INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Wagmüllerstraße 23 D–80538 München

Ludwig
Maximilians–
Universität
München

Implementierung von Anfrageauswertungsverfahren für Datalog-Programme in Prolog

Gerhard Hingerl und Jörg Sander

Fortgeschrittenenpraktikum

Beginn der Arbeit:	1. Oktober 1994
Abgabe der Arbeit:	8. Mai 1995
Betreuer:	Dipl.-Inform. Slim Abdennadher

Inhaltsverzeichnis

1 Einleitung

In diesem Fortgeschrittenen-Praktikum sind einige Anfrageauswertungsverfahren für Datalog-Programme in Prolog implementiert worden. Ein Datalog-Programm (im folgenden auch "Deduktive Datenbank" (DDB) genannt) ist ein einfaches durch eine endliche Menge von Fakten und Regeln gegebenes Logikprogramm, das folgende Bedingungen erfüllt:

- Fakten sind Atome der Form $H \leftarrow$.
- Regeln sind Ausdrücke der Form $H \leftarrow L_1 \wedge \ldots \wedge L_n, (n \geq 1)$[1], wobei H ein Atom und die L_i *positive* Literale sind.
- Die Sprache enthält außer Konstantensymbolen keine weiteren Funktionssymbole.

Prädikate, die ausschließlich durch Fakten gegeben sind, nennen wir "Extensionale Datenbank-Prädikate" (EDB). Solche Prädikate, die durch mindestens eine Regel definiert sind, nennen wir "Intensionale Datenbank-Prädikate" (IDB).
Eine *Anfrage* A an eine Deduktive Datenbank P ist ein Ausdruck der Form $\leftarrow L_1 \wedge \ldots \wedge L_n, (n \geq 1)$, wobei die $L_i's$ positive Literale sind.
Antwortsubstitutionen auf eine solche Anfrage sind Substitutionen Θ für alle Variablen in A, so daß gilt: $P \models A\Theta$. $A\Theta$ heißt dann *Antwort* auf A.
Ein Anfrageauswertungsverfahren für eine Deduktive Datenbank ist ein Verfahren, welches zu einem (beliebigen) gegebenen Datalog-Programm und einer (beliebigen) Anfrage an dieses Datalog-Programm Antworten findet. Idealerweise ist ein solches Verfahren korrekt, vollständig und terminiert immer.

Wir behandeln drei Strategien zur Anfrageauswertung in Deduktiven Datenbanken:

- Vorwärtsschließende Verfahren
  - Naive Vorwärtsauswertung
  - Seminaive Vorwärtsauswertung
- Transformation der DDB zur Optimierung der Vorwärtsauswertung
  - Magic Sets (und Unique Binding Property)
- Rückwärtsschließende Verfahren

[1] In unserer Implementierung setzen wir Prolog-Syntax für Regeln und Fakten von Datalog-Programmen voraus, d.h. alle Klauseln werden mit einem Punkt abgeschlossen, für den Implikationspfeil "$\leftarrow$" wird "`:-`" und für die Konjunktion "$\wedge$" wird ein Komma geschrieben.

– OLDT
– RQSQ

Bei den Bottom-Up oder Vorwärtsschließenden Verfahren wird zunächst der kleinste Fixpunkt eines Immediate-Consequence-Operators berechnet. Ein Immediate-Consequence-Operator bildet eine Menge von Fakten M anhand einer Menge von Regeln ab in eine neue Menge von Fakten M', welche neben den Fakten M noch alle Fakten enthält, die mit Hilfe der Regeln und den Fakten aus M abgeleitet werden können. Der Fixpunkt wird durch iterative Anwendung des Immediate-Consequence-Operators berechnet. Die Berechnung ist abgeschlossen, wenn in einem Iterationsschritt $M = M'$ gilt. Das heißt, es werden alle Fakten gefunden, die aus den gegebenen Fakten mit Hilfe der Regeln des Programms ableitbar sind. Ein solches Vorgehen heißt "Bottom Up" oder "vorwärtsauswertend", weil in diesem Fall die Datenbank-Regeln "in der Richtung des Implikationspfeils", bei unserer Notation also von rechts nach links, angewendet werden. Dies entspricht einem Durchlaufen des Suchraums (anschaulich der Graph aus Regeln und Fakten) "von unten nach oben". Die Anfrage wird dann erst in einem zweiten Schritt dazu verwendet, um aus der Menge aller generierten Fakten diejenigen zu selektieren, die Antworten auf die Anfrage darstellen. Die Verfahren *Naive* und *Seminaive Evaluation* werden in Abschnitt 3 beschrieben.
Bei Top-Down oder rückwärtsschließenden Verfahren wird die Anfrage an die Deduktive Datenbank schon am Anfang der Auswertung dazu verwendet, die Anzahl der möglichen Beweise und der zu generierenden Fakten einzuschränken: es werden nur die relevanten Regeln —das sind solche, deren Kopf mit der Anfrage unifiziert— betrachtet und in der Anfrage eventuell vorhandene, einschränkende Konstanten werden über den Kopf der Regel als Variablenbindungen auf die Rumpfliterale propagiert. Die Herleitung oder der Beweis einer Antwort auf die Anfrage geschieht dann durch den Beweis des Rumpfs einer solchen Regelinstanz. Dazu werden die Instanzen der Rumpfliterale als "Unteranfragen" auf die gleiche Weise wie die "Ausgangsanfrage" behandelt. Dieser Prozeß endet bei den EDB-Prädikaten. Anschaulich wird damit der Suchraum "von oben nach unten" durchlaufen. Die Verfahren *OLD-Resolution with Tabulation* (OLDT) und *Recursive Query-Sybquery* (RQSQ) werden In Abschnitt 5 genauer dargestellt.
Die rückwärtsschließenden Verfahren haben also den Vorteil, daß der Suchraum für mögliche Antworten von vornherein eingeschränkt und damit die Anzahl der generierten Fakten und Beweise im allgemeinen deutlich geringer ist als bei den vorwärtsschließenden Verfahren ist. Vorwärtsschließende Auswerter haben dagegen eine einfachere Realisierung.[2]
Der dritte von uns behandelte Ansatz zur Anfrageauswertung verbindet nun in gewissem Sinn die Vorteile der Bottom-Up und Top-Down Auswertung. Bei

[2]Genauer gesagt ist die Sicherstellung der *Vollständigkeit* von rückwärtsauswertenden Verfahren mit komplexeren Algorithmen verbunden als bei den Vorwärtauswertern.

diesem Ansatz wird einer Anfragebeantwortung mit Hilfe eines gewöhnlichen Bottom-Up Auswerters eine *Transformation* der Deduktiven Datenbank vorangestellt. Durch die Transformation wird aus der Deduktiven Datenbank und einer gegebenen Anfrage ein neues Datalog-Programm generiert, das bezüglich dieser speziellen Anfrage dieselbe Antwortmenge liefert wie das Ursprungsprogramm, jedoch bei der Vorwärtsauswertung die den Suchraum einschränkenden Merkmale der Top-Down-Auswertung simuliert. Die hier implementierte Transformation ist die *Magic Set Methode*, die im Anschluß an die Vorwärtsauswerter in Abschnitt 4 ausführlicher dokumentiert ist.[3]

2 Allgemeines zur Implementierung der Verfahren

Die Implementierung der Verfahren verwendet die folgenden Datenstrukturen:

```
fact(Fact,Step,Id,History)
```

für Fakten der eingelesenen DDB und abgeleitete Fakten bei den vorwärtsschließenden Verfahren. Die Argumente sind:

- `Fact`, der eingelesene bzw. abgeleitete Fakt.
- `Step`, der Iterationsschritt, in dem `Fact` abgeleitet wurde.
- `Id`, die (eindeutige) Nummer von `Fact`.
- `History`, eine Liste von Ids. Das erste Element ist die Id der Regel, die restlichen Elemente sind die Ids der Fakten, mit der `Fact` abgeleitet wurde. Bei Fakten der eingelesenen Datenbank ist diese Liste leer.

```
ddb_rule((Head :- Body),Id)
```

für Regeln der eingelesenen DDB. Die Argumente sind:

- `Head`, der Kopf der eingelesenen Regel.
- `Body`, der Rumpf[4] der eingelesenen Regel.
- `Id`, die (eindeutige) Nummer der Regel.

Für transformierte Regeln werden analog die folgenden Prädikate verwendet:

[3] Es gibt auch Verfahren, bei denen die DDB zur Optimierung eines rückwärtsauswertenden Verfahrens transformiert wird. Solche Verfahren gehörten jedoch nicht zur Aufgabenstellung dieses Fopras.

[4] Da die Rümpfe der Regeln oftmals analog wie Listen verarbeitet werden, erhält jeder Regelrumpf beim Einlesen ein eindeutiges letztes Element. Wir haben hierfür das Prolog-Atom *true* gewählt, da es sich zu Konjunktionen ähnlich verhält wie die leere Liste zu allgemeinen Listen.

- `rule((Head :- Body),Id)`, für Regeln, die bei der Vorwärts- oder Rückwärtsauswertung ausgewertet werden.
- `ubp_rule((Head :- Body),Id)`, für Regeln, die die "unique binding property" erfüllen (adorned rules).
- `magic_rule((Head :- Body),Id)`, für Regeln, die aus der Magic-Set-Transformation entstanden sind.

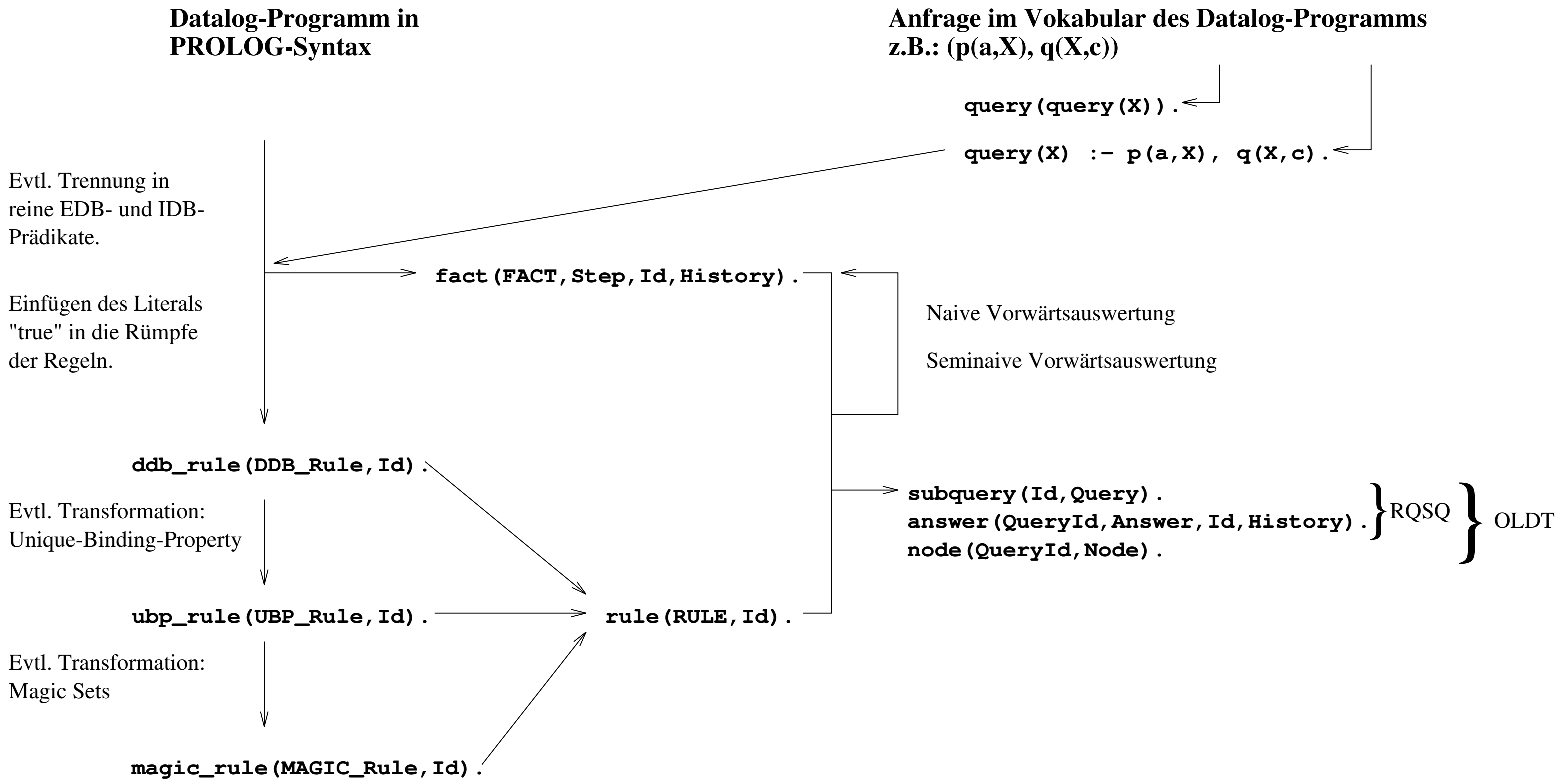
Eine Anfrage wird in eine Regel mit dem Kopf `query`$(X_1, \ldots, X_n)$ transformiert, wobei $X_1, \ldots, X_n$ die freien Variablen der Anfrage sind. Die Anfrage selbst bildet den Rumpf dieser Regel. Um auf diese Regel, die je nach Anfrage unterschiedlich viele Argumente im Kopf haben kann, zugreifen zu können, wird der Kopf dieser Regel in dem Prädikat `query_aux(Query)` abgespeichert.

Für die rückwärtsauswertenden Verfahren werden noch folgende Prädikate zur Realisierung einer Memotabelle verwendet:

- `subquery(Id,Query)`, für generierte Unteranfragen.
- `answer(QueryId,Answer,Id,History)`, für Antworten auf die (Unter)Anfrage mit der Nummer `QueryId`.
- `node(QueryId,Node)`, für die Memoisierungsstrategie bei OLDT.

Das folgende Schema soll einen Überblick über den Zusammenhang dieser Formate geben.

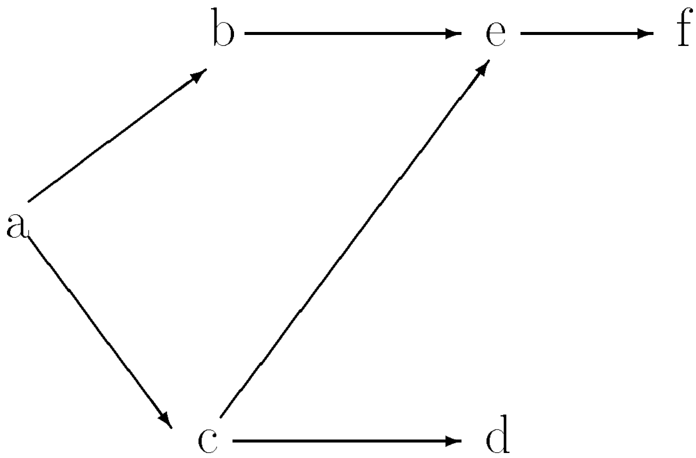
Interne Formate für Fakten, Regeln und Queries



Um diese Formate an einem Beispiel zu verdeutlichen, seien folgende Fakten gegeben:

```
k(a,b).
k(b,e).
k(e,f).
k(a,c).
k(c,d).
k(c,e).
```

Die graphische Darstellung dieser Fakten sieht wie folgt aus:



Des weiteren betrachten wir zwei Regelmengen zu diesen Fakten:

```
Regelmenge 1:

v(Start,Ziel) :-
k(Start,Ziel).
v(Start,Ziel) :-
k(Start,Knoten),
v(Knoten,Ziel).
```

```
Regelmenge 2:

v(Start,Ziel) :-
v(Knoten,Ziel),
        k(Start,Knoten).
v(Start,Ziel) :-
k(Start,Ziel).
```

und folgende Anfrage:

```
v(b,X)
```

Beim Einlesen dieser Datenbanken werden folgende Prädikate in die Prolog-Datenbasis geschrieben:

```
query_aux(query(A), 0, 0).

fact(k(a, b), 0, 1, []).
fact(k(b, e), 0, 2, []).
fact(k(e, f), 0, 3, []).
fact(k(a, c), 0, 4, []).
fact(k(c, d), 0, 5, []).
fact(k(c, e), 0, 6, []).
```

DDB-Regeln zu Regelmenge 1:

```
ddb_rule((query(A) :- v(b, A), true), 0).
ddb_rule((v(A, B) :- k(A, B), true), 1).
ddb_rule((v(A, B) :- k(A, C), v(C, B), true), 2).
```

DDB-Regeln zu Regelmenge 2 (PROLOG würde hier nicht terminieren):

```
ddb_rule((query(A) :- v(b, A), true), 0).
ddb_rule((v(A, B) :- v(C, B), k(A, C), true), 1).
ddb_rule((v(A, B) :- k(A, B), true), 2).
```

Die "Unique-binding-property-Transformation"erzeugt aus den DDB-Regeln folgende UBP-Regeln:
UBP-Regeln zu Regelmenge 1:

```
ubp_rule((query_f(A) :- v_bf(b, A), true), 1).
ubp_rule((v_bf(A, B) :- k(A, B), true), 2).
ubp_rule((v_bf(A, B) :- k(A, C), v_bf(C, B), true), 3).
```

UBP-Regeln zu Regelmenge 2:

```
ubp_rule((query_f(A) :- v_bf(b, A), true), 1).
ubp_rule((v_bf(A, B) :- v_ff(C, B), k(A, C), true), 2).
ubp_rule((v_ff(A, B) :- v_ff(C, B), k(A, C), true), 3).
ubp_rule((v_ff(A, B) :- k(A, B), true), 4).
ubp_rule((v_bf(A, B) :- k(A, B), true), 5).
```

Die Magic-Transformation erzeugt aus den UBP-Regeln folgende Magic-Regeln:
Magic-Regeln zu Regelmenge 1:

```
magic_rule((query_f(A) :- magic_query_f, v_bf(b, A), true), 1).
magic_rule((magic_v_bf(b) :- magic_query_f, true), 2).
magic_rule((v_bf(A, B) :- magic_v_bf(A), k(A, B), true), 3).
magic_rule((v_bf(A, B) :- magic_v_bf(A), k(A, C), v_bf(C, B), true), 4).
magic_rule((magic_v_bf(A) :- magic_v_bf(B), k(B, A), true), 5).
```

Magic-Regeln zu Regelmenge 2:

```
magic_rule((query_f(A) :- magic_query_f, v_bf(b, A), true), 1).
magic_rule((magic_v_bf(b) :- magic_query_f, true), 2).
magic_rule((v_bf(A, B) :- magic_v_bf(A), v_ff(C, B), k(A, C), true), 3).
magic_rule((magic_v_ff :- magic_v_bf(A), true), 4).
magic_rule((v_ff(A, B) :- magic_v_ff, v_ff(C, B), k(A, C), true), 5).
magic_rule((magic_v_ff :- magic_v_ff, true), 6).
magic_rule((v_ff(A, B) :- magic_v_ff, k(A, B), true), 7).
magic_rule((v_bf(A, B) :- magic_v_bf(A), k(A, B), true), 8).
```

3 Vorwärtsauswerter

Vorwärtsauswertung ist von den hier behandelten Auswertungsverfahren das einfachste: Zu einem Datalog-Programm und einer Anfrage wird zuerst der Fixpunkt zu diesem Programm berechnet und anschliessend werden aus dieser Faktenmenge die Antworten auf die Anfrage herausgesucht. Der Fixpunkt wird schrittweise berechnet mittels eines sogenannten *Immediate-Consequence-Operators*[5].
Die Implementierung der Fixpunktberechnung sieht folgendermaßen aus:

```
calculate_fixpoint(EvaluationType) :-
        new_id(step,Step),
        doall(immediate_consequence(Step,EvaluationType)),
        fact(_,Step,_,_),
        !,
        calculate_fixpoint(EvaluationType).
calculate_fixpoint(_).

immediate_consequence(Step,EvaluationType) :-
        rule((Fact :- Body),RuleId),
        evaluate(Body,Step,EvaluationType,BodyIds),
        update_database(Fact,Step,[RuleId|BodyIds]).
```

3.1 Naive Vorwärtsauswertung

Bei der *naiven* Vorwärtsauswertung besteht ein Schritt darin, zu einer gegebenen Menge von Fakten und Regeln *jede mögliche Ableitung* zu bilden und die neuen Fakten der Datenbasis hinzuzufügen. Das Verfahren wird naiv genannt, weil in jedem Schritt der vorhergehende komplett wiederholt und daher unter Umständen ein erhebliches Maß an redundanter Arbeit geleistet wird.

```
evaluate(true,_,naive,[]).
evaluate((B1,B2),Step,naive,[Id|Ids]) :-
        fact(B1,FactStep,Id,_),
        FactStep < Step,
         evaluate(B2,Step,naive,Ids).
```

3.2 Seminaive Vorwärtsauswertung

Bei der *seminaiven* Vorwärtsauswertung werden in einem Schritt nur solche Fakten abgeleitet, zu deren Ableitung mindestens ein in der vorhergehenden

[5] vgl. Einleitung.

Runde abgeleiteter Fakt verwendet wird. Das heißt, es werden nur *neue Ableitungen* gebildet.
In unserer Implementation der seminaiven Auswertung eines Regelrumpfs wird nacheinander ein Literal des Rumpfs ausgewählt und sichergestellt, daß dieses Literal mit Fakten des letzten Iterationsschrittes bewiesen wird. Die Literale des Rumpfs, die vor dem ausgewählten Literal stehen, werden mit Fakten bewiesen, die bereits *vor* dem letzten Iterationsschritt generiert wurden; die nach dem ausgewählten Literal stehenden Literale dürfen mit allen Fakten, außer den in der aktuellen Iteration generierten, bewiesen werden.

```
evaluate(Body,Step,seminaive,BodyIds) :-
        append_c(Prefix,(Middle,Postfix),Body),
        delta_fact(Middle,Step,Id),
        old_facts(Prefix,Step,Ids1),
        any_facts(Postfix,Step,Ids2),
        append(Ids1,[Id|Ids2],BodyIds).
```

Durch diese Implementierung wird sogar sichergestellt, daß jeder Beweis nur genau einmal geführt wird. Würde man nur vom ausgewählten Literal fordern, daß es mit Fakten der letzten Iteration bewiesen wird, würden unter Umständen identische Beweise mehrfach geführt werden.[6]

4 Transformationen für Vorwärtsauswerter

4.1 Die Unique-Binding-Property (UBP)

Voraussetzung für die nachfolgende Magic Set Transformation ist, daß kein IDB-Prädikat bei einer Top-Down-Auswertung einer gegebenen Anfrage mit unterschiedlichen Bindungsmustern aufgerufen wird. Diese Eigenschaft wird durch die UBP-Transformation hergestellt. Wäre beispielsweise für ein zweistelliges Prädikat *pred* bei einem Aufruf das erste Argument gebunden und das zweite frei, ergäbe sich eine neue Prädikatdefinition *pred_bf*, wobei *b* für gebunden und *f* für frei steht. Diese Definition unterscheidet sich von der alten lediglich darin, daß sämtliche Prädikate in Kopf und Rumpf /der Klausel mit Bindungsmustern markiert sind. Solche Bindungsmuster nennt man auch *Adornments*.
Der Algorithmus zur Herstellung der "unique binding property" simuliert "abstrakt" die Top-Down-Auswertung einer gegebenen Anfrage: Es wird nacheinander jede Regel, bei der der Funktor des Kopfprädikats mit dem Funktor der Anfrage übereinstimmt, rekursiv bearbeitet. Zuerst werden die

[6] Betrachte zum Beispiel folgende Regel: `p(X) :- q(X),r(X)`. Angenommen im letzten Iterationsschritt sind die Fakten q(a) und r(a) abgeleitet worden. Dann sind zwei identische neue Beweise für `p(a)` möglich.

Bindungen bzw. Bindungsmuster der Anfrage auf den Kopf der Regel und damit zugleich auf den Rumpf übertragen.[7] Dann werden die Bindungsmuster für die Rumpfprädikate gemäß der Selektionsstrategie[8] ermittelt. Zuerst wird das Bindungsmuster des ersten ausgewählten Rumpfprädikats bestimmt. Danach werden noch freie Variablen in diesem Prädikat mit einer Konstante gebunden, um das Bindungsmuster der nachfolgenden Prädikate zu vervollständigen. Auf die gleiche Weise verfährt man dann in der durch die Selektionsstrategie festgelegten Reihenfolge, bis der Rumpf abgearbeitet ist. Die so teilweise instantiierten Rumpfprädikate werden als neue Anfragen rekursiv wie die Ausgangsanfrage behandelt.
Das zentrale Prädikat der Implementation ist `abstrakt_evaluation`:

```
abstract_evaluation([]).
abstract_evaluation([[Predicate,Functor_ado]|Rest]) :-
        get_rule(Predicate,(Head :- Body),Id),
        propagate_bindings(Predicate,Head),
        find_body_adornments(Body,BodyPredList,Functor_adoList),
        assert_ubp_rule([Functor_ado|Functor_adoList],Id),
        append(BodyPredList,Rest,NewGoals),
        abstract_evaluation(NewGoals).
```

4.2 Magic-Set-Transformation

Die Magic-Set-Transformation wird an einer Menge von Regeln, welche die "unique binding property" erfüllen, durchgeführt. Diese Eigenschaft wird in unserer Implementation vor Aufruf der Magic-Set-Methode automatisch durch die im vorigen Abschnitt besprochene UBP-Transformation hergestellt.
Ergebnis der Transformation ist eine neue Menge von Regeln, welche bei Vorwärtsausführung nur zur Beantwortung der Query relevante Fakten generiert. Die Transformation modelliert die Konstantenpropagierung von Top-Down-Methoden durch die Einfügung von "magic subgoals" in die Ausgangsregeln und die Generierung weiterer Regeln für die Literale in den Regelrümpfen der Ausgangsregeln:

- Zu jedem IDB-Prädikat p wird ein neues Prädikat $magic\_p$ eingeführt. Die Argumente von $magic\_p$ sind die beim Aufruf gebundenen Variablen von p.

- Ersetzen aller Regeln $H \leftarrow B_1, \ldots, B_n$ durch Regeln $H \leftarrow magic\_H, B_1, \ldots, B_n$.

[7] Da im Verlauf der abstrakten Auswertung eine willkürlich gewählte Konstante benutzt wird um Variablen zu binden, andererseits in den Regeln selbst schon Konstanten vorkommen können, reicht einfaches Matching zur Propagierung der Variablenbindungen nicht aus.

[8] in unserem Fall die Prolog-Standardselektionsfunktion "von links nach rechts"

- Generierung weiterer Regeln für die IDB-Literale in den Regelrümpfen $(B_1, \ldots, B_n)$: $magic\_B_i \leftarrow magic\_H, B_1, \ldots, B_{i-1}$ für jedes Bodyliteral B_i.

- Generierung eines Magic-Fakts aus der Query.

Die Implementierung der Magic-Set-Transformation ist verhältnismäßig einfach, da sie sich auf UBP-Regeln stützt, in denen die Bindungsmuster der Variablen schon gekennzeichnet sind. Der Code ist eine direkte Übersetzung der obigen Punkte in Prolog-Notation.

```
magic :-
        ubp_rule((Head :- Body),_),
        magic_predicate(Head,MagicHead),
        new_id(magic_rule,Id1),
        assert(magic_rule(Head :-(MagicHead,Body),Id1)),
        append_c(Prefix,(Pred,_),Body),
        magic_predicate(Pred,MagicPred),
        new_id(magic_rule,Id2),
        assert(magic_rule(MagicPred :-(MagicHead,Prefix),Id2)),
        fail.
magic :-
        adorned_predicate(query,_,QueryF),
        concat_atom(['magic_',QueryF],MagicQuery),
        asserta(fact(MagicQuery,0,0,[])).
```

Die folgende Abbildung zeigt die Konstantenpropagierung der Bottom-Up-Auswertung an einem Beispiel.

Vorwärtsauswertung der zur Anfrage v(b,X) generierten Magic-Regeln (Regelmenge 1)

Step 1

`magic_v_bf(b)  <- magic_query_f`

Step 2

`v_bf(b,e) <- magic_v_bf(b), kante(b,e)`

`magic_v_bf(e)  <- magic_v_bf(b), kante(b,e)`

Step 3

`query_f(e)  <- magic_query_f, v_bf(b,e)`

`v_bf(e,f) <- magic_v_bf(e), kante(e,f)`

`magic_v_bf(f)  <- magic_v_bf(e), kante(e,f)`

Step 4

`v_bf(b,f) <- magic_v_bf(b), kante(b,e), v_bf(e,f)`

Step 5

`query_f(f)  <- magic_query_f, v_bf(b,f)`

5 Rückwärtsauswerter

Das Prinzip der hier behandelten rückwärtsauswertenden Verfahren ist Resolution mit *Wiederholungskontrolle*. Reine Resolutionsverfahren haben bei mehrfachen Vorkommnissen des gleichen Teilbaums im Resolutionsbaum folgende Nachteile:

- Die Knoten werden mehrfach expandiert.
- Es kann vorkommen, daß ein Ableitungspfad unendlich wird.

Durch Wiederholungskontrolle soll sowohl redundante Arbeit vermieden als auch die Terminierung für jede Anfrage garantiert werden.

- Subqueries und zugehörige Antworten werden memoisiert.
- Eine Subquery, die erneut gestellt wird, wird mit den für diese Subquery memoisierten Antworten beantwortet. [9]
- Werden zu einer Subquery neue Antworten generiert, so müssen alle Beweise, die sich auf Antworten zu dieser Subquery stützen, mit diesen neuen Antworten weitergeführt werden.

[9] Zu diesem Zeitpunkt liegen i.Allg. aber noch nicht *alle* Antworten auf die Subquery vor.

5.1 OLDT

OLDT-Resolution ist eine Erweiterung der SLD-Resolution um Memoisierung. Ihr liegt ein Spezialfall der SLD-Resolution zugrunde, die sogenannte OLD-Resolution ("Ordered selection strategy with linear resolution for definite clauses"). Diese entspricht der SLD-Resolution eingeschränkt auf die Verwendung der Standardberechnungsregel (das am weitesten links stehende Literal wird ausgewählt). OLDT steht für "OLD-Resolution with Tabulation".
Das Konzept des Memoisierens von Zwischenergebnissen garantiert bei der OLDT-Resolution die Terminierung für Datalog-Programme, auch wenn ein entsprechender SLD-Resolutionsbaum unendliche Pfade enthält. Dazu wird im Verlauf einer Anfragebeantwortung eine globale Tabelle mit Teilzielen und zugehörigen berechneten Antworten (Lemmata) generiert. Soll eine Variante eines Teilziel, für das bereits ein Tabelleneintrag existiert, erneut bewiesen werden, so werden nicht mehr die Regeln der DDB, sondern nur die Lemmata aus der Tabelle herangezogen.
Eine Komplikation bei diesem Verfahren besteht darin, daß zu einem bestimmten Zeitpunkt, an dem auf die Antworten zu einer Anfrage in der Memotabelle zugegriffen wird, diese Anfrage noch nicht vollständig beantwortet sein braucht, d.h., daß nicht *alle* Antworten schon vorliegen. Daher muß in der Memotabelle zusätzlich der Zustand des Beweises für ein Teilziel, welches mit Hilfe der Lemmata bewiesen wird, abgespeichert werden, um ihn später, wenn zusätzliche Antworten hinzukommen, mit diesen neuen Antworten fortsetzen zu können. Der Zustand eines Beweises für ein Teilziel entpricht dem Pfad im SLD-Baum von der Wurzel zu diesem Teilziel. Unsere Implementierung von OLDT orientiert sich zu diesem Zweck an einem Metainterpreter für OLD-Resolution, bei dem der Suchbaum aus speziellen Knoten besteht, die Wurzelpfaden im SLD-Baum entsprechen (vgl. [BT92]). Diese Knoten sind eine Folge von sogenannten *Frames* der Form
$(H \leftarrow [G_1, \ldots, G_n])\Theta$
Für solche Frames gilt:

- Es gibt eine Programmklausel $H \leftarrow B_1, \ldots, B_m$
- $[G_1, \ldots, G_n]$ sind die Rumpfliterale $B_i, B_{i+1}, \ldots, B_m$, die noch zur Verarbeitung anstehen
- Θ ist Substitution, die sich durch die Beantwortung der Teilziele $B_1, \ldots, B_{i-1}$ ergeben hat.

Die Knoten, die sich im OLDT-Baum ergeben, sind Folgen solcher Frames und müssen unter Umständen in der Memotabelle vermerkt werden. Die Memotabelle besteht also abstraktaus Einträgen der Form

```
<Teilziel, Knoten, Lemmata>
```

Wir realisieren diese Tabelle in unserem Programm durch drei Prädikate, die die Knoten und Antwortmenge durch einzelne Fakten in der Prolog-Datenbasis repräsentieren:

- `subquery(CallId,Call)` : der Eintrag für ein Teilziel mit Id.
- `node(CallId, Node)` : die Liste der auf Antworten wartenden Knoten.
- `answer(CallId, Answer, AnwerId, History)` : Die Antworten auf Call mit einer Id und der "History"(vgl. Abschnitt 2).

Die Anfrageauswertung mit OLDT geschieht intuitiv dann folgendermaßen:
Soll ein Knoten K mit selektiertem Teilziel G expandiert werden, so sucht man in der Memotabelle zunächst nach einem Eintrag für einen $Call$, der Variante von G ist. Findet sich ein solcher Eintrag, wird für jedes Lemma aus der Liste der schon vorhandenen Antworten auf $Call$ ein Sohn-Knoten erzeugt, und K wird in die Liste der auf Antworten für $Call$ wartenden Knoten eingetragen. Ist noch kein Eintrag in der Memotabelle vorhanden, werden Sohn-Knoten mit Hilfe der relevanten Regeln und Fakten des Programms erzeugt.
Sobald eine Lösung L für ein Teilziel gefunden ist, wird L in die Liste der Antworten zu diesem Teilziel eingetragen, sofern nicht bereits eine Variante von L als Eintrag vorhanden ist. Ist L "neu", wird anschließend für alle Knoten, die auf Antworten auf das genannte Teilziel warten, mit Hilfe dieser neuen Antwort L ein weiterer Sohn-Knoten erzeugt.
Die im Vergleich zu den Vorwärtsauswertern hohe Komplexität der Rückwärtsauswerter spiegelt sich auch in der Implementierung wieder. Die Implementierung des zentralen Prädikats `oldt(+Node)` unterscheidet drei Fälle für das Argument `Node`:

```
oldt([(Answer,History) <- true]) :-
answer(0,OldAnswer,_,_), variant(Answer,OldAnswer)
   -> new_id(duplicate,_)
   ;  new_id(answer,AId), assert(answer(0,Answer,AId,History)).
oldt([(Answer,History) <- true | ParentNode]) :-
ParentNode = [_ <- (Call,_) | _],
( add_answer(Call,Answer,AId,History,Nodes)
    -> apply_subst(Answer,AId,ParentNode), map_nodes(Nodes,Answer,AId)
    ;  true ).
oldt(Node):-
Node = [_ <- (Call,_) | _],
( get_answer(Call,Node,Answers)
    -> map_answers(Answers,Node)
    ;  findall((Call,[Id]) <- Body,relevant_clause(Call,Body,Id),Frames),
```

```
new_id(query,QId), assert(subquery(QId,Call)),
map_oldt(Frames,Node) ).
```

5.2 RQSQ

Recursive Query/Subquery ist das zweite rückwärtsauswertende Verfahren, das wir im Rahmen dieses Fopras implementiert haben. Wir haben uns dabei auf die Artikel [Vie87b] und [BR86] gestützt. Leider hat sich kurz vor Ende des Fopras herausgestellt, daß RQSQ, so wie es in den genannten Artikeln dargestellt wird, unvollständig ist.
Die Memotabelle besteht bei RQSQ aus Einträgen der Form ¡Query, Answers¿. RQSQ merkt sich also im Unterschied zu OLDT *keine* Ableitungspfade. Wird eine Anfrage erneut gestellt, so wird sie mit den zu dieser Anfrage gehörenden Antworten aus der Memotabelle beantwortet. Nur neue Anfragen bewirken eine Expansion des Ableitungsbaumes . Vollständigkeit soll durch eine Iteration "Solange neue Antworten auf die Anfrage Q generiert werden, führe rqsq(Q) aus" erreicht werden, die rekursiv für jede Subquery durchlaufen wird.
Die Unvollständigkeit dieses Verfahrens zeigt sich an folgenden Beispielen: Im ersten Fall findet RQSQ alle Lösungen, jedoch nicht im zweiten Fall, der sich vom ersten nur dadurch unterscheidet, daß ein zusätzliches "Delay-Prädikat"in die rekursive Regel geschoben wurde.
Um vollständig zu sein, müßte RQSQ bei jeder Iteration den gesamten Ableitungsbaum nochmals durchgehen. Dies wird aber ausgeschlossen, wenn es zu einem Anfrageknoten in der letzten Iteration keine neuen Antworten gegeben hat.
Nach unserem Dafürhalten würde eine Lösung dieses Problems entweder auf ein OLDT-ähnliches Verfahren oder das ineffizientere Iterative-Query/Subquery-Verfahren hinauslaufen. IQSQ unterscheidet sich von RQSQ im wesentlichen darin, daß nach jeder Iteration die Unteranfragen aus der Memotabelle wieder gelöscht werden.

6 Graphische Oberfläche für die Auswertungsverfahren

Die Auswertungsverfahren werden über eine graphische Oberfläche bedient.
Die folgende Abbildung zeigt den Hauptbildschirm.
Der Hauptbildschirm ist in folgende Bereiche aufgegliedert:

- DDB-Bereich
  - Eingabefeld für den Namen der Datei, die die DDB enthält
  - Listbox in der die EDB-Prädikate der DDB angezeigt werden
  - Listbox in der die IDB-Prädikate der DDB angezeigt werden

k(a, b)
k(b, c)
v(Start, Ziel) :- k(Start, Ziel).
v(Start, Ziel) :- v(Start, Zwischen), k(Zwischen, Ziel).

Anfrage: ?- v(Start, Ziel).

1. Runde
v(Start, Ziel)

k(Start, Ziel)   v(Start, Zwischen)

Start = a, Ziel = b
Start = b, Ziel = c
=> neue Fakten: v(a, b), v(b,c)

2. Runde
v(Start, Ziel)

v(Start, Zwischen), k(Zwischen, Ziel)

Start = a, Zwischen = b, Ziel = c
Start = b, Zwischen = c, Ziel : keine Lösung
=> neue Fakten: v(a, c)

3. Runde
v(Start, Ziel)

v(Start, Zwischen), k(Zwischen, Ziel)

Start = a, Zwischen = c, Ziel : keine Lösung
=> keine neue Fakten !!!

insgesamt abgeleitete Fakten :
v(a,b), v(b, c), v(a, c)

k(a, b)
k(b, c)
v(Start, Ziel) :- k(Start, Ziel).
v(Start, Ziel) :- v_delay(Start, Zwischen), k(Zwischen, Ziel).
v_delay(Start, Ziel) :- v(Start, Ziel).

1. Runde
v(Start, Ziel)

k(Start, Ziel)   v_delay(Start, Zwischen)

Start = a, Ziel = b   v(Start, Zwischen)
Start = b, Ziel = c
=> neue Fakten: v(a, b), v(b,c)

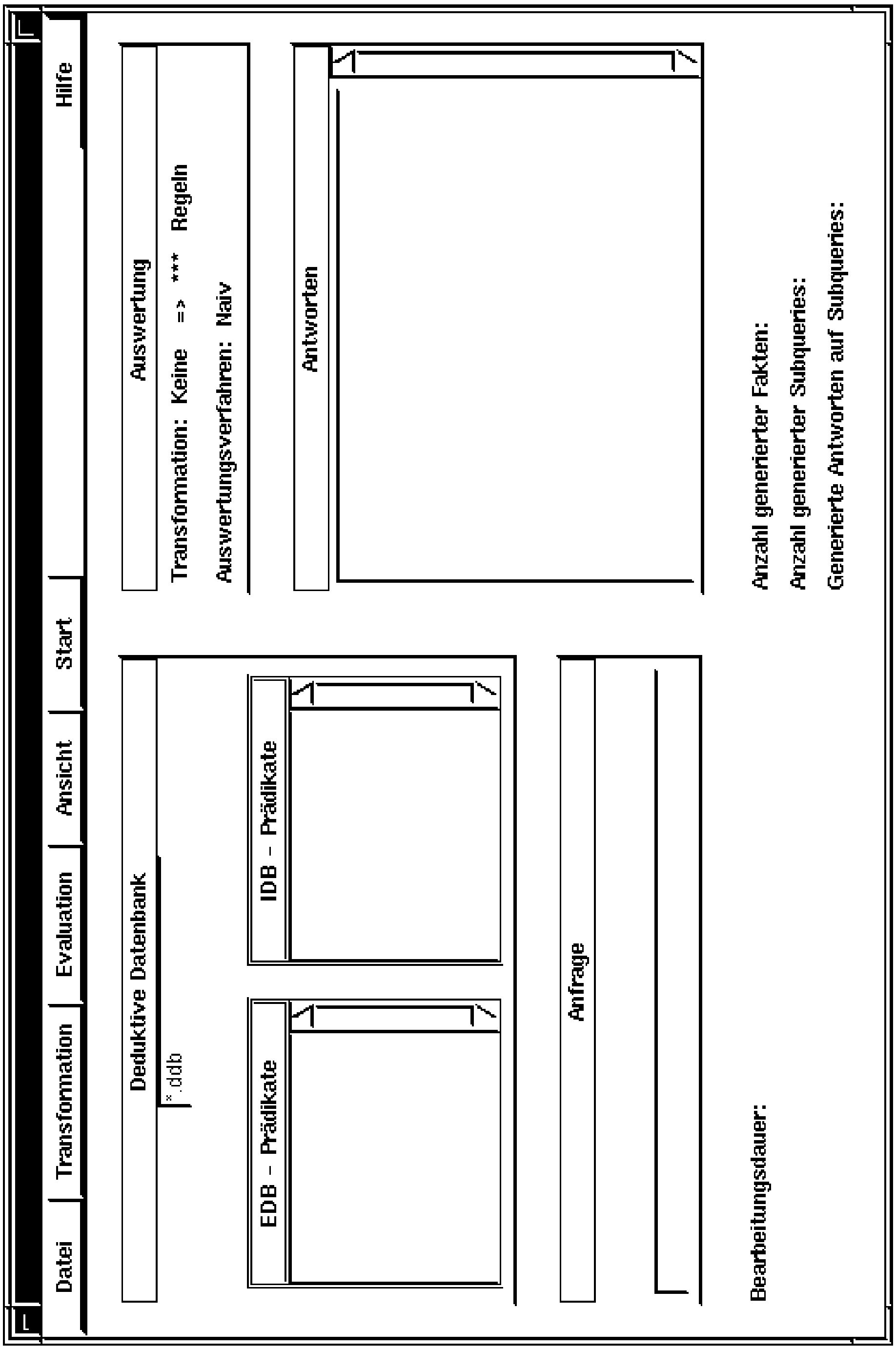
2. Runde
v(Start, Ziel)

v_delay(Start, Zwischen)

keine neuen Fakten
für v_delay(Start, Zwischen)
in der letzten Runde generiert !!!

=> keine neue Fakten !!!

insgesamt abgeleitete Fakten :
v(a,b), v(b, c)


Datei
Transformation
Evaluation
Ansicht
Start
Hilfe
Deduktive Datenbank
*.ddb
EDB - Prädikate
IDB - Prädikate
Anfrage
Auswertung
Transformation: Keine => *** Regeln
Auswertungsverfahren: Naiv
Antworten
Bearbeitungsdauer:
Anzahl generierter Fakten:
Anzahl generierter Subqueries:
Generierte Antworten auf Subqueries:

Eingabe des Dateinamens und ¡Return¿ bewirkt, daß die DDB geladen wird und die Listboxen für die EDB- bzw. IDB-Prädikate gefüllt werden.

- Anfrage-Bereich
  - Eingabefeld für die Anfrage

Eingabe der Anfrage und `<Return>` bewirkt, daß die Anfrage der Datenbank hinzugefügt wird. Bei konjunktiven Anfragen müssen die einzelnen Konjunktionsglieder durch Kommata getrennt werden.[10]

- Transformations-Bereich
  - Anzeige des gewählten Auswertungsverfahrens sowie der Transformationart der DDB-Regeln, falls eine solche gewählt wurde.

Die Anzahl der Regeln wird ebenfalls angezeigt.

- Antworten-Bereich
  - Anzeige der Antworten auf eine Anfrage
- Statistik-Bereich
  - Anzeige der Bearbeitungsdauer
  - Anzeige der Anzahl generierter Fakten bei Vorwärtsauswertung
  - Anzeige der Anzahl generierter Subqueries bei Rückwärtsauswertung
  - Anzeige der Anzahl an Antworten auf die generierten Subqueries [11]

Die Pull-Down-Menüs.

- Datei
  - `DDB Laden`: öffnet ein Fenster, in dem die DDB über eine Listbox ausgewählt werden kann. Das Fenster ermöglicht auch eine Voransicht der Dateien
  - `Beenden`: Schließt die graphische Oberfläche
- Transformation

[10]Durch Doppelclick mit der Maus auf ein Prädikat in den DDB-Listboxen erscheint dieses im Anfragefenster und kann dort editiert werden.

[11]Das IQSQ-Verfahren bildet hierbei eine Ausnahme, da es keine Antworten auf Subqueries memoriert, sondern nur Fakten, die nicht einer bestimmten Subquery zugeordnet werden.

  - Das Menü enthält Radio-Buttons zur Auswahl einer Transformation (voreingestellt ist "Keine")

- Evaluation

  - Das Menü enthält Radio-Buttons zur Auswahl eines Auswertungsverfahrens (voreingestellt ist "Naiv")

- Ansicht

  - `DDB Fakten`: zeigt die Fakten der gewählten DDB
  - `DDB Regeln`: zeigt die Regeln der gewählten DDB
  - `Adorned Regeln`: zeigt die Regeln nach der UBP-Transformation an (Siehe Menüpunkt "Start")
  - Magic Regeln:zeigt die Regeln nach der Magic-Transformation an (Siehe Menüpunkt "Start")
  - `Generierte Fakten`: zeigt die von vorwärtsauswertenden Verfahren generierten Fakten an.[12]
  - `Subqueries`: zeigt die von rückwärtsauswertenden Verfahren generierten Subqueries an
  - `Antworten auf Subqueries`: zeigt die von rückwärtsauswertenden Verfahren generierten Antworten an (vgl. vorige Fußnote)

- Start

  - `Auswertung (+ evtl. Transformation)`: startet die Auswertung mit den gewählten Verfahren
  - `Nur Regeln transformieren`: führt lediglich die gewählte Transformation der Regeln durch, ohne ein Auswertungsverfahren zu starten

[12]Bei diesem und dem übernächsten Fenster kann man sich durch Eingabe der ID, die links von den generierten Fakten bzw. Antworten angezeigt wird, den zugehörigen Ableitungsbaum anzeigen lassen.

7 Anhang

Die Implementierung der Vorwärtsauswertung (→ FORWARD.PL)

```
/******************************************************************************
                    Naive and seminaive fixpoint calculation
 ******************************************************************************/

/******************************************************************************
 calculate_fixpoint(+Evaluationtype)
 Berechnet iterativ den Fixpunkt des Datalog-Programms gemäß des gewählten
 Evaluationstyps (naiv oder seminaiv)
 ******************************************************************************/
calculate_fixpoint(EvaluationType) :-
        new_id(step,Step),
        doall(immediate_consequence(Step,EvaluationType)),
        fact(_,Step,_,_),
        !,
        calculate_fixpoint(EvaluationType).
calculate_fixpoint(_).

/******************************************************************************
 immediate_consequence(+Step,+EvaluationType)
 Leitet gemäß des gewählten Evaluationstyps (naiv oder seminaiv) einen Fakt
 aus dem Datalog-Programm ab
 ******************************************************************************/
immediate_consequence(Step,EvaluationType) :-
        rule((Fact :- Body),RuleId),
        evaluate(Body,Step,EvaluationType,BodyIds),
        update_database(Fact,Step,[RuleId|BodyIds]).

/******************************************************************************
 update_database(+Fact,+Step,+History)
 Falls der abgeleitete Fakt Fact noch nicht in der Datenbasis vorhanden ist,
 bekommt er eine eindeutige Nummer Id und wird mit Step und History in die
 Datenbasis geschrieben. Ist er bereits in der Datenbasis vorhanden, wird er
 als Duplikat mit Nummer Id, mit Step und der Nummer FactId des Fakts mit dem
 er identisch ist, in die Datenbasis geschrieben.
 ******************************************************************************/
update_database(Fact,_,_) :-
        fact(OldFact,_,_,_),
        variant(Fact,OldFact),
        new_id(duplicate,_),
        !.
update_database(Fact,Step,History) :-
        new_id(fact,Id),
        assert(fact(Fact,Step,Id,History)).

/******************************************************************************
 evaluate(+Body,+Step,+Evaluationtype,-BodyIds)
 Wertet einen Rumpf gemäß des gewählten Evaluationstyps (naiv oder seminaiv)
 über den Fakten der Datenbasis aus. BodyIds ist eine Liste mit den Ids der
 in die Auswertung eingegangenen Fakten.
```

```
 *****************************************************************************/
evaluate(true,_,naive,[]).
evaluate((B1,B2),Step,naive,[Id|Ids]) :-
        fact(B1,FactStep,Id,_),
        FactStep < Step,
        evaluate(B2,Step,naive,Ids).

evaluate(Body,Step,seminaive,BodyIds) :-
        append_c(Prefix,(Middle,Postfix),Body),
        delta_fact(Middle,Step,Id),
        old_facts(Prefix,Step,Ids1),
        any_facts(Postfix,Step,Ids2),
        append(Ids1,[Id|Ids2],BodyIds).

/*****************************************************************************
 old_facts(+Conjunction,+Step,-IdList)
 Wertet die Konjunktion Conjunction über den "alten", d.h. nicht im letzten
 Iterationsschritt generierten, Fakten aus. IdList ist eine Liste mit den Ids
 in die Auswertung eingegangenen Fakten.
 *****************************************************************************/
old_facts(true,_,[]).
old_facts((Fact,Facts),Step,[Id|Ids]) :-
        fact(Fact,OldStep,Id,_),
        OldStep < Step - 1,
        old_facts(Facts,Step,Ids).

/*****************************************************************************
 delta_fact(+Fact,+Step,-Id)
 Wertet Fact über den "neuen", d.h. im letzten Iterationasschritt generierten,
 Fakten aus. Die Id des gewählten Fakts wird zurückgegeben.
 *****************************************************************************/
delta_fact(Fact,Step,Id) :-
        PreviousStep is Step - 1,
        fact(Fact,PreviousStep,Id,_).

/****************************************************************************
 any_facts(+Conjunction,+Step,-IdList)
 Wertet die Konjunktion Conjunction über den Fakten der Datenbasis aus.
 IdList ist eine Liste mit den Ids in die Auswertung eingegangenen Fakten.
 ****************************************************************************/
any_facts(true,_,[]).
any_facts((Fact,Facts),Step,[Id|Ids]) :-
        fact(Fact,OldStep,Id,_),
        OldStep < Step,
        any_facts(Facts,Step,Ids).
```

Die Implementierung von UBP (→ UBP.PL)

```
/*************************************************************
 unique_binding_proberty
 Bildet zur Ausgangsquery das entsprechende Adornment und
 startet die abstrakte Evaluierung.
*************************************************************/
unique_binding_property :-
        query(Query,_,_),
        Query =.. [Functor|ArgList],
        adornment(Functor,ArgList,AdoList,Functor_ado),
        assert(adorned_predicate(Functor,AdoList,Functor_ado)),
        AdoQueryHead =.. [Functor_ado|ArgList],
        asserta(query(AdoQueryHead,0,0)),
        doall(abstract_evaluation([[Query,Functor_ado]])).

/*************************************************************
 abstract_evaluation(+Predicate_AdorndedFunctor_List)
 Predicate_AdorndedFunctor_List ist eine Liste von zwei-
 elementigen Listen, deren erstes Element ein Prädikat und
 deren zweites Element der mit einem Adornment versehene zu
 diesem Prädikat gehörende Funktor ist.
 Das Argument Predicate_AdorndedFunctor_List fungiert als
 Queue, um die "abstrakte" Top-Down Auswertung breadth-first
 zu simulieren.
 Die Regelmenge wird dabei in eine Form mit Adornments trans-
 formiert.
*************************************************************/
abstract_evaluation([]).
abstract_evaluation([[Predicate,Functor_ado]|Rest]) :-
        get_rule(Predicate,(Head :- Body),Id),
        propagate_bindings(Predicate,Head),
        find_body_adornments(Body,BodyPredList,Functor_adoList),
        assert_ubp_rule([Functor_ado|Functor_adoList],Id),
        append(Rest,BodyPredList,NewGoals),
        abstract_evaluation(NewGoals).

/*************************************************************
 get_rule(+Predicate,-Rule,-Id)
 Rule ist eine Regel mit der Nummer Id, deren Kopf denselben
 Funktor hat wie Predicate.
*************************************************************/
get_rule(Predicate,(Head :- Body),Id) :-
        functor(Predicate,Functor,Arity),
        functor(Head,Functor,Arity),
        ddb_rule((Head :- Body),Id).

/*************************************************************
 propagate_bindings(+Predicate,+Head)
 Überträgt Bindungen von Variablen in Predicate auf die
 Variablen in Head. Predicate und Head haben denselben
 Funktor.
*************************************************************/
```

```
propagate_bindings(Predicate,Head) :-
        Predicate =.. [Functor|ArgPred],
        Head      =.. [Functor|ArgHead],
        bind_arguments(ArgPred,ArgHead).


/***************************************************************
 bind_arguments(+PredicateArgumentList,+HeadArgumentList)
 Unifiziert korrespondierende Elemente der beiden Listen, wenn
 diese unifizierbar sind.
 Anwendung: Variablen in HeadArgumentList, die noch nicht
 gebunden sind, sollen entsprechend der Bindungen in
 PredicateArgumentList instantiiert werden, um das Bindungs-
 muster zu ermitteln.
***************************************************************/
bind_arguments([],[]).
bind_arguments([Arg|ArgRest1],[Arg|ArgRest2]) :-
        !,
        bind_arguments(ArgRest1,ArgRest2).
bind_arguments([_|ArgRest1],[_|ArgRest2]) :-
        bind_arguments(ArgRest1,ArgRest2).

/***************************************************************
 find_body_adornments(+Body,-BodyPredicateList,-AdornedFunctorList)
 Body ist der zu bearbeitende Rumpf, für den Adornements gefunden
 werden sollen.
 BodyPredicateList ist eine Liste von zweielementigen Listen,
 deren erstes Element ein Prädikat und deren zweites Element der
 mit einem Adornment versehene zu diesem Prädikat gehörende Funktor
 ist. Diese Liste dient zur Steuerung der Rekursion in
 abstract_evaluation. Sie enthält nur für solche Prädikate Einträge,
 deren mit Adornments versehene Form noch nicht in der Datenbasis
 vorhanden ist.
 AdornedFunctorList enthält die mit Adornments versehenen Funktoren
 aller Rumpfprädikate. Sie dient zur Transformation der Regel durch
 das Prädikat assert_ubp_rule.
***************************************************************/
find_body_adornments(true,[],[true]).
find_body_adornments((B1,B2),[[B1Copy,Functor_ado]|BR1],[Functor_ado|BR2]) :-
        copy_term(B1,B1Copy),
        B1 =.. [Functor|Args],
        functor(B1,Functor,Arity),
        idb(Functor,Arity),
        adornment(Functor,Args,Ado,Functor_ado),
        not(adorned_predicate(Functor,Ado,Functor_ado)),
        !,
        assert(adorned_predicate(Functor,Ado,Functor_ado)),
        assert(idb(Functor_ado,Arity)),
        instantiate_argumentlist(Args),
        find_body_adornments(B2,BR1,BR2).
find_body_adornments((B1,B2),BR1,[Functor_ado|BR2]) :-
        B1 =.. [Functor|Args],
        functor(B1,Functor,Arity),
        idb(Functor,Arity),
```

```
        adornment(Functor,Args,_,Functor_ado),
        !,
        instantiate_argumentlist(Args),
        find_body_adornments(B2,BR1,BR2).
find_body_adornments((B1,B2),BR1,[Functor|BR2]) :-
        B1 =.. [Functor|Args],
        instantiate_argumentlist(Args),
        find_body_adornments(B2,BR1,BR2).

/************************************************************
 adornment(+Functor,+ArgList,-AdoList,-Functor_ado)
 Konkateniert den Funktor Functor mit dem Adornment, das sich
 aus der Analyse von ArgList ergibt, zu dem Funktor Functor_ado.
************************************************************/
adornment(Functor,ArgList,AdoList,Functor_ado) :-
        ado(ArgList,AdoList),
        concat_atom(AdoList,Ado),
        (AdoList == [] -> Functor_ado = Functor
                        ; concat_atom([Functor,´_´,Ado],Functor_ado)
        ).

/************************************************************
 ado(+ArgumentList,-AdornmentList)
 Analysiert ArgumentList und erstellt eine Liste aus b´s und f´s
 für gebundene bzw. freie Element aus ArgumentList.
************************************************************/
ado([],[]).
ado([Arg|Args],[f|Bindings]) :-
        var(Arg),
        ado(Args,Bindings).
ado([Arg|Args],[b|Bindings]) :-
        nonvar(Arg),
        ado(Args,Bindings).

/************************************************************
 instantiate_argumentlis(+ArgumentList)
 Instantiiert freie Variablen in ArgumentList mit der Konstante $.
************************************************************/
instantiate_argumentlist([]).
instantiate_argumentlist([Arg|Args]) :-
        var(Arg),
        !,
        Arg = ´$´,
        instantiate_argumentlist(Args).
instantiate_argumentlist([_|Args]) :-
        instantiate_argumentlist(Args).

/************************************************************
 assert_ubp_rule(+AdornedFunctorList,+Id)
 Transformiert die Regel mit der Nummer Id in die "adornte" Form.
************************************************************/
assert_ubp_rule([HeadFunctor|BodyFunctors],Id) :-
        ddb_rule((Head :- Body),Id),
        Head =.. [_|Args],
```

```
        Head_ado =.. [HeadFunctor|Args],
        adorn_body(Body,BodyFunctors,AdoBody),
        new_id(ubp_rule,NewId),
        assert(ubp_rule((Head_ado :- AdoBody),NewId)).

/************************************************************
 adorn_body(+Body,+AdornedFunctorList,-AdornedBody)
 Ersetzt in allen Prädikaten des Rumpfs Body den Funktor durch
 seine "adornte" Form.
************************************************************/
adorn_body(true,[true],true).
adorn_body((B1,B2),[FunctorB1_ado|FunctorB2_ado],(B1_ado,B2_ado)) :-
        B1 =.. [_|Args],
        B1_ado =.. [FunctorB1_ado|Args],
        adorn_body(B2,FunctorB2_ado,B2_ado).
```

Die Implementierung der Magic-Set Transformation (→ MAGIG.PL)

```
/****************************************************************************
                    Magic templates transformation
 ****************************************************************************/

magic :-
        ubp_rule((Head :- Body),_),
        magic_predicate(Head,MagicHead),
        new_id(magic_rule,Id1),
        assert(magic_rule(Head :-(MagicHead,Body),Id1)),
        append_c(Prefix,(Pred,_),Body),
        magic_predicate(Pred,MagicPred),
        new_id(magic_rule,Id2),
        assert(magic_rule(MagicPred :-(MagicHead,Prefix),Id2)),
        fail.
magic :-
        adorned_predicate(query,_,QueryF),
        concat_atom([´magic_´,QueryF],MagicQuery),
        asserta(fact(MagicQuery,0,0,[])).

magic_predicate(Predicate,MagicPredicate) :-
        Predicate =.. [Functor|Args],
        adorned_predicate(_,AdoList,Functor),
        concat_atom([´magic_´,Functor],MagicFunctor),
        bound_variables(Args,AdoList,VarList),
        MagicPredicate =.. [MagicFunctor|VarList].

bound_variables([],[],[]).
bound_variables([A|AR],[b|R1],[A|R2]) :-
        !,
        bound_variables(AR,R1,R2).
bound_variables([_|AR],[f|R1],R2) :-
        bound_variables(AR,R1,R2).
```

Die Implementierung von OLDT (→ OLDT.PL)

```
/***********************************************************************
                       OLD-Resolution With Tabulation
Die Memotabelle der Form (ZeilenNr, Call, KnotenListe, AntwortListe)
wird hier durch drei Prädikate realisiert, die die Knoten- und Antwortliste
als Fakten in der Prolog-Datenbasis repräsentieren:
 - subquery(QueryId,Call)  % QueryId entspricht der ZeilenNr
 - node(QueryId, Node)     % die Liste der auf Antworten wartenden Knoten
 - answer(QueryId, Answer, AnwerId, History)  % Die Antworten auf Call
***********************************************************************/

/***********************************************************************
Der Operator "<-" dient zur Konstruktion der "Frames".
Eine Folge von Frames (H <- [G1,...,Gn]) ist ein OLD-Knoten und entspricht
einem Pfad von der Wurzel zu einem Knoten in einem SLD-Resolutionsbaum.
Die Frames werden hier im Kopfliteral um eine "Historie" der Form
[RegelId,FactId1,...,FactIdn] bzw. [FactId] für ihre Ableitung aus Regeln
bzw. Fakten, erweitert. Damit kann später der Resolutionsbaum für die
graphische Darstellung rekonstruiert werden.
***********************************************************************/
:- op(500,xfy,<-).


/***********************************************************************
oldt/0
Sucht die Ausgangsanfrage und startet die Evaluation für den OLDT-Wurzel-
knoten [Query <- Body].
***********************************************************************/
old_resolution_with_tabulation  :-
        query(Query,_,_),
        rule((Query :- Body),RId),
        oldt([(Query,[RId]) <- Body]).


/***********************************************************************
oldt(+Node)
1) Node = [(Answer,History) <- true]: eine Antwort "Answer" mit der
   Historie "History" für die Ausgangsanfrage in der Wurzel des OLDT-Baums
   ist gefunden.
2) Node = [(Answer,History) <- true | [_ <- (Call,_) | _]]: Ein Blattknoten
   ist erreicht, der eine Antwort "Answer" auf "Call" liefert. Ist die
   Antwort neu, so wird sie in die Memo-Tabelle eingetragen, der Vorgänger-
   knoten und alle Knoten aus der Memo-Tabelle, die auf Antworten auf Call
   warten, werden mit dieser Substitution weiterbearbeitet.
3) Node = [_ <- (Call,_) | _] ist ein "innerer" Knoten.
   Sind schon Antworten für diesen Knoten in der Memotabelle vorhanden, so
   wird er mit diesen Substitutionen für Call weiterbearbeitet. Sonst ist
   Call eine neue Subquery und der Knoten wird mit den relevanten Klauseln
   - Regeln und Fakten, die mit Call unifizierbar sind - expandiert.
***********************************************************************/
oldt([(Answer,History) <- true]) :-
       (answer(0,OldAnswer,_,_), variant(Answer,OldAnswer))
```

```
         -> new_id(duplicate,_)
         ;
            new_id(answer,AId),
            assert(answer(0,Answer,AId,History)).
oldt([(Answer,History) <- true | ParentNode]) :-
        ParentNode = [_ <- (Call,_) | _],
        (add_answer(Call,Answer,AId,History,Nodes) ->
             apply_subst(Answer,AId,ParentNode),
             map_nodes(Nodes,Answer,AId)
             ;
             true).
oldt(Node):-
        Node = [_ <- (Call,_) | _],
        (get_answer(Call,Node,Answers) ->
             map_answers(Answers,Node)
             ;
             findall((Call,[Id]) <- Body,relevant_clause(Call,Body,Id),Frames),
             new_id(query,QId),
             assert(subquery(QId,Call)),
             oldt_map_rules(Frames,Node)
             ).


/**********************************************************************
oldt_map_rules(+FrameList,+Node)
Expandiert den Knoten Node mit den Frames (diese entsprechen Fakten und
Regeln) aus FrameList und ruft oltd/1 mit diesen neuen Knoten zur
Bearbeitung auf.
**********************************************************************/
oldt_map_rules([],_).
oldt_map_rules([Frame|Frames],Node) :-
        oldt([Frame|Node]),
        oldt_map_rules(Frames,Node).


/**********************************************************************
apply_subst(+Answer,+AId,+FrameList)
Wendet die Substitution der gefundenen Antwort "Answer" für das Rumpf-
literal "Call" auf dieses Literal an, fügt die AntwortId "AId" in den Kopf
des Frames ein und ruft dann oldt/1 zur Bearbeitung der restlichen Rumpf-
literale des aktuellen Frames auf.
**********************************************************************/
apply_subst(Answer,AId,[Frame|Frames]) :-
        copy_term(Frame,(A,H) <- (Call,Goals)),
        Call = Answer,
        append(H,[AId],Hist),
        oldt([(A,Hist) <- Goals | Frames]).


/**********************************************************************
map_nodes(+NodeList,+Answer,+AId)
Weiterverarbeitung aller Knoten in NodeList (aus der Memotabelle) mit der
Antwort "Answer".
**********************************************************************/
```

```
map_nodes([],_Answer,_AId).
map_nodes([Node|Nodes],Answer,AId) :-
        apply_subst(Answer,AId,Node),
        map_nodes(Nodes,Answer,AId).


/*************************************************************************
map_answers(+AnswerList,+Node)
Weiterverarbeitung des Knotens "Node" mit den Antworten in "AnswerList"
(aus der Memotabelle).
*************************************************************************/
map_answers([],_Node).
map_answers([(Answer,AId)|Answers],Node) :-
        apply_subst(Answer,AId,Node),
        map_answers(Answers,Node).


/*************************************************************************
add_answer(+NewCall,+Answer,-AId,+History,-NodeList)
Wenn "NewCall" eine Variante eines Call in der Memotabelle ist, dann wird
die Antwort "Answer" mit Historie "History" und neuer Id "AId" in der
Tabelle bei Call eingetragen. Die AntwortId und die Liste aller Knoten
aus der Memotabelle, die auf Antworten für Call warten, werden zurück-
gegeben.
*************************************************************************/
add_answer(NewCall,Answer,AId,History,Nodes) :-
        subquery(QId,Call),
        variant(NewCall,Call),
        !,
        ( answer(QId,OldAnswer,_,_), variant(OldAnswer,Answer)
         -> new_id(duplicate,_),fail
         ;
            findall(Node,node(QId,Node),Nodes),
            new_id(answer,AId),
            assert(answer(QId,Answer,AId,History))
        ).


/*************************************************************************
get_answer(+NewCall,+Node,-AnswerList)
Wenn "NewCall" aus dem Knoten "Node" eine Variante eines Call in der Memo-
tabelle ist, dann werden die Antworten auf Call in "AnswerList" zurück-
gegeben und der Knoten "Node" in der Tabelle bei Call eingetragen.
*************************************************************************/
get_answer(NewCall,Node,Answers) :-
        subquery(QId,Call),
        variant(NewCall,Call),
        new_id(duplicate_query,_),
        findall((Answer,AId),answer(QId,Answer,AId,_),Answers),
        assert(node(QId,Node)).


/*************************************************************************
relevant_clause(+Call,-Body,-Id)
```

```
Der Rumpf von Regeln oder Fakten, die mit dem Call unifizieren, werden
mit Regel- bzw. Fakt-Id zurückgegeben.
***********************************************************************/
relevant_clause(Call,Body,Id) :-
        rule((Call :- Body),Id)
        ;
        fact(Call,_,Id,_),
        Body = true.
```

Die Implementierung von RQSQ (→ RQSQ.PL)

```
/***********************************************************************
                     Recursive Query/Subquery
***********************************************************************/

/***********************************************************************
 recursive_query_subquery
 Ruft das Prädikat rqsq mit der Anfrage Query und deren Id-Nummer QId auf.
***********************************************************************/
recursive_query_subquery :-
        query(Query,QId,_),
        rqsq(Query,QId).

/***********************************************************************
 rqsq(+Query,+QId)
 Alle Lösungen auf die Anfrage Query werden mittels aller Fakten und
 Regeln, deren Kopf mit Query matcht, durch eine iterative Schleife
 gefunden. Die Schleife bricht ab, wenn keine neuen Antworten auf die
 Anfrage mehr gefunden werden.
***********************************************************************/
rqsq(Query,QId) :-
        consult_facts(Query,QId,Answers),
        findall(rule((Query :- Body),Id),rule((Query :- Body),Id),Rules),
        rqsq_map_rules(Rules,AnswerList),
        flatten([Answers|AnswerList],As),
%        write(As),nl,nl,
        last_id(answer,AId),
        add_answers(As,QId,[_|_]),
        rqsq(Query,QId).
rqsq(_,_).

/***********************************************************************
 consult_facts(+Query,+QId)
 Wurde die Anfrage Query mit Id QId noch nicht über den Fakten der ge-
 ladenen Datenbasis ausgewertet, werden alle Fakten, welche mit Query
 matchen als Antworten auf Query in die Datenbasis geschrieben. Im anderen
 Fall gelingt consult_facts ohne daß etwas gemacht wird.
***********************************************************************/
consult_facts(Query,QId,Answers) :-
        answer(QId,_,_,[_])
          -> Answers = []
          ; findall((Query,[Id]),fact(Query,_,Id,_),Answers).

/***********************************************************************
 rqsq_map_rules(+RuleList,-AnswerList)
 Zu jeder Regel R aus RuleList werden alle Möglichkeiten den Kopf mittels
 des Rumpfs über der Datenbasis zu erfüllen in einem Element der Liste
 AnswerList gesammelt. Diese ist eine Liste von Listen, welche wiederum
 Paare der Form (Head,History) enthalten. History ist die Liste der Ids
 der bei der Ableitung von Head verwendeten Antworten.
***********************************************************************/
rqsq_map_rules([],[]).
```

```
rqsq_map_rules([rule((Head :- Body),Id)|Rules],[Answers|RestAnswers]) :-
        findall((Head,[Id|History]),rqsq_body(Body,History),Answers),
        rqsq_map_rules(Rules,RestAnswers).

/************************************************************************
 rqsq_body(+BodyLiterals,-History)
 BodyLiterals ist beim Aufruf das noch unausgewertete Reststück eines
 Regelrumpfs. Besteht dieses Reststück nur noch aus dem Literal true, ist
 der Rumpf abgearbeitet und das Prädikat terminiert. Andernfalls wird für
 das erste Literal B von BodyLiterals geprüft, ob es bereits als Subquery
 behandelt wurde. Wenn ja, wird der Duplikatszähler für die Subqueries
 hochgesetzt und versucht, B mit einer bereits in der Datenbasis befind-
 lichen Antwort auf diese Subquery zu matchen. Gelingt dies, wird
 rqsq_body rekursiv mit dem Bodyreststück ohne das erste Literal aufgeru-
 fen. Wurde das erste Literal B noch nicht als Subquery gestellt, wird
 B mit einer Query-Id als Subquery in die Datenbasis geschrieben, das
 Hauptprädikat rqsq mit B aufgerufen und anschliessend versucht B mit
 einer der generierten Antworten zu matchen und in die Rekursion wie im
 ersten Fall gegangen. Die Ids der gefundenen Antworten werden in der
 Liste History gesammelt.
************************************************************************/
rqsq_body(true,[]).
rqsq_body((B,Bs),Ids) :-
        not (subquery(_,Q),variant(B,Q)),
        new_id(query,QId),
        assert(subquery(QId,B)),
        rqsq(B,QId),
        !,
        rqsq_body((B,Bs),Ids).
rqsq_body((B,Bs),[Id|Ids]) :-
        subquery(QId,Q),
%        write(Q),nl,
        variant(B,Q),
        new_id(duplicate_query,_),
        answer(QId,B,Id,_),
        rqsq_body(Bs,Ids).
/************************************************************************
 add_answers(+AnswerList,+QueryId,-NewAnswerList)
 AnswerList ist die Liste von Paaren, deren erstes Element eine Antwort
 auf die Query mit der Id-Nummer QueryId und deren zweites Element die
 History dieser Antwort ist. Für jedes Paar wird geprüft, ob eine Variante
 des ersten Elements bereits als Antwort mit derselben QueryId in der
 Datenbasis ist. In diesem Fall wird sie als Duplikat lediglich gezählt.
 Andernfalls erhält die Antwort eine Antwort-Id und wird als Antwort auf
 die Query mit der Id-Nummer QueryId mit der History gespeichert und wird
 außerdem in die Liste NewAnswerList übernommen.
************************************************************************/
add_answers([],_,[]).
add_answers([(S1,_)|S2],QId,News) :-
        answer(QId,Answer,_,_),
        variant(S1,Answer),
        new_id(duplicate,_),
        !,
        add_answers(S2,QId,News).
```

```
add_answers([(S1,Hist)|S2],QId,[S1|News]) :-
        new_id(answer,AId),
        assert(answer(QId,S1,AId,Hist)),
        add_answers(S2,QId,News).
```

Die Implementierung von IQSQ (→ IQSQ.PL)

```
/****************************************************************************
                Iterative Query/SubQuery (IQSQ)
 ****************************************************************************/

/****************************************************************************
 iterative_query_subquery
 Setzt den Step-Zähler auf 1 und ruft die Iterationsschleife iqsq/1 mit diesem
 Step auf.
 ****************************************************************************/
iterative_query_subquery :-
        new_id(step,Step),
        iqsq(Step).

/****************************************************************************
 iqsq(+Step)
 Ruft iqsq/3 solange mit der Query auf, bis in einer Runde keine neuen Fakten
 mehr generiert werden. Ist ein neuer Fakt generiert worden, wird Step um eins
 hochgezählt, es werden alle Subqueries aus der Datenbank gelöscht und es wird
 in die Iteration gegangen.
 ****************************************************************************/
iqsq(Step) :-
        query(Query,_,_),
        doall(iqsq(Query,Step,_)),
        fact(_,Step,_,_),
        new_id(step,NextStep),
        retract_all(subquery(_,_)),
        iqsq(NextStep).
iqsq(_).

/****************************************************************************
 iqsq(?Query,+Step,-Id)
 Ein Beweis für Query wird Top-Down geführt. Gelingt der Beweis, wird die Id
 des neu generierten Fakts zurückgegeben und dieser der Datenbasis hinzuge-
 fügt. Subqueries werden zur Wiederholungskontrolle gespeichert.
 ****************************************************************************/
iqsq(Query,_,Id) :-
        new_id(query,QueryId),
        fact(Query,_,Id,_).
iqsq(Query,Step,Id) :-
        not((subquery(_,OldQuery),variant(Query,OldQuery))),
        rule((Query :- Body),RuleId),
        new_id(query,QueryId),
        assert(subquery(QueryId,Query)),
        iqsq_body(Body,Step,Ids),
        assert_answer(Query,Step,Id,[RuleId|Ids]).

/****************************************************************************
 iqsq_body(?Body,+Step,-IdList)
 Ein Beweis für Body wird Top-Down geführt, indem iqsq/3 für jedes Bodyliteral
 aufgerufen wird.
 ****************************************************************************/
```

```
iqsq_body(true,_,[]).
iqsq_body((C1,C2),Step,[Id1|Id2]) :-
        iqsq(C1,Step,Id1),
        iqsq_body(C2,Step,Id2).

/*****************************************************************************
 assert_answer(+Fact,+Step,-Id,+History)
 Steht eine Variante von Fact bereits in der Datenbasis, wird Fact lediglich
 gezählt; anderenfalls bekommt Fact eine Id und wird mit Step, Id und History
 in die Datenbank geschrieben.
 *****************************************************************************/
assert_answer(Fact,_,_,_) :-
        fact(OldFact,_,_,_),
        variant(Fact,OldFact),
        new_id(duplicate,_),
        !.
assert_answer(Fact,Step,Id,History) :-
        new_id(fact,Id),
        assert(fact(Fact,Step,Id,History)).
```

Literatur

[Abd94] S. Abdennadher. Anfrageauswertung und -optimierung - Klassifizierung. Technical Report PMS-1994-3, LMU, 1994.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGMOD-SIGART symposion on principles of database systems*, pages 1–15, Cambridge, Massachusetts, 1986.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM-SIGMOD international conference on management of data*, pages 16–52, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART symposion on principles of database systems*, pages 269–283, San Diego, California, 1987.

[Bry90] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[BT92] C. Beckmann and G. Tobermann. Aktuelle Probleme der Logikprogrammierung - Neue Verarbeitungsmodelle durch Meta-Interpreter. Vorlesung SS 1992, Juli 1992.

[CGH94] A.B. Cremers, U. Griefahn, and R. Hinze. *Deduktive Datenbanken. Eine Einführung aus der Sicht der logischen Programmierung.* Vieweg, 1994.

[Das92] S.K. Das. *Deductive databases and logic programming.* Addison-Wesley Publishing Company, 1992.

[Ull88] J. Ullman. *Principles of database and knowledge-base systems*, volume 1. Computer science press, 1988.

[Ull89] J. Ullman. *Principles of database and knowledge-base systems*, volume 2. Computer Science Press, 1989.

[Vie87a] L. Vieille. Database-complete proof procedures based on SLD resolution. In *Proceedings of the 4th int. conference on logic programming*, pages 74–103, 1987.

[Vie87b] L. Vieille. Recursive axioms in deductive databases: the query/subquery approach. In L. Kerschberg, editor, *Expert Database*

Systems, pages 253–267. Benjamin-Cummings, Menlo Park, CA, 1987.