputations of independent, deterministic goals during backtracking. Whether execution on the whole is more efficient depends on the savings achieved by reduced search, as we have to take into account the additional overhead spend maintaining queues of delayed goals and the associated backtrack information. As a particularly nice effect, the search space becomes finite in some cases where Prolog would not terminate.

Another result is that flat terms suffice in type constraints to define bindings sets. This is important since type inference of flat types is easier than for nested types. In particular, type equivalence is decidable.

We remark that the proposed operational semantics is quite similar to lazy evaluation in functional languages since the evaluation order is scheduled dynamically through demands issued by pattern matching. However, an essential difference exists: in logical programming, all atoms must be evaluated in order to establish the proof, whereas in the functional context some expressions might be suspended forever.

Furthermore, our semantics has strong relations to consistency techniques as used in constrained logic programming with finite domains [Hen89]. So we could view our types as finite domains. The essential difference is that our domains contain term structures instead of constants, which entails for instance that inequality must be treated differently.

As future work, it might be worth investigating the issue of implementation. Delaying and releasing goals, as well as propagating type constraints are challenging novelties. It remains to be seen if a WAM based approach is suitable. Furthermore, the semantics presented here might be used for a logic programming language, with the effect that it will be semantically closer to Horn logic than Prolog is. A problem then is that the order of execution cannot be predicted statically, which is annoying when I/O is intended. However, this can be overcome by mechanisms similar to those used in lazy functional languages, e.g. monads in Haskell [Wad90, Wad92].

References

[AKLN87] Hassan Ait-Kaci, P. Lincoln, and Roger Nasr. LeFun: Logic, equations, and functions. In *Proceedings of the 4th Symposium on Logic Programming, San Francisco*, pages 17–23, 1987.

[Gol93] Achim Goltz, May. 1993. private communications.

[Han92a] Michael Hanus. Improving control of logic programs by using functional logic languages. In *Fourth Int. Symposium on Programming Language Implementation and Logic Programming*, pages 1–23. LNCS 631, 1992.

[Han92b] Michael Hanus. On the completeness of residuation. In *Proceedings of the Joint Int. Conference and Symposium on Logic Programming*. MIT Press, 1992.

[Hen89] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, Heidelberg, 1984.

[MDV88] M. Meier, P. Dufresne, and D. Henry de Villeneuve. SEPIA. Technical Report TR-LP-36, European Computer Industry Research Center, Munich, 1988.

[Nai84] Lee Naish. Mu-Prolog 3.11db referencce manual. Technical report, Melbourne University edition, 1984.

[Sha89] Ehud Y. Shapiro. The family of concurrent logic programming languages. In F.L. Bauer, editor, *Logic, Algebra and Computation, International Summer School, Marktoberdorf*, NATO ASI Series, Vol. 79. Springer Verlag, 1989.

[Wad90] Phil Wadler. Comprehending monads. In *ACM Conference on LISP and Functional Programming*. ACM, 1990. also in *Journal of Mathematical Structure in Computer Science*.

[Wad92] Phil Wadler. The essence of functional programming. In 19th *Annual Symposium on Principles of Programming Languages*, 1992. invited talk, manuscript by ftp.

Polymorphe Featuretypen — Typinferenz und Typüberprüfung

Gregor Meyer und Sybilla Weigel *

1 Einleitung

Im Verlauf der 80er Jahre sind etliche Erweiterungen für die logische Programmierung und insbesondere für Prolog entstanden, von denen hier zwei Richtungen näher untersucht werden sollen. Zum einen wurde das Konzept der 'frames' aus der Wissensrepräsentation und KI [Min75] verbunden mit objektorientierten Konzepten auf Prolog übertragen. Zentrale Merkmale sind die Vererbung von Attributen und Methoden sowie die Kapselung von Methoden in Objekten oder Klassen. Besonders für die Vielzahl von parallelen logischen Programmiersprachen ist das Bild des Versendens von Nachrichten bestimmend (vgl. Übersicht in [DP91]). Wichtige neue Vertreter von (nicht-parallelen) objektorientierten Prologerweiterungen sind Prolog++ der Firma LPA und L&O [McC92].
Die zweite relevante Richtung ersetzt oder erweitert die Termdarstellung (1. Ordnung) durch attributierte Terme (Featureterme) kombiniert mit einem geordneten Typverband (auch Sortenverband genannt) wie in LogIn [AKN86] und LIFE [AKP91]. Featureterme oder Featurestrukturen haben insbesondere für die in der Computerlinguistik verwendeten Formalismen eine wichtige Bedeutung [Car92]. Sie erlauben eine 'record'-ähnliche partielle Beschreibung der Daten mit Attributen (Feature) und den zugehörigen Werten. So beschreiben z.B. die Featureterme

```
person { vorname => "Hans", alter => 24 }
student { uni => "Heidelberg" }
```

eine Person mit dem Namen 'Hans' und dem Alter '24', sowie einen Studenten an der Universität Heidelberg. Ist weiterhin '**student**' als Untertyp von '**person**' deklariert, dann liefert die ordnungssortierte Unifikation der beiden obigen Terme

```
student { vorname => "Hans", alter => 24 , uni => "Heidelberg" }.
```

In der logischen Programmiersprache 'Feature PROTOS-L', die aus den ordnungssortierten Sprachen PROTOS-L [Bei92] und TEL [Smo88] hervorgegangen ist, sind (zusätzlich zu den bereits vorhandenen Konstruktortermen) ebenfalls Featureterme im Datenmodell enthalten. Darüber hinaus existiert eine strenge Typdisziplin: Alle Featuretypen werden explizit deklariert, u.a. werden die erlaubten Attributnamen und die zugehörigen Typen definiert und auch Prädikate besitzen Typdeklarationen, die angeben, welche Typen die Argumente haben. Es gibt eine genaue Trennung zwischen Featuretypen und Featuretermen. Ein ähnlicher Ansatz wurde auch schon

*Postanschrift: Gregor Meyer, c/o IBM Deutschland, WZH - IWBS, Postfach 103068, 69020 Heidelberg, email: gmeyer @ dhdibm1.bitnet

mit 'Feature TEL' [Seu90] verwirklicht. Die im folgenden diskutierten Aspekte gehen darüber hinaus.

- Featuretypen können parametrisch polymorph sein. Die Typen von Attributen sind nicht immer durch eine Typkonstante gegeben, sondern können auch durch Typparameter definiert sein.
- Bei Koreferenzen überprüft der Compiler nicht nur die reine Typinformation sondern kann auch inkonsistente Werte entdecken.
- Featuretypen können über Modulgrenzen hinweg erweitert werden. Die Module werden getrennt übersetzt, wobei eine systemweite Typsicherheit gewährleistet wird.
- Die Typinferenz leitet nicht nur Typen von Variablen her, die lokal in einer Klausel verwendet werden, sondern sie wurde auch auf die Inferenz von Relationsdeklarationen erweitert.

Die entsprechenden Phasen des Compilers wurden bereits implementiert und sind in [Wei93] dokumentiert. In diesem Bericht wird die Typüberprüfung und Typherleitung für 'Feature PROTOS-L' skizziert und auf den Bezug zu objektorientierten Sprachen eingegangen.

2 Grundlagen

Ein **Featuretyp** ist mit einer Recordstruktur vergleichbar. Seine Elemente nennt man **Feature** oder **Attribute**. Diese werden mit einem Attributnamen angesprochen und beschreiben die Eigenschaften der Objekte dieses Typs. Im nachfolgenden Beispiel werden Objekte der Featuretypen **'person'** und **'datum'** beschrieben. Zu den Attributen des Typs **'person'** gehören **'vorname'**, **'nachname'** und **'geburtstag'** mit den Typen **'string'** und **'datum'**.

```
person := [ vorname    : string;
            nachname   : string;
            geburtstag : datum ].

datum := [ tag   : int;
           monat : int;
           jahr  : int
         ]
```

Die Featuretypen können in Hierarchien, analog zu Klassenhierarchien in objektorientierten Programmiersprachen, angeordnet werden. So kann der Featuretyp **'student'**, der ein Untertyp des Featuretyps **'person'** ist, folgendermaßen deklariert werden:

```
student  := person  [ uni : string ].
```

Dieser Featuretyp erbt die Attribute **'vorname'**, **'nachname'** und **'geburtstag'**. Zusätzlich besitzt er ein Attribut **'uni'**.
Die Werteterme eines Featuretyps werden Featureterme genannt.

Eine der ersten logischen Programmiersprachen mit Featuretermen war LogIn, in der Prolog um sogenannte ψ-Terme erweitert wurde. Die ψ-Terme entsprechen den Featuretermen in Feature PROTOS-L. Die Unifikation von Termen in Prolog wird durch ψ-Termunifikation ersetzt. Die Beschreibung einer Person namens `'Otto'` kann in LogIn mittels eines ψ-Termes folgendermaßen dargestellt werden:

```
person { vorname    => "Otto"
         geburtstag => { tag   => 10;
                         monat => 10;
                         jahr  => 1942
                       }
       }
```

Der oben angegebene Featureterm repräsentiert Personen, deren Vorname `'Otto'` ist und die ihren Geburtstag am `'10.10.1942'` haben. Ein Featureterm beschreibt implizit eine Menge von Werten, die subsumiert werden [AKP91]. Die Sprache LogIn unterscheidet nicht zwischen Featuretypen und Featuretermen. In 'Feature PROTOS-L' hingegen beschreibt der Typ explizit die Struktur eines Terms. Der Typ beschreibt, was für jeden Term des Typs gilt, bzw. was für einen Term gelten muß, damit er für diesen Typ erlaubt ist. Dies unterstreicht den starken Bezug des Typkonzepts zur (statischen) Typüberprüfung im Compiler. In LogIn werden ψ-Term-Hierarchien im Gegensatz zu Relationen deklariert. Eine Typüberprüfung findet aber nicht statt.
In LogIn (wie auch in Feature PROTOS-L) werden auch Koreferenzen, die verschiedene Attribute eines ψ-Terms an einen Werteterm binden, verwendet. So könnte z.B. der ψ-Term, der die Person `'Otto'` beschreibt, erweitert werden zu:

```
person { vorname   => "Otto"
         nachname  => X : string
         geburstag => { tag   => 10;
                        monat => 10;
                        jahr  => 1942
                      }
         vater => { nachname => X }
       }
```

Durch die Koreferenzvariable `'X'` wird definiert, daß der Nachname des Vaters der Person mit dem Nachnamen dieser Person übereinstimmt. Diese beiden Attribute sind immer mit dem gleichen Werteterm unifiziert.

3 Parametrischer Polymorphismus

Eine mögliche Definition des Typs 'list' als Featuretyp ohne Verwendung des parametrischen Polymorphismus ist

```
type list.
type elist := list.          % an empty list is a list (subtyping)
type nelist := list          % a non empty list is a list
        [ head : term;       % with any term as head
          tail : list ].     % and a list as tail
```

Der Typ 'term' steht hier für den allgemeinsten Typ. Unter der Annahme, daß die Typen 'int' und 'string' mit der üblichen Bedeutung Untertypen von 'term' sind, wären mögliche Terme des Typs 'list'

```
nelist { head => 1 , tail => nelist { head => 2 , tail => elist } }
nelist { head => 1 , tail => nelist { head => "ab", tail => elist } }
```

Während die 'head'-Werte im ersten Term beide vom Typ 'int' sind, ist auch eine annähernd beliebige Mischung möglich. Die einzige Bedingung ist, daß die 'head'-Werte zu dem relativ allgemeinen Typ 'term' gehören müssen. Eine solche Liste wird auch als heterogene Datenstruktur bezeichnet. Der Polymorphismus, der dabei zum Tragen kommt, wird in [CW86] 'inclusion polymorphism' genannt.

Eine analoge Definition des Typs 'list_p', die parametrischen Polymorphismus verwendet, ist

```
type list_p(T).
type elist_p := list_p(_).              % an empty list is a list of terms of any type
type nelist_p(T) := list_p(T)           % a non empty list is a list
                 [ head : T;             % with a term of type T as head
                   tail : list_p(T) ].   % and a list as tail
```

Der Typ der 'head'-Werte ist nicht statisch durch eine Konstante festgelegt, sondern wird durch einen Typparameter (hier mit dem Namen 'T') bestimmt. Dieser wird erst an den Stellen, wo Terme des Typs list_p vorkommen, angegeben wie z.B. als Argument einer Relation

```
rel sum : list_p(int) x int.
    sum(L,S) <-- S is sum of values in L.
```

Die Typüberprüfung des Compilers stellt für jeden Aufruf der Relation 'sum' sicher, daß das erste Argument tatsächlich eine (homogene) Liste von Zahlen ist. Ohne die Möglichkeit des parametrischen Polymorphismus müßte jedesmal zur Laufzeit geprüft werden, ob die jeweiligen 'head'-Werte zum Typ 'int' gehören, damit die Addition sinnvoll ist.

Folgender Grammatikausschnitt definiert den syntaktischen Aufbau einer Deklaration einer Featuretyphierarchie.

```
Featuretyp      ::= ['type'] Featuretypname ['('Args')']
                    [':=' Obertypliste '[' Featureliste ']'].

Obertypliste    ::= [ Featuretypterm {'*' Featuretypterm }].

Featureliste    ::= [ Featuredekl {';' Featuredekl }].

Featuredekl     ::= ( [Visib] Featurename ':' Typterm
                    | [Visib] Featurename '=>' Initialwert [ ':' Typterm ]
                    | [Visib] Featurename '=>' Koreferenz
                    )

Initialwert     ::= ( Konstruktorterm
                    | Featureterm
                    )

Koreferenz      ::= Variable [ ':' Typterm ]
```

```
Visib           ::=  'public' | 'private'

Featureterm     ::= Featuretypname '{' Feature {';' Feature } '}'

Feature         ::= ( Featurename '=>' Variable [ ':' Typterm ]
                    | Featurename '=>' Werteterm
                    | Featurename '=>' Featureterm
                    )

Featuretypterm ::= Featuretypname [ '(' Args ')' ]

Typterm         ::= Typname [ '(' Args ')' ]

Typname         ::= ( Featuretypname
                    | Konstruktortypname
                    )

Args            ::= Arg { ',' Arg }

Arg             ::= ( Typterm
                    | Typparameter
                    )

Typparameter    ::= Variable
```

Die Liste der Parameter *Args* ist bei einem monomorphen Featuretyp leer, bei einem polymorphen Featuretyp besteht sie aus Typparametern. Die *Obertypliste* besteht aus einer eventuell leeren Liste der Obertypen, von denen Feature ererbt werden. Die *Featureliste* gibt die neu hinzukommenden Attribute an, bzw. schränkt den Typ von ererbten Attributen ein. In einer Featuretypdeklaration muß immer der Featurename und der Typ oder der Initialwert angegeben werden. Optional kann die Sichtbarkeit des Attributs angegeben werden (s. Abschnitt 5).
Die Diskussion, ob parametrische Featuretypen nötig sind, kann sich an den aktuellen Entwicklungen bei prozeduralen Programmiersprachen orientieren. Während Eiffel [Mey92] die Möglichkeiten zur Definition generischer Klassen, die den parametrisch polymorphen Featuretypen in etwa entsprechen, stark ausgeprägt sind, sind in Oberon keine parametrisch polymorphen Datentypen möglich. Stattdessen wird darauf hingewiesen, daß sich der parametrische Polymorphismus mit den gegebenen Mitteln simulieren läßt (z.B. [Mös92]). Diese Simulation erfordert aber zusätzliche Typtests zur Laufzeit und schränkt die Möglichkeiten zur statischen Typüberprüfung ein. In 'Feature PROTOS-L' wurde daher der parametrische Polymorphismus in das Featuretypkonzept integriert. Parametrischer Polymorphismus ist insbesondere für die Definition von allgemein verwendbaren Datenstrukturen nützlich, bei denen beliebige andere Terme in speziellen Strukturen wie z.B. Bäumen oder Graphen verwaltet werden (sog. Container-Datenstrukturen).

4 Typüberprüfung vs. Werteüberprüfung

Um Bindungen verschiedener Attribute an den gleichen Werteterm darzustellen, können Typdefinitionen Koreferenzen enthalten. Weiterhin können Werte von Attributen schon in der Typdefinition als Initialwert angegeben werden, generell sind beliebige Terme als Initialwerte arlaubt.
Beispiel:

```
person := [ info => "eine natürliche Person", ...]
gleichalt := person [
                info => "Person dessen Vater und Mutter gleich alt sind";
                vater => person { alter => N : nat };
                mutter => person { alter => N } ].
```

Werte von Attributen werden durch ein vorangestelltes '=>' gekennzeichnet und Typen durch ':'. Die Gleichheit der Attribute 'alter' wird durch die Koreferenzvariable 'N' angegeben. Bei jedem Wert, der zu dem Typ 'gleichalt' gehört, sind die Attribute 'vater.alter' und 'mutter.alter' identisch. Es zeigt sich, daß die Überprüfung von Typen und von Werten ineinander übergeht. Der Typ des Attributs 'info' wird aus dem Initialwert automatisch als 'string' abgeleitet. Der Compiler kann in der Definition von 'gleichalt' eine Inkonsistenz der Werte für 'info' erkennen, obwohl der Typ der Attribute immer 'string' ist. Derartige Inkonsistenzen können auch erkannt werden, wenn Koreferenzen an verschiedene Werte gebunden werden. Diese erweiterte Typüberprüfung ist natürlich nicht vollständig in dem Sinne, daß nicht alle derartigen Situationen erkannt werden, weil sonst das Programm z.B. abstrakt interpretiert werden müßte. Die Verbindung von Werten und Typen bei der Definition von Featuretypen ermöglicht, daß der Compiler einerseits eine strenge statische Typüberprüfung durchführen kann, d.h. der Typanteil wird vollständig geprüft; andererseits werden die Ausdrucksmöglichkeiten für die Definition der Typen nicht eingeschränkt.

5 Featuretypen und Module

In objektorientierten Sprachen werden die Attribute und Methoden einer Klasse häufig aufgeteilt in solche, die außerhalb der Klasse zugreifbar sind und solche, die nur privat innerhalb der Klasse (von deren Methoden) verwendet werden. Das Konzept, das für die Strukturierung der Daten verwendet wird, bestimmt gleichzeitig auch die Sichtbarkeit von Informationen. In 'Feature PROTOS-L' werden diese beiden Konzepte getrennt. Die Sichtbarkeit wird durch das Modulkonzept gesteuert. Die Sichtbarkeit von Namen, z.B. Typen und deren einzelne Attribute wird an der Exportschnittstelle eines Moduls (in gewissen Grenzen) unabhängig von der Struktur der Daten angegeben. Innerhalb eines Moduls sind alle Namen, die importiert oder lokal neu deklariert werden, zugreifbar. Für diesen Zweck gibt es eigene Schlüsselwörter, mit denen man z.B. die Sichtbarkeit von einzelnen Attributen definieren kann. Die Einhaltung der Sichtbarkeitsregeln wird statisch vom Compiler geprüft.
Für die objektorientierte Programmierung ist ein wichtiges Grundprinzip, daß die Definitionen von Klassen an beliebigen Stellen wiederverwendet werden, um einfach und sicher neue Klassen zu erstellen, die sich nur wenig von schon bestehenden unterscheiden. Ähnliche Prinzipien gibt es für die Wissensrepräsentation, wenn umfangreiches Wissen über Konzepte so dargestellt werden soll, daß Gemeinsamkeiten

einfach ausgedrückt werden können und die Wissensbasis modularisiert ist. Bezogen auf Featuretypen ergibt sich dann in Verbindung mit dem Modulkonzept die Anforderung, Strukturen, die in einem Modul exportiert werden, in einem beliebigen anderen Modul erweitern zu können. Allerdings wird häufig in Systemen zu Sprachen mit Featuretermen bei der Übersetzung bzw. Interpretation eines Programms implizit die Annahme gemacht, daß die gesamte Hierarchie der Featuretypen bekannt ist. Da dann die maximale Anzahl von Attributen zu jedem Typ und seinen Untertypen bekannt ist, ist die interne Darstellung der Featureterme einfach. Eine spätere Erweiterung eines Featuretyps (in einem anderen Modul) ist meist mit einer Neuübersetzung aller Programmteile verbunden.
In 'Feature PROTOS-L' wird diese Annahme nicht verwendet, weil sie als unrealistisch betrachtet wird. Für eine abstrakte Maschine zur Abarbeitung von logischen Programmen mit Featuretermen hat das zwar zur Folge, daß die Indizierung von Attributen komplizierter wird, aber Erfahrungen mit der Implementierung von konventionellen objektorientierten Sprachen haben gezeigt, daß der Zusatzaufwand im Vergleich zum Nutzen vernachlässigbar ist. Eine Umsetzung der Techniken in eine abstrakte Maschine für 'Feature PROTOS-L' ist Gegenstand aktueller Arbeiten.

6 Inferenz von Relationsdeklarationen

Im Gegensatz zu TEL und PROTOS-L wo die Typen der Argumente für jede Relation und Funktion explizit deklariert werden, wurde mit dem Compiler für 'Feature PROTOS-L' der Versuch unternommen, diese Deklarationen automatisch herzuleiten. Die Typinferenz leitet zum einen die Typen von lokalen Variablen in Klauseln auf der Basis der Typdefinitionen her. Diese Inferenz wurde dahingehend erweitert, daß auch die Typen der Argumente der Relation bzw. Funktion insgesamt über alle Klauseln hinweg bestimmt werden können. So kann für den Programmausschnitt

```
person    := [alter :posint].
student   := person [matr_nr: posint].
arbeiter  := person [gehalt: posint ].

p(X) <--  gehalt(X) < 2000.
```

die Deklaration

```
rel p: arbeiter.
```

automatisch hergeleitet werden. Diese Typinferenz ist nicht mehr so einfach wie z.B. in ML, da Typhierarchien beachtet werden müssen. Aufgrund dieser Schwierigkeit wird auch z.B. für eine Erweiterung von ML um hierarchische Typen in [MMM91] die Typinferenz weggelassen. Die auftauchenden Probleme sollen kurz an einer Modifikation des vorherigen Beispiels gezeigt werden.

```
p2(X) <-- gehalt(X) < 2000.    % keine typkorrekte
p2(X) <-- X: student.          % Deklaration moeglich
```

Nach Herleitung des Argumenttyps '`arbeiter`' für die erste Klausel von '`p2`' und '`student`' für die zweite Klausel wird der Argumenttyp des Prädikats als kleinster gemeinsamer Obertyp, nämlich '`person`' bestimmt. Dies führt im nachhinein dazu, daß in

der ersten Klausel auf ein Attribut zugegriffen wird, daß für den Typ 'person' nicht definiert ist und somit als Typfehler gemeldet wird. Wegen der Vererbungsbeziehungen reicht also die erste Phase der Typinferenz allein nicht aus, um die korrekte Typisierung der Klauseln zu gewährleisten. Die Typinferenz im Feature PROTOS-L-Compiler ist korrekt in dem Sinne, daß keine falschen Relationsdeklarationen hergeleitet werden. Aber sie ist derzeit bei weitem nicht vollständig, da nicht immer eine Deklaration gefunden wird, wenn es eine korrekte gibt.

7 Objektorientierte Featurestrukturen?

In der Einleitung wurden außer den Erweiterungen von Prolog um Featurestrukturen auch objektorientierte Erweiterungen erwähnt wie z.B. Prolog++. Bezüge zu weiteren OO-Sprachen wurden in den anderen Abschnitten erwähnt. Z.T. werden auch Sprachen, die auf Featuretermen basieren, als objektorientiert bezeichnet. Auf einen wichtigen Unterschied zwischen Featuretermen mit ordnungssortierter Unifikation und objektorientierten Ansätzen soll im folgenden hingewiesen werden. Da es keine allgemein akzeptierte Definition von 'Objektorientierung' gibt, ist es allerdings müßig, definitorische Debatten zu führen. Ein recht bekannter Versuch einer Definition stammt von Wegner [Weg90], der zwischen objektbasiert und objektorientiert unterscheidet. Das entscheidende Kriterium für die Objektorientierung ist dort die Vererbung. Auch für Featuretypen ist die Vererbung ein wichtiges Konzept. Eine weitere, insbesondere für die praktische Programmierung wichtige Möglichkeit in objektorientierten Sprachen ist, Methoden in Unterklassen neu definieren zu können. Ein Smalltalk-System ohne diese Möglichkeit wäre ohne jegliche Bedeutung. In Feature-Sprachen fehlt allerdings diese Redefinition von Methoden. Featuretypen beschreiben eher die Struktur von Termen und nicht primär das Verhalten von Objekten wie in objektorientierten Sprachen oder auch in den ursprünglichen Frame-Systemen der KI [Min75].

Teilweise, wie auch in Feature PROTOS-L, fehlt zusätzlich auch die Möglichkeit, überhaupt Methoden lokal zu Featuretypen definieren zu können. Aber auch ohne Methoden zeigt sich der Unterschied, da es i.a. nicht möglich ist, Eigenschaften (Attribute) nicht-monoton zu redefinieren. Sei der Typ 'nat' Untertyp von 'int' aber disjunkt zu 'string', dann ist in der Typhierarchie

```
type t := [ f1:int , f2:string ].
type subt := t [ f1 : nat          % i.O.
                 f2 : nat ].       % Fehler
```

die Redefinition von 'f1' in 'subt' monoton, da der Typ nur eingeschränkt wird. Die Redefinition von 'f2' ist allerdings in Feature PROTOS-L nicht erlaubt, da der Typ von 'f2' ein gemeinsamer Untertyp von 'string' und 'nat' sein müßte, der aber nicht existiert. Nimmt man an, daß 'f1' nicht für ein einfaches Attribut steht, sondern für Funktionen oder allgemein Methoden vom gleichen Ergebnistyp (entsprechende Möglichkeiten gibt es z.B. in LIFE [AKMV92] oder auch F-logic [KL89]), dann ergibt sich zusätzlich die Frage, welche Methode verwendet wird, wenn bei einem Term des Typs 'subt' auf 'f1' zugegriffen wird. Für Feature-Sprachen lautet aus Gründen der einfacheren Semantik die Antwort: 'beide zusammen'. In objektorientierten Sprachen wird aber i.a. die Möglichkeit geboten, die im Obertyp definierte Methode nicht-

monoton zu redefinieren. Nimmt man eine monotone Semantik dann gibt es in dem Beispiel

```
type t := [ f1 => 1 ].
type subt := t [ f1 => 2 ].
```

keinen Term, der die Bedingungen des Typs 'subt' erfüllt. Mit einer nicht-monotonen Semantik wäre hingegen der Term 'subt{f1 => 2} ' zulässig.
Erschwerend kommt hinzu, daß Featureterme meist als intensionale Beschreibungen interpretiert werden, die im Laufe der Berechnungen immer genauer werden. Ein Featureterm unterscheidet sich damit deutlich von einem Objekt, das zu einem bestimmten Zeitpunkt mit einer eindeutigen Identifikation erzeugt wird. Die Übertragung einer nicht-monotonen Semantik auf Featureterme würde zu schwerwiegenden Komplikationen führen. Sei die Sequenz

```
X : t & F = f1(X) & X : subt
```

Teil einer Klausel mit der Bedeutung, daß zunächst die Variable 'X' auf den Typ 't' eingeschränkt wird. Danach wird der Wert des Features 'f1' gelesen und zum Schluß wird 'X' auf den Typ 'subt' eingeschränkt. Läuft die Berechnung genau in dieser Reihenfolge ab, dann hat 'F' den Wert 1, wird aber (z.B. von der Typinferenz des Compilers) die Typrestriktion 'X : **subt** ' vorgezogen, dann hat 'F' den Wert 2, obwohl sich logisch gesehen die Semantik nicht ändern sollte. In objektorientierten Sprachen bleiben die Identität und insbesondere auch die Methoden, die einem Objekt bei der Erzeugung zugeordnet werden, i.a. konstant (der interne Zustand kann sich allerdings ändern). Die obige Sequenz wäre also nicht erlaubt, da das Objekt 'X' zweimal erzeugt wird, oder aber die Sequenz würde immer 'fail' liefern, da 'X' nicht an zwei verschiedene Werte gebunden werden kann.
Aufgrund dieser semantischen Schwierigkeiten, die sich nicht leicht umgehen lassen, gilt für die Feature-Typen in Feature PROTOS-L, wie in anderen verwandten Feature-Sprachen auch, die monotone Semantik und die Sprache wird daher in diesem Sinne nicht als objektorientiert bezeichnet. Das ändert aber nichts an der Tatsache, daß die Möglichkeiten zur Datenmodellierung deutlich über die in der logischen Programmierung übliche Beschränkung auf Konstruktorterme hinausgeht.

Literatur

[AKMV92] Hassan Aït-Kaci, Richard Meyer, and Peter Van Roy. Wild LIFE - a user manual (draft). Technical report, DEC, Paris, November 1992.

[AKN86] Hassan Aït-Kaci and Roger Nasr. Login: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3:185 - 215, 1986.

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528 in Lecture Notes in Computer Science, pages 255–274. Springer Verlag, August 1991.

[Bei92] Christoph Beierle. Logic programming with typed unification and its realization on an abstract machine. *IBM Journal of Research and Development*, 36(3):375–390, May 1992.

[Car92] Bob Carpenter. *The Logic of Typed Feature Structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[CW86] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471 – 522, December 1986.

[DP91] Andrew Davison and The PARLOG Group. Design issues for logic programming-based object oriented languages. Technical report, Dept. of Computing, Imperial College, London, May 1991. 3rd revision.

[KL89] Michael Kifer and Georg Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In James Clifford, Bruce Lindsay, and David Maier, editors, *ACM SIGMOD Int. Conf. on Management of Data, Portland, Oregon*, 1989. published as SIGMOD Record Vol.18, No.2, June '89.

[McC92] Francis McCabe. *Logic and Objects*. Series in computer science. Prentice Hall International, 1992.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Min75] Marvin Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The psychology of computer vision*, pages 211–277. McGraw-Hill, 1975.

[MMM91] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 270–278, Orlando, Florida, January 1991. acm Press.

[Mös92] Hanspeter Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer Verlag, 1992.

[Seu90] Georg Seul. Logisches Programmieren mit Feature-Typen. Diplomarbeit, Universität Kaiserslautern, Fachbereich Informatik, Kaiserslautern, May 1990. DFKI Document D-90-02.

[Smo88] Gert Smolka. TEL (version 0.9), Report and User Manual. SEKI-Report SR 87-17, FB Informatik, Universität Kaiserslautern, 1988.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, 1990.

[Wei93] Sybilla Weigel. Typüberprüfung und Typinferenz in einer logischen Programmiersprache mit polymorphen Featuretypen. Diplomarbeit, Universität Karlsruhe, April 1993.

Finite Domain Constraints: eine deklarative Wissensrepräsentationsform mit effizienten Verarbeitungsverfahren

Manfred Meyer
Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
Erwin-Schrödinger-Straße
D-67663 Kaiserslautern

Deklarativität ist ein wichtiges Kriterium für Programmiersprachen und Wissensrepräsentationsformalismen geworden. Im Bereich der Programmiersprachen haben die Arbeiten von Kowalski [Kowalski, 1979] ("Program = Logic + Control") und die Entwicklung von PROLOG den Weg aufgezeigt. Im Bereich der Wissensrepräsentationsformalismen waren es insbesondere die Arbeiten an Sprachen der KL-ONE-Familie [Brachman and Schmolze, 1985], die eine deklarative oder *deskriptive* Wissensrepräsentation thematisierten.

Wesentliche Vorteile einer deklarativen Repräsentationssprache — gleichgültig, ob zur Repräsentation eines Programmes oder einer Wissensbasis — sind leichte Les- und Verstehbarkeit, Portabilität und Wartbarkeit. Leider werden diese wünschenswerten Eigenschaften in der Regel durch ineffiziente Verarbeitungsverfahren erkauft:

- Die Sprache PROLOG stellt in ihrer Reinform ("pure PROLOG") zusammen mit einem Breitensuche-Interpreter einen rein deklarativen Formalismus zur Beschreibung und Verarbeitung relationalen Wissens dar. Ein solches Abarbeitungsverfahren ist aber aufgrund seiner Ineffizienz völlig unpraktikabel. Bereits die Standardabarbeitungsstrategie von PROLOG erfordert aufgrund der Unvollständigkeit ihrer Tiefensuche vom Benutzer, daß dieser bereits bei der Formulierung seines Wissens/Programms dessen Abarbeitung berücksichtigt. Hier zeigt sich bereits der allgemeine Trade-off zwischen Deklarativität und Effizienz. Durch die Einführung von "extra-logischen" Elementen wie dem cut-Operator oder den `assert` und `retract` Operationen wird das Gleichgewicht weiter zugunsten effizienter ausführbarer aber weniger deklarativer Programme verschoben.

- Die deskriptiven Wissensrepräsentationssprachen in der Tradition von KL-ONE bieten sehr ausdrucksstarke Formalismen zur Repräsentation terminologischen Wissens in einer sehr deklarativen Form: Jedes konkrete oder abstrakte Objekt

(Konzept) wird ggfs. unter Rückgriff auf andere Konzepte in seinen Eigenschaften *definiert*. Die Situation auf der Seite der Abarbeitungsverfahren ist hier allerdings noch schlimmer: Die wichtigsten Dienste eines solchen terminologischen Systems, wie z.B. der Test auf Subsumption von Konzepten, sind für die volle KL-ONE-Sprache unentscheidbar. Auch hier muß also entweder die Sprache eingeschränkt oder mit einem unvollständigen Formalismus gelebt werden. Aber auch mit diesen Einschränkungen haben terminologische Wissensrepräsentationssprachen bisher den Weg in praktische Anwendungen kaum gefunden; hier werden überwiegend noch frame-orientierte Repräsentationssprachen verwendet.

Es ist also festzustellen, daß deklarative Repräsentationsformalismen für Relationen oder Terminologien durchaus verfügbar sind, der Gewinn an deklarativer Ausdruckskraft aber in der Regel mit einem Verlust an effizienten Verarbeitungsverfahren einhergeht.

Für die Wissensrepräsentation mit *finite domain constraints*, ist dies nicht notwendigerweise der Fall: Constraints über endlichen Domänen erlauben eine absolut deklarative Repräsentation relationalen Wissens in Form von Zusammenhängen (Relationen) zwischen Ausprägungen ausgewählter Attribute von Objekten des Gegenstandsbereichs. Zur Verarbeitung solchen Wissens, d.h. zur Lösung des damit verbundenen Constraint-Problems, sind in der Literatur verschiedene, teilweise sehr effiziente Verfahren beschrieben worden. Aber trotzdem haben diese Verfahren in praktisch relevanten Anwendungen noch kaum Verwendung gefunden. Das liegt überwiegend daran, daß die meisten Anwendungsprobleme sich nicht direkt als Constraint-Probleme formulieren lassen:

- Die bislang verfügbaren Constraint-Systeme bieten kaum Möglichkeiten, Zusammenhänge über Gruppen oder *Klassen* von Objekten auszudrücken; die Constraints müssen oftmals für jedes Individuum getrennt formuliert werden.

- In realen Anwendungen, insbesondere wenn Erfahrungswissen von Experten akquiriert und formalisiert werden muß, kommt es oft vor, daß einige Wissenseinheiten zusammen strenggenommen inkonsistent sind. Der menschliche Experte ist aber in der Lage, im konkretem Einzelfall einzelne, weniger "wichtige" Anforderungen zu verletzen, um insgesamt zu einer akzeptablen Problemlösung zu kommen. Diese Gewichtung von Constraints wird oftmals, wenn überhaupt möglich, nur über Eingriff in Kontrollstrukturen (Prioritäten usw.), realisiert. Damit wird ein großer Teil der Deklarativität aufgegeben.

- Viele Anwendungen lassen sich zwar nicht vollständig als Constraint-Probleme beschreiben, beinhalten aber Unterprobleme, die sich sehr gut als solche formulieren und lösen lassen. Für solche Anwendungen ist es entscheidend, einen Constraint-Formalismus in einem allgemeinen Wissensrepräsentationsformalismus, wie etwa einer Regelsprache, verfügbar zu machen.

Zentraler Gegenstand ist also die Untersuchung von Constraints als deklarativem Wissensrepräsentationsformalismus. Da in vielen technischen Anwendungsproblemen große, meist hierarchisch strukturierte Domänen auftreten, werden wir insbesondere untersuchen, wie sich sowohl Constraints über solchen hierarchisch strukturierten Bereichen als auch Gewichtungen von Constraints *deklarativ formulieren* und zugleich aber auch *effizient verarbeiten* lassen.

Damit sind die Anforderungen für das im Vortrag vorzustellende Constraint-System CONTAX beschrieben, dessen Eignung für den Einsatz in praktischen Anwendungen am Beispiel der Werkzeugauswahl im Bereich der Arbeitsplanung für CNC-Fertigung untersucht werden wird.

Darüberhinaus interessiert aber auch, wie *finite domain constraints* und ihre Verarbeitungsverfahren in eine logische Programmiersprache integriert werden können, um auch in größeren komplexen Anwendungen für bestimmte Teilaspekte von den Constraint-Verarbeitungsverfahren Gebrauch machen zu können. Für die solchermaßen PROLOG um FInite DOmain constraints erweiternde Sprache FIDO haben wir verschiedene Implementationstechniken untersucht. Die These dabei ist, daß durch die Integration von *finite domain constraints* das gesamte Anwendungsprogramm zugleich **deklarativer <u>und</u> effizienter** werden kann.

1 CONTAX

In CONTAX können *extensionale* (durch Aufzählen aller Tupel der Relation), prädikative (durch Angabe eines Entscheidungsprädikats) sowie zusammengesetzte (compound) Constraints definiert werden. Dafür wurde zunächst ein modifizierter AC-3-Algorithmus [Mackworth, 1977] implementiert.

Im nächsten Schritt wurde CONTAX um hierarchisch strukturierte Domänen erweitert: Die in praktischen Anwendung häufig vorkommenden hierarchischen Strukturen der Domänen, über denen die Constraints ausgedrückt werden, können nun auch direkt repräsentiert und für die Verarbeitung berücksichtigt werden. Domänen können in CONTAX entweder direkt durch Angabe der Klassen und ihrer Beziehungen (Ober-/Unterklassen, Elemente) oder auch mit Hilfe einer terminologischen Sprache (TAXON [Hanschke *et al.*, 1991] definiert werden. Durch Berücksichtigung der hierarchischen Struktur kann auch eine effizientere Propagierung erreicht werden. Hierzu wurde der HAC-Algorithmus (*hierarchical arc-consistency*, [Mackworth *et al.*, 1985]) modifiziert und erweitert.

Im Rahmen des ARC-TEC-Projekts am DFKI [Bernardi *et al.*, 1991] wurde CONTAX für das Teilproblem der Werkzeugauswahl eingesetzt. Die hierarchische Struktur der dabei zu repräsentierenden Domänen (Schneidplatten, Klemmsystem, Material usw.) konnte sehr natürlich in CONTAX modelliert werden. Es zeigte sich aber, daß für die praktische Anwendung die verschiedenen Constraints gewichtet werden müssen. So ist es z.B. allgemein sinnvoll, beim Schruppen eine quadratische Schneidplatte zu verwenden, da diese insgesamt viermal verwendet werden kann. Wenn dies aber

z.B. aus geometrischen Gründen (Eckenradius zu groß für die zu fertigende Kontur) nicht möglich ist, kann auch eine rautenförmige Schneidplatte verwendet werden. Um derartiges in praktischen Anwendungen häufig auftretendes Wissen darstellen zu können, wurde CONTAX auch noch um gewichtete Constraints erweitert. Dabei wurde allerdings großer Wert auf die Erhaltung des deklarativen Charakters gelegt: Constraints können in CONTAX mit einem von fünf Gewichten zwischen `hard` und `soft` belegt werden. Eine feinere oder gar kontinuierliche Abstufung (Gewicht z.B. zwischen 0 und 1) ist in diesem Sinne nicht sinnvoll, weil damit der Anwender direkt in den Ablauf der Propagierung eingreifen und damit Kontrollinformation codieren könnte.

Das so erweiterte Constraint-System wurde inzwischen erfolgreich für die Werkzeugauswahl eingesetzt [Tolzmann, 1992, Meyer, 1992b, Meyer, 1992a].

Die weiteren Arbeiten an CONTAX zielen jetzt darauf ab, die Integration mit dem terminologischen System TAXON auszubauen. Derzeit können Konzepthierarchien und damit die Domänen für CONTAX in TAXON definiert und dann von CONTAX verwendet werden. Oft kommt es aber auch vor, daß bei der Konzeptdefinition auch finite domain constraints benutzt werden könnten, um Abhängigkeiten zwischen einzelnen Attributen elegant zu beschreiben und bei der Klassifikation auch entsprechend zu verarbeiten. Da TAXON die Möglichkeit bietet, in der Konzeptdefinition Ausdrücke über *konkreten Bereichen* [Baader and Hanschke, 1991], z.B. den reellen Zahlen, zu verwenden, ist es sehr interessant, auch CONTAX als konkreten Bereich in TAXON einzubinden. Um die formalen Anforderungen an einen konkreten Bereich (Abgeschlossenheit unter Negation und Inkrementalität) zu erfüllen, wird CONTAX um Negation von Constraints und wegen der konjunktiven *compound constraints* auch noch um disjunktive Constraints erweitert sowie eine inkrementelle Propagierung realisiert [Frank Steinle, 1993].

Damit steht dann mit der Kopplung TAXON/CONTAX ein leistungsfähiges deklaratives Wissensrepräsentationssystem zur Verfügung: Die Begriffswelt wird mit TAXON *beschrieben*, wobei Beziehungen zwischen Attributen auch als Constraints dargestellt werden können. Bei der konkreten Problemlösung wird dann auf der Ebene der Constraints propagiert, wobei insbesondere die Ausnutzung der hierarchischen Domänenstruktur sowie die Gewichtung einzelner Constraints auch die Bearbeitung realistischer Anwendungen ermöglichen.

2 FIDO

Neben diesen Ansätzen wurde wie eingangs angesprochen auch die Einbettung von Constraint-Propagierungs-Verfahren in eine logische Programmiersprache wie PROLOG untersucht. Grundlage dieser Integration ist die Tatsache, daß Constraints prinzipiell wie gewöhnliche Relationen dargestellt und auch behandelt werden können. Faßt man sie aber als Constraints auf und unterwirft sie einer erweiterten Auswahlstrategie wie *forward checking* oder *looking-ahead*, so lassen sich dadurch sehr früh

große Teile des sonst von PROLOG zu bearbeitenden Suchraums abschneiden, die nicht zu einer Lösung beitragen können.

Während bei der Anwendung von *forward-checking* jeweils alle bis auf eine an einem Constraint beteiligten Variablen instanziiert sein müssen um eine Einschränkung für die verbleibende Variable zu erhalten, kann *looking-ahead* jederzeit auf ein Constraint angewendet werden: In jedem looking-ahead-Schritt wird für jeden Wert in der Domäne einer jeden an dem Constraint beteiligten Variable überprüft, ob sich in den Domänen der anderen betroffenen Variablen Werte finden lassen, so daß das Constraint erfüllt wird. Somit entspricht *looking-ahead* für ein Constraint mit k Variablen genau der Berechnung einer *k-konsistenten* [Mackworth, 1977] Werte-Belegung.

Jede dieser Konsistenztechniken hat ihre spezifischen Vor- und Nachteile:

- Mit *forward-checking* können die Domänen — und damit der Suchraum für eine konsistente Lösung — in der Regel stark eingeschränkt werden, der Berechnungsaufwand ist relativ gering, und das Verfahren ist korrekt und vollständig [van Hentenryck, 1989]. Allerdings kann forward-checking durch die strenge Anwendungsbedingung, daß alle bis auf eine Variable instanziiert sein müssen, oftmals erst sehr spät in einer Anwendung eingesetzt werden.

- Mit *looking-ahead* kann dagegen allerdings schon sehr früh eine Domänen- und Suchraumeinschränkung erreicht werden. Für viele Anwendungen ist eine vollständige Verwendung von looking-ahead, also die Durchführung eines looking-ahead-Schritts bei jeder Änderung der Domäne einer der betroffenen Variablen, viel zu aufwendig. Es eignet sich also nur für einzelne Spezialfälle und ist noch dazu theoretisch unvollständig.

Die Sprache FIDO stellt also normales PROLOG zusammen mit den Konsistenztechniken *forward-checking* und *looking-ahead* über endlichen Domänen zur Verfügung. Dazu wurden verschiedene Implementationstechniken untersucht:

- ein seinerseits wiederum in PROLOG implementierter Meta-Interpreter (FIDO-I, [Schrödl, 1991]),

- ein horizontaler Compiler (FIDO-II, [Müller, 1991]), der FIDO-Programme nach PROLOG übersetzt und dabei insbesondere den Coroutining-Mechanismus (`delay`) ausnutzt, um die ausgezeichneten Constraints gesondert abzuarbeiten,

- sowie eine erweiterte abstrakte Maschine (FIDO-III, [Hein, 1992]) und einen optimierenden Compiler für FIDO-Programme in Code für diese Maschine [Stein, Forthcoming 1993].

Auch hier hat sich bei der Erprobung dieses Ansatzes an praktischen Anwendungen gezeigt, daß die bekannten Konsistenztechniken in der Praxis nicht ausreichen: *Forward checking* bietet nur eine geringe Einschränkung des Suchraums mit allerdings geringem Aufwand, *looking-ahead* dagegen kann zwar zu großen Einsparungen

an Berechnungsschritten führen, ist allerdings sehr aufwendig. Für praktische Anwendungen wurde deshalb eine neue Konsistenztechnik *weak looking-ahead (WLA)* [Meyer and Müller, 1993a] als Kombination von forward checking und looking-ahead entwickelt:

Weak looking-ahead als Kombination von *forward-checking* und *looking-ahead* verwendet daher nur einen looking-ahead-Schritt, der z.B. initiale Wertebeschränkungen durch das Constraint propagiert. Danach wird ein *weak-looking-ahead* Constraint nur noch mit *forward-checking* behandelt [Meyer and Müller, 1993b]. Damit kann mit wesentlich geringerem Aufwand als mit looking-ahead in der Regel eine größere Einschränkung des Suchraums als mit forward checking erzielt werden. Außerdem kann gezeigt werden, daß die um weak-looking-ahead erweiterte SLD-Resolution (*SLDW-Resolution*) auch wiederum korrekt und vollständig ist [Meyer and Müller, 1993a].

Literatur

[Baader and Hanschke, 1991] F. Baader and P. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. Research Report RR-91-10, DFKI GmbH, 1991. Includes algorithms.

[Bernardi *et al.*, 1991] Ansgar Bernardi, Harold Boley, Knut Hinkelmann, Philipp Hanschke, Christoph Klauck, Otto Kühn, Ralf Legleitner, Manfred Meyer, Michael M. Richter, Gabi Schmidt, Franz Schmalhofer, and Walter Sommer. ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge. In *Expert Systems and their Applications: Tools, Techniques and Methods*, Avignon, France, 1991. Also available as Research Report RR-91-27, DFKI GmbH.

[Boley and Richter, 1991] Harold Boley and Michael M. Richter, editors. *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, Berlin, Heidelberg, 1991.

[Brachman and Schmolze, 1985] J. Brachman, R. and G. Schmolze, J. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[Frank Steinle, 1993] Frank Steinle. HAMLET: Erweiterung eines Constraint-Systems um Negation und Disjunktion und dessen Anbindung an eine Konzeptbeschreibungssprache. Projektarbeit, March 1993. In German.

[Hanschke *et al.*, 1991] Philipp Hanschke, Andreas Abecker, and Dennis Drollinger. TAXON: A Concept Language with Concrete Domains. In *[Boley and Richter, 1991]*, pages 411–413, 1991.

[Hein, 1992] H.-G. Hein. Consistency Techniques in WAM-based Architectures. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, October 1992.

[Kowalski, 1979] Robert Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. Elsevier North Holland, 1979.

[Mackworth *et al.*, 1985] A. Mackworth, J. Mulder, and W. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[Mackworth, 1977] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Meyer and Müller, 1993a] Manfred Meyer and Jörg Müller. Combining Forward-Checking and Looking-Ahead in Constraint Logic Programming. In Jianping Wu, Jin Yang, Yamin Li, and Wen Gao, editors, *Towards the future: Proceedings of the Third International Conference for Young Computer Scientists (ICYCS'93), Beijing, China*, pages 460–463. Tsinghua University Press, Beijing, China, July 1993.

[Meyer and Müller, 1993b] Manfred Meyer and Jörg Müller. Using Weak Looking-Ahead for Lathe-Tool Selection in a CIM Environment. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93), Edinburgh, Scotland*, pages 26–35. Gordon and Breach Science Publishers, June 1993. This paper received the **IEA/AIE-93 Best Paper Award**. Also available as DFKI Research Report RR-93-22, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany.

[Meyer, 1992a] Manfred Meyer. Hierarchical Constraint Satisfaction and its Application in Computer-Aided Production Planning. In *Expert Systems 92, Cambridge, U.K.*, December 1992.

[Meyer, 1992b] Manfred Meyer. Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment. In *Fifth International Symposium on Artificial Intelligence (ISAI), Cancun, Mexico*, pages 167–177. AAAI Press, December 1992.

[Müller, 1991] Jörg Müller. Design and implementation of a finite domain constraint logic programming system based on prolog with coroutining. Document D-91-02, DFKI, DFKI GmbH, Postfach 2080, 67608 Kaiserslautern, Germany, November 1991.

[Schrödl, 1991] S. Schrödl. FIDO: Ein Constraint-Logic-Programming-System mit Finite Domains. ARC-TEC Discussion Paper 91-05, DFKI GmbH, Postfach 2080, D-6750 Kaiserslautern, June 1991.

[Stein, Forthcoming 1993] W. Stein. Nutzung globaler Analysetechniken in einem optimierenden Compiler für die Constraint-Logic-Programming-Sprache FIDO. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, Forthcoming 1993.

[Tolzmann, 1992] E. Tolzmann. Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX, June 1992.

[van Hentenryck, 1989] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Ma., 1989.

Ein erweitertes CLP–Schema für eine hybride Wissensverarbeitung

Holger Wache, Panagiotis Tsarchopoulos
DFKI
Postfach 2080
67608 Kaiserslautern
Deutschland
e-mail:{*wache, tsarcho*}@dfki.uni-kl.de

16. August 1993

Zusammenfassung

In diesem Papier stellen wir einen Ansatz für ein hybrides Wissensrepräsentations- und Inferenzsystem vor. Sowohl Algebraische Constraints, wie sie in CLP($\mathbb{R}$) Systemen formuliert werden können, als auch Constraints über endlichen Wertemengen können als Konkrete Bereiche in ein terminologisches System integriert werden. Terminologische Logik und Constraints werden unter Zuhilfenahme des erweiterten CLP–Schemas in Hornklauseln eingebettet. Der Ansatz besitzt eine wohl–fundierte Semantik und das Potential für einen Effizienzgewinn, da auf spezialisierte Algorithmen zurückgegriffen werden kann. Die Integration wird an einem Beispiel aus der Konfigurierung verdeutlicht.

1 Motivation

Das Expertenwissen einer Domäne läßt sich im allgemeinen nicht in einem uniformen Formalismus (z.B. Regeln) adäquat repräsentieren. Dies gilt insbesondere für komplexe Anwendungen, wie sie zum Beispiel im Bereich der Konfigurierung auftreten. In einem uniformen Formalismus werden unterschiedliche Wissensarten wie zum Beispiel Wissen über Objekte, Wissen über deren (strukturelle und funktionale) Abhängigkeiten oder Reihenfolgewissen miteinander vermischt. Um dies zu vermeiden, sollten mehrere Repräsentationsformalismen zur Verfügung gestellt werden, die die Beschreibung von unterschiedlichen Wissensarten deklarativ und adäquat erlauben. Aus diesem Grunde wurden verschiedene hybride Wissensrepräsentationssysteme wie das CLP–Schema [JL87] oder CoLab [BHHM93] entwickelt. Die Systeme erlauben die Verwendung von verschiedenen Formalismen zur Repräsentation unterschiedlicher Teilaspekte eines komplexen Problems.

Ein wichtiger Punkt bei hybriden Repräsentationssystemen ist die effiziente Verarbeitung des dargestellten Wissens. Prinzipiell kann die Inferenz auf zwei Arten erfolgen. Zum einen kann man versuchen, die verschiedenen Repräsentationsformen mittels Kompilationstechniken auf einen einzigen Formalismus abzubilden, sodaß nur eine Inferenzmethode benötigt wird. Zum anderen kann man für jeden Formalismus eine separate Inferenzmethode verwenden, wobei die Methoden in irgendeiner Form miteinander gekoppelt werden müssen. Dieser Ansatz hat den Vorteil, daß die einzelnen Inferenzmethoden weitgehend erhalten bleiben und auf effiziente Algorithmen, die bisher

separat für die Methoden entwickelt wurden, zurückgegriffen werden kann. Dieses Potential für einen Effizienzgewinn geht im ersten Ansatz verloren.

Die Ansätze zur Kopplung verschiedener Inferenzmethoden lassen sich tendenziell zwischen zwei Polen einordnen:

- lose Integration über klar definierte Schnittstellen
- eine enge, verzahnte Integration der Inferenzmethoden mit einer klaren Semantik

Beispiele für eine lose Integration sind BABYLON [CdPV89] oder COLAB [BHHM93]. Das CLP-Schema [JL87] entspricht konzeptionell eher einer engen, verzahnten Integration von Inferenzmethoden. Es hat sich gezeigt, daß in beiden Ansätzen ein semantisch klares Konzept für die Integration der Methoden notwendig ist!

In diesem Papier werden wir einen Ansatz vorstellen, in dem terminologische Logiken zur Repräsentation der Begriffe (z.B. Objekte) einer Domäne, algebraische Constraints und Constraints über endlichen Wertebereichen in Hornklauseln eingebettet werden. Die Inferenzmethoden sind auf der Basis einer logisch fundierten Semantik miteinander gekoppelt. An einem einfachen Beispiel, der Konfigurierung eines PC's, wollen wir veranschaulichen, wie die unterschiedlichen Wissensrepräsentationsformalismen verwendet werden können.

2 Wissensrepräsentation auf der Basis terminologischer Logiken

Terminologische Systeme [BS85, Neb90] erlauben die Formulierung von Begriffen (Objekten, *Konzepten*) einer Domäne. Die Beschreibung der Konzepte erfolgt explizit über ihre strukturellen Beziehungen zu anderen Konzepten. Die terminologischen Logiken lassen sich auf eine entscheidbare Teilmenge der Prädikatenlogik zurückführen. Sie zeichnen sich daher gegenüber anderen objektorientierten Repräsentationssystemen durch eine klare Semantik aus.

Terminologische Systeme bestehen in der Regel aus zwei Teilen: einer T–Box und einer A–Box. In der T–Box wird das terminologische Wissen abgelegt. Das heißt, in der T–Box werden die Konzepte einer Domäne definiert. Dabei werden, ausgehend von einfachen Konzepten, mittels *Konstruktionsoperatoren* wie $C \sqcap D$ (AND), $C \sqcup D$ (OR) und $\neg C$ (NOT) neue, komplexere Konzepte gebildet (z.B. $E := C \sqcup D$). Konzepte werden über zweistellige Relationen (Rollen R oder Attribute (Features) F) mittels $\exists\ R.C$ (für features: $F.C$) und $\forall\ R.C$ miteinander in Beziehung gesetzt.

Terminologische Systeme bieten den Vorteil, daß aus den Konzeptbeschreibungen mit vollständigen und korrekten Algorithmen die Objekttaxonomie (Subsumptionshierarchie) automatisch generiert werden kann. Diese Dienstleistung kann zur Wissensakquisition und -validierung herangezogen werden.

Neben der Definition in der T–Box lassen sich Begriffe in der A–Box eines terminologischen Systems instanziieren (assertionales Wissen). Die Instanzen (*Individuen*) werden in Form von *assertionalen Ausdrücken* in die A–Box eingetragen. Assertionale Ausdrücke umfassen Zusicherungen über die Zugehörigkeit von Individuen zu Konzepten $(i : C)$[1] und über Rollen– oder Attributfüller $(R(i, j))$,

[1] Zur besseren Unterscheidung werden wir Individuen der A–Box mit einem kleinen und Konzepte der T–Box mit einem großen Anfangsbuchstaben schreiben.

ebenso wie Gleichheitsaussagen über Individuen ($i = j$). Ein Individuum läßt sich als Ausprägung einer Klasse (Konzept) oder — im Sinne der Sortenlogik — als ein Element einer Sorte (= Konzept) verstehen.

Die A-Box stellt verschiedene Inferenzmethoden zur Verfügung (vgl. [Hol90]):

- *Konsistenztest*
  Die Konsistenz des in der A-Box enthaltenen assertionalen Wissens bezogen auf das terminologische Wissen in der T-Box kann überprüft werden.
- *Realisierung*
  Die Realisierung klassifiziert ein Individuum aufgrund des assertionalen Wissens als eine Menge von Konzepten zugeordnet. Im Gegensatz zu anderen objektorientierten Systemen, wo Instanzen Klassen fest zugeordnet werden, kann sich die Zuordnung von Individuen zu Konzepten in einem laufenden Inferenzprozeß dynamisch ändern. In [BHHM93] werden zwei Arten von Realisierung unterschieden:
  - *starke Realisierung*
    Die starke Realisierung eines Individuums ist eine (minimale) Menge von bezüglich der Subsumptionshierarchie speziellsten Konzepten, zu denen das Individuum unter einer Open-World-Semantik *sicher* zugeordnet werden kann. Mit anderen Worten, das Individuum *muß* den Konzepten zugeordnet werden.
  - *schwache Realisierung*
    Die schwache Realisierung liefert eine (minimale) Menge von bezüglich der Subsumptionshierarchie speziellsten Konzepten zurück, für die eine Zuordnung des Individuums als *konsistent angenommen* werden kann. Ein Individuum *kann* den Konzepten zugeordnet werden. In der Regel besteht die Menge aus terminalen Konzepten.

Die Inferenzmethoden können auf einen grundlegenden Mechanismus, den Konsistenztest der A-Box, abgebildet werden [Hol90]. Der Konsistenztest wird bei der Einbettung in Hornklauseln als "Unifikationsmethode" benötigt. Auf die Realisierung werden wir bei der Diskussion des Beispiels in Abschnitt 5 zurückkommen.

3 Konkrete Bereiche

Terminologische Systeme besitzen eine wohl-definierte Semantik und ein hohes Maß an Deklarativität. Für reale Anwendungen aber ist ihre Ausdrucksmächtigkeit in der Regel zu schwach. Zum Beispiel läßt sich ein Erwachsener nicht ohne weiteres als ein Mensch mit einem Alter *größer als 18* Jahren in der T-Box modellieren, da keine Vergleichsprädikate über Zahlen definiert werden können. Dieser Schwachpunkt macht sich besonders in technischen Domänen bemerkbar.

Zur Beseitigung dieses Defizits erweiterten Baader und Hanschke die terminologische Logik um das Konzept der *Konkreten Bereiche* [BH91]. In einem Konkreten Bereich können spezielle Inferenzmethoden, die in das terminologische System integriert werden sollen, realisiert werden. Die Integration erfolgt über *Konkrete Prädikate*, die im terminologischen System gewissermaßen als "Built-In's" verwendet werden können. Ein Konkretes Prädikat P kann in der T-Box in der Form

$\exists\ P[p_1,\ldots,p_n]$ oder $\forall\ P[p_1,\ldots,p_n]$ verwendet werden, wobei p_i eine Rollen- und/oder Attributkette (z.B. $F_1 \circ R_1 \circ F_2$) bezeichnet. In die A-Box kann ein Konkretes Prädikat P als $P[i_1,\ldots,i_n]$ über den Individuen $i_1,\ldots,i_n$ instanziiert werden.

Über die Struktur und Beschaffenheit des Konkreten Bereichs werden aus der Sicht der terminologischen Logik nur wenige Aussagen gemacht. Es wird im wesentlichen für den Konsistenztest der A-Box ein (effizientes) Entscheidungsverfahren gefordert, das eine Konjunktion von (instanziierten) Konkreten Prädikaten auf Konsistenz hin überprüft.

Beispielsweise läßt sich ein algebraischer Constraintsolver über reelle Zahlen als Konkreter Bereich einbinden. Mittels des konkreten Prädikats > läßt sich z.B. ein Erwachsener als $Mensch \sqcap (\exists\ Alter > 18)$ beschreiben.[2] Es können aber auch Methoden verwendet werden, die eigenständiges Wissen benutzen. Datenbanken oder Constraints über endlichen Wertebereichen können über die Schnittstelle der Konkreten Bereiche angeschlossen werden, sofern sie die Restriktionen der Konkreten Bereiche erfüllen.

Konkrete Bereiche definieren eine schmale und klare Schnittstelle, für die eine wohl-fundierte Semantik angeben werden kann [BH91]. Durch die Einbindung spezieller Repräsentationsformalismen gewinnt die Konzeptsprache an Ausdrucksmächtigkeit. Dabei bleiben die Inferenzmethoden weitgehend erhalten, sodaß durch den Einsatz von spezialisierten Algorithmen in den Konkreten Bereichen ein Effizienzgewinn möglich ist.

Das Konzept der Konkreten Bereiche ähnelt dem CLP-Ansatz. Im CLP-Schema [JL86] wird eine Schnittstelle definiert, über die ein (beliebiger) Constraintsolver in die logische Programmierung wie z.B. *PROLOG* integriert werden kann. Die gleiche Funktion übernehmen die Konkreten Bereiche für das terminologische System.

4 Einbettung in die logische Programmierung

Die A-Box terminologischer Systeme ist in der Regel nur in beschränktem Maße dafür ausgelegt, das assertionale Wissen (z.B. im Verlauf eines Suchprozesses) dynamisch zu erweitern. Die Inferenzmethoden der A-Box können im wesentlichen nur die Konsistenz des assertionalen Wissens überprüfen. Viele Anwendungen (z.B. in der Konfigurierung) zeichnen sich jedoch durch einen inkrementellen Lösungsgenerierungsprozeß aus.[3] Es ist daher notwendig, terminologische Systeme in eine größere Umgebung einzubetten.

In [AH93] wird die Integration eines terminologischen Systems in die funktionsfreie Hornlogik vorgeschlagen. Hornklauseln fungieren als Trägersprache, in die eine terminologische Komponente eingebettet ist. Diesen Ansatz wollen wir hier aufgreifen.

Analog zum CLP-Schema, wo Hornklauseln um die Möglichkeit zur Formulierung von Constraints erweitert werden, werden hier Aussagen über das assertionale Wissen einer terminologischen Komponente in Hornklauseln erlaubt. Im Rumpf einer Hornklausel sind neben Prädikaten eine Menge von assertionalen Ausdrücken zugelassen. Assertionale Ausdrücke können Zusicherungen an eine A-Box ($i : C,\ R(i,j),\ i = j$) oder Konkrete Prädikate ($P[i,j,k]$) sein.

[2] Zur besseren Lesbarkeit verwenden wir algebraische Constraints in der Infixnotation.

[3] Wir gehen hier von der (sinnvollen) Annahme aus, daß die Lösung in der A-Box repräsentiert werden soll.

Über die Schnittstelle der Konkreten Bereiche lassen sich Constraints als Konkrete Prädikate in das terminologische System einbinden (siehe Abschnitt 3). Da Konkrete Prädikate in den Hornklauseln verwendet werden können, sind neben dem eigentlichen terminologischen System auch die Constraintsolver in die Hornlogik integriert.

Es läßt sich zeigen, daß der Ansatz eine Instanz des allgemeinen Höhfeld/Smolka-Schemas [HS88] ist. Das Höhfeld/Smolka-Schema erweitert das CLP-Schema insofern, als schwächere Annahmen über das eingebundene Constraintssystem gemacht werden. Die assertionalen Ausdrücke in einer Hornklausel können in diesem Sinne als Constraints verstanden werden. Eine deklarative Semantik für diesen Integrationsansatz läßt sich somit direkt aus der Fixpunktsemantik des Schemas ableiten. Aufgrund des Höhfeld/Smolka-Schemas basiert also die Integration von terminologischer Logik in Hornklauseln auf einer klaren Semantik.
Eine operationale Semantik läßt sich durch die Angabe eine Interpreterss beschreiben. An dieser Stelle wollen wir den Interpreter nur grob skizzieren:[4]

1. Von einer Menge noch zu beweisender Ziele (*Goalstack*) wird ein Ziel ausgewählt.
2. Aus der Regelbasis wird eine Hornklausel zum Beweisen des Ziels selektiert. Die Hornklausel wird entsprechend der Variablenbelegung des gewählten Ziels umbenannt.
3. Die Hornklauseln wird abgearbeitet, indem die im Rumpf enthaltenen assertionalen Ausdrücke einfach in die A-Box eingetragen und die Prädikate als noch zu beweisende (Unter-) Ziele auf den Goalstack geschrieben werden.
4. Wenn der abschließende Konsistenztest eine Inkonsistenz der A-Box entdeckt oder für ein Ziel keine Hornklausel gefunden werden kann, wird der Backtrackingmechanismus ausgelöst. Ansonsten fährt der Interpreter mit dem Beweis des nächsten Zieles fort (Schritt 1).

Der Algorithmus terminiert, wenn der Goalstack vollständig abgearbeitet wurde. Als Ergebnis wird der Inhalt der konsistenten A-Box zurückgeliefert.

Mit der Integration des terminologischen Systems und der Constraints (im Konkreten Bereich des terminologischen Systems) können Einschränkungen an eine mögliche Lösung formuliert werden. Während des Inferenzprozesses können aufgrund der Einschränkungen inkonsistente (Teil-)Lösungen früher erkannt und dadurch der Suchaufwand reduziert werden. Dieselbe Idee liegt auch dem CLP-Ansatz [JL87] und dem Ansatz zur Verwendung von Sorten in der logischen Programmierung zugrunde. Da die einzelnen Inferenzmethoden für die Repräsentationsarten weitgehend erhalten bleiben, kann auf effiziente, da spezialisierte Verfahren zurückgegriffen werden. Die semantisch klare Integration von terminologischen Logiken und Constraints bietet somit das Potential für eine Effizienzsteigerung.

5 Ein Beispiel

Als Beispiel wollen wir die Konfigurierung eines PC's behandeln. Ein PC soll in unserem, vereinfachten Beispiel aus einer Menge von *Board*'s (*Komponenten*) zusammengestellt werden, die

[4] In [AH93] wird der Interpreter genauer ausgeführt.

in ein Gehäuse gesteckt werden. *Board*'s können Motherboards, Festplatte, Diskettenstation oder Streamer sein. Die Boards kommunizieren über ein Bussystem, zu dem alle Boards des PC's kompatibel sein müssen (interne Abhängigkeit). Die Konfigurierungsaufgabe besteht darin, die geeigneten Boards aufgrund der Anforderungen des Benutzers (z.B. langsamer oder schneller PC oder Größe der Festplatte) unter Berücksichtigung der internen Abhängigkeiten zu bestimmen.

Die betrachteten Objekte sind die Komponenten eines PC's. Sie werden daher durch Konzepte in der T-Box des terminologischen Systems modelliert. In unserem Beispiel wollen wir uns auf die Modellierung von Motherboards und Festplatten beschränken:[5]

$$
\begin{aligned}
KindOfCPUs &:= i386 \sqcup i486 \\
KindOfBustypes &:= AT \sqcup LocalBus \sqcup EISA \\
\\
Board &:= Bustype.KindOfBustypes \\
\\
Motherboard &:= Board \sqcap Speed.(Slow \sqcup Fast) \sqcap CPU.KindOfCPUs \\
Motherboard386 &:= Motherboard \sqcap Speed.Slow \sqcap Bustype.AT \sqcap CPU.i386 \\
Motherboard486 &:= Motherboard \sqcap Speed.Fast \sqcap Bustype.EISA \sqcap CPU.i486
\end{aligned}
$$

Board's werden allgemein als Konzept eingeführt, die auf das Bussystem des PC's zugreifen. Das (abstrakte) Konzept *Motherboard* wird durch drei Attribute *Speed*, *Bustype* und *CPU* beschrieben. Als "Werte" für die Attribute kommen nur Elemente einer Menge von (primitiven) Konzepten in Frage (Value Restriction). *Motherboard386* und *Motherboard486* sind konkrete Motherboards (Komponenten).

Neben den *Motherboard*'s sollen Festplatten repräsentiert werden:

$$
\begin{aligned}
Harddisk &:= Board \sqcap (\exists\, Size > 0) \\
Harddisk20MBAT &:= Harddisk \sqcap (\exists\, Size = 20) \sqcap Bustype.LocalBus \\
Harddisk20MBEISA &:= Harddisk \sqcap (\exists\, Size = 20) \sqcap Bustype.EISA \\
Harddisk40MBAT &:= Harddisk \sqcap (\exists\, Size = 40) \sqcap Bustype.AT
\end{aligned}
$$

Festplatten sind durch die Attribute *Size* und *Bustype* charakterisiert. Das Attribut *Bustype* wird von dem Konzept *Board* vererbt und in den möglichen Attributausprägungen eingeschränkt (Value Restriktion). Bei der Definition der Festplatten wurden die Konkreten Prädikate > und = verwendet, die auf den algebraischen Constraintsolver im Konkreten Bereich abgebildet werden.

Ein PC läßt sich folgendermaßen darstellen:

$$BusCompatible \Leftrightarrow \{(AT,\ AT);\ (AT,\ localBus);\ (EISA,\ EISA)\ \ldots\}$$

$$
\begin{aligned}
PC := \ & (\forall\, HasBoards.Board) \sqcap \\
& (\forall\, BusCompatible[(HasBoards \circ Bustype), \\
& \qquad (HasBoards \circ Bustype)])
\end{aligned}
$$

[5] Rollen und Attribute können anhand der Restriktoren unterschieden werden. Attribute sind in unserem terminologischen System immer existanziell quantifiziert; deshalb wurde der Quantor weggelassen.

Der PC besteht aus einer Menge von Boards (Rolle *HasBoards*). Die (paarweise) Verträglichkeit der Boards untereinander wird durch das Constraint *BusCompatible* gewährleistet, welches als Konkretes Prädikat in der Beschreibung des PC's verwendet wird. Durch die Darstellung von Abhängigkeiten zwischen Attributen als Constraints können Äquivalenzklassen über den Attributwerten definiert werden. Allgemein lassen sich mittels Constraints Korrelationen über Attributausprägungen formulieren.

Neben der Modellierung der Komponenten im terminologischen System muß das Problemlösungswissen zur Generierung einer Konfiguration repräsentiert werden. Allgemein kann man zwischen Methoden zur Auswahl, zum Parametrieren und zum Zusammenfügen von Komponenten unterscheiden [Hei93]. Die Auswahl von Komponenten kann durch die (schwache) Realisierung der A-Box unterstützt werden, während das Parametrieren und Zusammenfügen in dem vorgestellten Schema auf der Ebene der Hornklauseln repräsentiert werden muß.

```
configure                 :-  ?X:PC,
                              config-motherboard(?X),
                              config-harddisk(?X).

config-motherboard(?PC)   :-  ?MB: Motherboard, HasBoards(?PC, ?MB),
                              parameter(speed, ?MB-SPEED),
                              Speed(?MB, ?MB-SPEED),
                              refine(?MB).
```

Zur Konfigurierung des PC's müssen das Motherboard **config-motherboard** und die Festplatte **config-harddisk** bestimmt werden. Zur Bestimmung des Motherboards wird ein Motherboard *?MB* in die A-Box instanziert und als Rollenfüller der Relation *hasBoards* des PC's *?PC* eingetragen. Die Geschwindigkeit des Boards wird durch den Benutzer festgelegt, indem er zum Beispiel das Faktum **parameter**(speed, slow) in die Faktenbasis einträgt. Die Geschwindigkeitsanforderung wird als Attributfüller des Features *Speed* in die A-Box eingetragen. Auf das Prädikat **refine** werden wir später zurückkommen.

```
config-harddisk(?PC)  :-  ?BD: Harddisk, HasBoards(?PC, ?BD), Size(?BD, ?BD-SIZE),
                          parameter(memory-size, ?MEMORY-SIZE),
                          ?MEMORY-SIZE <= ?BD-SIZE,
                          refine(?BD).

config-harddisk(?PC)  :-  ?BD1: Harddisk, HasBoards(?PC, ?BD1), Size(?BD1, ?BD1-SIZE),
                          ?BD2: Harddisk, HasBoards(?PC, ?BD2), Size(?BD2, ?BD2-SIZE),
                          parameter(memory-size, ?MEMORY-SIZE),
                          ?MEMORY-SIZE <= ?BD1-SIZE + ?BD2-SIZE,
                          refine(?BD1), refine(?BD2).
```

Bei der Formulierung des **config-harddisk** Prädikats wurde implizit heuristisches Vorgehenswissen verwendet. Zuerst soll versucht werden, mit *einer* Festplatte die Anforderung des Benutzers zu erfüllen. Mit dem algebraischen Constraint ?MEMORY-SIZE <= ?BD-SIZE stellt den Test dar. Wenn die Anforderung nicht erfüllt werden kann, dann kann noch eine zweite Festplatte hinzugenommen werden.

Der Nachteil dieser Modellierung ist, daß das Vorgehenswissen und die deklarative Formulierung eines Konfigurierungsschritts miteinander vermischt werden. Das Schema sollte daher um eine Komponente erweitert werden, in der Vorgehenswissen explizit dargestellt werden kann (z.B. Meta-Regeln). Dieses Wissen kann dann z.B. zur Auswahl der Regeln, die die einzelnen Konfigurierungsschritte repräsentieren, herangezogen werden.

Das Prädikat **refine** soll die Verfeinerung eines Individuums bezüglich der Subsumptionshierarchie in der T-Box herbeiführen. Mit der schwachen Realisierung kann eine Menge von terminalen Konzepten (Komponenten) ermittelt werden. Das Individuum soll dann einem Konzept aus dieser Menge hypothetisch zugeordnet werden.

Die Funktionalität des **refine**-Prädikats läßt sich zum einen als "*Built-In*-Prädikat" direkt zur Verfügung stellen. Das Prädikat sollte sich aber auch wie folgt repräsentieren lassen:

```
refine(?X)                   :-  weakly-realize(?X, ?LIST),
                                 assume(?X, ?LIST).

assume(?X, [?CONC | ?REST])  :-  ?X:?CONC.
assume(?X, [?CONC | ?REST])  :-  assume(?X, ?REST]).
```

Diese Modellierung ist im vorgestellten Ansatz nicht zulässig, da im Höhfeld/Smolka-Schema funktionsfreie Prädikate gefordert werden. Für einen praktischen Einsatz sollte aber die Einschränkung fallengelassen und zumindest Listen erlaubt werden. Das Konstrukt *?X:?CONC* ist ebenfalls nicht zulässig, da Variablen in der Hornlogik ausschließlich auf Individuen in der A-Box und nicht auf Konzeptnamen abgebildet werden.

6 Zusammenfassung und Ausblick

Bisherige Systeme zur hybriden Wissensrepräsentation und -verarbeitung (z.B. [CdPV89, BHHM93]) lassen eine klare Semantik der Schnittstellen und/oder eine effiziente Integration vermissen.

Das Höhfeld/Smolka-Schema erlaubt die Einbettung von terminologischen Logiken in die Hornlogik. Algebraische Constraints und Constraints über endlichen Wertebereichen können über die Konkreten Bereiche in die terminologische Komponente integriert werden. Beide Integrationsansätze basieren auf einer wohl-fundierten Semantik.

Im vorgestellten Ansatz bleiben die verschiedenen Inferenzmethoden weitgehend erhalten, sodaß spezialisierte Algorithmen eingesetzt werden können. Die Informationen im terminologischen System und die Constraints können zum frühen Erkennen von Inkonsistenzen herangezogen werden und somit den Suchraum beschränken. Spezialisierte Algorithmen und das frühzeitiges Beschränken des Suchraums machen eine Effizienzsteigerung möglich.

In dem Beispiel wurden Erweiterungen des Ansatzes vorgestellt, die für einen praktischen Einsatz wünschenswert wären. Die Einschränkung auf funktionsfreie Prädikate in der Hornlogik sollte fallengelassen werden. Das Vorgehenswissen sollte nicht implizit, sondern explizit und deklarativ dargestellt werden können (z.B. Meta-Regeln).

In vielen Anwendungen ist eine (ausschließlich) Anfragen-orientierte Inferenz nicht adäquat. Wir wollen daher untersuchen, wie der Ansatz z.B. um eine Daten-getriebene Inferenz (Vorwärtsregeln) effizient erweitert werden kann (vgl. [Han93]).

7 Danksagungen

Für die Unterstützung möchten wir dem gesamten ARC-TEC Team am DFKI danken. Hier sind insbesondere Andreas Abecker und Philipp Hanschke für ihre tatkräftige Unterstützung hervorzuheben.

Literatur

[AH93] A. Abecker and Ph. Hanschke. TaxLog: A Flexible Architecture for Logic Programming with Structured Types and Constraints. In M. Meyer, editor, *Proc. Workshop Constraint Processing, CSAM'93.* DFKI, 1993. Research Report RR-93-393s4s3s.

[BH91] F. Baader and Ph. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[BHHM93] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A Hybrid Knowledge Representation and Compilation Laboratory. Research Report RR-93-08, DFKI, Postfach 2080, 67608 Kaiserslautern, January 1993.

[BS85] J. Brachman, R. and G. Schmolze, J. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–216, 1985.

[CdPV89] Thomas Christaller, Franco di Primio, and Angi Voss, editors. *Die KI-Werkbank BABYLON: eine offene und portable Entwicklungsumgebung für Expertensysteme.* Künstliche Intelligenz. Addison–Wesley (Deutschland) GmbH, 1989.

[Han93] Ph. Hanschke. *A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning.* Dissertation, Universität Kaiserslautern, Postfach 2080, 67608 Kaiserslautern, (Germany), 1993.

[Hei93] Michael Heinrich. Ressourcenorientiertes Konfigurieren. *KI*, (1/93):11–16, 1993.

[Hol90] B. Hollunder. Hybrid Inferences in KL-ONE-Based Knowledge Representation Systems. In GWAI-90*; 14th German Workshop on Artificial Intelligence*, volume 251 of *Informatik-Fachberichte*, pages 38–47. Springer, 1990.

[HS88] Markus Höhfeld and Gert Smolka. Definite Relations over Constraint Languages. Lilog-Report 53, IBM Deutschland, October 1988.

[JL86] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Monash University, Department of Computer Science, June 1986.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, pages 111–119. ACM, January 1987.

[Neb90] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *LNAI.* Springer, 1990.

Update, Contraction and Revision in Knowledge Representation Systems

Gerd Wagner
Gruppe Logik, Wissenstheorie und Information
Institut für Philosphie, Freie Universität Berlin
Habelschwerdter Allee 30, 14195 Berlin, Germany
gw@inf.fu-berlin.de

Preliminary Report

Zusammenfassung Ein Wissensrepräsentationssystem (KRS) besteht aus zwei Hauptkomponenten: einer Inferenz- und einer Aktualisierungsoperation, deren Argumente Wissensbasen und Anfrage- bzw. Input-Ausdrücke sind. Ein *vivides* KRS (VKRS) besitzt darüberhinaus zwei Negationen: erstens eine *starke* Negation, mit deren Hilfe explizit negative Information ausgedrückt wird, und zweitens eine *schwache* Negation, die es erlaubt, Falschheit per 'default' im Stil der Negation-As-Failure auszudrücken. In einem VKRS läßt sich eine Kontraktionsoperation als Aktualisierung mit schwach negierten Inputs definieren. Revision mit F ergibt sich dann durch Kontraktion mit $\sim F$ und anschließende Aktualisierung mit F. Die Kontraktionsoperation ist daher fundamentaler als die Revisionsoperation, beide basieren jedoch auf der in einem KRS vorhandenen Aktualisierungsoperation. Zur formalen Charakterisierung der Kontraktion in einem VKRS werden die bekannten Rationalitätspostulate von Alchourrón, Gärdenfors and Makinson (siehe [Gärdenfors 1988]) entsprechend modifiziert und um das Relevanz-Postulat von Hansson [1991] ergänzt. Die jeweilige Ausprägung der Kontraktion wird dann für mehrere konkrete VKRS diskutiert: nämlich für Faktenbasen, Disjunktive Faktenbasen und Deduktive Datenbanken.

Abstract We define the general concept of a ***knowledge representation and reasoning system (KRS)*** consisting of two main components: an inference and an update operation manipulating knowledge bases as abstract objects. A *vivid* KRS has, in addition to a standard negation '$\sim$', a second negation in the style of negation-as-failure, called ***weak negation*** here, denoted by '$-$'. In a vivid KRS a contraction operation is definable as updating by weakly negated inputs: $\mathrm{KB} - F := \mathbf{Upd}(\mathrm{KB}, -F)$. Finally, revision can be defined according to $\mathrm{KB} * F := \mathbf{Upd}(\mathbf{Upd}(\mathrm{KB}, -\sim F), F)$.[1] Thus, in a KRS, update is more fundamental than contraction which is, in turn, more fundamental than revision. We present appropriate modifications of the rationality postulates for contraction and compare them to the original ones proposed in the AGM theory[2] and by Hansson [1991]. And we discuss the application of our notions to several VKRSs, including fact bases, disjunctive fact bases and their rule-based extensions.

1 Introduction

The concept of a knowledge representation and reasoning system (KRS) consists essentially of two main components: an inference and an update operation manipulating knowledge bases

[1] This definition of revision in terms of contraction is due to Levi [1977].

[2] Called after its originators: C. Alchourrón, P. Gärdenfors and D. Makinson. See [Gärdenfors 1988].

as abstract objects,[3] together with a set of formal properties these operations may have. In general, there are no specific restrictions on the internal structure of a knowledge base. We will see, however, that a computational design can be achieved by 'compiling' incoming information into some normal form rather than leaving it in the form of arbitrarily complex formulas. This is the case, for instance, in Belnap's KRS[4] which can be considered as the paradigm for KRSs.

The concept of a KRS constitutes a useful framework for the classification and comparison of various AI systems and formalisms. It is more general than that of a logic (i.e. a consequence relation). A standard logic can be viewed as a special kind of KRS. On the other hand, by defining the inference and update operations procedurally, KRSs can serve as the basis for the operational definition of logics. There is even a strong analogy between the concept of a KRS and that of a Gentzen sequent system.

Both relational and deductive databases can be considered as computational paradigms of real world KRSs. They implement a form of nonmonotonic reasoning caused by the use of negation-as-failure expressing falsity by default, and hence allowing to take into account default-implicit negative information. On the other hand, relational and deductive databases, as well as normal logic programs, are not capable of representing and processing explicit negative information. This shortcoming has led to the extension of logic programming by adding a second negation (in addition to negation-as-failure) as proposed independently in [Gelfond and Lifschitz 1990] and in [Pearce and Wagner 1990]. We will call the general concept of an operator expressing default-implicit negative information in the style of negation-as-failure *weak negation*, while the concept of an operator expressing explicit negative information will be called *strong negation*.

The concept of a *vivid* KRS *(VKRS)* is a two-fold generalization:

1. it extends already known logics, such as Belnap's 4-valued or Nelson's paraconsistent constructive logic,[5] by adding *weak* negation, and
2. it extends already known knowledge representation systems, such as relational or deductive database systems, by adding *strong* negation.

In the framework of a VKRS, a specific meaning is assigned to the *Closed-World Assumption*:[6] it connects the use of weak and strong negation in combination with partially and totally represented predicates. If the Closed-World Assumption holds for a predicate, its weak negation implies its strong negation, in other words, an atomic sentence formed with such a predicate is already false if it is implicitly (i.e. by default) false.

In contrast to standard logics, where dynamic aspects of reasoning are largely neglected, belief change operations, such as update, contraction and revision, are essential ingredients of KRSs. The term 'belief revision' is used in at least two ways: in its broader sense belief revision subsumes any systematic approach to the question of how to change a KB when new information comes in, and in its narrower sense it denotes the specific change operation axiomatised by the AGM postulates. In [GR 92] the goal of consistency maintenance is considered the dominating motive for belief revision. We would like to point out that there are reasonable systems, however, not maintaining consistency but nevertheless employing a revision operation.[7]

The main principles for belief change operations, according to Gärdenfors and Rott, are:

---

[3] This with respect to KR fundamental distinction was first proposed in [Levesque 1984] where the resp. operations are called *ASK* and *TELL*.

[4] Cf. [Belnap 1977].

[5] See [Belnap 1977], resp. [Almukdad & Nelson 1984].

[6] Cf. [Reiter 1978].

[7] E.g., [Cross & Thomason 1992].

1. The beliefs in a KB should be kept consistent whenever possible.
2. The amount of information lost in a belief change should be kept minimal.
3. In so far as some beliefs are considered more important than others, one should retract the least important ones.

The first principle is called *minimal inconsistency* here, and the second one *minimal change*. Gärdenfors and Rott also include the principle that KBs should be deductively closed, because this is essential for the formulation of the AGM theory. We will not adopt this principle, since it seems to be inappropriate for KRSs where KBs are always finite objects. Therefore, we will rather be concerned with change operations for 'belief bases' instead of infinite 'belief sets', as KBs are called in the AGM terminology. Hansson [1991] has proposed postulates for belief bases. The main difference to the AGM postulates seems to be that Hansson does not require recovery which indeed fails in disjunctive KRSs (see below).

In summary, our approach differs from the AGM approach (and related ones) in several respects:

1. KBs are not deductively closed.
2. Contraction is more fundamental than revision. In a VKRS, contraction is nothing else as weak negation on the left, i.e. the negative counterpart of ampliative updating,
3. We prefer *direct* constructive notions (such as *strong negation* '$\sim$') to indirect and nonconstructive ones (such as classical negation '$\neg$', or $K \bot F$, or $K_\bot$).
4. The preferred form of a KB is not a set of formulas, but rather some form of 'compiled' information (since every input formula can be compiled, this is no loss of generality). Contraction in such KBs is syntax-independent: e.g. $\lceil p \wedge q \rceil - p = \lceil p, q \rceil - p$. Also, KBs are not compared by means of set inclusion, $\subseteq$, but rather by means of an informational ordering $\leq$.
5. The basic pieces of information in a KRS are atoms, resp. literals, and not arbitrary formulas. We are, therefore, most interested in the formalization of contraction and revision for atoms, resp. literals, whereas disjunctions and conditionals may be included in the picture as optional extensions in a second step.
6. Vacuity and Recovery do not hold in general. In many cases, however, they hold for literals.
7. Consistency has not to be maintained under all circumstances in a VKRS. Paraconsistency is also o.k., though the ultimate goal is minimal inconsistency. This implies that our logics do not contain the concept of absurdity (expressed by $\bot$ and $K_\bot$) as a fundamental ingredient.
8. Strong negation does not satisfy the inference rule of (Negation as Inconsistency). Therefore, it is not possible to express that F is inconsistent with KB by the condition KB $\vdash \sim F$. This affects the formulation of the resp. proviso of several of the rationality postulates for revision.

2 Knowledge Representation and Reasoning

The language of KRSs consists of the logical operator symbols $\wedge, \vee, \sim, -$ and 1 standing for conjunction, disjunction, strong negation, weak negation and the verum, respectively; the predicate symbols $p, q, r, \ldots$; the constant symbols $c, d, \ldots$ and variables $x, y, \ldots$. Notice that there are no functional terms but only variables and constants. Further connectives, such as the epistemic modalities $\Box$ and $\Diamond$, and the (possibly restricted) quantifiers $\exists$ and $\forall$ can be optionally added to the language.

An *atom* is an atomic formula, it is called *proper*, if it is not 1. *Literals* are either atoms or strongly negated atoms. *Extended literals* are either literals or weakly negated literals. We use $a, b, \ldots, l, k, \ldots, e, f, \ldots$, and $F, G, H, \ldots$ as metavariables for atoms, literals, extended literals and formulas, respectively.[8] A variable-free expression is called *ground*. The set of all proper ground atoms (resp. literals, resp. extended literals) of a given language is denoted by At (resp. Lit, resp. XLit). If not otherwise stated, a formula is assumed to be ground. If *op-list* is a set of logical operators, say *op-list* $\subseteq \{1, -, \sim, \wedge, \vee, \rightarrow, \Box, \Diamond, \exists, \forall\}$, then $L(\textit{op-list})$ denotes the respective set of well-formed formulas.

With each negation a complement operation for the resp. type of literal is associated: $\tilde{a} = {\sim}a$ and $\widetilde{\sim a} = a$, $\bar{l} = -l$ and $\overline{-l} = l$. These complements are also defined for sets of resp. literals $L \subseteq \text{Lit}$, and $E \subseteq \text{XLit}$: $\tilde{L} = \{\tilde{l} : l \in L\}$, resp. $\overline{E} = \{\bar{e} : e \in E\}$. We distinguish between the positive and negative elements of $E \subseteq \text{XLit}$ by writing $E^+ := E \cap \text{Lit}$ and $E^- := \{l : -l \in E\}$.

The inference relation of a KRS is not uniform, in general. Only certain formulas may make sense for representing vivid knowledge, that is, there will be a specific representation language L_{Repr}, and a KB will be a finite collection of elements of L_{Repr}, possibly constrained in some way determined by the set L_{KB} of all admissible KBs: $\text{KB} \in L_{\text{KB}} \subseteq 2^{L_{\text{Repr}}}$.

Likewise, since not every formula may be appropriate as a sensible query, the set of admissible queries is specified by L_{Query}. The inference relation of a KRS, thus, is in general not based on a single universal language applying to premises as well as to queries (resp. consequences), but on two, usually different, languages: $\vdash \subseteq L_{\text{KB}} \times L_{\text{Query}}$.

Moreover, the axiomatization of standard consequence seems to be overidealised and not adequate for commonsense reasoning. This is no longer debated today in the case of the monotonicity postulate which requires that all previously obtainable inferences remain valid whenever new information comes in. In many forms of commonsense reasoning, such as default reasoning, and in many computational systems, such as Prolog, monotonicity is violated, and one has to look for other principles replacing it, or one can just drop it and allow for unrestricted nonmonotonicity.[9]

The basic scenario of a KRS consists of two operations: an inference operation processing queries posed to the KB, and an update operation processing input formulas entered by users or by other (e.g. sensoric) information suppliers. While in standard logics an update is a simple addition of a formula $F \in L$ to the premise set $X \subseteq L$, i.e. $X \cup \{F\}$, a KRS restricts the admissible inputs to elements of a specific input language L_{Input}, and an update is performed by processing the input formula in an appropriate way in order to add its information content to the KB.

In general, a KB can consist of any kind of data structures capable of representing knowledge, e.g. a set, or multiset, or sequence, of (logical) expressions, or a directed graph, etc. For the sake of simplicity, I will assume that a $\text{KB} \in L_{\text{KB}}$ is a finite set of expressions from a representation

[8] I will frequently just say "literal" when I, precisely speaking, mean a proper ground literal.

[9] It is an open debate if, and in what form, nonmonotonicity should be restricted in a KRS. The currently most popular proposal is *Cautious Monotonicity* which will be called *Lemma Compatibility* below.

language, i.e. $L_{\mathrm{KB}} \subseteq 2^{L_{\mathrm{Repr}}}$. There will always be an informationally 'empty' KB, denoted by 0, which is not necessarily equal to the empty set.

2.1 Knowledge Representation Systems

A KRS is a quintuple:

$$\langle L_{\mathrm{KB}}, \vdash, L_{\mathrm{Query}}, \mathbf{Upd}, L_{\mathrm{Input}} \rangle$$

where the inference relation $\vdash \subseteq L_{\mathrm{KB}} \times L_{\mathrm{Query}}$ is asscociated with a resp. inference operation C in the usual way:

$$C(\mathrm{KB}) = \{F \in L_{\mathrm{Query}} : \mathrm{KB} \vdash F\} \qquad \text{and} \qquad C^l(\mathrm{KB}) = \{l \in \mathrm{Lit} : \mathrm{KB} \vdash l\}$$

and the update operation **Upd** takes a KB and an input formula F, and provides an appropriately updated KB:

$$\mathbf{Upd} : L_{\mathrm{KB}} \times L_{\mathrm{Input}} \to L_{\mathrm{KB}}$$

An update may be simply an addition of a formula $F \in L_{\mathrm{Input}} \cap L_{\mathrm{Repr}}$ to the KB:

$$\mathbf{Upd}(\mathrm{KB}, F) = \mathrm{KB} \cup \{F\}$$

But it may also be necessary to process the input formula in some way, especially if it is not an L_{Repr} expression. For instance, an RDB can be updated by a conjunction of atoms by simply adding all atoms to it.

Notice that L_{Input} and L_{Repr} may be totally distinct (one may expect that an input formula is a kind of logical expression whereas a KB can be any kind of data structure). We use the following abbreviations: $\mathrm{KB} + F := \mathbf{Upd}(\mathrm{KB}, F)$, $\mathrm{KB} - F := \mathbf{Upd}(\mathrm{KB}, -F)$, and $\mathrm{KB} * F := \mathbf{Upd}(\mathbf{Upd}(\mathrm{KB}, -\sim F), F)$, standing for expansion, contraction and revision.

The formulation of a KRS in terms of query and input processing was already implicitly present in Belnap's [1977] view of a KRS. In [Levesque 1984] it was proposed as a 'functional approach to knowledge representation'.[10]

Standard logics, such as classical or intuitionistic logic, can be viewed as special (very idealized) cases of KRSs. Here, the KB consists of a (possibly infinite) set of arbitrary first order sentences, and is usually called a 'theory'. The inference operation is monotonic. Updates are simple additions of formulas, and the representation language is equal to the query and input language, $L_{\mathrm{Repr}} = L_{\mathrm{Query}} = L_{\mathrm{Input}}$.

2.2 Some Formal Properties of KRSs

Although it seems desirable, lemmas are not always redundant and compatible in KRSs. In other words, there are reasonable cases of KRSs which are not cumulative.

Lemma Redundancy (alias: Cut, Transitivity)

$$\mathrm{KB} \vdash F \;\&\; \mathbf{Upd}(\mathrm{KB}, F) \vdash G \;\Rightarrow\; \mathrm{KB} \vdash G$$

Lemma Compatibility (alias Cautious Monotonicity, due to [Gabbay 1985])

$$\mathrm{KB} \vdash F \;\&\; \mathrm{KB} \vdash G \;\Rightarrow\; \mathbf{Upd}(\mathrm{KB}, F) \vdash G$$

[10] However, Levesque is not so much concerned with the investigation of the formal properties of KRSs in general, but rather with the modelling of the concept of meta-knowledge by means of an epistemic modality on the basis of classical logic.

Lemma Redundancy and Compatibility can be combined in the following condition of

Cumulativity $\quad \mathrm{KB} \vdash F \;\Rightarrow\; C(\mathbf{Upd}(\mathrm{KB}, F)) = C(\mathrm{KB})$

The following condition of Monotonicity seems to be too strong for commonsense reasoning. In a specifically restricted KRS, however, it may hold.

Monotonicity $\quad C(\mathrm{KB}) \subseteq C(\mathbf{Upd}(\mathrm{KB}, F))$

Clearly, Monotonicity implies Lemma Compatibilty.

It will be useful to compare knowledge bases in terms of their information content, that is, to have an informational ordering between KBs such that

$\mathrm{KB}_1 \leq \mathrm{KB}_2$ if KB_2 contains more information than KB_1.

The informational ordering '$\leq$' should be defined in terms of the structural components of knowledge bases and not in terms of higher-level notions (like derivability).

The informationally empty KB will be denoted by 0. By definition, $0 \leq \mathrm{KB}$ for all KBs.

Definition 1 (Unique Representation) *A KRS enjoys the property of* Unique Representation *if the information contained in a KB is uniquely represented, i.e. if*

$$C(\mathrm{KB}_1) = C(\mathrm{KB}_2) \;\Rightarrow\; \mathrm{KB}_1 = \mathrm{KB}_2$$

A standard logical system, where a KB is a set of well-formed formulas, does not enjoy the Unique Representation property. On the other hand, a relational database, for instance, is a unique representation. Also a *belief set* in the sense of [Gärdenfors 1988] is a unique representation.

In general, it will not suffice in order to infer $-F$ that F fails (this will only be the case for definite KBs). However, the following restriction characterizes weak negation:

(Inherent Consistency) $\quad \mathrm{KB} \vdash -F \;\Rightarrow\; \mathrm{KB} \not\vdash F$

Notice that this does not hold for classical negation. In fact, Inherent Consistency is violated by every negation satisfying the classical principle *ex contradictione sequitur quodlibet*, $\{F, \sim F\} \vdash G$. Also strong negation, in general, does not satisfy Inherent Consistency. Yet, it seems desirable that the following *coherence*[11] property holds:

(Coherence) $\quad \mathrm{KB} \vdash \sim F \;\Rightarrow\; \mathrm{KB} \vdash -F \;\Rightarrow\; \mathrm{KB} \not\vdash F$

In general, more information does not mean more consequences. In other words: answers are not necessarily preserved under growth of information. Queries, for which this is the case, are called *persistent*.

Definition 2 (Persistent Query) *A closed, resp. open, query formula F is called* persistent *if* $\mathrm{KB}_1 \vdash F \;\Rightarrow\; \mathrm{KB}_2 \vdash F$, *whenever* $\mathrm{KB}_1 \leq \mathrm{KB}_2$.

Definition 3 (Ampliativity) *A KRS and its update operation* **Upd** *are called* ampliative,[12] *if for all* $F \in L_{\mathrm{Input}}$, $\mathrm{KB} \leq \mathbf{Upd}(\mathrm{KB}, F)$.

[11] The name is adopted from [Pereira & Alferes 1992].

[12] The name is adopted from [Belnap 1977].

If a KRS is monotonic, its update operation is ampliative. Ampliative updating is the formal counterpart of persistent answering. If **Upd** is not ampliative one can still discriminate ampliative input formulas (as the formal counterpart of persistent query formulas).

Definition 4 *An input formula F is called (i)* ampliative *if* $\mathrm{KB} \leq \mathbf{Upd}(\mathrm{KB}, F)$, *or (ii)* reductive *if* $\mathrm{KB} \geq \mathbf{Upd}(\mathrm{KB}, F)$.

Permutation

$$\mathbf{Upd}(\mathbf{Upd}(\mathrm{KB}, F), G) = \mathbf{Upd}(\mathbf{Upd}(\mathrm{KB}, G), F)$$

Conjunction Composition

$$\mathbf{Upd}(\mathrm{KB}, F \wedge G) = \mathbf{Upd}(\mathbf{Upd}(\mathrm{KB}, F), G)$$

Observation 1 *The condition of Conjunction Composition can be used as a natural definition of conjunction processing. That is, with its help L_{Input} can always be closed under conjunction. Because standard conjunction is commutative, Conjunction Composition can only be used as a definition of conjunctive input processing if* **Upd** *satisfies Permutation, that is, if time does not matter for updates. Below, a simple system, V_0, will be presented where Permutation is violated, and first learning F and then learning G is not always the same as learning $F \wedge G$ in a single step. Therefore, Conjunction Composition does not hold.*

Observation 2 *The expressiveness of L_{KB} does not matter. More important is the expressiveness of the input and the query language as the following construction of the entailment relation associated with a KRS shows. For $F \in L_{\mathrm{Input}}$ and $G \in L_{\mathrm{Query}}$ define*

$$F \vdash G :\Longleftrightarrow \mathbf{Upd}(\mathrm{KB}, F) \vdash G \text{ for any } \mathrm{KB} \in L_{\mathrm{KB}}$$

For finite $X \subseteq L_{\mathrm{Input}}$ we define: $X \vdash F$ iff $\bigwedge X \vdash F$. The generality of the so-defined (finite) entailment relation does not depend on L_{KB}. Notice that the standard definition of entailment (compare with $\models F :\Longleftrightarrow \emptyset \models F$) corresponds to

$$F \vdash G :\Longleftrightarrow \mathbf{Upd}(0, F) \vdash G,$$

which is equivalent to the general definiton if the KRS satisfies Monotonicity and Permutation.

Observation 3 (Update Postulates) *The following postulates are adapted from [Gärdenfors 1988] where they were formulated for* expansion *and* contraction.

(Success)	$\mathbf{Upd}(\mathrm{KB}, F) \vdash F$
(Vacuity)	$\mathrm{KB} \vdash F \Rightarrow \mathbf{Upd}(\mathrm{KB}, F) \cong \mathrm{KB}$
(Extensionality)	$F \vdash G \;\&\; G \vdash F \Rightarrow \mathbf{Upd}(\mathrm{KB}, F) \cong \mathbf{Upd}(\mathrm{KB}, G)$

Instead of equality, we have used equivalence in the formulation of Vacuity and Extensionality, since KBs may not be unique representations. The postulate of Success is nothing else as unrestricted reflexivity, whereas Vacuity is slightly stronger than Cumulativity. We will see that Vacuity will in many cases only hold for literals. Another postulate, called Inclusion *in the AGM theory, expresses the property of updates being ampliative in the case of expansion, and reductive in the case of contraction. Notice that Extensionality, in the AGM theory, is a kind of soundness condition with respect to classical logic, whereas here every KRS defines its own logic.*

2.3 Vivid Knowledge Representation Systems

Depending on the specific requirements of an application, and on the computational resources available, different answer and update operations constituting different KRS's may be appropriate. However, there are some minimal requirements any vivid KRS has to satisfy, namely Restricted Reflexivity, Constructivity and Non-Explosiveness.[13] In the sequel, a KRS that satisfies these conditions is called a *basic VKRS*.

Additionally, besides a principal negation (called *strong*), expressing explicit falsity, there should be a second negation (called *weak*) which handles default-implicit negative information in the style of negation-as-failure. Thus, the query language, and possibly also the input language, should contain $\sim$ and $-$.

Some form of CWA, restricted to specific predicates (namely those which are totally represented in the knowledge base) then relates explicit with implicit falsity, i.e. strong with weak negation: an atomic sentence formed with a totally represented predicate is explicitly false if it is implicitly false, i.e. its strong negation holds if its weak negation does (see also [Wagner 1993b]). A basic VKRS equipped with a weak negation and a predicate specific CWA mechanism is called a (full) *VKRS*.

If a VKRS enjoys the properties of Lemma Redundancy and Compatibility, it will be called *cumulative*. In certain respects, a cumulative VKRS is preferable to a non-cumulative one since it is more regular. However, there seems to be a trade-off between regularity and the capability to capture all aspects of commonsense reasoning. In particular, the price for cumulativity seems to be a more cautious inference operation not allowing for certain inferences suggested by respective commonsense reasoning schemes.

In a VKRS we have the following rewrite rules in order to simplify complex formulas:

$$\begin{array}{rcl}
-(F \wedge G) & \longrightarrow & -F \vee -G \\
\sim(F \wedge G) & \longrightarrow & \sim F \vee \sim G \\
-\sim(F \wedge G) & \longrightarrow & -\sim F \wedge -\sim G \\
-(F \vee G) & \longrightarrow & -F \wedge -G \\
\sim(F \vee G) & \longrightarrow & \sim F \wedge \sim G \\
-\sim(F \vee G) & \longrightarrow & -\sim F \vee -\sim G \\
\sim\sim F & \longrightarrow & F \\
-\sim\sim F & \longrightarrow & -F \\
\sim -F & \longrightarrow & F \\
--F & \longrightarrow & F \\
-\sim -F & \longrightarrow & -F
\end{array}$$

For inductive definitions, then, it is sufficient to treat the cases of the verum (1), of extended literals (e), of conjunctions ($\wedge$), and of disjunctions ($\vee$). All other cases are covered by the above rewrite rules.[14]

In the sequel we frequently use the fact that formulas $F \in L(1, -, \sim, \wedge, \vee)$ can be normalized

[13] See [Wagner 1993].

[14] For their justification see [Wagner 1993].

according to the following definition:

$$\begin{array}{lcl} \mathrm{DNS}(1) & = & \{\emptyset\} \\ \mathrm{DNS}(e) & = & \{\{e\}\} \\ \mathrm{DNS}(F \wedge G) & = & \{K \cup L : K \in \mathrm{DNS}(F),\ L \in \mathrm{DNS}(G)\} \\ \mathrm{DNS}(F \vee G) & = & \mathrm{DNS}(F) \cup \mathrm{DNS}(G) \end{array}$$

This formulation is inspired by a similar one (without weak negation) in [Miller 1989]. The disjunctive normal form of a formula $G \in L(1, -, \sim, \wedge, \vee)$ is obtained as

$$\mathrm{DNF}(G) = \bigvee_{K \in \mathrm{DNS}(G)} \bigwedge K$$

We will use the following notation for a KRS: let $\boldsymbol{K}$ denote a KRS where only the query but not the input language contains weak negation, then $\boldsymbol{K}^+$ denotes its weak-negation-free fragment not allowing for weak negation in queries, and $\boldsymbol{K}^-$ denotes its extension by adding weak negation to the input language.

3 Contraction

The minimal change principle for literal contraction in KRSs can be formalized as follows:

$(l\text{-}1)$ $\mathrm{KB} - l \leq \mathrm{KB}$
$(l\text{-}2)$ for all $\mathrm{KB}' \leq \mathrm{KB} : \mathrm{KB}' \vdash -l \ \Rightarrow\ \mathrm{KB}' \leq \mathrm{KB} - l$

$(l\text{-}1)$ says that the contraction by a literal causes a loss of information, while $(l\text{-}2)$ says that this loss is minimal: all other informational reductions of KB yielding $-l$ contain less information than $\mathrm{KB} - l$.

We now present a list of rationality postulates for contraction in KRSs. They are appropriate modifications of postulates originally proposed in [Gärdenfors 1988] and [Hansson 1991] as an axiomatization of contraction for belief sets, resp. bases. In our direct approach, however, these postulates are not axioms, but rather expressing interesting properties the contraction operation defined in a KRS may have.

(Success)	$\not\vdash F \Rightarrow \mathrm{KB} - F \vdash -F$
(Inclusion)	If F is ampliative, then $\mathrm{KB} - F \leq \mathrm{KB}$
(Vacuity)	$\mathrm{KB} \vdash -F \Rightarrow \mathrm{KB} - F \cong \mathrm{KB}$
(Extensionality)	$F \vdash G \ \&\ G \vdash F \Rightarrow \mathrm{KB} - F \cong \mathrm{KB} - G$
(Relevance)	$\mathrm{KB} \vdash F, G \ \&\ \mathrm{KB} - F \vdash -G \Rightarrow (\mathrm{KB} - F) + G \vdash F$
(Recovery)	$\mathrm{KB} \vdash F \Rightarrow (\mathrm{KB} - F) + F \cong \mathrm{KB}$
(Conjunction 1)	$C^l(\mathrm{KB} - F) \cap C^l(\mathrm{KB} - G) \subseteq C^l(\mathrm{KB} - F \wedge G)$
(Conjunction 2)	$\mathrm{KB} - F \wedge G \not\vdash F \Rightarrow \mathrm{KB} - F \wedge G \leq \mathrm{KB} - G$

Notice that in our formulation of Vacuity, Success and Relevance we have used $\mathrm{KB} \vdash -F$ instead of the original $\mathrm{KB} \not\vdash F$. Because of $\mathrm{KB} \vdash -F \Rightarrow \mathrm{KB} \not\vdash F$, this strengthens the original formulation.

While Relevance and Recovery are conditions on the interaction between contraction and expansion, the postulates of success, vacuity and extensionality describe properties of the update operation.

4 Consolidation and Revision

While the axiomatization of revision in the AGM theory is inspired by the kind of theory revision investigated in the philosophy of science, our prime concern is revision in knowledge bases (as a generalization of databases) where we have quite a different situation. This concerns, for instance, the success postulate giving top-priority to new inputs, so that all new information will have to be accepted no matter how much of the already represented information is incompatible with it – something we do not want in general in a KRS. Instead of the AGM revision operator, a non-prioritized revision operator seems to be more fundamental for a KRS. Such a belief change operation is called *consolidation* in [Hansson 1991]. It proceeds in two steps: first update KB by the new input F, and then, if necessary, consolidate $\text{KB} + F$, i.e. make it consistent (or minimally inconsistent) by appropriate contractions.

In the sequel, we will simply say *revision* for a non-prioritized consolidating update operation, while we call the special AGM concept of revision, based on the Levi identity, *recency-preferring revision*. Thus, we will deal with four kinds of update operations in KRSs:

1. Simple (possibly inconsistent) update, denoted by $\mathbf{Upd}(\text{KB}, F)$, or in short, $\text{KB} + F$.
2. Minimally inconsistent consolidation (in disjunctive KRSs), denoted by $\text{KB} \uplus F$.
3. Consistent consolidation, denoted by $\text{KB} \oplus F$.
4. Recency-preferring revision, denoted by $\text{KB} * F$.

The AGM postulates for a revision operator $\circ = \uplus, \oplus, *$ (supplemented with Hansson's relevance postulate) adapted to the KRS framework are:[15]

(Inclusion)	$\text{KB} \circ F \leq \text{KB} + F$
(Vacuity)	If F is consistent with KB, then $\text{KB} \circ F \cong \text{KB} + F$
(Consistency Preservation)	$\text{KB} \oplus F$ is consistent, while $\text{KB} \uplus F$ is minimally inconsistent.
(Extensionality)	$F \vdash G \ \&\ G \vdash F \Rightarrow \text{KB} \circ F \cong \text{KB} \circ G$
(Relevance)	$\text{KB} \vdash G \ \&\ \text{KB} \circ F \not\vdash G \Rightarrow (\text{KB} \circ F) + G$ is inconsistent
(Conjunction 1)	$\text{KB} \circ (F \wedge G) \leq (\text{KB} \circ F) + G$
(Conjunction 2)	If G is consistent with $\text{KB} \circ F$, then $(\text{KB} \circ F) + G \leq \text{KB} \circ F \wedge G$

The recency-preferring revision $\text{KB} * F$, according to the Levi identity, is the result of first removing $\sim F$ from $C(\text{KB})$, and then adding F to KB. It is characterized by the postulate of success, and a restricted consistency preservation postulate:

(Success)	If F is consistent, then $\text{KB} * F \vdash F$
(Consistency Preservation)	$\text{KB} * F$ is only inconsistent if F is inconsistent

4.1 Inconsistency in KRSs

Notice that strong negation does not satisfy the following (classically valid) condition of

(Negation as Inconsistency) $\quad \text{KB} \vdash \neg F \quad$ iff $\quad \text{KB} + F$ is inconsistent

Therefore, in contrast to classical logic, $\text{KB} \not\vdash \sim F$ does not guarantee that F is consistent with KB. Consider the extended logic program $\Pi = \{p, \sim p \leftarrow q\}$. Clearly, $\Pi \not\vdash \sim q$. However,

[15] In [GR 92] (Inclusion) is called (Expansion 1), and (Vacuity) is called (Expansion 2).

$\Pi * q = \Pi + q$ is inconsistent. This means that several conditions for recency-preferring revision, notably (Consistency Preservation) and (Success), do not hold if it is defined according to the Levi identity, $\mathrm{KB} * F := (\mathrm{KB} - \sim F) + F$.

We have two options to remedy this problem. They are associated with the preferred approach to inconsistency handling: either consistency maintenance or paraconsistency, resp. minimal inconsistency. If we want consistency maintenance we could look for a definition of revision capturing the spirit of Levi's definition (something like: first make KB consistent with F, and then add F).

If, on the other hand, paraconsistency (resp. minimal inconsistency) is preferred, the Levi definition has to be modified in order to prevent the inconsistency of F in $\mathrm{KB} * F$, e.g. in the following way:

$$\mathrm{KB} * F := \mathrm{KB} - (F \rightarrow \sim F) + F$$

For instance, if $\Pi = \{p, \sim q \leftarrow q\}$, this gives $\Pi * q = \{p, q\}$ instead of $\{p, \sim q \leftarrow q, q\}$, where the new input q is inconsistent.

4.2 Implicit Consolidation versus Consistency Maintenance

There is a computationally attractive alternative to global consistency maintenance: nonexplosive inference operations based on mutual neutralization of contradictory information. Such inference operations[16] yield consistent conclusions only. Thus, they can be viewed as *implicit consolidation*. While explicit consolidation is characterised by Consistency Preservation,

$$\mathrm{KB} \oplus F \text{ is always consistent,}$$

implicit consolidation by means of a resp. neutralization-based inference operation $C^{\oplus}$ is characterised by the postulate of *Inherent Consistency*,

$$C^{\oplus}(\mathrm{KB} + F) \text{ is always consistent.}$$

The following question arises: *Given a revision operation* $\oplus$, *and standard inference operation* C *which only works for consistent KBs, is there an inherently consistent inference operation* $C^{\oplus}$ *such that*

$$C^{\oplus}(\mathrm{KB} + F) \quad = \quad C(\mathrm{KB} \oplus F)$$

In a KRS, both $\oplus$ *and* $C^{\oplus}$ *make sense depending on the specific requirements of an application. If a KB has to process a lot of queries, and only a few inputs, then consistency maintenance by means of a revision operation seems to be feasible. If, on the other hand, a KB has to process a lot of inputs, and only a few queries, then revision may be not feasible due to the fact that global inconsistency detection is hard to compute, and an inherently consistent inference operation might be the better choice.*

5 Fact Bases

A KB consisting of ground literals (viewed as positive and negative facts) is actually a slight generalization of a vivid knowledge base *in the restricted sense of Levesque [1988], where only positive facts, i.e. ground atoms, are allowed. For example,* $X_1 = \{\sim b(S), m(P, L)\}$ *represents the information that Susan is not blonde, and that Peter is married to Linda.*

[16] Investigated, e.g., in the *logic of argumentation*, and the area of *defeasible inheritance*. In [Wagner 1993a], neutralization-based top-down reasoning procedures have been proposed for extended logic programming.

Definition 5 (Informational Ordering) *Let X and X' be fact bases, i.e. $X, X' \subseteq$ Lit. Then X' is an informational extension of X, symbolically $X' \geq X$, if $X' \supseteq X$.*

As a kind of natural deduction from positive and negative facts a derivability relation $\vdash$ between a set of proper ground literals X and a ground formula $F \in L(1, -, \sim, \wedge, \vee)$ is defined:

$(\vdash l)$ $X \vdash l$ if $l \in X$
$(\vdash -l)$ $X \vdash -l$ if $l \notin X$
$(\vdash \wedge)$ $X \vdash F \wedge G$ if $X \vdash F$ & $X \vdash G$
$(\vdash \vee)$ $X \vdash F \vee G$ if $X \vdash F$ or $X \vdash G$

For example, $X_1 \vdash \sim b(S) \wedge -\sim b(L)$. Notice that both a literal l and its complement $\bar{l}$ are acceptable independently from each other. This kind of inference is called liberal.

Observation 4 *A liberal inference relation is not coherent, i.e. $X \vdash \sim p$ does not imply that $X \vdash -p$. In other words, it is possible to infer contradictory queries, such as $p \wedge \sim p$.*

Definition 6 *Updating by extended literals is defined as insertion, resp. deletion: $\mathbf{Upd}(X, l) := X \cup \{l\}$, resp. $\mathbf{Upd}(X, -l) := X - \{l\}$. More generally,* **Upd** *can be defined for sets (corresponding to conjunctions) of extended literals, $E \subseteq$ XLit, as $\mathbf{Upd}(X, E) := X \cup E^+ - E^-$.*[17]

For example, if we learn that Peter gets divorced from Linda and marries Susan, we perform the following update: $\mathbf{Upd}(X_1, -m(P, L) \wedge m(P, S)) = \{\sim b(S), m(P, S)\}$. Such a sequence of basic inputs (insertions and deletions of atoms or literals) is also called a knowledge base transaction. *Notice that if E is weakly inconsistent, i.e. $E^+ \cap E^- \neq \emptyset$, the inconsistent inputs neutralize each other by first being added and then being deleted again. In this case,* **Upd** *does not satisfy Conjunction Composition. Therefore, consecutive updates at different time stamps cannot be expressed by a conjunction, in general.*

Definition 7 *The KRS $\langle 2^{\mathrm{Lit}}, \vdash, L(1, -, \sim, \wedge, \vee), \mathbf{Upd}, \mathrm{XLit} \rangle$ is denoted by $\mathbf{V}_\circ^-$.*

Claim 1 *$\mathbf{V}_\circ^-$ is a cumulative VKRS.*

Claim 2 *In $\mathbf{V}_\circ^+$ and $\mathbf{V}_\circ^-$, where we only have literals, resp. extended literals as input, the contraction operation '$-$' satisfies (l-1) and (l-2), and furthermore inclusion, vacuity, success, relevance, extensionality and literal recovery.*

6 The Disjunctive Extension of a KRS

Let $\mathbf{K}$ be a definite KRS, that is, KBs in $\mathbf{K}$ are definite, and hence, any input formula is also definite,[18] *implying that the input laguage does not allow for disjunctions. The* disjunctive extension *of $\mathbf{K}$, symbolically $D\mathbf{K}$, is defined as the KRS $\langle L^d_{\mathrm{KB}}, \vdash_d, L^d_{\mathrm{Query}}, \mathbf{Upd}_d, L^d_{\mathrm{Input}} \rangle$ where $L^d_{\mathrm{KB}} := 2^{L_{\mathrm{KB}}}$ and for $Y \in L^d_{\mathrm{KB}}$,*

$$Y \vdash_d F :\Longleftrightarrow \text{for all } X \in Y : X \vdash F$$

The elements $X \in Y \subseteq L_{\mathrm{KB}}$ of a disjunctive KB are also called (possible) situation descriptions.

Notice that in $D\mathbf{K}$, $Y = \emptyset$ is not a meaningful KB: it trivially confirms every sentence. The 'empty' KB of $D\mathbf{K}$ contains as its only element the empty set: $0 = \{\emptyset\}$.

The informational ordering of KBs is extended from $\mathbf{K}$ to $D\mathbf{K}$:

[17] Thus, an input formula can be any definite formula F, such that its disjunctive normal set is a singleton: $\mathrm{DNS}(F) = \{E\}$.

[18] A formula F is definite if its disjunctive normal set is a singleton: $\mathrm{DNS}(F) = \{E\}$.

Definition 8 (Informational Ordering) *Let $\geq$ be an informational ordering of KBs in $\mathbf{K}$, then an ordering relation between the disjunctive KBs of $D\mathbf{K}$ can be defined according to*

$$Y' \geq_d Y :\Longleftrightarrow \forall X' \in Y' \exists X \in Y : X' \geq X$$

Finally, a disjunction operator is added to the input language of $\mathbf{K}$, $L^d_{\text{Input}} := L_{\text{Input}}(\vee)$, and the update operation is inductively extended in the following way:

$$\begin{array}{lll} (Ue) & \mathbf{Upd}_d(Y,e) & = \{\mathbf{Upd}(X,e) : X \in Y\} \\ (U\wedge) & \mathbf{Upd}_d(Y,F \wedge G) & = \mathbf{Upd}_d(\mathbf{Upd}_d(Y,F),G) \\ (U\vee) & \mathbf{Upd}_d(Y,F \vee G) & = \mathbf{Upd}_d(Y,F) \cup \mathbf{Upd}_d(Y,G) \cup \mathbf{Upd}_d(Y,F \wedge G) \\ (U\,|) & \mathbf{Upd}_d(Y,F \mid G) & = \mathbf{Upd}_d(Y,F) \cup \mathbf{Upd}_d(Y,G) \end{array}$$

All other inductive cases are handled by the above rewrite rules for complex formulas.[19] *Notice that the definition of disjunctive update, $(U\vee)$, is* inclusive *(as is standard disjunction).*

6.1 Disjunctive Fact Bases

Following Belnap [1977] we call a set Y of partial interpretations (represented as sets of literals) an epistemic state. *Since an epistemic state is able to represent disjunctive information, it can also be viewed as a* disjunctive fact base *where - in addition to Belnap's considerations - we also allow for weak negation in the query and in the input language.*

Definition 9 $\mathbf{V}_B := D\mathbf{V}_0$. *The weak-negation-free fragment of the disjunctive extension of fact bases, $\mathbf{V}_B^+$, is called* Belnap's KRS.

Example 1 $\{\{p\},\{\sim q\}\} \vdash (p \vee \sim q) \wedge -(p \wedge \sim q)$. *However, neither p nor $\sim q$, and neither $-p$ nor $-\sim q$ are derivable.*

Example 2 *Let $Y_1 = \{X_1\}$ where $X_1 = \{\sim b(S), m(P,L)\}$. If we learn that not both Peter and Linda are blonde, we perform the following update:*

$$Y_2 := \mathbf{Upd}(Y_1, \sim(b(P) \wedge b(L))) = \{\, X_1 \cup \{\sim b(P)\},\ X_1 \cup \{\sim b(L)\}\ X_1 \cup \{\sim b(P), \sim b(L)\}\,\}$$

We obtain, for instance, $Y_2 \not\vdash -\sim b(L)$, and also $Y_2 \geq Y_1$.

Observation 5 (Failure of Vacuity) *Vacuity fails in many cases of compound formulas:*

$$\{\{p\}\} \vdash -(p \wedge q), \quad \text{but} \quad \{\{p\}\} - p \wedge q = \{\{\},\{p\}\} \not\cong \{\{p\}\}$$

Observation 6 (Failure of Recovery) *Recovery fails in the case of disjunctions:*

$$\{\{p,q\}\} - p \vee q + p \vee q = \{\emptyset\} + p \vee q = \{\{p\},\{q\},\{p,q\}\} \not\geq \{\{p,q\}\}$$

Claim 3 *In $\mathbf{V}_B^+$ and $\mathbf{V}_B^-$, the contraction operation '$-$' satisfies (l-1) and (l-2), and furthermore inclusion, literal vacuity, success, relevance, extensionality and literal recovery.*

[19] $(\wedge U)$ requires that $F \wedge G$ are weakly consistent. Alternatively, the update operation can be defined by means of the disjunctive normal set which also works for weakly inconsistent inputs.

7 Rule-Based Systems

With each KRS $\boldsymbol{K}$ *a* rule-based *extension,* $\boldsymbol{RK}$*, can be associated. In* $\boldsymbol{RK}$ *a knowledge base* $X \in L_{\text{KB}}$ *is supplemented by a set* $R \subseteq L_{\text{Input}} \times L_{\text{Query}}$ *containing rules* $r = \langle Conclusion, Premise \rangle$ *with* $Conclusion \in L_{\text{Input}}$ *and* $Premise \in L_{\text{Query}}$*, also written as '*$Conclusion \leftarrow Premise$*'. These rules are mappings between KBs,*

$$r : L_{\text{KB}} \to L_{\text{KB}}$$

since – in the standard case – their application is defined as

$$r(X) = \begin{cases} \mathbf{Upd}(X, Conclusion) & \text{if } X \vdash Premise \\ X & \text{otherwise} \end{cases}$$

Definition 10 *A mapping* $f : A \to A$ *from a preorder* $\langle A, \leq \rangle$ *into itself is called* monotonic *if* $f(x) \leq f(y)$ *whenever* $x \leq y$*. It is called* ampliative *if* $x \leq f(x)$*. A rule is called* monotonic *(resp.* ampliative*) if it is a monotonic (resp. ampliative) mapping.*

Observation 7 *The rule* $F \leftarrow G$ *is monotonic if its conclusion* F *is ampliative, and its premise* G *is persistent.*

The semantics of a rule knowledge base $\langle X, R \rangle$ *is determined by the definition of a preferred closure of* X *under* R*, being a knowledge base* Z *closed under all rules of* R*:*

(i) $Z \in L_{\text{KB}}$

(ii) $r(Z) = Z$ *for all* $r \in R$

In general, however, there may be several preferred closures, or none. We denote their collection by $R(X)$*. If there are several preferred closures, a valid consequence must be inferrable from all of them:*

$$C(\langle X, R \rangle) \quad := \quad \bigcap \{C(Z) : Z \in R(X)\}$$

7.1 Extended Deductive Databases

Extended deductive databases (XDBs) correspond to rule knowledge bases of $\mathbf{RV}_0$*. For instance, the following XDB*

$$\Pi_1 \ = \{\, m(P, L), {\sim} b(S), {\sim} m(x, y) \leftarrow -m(x, y) \,\}$$

corresponds to the rule knowledge base $\langle X_1, R_1 \rangle$ *in* $\mathbf{RV}_0$ *such that* X_1 *is as above, and* $R_1 = \{{\sim} m(x, y) \leftarrow -m(x, y)\}$*. We obtain as the unique preferred closure*

$$R_1(X_1) = \{{\sim} b(S), m(P, L), {\sim} m(P, S)\},$$

and consequently, $\langle X_1, R_1 \rangle \vdash {\sim} m(P, S)$*.*

The following definition of informational ordering for XDBs is preliminary:

$$\Pi \leq \Pi' :\Longleftrightarrow C^l(\Pi) \subseteq C^l(\Pi')$$

where $C^l(\text{KB}) := \{l \in \text{Lit} : \text{KB} \vdash l\}$*. Updating by a weakly negated ground literal in an extended deductive database is defined as follows:*

Definition 11 (Literal Contraction) *Let l be a ground literal. Then* $\mathbf{Upd}(\Pi, -l)$ *denotes the XDB obtained from Π by making the following changes:*

1. *If Π contains l as a fact (resp. a rule $l \leftarrow F$), then l (resp. $l \leftarrow F$) is deleted from Π.*
2. *If $l = p(c)$, resp. $l = \sim p(c)$, and Π contains the rule $p(x) \leftarrow F(x)$, resp. $\sim p(x) \leftarrow F(x)$, such that $\Pi \vdash F(c)$, then this rule is replaced by $p(x) \leftarrow x \neq c \wedge F(x)$, resp. $\sim p(x) \leftarrow x \neq c \wedge F(x)$.*

Example 3 *We can now apply this contraction operation to an example given in [GR 92]. Let*

$$\Pi = \{p, q, r \leftarrow q, s \leftarrow p \wedge r\}$$

According to Gärdenfors and Rott, it is not clear which element of Π has to be given up in order to obtain the contraction $\Pi - s$. According to our approach, however, this is clear: only the deletion of the rule $s \leftarrow p \wedge r$ leads to an informationally minimal change. The deletion of q, which is supposed to be an equally good candidate according to Gärdenfors and Rott, violates the minimal change principle (1-2):

$$C^l(\{p, r \leftarrow q, s \leftarrow p \wedge r\}) = \{p\} \subset C^l(\{p, q, r \leftarrow q\}) = \{p, q, r\}$$

Example 4 (View Updates) *In the literature on 'view updates' in deductive databases it is common to ignore the minimal change principle in favour of the (hardly justifiable) idea that only facts should be changed but not rules. For instance, in order to 'delete' $q(c)$ from the database*

$$\Pi = \{p(c), r(d), q(x) \leftarrow p(x) \wedge r(d), s(x) \leftarrow p(x) \vee r(x)\}$$

two possibilities are considered: in the first one, $p(c)$ is deleted, and in the second one, $r(d)$ is deleted. Both changes are not minimal: clearly,

$$\Pi - \{p(c)\} < \mathbf{Upd}(\Pi, -q(c))$$

as well as

$$\Pi - \{r(d)\} < \mathbf{Upd}(\Pi, -q(c))$$

References

[Almukdad & Nelson 1984] A. Almukdad and D. Nelson: Constructible Falsity and Inexact Predicates, *JSL* 49:1 (1984), 231–233.

[Belnap 1977] N.D. Belnap: A Useful Four-valued Logic, in G. Epstein and J.M. Dunn (Eds.), *Modern Uses of Many-valued Logic*, Reidel 1977, 8–37.

[Cross & Thomason 1992] C.B. Cross and R.H. Thomason: Conditionals and Knowledge Base Update, in P. Gärdenfors (Ed.), *Belief Revision*, Cambridge University Press, 1992, 247–275.

[Gabbay 1985] D. Gabbay: Theoretical Foundations for Nonmonotonic Reasoning in Expert Systems, in K.R. Apt (Ed.), *Proc. NATO Advanced Study Institute on Logics and Models of Concurrent Systems*, Springer Verlag, 1985, 439–457.

[Gärdenfors 1988] P. Gärdenfors: *Knowledge in Flux*, MIT Press, Cambridge, 1988.

[GR 92] P. Gärdenfors and H. Rott: *Belief Revision*, Report 30, Konstanzer Berichte zur Logik und Wissenschaftstheorie, Universität Konstanz, 1992.

[Gelfond & Lifschitz 1990] M. Gelfond and V. Lifschitz: Logic Programs with Classical Negation, *Proc. ICLP 1990*, MIT Press, 1990.

[Hansson 1991] S.O. Hansson: *Belief Base Dynamics*, dissertation, Uppsala University, 1991.

[Levesque 1984] H.J. Levesque: Foundations of a Functional Approach to Knowledge Representation, *AI* 23:2, 1984, 155-212.

[Levesque 1988] H.J. Levesque: Logic and the Complexitiy of Reasoning, J. Philosophical Logic 17 (1988), 355–389.

[Levi 1977] I. Levi: Subjunctives, dispositions and chances, *Synthese* 34 (1977), 423–455.

[Miller 1989] D. Miller: A Logical Analysis of Modules in Logic Programming, *J. Logic Programming* 1989, 79–108.

[Pearce & Wagner 1990] D. Pearce and G. Wagner: Reasoning with Negative Information I - Strong Negation in Logic Programs, in L. Haaparanta, M. Kusch and I. Niiniluoto (Eds.), *Language, Knowledge and Intentionality*, Acta Philosophica Fennica 49, 1990.

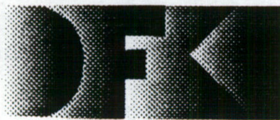
[Pereira & Alferes 1992] L.M. Pereira and J.J. Alferes: Wellfounded Semantics for Logic Programs with Explicit Negation, *Proc. ECAI'92*, Wiley, 1992.

[Reiter 1978] R. Reiter: On Closed-World Databases, in J. Minker and H. Gallaire (Eds.): *Logic and Databases*, Plenum Press, 1978.

[Wagner 1993] G. Wagner: *Vivid Logic - Knowledge-Based Reasoning with Two Kinds of Negation*, dissertation, Freie Universität Berlin, 1993, to appear as Springer LNAI.

[Wagner 1993a] G. Wagner: Reasoning with Inconsistency in Extended Deductive Databases, in L.M. Pereira and A. Nerode (Eds.), *Proc. 2nd Int. Workshop on Logic Programming and Nonmonotonic Reasoning*, MIT Press, 1993.

[Wagner 1993b] G. Wagner: Epistemic Modalities and the CWA in Disjunctive Knowledge Representation Systems, 1993, submitted.



Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
67608 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-37
Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer:
An Alternative to H-Subsumption Based on Terminological Reasoning
9 pages

RR-92-40
Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41
Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42
John Nerbonne:
A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster:
WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi:
Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51
Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52
Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-53
Werner Stephan, Susanne Biundo:
A New Logical Framework for Deductive Planning
15 pages

RR-92-54
Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55
John Nerbonne, Joachim Laubsch, Abdel Kader Diagne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56
Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58
Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59
Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
13 pages

RR-92-60
Karl Schlechta: Defaults, Preorder Semantics and Circumscription
19 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist:
Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi:
An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-04
Christoph Klauck, Johannes Schwagereit:
GGD: Graph Grammar Developer for features in CAD/CAM
13 pages

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: On Skolemization in Constrained Logics
40 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: Concept Logics with Function Symbols
36 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09
Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 Seiten

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf: Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-11
Bernhard Nebel, Hans-Juergen Buerckert:
Reasoning about Temporal Relations:
A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-12
Pierre Sablayrolles: A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-13
Franz Baader, Karl Schlechta:
A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-14
Joachim Niehren, Andreas Podelski,Ralf Treinen:
Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support Final Project Report
33 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-17
Rolf Backofen:
Regular Path Expressions in Feature Logic
37 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional μ-Calculus
32 pages

RR-93-20
Franz Baader, Bernhard Hollunder:
Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-22
Manfred Meyer, Jörg Müller:
Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy:
Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel:
Document Highlighting —
Message Classification in Printed Business Letters
17 pages

RR-93-26
Jörg P. Müller, Markus Pischel: The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-27
Hans-Ulrich Krieger:
Derivation Without Lexical Rules
33 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker: Feature-Based Allomorphy
8 pages

RR-93-33
Bernhard Nebel, Jana Koehler:
Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis
33 pages

RR-93-34
Wolfgang Wahlster:
Verbmobil Translation of Face-To-Face Dialogs
10 pages

RR-93-35
Harold Boley, François Bry, Ulrich Geske (Eds.):
Neuere Entwicklungen der deklarativen KI-Programmierung — *Proceedings*
150 Seiten

RR-93-36
Michael M. Richter, Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Gabriele Schmidt: Von IDA bis IMCOD: Expertensysteme im CIM-Umfeld
13 Seiten

RR-93-38
Stephan Baumann: Document Recognition of Printed Scores and Transformation into MIDI
24 pages

DFKI Technical Memos

TM-91-13
Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Busemann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh:
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben:
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

TM-93-01
Otto Kühn, Andreas Birk: Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

TM-93-02
Pierre Sablayrolles, Achim Schupeta:
Conlfict Resolving Negotiation for COoperative Schedule Management
21 pages

DFKI Documents

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.):
UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-22
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.):
Kooperierende Agenten
78 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann:
Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

D-92-28
Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-01
Philipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-02
Gabriele Schmidt, Frank Peters, Gernod Laufkötter: User Manual of COKAM+
23 pages

D-93-03
Stephan Busemann, Karin Harbusch(Eds.):
DFKI Workshop on Natural Language Systems: Reusability and Modularity - Proceedings
74 pages

D-93-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1992
194 Seiten

D-93-05
Elisabeth André, Winfried Graf, Jochen Heinsohn, Bernhard Nebel, Hans-Jürgen Profitlich, Thomas Rist, Wolfgang Wahlster:
PPP: Personalized Plan-Based Presenter
70 pages

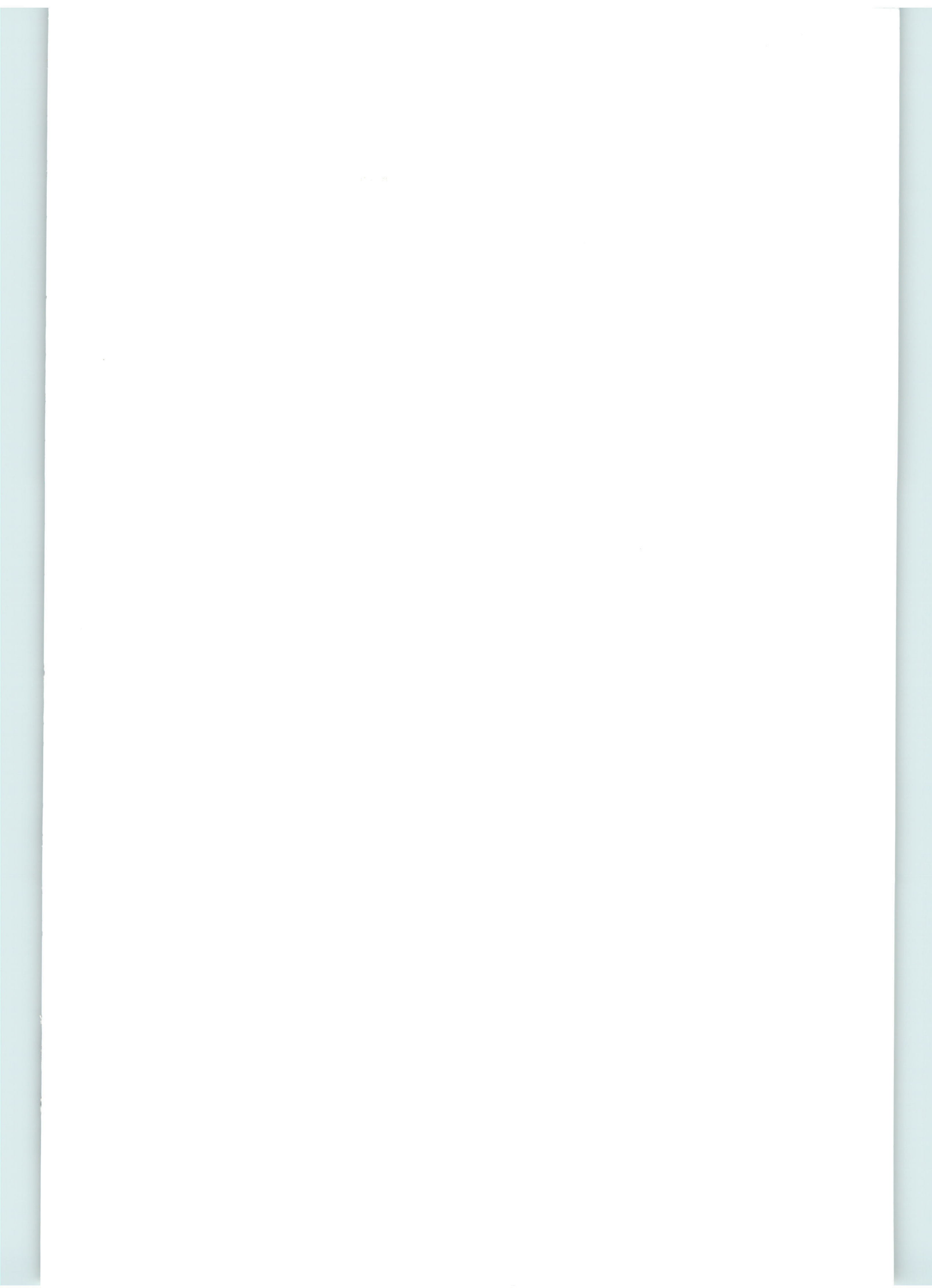
D-93-06
Jürgen Müller (Hrsg.):
Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz Saarbrücken 29.-30. April 1993
235 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-07
Klaus-Peter Gores, Rainer Bleisinger:
Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-08
Thomas Kieninger, Rainer Hoch: Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse
125 Seiten

D-93-09
Hans-Ulrich Krieger, Ulrich Schäfer:
TDL ExtraLight User's Guide
35 pages

D-93-12
Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, Werner Stein:
RELFUN Guide: Programming with Relations and Functions Made Easy
86 pages



Harold Boley, François Bry, Ulrich Geske (Eds.)

Research Report