FAKULTÄT FÜR !NFORMATIK

Analyse von Redundanzen beim Datenaustausch

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Florian Wagner
Matrikelnummer 0226296

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ. Prof. Dr. Reinhard Pichler
Mitwirkung: MSc. Vadim Savenkov

Wien, 08.07.2009

_____ _____
(Unterschrift Verfasser) (Unterschrift Betreuer)

Florian Wagner
Platzl 14
3180 Lilienfeld

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 08.07.2009

_____

(Unterschrift Verfasser)

Kurzfassung

Beim Datenaustausch (*data exchange*) wird versucht, die Daten eines Quellschemas möglichst adäquat in Daten eines Zielschemas zu überführen. Im Allgemeinen gibt es eine Vielzahl von Lösungen unterschiedlicher Größe. Die Kompakteste aller Lösungen wird Kern (*core*) genannt und sollte, wie Fagin et al. in [4, 5] beschreibt, in der Zieldatenbank gespeichert werden. Mittels Chase kann eine Lösung berechnet werden. Um aus einer Lösung den Kern zu berechnen, existieren verschiedene Algorithmen, wie der Blocks- oder FindCore Algorithmus.

Bei größeren Datenbeständen ist die Kernberechnung trotz polynomieller Laufzeit extrem aufwändig und nicht mehr sinnvoll durchzuführen, weshalb ein anderer Ansatz notwendig wird. Dieser wurde von Pichler und Savenkov [17] entwickelt und versucht durch Änderung der Abhängigkeiten zwischen Quelle und Ziel den Kern direkt durch den Chase zu erstellen. Dazu, muss das Zusammenspiel aller Abhängigkeiten untersucht werden. Die Abhängigkeiten können dabei ohne Berücksichtigung der eigentlichen Daten, jedoch für verschiedene Eingabemuster untersucht werden.

Genau diese wesentliche Idee des neuen Ansatzes, nämlich das Zusammenspiel der Quelle-zu-Ziel Abhängigkeiten zu analysieren wird in dieser Diplomarbeit durch eine Implementierung abgedeckt. Damit ist die Basis für eine direkte Erzeugung des Kerns durch einen abgewandelten Chase geschaffen und die Voraussetzung, um solch einen Chase überhaupt entwickeln zu können.

Die Berechnung der Abhängigkeitskombinationen erfolgt mittels Brute-Force Methode, die Laufzeit verändert sich somit exponentiell mit der Anzahl der gegebenen Abhängigkeiten, wobei dieser hohe Aufwand nur einmalig anfüllt und dafür auf die aufwändige Berechnung des Kerns aus einer allgemeinen Lösung verzichtet werden kann. Damit ist diese Methode für große Datenmengen besser geeignet als der frühere Ansatz.

Den größten Teil der Arbeit stellt die Implementierung in Java dar. In diesem Dokument wird die Implementierung und deren theoretische Überlegungen behandelt. Um eine Übersicht über den Themenbereich Datenaustausch zu bekommen, wird ein Überblick über dessen wesentliche Grundlagen gegeben. Außerdem werden einige frühere Algorithmen zur Kern-Berechnung erläutert, nämlich der Block-Algorithmus [5] von Fagin et al. sowie der FindCore Algorithmus [12] von Gottlob und Nash.

Abstract

Data exchange tries to transfer data from a source schema as accurate as possible into a target schema. Each source may have different solutions with different sizes. Fagin et al. illustrate in [4, 5] that among these solutions the smallest solution, which is called the core, should be stored in the target database. The chase is able to compute one solution. To get the core from this solution we can use some different algorithms for the so called core computation. There are some algorithms that are able to compute the core like the Blocks algorithm and the FindCore Algorithm.

Even though the algorithms have polynomial runtime, dealing with large databases becomes extremely expensive and can't be done in acceptable time, thus a new method is needed. Pichler and Savenkov [17] developed a new method, which tries to change the dependencies between source and target to compute the core via chase. To directly compute the core, we have to analyze all dependencies and their interaction. The dependencies can be analyzed without the data, but for all possible input schemas.

The main idea of this new method, the analysis of the source to target dependencies, is implemented in this thesis. The new generated versions of source-to-target dependencies are essential for developing a modified chase, which is able to generate the core.

Analyzing the combination of dependencies is done by Brute-Force, which means that the runtime is increasing exponentially with the number of dependencies. On the other hand this effort has to be done only once. Thus this method is more suitable for large databases than the earlier method.

The biggest part of the thesis is the implementation in Java. This document shows the implementation and their theoretical base. To get an overview of data exchange, the main aspects are shown. Furthermore we present some earlier algorithms for core computation: the Blocks algorithm [5] from Fagin et al. and the FindCore algorithm [12] from Gottlob and Nash.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Grundlagen

Weltweit schreitet die Vernetzung von Computersystemen immer weiter voran, was Organisationen in die Lage versetzt, Daten zwischen verschiedensten Systemen miteinander austauschen und Anfragen auf diese Daten beantworten zu müssen. Daraus ergibt sich das Bedürfnis, Daten zwischen verschiedenen Systemen zu transferieren und ineinander umzuwandeln, mit dem Ziel, eine konsistente globale Sicht auf alle Datenquellen zu ermöglichen.

Aus diesem Umstand haben sich zwei ähnliche Bereiche entwickelt, die sich mit dieser Thematik befassen: Datenintegration (*data integration*) und Datenaustausch (*data exchange*).
Bei der Datenintegration, oder auch virtueller bzw. logischer Integration, gibt es eine (globale) Zielinstanz. Diese ist hauptsächlich dafür verantwortlich, eine zentrale Anlaufstelle für Datenabfragen auf verschiedene Quellinstanzen zu sein. Bei einer Anfrage auf die Zielinstanz werden die Anfragen passend zum jeweiligen Quellschema umgeschrieben und auf die einzelnen Quellinstanzen angewendet. Sobald alle Einzelanfragen abgearbeitet wurden, werden die Ergebnisse miteinander verknüpft und als ein Datensatz (der Zielinstanz) zurückgegeben. Weil die Daten auf die einzelnen Quellinstanzen verteilt bleiben ist die Zielinstanz nichts weiter, als ein virtuelles Datenbankschema, welches selber keine Daten enthält, sondern Anfragen auf dieses Schema nur in Anfragen an einzelne Quellschemen umschreibt.
Auf der anderen Seite beschäftigt sich Datenaustausch, oder auch materialisierte bzw. physische Integration, damit, Daten aus einer oder mehreren Quelle in ein Ziel zu übernehmen. Um diese Daten zu übernehmen werden die sogenannten Quelle-zu-Ziel Abhängigkeiten (*source-to-targed dependencies STDs*) verwendet, welche zu den Tupel-generierenden Abhängigkeiten (*tuple generating dependencies TGD*) gehören. Sie sind eine Folgerungen mit einer Verknüpfung von Atomen in der Voraussetzung (*premise*) als auch in der Aussage (*conclusion*). STDs generalisieren Global-als-Sicht (*global-as-view* GAV) Abbildungen und Lokal-als-Sicht (*local-as-view* LAV)

Abbildungen und werden manchmal als GLAV (*globla-local-as-view*) Abbildungen bezeichnet [1, 2]. Bei LAV werden die lokalen Datenquellen als eine Sicht (*view*) auf das (virtuelle) globale Schema betrachtet, während bei GAV das globale Schema erstellt wurde, um eine Sicht auf die lokalen Schemen zu sein.
Neben den TGDs ist es in beim Datenaustausch auch möglich, dass die Zieldatenbank Einschränkungen (*constraints*) der Daten enthält, welche Ziel-Abhängigkeiten (*target dependencies* TDs) genannt werden. Diese können neben STDs auch gleichheitserzeugende Abhängigkeiten (*equality-generation dependencies* EGDs) enthalten.

Von Fagin, Kolaitis, Miller und Popa wurde in [2] das Datenaustauschproblem (*data exchange problem*) definiert. Hierbei werden das Quell- und Zielschema als auch STDs und TDs als unveränderlich angesehen und als Datenaustauschszenario (*data exchange szenario*) bezeichnet. Das zugehörige Datenaustauschproblem beschäftigt sich dann damit, eine Zielinstanz zu finden, welche zusammen mit der Quellinstanz alle Abhängigkeiten erfüllt. Die Zielinstanz entsteht dabei durch schrittweises überführen der Daten aus der Quellinstanz in die Zielinstanz. Eine gute Lösung muss dabei so generell wie möglich sein und wird in diesem Fall als universelle Lösung (*universal solution*) bezeichnet. Die Chase-Prozedur ist in der Lage eine solche Lösung zu generieren. Dazu wird schrittweise die Zielinstanz aus der Quellinstanz mithilfe der STDs und TDs aufgebaut bis alle Abhängigkeiten erfüllt sind, oder eine oder mehrere Abhängigkeiten nicht erfüllt werden können. Als Ergebnis liefert diese Prozedur im Erfolgsfall die sogenannte kanonische universelle Lösung (*canonical universal solution*). Wenn eine der Abhängigkeiten nicht erfüllt werden kann gibt es keine Lösung.

Das Problem bei diesen Lösungen ist jedoch, dass diese im Allgemeinen nicht einzigartig und verschieden groß sind. Unter allen Lösungen ist der sogenannte Kern (*core*) die kleinste und sollte in der Zieldatenbank gespeichert werden. Die Idee für Kerne stammt ursprünglich aus der Graphentheorie und kann auf den Datenaustausch übertragen werden, wie in [5] gezeigt wird. Außerdem wird von Fagin, Kolaitis und

Popa in diesem Artikel angeführt, dass der Kern besonders vorteilhafte Eigenschaften hat und daher beim Datenaustausch bevorzugt werden sollte.

Mit dem Berechnen des Kernes beschäftigen sich mehrere Artikel verschiedener Autoren [5, 19, 20]. Die einfachsten vorgestellte Algorithmen sind der Greedy- und Blocks Algorithmus aus [5]. In [17] wird der FindCore Algorithmus vorgestellt, welcher in polynomieller Zeit den Kern berechnen kann und EGDs sowie schwach azyklische (*weakly acyclic*) TGDs unterstützt. Der Algorithmus stellt EGDs als TGDs dar, indem zusätzliche Zielrelationen eingefügt werden, um gleiche Terme abzuspeichern und zusätzliche TGDs hinzugefügt werden. Hier liegt auch der Nachteil des Algorithmus, da eine Berechnung irgendeiner Lösung nicht ohne Berechnung des Kerns möglich ist. Dieses Problem behandelt Savenkov im Rahmen einer Diplomarbeit [20], sowie Pichler und Savenkov in [18] und stellen einen verbesserten FindCore Algorithmus vor, welchen als FindCore^{E} bezeichnet wird. Dessen Vorteil ist es, dass EGDs direkt unterstützt werden und somit zuerst die kanonische Lösung und danach optional die Berechnung des Kerns durchgeführt werden kann.

In [3] setzen sich Fagin, Kolaitis, Miller und Popa damit auseinander, wann Abbildungen gleichwertig sind und gehen einen Schritt in Richtung der Optimierung von Abhängigkeiten, doch ist dies erst der erste Schritt Schemenabbildungen (*schema mappings*) in einfachere, aber gleichwertige Schemenabbildungen zu verwandeln.

1.2. Zusammenfassung der Resultate

Der verbesserte FindCore Algorithmus (wie auch der ursprüngliche FindCore Algorithmus) ist in der Lage den Kern in polynomieller Zeit zu berechnen, er besitzt also einen konstanten Exponenten. Wenn jedoch die Anzahl der Datensätze ansteigt wird die Berechnung dennoch sehr zeitaufwändig und ist dann praktisch nicht mehr durchführbar.

Um dieses Problem zu lösen wurde von Pichler und Savenkov in [17] ein neuer Ansatz vorgestellt, welcher den Kern direkt berechnet. Dazu werden die Quelle-zu-Ziel Abhängigkeiten (EGDs werden nicht unterstützt) so abgewandelt, dass ein veränderter Chase in der Lage ist aus diesen direkt den Kern zu berechnen. Es entsteht durch das Untersuchen der Abhängigkeiten ein beträchtlicher Aufwand, welcher jedoch

unabhängig von den Daten der Quellinstanz ist und alle Eingebemuster berücksichtig. Im Gegenzug entfällt die aufwändige Kernberechnung, da ein abgewandelter Chase in der Lage ist aufbauend auf den berechneten neuen Abhängigkeiten den Kern direkt zu berechnen.

Das Ziel dieser Diplomarbeit ist die Umsetzung des wesentlichen Teils, der Analyse von Abhängigkeiten, des oben beschriebenen Ansatzes von Pichler und Savenkov. Die Implementierung ist in der Lage einzelne TGDs zu minimieren, als auch das Zusammenspiel aller vorhandenen TGDs zu untersuchen. Dabei wird jede Möglichkeit der angetroffenen Daten betrachtet, welche in einer TGD auftreten können und diese dann mit allen passenden anderen TGDs kombiniert und so eine Vielzahl neuer TGDs geschaffen, die immer weiter spezialisiert sind. Wie man vermuten kann ist diese Berechnung recht aufwendig, jedoch sind in der Regel nur wenige TGDs zu untersuchen und die Berechnung nur einmalig durchzuführen. Um Speicherplatz zu sparen und in weiterer Folge den Chase zu beschleunigen wird für jede Spezialvariante untersucht, ob diese ein besseres Ergebnis (also weniger Tupel im Ziel einfügt) als die Basis-TGD liefert und nur solche Fälle gespeichert, in denen das Ergebnis besser ist.
Diese berechneten Spezialfälle sind die Voraussetzung, in weiterer Folge einen angepassten Chase zu entwickeln, welcher den Kern direkt berechnet.

1.3. Gliederung

In Kapitel 2 wird auf die Grundlagen des Datenaustausches eingegangen. Darauf aufbauend werden in Kapitel 3 einige verschiedene Arten zur Berechnung von Kernen im Überblick aufgezeigt. Kapitel 4 befasst sich mit den theoretischen Grundlagen zum direkten Berechnen der Kerne durch den Chase. Kapitel 5 beschreibt die Implementierung zur Untersuchung des Zusammenspiels von Abhängigkeiten und in Kapitel 6 wird das ganze schließlich zusammengefasst und ein Ausblick auf Optimierungen und Vervollständigungen gegeben.

2. Grundlagen des Datenaustausches

Bei Datenaustausch (*data exchange*) handelt es sich um das Problem, Daten die in einem Quellschema strukturiert sind, in eine Instanz eines Zielschemas zu überführen, welches die Quelldaten so gut wie möglich widerspiegelt.

Dieser Abschnitt behandelt Grundsätzliches zum Datenaustausch und bezieht sich im Wesentlichen auf Artikel von Fagin, Kolaitis, Popa (und Miller) [2, 5] sowie einen Artikel von Kolaitis [15].

2.1. Das Datenaustauschproblem

Ein Schema ist eine endliche Menge $R = \{R_1, ..., R_k\}$ von relationalen Symbolen, welche einen Namen und eine positive ganze Stelligkeit besitzen. Bei einer Instanz von R handelt es sich um eine Funktion, welche relationale Symbole in eine Relation mit derselben Stelligkeit abbildet. Oftmals werden relationale Symbole missbraucht, um die Relation selbst zu zeigen und zwar speziell dann, wenn die Instanz aus dem Zusammenhang klar ist, oder nicht wichtig ist. Ein Tupel aus der Relation wird als Fakt (*fact*) bezeichnet, daher kann auch die Instanz anhand ihrer Tupel identifiziert werden.
Tupel einer Relation können zwei verschiedene Arten von Termen enthalten: Konstanten (Const) und Variablen (Var). Erstere ist die Menge alle Werte, die in der Quellinstanz vorkommen. Zweitere werden auch als benannte Nullen (*labeled nulls*) bezeichnet, wobei zwei Nullen gleich sind, wenn deren Benennung gleich ist. Die Vereinigung $\underline{Var} \cap \underline{Const} = \varnothing$ beider ergibt die Leere Menge.

Beim Datenaustausch kommen zwei Schemata vor, welche zueinander disjunkt sein müssen. Das Quellschema (*source schema*) $\mathbf{S} = \{S_1, ..., S_n\}$ mit den Quell-relationalen Symbolen S_i und das Zielschema (*target schema*) $\mathbf{T} = \{T_1, ..., T_n\}$ mit den Ziel-relationalen Symbolen T_j. Instanzen von **S** werden Quellinstanzen und Instanzen von **T** Zielinstanzen genannt. Wir schreiben $\langle S, T \rangle$ für das Schema $\langle S_1, S_2, ...S_n, T_1, T_2, ...T_m \rangle$

um die beiden als eine Einheit zu betrachten. Wenn *I* eine Instanz von **S** ist und *J* eine Instanz von **T**, dann schreiben wir $\langle I, J \rangle$ für die Instanz *K* über das Schema $\langle S, T \rangle$.

Eine Quelle-zu-Ziel-Abhängigkeit (*source-to-target dependency, STD*) ist eine Abhängigkeit der Form $\forall \vec{x}(\phi_S(\vec{x}) \rightarrow \chi_T(\vec{x}))$. Hierbei ist $\phi_S(\vec{x})$ ein logischer Formalismus mit freien Variablen $\vec{x}$ über dem Quellschema **S** und $\chi_T(\vec{x})$ ist ein logischer Formalismus mit freien Variablen $\vec{x}$ über dem Zielschema **T**, wobei diese beiden Formalismen verschieden sein können. $\vec{x}$ bezeichnet einen Vektor von Variablen $x_1, \ldots x_k$.
Zielabhängigkeiten (*target dependencies*) sind Abhängigkeiten über dem Zielschema **T**. Auch das Quellschema kann Abhängigkeiten enthalten, diese werden jedoch als erfüllt angenommen und daher für den Datenaustausch nicht weiter relevant.

Definition 2.1. Ein Datenaustauschszenario (*data exchange setting*) (**S**, **T**, $\sum_{ST}$, $\sum_{T}$) besteht aus einem Quellschema **S**, einem Zielschema **T**, einer Menge $\sum_{ST}$ von STDs und einer Menge $\sum_{T}$ von Zielabhängigkeiten. Das Datenaustauschproblem, welches mit diesem Szenario zusammenhängt, sieht folgendermaßen aus: Finde für eine gegebene endliche Quellinstanz *I* eine endliche Zielinstanz *J* sodass $\langle I, J \rangle$ $\sum_{ST}$ erfüllt und *J* $\sum_{T}$ erfüllt. Ein derartiges *J* wird Lösung (*solution*) für *I* genannt. Die Menge aller Lösungen wird mit Sol(*I*) bezeichnet.

Diese Arbeit behandelt nur zwei Typen von Abhängigkeiten, während die von Fagin, Kolaitis, Popa und Tan in [6, 7] beschriebenen prädikatenlogischen tuplegenerierende Abhängigkeiten (*second order TGDs*) außen vor bleiben.
Der erste Typ sind tupelgenerierende Abhängigkeiten (*tupel generating dependency TGD*), welche die Form $\forall \vec{x}(\phi_T(\vec{x}) \rightarrow \exists y \psi_T(\vec{x}, \vec{y}))$ besitzen bei denen ϕ sowie ψ jeweils eine Verknüpfung von Atomen sind. Sollte $\vec{x}$ leer sein spricht man von vollen (*full*) TGDs. Der zweite Typ sind die gleichheitsgenerierenden Abhängigkeiten (*equality generating dependencies EGDs*), welche die Form $\forall \vec{x}(\phi_T(\vec{x}) \rightarrow (x_1 = x_2))$ besitzen, wobei x_1 und x_2 Variablen von $\vec{x}$ sind.

In weiterer Folge werden alle Allquantoren am Anfang von TGDs und EGDs weggelassen, Existenzquantoren werden aber weiterhin angeführt.
Die linke Seite einer Abhängigkeit wird Prämisse (*premise*) und die rechte Seite Folgerung (*conclusion*) genannt. Alle Quelle-zu-Ziel-Abhängigkeiten sind TGDs, während Zielabhängigkeiten sowohl TGDs als auch EGDs sein können. Miteinander bilden diese beiden Typen die eingebauten Folgerungsabhängigkeiten (*embedded implicational dependencies*), mit deren Hilfe die wichtigsten Arten Datenbankeinschränkungen ausgedrückt werden können.

Wenn von einem Schema zwei verschiedene Instanzen vorliegen ist es oft notwendig, diese zu vergleichen, wofür auf Homomorphismus beruhende Relationen benutzt werden. All diese Relationen sind in der Lage Konstanten zu erhalten, was bedeutet, dass Konstanten nur auf sich selbst abgebildet werden. Sei *r* eine Relation zwischen den Einflussbereichen (Domänen, *domain*) zweier Instanzen *I* und *J*, welche dom(*I*) auf dom(*J*) abbildet. Dann ist dom(*I*) auch im Einflussbereich von *r* und $r(\mathrm{dom}(I)) \subseteq J$ eine Reichweite (*range*) von *r*.
Bei zwei Instanzen *I* und *J* ist eine Relationen *h* zwischen dom(*I*) und dom(*J*) ein Homomorphismus, wenn für jeden Fakt $S(x_1, \ldots x_n)$ der Relation *S/n* welcher in *I* vorkommt, ein Fakt $S(h(x_1), \ldots, h(x_n))$ in *J* vorhanden ist. Ein Homomorphismus wird als $h: I \rightarrow J$ geschrieben. Ein Homomorphismus, der einen Fakt auf sich selbst abbildet wird Endomorphismus genannt. Besonders interessant ist für uns die Klasse der idempodenten Endomorphismen, welche für einen Endomorphismus *r* folgendermaßen aussieht: $\forall x \in dom(r)\ r(r(x)) = r(x)$. Solch ein Endomorphismus wird Rückziehung (*retraction*) genannt. Wenn es eine zurückgezogene Abbildung einer Instanz K auf eine Unterinstanz $K' \subseteq K$ gibt, so wird das K' eine Zurückstellung von K genannt.

2.2. Universelle Lösung

Es ist durchaus möglich, dass für ein Datenaustauschproblem eine Vielzahl von Lösungen existiert. Daraus ergibt sich die Frage, welche Lösung man am besten auswählen sollte.

Bevor wir zur Definition der universellen Lösung (*universal solution*) sollen hier noch einige wichtige Vorkenntnisse aufgezeigt werden.

Wenn $R = \{R_1, \ldots, R_k\}$ ein Schema und K eine Instanz von **R** mit Werten aus $\underline{\text{Const}} \cup \underline{\text{Var}}$ ist, dann *b*eschreibt $\underline{\text{Var}}(K)$ die Menge der benannten Nullen, welche in der Relation K vorkommen.

Definition 2.2. Seien K_1 und K_2 zwei Instanzen von R mit Werten aus $\underline{\text{Const}} \cup \underline{\text{Var}}$.

1) Homomorphismus h: $K_1 \rightarrow K_2$ ist eine Abbildung von $\underline{\text{Const}} \cup \underline{\text{Var}}(K_1)$ nach $\underline{\text{Const}} \cup \underline{\text{Var}}(K_2)$, sodass
   a) $h(c) = c$, für jedes $c \in \underline{\text{Const}}$;
   b) für jeden Fakt $R_i(t)$ aus K_1 gilt dass $R_i(h(t))$ ein Fakt von K_2 ist. Wobei, wenn $t = (a_1, \ldots a_s)$, gilt dass $h(t) = (h(a_1), \ldots, h(a_s))$.
2) K_1 ist homomorphisch gleichwertig zu K_2, wenn es einen Homomorphismus h: $K_1 \rightarrow K_2$ und einen Homomorphismus h': $K_2 \rightarrow K_1$ gibt.

Vereinfacht ausgedrückt ist ein Homomorphismus zwischen zwei relationalen Strukturen eine Abbildung, welche Fakten erhält. Die obige Definition enthält auch den speziellen Fall von Konstanten, die nur auf sich selbst abgebildet werden können.

Definition 2.3. Eine universelle Lösung J eines Datenaustauschszenarios (**S**, **T**, $\sum_{ST}$, $\sum_T$) ist dann gegeben, wenn I eine Instanz der Quelle ist und J eine Lösung für I ist und für jede Lösung J' ein Homomorphismus h: $J \rightarrow J'$ welcher J auf J' abbildet existiert.

Die universellen Lösungen können also homomorphisch auf jede andere Lösung abgebildet werden, daher kann aus jeder Lösung durch Hinzufügen von Fakten eine universelle Lösung gebildet werden. Daraus ergibt sich, dass universelle Lösungen die generellsten Lösungen im Datenaustausch sind, da sie genauso viel Information beinhalten wie es für den Datenaustausch notwendig ist.

Beispiel 2.1. Wir gehen von einem Szenario aus, indem das Quellschema die relationalen Symbole P, Q und R und das Zielschema das relationale Symbol T besitzt. Beide haben die Attribute a, b, c. Es gibt keine Zielabhängigkeiten und die STDs sowie die Quellinstanz sind folgendermaßen gegeben:

$\sum_{ST}$: $P(a, b, c) \rightarrow \exists Y \exists Z\, T(a, Y, Z)$

$Q(a, b, c) \rightarrow \exists X \exists U\, T(X, b, U)$

$R(a, b, c) \rightarrow \exists V \exists W\, T(V, W, c)$

$I = \{ \; P(a_0, b_0', c_0'),\; Q(a_0'', b_0, c_0''),\; R(a_0''', b_0''', c_0) \}$

Wir sehen, dass $\sum_{ST}$ die Zielinstanz nicht vollständig beschreibt. Beispielsweise ergibt sich aus *P* dass in *T* an der ersten Stelle ein a aus der Quellinstanz stehen muss. Welche Werte *Y* und *Z* annehmen ist jedoch offen. Eine mögliche Lösung für das Problem wäre die folgende:

$$J = \{T(a_0, Y_0, Z_0), T(X_0, b_0, U_0), T(V_0, W_0, c_0)\}$$

Hier stehen X_0, Y_0, … für die sogenannten benannten Nullen, also Werte die nicht bekannt sind. Es können jedoch auch andere Lösungen entstehen, wie etwa die beiden folgenden:

$$J_1 = \{T(a_0, b_0, c_0)\} \qquad J_2 = \{T(a_0, b_0, Z_1), T(V_1, W_1, c_0)\}$$

In J_1 gibt es keine Variablen mehr. Sowohl J_1 als auch J_2 sind weniger generell als *J*, weil sie Annahmen beinhalten, die nicht Teil von $\sum_{ST}$ sind. Daher ist *J* die beste Lösung, weil es genau so viel enthält, wie notwendig ist.

J_1 und J_2 sind nicht universell, da es von beiden keinen Homomorphismus zu *J* gibt. Andererseits gibt es aber von *J* einen Homomorphismus zu den beiden Lösungen und allen anderen möglichen Lösungen wodurch *J* eine universelle Lösung ist.

2.3. Berechnung von universellen Lösungen

Aus der Definition einer universellen Lösung ergibt sich, dass unter Umständen alle Lösungen überprüft werden müssen. Dieses Kapitel befasst sich mit der Berechnung einer universellen Lösung und der Überprüfung, ob es eine solche überhaupt gibt. Zu diesem Zweck wird der Chase verwendet, der, wenn er nicht abbricht, eine universelle Lösung berechnet. Dieser führt Schritt für Schritt Einschränkungen mit einer eingeschränkten Anzahl von Tupel aus.

Bei Quelle-zu-Ziel Abhängigkeiten wird mit einer Instanz der Quellinstanz und der leeren Zielinstanz begonnen. Dann verändert der Chase die Zielinstanz immer wieder, was zu einer voruniversellen Instanz führt.

Der Chase bei Quellabhängigkeiten beginnt mit einer einzelnen Zielinstanz, welche solange aktualisiert wird, bis schließlich die sogenannte kanonische universelle Lösung (*canonical universal solution*) entsteht.

Betrachtet man eine TGD $\phi(\vec{x}) \rightarrow \exists y \psi(\vec{x}, \vec{y})$ bei der es einen Homomorphismus von $\phi(\vec{x})$ zu einer Instanz I gibt, jedoch keinen von einem von diesem erweiterten Homomorphismus $\phi(\vec{x}) \wedge \psi(\vec{x}, \vec{y})$. Es muss daher I mit Fakten erweitert werden, welche zu $\psi(\vec{x}, \vec{z})$ passen, wobei $\vec{z}$ neue Variablen enthält.

Bei einer EGD $\phi(\vec{x}) \rightarrow (x_1 = x_2)$ bei der es einen Homomorphismus h von $\phi(\vec{x})$ zu einer Instanz I bei dem $h(x_1) \neq h(x_2)$. Sollten $h(x_1)$ und $h(x_2)$ verschiedene Konstanten sein, dann beendet der Chase mit einem Fehler. Wenn nur einer der beiden eine Konstante ist, wird die benannte Null überall durch die Konstante ersetzt. Sollten beide benannte Nullen sein, wird eine der beiden Variablen überall durch die andere ersetzt.

Wenn man nun die Menge aller TGDs und EGDs durchlaufen lässt und keine davon fehlschlägt, war der Chase erfolgreich.

Beispiel 2.2.: Das folgende Beispiel führt den Chase mit den Abhängigkeiten und der Instanz aus Beispiel 2.1. durch.

Zu Beginn benötigen wir die leere Quellinstanz. Wir durchlaufen dann Schritt für Schritt die Tupel der Quellinstanz und suchen die Regeln, welche dazu passen und fügen auf diese Weise Tupel in der Zielinstanz ein. Die nachfolgende Tabelle zeigt in welchem Schritt welches Tupel betrachtet wird, welche Regeln dann ausgeführt werden und wie danach die Zielinstanz aussieht. Prinzipiell kann die Reihenfolge, in der die Tupel betrachtet werden, beliebig sein und auch für die Regeln gibt es keine vorher bestimmte Reihenfolge. In diesem Beispiel gibt es für jedes Tupel genau eine Regel, welche zur Ausführung kommt. Variablen werden wie schon im vorherigen Beispiel groß geschrieben, Werte die aus der Quelle stammen werden klein geschrieben.

Chase Schritt	Tupel der Quellinstanz	Abhängigkeit aus $\sum_{ST}$	Zielinstanz
0			∅

1	$P(a_0, b_0', c_0')$	P(a, b, c) → $\exists Y \exists Z$ T(a, Y, Z)	T(a_0, Y_0, Z_0)
2	$Q(a_0'', b_0, c_0'')$	Q(a, b, c) → $\exists X \exists U$ T(X, b, U)	T(a_0, Y_0, Z_0), T(X_0, b_0, U_0)
3	$R(a_0''', b_0''', c_0)$	R(a, b, c) → $\exists V \exists W$ T(V, W, c)	T(a_0, Y_0, Z_0), T(X_0, b_0, U_0), T(V_0, W_0, c_0)

Wie man sieht entspricht die Zielinstanz nach dem letzten Chase Schritt der Lösung J aus Beispiel 2.1. Weil es keine Zielabhängigkeiten gibt, ist diese Lösung bereits die universelle Lösung.

2.4. Berechnung von universellen Lösungen mit schwacher Azyklichkeit

Beispiel 2.3. Im Folgenden aus [2] stammendem Beispiel, besteht das Quellschema aus der Relation *DeptEmp*(*dptId*, *mgrName*, *eId*), welche Abteilungen mit deren Managern und Angestellten auflistet. Das Zielschema besitzt die Relation *Dept*(*dptId*, *mgrId*, *mgrName*) welche Abteilungen mit deren Verantwortlichen auflistet und die Relation *Emp*(*eId*, *dptId*) welche Angestellte einer Abteilung zuordnet. Die Abhängigkeiten sind die folgenden:

$$\Sigma_{ST} = \{DeptEmp(d,n,e) \rightarrow \exists M.Dept(d,M,n) \wedge Emp(e,d)\}$$

$$\Sigma_T = \{Dept(d,m,n) \rightarrow \exists D.Emp(m,D),$$
$$Emp(e,d) \rightarrow \exists M \exists N.Dept(d,M,N)$$

Angenommen in der Quellinstanz gibt es folgendes Tupel in *DeptEmp*(*F&E*, *Maria*, *E003*). Wenn der Chase $\sum_{ST}$ ausführt entsteht die Zielinstanz

J_1 = {*Dept*(*F&E*, *M*, *Maria*), *Emp*(*E003*, *F&E*)}

bei der *M* eine benannte Null ist, welche für die unbekannte ManagerId von Maria steht. J_1 erfüllt aber $\sum_T$ nicht, weshalb der Chase fortfährt und der Zielinstanz *Emp*(*M*, *D*) hinzufügt, bei dem *D* für die unbekannte Abteilung von Maria steht. Nun wird die zweite Abhängigkeit in $\sum_T$ wirksam, dann wieder die erste usw. wodurch der Chase

endlos weiterläuft. Für das Beispiel gibt es zwar endliche Lösungen, welche aber aufgrund von getroffenen Annahmen nicht universell sind.

Das vorherige Beispiel zeigt, dass aufgrund der Tatsache, dass TGDs immer neue Tupel einfügen, was wiederum einen Chase-Schritt notwendig machen kann es möglich ist, dass der Chase nicht terminiert. Daher wird in der Folge das Konzept der schwachen Azyklichkeit *(weakly acyclic*) betrachtet, welche es dem Chase erlaub, in polynomieller Zeit zu einem Ergebnis zu kommt.

Eine Menge von TGDs ist dann schwach Azyklisch, wenn der Chase benannte Nullen nicht kaskadiert erstellt. Formell wurde sie von Fagin durch Abhängigkeitsgraphen definiert. In diesem gerichteten Graphen finden sich Knoten, welcher aus einem relationalen Symbol des Schemas (in einer Datenbank der Tabellenname) und einem Attribut dieses Schemas (in einer Datenbank der Spaltenname) besteht. Solch ein Knoten wird Position (R, A) genannt. In weiterer Folge werden Kanten nach folgendem Schema dem Graphen hinzugefügt: Für jede TGD $\phi(\vec{x}) \rightarrow \exists \vec{y} \psi(\vec{x}, \vec{y})$ aus der Menge $\sum$ und für jedes x_i aus $\vec{x}$, welches in ψ vorkommt: Für jedes Vorkommen von x_i aus ϕ an der Position (R, A_i):

1) Füge für jedes Auftreten von x_i aus ψ an der Position (S, B_j) eine Kante (R, A_i) → (S, B_j) hinzu, sofern eine solche noch nicht existiert.
2) Füge zusätzlich für jede mit der existenziell quantifizierte Variable *y* und für jedes Vorkommen von *y* in ψ an der Position (T, C_k) eine spezielle Kante (spezial edge) $(R, A_i) \xrightarrow{*} (T, C_k)$ hinzu, sofern eine solche noch nicht existiert.

Es ist möglich, dass zwei Kanten zwischen 2 Knoten existieren, sofern genau eine der beiden eine spezielle Kante ist. Der Graph ist schwach Azyklisch, wenn kein Kreislauf durch spezielle Kanten existiert.

Der erste Punkt ist dafür zuständig, Werte während des Chase von einer Position auf eine andere abzubilden. Der zweite Teil kümmert sich um die Erstellung benannter Nullen. Falls es einen Kreislauf mit speziellen Kanten geben sollte, würde die Erstellung einer Variablen an einer Position durch den Chase unter Umständen an derselben Stelle wieder eine benannte Null erstellt werden und so der Chase niemals enden. Zyklen sind jedoch im Graph nicht generell verboten, sondern eben nur durch

spezielle Kanten. Sollten keine existenziell quantifizierten Variablen in einer Menge von TGDs existieren spricht man von vollen TGDs (*full tgds*), welche ein Spezialfall der schwachen Azyklischkeit sind.

Beispiel 2.4. Der Graph in Abbildung 1 veranschaulicht das vorherige Beispiel. Wie aus diesem ersichtlich ist, existiert ein Zyklus durch spezielle Kanten (der Graph ist also nicht schwach azyklisch), wodurch der Chase nicht terminiert.

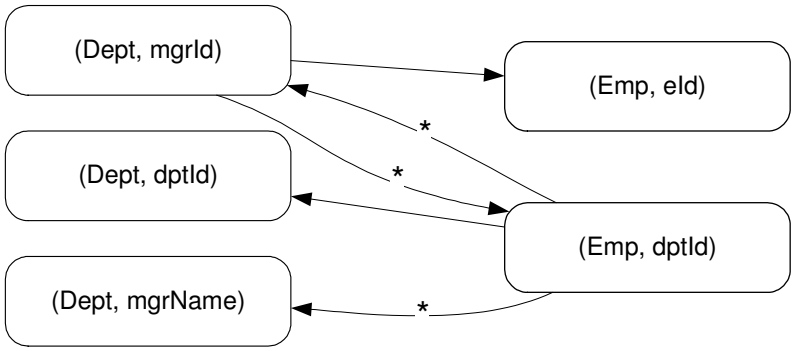


Abbildung 1: Graph für Beispiel 2.3.

Wir ändern das Beispiel in der Folge etwas ab, indem wir annehmen, dass jeder Manager einer Abteilung auch in dieser Beschäftigt ist. Wir ersetzen daher Σ_T aus vorherigem Beispiel durch die folgende:

$\Sigma_T^{'} = \{Dept(d,m,n) \rightarrow \exists D.Emp(m,d),$

$Emp(e,d) \rightarrow \exists M \exists N.Dept(d,M,N)\}$

Es entsteht durch diese Änderung der Graph aus Abbildung 2. Wie man sieht existiert hier kein Zyklus durch spezielle Kanten, die Menge $\Sigma_T^{'}$ ist daher schwach azyklisch.

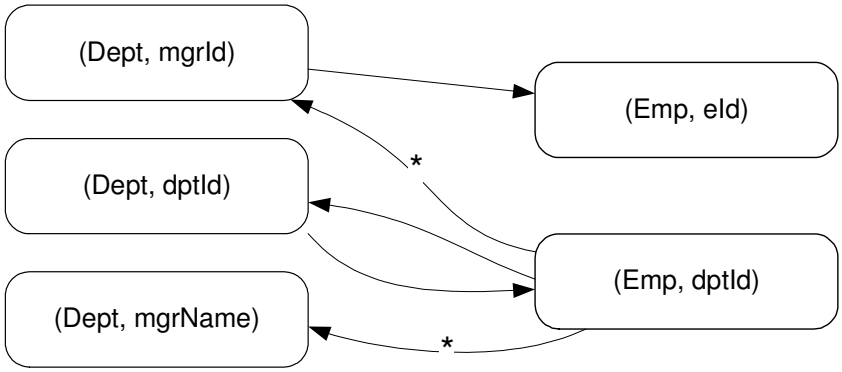


Abbildung 2: Graph für geändertes Beispiel 2.4.

Als Rang (*rank*) eines Knotens wird die maximale Anzahl an eingehenden speziellen Kanten bezeichnet.

Fagin beweist in [2], dass für eine Menge von schwach azyklischen TGDs und EGDs und eine Instanz K ein Polynom der Größe K existiert, welches die obere Schranke für die Anzahl von Chase Schritten darstellt. Es sind daher $O(|K|^a)$ Schritte bzw. die Zeit $O(|K|^b)$ zum Berechnen der universellen Lösung notwendig. Somit kann in polynomieller Zeit sowohl geprüft werden, ob es eine Lösung gibt, als auch diese berechnet werden.

2.5. Definition des Kerns

Nachdem nun universelle Lösungen berechnet werden können, taucht das Problem auf, dass mehrere universelle nicht isomorphe Lösungen existieren können. Diese besitzen zueinander einen Homomorphismus, können aber verschiedene Größen haben. Intuitiv ist der Kern die kleinste eindeutige universelle Lösung.

Hell und Nešetřil gehen in [14] auf die Kerne von Graphen ein und führen die Struktur (*structure*) ein, welche der Instanz sehr ähnlich ist. Formal ist eine Struktur A (über einem Schema $R = \langle R_1, ..., R_k \rangle$) eine Sequenz $\langle A, R_1, ..., R_k \rangle$ bei der A eine nichtleere Menge ist, welche Universum (*universe*) genannt wird und R_i relationale Symbole sind, welche Relationen über A darstellen. In Strukturen gibt es keine speziellen

Elemente (wie z.B. Konstanten), welche durch den Homomorphismus bewahrt werden müssen.

Definition 2.4. [5]: Eine Unterstruktur *C* einer Struktur *A* wird Kern von *A* genannt, wenn ein Homomorphismus von *A* nach *C* existiert, aber keiner von *A* zu einer korrekten Unterstruktur von *C*. Eine Struktur *C* wird Kern genannt, wenn sie ein Kern von sich selbst ist, was dann zutrifft, wenn es keinen Homomorphismus zu einer Unterstruktur von *C* gibt.

Aus der Graphentheorie stammen auch die folgenden Aussagen, welche in [14] bewiesen wurden:

- Jede endliche Struktur besitzt einen Kern und alle Kerne derselben endlichen Struktur sind isomorph.
- Jede endliche Struktur ist homomorphisch gleich mit ihrem Kern. Als Konsequenz sind zwei endliche Strukturen homomorhpisch gleich wenn deren Kerne Isomorph sind.

In ähnlicher Weise werden Subinstanz eines Kernes und einer Instanz definieren. Die Instanz wird der zugehörigen Struktur zugeordnet und deren Universum ist die aktive Domäne der Instanz. Wir müssen einen Teil der Elemente des Universums als Konstante identifizieren. Ein Homomorphismus einer Konstanten muss diese also auf sich selbst abbilden ($h(c) = c$). Die Aussagen über die Kerne von Strukturen sind somit auch für Kerne über Instanzen anwendbar.
Der folgende Satz, welcher in [5] bewiesen wird, begründet die Verwendung von Kernen im Datenaustausch.

Satz 2.1. [5]: Sei ($\mathbf{S}$, $\mathbf{T}$, $\sum_{ST}$, $\sum_{T}$) ein Datenaustauschszenario bei dem $\sum_{ST}$ eine Menge von TGDs und $\sum_{T}$ eine Menge von TGDs und EGDs ist. Wenn *I* eine Quellinstanz und *J* eine Lösung ist, dann ist der Kern von *J* (core(*J*)) eine Lösung für *I*. Analog gilt, dass wenn *J* eine universelle Lösung für *I* ist, dass auch core(*J*) eine universelle Lösung ist.

3. Berechnung des Kerns

Im Gegensatz zur Berechnung des Kerns einer beliebigen Instanz kann der Kern einer universellen Lösung in polynomieller Zeit berechnet werden. Die Berechnung des Kerns wird auch Kernberechnung (*core computation*) genannt.

Fagin, Kolaitis und Popa haben in [1] gezeigt, dass die Kernberechnung für den Fall dass $\sum_T$ nur EGDs enthält in polynomieller Zeit möglich ist. Gottlob hat in [10] die polynomielle Berechenbarkeit auch für Fälle bewiesen, in denen $\sum_T$ auch volle TGDs enthält. In [10, 12] wurde eine Methode vorgestellt, in der $\sum_T$ auch schwach azyklische TGDs enthalten darf. In [11] werden weitere Methoden zur Berechnung von Kernen untersucht.

In der Folge werden die beiden Algorithmen aus [1] vorgestellt bei denen $\sum_{ST}$ eine beliebige Menge von TGDs und $\sum_T$ eine Menge beliebiger EGDs ist. Wir gehen davon aus, dass wir ein Datenaustauschszenario (S, T, $\sum_{ST}$, $\sum_T$) gegeben haben. Aus der gegebenen Quellinstanz *I* entsteht die Zielinstanz *J* durch den Chase von $\langle I, \varnothing \rangle$ mit $\sum_{ST}$, welche voruniverselle Lösung (*pre-universal solution*) genannt wird. Aus dieser Lösung wird dann in weiterer Folge der Kern berechnet.

3.1. Greedy Algorithmus

Der Greedy Algorithmus, dessen Korrektheit in [5] bewiesen wird, beginnt mit der Quellinstanz und führt den Chase mit $\sum_{ST}$ durch, wodurch die voruniverselle Lösung *J* entsteht. Danach wird der Chase mit *J* und $\sum_T$ durchgeführt. Falls der Chase nicht erfolgreich ist wird ein Fehler zurückgeworfen und die Berechnung abgebrochen, andernfalls entsteht die kanonische universelle Lösung *J'*. Nun wird *J'* nach *J** kopiert und danach solange Tupel aus *J** entfernt, solange *I* und die neu entstandene Lösung die TGDs in $\sum_{ST}$ erfüllen. Wenn kein Entfernen mehr möglich ist, wird *J** als Ergebnis zurückgegeben.

Der Algorithmus mit der Laufzeit $O(n^{\text{Größe von } I})$ arbeitet nur korrekt, wenn $\sum_T$ keine TGDs enthält. Der größte Nachteil dieses Algorithmus ist, dass die Quellinstanz während der ganzen Laufzeit verfügbar sein muss.

3.2. Blocks Algorithmus ohne Zielabhängigkeiten

Dieser Nachteil wird durch den ebenfalls in [5] vorgestellten Blocks Algorithmus behoben. Um auf diesen einzugehen sind jedoch zuvor einige Definitionen notwendig.

Definition 3.1.: Der Gaifman Graph der Nullen einer Instanz *K* ist ein ungerichteter Graph, dessen Knoten aus Variablen (benannten Nullen) von *K* bestehen. Wenn zwei verschiedene Variablen in einem Tupel von *K* vorkommen, so ist zwischen diesen eine Kante zu finden. Ein Block ist eine verbundene Komponente des Graphen und jede Variable kommt in genau einem Block vor. Die Blockgröße der Instanz *K* ist die maximale Anzahl benannter Nullen aller Blöcke.
Jeder Homomorphismus von *K* nach *K*' kann als Vereinigung aller Homomorphismen auf die einzelnen Blöcke von *K* dargestellt werden. Die einzelnen Homomorphismen können unabhängig voneinander gebildet werden, da die einzelnen Blöcke keine gemeinsamen Variablen besitzen und sich die Blöcke nicht ändern.

Das folgende Theorem schätzt den Aufwand des Blocks Algorithmus grob ab.
Satz 3.1.: Seien *A* und *B* Instanzen und die Blockgröße von *A* sei kleiner gleich *c*. Dann kann sowohl die Überprüfung ob es einen Homomorphismus $h: A \rightarrow B$ gibt als auch die Berechnung von *h* mit der Laufzeit $O(|A| \bullet |B|^c)$ erfolgen.
Begründung: Die wichtigste Überlegung ist, dass wir anstatt der Suche nach einem Homomorphismus von $A \rightarrow B$ einen Homomorphismus in den einzelnen Blöcke suchen. *A* hat natürlich $\leq |A|$ Blöcke, welche maximal *c* Variablen enthalten. Daher müssen von einem Block von *A* maximal $|B|^c$ mögliche Abbildungen nach *B* betrachtet werden.

Nun wird der Blocks Algorithmus für $\sum_T = \varnothing$ vorgestellt.

Zuerst wird die kanonische universelle Lösung U aus $\langle I, \varnothing \rangle$ durch Chase mit $\sum_{ST}$ berechnet und U in seine Blöcke aufgeteilt. Nun versucht der Algorithmus in jedem Schritt eine Variable durch einen Endomorphismus aus dem Kernkandidaten U' zu eliminieren. Daher muss jeder Endomorphismus mindestens eine Variable eliminieren. Wenn y so eine Variable ist, dann kann ein gewünschter Endomorphismus als eine Vereinigung von Homomorphismen $h_i : B_i^{U'} \to U'$ berechnet werden, wobei h_j eine Gleichheitsabbildung ist, wenn $y \notin B_j^{U'}$. Daher ist es, um eine Variable loszuwerden, nur notwendig im Block von j nach einer Abbildung zu suchen.

Die Blöcke müssen nur einmal berechnet werden, wie im folgenden Satz erklärt wird:

Satz 3.2.: K und K' seien zwei Instanzen, bei denen die benannten Nullen von K' eine Untermenge der Variablen von K sind, daher ist $\underline{\mathrm{Var}}(K') \subseteq \underline{\mathrm{Var}}(K)$. h ist ein Endomorphismus von K' und B ein Block mit benannten Nullen von K. Man sagt, dass h K-lokal für B ist, wenn $h(x) = x$ und zwar immer wenn $x \notin B$. h ist K-lokal für K wenn h K-lokal für den Block B von K ist.

Für jeden Block $B^{U'}$ aus $U' \subseteq U$ existiert ein Block B^U aus U, sodass $B^{U'} \subseteq B^U$. Um einen Endomorphismus zu erstellen, der eine Variable y eliminiert reicht es aus, nur Variablen aus U' zu betrachtet welche auf B_y^U aus U beschränkt ist:

Satz 3.3.: Wir gehen von einem Datenaustauschszenario aus, indem $\sum_{ST}$ eine Menge von TGDs ist und $\sum_T = \varnothing$. Sei J' eine Unterinstanz der kanonischen universellen Lösung J. Wenn es einen geeigneten Endomorphismus von J' gibt, dann existiert auch ein geeigneter J-lokaler Endomorphismus von J'.

3.3. Blocks Algorithmus mit Zielabhängigkeiten

Der Vorteil der vorherigen Lösung war, dass STDs die Blockgröße nicht verändern. Dies ist bei Zielabhängigkeiten nicht mehr der Fall, weshalb in diesem Abschnitt der vorherige Algorithmus so erweitert wird, dass auch beim Vorhandensein von Zielabhängigkeiten der Kern gefunden werden kann.

Das größte Problem stellen die EGDs dar, weil diese Variablen ersetzen und daher Probleme beim Finden einer Blockgröße die unabhängig der in ihr enthaltenen Fakten ist. Fagin et al. zeigen aber, dass dieses Problem bei kanonischen voruniversellen Lösungen (nachdem Σ_{ST} auf eine Instanz angewandt wurde) auf die EGDs der Zielabhängigkeiten angewandt werden nicht auftreten. Wenn daher die Zielabhängigkeiten nur EGDs enthalten kann auf die fixe Blockgröße der voruniversellen Lösung aufgebaut werden. In der folgenden Definition wird auf diesen Umstand der starren (*rigid*) Variablen eingegangen.

Definition 3.2.: Es sei K eine Instanz, deren Elemente Konstante und Nullen sind und y ein Element aus K ist. Man sagt das y starr ist, wenn $h(y) = y$ für jeden Homomorphismus h auf K. Daher sind auch alle Konstanten, die in K vorkommen starr.

Diese Definition impliziert, dass sich Variablen im Bezug auf den Homomorphismus wie Konstante verhalten, also auf sich selbst abgebildet werden und daher beim Berechnen der Blöcke nicht weiter berücksichtigt werden. Ein Block wird nicht-starr genannt, wenn dieser nur unter Berücksichtigung von nicht-starren Variablen berechnet wurde.

Hilfssatz 3.1. (*Rigidity Lemma*): Wir gehen von einem Datenaustauschszenario bei dem Σ_{ST} eine Menge von TGDs und Σ_T eine Menge von EGDs ist. Sei J eine kanonische voruniverselle Lösung und J' das Ergebnis vom Chase von J mit Σ_T. Es seien x und y Nullen von J sodass $x \sim y$ und dass $[x]$ eine nicht-starre Null von J' ist. Dann sind x und y im selben Block von J.
In [17] wird der Satz erweitert, sodass Σ_T auch TGDs enthalten darf sowie dessen Gültigkeit in [5] bzw. [17] bewiesen.

Der Satz zeigt, dass wenn eine EGD zwei Variablen vereinheitlicht, welche zu verschiedenen Blöcken der voruniversellen Lösung J gehörten, die entstehende Variable starr ist. Aufgrund dieser Tatsache ist es möglich, den Blocks-Algorithmus abzuwandeln, um mit EGDs in den Zielabhängigkeiten umzugehen, was der folgende Satz aus [5] verdeutlicht.

Satz 3.4.: Wir gehen von einem Datenaustauschszenario bei dem Σ_{ST} eine Menge von TGDs und Σ_T eine Menge von EGDs ist. Es sei J eine kanonische voruniverselle Lösung und J" ein endomorphe Abbildung der kanonischen Lösung J'. Wenn es einen nicht injektiven Endomoprphismus von J" gibt, dann existiert auch ein nicht injektiver J-lokaler Endomorphismus von J".

3.4. FindCore Algorithmus

Dieser Abschnitt befasst sich mit dem von Gottlob und Nash in [12] vorgestellten FindCore Algorithmus, welcher einige neue Ansätze zur Berechnung des Kernes vorstellt.
Der unten stehende Pseudocode zeigt den FindCore Algorithmus und wird danach erklärt.

FindCore Algorithmus

```
Eingabe: Quellinstanz S
Ausgabe: Kern einer universellen Lösung von S
(1)   Chase (S, ∅) mit Σ_ST um (S, T) := (S, ∅)^Σ_ST zu erhalten
(2)   Berechne Σ̄_T aus Σ_T
(3)   Chase T mit Σ̄_T (unter Verwendung der guten Chase Reihenfolge) um U := T^Σ_T zu bekommen
(4)   Für alle x ∈ var(U), y ∈ dom(U), x ≠ y
      {
(5)         Berechne T_xy (siehe Hilfssatz 3.7.)
(6)         Suche ein h: T_xy → U sodass h(x) = h(y)
(7)         Wenn es solch ein h gibt
            {
(8)               Erweitere h zu einem Endomorphismus h' auf U (siehe Satz 3.8.)
(9)               Transformiere h' in eine Rückstellung r (siehe Satz 3.6.)
(10)              Setze U := r(U)
            }
(11)  }
      Retourniere U
```

Der Algorithmus startet bei (1) mit dem Berechnen der voruniversellen Lösung, fährt dann aber nicht wie bei den bisherigen Ansätzen mit dem Berechnen der kanonischen universellen Lösung durch den Chase von T mit Σ_T fort, sondern ersetzt die in Σ_T

vorhandenen EGDs durch TGDs. Dazu werden in (2) die TGDs und EGDs aus $\sum_T$ über der Signatur τ in eine Menge $\overline{\Sigma}_T$ bestehend aus TGDs über eine Signatur und $\tau \cap E$ umgewandelt. E wird Kodierqualität (*encoding quality*) genannt und ist eine binäre Relation welche in τ nicht vorhanden ist. Die Umwandlung erfolgt folgendermaßen:

1. Ersetze alle Gleichheiten $x = y$ mit $E(x, y)$, um jede EGD in eine TGD umzuwandeln
2. Füge die folgenden Gleichheitsbedingungen ein:
   - $E(x, y) \rightarrow E(y, x)$
   - $E(x, y) \wedge E(y, z) \rightarrow E(x, z)$
   - $R(x_1, \ldots, x_k) \rightarrow E(x_i, x_i)$
     für jedes $R \in \tau$ und $i \in \{1, 2, \ldots, k\}$ wobei k die Stelligkeit von R ist.
3. Füge die folgenden Konsistenzbedingungen ein:
   - $R(x_1, \ldots, x_k) \wedge E(x_i, y) \rightarrow R(x_1, \ldots, y, \ldots, x_k)$
     für jedes $R \in \tau$ und $i \in \{1, 2, \ldots, k\}$

Selbst wenn $\sum_T$ schwach azyklisch war, muss dies für $\overline{\Sigma}_T$ nicht unbedingt zutreffen, weshalb in [12] die sogenannte gute Chase Reihenfolge (*nice chase order*) eingeführt wird, welche sicherstellt, dass der Chase für $\overline{\Sigma}_T$ terminiert. Die Lösung U nach dem Schritt (3) welche durch Chase mit $\overline{\Sigma}_T$ berechnet wurde ist keine universelle Lösung, weil im Allgemeinen die EGDs von $\sum_T$ nicht erfüllt sind. Diese werden erst als Teil der Kernberechnung betrachtet. U erfüllt aber natürlich die Abhängigkeiten in $\sum_{ST}$ und $\overline{\Sigma}_T$.
Der Kern wird vom FindCore Algorithmus durch iteratives Berechnen einer Folge von verschachtelten Rückstellungen. Der Grund dafür ist, dass diese Rückstellung einige gute Eigenschaften besitzt: Eingebettete Abhängigkeiten sind geschlossen unter der Rückstellung und jeder gültige Endomorphismus kann effizient in eine Rückstellung überführt werden, was die beiden folgenden Sätze aus [12] belegen:

Satz 3.5.: Sei r: $A \rightarrow A$ eine Rückstellung mit $B = r(A)$ und sei $\sum$ eine Menge von eingebetteten Abhängigkeiten. Wenn $A \models \sum$ gilt, so gilt auch $B \models \sum$.
Die Berechnung erfolgt laut folgendem Satz:

Satz 3.6.: Sei h ein Endomorphismus h: $A \rightarrow A$ sodass $h(x) = h(y)$ für $x, y \in$ dom(A), so gibt es eine gültige Rückstellung r auf A sodass $r(x) = r(y)$ ist. Solch eine Rückstellung kann mit der Laufzeit O(|dom(A)|²) gefunden werden.

Wenn die Suche nach einem Endomorphismus h aus U in den Schritten (4) - (8) erfolgreich war, wird in Schritt (9) mithilfe des Satzes 3.5. in eine Rückstellung r umgewandelt und schließlich in Schritt (10) U durch $r(U)$ ersetzt. Der Satz 3.6. stellt sicher, dass $\sum_{ST}$ und $\overline{\Sigma}_T$ weiterhin erfüllt sind.

In jedem Durchlauf versucht der FindCore Algorithmus in den Schritten (5) – (8) einen gültigen Endomorphismus für die gerade gültige Instanz U zu finden. Bei einer gegebenen Variable x und einer anderen benannten Null y aus der Domäne wird versucht, einen Endomorphismus zu finden, welcher x und y gleichsetzt. Die dafür benötigte Zeit dafür kann exponentiell zur Blockgröße sein. Aus diesem Grund ist die grundlegende Idee des FindCore Algorithmus, die Suche nach einem gültigen Endomorphismus in zwei Teile aufzuteilen:
Für die gegebenen benannten Nullen x und y existiert eine Instanz T_{xy}, deren Blockgröße durch eine Konstante, welche nur von $\sum_{ST} \cup \sum_T$ abhängt, begrenzt ist. Daher wird im ersten Schritt nach einem Homomorphismus h: $T_{xy} \rightarrow U$ mit $h(x) = h(y)$ gesucht. Danach wird h zu einem Homomorphismus h: $U \rightarrow U$ erweitert, bei dem $h(x) = h(y)$ immer noch gilt. Daher ist h immer noch nicht-injektiv und solange wir nur endliche Instanzen betrachten, auch ein gültiger Endomorphismus.
Die Eigenschaften von T_{xy} und das Vorhandensein einer Erweiterung h' von h werden durch die folgenden Sätze aufgezeigt. Die Beweise dafür sind in [12] zu finden:

Hilfssatz 3.7.: Für jede schwach azklyische Menge $\sum$ von TGDs, Instanz T und $x, y \in$ dom(T^{Σ}) existieren Konstanten b und c. Diese hängen nur von $\sum$ und einer Instanz T_{xy} ab und erfüllen folgendes:

1. $x, y \in$ dom(T_{xy})
2. $T \subseteq T_{xy} \subseteq T^{\Sigma}$
3. dom(T_{xy}) ist geschlossen unter Eltern und Geschwistern und
4. $|\text{dom}(T_{xy})| \leq |\text{dom}(T)| + b$

Weiters kann T_{xy} mit der Laufzeit O(|dom(T)^c) berechnet werden.

Satz 3.8.: (Hebung, *lifting*): Sei T^{Σ} eine universelle Lösung eines Datenaustauschszenarios, welches durch den Chase einer voruniversellen Instanz von T mit der schwach azyklischen Menge $\sum$ von Ziel-TGDs entstanden ist. Wenn B und W Instanzen sind für die folgendes gilt

1. $B \models \sum$
2. $T \subseteq W \subseteq T^{\Sigma}$ und
3. dom(W) ist geschlossen unter den allen Vorfahren und den Geschwistern

dann kann jeder Homomorphismus h: $W \rightarrow B$ mit der Laufzeit O(|dom(T)^b|) zu einem Homomorphismus h': $T^{\Sigma} \rightarrow B$ erweitert werden, bei dem b nur von $\sum$ abhängt.

In den vorherigen Sätzen wurden die Begriffe Geschwister, Eltern und Vorfahren gebraucht, welche jetzt beschrieben werden sollen. Wir gehen von einem Chase auf eine voruniverselle Lösung der Instanz T mit dem Zielabhängigkeiten Σ_T aus und nehmen an, dass $\vec{y}$ ein Tupel mit Variablen ist, das durch die TGD $\phi(\vec{x}) \rightarrow \psi(\vec{x}, \vec{y})$ aus Σ_T bei Anwendung auf $\vec{a}$ erstellt wurde. In diesem Fall sind die Elemente aus $\vec{y}$ untereinander Geschwister. Alle Variablen aus $\vec{a}$ sind Eltern jedes Elementes aus $\vec{y}$ und die Vorfahren sind die transitive Hülle der Elternrelation.

Zusammenfassend ist zu sagen, dass das Hilfsprädikat E genutzt wird, um EGDs zu simulieren. Daher bricht der Algorithmus ab, wenn der Schritt (3) ein Faktum $E(a_i, a_j)$ bei dem $a_i \neq a_j$ generiert und damit das Datenaustauschszenario keine Lösung hat. Andernfalls wird in den Schritten (4) – (10) dom(U) immer weiter verkleinert, bis der Kern schließlich mit der nicht weiter verkleinerbaren dom(U) gefunden ist. In [12] wird bewiesen, dass solch eine von FindCore berechnete Instanz U alle EGDs erfüllt. Daher stellt U ohne die Hilfsfakten mit dem führenden E den Kern einer universellen Lösung dar und es ergibt sich der folgende Satz aus [12]:

Satz 3.9.: Sei $(\mathbf{S}, \mathbf{T}, \Sigma_{ST}, \Sigma_T)$ ein Datenaustauschszenario mit den Quelle-zu-Ziel Abhängigkeiten Σ_{ST} und den Zielabhängigkeiten Σ_T. Weiters sei S eine Grundinstanz

eines Zielschemas **S**. Wenn das Szenario eine Lösung hat, berechnet der FindCore Algorithmus den Kern einer kanonischen Universellen Lösung in der Zeit $O(|\text{dom}(S)|^b)$ korrekt. Die Konstante b hängt dabei nur von $\Sigma_{ST} \cap \Sigma_T$ ab.

3.5. Erweiterter FindCore Algorithmus

Der im vorherige Kapitel beschriebene FindCore Algorithmus hat den Nachteil, dass er EGDs durch TGDs simulieren muss und daher keine universelle Lösung ohne Berechnung des Kerns erstellen kann. Diesen Nachteil beseitigt der von Savenkov und Pichler in [4, 14] beschriebene erweiterte FindCore (FindCore^E) Algorithmus, welcher hier aufbauend auf diesen Arbeiten im Überblick vorgestellt werden soll.
Der entscheidende Punkt ist des Algorithmus ist, dass die EGDs direkt behandelt werden und somit zuerst die kanonische universelle Lösung U berechnet wird und dann optional die Berechnung des Kerns erfolgen kann. Oberflächlich betrachtet arbeitet der FindCore^E Algorithmus gleich wie der bereits beschriebene FindCore Algorithmus, jedoch müssen wegen der direkten Behandlung der EGDs einige Anpassungen vorgenommen werden:

- T_{xy} muss anders definiert werden, weil beim Berechnen dieses im vorherigen Abschnitt nur einen kleinen Teil des Zielchases berücksichtigt wurde und daher eine Unterinstanz von U erstellt wurde. Weil aber nun EGDs vorhanden sein dürfen ist es möglich dass die Domäne U nicht mehr alle Elemente beinhaltet, die in T (oder irgendwelchen Zwischenschritten des Chases) vorhanden waren.
- Der Aufwand zum Berechnen des Homomorphismus h: $T_{xy} \rightarrow U$ hängt von der Blockgröße von T_{xy} und damit von der Blockgröße der voruniversellen Lösung T ab. Variablen werden durch EGDs eliminiert, wodurch der positive Effekt verkleinerter Blöcke auftritt. Andererseits können EGDs aber auch verschiedene Blöcke von T zusammenfügen, was den Nachteil hat, dass, ohne weitere Maßnahmen, die Suchen nach h nicht mehr polynomiell berechenbar ist.
- Weil wir T_{xy} anders definieren müssen, ist es auch notwendig, die Hebung (*lifting*) von h: $T_{xy} \rightarrow U$ zu einem passenden Endomorphismus h': $U \rightarrow U$ abzuändern, was einen komplett neuen Ansatz erfordert.

All diese Aspekte werden in [20] und [14] ausführlich behandelt, sie sind jedoch für diese Arbeit nicht wesentlich, weshalb diese hier nicht weiter ausgeführt werden sollen, jedoch zum Abschluss noch der essenzielle Satz erwähnt werden soll, welcher die Richtigkeit des Algorithmus und dessen Laufzeit beschreibt.

Satz 3.10.: Sei (**S**, **T**, Σ_{ST}, Σ_T) ein Datenaustauschszenario mit den Quelle-zu-Ziel Abhängigkeiten Σ_{ST} und den Zielabhängigkeiten Σ_T. Weiters sei *S* eine Grundinstanz des Zielschemas **S**. Wenn es eine Lösung für das Datenaustauschszenario gibt, berechnet der FindCore^E Algorithmus korrekt den Kern einer kanonischen universellen Lösung in der Zeit $O(|dom(S)|^b)$, bei dem b nur von $\Sigma_{ST} \cup \Sigma_T$ abhängt.

3.6. Performance Vergleich zwischen Algorithmen

Die Entscheidung, ob ein Graph H der Kern eines Graphs G ist, ist DP-vollständig, was bedeutet, dass solange P ≠ NP kein Algorithmus in polynomieller Zeit den Kern einer beliebigen Struktur berechnen kann. Beim Datenaustausch gibt es jedoch einige Einschränkungen, welche es erlauben, den Kern dennoch in polynomieller Zeit zu berechnen.

Der als erstes vorgestellte Greedy-Algorithmus besitzt eine Laufzeit $O(m^{2a+ab} + m^{2a+aa'})$. Dabei ist m die Größe der Quellinstanz (also die Anzahl der Tupel in ihr), a die maximale Anzahl aller allquantifizierten Variablen in den TGDs von Σ_{ST}, b die maximale Anzahl aller existenzuell quantifizierten Variablen in den TGDs von Σ_{ST} und a' die Anzahl aller allquanitifizierten Variablen aller EGDs in Σ_T. Solange das Datenaustauschszenario fix ist, sind a, a' und b Konstanten. Die Verbesserung dessen ist der Blocks-Algorithmus, welcher eine polynimielle Laufzeit besitzt.

Einen weitere Verbesserung sind die Algorithmen FindCore und FindCore^E, welche beide im schlechtesten Fall die Laufzeit $O(|dom(S)|^b)$ besitzen, wobei b nur von $\Sigma_{ST} \cup \Sigma_T$ abhängt. Der Vorteil von FindCore^E ist, dass damit universelle Lösungen ohne Kernberechnung berechnet werden können. Einige Experimente in [18] haben gezeigt, dass der FindCore^E Algorithmus teilweise deutlich schneller als dessen Urfassung ist,

was die nachfolgende aus [18] stammende Abbildung 3 aufzeigt. Darauf zu sehen sind Laufzeiten einer Implementierung, bei denen verschiedene Szenarios durchlaufen wurden. Die Tests wurden auf einer Workstation (Suse Linux, zwei 2,3 GHz Quad-Core Prozessoren, 16 GB RAM, Oracle 11g) durchgeführt.

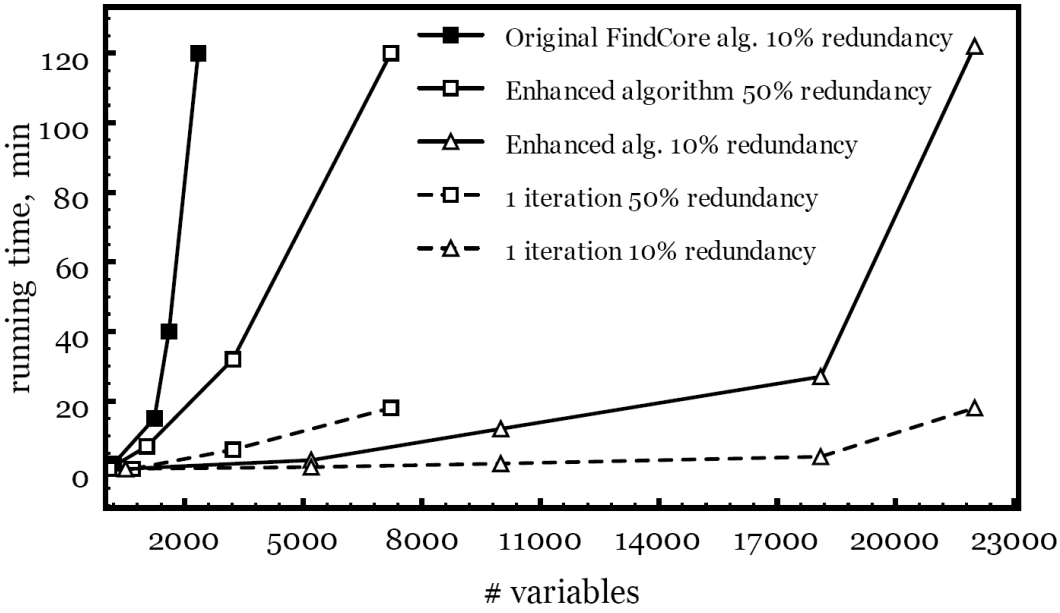


Abbildung 3: Laufzeiten der Kernberechnung [18]

Bei einem Szenario, bei dem die kanonische Lösung etwa 50% mehr Variablen hatte als der Kern, konnten nur ca. 7.000 Variablen in der Zieldatenbank innerhalb von 2 Stunden verarbeiten werden (zweite durchgehende Linie von links). Auf der anderen Seite konnten rund 22.000 benannte Nullen in einer ähnlichen Zeit verarbeitet werden, wenn die kanonische Lösung nur 10% größer als der Kern war (dritte durchgehende Linie von links). Der normale FindCore Algorithmus ist bei der Laufzeit wesentlich langsamer und konnte bei 10% Redundanz nur ca. 2.500 Variablen innerhalb der gleichen Zeit verarbeiten.

Wie die Grafik zeigt, bleibt die Problematik, dass die Kernberechnung bei großen Datenmengen extrem aufwendig ist selbst beim FindCore^{E} Algorithmus erhalten (besonders bei hoher Redundanz). Das nächste Kapitel stellt daher eine andere Methode zum direkten Berechnen von Kernen vor.

4. Kernberechnung durch Chase

Die Erläuterungen dieses Kapitels basieren auf der Arbeit von Pichler und Savenkov [17] und sind die theoretische Basis zur Implementierung, welche im nächsten Kapitel erläutert wird und den praktischen (Haupt-) Teil der Arbeit darstellt.

4.1. Motivation und Idee

Einleitung

Im vorherigen Kapitel haben wir gesehen, wie aus einer Quellinstanz durch den Chase und durch verschiedene Algorithmen zur Berechnung von Kernen eine Zielinstanz gebildet wurde. Dieser Ansatz ist in der Abbildung 4 dargestellt:

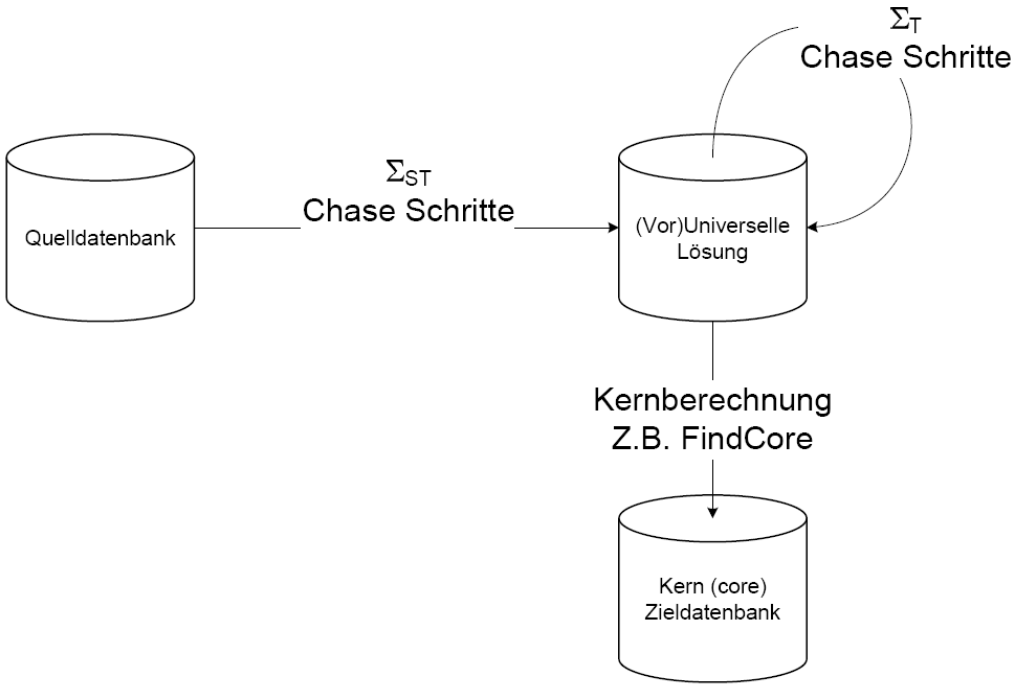


Abbildung 4: Bisherige Lösung

Wir wollen in den folgenden Kapiteln einen Ansatz vorstellen, wie der Kern direkt durch den Chase berechnet werden kann. Im ursprünglichen Chase werden die

einzelnen Abhängigkeiten mehr oder minder zufällig aus der gesamten Menge gewählt und ausgeführt. Je nachdem, wie die Reihenfolge gewählt wurde, kann die Größe der Lösung unterschiedlich ausfallen, wie das folgende Beispiel zeigt:

Beispiel 4.1.: In der Quellinstanz gibt es zwei Relationen: *Grundlehrgang*(*Kursname*, *Vortragender*, *Betreuer*) und *Aufbaulehrgang*(*Kursname*). In der ersten Relation befindet sich das Tupel *Grundlehrgang*(*C#*, *Hans*, *Hans*), in der zweiten das Tupel *Aufbaulehrgang*(*C#*). Folglich hält Hans den C# Lehrgang und ist auch dessen Betreuer, weiters gibt es einen erweiterten C# Kurs. In der Zielinstanz gibt es die folgenden Relationen: *Kursname*(*id_k*, *name*), *Fachbereich*(*id_f*, *name*), *Lehrt*(*id_Fachbereich*, *id_Kursname*) und *BenoetigtLabor*(*id_Fachbereich*, *lab*) Die Quelle-zu-Ziel Abhängigkeiten sind die folgenden:

STD1: *Aufbaulehrgang*(*C*) → ∃*id_k*, *id_f*, *n Kursname*(*id_k*, *C*), *Fachbereich*(*id_f*, *n*), *Lehrt*(*id_f*, *id_k*)

STD2: *Grundlehrgang*(*C*, *L*, *T*) → ∃*id_k*, *idt*, *id_l Kursname*(*id_k*, *C*), *Fachbereich*(*id_l*, *L*), *Lehrt*(*id_l*, *id_k*), *Fachbereich*(*idt*, *T*), *Lehrt*(*idt*, *id_k*)

STD3: *Lehrt*(*id_f*, *id_k*) → ∃*l BenoetigtLabor*(*id_f*, *l*)

Die folgenden Instanzen sind allesamt gültige Lösungen:

J = {*Kursname*(C_1, *C#*), *Kursname*(C_2, *C#*),
Fachbereich(F_1, N_1), *Fachbereich*(F_2, *Hans*), *Fachbereich*(F_3, *Hans*),
Lehrt(F_1, C_1), *Lehrt*(F_2, C_2), *Lehrt*(F_3, C_2),
BenoetigtLabor(F_1, L_1), *BenoetigtLabor*(F_2, L_2), *BenoetigtLabor*(F_2, L_2)}

J_c = {*Kursname*(C_1, *C#*), *Fachbereich*(F_1, *Hans*), *Lehrt*(F_1, C_1), *BenoetigtLabor*(F_1, L_1)}

Die erste kanonische universelle Lösung ist entstanden, indem die Abhängigkeiten in der Reihenfolge wie sie oben steht durchgeführt wurden. Demnach erstellt STD1 die Fakten *Kursname*(C_1, *C#*), *Fachbereich*(F_1, N_1), *Lehrt*(F_1, C_1). Daraufhin generiert STD2 die Tupel *Kursname*(C_2, *C#*), *Fachbereich*(F_2, *Hans*), *Fachbereich*(F_3, *Hans*), *Lehrt*(F_2, C_2), *Lehrt*(F_3, C_2) und im Anschluss noch STD3 die *BenoetigtLabor* Fakten.

Wenn die beiden ersten STDs getauscht würden, müssten die ersten 3 Fakten nicht erstellt werden. In der angegebenen Quelldatenbank ist durch STD2 auch STD1 erfüllt und das einfache Umsortieren kann das Erstellen von redundanten Tupeln verhindern, wenn die gleiche Bezeichnung in *Grundlehrgang* und *Aufbaulehrgang* vorkommt.

Die reine Änderung der Reihenfolge bewirkt jedoch nicht automatisch, dass keine unnötigen Tupel generiert werden, wie das folgende einfache Beispiel zeigt:

Beispiel 4.2.: Wir haben eine TGD $S(x, y) \rightarrow \exists w, z P(x, w) \wedge P(z, w)$. Wenn diese Regel auf eine Datenbank, welche nur aus dem Atom $\{S(1, 2)\}$ besteht, angewandt wird, entstehen daraus die Tupel $\{P(1, w)$ und $P(z, w)\}$. Hier ist das zweite Tupel nicht Teil des Kerns und wir hätten es nicht einfügen müssen, wenn wir zuerst die TGD zu einer logisch gleichwertigen kleineren TGD minimiert hätten, nämlich $S(x, y) \rightarrow \exists w P(x, w)$. Das Umschreiben der TGD hat keinen Einfluss auf die Semantik des Datenaustausches, weil sowohl $\{P(1, w)\}$ als auch $\{P(1, w)$ und $P(z, w)\}$ universelle Lösungen des ursprünglichen Problems sind.

Wir wollen nun einen Schritt weiter gehen und auf die Quelldaten des ersten Beispiels eingehen. Wie bereits kurz erwähnt, kann in speziellen Situationen durch Änderung der Ausführungsreihenfolge von STDs einige davon übersprungen werden, woraus man ableiten kann, dass speziellere Tupel als erstes eingefügt werden sollten. Betrachten wir nun die STD2 genauer: Im speziellen Fall, dass eine Person sowohl einen Kurs hält als auch dessen Betreuer ist wird diese Person zweimal im Fachbereich eingetragen. Man könnte diesen Effekt verhindern, indem man eine Spezialversion von STD2 einführt, welche vor der normalen STD2 ausgeführt wird und damit nur die speziellere Version während des Chase ausgeführt wird. Solche eine spezielle STD2 würde

Grundlehrgang(C, T, T) → *∃idc, idt Kursname(id_k, C),*
Fachbereich (id_t, T), Lehrt (id_t, id_k)

sein und das doppelte Einfügen in *Fachbereich* vermeiden.

Wie dieses Beispiel zeigt hängt es von den Daten der Quelle ab, ob eine TGD überflüssige Tupel einfügt oder nicht. Dies kann außerdem nicht nur eine einzelne TGD betreffen sondern auch mehrere, wie das folgende Beispiel von Fagin, Kolaitis und Popa aus [5] zeigt.

Beispiel 4.3.: [5] Wir gehen davon aus, dass das Quellschema aus einer vierwertigen Relation *S* und das Zielschema aus einer fünfstelligen Relation *R* besteht. Es gibt zwei Quelle-zu-Ziel Abhängigkeiten TGD1 und TGD2:

TGD1: $S(a, b, c, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5 \, (R(x_5, b, x_1, x_2, a) \wedge$
$R(x_5, c, x_3, x_4, a) \wedge$
$R(d, c, x_3, x_4, b))$

TGD2: $S(a, b, c, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5 \, (R(d, a, a, x_1, b) \wedge$
$R(x_5, a, a, x_1, a) \wedge$
$R(x_5, c, x_2, x_3, x_4))$

Die Quellinstanz I ist $\{S(1, 1, 2, 3)\}$.

Wenn der Chase mit TGD1 auf *I* durchgeführt wird entsteht
$\{R(N_5, 1, N_1, N_2, 1), R(N_5, 2, N_3, N_4, 1), R(3, 2, N_3, N_4, 1)\}$.
N_1, N_2, N_3, N_4 *und* N_5 sind benannte Nullen. (1)

Bei Durchführung des Chase von TGD2 auf *I* entsteht
$\{R(3, 1, 1, N'_1, 1), R(N'_5, 1, 1, N'_1, 1), R(N'_5, 2, N'_2, N'_3, N'_4)\}$.
N'_1, N'_2, N'_3, N'_4 *und* N'_5 sind Variablen. (2)

Sei *J* die universelle Lösung die durch Kombinieren der beiden obigen Lösungen entsteht. Wir zeigen nun, dass der Kern die folgende Instanz J_0 ist, welche das dritte Tupel aus dem ersten Chase und das erste Tupel aus dem zweiten Chase enthält. $J_0 = \{R(3, 2, N_3, N_4, 1), S(3, 1, 1, N'_1, 1)\}$.
Es ist leicht fest zustellen, dass J_0 ein Abbild der universellen Lösung *J* ist, indem man die folgenden Endomorphismen *h* betrachtet: $h(N_1) = 1$, $h(N_2) = N'_1$, $h(N_3) = N_3$, $h(N_4)$

$= N_4$, $h(N_5) = 3$, $h(N'_1) = N'_1$, $h(N'_2) = N3$, $h(N'_3) = N4$, $h(N'_4) = 1$ *und* $h(N'_5) = 3$. Weiters gibt es keinen Endomorphismus von J_0 in eine korrekte Unterstruktur von J_0. Daraus folgt, dass J_0 der Kern ist.

Aufgrund der Tatsache, dass der Chase mit TGD1 drei Tupel besitzt, der Kern aber nur zwei Tupel hat, kann man sehen, dass das Ausführen von TGD1 und danach TGD2 nicht den Kern berechnet. Genau dasselbe gilt auch, wenn man die Reihenfolge umdreht. Keiner der beiden Chases ergibt den Kern, was hier gezeigt werden sollte.
Dieses Beispiel hat noch weiter Besonderheiten, auf die in Folge eingegangen wird.
Erstens ist es nicht möglich, eine der Konjunktionen aus der rechten Seite von TGD1 oder TGD2 zu entfernen und immer noch eine gleichwertige Abhängigkeit zu behalten (die TGD kann also nicht minimiert werden). Daher liegt es nicht an einer Redundanz in einer der TGDs, dass der Kern nicht durch den Chase berechnet werden kann.
Zum Zweiten ist der Gaifman Graph der Nullen, welcher in (1) entsteht, verbunden. Das sagt uns, dass TGD1 nicht in mehrere TGDs mit derselben linken Seite aufgespaltet werden kann. Dasselbe gilt auch für (2). Daher liegt das Problem, dass der Kern nicht durch den Chase berechnet werden kann, auch nicht daran, dass eine der TGDs aufgespaltet (normalisiert) werden könnte.
Die dritte Besonderheit ist, dass nicht nur die Menge (1) nicht komplett im Kern enthalten ist, sondern auch deren Kern, welcher aus dem ersten und dem dritten Tupel besteht, nicht komplett Teil des Kerns ist. Für die Menge (2) gilt dasselbe, wobei deren Kern ebenfalls aus dem ersten und dritten Tupel besteht. Es folgt daraus, dass selbst wenn wir in der Lage wären den Kern jeder TGD direkt zu berechnen, der Kern des Ganzen nicht durch den Chase berechnet werden kann.

Das obige Beispiel zeigt ebenfalls auf, dass der Grad der Redundanz des Ziels von den Quelldaten abhängt. Daraus ergibt sich die Idee, das Datenaustauschszenario abzuwandeln indem die Abhängigkeiten so umgeschrieben werden, dass diese bei speziellen Eingabewerten keine Redundanzen mehr generieren (Die Redundanz in *J* entstand wegen den ersten beiden Attributen in *R,* welche gleich waren). Es wird durch das Beispiel auch klar, dass man um alle Besonderheiten abzudecken, mehrere

Abhängigkeiten miteinander vergleichen muss. Aus dieser Überlegung ergibt sich die in weiterer Folge vorgestellte neue Idee.

Die Idee

Wir wollen sicherstellen, dass jeder Schritt des Chase wenn überhaupt nur eine minimale Datenbankänderung vornimmt und gleichzeitig auf vergangene oder zukünftige Chaseschritte Rücksicht nimmt. Aus diesem Grund wird jeder Schritt mit zusätzlichen Abfragen ausgeführt sodass nur eine minimale (oder keine) Änderung an der Zieldatenbank vorgenommen wird. Im Gegensatz zum bisherigen Chase wird das Erfüllen einer Abhängigkeit für ein einzelnes Tupel auf mehrere Schritte aufgeteilt.

Um dies zu veranschaulichen gehen wir zurück zum vorherigen Beispiel und gehen von einem Tupel der Quelldatenbank mit drei gleichen Werten aus *S* aus: (*1, 1, 1, 3*). Es fällt sofort auf, dass die ersten drei Attribute gleich sind und daher das Beispiel eine Instanz des Eingabemusters (*a, a, a, d*) ist. Für jedes solches Quelltupel würden die beiden TGDs folgendermaßen aussehen:
TGD1: $S(a, a, a, d) \rightarrow \exists N_3, N_4\ R(d, a, N_3, N_4, a)$
TGD2: $S(a, a, a, d) \rightarrow \exists N_1'\ R(d, a, a, N_1', a)$

Man sieht hier, dass die rechte Seite von TGD1 immer allgemeiner als die von TGD2 ist, weil es einen Homomorphismus $rhs(d_1) \rightarrow rhs(d_2)$ gibt, aber nicht umgekehrt. Dies bedeutet, dass die erste Abhängigkeit Redundanzen für das Eingabeschema enthält. Wenn also TGD1 nach der TGD2 für ein Tupel aus *S* nach diesem Eingabeschema ausgeführt wird, hat diese keinen Effekt, im umgekehrten Fall generiert es aber ein Tupel, das nicht zum Kern gehört.
In der Folge beschäftigen wir uns damit, wie alle möglichen Spezialfälle aus den Abhängigkeiten berechnet werden können. Danach wollen wir den Kern durch den Chase berechnen, welche auf zuvor berechneten Abhängigkeiten beruht. Da der Chase keine Tupel aus dem Ziel entfernt müssen wir beim Chase aufpassen, keine überflüssigen Tupel in das Ziel einzufügen. Aufgrund der Tatsache dass der Chase rekursiv ablaufen kann, müssen wir die Zielabhängigkeiten einschränken. In dieser

Arbeit wird nur der Fall behandelt, dass es keine Zielabhängigkeiten gibt und somit unser Szenario nur Quelle-zu-Ziel Abhängigkeiten enthält. Die grundsätzliche Idee ist in Abbildung 5 dargestellt.

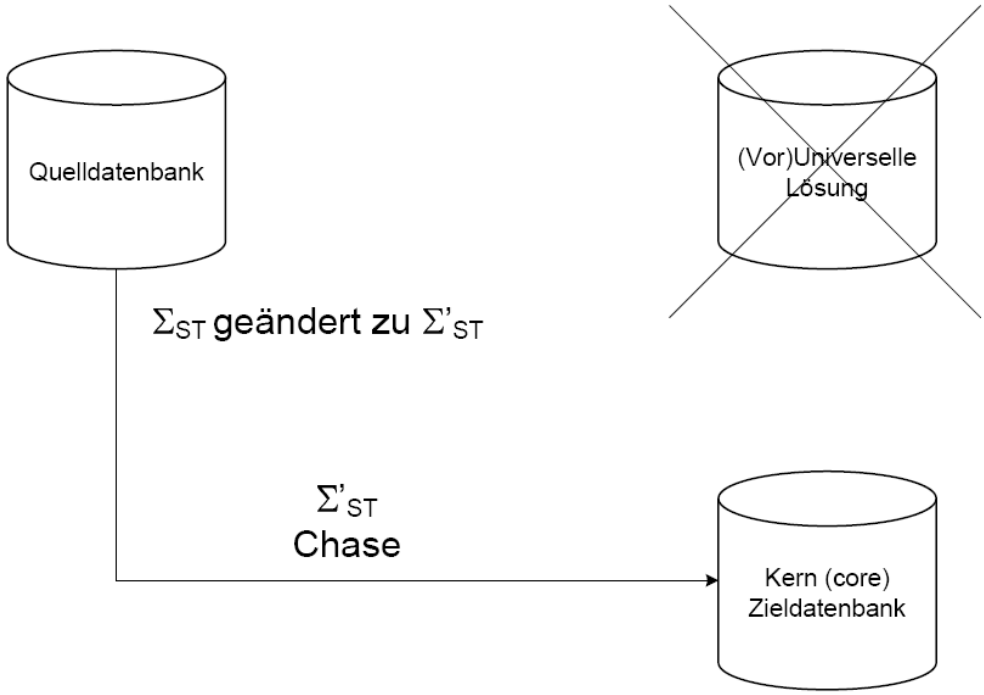


Abbildung 5: Neuer Ansatz

4.2. Berechnung von TGD-Varianten

Eine Ersetzung (*substitution*) σ ist eine Abbildung, welche Variablen in andere Elemente der Domäne abbildet. Wir schreiben $\sigma = \{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$ wenn σ jedes x_i auf a_i abgebildet wird.

Neben den Funktionen *dom(·)* und *range(·)* welche die Domäne bzw. die Bereich der Variablenersetzung beschreiben, werden die Funktionen *vrange(·)* und *crange(·)* definiert. Diese liefern Teile des Bereichs zurück, der Variablen bzw. Konstanten beinhaltet. Daher ist für jede Ersetzung σ, $range(\sigma) = vrange(\sigma) \cup crange(\sigma)$.

Eine Variablenzuweisung (*variable assignment*, oder *valuation*) υ ersetzt Variablen in einer Formel mit Werten aus einer Datenbank. Wir schreiben $\upsilon = \{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$, wenn σ alle x_i auf a_i abbildet.

Der wesentliche Unterscheid zwischen diesen beiden Typen ist, dass die Variablenersetzung eine Formel in eine andere umwandelt, während die Variablenzuweisung Datenbankinstanzen mit der gegebenen Abhängigkeit erstellt.
In diesem Kapitel beschreibt *a(·)* die Prämisse (linke Seite) und *c(·)* die Aussage (rechte Seite) einer Abhängigkeit. Bei einer TGD $\tau : \varphi(\vec{x}, \vec{y}) \rightarrow \exists \vec{z} \psi(\vec{x}, \vec{z})$ wäre $a(\tau) = \varphi(\vec{x}, \vec{y})$ und $c(\tau) = \exists \vec{z} \psi(\vec{x}, \vec{z})$.

Für eine Abhängigkeit τ mit der Prämisse $\varphi(\vec{x})$ und einer Variablenzuweisung υ für $\vec{x}$ ist die Abhängigkeitsdatenbank (*antedecedent database*) von τ unter υ, welche als $A(\tau, \upsilon)$ bezeichnet wird, eine Instanz, bei der jedes Atom $R(\vec{x}_i)$ aus $\varphi(\vec{x})$ mit einem Fakt $R(\vec{x}_i, \upsilon) \in A(\tau, \upsilon)$ in Zusammenhang steht.

Wenn υ jeder Variable in $\vec{x}$ eine neue Konstante zuordnet dann wird $A(\tau, \upsilon)$ kanonische Abhängigkeitsdatenbank von τ genannt.
Eine Aussagendatenbank (*conclusion database*) $C(\tau, \upsilon)$ wird in ähnlicher Weise definiert und die kanonische Aussagendatenbank ist mit dem Unterschied definiert, dass υ jeder existenziell quantifizierten Variable eine frische benannte Null zuweist.
Beim Anschreiben der Abhängigkeits- und Aussagendatenbank werden Konstante mit einem Punkt markiert. So wären zum Beispiel bei der Instanz $I = \{R(\dot{x}, y), P(\dot{z}, w)\}$ $\dot{x}$ und $\dot{z}$ Konstanzen, während *y* und *w* Variablen sind.

Beispiel 4.4. Sei τ die folgende TGD:
$R(x, y) \wedge P(y, z) \rightarrow \exists v \exists w\ (S(x, v) \wedge S(w, y) \wedge Q(v, w))$
Dann sind die Abhängigkeits- und Aussagendatenbank die folgenden:
$A(\tau) = \{R(\dot{x}, \dot{y}), P(\dot{y}, \dot{z})\}$
$C(\tau) = \{S(\dot{x}, v), S(w, \dot{y}), Q(v, w)\}$

Eine Variablenersetzung σ ist relevant im Bezug auf eine TGD, wenn diese relevante Variablen in der TGD durch andere relevanten Variablen ersetzt. Solch ein σ wir r-Ersetzung (*r-substitution*) für die TGD genannt.

Beispiel 4.5. Betrachten wir die folgende TGD

$R(x, z) \rightarrow \exists y\, P(x, y) \wedge P(y, z)$

und eine r-Ersetzung {z → x}, dann hat die entstehende TGD $\tau\sigma$ die Form

$R(x, x) \rightarrow \exists y\, P(x, y) \wedge P(y, x)$

Satz 4.1. Sei Σ_{ST} eine Menge von TGDs, ξ eine Chase-Sequenz für Σ_{ST} und I eine Quellinstanz. Für jeden Chase-Schritt $(\tau, \upsilon) \in \xi$ und für eine r-Ersetzung σ für τ gilt, dass, wenn eine Variablenzuweisung υ' für die Prämissenvariablen von $\tau\sigma$ existiert, sodass für alle $x \in dom(\upsilon)$ $\upsilon(x) = \upsilon'(\sigma(x))$ gilt, der Chase-Schritt $(\tau\sigma, \sigma\upsilon')$ (τ, υ) in ξ ersetzen kann. Die so veränderte Chase-Sequenz erstellt damit eine universelle Lösung für I mit Σ_{ST}.

Das Einfügen einer zusätzlichen Variablenersetzung ist notwendig, damit die Redundanz in der Aussage der TGD explizit gemacht wird.

Beispiel 4.6. Betrachten wir die TGD1 aus Beispiel 4.3.:

$TGD1^{abcd}$: $S(a, b, c, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5\, (R(x_5, b, x_1, x_2, a) \wedge$
$R(x_5, c, x_3, x_4, a) \wedge R(d, c, x_3, x_4, b))$

Es gibt hier zahlreiche Möglichkeiten, wie die r-Ersetzung erfolgen kann:

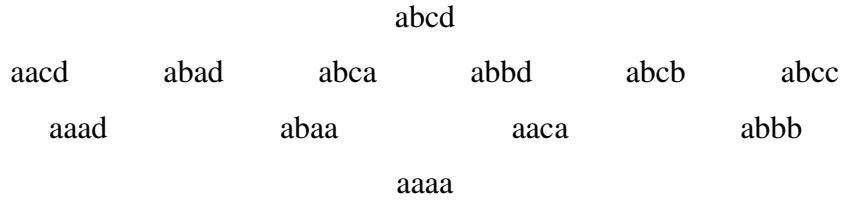
Mit der r-Ersetzung $\sigma = \{c \rightarrow b\}$ bekommt die TGD die folgende Form:

$TGD1^{abbd}$: $S(a, b, b, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5\, (R(x_5, b, x_1, x_2, a) \wedge$
$R(x_5, b, x_3, x_4, a) \wedge R(d, b, x_3, x_4, b))$

Wenn der Chase mit der $TGD1^{abbd}$ durchgeführt wird, entstehen immer redundante Fakten. Der Grund dafür ist, dass die rechte Seite nicht der Kern ist. Es gibt nämlich einen gültigen Endomorphismus h auf die kanonische Aussagendatenbank von τ, sodass $h(C(\tau_1\sigma)) = \{R(x_5, \dot{b}, x_3, x_4, \dot{a}), R(\dot{d}, \dot{b}, x_3, x_4, \dot{a})\}$. Es genügt $h(x_1) = x_3$ und $h(x_2) = x_4$ zu setzen, sowie x_3, x_4 und x_5 auf sich selbst abzubilden. Dies ist auch für jeden Chase-

Schritt (τ_l, υ) der Fall, bei dem υ die zweite oder die dritte allquanitfizierte Variable in τ vereinheitlicht.

Prinzipiell muss man alle Möglichkeiten für die r-Ersetzung betrachten, die für die obige TGD folgende wären:



Je höher die Ebene spezieller werden die Ersetzungen. Ersetzungen auf der gleichen Ebene sind gleichwertig.

Hier sind alle weiteren Beispiele der r-Ersetzung für TGD1 und TGD2 aus Beispiel 4.3., wobei (lokal) redundante Tupel durchgestrichen sind.

$TGD1^{abcd}$: $R(x_5, b, x_1, x_2, a), R(x_5, c, x_3, x_4, a), R(d, c, x_3, x_4, b)$
$TGD2^{abcd}$: $R(d, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, c, x_2, x_3, x_4)$

$TGD1^{aacd}$: $R(x_5, a, x_1, x_2, a), R(x_5, c, x_3, x_4, a), R(d, c, x_3, x_4, a)$
$TGD2^{aacd}$: $R(d, a, a, x_1, a), R(x_5, a, a, x_1, a), R(x_5, c, x_2, x_3, x_4)$

$TGD1^{abad}$: $R(x_5, b, x_1, x_2, a), R(x_5, a, x_3, x_4, a), R(d, a, x_3, x_4, b)$
$TGD2^{abad}$: $R(d, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, a, x_2, x_3, x_4)$

$TGD1^{abca}$: $R(x_5, b, x_1, x_2, a), R(x_5, c, x_3, x_4, a), R(a, c, x_3, x_4, b)$
$TGD2^{abca}$: $R(a, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, c, x_2, x_3, x_4)$

$TGD1^{abbd}$: ~~$R(x_5, b, x_1, x_2, a)$~~, $R(x_5, b, x_3, x_4, a), R(d, b, x_3, x_4, b)$
$TGD2^{abbd}$: $R(d, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, b, x_2, x_3, x_4)$

$TGD1^{abcb}$: $R(x_5, b, x_1, x_2, a), R(x_5, c, x_3, x_4, a), R(b, c, x_3, x_4, b)$

$TGD2^{abcb}$: R(b, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, c, x_2, x_3, x_4)

$TGD1^{abcc}$: R(x_5, b, x_1, x_2, a), R(x_5, c, x_3, x_4, a), R(c, c, x_3, x_4, b)
$TGD2^{abcc}$: R(c, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, c, x_2, x_3, x_4)

$TGD1^{aaad}$: ~~R(x_5, a, x_1, x_2, a)~~, ~~R(x_5, a, x_3, x_4, a)~~, R(d, a, x_3, x_4, a)
$TGD2^{aaad}$: R(d, a, a, x_1, a), ~~R(x_5, a, a, x_1, a)~~, ~~R(x_5, a, x_2, x_3, x_4)~~

$TGD1^{abaa}$: R(x_5, b, x_1, x_2, a), R(x_5, a, x_3, x_4, a), R(a, a, x_3, x_4, b)
$TGD2^{abaa}$: R(a, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, a, x_2, x_3, x_4)

$TGD1^{aaca}$: R(x_5, a, x_1, x_2, a), R(x_5, c, x_3, x_4, a), R(a, c, x_3, x_4, a)
$TGD2^{aaca}$: R(a, a, a, x_1, a), R(x_5, a, a, x_1, a), R(x_5, c, x_2, x_3, x_4)

$TGD1^{abbb}$: ~~R(x_5, b, x_1, x_2, a)~~, R(x_5, b, x_3, x_4, a), R(b, b, x_3, x_4, b)
$TGD2^{abbb}$: R(b, a, a, x_1, b), R(x_5, a, a, x_1, a), R(x_5, b, x_2, x_3, x_4)

$TGD1^{aaaa}$: ~~R(x_5, a, x_1, x_2, a)~~, ~~R(x_5, a, x_3, x_4, a)~~, R(a, a, x_3, x_4, a)
$TGD2^{aaaa}$: R(a, a, a, x_1, a), ~~R(x_5, a, a, x_1, a)~~, ~~R(x_5, a, x_2, x_3, x_4)~~

Definition 4.1. Sei τ eine TGD, dann ist *core-right*(τ) eine umgeschriebene Abbildung, welche die Aussage der TGD auf der rechten Seite mit deren Kern ersetzt.

4.3. Chase für TGD-Varianten

Nachdem die verbesserten Varianten der TGDs gefunden wurden, wollen wir in diesem Kapitel einen darauf aufbauenden Chase betrachten.

Satz 4.2. Wir verwenden eine Chase-Sequenz ξ, eine Quellinstanz I mit einer Menge Σ_{ST} von Quelle-zu-Ziel Abhängigkeiten, eine Instanz J_ξ, welche durch ξ generiert wurde und einem Chase-Schritt $(\tau, \upsilon) \in \xi$, sodass die Aussage $c(\tau)$ nicht der Kern ist. Außerdem sei ξ' eine Chase-Sequenz welche von ξ abgeleitet wurde, indem (τ,

υ) durch *(core-right(*τ*),* υ) ersetzt wurde und $J_{\xi'}$ eine Datenbankinstanz, welche durch ξ' generieret wurde, dann gilt das folgende:

1. *(*τ*,* υ*)* erstellt Fakten, die nicht Teil des Kerns von $J_{\xi'}$ sind. Dies sind die Atome, welche zu den Redundanten Fakten in *c(*τ*)* gehören
2. $J_{\xi'}$ ist homomorphisch äquivalent zu J_ξ und
3. $|J_{\xi'}| < |J_\xi|$.

Wie man sieht, kann eine durch die *core-right* Regel umgeschriebene TGD in jeder Chase-Sequenz anstelle des Originals verwendet werden und die so geänderte Abfolge von Chase-Schritten erzeugt weniger (oder zumindest gleich viele) Tupel im Ziel, wie die ursprüngliche Abhängigkeit.

Definition 4.2. (Lokal adaptiver Chase): Eine lokal adaptive Version einer Chase-Sequenz ξ wird gefunden, indem beliebig viele Chase-Schritte *(*τ*,* υ*)* $\in \xi$ durch (*core-right(*τ*),* $\sigma\upsilon$') ersetzt werden, wobei σ und υ' die Vorbedingungen aus Satz 4.1. erfüllen und $|c(core\text{-}right(\tau\sigma))| < |c(\tau)|$ ist.

Satz 4.3. Betrachten wir eine Chase-Sequenz ξ einer Quellinstanz I mit einer Menge Σ_{ST} von Quelle-zu-Ziel Anbhängigkeiten und eine Instanz J_ξ welche durch ξ entstanden ist. Eine lokal adaptive Version ξ' von ξ liefert eine universelle Lösung $J_{\xi'}$ für I mit Σ_{ST}, sodass $|J_{\xi'}| < |J_\xi|$ ist.

Wir ermitteln bei jedem Chase-Schritt welche zusätzlichen Bedingungen zwischen den Elementen einer Variablenzuordnung υ bestehen. Die daraus entstehenden Versionen von TGDs können ohne die eigentlichen Daten zu betrachten berechnet werden und dann entsprechend ausgewählt werden, je nachdem auf welche Eingabedaten man stößt.

4.4. Berechnung von TGD-Varianten

Wie schon das Beispiel 4.3. zeigt reicht es nicht immer aus, Redundanzen innerhalb einer TGD zu suchen, um das Einfügen von nicht benötigten Variablen zu verhindern. Man muss daher das Zusammenspiel mehrerer TGDs in allen Varianten untersuchen.

Beispiel 4.7.: Betrachten wir nun das Zusammenspiel zweier Spezialfälle von Abhängigkeiten aus dem Beispiel 4.3.:
$TGD1^{aacd}$: $R(x_5, a, x_1, x_2, a)$, $R(x_5, c, x_3, x_4, a)$, $R(d, c, x_3, x_4, a)$
$TGD2^{aacd}$: $R(d, a, a, x_1', a)$, $R(x_5', a, a, x_1', a)$, $R(x_5', c, x_2', x_3', x_4')$

Für sich betrachtet kann keines der Tupel weggelassen werden, wenn beide Abhängigkeiten kombiniert werden ergibt sich folgende TGD (rechte Seite):
~~$R(x_5, a, x_1, x_2, a)$~~, ~~$R(x_5, c, x_3, x_4, a)$~~, $R(d, c, x_3, x_4, a)$,
$R(d, a, a, x_1', a)$, ~~$R(x_5', a, a, x_1', a)$~~, ~~$R(x_5', c, x_2', x_3', x_4')$~~

Nun können die durchgestrichenen (redundanten) Tupel eliminiert werden und übrig bleibt eine minimale Datenbankänderung.

Die einfachste Möglichkeit, alle Varianten einer TGD mit allen Varianten aller anderen TGDs zu kombinieren, ist mittels Brute-Force, also dem ausprobieren aller Möglichkeiten. Die Anzahl der zu untersuchenden Möglichkeiten steigt dabei mit der Anzahl der TGDs exponentiell an, ist also sehr aufwändig. Diese Analyse muss jedoch nur einmal ohne Berücksichtigung der eigentlichen Daten (jedoch für jedes Eingabeschema) erfolgen. In weiterer Folge muss der Chase so angepasst werden, dass dieser die speziellsten Varianten zuerst ausführt, um nicht unnötige Tupel im Ziel einzufügen.
Die Berechnung von Kernen auf diese Weise muss zwar am Anfang eine große Menge von möglichen Kombinationen durchprobieren, doch im Gegenzug kann auf die, für große Datenmengen nicht mehr sinnvoll durchführbare, Berechnung des Kerns aus einer universellen Lösung verzichtet werden.

5. Implementiertes Framework

In diesem Kapitel wird beschrieben, wie die im vorherigen Kapitel theoretischen Grundlagen umgesetzt wurden. Die Implementierung setzt auf der Diplomarbeit von Savenkov [20] auf, wobei Details der Implementierung seither etwas angepasst wurden. Als Programmiersprache wurde wie auch von Savenkov Java verwendet

Der Java – Code ist in drei SVN – Archiven am Institut für Informationssysteme verfügbar:

- `ssh://<login>@sg1.dbai.tuwien.ac.at//web/proj/InfInt/Code/codedx`
  Dieses Archiv beinhaltet das bereits vorhandene System zur Kernberechnung, wobei auch hier, wie weiter unten beschrieben, einige Details neu implementiert bzw. verändert wurden, um den neuen Anforderungen zu genügen.
- `ssh://<login>@sg1.dbai.tuwien.ac.at//web/proj/InfInt/Code/codedx-lib`
  In diesem Archiv finden sich die benötigten Bibliotheken, die zum Ausführen des Systems benötigt werden.
- `ssh://<login>@sg1.dbai.tuwien.ac.at//web/proj/InfInt/Code/codedx-chase`
  Dieses Archiv beinhaltet den im Zuge dieser Diplomarbeit implementierten Code zum Berechnen von Varianten von Abhängigkeiten, welcher in diesem Kapitel beschrieben wird.

5.1. Übernommene Bestandteile

Die für diese Arbeit wesentlichen Pakete aus der Arbeit von Savenkov werden hier noch einmal kurz zusammengefasst. Details sind in den Kommentaren der einzelnen Klassen ersichtlich. Die hier nicht erwähnten Pakete können in der oben genannten Diplomarbeit von Savenkov nachgelesen werden.

Paket dependency

Term

Diese Klasse bildet die einzelnen Variablen und Konstanten einer Abhängigkeit ab. Es werden Funktionen bereitgestellt, um Variablennamen zu vergleichen, frische

Variablennamen zu generieren und zu überprüfen, ob es sich um eine Variable oder eine Konstante handelt.

Atom

Diese Klasse repräsentiert die einzelnen Prädikate einer Datenbank wie z.B. S(A, B, 2). Es sind Funktionen zum Auslesen der Feldnamen und Werte, des Ralationennamens und zum Ersetzen von Variablen vorhanden. Besonders zu erwähnen ist die Funktion getMapping(Atom), welche eine Abbildung von diesem Atom in ein gegebenes in der Form Variablenname → Term zurückliefert, sofern eine solche Abbildung existiert.
Eine solche Abbildung existiert etwa, wenn eine Variable auf eine Konstante abgebildet wird (aber nicht umgekehrt) oder eine Variable auf eine andere benannte Null. Es dürfen jedoch keine Konflikte zwischen zwei Abbildungen entstehen, etwa dass eine Variable A einmal auf die Konstante 1 und einmal auf die Konstante 2 abgebildet wird. Natürlich kann auch eine Konstante nicht auf eine andere abgebildet werden (auf dieselbe natürlich schon).

Expression

Dieses Interface stellt Funktionen zum Auslesen der Variablen, zum Überprüfen ob eine Formel eine andere impliziert und um die Anzahl der Prädikate zu zählen zur Verfügung.

AtomConjunction

Diese Klasse, welche im Wesentlichen eine Menge von Atomen ist und das Interface Expression verwendet, bildet eine Seite einer Abhängigkeit ab. Es werden Funktionen zum Umbenennen von Variablen, zum Auslesen verschiedener Werte und vor allem zum Finden von Abbildungen bereitgestellt. Die hierfür wesentliche Funktion getConsistantMapping(…), welche eine AtomConjunction auf eine andere abzubilden versucht wurde im Zuge diese Arbeit implementiert und wird später genauer erläutert.

ImplicationalDependency

Diese abstrakte Klasse stellt Funktionen zum Auslesen von Prämisse und Aussage, zum Vorbereiten einer Abhängigkeit und zum Ändern von Variablennamen zur Verfügung.

PositiveConjunctiveEGD

Mit dieser Klasse, welche von ImplicationalDependency erbt ist es möglich, gleichheitserzeugende Abhängigkeiten (EGDs) darzustellen. Diese Klasse wird im entwickelten System nicht verwendet, weil das System (und auch [17], welches der theoretische Unterbau der Implementierung ist) TGDs, jedoch keine EGDs unterstützt.

PositiveConjunctiveTGD

Auch diese Klasse erbt von ImplicationalDependency und dient dem Darstellen von tupelgenerierenden Abhängigkeiten (TGDs). Es werden sowohl die rechte als auch die linke Seite (jeweils AtomConjunctions) einer Abhängigkeit gespeichert. Neben den in ImplicationalDependency zur Verfügung gestellten Funktionen steht auch eine Funktion zum Auslesen von existenziell quantifizierten Variablen zur Verfügung, welche aber in der vorliegenden Fassung noch nicht unterstützt wird.

VariableSubstitution

Diese von HashMap erbende Klasse ermöglicht es, Variablenersetzungen der Form Variablenname → Term abzuspeichern und auszulesen und ist für die Ersetzung von Variablen essentiell. Neben Funktionen für das Ein- und Auslesen solcher Ersetzungen gibt es eine Funktion um triviale Ersetzungen (A → A) zu entfernen und zu überprüfen ob zwei VariableSubstitutions widerspruchsfrei sind.

Paket dependency.parse

DependencyParser

Diese Helferklasse dient dem Umwandeln einer Zeichenkette in eine Abhängigkeit und wird zum Einlesen verwendet.

Nachdem hier einige grundlegende Funktionen der vorliegenden Implementierung erläutert wurden wollen wir in den nächsten Kapiteln die Funktionen des neu geschaffenen Systems zum Analysieren von Abhängigkeiten näher betrachten.

5.2. Systemarchitektur

Die grobe Systemarchitektur ist in Abbildung 6 dargestellt. Die gegebenen TGDs werden als Zeichenkette der Form „S(A, B, 2), T(A, 3, C) → P(A, 2), Q(B, C, 3)“ angegeben. Wie man sieht, werden hier die All- und Existenzquantoren in der Eingabe der Einfachheit halber nicht extra angeführt, sondern werden automatisch erkannt. Der Mappings Manager erstellt dann aus einer Basis-TGD und einer Menge von weiteren TGDs alle Spezialfälle der Basis-TGD, welche wie im Kapitel 4 beschrieben eine baumartige Struktur besitzt, deren höhere Ebenen spezialisierter sind als die niedrigeren. In dieser Struktur werden nur Ergebnisse abgespeichert, welche weniger Atome enthalten als der Basisfall.

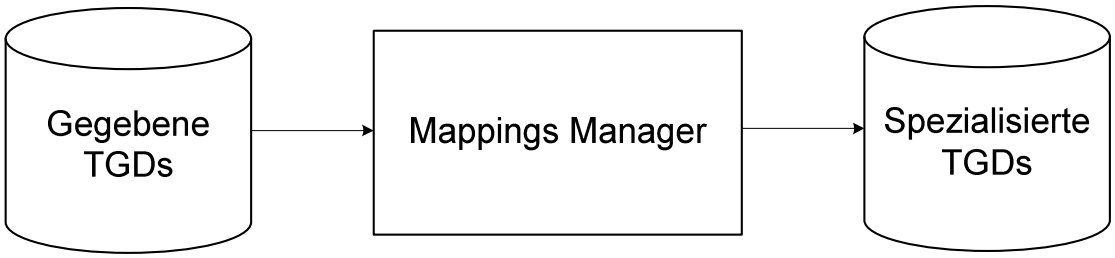


Abbildung 6: Übersicht Systemarchitektur

5.3. Algorithmus zur Minimierung einer Abhängigkeit

Dieser Abschnitt behandelt die Frage, wie die Minimierung von einzelnen Abhängigkeiten implementiert wurde, also der Kern einer einzelnen TGD gefunden werden kann.

Dazu wollen wir zuerst ein Beispiel betrachten:

Aus dem Beispiel von Fagin $TGD1: S(a, b, c, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5 (R(x_5, b, x_1, x_2, a) \wedge R(x_5, c, x_3, x_4, a) \wedge R(d, c, x_3, x_4, b))$

für den speziellen Eingabefall $TGD1^{aaad}$ entsteht aus dieser TGD die folgende, minimierte TGD, die keine unnötigen Atome mehr enthält:

$TGD1^{aaad}: S(a, a, a, d) \rightarrow \exists x_3 \exists x_4 \; R(d, a, x_3, x_4, a)$

Im Wesentlichen folgt die Minimierung dem folgenden Pseudocode, welcher in [16] als „Generic Query Minimization“ Algorithmus beschrieben wird:

```
Funktion: minimizeConjunction
für jedes Atom a aus originalConjunction
{
     minimizedConjunction = originalConjunction
     Entferne a aus minimizedConjunction
     Überprüfe ob es einen Homomorphismus von originalConjunction nach
          minimizedConjunction gibt
     Wenn es eine solche gibt
          originalConjunction = minimizedConjunction
}
Gib originalConjunction zurück
```

Betrachten wir den Spezialfall aaad des obigen Beispiels:

$S(a, a, a, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5 (R(x_5, a, x_1, x_2, a) \wedge R(x_5, a, x_3, x_4, a) \wedge R(d, a, x_3, x_4, a))$

Zuerst versucht der Algorithmus das erste Atom $R(x_5, a, x_1, x_2, a)$ aus der rechten Seite zu entfernen. Es gibt zwei mögliche Abbildungen von dem Ergebnis auf das Original, eine davon ist x1 → x3 und $x_2 \rightarrow x_4$. Die neue rechte Seite wird daher die alte rechte Seite ohne dieses Atom. Als nächstes wird das Atom $R(x_5, a, x_3, x_4, a)$ entfernt. Auch diesmal gibt es eine Abbildung, nämlich $x_5 \rightarrow d$, die neue rechte Seite enthält daher nach dem Ersetzen der alten nur noch $R(d, a, x_3, x_4, a)$. Als letztes wird nun auch noch versucht dieses letzte Atom zu entfernen. Jetzt gibt es aber keine Abbildung mehr auf die nun leere rechte Seite und daher wird die vorherige Lösung beibehalten. Nachdem es keine weiteren Atome zu untersuchen gibt, beendet die Schleife und es wird die (neue) `originalConjunction` mit der rechten Seite $R(d, a, x_3, x_4, a)$ zurückgegeben.

In der Implementierung ist diese Funktionalität in der privaten Funktion `minimizeConjunction` implementiert, welche der Idee des Pseudocodes folgt, jedoch anstatt `minimizedConjunction = originalConjunction` zuerst aus der `originalConjunction` a entfernt und dieses dann, wenn keine Abbildung gefunden werden kann, wieder ergänzt.

Diese Methode ist privat und kann daher nur durch andere Methoden aufgerufen werden:

- `minimizePremise(AtomConjunction toMinimize)`
- `minimizePremise(AtomConjunction toMinimize, Set<Atom> importantAtoms)`
- `minimizeConclusion(AtomConjunction toMinimize)`
- `minimizeConclusion(AtomConjunction toMinimize, Set<Atom> importantAtoms)`

Die `AtomConjunction` enthält entweder die rechte oder linke Seite der Abhängigkeit, die es zu minimieren gilt. In zwei Fällen gibt es einen zusätzlichen Parameter, nämlich `importantAtoms`. In dieser Menge von Atomen können Atome angegeben werden, die minimiert werden sollen (anstatt aller in `AtomConjunction` enthaltenen Atome). Der Algorithmus verwendet in diesem Fall nicht alle Atome aus `toMinimize` sondern nur die in `importantAtoms` angegebenen Atome.

Wenn im obigen Beispiel für den Spezialfall TGD1^{aaad} beispielsweise `importantAtoms` nur das Atom R(x5, a, x1, x2, a) enthält, würde das Ergebnis der Minimierung folgendermaßen aussehen: TGD1: S(a, a, a, d) → $\exists x_3 \exists x_5$ (R(x_5, a, x_3, x_4, a) ∧ R(d, a, x_3, x_4, a)). Das (redundante) Atom R(x_5, a, x_3, x_4, a) wäre also noch im Ergebnis vorhanden. Dieses Verhalten scheint auf den ersten Blick nicht besonders sinnvoll, wir benötigen es allerdings beim Minimieren von mehreren Abhängigkeiten miteinander.

Der Knackpunkt bei diesem Algorithmus ist es, festzustellen ob es eine Abbildung von `originalConjunction` nach `minimizedConjunction` gibt. `minimizedConjunction` ist immer `originalConjunction` \ {a}.
In weiterer Folge wollen wir untersuchen, wie das Finden einer Abbildung von `originalConjunction` nach `minimizedConjunction`, die ja ein Homomorphismus ist, vonstatten geht.

5.3.1. Abbildung zwischen AtomConjunctions finden

Das vorher beschriebene Problem, eine Abbildung von einer `Conjunction` auf eine andere zu finden wird in der bereits beschriebenen Klasse `AtomConjunction` von der

Funktion `getConsistantMapping` durchgeführt. Diese findet eine (von möglicherweise mehreren) gültige Abbildung, sofern es eine gibt. Wichtig ist hierbei, dass die Funktion „Backtracking“ unterstützt, was bedeutet, dass eine einmal gefundene Abbildung auch wieder verworfen werden kann, was anhand des folgenden Beispiels erläutert wird:

Angenommen wir wollen die `AtomConjunction` (also entweder die linke oder rechte Seite einer Abhängigkeit) $S(X,Y) \wedge R(X,Y)$ auf $S(1,2) \wedge S(1,3) \wedge R(1,3)$ abbilden (der Umgekehrte Weg ist wegen der Unmöglichkeit einer Abbildung von Konstanten auf Variablen ohnedies nicht möglich). Ohne das Zurücknehmen von vorher getroffenen Entscheidungen würde der Ablauf folgendermaßen sein: Zuerst wird *S(X, Y)* und *S(1, 2*) betrachtet und *X* daher auf *1* abgebildet und *Y* auf *2*. Im nächsten Schritt wird versucht *S(X, Y)* auf *S(1, 3)* abzubilden. Hier würde X auf 1 und Y auf 3 abgebildet. Die Abbildung von Y auf 3 ist jedoch nicht möglich, weil dieses ja bereits auf 2 abgebildet ist und eine Abbildung auf zwei verschiedene Konstanten nicht möglich ist, daher wird die Abbildung Y → 3 dem Ergebnis nicht hinzugefügt. Zum Schluss wird noch versucht R(X, Y) auf R(1, 3) abzubilden, was wie beim vorherigen Atom scheitert. Damit liefert die gesamte Abbildung kein Ergebnis weil für die Quelle R(X, Y) kein passendes Atom im Ziel gefunden werden kann.

Die Idee beim Backtracking ist es nun, eine vorher getroffene Entscheidung zurückzunehmen, was für das Beispiel bedeuten würde, dass zwar auch zuerst Y auf 2 abgebildet würde, am Ende jedoch diese Entscheidung zurückgenommen wird und versucht wird Y auf 3 abzubilden, was schließlich auch eine korrekte Abbildung liefert, da ja das Y aus S sowohl ein Pendant für 2 als auch für 3 besitzt.
Die derzeitige Implementierung geht beim Backtracking einen etwas anderen Weg. Es wird von vorneherein unterbunden, dass unmögliche Lösungen zum Endergebnis hinzugefügt werden. Das Backtracking wird also durch einen Blick in die Zukunft ersetzt, was der folgende Pseudocode zeigt. Die grundsätzliche Idee ist jedoch dieselbe wie beim Backtracking. Die Funktion liefert bei Erfolg eine `VariableSubstitution`, welche eine passende Ersetzung enthält, zurück, ansonsten die Leere Menge.

```
Funktion: getConsistantMapping
VariableSubstitution Ergebnis

Für jedes Atom atom aus AtomConjunction1
{
     Für jedes Atom image aus AtomConjunction2
     {
          VariableSubstitution vsTemp = atom.getMapping(image)
          Wenn Backtracking erforderlich fahre mit nächstem Atom fort
          Für alle Abbildungen aus vsTemp: Prüfe auf Konflikte mit dem
             Ergebnis
          {
               Falls es Konflikte gibt gib Null zurück
               Andernfalls füge Abbildung dem Ergebnis hinzu
          }
     }
     Falls es für Atom keine Abbildung gibt gib Null zurück
}
Gib das Ergebnis zurück
```

Der Algorithmus für das Backtracking sieht folgendermaßen aus:

```
Funktion: Backtrack
VariableSubstitution vsTempCheck = vsTemp
Für jedes Atom atomCheck aus AtomConjunction1
{
     Für jedes Atom checkImage aus AtomConjunction2
     {
          Suche eine Abbildung von atomCheck nach checkImage unter der
             Voraussetzung von vsTempCheck
          Wenn eine gültige Abbildung gefunden wurde beende die Suche
     }
     Wenn ein keine Abbildung gefunden wurde beende die Schleife
}
Wenn keine Abbildung gefunden wurde ist Backtracking erforderlich
```

Der Algorithmus soll nun anhand des vorherigen Beispiels durchgegangen werden:

Zuerst wird versucht das Atom *S(X, Y)* auf *S(1, 2)* abzubilden. Die passende Abbildung hierfür ist X→1, Y→2. Nun wird das Backtracking durchgeführt um zu überprüfen ob mit dieser Abbildung jedes Atom der Quelle ein Gegenstück im Ziel finden kann. Es wird also versucht, für das durch die Abbildung geschaffene Atom *S(1, 2)* eine Abbildung zu finden. Mit dem ersten Atom wird im Ziel auch gleich eine gefunden, daher die weitere Suche für dieses Atom abgebrochen. Nun wird mit dem zweiten Atom der Quelle, welches *R(1, 2)* lautet weiter gemacht. Im Ziel existiert jedoch nur *R(1, 3)* und daher wird keine Abbildung für dieses Atom gefunden und somit diese Abbildung nicht in das Ergebnis aufgenommen. Die zweite mögliche Abbildung ist X→1, Y→3. Auch diese wird dem Backtracking unterzogen. Das erste Atom des Ziels passt nicht zu *S(1, 3)*, das zweite jedoch schon. Nun wird nach *R(1, 3)* gesucht und dieses auch gefunden. Damit wurde für alle Atome der Quelle ein Gegenstück gefunden und somit kann die gefundene Abbildung dem Ergebnis hinzugefügt werden.
In dem Algorithmus stellt sich die Frage, wie eine Abbildung eines einzelnen Atoms auf ein anderes gefunden werden kann. Auf diese Frage wird im folgenden Abschnitt eingegangen.

5.3.2. Abbildung zwischen zwei Atomen

Um eine Verkettung von Atomen auf eine andere abzubilden ist es notwendig, die einzelnen Atome aufeinander abbilden zu können. Diese Funktionalität wird von der Funktion `getMapping` in der bereits beschriebenen Klasse `Atom` zur Verfügung gestellt.
Zuerst wollen wir jedoch untersuchen, wann eine Abbildung überhaupt möglich ist. Zuallererst kann eine Abbildung nur auf Relationen mit demselben Namen und selber Stelligkeit durchgeführt werden wie z.B. S(X) → S(1), nicht aber S(X) → R(1) oder S(X) → S(1, Y). Konstante können nur auf sich selbst abgebildet werden wie z.B. S(1) → S(1), nicht aber S(1) → S(X). Durch den Allquantor gebundene Variablen werden wie Konstanten behandelt und in der Implementierung als `VariableSubstitution` welche Werte auf sich selbst beinhaltet z.B. X→X. Es können also freie Variablen auf sich selbst, andere Variablen oder auf Konstante abgebildet werden.
Diese Variablen dürfen innerhalb des Tupels aber keine Konflikte verursachen. Ein Konflikt besteht dann, wenn eine Variable auf zwei verschiedene Variablen oder

Konstante abgebildet wird. Natürlich darf ein solcher Konflikt auch nicht zum `fixedImage` bestehen.
Der implementierte Algorithmus folgt diesen Vorgaben indem er jede Stelle untersucht und versucht, einen Term auf einen anderen abzubilden, bis entweder alle Terme abgearbeitet worden sind, oder aber ein Konflikt entstanden ist. Im Erfolgsfall liefert er eine `VariableSubstitution` zurück, im Fehlerfall Null.

5.4. Normalisierung

Neben der Minimierung der linken und rechten Seite einer Abhängigkeit kann diese auch normalisiert, also in einzelne unabhängige Bestandteile aufgeteilt werden, sofern diese existieren. Dies ist der Fall, wenn der Gaifman Graph nicht verbundene Teile enthält. Dieses Kapitel befasst sich mit der Umsetzung der Normalisierung.

Betrachten wir die folgende TGD: $S(X, Y) \rightarrow \exists V, W: P(X, V) \wedge R(Y, X, W)$. Wie man sieht sind die beiden durch den Existenzquantor quantifizierten Variablen V und W nur in jeweils einem Tupel vorhanden. Daher kann die TGD in eine äquivalente Menge von zwei TGDs aufgespaltet werden: $S(X, Y) \rightarrow \exists V: P(X, V)$ und $S(X, Y) \rightarrow \exists W: R(Y, X, W)$.
Die TGD $S(X, Y) \rightarrow \exists V, W: P(X, V) \wedge R(Y, X, W) \wedge Q(V, W)$ kann jedoch nicht aufgespaltet werden, weil das Atom Q(V, W) beide Existenzquantifizierten Variablen enthält und daher Tupel, welche entweder V oder W enthalten, nicht aufgeteilt werden können.

Der folgende Algorithmus überprüft, ob eine TGD normalisiert werden kann und liefert als Ergebnis die normalisierten Abgängigkeiten, oder die Gegebene, falls keine Normalisierung möglich ist. Der Algorithmus bildet zusammengehörige Blöcke, wobei zu Beginn jedes Atom ein eigener Block ist. Bei jedem Durchlauf wird ein Tupel entweder zu einem bestehenden Block hinzugefügt, oder bleibt in seinem Block. Wenn ein neuer Block gebildet wird, bedeutet das, dass das Atom vom Rest der TGD entfernt werden kann. Der Algorithmus ist in der Lage eine Seite einer TGD abzuarbeiten, falls man die komplette TGD (bei der Analyse von Redundanzen reicht die rechte Seite)

normalisieren will muss man demnach die Funktion zweimal ausführen, einmal für die linke und einmal für die rechte Seite.

Die Funktion zum Normalisieren ist im DependencyMinimizeUtil in der Funktion normalizeSide zu finden und erledigt genau die im Beispiel dargestellte Aufgabe. Diese Methode ist privat und kann daher nicht direkt aufgerufen werden. Dies übernimmt die Funktion normalizeDependency, welche sich auch darum kümmert, dass sowohl die Prämisse als auch die Aussage der Abhängigkeit normalisiert werden.

```
Funktion: normalizeSide
Vector bearbeitet
Zaehler = 0
Hashtable TermBlocks
Für jedes Atom t
{
      Wenn bearbeitet t enthält, continue
      tmpTuples = t
      bearbeitet += t
      suche alle Tupel t1, die dieselben freien Variablen wie t haben
           und füge sie tmpTuples und bearbeitet hinzu
      TermBlocks += (zaehler, tmpTuples)
      zaehler++
}
Vector blocks
Für jeden Schlüsselwert aus Termblocks
{
      Erstelle neue TGD aus Termblocks und füge sie blocks hinzu
}
Return blocks
```

Wie man sieht, geht der Algorithmus jedes einzelne Atom durch und betrachtet alle weiteren, ob diese dieselben Variablen wie das ursprüngliche haben und fügt diese dem Block hinzu. Auf diese Weise entstehen unabhängige Blöcke, die zum Schluss nur noch in eine oder mehrere neue TGDs gegossen werden müssen.
Die Normalisierung muss, so sie gewünscht ist, vor dem Minimieren angestoßen werden, wird also nicht automatisch bei der Minimierung durchgeführt.

5.5. Kombinatorik

In dem Framework sind verschiedene kombinatorische Fähigkeiten notwendig um beispielsweise die zu kombinierenden Abhängigkeiten zu finden, oder Spezialfälle zu erkennen.
In der Folge werden die einzelnen implementierten Funktionen erläutert, welche allesamt über das `CombinatoricsUtil` zugänglich sind. Die eigentliche Funktionalität ist im Paket `combinatorics` zusammengefasst.

5.5.1. Kombinationen finden

Ziel dieser Funktion ist es, aus einer Menge mit n Elementen alle möglichen Anordnungen zu finden, die r Elemente beinhalten. Die Reihenfolge wird nicht beachtet und die einzelnen Elemente werden nicht wiederholt.

Beispielsweise sind alle Kombinationen mit drei Elementen aus {A, B, C, D} die folgenden:
ABC, ABD, ACD, BCD. Bei zwei Elementen wären alle Kombinationen AB, AC, AD, BC, BD, CD. Zu beachten ist, dass natürlich ein r größer als die die Anzahl der Eingabeelemente oder kleiner als null nicht möglich ist. Im Fall dass r=0 ist wird die Leere Menge zurückgeliefert, bei r=1 werden die einzelnen Elemente der Menge zurückgeliefert und die Lösungsmenge enthält n Elemente, in diesem Fall also A, B, C und D. Sollte r gleich der Anzahl der Elemente sein, gibt es nur eine Lösung, die alle Elemente beinhaltet, in unserem Beispiel wäre das ABCD. Für jedes r gibt es $\binom{n}{r}$ mögliche Kombinationen und daher insgesamt $\sum_{r=0}^{n}\binom{n}{r}$ ($=2^n$) mögliche Kombinationen, was in unserem Fall $1 + 4 + 6 + 4 + 1 = (2^4) = 16$ wären. Bei zehn Elementen wäre es insgesamt bereits 1024 mögliche Kombinationen.

In der Implementierung ist diese Funktionalität im `CombinationGenerator` untergebracht und entstammt [8]. Der verwendete Algorithmus der Implementierung wurde in [19] beschrieben.
Die Implementierung erwartet im Konstruktor ein Array mit den Elementen und liefert mit jedem Aufruf der Funktion `getNext()` ein Array mit Integer-Werten in welchem die Indexwerte aus dem Originalarray ersichtlich sind. Als Beispiel dient hier wieder die vorherige Menge

Index	0	1	2	3
Wert	A	B	C	D

Das Ergebnis wäre für den Fall ABD beispielsweise ein Feld mit den Indizes {0, 1, 3}.

Beim Initialisieren wird zuerst die gesamte Anzahl an möglichen Kombinationen $\binom{n}{r}$ berechnet und die noch möglichen Kombination festgesetzt, sowie die erste Kombination a[], welche aus den Indizes 0…r besteht. Die Funktion `hasMore()` gibt an, ob es eine weitere Kombinationen gibt und die Funktion `getNext` berechnet diese, wobei der Algorithmus der folgende ist:

```
Funktion: getNext()
Bei der ersten Abfrage reduziere die verbleibenden Kombinationen um 1
   und liefere a zurück
i = r -1
reduziere i solange bis a[i] ungleich n-r+i ist
erhöhe den Wert in i um eins
j = i + 1
so lange j < r
{
     a[j] = a[i] + j - i
     erhöhe j um eins
}
reduziere die Anzahl der noch möglichen Kombinationen um eins
returniere a
```

Der erste Aufruf mit unserer Beispielmenge und einem r = 3 würde folgendermaßen ablaufen: die Anzahl von Kombinationen ist 4 und das erste a[] = 0, 1, 2, das Ergebnis

(lt. Tabelle) demnach ABC. Beim ersten Aufruf von getNext() würde dieses Ergebnis zurückgeliefert. Der zweite Aufruf setzt i zuerst auf den Wert 2. Die Abbruchbedingung ist sofort erfüllt, i bleibt also 2. Der Index-Wert an dieser Stelle wird daher um eins erhöht und ist nun 3. j ist gleich r, damit wird nichts mehr getan und a zurückgegeben. Beim nächsten Durchlauf ist i am Beginn wieder 2. Diesmal wird i um eins reduziert und der Wert an dieser Stelle um eines erhöht. j ist diesmal kleiner als r, der Wert an der Stelle zwei auf den Wert 3 gesetzt, die noch vorhandenen Kombinationen um eins reduziert und das Ergebnis zurückgeliefert. Der letzte Durchlauf erfolgt nach demselben Schema, die Werte sind in der nachstehenden Tabelle zusammengefasst:

Durchlauf	a[0]	a[2]	a[5]	i
1	0	1	2	-
2	0	1	3	2
3	0	2	3	1
4	1	2	3	0

Die Klasse `CombinatoricsUtil` vereinfacht die Verwendung weiter, indem bei Aufruf der generischen Funktion `getCombinationsWithoutRepetition(testVektor, 3)` in einem Vector alle möglichen Kombinationen mit der gewünschten Länge zurückgegeben werden. Der Vektor würde für unser Beispiel demnach 4 weitere Vektoren, welche die oben genannten Lösungen der Länge 3 beinhalten.

5.5.2. Variationen finden

Variationen sind Kombinationen, deren Reihenfolge beachtet und deren Elemente öfter vorkommen dürfen. Ziel dieser Funktion ist es aus einer Menge mit n Elementen alle möglichen Anordnungen zu finden, die r Elemente beinhalten.

Betrachten wir wieder die Menge aus dem vorherigen Kapitel {A, B, C, D} mit n (=4) Elementen. Die Variationen mit der Länge r = 2 für diese Menge sind AA, AB, AC, AD, BA, BB, BC, BD, CA, CB, CC, CD, DA, DB, DC und DD. Im Fall r = 0 wird wie im vorherigen Kapitel die Leere Menge geliefert, für r = 1 ist das Ergebnis mit A, B, C,

D ebenfalls dasselbe. Anders als im vorherigen Kapitel ist hier ein r, welches größer als die Anzahl der Elemente ist erlaubt, weil sich die einzelnen Elemente ja wiederholen dürfen. Wir haben für jedes r n^r Lösungen. Für das obige Beispiel $4^2 = 16$ bei r = 3 wären es 64 und für r = 4 bereits 256 Lösungen. Für große n bzw. vor allem r ist das Berechnen aller Variationen problematisch, so gibt es beispielsweise für n = 9 und r = 10 bereits 3.486.784.401 (≈ 3,5 Milliarden) Möglichkeiten.

Der Algorithmus berechnet rekursiv einen Vector, der alle möglichen Variationen enthält.

```
Funktion: getNext()
Vector<T[]> returnTerms
T[] tmpTerm = berechnetesObject
Für alle i zwischen 0 und Anzahl der Elemente
{
      tmpTerm[aktuellePosition] = Objects[i]
      Wenn aktuelle Position die letzte Position ist
           Füge tmpTerm zum Ergebnis hinzu
      Sonst
           Erhöhe aktuelle Position um eins
           Führe Funktion für nächste Position rekursiv aus
}
Gib returnTerms zurück
```

Vereinfacht gesprochen fängt der Algorithmus an der ersten Position des Ergebnisses an und setzt an dieser Stelle alle angegebenen Werte und ruft mit jedem dieser Werte sich selbst auf um wieder alle Werte an der nächsten Position durchzuprobieren. Der Algorithmus läuft so baumförmig auseinander, bis alle Stellen des Ergebnisses gefüllt sind und liefert dann alle Ergebnisse zurück. Die untenstehende Tabelle folgt beispielhaft jeweils dem aller weitest links stehendem Pfad der Ausführung.

Rekursion	Werte	Werte	Werte	Werte		
0	a[A, ., ., .]	a[A, ., ., .]	a[A, ., ., .]	a[A, ., ., .]		
1	a[A, A, ., .], a[A, B, ., .], a[A, C, ., .], a[A, D, ., .]			…	…	…

2	a[A, A, A, .], a[A, A, B, .], a[A, A, C, .], a[A, A, D, .]	…	…	…
3	a[A, A, A, A], a[A, A, A, B], a[A, A, A, C], a[A, A, A, D]	…	…	…

Das `CombinatoricsUtil` hält wieder eine generische Funktion zum einfacheren Umgang bereit, welche mit `getCombinations(…)` aufgerufen werden kann und einen Vektor mit allen Ergebnissen als Array zurückliefert.

5.5.3. Kombinationen mit Wiederholungen finden

Ziel dieser Funktion ist es aus einer Menge mit n Elementen alle möglichen Anordnungen zu finden, die r Elemente beinhaltet. Hierbei dürfen Elemente öfter vorkommen, jedoch ist deren Reihenfolge nicht von Bedeutung.
Wir betrachten wieder die Menge {A, B, C, D} mit n (=4) Elementen. Die Kombinationen mit Wiederholungen wären für r = 2 wären in diesem Fall AA, AB, AC, AD, BB, BC, BD, CC, CD und DD. Die Ergebnisse für r = 0 und r = 1 decken sich mit denen der vorherigen Kapitel, nämlich zum einen die Leere Menge und zum anderen die einzelnen Elemente. Es ist erlaubt, ein r zu wählen, das Größer als n ist, weil die einzelnen Elemente öfter vorkommen dürfen. Für jedes r gibt es $\binom{n+r-1}{r}$ verschiedene Möglichkeiten, was für r = 2 zehn Lösungen bedeutetet. Für r = 3 wären es 15 und für r = 4 wären es 21 Möglichkeiten.

Der implementierte Algorithmus setzt auf dem zuvor beschriebenen Algorithmus zum Berechnen der Variationen auf und ist ebenfalls in der Klasse `CombinationsGeneratorRepetiton` beheimatet. Das verwenden des Algorithmus für Variationen hat den Nachteil dass zuerst eine Vielzahl von Möglichkeiten berechnet wird, welche zum großen Teil wieder verworfen werden müssen. So werden für unser Beispiel bei r = 4 256 Varianten berechnet, aber nur 21 davon sind auch im Ergebnis zu finden.
Der implementierte Algorithmus arbeitet folgendermaßen:

```
Funktion: computeSortedCombinations
Für jede Variation v:
{
      Sortiere die Elemente in v der Größe nach
      Wenn die sortierte Variation noch nicht im Ergebnis e vorhanden
         ist, füge sie hinzu
}
Gib das Ergebnis zurück
```

Aufgrund des Sortierens ergibt sich ein weiterer Nachteil: Die Elemente müssen sortierbar sein, was zwar auf Zahlen und Zeichenketten, im Allgemeinen aber nicht auf Objekte zutrifft. Für die darauf aufbauenden Funktionen sind beide Nachteile jedoch nicht weiter dramatisch, wobei der zweite recht einfach durch das Verwenden von Indizes umgangen werden kann, wie es auch der Algorithmus zum Finden von Kombinationen tut.

Für unser Beispiel würden also zuerst die Varianten AA (AA), AB (AB), AC (AC), AD (AD), ~~BA (AB)~~, BB (BB), BC (BC), BD (BD), ~~CA (AC)~~, ~~CB (BC)~~, CC (CC), CD (CD), ~~DA (AD)~~, ~~DB (BD)~~, ~~DC (CD)~~ und DD (DD) generiert. In den Klammern stehen diese sortiert, wobei die durchgestrichenen Varianten nicht Teil des Ergebnisses sind, jede Variante also nur einmal im Ergebnis vorkommt. Der Algorithmus überprüft demnach, ob eine sortierte Variante bereits vorhanden ist und fügt diese gegebenenfalls dem Ergebnis hinzu.

Die Klasse `CombinatoricsUtil` hält die Funktion `getSortedCombinations(Vector werte)` bereit, welche die Verwendung der Funktionalität vereinfacht und als Ergebnis einen Vector mit Arrays aller Ergebnisse zurückliefert. Zu beachten ist, dass die verwendeten Werte von `Comparable` erben müssen, weil die Werte ja immer sortiert werden müssen.

5.5.4. Permutationen ohne Wiederholungen finden

Diese Funktion hat den Zweck, alle möglichen Anordnungen von n Elementen zu finden, in denen alle diese Elemente verwendet werden und kein Element doppelt vorkommt. Die Permutationen für unser Beispiel (A, B, C, D) mit n = 4 sind ABCD, ABDC, ACBD, ACDB, ADBC, ADCB, BACD, BADC, BCAD, BCDA, BDAC, BDCA, CABD, CADB, CBAD, CBDA, CDAB, CDBA, DABC, DACB, DBAC, DBCA, DCAB und DCBA. Es gibt somit $n!$ viele Möglichkeiten, die Elemente zu permutieren. Für n = 4 wären dies 1*2*3*4 = 24 Varianten, bei n = 6 bereits 720 verschiedene Arten. Entsprechend ist es problematisch, n besonders groß werden zu lassen. Der Algorithmus entstammt [9] und wird in [19] beschrieben, wird aber zurzeit von keiner Funktion verwendet.

```
Funktion: getNext()
Finde den größten Index j sodass a[j] < a[j+1]
Finde Index k dass a[k] die kleinste ganze Zahl größer als a[j] auf
   der rechten Seite von a[j] ist
Tasche a[j] mit a[k] aus
Füge Ende der Permutation nach der j-ten Position in aufsteigender
   Reihenfolge ein
Reduziere die noch möglichen Permutationen um eins
Gib a zurück
```

Zum einfacheren Verwenden gibt es in der Klasse `CombinatoricsUtil` die Funktion `getPermutations(Vector)`, welche einen Vektor mit allen möglichen Permutationen zurückgibt.

5.6. Spezialisierte Versionen berechnen, die bessere Ergebnisse liefern

In diesem Kapitel wollen wir untersuchen, wie alle Spezialisierungen einer einzelnen Abhängigkeit berechnet werden können. Wir betrachten wieder das Beispiel von Fagin [5], dessen Spezialisierungen bereits beschrieben wurden, wobei wir uns in dieser Betrachtung auf die erste TGD beschränken:

TGD1: $S(a, b, c, d) \rightarrow \exists x_1 \exists x_2 \exists x_3 \exists x_5 (R(x_5, b, x_1, x_2, a) \wedge$
$R(x_5, c, x_3, x_4, a) \wedge$
$R(d, c, x_3, x_4, b))$

Wie bereits beschrieben gibt es für diese TGD die folgenden Spezialisierungen:

abcd

aacd abad abca abbd abcb abcc

aaad abaa aaca abbb

aaaa

Wir müssen zwar alle möglichen Spezialfälle einmal betrachten, für die weitere Betrachtung sind jedoch nur Kombinationen interessant, deren Minimierung weniger Tupel enthält als höhere Schichten. Wir müssen alle Spezialfälle betrachten und diese dann minimieren, wobei wir beim allgemeinsten Fall (abcd) beginnen und immer weiter spezialisieren, bis wir bei aaaa angelangt sind. Alle besseren Spezialfälle speichern wir, den Rest verwerfen wir.

Für unser Beispiel würden folgende Spezialfälle gespeichert:

Schicht 1	$S(a, b, c, d) \rightarrow R(x_5, b, x_1, x_2, a) \wedge R(x_5, c, x_3, x_4, a) \wedge R(d, c, x_3, x_4, b)$
Schicht 2	$S(a, b, b, d) \rightarrow R(x_5, b, x_3, x_4, a) \wedge R(d, b, x_3, x_4, b)$
Schicht 3	$S(a, a, a, d) \rightarrow R(d, a, x_3, x_4, a)$

Zwei Spezialfälle, welche im Vergleich zur ursprünglichen TGD ein besseres Ergebnis liefern sind hier nicht angeführt: S(a, b, b, b) und S(a, a, a, a). Erstere ist neben dem originalen Fall eine Spezialisierung von S(a, b, b, d) und wäre in Schicht 3 anzutreffen. Die Anzahl der Atome in der Aussage ist aber nicht weniger als die von Schicht 2, weswegen dieser Fall nicht abgespeichert wird. Ähnlich verhält es sich mit dem speziellsten Fall S(a, a, a, a), welcher eine Spezialisierung aller Schichten ist und in Schicht 4 anzutreffen wäre. Die Aussage besitzt aber gleich viele Tupel wie der Fall S(a, a, a, d) weswegen wir diesen Fall nicht speichern müssen.
Wenn wir in einer Quelldatenbank das Tupel (1, 2, 2, 4) vorfinden würden, würde zuerst in Schicht 3 nachgesehen, ob das Muster zur Eingabe passt, was nicht der Fall ist, also

wird in der nächst höheren Schicht nachgesehen, ob das Schema zur Eingabe passt. In diesem Fall haben wir Glück, das Schema abbd passt zu unserem Tupel, daher kann der Chase entsprechend mit diesem Spezialfall durchgeführt werden.

Ein Algorithmus, welcher die Suche nach Spezialfällen und die Schichteinteilung durchführt, wurde von Vadim Savenkov aufbauend auf dem Algorithmus zum Finden von Kombinationen und dem Minimieren von Abhängigkeiten implementiert und liefert als Ergebnis eine Einteilung in die entsprechenden Schichten. Dieser Idee des Algorithmus wird nun kurz vorgestellt.

```
Funktion: getSpecializedVariants
DependencyLattice lattice
Minimiere ursprüngliche Abhängigkeit und füge sie lattice in Schicht 1
   hinzu
für alle Schichten < Anzahl relevanter Variablen
{
      Setzte alle Kombinationen mit Länge der aktuellen Schicht gleich
      Wenn Minimierung weniger Tupel enthält als darunterliegende
         allgemeinere Schichten füge Spezialisierung der aktuellen
         Schicht hinzu
}
Gib lattice zurück
```

Die Implementierung dieses Algorithmus ist in der Klasse `DependencyMinimizeUtil` in der Funktion `getSpecializedVariants` zu finden, welche als Argumente die zu untersuchende Abhängigkeit und optional einen Wahrheitswert, welcher angibt dass alle Spezialisierungen (also auch die, welche nicht besser sind) gespeichert werden sollen. Im Standardfall werden nur bessere Varianten gespeichert.

5.7. Zusammenfassen von Abhängigkeiten

Bis jetzt haben sich alle Betrachtungen der Implementierung um isolierte einzelne Abhängigkeiten gedreht. Der nächste Schritt ist nun, das Zusammenspielen mehrerer Abhängigkeiten miteinander zu untersuchen. In diesem Kapitel beschäftigen wir uns nur

mit dem Kombinieren einer Basis-TGD mit mehreren anderen und erhalten so eine Vielzahl speziellerer Varianten der Basis-TGD. Das nächste Kapitel beschäftigt sich dann mit dem Minimieren dieser Abhängigkeiten.

Wir müssen als ersten Schritt alle möglichen Varianten finden, wie Abhängigkeiten untereinander in Verbindung stehen. Im nächsten Schritt müssen wir dann alle möglichen Variablenspezialisierungen durchspielen.

Betrachten wir das folgende Beispiel mit den folgenden Abhängigkeiten:

TGD1: $S(X, Y) \rightarrow R(X, Y) \wedge R(Y, 2) \wedge P(Y, Z)$
TGD2: $Q(A) \rightarrow R(A, 2)$

Wir generieren als nächstes die möglichen Kombinationen der Abhängigkeiten:

Kombinationen mit 2 Abhängigkeiten:

TGD1 mit sich selbst:

$S(X, Y), S(X1, Y_1) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z),$	$R(X_1,Y_1), P(Y_1,Z_1), P(2, Z_1)$
$S(X, Y), S(X, Y_1) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z),$	$R(X,Y_1), P(Y_1,Z_1), P(2, Z_1)$
$S(X, Y), S(X_1, Y) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z),$	$R(X_1,Y), P(Y_1,Z_1), P(2, Z_1)$
…		
$S(X, X), S(X, X) \rightarrow$	$R(X, X), R(X, Y), P(2, Z),$	$R(X, X), P(X, Y), P(2, Z_1)$

TGD1 mit der Annahme, dass *TGD2* auch zutrifft:

$S(X,Y), Q(X_1) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z),$	$R(X_1, 2)$
$S(X,Y), Q(X) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X, 2)$
$S(X,Y), Q(Y) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(Y, 2)$
$S(X,X), Q(X) \rightarrow$	$R(X, X), P(X, Z), P(2, Z)$	$R(X, 2)$

Kombinationen mit 3 Abhängigkeiten:

3x *TGD1* mit sich selbst:

$S(X,Y), S(X_1, Y_1), S(X_2, Y_2) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X_1, Y_1), P(Y_1, Z_1), P(2, Z_1)$	$R(X_2, Y_2), P(Y_2, Z_2), P(2, Z_2)$
$S(X,Y), S(X, Y_1), S(X_2, Y_2) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X, Y_1), P(Y_1, Z_1), P(2, Z_1)$	$R(X_2, Y_2), P(Y_2, Z_2), P(2, Z_2)$
…			
$S(X, X), S(X, X), S(X, X) \rightarrow$	$R(X, X), P(X, Z), P(2, Z)$	$R(X, Y), P(Y, Z_1), P(2, Z_1)$	$R(X, X), P(X, Z_2), P(2, Z_2)$

2x *TGD1* und 1x *TGD1*:

$S(X, Y), S(X_1, Y_1), Q(X_2) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X_1, Y_1), P(Y_1, Z_1), P(2, Z_1)$	$R(X_2, 2)$
$S(X, Y), S(X_1, Y_1), Q(X) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X_1, Y_1), P(Y_1, Z_1), P(2, Z_1)$	$R(X, 2)$
$S(X, Y), S(X_1, Y_1), Q(Y) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X_1, Y_1), P(Y_1, Z_1), P(2, Z_1)$	$R(Y, 2)$
…			
$S(X, X), S(X, X), Q(X) \rightarrow$	$R(X, X), R(X, Y), P(2, Z)$	$R(X, X), R(X, Y), P(2, Z_1)$	$R(X, 2)$

1x *TGD1* und 2x *TGD2*

$S(X, Y), Q(X_1), Q(X_2) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X_1, 2)$	$R(X_2, 2)$
$S(X, Y), Q(X), Q(X_2) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X, 2)$	$R(X_2, 2)$
$S(X, Y), Q(X_1), Q(Y) \rightarrow$	$R(X, Y), P(Y, Z), P(2, Z)$	$R(X_1, 2)$	$R(Y, 2)$
…			
$S(X, X), Q(X), Q(X) \rightarrow$	$R(X, X), R(X, Y), P(2, Z)$	$R(X, 2)$	$R(X, 2)$

Wie man aus dem recht einfachen Beispiel erahnen kann, steigt die Anzahl der verschiedenen Spezialisierungen rapide an. Es müssen jedoch nur Abhängigkeiten betrachtet werden, welche in der Aussage übereinstimmende Tupelnamen haben. Wenn nicht mindestens ein Tupelname in beiden Abhängigkeiten vorkommt, kann nichts minimiert werden, wie das nächste Kapitel zeigt.

Die Funktionen zum Finden der Kombinationen sind in der Klasse `DependencyMergeUtil` untergebracht. Zum Aufruf stehen die Funktionen `getConditionalDependencies` und `getConditionalDependenciesWithDepVariants` zur Verfügung. Der Unterschied zwischen den beiden ist, dass erstere die Abhängigkeit, mit der die angegebenen Abhängigkeiten kombiniert werden sollen nicht in das Ergebnis aufnimmt und die zweite schon.
Hier wird in der Folge der Algorithmus zum Zusammenfügen mehrerer Abhängigkeiten vorgestellt, wobei zuerst die Funktion `getConditionalDependencies` beschrieben wird.

```
Funktion: getConditionalDependencies
Vector dependencies
Vector dependencyCombIndexes  = alle Indizes der Varianten der zu
   kombinierenden Abhängigkeiten mit Länge <= maxConditions

Hashtable hash = Index, Vector mit alle Spezialfälle der Abhängigkeit

Für jeden Indexsatz aus dependencyCombIndexes
{
      Für jede Liste von Abhängigkeit, die mappingCombinator(index,
         hash) erstellt
      {
           dependencies += Abhängigkeiten in eine Abhängigkeit
              zusammengefügt
      }
}
Gib dependencies zurück
```

Die Funktion `getConditionalDependenciesWithDepVariants` kombiniert zusätzlich noch alle Varianten der Basisabhängigkeit zu den zu kombinierenden Abhängigkeiten:

```
Funktion: getConditionalDependenciesWithDepVariants
Vector dependencies
Vector dependencyCombIndexes  = alle Indizes der Varianten der zu
   kombinierenden Abhängigkeiten mit Länge <= maxConditions

Hashtable hash = Index, Vector mit alle Spezialfälle der Abhängigkeit


Für jeden Indexsatz aus dependencyCombIndexes
{
      Für jeden Spezialfall s der Basisabhängigkeit
      {
           Für jede Liste von Abhängigkeit, die
              mappingCombinator(index, hash) erstellt
           {
                dependencies += s und Abhängigkeiten in eine
                   Abhängigkeit zusammengefügt
           }
      }
}
Gib dependencies zurück
```

Beide Algorithmen benötigen die Hilfsfunktion `MappingCombinator`. Diese berechnet rekursiv alle Abbildungen von einer Liste von Indizes.

Wie das Beispiel oben zeigt ist es wichtig, frische Variablennamen generieren zu können. Diese Funktionalität wird in der Klasse `Term` von der statischen Funktion `freshVariable()` zur Verfügung gestellt. Diese wird beim Berechnen von `hash` Aufgerufen und die Variablen einer Abhängigkeit entsprechend umbenannt.

In diesem Kapitel wurde dargestellt, wie alle möglichen Kombinationen von Abhängigkeiten generiert werden können, doch sind wir in der Regel nur an kombinierten Abhängigkeiten interessiert, welche in der Zielmenge weniger Tupel generieren. Wie dies bewerkstelligt wird, zeigt das nächste Kapitel.

5.8. Minimieren von Abhängigkeiten

Dieser Abschnitt veranschaulicht, wie die im vorherigen Kapitel vorgestellten kombinierten Abhängigkeiten minimiert werden können. Diese Minimierung ist notwendig, um nicht unnötig viel Speicherplatz für Spezialfälle zu verbrauchen, die ohnedies nicht weniger Tupel im Ziel einfügen. Außerdem muss dann auch der nachfolgende Chase nicht viele unnötige Spezialfälle untersuchen und kann daher schneller arbeiten.

Betrachten wir aus dem Beispiel des vorherigen Kapitels die Kombination von TGD1 und TGD2, bei der die benannte Variable aus TGD2 die selbe ist wie eine der Variablen aus TGD1, bei einer Eingabemenge also die selben Werte vorliegen:
$S(X, Y) \wedge R(X) \rightarrow R(X, Y) \wedge P(Y, Z) \wedge P(2, Z) \wedge R(X, 2)$
Für uns interessant sind nur die Atome, welche von TGD1 stammen, also R(X, Y), P(Y, Z), P(2, Z) und wollen überprüfen, ob es eine Abbildung gibt, welche deren Anzahl reduziert. Hier erschließt sich die Sinnhaftigkeit, bei der Minimierung einer Abhängigkeit die Atome angeben zu können, welche minimiert werden sollen.
In diesem Fall gibt es eine solche Abbildung, nämlich $Y \rightarrow 2$. Die TGD würde demnach folgendermaßen aussehen:
$S(X, 2) \wedge R(X) \rightarrow$ ~~$R(X, 2) \wedge$~~ $P(2, Z) \wedge$ ~~$P(2, Z) \wedge$~~ $R(X, 2)$
Wie zu sehen ist, sind die durchgestrichenen Atome redundant und können eliminiert werden, die resultierende TGD wäre also $S(X, 2) \wedge R(X) \rightarrow P(2, Z) \wedge R(X, 2)$. Wir sind in der Aussage jedoch nur an Atomen interessiert, welche zur Basisabhängigkeit gehören. Es bleibt also von der TGD nur $S(X, 2) \wedge R(X) \rightarrow P(2, Z)$ für den Fall, dass auch TGD2 ausgeführt wird, übrig.

Implementiert wurde diese Funktionalität im `DependencyMinimizeUtil` in der Methode `getMinimizedDependencies2(TGD dependency, TGD[] conditionals)`. Diese verwendet so gut wie alle bisher beschriebenen Funktionen und berechnet für eine gegebene Abhängigkeit und gegebene zu kombinierende Abhängigkeiten die TGDs, welche wie das obige Beispiel die Anzahl der Atome in der Aussage reduzieren. Es ist

auch möglich, dass in der Aussage gar keine Atome mehr stehen. In diesem Fall ist die Kombination komplett anderswo vorhanden. Die TGD `dependency` ist hier die Basis-TGD, mit der alle `conditionals` unter Einbeziehung aller Spezialfälle kombiniert werden sollen.
Der Ablauf der Methode ist dem einer Minimierung sehr ähnlich. Als Eingabe muss die Basisabhängigkeit und die zu kombinierenden Abhängigkeiten angegeben werden, das Ergebnis ist die bereits beschriebene Einteilung in Schichten.

```
Funktion: getMinimizedDependencies2
List relevantDeps
Füge bessere Spezialfälle von dependency relevantDeps hinzu
Für Spezialfälle t der dependency
{
      Origsize = t.Conclusion.size
      Für alle Kombinationen k der zu kombinierenden Abhängigkeiten
      {
           toMin = t kombiniert mit k
           minimiere toMin
           entferne Atome, die nicht von t sind aus toMin
           wenn toMin.size < origsize
                relavantDeps += toMin
      }
}
Returniere neue DependencyLattice(relevantDeps)
```

Der Algorithmus fügt also zuerst die besseren Spezialfälle der Basisabhängigkeit `relevantDeps` hinzu und untersucht dann alle Spezialfälle derer mit allen Variationen von Spezialfällen, welche die im vorherigen Kapitel beschriebene Funktion berechnet. Der Wert `maxConditions` ist die Anzahl der Atome der rechten Seite. Nach Minimierung der neuen Abhängigkeit und entfernen aller Atome, die nicht zu der Basisabhängigkeit gehören, wird überprüft, ob sich eine Verbesserung zum Basis(spezial)fall einstellt. Sollte dies der Fall sein, wird die entstandene TGD in `relevantDeps` aufgenommen. Zum Schluss wird mit `DependencyLattice` eine baumartige, in Schichten eingeteilte Struktur zurückgegeben, die die besseren Spezialfälle enthält.

Selbstverständlich muss für jede Basisabhängigkeit solch eine Struktur erstellt werden, was einen nicht unerheblichen Rechenaufwand bedeutet. Diese Struktur ist jedoch erforderlich um einen Chase, welcher in der Lage ist direkt den Kern zu erzeugen überhaupt entwickeln zu können.

5.9. Graphische Darstellungen

Dieses Kapitel gibt mittels zweier Diagramme Auskunft über den Aufbau und den Ablauf im implementierten System.

5.9.1. Klassendiagramm

Das nachfolgende Klassendiagramm (Abbildung 7) bildet die wichtigsten Klassen und Methoden des Systems ab.

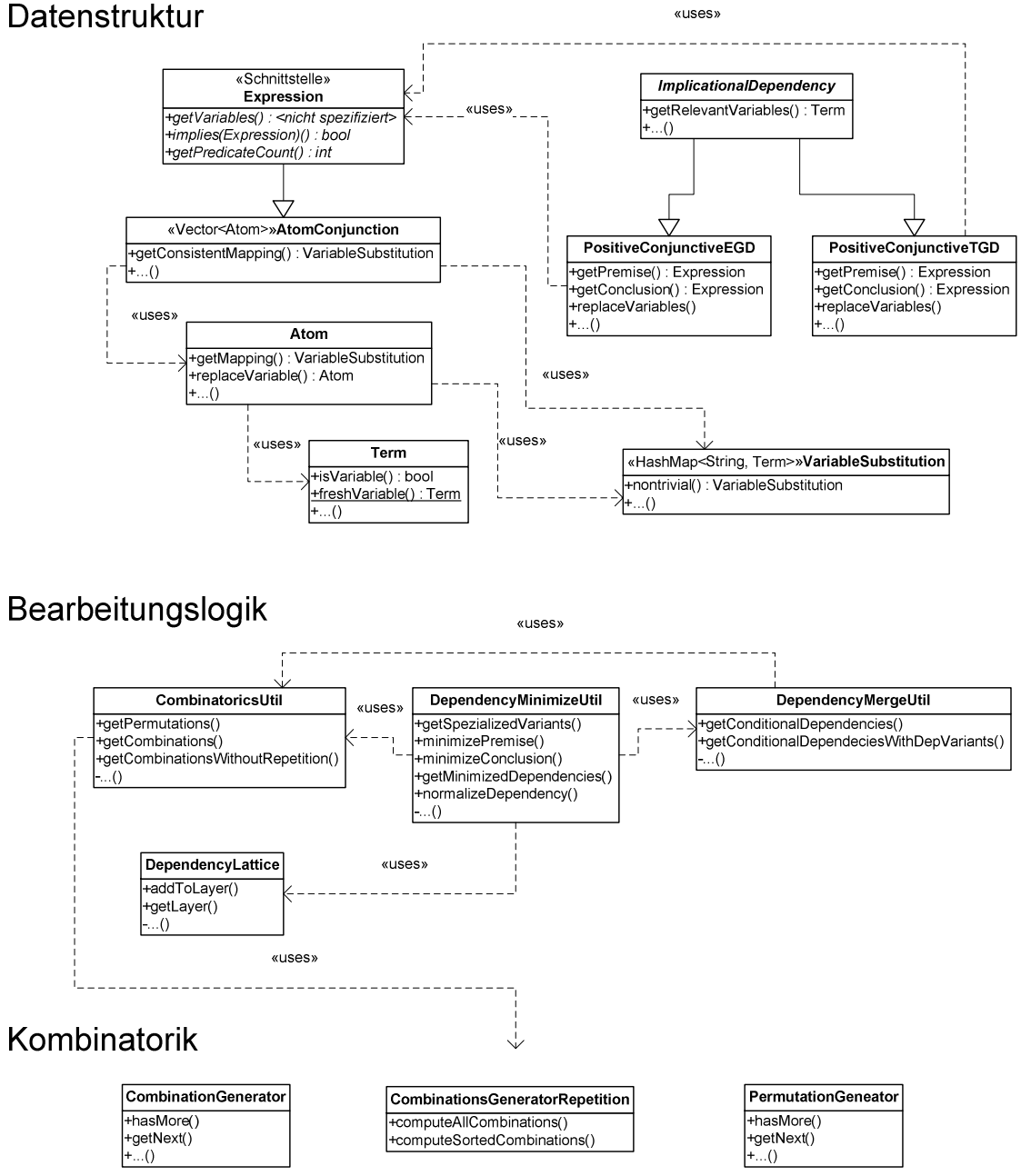


Abbildung 7: Klassendiagramm

5.9.2. Grober (interner) Ablauf – Ablaufdiagramm

Das folgende Ablaufdiagramm (Abbildung 8) zeigt exemplarisch, welche Methoden zusammenspielen, um die zuletzt beschriebene Minimierung von Abhängigkeiten durchzuführen. Der Ablauf ist sehr stark vereinfacht und soll lediglich einen Überblick

verschaffen, welche wichtigen Teile an der Minimierung mehrerer Abhängigkeiten notwendig sind.

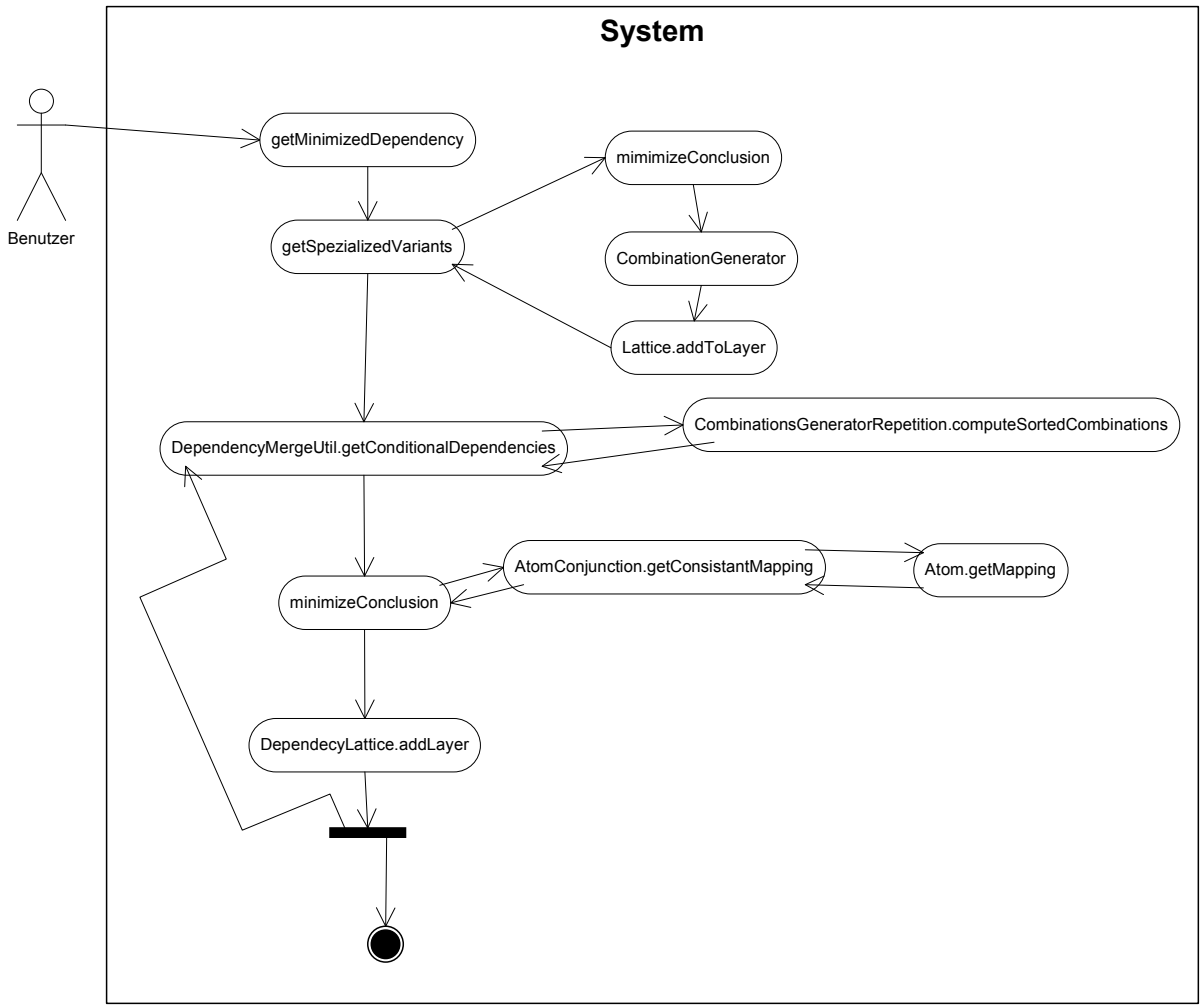


Abbildung 8: Ablaufdiagramm

5.10. Beispielhafte Verwendung

Dieses Kapitel befasst sich damit, wie einzelne Funktionen der Implementierung verwendet werden können und welche Ergebnisse sie liefern. Wir gehen bei der Beschreibung vom Ablauf her analog zu der Beschreibung vor.

5.10.1. Minimierung einer Abhängigkeit

Die Funktionalität zum Minimieren einer Abhängigkeit ist im `DependencyMinimizeUtil` zu finden. Je nachdem ob man die Prämisse oder die Aussage einer Abhängigkeit minimieren will bietet entweder die Funktionen `minimizePremise(PositiveConjunctiveTGD tgd)` und `minimizePremise(PositiveConjunctiveTGD tgd, Set<Atom> importantAtoms)` oder `minimizeConclusion(PositiveConjunctiveTGD tgd)` bzw. `minimizeConclusion(PositiveConjunctiveTGD tgd, Set<Atom> importantAtoms)` die gewünschte Funktionalität.

`PositiveConjunctiveTGD` ist jeweils die TGD, von der eine der Seiten minimiert werden soll. Optional kann im `Set<Atom>` definiert werden, welche Atome zum Minimieren berücksichtigt werden sollen. Als Rückgabe liefern alle Funktionen eine `VariableSubstitution`, also eine Ersetzung von Variablen durch andere, welche dann auf die TGD angewendet wird.
Das folgende Beispiel zeigt, wie man die Aussage einer TGD, welche in unserem Fall zwei redundante Atome enthält minimieren kann.

Codebeispiel: Minimierung einer Abhängigkeit

```
//TGD definieren
PositiveConjunctiveTGD TGD_MIN =
   (PositiveConjunctiveTGD)DependencyParser.parse(
   "S(X,Y) -> P(X,V,W),P(X,Y,2),P(Z,Y,W)");

//Variablenersetzung zum Minimieren der TGD
VariableSubstitution vs =
   DependencyMinimizeUtil.minimizeConclusion(TGD_MIN);

//Variablen Ersetzen
TGD_MIN = (PositiveConjunctiveTGD)TGD_MIN.replaceVariables(vs);

//Ergebnis ausgeben
system.out.println(TGD_MIN);

--- Ausgabe ---
S(X,Y) -> P(X,Y,2)
```

5.10.2. Normalisieren einer Abhängigkeit

Die Implementierung zum Normalisieren einer Abhängigkeit befindet sich, wie auch die Funktionalität zum Minimieren einer TGD, im `DependencyMinimizeUtil` in der Funktion `normalizeDependency(PositiveConjunctiveTGD dependency)`. Die Methode benötigt als einzigen Wert eine TGD, die normalisiert werden soll. Als Rückgabewert wird ein `Vector<PositiveConjunctiveTGD>`, also eine Liste mit neuen TGDs geliefert. Wenn die TGD nicht weiter normalisiert werden kann, enthält die Liste die Eingabe-TGD.
Das folgende Beispiel zeigt, wie eine TGD normalisiert werden kann, also in mehrere TGDs aufgeteilt werden kann.

Codebeispiel: Normalisieren einer Abhängigkeit

```
//TGD definieren
PositiveConjunctiveTGD TGD_REP =
   (PositiveConjunctiveTGD)DependencyParser.parse(
   "T(A,B,C) -> Q(Y,2),R(3,X,Y),P(X,B),R(3,B,A)");

//TGD normaliesieren
Vector<PositiveConjunctiveTGD> norm =
   DependencyMinimizeUtil.normalizeDependency(TGD_REP);

//Ergebnis ausgeben
System.out.println(norm);

--- Ausgabe ---
[T(A,B,C) -> R(3,B,A),
T(A,B,C) -> Q(Y,2), R(3,X,Y), P(X,B)]
```

5.10.3. Spezialisierte Versionen einer Abhängigkeit berechnen

Um die baumartige Struktur zu erhalten, welche die in Schichten angeordneten Spezialisierungen (also verschiedene Eingabemuster) einer Abhängigkeit beinhaltet, stehen im `DependencyMinimizeUtil` die Methoden

`getSpecializedVariants(PositiveConjunctiveTGD dependency)` und

```
getSpecializedVariants(PositiveConjunctiveTGD dependency,
                       boolean smallerConclusionsOnly)
```

zur Verfügung. Als Eingabe wird zumindest eine TGD erwartet. Optional kann mittels eines Wahrheitswertes angegeben werden, ob alle Spezialfälle, oder nur welche, die ein besseres Ergebnis als höher liegende Schichten gespeichert werden sollen. Letzteres ist das Standardvorgehen. Als Ergebnis liefert die Funktion eine `DependencyLattice`, welche die Spezialfälle enthält.

Im folgenden Beispiel ist zu sehen, wie die Funktion verwendet werden kann.

Codebeispiel: Spezialisierte Versionen einer TGD berechnen

```
//TGD definieren
PositiveConjunctiveTGD tgd =
   (PositiveConjunctiveTGD)DependencyParser.parse(
   "S(X,Y,Z)->R(X,Y), R(Y,X), R(Z,X)");

//Spezialfälle berechnen
DependencyLattice lattice =
   DependencyMinimizeUtil.getSpecializedVariants(tgd);

//Ergebnis ausgeben
PrettyPrinter.print(lattice)

--- Ausgabe ---
Layer 1 ++++++++++++++++
        S(X,Y,Z) -> R(Z,X), R(Y,X), R(X,Y)
Layer 2 ++++++++++++++++
        S(X,X,Z) -> R(X,X), R(Z,X)        S(X,Y,Y) -> R(X,Y), R(Y,X)
Layer 3 ++++++++++++++++
        S(X,X,X) -> R(X,X)
```

5.10.4. Zusammenspiel mehrerer TGDs untersuchen

Die Verwendung dieser Funktion, welche ebenfalls im `DependencyMinimizeUtil` in der Methode `getMinimizedDependencies2(PositiveConjunctiveTGD dependency, PositiveConjunctiveTGD[] conditionals)` anzutreffen ist, berechnet alle Möglichkeiten zur Kombination der Basis-TGD (`dependency`) und einer Menge von TGDs (`conditionals`), wobei alle möglichen Eingabemuster der Basis-TGD berücksichtigt werden. Hierbei werden nur Varianten berücksichtigt, welche bessere Ergebnisse als die Basis-TGD für dieses Eingabemuster liefern. Das Ergebnis ist eine `DependencyLattice`, welche die Spezialfälle enthält.

Das folgende Beispiel zeigt, wie diese Funktion, welche alle Implementierungen zusammenführt, benutzt werden kann. Die Ausgabe ist nicht vollständig, weil diese sehr lange ist (die `DependencyLattice` enthält 10 Schichten)

Codebeispiel: Zusammenspiel mehrerer TGDs untersuchen

```
//TGDs definieren
PositiveConjunctiveTGD TGD_D1 =
   (PositiveConjunctiveTGD)DependencyParser.parse(
   "S(X,Y) -> R(X,Y),R(Y,2),P(Y,Z)");
PositiveConjunctiveTGD TGD_D2 =
   (PositiveConjunctiveTGD)DependencyParser.parse(
   "Q(A) -> R(A,2)");

//Array mit zu kombinierenden TGDs füllen
PositiveConjunctiveTGD[] conditionals = {TGD_D1, TGD_D2};

//Kombinationen mit Spezialfälle berechnen
DependencyLattice lattice =
   DependencyMinimizeUtil.getMinimizedDependencies2(TGD_D1,
   conditionals);

//Ergebnis ausgeben
PrettyPrinter.print(lattice);

--- Ausgabe ---
Layer 1 ++++++++++++++++
        S(X,Y) -> P(Y,Z), R(Y,2), R(X,Y)
Layer 2 ++++++++++++++++
        S(ZV,X), S(X,2), S(X,ZV) ->         S(X,ZU), S(ZU,X), S(X,2) -
   >         S(ZV,X), S(X,2) ->
        S(ZU,X), S(X,2), S(X,ZV) ->          S(ZV,X), S(ZU,ZV), S(X,2)
   ->         S(ZU,X), S(X,2) ->
        S(ZU,ZV), S(ZV,X), S(X,2) ->         S(ZV,ZU), S(ZV,X), S(X,2)
   ->         S(ZU,X), S(ZV,ZU), S(X,2) ->
        S(ZU,X), S(X,ZU), S(X,2) ->         S(ZU,X), S(ZV,X), S(X,2) -
   >         S(ZV,X), S(ZV,ZU), S(X,2) ->
        S(ZU,X), S(ZU,ZV), S(X,2) ->                S(X,ZU), S(ZV,X),
   S(X,2) ->
Layer 3 ++++++++++++++++
...
```

Die hier berechneten Schichten bilden die Basis für einen abgewandelten Chase, der aus diesen in der Lage ist, den Kern direkt zu bilden.

6. Zusammenfassung und Ausblick

6.1. Hauptergebnisse

Die vorliegende Diplomarbeit untersucht die Frage, ob der Kern eines Datenaustauschszenarios auch ohne Kernberechnung gefunden werden kann. Letztere hat bei großen Datenmengen den Nachteil, dass der Aufwand zum Berechnen des Kerns stark ansteigt und daher praktisch der Kern nicht mehr in vertretbarer Zeit berechnet werden kann. Der präsentierte neue Ansatz von Pichler und Savenkov umgeht das Problem, indem der Kern direkt durch den Chase berechnet wird. Die vorliegende Implementierung ist in der Lage alle Möglichkeiten des Zusammenspiels von TGDs zu untersuchen und daraus neue, spezialisiertere TGDs zu generieren. Darauf aufbauend ist es möglich einen angepassten Chase zu entwickeln, der den Kern einer Lösung ohne Umweg über eine universelle Lösung und der damit verbundenen aufwändigen Kernberechnung erstellt [17].

6.2. Zusätzliche Erweiterungen und Verbesserungen

Die Implementierung ist in der vorliegenden Form nicht in der Lage, den Kern zu berechnen. Dazu ist ein angepasster Chase notwendig. Dieser kann die vom vorliegenden System bereitgestellten TGDs als Basis seines Vorgehens verwenden und immer die spezialisierteste aller passenden TGDs auswählen und den Chase Schritt darauf ausführen. Wenn der Chase vollständig durchlaufen ist, liefert dieser als Ergebnis den Kern des Datenaustauschszenarios.

Das System unterstützt weiters nur Quelle-zu-Ziel Abhängigkeiten, keine Zielabhängigkeiten (weder TGDs noch EGDs), allerdings gehen Pichler und Savenkov in [17] ebenfalls nur von STDs aus, weil TDs den ganzen Ablauf noch einmal erheblich erschweren würden. Um die Unterstützung von TDs nachzurüsten, müsste zuerst gezeigt werden, dass ein angepasster Chase den Kern direkt berechnen kann und dann

sowohl die theoretischen Grundlagen als auch aufbauend darauf die Implementierung für eine Unterstützung von TDs angepasst werden.

In [16] beschreiben Kunen und Suciu, wie eine Abfrage minimiert werden kann. Die Implementierung verwendet den einfachsten Ansatz zum Minimieren von TGDs, bei dem Tupel entfernt werden und überprüft wird, ob die TGD noch dieselbe Aussage hat. Der Artikel untersucht einen skalierbaren Abfrageminimierungsalgorithmus, welcher deutlich schneller als der derzeitige ist.

Eine weitere Verbesserung könnte erreicht werden, wenn die STDs vor deren Analyse Minimiert würden. Wie STDs (und EGDs) minimiert werden können wird von Gottlob, Pichler und Savenkov in [13] untersucht. Damit würde die Anzahl der STDs geringer werden, was sich unmittelbar auf die Laufzeit der Analyse auswirkt.

7. Referenzen

[1] Fagin, R., Kolaitis, P.G., Miller, R.J. & Popa, L.: “Data Exchange: Semantics and Query Answering”, ICDT, 2003, pp. 207-224

[2] Fagin, R., Kolaitis, P.G., Miller, R.J. & Popa, L.: “Data Exchange: Semantics and Query Answering”, Theor. Comput. Sci., Elsevier Science Publishers Ltd., 2005, Vol. 336(1), pp. 89-124

[3] Fagin, R., Kolaitis, P.G., Nash, A. & Popa, L.: “Towards a theory of schema-mapping optimization”, PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, 2008, pp. 33-42

[4] Fagin, R., Kolaitis, P.G. & Popa, L.: “Data exchange: getting to the core”, PODS, 2003, pp. 90-101

[5] Fagin, R., Kolaitis, P.G. & Popa, L.: “Data exchange: getting to the core”, ACM Trans. Database Syst., ACM, 2005, Vol. 30(1), pp. 174-210

[6] Fagin, R., Kolaitis, P.G., Popa, L. & Tan, W.C.: “Composing Schema Mappings: Second-Order Dependencies to the Rescue”, PODS, 2004, pp. 83-94

[7] Fagin, R., Kolaitis, P.G., Popa, L. & Tan, W.C.: “Composing schema mappings: Second-order dependencies to the rescue”, ACM Trans. Database Syst., ACM, 2005, Vol. 30(4), pp. 994-1055

[8] Gilland, M.: “Combination Generator”, http://www.merriampark.com/comb.htm, Zugriff am 5.5.2009

[9] Gilland, M.: “Permutation Generator“, http://www.merriampark.com/perm.htm, Zugriff am 5.5.2009

[10] Gottlob, G.: “Computing cores for data exchange: new algorithms and practical solutions”, PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-

SIGACT-SIGART symposium on Principles of database systems, ACM, 2005, pp. 148-159

[11] Gottlob, G. & Nash, A.: “Efficient core computation in data exchange”, J. ACM, ACM, 2008, Vol. 55(2), pp. 1-49

[12] Gottlob, G. & Nash, A.: “Data exchange: computing cores in polynomial time”, PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, 2006, pp. 40-49

[13] Gottlob, G., Pichler, R. & Savenkov, V.: “Normalization and Optimization of Schema Mappings”, VLDB '09: 35th International Conference on Very Large Data Bases, to appear 2009

[14] Hell, P. & Nešetřil, J.: „The core of a graph”, Discrete Math., Elsevier Science Publishers B. V., 1992, Vol. 109(1-3), pp. 117-126

[15] Kolaitis, P.G.: “Schema mappings, data exchange, and metadata management”, PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, 2005, pp. 61-75

[16] Kunen, I.K. & Suciu, D.: “A Scalable Algorithm for Query Minimization”, Technical Report, University of Washington, 2002

[17] Pichler, R. & Savenkov, V.: “Analysing Redundancy Generation in Data Exchange”, Technical Report, TU-Wien, 2009

[18] Pichler, R. & Savenkov, V.: “Towards Practical Feasibility of Core Computation in Data Exchange”, LPAR, 2008, pp. 62-78

[19] Rosen, K.H.: “Discrete mathematics and its applications”, McGraw-Hill, 1991, pp. 282 – 286

[20] Savenkov, V.: “Implementing Core Computation for Data Exchange”, Master’s thesis, TU-Wien, 2007