Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorarbeit Informatik

Provenance of SQL Transactions

Martin Fuß

29.09.2018

Gutachter

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Betreuer

Benjamin Dietrich
Universität Tübingen

Fuß, Martin:
Provenance of SQL Transactions
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 29.05.2018-29.09.2018

Zusammenfassung

Aufbauend auf einer vorherigen Arbeit die sich mit der *Language-Level Provenance* Analyse von SQL-`SELECT` Queries befasst hat, beschäftigt sich diese Arbeit damit diese Umsetzung auf die SQL Anweisungen `DELETE`, `INSERT` und `UPDATE` zu erweitern. Zusätzlich werden Teile der prozeduralen Sprache *PL/pgSQL* implementiert, um damit auch entsprechende *benutzerdefinierte Funktionen* übersetzen zu können.

Das Ziel ist es, ein *Haskell*-Programm zu entwickeln das aus einer gegebenen SQL Anweisung zwei Anweisungen generiert. Die erste dieser Abfragen liefert primär das originale Ergebnis und schreibt dazu Logs der verarbeiteten Daten. Die zweite Abfrage liest dann diese Logs und leitet damit die *Data Provenance* ab.

Inhaltsverzeichnis

Abkürzungsverzeichnis

UDF Userdefined Function

DP Data Provenancehttps://www.facebook.com/

DPA Data Provenance Analyse

AST Abstract Syntax Tree

ADT Algebraic Data Type

GADT Generalized Algebraic Data Type

SQL Structured Query Language

DDL Data Definition Language

DML Data Manipulating Language

1 Einleitung

Egal ob wir mit Menschen darüber reden was sie erlebt haben und wer oder was sie geprägt hat, uns die Frage stellen welche Züge ein Schachspieler gemacht hat um seinen Gegner zu besiegen oder uns darüber Gedanken machen warum eine bestimmte Abfrage an eine Datenbank gerade dieses Ergebnis liefert - all diese Informationen geben uns Aufschluss über die Geschichte von Daten, Objekten und Personen. Sie geben uns Hinweise wie verlässlich das was wir gerade betrachten unter Umständen ist (wenn wir die richtigen Schlüsse ziehen) oder geben uns die Möglichkeit nachzuvollziehen wo und warum unter Umständen etwas falsch lief. Warum eine Abfrage ein anderes Ergebnis liefert als wir erwarten, der Schachspieler vielleicht schneller jubeln hätte können wenn er einen bestimmten, vielleicht sinnlosen, Zug nicht gemacht hätte oder warum der Familienvater in der Nachbarschaft auf die schiefe Bahn geraten ist.

Natürlich interessiert uns in dieser Arbeit hier der Datenaspekt und nicht so sehr das Menschliche. Gut, selbstverständlich ist auch der Mensch wenn es um Daten geht ein nicht zu verachtender Faktor. Daten können falsch eingepflegt worden sein oder die Abfrage die zum falschen Ergebnis führt fehlerhaft formuliert worden sein. In dieser Arbeit wird es aber darum gehen welche Daten in ein bestimmtes Ergebnis eingeflossen sind bzw. primär darum uns zu überlegen wie sich Daten verändern wenn sie manipuliert werden.

1.1 Beispiel

In Tabelle 1.1 sehen wir eine Liste mit den Bundesliga-Ergebnissen aus den Spielen der baden-württembergischen Bundesligateams gegeneinander . Die Query A in Listing 1.1 fragt nun ab gegen wen Freiburg an den ersten drei Spieltagen ein Heimspiel gewonnen

spiel	heim	gast	sieger
1	Stuttgart	Freiburg	1
1	Augsburg	Hoffenheim	2
2	Hoffenheim	Stuttgart	1
2	Freiburg	Augsburg	1
3	Stuttgart	Augsburg	0
3	Freiburg	Hoffenheim	1
4	Freiburg	Stuttgart	2
⋮			

Tabelle 1.1: Bundesliga Ergebnisse

team	punkte	spiele
Hoffenheim	0	0
Augsburg	0	0
Freiburg	0	0
Stuttgart	0	0

Tabelle 1.2: Bundesliga Tabelle

hat.

```
1 SELECT Spiel AS Spieltag,
2        Gast  AS Gegner
3 FROM   spiele
4 WHERE  Heim='Freiburg'
5    AND Sieger=1
6    AND Spieltag <= 3 ;
```

Listing 1.1: Query A

spieltag	gegner
2	Augsburg
3	Hoffenheim

Tabelle 1.3: Ergebnis von Query A

So weit ist einigermaßen noch offensichtlich wie sich die Daten in der Ergebnistabelle zusammensetzen. Schauen wir auf das Gegner -Feld in der zweiten Zeile, dann können wir uns denken dass das blau markierte Feld in Tabelle 1.1 ausgewählt wurde und dass die grün markierten Felder für die Auswahl herangezogen wurde warum der Eintrag ausgewählt wurde. Zusätzlich verrät uns ein Blick auf die Abfrage unter Umständen noch dass die rot markierten Zeilen dafür gesorgt haben dass dieser Wert schließlich den Weg ins Output gefunden hat.

Betrachten wir aber jetzt die Anweisung in Listing 1.2 und fragen danach die Tabelle 2.5 ab, dann ist z.B. plötzlich schwierig nachzuvollziehen welcher der Freiburger Heimsiege dazu geführt hat dass in der Tabelle dieses Update stattgefunden hat.

```
1 UPDATE liga l
2 SET    punkte=punkte+3,
3        gespielt=gespielt+1
4 FROM   bawue b
5 WHERE  l.team=b.heim
6   AND  b.sieger=1;
```

Listing 1.2: Anweisung B

team	punkte	gespielt
Augsburg	0	0
Hoffenheim	3	1
Freiburg	3	1
Stuttgart	3	1

Tabelle 1.4: BuLi Tabelle nach dem Update

1.2 Data Provenance

Die *Provenance* Analyse von Daten gibt uns hier ein Werkzeug an die Hand mit der wir nachverfolgen können was passiert ist wenn wir z.B. eine bestimmte Zelle (oder andere Einheit) einer Tabelle in einem relationalen Datenbanksystem betrachten. Der Lehrstuhl für Datenbanksysteme an der Universität Tübingen hat dafür eine Methode entwickelt die eine Data Provenance Analyse für eine Vielzahl von SQL Konstrukten ermöglicht. Dabei werden SQL Queries so umgeschrieben dass sie mit Hilfe von Datenabhängigkeiten Aufschluss geben können **woher** Elemente einer Datenausgabe stammen und **warum** diese Elemente ausgewählt wurden ausgegeben zu werden. [Müller u. a. 2018]

1.3 Ziel

Diese Arbeit baut auf einer vorhergehenden Bachelorarbeit von Gabriel Paradzik [1] die sich mit der automatisierten Transformation von gängigen SQL Abfragen und *Provenance Sets* befasst hat. [Paradzik 2018] Weiterführend wurde jetzt zuerst das Regelwerk für die Transformation von Befehlen der Data Manipulating Language (DML) implementiert. Außerdem beschäftigt sich die Arbeit mit der Übersetzung von *PL/pgSQL* Befehlen und benutzerdefinierten Funktionen (Userdefined Functions (UDFs)) die diese Befehle benutzen.

[1] Im Git Repository *PgLlProvenance* zu finden.

1.4 Aufbau

Nach dieser Einführung betrachten wir zuerst die Grundlagen und benutzten Werkzeuge. Besonders gehen wir hier noch einmal auf Begriff und Implementierung von *Data Provenance* ein. Das dritte Kapitel macht sich Gedanken zur Herleitung der Ableitungen für die Data Provenance Analyse (DPA) der implementierten Befehle. Das Kapitel 4 befasst sich dann mit der eigentlichen Übersetzung. Es beschreibt die Änderungen und Ergänzungen die am bestehenden Code von Gabriel Paradzik vorgenommen werden mussten und definiert die Regeln mit denen die SQL Statements übersetzt werden. Abschließend soll in Kapitel 5 ein abschließendes Fazit gezogen werden und über mögliche weitere Schritte nachgedacht werden.

2 Grundlagen

Dieses Kapitel bietet zuerst einen Überblick über die Werkzeuge und bestehenden Programme die bei der Erstellung von *SPTransactions* mit eingeflossen sind. Danach werden mögliche Arten von Data Provenance (DP) und Herangehensweisen diese zu ermitteln betrachtet und aufgezeigt wie die abgeleiteten DML und *PL/pgSQL* Anweisungen schlussendlich aussehen bzw. warum diese so abgeleitet wurden.

2.1 Werkzeuge

2.1.1 Haskell

Haskell ist die Programmiersprache die genutzt wurde um *SPTransactions* zu entwickeln. Dabei handelt es sich um eine funktionale Programmiersprache die es sich unter anderem auf die Fahnen geschrieben hat

- für jeden frei verfügbar zu sein.
- dass sie durch die Publikation von formalem Syntax und Semantiken komplett beschrieben wird.
- und sich für Lehre, Forschung und Anwendungen, auch für große Systeme, eignen soll.

[Marlow 2010]

Haskell ist dabei eine Programmiersprache die der Lehrstuhl für Datenbanksysteme in Forschung und Lehre immer wieder einsetzt, wodurch auch für die DPA eine Basis an Code

vorhanden ist. Unter anderem wurde der verwendete *LogParser* in Haskell geschrieben und liefert uns die zu manipulierenden Daten entsprechend in Haskell Code.

2.1.1.1 Eigenschaften

Eigenschaften von Haskell die die Sprache für uns interessant machen:

- **Purity** - Wie mathematische Funktionen sind
- **Laziness** - Haskell wertet Ausdrücke dann aus wenn sie benötigt werden.
- Haskells **Typsystem** - Haskell nutzt *statische Typisierung* so dass schon beim Kompilieren klar ist wenn die verwendeten Typen für Probleme sorgen bzw. dann wird der Compiler auch die Kompilierung verweigern.
- **(Generalisierte) algebraische Datentypen** ((G)ADTs) - Damit lassen sich mehrere Datenkonstruktoren zu einem zusammenfassen. Ein Beispiel dafür ist der Abstract Syntax Tree (AST) den uns der *LogParser* liefert.
- **Erweiterbarkeit** - Haskell und auch der Compiler *ghc* lassen sich einfach durch eine reiche Bibliothek an Modulen und Erweiterungen erweitern.

2.1.1.2 Erweiterungen

Drei dieser Erweiterungen seien dabei hier explizit erwähnt:

- **Lens** - Mit der Verwendung von Generalized Algebraic Data Type (GADT) und dem AST wird die Manipulation der Daten stellenweise recht mühsam, speziell wenn wir mehrere Ebenen tief in einem Objekt eine Veränderung vornehmen können. Das *Lens* Paket hilft dabei mit *Gettern* und *Settern* die ein mühsames Extrahieren und wieder einfügen vereinfachen.
- **Pattern Synonyms** - So angenehm das *Pattern Matching* in Haskell Funktionen ist, aber je nach Datentyp kann damit die Lesbarkeit einer Funktion deutlich eingeschränkt werden. *Pattern Synonyms* bieten uns die Möglichkeit das Pattern Matching

auszulagern und im Funktionsaufruf nur noch die relevanten Informationen zu verwenden. Listing 2.1 stellt ein mögliches Beispiel vor bei dem die Funktionsdefinition deutlich aufgeräumt wird.

In diesem Beispiel wird ein unidirektionales Pattern Synonym verwendet, vereinzelt werden auch bidirektionale Pattern Synonyms genutzt welche sich dann wie Konstruktoren verwenden lassen. Problematik bei diesen ist vor allem dass in den Pattern Synonyms nicht eine Variable der linken Seite der Definition mehrmals auf der rechten Seite verwendet werden darf was die Nützlichkeit leider etwas herabsetzt.

- **RecordWildcards** - Das obige Beispiel für ein *Pattern Synonym* lässt sich tatsächlich noch angenehmer implementieren indem sogenannte *RecordWildcards* verwendet werden. Listing 2.2 zeigt wie sich das *Pattern Matching* aus Listing 2.1 noch etwas vereinfachen lässt.

```
1 pattern PSelect
2       <- A.QBlockGeneric _ _ _
3                          _ _ _
4                          _ _ _
5                          _ _ _
6                          _ _ _
7                          _ _ _
8                          CMD_SELECT
9 translateQuery inp@PSelect    = (~~>) inp
```

Listing 2.1: Beispiel für die Verwendung von Pattern Synonyms

```
1 translateQuery inp@QBlockGeneric{cmdType=CMD_SELECT}
2       = {~~>} inp
```

Listing 2.2: Beispiel für die Verwendung von RecordWildcards

2.1.2 PostgreSQL

PostgreSQL (oder kurz *Postgres*) ist ein quelloffenes objektrelationales Datenbankmanagementsystem (ORDBMS) das seit 1986 aktiv entwickelt wird. *Postgres* unterstützt in der aktuellen Version 10 die meisten der Hauptbestandteile des SQL:2011 Standards und wurde dafür entwickelt sich durch Erweiterungen einfach ergänzen zu lassen. [PostgreSQL 2018]

2.1.2.1 PL/pgSQL

PL/pgSQL ist eine prozedurale Sprache für *Postgres* die einfach zu nutzen ist und Zugriff auf den Funktionsumfang von Postgres hat. Wir können damit außerdem einfach Funktionen erstellen und auch anders als mit reinen SQL Befehlen, als großen Vorteil, mehrere Kommandos auf einmal an Postgres weitergeben ohne alle diese Befehle einzeln abschicken zu müssen. Ausserdem haben wir die Möglichkeit Zwischenergebnisse können in Variablen gespeichert und später weiterverarbeitet werden. PL/pgSQL Funktionen können dabei jegliche elementaren Datentypen, sowie Arrays daraus aber auch Zeilentypen die sich aus mehreren einzelnen Typen zusammensetzen als Argumente erhalten und auch als Rückgabewerte zurückgeben. Auch Funktionen die keine Rückgabe haben sind möglich, diese haben dann den Rückgabewert `void`.

Zusätzlich interessant wird *PL/pgSQL* dadurch dass SQL damit mit zusätzlichem Kontrollfluß ergänzt wird. So bekommen wir zum Beispiel Schleifen und Konstrukte wie `IF` dazu. In Listing 2.3 ist ein Beispiel für eine *PL/pgSQL* Funktion aufgeführt.

2.1.2.2 Normalisierung

Während bei den DML Anweisungen keine Normalisierung der Anweisungen nötig ist, macht es uns hier das Leben leichter wenn wir bei UDF auf eine Normalisierungsschritt bestehen. In Listing 2.3 sehen wir einmal eine UDF bei der wir die Funktionsparameter „normal“ im Code mit verwenden. In der normalisierten Variante in Listing 2.4 haben wir dann diese Parameter bei der Verwendung in einer SQL Anweisung gebunden. Damit können wir später wesentlich einfacher mit der Transformation der Funktion arbeiten.

```
1  CREATE FUNCTION heimspiele(team text, gewinner int)
2                      RETURNS int4 AS $$
3  DECLARE
4    n int4 := 0;
5    t text;
6  BEGIN
7    FOR t IN (SELECT heim
8              FROM spiele
9              WHERE heim=team  AND sieger)
10   LOOP
11     n:=n+1;
12    END LOOP;
13    RETURN n;
14 END; $$ LANGUAGE PLPGSQL;
```

Listing 2.3: PL/pgSQL Beispielfunktion

```
1  CREATE FUNCTION heimspiele(team text, gewinner int)
2                      RETURNS int4 AS $$
3  DECLARE
4    n int4 := 0;
5    t text;
6  BEGIN
7    FOR t IN (SELECT heim
8              FROM spiele,
9                   (select team, gewinner) AS udf(u1, u2)
10             WHERE heim=udf.u1 AND sieger=gewinner)
11   LOOP
12     n:=n+1;
13    END LOOP;
14    RETURN n;
15 END; $$ LANGUAGE PLPGSQL;
```

Listing 2.4: Normalisierte Funktion aus 2.3

Zusätzlich war zu beachten, dass der LogParser die Funktionsparametern nicht mit Namen an uns weitergibt, nur die Typen bleiben uns erhalten. Dafür können wir mit den alternativen Bezeichnern `$n` arbeiten, wobei $n \geq 1$ einfach die Position in der Parameter Liste bezeichnet. Statt `gewinner` wird also `$2` verwendet im Beispiel aus den beiden Listings 2.3, 2.4.

2.1.3 LogParser

Diese Werkzeug[1] ist essentiell für diese Arbeit da es die Haskell-Repräsentation für unsere SQL Anweisungen liefert. Der LogParser, entwickelt am Lehrstuhl für Datenbanksysteme, liest mittels einer *Postgres* Erweiterung die *Postgres* Logs für ein gegebenes Statement. Da wir mit dem dadurch enthaltenen JSON Baum noch nicht viel anfangen können, werden diese Daten so aufbereitet dass wir einen AST erhalten den wir in *Haskell* dann weiterverarbeiten können. [Hirn 2017]

2.1.4 ASTPrettyprinter

Der *Prettyprinter* wandelt unsere Daten, in Form von ASTs schließlich wieder zurück in lesbare SQL Anweisungen. Dabei wird der Code sinnvoll eingerückt und, sofern nicht anders gewünscht, auch farbige Hervorhebungen von einzelnen Teilen verwendet.

2.1.5 SQLProv

Dabei handelt es sich um die Vorarbeit bzw. Grundlage dieser Arbeit. Gabriel Paradzik hat mit diesem Programm die Transformation von SQL Queries für die DPA implementiert, die auch von *SPTransactions* genutzt werden. *SQLProv* kann dabei mit folgenden Konstrukten umgehen:

- Aggregatsfunktionen
- Table Functions

[1] Zu finden im git des Lehrstuhl für Datenbanksysteme im Repository *PgQueryHauler*

- CTEs
- `SELECT ...`
- `SELECT ... FROM t`
- `SELECT ... FROM t` (mit Window Functions)
- `SELECT ... FROM t_1,... ,t_n WHERE p`
- `SELECT ... FROM t ORDER BY ... LIMIT ...`
- `SELECT ... FROM t GROUP BY ...`
- `SELECT DISTINCT ... FROM t ORDER BY ...`

Die Normalisierung einer SQL Query stellt sicher dass mögliche Kombinationen der entsprechenden Klauseln von diesen Konstrukten abgedeckt wird ohne eine Vielzahl an Spezialfällen bei der Implementierung abdecken zu müssen. [Paradzik 2018]

2.2 SQL

Die Structured Query Language (SQL) wird bereits seit den 70er Jahren entwickelt und zeichnet sich durch eine einfache Syntax aus. Sie gilt heute als *die* Datenbanksprache für relationale Datenbanksyteme und wird von den meisten gängigen Systemen unterstützt, wenn auch üblicherweise nicht mit dem vollen Sprachumfang. [Saake u. a. 2010] Dabei besteht SQL selbst aus mehreren Sprachteilen von denen für uns vor allem die *Data Definition Language (DDL)* und die *DML* interessant sind.

2.2.1 Datendefinitionssprache

Die DDL dient uns im einfachsten Fall erst einmal dazu die Datenbank mit Leben zu füllen und *Relationen* (was wir im üblichen Sprachgebrauch unter *Tabellen* kennen) zu erstellen. Dabei haben wir zwei Varianten zur Auswahl von denen die erste die gebräuchlichere sein dürfte. Dabei wird eine Tabelle erstellt der wir einen Namen geben sowie die Namen

ihrer Spalten und deren Datentypen. Zusätzlich können wir hier schon *Constraints* für die einzelnen Spalten vergeben.

```
1 CREATE TABLE tabelle (feld1 typ1 [, feld2 typ,...] );
```

Listing 2.5: Tabelle erzeugen durch Angabe von Spalten und Typen

Die zweite Variante, die im Falle von *SPTransactions* verwendet wird, erstellt eine Tabelle aus dem Ergebnis einer Query. Dabei können wir die Spaltennamen im `CREATE` Statement selbst benennen, den Spaltentyp ermittelt SQL automatisch. Wir können aber auch die Namen wie sie in der Query vorkommen verwenden, dann fällt die Angabe der Namen im `CREATE` Statement einfach weg.

```
1 CREATE TABLE anderetabelle (feld1,feld2) AS
2       SELECT feld2 AS feld2, feld3 as feld3 FROM tabelle;
```

Listing 2.6: Tabelle wird aus einer Query generiert

Auch das Löschen von Tabellen fällt in in den Bereich von Datendefinition, ebenso können wir mit DDL Anweisungen *Constraints* (Einschränkungen), wie z.B. Schlüssel, für einzelne oder mehrere Spalten einer Tabelle festlegen oder Indexe definieren.

2.2.2 Datenmanipulation

Die Sprache von DML Statements dienen, offensichtlich, dazu Daten zu manipulieren. Dazu sind im folgenden die Möglichkeiten aufgeführt die von *SPTransactions* transformiert werden können.

2.2.2.1 Daten einfügen

```
1   INSERT INTO tabelle [AS alias] [(feld1, (feld2,...)]
2       query
3     [RETURNING * | ausdruck [AS name] [, ... ] ]
```

```
1   INSERT INTO andereta
```

2.2.2.2 Daten löschen

```
1  DELETE FROM tabelle
2    [USING using_list]
3    [WHERE condition]
4    [RETURNING * | ausdruck [AS name] [, ... ] ]
```

2.2.2.3 Daten ändern

```
1  UPDATE tabelle [AS alias]
2    SET feld = ausdruck [, ...]
3    [FROM from_list]
4    [USING using_list]
5    [WHERE condition]
6    [RETURNING * | ausdruck [AS name] [, ... ] ]
```

2.2.3 Datenabfrage

Damit dem Nutzer klar wird was überhaupt in den Tabellen drinsteht, haben wir noch die Möglichkeit Abfragen (*Queries*) auf unseren Tabellen auszuführen. Dabei erwartet das Programm die in 2.1.5 angedeutete Normalisierung. In den Listings 2.7 , 2.8 wird diese kurz angerissen [Paradzik 2018].

```
1 SELECT heim
2 FROM bawue
3 WHERE sieger=1
4 GROUP BY heim
5 ORDER BY heim;
```

Listing 2.7: Query

```
1 SELECT heim
2 FROM
3   (SELECT heim
4     FROM (SELECT heim
5           FROM bawue
6           WHERE sieger=1) t
7     GROUP BY heim) u
8 ORDER BY heim;
```

Listing 2.8: normalisierte Query

Es gilt also die Queries so zu verschachteln dass sie jeweils nur noch eine Klausel haben.

2.3 Data Provenance

Wenn man von Data Provenance spricht, geht es darum die Geschichte von Daten festzuhalten bzw nachvollziehen zu können. In einer Zeit in dem es Unmengen an, zum Teil riesigen Sammlungen von Daten gibt, wird es wichtig aber auch schwierig nachvollziehen zu können was diese „erlebt“ haben. Wir wollen also wissen woher diese Daten stammen, von wem sie gesammelt, eingepflegt und verwaltet werden und welche Veränderungen an diesen Daten vorgenommen wurden. Wenn diese Informationen verfügbar sind und wir lernen wie vertrauenswürdig erhaltene Daten sind, können diese im Wert steigen und sinken. [Cheney u. a. 2009]

Fragen die im Zusammenhang mit Data Provenance dabei auftauchen sind üblicherweise die Fragen danach *wie* ein Ergebnis zustande kommt und *woher* die Daten stammen die in das Ergebnis mit eingeflossen sind.

Dabei haben wir ein weites Spektrum dessen wie detailgetreu wir an die Sache herangehen wollen (und wenn wir nur Konsument sind, auch können). Mal reicht es uns zu wissen aus welchem Datensatz ein Datum stammt, mal wollen wir genau herauskriegen welche Zellen genau zur Berechnung herangezogen wurden. Diese Detailgenauigkeit kennen wir unter *Granularität*.

In unserem Falle beschäftigen wir uns konkret mit den Daten in relationalen Datenbanken und der Möglichkeit mit SQL Anweisungen *Provenance Informationen* auszulesen und zu verändern. Uns geht es um die feinste *Granularität* und wir betrachten damit die Informationen auf Zelllevel. Dazu gibt es eine ganze Reihe von Ansätzen.

2.4 Ansätze

Ein Ansatz den die Arbeitsgruppe am Lehrstuhl für Datenbanksysteme verfolgt und der die Basis für diese Arbeit ist, ist der, einen Interpreter direkt aus der originalen Query abzuleiten. Während diese Query noch auf konkreten Daten arbeitet, wird in diesem Prozess die Verbindung zwischen Ein- und Ausgabedaten interessant und die Werte der Daten werden nebensächlich. Dieser Prozess findet schließlich durch eine Aufteilung in zwei Phasen statt. [Müller u. a. 2018]

Um die Daten in Relation setzen zu können brauchen wir einige Vorbereitungsschritte. Die im folgenden vorgenommen Aktionen werden hier anhand der Tabellen 1.1 und 1.2 aus dem Eingangsbeispiel veranschaulicht.

2.4.1 Vorbereitung

Zuerst benötigen die originalen Tabellen eine Anpassung. Jede der benutzten Tabellen erhält für jede Zeile eine Tupel-ID ρ die später hilft die Werte zu loggen. In unserem Falle ergibt das dann die Tabellen 2.1, 2.2.

tuid	spiel	heim	gast	sieger
ρ_1	1	Stuttgart	Freiburg	1
ρ_2	1	Augsburg	Hoffenheim	2
ρ_3	2	Hoffenheim	Stuttgart	1
ρ_4	2	Freiburg	Augsburg	1
		⋮		

Tabelle 2.1: Ergebnisse[1]

tuid	team	punkte	spiele
ρ_{61}	Hoffenheim	0	0
ρ_{62}	Augsburg	0	0
ρ_{63}	Freiburg	0	0
ρ_{64}	Stuttgart	0	0

Tabelle 2.2: Tabelle[1]

In der zweiten Phase wollen wir die Form dieser Tabellen erhalten, ersetzen aber alle Werte mit Mengen die jeweils eine ID (ρ benannt) enthalten. Dies sehen wir in den Tabellen 2.3, 2.4. Wir sehen also, für jede Zeile einer Phase $\mathbb{1}$ Tabelle gibt es eine entsprechende Zeile

tuid	spiel	heim	gast	sieger
ρ_1	$\{\rho_{13}\}$	$\{\rho_{14}\}$	$\{\rho_{15}\}$	$\{\rho_{16}\}$
ρ_2	$\{\rho_{17}\}$	$\{\rho_{18}\}$	$\{\rho_{19}\}$	$\{\rho_{20}\}$
ρ_3	$\{\rho_{21}\}$	$\{\rho_{22}\}$	$\{\rho_{23}\}$	$\{\rho_{24}\}$
ρ_4	$\{\rho_{25}\}$	$\{\rho_{26}\}$	$\{\rho_{27}\}$	$\{\rho_{28}\}$
		$\vdots$		

Tabelle 2.3: Ergebnisse$^{\mathbb{2}}$

tuid	team	punkte	spiele
ρ_{61}	$\{\rho_{65}\}$	$\{\rho_{66}\}$	$\{\rho_{67}\}$
ρ_{62}	$\{\rho_{68}\}$	$\{\rho_{69}\}$	$\{\rho_{70}\}$
ρ_{63}	$\{\rho_{71}\}$	$\{\rho_{72}\}$	$\{\rho_{73}\}$
ρ_{64}	$\{\rho_{74}\}$	$\{\rho_{75}\}$	$\{\rho_{76}\}$

Tabelle 2.4: Tabelle$^{\mathbb{2}}$

in der Phase $\mathbb{2}$ Tabelle die eine entsprechenden ID hat. Mit einer Reihe von Log-Tabellen und Funktionen die uns diese Tabellen auslesen und mit Hilfe dieser die erhaltenen Werte schließlich die Verknüpfungen herstellen können wir so unsere DPAs herleiten.

2.4.2 Phasen

Nach diesen Vorbereitungsschritten haben wir im Kern zusammen was wir brauchen. Für die Verarbeitung der beiden Phasen benutzen wir UDFs die auf Log-Tabellen zugreifen. Dies sind in unserem Falle größtenteils Funktionen die einen Verknüpfung auf 2 oder mehreren ρ von verwendeten Tabellen ausführen und eine daraus neu erstellte ρ zurückgibt. In der *Instrumentationsphase* (Phase $\mathbb{1}$) handelt es sich dabei um `write` Funktionen, in der *Interpretationsphase* (Phase $\mathbb{2}$) um `read` Funktionen.

Beispiele für solche Funktionen finden sich in Listings 2.9 (`writeJoin`) und 2.10 (`readJointable`). Hierbei machen wir uns zu Nutzen dass wir in *Postgres* mehrere Funktionen eines Namens erstellen können wenn diese sich in ihren Argumenten unterscheiden können. So brauchen wir nicht für jede verschiedene Argumentzahl aufpassen die richtige Funktion aufzurufen.

```
1  CREATE FUNCTION log.writeJoin(v_loc integer,
2                                v_tuid_1 integer,
3                                v_tuid_2 integer)
4  RETURNS integer AS $$
5  DECLARE
6    v_tuid int;
7  BEGIN
8    INSERT INTO log.logJoin2 (location, tuid_1, tuid_2)
9        VALUES (v_loc, v_tuid_1, v_tuid_2)
10       RETURNING tuid INTO v_tuid;
11   RETURN v_tuid;
12   EXCEPTION
13      WHEN UNIQUE_VIOLATION THEN
14       RETURN log.readJoin(v_loc, v_tuid_1, v_tuid_2);
15 END; $$ LANGUAGE PLPGSQL;
```

Listing 2.9: writeJoin Funktion

```
1  CREATE FUNCTION log.readJoinTable(v_loc integer,
2                                    v_tuid_2 integer)
3  RETURNS table (tuid integer, tuid_1 integer) AS $$
4  BEGIN
5    SELECT j.tuid, j.tuid_1
6    FROM   log.logJoin2 j
7    WHERE  j.location=v_loc
8       AND j.tuid_2=v_tuid_2;
9  END; $$ LANGUAGE PLPGSQL
```

Listing 2.10: readJoinTable Funktion

2.4.3 Phase $\mathbb{1}$

In **Phase** $\mathbb{1}$ modifizieren wir unsere Anweisung q dahingehend dass eine Anweisung $q^{\mathbb{1}}$ entsteht. Diese trifft die selben wertebasierten Entscheidungen wie die Originalanweisung. Wird eine solche Entscheidung getroffen, wird dabei zusätzlich mitgeschrieben (geloggt) welche Daten in diese Entscheidung eingeflossen sind. [Müller u. a. 2018]

Am Beispiel des `UPDATE`s aus dem Einleitungsbeispiel sehen wir in Listing 2.11 die Transformation in die Phase $\mathbb{1}$. InTabelle 2.6 lässt sich sehen wir die geänderten ρ Zellen der bearbeiteten Tupel und in Tabelle 2.5 die geloggten Informationen des Update Prozesses. Die *Location* gibt dabei jeweils die Position in der transformierten Anweisung.

```
1 UPDATE liga_1 l
2 SET    punkte=punkte+3,
3        gespielt=gespielt+1,
4        tuid=log.writeJoin(1, l.tuid, b.tuid)
5 FROM   bawue_1 b
6 WHERE  b.spiel<=2
7   AND  l.team=b.heim
8   AND  b.sieger=1;
```

Listing 2.11: Phase$^{\mathbb{1}}$

location	tuid	tuid1	tuid2
1	$\{\rho_{584}\}$	$\{\rho_{61}\}$	$\{\rho_3\}$
1	$\{\rho_{585}\}$	$\{\rho_{63}\}$	$\{\rho_4\}$
1	$\{\rho_{586}\}$	$\{\rho_{64}\}$	$\{\rho_1\}$

Tabelle 2.5: Tabelle logJoin2

tuid	team	punkte	spiele
ρ_{62}	Augsburg	0	0
ρ_{596}	Hoffenheim	3	1
ρ_{597}	Freiburg	3	1
ρ_{598}	Stuttgart	3	1

Tabelle 2.6: Tabelle$^{\mathbb{1}}$ nach update

Mit diesen Informationen im Gepäck können wir damit den entscheidenden Schritt hin zur DP machen.

2.4.4 Phase 2

In **Phase** 2 erhalten wir eine Anweisung q^2 die als Interpreter für die geloggten Daten dient und die die Operationen auf sogenannten *Dependency Sets* („Abhängigkeitsmengen“) ausführt. Die Entscheidungen werden also nicht mehr auf den originalen Daten gefällt sondern arbeiten jeweils mit den ρ IDs die die Daten nachvollziehbar machen. [Müller u. a. 2018]

Wenn wir das Beispiel aus Phase 1 weiterführen, erhalten wir damit die `UPDATE` Anweisung im Listing 2.12. Der Leseprozess findet in der `FROM` Klausel statt, im Grunde aber mit den gleichen Informationen (Location, ρs. Einzig der Zugriff auf die Felder der geänderten Tabelle bleibt uns verborgen, deshalb muss der finale Abgleich der Daten in der `WHERE` Klausel erfolgen.

```
1 UPDATE liga_2 l
2 SET    punkte=punkte || array[],
3        gespielt=gespielt || array[],
4        tuid=j.tuid
5 FROM   bawue_2 b,
6        LATERAL log.readJoinTable(1, b.tuid)
7                AS j(tuid, tuid1)
8 WHERE  j.tuid1=l.tuid;
```

Listing 2.12: Phase^2

Dieses Statement liefert uns jetzt unsere *Provenance Informationen*. Dabei gehen wir weg von Operationen wie der Addition oder Multiplikation und sammeln einfach alle Informationen die in Phase 1 hier genutzt wurden und nutzen die offensichtlichste Operation die wir auf den Mengen die wir jetzt nutzen haben: die Vereinigung. Für Daten die wir aus Tabellen auslesen, nehmen wir hier dann ihre vorbereiteten (oder später modifizierten) *Provenance Sets* für diese Vereinigung heran, für konstante Werte wie wir sie in unserem Beispiel benutzen ersetzen wir diese durch leere Mengen (in der aktuellen Variante repräsentiert durch leere Array).

tuid	team	punkte	spiele
ρ_{62}	$\{\rho_{68}\}$	$\{\rho_{69}\}$	$\{\rho_{70}\}$
ρ_{616}	$\{\rho_{74}\}$	$\{\rho_{75}\}$	$\{\rho_{76}\}$
ρ_{614}	$\{\rho_{65}\}$	$\{\rho_{66}\}$	$\{\rho_{67}\}$
ρ_{615}	$\{\rho_{71}\}$	$\{\rho_{72}\}$	$\{\rho_{73}\}$

Tabelle 2.7: Where-Provenance

2.4.5 Arten von Provenance

Dabei unterscheiden wir hier in zwei Arten wobei sie sich in unserem Falle wie wir sehen werden nicht völlig ausschließen werden weil die *Where-Provenance* eine Teilmenge der *Why-Provenance* darstellen wird.

2.4.5.1 Where-Provenance

Die **Where-Provenance** stellt die Frage, bzw. beantwortet sie für uns, woher diese Daten kommen. Es wird also nachvollziehbar welche Tabellenzellen mit in die Berechnung eingeflossen sind die zum entscheidenden Ergebnis geführt hat. Wenn wir uns in unserem Falle die geänderte Tabelle (dargestellt in Tabelle 2.7)wieder anschauen stellen wir fest dass sich leider nicht so furchtbar viel geändert hat. Immerhin hat sich die Änderung von ρ in der Phase 2 niedergeschlagen.

2.4.5.2 Why-Provenance

Etwas interessanter in unserem Beispiel wird warum die Daten ihren Weg in die Tabelle gefunden haben. Diese Frage beantwortet die sogenannte **Why-Provenance** zu sehen in Tabelle 2.8. Dabei fließen hier die Tabellenzellen mit in die *Why-Provenance* ein die dazu geführt haben dass in unserem Falle der Update Prozess stattgefunden hat.

In unserem Fall werden wir allerdings die *Why-Provenance* nicht losgelöst von der *Where-Provenance* betrachten, wir bekommen eine Ergebnismenge und können entscheiden ob dabei die Why Informationen verfügbar sind oder nicht, *Where-Provenance* wird in diesen

tuid	team	punkte	spiele
ρ_{62}	$\{\}$	$\{\}$	$\{\}$
ρ_{625}	$\{\}$	$\{\rho_{14}, \rho_{74}, \rho_{16}, \rho_{13}\}$	$\{\rho_{14}, \rho_{74}, \rho_{16}, \rho_{13}\}$
ρ_{623}	$\{\}$	$\{\rho_{22}, \rho_{21}, \rho_{24}, \rho_{65}\}$	$\{\rho_{22}, \rho_{21}, \rho_{24}, \rho_{65}\}$
ρ_{624}	$\{\}$	$\{\rho_{28}, \rho_{26}, \rho_{25}, \rho_{71}\}$	$\{\rho_{28}, \rho_{26}, \rho_{25}, \rho_{71}\}$

Tabelle 2.8: Why-Provenance

Informationen **immer** enthalten sein.

3 Herleitung benötigter Ableitungen

In diesem Kapitel sollen Gedankengänge die zu den in 4.3 dargestellten Ableitungen geführt haben aufgeführt werden. Zuerst beschäftigen wir uns mit der Problematik bei SQL Statements, werfen dann einen Blick auf die Verarbeitung von UDFs und *PL/pgSQL* Statements und machen uns zum Schluß einige Gedanken zu benötigten Veränderungen in Ausdrücken.

3.1 SQL Anweisungen

Eine große Problematik die wir im Gegensatz zu `SELECT`-Queries bei den DML Statements haben, ist dass die Tabellen die wir gerade bearbeiten in der `FROM` bzw `USING` Klausel nicht bekannt sind und wir dieses Problem auch nicht `LATERAL` behoben bekommen. Die Felder dieser Tabelle sind einfach nicht bekannt. Damit können wir das Lesen der Logtabellen nicht völlig *inline* handhaben sondern müssen als Zwischenschritt eben die ρ der Tabelle lesen und dann die finale Verknüpfung in der `WHERE` Klausel „manuell“ machen. Dies wird später von der Filterfunktion $log.readjointable$ übernommen.

3.1.1 INSERT

Diese Anweisung ist erst einmal die pflegeleichteste der drei betrachteten. Hier finden eh alle Prozesse in Query statt die die einzufügenden Daten ermittelt. Die entsprechende Form ist in 4.3.4.3 zu finden.

Problem bei der gewählten Implementation ist allerdings noch dass bei Anweisungen der

Form in Listing 3.1 Probleme auftreten.

```
1 INSERT INTO TABLE table VALUES v1, ..., vn
```

Listing 3.1: problematisches INSERT

3.1.2 DELETE

Abgesehen von der eingangs erwähnten Problematik standen wir hier vor dem Frage: „Wo bringen wir das Logging unter?“ Ein erster, und final auch gewählter, Ansatz war der, die Schreibfunktion in der `RETURNING` Klausel unterzubringen. Dieses Vorgehen wurde dann spätestens in der Phase als *PL/pgSQL* Funktionen interessant wurde aber wieder in Frage gestellt. Hier stellte sich dann, anders als bei SQL Anweisungen, das Problem, dass wir in Fällen in denen kein `RETURNING` angegeben wurde auch kein `RETURNING` haben wollen.

Die Problematik im `USING` nicht loggen zu können, weil dort die Felder der zu bearbeitenden Tabelle nicht bekannt waren, blieb erhalten. Damit war nur noch die `WHERE` Klausel vorhanden um den gewünschten Seiteneffekt zu erzielen. Hier wurden verschiedene Möglichkeiten in Betracht gezogen aber wieder verworfen.

- `WHERE(writefunction(loc, rho, predicate)`

  Wobei die `writefunction` hier dann erst das Prädikat abfragen und nur im Falle wo dieses wahr ist das Logging vornehmen würde. Problem dabei macht uns diese Lösung aber zum einen wenn wir auf einer Tabelle und Spalten mit Index arbeiten da dieser dann nicht mehr verwendet wird um die zu löschenden Zeilen zu finden. Zum anderen nimmt diese Lösung aber auch deutlich mehr Ausführungszeit in Anspruch selbst wenn nicht auf indizierten Tabellen gearbeitet wird.

- `predicate AND writefunction(loc, rho)`

  Problem in diesem Falle ist die Tatsache dass nicht klar ist wie *Postgres* Prädikate auswertet und wir können so nicht mit Sicherheit sagen dass nicht ein versehentliches Logging stattfindet das uns in Phase `2` später Zeilen löscht die in Phase `1` wegen der `AND`-Verknüpfung unangetastet geblieben sind.

Aus diesen Vorüberlegungen heraus blieb es dann bei der entstandenen Form wie sie in 4.3.4.1 zu finden ist.

3.1.2.1 Why-Provenance

Die ursprüngliche Reaktion auf die Frage was denn die *Why-Provenance* darstellt wenn Dinge gelöscht werden war die: Wenn Dinge gelöscht sind brauchen wir auch nicht über Provenance nachdenken. Die Tatsache dass wir mit der `RETURNING` Klausel aber durchaus auch Daten zurückgeben können bringt das Thema wieder auf den Tisch. Da wir mit *PL/pgSQL* (aber auch mit Hilfe von CTEs in SQL) diese Daten sammeln und wieder in Tabellen einfügen können war klar dass *Why-Provenance* benötigt wird. Anders als bei Queries können wir diese Provenance nicht inlinen sondern benutzen eine Funktion `toYarray` die an jedes Feld im `RETURNING` angehängt wird. Damit wird das ganze zwar nicht so übersichtlich wie die Query-Variante, erfüllt aber seinen Zweck.

3.1.3 UPDATE

Für die *Why-Provenance* benutzen wir die gleiche Funktion wie auch im `DELETE` um die Daten aus dem Prädikat einzusammeln. Diese DP Informationen werden dann an die gesetzten Felder angehängt. Der Logging Prozess findet dort statt wo auch eine neue ρ gebraucht wird: In der `SET` Klausel. Diese Lösung scheint allerdings ungünstig bzw problematisch sobald mehrere mögliche Update Kandidaten vorhanden sind. Transformieren wir die Anweisung aus Listing 1.1 aus dem Einleitungskapitel verbunden sind und rufen wir die Phase $\mathbb{1}$ und $\mathbb{2}$ Anweisungen auf, finden wir die geloggten Informationen wie wir sie in Tabelle 3.1 sehen. Es wurden also nicht nur je eine Zeile pro geupdateter Zeile geloggt sondern zum Teil zwei Zeilen. Dies sorgt im Zweifel im Nachhinein für Probleme, je nachdem welche der geloggten Zeilen in der Phase $\mathbb{2}$ dann verwendet werden. Die gewählte Lösung hat also noch Verbesserungspotential.

location	tuid	tuid1	tuid2
1	$\{\rho_{579}\}$	$\{\rho_{61}\}$	$\{\rho_3\}$
1	$\{\rho_{580}\}$	$\{\rho_{63}\}$	$\{\rho_6\}$
1	$\{\rho_{581}\}$	$\{\rho_{63}\}$	$\{\rho_4\}$
1	$\{\rho_{582}\}$	$\{\rho_{64}\}$	$\{\rho_9\}$
1	$\{\rho_{583}\}$	$\{\rho_{64}\}$	$\{\rho_1\}$

Tabelle 3.1: Fehlerhaft geloggte Daten beim UPDATE

3.2 PL/pgSQL UDF

Bei der Transformation müssen unabhängig von den Anweisungen einige Dinge beachtet werden. Zum einen hat die UDF Ein- und Ausgabewerte. Die Typen der Eingabewerte wurden in Phase 𝟙 belassen wie sie sind, in Phase 2 zum Provenance Mengen Typen geändert. Die Rückgabewerte waren da etwas kritischer. Prinzipiell wurde hier der `VOID` Typ nicht verändert, bei atomaren Datentypen und den Teilen von zusammengesetzten Datentypen wurden ebenfalls *Provenance Sets* gewählt.

Außerdem wurden für beide Phasen jeweils eine ρ Variable in der Liste der Eingabeparameter hinzugefügt. Entsprechend bekamen bei den Rückgabewerten die zusammengesetzten Datentypen ebenfalls ein ρ Feld.

3.3 PL/pgSQL Anweisungen

Durch die Bank weg haben wir das Problem ein passendes ρ für unsere Statements zu brauchen. Dieses Problem lösen wir dass wir als Standard Startwert immer das ρ der Funktion selbst nutzen und dies entsprechend innerhalb von Schleifen abhängig neu setzen.

3.3.1 Schleifen

Um hier zu gewährleisten dass Schleifen immer gleich durchlaufen wurde darauf zurückgegriffen in Phase 1 eine Tabelle mitzuloggen die zusätzlich zum neu erzeugten ρ auch einen Zähler mit erstellt und hochzählt. Diese Tabelle und den Zähler nutzen wir in Phase 2 dann und lassen hier die Schleife über eine Query laufen die diese Tabelle ausliest und das Ergebnis anhand der Zählervariablen sortiert.

Als Problematich erweist sich die Implementation der Schleife über Arrays ebenso wie die Frage nach der *Why-Provenance* wenn `CONTINUE WHEN` und `EXIT WHEN` verwendet werden. Für `CONTINUE WHEN` macht es auf alle Fälle Sinn die Informationen in der Schleife weiterzuverwenden, allerdings wäre es durchaus sinnvoll dies auf Dauer für die Informationen die durch `EXIT WHEN` entstehen auch ausserhalb der Schleife zu kennen.

3.3.2 Zuweisungen

Zuweisungen (**:** = bzw. =) werden einfach dahingehend verarbeitet dass ihnen ihre jeweilige Phase 1 und Phase 2 Repräsentation zugeordnet wird. Im Falle von Zeilenvariablen ist dies aber noch nicht sinnvoll implementiert worden da die ρ Variable dabei verloren geht.

3.3.3 Conditionals

Bei den beiden bedingten Abfragen machen wir uns zu Nutzen dass wir, anders als bei SQL, im Anweisungsteil eine Vielzahl an Anweisungen ausführen können. Deshalb bleibt in Phase 1 das Prädikat in der Abfrage enthalten und wir fügen ein weiteres Statement das die `writeBool` Funktion ausführt hinzu. Wir schreiben in den meisten Fällen nur die Kombination aus Location und ρ und nutzen dann aus dass *Postgres* bei einem booleschen Vergleich ein `NULL` als `False` interpretiert. Das stellt sicher dass wirklich nur Zweige betreten werden die auch in Phase 1 genutzt wurden da die entsprechende `readBool` Funktion dann keinen Eintrag in der Logtabelle finden und `NULL` zurückgeben wird. Diese `readBool` Funktion ersetzt das Prädikat in Phase 2. *Why-Provenance* bei den

Conditionals stellen in jedem Zweig dann, gleich wie beim Case in SQL, alle Informationen dar die in de Zweigen zuvor abgefragt wurden.

3.3.4 Statements mit möglichen Rückgabewerten

Wenn hier Statements verarbeitet wurden die tatsächlich einen Rückgabewert haben und diesen in eine Variable übergeben sind wir fein raus. Auch wenn wir `UPDATE` oder `INSERT` ausführen ohne etwas zurückzugeben. Haben wir in diesem Falle aber ein `DELETE` dann müssen wir, wegen dem oben angesprochenen Problem mit der `writeJoin` Funktion im der `RETURNING` Klausel, hier etwas mehr Veränderung anpacken als in den anderen Fällen. .

Die Lösung in diesem Falle wurde eine leere Schleife, eine Schleife die über die Rückgabewerte läuft ohne eine Aktion dabei auszuführen. Braucht in der Ausführung länger, scheint aber die praktikabelste Lösung zu sein.

3.3.5 Rückgabewerte

Hier gilt bei Rückgaben die einen Zeilentypen zurückgeben das gleiche wie bei den Zuweisungen. Dabei sollte es eigentlich sinnvoll machbar sein dieses Problem aufzufangen und weiterzugeben. Ansonsten wird hier einfach die Phase $\mathbb{1}$, Phase $\mathbb{2}$ Repräsentation ermittelt und verwendet.

3.4 Ausdrücke

Während schon ein reiches Regelwerk für *Expressions* (vgl. [Müller u. a. 2018], [Paradzik 2018]) existiert, wurde es trotzdem interessant einige davon nochmal anzupassen bzw. weiterzuentwickeln.

3.4.1 Parameter

Für unseren Fall war es nötig externe Parameter (deren ID von 1 bis n durchnummeriert sind) zu transformieren weil diese die Eingabewerte unserer UDF repräsentieren. Durch das Einfügen des ρ Verweises am Anfang der Liste müssen diese Parametern-IDs erhöht werden. Außerdem bekommt der Parameter in Phase 2 den Typen *Provenance Set*. Die anderen Parametertypen wurden dabei nicht angepasst.

3.4.2 Array

Mit der Möglichkeit bei einer Schleife über einen Array auch tatsächliche *Provenance Informationen* in einem Array stehen zu haben, wurde die Phase 2 Interpretation dahingehend modifiziert dass wir die einzelnen Werte des Arrays transformieren und die so enthaltenen *Provenance Sets* vereinigen.

3.4.3 UDF

Mit der Implementation von UDFs wurde es natürlich auch interessant die bisherige Implementation, in der die *Provenance* die Vereinigung der *Provenance Sets* der Argumente beim Funktionaufruf darstellt zu verändern.

Die Alternative, bei Funktionen in denen wir Transformationen vorgenommen haben, ist es jetzt dem Funktionsaufruf einen Wert ρ hinzuzufügen. Dazu wird die aktuelle *Location* des Aufrufs und die zugehörige ρ ermittelt und anhand dieser Werte per `writeCall` eine passende ρ vergeben und mitgegeben.

Im Nachhinein betrachtet sollte man evtl. aber darüber nachdenken ob es nicht sinnvoller wäre hier die Funktionen als tabellenwertige Funktionen zu behandeln und hier weitreichendere Änderungen vorzunehmen.

4 Implementation

Dieses Kapitel beschäftigt sich damit wie *SPTransactions* aufgebaut ist und macht sich Gedanken darüber was bei der Anpassung an *SQLProv* vorgenommen wurde. Anhand einer Funktion wird versucht die Transformation von Anweisungen transparent zu machen bevor der große Block das Regelwerk der implementierten Transformationen aufstellt.

4.1 SPTransactions

Zu finden ist das Programm im git des Lehrstuhls für Datenbanksysteme im Repository PgLlProvenance im Branch transactions. Dort enthalten ist auch eine *README* Datei die die Bedienung erklärt.

4.2 Module

Am Anfang stand, wie Eingangs schon erwähnt, das bereits existierende Tool *SQLProv* [1] von Gabriel Paradzik. Um die Möglichkeit zu haben spätere Weiterentwicklungen seines Codes mit überschaubarem Aufwand mit in das neue Tool hineinzupatchen (bzw. andersherum) orientiert sich *SPTransactions* weitestgehend an der bestehenden Modulstruktur. [Paradzik 2018, S. 19] Dabei wurde versucht wenig in den existierenden Daten zu ändern und stattdessen zusätzliche Module angelegt die die neuen Funktionen implementieren. Diese sind namentlich an die Originale angelehnt, das Hauptregelwerk findet sich in *RulesTrans*, der Provenance Helper ist als *ProvHelperTrans* benannt, etc..

[1] Zwischenzeitlich wurde die Version die unter folgendem Hashcode im git zu finden ist eingepflegt: *228fba00bd456f2e7a617af40304ae118f82eede*

4.2.1 Anpassungen

Ganz ohne Änderungen ging der Prozess aber nicht von statten. Vor allem das Kernmodul Rules erforderte einige Anpassungen um die Anbindung zu erlauben. In OptParse war die Änderung so minimal dass die Ergänzung als der sinnvollere Schritt erschien und auch vom Main Modul und der dortigen Hauptroutine erschien es als sinnvoller zu ergänzen anstatt das Rad neu zu erfinden, von dem, Anfangs des Kapitel schon angesprochenen, Hintergrund der Zusammenführbarkeit mal abgesehen.

4.2.1.1 Rules

Hier muss vor allem dafür gesorgt werden dass alles was *SQLProv* nicht abdeckt auf *RulesTrans* umgeleitet wird. Deshalb wird die Funktion `(~~>)` auf `translateQuery` im Modul *RulesTrans* weitergeleitet.

```
(~~>) inp@_ = translateQuery inp
```

Listing 4.1: Nicht abgedeckte Anweisungen werden weitergeleitet

Entsprechendes wird für einige *Expressions*, die in *RulesTrans* anders als als bisher implementiert wurden, verfahren. Diese ist das in Listing 4.2 dargestellt.

```
(~~~>) inp@EParam{}     = translateExpr inp
(~~~>) inp@EFuncCall{} = translateExpr inp
(~~~>) inp@EArrayExpr{} = translateExpr inp
```

Listing 4.2: Nicht abgedeckte Ausdrücke werden weitergeleitet

Der große Teil der Anpassung betrifft dann die *Rule* Monade und deren *State* und *Environment*. Speziell die Implementierung der *PL/pgSQL* Statements und Funktionen hat hier einen Bedarf hervorgerufen der mit zwei neuen Datentypen *TransConfig* (Listing 4.3) und *EnvConfig* (Listing 4.4) angegangen wurde.

```
data TransConfig =
        TransConfig { v1 :: [PLpgVar]
                    , v2 :: [PLpgVar]
```

```
                , y    :: [A.Expr]
                , n    :: Integer }
```

Listing 4.3: ADT für den State Container

```
data EnvConfig      =
          EnvConfig   { rho :: [A.Expr]
                      , udf   :: [String]
                      , composite :: Bool }
```

Listing 4.4: ADT für den Environment Container

Im Abschnitt „Die Monade“ findet sich die Beschreibung zum Sinn der einzelnen Felder

Diese Definition erspart uns schlichtweg die Verwendung von hässlichen, ausartenden Tupeln auch wenn mit der Verwendung von *Lenses* der Zugriff auf die einzelnen Tupeleinträge noch einigermaßen schön realisierbar gewesen wäre.

Der Versuch diese im Modul *RulesTrans* zu definieren scheiterte an der Problematik dass dann eine gegenseitige Abhängigkeit (vergleiche Abschnitt 4.2.2) zwischen den beiden Regelwerk Modulen aufgetreten wäre die sich nicht mehr hätten beheben lassen.

Mit dieser Anpassung zusammenhängend ändert sich in der Signaturdefinition von *Rule* (Listing 4.5) dann eben die Verwendung von *State* und *Environment* wie in Listing 4.5 dargestellt sowie die (trivialen) Startwerte für die `runRule` Funktion.

```
type Rule a b =
 a -> OperSem
       (Integer,TransConfig)        -- State
       (RuleConfig, EnvConfig)      -- Environment
       b                            -- Output
        [TableHead]                 -- Logs
```

Listing 4.5: Neue Signaturdefinition

Final wurde dann die von *SQLProv* verwendeten Funktionen die mit *State* und *Environment* in Zusammenhang stehen angepasst werden. Dabei handelt es sich um die Funktionen

`yProvEnabled`, `ifYProv`, `setRhoCtx`, `setRhoCtx'` und `getRhoCtx`. Neben den entsprechenden Signaturergänzungen und -modifikationen musste der Zugriff auf das Tupel und den Inhalt des Containers geändert werden.

4.2.1.2 OptParse

Hier wurden minimale Ergänzungen vorgenommen (deshalb hier auch die Manipulation im vorhandenen Modul und kein neues). Da es unter Umständen von Vorteil sein kann keine zusätzlichen Tabellen zu generieren wird eine zusätzliche Option für den Programmaufruf hinzugefügt. Mit den Schaltern `--no-table` bzw `-0` kann diese Vorbereitung unterdrückt werden. Dazu wird der Algebraic Data Type (ADT) `Options` um `oPrepare` ergänzt und dem *Option Parser* die entsprechende Option hinzugefügt.

4.2.1.3 Main

Hier beginnen die Änderungen damit die Eingabe-Query mit der Funktion `parseQueryAndUDFs` (anstatt nur die Query) zu parsen. Diese liefert uns die SQL Query und eine Liste von *PL/pgSQL* Funktionen.

Beides geben wir mitsamt der Liste der Namen der UDFs an die Übersetzungsfunktion `translateAll` aus dem *RulesTrans* Modul weiter.

Schließlich wird hier noch die Option ob die Tabellen vorbereitet werden sollen sowie die Formatierungsoptionen für die UDFs abgefragt und angewandt bevor die komplette Ausgabe erfolgt.

4.2.2 Gegenseitige Abhängigkeiten

Wie in Abbildung 4.2.1.1 aufgezeigt, rufen die Module *Rules* und *RulesTrans* sich gegenseitig auf. Dieser gegenseitige, rekursive Import [2] ist mit dem Haskell Compiler *GHC* aber leider nicht völlig unfallfrei zu bewerkstelligen.

[2] https://wiki.haskell.org/Mutually_recursive_modules

Da wir ohne diesen Import aber nur im alten Modul weiter hätten arbeiten können, musste als Fix zum einen für eines der Module (hier das Modul *Rules*) eine zusätzliche *.hs-boot Datei* erzeugt werden. Dann wird dieses Modul im anderen Modul über den Befehl im Listing 4.6 importiert.

```
import {-# SOURCE #-} Rules
```

Listing 4.6: Import des Rules Moduls

In der neu erzeugten Datei *Rules.hs-boot* werden die Signaturen der Funktionen angegeben die von *RulesTrans* verwendet werden.

4.3 Regelwerk

Den Kern des ganzen Programms bilden die beiden Module *Rules* und *RulesTrans*. Beide sind dafür zuständig gegebene Anweisungen zu übersetzen. Seien es Queries, DML Statements, Ausdrücke oder *PL/pgSQL* Statements und Funktionen.

Dabei werden in den Fällen *SQL* und *PL/pgSQL* im Grundsatz verschiedene ASTs verarbeitet, wobei wir im *PL/pgSQL*-Baum auch auf SQL Elemente treffen können. Da dieser Baum parametrisiert auftritt, besteht hier auch die Möglichkeit ihn als *Funktor* Instanz zu definieren und damit `fmap` darauf zu verwenden.

Die eigentliche Übersetzung wird dann mit der Funktion `translateAll` initiert die den Aufruf der Übersetzungsfunktionen für SQL (`(~~>)`) und *PL/pgSQL* (`translateUDF`) vornimmt. In Listing 4.8 ist die letztere aufgezeigt. Zur besseren Veranschaulichung sei in Listing 4.7 auch noch der AST einer *PL/pgSQL* UDF dargestellt. Dabei enthält die `dataArea` die Variablenliste und der `block` eine Liste mit *PL/pgSQL* Anweisungen.

Die Funtion benennt die UDF um und ermittelt welche zusätzlichen Variablen gebraucht werden bzw. modifiziert die vorhandenen Variablen. Dabei muss die ID jeder Variable erhöht werden, da am Anfang später eine Variable für die Tuid ρ eingefügt wird, ebenso die Namen (`$1,...`) der Parameter angepasst werden die für den Funktionsaufruf

verwendet werden. Für Zeilenvariablen wird ein zusätzliches ρ Feld geschaffen. Sollten Variablen schon eine Zuweisung (in Form einer Query) erhalten haben, wird diese in die Repräsentation der beiden Phasen transformiert. Abschließend werden die Namen von möglichen tabellenabhängigen Zeilentypen angepasst. Und jeweils noch eine Variable für die ρ der UDF eingefügt.

Ab Zeile 16 beginnt dann die eigentliche Transformation der Statements. Das *Pattern Synonym* `RhoCC` stellt dabei ein bidirektionales Pattern dar dass wir in der Folge zum Pattern Matching nutzen aber auch zum Erstellen eine neuen Pärchens von ρ Variablen nutzen können. Dieses neue Pärchen setzen wir ins *Environment* und übersetzen mit diesem Kontext dann die Statements.

Abschließend holen wir uns dann die finale Variante der Variablenlisten aus dem *State* und sorgen noch dafür dass auch im Rückgabewert der Funktion die richtigen Typen vorkommen.

Schließlich geben wir dann die beiden Funktionen mit den ermittelten Werten zurück.

```
1 type PLpgUDF     = PLpgUDFF PLpgVar A.Type PLpgStmt
2 data PLpgUDFF a b c = PLpgFunction
3       { oid         :: Integer
4       , pludfname   :: String
5       , dataArea    :: [ a ]
6       , block       :: [ c ]
7       , rettype     :: b
8       }
```

Listing 4.7: AST einer PLpgUDF

```
1  translateUDF :: Rule PLpgUDF (PLpgUDF, PLpgUDF)
2  translateUDF inp@A.PLpgFunction{..}= do
3    let p1named  = phase1Tablename pludfname
4    let p2named' = phase2Tablename pludfname
5    p2named <- ifYProv (p2named'++"y") p2named'
6
7    setVarList dataArea []
8    (area1, area2)
9          <- mapAndUnzipM translatePLpgVAR dataArea
10   (_,new) <- getVarLists
11   let (new1,new2) = (unzip . (map H.fixVarNames)) new
12   let udfTuid = DefVar "$1" 0 sqlTypeInt
13   setVarList ((udfTuid:area1)++new1)
14              ((udfTuid:area2)++new2)
15
16   let translateThem
17        = setRhoCtx (RhoCC defaultParam defaultParam)
18                    translateSTMTs
19   (block1, block2) <- mapAndUnzipM translateThem block
20   (area1'', area2'') <- getVarLists
21   let (rettype1, rettype2) = H.translateRetType rettype
22
23   return (inp{pludfname=p1named
24               , dataArea = area1''
25               , block=block1
26               , rettype=rettype1}
27           , inp{pludfname=p2named
28               , dataArea = area2''
29               , block=block2
30               , rettype=rettype2})
```

Listing 4.8: Übersetzungsfunktion einer UDF

4.3.1 Die Monade

Der ganze Transformationsprozess findet in der *Rule Monade* statt. In dieser Monade rufen wir rekursiv weitere Übersetzungsschritte auf und behalten dabei im *State* Listen und Informationen die wir im Verlauf dieser Übersetzung benötigen und die dabei auch verändert werden. Dazu gehört die Erzeugung eines Zählers für die aktuelle Aufrufs-Location wie sie auch in *SQLProv* verwendet wurde, die beiden Variablenlisten für *PL/pgSQL* Funktionen sowie ein Variablenzähler der den Namen von neu erzeugten Variablen mit einem Zähler versieht. Zusätzlich sammeln wir in *PL/pgSQL* Funktionen die *Why-Provenance* mit im *State*. Dies hätte an sich zwar auch im *Environment* reingepasst, die Tatsache dass wir bei aber auch die Informationen aus dem verwendeten Prädikat in der entsprechenden Schleife uns für nachfolgende *Statements* merken wollen, hat die Implementation im *State* nötig gemacht.

Dazu existiert ein *Environment* in denen wir lokale Variablen vergeben. Die erste dieser Variablen ist die Verwendung des Schalters für die *Why-Provenance*. Zusätzlich merken wir uns im *Environment* die Namen der UDFs die wir geparset haben. Diese Information wird dann bei der Transformation von UDF (4.3.5.2 relevant. Final brauchen wir an manchen Stellen die Informationen darüber ob die aktuell bearbeitete UDF einen zusammengesetzten Datentyp als Rückgabe hat.

4.3.2 Der Abstract Syntax Tree

Der AST fasst uns die Informationen die der *LogParser* aus *Postgres* extrahiert hat mehr oder weniger übersichtlich in einem ADT zusammen. Viele der Informationen sind für uns interessant, leider enthält er aber auch einiges an Informationen die für unsere Zwecke uninteressant ist. Die *Location* einer Query gehören dazu oder die *OID* einer UDF. Da manche dieser Datentypen inzwischen bis zu 19 Felder haben, verwenden wir, die eingangs erwähnten, *Pattern Synonyms* und *Record Wildcards* die uns ein übersichtlicheres *Pattern Matching* ermöglichen. Dabei finden auch die erwähnten bidirektionalen Pattern Verwendung um als Konstruktoren herzuhalten.

Leider stelle ich im Nachhinein fest, dass hier der Einsatz von „normalen“ Funktionen und

bidirektionalen Patterns etwas inkonsistent und schwammig gehandhabt wurde.

4.3.3 Vorbereitung

Anders als bei *SQLProv* wo *Views* verwendet wurden, wählen wir hier Tabellen zur Speicherung und vor allem Manipulation der beiden Tabellen. Sofern der Nutzer es nicht explizit wünscht keine Tabellen zu generieren, werden im Endeffekt die gleichen Vorbereitungsmaßnahmen ergriffen wie bei *SQLProv*, nur dass (temporäre) Tabellen erstellt werden. Das basiert einfach auf der Tatsache dass wir *Materialized Views* gar nicht verändern können und *Views* nicht manipulieren können ohne die Daten der Originaltabelle dabei auch zu verändern. Dieses Original wollen wir aus Gründen der Nachverfolgung der Änderungen im Zweifel aber gerne noch verfügbar haben. Wie im Abschnitt 4.2.1.2 schon beschrieben, wurde dem Programm die Option hingefügt diese Vorbereitungsschritte zu überspringen. Eine Alternative für die Zukunft könnte hier sein die Phase 𝟙 Tabelle direkt zur Verfügung zu stellen (was gerade bei Tabellen die mit Indexen oder irgendwelchen Constraints arbeiten auf alle Fälle Sinn machen könnte).

So wie die Arbeit ist, gibt es soweit nur die Option „ganz oder gar nicht“. Wenn wir die Vorbereitungsschritte nicht unterdrücken, wird eine Sequenz *ProvId* erstellt und für jede der Tabellen nach dem Schema aus den Grundlagen je eine Phase 𝟙 und eine Phase 𝟚 Tabelle erstellt.

Anschließend geben wir die transformierten UDFs aus bevor schließlich die verwendeten Phase 𝟙 und Phase 𝟚 Anweisungen ausgegeben werden.

Die folgenden Abschnitte enthalten die dabei verwendeten Regeln. Zuerst für die SQL Anweisungen `UPDATE, DELETE, INSERT`. Schließlich für die veränderten Ausdrücke *Arrays*, *Parameter* und UDF bevor dann der große Abschnitt mit *PL/pgSQL* Statements behandelt wird.

4.3.4 Regeln für SQL Anweisungen

Hier, und auch in den folgenden Kapiteln, wird der Einsatz von optionalen Why-Provenance Informationen grau hinterlegtwerden.

4.3.4.1 DELETE

$$\mid q_i \Mapsto \langle q_i^{\mathbb{1}}, q_i^{?} \rangle \mid_{i=1..n}, \quad p \Mapsto \langle p^{\mathbb{1}}, p^{?} \rangle, \quad \mid e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{?} \rangle \mid_{i=1..m}, \quad \textcircled{\ell} = generateId$$

$$i^{\mathbb{1}} = \begin{array}{ll} \text{DELETE} & \text{FROM } q_1^{\mathbb{1}} \text{ AS } t_1 \\ \text{USING} & q_2^{\mathbb{1}} \text{ AS } t_2, ..., q_n^{\mathbb{1}} \text{ AS } t_n \\ \text{WHERE} & p^{\mathbb{1}} \\ \text{RETURNING} & log.write_{join}(\textcircled{\ell}, t_1.\rho, t_2.\rho, ..., t_n.\rho), e_1 \text{ AS } c_1, ..., e_m \text{ AS } c_m \end{array}$$

$$i^{?} = \begin{array}{ll} \text{DELETE} & \text{FROM } q_1^{?} \text{ AS } t_1 \\ \text{USING} & q_2^{?} \text{ AS } t_2, ..., q_n^{?} \text{ AS } t_n \\ & \text{LATERAL } log.read_{jointable}(\textcircled{\ell}, t_2.\rho, ..., t_n.\rho) \\ & \qquad \text{AS } filter(\rho, \rho_1) \\ \text{WHERE} & t_1.\rho = filter.\rho_1 \\ \text{RETURNING} & filter.\rho \text{ AS } \rho, \\ & e_1^{?} \cup toYarray(p^{?}) \text{ AS } c_1, ..., e_m^{?} \cup toYarray(p^{?}) \text{ AS } c_m \end{array}$$

$$\begin{array}{ll} \text{DELETE} & \text{FROM } q_1 \text{ AS } t_1 \\ \text{USING} & q_2 \text{ AS } t_2, ..., q_n \text{ AS } t_n \\ \text{WHERE} & p \\ \text{RETURNING} & e_1 \text{ AS } c_1, ..., e_m \text{ AS } c_m \end{array} \quad \Mapsto \langle i^{\mathbb{1}}, i^{?} \rangle$$

4.3.4.2 DELETE (alles)

Um unnötiges Loggen beim (vorsätzlichen) Löschen aller Daten zu vermeiden, wurde dieser Spezialfall implementiert. Sollte doch ein Logging gewünscht sein, kann dies mit einem einfachen Workaround in die erste Variante umgeschrieben werden, dazu braucht es nur ein `WHERE true` als „Schalter“ um dies zu ermöglichen.

$$q_i \Rightarrow \langle q_i^{\mathbb{1}}, q_i^{\mathbb{2}} \rangle, \quad \mid e_i \Rightarrow \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \mid_{i=1..m}$$

$$i^{\mathbb{1}} = \begin{array}{ll} \text{DELETE} & \text{FROM } q_1^{\mathbb{1}} \text{ AS } t_1 \\ \text{RETURNING} & q_1^{\mathbb{1}}.\rho, e_1 \text{ AS } c_1, ..., e_m \text{ AS } c_m \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{ll} \text{DELETE} & \text{FROM } q_1^{\mathbb{2}} \text{ AS } t_1 \\ \text{RETURNING} & q_1^{\mathbb{2}}.\rho \text{ AS } \rho, e_1^{\mathbb{2}} \text{ AS } c_1, ..., e_m^{\mathbb{2}} \text{ AS } c_m \end{array}$$

$$\begin{array}{ll} \text{DELETE} & \text{FROM } q_1 \text{ AS } t_1 \\ \text{RETURNING} & e_1 \text{ AS } c_1, ..., e_m \text{ AS } c_m \end{array} \Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$$

4.3.4.3 INSERT

$$q \Rightarrow \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle, \quad query \Rightarrow \langle query^{\mathbb{1}}, query^{\mathbb{2}} \rangle, \quad \mid e_i \Rightarrow \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \mid_{i=1..m}$$

$$i^{\mathbb{1}} = \begin{array}{ll} \text{INSERT} & \text{INTO } q^{\mathbb{1}} \text{ AS } t \\ & query^{\mathbb{1}} \\ \text{RETURNING} & t.\rho, e_1^{\mathbb{1}} \text{ AS } c_1, ..., e_m^{\mathbb{1}} \text{ AS } c_m \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{ll} \text{INSERT} & \text{INTO } q^{\mathbb{2}} \text{ AS } t \\ & query^{\mathbb{2}} \\ \text{RETURNING} & t.\rho, e_1^{\mathbb{2}} \text{ AS } c_1, ..., e_m^{\mathbb{2}} \text{ AS } c_m \end{array}$$

$$\begin{array}{ll} \text{INSERT} & \text{INTO } q \text{ AS } t \\ & query \\ \text{RETURNING} & e_1 \text{ AS } c_1, ..., e_m \text{ AS } c_m \end{array} \Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$$

4.3.4.4 UPDATE

$$\mid q_i \Mapsto \langle q_i^{\mathbb{1}}, q_i^{\mathbb{2}} \rangle \mid_{i=1..n}, \quad \mid e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \mid_{i=1..t}, \quad p \Mapsto \langle p^{\mathbb{1}}, p^{\mathbb{2}} \rangle, \quad \textcircled{\ell} = generateId$$

$i^{\mathbb{1}} =$

UPDATE	$q_1^{\mathbb{1}}$ AS t_1
SET	$\rho = log.write_{join}(\textcircled{\ell}, t_1.\rho, t_2.\rho, ..., t_n.\rho)$,
	$c_1 = e_1^{\mathbb{1}}, ..., c_k = e_k^{\mathbb{1}}$
FROM	$q_2^{\mathbb{1}}$ AS $t_2, ..., q_n^{\mathbb{1}}$ AS t_n
WHERE	$p^{\mathbb{1}}$
RETURNING	$t_1.\rho$ AS $\rho, e_l^{\mathbb{1}}$ AS c_l, ..., $e_t^{\mathbb{1}}$ AS c_t

$i^{\mathbb{2}} =$

UPDATE	$q_1^{\mathbb{2}}$ AS t_1
SET	$\rho = filter.\rho$,
	$c_1 = e_1^{\mathbb{2}} \cup toYarray(p^{\mathbb{2}}), ..., c_k = e_k^{\mathbb{2}} \cup toYarray(p^{\mathbb{2}})$
FROM	$q_2^{\mathbb{2}}$ AS $t_2, ..., q_n^{\mathbb{2}}$ AS t_n
	LATERAL $log.read_{jointable}(\textcircled{\ell}, t_2.\rho, ..., t_n.\rho)$ AS $filter(\rho, \rho_1)$
WHERE	$t_1.\rho = filter.\rho_1$
RETURNING	$filter.\rho$ AS $\rho, e_l^{\mathbb{2}}$ AS c_l, ..., $e_t^{\mathbb{2}}$ AS c_t

---

UPDATE	q_1 AS t_1
SET	$c_1 = e_1, ..., c_k = e_k$
FROM	q_2 AS $t_2, ..., q_n$ AS $t_n \Mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$
WHERE	p
RETURNING	e_l AS c_l, ..., e_t AS c_t

4.3.5 Geänderte Regeln für Expressions

4.3.5.1 Array

Eine Änderung der bisherigen Herangehensweise, in Phase 2 den Array als $\emptyset$ zu betrachten, wurde interessant durch die Möglichkeit dass in *PL/pgSQL* Funktionen Informationen mit *Provenance* gespeichert sein könnten.

$$\mid e_i \Mapsto \langle e_i^{\mathbb{1}}, es_i^{\mathbb{2}} \rangle \mid_{i=1..n}$$

$$\frac{i^{\mathbb{1}} = \quad array[e_1^{\mathbb{1}}, ..., e_n^{\mathbb{1}}] \qquad\qquad i^{\mathbb{2}} = \quad e_1^{\mathbb{2}} \cup \ ... \ \cup e_n^{\mathbb{2}}}{array[e_1, ..., e_n] \Mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle}$$

4.3.5.2 UDF

Hier haben wir leider den Bug mit drin, dass die Argumente der Funktion eigentlich die Why-Provenance mitbekommen sollten, wir diese Information aber bei der Transformation der UDF nicht kennen. Unter Umständen ließe sich dies unter der Verwendung des *Y-Contexts* auch in Queries oder Mapping in irgendeiner Form angehen.

$$\mid e_i \Mapsto \langle e_i^{\mathbb{1}}, es_i^{\mathbb{2}} \rangle \mid_{i=1..n}, \quad (\rho^{\mathbb{1}}, \rho^{\mathbb{2}}) = getRhoCtx, \quad \textcircled{\ell} = generateId$$

$$\frac{i^{\mathbb{1}} = \quad f(write_{call}(\textcircled{\ell}, \rho^{\mathbb{1}}), e_1^{\mathbb{1}}, ..., e_n^{\mathbb{1}}) \qquad i^{\mathbb{2}} = \quad f(read_{call}(\textcircled{\ell}, \rho^{\mathbb{2}}), e_1^{\mathbb{2}}, ..., e_n^{\mathbb{2}})}{f(e_1, ..., e_n) \Mapsto \begin{array}{ll} \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle & \text{wenn UDF die geparsed wurde} \\ \langle f(e_1^{\mathbb{1}}, ..., e_n^{\mathbb{1}}), e_1^{\mathbb{2}} \cup ... \cup e_n^{\mathbb{2}}) \rangle & \text{sonst} \end{array}}$$

4.3.5.3 Parameter

$$EParam\ id\ type \Mapsto \langle EParam\ id + 1\ type, EParam\ id + 1\ ProvenanceSet \rangle$$

4.3.6 PL/pgSQL

Hier finden sich die Übersetzungsregeln für die Anweisungen die von *PL/pgSQL* Funktionen erzeugt und genutzt werden. Bei diesem Abschnitt kommen verstärkt *State* und *Environment* zum Einsatz um übertrieben viele Variablenbindung und -einführung in den Funktionen zu vermeiden. Dabei wird der Einsatz des Setzens von Why kontext Informationen blau notiert während das setzen von ρ Kontext rot markiert ist.

4.3.6.1 BLOCK

$$\mid s_i \mapsto \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \mid_{i=1..n}$$

$$\frac{\begin{array}{ll} & \text{BEGIN} \\ i^{\mathbb{1}} = & \quad s_1^{\mathbb{1}}, ..., s_n^{\mathbb{1}} \\ & \text{END;} \end{array} \qquad \begin{array}{ll} & \text{BEGIN} \\ i^{2} = & \quad s_1^{2}, ..., s_n^{2} \\ & \text{END;} \end{array}}{\begin{array}{l} \text{BEGIN} \\ \quad s_1, ..., s_n \mapsto \langle i^{\mathbb{1}}, i^{2} \rangle \\ \text{END;} \end{array}}$$

4.3.6.2 RETURN

$$e \mapsto \langle e^{\mathbb{1}}, e^{2} \rangle$$

$$\text{RETURN NEXT } var \mapsto \langle \text{ RETURN NEXT } var + 1, \text{RETURN NEXT } var + 1 \rangle$$

$$\text{RETURNNEXT } e \mapsto \langle \text{ RETURN NEXT } e^{\mathbb{1}}, \text{RETURN NEXT } e^{2} \rangle$$

4.3.6.3 RETURN NEXT

$$e \mapsto \langle e^{\mathbb{1}}, e^{2} \rangle$$

$$\text{RETURN } var \mapsto \langle\ \text{RETURN } var + 1, \text{RETURN } var + 1\ \rangle$$

$$\text{RETURN } e \mapsto \langle\ \text{RETURN } e^{\mathbb{1}}, \text{RETURN } e^{2}\ \rangle$$

4.3.6.4 RETURN QUERY

$$q \mapsto \langle q^{\mathbb{1}}, q^{2} \rangle$$

$$\text{RETURN QUERY } q \mapsto \langle\ \text{RETURN QUERY } q^{\mathbb{1}}, \text{RETURN QUERY } q^{2}\ \rangle$$

4.3.6.5 ASSIGN

$$e \mapsto \langle e^{\mathbb{1}}, e^{2} \rangle, \quad Y = getYCtx$$

$$\frac{i^{\mathbb{1}} = \quad v := e^{\mathbb{1}} \qquad\qquad i^{2} = \quad v := e^{2} \ \cup Y}{v := e \mapsto \langle i^{\mathbb{1}}, i^{2} \rangle}$$

4.3.6.6 EXECSQL = Query mit möglichem Rückgabewert ausführen

Der *LogParser* liefert uns hier beim parsen von Statements die eine Zuweisung (`INTO`) haben statt der gewünschten Variablen eine Zeilenvariablen `*internal*` die wir im Falle der Fälle umbenennen (in die Namen ihrer Einzelvariablen) und rufen mit dieser veränderten Zielvariablen die Übersetzung nochmal auf.

$$query \mapsto \langle query^{\mathbb{1}}, query^{2} \rangle$$

$$i^{\mathbb{1}} = \begin{array}{l} \text{FOR} \quad query\_dummy \quad \text{IN} \quad query^{\mathbb{1}} \\ \text{LOOP} \\ \text{END} \quad \text{LOOP;} \end{array} \qquad i^{2} = \quad query^{2}$$

$$query \mapsto \begin{array}{ll} \langle i^{\mathbb{1}}, i^{2} \rangle & \text{wenn DELETE ohne } \textit{RETURNING} \\ \langle query^{\mathbb{1}}, query^{2} \rangle & \text{sonst} \end{array}$$

4.3.6.7 IF ... THEN ... ELSE ...

$$e \mapsto \langle e^{\mathbb{1}}, e^{2} \rangle, \quad \mid s_i \mapsto \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \mid_{i=1..t}, \quad \ell = generateId, \quad \langle \rho^{\mathbb{1}}, \rho^{2} \rangle = getRhoCtx$$
$$add2YCtx(e^{2})$$

$$i^{\mathbb{1}} = \begin{array}{lll} \text{IF} & e^{\mathbb{1}} & \\ & \text{THEN} & \text{PERFORM } write_{bool}(\ell, \rho^{\mathbb{1}}) \\ & & s_1^{\mathbb{1}}, ..., s_k^{\mathbb{1}} \\ & \text{ELSE} & s_l^{\mathbb{1}}, ..., s_t^{\mathbb{1}} \\ \text{END} & \text{IF;} & \end{array}$$

$$i^{2} = \begin{array}{lll} \text{IF} & read_{bool}(\ell, \rho^{2}) & \\ & \text{THEN} & s_1^{2}, ..., s_k^{2} \\ & \text{ELSE} & s_l^{2}, ..., s_t^{2} \\ \text{END} & \text{IF;} & \end{array}$$

$$\begin{array}{lll} \text{IF} & e & \\ & \text{THEN} & s_1, ..., s_k \\ & \text{ELSE} & s_l, ..., s_t \\ \text{END} & \text{IF;} & \end{array} \mapsto \langle i^{\mathbb{1}}, i^{2} \rangle$$

4.3.6.8 CASE...

$$\mid s_i \Mapsto \langle s_i^{\mathbb{1}}, s_i^2 \rangle \mid_{i=1..n}, \quad , e \mapsto \langle e^{\mathbb{1}}, e^2 \rangle, \quad add2YCtx(e^2$$

$$i^{\mathbb{1}} = \begin{array}{lll} \text{CASE} & e^{\mathbb{1}} & \\ & s_1^{\mathbb{1}}, ..., s_n^{\mathbb{1}} & \\ \text{END} & \text{CASE} & \end{array} \qquad i^2 \begin{array}{lll} \text{CASE} & & \\ & s_1^2, ..., s_n^2 & \\ \text{END} & \text{CASE} & \end{array}$$

$$\begin{array}{lll} \text{CASE} & e & \\ & s_1, ..., s_n & \Mapsto \langle i^{\mathbb{1}}, i^2 \rangle \\ \text{END} & \text{CASE} & \end{array}$$

4.3.6.8.1 WHEN ...

$$\mid s_i \Mapsto \langle s_i^{\mathbb{1}}, s_i^2 \rangle \mid_{i=1..n}, \quad , e \mapsto \langle e^{\mathbb{1}}, e^2 \rangle, \quad \ell = generateId, (\rho^{\mathbb{1}}, \rho^2) = getRhoCtx$$
$$add2YCtx(e^2)$$

$$i^{\mathbb{1}} = \begin{array}{ll} \text{WHEN} & e^{\mathbb{1}} \\ \text{THEN} & \text{PERFORM } log.write_{bool}(\ell, \rho^1), \\ & s_1^{\mathbb{1}}, ...,^{\mathbb{1}} \end{array}$$

$$i^2 = \begin{array}{ll} \text{WHEN} & log.read_{bool}(\ell, \rho^2) \\ \text{THEN} & s_1^2, ..., s_n^2 \end{array}$$

$$\begin{array}{lll} \text{WHEN} & e & \\ \text{THEN} & s_1, ..., s_n & \Mapsto \langle i^{\mathbb{1}}, i^2 \rangle \end{array}$$

4.3.6.9 LOOP

$$\textcircled{\ell} = generateId, \quad \langle \rho^{\mathbb{1}}, \rho^{2} \rangle = getRhoCtx$$
$$setRhoCtx(ctuid^{\mathbb{1}}, ctduid^{2}), \quad \mid s_i \Mapsto \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \mid_{i=1..n}$$

$query^{2}$= SELECT $loop.\rho$ AS ρ
FROM $log.read_{loop}(\textcircled{\ell}, \rho^{2})$ AS $loop(\rho, v)$
ORDER BY $loop.v$ ASC

$i^{\mathbb{1}} =$ LOOP
$cTuid^{\mathbb{1}} := log.write_{log}(\textcircled{\ell}, \rho^{\mathbb{1}})$,
$s_1, ..., s_n$
END LOOP;

$i^{2} =$ FOR $cTuid^{2}$ IN $query^{2}$
LOOP
$s_1^{2}, ..., s_n^{2}$
END LOOP;

LOOP
$s_1, ..., s_n \Mapsto \langle i^{\mathbb{1}}, i^{2} \rangle$
END LOOP;

4.3.6.10 CONTINUE

$$e \mapsto \langle e^{\mathbb{1}}, e^{2} \rangle, (\rho^{\mathbb{1}}, \rho^{2}) = getRhoCtx, \textcircled{\ell} = generateId, \quad add2YCtx(e^{2})$$

$$i^{\mathbb{1}} = \quad \text{CONTINUE WHEN } write_{bool}(\textcircled{\ell}, \rho^{\mathbb{1}}, e^{\mathbb{1}})$$

$$i^{2} = \quad \text{CONTINUE WHEN } read_{bool}(\textcircled{\ell}, \rho^{2})$$

$$\text{CONTINUE WHEN } p \quad \mapsto \langle i^{\mathbb{1}}, i^{2} \rangle$$

4.3.6.11 EXIT

$$e \mapsto \langle e^{\mathbb{1}}, e^{2} \rangle, (\rho^{\mathbb{1}}, \rho^{2}) = getRhoCtx, \textcircled{\ell} = generateId$$

$$i^{\mathbb{1}} = \quad \text{EXIT WHEN } write_{bool}(\textcircled{\ell}, \rho^{\mathbb{1}}, e^{\mathbb{1}})$$

$$i^{2} = \quad \text{EXIT WHEN } read_{bool}(\textcircled{\ell}, \rho^{2})$$

$$\text{EXIT WHEN } e \quad \mapsto \langle i^{\mathbb{1}}, i^{2} \rangle$$

4.3.6.12 WHILE

$$\textcircled{\ell} = generateId, \quad p \mapsto \langle p^{\mathbb{1}}, p^{\mathbb{2}} \rangle, \quad \langle \rho^{\mathbb{1}}, \rho^{\mathbb{2}} \rangle = getRhoCtx$$
$$setRhoCtx(ctuid^{\mathbb{1}}, ctduid^{\mathbb{2}}), \quad \mid s_i \mapsto \langle s_i^{\mathbb{1}}, s_i^{\mathbb{2}} \rangle \mid_{i=1..n}, \quad add2YCtx(p^{\mathbb{2}})$$

$$query^{\mathbb{2}} = \begin{array}{ll} \text{SELECT} & loop.\rho \text{ AS } \rho \\ \text{FROM} & log.read_{loop}(\textcircled{\ell}, \rho^{\mathbb{2}}) \text{ AS } loop(\rho, v) \\ \text{ORDER BY} & loop.v \text{ ASC} \end{array}$$

$$i^{\mathbb{1}} = \begin{array}{l} \text{WHILE} \quad p^{\mathbb{1}} \\ \text{LOOP} \\ \qquad cTuid^{\mathbb{1}} := log.write_{log}(\textcircled{\ell}, \rho^{\mathbb{1}}) \\ \qquad s_1^{\mathbb{1}}, ..., s_n^{\mathbb{1}} \\ \text{END} \quad \text{LOOP;} \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{l} \text{FOR} \quad cTuid^{\mathbb{2}} \quad \text{IN} \quad query^{\mathbb{2}} \\ \text{LOOP} \\ \qquad s_1^{\mathbb{2}}, ..., s_n^{\mathbb{2}} \\ \text{END} \quad \text{LOOP;} \end{array}$$

$$\begin{array}{ll} \text{WHILE} & p \\ \text{LOOP} & \\ & s_1, ..., s_n \end{array} \mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$$
$$\text{END} \quad \text{LOOP;}$$

4.3.6.13 FOR (over arrays)

Die Implementation ist hier leider noch etwas verbuggt und vermutlich eher undurchdacht.

$$v \mapsto \langle v^{\mathbb{1}}, v^{\mathbb{2}} \rangle, \quad x \mapsto \langle x^{\mathbb{1}}, x^{\mathbb{2}} \rangle, a \mapsto \langle a^{\mathbb{1}}, a^{\mathbb{2}} \rangle, \quad \langle \rho^{\mathbb{1}}, \rho^{\mathbb{2}} \rangle = getRhoCtx$$

$$\textcircled{\ell} = generateId, \quad setRhoCtx(ctuid^{\mathbb{1}}, ctduid^{\mathbb{2}}), \quad \mid s_i \mapsto \langle s_i^{\mathbb{1}}, s_i^{\mathbb{2}} \rangle \mid_{i=1..n}$$

$$query^{\mathbb{2}} = \begin{array}{ll} \text{SELECT} & loop.\rho \text{ AS } \rho \\ \text{FROM} & log.read_{loop}(\textcircled{\ell}, \rho^{\mathbb{2}}) \text{ AS } loop(\rho, b) \\ \text{ORDER BY} & loop.v \text{ ASC} \end{array}$$

$$i^{\mathbb{1}} = \begin{array}{l} \text{FOR EACH} \quad v^{\mathbb{1}} \quad \text{SLICE } x^{\mathbb{1}} \text{ IN ARRAY} \quad a^{\mathbb{1}} \\ \text{LOOP} \\ \qquad cTuid^{\mathbb{1}} := log.write_{log}(\textcircled{\ell}, \rho^{\mathbb{1}}), \\ \qquad s_1, ..., s_n \\ \text{END LOOP;} \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{l} \text{FOR} \quad cTuid^{\mathbb{2}} \quad \text{IN} \quad query^{\mathbb{2}} \\ \text{LOOP} \\ \qquad s_1^{\mathbb{2}}, ..., s_n^{\mathbb{2}} \\ \text{END} \quad \text{LOOP;} \end{array}$$

$$\begin{array}{l} \text{FOR EACH} \quad v \quad \text{SLICE } x \text{ IN ARRAY} \quad a \\ \text{LOOP} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle \\ \qquad s_1, ..., s_n \\ \text{END LOOP;} \end{array}$$

4.3.6.14 FOR (over integers)

$$\langle \rho^{\mathbb{1}}, \rho^{2} \rangle = getRhoCtx, \quad (\ell) = generateId$$

$$(x, y, z) \mapsto \langle (x^{\mathbb{1}}, y^{\mathbb{1}}, z^{\mathbb{1}}), (x^{2}, y^{2}, z^{2}) \rangle, \quad add + set0v^{2}, yv, \quad add2YCtx(e^{2})$$

$$query^{2} = \begin{array}{ll} \text{SELECT} & loop.\rho \text{ AS } \rho \\ \text{FROM} & log.read_{loop}((\ell), \rho^{2}) \text{ AS } loop(\rho, v) \\ \text{ORDER BY} & loop.v \text{ ASC} \end{array}$$

$$setRhoCtx(ctuid^{\mathbb{1}}, ctduid^{2}), \quad \mid s_i \mapsto \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \mid_{i=1..n}$$

$i^{\mathbb{1}} =$

FOR $v^{\mathbb{1}}$ IN $x^{\mathbb{1}}..y^{\mathbb{1}}$ BY $z^{\mathbb{1}}$
LOOP
$cTuid^{\mathbb{1}} := log.write_{log}((\ell), \rho^{\mathbb{1}})$
$s_1^{\mathbb{1}}, ..., s_n^{\mathbb{1}}$
END LOOP;

FOR $cTuid^{2}$ IN $query^{2}$
LOOP
$v^{2} := v^{2} \cup x^{2}$
$yv := yv \cup x^{2} \cup z^{2}$
$s_1^{2}, ..., s_n^{2}$
$v^{2} := v^{2} \cup y^{2}$
$yv := yv \cup y^{2}$
END LOOP;

FOR v IN $x..y$ BY z
LOOP
$s_1, ..., s_n$
END LOOP;

$\mapsto \langle i^{\mathbb{1}}, i^{2} \rangle$

4.3.6.15 FOR (over query results)

$$rowvar \Mapsto \langle rowvar^{\mathbb{1}}, rowvar^{\mathbb{2}} \rangle, \quad q \Mapsto \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle$$
$$add2YCtx(q^{\mathbb{2}}), \quad setRhoCtx(rowvar^{\mathbb{1}}.\rho, rowvar^{\mathbb{2}}.\rho), \quad \mid s_i \Mapsto \langle s_i^{\mathbb{1}}, s_i^{\mathbb{2}} \rangle \mid_{i=1..n}$$

$$
\begin{array}{lll}
 & \text{FOR} \quad rowvar^{\mathbb{1}} \quad \text{IN} \quad q^{\mathbb{1}} & \\
 & \text{LOOP} & \\
i^{\mathbb{1}} = & s_1^{\mathbb{1}} & \\
 & \vdots & \\
 & s_n^{\mathbb{1}} & \\
 & \text{END} \quad \text{LOOP;} &
\end{array}
$$

$$
\begin{array}{lll}
 & \text{FOR} \quad rowvar^{\mathbb{2}} \quad \text{IN} \quad q^{\mathbb{2}} & \\
 & \text{LOOP} & \\
i^{\mathbb{2}} = & s_1^{\mathbb{2}} & \\
 & \vdots & \\
 & s_n^{\mathbb{2}} & \\
 & \text{END} \quad \text{LOOP;} &
\end{array}
$$

$$
\begin{array}{lll}
 & \text{FOR} \quad rowvar \quad \text{IN} \quad q & \\
 & \text{LOOP} & \\
 & s_1 & \\
 & \vdots & \Mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle \\
 & s_n & \\
 & \text{END} \quad \text{LOOP;} &
\end{array}
$$

5 Diskussion

5.1 Fazit

Es wurde ein Programm entwickelt dass, aufbauend auf *SQLProv*, die Ableitungen von SQL-DML Anweisungen automatisiert durchführen kann. Völlig zufrieden bin ich mit diesem Ergebnis noch nicht. Die Implementation von `INSERT` hätte durch sinnvolleres Testen durchaus eine sinnvollere Implementation des Einfügens einer Liste von `VALUES` haben können. Die Tatsache dass die `UPDATE` Anweisung mehr loggt als final verarbeitet wird hätte auch früher auffallen können.

5.2 Ausblick

Eine Verbesserung der angesprochenen Probleme tut auf alle Fälle not wenn die implementierten Regeln sinnvoll angewendet werden sollen.

Darüber hinaus besteht durchaus die Möglichkeit die daraus entstandenen Transformationen noch voranzutreiben.

Durch eine Erweiterung des AST und den *LogParser* dahingehend dass auch Klauseln wie `ON CONFLICT`, `ONLY` und Ähnliches verfügbar sind wird die Transformation von DML Statements weiter an Tiefe bekommen.

Auch die Implementation der *PL/pgSQL* Statements und UDFs ist noch ausbaubar. Die Nutzung von *PL/pgSQL* Funktionen als tabellenwertige Funktionen würde hier vermutlich mehr Sinn machen als der bisherige Versuch den Rückgabewert einigermaßen zu erhalten.

Darüber hinaus gibt es auch hier noch die Möglichkeit weitere Funktionalität hinzuzufügen. Die Verwendung der Statements die die Cursor betreffen sollte noch Potential mit sich bringen ebenso müssten sich *Exceptions* und andere Meldungen durchaus sinnvoll mit verarbeiten lassen.

Literaturverzeichnis

[Cheney u. a. 2009] CHENEY, James ; CHITICARIU, Laura ; TAN, Wang-Chiew: Provenance in Databases: Why, How, and Where. In: *Found. Trends databases* 1 (2009), April, Nr. 4, 379–474. http://homepages.inf.ed.ac.uk/jcheney/publications/provdbsurvey.pdf. – ISSN 1931–7883

[Hirn 2017] HIRN, Denis: *Compilation of SQL into KL*. Version: 2017. https://db.inf.uni-tuebingen.de/attachments/thesis-hirn-2017.pdf

[Marlow 2010] MARLOW, Simon: *Haskell 2010 Language Report*. https://www.haskell.org/onlinereport/haskell2010/. Version: 2010

[Müller u. a. 2018] MÜLLER, Tobias ; DIETRICH, Benjamin ; GRUST, Torsten: You Say 'What', I Hear 'Where' and 'Why': (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. In: *CoRR* abs/1805.11517 (2018). http://arxiv.org/abs/1805.11517

[Paradzik 2018] PARADZIK, Gabriel: *Language-Level Provenance Analysis of SQL*. Version: 2018. https://db.inf.uni-tuebingen.de/attachments/paradzik-2018.pdf

[PostgreSQL 2018] POSTGRESQL: *Postgres Dokumentation*. https://www.postgresql.org/docs/10/static/index.html. Version: 2018

[Saake u. a. 2010] SAAKE, Gunter ; SATTLER, Kai-Uwe ; HEUER, Andreas: *Datenbanken - Konzepte und Sprachen, 4. Auflage*. MITP, 2010. – ISBN 978–3–8266–9057–0

Alle URLs wurden zuletzt am 29.10.2018 geprüft

Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Tübingen, 29.09.2018 Unterschrift