Technische Universität Hamburg-Harburg
Arbeitsbereich Softwaresysteme

Entwickung und Umsetzung einer deduktiven Komponente für AlegroGraph

Bachelorarbeit im Studiengang Informationstechnologie

vorgelegt von

Oskar Maier

Matrikelnr. 20522714

betreut durch

Prof. Dr. Ralf Möller

10. Juli 2009

Erklärung

Ich, Oskar Maier, erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbstständig sowie ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.
Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Hamburg, den 10. Juli 2009

Oskar Maier

Danksagung

Ich danke Prof. Dr. Ralf Möller vom Institut für Softwaresysteme der Technischen Universität Hamburg-Harburg für die Betreuung dieser Bachelorarbeit. Außerdem danke ich Prof. Dr. Stefan Brass vom Institut für Informatik der Martin-Luther-Universität Halle-Wittenberg und Dr. Ulrich Zukowski von der Universität Passau, dass sie sich Zeit genommen haben, meine Fragen ausführlich zu beantworten. Aus der Korrespondenz mit ihnen sind wertvolle Denkanstöße hervorgegangen. Josephina Maier und Dr. Christine Müller danke ich für die viele Zeit, die sie in in das formale Korrekturlesen investiert haben; ohne sie wäre die Arbeit in dieser Form nicht möglich gewesen. Zum Schluss danke ich noch Juliane Mahro und Eva Bollen, die meine Launen und Zurückgezogenheit während der Entstehung dieser Arbeit geduldig ertragen haben.

Inhaltsverzeichnis

1 Einleitung

Mit der stetig vorangetriebenen Entwicklung von Semantic-Web-Applikationen steigt auch die Verbreitung maschinenlesbarer Inhalte im World Wide Web und die Zahl der Fachveröffentlichungen zu diesem Thema. Suchmaschinen wie Sindice[1], Falcons[2] und Swoogle[3] indizieren diese Inhalte und ermöglichen einfache Suchanfragen, während Semantic-Web-Browser Zugriff auf die Daten bieten. Als Formate dienen meist die W3C-Empfehlungen und de-facto-Standards RDF mit RDFS und OWL[4].
Auf dem Gebiet der maschinellen Verarbeitung der Daten und ihrer sinnvollen Auswertung ist die Entwicklung jedoch noch nicht weit fortgeschritten. Es stellt zwar keine Schwierigkeit mehr dar, Fakten als RDF-Tripel aus dem Internet zu erhalten. Diese jedoch zum Beispiel als Grundlage für Experten- oder Agentensysteme zu nutzen und abgeleitete Schlüsse daraus zu ziehen, erfordert meist eine eigene, situationsabhängige Programmierung. Werden die Informationen, auf die über Zugriff verfügt wird, jedoch als Datenbank von Tripeln betrachtet, kann ein deduktives Datenbanksystem eine Möglichkeit bieten, gezielt implizites Wissen aus den Daten zu extrahieren. In letzter Zeit wurde daran geforscht, eine Verbindung der OWL *Description Logic Programs (DLP)* und der eng verwandten Datalog *Description Horn Logic* zu erreichen. In der Veröffentlichung von Grosof et al. 2003 [GHVD03] wird OWL DLP als die Schnittmenge von OWL und Datalog definiert und damit die Möglichkeit eröffnet, die weiterentwickelten Schlussfolgerungsmechanismen von Datalog für Ontologien zu verwenden.
Die vorliegende Arbeit beschäftigt sich mit dem Entwurf und der Umsetzung einer deduktiven Komponente für das AlegroGraph Triple Storage System von Franz Inc., die im Folgenden mit AGDC bezeichnet wird. Ziel ist es, eine Präsentation der expliziten Daten durch implizite Regeln zu implementieren und damit die elegante Möglichkeit zu eröffnen, eine Teilmenge von OWL ausgedrückt als Datalogregeln zur Anwendung auf eine Tripelmenge zu verwenden. Die Problemstellung wird dabei rein unter dem Aspekt der Anfrageauswertung betrachtet, Datenbankänderungen werden als unabhängig von der deduktiven Komponente begriffen. Die zugrundeliegenden theoretischen Aspekte werden im Rahmen der vorliegenden Arbeit nur insofern angerissen, als sie zur Klärung der Sachverhalte und Entscheidungen notwendig sind.
Im nachfolgenden Abschnitt 2 der Arbeit wird eine kurze Einführung in die theoretische Basis von deduktiven Datenbanken gegeben. Daraufhin wird in Abschnitt 3 explizit auf die Besonderheiten eingegangen, die durch die Wahl von AlegroGraph als zugrundeliegendes Datenbankmanagementsystem beim Entwurf der deduktiven Komponente bedacht werden müssen. Zusätzlich werden Implementierungsgrundlagen erklärt. Abschnitt 4 definiert und beschreibt mit der Auswertungsstrategie das Herzstück von AGDC. Im Abschnitt 5 wird auf die implementierten Erweiterungen der Datalog Ausdrucksstärke eingegangen. In Abschnitt 6 wird die konkrete Implementierung vorgestellt. Abschließend werden in Abschnitt 7 die Ergebnisse einer Geschwindigkeitsevaluierung mit dem

[1] Sindice - The Semantic Web Index: http://sindice.com/, Stand: 17.06.2009
[2] Falcons Object Search: http://iws.seu.edu.cn/services/falcons/objectsearch/index.jsp, Stand: 17.06.2009
[3] Swoogle Semantic Web Search Web Engine: http://swoogle.umbc.edu/, Stand: 17.06.2009
[4] Siehe http://www.w3.org/RDF/, http://www.w3.org/TR/rdf-schema/ und http://www.w3.org/2004/OWL/ für eine Einführung. Stand 18.06.2009.

Lehigh University Benchmark besprochen. In Abschnitt 8 werden abschließend die Ergebnisse präsentiert.
Definitionen werden im Text mit •, Beispiele mit ◇ und Beweise mit □ abgeschlossen.

1.1 Grundlegendes, Richtlinien und Ziele

Das implementierte System baut auf AlegroGraph auf, eine der ausgereiftesten und schnellsten Triple-Datenbanken, die derzeit verfügbar sind[5]. Als Implementierungssprache wurde Common Lisp gewählt. Dafür spricht einerseits die starke Kopplung zwischen AlegroGraph und Alegro CommonLisp[6], andererseits die Eignung von Common Lisp als KI-Programmiersprache, ein Bereich, der Überschneidungen mit der Programmierung deduktiver Datenbanken aufweist.
EDB und IDB werden als Stores[7] realisiert, wodurch eine persistente Speicherung und ein einfacher und schneller Wechsel zwischen verschiedenen Regel- und vor allem Faktenmengen möglich ist.
Der Entwurf wird unter drei Gesichtspunkten mit unterschiedlicher Gewichtung angegangen: Neben der Zuverlässigkeit wird das Hauptaugenmerk auf die Geschwindigkeit der Anfragebeantwortung gelegt, danach folgt eine möglichst große Ausdrucksstärke des verwendeten Datalog und als letzter Punkt ein modularer Aufbau, um eine schnelle Adaption an zukünftige Änderungen von AlegroGraph zu ermöglichen und die Komponente in Zukunft um neue Konzepte erweitern zu können.
Ziel dieser Arbeit ist es, eine möglichst flexible deduktive Komponente für AlegroGraph zu entwerfen und zu implementieren. Ihre erlaubte Datalog Ausdrucksstärke soll ausreichen, um verschiedene Ansätze zur OWL-Datalog-Verbindung zu realisieren und die Schlussfolgerungen in einem akzeptablen Zeitrahmen abzuschließen.

2 Theorie

2.1 Datenbanksysteme

Datenbanksysteme (DBS) sind Systeme zur elektrischen Datenverwaltung und dienen der effizienten, wiederspruchsfreien und dauerhaften Speicherung großer Datenmengen. Sie bestehen meist aus den zwei Teilsystemen des Datenbankmanagementsystems (DBMS) und der Datenbank (DB), den eigentlichen Daten. Die bekannteste und am weitestern verbreitete Form der DBMS bilden die relationalen Datenbanksysteme (RDBMS), die auf dem Modell der relationalen Algebra aufbauen.

2.2 Deduktive Datenbanksysteme

Deduktive Datenbanken erweitern das relationale Modell um eine deduktive Komponente. Unter Verwendung von Logikprogrammierung kann über die Aus-

[5] Stand 06.2009
[6] AlegroGraph stellt auch Schnittstellen für andere Programmiersprachen wie Java und standardisierte wie SPARQL bereit.
[7] Die AlegroGraph Datenbankbezeichnung

wertung von Deduktionsregeln weiteres "Wissen" aus den in der DB gespeicherten Fakten gewonnen werden.
So können Expertensysteme moduliert werden, die auf Basis von Expertenwissen, das in Form von Regeln ausgedrückt wird, Lösungen zu oder Bewertungen für bestimmte Problemstellungen liefern.

Definition 2.1 (Expertensystem): Ein *Expertensystem* ist ein Softwaresystem, das unter Verwendung einer Wissensbasis, die Fachwissen eines bestimmten Bereiches enthält, dieselben Schlüsse bezüglich fachspezifischer Problemstellungen ziehen kann, wie es auch einem Experten in diesem Bereich möglich wäre.

•

Die explizites Wissen beschreibenden Regeln der DB, auch Fakten genannt, werden unter dem Begriff der *Extensional Database* (EDB) zusammengefasst, während die Menge der implizites Wissen repräsentierenden Regeln als *Intensional Database* (IDB) bezeichnet wird. Auf diese Einteilung wird im Kapitel 2.4 über Datalog noch genauer eingegangen. Ein *Inferenz-* oder *Schlussfolgerungsmechanismus* verarbeitet die Regeln und kann logische Schlussfolgerungen aus ihnen ziehen.

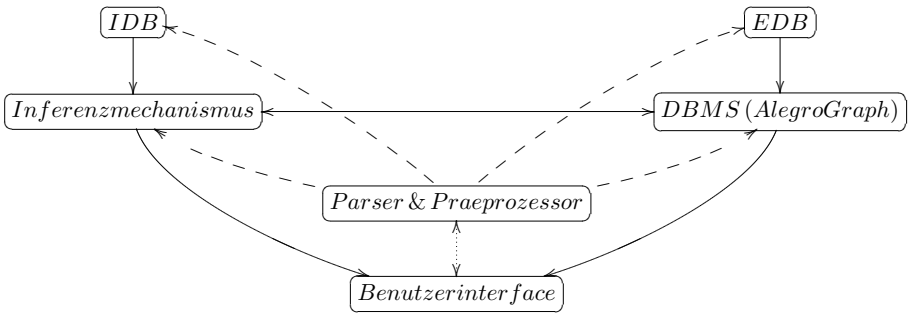

Abbildung 1: Skizze der Software-Architektur einer deduktiven Komponente (nach dem Buch von Nussbaum 1991 [Nus91, Preface, ix])

2.2.1 Datensicherheit, Transaktionssicherheit und Datenintegrität

Häufig geforderte Eigenschaften von DBMS sind Datensicherheit, der Schutz gegen unerlaubten Zugriff und Datenverlust, Transaktionssicherheit, die Verhinderung von undefinierten, zufälligen Zuständen im Mehrbenutzerbetrieb und Datenintegrität, bei der die Integrität der Daten durch Constraints sichergestellt wird.
Da sich die deduktive Komponente nur auf das Auslesen von Daten auswirkt, werden die vom zugrundeliegenden DBMS, AlegroGraph, bereitgestellten Datensicherheitskonzepte nicht beeinflusst. Auf Transaktionssicherheit im Mehrbenutzerbetrieb wird im Rahmen dieser Arbeit nicht eingegangen, d.h. wenn eine Datenänderung während des Deduktionsprozesses stattfindet, können die ausgegebenen Daten fehlerhaft sein, da es sich bei der Auswertung einer Anfrage durch die deduktive Komponente nicht um eine atomare Operation handelt.

Beispiel 2.1: Vorhandene Datentupel in der IDB:

$(1, 2)$
$(1, 3)$

Benutzer A liest Datensätze aus der Datenbank. Dabei erhält er die Tupel einzeln, vergleicht sie mit seiner bisherigen Ergebnismenge und fügt sie, falls sie in dieser noch nicht existieren, hinzu. Tabelle 1 zeigt diesen Vorgang schrittweise auf. Benutzer A erhält so die Tupelmenge $R = \{(1, 2), (1, 3), (1, 4)\}$ als Ergebnis

Schritt	Benutzer A	Benutzer B
I	`read(1,2)`	
II	`read(1,3)`	
II		`alter(1,2)set(1,4)`
IV	`read(1,4)`	

Tabelle 1: Beispielhafter Ablauf eines unsicheren Auslesevorganges

der Anfrage, obwohl sich in der Datenbank nur zwei Werte, und zwar $(1, 3)$ und $(1, 4)$ befinden. ◇

Datenintegrität, in Bezug auf Datenbanken, wird von den meisten modernen RDBMS unterstützt. Sie lässt sich in die Kategorien Entititäts- (UNIQUE, PRIMARY KEY, ...), Domänen- (FOREIGN KEY, DEFAULT, ...), referenzielle (*tabellenübergreifende*) und benutzerdefinierte Integrität partitionieren. Es wird dabei garantiert, dass keine neuen Daten in die Datenbank eingepflegt werden können, die eine der Restriktionen verletzen würden. Da sich dieser Mechanismus allerdings ausschließlich auf den Schreibkontext bezieht, kann er im Rahmen dieser Arbeit unberücksichtigt bleiben.

2.3 Prädikatenlogik

Das Datenmodell deduktiver Datenbanken wird von der einsortigen[8], klassischen[9] Prädikatenlogik erster Stufe gebildet, welche zur Beschreibung von Daten und Operationen über diese dient. Logik erster Stufe setzt sich aus Objekten mit individuellen Identitäten und Eigenschaften und aus Relationen zwischen diesen zusammen. Als beschreibendes Element dienen Formeln, mit denen Daten beschrieben, Regeln definiert und Anfragen formuliert werden können.
Die gesamte Ausdrucksstärke der Prädikatenlogik wird für die vorliegende Aufgabe nicht benötigt und deshalb beschränkt, um Entscheidbarkeit zu erhalten und den Rahmen der Arbeit nicht zu überschreiten. So wird auf eine Teillogik, die *funktionsfreie kausale Logik*, wie sie im Buch Paton et al. 1996 [PCWT96, S. 39] definiert wird, zurückgegriffen.

[8] Bei der mehrsortigen Prädikatenlogik werden den Termen von Formeln Klassen (sog. Sorten) zugeordnet, die die Unifikation verhindern, solange beide Terme nicht von derselben Sorte sind. Ein Datenmodell unter Verwendung der Sortenlogik kann im Buch von Cremers et al. 1994 [CG94] gefunden werden, die theoretische Grundlage im Buch von Walther 1987 [Wal87].

[9] Das System ist zweiwertig und der Wahrheitswert von zusammengesetzten Aussagen ist durch die Wahrheitswerte der Unteraussagen eindeutig bestimmt.

Des Weiteren muss die betrachtete Welt beschränkt werden. Durch eine Datenbank wird ein für den Anwender relevanter Weltausschnitt moduliert. Dieser wird im Folgenden mit dem Ausdruck *Diskursuniversum* bezeichnet.
Der Aufbau und die innere Struktur der Formeln wird im Abschnitt 2.3.1 *„Syntax“* behandelt. Der Abschnitt 2.3.2 *„Semantik“* beschreibt die Regeln zur Zuweisung von Wahrheitswerten. Mit der automatischen Feststellung von Wahrheitswerten bzw. der Herleitung von Formeln aus Formelmengen beschäftigt sich der dritte Abschnitt 2.3.3 unter der Überschrift *„Deduktionssystem“*. In seiner Aufteilung orientiert sich dieses Kapitel stark an der im Buch von Cremers et al. 1994 [CG94] verwendeten, während die Begrifflichkeiten zumeist aus dem Buch Paton et al 1996 [PCWT96] übernommen wurden.

2.3.1 Syntax

Zur Definition einer formalen Sprache muss zuerst der verwendbare Zeichenvorrat festgelegt werden. Aus diesen Symbolen werden dann die Terme und mit Hilfe dieser schließlich die Formeln gebildet. Die Menge aller Formeln bildet eine Sprache der Prädikatenlogik erster Stufe. Man beachte, dass die Gesetze der Prädikatenlogik nur gelten, wenn das betrachtete Diskursuniversum mindestens ein Element enthält. Im Folgenden werden als erstes die Grundelemente der Prädikatenlogik definiert.

Definition 2.2 (Konstante): Eine *Konstante* oder ein *Eigenname* ist ein innerhalb des Diskursuniversums eindeutiger, unveränderlicher Wert, der genau ein Individuum unverwechselbar bezeichnet.
M_e sei die Menge aller Konstanten des Diskursuniversums und $\varsigma(x)$ das Individuum, das durch die Konstante x bezeichnet wird. Dann gilt:

$$\forall x, y \in M_e : x = y \Leftrightarrow \varsigma(x) = \varsigma(y)$$

•

Definition 2.3 (Variable): Eine *Variable* ist ein Symbol, das als Platzhalter für ein beliebiges Individuum des Diskursuniversums dient. •

Definition 2.4 (Term): Ein *Term* ist eine Variable oder eine Konstante. Terme werden für die Beschreibung von Individuen des Diskursuniversums verwendet. •

Variablen und Konstanten dürfen sich aus einer beliebigen Kombination alphanumerischer Zeichen[10] zusammensetzen. Variablen müssen mit einem großen alphabetischen Buchstaben beginnen. Konstanten können in Anführungszeichen (") eingeschlossen sein. Ist dies nicht der Fall und beginnen sie mit einem alphabetischen Buchstaben, so muss dieser in Kleinschreibweise vorliegen, um eine Unterscheidung zu den Variablen zu ermöglichen.
Diese Vorgaben entsprechen den in der Literatur üblichen. Für das in dieser Arbeit verwendete Datalog gilt eine andere Konvention, die in Abschnitt 3.5 beschrieben wird.

[10] Dies entspricht der Lisp Konvention für Symbole.

Definition 2.5 (Prädikat): Ein *Prädikat* besteht aus einem Prädikatensymbol und einer Reihe von definierten Leerstellen und wird zu einer - `wahren` oder `falschen` - Aussage reduziert, wenn die Leerstellen mit Konstanten besetzt werden. Die Stelligkeit des Prädikates ergibt sich aus der Anzahl der unterschiedlichen Leerstellen. •

Für Prädikatensymbole steht der gleiche Zeichensatz wie für die Terme zur Verfügung, wobei allerdings keine Regel zur Groß- und Kleinschreibung des ersten Zeichens existiert, da die Prädikatensymbole anhand ihrer Position als solche identifiziert werden können.

Definition 2.6 (Atom): Ein *Atom* ist ein Audruck der Form

$$\mathtt{p}(\mathtt{t}_1, ..., \mathtt{t}_n)$$

wobei $\mathtt{p}$ ein n-stelliges Prädikatensymbol darstellt, während $\mathtt{t}_1, ..., \mathtt{t}_n$ Terme sind und $n \geq 0$. Ein variablenfreies Atom wird auch als Grundatom bezeichnet. •

Definition 2.7 (Literal): Ein *Literal* ist ein negiertes oder nicht-negiertes Atom. •

Ein negiertes Atom (auch negatives Literal) wird im Folgenden in der Form

$$\neg\mathtt{p}(\mathtt{t}_1, ..., \mathtt{t}_n)$$

oder

$$\mathtt{not\ p}(\mathtt{t}_1, ..., \mathtt{t}_n)$$

dargestellt.

Definition 2.8 (Gültige Formel): Eine *Gültige Formel* oder *Wohldefinierte Formel* ist jede Formel, die sich aus gültigen Formeln und logischen Verknüpfungen zusammensetzt. *Gültige Verknüpfungen* sind dabei $\neg$ (Negation), $\vee$ (Disjunktion), $\wedge$ (Konjunktion), $\Rightarrow$ (Implikation) und $\Leftrightarrow$ (Äquivalenz) sowie die *logischen Quantoren* $\forall$ (Allquantor) und $\exists$ (Existenzquantor). Ein Atom ist eine Gültige Formel, ebenso wie die *aussagenlogischen Konstanten* `wahr` und `falsch`. •

In der vorliegenden Arbeit wird nur auf eine einfache Form der Formeln zurückgegriffen, die als Bedingung bekannt ist.

Definition 2.9 (Bedingung, Fakt): Eine *Bedingung* oder auch *Eindeutige Schlussfolgerung* ist eine Formel der Ausprägung

$$\mathtt{p} \Leftarrow \mathtt{q}_1 \wedge ... \wedge \mathtt{q}_k$$

wobei $\mathtt{p}$ ein Atom ist, $\mathtt{q}_1 \wedge ... \wedge \mathtt{q}_k$ Literale und $k \geq 0$. Eine Bedingung mit leerer rechter Seite ($k = 0$) impliziert, dass $\mathtt{p}$ immer `wahr` und wird als *Fakt* bezeichnet.

•

Diese Bedingung kann gedeutet werden als "$\mathtt{p}$ ist wahr, wenn alle $\mathtt{q}_1$ und ... und

$\mathtt{q_k}$ wahr sind". Im Folgenden werden Bedingungen in Anlehnung an die Syntax von Datalog in der Form

$$\mathtt{p} :- \mathtt{q_1}, ..., \mathtt{q_k}$$

angegeben, wobei ": –" als links impliziert ($\Leftarrow$) und "," als und ($\wedge$) angesehen werden muss. Die linke Seite ($\mathtt{p}$) wird dabei als *Kopf*, die rechte Seite ($\mathtt{q_1}, ..., \mathtt{q_k}$) als *Körper* der Bedingung bezeichnet. Die einzelnen Literale des Körpers $\mathtt{q_i}$ ($1 \leq i \leq k$) heißen auch *Unteranfragen* bzw. *Unterziele*.

Definition 2.10 (Logikprogramm): Ein *Logikprogramm* ist eine endliche Menge von Bedingungen. •

Die beschriebene, kausale Logik ist äquivalent zum konventionellen Prädikatenkalkül. Jede Prädikatenkalkülformel kann in eine Menge mit einer oder mehreren Bedingungen umgewandelt werden, während jede Bedingung wiederum eine Formel des Prädikatenkalküls ist wie im Buch Clocksin et al. 2003 [CM03, Kapitel 10] gezeigt wird.

2.3.2 Semantik

Während die Syntax die äußere Form der Formel beschreibt, ordnet die Semantik ihr eine Bedeutung zu. Über Interpretation wird den Formeln ein Wahrheitswert, d.h. eine aussagenlogische Konstante `wahr` bzw. `falsch`, zugewiesen, wobei dieser Vorgang als *Reduktion* bezeichnet wird. Die Semantik eines Logikprogramms wird über Herbrand-Modelle und die Herbrand-Interpretationen definiert, auf Interpretationen und Modelle im Allgemeinen wird im Rahmen dieser Arbeit nicht eingegangen.

Definition 2.11 (Grundterm, Grundatom): Ein *Grundterm* bzw. *Grundatom* ist ein Term bzw. Atom ohne Variablen. Im Falle der Terme ist die Menge aller Grundterme äquivalent mit der Menge aller Konstanten eines Logikprogramms.[11] •

Die folgenden Definitionen von Herbrand-Universum, -Basis, -Interpretation und -Modell beziehen sich auf Logikprogramme.

Definition 2.12 (Herbrand-Universum): Das *Herbrand-Universum* U_p eines Logikprogramms P ist die Menge aller Grundterme. •

Definition 2.13 (Herbrand-Basis): Die *Herbrand-Basis* B_p eines Logikprogramms P ist die Menge aller Grundatome, die aus der Menge aller Prädikatensymbole in P und den Grundtermen von U_p gebildet werden können.

•

Definition 2.14 (Herbrand-Interpretation): Eine *Herbrand-Interpretation* $\mathcal{I}_p$ eines Logikprogramms P ist eine Zuweisung der aussagenlogischen Konstanten `wahr` oder `falsch` an jedes Element von B_p und repräsentiert einen mög-

[11] Diese Aussage ist nur im betrachteten Fall der funktionsfreien kausalen Logik gültig.

lichen Zustand des Diskursuniversums. Besonders interessant für diese Arbeit sind hier die Herbrand-Interpretationen I_p, in denen jedem Grundatom der Wert `wahr` zugewiesen ist. •

Definition 2.15 (Substitution, Instanz): [12] Eine *Substitution* θ ist eine endliche Menge der Form $\{v_1 \mapsto t_1, ..., v_n \mapsto t_n\}$, wobei $v_1, ..., v_n$ Variablen und $t_1, ..., t_n$ Konstanten sind. Zudem sind $v_1, ..., v_n$ verschieden. E_θ, die *Instanz* des Ausdruckes E mit der Substitution θ, wird aus E gebildet, indem jedes Vorkommen der Variable v_i durch die Konstante t_i ersetzt wird. Ist E_θ dann variablenfrei, wird es als Grundinstanz bzw. als instanziiert bezeichnet. Eine Variable, die durch eine Konstante substituiert wurde, wird ebenfalls als instanziiert bezeichnet. •

Definition 2.16 (Herbrand-Modell): I_p sei eine Herbrand-Interpretation von P. C ist die Menge aller Bedingungen in P. G ist die Menge aller Grundinstanzen in P und G_c die Menge aller Grundinstanzen einer Bedingung $c \in C$ in P. L_c ist Menge aller Literale im Körper und H_c der Kopf dieser Bedingung c.

$$\forall g \in G_c : g = true \Rightarrow c = true$$
$$\exists l \in L_c : l = false \vee H_c = true \Rightarrow c = true$$
$$a \in B_p \wedge a \in Ip \Rightarrow a = true$$

I_p ist dann ein *Herbrand-Modell* von P, wenn alle Bedingungen in P wahr sind $\forall c \in C : c = true$. •

Für jedes Programm existieren mehrere Herbrand-Modelle. Für das weitere Vorgehen interessant ist dabei das minimale Modell, das im Folgenden als Minimalmodell bezeichnet wird. Es stellt die Antwort auf eine Anfrage über einem Logikprogramm dar.

Definition 2.17 (Minimalmodell): Das *Minimalmodell* ist das minimalste aller Herbrand-Modelle eines Programms. Es kann gebildet werden, indem die Schnittmenge aller Herbrand-Modelle gebildet wird.
Sei M_H die Menge aller Herbrand-Modelle eines Logikprogrammes P und M_m das zugehörige Minimalmodell. Dann gilt $M_m = M_1 \cup ... \cup M_i$ mit $M_i \in M_m$. •

Die vorgenommene Einschränkung der verwendeten Formeln auf Bedingungen, wie sie in Definition 2.9 definiert wird, hat zur Folge, dass die verwendeten Logikprogramme keine indefiniten Formeln enthalten können[13] und es kann deshalb bei den weiteren Überlegungen von definiten Logikprogrammen ausgegangen werden. Es kann gezeigt werden, dass im Falle eines definiten Logikprogrammes ohne Negation ein einziges, eindeutiges Minimalmodell existiert.
Die vorgestellte Variante der kausalen Logik bedient sich der Annahme einer geschlossenen Welt. In reiner Logik gibt es drei mögliche Zustände für eine Bedingung: `wahr`, `falsch` und `undefiniert`. Dies ergibt sich aus der Betrachtung einer offenen Welt. Wenn kein explizites Statement über den Wahrheitswert einer Aussage bekannt ist, kann ihr auch keine aussagenlogische Konstante zu-

[12] Aus Schmidt 1991 [Sch91] entlehnt.
[13] Eine solche wäre beispielsweise $\mathtt{P_1} \vee \mathtt{P_2} \leftarrow \mathtt{Q_1} \wedge ... \wedge \mathtt{Q_n}$.

gewiesen werden. Dieses Problem durch die Annahme einer geschlossenen Welt eliminiert.

Definition 2.18 (Annahme einer geschlossenen Welt): Die *Annahme einer geschlossenen Welt* sagt aus, dass wenn zu einer Aussage kein Fakt gefunden oder hergeleitet werden kann, diese die aussagenlogische Konstante `falsch` zugewiesen bekommt. •

Der folgende Abschnitt beschäftigt sich mit dem Problem, wie Bedingungen aneinander angeglichen und Folgerungen aus ihnen gezogen werden können. Dies stellt den ersten Schritt zum Erhalt des Minimalmodells dar, umgesetzt als Auswertungstrategie und dargelegt in Abschnitt 4 und Folgenden.

2.3.3 Deduktionssystem

Der große Vorteil deduktiver Datenbanken gegenüber relationaler ist, dass sie nicht nur explizites, sondern auch implizites Wissen darstellten können. Um dies zu erreichen, fehlt noch ein Mechanismus, um aus der gegebenen Formelmenge eines Programms und den expliziten Fakten der EDB dieses implizite Wissen abzuleiten.
In diesem Abschnitt wird eine als *Deduktionssystem* bezeichnete Regelmenge vorgestellt, die es ermöglicht, aus den gegebenen Formeln des Diskursuniversums neue Formeln abzuleiten. Dies wird erreicht durch das Einsetzen von Konstanten bzw. Variablen für Variablen, der Substitution und das Anwenden schon vorhandener Formeln. Es bildet den Kern des in Abbildung 1, S.3 angeführten Inferenzmechanismus.
Der erste Schritt hin zur Folgerung einer Formel aus einer Menge anderer Formeln ist die syntaktische Angleichung der beteiligten Atome. Dies wird durch Substitution erreicht und im Folgenden als Unifikation bezeichnet. Von der in 2.15 gegebenen Definition der Substitution weicht die Folgende dahingehend ab, dass sie einen allgemeineren Fall beschreibt. Während vorher nur Variablen durch Konstanten substitutiert wurden, wird nun auch die Substitution von Variablen durch andere Variablen beschrieben. Ersterer Fall wird im Folgenden als Grundsubstitution bezeichnet, letzterer als Variablenumbenennung.

Definition 2.19 (Substitution allgemein, Grundsubstitution, Variablenumbenennung): Sei M_v die Menge aller Variablen und M_k die Menge aller Konstanten des Diskursuniversums U. S_v sei eine beliebige Variablenmenge mit $S_v \subseteq M_v$ und $M_t := M_k \cup M_v$ die Vereinigungsmenge von M_v und M_k, also die Menge aller Terme des Diskursuniversums.
Eine allgemeine *Substitution* σ ist eine Abbildung $\sigma : S_v \rightarrow M_t$. Wenn $\sigma : S_v \rightarrow M_k$ gilt, also alle Variablen von S_v auf Konstanten abgebildet werden, wird sie *Grundsubstitution* bezeichnet. Gilt hingegen $\sigma : S_v \rightarrow M_v$, so wird sie *Variablenumbenennung* genannt und die Notation ϵ statt σ verwendet. Grundsubstitutionen werden, wie schon in Definition 2.15 festgelegt, weiterhin mit θ bezeichnet. Für die Anwendung einer Substitution auf einen Ausdruck E wird die Form $E\sigma$ statt $\sigma(E)$ gewählt. •

Beispiel 2.2: Die folgenden Bedingungen und Fakten werden betrachtet

$\texttt{grosseltern}(\texttt{X},\texttt{Y}) :- \texttt{eltern}(\texttt{X},\texttt{Z}), \texttt{eltern}(\texttt{Z},\texttt{Y}).$
$\texttt{eltern}(\texttt{mary},\texttt{peter}).$

Aus der Anwendung der Substitution zur Angleichung von $\texttt{eltern}(\texttt{X},\texttt{Z})$ und $\texttt{eltern}(\texttt{mary},\texttt{peter})$ folgt $\sigma = \{X \mapsto mary, Y \mapsto peter\}$ und somit die Bedingung $\texttt{grosseltern}(\texttt{mary},\texttt{Y}) :- \texttt{eltern}(\texttt{mary},\texttt{peter}), \texttt{eltern}(\texttt{peter},\texttt{Y})$. ◇

Wenn eine Variablenumbenennung durchgeführt wird, sind die beteiligten Ausdrücke zwar syntaktisch gleich[14], jedoch noch nicht instanziiert, da den Variablen keine Konstanten zugeordnet sind. Um eine Formel zu reduzieren und somit schließlich das Minimalmodell zu erhalten, muss sie jedoch instanziiert sein. Dazu muss es möglich sein, Substitution auch auf bereits substituierte Ausdrücke anzuwenden, um u.a. mehrfache Ableitungen zu ermöglichen.

Definition 2.20 (Komposition von Substitutionen): $E\sigma_1\sigma_2$ bezeichnet die Anwendung zweier Substitutionen auf einen Ausdruck. Dabei wird zuerst σ_1, dann σ_2 angewendet. Also $E\sigma_1\sigma_2 := (E\sigma_1)\sigma_2$. •

Wie schon in Definition 2.15 festgelegt, wird $E\sigma$ als Instanz des Ausdruckes E bezeichnet, wenn die Variablen in E durch σ instanziiert, d.h. ihnen Konstanten zugeordnet wurden. Eine Reihe von Substitutionen, die zwei Ausdrücke E_1 und E_2 gleichmachen, wird als Unifikator bezeichnet. Existiert eine solche, existieren automatisch unendlich viele. Im Folgenden wird nur die kleinste Komposition von Substitutionen, die diese Wirkung entfaltet, als Unifikator bezeichnet.

Definition 2.21 (Unifikator): Existiert zu zwei Ausdrücken E_1 und E_2 eine Komposition von Substitutionen σ_{n1}, die sich aus k_1 Einzelsubstitutionen zusammensetzt, so dass $E_1 = E_2\sigma_{n1}$ gilt, wird diese als Unifikator von E_1 und E_2 bezeichnet, wenn keine andere Komposition von Substitutionen σ_{n2} mit $E_1 = E_2\sigma_{n2}$ und $k_2 < k_1$ existiert. •

Beispiel 2.3: Die folgende Liste von Atomen soll per Unifikation auf syntaktische Gleichheit gebracht werden.

$\texttt{P}_1 : \texttt{p}(\texttt{X},\texttt{Y},\texttt{Z})$
$\texttt{P}_2 : \texttt{p}(\texttt{Z},\texttt{X},\texttt{Y})$
$\texttt{P}_3 : \texttt{p}(\texttt{A},\texttt{B},\texttt{delta})$
$\texttt{P}_4 : \texttt{q}(\texttt{X},\texttt{Y},\texttt{Z})$

Für das Atom P_4 ist die Unifikation mit den anderen nicht möglich, da sich die Prädikatsymbole unterscheiden. Eine Unifikation von P_1 und P_2 kann mit Hilfe der Substitution $\sigma_1 = \{X \mapsto Z, Y \mapsto X, Z \mapsto Y\}$ erreicht werden, wobei es sich um eine Variablenumbenennung handelt. Zum Angleichen von P_2 an P_3 dient der Unifikator $\sigma_1 = \{Z \mapsto A, X \mapsto B, Y \mapsto delta\}$. Hierbei handelt es sich um eine Mischform von Grundsubstitution und Variablenumbenennung.

[14] Zwei Ausdrücke oder Anweisungen werden als syntaktisch gleich bezeichnet, wenn sie denselben Ableitungsbaum besitzen, d. h. wenn die abstrakte Syntax gleich ist. Oder auch: Uniformität.

Zusammenfassend wird also $P_3 = P_2\sigma_1 = P_1\sigma_1\sigma_2$ und somit die gewünschte Gleichheit erreicht. $\diamond$

Jetzt liegt ein Werkzeug vor, um Formeln mit gleichen und gleichstelligen Prädikatensymbolen anzugleichen. Die Unifikation ist eine zentrale Operation für das erforderliche automatische Beweisen für deduktive Datenbanken. Mit ihnen können die Formeln des Diskursuniversums umgeformt, ihr Zusammenhang hergestellt und sie dahingehend abgeleitet werden, dass eine Anfrage beantwortet werden kann.
Der in der Literatur als *Bottom-up* bezeichnete, mengenorientierte Ansatz der Auswertung stellt die einfachste Methode dar, alle Ausprägungen der Formeln zu erhalten. Er berücksichtigt zuerst, wie der Name schon impliziert, von unten ausgehend die bereits instanziierten Formeln, die sich größtenteils aus den Fakten zusammensetzen. Mit Hilfe der Unifikation werden weitere Formeln instanziiert. Diese können dann wieder zu Instanziierung weiterer dienen, usw., bis schließlich alle möglichen Ausprägungen aller Formeln erzeugt wurden.

Beispiel 2.4: Dieses Beispiel befasst sich mit einer naiven Anwendung der *Bottom-up*-Methode der Auswertung. Dabei wird für alle Körperprädikate aller Formeln, die mit den Köpfen schon instanziierter Formeln unifiziert werden können, genau dieser Schritt durchgeführt. Die durch die Ableitung neu entstandenen Formeln werden der Formelmenge hinzugefügt. Diese Schritte werden so lange wiederholt, bis sich die Formelmenge nicht mehr verändert, woraufhin das Minimalmodell erreicht ist.

Programm:	*Fakten:*	*Zielanfrage:*
$\mathtt{p(X,Y)} : -\mathtt{r(X,Z)}, \mathtt{s(Z,Y)}.$	$\mathtt{u(1,2)},\ \mathtt{u(2,4)},\ \mathtt{t(5,3)},$	$? - \mathtt{p(X,Y)}.$
$\mathtt{q(X,Y)} : -\mathtt{t(X,Y)}.$	$\mathtt{t(6,8)},\ \mathtt{r(7,4)}$	
$\mathtt{s(X,Y)} : -\mathtt{u(Y,X)}.$		

Die Menge aller Formeln F enthält zu Anfang alle Formeln des Programms und alle Fakten.

Schritt I: Erster Durchlauf.

1. Die Menge aller instanziierten Formeln von F mit dem Kopfprädikat $\mathtt{u}$, genauer $\mathtt{u(1,2)}$ und $\mathtt{u(2,4)}$, sind mit dem Körperprädikat $\mathtt{u(Y,X)}$ von $\mathtt{s}$ unifizierbar. Die dadurch ableitbaren Fomeln $\mathtt{s(2,1)} : -\mathtt{u(1,2)}.$ und $\mathtt{s(4,2)} : -\mathtt{u(2,4)}.$ werden der Formelmenge F hinzugefügt.
2. Gleiches gilt für die instanziierten Formeln mit dem Kopfprädikat $\mathtt{t}$ und der Formel $\mathtt{q(X,Y)} : -\mathtt{t(X,Y)}.$. Es werden $\mathtt{q(5,3)} : -\mathtt{t(5,3)}.$ und $\mathtt{q(6,8)} : -\mathtt{t(6,8)}.$ zu F hinzugefügt.
3. $\mathtt{r(7,3)}$ unifiziert mit $\mathtt{r(X,Y)}$ aus $\mathtt{p}$ und so erhält man $\mathtt{p(7,Y)} : -\mathtt{r(7,4)}, \mathtt{s(4,Y)}.$.

Schritt II: F hat sich geändert, die Auswertung wird fortgesetzt.

1. Die vorhandenen Instanzen von $\mathtt{u}$, $\mathtt{t}$ und $\mathtt{r}$ führen nicht mehr zu neuen Formeln.
2. Die nun instanziierten $\mathtt{q}$ sind mit keinem Körperprädikat unifizierbar.

3. Eine der Auprägungen von $\mathtt{s}$, genauer $\mathtt{s}(4,2)$, unifiziert mit der Formel $\mathtt{p}(7,\mathtt{Y}) :- \mathtt{r}(7,4), \mathtt{s}(4,\mathtt{Y})$.. Für $\mathtt{s}(2,1)$ ist dies nicht der Fall, da $\mathtt{s}(4,\mathtt{Y})$ schon teilinstanziiert ist und die zugeordneten Konstanten sich nicht gleichen. Dier erhaltene Regel ist die Instanz $\mathtt{p}(7,2) :- \mathtt{r}(7,4), \mathtt{s}(4,2)$..

Schritt III: F hat sich geändert, die Auswertung wird fortgesetzt.

1. Keine neue Regel kann abgeleitet werden.

Schritt IV: F hat sich nicht geändert, die Auswertung terminiert.

Die Menge F enthält nun alle möglichen Ausprägungen aller Regeln des Programms und mit ihr kann die Anfrage komplett beantwortet werden:
Frage: $? - \mathtt{p}(\mathtt{X},\mathtt{Y})$.
Antwort: $\mathtt{p}(7,2)$. ◇

Wie aus dem Beispiel 2.4 hervorgeht, werden bei einer naiven Anwendung der Bottom-Up-Methode auch Formeln instanziiert, die von keiner Relevanz für die gestellte Zielanfrage sind, wodurch unnötig Prozessor- und Speicherresourcen belegt werden. Es wäre wünschenswert, bei der Anfragebearbeitung nur die Formeln, von denen die Beantwortung der Zielanfrage abhängt zu berücksichtigen. Der in dieser Arbeit verwendete Ansatz verwendet die semi-naive Bottom-up Auswertung, die in Abschnitt 4.1 beschrieben wird und bringt mit Generalized Supplementary Magic Sets in Abschnitt 4.2 eine Programmtransformationsmethode zum Einsatz, die Ansätze der *Top-down*-Methode der Auswertung mit einbringt.
Die Anfragesprache, die für dieses Projekt verwendet wird, nennt sich *Datalog* und basiert auf dem vorgestellten Datenmodell. Im folgenden Kapitel wird ihre Syntax und Semantik beschrieben und auf notwendige Beschränkungen ihrer Ausdrucksstärke eingegangen.

2.4 Datalog

Datalog ist eine Abfragesprache für Datenbanken mit großer Ähnlichkeit zu Prolog. Mit ihr lassen sich Anfragen an ein definites, deduktives Datenbanksystem stellen. Gleichzeitig kann sie zur Formulierung von Bedingungen, also dem Aufbau der IDB, und der Darstellung von Fakten, also der EDB, dienen. Das für diese Arbeit gewählte Datalog ist funktionsfrei und nicht disjunktiv. Rekursion kann in Datalog durch deduktive Abgeschlossenheit verarbeitet werden.
Die im vorangegangenen Kapitel beschriebende Logik enspricht der der von Datalog verwendeten, der Aufbau dieses Kapitels ist deshalb gleich und ermöglicht es, direkte Paralellen zu ziehen.

2.4.1 Syntax

Ein *Datalogprogramm* ist eine endliche Menge an *Datalogregeln*. Diese werden in der Form

$$\mathtt{a}(\mathtt{X},\mathtt{Y}) :- \mathtt{b}(\mathtt{X},\mathtt{Z}), \mathtt{c}(\mathtt{Z},\mathtt{Y}).$$

dargestellt. Sie entsprechen den Bedingungen des vorangegangenen Kapitels und werden kurz als *Regeln* bezeichnet. Die linke Seite wird als *Kopf*, die rechte

als *Körper* der Regel bezeichnet, wobei zwischen Regeln mit leerem Körper[15] (*Fakten*) und solchen mit Körper unterschieden wird. Die Menge der Ersteren bildet die *Extensional Database* (EDB), während sich die *Intensional Database* (IDB) aus der Menge der Zweiteren zusammensetzt.
Um die von Datalog geforderten finiten Ergebnismengen zu erhalten, gilt eine Bereichsbegrenzung für die Regeln. Kommt eine Variable im Kopf einer Regel vor, muss sie bereichsbeschränkt sein. Effektiv wird dies erreicht, indem die Bedingung der Beschränkung auferlegt wird, die aussagt, dass jede im Kopf einer Regel vorkommende Variable in mindestens einem Körperprädikat vorkommen muss. Diese Einschränkung führt auch dazu, dass bei der Bottom-up-Auswertung die Unifizierung durch Übereinstimmung ersetzt werden kann, die viel einfacher durchzuführen ist. Das ist möglich, da die Köpfe aller Regeln der Körperprädikate einer weiteren Regel bereits instanziiert sind, wenn die Unifikation durchgeführt wird. Somit werden nur Grundsubstitutionen und keine Variabelnumbenennungen benötigt.

Definition 2.22 (Bereichsbegrenzung): [16] R sei eine Datalogregel, V sei die Menge aller Variablen, die im Kopf der Regel und W die Menge aller Variablen die in den Unterzielen U des Körpers der Regel vorkommen. Dann gilt:

$$V \backslash W = \emptyset$$

•

Proposition 2.1: Sei P ein Datalogprogramm mit funktionslosem Datalog, für das die Bereichsbegrenzung gilt. Dann liefert eine Anfrage q über P immer eine endliche Lösungsmenge.

Beweis 2.1: Ein Datalogprogramm P ist per Definition eine endliche Menge an Regeln. Das Herbrand-Universum U_p setzt sich aus allen Grundtermen aus P zusammen. Diese Menge ist äquivalent zu allen Konstanten k, $k \in P$, also ist U_p eine endliche Menge. Die Menge G aller Prädikatensymbole g, $g \in P$ ist auch endlich, also ist auch die Herbrandt-Basis B_p eine endliche Menge, da sie sich aus G und P zusammensetzt. M_p, das Minimalmodell ist auch ein Herbrandt-Model M und alle diese sind Teilmengen von B_p, also auch endlich. Daraus folgt, dass die Lösungsmenge L einer Anfrage q über P endlich ist, da $L_q = M_p$ gilt. □

Beispiel 2.5: Läge eine Regel der Form

$$\mathtt{q(X,Y)} :- \mathtt{p(X)}.$$

vor, wäre die Bereichsbegrenzung verletzt. Eine Anfrage $? - \mathtt{q(X,Y)}.$ würde zu einer unendlichen Lösungsmenge $M_L = \{(x, y) | \forall x : \mathtt{p}(x) = \mathtt{true}\}$ führen, was der Forderung, dass eine Dataloganfrage immer finite Lösungsmengen liefert, nicht genügen würde. ◇

[15] Und damit auch variablenlosem Kopf, wie aus der Definition der Bereichsbegrenzung in Definition 2.22 folgt.
[16] Die praktischen Auswirkungen der Bereichsbegrenzung sind in Abschnitt 3.5 gegeben.

2.4.2 Semantik

Substitutionen sind gleich definiert wie in Definition 2.19 beschrieben. Mit ihrer Hilfe können, wie auch dort, Regeln aus anderen Regeln abgeleitet werden. Im Folgenden ist ein Beispiel gegeben, das der von Datalog verwendete Inferenzmechanismus *Elementary Production Principle* (EPP) auf einige Regeln anwendet.

Beispiel 2.6: Wir haben die Regeln (IDB)

$$\texttt{weg(X,Y)} :- \texttt{direct(X,Y)}.$$

$$\texttt{weg(X,Y)} :- \texttt{direct(X,Z)}, \texttt{weg(Z,Y)}.$$

und die Fakten (EDB)

$$\texttt{direkt(a,b)}.$$

$$\texttt{direkt(c,a)}.$$

womit durch den EPP abgeleitet werden kann dass

$$\texttt{weg(a,b)}.$$

$$\texttt{weg(a,a)}.$$

$$\texttt{weg(c,a)}.$$

$$\texttt{weg(c,b)}.$$

◇

EPP kann zur Erstellung von Ableitungen von unten nach oben (Bottom-up) verwendet werden.

2.4.3 Deduktionssystem

Um Rekursion zu lösen, nutzt Datalog deduktive Abgeschlossenheit, der Interferenzmechanismus EPP wird dabei so lange immer wieder auf die Regelmenge angewandt, bis diese sich nicht mehr ändert und somit der sogenannte *Least Fixed Point* erreicht ist. Die Mengen der erhaltenen Fakten entspricht dann dem Minimalmodell. Hierbei ist zu beachten, dass diese Terminierung nur garantiert ist, wenn die Regelmenge keine mehrfachen Vorkommen einer Regel enthält, also disjunkt mit sich selber ist.

Definition 2.23 (Deduktive Abgeschlossenheit): F sei eine Menge Regeln eines Datalogprogrammes P und $EEP(F)$ stelle die Anwendung des Interferenzmechanismus auf die Regelmenge F dar. Wenn gilt

$$\texttt{EEP(F) = F}$$

wird F als *deduktiv abgeschlossen* bezeichnet und der *Least Fixed Point* ist erreicht. •

3 Grundlagen

3.1 Möglichkeiten und Beschränkungen von AlegroGraph

AlegroGraph ist ein Tripel-Datenbanksystem mit einem Schwerpunkt zur Verwendung in Semantic-Web-Applikationen als RDF-Tripel-Datenbank. In diesem Abschnitt wird auf die Besonderheiten des DBS eingegangen und seine Funktionen im Hinblick auf eine Erweiterung durch eine deduktiven Komponente untersucht.
Franz Inc. Philosophie bei der Entwicklung von AlegroGraph ist eine solide Unterstützung der W3C-Standards bei gleichzeitiger Bereitstellung möglichst vieler Erweiterungen, die für einen Anwender sinnvoll sein könnten. Laut Herstellerangaben wird versucht, dem Programmierer möglichst wenig Vorgaben zu machen und ihm damit die Umsetzung seines eigenen Stils zu ermöglichen. Dementsprechend gibt es meistens mehrere Wege, ein Ziel zu erreichen.
Die vom W3C-Standard abweichenden Funktionalitäten können teilweise für eine effizientere und einfachere Implementation von AGDC genutzt werden. Dabei muss jedoch bedacht werden, dass nur standard-konforme Datengrundlagen vorausgesetzt werden können.

Dieses Kapitel ist mit fundiertem Hintergrundwissen zu AlegroGraph leicher zu verstehen und setzt Kentnisse über RDF nach der W3C-Spezifikation voraus. Beschreibungen von AlegroGraph können unter http://www.franz.com/agraph/allegrograph/[17] gefunden werden, die W3C-RDF-Spezifikation unter http://www.w3.org/RDF/[18].

3.1.1 Stores und Federated Stores

Die AlegroGraph Datenbanken werden Stores genannt und sind ungeordnete Tripel-Speicher. Mehrere Stores können gleichzeitig geöffnet sein und werden über einen Storepointer gehalten. AlegroGraph bietet sich auch als Wrapper für andere Triple-Store-Systeme an. Dabei wird keine Unterscheidung zwischen den tatsächlich unterliegenden Datenbanken gemacht, solange nur Methoden genutzt werden, die dem W3C-RDF-Standard entsprechen. Die AlegroGraph eigenen Erweiterungen stehen somit für fremde Datenbanken nicht zur Verfügung. Es kann auf lokale oder entfernte Stores zurückgegriffen werden und es gibt zusätzlich die Möglichkeit, mehrere offene Stores zu einem Federated Store zusammenzufassen. Dieser präsentiert sich nach außen als ein einzelner Tripel Store, es stehen jedoch die Tripels aller zusammengefassten Stores zur Verfügung. In Federated Stores können wiederum eigene und fremde sowie lokale und entfernte Stores eingebunden werden. Durch diese Optionen können die Fakten- und Regelmengen partitioniert und in verschiedenen Stores gehalten werden. Je nach Anwendungsfall können die benötigten Datenbanken dann zu Federated Stores zusammengefasst und der deduktiven Komponente übergeben werden.

3.1.2 RDF-Tripel und W3C-Standards

AlegroGraph-Stores sind auf die Speicherung von Triples beschränkt. Deshalb müssen Regeln, um sie in einem Store speichern zu können, durch eine Menge

[17] Stand 19.06.2009
[18] Stand 19.06.2009

von Tripels ausgedrückt werden. Das genaue Vorgehen wird in Abschnitt 3.4 dargelegt. Der W3C-Standard zu RDF-Tripels gibt für viele Tripel-Elemente einen Namespace vor. Ein Tripel besteht immer genau aus den drei Elementen Subjekt, Prädikat[19] und Objekt, die durch ihren Namespace und ihren Wert definiert sind. Das bedeutet, dass zwei Elemente nur dann als gleich angesehen werden, wenn sie denselben Wert und denselben Namespace besitzen. Dies muss bedacht werden, wenn es um die Deckung von Prädikaten und Triple-Prädikaten und die Gleichheit von Konstantenelementen geht und wird in Abschnitt 3.3 genauer dargelegt. AlegroGraph nutzt intern eigentlich keine Tripel, sondern Quintupel der Form (Subjekt, Prädikat, Objekt, Tripel-Id, Graph). Dies stellt eine Erweiterung dar, die für die Fakten der EDB nicht genutzt werden kann, da 1. dadurch eine Verwendung von fremden Stores unmöglich gemacht werden würde, da diese die AlegroGraph Erweiterungen nicht unterstützen und 2. die Herkunft der Tripel in der EDB nicht der Kontrolle der deduktiven Komponente unterliegt. Sie können also aus Quellen stammen, denen die erweiterten Möglichkeiten nicht zur Verfügung stehen. Für die Regeltripel ist eine Verwendung jedoch angemessen, da diese durch einen Parser von AGDC erstellt und in der IDB-Datenbank abgelegt werden. Es wird IDB-Datenbanken somit die Beschränkung auferlegt, nur AlegroGraph eigene Stores zu nutzen.

3.1.3 Graphen

AlegroGraph ordnet Tripel einem RDF-Graphen zu, wodurch mehrere RDF-Graphen in einem Store gespeichert werden können, ohne die Bindung der Tripel zum Graphen zu verlieren. Graph-Elemente können dabei AlegroGraph-Tripel-Parts sein. Dadurch bietet sich die Möglichkeit mehrere Regeln in einem Store abzulegen, ohne die Zuordnung der Tripel zu den Regeln zu verlieren. Jede Regel wird somit einem eigenen RDF-Graphen zugeordnet.

3.1.4 First-Order-Tripel

Tripel besitzen in AlegroGraph, wie in Abschnitt 3.1.2 erwähnt, eine ID im Store. AlegroGraph bietet die Möglichkeit an, über diese Statements bezüglich der Triple zu erstellen, indem die Triple-ID als Subjekt oder Objekt eines anderen Triples verwendet wird. Abbildung 2 stellt dies grafisch dar.

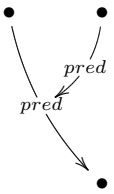

Abbildung 2: First-Order-Triple: Statements über Tripel in grafischer Darstellung

Dies ist eine Abweichung vom W3C-RDF-Standard und kann deshalb nur für die IDB genutzt werden. Hier kann sie dazu dienen, die benötigte Tripelanzahl

[19] Um zwischen Regel- und Tripel-Prädikaten zu unterscheiden, werden erstere im Folgenden als Prädikate und zweitere als Tripel-Prädikate bezeichnet.

zur Darstellung einer Regel zu verkleinern.

3.1.5 Typbezeichner

AlegroGraph kennt und unterscheidet zwischen einigen Typen von Elementen, die als Subjekt, Prädikat oder Objekt der Tripel verwendet werden können. Dies schließt vom W3C definierte Typen und noch einige eigene und spezielle Typen mit ein. Dadurch wird eine effizientere Speicherung der Tripel ermöglicht, da beispielsweise eine Zahl als Integer abgelegt weniger Speicher verbraucht als im String-Format. Des Weiteren kann die Suche nach Tripeln beschleunigt werden, da ein Vergleich von Strings wiederum länger dauert als einer von Integers. Bei der Erstellung eines Elements (im Folgenden auch als UPI[20] bezeichnet) kann der gewünschte Typ mit Wert und Namespace übergeben werden, beim Parsen bestimmter Dateiformate kann AlegroGraph den Typ auch anhand der XML-Notation erkennen[21]. Dies bietet einen guten Ansatz für die Verwendung eines Typsystems in AGDC, der in Kapitel 5.3 weiter ausgeführt wird.

3.1.6 Indexierung von Tripeln

AlegroGraph speichert Tripel normalerweise ungeordnet ab, was die Suche nach bestimmten Tripeln langwierig macht. Es wird deshalb die Möglichkeit geboten, einen Index für die Tripel eines Stores anzulegen, der die Suche beschleunigt. Ein Store kann teilweise oder ganz indexiert werden. Gerade für die IDB, die häufig gezielt nach bestimmten Tripeln durchsucht wird, bietet sich eine Indexierung der Tripel an, sie ergibt aber auch für die EDB einen Sinn.

3.1.7 Anfrageschnittstellen: SPARQL, Prolog und native Lisp-Methoden

Gerade im Hinblick auf das Auslesen von Tripeln aus einem Store bietet AlegroGraph mehrere Möglichkeiten für den Benutzer, die hier kurz mit ihren Vor- und Nachteilen vorgestellt werden.

SPARQL SPARQL ist die W3C-Empfehlung zu RDF-Anfragesprachen. Ihre Syntax ähnelt der von SQL. Mit ihr sind bedingte Anfragen möglich. Eine Anfrage kann also zu mehreren, zusammenhängenden und einander bedingenden Tripeln als Ergebnis führen. Ein Nachteil ist, dass SPAQRL als W3C-Standard nur mit RDF-Standard-konformen Eigenschaften von Tripeln arbeitet, die AlegroGraph eigenen Erweiterungen also nicht genutzt werden könnten. Ein potentieller Vorteil wäre allerdings die Möglichkeit, in Zukunft eventuell ein anderes unterliegendes DBS als AlegroGraph zu nutzen. Als ein weiterer Nachteil könnte sich erweisen, dass die AlegroGraph SPARQL-Schnittstelle einen Wraper für die nativen Lisp Methoden darstellt, die Anfrage also in eine Reihe von diesen umgeformt wird. Dadurch könnte ein Geschwindigkeitsverlust entstehen, da teilweise redundante Informationen abgefragt werden.

Prolog Franz Inc. bietet einen in Alegro Common Lisp integrierten, optimierten Prolog Interpreter Alegro Prolog an. AlegroGraph erweitert diesen

[20] `Unique Part Identifier`
[21] Beispielsweise 1^^xsd:integer

um die Möglichkeit, Tripel als Prolog-Fakten zu interpretieren und somit bedingte, deklarative Anfragen an ein Store über das Prologinterface zu stellen. Dabei können auch die AlegroGraph Erweiterungen des RDF-Standards genutzt werden. Wie bei SPARQL werden die Anfragen in native Lisp Methoden übersetzt, es besteht also die Gefahr eines Geschwindigkeitsverlustes. Allerdings eröffnet sich die Möglichkeit, ein Teil der Schlussfolgerungen des Deduktionssystems an den Prolog Interpreter auszulagern, was wegen der großen Ähnlichkeit von Datalog und Prolog leicht erreichbar wäre.

Native Lisp Methoden Mit den nativen Lisp Methoden können Tripel nur direkt anhand ihrer Eigenschaften ausgelesen werden, ohne sich gegenseitig zu bedingen. Dadurch würde ein programmatischer Ansatz der Anfragen erzwungen, was ein Mehr an Programmierarbeit erfordert und den Quellcode unübersichtlicher macht. Als positiver Punkt lässt sich nur die erwartete höhere Geschwindigkeit angeben, die nach den in Abschnitt 1.1 dargelegten Präferenzen aber schwer wiegt.

Alle beschriebenen Ansätze haben ihre Vor- und Nachteile. Es soll davon abgesehen werden die Schnittstellen zu mischen, sondern sich für eine einzige entschieden werden. In Anhang C werden die Schnittstellen auf ihre Geschwindigkeit hin getestet und durch dieses Kriterium eine Entscheidung zu Gunsten der nativen Lisp-Methoden getroffen. Die Geschwindigkeit der nativen Lisp-Methoden ist in manchen Fällen um eine Zehnerpotenz höher al die der anderen Schnittstellen und rechtfertigt damit den größeren Arbeitsaufwand durch den programmatischen Ansatz.

3.2 IDB und EDB als Tripel-Stores

Durch die Verwendung von AlegroGraph-Stores als EDB- und IDB-Datenbanken ergeben sich einige Besonderheiten, die berücksichtigt werden müssen.
Die Erweiterungen, die AlegroGraph anbietet und die vom RDF-Standard abweichen, können für die EDB aus schon genannten Gründen nicht berücksichtigt werden. Wird allerdings eine EDB vorgefunden, die diese Erweiterungen beinhaltet, wäre es wünschenswert, wenn AGDC dies erkennt und unterstützt.
In den Stores können nur Triple gespeichert werden, was dazu führt, dass die Faktengrundlage nur aus zweistelligen Prädikaten besteht. First-Order-Triple würden eine Möglichkeit bieten, auch mehrstellige Prädikate als Fakten in der EDB abzulegen; dafür müsste allerdings eine Syntax festgelegt werden, was wegen der fehlenden Kontrolle über die EDB nicht möglich ist. Es ist aber möglich, Fakten als IDB-Regeln darzustellen. Da Fakten Regeln ohne Körper sind, wird durch das Erstellen einer Regel ohne Körper ein Fakt definiert, der in der Auswertung berücksichtigt wird. Somit kann AGDC auch außerhalb des Kontextes von RDF Tripeln als deduktives DBMS genutzt werden.
Es existiert somit keine strikte Trennung zwischen Regeln und Fakten, was dem Sinne von Datalog entspricht, wonach Fakten nichts anderes als körperlose Regeln sind. AGDC macht somit auch keine Unterscheidung zwischen IDB und EDB in dem Sinne, dass ein Fakt der EDB mit p/2 nicht anders behandelt wird als eine körperlose Regel der IDB mit einem Kopfprädikat p/2. Eine Speicherung von Fakten über diesen Weg erfordert allerdings viel mehr Speicherplatz und ist nicht zu empfehlen. Soll beispielsweise ein dreistelliger Fakt.

$\texttt{p}(\texttt{X}, \texttt{Y}, \texttt{Z})$.

erzeugt werden, so ist es unter dem Blickwinkel der benötigten Speicherkapazitäten sinnvoller eine Regel

$\texttt{p}(\texttt{X}, \texttt{Y}) :- \texttt{edbl}(\texttt{X}, \texttt{Y}), \texttt{edbr}(\texttt{Y}, \texttt{X})$

mit den entsprechenden EDB-Fakten zu definieren, als mehrere körperlose, dreistellige IDB-Fakten.
AlegroGraph bietet auch die Möglichkeit Elemente ohne Namespaces abzulegen, was von AGDC erkannt wird und wodurch diese Erweiterung auch in der EDB verwendet werden kann. Praktisch wird ein leerer Namespace als eigener Namespace betrachtet, was bedeutet, dass Gleichheit zwischen zwei Elementen mit leerem Namespace besteht, wenn sich ihre Werte decken. Ein leeres Namespaceattribut eines UPI wird also nicht als Wildcard interpretiert.

3.3 Namespaces für Konstanten und Prädikate

Das schon angeschnittene Thema der Namespaces ist besonders relevant für Konstanten und Prädikate. AGDC muss die Möglichkeit haben, zwei Prädikate anhand ihrer Stelligkeit und ihres Prädikatensymbols als deckend zu erkennen. Gleiches gilt für Konstanten. Prädikatsymbole und Konstanten werden durch UPIs repräsentiert und beinhalten neben einem Wert und Typbezeichner auch einen Namespace. Bei der Regelformulierung muss deshalb darauf geachtet werden, nicht nur den Wert der Konstanten und Prädikatensymbole anzugeben, sondern auch den Namespace, da beide in Einheit das eigentliche Element darstellen. Namespaces können beliebig gewählt werden, solange sie dem W3C-Standard folgen oder einer Erweiterung durch AlegroGraph entsprechen. Eine Einführung von Namespace-Wildcards wurde angedacht, aber mit der Begründung verworfen, dass Werte ohne Namespace im Kontext des Semantic Web nur dann einen Sinn ergeben, wenn es sich um atomare String- oder Zahlenwerte handelt. In diesen Fällen ist ihnen aber kein Namespace zugeordnet, es bedarf also ohnehin keines Wildcard-Attributes. Für andere Elemente wie Ressourcen kann nur über die Kombination aus Namespace und Wert die Bedeutung eines Prädikates extrahiert werden, die Ausgabe eines Wertes ohne Namespace ergebe somit keinen Sinn.
Eine Regel wird dementsprechend üblicherweise nicht in der Form

```
(:- (...)
    ( (p ?x ?y) ...) )
```

definiert, sondern mit zum Prädikatensymbol gehörigen Namespace, also

```
(:- (...)
    ( (http\://example.com/definitions#p ?x ?y) ...) )
```

Dies trifft natürlich nur für Prädikate zu, die sich mit gleichnamigen EDB-Fakten decken sollen. Wenn beispielsweise der Kopf einer Regel sich nie mit einem EDB-Fakt zu decken braucht, so ist es auch nicht notwendig, einen Namespace mit anzugeben. In einem solchen Fall hat die Ressource, der das Prädikat entspricht, keinen Namespace, was durch die Erweiterungen von AlegroGraph über den RDF-Standard hinaus ermöglicht wird. Sollte die EDB einen ebenfalls namespacelosen, gleichwertigen und gleichstelligen Fakt beinhalten, würde

dieser sich mit dem Prädikat decken. Intern verwendet AGDC die AlegroGraph-Funktion (resource <predicate>) um das Prädikat in ein für die Datenbank akzeptables Format zu konvertieren, wodurch auch ein eventuell mit angegebener Namespace präserviert wird.

3.4 Regeln als RDF-Graphen und interne Regelrepräsentation

Regeln müssen als Menge von Tripeln dargestellt werden, um sie im IDB-Store speichern zu können. Wichtig ist dabei, dass eine Repräsentation gewählt wird, bei der keine Information verloren geht. Dafür wird die Regel in einen RDF Graphen umgewandelt, dessen Subjekte und Objekte den Elementen und Informationen der Regel entsprechen, während die AGDC eigenen Triple-Prädikate nur als fest definierte Elemente mit einem eigenen Namespace auftreten. Dieser wird global als System-Namespace festgelegt und kann eine beliebige Ausprägung erfahren. In der Regelrepräsentation als RDF-Graph tauchen die Regelelemente nur als Subjekte und Objekte auf, während die eigenen Triple-Elemente immer als Prädikate verwendet werden. Damit ist eine Unterscheidung immer möglich und auch eine Verwendung desselben Namespaces für Regeln oder Fakten führt nicht zu Überschneidungen. Die Transformation einer Regel lässt sich am besten anhand eines Beispiels aufzeigen, siehe dazu Anhang B.3.
Zur Verarbeitung werden die Regeln und Fakten als Objekte im System gehalten. Die dafür gewählte Struktur setzt sich aus Klassen und Listen zusammen. Eine Regel wird durch eine Instanz einer *rule*-Klasse repräsentiert, Literale als Instanzen von *literal*-Klassen. Die Liste 4 in Anhang B.2 stellt die formale Definition dar.
Sowohl EDB- als auch abgeleitete Fakten werden als Instanzen der *literal*-Klasse gehalten.

3.4.1 Die Regeln im Speicher

Der gewählte Ansatz für AGDC läd die Regeln und Fakten in den Hauptspeicher bevor die verarbeitet werden. Dieses Vorgehen kann bei großen Regel- bzw. Faktenmengen zu Problemen führen, wenn nicht genug Speicher zur Verfügung steht und somit Page-Faults auftreten. Das Problem wird noch dadurch verstärkt, dass die abgeleiteten Regeln auch im Speicher gehalten werden müssen, wobei auch wenige IDB-Regeln zu einer sehr großen Menge abgeleiteter Regeln führen können. Eine Alternative wäre es, immer nur die Regeln im Speicher zu halten, mit denen gerade gearbeitet wird. Dies würde jedoch ständige Lese- und Schreibvorgänge über den Stores und die Abspeicherung der nur temporär benötigten abgeleiteten Regeln in einem Store erfordern. Des Weiteren müssten viele spezielle Suchoperationen über den Stores ausgeführt werden. Es ist zu erwarten, dass dies zu einem signifikanten und nicht zu akzeptierenden Geschwindigkeitsverlust führen würde.
Unter der Annahme, dass meistens mit großen Fakten- und verhältnismäßig kleinen Regelmengen gearbeitet wird, werden die Fakten der EDB immer nur konkret für die unifizierenden Subgoals der Regeln ausgelesen. Nicht relevante Fakten werden somit nicht berücksichtigt und führen nicht zu einer Vergrößerung der zu haltenden Daten. Fakten die auf ein Unterziel einer Regel anwendbar

waren müssen jedoch im Speicher gehalten werden, das sie noch für weitere Schlussfolgerungen benötigt werden[22].
Gleiches gilt für die IDB; hier kann von der Anfrage ausgehend ein Abhängigkeitsbaum erstellt und diesem folgend nur relevante Regeln aus der IDB gelesen werden.
Diese Ansätze werden im Abschnitt 4.1.2 für die Fakten und Regeln noch einmal aufgegriffen und konkretisiert.

3.4.2 AlegroCache zur Speicherauslagerung

Franz Inc. bietet mit AlegroCache ein Framework für persistente Objekte mit transparenter Handhabung an. Dem Framework zugeordnete Objekte können behandelt werden als ob sie im Speicher vorhanden wären, während AlegroCache sich um die Speicherung im Falle eines Speichermangels und das Nachladen bei Bedarf kümmert.
Damit könnte eine Möglichkeit gegeben sein, auch sehr große Regel- und Faktenmengen zu verarbeiten, indem die *rule*- und *literal*-Klassen als cache-verwaltete Klassen definiert werden.
Die Umsetzung dieses Ansatzes würde allerdings den Rahmen der Arbeit überschreiten.

3.5 Regel- und Anfragesyntax

Eine neue Regel wird der IDB mit dem Makro :− hinzugefügt.

```
(:− (head−predicate [variable|constant]*)
        ( [([not] predicate [variable|constant]*)]* ))
```

Variablen werden durch ein vorangestelltes *?* bezeichnet, Konstanten können in Anführungszeichen *"* eingeschlossen werden oder als Symbol repräsentiert werden.

```
?var     − Variable
const    − Konstante
"const"  − Konstante
```

Eine Datalogregel, in der Literatur häufig verwendeten Form

$$\mathtt{p(X,Y)} :- \mathtt{q(X,Z)}, \mathtt{r(Z,Y)}, \neg\mathtt{q(X,Y)}.$$

dargestellt, würde für AGDC in der Form

```
(:− (p ?x ?y)
        ( (q ?x ?z) (r ?z ?y) (not q ?x ?y) ))
```

definiert werden.
Neben der Übergabe als Symbole können Prädikatensymbole und Konstanten auch als AlegroGraph Future-Parts an das Regelmakro übergeben werden. Dies ermöglicht eine einfachere Form der Namespacedefinition.
Die Regeln werden vom Makro geparst und in die IDB eingetragen. Dabei ist die in Definition 2.22 im Abschnitt 2.3 vorgestellte Bereichsbeschränkung für Variablen zu berücksichtigen.

[22] Beispielsweise für die Transformationsmethode Failure, siehe Abschnitt 5.1.1

i Variablen, die im Kopf einer Regel vorkommen, müssen auch in mindestens einem nicht negierten, relationen Unterziel derselben Regel vorkommen.

ii Variablen, die in einem negierten Unterziel einer Regel vorkommen, müssen auch in mindestens einem nicht negierten, relationen Unterziel derselben Regel vorkommen.

iii Variablen, die in einem arithmetischen Unterziel einer Regel vorkommen, müssen auch in mindestens einem nicht negierten, relationalen Unterziel derselben Regel vorkommen.

Ein Anfrage an das System wird über das Makro ?− gestellt und stellt ein positives Literal dar.

```
(?- predicate [variable|constant]*)
```

Im Hinblick auf die Darstellung der Variablen und Konstanten gelten die selben Regeln wie für die Regelerstellung mit dem Makro :−. Es ist nicht möglich, die Anfrage als eine eigene Regel zu definieren. Als Ergebnis werden alle EDB und abgeleiteten Fakten als Literale zurückgegeben, die mit dem Anfrageliteral unifizieren.
Für eingebaute Prädikate und Typbezeichner wird die Regel- und Anfragesyntax erweitert, genaueres ist in den entsprechenden Abschnitten 5.2 und 5.3 zu finden.

4 Auswertungsstrategie

Dieser Abschnitt stellt die Auswertungsstrategie vor, die in AGDC Verwendung findet. Es werden Grundkenntnisse von semi-naiver Auswertung und magischen Prädikaten vorausgesetzt.

4.1 Semi-naives Bottom-up-Verfahren

Die semi-naive Auswertung ist ein Bottom-Up-Verfahren, das mengenorientiert ist und im Vergleich zur naiven Auswertung einige Redundanzen während der Ableitung von Regeln entfernt. Die Unterziele der Regeln des Programms werden mit den Fakten durch Substitution in allen möglichen Ausprägungen verschränkt und somit werden neue, abgeleitete Fakten berechnet. Dies geschieht rundenweise, wobei in der ersten Runde die EDB-Fakten in allen möglichen Varianten mit den Regeln der IDB inferieren. In der nächsten Runde werden nur die neu abgeleiteten Fakten berücksichtigt. Dieser Vorgang wird solange wiederholt, bis keine neuen Fakten mehr abgeleitet werden können. Dieser Haltepunkt wird als *Least Fixed Point* bezeichnet.
Formal beschreiben lässt sich dieses Vorgehen durch den *Standard Immediate Consequence Operator* $T_P(I)$, wie er in der Veröffentlichung von Brass et al. 2001 [BDFZ01] gegeben ist.

Definition 4.1 (Standard Immediate Consequence Operator (nach Brass et al. 2001 [BDFZ01])): Sei P ein bereichsbeschränktes Datalogprogramm, $ground(P)$ die Herbrandt Instanziierung von P, $\check{L}$ die Menge aller Körperliterale einer Regel, A der Kopf einer Regel, $pos(\check{L})$ alle positiven, relationalen Literale in $\check{L}$ und I eine Menge von Fakten. Dann gilt

$$T_P(I) := \{A \mid \text{es gibt ein } A \leftarrow \check{L} \in ground(P) \text{ mit } pos(\check{L}) \subseteq I\}.$$

•

Zuerst wird $T_P(I_0)$ auf das Programm mit $I_0 = EDB - Fakten$ angewendet und man erhält $I_1 = T_P(I_0)$. Anschließend folgt $I_{n+1} = T_P(I_n)$ mit $n \geq 1$ bis der *Least Fixed Point* erreicht wurde, also $T_P(I_n)$ keine Fakten mehr liefert, die nicht schon bekannt sind. Dies wird mit $lfp(T_P)$ bezeichnet.
Es ist zu beachten, dass T_P negative Literale und eingebaute Prädikate als Unterziele einer Regel ignoriert, also Regeln in denen diese vorkommen nicht vollständig instanziiert werden können. Die Zuweisung eines Wahrheitswertes an diese wird verzögert; die Auswertung negativer Literale wird erschöpfend in Abschnitt 5.1 behandelt. Eingebaute Prädikate erhalten ihren Wahrheitswert durch eine Nebenroutine, die im Schlussfolgerungsprozess integriert ist und ausgelöst wird, sobald alle Variablen eines eingebauten Prädikates instanziiert wurden.
Um die semi-naive Auswertung verwenden zu können und trotzdem alle möglichen Ausprägungen der Regeln zu erhalten, arbeitet AGDC mit *verzögerten, positiven Literalen*. Nach der gegebenen Definition von T_P kann der im folgenden Beispiel 4.1 aufgezeigte Fall auftreten.

Beispiel 4.1: Wird T_P auf ein Program P mit den IDB-Regeln

```
p(X) :- q(X), edb(X).
q(X) :- edb(X).
```

angewendet und werden nach der semi-naiven Methode in der ersten Runde zum Erhalt von $lfp(T_P)$ die EDB-Fakten `edb(1).` verwendet, so ergibt sich

```
p(X) :- q(X), edb(X).
q(X) :- edb(X).
q(1).
```

Damit stehen die abgeleiteten Fakten `p(1).` für die nächste Runde zur Verfügung. Da allerdings `edb(1).` aus der ersten Runde nicht mehr berücksichtigt wird, kann nie die Schlussfolgerung

```
p(1) :- q(1), edb(1).
```

gezogen werden und das Ergebnis ist unvollständig. ◇

Dieses Problem kann gelöst werden indem erlaubt wird teilinstanziierte Regeln zu erzeugen, was durch Ignorieren der negativen Literale ohnehin schon geschieht. Der *Immediate Consequence Operator mit verzögerten Literalen* $\tilde{T}_P(I)$ wird wie folgt definiert:

Definition 4.2 (Immediate Consequence Operator mit verzögerten Literalen): Es gelten dieselben Rahmenbedingungen wie in Definition 4.1, $\tilde{T}_P(I)$ führt dieselben Operationen durch wie auch $T_P(I)$, transformiert aber gleichzeitig das Programm P nach $\tilde{P}$ durch Bildung von

$$\hat{P} := \{A \leftarrow \{\check{L} - L\} | \text{ es gibt ein } A \leftarrow \check{L} \in \mathit{ground}(P) \text{ mit } L \in \mathit{pos}(\check{L}) \wedge L \in I\}$$

und schließlich $\tilde{P} := \hat{P} \cup P$ und

$$\tilde{T}_P(I) := \{A | \text{ es gibt ein } A \leftarrow \check{L} \in \hat{P} \text{ mit } \check{L} = \emptyset\}$$

•

Es gilt $\tilde{T}_P(I) = T_P(I)$; nur ändert sich bei $\tilde{T}_P(I)$ in jeder Runde das Proramm P mit, indem es um teilinstanziierte Regeln erweitert wird. Damit ist die seminaive Auswertungsmethode anwendbar.

4.1.1 Partitionierung nach Strongly Connected Components

In Zukowski et al. 1997 [ZFB97] wird eine Auswertung vorgestellt, die auch in Brass et al. 2001 [BDFZ01] als *Intelligent Grounding* Verwendung findet und sich an *Strongly Connected Components* (SCC) orientiert.
Partitionierung nach SCC sieht vor, ein Datalogprogramm P in Teilprogramme mit fester Reihenfolge $P^{(1)}, P^{(2)}, ...P^{(n)}$ zu zerlegen, so dass alle relevanten Instanziierungen einer Regel aus einer Teilmenge $P^{(i)}$ nur von den vorherigen Instanziierungen der Teilmengen $P1(j), j < i$ abhängen. Damit kann der Instanziierungsprozess schrittweise erfolgen und es müssen immer nur die abgeleiteten Fakten der vorherigen, instanziierten Mengen an den Instanziierungsprozess der nächsten Teilmenge weitergegeben werden.
Zur Partitionierung eines Programms P wird ein Abhängigkeitsgraph G der Menge der Regeln M_R aus P von der Anfrage Q als Wurzelknoten ausgehend erstellt und anschließend ein an Pearce 2005 [Pea05] angelehnter Algorithmus zur Partitionierung nach SCC angewendet. Die Untermengen $P^{(i)}$ des Programms $P = P^{(1)} \cup ... \cup P^{(n)}$ werden abschließend in Reihenfolge instanziiert, wobei jeweils die abgeleiteten Fakten der vorangehenden für die aktuell bearbeitete Menge mit berücksichtigt werden.
Durch die Anwendung dieses Ansatzes muss AGDC viel weniger Regelinstanziierungen und Fakten gleichzeitig im Speicher halten und kann mit größeren Datenmengen arbeiten. Zwar wird insgesamt immer noch mit einer vergleichbar großen Datenmenge gearbeitet, aber da diese partitioniert ist, werden einige Iterationsschleifen über den Daten schneller terminieren. Es ist also ein Geschwindigkeitsgewinn für Programme zu erwarten, deren Regeln sich nicht in einer einzigen SCC befinden.

4.1.2 Optimierungen

Begrenzung auf relevante Regeln Während der Auswertung mit der seminaiven Methode werden eine große Menge teilinstanziierter Regeln gebildet und müssen im Speicher gehalten werden. Diese Anzahl kann leicht das vielfache der eigentlichen IDB-Regeln und EDB-Fakten betragen. Eine einfache Optimierungsmöglichkeit besteht darin, nur die IDB-Regeln zu beachten die eine

Relevanz für die gestellte Anfrage haben. Dafür wird von der Anfrage ausgehend ein Abhängigkeitsgraph erstellt und alle Regeln, die in diesem auftauchen dem Inferenz-Mechanismus übergeben.
Dieser Optimierungsansatz ist nur für IDB-Regelmengen sinnvoll, die nicht vollständig aufeinander aufbauen. Wegen der großen Anzahl abgeleiteter Regeln pro anfänglicher IDB-Regel macht sich ein Geschwindigkeitsgewinn schon bei wenigen ausgelassenen Regeln bemerkbar, wie der in Tabelle 2 dargestellte Vergleich zeigt.

	# anfängliche Regeln/Fakten	# abgeleitete Regeln/Fakten	Zeit für 100 Anfragen
Ohne Opt.	23/30	580/178	26879 ms
Mit Opt.	9/30	402/93	17881 ms

Tabelle 2: Semi-naive Auswertung: Optimierung über Beschränkung der anfänglichen Regelmenge

EDB-Fakten des ersten Durchlaufes beschränken Statt alle Fakten aus dem EDB-Store zu laden und auf die Regeln anzuwenden, ist es von Vorteil in der ersten Runde der semi-naiven Auswertung, an der nur und zum einzigen Mal EDB-Fakten beteiligt sind, nicht die sonst in AGDC verwendete Schleifenreihenfolge (**loop** for facts ... (**loop** for rules ...)) zu verwenden, sondern die Regeliteration nach außen zu heben, also (**loop** for rules ... (**loop** for facts ...)) . Dies eröffnet die Möglichkeit, die Fakten jeweils einzeln und gezielt aus der Datenbank auszulesen und dadurch die insgesamt zu ladende Menge zu beschränken. Hierdurch müssen auch weniger Fakten im Speicher gehalten werden. Dieses Vorgehen ist gerade bei den zu erwartenden Anwendungsfällen mit sehr vielen Fakten von Vorteil. Die Tabelle 3 zeigt die vergleichenden Ergebnisse eines einfachen Testes auf.

	EDB nicht indexiert		EDB indexiert	
Query Optimierung	an	aus	an	aus
5^1 / 5^2	19	10	-	-
5^1 / 10^2	21	10	-	-
5^1 / 100^2	21	11	-	-
5^1 / 1.000^2	25	17	1	17
5^1 / 10.000^2	63	90	1	90
5^1 / 100.000^2	684	HO^3	1	HO^3

Tabelle 3: Auswirkungen der EDB Query Optimierung

Die Ergebniswerte sind Zeiten in Millisekunden, ^{1} bezeichnet die Anzahl der für die Anfrage relevanten Fakten, ^{2} die Gesamtzahl der EDB-Fakten im Store. HO^3 weist darauf hin, dass es zu einem Heap-Overflow in der freien Version von AlegroCL kam, deren Heap begrenzt ist. Wie erwartet zeigt sich der Vorteil der beschränkten Fakten erst bei EDB-Stores mit vielen Tripeln von denen wenige von der Anfrage betroffen sind. Ein bedeutender Geschwindigkeitsvorteil wird

erreicht, wenn der EDB-Store indexiert ist, da hiervon die Geschwindigkeit eines Ladens aller Fakten auf einmal nicht beeinflusst wird, ein gezieltes Suchen nach dem Prädikatensymbol jedoch sehr viel schneller durchgeführt wird. Eine Indexierung ist somit nicht nur für den IDB-Store, sondern auch für den EDB-Store von Vorteil. Dies liegt in der Verantwortung des Benutzers und wird dringend empfohlen. Für die Dauer der Instanziierung mit EDB-Fakten werden die schon aus der EDB ausgelesenen Fakten in einem Cache gehalten, um ein erneutes Auslösen der zeitkritischen Ausleseoperation zu vermeiden.

Bei jeder Anfrage, unabhängig von ihrer Ausprägung, werden alle Regeln und Fakten berücksichtigt, also immer alle möglichen IDB-Fakten abgeleitet und erst dann die Menge der abgeleiteten IDB- und expliziten EDB-Fakten durch die Anfrage gefiltert. Dabei treten noch immer viele unnötige Berechnungen auf, da beispielsweise alle Eltern-Kinder-Beziehungen gebildet werden, obwohl nur eine Anfrage nach den Kindern einer bestimmten Person vorliegt. Eine Herangehensweise an dieses Problem stellen die Magic Sets dar, eine Methode des logischen Umschreibens von Regeln, die im folgenden Abschnitt 4.2 vorgestellt wird.

4.2 Programmtransformation: Magische Mengen

Magic Sets[23] (Erstmals vorgeschlagen von Bancilhon et al. 1986 [BMSU86]) ist eine Optimierungsmethode für die Bottom-up-Auswertung, bei der die Regeln des ursprünglichen Programms P umgeschrieben werden, um eine effizientere Verarbeitung zu ermöglichen (Siehe Ceri et al. 1989 [CGT89] für eine Einführung). Die in der transformierten Version P_M eines Programms P enthaltenen magischen Prädikate bringen die aus der Top-down-Auswertung bekannten Sideway Information Passing Strategies (im Folgenden *sips* abgekürzt) in die Bottom-up-Auswertung ein. Dies geschieht unter Berücksichtigung der gebundenen Variablen der Anfrage mit dem Ergebnis, dass für diese irrelevante Instanziierungen von Regeln weitgehend vermieden werden. Eine formale Definition von sips kann in der Veröffentlichung von Beeri et al. 1991 [BR91] nachgelesen werden. Ebenfalls dort findet sich ein Beweis zur Äquivalenz von P und P_M in Hinblick auf eine Anfrage Q bei Verwendung der semi-naiven Bottom-up-Auswertung.

4.2.1 Generalized Supplementary Magic Sets

Die allgemeine Magic Set Methode ist mit dem Nachteil behaftet, dass noch immer viele Fakten mehrfach ausgewertet werden. In der Veröffentlichung von Beeri et al. 1991 [BR91] wird mit dem Generalized Supplementary Magic Set (GSMS) Verfahren eine Methode vorgestellt diese Mehrfachauswertungen bei Verwendung beliebiger sips teilweise zu elimieren. GSMS ist der in AGDC implementierte Ansatz zur Verwendung von Magic Predicates als Optimierung der semi-naiven Auswertung, da es auf alle nicht disjunktiven Datalog-Programme anwendbar ist und eine freie Wahl der sips ermöglicht. Der Beweis der Äquivalenz vom P und dem mit GSMS umgeschriebenen Programm P_{SM} in Hinblick auf Q und der garantierten Terminierung bei Bottom-up-Auswertung von P_{SM} für Datalog Programme ist in der gleichen Veröffentlichung zu finden.

[23] dt.: Magische Mengen

Es soll hier noch darauf hingewiesen werden, dass eine Optimierung eines Datalogprogramms mit GSMS eigentlich nur dann in einem wirklichen Geschwindigkeitsvorteil resultiert, wenn die Anfrage gebundene Variablen enthält, in vielen Fällen aber auch für ungebundene Anfragen eine Geschwindigkeitsteigerung erzielt werden kann.

4.2.2 Wahl der sips

Es gibt verschiedene sips, wie beispielweise *left-to-right-* und *well-founded-sips*. Die Wahl der für die Implementierung am besten geeigneten muss mit Hinblick auf die Verarbeitungsstrategie der logischen Negation geschehen. Für den in AGDC implementierten Ansatz zur Auswertung von Datalog mit Negation, der im nächsten Abschnitt 5.1 beschrieben wird, findet eine sips Verwendung, die *left-to-right* Informationsweitergabe bis zum ersten negierten, nicht arithmetischen Literal des Körpers der Regel anwendet und im Folgenden als *non-negated-left-to-right* sips bezeichnet wird.
Dafür werden die Körperliterale einer Regel vor Anwendung der GSMS-Transformation wie folgt angeordnet:

positv, relational $<$ *positiv, arithmetisch* $<$ *negiert, arithmetisch* $<$ *negiert, relational*

Supplementary Magic Prädikate (*suplement_ **) werden für alle Literale bis zum ersten negierten, relationalen Literal gebildet, Magic Prädikate (*magic_ **) mit auslösender Funktion jedoch auch für diese. Genaueres zur Bildung der Prädikate kann der Veröffentlichung von Beeri et al. 1991 [BR91] entnommen werden, die Auswirkungen der gewählten sips zeigen sich im Beispiel 4.2. AGDC erstellt intern einen Regel/Anfrage-Baum aus dem Programm, um die Regeln nach Bindungsmuster[24] der Variablen aufzustellen und daraufhin die GSMS-Transformation für jede Regel/Bindungsmuster-Ausprägung gezielt anzuwenden.

Beispiel 4.2: Die Unterziele einer Regel

$$\texttt{p(X,Y)} :- \texttt{X} > \texttt{Y}, \texttt{not r(X)}, \texttt{edb(X,Y)}, \texttt{not s(Y)}.$$

werden im ersten Schritt neu angeordnet und ergeben

$$\texttt{p(X,Y)} :- \texttt{edb(X,Y)}, \texttt{X} > \texttt{Y}, \texttt{not r(X)}, \texttt{not s(Y)}.$$

woraufhin die GSMS-Transformation angewendet und

$$\begin{aligned}
&\texttt{supmagic_p_0}_0\texttt{(X,Y)} :- \texttt{magic_p()}, \texttt{edb(X,Y)}.\\
&\texttt{supmagic_p_0}_1\texttt{(X,Y)} :- \texttt{supmagic_p_0}_0\texttt{(X,Y)}, \texttt{X} > \texttt{Y}.\\
&\texttt{p(X,Y)} :- \texttt{supmagic_p_0}_1\texttt{(X,Y)}, \texttt{not r(X)}, \texttt{not s(Y)}.\\
&\texttt{magic_r(X)} :- \texttt{supmagic_p_0}_1\texttt{(X,Y)}.\\
&\texttt{magic_s(X)} :- \texttt{supmagic_p_0}_1\texttt{(Y,X)}.
\end{aligned}$$

erzeugt wird. In diesem Beispiel sind die Bindungsmuster der Übersichtlichkeit halber unberücksichtigt gelassen. ◇

[24] engl.: adornment

Diese Wahl verringert den durch die Anwendung der GSMS Optimierungsmethode erhaltenen Vorteil nicht, da die nach rechts geordneten negierten relationalen Literale durch die semi-naive Methode nicht ausgewertet werden und dementsprechend auch nicht durch bedingt auslösende Supplementary Magic Prädikate optimiert werden können. Für den Erhalt korrekter Ergebnisse mit der gewählten Methode zur Behandlung von Negation, die im nächsten Abschnitt 5.1 vorgestellt wird, ist die Verwendung der *non-negated-left-to-right* sips jedoch Voraussetzung.

5 Erweiterung der Datalog Ausdrucksstärke

5.1 Negation

Reines Datalog hat eine Ausdrucksstärke, die äquivalent zu der positiver relationaler Algebra ist, erweitert um Rekursion. Um auch die volle Ausdrucksstärke relationaler Algebra mit einzuschließen wird Datalog um logische Negation erweitert.
Dies ist möglich, wenn von der *Annahme einer geschlossenen Welt* ausgegangen wird. Diese sagt aus, dass wenn ein Fakt sich nicht logisch aus einer Menge Regeln herleiten lässt, er als falsch angenommen wird, wie schon in Definition 2.18 dargelegt. Trotzdem besteht die Möglichkeit, dass zu einem Datalog-Programm mit Negation[25] mehrere Minimalmodelle (nach Definition 2.17) existieren. Nur eines dieser Modelle ist das intuitiv richtige und damit das gesuchte.
Es existieren verschiedene Techniken um Datalog um gültige Negation zu erweitern, so unter anderen *Locally Stratified*, *Modularly Stratified*, das *Stable* und das *Well-Founded Modell*[26] (Siehe dazu in dieser Reihenfolge Przymusinski 1988 [Prz88], Ross 1994 [Ros94], Geflond et al. 1988 [GL88] und Van Gelder et al. 1991 [VGRS91]). Die erste Technik erfordert stratifizierte Programme und liefert ein Minimalmodell, das *perfektes Modell* genannt wird. Die zweite Technik stellt schwächere Bedingungen an die Programme, ist aber auch nicht allgemeingültig. Der Ansatz zu Berechnung der wohlfundierten Semantik arbeitet mit dreiwertiger Logik[27], wobei im abschließend erhaltenen Modell allen Fakten, für die ein eindeutiger Wahrheitswert abgeleitetet werden kann, auch einer zugewiesen ist. Ist das verwendete Programm stratifiziert wird auch das perfekte Modell gefunden. Es können jedoch auch nicht stratifizierte Programme verarbeitet werden, wobei einige Fakten den Wert `undefined` annehmen können. Das Ergebnismodell wird *akzeptables Modell* genannt, ist eindeutig und entspricht der intuitiv erwarteten Lösung.
Für AGDC wird die Auswertung von Negation mit wohlfundierter Semantik gewählt, um mit möglichst wenigen Einschränkungen für die erlaubten Programme auszukommen. Fakten mit dem Wahrheitswert `undefined` werden nicht zu der Ergebnismenge einer Anfrage hinzugerechnet, sondern intern gehalten. Die betroffenen Fakten können durch einen Aufruf von (get−undefined−facts) erhalten werden, wobei immer nur die Ergebnisse der letzten getätigen Anfrage zur Verfügung stehen[28].

[25] `Datalog`$^{\neg}$
[26] Im Folgenden mit dem deutschen Ausdruck *wohlfundierte Semantik* bezeichnet.
[27] `true`, `false`, `undefined`
[28] Diese Funktionalität steht nur zur Verfügung, wenn keine Optimierung durch GSMS verwendet wurde.

Die Verwendung der wohlfundierten Semantik wirft das Problem auf, dass das akzeptable Modell eines Programms P_{SM}, welches mit der Methode der GSMS transformiert wurde, nicht immer äquivalent zum akzeptablen Modell des ursprünglichen Programms P ist. In der Veröffentlichung von Kemp et al. 1991 [KSS91] wird gezeigt, dass die Äquivalenz für die Untermenge der stratifizierten Programme hält. Ebenfalls wird dort eine Äquivalenz für modular stratifizierte Programme bei der Verwendung von *left-to-right*-sips bewiesen und ebenfalls für *well-founded*-sips im Hinblick auf die Verwendung wohlfundierter Semantik. In der finalen Version des Artikels von Kemp et al. 1995 [KSS95] wurde der vorgestellte Ansatz der *well-founded magic sets method* erweitert und ist bei Verwendung beliebiger Arten von sips für alle Programme anwendbar.

5.1.1 Transformationsbasierte Bottom-up-Auswertung

In der Veröffentlichung von Brass et al. 2001 [BDFZ01] wird ein transformationsbasierter Ansatz vorgestellt, der mit $O(n^2)$[29] polynomial ist und korrekt mit Magic Sets arbeitet. Die vorgestellte Methode definiert klare Trennungen zwischen den verschiedenen Schritten und zeigt, dass diese, da sie beliebig kombinierbar sind, genutzt werden können um bekannte Methoden wie u.a. die *well-founded magic sets method* zu beschreiben. AGDC orientiert sich stark an dem dort beschriebenen Ansatz und verwendet einen großen Teil der präsentierten Möglichkeiten.
Brass et al. 2001 [BDFZ01] stellen in ihrer Veröffentlichung sechs Transformationsmethoden vor, die alle eine reduzierende Wirkung auf das Programm haben und damit garantiert terminieren. Alle zusammen bilden ein Transformationsystem.

- Positive reduction (P)
- Negative reduction (N)
- Success (S)
- Failure (F)
- Magic reduction (M)
- Loop detection (L)

Für die Vorstellung eines Transformationssystems nach Brass et al. 2001 [BDFZ01], das sich aus diesen Operationen in bestimmter Reihenfolge angewendet zusammensetzt, müssen noch die folgenden zwei Operatoren definiert werden:

G wie *Grounding* beschreibt die Bildung aller möglichen Ableitungen aus Fakten und Regeln. Entspricht einer Anwendung der semi-naiven Bottom-up-Auswertung bis zum Erreichen des *Least Fixed Point*

***** bezeichnet den *Least Fixed Point*, also den Punkt ab dem keine neuen Änderungen mehr am Programm auftreten

[29] n ist die Größe der extensionalen Datenbank (EDB)

Das folgende Transformationssystem ist ausreichend für die Berechnung der wohlfundierten Semantik:

$$P \longmapsto_{G((PGSGNF)*M)*L)*} \hat{P}$$

Aus dem *Program Reminder* $\hat{P}$ kann dann das akzeptable Modell von P extrahiert werden.
Nachteil dieses Ansatzes ist, dass das Programm wiederholt nach-instanziiert werden muss, da eine vollständige Instanziierung bei vorhandener Negation nicht möglich ist. Brass et al. 2001 [BDFZ01] schlagen deshalb die Verwendung eines *Immediate Consequence Operators* mit *bedingten Fakten* $\bar{T}_P(S)$ vor. Bedingte Fakten sind Regeln, die schon instanziiert wurden, aber noch Unterziele beinhalten, denen noch kein Wahrheitswert zugewiesen werden konnte. Wenn diese mit den noch nicht instanziierten Regeln des Programms inferiert werden, kann ein vollständig instanziiertes Programm abgeleitet werden. Dieses wird dann dem Transformationssystem übergeben.

Definition 5.1 (Immediate Consequence Operator mit bedingten Fakten (aus Brass et al. 2001 [BDFZ01])): Es gelten die selben Voraussetzungen wie in Definition 4.1 und S sei eine Menge von bedingten Fakten. Dann gilt

$$\bar{T}_P(S) := \{A \leftarrow \check{L} \in ground(P) \mid pos(\check{L}) \subseteq heads(S)\}.$$

•

$lfp(\bar{T}_P(S))$ dient dann als Startpunkt des Transformationssystems.
Äquivalent zur vorangegangenen Beschreibung eines Transformationssystems gilt

C bezeichnet Instanziierung mit bedingten Fakten.

womit das Transformationssystem zum Erhalt des akteptablen Modells eines GSMS transformierten Programms unter Verwendung der wohldefinierten Semantik wie folgt definiert wird:

$$P \longmapsto_C ((PSNF) * M) * L) * \hat{P}$$

Unter Berücksichtigung der in Abschnitt 4.1.1 vorgestellten SCC-orientierten Auswertung wird das Transformationssystem auf jede Untermenge $P^{(i)}$ von P angewendet. Brass et al. 2001 [BDFZ01] führen zu diesem Zweck noch den Operator $\hat{T}_{P,R}(S)$ ein.

Definition 5.2 (Operator $\hat{T}_{P,R}(S)$ (aus Brass et al 2001 [BDFZ01])): Es gelten die selben Voraussetzungen wie in Definition 5.1 und R sei ein instanziiertes Programm. Dann berechnet der Operator $\hat{T}_{P,R}(S)$ die nachfolgende Menge von bedingten Fakten:

$$\hat{T}_{P,R}(S) := \{A \leftarrow \check{L} \in ground(P) \mid pos(\check{L}) \subseteq heads(R) \cup heads(S) \; and \; neg(\check{L}) \cap facts(R) = \emptyset\}.$$

•

Die R_i orientieren sich an den $P^{(i)}$ von P und $R_0 := \emptyset$, R_i ist der *Program Reminder* von $R_{i-1} \cup lfp(\hat{T}_{P^{(i)},R_{i-1}})$, für $0 < i \leq m$. R_m ist dann der *Program Reminder* von P.

5.1.2 Auswertung nach SCC

Wenn das Programm wie in Abschnitt 4.1.1 vorgestellt nach SCC partitioniert wird, können diese Subprogramme der Reihe nach ausgewertet werden. Somit müssen weniger Regeln im Speicher gehalten werden. Wegen der Verwendung der wohldefinierten Semantik zu Auflösung von Negation müssen nicht nur die Fakten mit dem Wahrheitswert `true`, sondern auch solche mit dem Wahrheitswert `undefined` an die jeweils nächste Runde der SCC-Auswertung weitergereicht werden.
Details zur SCC-orientierten Auswertung mit Negation finden sich in der Veröffentlichung von Brass et al. 2001 [BDFZ01, S.14, Definition 35 und Theorem 36]. Um die im vorangegangenen Abschnitt vorgestellte Auswertung im SCC-Kontext verwenden zu können, müssen noch einige Änderungen vorgenommen werden.
Der bei der Instanziierung mit EDB-Fakten erstellte EDB-Cache wird nach Durchführung der Instanziierung mit EDB-Fakten nicht mehr verworfen, sondern gehalten und für die Auswertung der nächsten SCC-Ebene wiederverwendet. Somit erhöht sich die Anzahl der EDB-Store Zugriffe durch die SCC-orientierte Auswertung nicht.
Des Weiteren muss die Ergebnismenge jeder SCC-Ebene für die nächste Ebene zur Verfügung stehen und dort bei der Instanziierung mit Fakten Verwendung finden.

5.1.3 Negation und GSMS

Der genutzte Ansatz funktioniert nicht bei beliebiger Wahl von sips wenn eine Optimierung durch die GSMS Methode durchgeführt wurde. Eine Verwendung von *complete-left-to-right* sips führte bei Regeln mit mehreren negierten Körperliteralen nicht mehr zum aktzeptablen Modell, da die Bindung zwischen den negierten Literalen gelöst wird. Es reicht aus, dass eines der Literale den Wert `wahr` zugewiesen bekommt, damit der Inferenzmechanismus den Kopf der Regel ebenfalls als `wahr` ansieht. Die Verwendung der für AGDC entworfenen *non-negated-left-to-right* sips, die nur bis zum ersten Auftreten eines negierten, nicht arithmetischen Literals angewendet wird, wie in vorangegangenen Abschnitt 4.2.2 beschrieben, führt zum gewünschten Ergebnis.

5.2 Eingebaute Prädikate

Eingebaute Prädikate, im Folgenden mit BIP bezeichnet, werden nicht explizit in der EDB abgelegt, sondern lösen eine eingebaute Funktion des Systems aus. Zu Verfügung stehen die vergleichenden Operanden $>, <, <=, >=, ==$ und $/=$. Dargestellt werden BIPs in Infix-Notation, wofür das Regelmakro :− erweitert wird.

```
(:-  ...
        (  [([not]  predicate  [var|const]*)  |
            ([not]  var|const  <|<=|>|>=|==|/=  var|const)])  )
```

Statt eines Literals ist auch die Angabe eines BIP als Unterziel einer Regel möglich. Es ist zu beachten, dass die in Abschnitt 3.5 aufgeführte Bereichsbeschränkung für BIPs ebenfalls gilt.

Beispiel 5.1: .

$$\texttt{p(X,Y)} :- \texttt{q(X,Y)}, \texttt{X} < \texttt{Y}$$

wird für AGDC formuliert als

```
(:- (p ?x ?y)
    ((q ?x ?y) (?x < ?y)))
```

◇

Eingebaute Prädikate werden ausgewertet sobald alle ihre Variablen instanziiert sind. Als Funktionen werden die von AlegroGraph bereitgestellten Methoden zum Vergleich von UPIs verwendet, da Konstanten intern als solche repräsentiert werden. Bei diesen handelt es sich um upi= und upi<. Bei der Verwendung von BIPs für Regeln muss die genaue Verhaltensweise dieser Methoden im Auge behalten werden, da sie teilweise von den intuitiv erwarteten Lösungen abweicht. Ein Beispiel hierfür ist der Vergleich zweier als Typ `resource` bezeichneter UPIs mit den Werten "5" und "4". Ob (upi< "5" "4") ein t oder ein nil zurückgibt hängt nicht von dem Wert der Zahlen, sondern von weniger offensichtlichen Faktoren ab. Sind "5" und "4" allerdings als `int` bezeichnet, liefert (upi< "5" "4") das intuitiv erwartete Ergebnis.
Die folgende Tabelle 4 gibt einen Auszug aus den Verhaltenweisen von upi< in Bezug auf die übergebenen Typen.

Typen	<	>
num(4) & num(5)	t	nil
num(4) & literal(4)	nil	t
num(4) & literal(a)	nil	t
literal(4) & literal(5)	t	nil
literal(4,1) & literal(4.0)	t	nil
literal(a) & literal(b)	t	nil
resource(a) & resource(b)	nil	t

Tabelle 4: Auszug aus dem upi< Verhalten, *num* steht für alle Zahlentypen

5.2.1 Anpassen der angewendeten Methoden

Die Variable *build−in−predicates* im Paket `:agdc.inference.bip` enthält die Richtlinien, welche Funktionen beim Auftreten eines BIP ausgelöst werden. Mit Anlage einer Filterfunktion, die die Typen der Konstanten überprüft und von diesen Informationen ausgehend die richtige Auswertungsfunktion wählt, kann das Verhalten den persönlichen Anforderungen angepasst werden.

5.3 Typbezeichner

AlegroGraph erkennt RDF-Typinformationen wie xsd:integer in bestimmten Fällen[30] und verarbeitet diese entsprechend. Die Informationen über diese sind in der UPI mit abgelegt. Daneben definiert AlegroGraph auch einige eigene eingebaute Typen. Für jedes Element werden neben dem Namespace und dem Wert auch ein Typbezeichner mitgespeichert.
Es wäre wünschenswert, wenn die Möglichkeit bestehen würde Regelvariablen auf bestimmte Typen zu beschränken. Zum Beispiel ist es vorstellbar, dass die Regel

$$\texttt{gt(X,Y)} :- \texttt{num(X)}, \texttt{num(Y)}, \texttt{X} > \texttt{Y}.$$

nur mit Fakten inferiert werden sollte, die vom Typ `int` sind. Um dies zu ermöglichen enthält AGDC ein Typensystem, mit dem Variablen auf einen bestimmen, AlegroGraph bekannten, Typ beschränkt werden können. AlegroGraph kennt zwei Arten von Typbezeichnern: Einmal Typ-Codes und einmal Typ-Namen in Form von Lisp-Symbolen. Eine Liste der unterstützten Typen kann mit der Funktion (supported−types) ausgegeben werden. Die Lisp-Symbole können mit (type−name−>type−code <name>) in die entsprechenden Typ-Codes und mit (type−code−>type−name <code>) wieder zurückgewandelt werden. Die Tabelle 8 in Anhang E gibt eine Übersicht über die in Version 3.2 von AlegroGraph unterstützten Typen. Bei der Definition einer Regel mit dem :− Makro kann jeder Variable und Konstante ein Typbezeichner zugeordnet werden:

```
(:- (...)
    ( (pred-symbol [variable |
                    constant |
                    (variable type-id) |
                    (constant type-id)]*) ))
```

type−id kann dabei in der Symbol- oder der Codeform auftauchen. Die betroffene Variable unifiziert dann nur noch mit Werten, die vom gleichen Typ sind.
Wenn Future-Parts für die Definition einer Konstanten verwendet werden, können die Typinformationen auch in der dort erlaubten Form wie beispielsweise !5^^<http://www.w3.org/2000/01/rdf−schema#integer> übergeben werden. Näheres zu dieser Syntax kann in der AlegroGraph Dokumentation gefunden werden.
Prädikate definieren sich somit nicht nur über ihr Prädikatensymbol und ihre Stelligkeit, sondern auch über die Typbezeichner ihrer Variablen. Ist kein Typbezeichner angegeben, unifiziert die Variable mit allen Werten, unabhängig von deren Typ. Kein Typbezeichner fungiert somit als Wildcard. Das bedeutet, dass p((?x 20)) und p(?x) zwei unterschiedliche Prädikate darstellen. Existiert allerdings eine Regel $\texttt{q(X)} :- \texttt{p(X)}$ so unifiziert deren Unterziel p(?x) mit beiden Prädikaten.
Bei der Verwendung von Konstanten bei einer Regeldefinition ist zu beachten, dass ein Weglassen eines Typbezeichners zu einer Behandlung des Konstantenwertes als Ressource[31] führt und sie sich somit nur mit Fakten dieses Typs deckt.

[30] Bisher nur beim parsen von RDF-Dateien im N-Triple-Format und nach Festlegung eines entsprechenden Mappings

[31] Die zugehörige UPI wird durch (upi (resource <constant>)) erstellt.

Konstanten sollten deshalb immer mit einem Typbezeichner versehen werden, wenn sie nicht explizit eine Ressource darstellen sollen.
Bei Variablen ist noch zu beachten, dass das erste Auftauchen oder auch Weglassen eines Typbezeichners bindend ist.
Bei einer Definition der Form (:− (p (?x : int)) ((q ?x))) wird ?x auch für das Unterziel (q ?x) an den Typ : int gebunden. Daraus folgend wären die beiden Regeldefinitionen (:− (p (?x 20)) ((q (?x 21)))) und (:− (p ?x) ((q (?x 21)))) ungültig, da versucht wird die Variable ?x mit zwei inkonsistenten Typbezeichnern zu belegen. Das Makro :− würde in diesen Fällen einen Fehler aufwerfen und die Regel nicht in die IDB einspeisen.

6 Implementierung

Während in den vorangegangenen Abschnitten 4 und 5 die beschrieben Strategien und Möglichkeiten allgemein behandelt wurden, beschäftigt sich dieser Abschnitt mit der konkreten Implementierung des Inferenzmechanismus im Rahmen dieser Arbeit. Das Vorgehen bezieht sich auf die mit dieser Arbeit abgegebene Version 1.0 von AGDC. Im letzten Abschnitt werden noch einige Punkte vorgestellt, die AGDC sinnvoll erweitern könnten, deren Umsetzung im Rahmen dieser Arbeit aber nicht möglich war.
Für AGDC in der abgegebenen Version wird von einer Komplettinstanziierung mit bedingten Fakten abgesehen, da die Implementierung dieser Optimierungsstrategien noch nicht weit genug fortgeschritten ist, um verlässliche Ergebnisse zu liefern.
Umgesetzt wird ein Inferenzmechanisums, der ein Datalogprogramm nach SCC partitioniert. Die einzelnen Partitionen werden erst einmalig mit EDB-Fakten instanziiert. Daraufhin werden sie wiederholt mit abgeleiteten Fakten instanziiert und parallel eine Auflösung von Negationen unter Verwendung des Transformationssystems betrieben. Der Mechanismus liefert für Auswertung ohne und mit Anwendung von GSMS verlässliche Ergebnisse, AGDC kann somit für Testanwendungen genutzt werden. Allerdings kann die Auswertung in polynomialer Zeit $O(n^2)$ zur Anzahl der Fakten n nicht garantiert werden. Dies wird für die nächste Version von AGDC angestrebt wodurch dann auch eine Anwendung in zeitkritischem Kontext ermöglicht wird.
Im Folgenden werden die einzelnen Schritte des Inferenzmechanismus in der Reihenfolge des Auftretens beschrieben, wie im Pseudocode in Liste 1 dargelegt wird.

```
inference-mechanism (OrgProgram, Query)
  if Magic
    OrgProgram = rewrite-with-GSMS(OrgProgram, Query)
  end if
  for Program in scc-partitioning(OrgProgram, Query)
    Program = ground-with-edb(Program)
    Program = ground-with-facts(Program)
    while changes
      while changes
        while changes
          Program = positive-reduction (Program)
          Program = ground-with-facts(Program)
          Program = success(Program)
          Program = ground-with-facts(Program)
          Program = negative-reduction(Program)
          Program = failure(Program)
        end while
        Program = magic-reduction(Program)
        Program = ground-with-facts(Program)
      end while
      Program = loop-detection(Program)
    end while
  end for
end inference-mechanism
```

Listing 1: Pseudocode des Inferenzmechanismus von AGDC 1.0

6.1 Elemente des Inferenzmechanismus

6.1.1 Partitionierung nach SCC

Für eine Partitionierung des Programms nach SCC wird im ersten Schritt ein Abhängigkeitsgraph der Regeln erstellt. Auf diesen wird im zweiten Schritt ein an den in der Arbeit von Pearce 2005 [Pea05] angelehnter Algorithmus angewendet, der die Regeln partitioniert nach SCC zurückgibt. Aus diesen einzelnen Programmen werden in einem abschließenden Schritt noch einmal einige Redundanzen entfernt um somit schließlich eine Menge aus linear abhängigen Einzelprogrammen zu erhalten.
Die weiteren Schritte des Inferenzmechanismus werden für diese Programme jeweils einzeln ausgeführt, wobei die Fakten und instanziierten Regeln der vorausgegangenen Runde immer an die jeweils nächste Runde weitergegeben werden. Die Fakten dienen der weiteren Ableitung, während die instanziierten, aber nicht aufgelösten Regeln nach der Definition der wohlfundierten Semantik benötigt werden, da ihre Köpfe als Fakten mit dem Wahrheitswert `undefined` fungieren.
Die Details und Definitionen zur SCC-orientierten Auswertung sind u.a. in der Veröffentlichung von Brass et al. 2001 [BDFZ01] beschrieben.

6.1.2 Instanziierung mit EDB-Fakten

Definition 6.1 (Instanziieren mit EDB-Fakten): M_{EDB} sei die Menge aller Fakten der EDB und P ein nicht-instanziiertes, bereichsbeschränktes Programm. $P_{G^{edb}}$ folgt aus P wenn es eine Regel $A \leftarrow \check{L}$ in P gibt und ein positives Literal $L \in \check{L}$, so dass eine Substitution σ mit $L\sigma = F, F \in M_{EDB}$ existiert, und $P_{G^{edb}} = P \cup \{(A \leftarrow (\check{L} - \{L\}))\sigma\}$. •

EDB-Fakten werden mit den Regeln von P in allen möglichen Ausprägungen verschränkt und *Success* sofort angewendet, da das unifizierte Unterziel aus der Regel entfernt wird. Da IDB und EDB endlich sind, wird $P_{G^{edb}}$ eventuell erreicht. $P_{G^{edb}}$ enthält nun Fakten, instanziierte Regeln, halb-instanziierte Regeln und nicht-instanziierte Regeln.
Die EDB-Fakten sollen nur ein einziges Mal berücksichtigt werden, da der Zugriff auf den EDB-Store eine zeitintensive Operation darstellt. Die Implementation der Instanziierung mit EDB-Fakten arbeitet deshalb mit einem internen Cache, so dass schon einmal ausgelesene EDB-Fakten nicht erneut einen Zugriff auf den EDB-Store auslösen.
Die ausgelesenen EDB-Fakten werden im weiteren Verlauf der Auswertung nicht mehr benötigt, sondern erst wieder zum Schluss der Auswertung der Faktenmenge des Ergebnisprogramms hinzugefügt.
Dieses Vorgehen ist möglich, da alle möglichen Verschränkungen mit den Regeln hergestellt wurden und sie für die Transformationsmethoden nicht benötigt werden: *Loop Detection*, *Failure* und *Positive Reduction* betreffen die Fakten des Programms nicht und *Success* wird nach Definition 6.1, soweit die Transformationsmethode die EDB-Fakten betrifft, noch während der Instanziierung angewendet. *Negative Reduction* betrifft nur negative Literale, die in der Instanziierung mit EDB-Fakten nicht behandelt werden.
Bisher sind negative Literale im Körper der Regel, die mit einem EDB-Fakt unifizieren, unberücksichtigt geblieben. Diese werden allerdings für die `Negative Reduction` Transformation benötigt, weshalb sie ebenfalls aus dem EDB-Store ausgelesen und in einer Faktenmenge gehalten werden. Die folgende Definition 6.2 stellt den Formalismus zum Erhalt der Fakten dar; in der Implementierung wird dieser Prozess mit der Instanziierung mit EDB-Fakten verbunden, um den dort verwendeten Cache nutzen zu können, falls ein EDB-Fakt in positivem sowie negativem Kontext auftaucht. Die Menge der erhaltenen Fakten wird im Folgenden mit M_{NF} bezeichnet.

Definition 6.2 (Extrahierung von EDB-Fakten mit negativem Kontext): M_{EDB} sei die Menge aller Fakten der EDB und P ein nicht-instanziiertes, bereichsbeschränktes Programm. M_{NF} stellt die Menge aller EDB-Fakten dar, die mit den negativen Literalen der Regeln eines Programms P unifizieren, wenn es eine Regel $A \leftarrow \check{L}$ in P gibt und ein negatives Literal $L \in \check{L}$, so dass eine Substitution σ mit $L\sigma = F, F \in M_{EDB}$ existiert, und $M_{NF} = \{F\}$. •

Mit dieser Menge wird direkt die *Negative Reduction* des Transformationssystems angewendet; die EDB-Fakten in M_{NF} werden im Folgenden nicht mehr benötigt.
Die halb-instanziierten und nicht instanziierten Regeln in $P_{G^{edb}}$, die ein posi-

tives Unterziel enthalten, das niemals abgeleitet werden kann, können entfernt werden. Dafür wird die *Failure für nicht instanziierte Regeln* Transformation in 6.3 definiert.

Definition 6.3 (Failure für nicht instanziierte Regeln): P_1 und P_2 seien nicht instanziierte, bereichsbeschränkte Programme. P_2 folgt aus P_1 mit *Failure für uninstanziierte Regeln* ($P_1 \mapsto_U P_2$) wenn es eine Regel $A \leftarrow \check{L}$ in P_1 gibt und ein positives Literal $L \in \check{L}$, so dass es keine Regel über L in $P1$ gibt, d.h. es existiert keine Substitution σ, so dass $L\sigma = h, \forall h \in heads(P_1)$, und $P_2 = P_1 - \{A \leftarrow \check{L}\}$. •

6.1.3 Instanziierung mit Fakten

Die Instanziierung eines Programms beschreibt eine Verschränkung der Regeln mit einer übergebenen Menge von Fakten in allen möglichen Ausprägungen. Dabei werden neue Fakten abgeleitet, die in der nächsten Runde der Instanziierung mit Fakten verwendet werden, bis eventuell der *Least Fixed Point* Haltepunkt erreicht wird.

Definition 6.4 (Instanziierung mit Fakten): I sei die Menge aller Fakten die noch nicht auf P angewendet wurden und P ein nicht-instanziiertes, bereichsbeschränktes Programm. P_{G^f} folgt aus P wenn es eine Regel $A \leftarrow \check{L}$ in P gibt und ein positives Literal $L \in \check{L}$, so dass eine Substitution σ mit $L\sigma = F, F \in I$ existiert, und $P_{G^f} = P \cup \{(A \leftarrow (\check{L} - \{L\}))\sigma\}$. •

Die Instanziierung mit Fakten wie sie in 6.4 definiert wird, entspricht ziemlich genau der Instanziierung mit EDB-Fakten, mit dem Unterschied dass kein Zugriff auf die EDB-Datenbank benötigt wird. Allerdings ist eine wiederholte Anwendung vorgesehen, bis der *Least Fixed Point* erreicht wurde. Dafür wird der Standard Immediate Consequence Operator, wie er in Definition 4.1 definiert wurde, leicht abgeändert. Statt der Herbrand Instanziierung wird eine Instanziierung des Programms durch Fakten gebildet. Die *Success* Transformation ist dabei direkt in der Instanziierung mit Fakten enthalten, so dass nur die Regeln ohne Körperliterale zur Bestimmung der folgenden Faktenmenge verwendet werden.

Definition 6.5 (Facts Immediate Consequence Operator (nach Brass et al. 2001 [BDFZ01])): Sei P ein bereichsbeschränktes Datalogprogramm, $fground(P, I)$ die Instanziierung mit Fakten von P, A der Kopf einer Regel und I eine Menge von Fakten. Dann gilt

$$T_P^F(I) := \{A \mid \text{es gibt ein } A \leftarrow \emptyset \in fground(P, I)\}.$$

•

$\hat{P}_{G^F}$ wird somit durch $lfp(T^F_{P_{G^F}})$ erreicht und stellt eine teilweise Instanziierung von P dar. Da negative Unterziele jedoch nicht aufgelöst werden können, muss die Instanziierung mit Fakten nach jedem Schritt des Transformationssystems zur Auflösung von Negation, der neue abgleitete Fakten produziert, erneut

angewendet werden. Die Transformationsmethoden, die diese Eigenschaft aufweisen sind *Positive reduction*, *Success* und *Magic Reduction*. Dem Pseudocode des Inferenzmechanismus in Liste 1 kann der genaue Zeitpunkt der erneuten Instanziierung mit Fakten entnommen werden.

6.1.4 Programmreduktion durch das Transformationssystem

Das verwendete Transformationssystem nach Notation eines solchen wie sie in der Veröffentlichung von Brass et al. 2001 [BDFZ01] beschrieben wird und mit der Erweiterung um G die in Abschnitt 5.1.1 vorgenommen wurde, ergibt sich zu

$$P \mapsto_{G((PGSGNF)*M)*L)*} \hat{P}.$$

Die Transformationsmethoden werden jeweils nur auf die instanziierte Untermenge des Programms angewendet, mit der Ausnahme von *Failure*. Hier müssen nicht nur die Regelköpfe der instanziierten Regeln mit den positiven Körperliteralen verglichen werden, sondern auch die Köpfe der nicht oder halbinstanziierten Regeln auf eine mögliche spätere Unifizierung überprüft werden. Durch Anwendung des Transformationssystems auf $\hat{P}_{G^{cf}}$ erhält man den *Program Reminders* $\hat{P}$ von P, d.h. $\hat{P}_{G^{cf}} \mapsto_{G((PGSGNF)*M)*L)*} \hat{P}$. Das Ergebnis der Anfrage wird abschließend aus der Überschneidung der Ergebnisfakten und der EDB-Faktenmenge $facts(\hat{P}) \cup M_{EDB}$ gefiltert.

6.2 Mögliche Erweiterungen

Stärkere Verschränkung von Instanziierung und Transformationssystem Version 1.0 von AGDC orientiert sich stark an der formalen Definition des Transformationssystems, die für eine Implementierung teilweise ungünstig ist da eine Reduktion der Regelmenge nicht zum frühest möglichen Zeitpunkt stattfindet. In der Veröffentlichung von Niemelä et al. 1996 [NS96] wird eine effiziente Verschränkung des $PSNF*$-Teiles des Transformationssystems mit der Instanziierung durch Fakten beschrieben und auch in der Dissertation von U. Zukowski 2001 [Zuk01, Chapter 10] wird ein Ansatz geboten, der dort das gesamte Transformationssystem berücksichtigt, aber allgemeiner gehalten ist.

Einführung bedingter Fakten Die Einführung der Instanziierung mit bedingten Fakten, wie sie in den Arbeiten von Brass et al. 2001 [BDFZ01] und Zukowski 2001 [Zuk01] definiert wird bietet eine Möglichkeit, keine halb-instanziierten Regeln mehr im Speicher halten zu müssen. Dies kann, aus den schon zur SCC-orientierten Auswertung in Abschnitt 4.1.1 aufgeführten Gründen (Reduktion der Iterationschritte), neben einer Verminderung des benötigten Speichers zu einem Geschwindigkeitsgewinn führen.

Durch die Einführung der vorgestellten Erweiterungen sollte eine Auswertung in polynomialer Zeit $O(n^2)$ zur Anzahl der Fakten n erreichbar sein und es wird erwartet, dass sich die Geschwindigkeit pro Element des Programms noch einmal signifikant erhöht.

Disjunktives Datalog Ein großer Schritt nach vorne wäre die Einführung von disjunktivem Datalog (siehe u.a. die Veröffentlichungen Eiter et al. 1997 [EGM97], Leone et al. 1997 [LRS97] und Cumbo et al. 2004 [CFGL04] zu diesem Thema). Es würde so eine Umsetzung eines weiteren Teils von OWL in Datalog ermöglicht, da die Ausdrucksstärke von disjunktivem Datalog höher ist, als die von nicht disjunktivem Datalog. Allerdings ist eine effiziente Umsetzung sehr aufwendig, weshalb diese Erweiterung ein Projekt für die Zukunft ist.

7 Lehigh University Benchmark (LUBM)

LUBM[32] ist ein RDFS/OWL Reasoning Benchmark, der an der Lehigh Universität entwickelt wurde und zur Evaluierung von Reasonern über große Repositories verwendet wird. Er besteht aus einer Universitäts Domänen Ontology, anpassbaren und reproduzierbaren Daten und einer Menge von 14 Testanfragen, die sich in Selektivität und der Menge der betroffenen Daten unterscheiden. Die Universitäten sind in Departments aufgeteilt, denen jeweils akademische und nicht-akademische Mitarbeiter sowie Studenten angehören.
Das von AGDC verwendete Datalog¬ besitzt genügend Ausdruckskraft um die meisten der in OWL formulierten Zusammenhänge darzustellen, weshalb der Benchmark auch für die Belastungstestung von AGDC geeignet ist. Zur Testung selber wird eine abgewandelte Form der 12. Anfrage genutzt wie in Liste 3 dargestellt, genauer die Anfrage nach den Chairs der Departments, allerdings ohne Beschränkung auf diejenigen der Universität `http://www.University0.edu`. Der dafür relevante Teil der OWL Beziehungen ist in den in Liste 2 gelisteten sieben Datalogregeln ausgedrückt. Es sind nicht alle möglichen Personenbeziehungen durch die Regeln abgebildet, da diese größtenteils für die Anfrage irrelevant sind.

```
(:- (!ub:Chair ?x)
    ((!ub:Person ?x) (!ub:headOf ?x ?y) (!ub:Department ?y)))

(:- (!ub:FullProfessor ?x)
    ((!rdf:type ?x !ub:FullProfessor)))

(:- (!ub:Professor ?x)
    ((!ub:FullProfessor ?x)))

(:- (!ub:Faculty ?x)
    ((!ub:Professor ?x)))

(:- (!ub:Employee ?x)
    ((!ub:Faculty ?x)))

(:- (!ub:Person ?x)
    ((!ub:Employee ?x)))

(:- (!ub:Department ?x)
    ((!rdf:type ?x !ub:Department)))
```

[32] http://swat.cse.lehigh.edu/projects/lubm/, Stand 30.06.2009

Listing 2: Relevante Ontologieuntermenge für die 12. Anfrage

```
(?- !ub:Chair ?x)
```

Listing 3: Abgewandelte Form der 12. Anfrage

Der Benchmark wird einmal mit und einmal ohne GSMS Transformation für verschiedenen Faktenmengen durchgeführt; Ersteres wird in der Ergebnisabbildung mit *:semi*, Zweiteres mit *:magic* bezeichnet. Die Verarbeitung von Negation wurde in beiden Fällen deaktiviert. Die Daten zu jeder Universität sind in durchschnittlich 20 Departments partitioniert und bestehen insgesamt aus etwas mehr als 100.000 EDB-Fakten. Jedem Department ist ein Chair zugeordnet. LUBM(z) bezeichnet im Folgenden einen Durchlauf des Benchmarks unter Verwendung der Daten zu z Universitäten und dementsprechend ungefähr z$*$20 Deparments und z$*$100.000 Fakten.
Durchgeführt wurde der Benchmark auf einem Computer mit Linux Betriebsystem, einem Xeon X3320 64bit Prozessor mit 2,5 GHz Taktung und 8704 GB RAM unter Verwendung von Alegro CL 8.1 und AlegroGraph 3.2. Die verwendete AGDC Version ist 1.0. Die Zeiten zur Anfragebeantwortung sind in Grafik 3 dargestellt.
Ohne GSMS Transformation steigt die Auswertungszeit exponentiell zur Faktenmenge an, mit GSMS wird ein polynomialer Anstieg der Auswertzeit erzielt. Es ist nach der Veröffentlichung von Niemlä et al. 1996 [NS96] aber möglich für Datalog ohne Negation einen linearen Anstieg ($O(n)$) zu erzielen. Die für die nächste Version geplante Implementierung des dortigen Instantiierungsalgorithmus sollte zu diesem Ergebnis führen.
Die Auswertzeiten sind im direkten Vergleich sehr viel länger, als bei einer Verwendung des AlegroGraph eigenen RDFS/OWL Reasoners RDFS++, wie die auf http://www.franz.com/agraph/allegrograph/agraph_bench_lubm.lhtml[33] veröffentlichten LUBM(50) Ergebnisse zur AlegroGraph Version 3.2 deutlich machen. Eine praktische Verwendung von AGDC außerhalb von Testzwecken ist deshalb derzeit nicht zu empfehlen, wenn nur eine kleine Untermenge der OWL Ausdruckskraft benötigt wird, die von RDFS++ abgedeckt wird. In diesem Fall ist eine Verwendung des AlegroGraph eigenen Reasoners sinnvoller.
Allerdings ist das von AGDC verwendete Datalog sehr viel allgemeiner gehalten und nicht auf OWL zugeschnitten, so dass eine langsamere Antwortgeschwindigkeit erwartet wurde. Für Testszenarien, die eine große Ausdrucksstärke benötigen, nicht zu große Faktenmengen betreffen und keine Zeitvorgaben haben ist AGDC gut geeignet.

[33] Stand: 30.06.2009

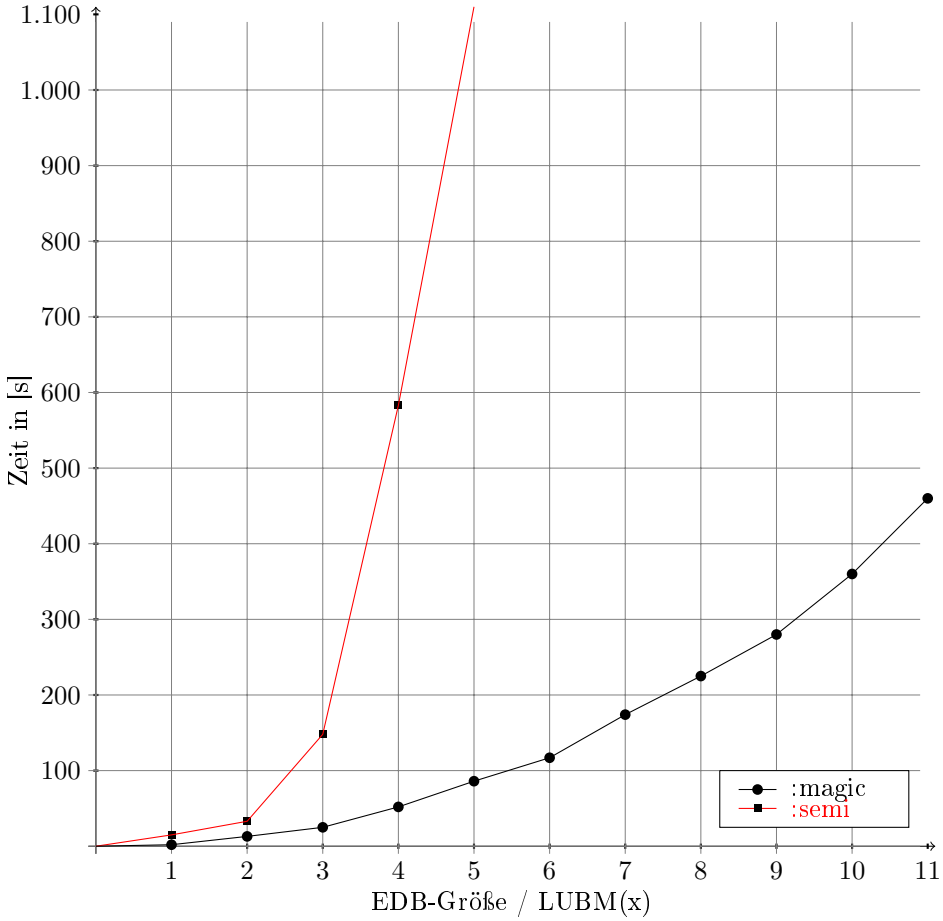

Abbildung 3: LUBM: Chair Query mit 7 Regeln

8 Zusammenfassung der Ergebnisse und Diskussion

Ziel dieser Arbeit war der Entwurf und die Implementierung einer deduktiven Komponente für AlegroGraph. Im Schlusswort wird das Ergebnis unter einigen Aspekten betrachtet, die für eine Nutzung im Kontext des Semantic Web interessant sind.

Ein Inferenzmechanismus, das Herzstück einer deduktiven Datenbank, erfüllt idealerweise einige Voraussetzungen. So sollte *Korrektheit* gegeben sein, es sollten also gültige Aussagen geliefert werden. AGDC genügt dieser Bedingung in allen bisher durchgeführten Tests. Ein weiterer wichtiger Punkt ist *Vollständigkeit*. AGDC liefert zu sämtlichen Anfragen zuverlässig alle aus der Fakten- und Regelmenge schließbaren Antworten. Des Weiteren wird *Entscheidbarkeit* verlangt, die bei einer Verwendung von Datalog schon in der zugrundeliegenden Semantik begründet liegt und somit von AGDC erreicht wird. Neben diesen drei Forderungen ist auch noch die *Effizienz* von großer Bedeutung. Die Antwortzeiten der bisherigen Version von AGDC sind noch zu lang, um eine wirklich effektive Anwendung zu erlauben. Dies liegt einmal an der Ausdrucksstärke des erlaubten Datalog, die einen aufwendigen Schlussfolgerungsalgorithmus erfordert. Andererseits ist der Entwurf und die Umsetzung eines effektiven Auswertungsalgorithums mit mehr Aufwand verbunden, als in der für eine Bachelorarbeit verfügbaren Zeit möglich ist. Es gibt viele Ansätze zur Weiterentwicklung und Optimierung, wie auch teilweise in dieser Arbeit beschrieben, die die Effizienz signifikant steigern könnten.

Gerade im Kontext des Semantik Web sind aufgrund seiner Struktur noch andere Punkte interessant, die teilweise den vorangehend genannten, klassischen Anforderungen widersprechen, wie in der Veröffentlichung von Balzer 2004 [Bal04] dargelegt wird.
Das wichtigste Merkmal, dass eine für das Semantic Web nutzbare deduktive Datenbank aufweisen sollte, ist *Skalierbarkeit*. Die verfügbaren Daten und damit die Wissensbasis des Inferenzmechnanismus wachsen mit großer Geschwindigkeit an. Um auch mit einer solchen Datenmenge umgehen zu können, sollte der Anstieg der Auswertungszeit eine geringe maximale Obergrenze aufweisen. Die theoretischen Grundlagen der in AGDC umgesetzten Konzepte ermöglichen eine Grenze von $O(n)$ (linear) für Datalog ohne Negation und $O(n^2)$ (polynomial) für Datalog mit Negation. Dieses Ziel konnte nicht erreicht werden.
Des Weiteren sollte der Inferenzmechanismus für das Semantic Web effektiv mit fehlerhaften Daten umgehen können. Gerade die Fakten der EDB werden häufig von unkontrollierten Quellen bereitgestellt. Dieser Gesichtspunkt wiegt für AGDC nicht so schwer, da die Korrektheit der EDB-Fakten durch AlegroGraph gesichert wird. Zugrundeliegende IDB-Regeln unterliegen der Kontrolle von AGDC und werden auf syntaktische Fehler überprüft. Sich widersprechende Regeln und andere Fehler dieser Ausprägung werden aber bisher nicht erkannt.

Für eine annähernd komplette Abbildung von OWL ist disjunktives Datalog erforderlich. Die Forschung in diesem Bereich wurde in den letzten Jahren intensiviert (siehe u.a. die Veröffentlichungen von Either et al. 1997 [EGM97], Leone et al. 1997 [LRS97], Leone et al. 2006 [LPF+06] und Cumbo et al. 2004

[CFGL04]), so dass eine effektive Implementation möglich sein sollte. Da eine Entwicklung und Umsetzung den Rahmen dieser Arbeit bei weitem überschreiten würde, wurde von der Unterstützung von disjunktivem Datalog abgesehen. Um den Schlussfolgerungsmechanismus von Datalog für OWL vollständig nutzbar zu machen, wird eine entsprechende Unterstützung jedoch notwendig sein.

Abschließend lässt sich sagen, dass AGDC einen vielversprechenden Ansatz aufzeigt, um Schlussfolgerungen mit Ontologien im Kontext des Semantic Web zu realisieren. Es ist allerdings noch weiterführende Arbeit notwendig, bis eine praktische Anwendung in Betracht gezogen werden kann.

Literatur

[Bal04] Steffen Balzer. Inferenzsysteme für das semantic web, 02 2004.

[BDFZ01] Stefan Brass, Jürgen Dix, Burkhard Freitag, and Ulrich Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory Pract. Log. Program.*, 1(5):497–538, 2001.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.

[BR91] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3-4):255–299, 1991.

[CFGL04] Chiara Cumbo, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Enhancing the magic-set method for disjunctive datalog programs. In *In Proc. 20th International Conference on Logic Programming (ICLP 04), Springer LNCS 3132*, pages 371–385. Springer, 2004.

[CG94] Armin B. Cremers and Ralf Griefhahn, Ulrike und Hinze. *Deduktive Datenbanken: Eine Einführung aus der Sicht der logischen Programmierung*. Vieweg, 1994.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, 1989.

[CM03] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag (Berlin, New York), 5 edition, 2003.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. pages 48–57, 2003.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *In Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080. MIT Press, 1988.

[KSS91] David B. Kemp, Peter J Stuckey, and Divesh Srivastava. Magic sets and bottom-up evaluation of Well-Founded models. *In Proceedings of the 1991 Int. Symposium on Logic Programming*, pages 337—351, 1991.

[KSS95] David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theor. Comput. Sci.*, 146(1-2):145–184, 1995.

[LPF+06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[LRS97] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: unfounded sets, fixpoint semantics, and computation. *Inf. Comput.*, 135(2):69–112, 1997.

[NS96] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303. MIT Press, 1996.

[Nus91] Miguel Nussbaum. *Building a deductive database.* Ablex Publishing Corporation, 1991.

[PCWT96] Norman Paton, Richard Cooper, Howard Williams, and Philip Trinder. *Programmiersprachen für Datenbanken.* Prentice Hall Verlag GmbH, 1996.

[Pea05] David J. Pearce. An improved algorithm for finding the strongly connected components of a directed graph. Technical report, Victoria University, Wellington, NZ, 2005.

[Prz88] T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[Ros94] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, 1994.

[Sch91] Helmut Schmidt. *Meta-Level Control for Deductive Database Systems*, volume 479/1991 of *Lecture Notes in Computer Science*, chapter 2. A standard deductive database system, pages 5–32. Springer Berlin / Heidelberg, 1991.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

[Wal87] Christopher Walther. *A many-sorted calculus based on resolution and paramodulation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[ZFB97] Ulrich Zukowski, Burkhard Freitag, and Stefan Brass. Improving the alternating fixpoint: The transformation approach. In *LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 4–59, London, UK, 1997. Springer-Verlag.

[Zuk01] Ullrich Zukowski. *Flexible Computation of the Well-Founded Semantics of Normal Logic Programs.* PhD thesis, Universität Passau - Fakultät für Mathematik und Informatik, Februar 2001.

A Datenstruktur der Regeln

Es ist sinnvoll die Regeln zur Verarbeitung intern in einfach zugreifbaren Datenstrukturen zu halten. Common Lisp bietet dafür mehrere Möglichkeiten Daten: Das Common-Lisp-Object-System (CLOS) mit Klassen, Structures und Grundelemente wie Listen und Arrays. Diese haben unterschiedliche Vor- und Nachteile.

CLOS-Klassen Vererbung, einfache Erstellung, automatische Zugriffsmakros, übersichtlich

Structures Einfache Erstellung, automatische Zugriffsmakros, semi-übersichtlich

Listen Erstell- und Zugriffsmakros müssen manuell programmiert werden

Bei der Verwendung von Klassen und Structs hat der Programmierer weniger Arbeit, da viele Operationen automatisiert sind und die Übersichtlichkeit größer ist. Gut definierte Makros zur Behandlung einer Datenstruktur in Listenform können den letzten Punkt allerdings ausgleichen. Auf den ersten Blick scheinen trotzdem mehr Gründe für die Verwendung des CLOS zu sprechen.
Wichtigstes Merkmal einer deduktiven Komponente ist jedoch die Geschwindigkeit. Es ist zu erwarten, dass diese bei Basisoperationen über Listen höher ist, da weniger Overhead erzeugt wird. Zur Überprüfung wurden einige Tests durchgeführt, deren Ergebnisse in Tabelle 5 präsentiert werden.

I : Erstellen der Datenstruktur (make−instance, make−struct, make−list)		
Klasse	Struct	List
724	1,443	458

II : Einfacher Schreibzugriff auf ein Element (slot−value, struct−slot, **nth**)		
Klasse	Struct	List
1,545	1,671	605

III : Erweiterter Schreibzugriff auf ein Element		
Klasse	Struct	List
1,587	1,709	639

IV : Einfacher Lesezugriff auf ein Element (slot−value, struct−slot, **nth**)		
Klasse	Struct	List
499	475	478

V : Erweiterter Lesezugriff auf ein Element der zweiten Ebene		
Klasse	Struct	List
543	502	519

Tabelle 5: Datenstrukturtest - Ergebnisse. Jeweils 100.000 Durchgänge. Zeit in Millisekunden.

Wie erwartet sind die Basisoperationen über den Listen in allen Bereichen schneller. Allerdings sind die Unterschiede vor allem im Bereich des Auslesens,

der häufigsten Operation, nicht signifikant. Die Unterschiede in der Erstellzeit können auf die automatische Zusatzarbeit zurückgeführt werden. Der einzige kritische Punkt, der Schreibzugriff auf die Elemente der Klassen und Structs, spricht für eine Verwendung von Listen. Allerdings muss hier bedacht werden, dass der Zugriff auf Listen nicht immer über **nth** erfolgen kann, sondern manchmal auch mehrere zusätzliche Operationen wie **member** benötigen werden.
Eine Darstellung über Klassen wird abschließend gewählt, da sie eine größere Übersicht bieten und bei einer Erweiterung von AGDC um neue Methoden die Arbeit signifikant erleichtern können. Des Weiteren ergibt sich der Vorteil, mit CLOS zielgenaue Methoden zur Verarbeitung der Klassen zu definieren. Diese Punkte überwiegen den geringen Geschwindigkeitsvorteil, den eine Verwendung von Listen als Struktur bringen würde.

B Repräsentation der Regeln

Ein wichtiger Punkt der Implementierung ist, wie die Regeln im System repräsentiert werden. Es darf keine Information über die Struktur und Einschränkungen verloren gehen, während gleichzeitig ein einfacher Zugriff auf alle Teile und eine einfache Verarbeitung möglich sein sollte. In diesem Abschnitt werden die verwendeten Regelrepräsentationen vorgestellt, die nach einer sorgfältigen Abwägung von Ausdrucksstärke gegen Simplizität entstanden sind.
Das System kennt drei verschiedene Repräsentationen von Regeln, die jeweils für ihren jeweiligen Anwendungsbereich optimiert wurden.

Benutzersicht beschreibt, wie der Benutzer Regeln abfasst, um sie in das System einzupflegen. Diese Darstellung muss möglichst einfach und kurz, aber gleichzeitig auch übersichtlich sein.

Interne Sicht beschreibt, in welcher Struktur das System Regeln im Speicher hält.

Datenbanksicht ist der RDF-Graph, in den eine Regel transformiert wird, um sie in einem AlegroGraph Triple Store ablegen zu können.

B.1 Benutzersicht

Diese Sicht orientiert sich an der in der Literatur verbreiteten Darstellung von Datalog-Regeln in Verbindung mit der Syntax von S-Expressions, um eine einfache Verarbeitung durch das auf Lisp aufbauende System zu ermöglichen.

Beispiel B.1: Die Datalogregel

$$\mathtt{r(X,Y)} :- \mathtt{p(X,Z)}, \mathtt{q(Z,Y)}, \mathtt{s(Y,Constant"},\mathtt{Constant}).$$

wird in der Form

```
(:-  (r  ?X  ?Y)  ((p  ?X  ?Z)
                   (q  ?Z  ?Y))
                   (s  ?Y  "Constant"  Constant))
```

dargestellt. ◇

Dabei wird die Lisp-typische Prefix- der Infixnotation vorgezogen. Variablen beginnen immer mit einem vorgestellten Fragezeichen (*?*), Konstanten können von Anführungszeichen () umgeben oder einfach ausgeschrieben werden, wobei sie im zweiteren Falle nicht mit einem Fragezeichen (*?*) beginnen und keine Klammern (*(* bzw. *)*) beinhalten dürfen[34]. Groß- und Kleinschreibung ist für beide Parameterarten erlaubt und wird von dem System unterschieden.
Negierte Literale werden durch ein Prefix *not* bezeichnet.

Beispiel B.2: Die Datalogregel

$$\mathtt{r(X,Y)} :- \mathtt{p(X,Z)}, \mathtt{q(Z,Y)}, \neg\mathtt{s(Y,Constant"},\mathtt{Constant}).$$

[34] Der *char*, der als Variablensuffix dient, und auch die Begrenzerzeichen der Konstanten können im System über einen globalen Parameter festgelegt werden. Wie dies genau funktioniert, kann in D.2 nachgelesen werden.

wird in der Form

```
(:- (r ?X ?Y) ((p ?X ?Z)
               (q ?Z ?Y))
               (not s ?Y "Constant" Constant))
```

dargestellt. ◇

Eingebaute Prädikate werden in der Infix-Notation dargestellt und ansonsten ebenso wie andere Unterziele einer Regel angegeben.

Beispiel B.3: Die Datalogregel

$\mathtt{r(X,Y)} :- \mathtt{p(X,Z)}, \mathtt{q(Z,Y)}, \mathtt{X} > \mathtt{Y}.$

wird in der Form

```
(:- (r ?X ?Y) ((p ?X ?Z)
               (q ?Z ?Y))
               (?X > ?Y))
```

dargestellt. ◇

Die Typbezeichner, die einer Variablen oder einer Konstanten zugewiesen werden können, bilden mit diesen eine feste Einheit und werden mit einem weiteren Klammerpaar umschlossen.

Beispiel B.4: Die Datalogregel

$\mathtt{r(X,Y)} :- \mathtt{p(X,Z)}, \mathtt{q(Z,Y)}, \mathtt{s(Y,"2.3",Constant)}.$

mit X <= :int, "2.3«= :double-float und Constant <= :literal-typed wird in der Form

```
(:- (r (?X :int) ?Y)
       ((p ?X ?Z)
        (q ?Z ?Y))
        (not s ?Y ("2.3" 12) (Constant 2)))
```

dargestellt. ◇

Um deutlich zu machen, dass Typbezeichner entweder als Typ-Symbole oder als Typ-Codes angegeben werden können, wurde einmal :int/20 mit Symbol und einmal :double−float/12 und :literal−typed/2 mit Codes angegeben.
Beim Hinzufügen einer neuen Regel in die IDB des Systems wird sie von der Benutzersicht in die Datenbanksicht transformiert und in der IDB-Datenbank abgelegt. Dies erfolgt über das Makro :−.

B.2 Interne Sicht

Die interne Sicht wird vom System verwendet, um Regeln zu bearbeiten und Operationen über ihnen durchzuführen. Dadurch ergeben sich die Anforderungen einer effizienten Darstellung mit schnellen Zugriffs- und Manipulationsmöglichkeiten bei einem gleichzeit geringen Speicherverbrauch.

Die gewählte Darstellung entspricht einem Konstrukt aus Klassen und Konstrukten, das Ähnlichkeiten mit dem RDF-Graphen der Datenbanksicht aufweist.

```
((class)rule
        :head ((class)literal
                :pred (upi)
                :parameters (list
                        (nil . type)
                        ...
                        (const-upi . nil)
                        ...
        :subgoals (list
                (class)literal
                        :pred (upi)
                        :parameters (list
                                (nil . type)
                                ...
                                (const-upi . nil)
                                ...
                        :negated (boolean)
                ...))
```

Listing 4: Struktur der internen Regelrepräsentation

Jede Regel wird durch die Klasse rule dargestellt, die einzelnen Literale des Körpers und der Kopf wiederum durch Instanzen der Klasse literal. Bei den Parametern der Literale wird unterschieden zwischen Konstanten, die durch die Struktur (const−upi . nil), und Variablen, die durch die Struktur (nil . nil) repräsentiert werden. Gleiche Variablen werden durch die gleiche `cons-cell` dargestellt. Die Reihenfolge des Vorkommens von Unterzielen und Parametern in den jeweiligen Listen ist relevant.
Für die Klassen rule und literal sind einige Methoden definiert, um eine einfache Manipulation und Extraktion der Daten zu ermöglichen. Anhang D gibt einen Überblick über die Methoden, die für den Anwender interessant sind.

B.3 Datenbanksicht

Die Darstellung der Regeln in einer Tripel-Form zur Speicherung in der Datenbank erfordert ein vorsichtiges Vorgehen. Es darf keine benötigte Metainformation verloren gehen, während es gleichzeitig wünschenswert ist, eine möglichst kleine Menge von Tripeln pro Regel zu erhalten.
Um eine gute Repräsentation zu erhalten können einige Besonderheiten von AlegroGraph ausgenutzt werden, die über den RDF-Standard hinausgehen. So bietet das Datenbankmanagementsystem die Möglichkeit, Tripel zu benannten Graphen zusammenzufassen. Außerdem werden First-Order-Tripel unterstützt, was bedeutet, dass ein Tripel über seine ID als Objekt oder Subjekt eines anderen Tripels fungieren kann.
Besonders kritische Punkte sind die fehlende Einzigartigkeit von Prädikatensymbolen innerhalb einer Regel und die feste Reihenfolge der Parameter. Da für den zweiten Punkt sowieso eine Lösung gefunden werden muss, wird mit

Hinblick auf zukünftige Optimierungen auch die, für Datalog an sich beliebige, Information über die Reihenfolge der Unterziele mitgespeichert. Somit können Unterzielanordnungen bereits vor der Anfrageauswertung durchgeführt werden. Die Abbildung 4 stellt den RDF-Graphen der Datalogregel

```
(:- (p (?x 20))
        ( (q ?x ?y) (not r ?y) (?y < ("5" 21)) ))
```

dar, Abbildung 5 ist eine Auflistung der entsprechenden RDF-Tripel.

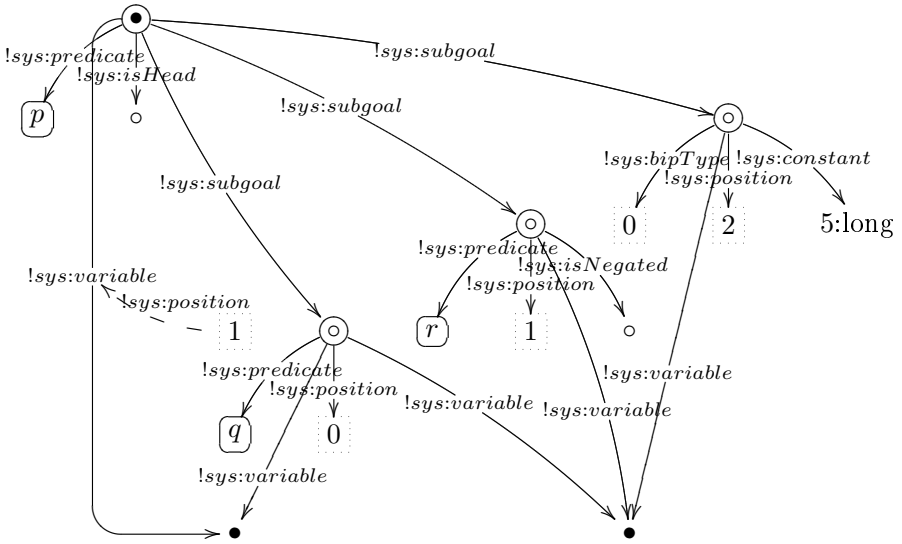

Abbildung 4: RDF-Graph einer Regel, ∘ und • stellen `blank-nodes` dar

Im Graph entsprechen die Beschriftungen der gerichteten Kanten den Prädikaten, die Ursprünge den Subjekten und die Ziele den Objekten. Die Literale bestehen aus einem `blank-node` mit einer einzigen *sys:predicate* Kante zum Prädikatensymbol und $0-*$ *sys:variablen* bzw. *sys:constanten* Kanten zu den Parametern des Literals. Das Kopfprädikat ist mit seinen Körperprädikaten durch eine *sys:subgoal* Kante verbunden. Die Abbildung enthält nur ein exemplarisches Beispiel der Positionstripel, der gestrichelte Pfeil (⇢), der von dem Integer *1* auf eine der *sys:variable* Kanten weist. Bei diesen handelt es sich um ein First-Order-Tripel, die eine Aussage über einen anderen Tripel treffen. Diese Positionstripel werden für alle Unterziele `blank-nodes` und Parameter (*sys:variable*, *sys:constant*) Beziehungen angelegt und beginnen ihre Zählung bei 0. Alle Tripel der Regel werden dem gleichen Graphen zugeordnet, wodurch die Regeln voneinander unterschieden werden können.

```
(:_0 sys:predicate r)
(:_0 sys:isHead :_1)
  (:_0 sys:variable :_2 regel id_00)
     (id_00 sys:position 0^^:short)
  (:_0 sys:subgoal :_3)
  (:_0 sys:subgoal :_4)
  (:_0 sys:subgoal :_5)
```

```
(:_3 sys:predicate q)
(:_3 sys:position 0^^:short)
  (:_3 sys:variable :_2 regel id_10)
     (id_10 sys:position 0^^:short)
  (:_3 sys:variable :_6 regel id_11)
     (id_11 sys:position 1^^:short)

(:_4 sys:predicate r)
(:_4 sys:isNegated :_7)
(:_4 sys:position 1^^:short)
  (:_4 sys:variable :_6 regel id_20)
     (id_20 sys:position 0^^:short)

(:_5 sys:bipType 0^^:short)
(:_5 sys:position 2^^:short)
  (:_5 sys:variable :_6 regel id_30)
     (id_30 sys:position 0^^:short)
  (:_5 sys:constant 5^^:long)
```

Listing 5: Beispiel für die RDF Tripel einer Regel

Die Einrückungen sind nur zur Übersicht angegeben, die Tripel sind alle gleichberechtigt. Wo erforderlich, sind noch `graph` und `id` der Tripel angegeben. Intern verwaltet AlegroGraph die Tripel also als Quintupel in der Form
(subject predicate object graph id). Sofern nicht anders angegeben, wird angenommen, dass jeder Tripel dem Graphen *regel* der Regel zugeordnet ist.
Die vorgestellte Regel mit einem Kopfprädikat, 3 Unterzielen und 5 Parametern erfordert alleine schon 23 RDF-Tripel für die Speicherung im IDB-Store.

C Schnittstelle für den Datenbankzugriff

AlegroGraph bietet verschiedene Möglichkeiten an, lesend auf die Tripel eines Stores zuzugreifen.

Lisp Lisp native Methoden.

Prolog Ein Interface zu Alegro Prolog über das Triple mit deklarativen Anfragen ausgelesen werden können.

SPARQL Die W3C-Standard-Anfragesprache, ebenfalls mit Unterstützung deklarativer Anfragen und zahlreicher zusätzlicher Optionen.

Die **Lisp nativen Methoden** erlauben ein einfaches und schnelles Einbetten von Datenbankanfragen in den Systemcode auf Kosten der Ausdruckskraft. Mit ihnen ist es nur möglich ein oder mehrere Tripel nach einem Kriterium auszulesen, während die Zusammenhänge zwischen ihnen programmatisch in Lisp-Code ausgedrückt werden müssen. Eine Frage nach allen Zwei-Generationen-Beziehungen einer Familie würde wie in 6 aussehen und noch weiteren Code erfordern, um die erhaltenen Tripel in Zusammenhang zu stellen.

```
(iterate-cursor (tripel (get-triples :p "isParentOf"))
      (get-triples :s (object triple) :p "isParentOf"))
```

Listing 6: Erfragen von Zwei-Generationen-Beziehungen mit Lisp nativen Methoden

Andererseits unterstützen diese Methoden alle AlegroGraph Erweiterungen wie Tripel Ids und First-Order-Tripel, die über die Möglichkeiten einer reinen RDF-Datenbank hinausgehen.

Bei einer Verwendung der **Prolog** Schnittstelle würde sich die Möglichkeit ergeben, deklarative Anfragen zu stellen. Das schon im vorhergehenden Absatz verwendete Beispiel würde in Prolog nur eine einzige Anfrage erfordern wie 7 zeigt, es lassen sich aber auch viel kompliziertere Verschachtelungen bilden.

```
(select0-distinct (?grandparent ?grandchild)
      (q- ?grandparent isParentOf ?-p)
      (q- ?-p isParentOf ?grandchild))
```

Listing 7: Erfragen von Zwei-Generationen-Beziehungen mit Prolog

Die AlegroGraph Erweiterungen des RDF-Prinzipes werden ebenfalls unterstützt. Ein Kritikpunkt ist die fehlende Möglichkeit, Anfrageergebnisse auf eine bestimmte Zahl zu begrenzen. Es werden also immer alle Beziehungen, die auf das Suchmuster zutreffen, zurückgegeben. Interessant wäre die Möglichkeit einen Teil der Beweisführung nach Alegro Prolog auszulagern. Das Entwickeln, Umsetzen und Testen dieses Ansatzes würde den Rahmen dieser Arbeit allerdings überschreiten.

Für die Verwendung von **SPARQL** spricht die Standardisierung und Portabilität. Es wäre möglich, mit nur sehr wenig Codeveränderung die Datenbank

hinter der deduktiven Komponente auszuwechseln. Des Weiteren bietet SPARQL eine Reihe von Möglichkeiten, die deklarativen Anfragen ausdrucksstark zu formulieren und die Ergebnismenge zu begrenzen[35]. Nachteil ist die fehlende Unterstützung für die AlegroGraph eigenen Erweiterungen. Dies führt auch dazu, dass bei der Nutzung von **SPARQL** als Datenbankschnittstelle die zugrundeliegenden EDB zwingend dem RDF-Standard genügen müssen. Dies ist von Nutzer sicherzustellen, der die EDB erstellt und mit Daten befüllt.

```
(run-sparql
        "SELECT ?grandparent ?grandchild WHERE {"
           "(?grandparent isFatherOf ?-p) ."
           "(?-p isFatherOf ?grandchild) ."
        "}" :results-format :lists)
```

Listing 8: Erfragen von Zwei-Generationen-Beziehungen mit Prolog

C.1 Schnittstellentest I

Wie im vorherigen Abschnitt gezeigt wurde, gibt es mehrere Gründe, die für und gegen jede der angebotenen Schnittstellen sprechen. Bei den Beschreibungen außer Acht gelassen wurde bisher die Geschwindigkeit, mit der die Abfragen behandelt werden. Dies ist jedoch wie in Abschnitt 1.1 der wichtigste Punkt bei der Entwicklung der deduktiven Komponente.
Der durchgeführte Test beschäftigt sich mit der Ausführgeschwindigkeit leichter bis semi-komplizierter Anfragen an die Datenbank durch die drei verschiedenen Schnittstellen. Die erzielten Geschwindigkeiten sind jeweils einmal für eine nichtindexierte und einmal für eine indexierte Datenbank angegeben. Die Ergebnisse sind in Tabelle 6 angegeben.
Die durchschnittlich höhere Geschwindigkeit bei Lisp-basierten Anfragen sollte mit Vorsicht betrachtet werden, da mehr Funktionalität nach Lisp ausgelagert werden muss. Die am häufigsten getätigten Anfragen werden voraussichtlich verkettete zum Auslesen der Regeln aus der IDB-Datenbank und Anfragen zum Erhalt mehrerer Tripel an die EDB-Datenbank sein. Relevant ist die Geschwindigkeitssteigerung bei einer verketteten Anfrage über eine indexierte Datenbank. Um festzustellen, ob dieser Geschwindigkeitszuwachs durch Verwendung der low-level Methoden das Mehr an Programmierarbeit rechfertigen kann, wird ein weiterer Test durchgeführt.

C.2 Schnittstellentest II

Prolog und SPARQL unterstützen beide deklarative Datenbankanfragen. Dieser Test soll die Frage beantworten, ob eine einzelne solche Anfrage schneller ist als mehrere einfache, bei denen der Zusammenhang zwischen den Elementen programmatisch hergestellt wird. Als Testoperation dient das Auslesen aller in der Datenbank gespeicherten Informationen zu einer Datalog-Regel. Im vorliegenden RDF-konformen Fall handelt es sich dabei um durchschnittlich 33 Tripel, die zueinander in Beziehung stehen. Die Ergebnisse sind in Tabelle 7 abgebildet.

[35] Leider wurde die Unterstützung für Subqueries vom W3C hinausgeschoben (Stand: 27.05.2009).

I : Auslesen eines einzelnen Tripels					
Lisp		Prolog		Sparql	
2,228	86*	3,381	1,249*	5,130	803*

II : Auslesen mehrerer Tripel					
Lisp		Prolog		Sparql	
2,216	100*	3,823	1,702*	4,981	698

III : Auslesen aller Tripel					
Lisp		Prolog		Sparql	
44,038	43,555*	47,290	49,532*	82,832	87,643*

IV : Semi-komplizierte Anfrage					
Lisp		Prolog		Sparql	
15,967	939*	17,783	2,478*	23,523	1,777*

V : Verkettete Anfrage					
Lisp		Prolog		Sparql	
47,113	46,807*	52,291	51,446*	130,171	134,083*

Tabelle 6: Schnittstellentest I - Ergebnisse. Jeweils 1000 Durchgänge. * = indexiert. Zeit in Millisekunden.

Der Gewinn durch eine Aufteilung der Anfrage und programmatische Herstellung des Zusammenhanges ist im Falle von Prolog, der einzigen Query-Engine, die beide Vorgehensweisen unterstützt, nicht signifikant. SPARQL ermöglicht keine Verwendung von BlankNodes in Anfragen außerhalb des deklarativen Kontextes, sodass hier nur die Geschwindigkeit einer zusammengefassten Anfrage getestet werden konnte. Erstaunlichstes Ergebnis ist die um den Faktor 10 (bei indexierten Tripeln sogar 20) höhere Geschwindigkeit bei der Verwendung der nativen Methoden im Vergleich zu Prolog. Dass diese den programmatischen Ansatz zwingend voraussetzt und so zu mehr Code führt, ist im Hinblick auf dieses Ergebnis vernachlässigbar.
Zusammen mit dem leichten Geschwindigkeitsvorteil, den die Lisp-Query-Engine auch im ersten Test gezeigt hat, ergibt sich die Wahl dieser Zugriffsmaschinerie für die deduktive Komponente.
Alle Funktionen und Methoden, die Anfragen an die Datenbank beinhalten werden in den Lisp Paketen `agdc.idb.query` und `agdc.edb.query` abgelegt, sodass die Erstellung einer Query Enginge für SPARQL nachträglich möglich sein sollte. Allerdings muss in diesem Fall auch der Parser umgeschrieben werden, da dann die Verwendung von First-Order-Tripeln nicht möglich ist. Als weiteres Problem tauchen nicht-RDF-gültige Tripel in der EDB auf, genauer Tripel ohne Namespace. Diese kann SPARQL nicht direkt abfragen und es wird ein Hack benötigt.

I : Aufgabe in zwei Anfragen komprimiert					
Lisp		Prolog		Sparql	
Nicht unterstützt		116,713	1,620*	184,080	5,309*

II : Aufgabe in mehrere Anfragen partitioniert				
Lisp		Prolog		Sparql
9,063	61*	92,024	1,217*	Nicht unterstützt

Tabelle 7: Schnittstellentest II - Ergebnisse. Jeweils 100 Durchgänge. * = indexiert. Zeit in Millisekunden.

D Methoden und Variablen des AGDC Benutzerinterface

:-

`head body`

Definition einer neuen Regel. Diese wird überprüft, geparst und dem IDB-Store aus ***idb-store*** hinzugefügt. Eine genaue Definition dieses Makros ist in den Abschnitten 3.5, 5.2 und 5.3 gegeben, in Anhang B finden sich einige Beispiele zur Anwendung.

?-

`predicate` *&rest* `parameter`

Dieses Makro dient der Formulierung und Beantwortung einer Anfrage. Die zu verwendende Syntax enspricht der von **:-** Eine genaue Definition ist in den Abschnitten 3.5 und 5.3 gegeben. Die Antwortliterale können in verschiedenen Formaten zurückgegeben werden, was durch **query-result-form** gesteuert wird.

D.1 Regeln & Fakten

get-rules

`nil`

Gibt eine `list` mit allen Regeln der IDB als Regelobjekte zurück.

get-undefined-facts

`nil`

Gibt die Fakten zurück, die nach der letzten Anwendung einer Auswertung unter Verwendung der wohlfundierten Semantik mit dem Wahrheitswert `undefinied` verblieben sind. Die Ausgabe ist nur dann verlässlich, wenn keine Optimierung durch GSMS verwendet wurde, also **query-mode** auf :semi.wfm gesetzt war.

pprint-fact

`fact` *&optional* `stream`

Diese Funktion gibt ein Literalobjekt in einer menschenlesbaren Form aus. Mehr Details unter **pprint-facts**.

pprint-facts

`facts` *&optional* `stream line-break`

Gibt die Literalobjekte der Liste `facts` in einer menschenlesbaren Form aus, die sich an der in der Literatur üblichen Notation von Datalogregeln orientiert. Konstanten werden als UPIs in der Form *{Wert}*, Variablen

als Zahlen der Form *_0* ausgegeben. Gleiche Zahlen innerhalb eines Literals bezeichnen gleiche Variablen. Beide können optional von einer Typinformation begleitet werden, die in der Form *_0^:typbezeichner* erscheint. `stream` wird standardmäßig zu `t`, dem Standard-Output-Stream, `line-break` zu `nil`.

pprint-rule
`rule` *&optional* `stream`

Diese Funktion gibt ein Ruleobjekt in einer menschenlesbaren Form aus. Mehr Details **pprint-rules** unter.

pprint-rules
`rules` *&optional* `stream line-break`

Gibt die Rulesobjekte der Liste `rules` in einer menschenlesbaren Form aus, die sich an der in der Literatur üblichen Notation von Datalogregeln orientiert. Die Ausgabe gleicht der von **pprint-facts**, nur dass die Variablen im Kontext der gesamten Regel gelten. `stream` wird standardmäßig zu `t`, dem Standard-Output-Stream, `line-break` zu `nil`.

literal->list
`literal`

Gibt das Literalobjekt als **list** der Form

```
(list boolean            ;; negated
      upi                ;; predicate
      (list upi ...))    ;; parameters
```

zurück. Diese Methode ist für instanziierte Literale gedacht, die keine Variablen mehr beinhalten. Sie arbeitet auch mit noch nicht instanziierten Literalen, die Variablen werden dann jedoch durch `nil`-Werte ersetzt.

literal->plain-list
`literal`

Gibt das Literalobjekt als **list** der Form

```
(list boolean            ;; negated
      string             ;; predicate
      (list string ...)) ;; parameters
```

zurück. Diese Methode ist **literal->list** sehr ähnlich, nur dass die UPIs in ihre Stringrepräsentationen übersetzt werden. Diese Methode ist für instanziierte Literale gedacht, die keine Variablen mehr beinhalten. Sie arbeitet auch mit noch nicht instanziierten Literalen, die Variablen werden dann jedoch durch `nil`-Werte ersetzt.

head
`rule`

Gibt den Kopf eines Regelobjektes als Literalobjekt zurück.

subgoals
`rule`

Gibt die Unterziele eines Regelobjektes als **list** von Literalobjekten zurück.

pred
`literal`

Gibt das Prädikat eines Literalobjektes als UPI zurück.

parameters
`literal`

Gibt die Parameter (Variablen und Konstanten) eines Literalobjektes als **list** von **cons** zurück. Diese enthält (nil . <type−code>) für Variablen und (<const−upi> . nil) für Konstanten, wie in B.2 genauer dargelegt.

negated
`literal`

Gibt `t` zurück, wenn das dem Literalobjekt zugrundeliegende Atom negiert ist.

D.2 Einstellungen

query-result-form
&optional `form`

Legt das Rückgabeformat der Ergebnisse einer mit **?-** gestellten Anfrage fest. Wenn keine `form` angegeben wird, gibt die Funktion das aktuell eingestellte Rückgabeformat zurück. Erlaubte Werte für `form` sind : literal , : **list** und : plainlist . Im ersteren Fall werden die Ergebnisliterale als Literalobjekte zurückgegeben, die beiden anderen Fälle entsprechen einer vorherigen Umwandlung mit **literal->list** bzw. **literal->plain-list**.

query-mode
&optional `mode`

Zeigt den aktuellen Modus an oder legt fest, mit welcher Strategie eine Anfrage beantwortet werden soll. Erlaubte Werte für `mode` sind :semi.non−neg, :semi.wfm, :magic.non−neg und :magic.wfm. Ihnen entspricht, in dieser Reihenfolge, eine semi-naive Auswertung ohne negierte Literale;

eine semi-naive Auswertung mit negierten Literalen nach der wohlfundierten Semantik; eine semi-naive Auswertung mit Optimierung durch GSMS ohne negierte Literale bzw. eine semi-naive Auswertung mit Optimierung durch GSMS mit negierten Literalen nach der wohlfundierten Semantik.

set-idb-store
`store`

Setzt den zu verwendenden IDB-Store. Dieser muss vom Typ AlegroGraph-Tripel-Store sein und über Schreibzugriff verfügen. Der Pointer zum Store wird in ***idb-store*** abgelegt. Die Festlegung des Stores interferiert nicht mit den globalen AlegroGraph Einstellungen (die AlegroGraph Variable ∗db∗ wird also nicht beeinflusst).

set-edb-store
`store`

Setzt den zu verwendenden EDB-Store. Dieser kann von beliebigem Typ sein, so u.a. auch ein Federated Store. Der Pointer zum Store wird in ***edb-store*** abgelegt. Die Festlegung des Stores interferiert nicht mit den globalen AlegroGraph Einstellungen (die AlegroGraph Variable ∗db∗ wird also nicht beeinflusst).

debug-modus
`t/nil`

Schaltet den Debug-Modus an oder aus. Als Rückgabewert treten `t` für eingeschalteten und `nil` für abgeschalteten Debug-Modus auf. Im Debug-Modus werden stetig Informationen über den Inferenzmechanismus ausgegeben, die eine Auswertung bedeutend verlangsamen und nur für einen Entwickler wirklich interessant sind.

idb-store

Variable, die den Pointer auf den IDB-Store hält, der gerade genutzt wird.

edb-store

Variable, die den Pointer auf den EDB-Store hält, der gerade genutzt wird.

system-namespace

Variable, die den von AGDC intern genutzten Namespace enthält, der mit dem !-reader-Shortcut `!sys:` erreichbar ist. Dieser interferiert nicht mit den sonstigen Namespaces der IDB und EDB, kann und sollte also

in seinem ursprünglichen Wert belassen werden. Eine Änderung kann zur Inkonsistenz mit zu einem früheren Zeitpunkt erstellten Regeln und IDB-Stores führen.

constant-delimiter

Enthält den Charakter, der genutzt wird, um Konstanten bei einer Regeldefinition einzuschließen. Dieser ist standardmäßig ". Er kann ohne große Bedenken geändert werden, allerdings sollte die darauf folgende Verhaltensänderung von **:-** und **?-** verstanden werden.

variable-prefix

Enthält den Charakter, der als Prefix für Variablen genutzt wird und über den diese identifiziert werden. Dieser ist standardmäßig *?*. Er kann ohne große Bedenken geändert werden, allerdings sollte die darauf folgende Verhaltensänderung von **:-** und **?-** verstanden werden.

negation-string

Enthält den String, der als Prefix vor einem Prädikatensymbol auf dessen Verneinung hinweist. Dieser ist standardmäßig *not*. Er kann ohne große Bedenken geändert werden, allerdings sollte die darauf folgende Verhaltensänderung von **:-** und **?-** verstanden werden. Im Gegensatz zu ***constant-delimiter*** und ***variable-prefix*** kann hier eine Änderung des Strings durchaus Sinn machen, da es bei Verwendungen von notäls Prädikatensymbol zu Uneindeutigkeiten beim Regelparsen kommen kann.

D.3 Sonstiges

with-idb-store
&body `body`

Dieses Makro ist ein Wrapper um die AlegroGraph Funktion with−triple−store und bindet die AlegroGraph Variable ∗db∗ an den IDB-Store aus ***idb-store***. Dieses Makro dient vor allem der korrekten Ausgabe von UPI-Werten als Strings, die den Store benötigen, aus dem sie entstanden sind.

with-edb-store
&body `body`

Dieses Makro ist ein Wrapper um die AlegroGraph Funktion with−triple−store und bindet

idb-indexed-p
`nil`

Gibt `t` zurück, wenn der IDB-Store indexiert ist.

edb-indexed-p
`nil`

Gibt `t` zurück, wenn der EDB-Store indexiert ist.

E Liste der AlegroGraph-Typbezeichner

Typ-Code	Typ-Symbol
30	:triple-id
31	:default-graph
41	:subscript
36	:geospatial
29	:longitude
28	:latitude
27	:telephone-number
8	:blank-node
3	:literal-language
2	:literal-typed
1	:literal
7	:literal-short
0	:node
0	:resource
11	:single-float
12	:double-float
26	:gyear
24	:time
25	:date-time
23	:date
42	:long-88
21	:long
19	:short
20	:int
18	:byte
43	:unsigned-long-88
17	:unsigned-long
15	:unsigned-short
16	:unsigned-int
14	:unsigned-byte

Tabelle 8: Unterstützte Typbezeichner in AlegroGraph 3.2

F Abkürzungsverzeichnis

AGDC	Alegro Graph Deductive Component
BIP	Build-in Predicate(s)
CLOS	Common Lisp Object System
DB	Datenbank
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem
DLP	Description Logic Program
EDB	Extensional Database
EPP	Elementary Production Principle
F	Failure (Transformationsmethode)
GSMS	Generalized Supplementary Magic Sets
IDB	Intensional Database
L	Loop detection (Transformationsmethode)
lfp	Least Fixed Point
LUBM	Lehigh University Benchmark
M	Magic reduction (Transformationsmethode)
N	Negative reduction (Transformationsmethode)
OWL	Web Ontology Language
P	Positive reduction (Transformationsmethode)
RDBMS	Relationales Datenbankmanagementsystem
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
S	Success (Transformationsmethode)
SCC	Strongly Connected Component
sips	Sideway Information Passing Strategy
SPARQL	SPARQL Protocol and RDF Query Language
UPI	Unique Part Identifier
W3C	The World Wide Web Consortium
XML	Extensible Markup Language
XMLS	Extensible Markup Language Schema

Tabelle 9: Abkürzungsliste

G Index der Tabellen und Abbildungen

Tabellenverzeichnis

Abbildungsverzeichnis