EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorarbeit Informatik

Compilation of SQL into KL

Denis Hirn

27. März 2017

Gutachter

Prof. Dr. Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Betreuer

Daniel O'Grady
Universität Tübingen

Tobias Müller
Universität Tübingen

Hirn, Denis:
Compilation of SQL into KL
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.12.2016 - 31.03.2017

Abstract

Um Data Provenance Analysen von Turing-vollständigen SQL-Queries durchführen zu können, ist es erforderlich die Queries in imperative Programme zu übersetzen. Im Rahmen dieser Bachelorarbeit wurde daher ein Compiler entwickelt, der diese Übersetzung ermöglicht und insbesondere Subqueries, Aggregierungen sowie rekursive Queries und Window Functions unterstützt.

Besonders wichtig war es zudem, dass typisierter Programmcode generiert wird, da dieser in einem weiteren, unabhängigen Schritt zu LLVM Code übersetzt werden soll.

Inhaltsverzeichnis

Abkürzungsverzeichnis vii

1 Einleitung 1
- 1.1 Motivation 1
  - 1.1.1 Zielsetzung 2
- 1.2 Aufbau der Arbeit 2

2 Grundlagen 3
- 2.1 Compilerbau 3
- 2.2 Das Eingabeformat 4
- 2.3 Kernel Language 6
  - 2.3.1 Funktionsaufrufe und Operatoren 6
  - 2.3.2 KL Grammatik 7
- 2.4 SQL Datentypen 7
- 2.5 Die Symboltabelle 8
- 2.6 Semantik relationaler Queries 9
- 2.7 Iterator Modell vs. Compilation 10
  - 2.7.1 Select-From-Where Block 10
  - 2.7.2 Spaltenreferenzen (ColRefs) 12
  - 2.7.3 Group By 13
  - 2.7.4 Aggregate 14
    - 2.7.4.1 Aggregat The 14
  - 2.7.5 Sortierverfahren für Order By 15
  - 2.7.6 Window Functions 15
  - 2.7.7 Common Table Expressions (CTEs) 16

3 Regelwerk 19
- 3.1 Definition Syntax 19
  - 3.1.1 Symbole 19
  - 3.1.2 KL-Type Notation 19
- 3.2 Definition Semantik 20
  - 3.2.1 Übersetzungsklassen 20
  - 3.2.2 Scoping Level 21
- 3.3 SQL Compiler Regelwerk 22
  - 3.3.1 Typenübersetzung trType 22
  - 3.3.2 Konstanten 23
  - 3.3.3 Block 23
  - 3.3.4 Select 24
  - 3.3.5 From 24

3.3.6 Having und Where . . . 25
3.3.7 Order By: Selection Sort . . . 26
3.3.8 Group By . . . 26
3.3.9 Aggregate Grundregel . . . 27
3.3.10 trAgg . . . 27
3.3.10.1 The . . . 27
3.3.10.2 Sum . . . 28
3.3.10.3 Count . . . 28
3.3.10.4 Average . . . 28
3.3.10.5 Array Agg . . . 29
3.3.10.6 Min / Max . . . 29
3.3.11 Function Calls . . . 30
3.3.11.1 Builtin Operators . . . 30
3.3.11.2 Array Union . . . 30
3.3.11.3 Recursive Union All . . . 30
3.3.11.4 Intersect . . . 31
3.3.11.5 Union . . . 31
3.3.11.6 Union All . . . 31
3.3.11.7 <>-Operator . . . 32
3.3.11.8 Degrees . . . 32
3.3.11.9 Abs . . . 32
3.3.11.10 greatest . . . 32
3.3.11.11 Generate Series . . . 33
3.3.12 trWindows . . . 33
3.3.12.1 trWindow . . . 34
3.3.12.2 Rank / Dense Rank . . . 36
3.3.13 Case When . . . 36
3.3.14 Sublink . . . 37

4 Queries 39
4.1 Konstante . . . 39
4.2 Filternde Query . . . 40
4.3 Aggregierende Query . . . 41
4.4 Deterministischer endlicher Automat . . . 43
4.5 Sliding Window . . . 48

5 Data Provenance 51
5.1 Aggregate . . . 51
5.2 Window Functions . . . 52

6 Diskussion und Ausblick 53
6.1 Einschränkungen der Implementierung . . . 54
6.1.1 Null Values . . . 54
6.1.2 Skalaroperationen . . . 54
6.1.3 Window Function Range peer groups . . . 54

6.2 Compiler als Lehrwerkzeug 55
6.3 Ausblick 56

Abbildungsverzeichnis 57

Literaturverzeichnis 57

ABKÜRZUNGSVERZEICHNIS

AST Abstract Syntax Tree
DP Data Provenance
DPA Data Provenance Analyse
CTE Common Table Expression
JSON JavaScript Object Notation
KL Kernel Language
SFW-Block SELECT-FROM-WHERE SQL Ausdruck
SQL Structured Query Language

1 EINLEITUNG

1.1 Motivation

Ein Forschungsbereich des Lehrstuhls für Datenbanksysteme an der Universität Tübingen beschäftigt sich mit der Ableitung und Analyse von *Data Provenance (DP)* im Kontext *lesender* relationaler SQL Queries. Die (*Why- und Where-*) DP beschreibt hierbei, wie und wieso die Daten des Inputs in das Ergebnis einfließen.

Das vom Lehrstuhl entwickelte Verfahren, ist dazu in der Lage die DP beliebiger Turing-vollständiger Queries herzuleiten. Unter anderem wird die Analyse von (korrelierten) Subqueries, Aggregierungen, rekursiven Queries und Window Functions unterstützt. Um die Provenance ableiten zu können, müssen Eingabequeries zunächst in eine imperative Programmiersprache übersetzt werden. Der generierte Programmcode wird anschließend mit einem neuen Verfahren, basierend auf bekannten Techniken aus dem Bereich der Programmanalyse, untersucht. Dadurch erhält man eine Berechnung der DP auf der Granularitätsebene einzelner Tabellenzellen. [Mül15, Mül16]

Der Übersetzungsvorgang einer Query kann einerseits von Hand durchgeführt werden, andererseits existiert bereits ein funktionsfähiger Compiler, der diese Aufgabe übernehmen kann. Der bestehende Ansatz ging aus einem experimentellen Forschungszweig hervor. Zu diesem Zeitpunkt war es nicht das primäre Ziel, einen *allgemeinen* Query Compiler zu entwerfen. Vielmehr wurden jeweils die aktuell für eine Übersetzung notwendigen Regeln implementiert oder angepasst und dabei nicht unbedingt Wert auf ein allgemeines und vollständiges Regelwerk für den Compiler gelegt.

Als mögliche Weiterentwicklung des Papers [Mül15], war eine Implementierung der Analysephase in LLVM angedacht, um die Performance zu verbessern. Dazu ist es erforderlich, dass Queries in *typisierten* Programmcode übersetzt werden. Diese Funktionalität ist im bestehenden Compiler ebenfalls nicht vorgesehen. Die Anforderung der Typisierung des Codes sowie der Allgemeinheit an die Übersetzung beeinflusst den Entwurf eines Compilers *maßgeblich*. Aus diesem Grund wäre es nicht Zielführend gewesen, den bestehenden Compiler zu überarbeiten.

1.1.1 Zielsetzung

Zielsetzung dieser Arbeit ist es, ein Konzept für einen möglichst *allgemeinen* Compiler für lesende relationale SQL-Queries zu entwickeln und zu implementieren. Dabei sollen neben Subqueries, Aggregierungen und rekursiven Queries insbesondere auch Window Functions unterstützt werden. Die Zielsprache des Compilers ist die vom Lehrstuhl entwickelte, *typisierte*, imperative Programmiersprache Kernel Language (KL). Als Implementierungssprache wurde Haskell gewählt, um die Kompatibilität zu bereits bestehenden Modulen zu gewährleisten.

1.2 Aufbau der Arbeit

Diese Arbeit gliedert sich, inklusive Einleitung, in insgesamt sechs Kapitel. Kapitel 2 beinhaltet zunächst eine allgemeine Einführung zum Thema Compilerbau. Weiter wird dort sowohl das Eingabe- als auch das Ausgabeformat definiert und die Semantik der wichtigsten SQL Konstrukte, sowie die zugehörige Implementierungsstrategie erläutert. In Kapitel 3 werden als nächstes Syntax und Übersetzungssemantik für eine *operationale Semantik* definiert. Damit werden im zweiten Teil des Kapitels die Übersetzungsregeln ausgedrückt.

Die Übersetzung einiger Queries wird dann in Kapitel 4 angeführt und erläutert. Im darauffolgenden Kapitel 5 werden schließlich DP-Analyseergebnisse von Queries, die mit dem Compiler übersetzt wurden gezeigt. Das letzte Kapitel 6 stellt ein abschließendes Fazit und einen Ausblick auf potentielle Weiterentwicklungen des Themas dar.

2 GRUNDLAGEN

In diesem Kapitel wird zunächst ein kurzer Überblick über den Compilerbau gegeben und der entwickelte Compiler in die bestehende Toolchain eingeordnet. Zudem werden sowohl das Eingabeformat als auch das Ausgabeformat des Compilers erläutert. Der Ansatz für die Übersetzung der SQL Semantik wird motiviert und für die verschiedenen SQL Ausdrücke definiert. Außerdem wird die zur Übersetzung wichtige Datenstruktur der Symboltabelle erläutert.

2.1 Compilerbau

Ein Compiler setzt sich in der Regel aus mehreren, verschiedenen *Phasen* zusammen, die jeweils eine spezifische Teilaufgabe erfüllen. Dabei wird primär zwischen dem *Front-* und *Backend* unterschieden. Das Frontend wird in diesem Zusammenhang auch als *Analyse-* und das Backend als *Synthesephase* bezeichnet. Im Verlauf der Analysephase wird der eingegebene Code – bzw. in diesem Fall die Query strukturiert, auf Fehler geprüft und in eine interne Abstract Syntax Tree (AST) Repräsentation übertragen. Die Übersetzung endet mit der Synthesephase, in der unter Verwendung des AST Programmcode in der Zielsprache erzeugt wird. Dieser Vorgang wird deshalb auch *Codegenerierung* genannt.

Es ist nicht erforderlich, dass ein Compiler als *monolithisches* Programm entworfen wird. Die verschiedenen Übersetzungsphasen können als eigenständige Programme implementiert werden, wobei jeweils eine neue Datenstruktur erzeugt wird. Die einzelnen Phasen werden in diesem Fall auch als *Pass* bezeichnet. Eine solche Aufteilung der verschiedenen Phasen in mehrere Passes war früher teilweise erforderlich, da der verfügbare Hauptspeicher nicht ausreichend war, um die gesamte Datenstruktur im Speicher zu halten. Heutzutage verfügen Computer über genug Hauptspeicher, um die Compilierung problemlos in einem Pass durchzuführen. Dennoch handelt es sich um keine veraltete Technik, da sie dazu beitragen kann, einzelne Teile des Compilers unkompliziert ersetzen zu können. Zudem kann eine solche Aufteilung die *Wartbarkeit* des Codes verbessern. [ALUS11, S. 4]

Im Kontext dieser Arbeit existiert das Frontend bereits in Form von PostgreSQL,

sowie dem nachfolgenden *Log-Parser*. Mehr Details zu dem verwendeten *Log-Parser* folgen im nächsten Abschnitt. Der Fokus dieser Arbeit liegt auf der Codegenerierung. Diese Phase wird aus diesem Grund im folgenden als (SQL-)Compiler bezeichnet und bildet ein eigenes Modul innerhalb der DP Toolchain. In Abbildung 2.1 ist eine Übersicht der verschiedenen Phasen der Übersetzung im Kontext der Data Provenance Analyse (DPA) zu sehen.

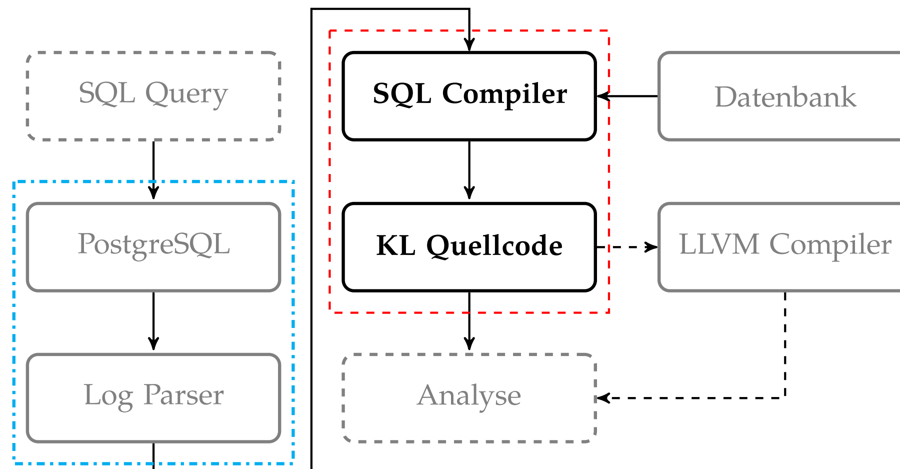

Abbildung 2.1: Übersicht der Compiler Struktur mit anschließender Analyse.

2.2 Das Eingabeformat

Das Eingabeformat des Compilers, ist die mit Typinformationen angereicherte JavaScript Object Notation (JSON) Repräsentation einer SQL-Query. Eine Konfiguration von PostgreSQL ermöglicht es, dass die Parse-Trees eingegebener Queries in einer Log Datei ausgegeben werden. Allerdings sind diese Daten in der Rohform nicht direkt zur Weiterverwendung geeignet und müssen zunächst durch den *Log Parser* aufbereitet werden. Der Log Parser ist ein Programm, das die PostgreSQL Log Datei ausliest und in eine nützliche Form überführt. Das Ergebnis dieser Transformation, ist der mit Typinformationen annotierte AST.

Die Verwendung des in PostgreSQL implementierten SQL Parsers bietet einige entscheidende Vorteile. Durch Expansion von beispielsweise `SELECT *`-Operationen, entsteht eine vereinfachte und typisierte Form der Eingabe-Query. Zudem werden fehlerhafte Queries bereits in diesem ersten Schritt der Verarbeitung abgewiesen, wodurch in nachfolgenden Verarbeitungsschritten weniger bis keine Fehlererkennung erforderlich ist.

Für die Implementierung der Übersetzung in Haskell wurden die *Inferenz-Regeln* basierend auf dem AST definiert. In Kapitel 3 werden trotz dessen möglichst *abstrahierte* Regeln für einzelne SQL Konstrukte angegeben, da diese nicht an die spezielle AST Struktur gebunden sein sollen. Die Gesamtheit dieser Übersetzungsregeln bildet das *Regelwerk* des Compilers, mit dem es möglich ist SQL-Queries in imperative Programme zu übersetzen. Damit das Regelwerk für die SQL Übersetzung funktionieren kann, ist es unvermeidlich auf die *Kompositionalität* der Regeln zu achten. Diese Eigenschaft ermöglicht es außerdem, dass der Compiler problemlos um bisher nicht übersetzbare Ausdrücke erweitert werden kann.

In Abbildung 2.15 auf Seite 17 sind die Haskell Datenstrukturen des AST dargestellt. Um den gewünschten SQL-Dialekt übersetzen zu können, muss für jede einzelne dieser Strukturen eine Übersetzungsregel definiert werden. Verletzt eine Regel die Kompositionalität oder ist semantisch fehlerhaft, kann kein korrektes Übersetzungsergebnis garantiert werden. Solche Fehler im Regelwerk können nicht während der Übersetzung einer Query erkannt werden. Schlimmstenfalls wird Code vom Compiler generiert, der entweder nicht funktioniert, oder ein falsches Ergebnis berechnet. Fehlt mindestens eine für die Übersetzung einer Query notwendige Regel, kann der Übersetzungsvorgang nicht abgeschlossen werden und endet mit einer Fehlermeldung, welche auf die nicht existierende Regel hinweist.

Die Fehlermeldungen sind besonders während der Aufbauphase des Regelwerks essentiell. Sie ermöglichen es systematisch per *try-and-error* Methode zu ermitteln, welche Regeln für eine Übersetzung der aktuellen Query fehlen. Anders gesagt kann der Übersetzungsvorgang einer beliebigen Query zunächst gestartet werden, durch die Fehlermeldungen können anschließend gezielt die zur Übersetzung fehlenden Regeln implementiert werden. Dadurch kann die Implementierung des Compilers mit einem kleinen Regelwerk begonnen werden, welches Stückweise erweitert wird. Diese Methode zur Entwicklung des Regelwerks erlaubt es zudem, den Fokus während der Implementierung jeweils gezielt auf einen kleinen Teil des Regelwerks zu beschränken.

Beispiel 2.2: Expansion einer Query durch den Postgres SQL-Parser

Unexpandiert	Expandiert

`SELECT *`
`  FROM foo;`
	`SELECT t1.a as A :: Int,`
	`       t1.b as B :: Text`
	`  FROM foo as t1;`

Beispiel 2.3: AST der Query `SELECT 42 as foo;`

```
Block
  (Select
    [Column (expr: ExprConst: ConstInt 42 (TypeAtom: "int4") "foo") ]
    ,agg: False, distinctOn: False, distinct: [])
  ,From []
  ,Where [...])
```

2.3 Kernel Language

KL ist eine minimale, typisierte *domänenspezifische Programmiersprache*, die für speziell die DPA von Queries entwickelt wurde. Zu den wichtigsten Einschränkungen zählt, dass nur *call-by-value* Funktionsaufrufe möglich sind, es keine globalen Variablen gibt und kein I/O möglich ist. Der Funktionsumfang ist durch die Inferenz-Regeln der DPA festgelegt. [Mül16]

Da kein I/O möglich ist, muss im Anschluss an die Übersetzung der Datenbankzustand der an der Query beteiligten Tabellen abgefragt und als KL Datenstrukturen an den Quellcode der Query angehängt werden.

2.3.1 Funktionsaufrufe und Operatoren

Der semantische Umfang von KL umfasst die wichtigsten mathematischen und logischen Operatoren. Zusätzlich sind String Operationen, Vergleiche und einige Type-Casts implementiert.

2.3.2 KL Grammatik

$$
\begin{aligned}
pp \;::=&\; \texttt{def}\; id(id : v, \ldots, id : v)\{ss;\ \texttt{return}\ e\}\ ;\ pp \\
\mid&\; ss \\[1em]
ss \;::=&\; s;\ ss \\
\mid&\; s \\[1em]
s \;::=&\; v = e \\
\mid&\; \texttt{if}\ e\ \texttt{then}\ ss_1\ \texttt{else}\ ss_2\ \texttt{fi} \\
\mid&\; \texttt{for}\ v\ \texttt{in}\ e^{list}\ \texttt{do}\ ss\ \texttt{od} \\
\mid&\; \texttt{while}\ e\ \texttt{do}\ ss\ \texttt{od} \\
\mid&\; \texttt{append}\ (v^{list},\ e_{elt}) \\
\mid&\; \texttt{skip} \\
\mid&\; \texttt{var}\ id : v \\[1em]
v \;::=&\; \langle variable\ name \rangle \\[1em]
e \;::=&\; id \\
\mid&\; e[e] \\
\mid&\; e\{e\} \\
\mid&\; \texttt{length}(e) \\
\mid&\; \texttt{keys}(e) \\
\mid&\; e\ \texttt{contains}\ e \\
\mid&\; e \circledast e \\
\mid&\; id(e, \ldots, e) \\
\mid&\; c \\[1em]
c \;::=&\; \texttt{None} \\
\mid&\; \langle integer \rangle \\
\mid&\; \langle float \rangle \\
\mid&\; \langle string \rangle \\
\mid&\; [e, \ldots, e] \\
\mid&\; \{c : e, \ldots, c : e\} \\
\mid&\; b \\
\mid&\; \texttt{True} \\
\mid&\; \texttt{False} \\[1em]
\circledast \;::=&\; + \mid - \mid * \mid / \mid \ldots \\[1em]
id \;::=&\; \langle variable/function\ name \rangle
\end{aligned}
$$

2.4 SQL Datentypen

Der SQL Standard beschreibt verschiedene Arten von Datentypen, die sich grob in die zwei Kategorien einordnen lassen. Es gibt *atomare* und *konstruierte* Datentypen. Zu den Atomen zählen unter anderem `Integers`, `Varchars`, `Floats`, während zu

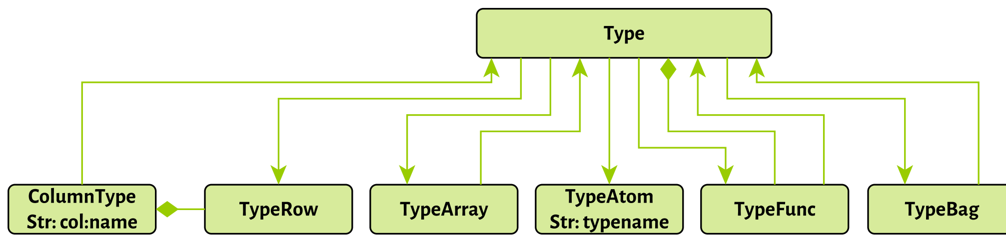

Abbildung 2.4: Für Typen relevanter Ausschnitt der JSON Repräsentation

den konstruierten Typen `Rows, Arrays, Multisets` zählen.

In Abbildung 2.4 sind alle für die Repräsentation der Typen im AST relevanten Haskell Datenstrukturen dargestellt. Die Strukturen der Typen bilden eine in sich abgeschlossene Gruppe. Das bedeutet in diesem Fall, dass SQL Typen statisch sind, also nicht zur Laufzeit inferiert werden und entweder atomar oder aus anderen Typen zusammengesetzt sind. Dies entspricht dem SQL Standard.

2.5 Die Symboltabelle

Es ist von zentraler Bedeutung *typisierten* Programmcode zu erzeugen, da dieser zur Geschwindigkeitsoptimierung der Analysephase in einem weiteren unabhängigen Compilationsschritt zu LLVM Code übersetzt werden soll. Viele der AST Strukturen sind bereits durch den Log Parser mit verwendbaren Typinformationen annotiert, allerdings sind nicht alle notwendigen Typen repräsentiert. Im Fall von nicht existierenden Typen, muss der Compiler eine *Typinferenz* durchführen. Dazu wird die *Symboltabelle* (auch *Type Environment* genannt) Γ benötigt. Dieses besteht aus einer Liste von assoziativen Datenfeldern, die jeweils einem Namen den zugehörigen Typen zuordnen. Ein Name kann dabei verschiedene Ursprünge haben. Unter anderem werden Namen durch Tabellen, Spalten, Row-Variablen und generierten KL Variablen gebunden.

$$\Gamma = [\{v \mapsto \tau_v\}, \ldots]$$

Es ist wichtig, dass bei der Übersetzung eines Row-Typs auch die Subtypen aufgenommen werden, da dieser Vorgang in folgenden Übersetzungsschritten nicht mehr trivial durchgeführt werden kann. Mit diesen Informationen können alle Typen, die nicht im AST repräsentiert sind, hergeleitet werden. Der Vorgang der Typinferenz wird nicht als eigenständiges Inferenz-Regelwerk angegeben, sondern ist implizit in den in Kapitel 3 angegebenen Übersetzungsregeln enthalten.

Beispiel 2.5: Rowtype einer Tabelle

```
Table "grp" (TypeRow [ ColumnType "a" (TypeAtom "int4")
                     , ColumnType "b" (TypeAtom "int4") ])
"A1"
```

In diesem Fall wird das Type Environment um vier Elemente erweitert:

$$\Gamma^{+} = [\{\mathtt{grp} \mapsto [<|\mathtt{a} : int, \mathtt{b} : int|>], \mathtt{A1} \mapsto <|\mathtt{a} : int, \mathtt{b} : int|>, \mathtt{a} : int, \mathtt{b} : int\}]$$

2.6 Semantik relationaler Queries

Um das Regelwerk zur Übersetzung zu entwerfen, ist es vorteilhaft die *bottom-up* und *top-down* Technik zu kombinieren. Dadurch ist es einerseits möglich ein Verständnis des gesamten Systems zu entwickeln, während andererseits der bereits teilweise funktionsfähige Compiler, die Implementierung neuer Übersetzungsregeln erleichtert. Dies ist möglich, da durch die Kompositionalität gewährleistet ist, dass die Semantik einer Regel unabhängig vom jeweiligen Kontext ist. Die einzelnen Regeln werden *bottom-up* entwickelt, da die Basisfälle dabei zum einen Aufschluss über die Repräsentation eines SQL Konstrukts als AST geben und zum anderen bereits die grundlegende *Semantik* definiert wird.

Zusätzlich zur strukturellen Analyse des AST, ist es unbedingt erforderlich, auf die *korrekte* Übersetzung der Semantik von SQL in das imperative Äquivalent zu achten. Für aussagekräftige Resultate der DP, muss die Auswertungsstrategie des Datenbanksystems logisch äquivalent durch die Übersetzungsregeln ausgedrückt werden. Um eine Übersetzungsregel aufstellen zu können, muss im Vorfeld klar sein *wie* ein bestimmtes SQL Konstrukt zu übersetzen ist.

Teilweise ist es möglich sich an der grundlegenden Theorie der relationalen Datenbank zu orientieren, bei der alle Operationen auf der *relationalen Algebra* basieren. [Cod90] Die relationale Algebra bietet allerdings beispielsweise für *rekursive* Queries keine Möglichkeit diese zu berechnen und ist daher hauptsächlich für die Grundoperationen eine sinnvolle Referenz. Eine andere Möglichkeit Informationen über die Query Auswertung zu erhalten, ist der `EXPLAIN` <query> Ausdruck, der vor beliebige Queries gestellt werden kann. PostgreSQL führt in diesem Fall die Query nicht aus, sondern liefert Informationen über den *Query Plan*.

2.7 Iterator Modell vs. Compilation

Das *Volcano-Iterator Modell* oder auch *Pipelined Query Execution Model*, ist die traditionelle Auswertungsstrategie von Datenbanksystemen für Queries und wird bis heute häufig verwendet. Jeder Operator erzeugt dabei einen *Stream* von Tupeln, die bei Bedarf generiert werden. Das Iterator Modell wurde zu einer Zeit entwickelt, als die Auswertungsgeschwindigkeit einer Query hauptsächlich durch die I/O Kosten limitiert war. Der Overhead, der durch die Vielzahl von *next* aufrufen entsteht, war zu vernachlässigen. Um Datenbanksysteme mit geringer Latenz basierend auf Compilation entwickeln zu können, waren die zu dieser Zeit vorhandenen Compiler nicht schnell genug. Durch den Fortschritt der Computer- sowie CPU- und Cachingtechnologien stößt das Modell allerdings mittlerweile an Optimierungsgrenzen. Eine Weiterentwicklung ist das Block-Volcano-Iterator Modell, bei dem bei jedem *next* Aufruf mehrere Tupel generiert werden.

Ein moderner Lösungsansatz versucht den Overhead, der mit dem Volcano-Iterator einher geht zu vermeiden und setzt zu diesem Zweck auf Query-Kompilierung, um die Auswertungsgeschwindigkeit zu erhöhen. Beispielsweise das Datenbanksystem *HyPer* verwendet diese Technik um Queries zu LLVM Code zu übersetzen. [Neu11, NL14]

2.7.1 Select-From-Where Block

Eine lesende SQL Query hat im allgemeinen eine Struktur wie sie in Listing 2.6 zu sehen ist. Die Auswertung einer solchen Query ist nach der Übersetzung in mehrere *Auswertungsphasen* gegliedert. Diese Phasen sind abhängig von der eingegebenen Query und werden im Übersetzten Code zu einem Funktionsrump zusammengefasst. Eine Übersicht der möglichen Konstellationen der Phasen ist in Abbildung 2.7 zu sehen. Im folgenden werden die wichtigsten SQL Fragmente erläutert und deren Übersetzungsstrategie motiviert. Für eine umfassendere Einführung und Vertiefung der Semantik ist [RG03] als Lektüre empfehlenswert.

1. From-Where: Der erste Schritt der Auswertung besteht darin, über den beteiligten Tabellen $t_1, \ldots, t_n$ das *kartesische Produkt* zu bilden. Eine häufig durchgeführte Optimierung, ist der sogenannte Predicate Pushdown. Dabei werden Filterkriterien so früh wie möglich angewendet, um Zwischenergebnisse klein zu halten. Diese Optimierung ist durch die Semantik der Übersetzungsregeln teilweise im Compiler implementiert und führt dazu, dass FROM und WHERE gemeinsam die erste Phase der Auswertung bilden.

2. Group By: Wird mindestens ein Group-By Kriterium angegeben, wird in einer zweiten Phase aus dem aktuellen Zwischenergebnis eine Hashtabelle aufgebaut. Nach einer Group-By Phase folgt zwingend eine Aggregierungsphase, in der die Hashtabelle wieder zu einer Listenform überführt wird.
3. Aggregate-Having: Aggregatsfunktionen werden speziell behandelt und bereits bevor die Übersetzung eines SFW-Blocks beginnt *substituiert*. Der genaue Mechanismus wird im Abschnitt 2.7.4 erläutert.

   Having bezieht sich auf Aggregationsfunktionen und kann nur in diesem Zusammenhang in einer Query vorkommen. Es ist deshalb sinnvoll das Filterkriterium direkt nach der Aggregierung anzuwenden.
4. Window-Functions: Window-Functions werden ähnlich wie Aggregationsfunktionen schon vor der SFW-Block Übersetzung substituiert um diese in einer eigenen Phase berechnen zu können.
5. Order By: Der letzte Schritt bevor das endgültige Ergebnis des SFW-Blocks generiert werden kann, besteht aus der Sortierung der Zwischenergebnistabelle. Die Details zu dem verwendeten Sortieralgorithmus sowie der zugehörigen Übersetzungsregel sind dem Abschnitt 2.7.5 bzw. 3.3.7 zu entnehmen.
6. Select-Offset-Limit: Offset und Limit nehmen direkten Einfluss auf die Generierung der Ergebnistabelle und werden aus diesem Grund in der letzten Phase mit Select ausgewertet.

```
SELECT c₁ AS name₁, ..., cₙ AS nameₙ
[ FROM t₁, ..., tₙ ]
[ WHERE p₁ ]
[ GROUP BY gb₁, ..., gbₘ ]
[ HAVING p₂ ]
[ ORDER BY o₁, ..., oₖ ]
[ OFFSET off ] [ LIMIT lim ]
```

Listing 2.6: Allgemeine lesende Query

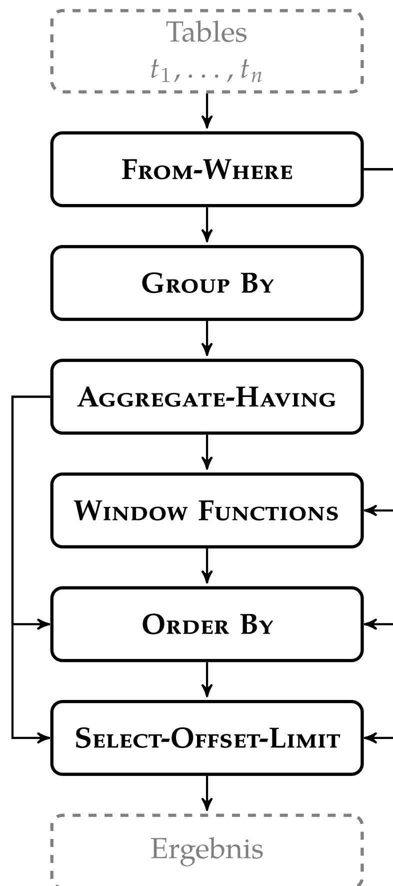

Abbildung 2.7: Organisation der verschiedenen Auswertungsphasen einer SFW-Query.

2.7.2 Spaltenreferenzen (ColRefs)

Eine Spaltenreferenz bzw. ein *ColRef* ist ein Zugriff auf eine Spalte einer Relation und erfolgt bei SQL durch den Ausdruck `<relation>.<spalte>`. Die Angabe der Relation ist dabei optional, sofern der Spaltenname *einzigartig* innerhalb der aktuellen Query ist. Während der Übersetzung werden allen Relationen, Row-Variablen sowie Spalten der Relationen neue eindeutige Namen zugeordnet. Diese Zuordnungen werden in der Symboltabelle verwaltet und können dadurch im Verlauf der Übersetzung aufgelöst werden.

2.7.3 Group By

Es ist grundsätzlich möglich Gruppierungen mit verschiedenen Algorithmen zu implementieren. Die einfachste Möglichkeit in KL ist Hashing. Dabei wird eine Relation mit einer Hashfunktion $h(i)$ zu einem *Dictionary* von Hashbuckets überführt. Die Hashfunktion $h(i)$ ist jeweils durch eine Menge von Spalten der Relation definiert. In Abbildung 2.8 ist der Vorgang schematisch dargestellt.

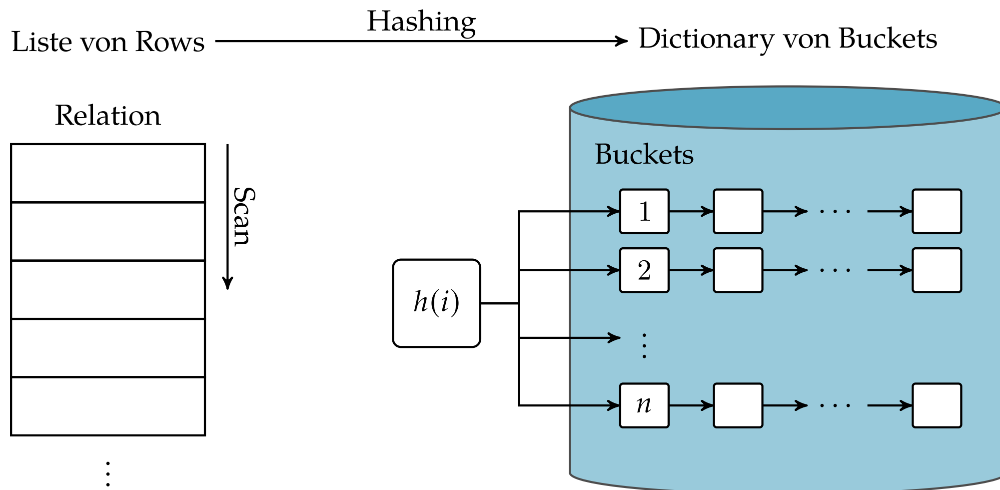

Abbildung 2.8: Schema der Gruppierung durch Hashing einer Relation

Beispiel 2.9: Hashing einer Relation anhand der ersten Spalte

```
input = [ (1,1), (1,2), (1,3)
        , (2,1), (2,2), (2,3)
        , (3,1), (3,2), (3,3) ]
output = { 1 : [(1,1), (1,2), (1,3)]
         , 2 : [(2,1), (2,2), (2,3)]
         , 3 : [(3,1), (3,2), (3,3)] }
```

2.7.4 Aggregate

Aggregationsfunktionen sind Berechnungen, die auf einer Gruppe von Tupeln durchgeführt werden. Wird kein *Group By* Kriterium angegeben, wird genau eine Gruppe gebildet, die alle Tupel der Tabelle beinhaltet. In diesem Fall besteht das Ergebnis einer Aggregationsfunktion aus genau einer Zeile.

Um die Übersetzung aggregierender Queries zu erleichtern, ist es möglich, die Query in eine äquivalente Form ohne Aggregationsfunktionen zu überführen. Dazu wird eine *Substitution* verwendet, die zunächst alle Aggregate aus der Query extrahiert und durch den Zugriff auf eine neu erzeugte Spalte ersetzt. Diese Form bietet erhebliche Vorteile, da es durch die Substitution möglich ist, die Aggregate in einer eigenständigen Phase zu Berechnen. Da die Gruppen für alle Aggregate dieselben sind, kann das Ergebnis Zeilenweise berechnet werden.

Nach der Gruppierung bzw. vor der Aggregierung bestehen die Daten nicht mehr aus einer Liste von Rows, sondern aus Hashbuckets. Problematisch würde dies, sofern in SELECT Duplikate eliminiert werden sollen, da dies ebenfalls über Hashing realisiert wird. Nach der Aggregierungsphase besteht das Zwischenergebnis wie vor der Gruppierung aus einer Liste von Rows, wodurch sich in folgenden Phasen keine weiteren Probleme ergeben.

Beispiel 2.10: Substitution vorkommender Aggregationsfunktionen

Input:

```
SELECT sum(a.foo)
    FROM bar as a
```

Output:

```
SELECT aggs.agg1
    FROM bar as a
```

$$\mathcal{A} = [\{agg_1 \mapsto sum(a.foo)\}]$$

2.7.4.1 Aggregat The

Bei aggregierenden Queries der Form `SELECT a, <agg>(...) FROM foo GROUP BY bar;`, ist die interne Repräsentation der Spalte 'a' durch die spezielle Aggregationsfunktion *The* dargestellt. 'The' beschreibt dabei den Zugriff auf Schlüsselspalten in der durch die Gruppierung generierte Datenstruktur.

2.7.5 Sortierverfahren für Order By

Die Sortierung in SQL wird über eine oder mehreren Spalten jeweils auf- oder absteigend durchgeführt. Es ist mit KL semantisch nicht möglich ein *allgemeines* Sortierverfahren zu definieren. Deshalb muss der Programmcode für das Prädikat, das diese Ordnung realisiert für *jeden* Datensatz *generiert* werden.

Der grundsätzlich verwendete Sortieralgorithmus basiert auf Selection-Sort und ist ein sehr einfaches, allerdings im Sinne der Komplexitätstheorie auch ineffizientes Verfahren in $O(n^2)$. Dennoch bietet sich der Algorithmus für die DPA an. Zudem ist das Verfahren günstig, da es sich um ein *stabiles* Sortierverfahren handelt.

2.7.6 Window Functions

Eine Window Function ermöglicht es Berechnungen über einer *Menge* von Tabellenzeilen durchzuführen. Window Functions sind mit Aggregaten verwand, mit dem wichtigen Unterschied, dass die einzelnen Zeilen des Inputs erhalten bleiben.

Um Window Functions ähnlich wie Aggregate in einer eigenen Phase berechnen zu können, wird auch hierfür eine Substitutionsoperation benötigt, die analog zu der in Abschnitt 2.7.4 erläuterten Methode funktioniert.

Im Vergleich zu Aggregaten, sind Window Functions mit einer ausdrucksstärkeren Semantik ausgestattet. Während bei Aggregaten stets eine Funktion über einer Gruppe bzw. intern einem Hashbucket berechnet wird, ist dies bei Window Functions nicht der Fall. Jede Window Function kann Berechnungen über andere Gruppen und Sortierungen durchführen. Damit sind beispielsweise Laufsummen berechenbar. Des Weiteren sind *Sliding Windows* möglich, bei denen jeweils die n Vorgänger und m Nachfolger eine Gruppe bilden. Window Functions werden aus diesem Grund Spaltenweise berechnet.

Beispiel 2.11: Substitution vorkommender Window Functions

Input:

```
SELECT sum(a.foo) over ()
    FROM bar as a
```

Output:

```
SELECT wins.win1
    FROM bar as a
```

$$\mathcal{W} = [\{win_1 \mapsto sum(a.foo)\ over()\}]$$

2.7.7 Common Table Expressions (CTEs)

Um SQL-Queries ähnlich wie in imperativen Programmiersprachen in kleine, leicht verständliche Gruppen zusammenzufassen, können *Common Table Expressions (CTEs)* verwendet werden. Der einfache Fall funktioniert dabei ähnlich wie eine *let* Bindung und ermöglicht es so, mehrere Queries zu definieren, wobei jeweils auf die vorangegangenen Queries zugegriffen werden kann. In Listing 2.12 ist die Struktur dargestellt.

```
WITH <q1> AS
   ( SELECT <...> ),
   <q2> AS (
   SELECT <...>
      FROM <q1>
      <...>
   )
)
SELECT <...>
```

Listing 2.12: Nicht rekursives With

Der interessantere Fall ist `WITH RECURSIVE`. Hierbei ist es möglich, dass sich eine CTE auf sich selbst bezieht und damit eine rekursive Berechnung ermöglicht. Die zugehörige Semantik ist in Listing 2.14 zu sehen.

Beispiel 2.13: Fakultät mit SQL

```
WITH RECURSIVE temp (n, fact) AS
(SELECT 0 as n, 1 as fact -- Initial Subquery
  UNION ALL
 SELECT n+1 as n, (n+1)*fact as fact FROM temp -- Rec Subquery
        WHERE n < 9)
SELECT * FROM temp;
```

```
r = <non-recursive SFW>
t = r
while len(t) > 0
    t = <recursive SFW>(t)
    r = r ∪ t
return <final SFW>(r)
```

Listing 2.14: Semantik rekursiver SQL Queries

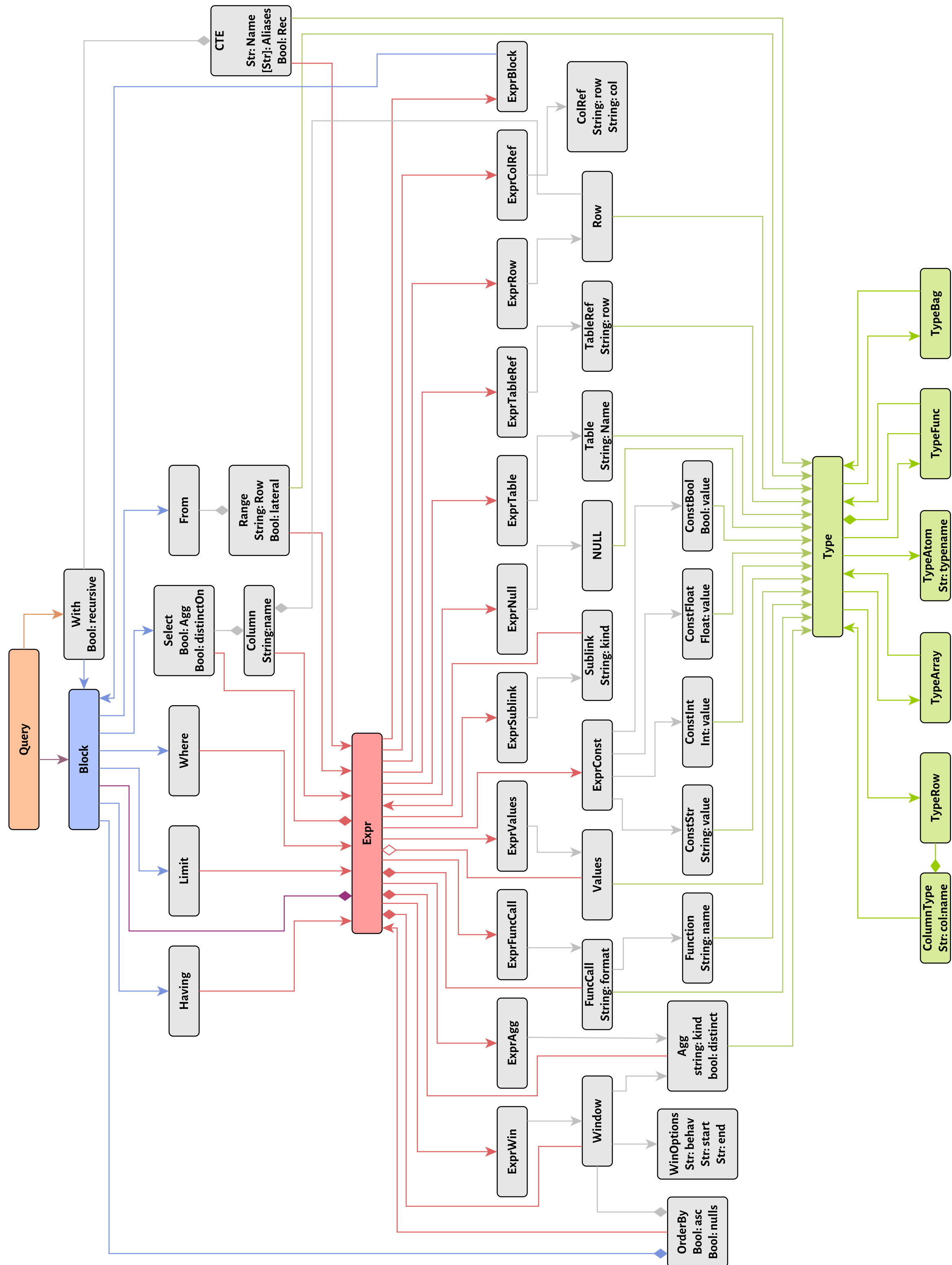

Abbildung 2.15: Vereinfachte Darstellung der JSON Repräsentation von SQL-Queries

3 REGELWERK

In diesem Kapitel wird zunächst die Syntax, die zur Repräsentation von Übersetzungsregeln verwendet wird definiert. Im zweiten Schritt wird eine *operationale Semantik* der Übersetzungsregeln in Form von *Inferenzregeln* angegeben. Die operationale Semantik beschreibt dabei die schrittweise Übersetzung einer Query.

3.1 Definition Syntax

3.1.1 Symbole

Um im Folgenden eine möglichst kompakte und übersichtliche Struktur für Übersetzungsregeln definieren zu können, werden zunächst Symbole eingeführt:

- κ Steht für ein Sourcecode Fragment in der Zielsprache KL
- ε KL Ausdruck (Zugriff auf Variablen, Konstanten etc.)
- τ Steht für einen KL Typen
- λ Funktion der Form `[Stmt] -> Maybe Expr -> [Stmt]`. Als Symbolnotation: $\kappa \rightarrow \varepsilon \rightarrow \kappa$. Die Parameter werden per Konvention $\lambda.\kappa$ und $\lambda.\varepsilon$ genannt
- Γ Type-Environment. Ist eine Liste von Namespaces $[\{v_q \mapsto v_k\}, \ldots]$

3.1.2 KL-Type Notation

Das KL-Typsystem setzt sich aus folgenden Komponenten zusammen:

```
-- All KL types
τ ::= v | fn
-- Value types
v ::= [v]               -- lists
   |  {v:v}             -- dictionary (unknown keys but single value type v)
   |  <|l:v,...,l:v|>   -- row (known key and associated value types)
   |  α                  -- atomic value
   |  null              -- null
fn ::= v ×...× v → v -- (First-order) function types
α ::= integer | float | bool | string -- Atomic types
```

3.2 Definition Semantik

3.2.1 Übersetzungsklassen

Die Übersetzungsregeln sind in mehrere semantische Klassen aufgeteilt, die sich jeweils in der Form des Ergebnis-Tupels unterscheiden. Die Metaklassen werden im folgenden Erläutert:

$\mapsto\!\!\triangleright$ Diese Regel entspricht dem Basisfall der Übersetzungsregeln, wird jedoch nur selten benötigt, da die Semantik vollständig durch die Lambda-Erweiterung abgedeckt wird. Das Resultat ist ein vier-Tupel $\langle \kappa, \varepsilon, \tau, \Gamma \rangle$.

$\overset{\lambda}{\mapsto}\!\!\triangleright$ Regeln dieser Klasse werden im folgenden als *Lambda-Regeln* bezeichnet. Das von der Regel erzeugte Resultat ist, ähnlich wie im Basisfall, ein vier-Tupel $\langle \lambda, \varepsilon, \tau, \Gamma \rangle$.

Die λ-Funktion ermöglicht es erzeugten Programmcode von mehreren Regeln ineinander zu *Falten*, statt zu *Konkatenieren*. Ein weiterer Anwendungsfall ist es, den Zugriff auf Iteratoren, die in einer vorherigen Übersetzungsregel erzeugt wurden zu ermöglichen. Ein Iterator entsteht bei Schleifenoperationen und entspricht einer Variable. Wird nun Programmcode in eine solche Schleife gefaltet, der Zugriff auf eben jenen Iterator benötigt, kann dieser trivial ermöglicht werden. Mit der Funktion `finalize` werden neutrale Argumente auf die λ-Funktion angewendet, um den resultierenden Quellcode zu erhalten. Die neutralen Argumente sind der leere Quellcode **`[]`** sowie **`Nothing`** als Expression.

$\overset{\text{trType}}{\longmapsto}\!\!\triangleright$ Übersetzt Typinformationen in die KL äquivalente. Das Ergebnis ist jeweils ein Tupel $\langle \tau_{KL}, \Gamma \rangle$ wobei die erste Komponente dem übersetzten Typen entspricht und die zweite Komponente der für die Typinferenz notwendige Kontexterweiterung wie in Abschnitt 2.5 definiert.

$\overset{\text{trColType}}{\longmapsto}\!\!\triangleright$ Dies ist ein Spezialfall der trType Regel und wird ausschließlich für die Typenübersetzung verwendet und benötigt.

Die hier angegebenen Klassen decken den Hauptteil der benötigten Semantik ab. Zusätzliche Regeln werden durch $\overset{tr\text{<}name\text{>}}{\longmapsto}\!\!\triangleright$ ausgedrückt und sind im Allgemeinen nur für die Übersetzung von genau einem Konstrukt definiert.

3.2.2 Scoping Level

In Abschnitt 2.7.1 wurde definiert, dass jeder Select-From-Where Block einer Query als eigener Funktionsrumpf gekapselt wird. Die Argumente einer solchen Funktion sind die beteiligten Input-Tabellen. Es gibt allerdings noch einen weiteren Fall der betrachtet werden muss. Bei korrelierten Unterabfragen muss zusätzlich eine einzelne Row als Parameter übergeben werden. Sowohl die Tabellen als auch Rows sind *freie Variablen*. Um diese zu erkennen ist es notwendig sogenannte *Scoping-Level* einzuführen. Das Scopinglevel wird für jeden Identifier mit den Typinformationen im Type Environment verwaltet.

Der Scope-Level ist zu Beginn der Übersetzung zunächst 0. Auf diesem Level werden die an der Query beteiligten Tabellen initialisiert. Die eigentliche Übersetzung beginnt auf Level 1 und wird bei Unterabfragen inkrementiert, siehe Listing 3.1. Bei einem Zugriff auf das Type Environment wird geprüft, ob das aktuelle Scope-Level $actLevel$ größer als das Scope-Level der Typinformation des Zugriffes ist. In diesem Fall handelt es sich um eine freie Variable.

$$\text{Level}(\Gamma(var)) < actLevel$$

```
-- Level 0
    SELECT first_name,last_name, salary  -- Level 1
    FROM employees
       WHERE salary >
          ( SELECT max(salary) -- Level 2
            FROM   employees
            WHERE  first_name = 'Alexander');
```

Listing 3.1: Scope-Level einer Query mit Subquery

3.3 SQL Compiler Regelwerk

In diesem Abschnitt wird das Regelwerk und damit der wichtigste Teil des Compilers definiert. Wie bereits erwähnt, kann mittels dieser Regeln SQL zu KL Übersetzt werden. Um die Regeln möglichst verständlich und soweit möglich unabhängig von der AST Repräsentation definieren zu können, wird der Input nicht als AST, sondern durch die entsprechenden SQL Fragmente dargestellt. Dabei wird implizit angenommen, dass beispielsweise die Typinformationen vorhanden sind, auch wenn sie nicht explizit im zugehörigen SQL Fragment aufgeführt sind.

Um die Inferenzregeln übersichtlich zu gestalten wird im Ergebnistupel teilweise das Schlüsselwort *code* angegeben. Dies entspricht dem Codefragment, das unter dem Tupel angeführt ist.

3.3.1 Typenübersetzung trType

$$\textsc{Integer}\ \frac{t \in \{\texttt{int4},\ \texttt{int8}\}}{\Gamma \vdash t \xmapsto{\text{trType}} \langle \texttt{TyInt}, \texttt{Nothing} \rangle} \qquad \textsc{Boolean}\ \frac{t = \texttt{bool}}{\Gamma \vdash t \xmapsto{\text{trType}} \langle \texttt{TyBool}, \texttt{Nothing} \rangle}$$

$$\textsc{Float}\ \frac{t \in \{\texttt{float},\ \texttt{numeric}\}}{\Gamma \vdash t \xmapsto{\text{trType}} \langle \texttt{TyFloat}, \texttt{Nothing} \rangle} \qquad \textsc{String}\ \frac{t \in \{\texttt{string},\ \texttt{text}\}}{\Gamma \vdash t \xmapsto{\text{trType}} \langle \texttt{TyStr}, \texttt{Nothing} \rangle}$$

$$\textsc{Row}\ \frac{\begin{array}{c} t_1 \in \mathbb{T}_{\text{SQL}} \xmapsto{\text{trColType}} \Gamma \vdash (name_1, \tau_1) \\ \ldots \\ t_n \in \mathbb{T}_{\text{SQL}} \xmapsto{\text{trColType}} \Gamma \vdash (name_n, \tau_n) \\ \Gamma^{+} = \left[(name_1, \tau_1), \ldots, (name_n, \tau_n)\right] \end{array}}{\Gamma \vdash [t_1, \ldots, t_n] \xmapsto{\text{trType}} \langle <|\ name_1 : \tau_1, \ldots, name_n : \tau_n\ |>, \Gamma^{+} \rangle}$$

$$\text{ColType}\ \frac{\text{t} \xmapsto{\text{trType}} [\tau, \text{-}]}{\Gamma \vdash (n, t) \xmapsto{\text{trColType}} \langle n, \tau \rangle}$$

$$\textsc{Function Type}\ \frac{\Gamma \vdash \text{args} \xmapsto{\text{trType}} [\tau_{\text{args}}, \text{-}] \qquad \Gamma \vdash \text{res} \xmapsto{\text{trType}} [\tau_{\text{res}}, \text{-}]}{\Gamma \vdash (\text{args}, \text{res}) \xmapsto{\text{trType}} \langle \texttt{TyFun}\ \tau_{\text{args}}\ \tau_{\text{res}}, \texttt{Nothing} \rangle}$$

3.3.2 Konstanten

Die Übersetzungsregeln für alle atomaren Konstanten [Int, Float, Bool, String, ...] sind identisch.

$$[\textsc{Const}_{Type}]\ \frac{value \in dom_{\mathsf{SQL}}(\mathtt{Type}) \quad value' \in dom_{\mathsf{KL}}(\mathtt{Type}) \quad \Gamma \vdash t \overset{\text{trType}}{\longmapsto} [\tau, \text{-}]}{\Gamma \vdash (\text{value, t}) \overset{\lambda}{\longmapsto} \langle [], value', \tau, \mathtt{Nothing} \rangle}$$

$$[\textsc{ColRef}]\ \frac{\Gamma((\mathtt{row},\mathtt{col})) = (\mathtt{row'},\mathtt{col'},\mathbf{typeof}(\mathtt{col'}))}{\Gamma \vdash \mathtt{row\ col} \overset{\lambda}{\longmapsto} (\mathtt{row'},\mathtt{col'},\mathbf{typeof}(\mathtt{col'}))}$$

3.3.3 Block

Ein *Block* ist die zu einem Select-From-Where Ausdruck äquivalente Datenstruktur im AST. Die in Abschnitt 2.7.4 und 2.7.6 definierten Substitutionsoperationen für Aggregate bzw. Window Functions werden hier verwendet.

$$\begin{array}{lll}
\Gamma & \vdash from\ [e_{f1}\ AS\ val_{f1},\ \ldots] & \overset{\lambda}{\longmapsto} \langle \kappa_f, \varepsilon_f, \tau_f, \Gamma_f \rangle \\
\Gamma;\Gamma_f & \vdash where\ e_w & \overset{\lambda}{\longmapsto} \langle \kappa_w, \varepsilon_w, \tau_w, [] \rangle \\
\Gamma;\Gamma_f & \vdash group\,by\ [e_{gb1},\ \ldots] & \overset{\lambda}{\longmapsto} \langle \kappa_{gb}, \varepsilon_{gb}, \tau_{gb}, [] \rangle \\
\Gamma;\Gamma_f;\mathcal{A} & \vdash (aggs, e_h) & \overset{\lambda}{\longmapsto} \langle \kappa_a, \varepsilon_a, \tau_a, \Gamma_a \rangle \\
\Gamma;\Gamma_f;\mathcal{A};\mathcal{W} & \vdash (wins, e_h) & \overset{trWins}{\longmapsto} \langle \kappa_w, \varepsilon_w, \tau_w, \Gamma_w \rangle \\
\Gamma;\Gamma_f;\mathcal{A};\mathcal{W} & \vdash ([e_{ob_1}\ asc_{ob_1}, \ldots], \tau_{a/w}) & \overset{\lambda}{\longmapsto} \langle \kappa_{ob}, \varepsilon_{ob}, \tau_{ob}, \Gamma_{ob} \rangle \\
\Gamma;\Gamma_f;\mathcal{A};\mathcal{W} & \vdash (select, e_o, e_l) & \overset{\lambda}{\longmapsto} \langle \kappa_s, \varepsilon_s, \tau_s, \Gamma_s \rangle \\
\hline
\end{array}$$

$\Gamma \vdash$

```
SELECT
   DISTINCT ON
      [e_d1, ...]
   e_s1 AS col_s1, ...
FROM
   [e_f1 AS val_f1, ...]
WHERE e_w
GROUP BY
   [ e_gb1, ... ]
HAVING e_h
ORDER BY
   [e_ob1 asc_ob1, ...]
OFFSET e_o
LIMIT e_l;
```

$\overset{\lambda}{\longmapsto} \langle [], \varepsilon_s, \tau_s, \Gamma + \Gamma_f \rangle;$

```
def sfwblock(val_f1 : τ_f1, ...) : τ {
  # From Code:
  κ_f (κ_w, ε_w)
  # GroupBy Code:
  κ_gb ([], ε_f)
  # Aggregation Code:
  κ_aggs ([], ε_gb)
  # Window Code:
  κ_wins ([], ε_aggs)
  # Sort Code:
  κ_ob ([], ε_wins)
  # Select Code:
  κ_s ([], ε_ob)
};
```

3.3.4 Select

$$\Gamma \vdash [e_{d1}, \ldots] \overset{groupBy}{\longmapsto\!\!\rhd} \langle \kappa_{gb}, \varepsilon_{gb}, \tau_{gb}, \Gamma_{gb} \rangle$$

$$\Gamma \vdash e_o \overset{\lambda}{\longmapsto\!\!\rhd} \langle \kappa_o, \varepsilon_o, \tau_o, \Gamma_o \rangle \qquad \Gamma \vdash e_l \overset{\lambda}{\longmapsto\!\!\rhd} \langle \kappa_l, \varepsilon_l, \tau_l, \Gamma_l \rangle$$

$$\Gamma \vdash \left[e_{s_k} \overset{\lambda}{\longmapsto\!\!\rhd} \langle \kappa_{e_{s_k}}, \varepsilon_{e_{s_k}}, \tau_{e_{s_k}}, \Gamma_{e_{s_k}} \rangle \right]_{k=1}^{n}$$

$$\Gamma \vdash \tau = \langle\!| \, col_{s_1} : \Gamma(e_{s_1}), \ldots, col_{s_n} : \Gamma(e_{s_n}) \, |\!\rangle \quad \varepsilon_s = result \quad \tau_s = [\tau]$$

$\Gamma \vdash$ `SELECT DISTINCT ON` $[e_{d1}, \ldots]$ e_{s1} `AS` $col_{s1}, \ldots$ `OFFSET` e_o `LIMIT` e_l; $\overset{\lambda}{\longmapsto\!\!\rhd} \langle$ code $, \varepsilon_s, \tau_s, \Gamma + \Gamma_f \rangle$

```
SELECT
  DISTINCT ON
   [e_d1, ...]
  e_s1 AS col_s1, ...
 OFFSET e_o
 LIMIT e_l;
```

```
κ_gb; κ_l; κ_o;
var row : τ;
var result : [τ];
result = [];
var offset : int;
var limit : int;
offset = ε_o;
limit = ε_l;
for select ε_gb do
  if (offset <= 0 and limit > 0) then
    κ_es1 ([], λ.ε); ...; κ_esn ([], λ.ε);
    row = <| col_s1 : ε_s1, ..., col_sn : ε_sn |>;
    append(result, row);
    limit = limit-1
  else
    offset = offset-1
  fi
od
```

3.3.5 From

$$\text{From}\ \frac{\varepsilon_f = from \quad \tau_f = [\langle\!| \, dummy : int \, |\!\rangle] \quad \Gamma_f = [\{from \mapsto (from, \tau_f)\}]}{\Gamma \vdash \texttt{FROM [ ]} \overset{\lambda}{\longmapsto\!\!\rhd} \langle \texttt{var from : [}\tau_f\texttt{]; from = [}\langle\!| \, dummy : 0 \, |\!\rangle\texttt{];}, \varepsilon_f, \tau_f, \Gamma_f \rangle}$$

$$\frac{\Gamma \vdash \left[e_{fi} \overset{\lambda}{\mapsto\!\!\rhd} \langle \kappa_{fi}, \varepsilon_{fi}, \tau_{fi}, \Gamma_{fi} \rangle \right]_{i=1}^{n} \quad \Gamma_f = \bigcup_{i=1}^{n} \Gamma_{fi} \cup \{from \mapsto (from, \tau_f)\} \qquad \kappa = \kappa_1; \ldots; \kappa_n \quad \varepsilon_f = from \quad \tau_f = <| row_1 : \tau_{f1}, \ldots, row_n : \tau_{fn} |>}{\Gamma \vdash \texttt{FROM } [e_{f1} \texttt{ AS } val_{f1}, \ldots] \overset{\lambda}{\mapsto\!\!\rhd} \langle \text{code}, \varepsilon_f, \tau_f, \Gamma_f \rangle}$$

where code is:

```
κ;
var from : [τ_f];
from = [];
for row_1 in ε_f1 do
  rowRange_1 = row_1;
  ...
  for row_n in ε_fn do
    rowRange_n = row_n;
    λ.κ;
    if λ.ε then
      var tempRow : τ_f;
      tempRow = <| row_1 : rowRange_1,...,row_n :
      ↪  rowRange_n |>;
      append(from, tempRow)
    else skip;
    fi
  od
  ...
od
```

3.3.6 Having und Where

$$\textsc{Having}\ \frac{\Gamma \vdash e_h \overset{\lambda}{\mapsto\!\!\rhd} \langle \kappa_h, \varepsilon_h, \tau_h, \Gamma_h \rangle \qquad \varepsilon_p = filter \qquad \tau_p = bool}{\Gamma \vdash \texttt{HAVING } e_h \overset{\lambda}{\mapsto\!\!\rhd} \langle \kappa_h\ ([], \lambda.\varepsilon);\ \texttt{var } filter : \tau_p;\ filter = \varepsilon_h, \varepsilon_p, \tau_p, [] \rangle}$$

$$\textsc{Where}\ \frac{\Gamma \vdash e_w \overset{\lambda}{\mapsto\!\!\rhd} \langle \kappa_w, \varepsilon_w, \tau_w, \Gamma_w \rangle \qquad \varepsilon_p = filter \qquad \tau_p = bool}{\Gamma \vdash \texttt{WHERE } e_w \overset{\lambda}{\mapsto\!\!\rhd} \langle \texttt{finalize } \kappa_w;\ \texttt{var } filter : \tau_p;\ filter = \varepsilon_w, \varepsilon_p, \tau_p, [] \rangle}$$

3.3.7 Order By: Selection Sort

$$op : bool \rightarrow Operator\ asc \mapsto \begin{cases} < & asc = true \\ > & asc = false \end{cases}$$

$$\left[e_{ob_k} \overset{\lambda}{\mapsto\!\!\triangleright} \langle \kappa_{ob_k}, \varepsilon_{ob_k}, \tau_{ob_k}, \Gamma_{ob_k} \rangle \right]_{k=1}^{n} \quad \lambda.\kappa_{sort} = \kappa_{ob_1}; \ldots; \kappa_{ob_n}$$

$$\lambda_{sort} = op_{asc_1}(e^{n}_{ob_1}, e^{min}_{ob_1})\ \texttt{or}\ (e^{n}_{ob_1} = e^{min}_{ob_1}\ \texttt{and}\ op_{asc_2}(e^{n}_{ob_2}, e^{min}_{ob_2}))\ \texttt{or}\ (\ldots)$$

$$\Gamma \vdash (\texttt{ORDER BY}\ [\ e_{ob_1}\ asc_1,\ \ldots\ ,\ e_{ob_n}\ asc_n\ ], \tau) \overset{\lambda}{\mapsto\!\!\triangleright} sort(\lambda_{sort}); \langle\ \texttt{var sorted}:[\tau];\ \texttt{sorted = sort}(data), \varepsilon_{ob}, \tau, [] \rangle$$

3.3.8 Group By

$$\Gamma \vdash \left[e_{gbi} \overset{\lambda}{\mapsto\!\!\triangleright} \langle \kappa_{gbi}, \varepsilon_{gbi}, \tau_{gbi}, _ \rangle \right]_{i=1}^{n}$$

$$\kappa = \texttt{concatFinalized}\ [\kappa_{gb1}, \ldots, \kappa_{gbn}] \quad \tau_{row} = <|\ k_1 : \Gamma(name(e_{gb1})),\ \ldots\ |>$$

$$\varepsilon_{gb} = groups \quad \tau_{list} = \Gamma(from) \quad \tau_{gb} = \{\tau_{row} : \tau_{list}\}$$

$\Gamma \vdash \texttt{GROUP BY}\ [\ e_{gb1}, \ldots, e_{gbn}\ ] \overset{\lambda}{\mapsto\!\!\triangleright} \langle$ code, $\varepsilon_{gb}, \tau_{gb}, [] \rangle$

```
var grpRow : τ_gb;
var groups : τ_gb;
groups = {} : τ_gb;
for f in λ.ε do
  κ; grpRow = <| k_1 : ε_gb1,... |>
  if not (groups contains grpRow) then
    update(groups, grpRow, [])
  else skip
  fi
  var grpVal : τ_list;
  grpVal = groups{grpRow};
  append(grpVal, f);
  update(groups, grpRow, grpVal)
od
```

3.3.9 Aggregate Grundregel

$$\frac{\begin{array}{lll}\Gamma \vdash & agg_1 \overset{\lambda}{\longmapsto} \langle \kappa_{agg_1}, \varepsilon_{agg_1}, \tau_{agg_1}, \Gamma_{agg_1} \rangle \\ \dots & \\ \Gamma \vdash & agg_n \overset{\lambda}{\longmapsto} \langle \kappa_{agg_n}, \varepsilon_{agg_n}, \tau_{agg_n}, \Gamma_{agg_n} \rangle \\ \tau_{row} & = & <| agg_1 : \tau_{agg_1}, \dots, agg_n : \tau_{agg_n} |> \\ \tau_{aggs} & = & <| aggs : \tau_{row} |> \\ \Gamma_{aggs} & = & [\{aggs \mapsto (aggRow, \tau_{aggs})\}, \Gamma_{agg_1}, \dots, \Gamma_{agg_n}] \\ \Gamma + \Gamma_{aggs} \vdash p_1 \overset{\lambda}{\longmapsto} & & \langle \kappa_h, \varepsilon_h, \tau_h, [] \rangle \end{array}}{\Gamma \vdash (\texttt{[}agg_1\texttt{, ... ,}agg_n\texttt{],} \; \texttt{HAVING} \; p_1\texttt{)} \overset{\lambda}{\longmapsto} \langle \text{code}, \varepsilon_{aggs}, [\tau_{aggs}], \Gamma_{aggs} \rangle}$$

```
var aggs : [τ_aggs];
aggs = [];
var aggRow : τ_aggs;
for g in keys(λ.ε) do
   κ_agg1 ([], λ.ε); ... ; κ_aggn ([], λ.ε);
   aggRow = <| agg1 : ε_agg1, ..., aggn : ε_aggn |>;
   κ_h;
   if ε_h then
      append(aggs, aggRow)
   else skip
   fi
od
```

3.3.10 trAgg

Die folgenden Regeln beschreiben die Übersetzung der einzelnen Aggregationsfunktionen.

3.3.10.1 The

$$\frac{\Gamma \vdash args \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle}{\Gamma \vdash \texttt{Agg the(args)} \overset{trAgg}{\longmapsto} \langle \text{code, Get the}, \tau_{agg}, [] \rangle}$$

```
var the : τ_args;
for x in λ.ε do
   κ_args;
   the = ε_args;
   λ.κ
od
```

3.3.10.2 Sum

$$\frac{\Gamma \vdash args \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle}{\Gamma \vdash \texttt{Agg sum(args)} \overset{trAgg}{\longmapsto} \langle \text{code, Get sum}, \tau_{agg}, [] \rangle}$$

```
var sum : τ_args;
sum = 0;
for x in λ.ε do
    κ_args;
    sum = sum + ε_args;
    λ.κ
od
```

3.3.10.3 Count

$$\frac{\Gamma \vdash args \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle}{\Gamma \vdash \texttt{Agg sum(args)} \overset{trAgg}{\longmapsto} \langle \text{code, Get cnt}, \tau_{agg}, [] \rangle}$$

```
var cnt : τ_args;
cnt = 0;
for x in λ.ε do
    κ_args;
    cnt = cnt + 1;
    λ.κ
od
```

3.3.10.4 Average

$$\frac{\Gamma \vdash args \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle}{\Gamma \vdash \texttt{Agg avg(args)} \overset{trAgg}{\longmapsto} \langle \text{code, Get avg}, \tau_{agg}, [] \rangle}$$

```
var avg : float;
avg = 0;
var sum : τ_args;
sum = 0;
var length : int;
length = 0;
for x in λ.ε do
    κ_args;
    sum = sum + ε_args;
    length = length+1;
    avg = float(sum/length);
    λ.κ
od
```

3.3.10.5 Array Agg

$$\frac{\Gamma \vdash args \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle}{\Gamma \vdash \texttt{Agg array_agg(args)} \overset{trAgg}{\longmapsto} \langle \text{code}, \text{Get cnt}, \tau_{agg}, [] \rangle}$$

```
var arrayAgg : τ_args;
arrayAgg = [];
for x in λ.ε do
    κ_args;
    append(arrayAgg, x);
    λ.κ
od
```

3.3.10.6 Min / Max

$$\frac{\Gamma \vdash args \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle \qquad op = \begin{cases} < & \$1 \text{ max} \\ > & \$1 \text{ min} \end{cases}}{\Gamma \vdash \texttt{Agg (min|max)(args)} \overset{trAgg}{\longmapsto} \langle \text{code}, \text{Get } \$1, \tau_{agg}, [] \rangle}$$

```
var $1 : τ_args;
var init : bool;
init = false;
$1 = 0;
for x in λ.ε do
    κ_args;
    if not(init) then
        $1 = ε_args;
        init = true
    else skip
    fi
    if ($1 op x)
        then $1 = x
        else skip
    fi;
    λ.κ
od
```

3.3.11 Function Calls

3.3.11.1 Builtin Operators

$$\frac{\Gamma \vdash a \overset{\lambda}{\mapsto} \langle \kappa_a, \varepsilon_a, \tau_a, \Gamma_a \rangle \quad \Gamma \vdash b \overset{\lambda}{\mapsto} \langle \kappa_b, \varepsilon_b, \tau_b, \Gamma_b \rangle \qquad lookup_{op}(\oplus) \mapsto \oplus_{KL} \quad T_\alpha \overset{trType}{\longmapsto} \tau_\alpha}{\Gamma \vdash \texttt{(a } \oplus \texttt{ b) :: } T_\alpha \overset{\lambda}{\mapsto} \langle\, \kappa_a\ (\lambda.\kappa, \lambda.\varepsilon);\ \kappa_b\ (\lambda.\kappa, \lambda.\varepsilon)\ , a \oplus_{KL} b, \tau_\alpha, [\Gamma_a, \Gamma_b] \rangle}$$

3.3.11.2 Array Union

$$\Gamma \vdash a \overset{\lambda}{\mapsto} \langle \kappa_a, \varepsilon_a, \tau_a, \Gamma_a \rangle \quad \Gamma \vdash b \overset{\lambda}{\mapsto} \langle \kappa_b, \varepsilon_b, \tau_b, \Gamma_b \rangle$$

$$(\alpha, \tau_\alpha) = \begin{cases} (a, \tau_a) & \tau_a \subset [] \\ (b, \tau_b) & \tau_b \subset [] \\ error & sonst \end{cases} \quad (\beta, \tau_\beta) = \begin{cases} (a, \tau_a) & \tau_a \not\subset [] \\ (b, \tau_b) & \tau_b \not\subset [] \end{cases}$$

$\Gamma \vdash$ `(a || b)` $\overset{\lambda}{\mapsto} \langle$ code, Get arrUn, $\tau_\alpha, [\Gamma_a, \Gamma_b] \rangle$

```
κa (λ.κ, λ.ε); κb (λ.κ, λ.ε);
var arrUn : τα;
var arrUn = α;
for it in β do
    append(arrUn, it)
od
```

3.3.11.3 Recursive Union All

$$\Gamma \vdash q_1 \overset{\lambda}{\mapsto} \langle \kappa_{q_1}, \varepsilon_{q_1}, \tau_{q_1}, \Gamma_{q_1} \rangle \quad \Gamma \vdash q_2 \overset{\lambda}{\mapsto} \langle \kappa_{q_2}, \varepsilon_{q_2}, \tau_{q_2}, \Gamma_{q_2} \rangle \quad T_{ret} \overset{trType}{\longmapsto} \tau_{ret}$$

$\Gamma \vdash$ **WITH RECURSIVE** (q_{init} **UNION ALL** q_{rec}) :: T_{ret} $\overset{\lambda}{\mapsto} \langle$ code, Get recUnionAllRes, $\tau_{ret}, [\Gamma_{q_1}, \Gamma_{q_2}] \rangle$

```
κq1 (λ.κ, λ.ε); κq2 (λ.κ, λ.ε);
var recUnionAll : [τret];
recUnionAll = [];
var baseRecQ : [τq1];
baseRecQ = εq1;
var recUnionAllRes : [τq2];
recUnionAllRes = [];
while (len(recUnionAll) > 0) do
    for row in recUnionAll do
        append(recUnionAllres, row)
    od;
    recUnionAll = εq2(recUnionAll, εq2)
od
```

3.3.11.4 Intersect

$$\frac{\Gamma \vdash q_1 \overset{\lambda}{\mapsto} \langle \kappa_{q_1}, \varepsilon_{q_1}, \tau_{q_1}, \Gamma_{q_1} \rangle \quad \Gamma \vdash q_2 \overset{\lambda}{\mapsto} \langle \kappa_{q_2}, \varepsilon_{q_2}, \tau_{q_2}, \Gamma_{q_2} \rangle \quad T_{ret} \overset{trType}{\longmapsto} \tau_{ret}}{\Gamma \vdash (q_1 \ \texttt{INTERSECT} \ q_2) :: T_{ret} \overset{\lambda}{\mapsto} \langle \text{code}, \text{Get union}, \tau_{ret}, [\Gamma_{q_1}, \Gamma_{q_2}] \rangle}$$

```
κ_q1 (λ.κ, λ.ε); κ_q2 (λ.κ, λ.ε);
var intersect : τ_q2;
intersect = ε_q2;
var union : τ_ret;
union = [];
for it in ε_q1 do
   if (intersect contains it) then
      append(union, it)
   else skip
   fi
od
```

3.3.11.5 Union

$$\frac{\Gamma \vdash q_1 \overset{\lambda}{\mapsto} \langle \kappa_{q_1}, \varepsilon_{q_1}, \tau_{q_1}, \Gamma_{q_1} \rangle \quad \Gamma \vdash q_2 \overset{\lambda}{\mapsto} \langle \kappa_{q_2}, \varepsilon_{q_2}, \tau_{q_2}, \Gamma_{q_2} \rangle \quad T_{ret} \overset{trType}{\longmapsto} \tau_{ret}}{\Gamma \vdash (q_1 \ \texttt{UNION} \ q_2) :: T_{ret} \overset{\lambda}{\mapsto} \langle \text{code}, \text{Get union}, \tau_{ret}, [\Gamma_{q_1}, \Gamma_{q_2}] \rangle}$$

```
κ_q1 (λ.κ, λ.ε); κ_q2 (λ.κ, λ.ε);
var union : τ_q2;
union = ε_q2;
for it in ε_q1 do
   if (union contains it) then skip
   else append(union, it)
   fi
od
```

3.3.11.6 Union All

$$\frac{\Gamma \vdash q_1 \overset{\lambda}{\mapsto} \langle \kappa_{q_1}, \varepsilon_{q_1}, \tau_{q_1}, \Gamma_{q_1} \rangle \quad \Gamma \vdash q_2 \overset{\lambda}{\mapsto} \langle \kappa_{q_2}, \varepsilon_{q_2}, \tau_{q_2}, \Gamma_{q_2} \rangle \quad T_{ret} \overset{trType}{\longmapsto} \tau_{ret}}{\Gamma \vdash (q_1 \ \texttt{UNION ALL} \ q_2) :: T_{ret} \overset{\lambda}{\mapsto} \langle \text{code}, \text{Get unionAll}, \tau_{ret}, [\Gamma_{q_1}, \Gamma_{q_2}] \rangle}$$

```
κ_q1 (λ.κ, λ.ε); κ_q2 (λ.κ, λ.ε);
var unionAll : τ_ret;
unionAll = [];
for it in ε_q1 do
   append(union, it)
od
for it in ε_q2 do
   append(union, it)
od
```

3.3.11.7 <>-Operator

$$\frac{\Gamma \vdash q_1 \overset{\lambda}{\longmapsto\!\!\triangleright} \langle \kappa_{q_1}, \varepsilon_{q_1}, \tau_{q_1}, \Gamma_{q_1} \rangle \quad \Gamma \vdash q_2 \overset{\lambda}{\longmapsto\!\!\triangleright} \langle \kappa_{q_2}, \varepsilon_{q_2}, \tau_{q_2}, \Gamma_{q_2} \rangle \quad T_{ret} \overset{trType}{\longmapsto\!\!\triangleright} \tau_{ret}}{\begin{array}{ll} \Gamma \vdash (q_1 \;\texttt{<>}\; q_2) \;\texttt{::}\; T_{ret} & \overset{\lambda}{\longmapsto\!\!\triangleright} \langle\, \text{code, Not}\,(\varepsilon_{q_1} == \varepsilon_{q_2}), \tau_{ret}, [\Gamma_{q_1}, \Gamma_{q_2}] \rangle \\ & \quad \kappa_{q_1}\ (\lambda.\kappa, \lambda.\varepsilon);\ \ \kappa_{q_2}\ (\lambda.\kappa, \lambda.\varepsilon); \end{array}}$$

3.3.11.8 Degrees

$$\frac{\Gamma \vdash e_1 \overset{\lambda}{\longmapsto\!\!\triangleright} \langle \kappa_{e_1}, \varepsilon_{e_1}, \tau_{e_1}, \Gamma_{e_1} \rangle}{\begin{array}{ll} \Gamma \vdash \texttt{degrees}(e_1) & \overset{\lambda}{\longmapsto\!\!\triangleright} \langle\, \text{code}, \frac{180}{\pi} \cdot \varepsilon_{e_1}, \text{Float}, [] \rangle \\ & \quad \kappa_{q_1}\ (\lambda.\kappa, \lambda.\varepsilon);\ \ \kappa_{q_2}\ (\lambda.\kappa, \lambda.\varepsilon); \end{array}}$$

3.3.11.9 Abs

$$\frac{\Gamma \vdash e_1 \overset{\lambda}{\longmapsto\!\!\triangleright} \langle \kappa_{q_1}, \varepsilon_{q_1}, \tau_{q_1}, \Gamma_{q_1} \rangle}{\begin{array}{ll} \Gamma \vdash \mathbf{abs}(e_1) & \overset{\lambda}{\longmapsto\!\!\triangleright} \langle\, \text{code, Get abs}, \tau_{e_1}, \Gamma_{e_1} \rangle \\ & \quad \kappa_{e_1}\ (\lambda.\kappa, \lambda.\varepsilon); \\ & \quad \mathbf{var}\ \texttt{abs : } \tau_{e_1}; \\ & \quad \texttt{abs = } \varepsilon_{e_1}; \\ & \quad \mathbf{if}\ (\varepsilon_{e_1} \texttt{ < 0}) \\ & \quad\quad \mathbf{then}\ \texttt{abs = } 0 - \varepsilon_{e_1} \\ & \quad\quad \mathbf{else}\ \texttt{skip} \\ & \quad \mathbf{fi} \end{array}}$$

3.3.11.10 greatest

$$\frac{\begin{array}{l} \Gamma \vdash e_1 \overset{\lambda}{\longmapsto\!\!\triangleright} \langle \kappa_{e_1}, \varepsilon_{e_1}, \tau_{e_1}, \Gamma_{e_1} \rangle \\ \dots \\ \Gamma \vdash e_n \overset{\lambda}{\longmapsto\!\!\triangleright} \langle \kappa_{e_n}, \varepsilon_{e_n}, \tau_{e_n}, \Gamma_{e_n} \rangle \end{array}}{\begin{array}{ll} \Gamma \vdash \texttt{greatest}(e_1, \dots, e_n) & \overset{\lambda}{\longmapsto\!\!\triangleright} \langle\, \text{code, Get greatest}, \tau_{e_1}, [\Gamma_{e_1}, \dots, \Gamma_{e_n}] \rangle \\ & \quad \kappa_{e_1}\ (\lambda.\kappa, \lambda.\varepsilon);\ \dots,\ \kappa_{e_n}\ (\lambda.\kappa, \lambda.\varepsilon); \\ & \quad \mathbf{var}\ \texttt{greatestList : } [\tau_{e_1}]; \\ & \quad \texttt{greatestList = } [\varepsilon_{e_1}, \dots, \varepsilon_{e_n}]; \\ & \quad \mathbf{var}\ \texttt{greatest : } \tau_{e_1}; \\ & \quad \texttt{greatest = } \varepsilon_{e_1} \\ & \quad \mathbf{for}\ \texttt{it}\ \mathbf{in}\ \texttt{greatestList}\ \mathbf{do} \\ & \quad\quad \mathbf{if}\ \texttt{(it > maxE)}\ \mathbf{then} \\ & \quad\quad\quad \texttt{maxE = it} \\ & \quad\quad \mathbf{else}\ \texttt{skip} \\ & \quad\quad \mathbf{fi} \\ & \quad \mathbf{od} \end{array}}$$

3.3.11.11 Generate Series

$$\frac{\Gamma \vdash e_{start} \overset{\lambda}{\longmapsto} \langle \kappa_{e_{start}}, \varepsilon_{e_{start}}, \tau_{e_{start}}, \Gamma_{e_{start}} \rangle \quad \Gamma \vdash e_{end} \overset{\lambda}{\longmapsto} \langle \kappa_{e_{end}}, \varepsilon_{e_{end}}, \tau_{e_{end}}, \Gamma_{e_{end}} \rangle \quad \Gamma \vdash e_{step} \overset{\lambda}{\longmapsto} \langle \kappa_{e_{step}}, \varepsilon_{e_{step}}, \tau_{e_{step}}, \Gamma_{e_{step}} \rangle}{\Gamma \vdash \texttt{generate_series}(e_{start}, e_{end}, e_{step}) \overset{\lambda}{\longmapsto} \langle \text{code, Get series}, \tau_{e_{start}}, [\Gamma_{e_{start}}, \Gamma_{e_{end}}, \Gamma_{e_{step}}] \rangle}$$

```
κ_{e_start} (λ.κ, λ.ε);  κ_{e_end} (λ.κ, λ.ε);
κ_{e_step} (λ.κ, λ.ε);
var start : τ_{e_start};
start = ε_{e_start};
var step : τ_{e_step};
step = ε_{e_step};
var series : [τ_{e_start}];
series = [];
var asc : bool;
if (start < end)
   then asc = true
   else asc = false
fi
while ((asc and start < end) or
↪  (not(asc) and start > end)) do
   append(list, start);
   start = start + step
od
```

3.3.12 trWindows

$$\frac{\Gamma \vdash win \overset{\lambda}{\longmapsto} \langle \kappa_{gb}, \kappa_{win}, \varepsilon_f, \varepsilon_{gb}, \tau_{gb}, \tau_f, \Gamma_f \rangle \quad \tau_{res} = <\mid from : \Gamma(from), wins :<\mid win : \tau_{win} \mid> + \tau_{row} \mid>}{\Gamma \vdash \texttt{([win:xs], } \tau\texttt{, } \tau_{row}\texttt{)} \overset{trWindow}{\longmapsto} \langle \text{code}, \varepsilon_f, \tau_{wins}, \Gamma_f \rangle}$$

```
var wins : [τ_res];
wins = [];
var winRows : τ_res;
for g in keys(ε_gb) do
   var group : τ_gb;
   group = ε_gb{g};
   κ_win
      (winRow = <| from : g <| from |>
      ↪  , wins :<| win : ε_f |> |>
      append(wins, winRow), group)
od
```

3.3.12.1 trWindow

$$\frac{\Gamma \vdash pb_{cols} \overset{\lambda}{\longmapsto} \langle \kappa_{gb}, \varepsilon_{gb}, \tau_{gb}, \Gamma_{gb} \rangle \quad \Gamma \vdash f_{win} \overset{\lambda}{\longmapsto} \langle \kappa_{f_{init}}, \kappa_{f_{logic}}, \varepsilon_f, \tau_f, \Gamma_f \rangle}{\Gamma \vdash f_{win} \text{ over (PARTITION BY } pb_{cols} \text{ RANGE (UNBOUNDED PRECEDING) (UNBOUNDED FOLLOWING)} \overset{trWindow}{\longmapsto} \langle \kappa_{gb}, \text{code}, \varepsilon_f, \varepsilon_{gb}, \tau_{gb}, \tau_f, \Gamma_f \rangle}$$

```
Γ ⊢ f_win
    over (PARTITION BY
            pb_cols
      RANGE
      (UNBOUNDED PRECEDING)
      (UNBOUNDED FOLLOWING)
```

code:

```
κ_finit;
for x in group do
    κ_flogic
od;
for tuple in group do
    λ.κ
od
```

$$\frac{\Gamma \vdash pb_{cols} \overset{\lambda}{\longmapsto} \langle \kappa_{gb}, \varepsilon_{gb}, \tau_{gb}, \Gamma_{gb} \rangle \quad \begin{array}{c} \Gamma \vdash ob_{cols} \overset{\lambda}{\longmapsto} \langle \kappa_{ob}, \varepsilon_{ob}, \tau_{ob}, [] \rangle \\ \Gamma \vdash f_{win} \overset{\lambda}{\longmapsto} \langle \kappa_{f_{init}}, \kappa_{f_{logic}}, \varepsilon_f, \tau_f, \Gamma_f \rangle \end{array}}{\Gamma \vdash f_{win} \text{ over (PARTITION BY } pb_{cols}, \text{ ORDER BY } ob_{cols} \text{ RANGE (UNBOUNDED PRECEDING) (UNBOUNDED FOLLOWING)} \overset{trWindow}{\longmapsto} \langle \kappa_{gb}, \text{code}, \varepsilon_f, \varepsilon_{gb}, \tau_{gb}, \tau_f, \Gamma_f \rangle}$$

```
Γ ⊢ f_win
    over (PARTITION BY
            pb_cols,
         ORDER BY ob_cols
      RANGE
      (UNBOUNDED PRECEDING)
      (UNBOUNDED FOLLOWING)
```

code:

```
κ_finit;
for x in λ.ε do
    κ_flogic;
    λ.κ
od
```

$$\Gamma \vdash ob_{cols} \overset{\lambda}{\longmapsto} \langle \kappa_{ob}, \varepsilon_{ob}, \tau_{ob}, [] \rangle \quad \Gamma \vdash pb_{cols} \overset{\lambda}{\longmapsto} \langle \kappa_{gb}, \varepsilon_{gb}, \tau_{gb}, \Gamma_{gb} \rangle$$

$$\Gamma \vdash f_{win} \overset{\lambda}{\longmapsto} \langle \kappa_{f_{init}}, \kappa_{f_{logic}}, \varepsilon_f, \tau_f, \Gamma_f \rangle \quad \Gamma \vdash e_{pre} \overset{\lambda}{\longmapsto} \langle \kappa_{e_{pre}}, \varepsilon_{e_{pre}}, \tau_{e_{pre}}, [] \rangle$$

$$\Gamma \vdash e_{fol} \overset{\lambda}{\longmapsto} \langle \kappa_{e_{fol}}, \varepsilon_{e_{fol}}, \tau_{e_{fol}}, [] \rangle$$

$$\varepsilon_{preExpr} = \begin{cases} 0 & \$1 == \text{UNBOUNDED PRECEDING} \\ current - \varepsilon_{e_{pre}} & \$1 == \text{VALUE PRECEDING} \end{cases}$$

$$\varepsilon_{folExpr} = \begin{cases} 0 & \$2 == \text{UNBOUNDED FOLLOWING} \\ current - \varepsilon_{e_{fol}} & \$2 == \text{VALUE FOLLOWING} \end{cases}$$

$$\varepsilon_{condSt} = \begin{cases} True & \$1 == \text{UNBOUNDED PRECEDING} \\ counter \geq preceding & \$1 == \text{VALUE PRECEDING} \end{cases}$$

$$\varepsilon_{condEn} = \begin{cases} True & \$2 == \text{UNBOUNDED FOLLOWING} \\ counter \leq following & \$2 == \text{VALUE FOLLOWING} \end{cases}$$

$\Gamma \vdash f_{win}$ $\overset{trWindow}{\longmapsto}$ $\langle \kappa_{gb}, \text{code}, \varepsilon_f, \varepsilon_{gb}, \tau_{gb}, \tau_f, \Gamma_f \rangle$

```
over (PARTITION BY
        pb_cols,
      ORDER BY ob_cols
   ROWS
   (UNBOUNDED PRECEDING|
    VALUE PRECEDING e_pre)
   (UNBOUNDED FOLLOWING|
    VALUE FOLLOWING e_fol))
```

```
var preceding : int;
var current : int;
current = 0;
var following : int;
for tuple in λ.ε do
   var counter : int;
   counter = 0;
   current = current+1;
   preceding = ε_preExpr;
   following = ε_folExpr;
   κ_finit;
   for x in λ.ε do
      counter = counter+1;
      if (ε_condSt and ε_condEn)
         then κ_flogic
      else skip
   od;
   λ.κ
od
```

3.3.12.2 Rank / Dense Rank

$$\frac{\begin{array}{lll} \Gamma \vdash args & & \overset{\lambda}{\longmapsto} \langle \kappa_{args}, \varepsilon_{args}, \tau_{args}, \Gamma_{args} \rangle \\ \kappa_d & = & \begin{cases} skip & \$1 == \text{dense_rank} \\ rank = rank + 1 & \text{sonst} \end{cases} \end{array}}{\Gamma \vdash \texttt{WinAgg (rank|dense_rank)(args)} \overset{trAgg}{\longmapsto} \langle \text{code, Get } \$1, \text{int}, [] \rangle}$$

```
Var $1 : int;
$1 = 0;
var rankB : int;
rankB = 0;
var ranked : [τ_from];
ranked = [];
for x in λ.ε do
    κ_args;
    if (ranked contains x)
       then κ_d
    else rank = rank + 1;
         rankB = rank;
         append(ranked, x)
    fi;
    λ.κ
od
```

3.3.13 Case When

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 & \overset{\lambda}{\longmapsto} \langle \kappa_{e_1}, \varepsilon_{e_1}, \tau_{e_1}, \Gamma_{e_1} \rangle \\ \dots & \\ \Gamma \vdash e_n & \overset{\lambda}{\longmapsto} \langle \kappa_{e_n}, \varepsilon_{e_n}, \tau_{e_n}, \Gamma_{e_n} \rangle \\ \Gamma \vdash o_1 & \overset{\lambda}{\longmapsto} \langle \kappa_{o_1}, \varepsilon_{o_1}, \tau_{o_1}, \Gamma_{o_1} \rangle \\ \dots & \\ \Gamma \vdash o_n & \overset{\lambda}{\longmapsto} \langle \kappa_{o_n}, \varepsilon_{o_n}, \tau_{o_n}, \Gamma_{o_n} \rangle \\ \Gamma \vdash o_{else} & \overset{\lambda}{\longmapsto} \langle \kappa_{o_{else}}, \varepsilon_{o_{else}}, \tau_{o_{else}}, \Gamma_{o_{else}} \rangle \end{array}}{\Gamma \vdash \texttt{CASE WHEN } e_1 \texttt{ THEN } o_1 \dots \texttt{ WHEN } e_n \texttt{ THEN } o_n \texttt{ ELSE } o_{else} \overset{\lambda}{\longmapsto} \langle \text{code, Get case}, \tau_{o_1}, [] \rangle}$$

```
κ_e1; var case : τ_o1;
if ε_e1 then
   case = ε_o1
   else if
      ...
         else case = ε_o_else
      fi
   ...
fi
```

3.3.14 Sublink

Ein Sublink entsteht immer dann, wenn in einer Query eine Subquery auftritt. In diesem Fall wird das Scopinglevel erhöht, um die freien Variablen erkennen zu können.

$$\frac{\Gamma, Scopelevel + 1 \vdash q \quad \overset{\lambda}{\mapsto} \langle \kappa_q, \varepsilon_q, \tau_q, \Gamma_q \rangle}{\Gamma \vdash \texttt{SUBQUERY}\ q \quad \overset{\lambda}{\mapsto} \langle \kappa_q, \varepsilon_q, \tau_q, \Gamma_q \rangle}$$

```
def sort(unsorted:[τ]):[τ]
{
  var sorted:[τ];
  sorted = []:[τ];
  var dict:{int:τ};
  dict = {}:{int:τ};
  # Convert list to dictionary: ;
  var it:int;
  it = 0;
  for e in unsorted do
     update(dict, it, e);
     it = (it + 1)
  od;
  it = 0;
  var l:int;
  l = len(unsorted);
  while (it < l) do
     var minElem:τ;
     minElem = dict{it};
     var k:int;
     k = (it + 1);
     var minIndex:int;
     minIndex = it;
     while (k < l) do
        λ.κ_sort;
        if (λ_sort)
           then minElem = dict{k};
                minIndex = k
           else # surplus statements, improved provenance;
                minIndex = minIndex;
                update(dict, minIndex, dict{minIndex})
        fi;
        k = (k + 1)
     od;
     append(sorted, minElem);
     if (minIndex > it) then update(dict, minIndex, dict{it})
        else skip
     fi;
     it = (it + 1)
  od;
  return sorted
};
```

Listing 3.2: Selection-Sort Algorithmus Prototyp

4 QUERIES

Im folgenden werden die Übersetzungen von einigen Queries, die teilweise direkt aus der aktuellen Forschung des Lehrstuhls entspringen präsentiert. Da die Quellcodes teilweise *sehr* viele Zeilen umfassen werden stellenweise *gekürzte* Versionen angegeben bzw. nur einige interessante Ausschnitte.

4.1 Konstante

Von allen übersetzbaren Queries ist die *Abfrage* einer Konstante die einfachste. In Listing 4.1 ist die Query angegeben. Da das Ergebnis der Übersetzung nur wenige Zeilen Code umfasst kann dieser ungekürzt angegeben werden.

```
1 SELECT 42 as foo;
```

Listing 4.1: Abfrage einer Konstante

```
1  def sfwblock6():[<|"foo":int|>]
2  { # From Code: ;
3    var from:[<|"dummy":int|>];
4    from = [<|"dummy":0|>]:[<|"dummy":int|>];
5    # Select Code: ;
6    var row3:<|"foo":int|>;
7    var result4:[<|"foo":int|>];
8    result4 = []:[<|"foo":int|>];
9    for select in from do
10      row3 = <|"foo":42|>;
11      append(result4, row3)
12   od;
13   return result4
14 };
15 var res7:[<|"foo":int|>];
16 res7 = sfwblock6();
17 print(res7)
```

Listing 4.2: Übersetzung von Listing 4.1

4.2 Filternde Query

Die in Listing 4.3 angegebene Query besteht nach der Übersetzung aus den zwei Auswertungsphasen *From-Where* und *Select-Limit-Offset*. Im Vergleich zu der in Listing 4.2 angegebenen Übersetzung, findet in der From-Where-Phase ein *Sequentieller-Scan* über die Eingabetabelle mit Filterung statt. Die Übersetzung des *WHERE-Prädikats* befindet sich in Listing 4.4, Zeile 10.

```
SELECT a, b
FROM   grp
WHERE  b > 5;
```

Listing 4.3: Filternde Query

```
1  def sfwblock9(table0:[<|"a":int, "b":int|>]):[<|"a":int, "b":int|>]
2  { # From Code: ;
3    var rowRange1:<|"a":int, "b":int|>;
4    var from:[<|"A1":<|"a":int, "b":int|>|>];
5    from = []:[<|"A1":<|"a":int, "b":int|>|>];
6    for A1 in table0 do
7       rowRange1 = A1;
8       # Where Code begin;
9       var filter3:bool;
10      filter3 = (rowRange1<|"b"|> > 5);
11      # Where Code end;
12      if filter3
13         then var tempRow2:<|"A1":<|"a":int, "b":int|>|>;
14              tempRow2 = <|"A1":rowRange1|>;
15              append(from, tempRow2)
16         else skip
17      fi
18   od;
19   # Select Code: ;
20   var row6:<|"a":int, "b":int|>;
21   var result7:[<|"a":int, "b":int|>];
22   result7 = []:[<|"a":int, "b":int|>];
23   for select in from do
24      rowRange1 = select<|"A1"|>;
25      row6 = <|"a":rowRange1<|"a"|>, "b":rowRange1<|"b"|>|>;
26      append(result7, row6)
27   od;
28   return result7
29 };
```

Listing 4.4: Kompletter Quellcode der Übersetzten Query 4.3

4.3 Aggregierende Query

```
SELECT   a, sum(b)
FROM     grp
GROUP BY a;
```

Die Auswertung dieser Query erfolgt in insgesamt vier Phasen. In der Gruppierungsphase wird eine Dictionary-Datenstruktur basierend auf dem Gruppierungskriterium aufgebaut. Diese Datenstruktur wird anschließend in der Aggregationsphase verarbeitet und zu einer Liste von Rows überführt. Insgesamt müssen zwei Aggregierungen berechnet werden, da die Spalte a ein implizites Aggregat ist, siehe auch 2.7.4.1. Danach folgt die Ergebnisgenerierung welche die Auswertung abschließt. Die From-Where Phase ist bis auf die Filterung in Zeile 10 identisch zu dem in Listing 4.4 angeführten Code.

```
def sfwblock16(table0:[<|"a":int, "b":int|>]):[<|"a":int, "sum":int|>]
{ # From Code: ;
  [...]
  # GroupBy Code: ;
  var grpRowVar6:<|"rowEntry5":int|>;
  var groups4:{<|"rowEntry5":int|>:[<|"A1":<|"a":int, "b":int|>|>]};
  groups4 = {}:{<|"rowEntry5":int|>:[<|"A1":<|"a":int, "b":int|>|>]};
  for f in from do
     rowRange1 = f<|"A1"|>;
     grpRowVar6 = <|"rowEntry5":rowRange1<|"a"|>|>;
     if (keys(groups4) contains grpRowVar6)
        then skip
        else update(groups4, grpRowVar6, [])
     fi;
     var grpValues7:[<|"A1":<|"a":int, "b":int|>|>];
     grpValues7 = groups4{grpRowVar6};
     append(grpValues7, f);
     update(groups4, grpRowVar6, grpValues7)
  od;
```

Listing 4.5: Funktionskopf und Gruppierungsphase

Die Berechnung der Aggregate findet in dem markierten Code-Abschnitt statt. Es wird in der äußeren Schleife über die zuvor gebildeten Gruppen iteriert und in den jeweils inneren Schleifen über die Elemente der aktuellen Gruppe. Im letzten Schritt wird eine Row mit den Ergebnissen der Berechnungen erzeugt und an die Zwischenergebnistabelle angehängt.

```
20     # Aggregation Code: ;
21     var aggs:[<|"aggs":<|"agg0":int, "agg1":int|>|>];
22     aggs = []:[<|"aggs":<|"agg0":int, "agg1":int|>|>];
23     var aggRow:<|"agg0":int, "agg1":int|>;
24     for g in keys(groups4) do
25        var the8:int;
26        for x in groups4{g} do
27           rowRange1 = x<|"A1"|>;
28           the8 = rowRange1<|"a"|>
29        od;
30        var sum9:int;
31        sum9 = 0;
32        for x in groups4{g} do
33           rowRange1 = x<|"A1"|>;
34           sum9 = (sum9 + rowRange1<|"b"|>)
35        od;
36        aggRow = <|"agg0":the8, "agg1":sum9|>;
37        append(aggs, <|"aggs":aggRow|>)
38     od;
39     # Select Code: ;
40     [...]
41     for select in aggs do
42        aggRow = select<|"aggs"|>;
43        row13 = <|"a":aggRow<|"agg0"|>, "sum":aggRow<|"agg1"|>|>;
44        append(result14, row13)
45     od;
46     return result14
47  };
```

Listing 4.6: Aggregationsphase und Ergebnisgenerierung

4.4 Deterministischer endlicher Automat

Die Query die diesen Automat ausführt, ist aus [Mül16] entnommen und ein komplexeres Beispiel für die DPA. Die Query prüft die Syntax einer chemischen molekularen Formel.

Die Query ist interessant für die Übersetzung, da es sich um eine rekursive Query handelt. Sie setzt sich aus drei expliziten und einem impliziten Select-From-Where Block zusammen, in der Übersetzung ergibt dies mehrere *sfwblock*-Funktionen. Der implizite Block ist dabei das rekursive Union All.

```sql
WITH RECURSIVE run(compound, step, state, input) AS
(
   SELECT compound,
          0 as step,
          0 as state,
          formula as input
   FROM   compounds
     UNION ALL
   SELECT current.compound,
          current.step + 1 AS step,
          edge.target AS state,
          right(current.input, -1) AS input
   FROM   run current,
          fsm edge
   WHERE  length(current.input) > 0
   AND    current.state = edge.source
   AND    strpos(edge.labels, left(current.input, 1)) > 0
)
SELECT r.step, r.state, r.input
FROM   run r
WHERE  compound = 'citrate';
```

Listing 4.7: Eingabe Query

Dieser Codeabschnitt entspricht den Zeilen 3 bis 7 in Listing 4.7.

```
def sfwblock9(table0:[...]):[...]
{ # From Code: ;
  [...]
  for CTE0_A1_B1 in table0 do
     rowRange1 = CTE0_A1_B1;
     var tempRow2:<|"CTE0_A1_B1":<|"compound":string, "formula":string|>|>;
     tempRow2 = <|"CTE0_A1_B1":rowRange1|>;
     append(from, tempRow2)
  od;
  # Select Code: ;
  [...]
  for select in from do
     row6 = <|"compound":select<|"CTE0_A1_B1"|><|"compound"|>,
              "step":0, "state":0,
              ↪  "input":select<|"CTE0_A1_B1"|><|"formula"|>|>;
     append(result7, row6)
  od;
  return result7
};
```

Listing 4.8: Initialisierung der rekursiven Query

Der Code in Listing 4.9 entspricht der rekursiven Berechnung in den Zeilen 9 bis 17 der Query 4.7. In der From-Where Phase wird das kartesische Produkt der Tabellen `table10` und `table12` gebildet und gefiltert. Die Inkrementierung von `step` sowie das Voranschreiten im Input geschieht in der Select Phase.

```
def sfwblock21(table10:[...], table12:[...]):[...]
{ # From Code: ;
  [...]
  for CTE0_A2_B1 in table10 do
     rowRange11 = CTE0_A2_B1;
     for CTE0_A2_B2 in table12 do
        rowRange13 = CTE0_A2_B2;
        var filter15:bool;
        filter15 = (((strlen(rowRange11<|"input"|>) > 0)
                    and (rowRange11<|"state"|> == rowRange13<|"source"|>))
                   and (strpos(rowRange13<|"labels"|>,
                        strleft(rowRange11<|"input"|>, 1)) > 0));
        if filter15
           then [...]
              tempRow14 = <|"CTE0_A2_B1":rowRange11, "CTE0_A2_B2":rowRange13|>;
               append(from, tempRow14)
           else skip
        fi
     od
  od;
  [...]
  for select in from do
     row18 = <|"compound":select<|"CTE0_A2_B1"|><|"compound"|>,
              "step":(select<|"CTE0_A2_B1"|><|"step"|> + 1),
              "state":select<|"CTE0_A2_B2"|><|"target"|>,
              "input":strright(select<|"CTE0_A2_B1"|><|"input"|>, -1)|>;
     append(result19, row18)
  od;
  return result19
};
```

Listing 4.9: Rekursiver Teil der Query

In diesem Teil des Codes wird wiederholt `sfwblock21`, also der rekursive Teil der Query mit jeweils der aktuellen Zwischenergebnistabelle aufgerufen. Dies entspricht der in Listing 2.14 auf Seite 16 beschriebenen Semantik.

```
def sfwblock33(table0:[...], table10:[...], table12:[...]):[...]
{ # From Code: ;
  # Start of recUnionAll;
  var recUnionAll23:[<|"compound":string, "input":string,
                      "state":int, "step":int|>];
  recUnionAll23 = [];
  var baseRecQ22:[<|"compound":string, "input":string,
                   "state":int, "step":int|>];
  baseRecQ22 = sfwblock9(table0);
  recUnionAll23 = baseRecQ22;
  var recUnionAllres24:[<|"compound":string, "input":string,
                         "state":int, "step":int|>];
  recUnionAllres24 = [];
  while (len(recUnionAll23) > 0) do
     for row in recUnionAll23 do
        append(recUnionAllres24, row)
     od;
     recUnionAll23 = sfwblock21(recUnionAll23, table12)
  od;
  # End of recUnionAll;
  [...]
};
```

Listing 4.10: Übersetzung des rekursiven Union All

Der letzte Schritt besteht aus der Ergebnisgenerierung. Diese wird wie in Abschnitt 4.2 beschrieben berechnet.

```
def sfwblock43(table34:[...]):[...]
{ # From Code: ;
  [...]
  for A1 in table34 do
     rowRange35 = A1;
     var filter37:bool;
     filter37 = (rowRange35<|"compound"|> == "citrate");
     if filter37
        then var tempRow36:<|"A1":<|"compound":string,
                                    "input":string,
                                    "state":int,
                                    "step":int|>|>;
             tempRow36 = <|"A1":rowRange35|>;
             append(from, tempRow36)
        else skip
     fi
  od;
  [...]
  for select in from do
     rowRange35 = select<|"A1"|>;
     row40 = <|"step":rowRange35<|"step"|>,
               "state":rowRange35<|"state"|>,
               "input":rowRange35<|"input"|>|>;
     append(result41, row40)
  od;
  return result41
};
```

Listing 4.11: Generierung des Resultats der gesamten Query

4.5 Sliding Window

In diesem Abschnitt wird die Übersetzung einer Window-Function im Sliding-Window Modus angeführt. Die angegebene Query berechnet eine Laufsumme mit einer Window-Größe von 3.

```
SELECT a, b,
       sum(b) over (ORDER BY a,b
                    ROWS BETWEEN 1 PRECEDING
                                 AND
                                 1 FOLLOWING)
FROM grp;
```

Da an der Query nur eine Tabelle beteiligt ist, wird diese in der From-Where Phase in die interne Datenrepräsentation übertragen ohne ein kartesisches Produkt zu bilden. Eine Filterung findet nicht statt.

```
1  def sfwblock27(table0:[<|"a":int,
2                          "b":int|>]):[<|"a":int,"b":int,"sum":int|>]
3  { # From Code: ;
4    var rowRange1:<|"a":int,"b":int|>;
5    var from:[<|"A1":<|"a":int,"b":int|>|>];
6    from = []:[<|"A1":<|"a":int,"b":int|>|>];
7    for A1 in table0 do
8       rowRange1 = A1;
9       var tempRow2:<|"A1":<|"a":int,"b":int|>|>;
10      tempRow2 = <|"A1":rowRange1|>;
11      append(from, tempRow2)
12   od;
```

Bevor die eigentliche Window-Function berechnet wird, muss in einem Zwischenschritt das Zwischenergebnis der From-Where Phase transformiert werden. Dies ist notwendig, um eine gleichmäßig tiefe Schachtelung der Daten zu gewährleisten. Anschließend wird anhand der im Window angegebenen Spalten Sortiert und Gruppiert. Danach kann die Window-Function berechnet werden. Dabei wird in der äußeren Schleife ähnlich wie bei Aggregationsfunktionen über die Gruppen iteriert und in den inneren Schleifen über die Elemente der aktuellen Gruppe. Anders als bei Aggregaten, bei denen nur einmal über die Gruppe iteriert wird, erfordert es die Implementierung der Semantik von Windowfunctions, dass verschachtelt über die Gruppe iteriert wird.

```
13  var wins:[<|"from":<|"A1":<|"a":int,"b":int|>|>|>];
14  wins = []:[<|"from":<|"A1":<|"a":int,"b":int|>|>|>];
15  for g in from do
16     append(wins, <|"from":g|>)
17  od;
18  var sorted:[<|"from":<|"A1":<|"a":int,"b":int|>|>|>];
19  sorted = sort5(wins);
20  var groups13:{<|"k14":int|>:[<|"from":<|"A1":<|"a":int,"b":int|>|>|>]};
21  groups13 =
    ↪ {<|"k14":0|>:sorted}:{<|"k14":int|>:[<|"from":<|"A1":<|"a":int,"b":int|>|>|>]};
22  var wins:[<|"from":<|"A1":<|"a":int,"b":int|>|>,"wins":<|"win0":int|>|>];
23  wins = []:[<|"from":<|"A1":<|"a":int,"b":int|>|>,"wins":<|"win0":int|>|>];
24  var winRow:<|"win0":int|>;
25  for g in keys(groups13) do
26     var group:[<|"from":<|"A1":<|"a":int,"b":int|>|>|>];
27     group = groups13{g};
28     var preceding16:int;
29     var current17:int;
30     var following18:int;
31     var current17:int;
32     current17 = 0;
33     for tuple in group do
34        var counter19:int;
35        counter19 = 0;
36        current17 = (current17 + 1);
37        preceding16 = (current17 - 1);
38        following18 = (current17 + 1);
39        var sum15:int;
40        sum15 = 0;
41        for x in group do
42           counter19 = (counter19 + 1);
43           if ((counter19 >= preceding16) and (counter19 <= following18))
44              then rowRange1 = x<|"from"|><|"A1"|>;
45                   sum15 = (sum15 + rowRange1<|"b"|>)
46              else skip
47           fi
48        od;
49        winRow = <|"win0":sum15|>;
50        append(wins, <|"wins":winRow, "from":tuple<|"from"|>|>)
51     od
52  od;
```

In der Select Phase wird das Endergebnis generiert. Hierbei kann man sehen, wie auf die neu generierte Spalte `win0` zugegriffen wird.

```
53    # Select Code: ;
54    var row24:<|"a":int,"b":int,"sum":int|>;
55    var result25:[<|"a":int,"b":int,"sum":int|>];
56    result25 = []:[<|"a":int,"b":int,"sum":int|>];
57    for select in wins do
58       rowRange1 = select<|"from"|><|"A1"|>;
59       winRow = select<|"wins"|>;
60       row24 =
       ↪  <|"a":rowRange1<|"a"|>,"b":rowRange1<|"b"|>,"sum":winRow<|"win0"|>|>;
61       append(result25, row24)
62    od;
63    return result25
64 };
```

5 DATA PROVENANCE

Der Fokus dieser Arbeit liegt zwar nicht bei der Data Provenance Analyse (DPA), dennoch war es ein wichtiges Kriterium, dass der generierte KL Code möglichst 'schöne' Ergebnisse bei der nachfolgenden DPA liefert. Detaillierte Informationen über die verwendete Analysemethode ist [Mül15, Mül16] zu entnehmen.

5.1 Aggregate

Bei der Provenance von Aggregationsfunktionen erkennt man im Ergebnis im Allgemeinen einen Zusammenhang zwischen einer Ergebniszelle zur Input-Gruppe.

```
SELECT a, sum(b)
FROM grp
GROUP BY a;
```

Listing 5.1: Einfaches Aggregat

table 0	
b	a
1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	3

result 12	
sum	a
6	1
15	2
24	3

Abbildung 5.2: Provenance Ergebnis von Query 5.1

5.2 Window Functions

Die Provenance dieser Query zeigt den Zusammenhang einer Ergebniszelle zu den jeweiligen Windows. Zudem kann man hier nochmals sehen, dass im Gegensatz zu Aggregaten, in denen jeweils eine Gruppe zu einer Zeile zusammengefasst wird, hier die Eingabezeilen erhalten bleiben.

```sql
SELECT a, b,
       sum(b) over (ORDER BY a,b
                    ROWS BETWEEN 1 PRECEDING
                                 AND
                                 1 FOLLOWING)
FROM grp;
```

Listing 5.3: Sliding Window

table 0	
b	a
1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	3

result 0		
b	a	sum
1	1	3
2	1	6
3	1	9
4	2	12
5	2	15
6	2	18
7	3	21
8	3	24
9	3	17

Abbildung 5.4: Provenance Ergebnis von Query 5.3

6 DISKUSSION UND AUSBLICK

Wie bereits in Kapitel 1 erwähnt, war es Teil der Zielsetzung, dass der Compiler Subqueries, Aggregierungen, rekursive Queries und auch Window Functions typisiert übersetzen kann. Dieses Ziel wurde erreicht.

Der Compiler ist wesentlich von der Input Datenstruktur abhängig. Diese muss allerdings, um neue Features unterstützen zu können bzw. neuen Anforderungen zu entsprechen, verändert oder zumindest erweitert werden. Abhängig vom Umfang dieser Anpassungen müssen unter Umständen auch die Übersetzungsregeln verändert werden. Während der Entwicklungszeit des Compilers wurde das Inputformat bereits mehrfach erweitert und teilweise verändert. Da der Compiler modular aufgebaut ist, stellt dies kein großes Problem dar, solange sich die Struktur des Inputs nicht *erheblich* verändert. In diesem Fall können diese Änderungen mit moderatem Aufwand auch im Compiler durchgeführt werden.

6.1 Einschränkungen der Implementierung

6.1.1 Null Values

Der SQL-Ausdruck **Null** ist alltäglich in Datenbanksystemen und steht für die Abwesenheit eines Datenwertes. Allerdings handelt es sich dabei nicht um einen speziellen Datenwert, sondern um einen *Zustand*. Damit stehen wesentliche semantische Komplikationen im Zusammenhang, die eine Übersetzung von SQL erheblich komplizierter machen, da viele Konstrukte zusätzliche Ausnahmeregeln erfordern. Diese Ausnahmen zu Formulieren ist nicht ohne weiteres umzusetzen und unter den verschiedenen SQL-Standards inkonsistent. Es scheint nicht möglich zu sein, eine *intuitive* semantische Behandlung von **Null** für SQL zu definieren. [vdM98, S. 344]

Der notwendige Aufwand, um sinnvolle und SQL konforme Ausnahmeregelungen in die Übersetzung einzubeziehen, hätte den Rahmen der ohnehin schon recht umfangreichen Zielsetzung gesprengt. Aus diesem Grund unterstützt der hier beschriebene Compiler keine Null-Values.

6.1.2 Skalaroperationen

In der Entwicklungszeit des Compilers haben sich einige Schwächen in dem genutzten *Log-Parser* gezeigt. Bei Skalaroperationen war es aufgrund von fehlerhaften bzw. fehlenden Typen nicht möglich *allgemeine* Übersetzungsregeln zu entwerfen, da keine Typinferenz realisierbar ist. Um entsprechende Queries übersetzen zu können, ist es durch den mangelhaften Input notwendig, eine für die Query angepasste *trType* Regel einzuführen. Die Einführung solcher speziellen Regeln ist weder intuitiv noch empfehlenswert und erfordert in der Regel eine manuelle Fehlerbeseitigung in der bereits übersetzten Query. Bei korrektem Input wäre es möglich, diese manuelle Nacharbeit zu vermeiden und korrekte Übersetzungen zu generieren.

6.1.3 Window Function Range peer groups

Werden Window Functions im *Range* Modus ausgewertet, entstehen bei Duplikaten laut SQL-Standard sogenannte *Peer Groups*, die gesondert behandelt werden müssen. Diese Feinheit der Semantik wird in der aktuellen Übersetzungsregel ignoriert. Trotz dessen wird gültiger KL Code der Query erzeugt. Da dieses Verhalten nicht SQL-Standard-konform ist, wird zusätzlich zur Übersetzung der Query eine Warnung mit einem Hinweis auf diesen semantischen Unterschied ausgegeben.

Beispiel 6.1: Peer Groups

```sql
SELECT salary, sum(salary) OVER (ORDER BY salary),
               rank(salary) OVER (ORDER BY salary)
FROM empsalary;
```

PSQL		
salary	sum	rank
3500	3500	1
3900	7400	2
4200	11600	3
4500	16100	4
4800	25700	5
4800	25700	5
5000	30700	7
5200	41100	8
5200	41100	8
6000	47100	10

Übersetzung		
salary	sum	rank
3500	3500	1
3900	7400	2
4200	11600	3
4500	16100	4
4800	20900	5
4800	25700	5
5000	30700	7
5200	35900	8
5200	41100	8
6000	47100	10

6.2 Compiler als Lehrwerkzeug

In erster Linie ist der SQL-Compiler für die DPA Forschung am Lehrstuhl für Datenbanksysteme entstanden. Eine alternative Verwendung als Lehrwerkzeug ist allerdings naheliegend und könnte ein vielversprechender Ansatz sein, um die Semantik von SQL aus einem imperativen Blickwinkel zu vermitteln. Die Zielsprache KL verfügt, wie bereits in Abschnitt 2.3 erwähnt, über eine einfache Syntax, die ohne große Einarbeitungszeit verstanden werden kann. Es ist denkbar sich diese Eigenschaft zu Nutze zu machen. Dadurch wäre es eventuell möglich, ein besseres Verständnis über die Auswertungsstrategie von Queries in einem compilierenden DBMS zu vermitteln. Dies könnte insbesondere bei komplexen Queries zum Verständnis beitragen.

6.3 Ausblick

Wie bereits bei den Einschränkungen der Implementierung erwähnt, werden Skalaroperationen aufgrund von mangelhaftem Input nur rudimentär unterstützt. Eine wünschenswerte Weiterentwicklung wäre es daher, den Log Parser entsprechend zu überarbeiten um anschließend stabile Übersetzungsregeln entwerfen zu können.

Aktuell werden Variablennamen generisch erzeugt. Dies erschwert das Debuggen einer übersetzten Query. Ein Mechanismus, der Namen aus der SQL-Query in der Übersetzung sinnvoll wiederverwendet, könnte hier Abhilfe schaffen, da sich Codeabschnitte dadurch besser den Ausdrücken in einer Query zuordnen lassen.

Abbildungsverzeichnis

2.1 Übersicht der Compiler Struktur mit anschließender Analyse. 4
2.4 Für Typen relevanter Ausschnitt der JSON Repräsentation 8
2.7 Organisation der verschiedenen Auswertungsphasen einer SFW-Query. 12
2.8 Schema der Gruppierung durch Hashing einer Relation 13
2.15 Vereinfachte Darstellung der JSON Repräsentation von SQL-Queries 17

5.2 Provenance Ergebnis von Query 5.1 51
5.4 Provenance Ergebnis von Query 5.3 52

Literaturverzeichnis

[ALUS11] A.V. Aho, M.S. Lam, J.D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2011.

[Cod90] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Mül15] Tobias Müller. Where-und why-provenance für syntaktisch reiches sql durch kombination von programmanalysetechniken. In *GvD*, pages 84–89, 2015.

[Mül16] Tobias Müller. Have your cake and eat it, too: Data provenance for turing-complete sql queries. 2016.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4:539–550, 2011.

[NL14] Thomas Neumann and Viktor Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37:3–11, 2014.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[vdM98] Ron van der Meyden. Logical approaches to incomplete information: A survey. In *Logics for databases and information systems*, pages 307–356. Springer, 1998.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Unterschrift