Title	Formal Semantics of Core SQL Language based on the K Framework
Author(s)	Sitthisak, Pakakorn
Citation	Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2015-001: 1-71
Issue Date	2015-02-17
Type	Technical Report
Text version	publisher
URL	http://hdl.handle.net/10119/12871
Rights	
Description	リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）

Formal Semantics of Core SQL Language based on the K Framework

Pakakorn Sitthisak

School of Information Science
Japan Advanced Institute of Science and Technology

February 17, 2015

IS-RR-2015-001

Formal Semantics of Core SQL Language based on the K Framework

By Pakakorn Sitthisak

A project paper submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

December, 2014

Abstract

There are lots of SQL dialects, e.g., MySQL, various versions of Oracle, and Microsoft SQL Server, ProgreSQL, and more. They share common semantics on standard table operations (with slight syntax differences), which appear in a textbook of relational database management systems. However, formal semantics of non-standard operations, e.g., type violation like `SELECT 1 + "1a"`, varies in detail. Most of programmers in system development do not aware of such differences, which will be crucial when applying formal methods. They are typically *coercion*, `NULL`, the *name space*, and the *error handling*. Even a standard operation JOIN varies depending on detailed types (including the bit-width) of arguments.

This thesis investigates detailed semantics of the core of SQL, specifically on MySQL and Oracle11. First, we observe their formal semantics by testing queries on boundary cases. Next, the semantics of the core of MySQL is implemented on the K framework. We call it *KSQL*, which covers basic table operations, like *selection, creation, deletion, update*, and *insertion*. They are defined with the features of *coercion*, `NULL`, and *the name space convention*. Lastly, we discuss on current limitations and difficulties in KSQL implementation.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Mizuhito Ogawa. He gave me an invaluable opportunity to study abroad. Without his support, encouragement, and kind advice, I could not complete my master course.

Beside my supervisor, I would like to thank Associate Professor Nao Hirokawa for his great guidance on writing thesis and precious lessons from his experience.

My sincere thanks also goes to Dr. Min Zhang for his kind tutorial on my research.

I thank my fellow for stimulation, worth discussion, and plentiful help.

Lastly, I would like to express special thanks to my family. They always give me financial support, continuous love, and great morale throughout my life.

Contents

Chapter 1

Introduction

Formal semantics of a programming language is required in many views. For instance, understanding detailed behaviour of languages reduces bugs of implementation. Automatic support by verification / analysis tools is constructed on formal semantics. Although formal semantics is often embedded into algorithms and/or implementation of such tools, there are several attempts to define formal semantics alone, e.g., Java [4], ANSI-C [3], PHP [11], Verilog [6], Scheme [5], x86 [2], and HTML5 [7]. Among them, for Java, ANSI-C, Verilog, and Scheme are implemented on K framework, thus executable.

Our aim is to give formal semantics of SQL, and clarify their differences among SQL dialects, e.g., MySQL, various versions of Oracle, Microsoft SQL Server, ProgreSQL, and more. They share common semantics on standard table operations (with slight syntax differences), like selection, insertion, deletion, creation, update, and join which are popular in a textbook of relational database management systems (RDBMSs). However, consider the following query. What do MySQL and Oracle11 return?

MySQL query : `SELECT 1 + "" ;`
Oracle query : `SELECT 1 + '' FROM DUAL;`

One possibility is simply an error, because the addition + accepts numbers as its arguments. However, the addition of integer and string is valid in MySQL and Oracle11, and they return `1` and `''` (empty string), respectively.

In this thesis, formal semantics of the core (a subset) of SQL is investigated. We first compare detailed semantics of MySQL and Oracle11, and next, the semantics of the core of MySQL is implemented on K-framework. We call it *KSQL*, which covers basic table operations, like *selection, creation, deletion, update*, and *insertion*.

In our study, we found two main issues to cause semantic differences between MySQL and Oracle11.

- Operations on boundary values, e.g., coercion, `NULL`, the name space convention, and error handling.

- Two different layers, logical and physical models of data types. Dialects have their own design of data types. For example, Oracle11 has NUMBER type, which allows users to specify the precision and the scale factor, while MySQL has predefined data types, such as TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT corresponding to 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integers, respectively.

In this thesis, we investigate the first issue, specifically, coercion, `NULL`, and the name space management, over basic operations. They are implemented on K framework [9], which is constructed on Maude, a programming language based on algebraic specification. Algebraic specification consists of rewriting rules (equations) over terms with sorts. We describe states of SQL as terms and SQL (small step) operational semantics of SQL as rewriting rules.

Lastly, we discuss on current limitations and difficulties in KSQL implementation.

Contributions

Our contributions are:

- The formal semantics of the core SQL language including selection, insertion, creation, deletion, and update statements.
- Differences between MySQL and Oracle11: coercion, `NULL`, and the name space convention.
- Semantics definition on K framework of detailed behaviour: coercion, `NULL`, and the name space convention in MySQL.
- Explanation and analysis of difficulties for defining semantics of SQL.

This thesis is organized as follows:

- Chapter 2 provides technical background about term rewriting systems and algebraic specification.
- Chapter 3 provides a brief introduction of K framework.
- Chapter 4 explains differences and choices among SQL semantics, based on observation on boundary cases.
- Chapter 5 describes basic description of semantics of MySQL on K framework.
- Chapter 6 describes semantics of coercion, `NULL`, and the name space convention of MySQL on K framework.
- Chapter 7 discuses on related work.
- Chapter 8 concludes the thesis and mentions future direction.

Chapter 2

Preliminaries

In this chapter, we explain term rewriting in the first section and algebraic specification in the second section.

2.1 Term rewriting system

Definition 2.1. *Let $\mathcal{V}$ be a countable set of variables, and $\mathcal{F}$ a set of function symbols associated with the arity mapping $\mathtt{ar} : \mathcal{F} \to \mathbb{N}$. We call $\mathcal{F}$ a* signature *and $f \in \mathcal{F}$ has arity n if there exists $n \in \mathbb{N}$ which satisfies $\mathtt{ar}(f) = n$. We call a function f* constant *if $\mathtt{ar}(f) = 0$.*

Definition 2.2. *The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of* terms *over the signature $\mathcal{F}$ is the smallest set satisfying the following conditions:*

- *if $x \in \mathcal{V}$ then $x \in \mathcal{T}(\mathcal{F}, \mathcal{V})$,*
- *if $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and $f \in \mathcal{F}$ which has arity n then $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.*

Example 2.3. *Let $\mathcal{V} = \{x, y\}$ and $\mathcal{F} = \{\mathtt{0}, \mathtt{s}, +\}$ with $\mathtt{ar}(\mathtt{0}) = 0$, $\mathtt{ar}(\mathtt{s}) = 1$, and $\mathtt{ar}(+) = 2$. Then the following terms are members of $\mathcal{T}(\mathcal{F}, \mathcal{V})$: $\mathtt{0}, \mathtt{s}(\mathtt{0}), \mathtt{s}(x)$, $\mathtt{s}(\mathtt{s}(x))$, $\mathtt{0} + \mathtt{s}(\mathtt{0})$, and $\mathtt{x} + \mathtt{s}(\mathtt{y})$.*

Definition 2.4. *Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. We inductively define the set $V(t)$ of variables occurring in t as follows:*

$$V(t) = \begin{cases} \{t\} & \text{if } t \in \mathcal{V} \\ \bigcup_{i=1}^{n} V(t_i) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Definition 2.5. *A* position *is a sequence of positive integers. The position of empty sequence is denoted by ϵ and the concatenation of positions p and q is $p.q$. The set $Pos(t)$ of positions of a term t is*

$$Pos(t) = \begin{cases} \{\epsilon\} & \text{if } t \in \mathcal{V} \\ \{\epsilon\} \cup \bigcup_{1 \le i \le n} \{i.p \mid p \in Pos(t_i)\} & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

Definition 2.6. *A* subterm $t|_p$ *of t at the position p is inductively defined as follows:*

$$t|_p = \begin{cases} t & \text{if } p = \epsilon \\ t_i|_q & \text{if } t = f(t_1, \dots, t_n) \text{ and } p = i.q \end{cases}$$

Definition 2.7. *If t' is a term, A term $t[t']_p$ denotes a term that is obtained from t by replacing the subterm at the position p with t':*

$$t[t']_p = \begin{cases} t' & \text{if } p = \epsilon \\ f(t_1, \dots, t_i[t']_q, \dots, t_n) & \text{if } t = f(t_1, \dots, t_n) \text{ and } p = i.q \end{cases}$$

Example 2.8 (Continued from Example 2.3)**.** *Let $t = (\mathsf{0}+\mathsf{s}(x))+(y+\mathsf{s}(x))$. Then we have $Pos(t) = \{\epsilon, 1, 11, 12, 121, 2, 21, 22, 221\}$. We have $t|_{11} = 0, t|_{221} = x$ and $t|_2[\mathsf{0}]_1 = \mathsf{0}+\mathsf{s}(x)$.*

Definition 2.9. *A* rewrite rule *is a pair (l, r) of terms that $l \notin \mathcal{V}$ and $V(r) \subseteq V(l)$. A rewrite rule (l, r) is denoted by $l \to r$. A* term rewriting system (TRS) *$\mathcal{R}$ is a set of rewrite rules over the signature $\mathcal{F}$.*

Example 2.10 (Continued from Example 2.3)**.** *We can define the term rewriting system $\mathcal{R}$ as below:*

$$\mathsf{0} + y \to y$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y).$$

2.2 Algebraic specification

Sort is a set of values. Ordered sorts are sorts with partial relation between them, called *subsort relation*. We use sorts to define a domain and a range of a function. The subsort relation gives the benefit of the sort inheritance, such that if A is a subsort of B, then every variable or constant of A is also a variable or a constant of B. Moreover, ordered sorts are useful for overloading functions. Since the SQL language has many data types, sets of values, throughout this thesis, we adopt order-sorted term rewriting. In this section we recall the notations for order-sorted terms.

Let $\mathcal{S}$ be a set of sorts equipped with a subsort relation $\sqsubseteq$ on $\mathcal{S}$.

Definition 2.11. *Let $\mathcal{F}$ be a set of pairs (f, τ) with a* function symbol f *and* $\tau \in \mathcal{S}^+$*. The set $\mathcal{F}$ is an* order-sorted signature *if the implication*

$$\frac{\tau_1 \sqsubseteq \tau_1' \quad \cdots \quad \tau_n \sqsubseteq \tau_n'}{\tau_0 \sqsubseteq \tau_0'}$$

holds for all $(f, \tau_1 \cdots \tau_n \tau_0), (f, \tau_1' \cdots \tau_n' \tau_0') \in \mathcal{F}$. A pair $(f, \tau_1 \cdots \tau_n \tau_0)$ is also denoted by $f : \tau_1 \times \cdots \times \tau_n \to \tau_0$ where n is the arity of f.

Example 2.12. *Let $\mathcal{S} = \{Int, NeList, List\}$ and $\sqsubseteq$ its subsort order on relation with $Int \sqsubseteq NeList \sqsubseteq List$. The set $\mathcal{F}$ consisting of*

$$\begin{array}{lll} \mathsf{nil} : List & \mathsf{cons} : List \times List \to List & \mathsf{head} : NeList \to Int \\ & \mathsf{cons} : NeList \times List \to NeList & \mathsf{tail} : NeList \to List \end{array}$$

forms an order-sorted signature.

We extend terms with sorted terms. Let $\mathcal{F}$ be a set of order-sorted signatures and let $\mathcal{V}$ be a set of *variables* as the disjoint union of $\mathcal{V}^\tau$ for all sorts $\tau \in \mathcal{S}$.

Definition 2.13. *The* sort judgment $t : \tau$ *is defined by the next inference rules:*

$$\frac{x \in \mathcal{V}^\tau}{x : \tau} \qquad \frac{f : \tau_1 \times \cdots \times \tau_n \to \tau \in \mathcal{F} \quad t_i : \tau_i \text{ for all } i}{f(t_1, \ldots, t_n) : \tau} \qquad \frac{t : \tau' \quad \tau' \sqsubseteq \tau}{t : \tau}$$

The set $\{t \mid t : \tau$ for some $\tau\}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and its elements are called (well-sorted) terms*.*

Example 2.14. *Let $\mathcal{S} = \{Nat, Bool\}$ and $\mathcal{F}$ a set of $\mathcal{S}$-sorted signature consisting of the following:*

$$\begin{array}{lll} \mathtt{eq} : Nat \times Nat \to Bool & + : Nat \times Nat \to Nat & \mathtt{true} : Bool \\ \mathtt{s} : Nat \to Nat & \& : Bool \times Bool \to Bool & \mathtt{false} : Bool \\ \mathsf{0} : Nat & & \end{array}$$

Let $\mathcal{V}^{Nat} = \{x, y\}$ and $\mathcal{V}^{Bool} = \{p, q\}$. Then sorted terms $x, y, \mathtt{0}, \mathtt{s}(0), \mathtt{s}(x), \mathtt{s}(y), \mathtt{s}(\mathtt{s}(x))$, and $\mathtt{eq}(\mathtt{s}(x), \mathtt{s}(y))$, p, $\mathtt{true}$ & p, p & q are members of $\mathcal{T}(\mathcal{F}, \mathcal{V})$ while $\mathtt{true} + \mathtt{s}(0)$, 1 & $\mathtt{false}$, and $\mathtt{eq}(\mathtt{true}, \mathtt{0})$ are not.

Definition 2.15. *A* substitution *σ is a map from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ if the following conditions hold:*

- *If $x : \tau$ and $\sigma(x) : \tau'$ are the sort judgments of terms x and $\sigma(x)$, then $\tau = \tau'$.*
- *The domain of σ is finite, where the domain of σ is given by $dom(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$.*

We extend substitution definition to a term t as follow:

$$\sigma(t) = \begin{cases} t' & \text{if } t \text{ is a variable and } \sigma(t) = t' \\ f(\sigma(t_1), \ldots, \sigma(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

We write $t\sigma$ for $\sigma(t)$.

Example 2.16. *Consider the substitution $\sigma = \{x \mapsto x + z, y \mapsto x\}$. If $t = x + (\mathsf{s}(y) + (z + x))$, then $t\sigma = (x + z) + (\mathsf{s}(x) + (z + (x + z)))$.*

Definition 2.17. *An* equality *is a pair denoted by $l \approx r$ where l, r are order-sorted terms which satisfy $l : \tau$ and $r : \tau'$, and then $\tau = \tau'$. We call a set of equations $\mathcal{E}$ an* equation system*. We define $\approx_{\mathcal{E}}$ the smallest equivalence relation which $C[l\sigma] \approx_{\mathcal{E}} C[r\sigma]$ holds for all equations $l \approx r \in \mathcal{E}$, contexts C, and substitution σ.*

Definition 2.18. *An* order-sorted rewrite rule *is a rewrite rule $l \to r$ which satisfies: $l : \tau$ and $r : \tau'$, and then $\tau' \sqsubseteq \tau$. We call a set of order-sorted rewrite rules $\mathcal{R}$ an* order-sorted term rewriting system.

Definition 2.19. *Given an order-sorted rewriting system $\mathcal{R}$ and an order-sorted equation system $\mathcal{E}$. A term t rewrites to t' with a rewrite relation $\to_{\mathcal{R}/\mathcal{E}}$, in rewriting modulo equations, if there exists a rewrite rule $l \to r \in \mathcal{R}$, a term C, a position p, and a substitution σ such that $t \approx_{\mathcal{E}} C[\sigma(l)]_p$, and $t' \approx_{\mathcal{E}} C[\sigma(r)]_p$. We write $t \to_{\mathcal{R}/\mathcal{E}} t'$ and call it a* rewrite step*. When $\mathcal{E}$ is empty, we simply write $\to_{\mathcal{R}}$ instead of $\to_{\mathcal{R}/\mathcal{E}}$.*

Example 2.20 (Continued from Example 2.14)**.** *We can define the sorted term rewriting system $\mathcal{R}$ as below:*

$$\begin{aligned}
\mathtt{eq}(\mathtt{0},\mathtt{0}) &\to \mathtt{true}\\
\mathtt{eq}(\mathtt{0},\mathtt{s}(\mathtt{0})) &\to \mathtt{false}\\
\mathtt{eq}(\mathtt{s}(\mathtt{0}),\mathtt{0}) &\to \mathtt{false}\\
\mathtt{eq}(\mathtt{s}(x),\mathtt{s}(y)) &\to \mathtt{eq}(x,y)\\
\mathtt{0}+y &\to y\\
\mathtt{s}(x)+y &\to \mathtt{s}(x+y).\\
\mathtt{false}\ \&\ p &\to \mathtt{false}\\
\mathtt{true}\ \&\ \mathtt{false} &\to \mathtt{false}\\
\mathtt{true}\ \&\ \mathtt{true} &\to \mathtt{true}
\end{aligned}$$

For instance, computation of $\mathsf{s}(0)+\mathsf{s}(0))+\mathsf{s}(0)$ is done by the following rewrite steps:

$$\begin{aligned}
\mathtt{eq}(\mathsf{s}(0)+\mathsf{s}(0),\mathsf{s}(0))\ \&\ \mathtt{true} &\to_{\mathcal{R}} \mathtt{eq}(0+\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(\mathsf{s}(0)))\ \&\ \mathtt{true}\\
&\to_{\mathcal{R}} \mathtt{eq}(\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(\mathsf{s}(0)))\ \&\ \mathtt{true}\\
&\to_{\mathcal{R}} \mathtt{eq}(\mathsf{s}(0),\mathsf{s}(0))\ \&\ \mathtt{true}\\
&\to_{\mathcal{R}} \mathtt{eq}(0,0)\ \&\ \mathtt{true}\\
&\to_{\mathcal{R}} \mathsf{true}\ \&\ \mathtt{true}\\
&\to_{\mathcal{R}} \mathsf{true}.
\end{aligned}$$

Example 2.21. *Given a sorted term rewriting system $\mathcal{R}$ and an equation system $\mathcal{E}$ as below:*

$$\mathcal{R} = \left\{ \begin{array}{ll} x\cdot x & \to x \\ \epsilon\cdot x & \to x \end{array} \right\} \qquad \mathcal{E} = \left\{ \begin{array}{ll} x\cdot y & \approx y\cdot x \\ (x\cdot y)\cdot z & \approx x\cdot(y\cdot z) \end{array} \right\}$$

Then the rewrite step $(((1\cdot 2)\cdot 1)\cdot 3) \to_{\mathcal{R}/\mathcal{E}} ((3\cdot 2)\cdot 1)$ *holds by the following sequence:* $(((1{\cdot}2){\cdot}1){\cdot}3) \approx_{\mathcal{E}} (((1{\cdot}1){\cdot}2){\cdot}3) \to_{\mathcal{R}} ((1{\cdot}2){\cdot}3) \approx_{\mathcal{E}} ((3{\cdot}2){\cdot}1)$ *while* $(((1{\cdot}2){\cdot}1){\cdot}3) \to_{\mathcal{R}} ((3{\cdot}2){\cdot}1)$ *does not hold.*

Chapter 3

K framework

The K framework is an executable framework of the language definition. Formalizing a language in the K framework automatically supplies the K analysis tools. K defines a TRS $\mathcal{R}$ and an equation system $\mathcal{E}$ together with their (sorted) signature. In this chapter, we will explain and basic definitions using in the K framework, and show a simple example, called language `SIMPLE`, to show the use in the K framework.

3.1 Basic description in K

Syntax

We define the syntax of sorts of a language as follows:

$$\tau ::= f(\tau_1, \ldots, \tau_n)$$

which stands for

$$f : \tau_1 \times \cdots \times \tau_n \to \tau$$

where $\tau_1, \ldots, \tau_n$ and τ are sorts of the language and f is a function symbol of the language. In the K framework, such a syntax is declared by the keyword `syntax`. We can extend the syntax of sort τ by overwriting new BNF definitions. For instance, we want the sort τ to have terms $\tau_1 \cdots \tau_n$, we can overwrite the BNF syntax of sort τ as follows:

$$\tau ::= t_1 \ \cdots \ \tau ::= t_n$$

This is equivalent to $\tau ::= t_1 \mid \ldots \mid t_n$. In addition, when we define the structure of sort τ as follows:

$$\tau ::= \tau_1 \mid \ldots \mid \tau_n$$

where $\tau_1, \ldots, \tau_n$ are sorts, this yields subsort relations $\tau_1 \sqsubseteq \tau, \ldots, \tau_n \sqsubseteq \tau$.

Definition 3.1. *Let $\mathcal{W}$ is a set of context variables denoted by $\{\Box_1, \ldots, \Box_n\}$. An n-*hole context *is a term in $\mathcal{T}(\mathcal{F}, \mathcal{V} \cup \mathcal{W})$ with the constraint that each hole $\Box \in \mathcal{W}$ appears at most once. Given a substitution $\sigma = \{\Box_1 \mapsto t_1, \ldots, \Box_n \mapsto t_n\}$ we write $C[t_1, \ldots, t_n]$ for $C\sigma$.*

We prepare new syntactical notations of rewrite rules.

Notation 3.2. *Let C be an n-hole context. A single step rewrite rule of form $C[\ell_1, \ldots, \ell_n] \to C[r_1, \ldots, r_n]$ is denoted by*

$$C\left[\frac{\ell_1}{r_1}, \ldots, \frac{\ell_n}{r_n}\right]$$

In the K framework such a rule is declared as a keyword `rule`.

Example 3.3. *Consider a rewrite system $\mathcal{R}$ written by the new syntactical form:*

$$\frac{\mathtt{eq}(0,0)}{\mathtt{true}} \qquad \frac{\mathtt{eq}(0,\mathtt{s}(0))}{\mathtt{false}} \qquad \frac{+(0,y)}{y}$$

$$\mathtt{eq}(\frac{\mathtt{s}(x)}{x}, \frac{\mathtt{s}(y)}{y}) \qquad \frac{\mathtt{eq}(\mathtt{s}(0),0)}{\mathtt{false}} \qquad \frac{+(\mathtt{s}(x),y)}{\mathtt{s}(+(x,y))}.$$

These rules are corresponding to normal rewrite rules as follows:

$$\begin{aligned}
\mathtt{eq}(0,0) &\to \mathtt{true} \\
\mathtt{eq}(0,\mathtt{s}(0)) &\to \mathtt{false} \\
\mathtt{eq}(\mathtt{s}(0),0) &\to \mathtt{false} \\
\mathtt{eq}(\mathtt{s}(x),\mathtt{s}(y)) &\to \mathtt{eq}(x,y) \\
+(0,y) &\to y \\
+(\mathtt{s}(x),y) &\to \mathtt{s}(+(x,y)).
\end{aligned}$$

The variables in $\mathcal{W}$ are used to identify the positions where rewriting takes place. The notation above specifies the subterms to be rewritten and their reducts.

The K framework provides several (predefined) sorts together with related rewrite rules.

Definition 3.4. *A* list, map, *and* bag *are defined as follows:*

$$\begin{aligned}
\mathit{List} &::= \epsilon_L \mid \mathit{List} :: \mathit{List} \mid \tau \\
\mathit{Map} &::= \epsilon_M \mid \mathit{Map} :: \mathit{Map} \mid \mathit{Binding} \\
\mathit{Bag} &::= \epsilon_B \mid \mathit{Bag} * \mathit{Bag} \mid \tau
\end{aligned}$$

where Binding $::= \tau_1 \mapsto \tau_2$ *and* $\tau, \tau_1,$ *and* τ_2 *are sorts.*

A list of sort τ *is a term of concatenation, denoted by* $::$, *of sorts* τ *in* $\mathcal{T}(\mathcal{F})$ *equipped with a term rewriting system* $\mathcal{R}_L$ *and a map of binding is a term of concatenation of Binding, denoted by* $::$, *equipped with a term rewriting system* $\mathcal{R}_M$ *where:*

$$\mathcal{R}_M = \left\{ \ \epsilon_L :: x \quad \to x \ \right\} \qquad\qquad \mathcal{R}_M = \left\{ \ \epsilon_L :: x \quad \to x \ \right\}$$

Remark that the parametric polymorphism is not supported. Therefore, we have to explicitly declare the sorts of elements, However, if the ordered sort of an element is clear from the context, we will omit the sort information.

A cell *of sort* τ *is a term denoted by:*

$$Cell ::= \langle \tau \rangle_{Label}$$

where Label can be any string. A bag *of sort* τ *is a term of the AC operator* $*$ *of sorts* τ *in* $\mathcal{T}(\mathcal{F})$ *equipped with a term rewriting system* $\mathcal{R}_B$ *and an equation system* $\mathcal{E}_B$ *where:*

$$\mathcal{R}_B = \left\{ \ \epsilon_B * x \quad \to x \ \right\} \qquad\qquad \mathcal{E}_B = \left\{ \begin{array}{ll} x * y & \approx y * x \\ (x * y) * z & \approx x * (y * z) \end{array} \right\}$$

We denote

$$\begin{array}{rcl} x_1 :: \cdots :: x_n :: \epsilon_L & by & [x_1, \ldots, x_n],\ or\ (x_1, \ldots, x_n) \\ (x_1 \mapsto y_1) :: \cdots :: (x_n \mapsto y_n) :: \epsilon_M & by & \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\},\ and \\ x_1 * \cdots * x_n * \epsilon_B & by & \{x_1, \cdots, x_n\}. \end{array}$$

For any list $L = [x_1, \ldots, x_i, \ldots, x_n]$, *we denote i-th element* x_i *of list* L *as* $L[i]$. *A* parallel product *of lists is a function* $\otimes : List \times List$ *which is defined as follows:*

$$[x_1, \ldots, x_2] \otimes [y_1, \ldots, y_n] = [(x_1, y_1), \ldots, (x_n, y_n)]$$

Given a list $A = [(x_1, y_1), \ldots, (x_n, y_n)]$, *we write* $A(x_i)$ *for* y_i. Configuration *is a bag of cells.*

Computation

Computation is the top sort over all defined sorts in the language definition. Let K be the sort of the computation, given by

$$K \ ::= \ \epsilon_K \mid K \curvearrowright K \mid \Diamond$$

Sort K is the smallest sort with respect to $\sqsubseteq$ among sorts defined in K. We can regard the sort K as a list of any sort, in which $\curvearrowright$ is the concatenation operator. $\Diamond$ is a reserved constant of the sort K for pointing a K-subterm to be executed. Sort K is equipped with the next rewriting system $\mathcal{R}$:

$$\mathcal{R} = \left\{ \ \epsilon_K \curvearrowright x \quad \to x \ \right\}$$

Strictness Attribute

Strictness attribute is an attribute on a function symbol to define its evaluation strategy. For function $f : \tau_1 \times \cdots \times \tau_n$, evaluation strategy of f is a list of integers i where $1 \leq i \leq n$. The K framework will automatically generate rewrite rules depending on the strictness attribute. The *strict* attribute is corresponding to non-deterministic evaluation strategy. The attribute `seqstrict` shows an sequential ordering of evaluation among arguments.

Example 3.5. *We set a function $\_+\_$ to evaluate its arguments from the left-to-right manner. We annotate* `seqstrict` *attribute of $\_+\_$, which is equivalent to the evaluation strategy $(1, 2)$. The K framework automatically generates the following four rewrite rules [10].*

$$\begin{aligned} a_1 + a_2 &\to a_1 \curvearrowright (\Diamond + a_2) \\ i_1 \curvearrowright (\Diamond + a_2) &\to i_1 + a_2 \\ i_1 + a_2 &\to a_2 \curvearrowright (i_1 + \Diamond) \\ i_2 \curvearrowright (i_1 + \Diamond) &\to i_1 + i_2 \end{aligned}$$

where a_1, a_2 are variables of sort K, and i_1, i_2 are variables of sort Int. The evaluation of $(1 + 2) + (3 + 4)$ is the following rewrite steps:

$$\begin{aligned} (1 + 2) + (3 + 4) &\to (1 + 2) \curvearrowright (\Diamond + (3 + 4)) \\ &\to 3 \curvearrowright (\Diamond + (3 + 4)) \\ &\to 3 + (3 + 4) \\ &\to (3 + 4) \curvearrowright (3 + \Diamond) \\ &\to 7 \curvearrowright (3 + \Diamond) \\ &\to 3 + 7 \\ &\to 10 \end{aligned}$$

Configurations

A configuration of the K framework represents a state of a program. A configuration contains terms of a program and the environments. In the K definition, we have to specify an initial configuration for the initial state for a run of a program.

Example 3.6. *The initial configuration of program $\mathtt{x = 1; y = 2;}$ in the language* `SIMPLE` *(see section 3.2) is defined by the following term constructed by three cells.*

$$\Big\langle \mathtt{x = 1; y = 2;} \Big\rangle_K * \Big\langle \epsilon_M \Big\rangle_{env} * \Big\langle 0 \Big\rangle_{loc}$$

Notation 3.7. *The K framework provides notations, defined by '_' and '$\cdots$', to represent an anonymous, unnamed, variable in the rewrite rules. Symbol '_' is used when a variable is appeared only in the left-hand side of the rule. Symbol '$\cdots$' is used when a variable is appeared both in the left-hand side and the right-hand side of the rule. When we use the abbreviation '$\cdots$', we often omit ',' (comma) and brackets for the list notation, and '$\curvearrowright$' for the sort K.*

The K framework represents the empty value ϵ_τ by $.\tau$ (dot followed by the sort name).

Example 3.8. *The rewrite rule*

$$\left\langle \frac{V = I;}{\epsilon_K} \ \cdots \right\rangle_K \left\langle \cdots \ V \mapsto \frac{\_}{I} \ \cdots \right\rangle_{env}$$

in the language SIMPLE *definition can be defined as*

$$\left\langle \frac{V = I;}{\epsilon_K} \ \curvearrowright \ k \right\rangle_K \left\langle [e_1, \ V \mapsto \frac{i}{I}, \ e_2] \right\rangle_{env}$$

For cells on the top level, we often omit cells that we do not touch.

3.2 Example of K

We briefly describe the K framework by using a simple example to show how we can define a language in K.

Figure 2.1 shows the definitions in K of the language `SIMPLE`. There are three parts we have to define: syntax, configuration, and rewrite rules.

Syntax of `SIMPLE`

For syntax definition, we define two new sorts which are `Exp`, expression, and `Stmt`, statement. `Exp` is formed by Int, Integer, or construct of plus, `Exp` + `Exp`. The plus construct is associated with `strict` attribute which means that its arguments must be evaluated before applying any rule to the construct. `Stmt` is formed by assignment from expression to variable name, Id, or is formed by sequencing of statements. In assignment construct, it is associated with `strict(2)` which means Exp terms must be evaluated before applying any rule to assignment construct. Additionally, the associativity is associated to the syntax definition, for sequencing statement we associate `left`, which means left associativity.

Initial configuration of `SIMPLE`

For the language `SIMPLE`, we define the initial configuration as:

$$\langle \$PGM : K \rangle_K \ \langle \epsilon_M \rangle_{env}.$$

```
module SIMPLE
   syntax Exp ::= Int
          | Exp "+" Exp  [strict]
   syntax Stmt ::= Id "=" Exp ";"   [strict(2)]
          | Stmt Stmt    [left]
   syntax KResult ::= Int

   configuration
      <k> $PGM:K </k>
      <env> .Map </env>

   rule I1:Int + I2:Int => I1 +Int I2
   rule  <k> V = I:Int ; => . ... </k>
         <env> ... V |-> (_ => I) ... </env>
   rule  <k> V = I:Int ; => . ... </k>
         <env> ... . => V |-> I ... </env>
   rule S1 S2 => S1 ~> S2
endmodule
```

Figure 3.1: Definition of the language SIMPLE in K

K cell contains a term (abstract syntax tree) of the input language, denoted by $\$PGM$. The env cell contains an empty map.

Rules of SIMPLE

In simple language, there are four rewrite rules which are corresponding to the following rewrite rules:

$$
\begin{array}{llll}
1: & \dfrac{I_1 + I_2}{I_1 +_{\texttt{Int}} I_2} & 3: & \left\langle \dfrac{V = I;}{\epsilon_k} \ \cdots \right\rangle_K \left\langle \cdots V \mapsto \dfrac{-}{I} \ \cdots \right\rangle_{env} \\
2: & \dfrac{S_1 \ S_2}{S_1 \curvearrowright S_2} & 4: & \left\langle \dfrac{V = I;}{\epsilon_k} \ \cdots \right\rangle_K \left\langle \cdots \dfrac{\epsilon_M}{V \mapsto I} \right\rangle_{env}
\end{array}
$$

The first rule says that the operator of plus with two integer arguments are rewritten to the primitive operator $+_{\texttt{Int}}$ for the addition (on integers), which is predefined function in the K framework. The second rule manages statements S_1 and S_2 to be ordered in terms of sort K. The third rule is an assignment rule which rewrites the assignment statement into empty unit of sort K and change the value in env map when map already has index V. The fourth rule is an assignment rule which rewrites the assignment statement into the empty unit of sort K and inserts a new pair of $V \mapsto I$ to env.

Suppose that we have a program `TEST` as below:

$$\$PGM \ \equiv \ \mathsf{x{=}3;y{=}5;\ x{=}3+5;}$$

If we run the input program using K interpreter then we get the following rewrite steps:

$$
\begin{array}{l}
\left\langle [\mathsf{x{=}3;y{=}5;x{=}3+5;}] \right\rangle_K \left\langle \epsilon_M \right\rangle_{env} \\
\rightarrow^+ \left\langle [\mathsf{y{=}5;x{=}3+5;}] \right\rangle_K \left\langle [x \mapsto 3] \right\rangle_{env} \\
\rightarrow^+ \left\langle [\mathsf{x{=}3+5;}] \right\rangle_K \left\langle [x \mapsto 3, y \mapsto 5] \right\rangle_{env} \\
\rightarrow^+ \left\langle [\mathsf{x{=}8;}] \right\rangle_K \left\langle [x \mapsto 3, y \mapsto 5] \right\rangle_{env} \\
\rightarrow^+ \left\langle \epsilon_L \right\rangle_K \left\langle [x \mapsto 8, y \mapsto 5] \right\rangle_{env}
\end{array}
$$

3.3 Support environment of the K framework

Based on formal semantics definition, the K framework provides analysis/verification tools [9] which are automatically derived as in figure 3.2.

Once we compile ("kompile" command) the definition in the K framework, it is translated into Maude in which analysis tools are prepared.

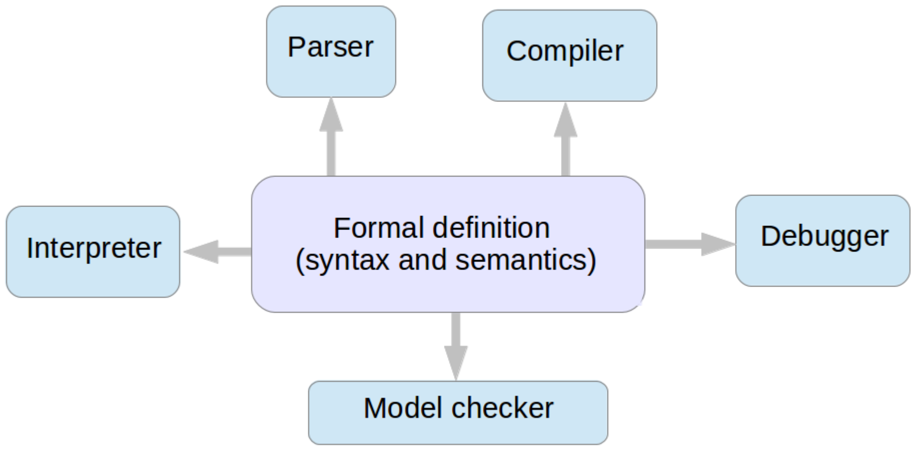

Figure 3.2: Analysis and verification tools [9]

Parser

The K framework uses SDF for parser generation. SDF generates the abstract-syntax tree from a grammar described in algebraic specification.

Interpreter

This is an immediate benefit of the language definition in the K framework. The K framework interprets K term by transforming it to Maude and Maude interprets it by rewriting.

Compiler

The K framework prepares a compiler written in Maude. It transforms K definition into Maude code. It has been replacing from Maude compiler to equivalent transformation in Java which is currently operating in some part of compilation.

Debugger

The K framework adopts Maude debugger by inserting a break point into K description. It is then translated into Maude code tagged with the break point. Maude debugger traces the execution and stop at each break point until the entire execution is done.

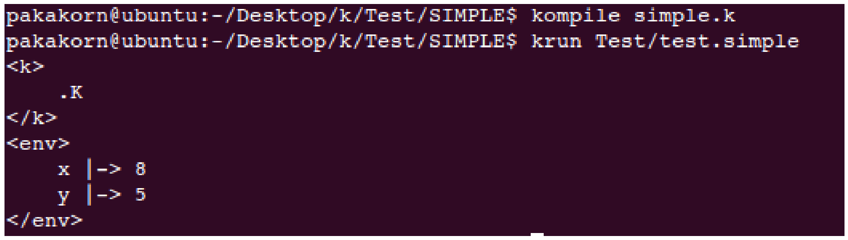

Figure 3.3: Result after running the program TEST with SIMPLE language

Model checker

The K framework supports for concurrent programming which can have non-deterministic behaviours. Maude provides search command to see all possible behaviours and the K framework makes use of it for model checking. The model checker in the K framework adopts linear temporal logic (LTL) benefited from a (model-checker) built-in provided by Maude.

For example, when we can compile this language SIMPLE by using K compiler and run the program by K interpreter, and then we get the following result as Figure 3.3.

Chapter 4

SQL semantics

4.1 SQL table operations and their treatments

In SQL, all data are stored in the tables. The following are basic elements.

- A *field*, *attribute*, and *column* refers to a datum.
- A *data type* of a field is a domain of values.
- A *record* is a composition of values.
- A *table* is a collection of records.
- A *database* is a set of tables.

Typically table operations include

- Selection: `SELECT` *fields* `FROM` *list_of_tables* `WHERE` *predicate* ;
  This statement defines which column to be retrieved in *fields*, which table in *list_of_tables*, and the condition for filtering in *predicate*.
- Creation: `CREATE TABLE` *table_name* (*list_of_column_definition*) ;
  This statement creates the table with names given by the *table_name* and the column definitions in *list_of_column_definition* with the data type, the column name, the key assignment, etc.
- Insertion: `INSERT INTO` *table_name* (*column_definition*) `VALUE` (*values*) ;
  This statement inserts a new record to the *table_name*. Such a new record has the fields as *list_of_column* and its values as *values*.

- Deletion: `DELETE FROM` *table_name* `WHERE` *predicate* ;
  This statement deletes elements in the table *table_name* such that the predicate *predicate* are satisfied.

- Update: `UPDATE` *table_name* `SET` *assignments* `WHERE` *predicate* ;
  This statement updates records of the table *table_name* with the assignment *assignments* that satisfied *predicate*.

Standard table operations among SQL dialects share similar core semantics described in a textbook except for variations of their syntax. However they vary in details, especially non-standard operations, such as:

- **(1) Treatment of `NULL` value** The treatment of `NULL` is one of the most important issues. The differences come from the meaning of this value. This different meanings bring confusion to the definition of semantics.

- **(2) Coercion among types** Coercion are implicitly conversion of types of arguments, e.g., $1 +$ `"1a"` requires the coercion from string to integer.

- **(3) Boolean data types** SQL dialects have different representation of the boolean data type. Some simply uses zero and non-zero like MySQL. Some omits the like Oracle11.

- **(4) Error handling** The ways of error handling can be the following: error constant, explicit error messages, and replacing with possible values. Normally, we can see error handling by printing error messages, but in real SQL dialects, they have their own specific purposes to use the error constant for error representation.

- **(5) Name space** When we want to refer to an specific object, such as a column name in SQL language, we have to identify the name space or path direct to such a column. The name space is designed differently among SQL dialects. This leads difference among SQL dialects.

Among these differences we focus on the coercion, `NULL`, and the name space by comparison between MySQL and Oracle11.

4.2 Coercion

Data type represents a set of values, e.g., 32-bit integer (denoted by `INT`) and text string (denoted by `TEXT`) in MySQL. SQL has types of arguments of operations. However, for flexibility, it converts types to fit an operation by the coercion. Coercion consists of rules to convert one type of an object to a new object/value with a different type.

Example 4.1. *MySQL executes the statement* `SELECT` $1 +$ `"1"`;. *The result of the this statement is* 2. *Basically, operator* $+$ *takes two arguments of integers, but MySQL extends the definition to cover other types. MySQL treats Int* $+$ *String as Int* $+$ *Int by applying implicit type conversion from String to Int, which converse* `"1"` *to* 1.

Here we show some possible choices of the coercion.
Coercion to boolean type

- Non-zero integers are converted to `TRUE` and 0 is converted to `FALSE`.
- Any string is converted to `FALSE`.
- A string is converted to an integer in some way, then converted to a boolean value.

Coercion to integer type

- `FALSE` is converted to 0 and `TRUE` is converted to 1.
- `FALSE` and `TRUE` are converted to *errors*.
- A integer string is converted to a concatenated integer; otherwise an *error*.
- A mixed string is converted to its maximum integer prefix; if it is empty, then 0.
- A string is converted to the concatenation of the ascii number of each element.

Coercion to string type

- A number is converted to an integer string.
- A number is converted to an *error*.

MySQL treats `TRUE` and `FALSE` as aliases of 1 and 0, respectively. We start with examples in Table 4.1 to observe the semantics definition of the operator + with the coercion For testing queries, we use the selection syntax `"SELECT` *Exp* `;"` in MySQL. However, Oracle11 does not allow the selection of boolean-valued expressions *BExp* as an argument of the column, we need to distinguish queries on *Bexp* such that `"SELECT` 1 `FROM DUAL WHERE` *BExp*;`"`. Table 4.1, 4.2, and 4.3 compare MySQL and Oracle11 on +, <=, and &&, respectively. Our observation is:

- In MySQL, `TRUE` is considered as 1 and `FALSE` is considered as 0.
- In MySQL, the coercion with “+” and ”<=” are different. “+” always tries to coerce arguments into integers. ”<=” changes its action depending on the types of arguments. For instance “*Int* <= *String*” coerces the string to the integer by taking the maximum number prefix. However, “*String* <= *String*” does not cause coercion; instead, simply overloading “<=” as the lexicographic extension of ascii numbers.

No.	*Exp* in MySQL	result (MySQL)	result (Oracle)
1	1 + 1 ;	2	2
2	1 + 2 ;	3	3
3	1 + `"1"` ;	2	2
4	`"1"` + 1 ;	2	2
5	`"1"` + `"1"` ;	2	2
6	`"1"` + `"2"` ;	3	3
7	`TRUE` + 1 ;	2	*error*
8	`FALSE` + 1 ;	1	*error*
9	`TRUE` + `FALSE` ;	1	*error*
10	0 + `"a"` ;	0	*error*
11	1 + `"a"` ;	1	*error*
12	0 + `"1a"` ;	1	*error*
13	1 + `"2a"` ;	3	*error*
14	1 + `"1a1"` ;	2	*error*
15	0 + `"-1"` ;	-1	-1
16	0 + `"-1a"` ;	-1	*error*
17	`"-1a"` + `"-1a"` ;	-2	*error*
18	`TRUE` + `"-1a"` ;	0	*error*

Table 4.1: Testing queries for coercion of + in MySQL and Oracle

- In Oracle11, we can see that all queries confirm the coercion from an integer to a string. However, the coercion can accept only numbered content and does not accept `TRUE` and `FALSE`.
- In Oracle11 all queries result errors. This confirms us that Oracle11 does not allow 1, 0, `TRUE`, and `FALSE` for the arguments.

Our hypothesis on their semantics design policy is,

- MySQL adopts Zero and Non-zero to represent `FALSE` and `TRUE`.
- In MySQL, the coercion from String to Integer types will return the maximum prefix of numbers; if it is empty, it returns 0.
- In Oracle11, boolean primitive data are kept implicit values such that they are encapsulated for users. Users can use boolean expressions only in conditional expressions and there is no coercion from other types to the boolean type. Oracle11 allows the coercion from the string to the integer types, only when a string is an integer string.

No.	*Exp* in MySQL	result (MySQL)	result (Oracle)
1	1 <= 1 ;	1	1
2	1 <= 0 ;	0	*emp tbl*
3	1 <= ”1” ;	1	1
4	`"1"` <= 1 ;	1	1
5	1 <= `"0"` ;	0	*emp tbl*
6	1 <= `"a"` ;	0	*error*
7	1 <= `"1a"` ;	1	*error*
8	`TRUE` <= 0 ;	0	*error*
9	`TRUE` <= 1 ;	1	*error*
10	2 <= `TRUE` ;	0	*error*
11	`FALSE` <= 1 ;	1	*error*
12	`TRUE` <= `FALSE` ;	0	*error*
13	`"1"` <= `"2"` ;	1	1
14	`"2"` <= `"1"` ;	0	*emp tbl*
15	`"-1"` <= `"1"` ;	1	1
16	`"-2"` <= `"-1"` ;	0	*emp tbl*
17	`"a"` <= `"b"` ;	1	1
18	`"b"` <= `"a"` ;	0	*emp tbl*
19	`"a"` <= `"ab"` ;	1	1
20	`"bb"` <= `"ac"` ;	0	*emp tbl*
21	`"-"` <= `"-2"` ;	1	1
22	`"+1"` <= `"1"` ;	1	1
23	`"+2"` <= `"1"` ;	1	1

Table 4.2: Testing queries for coercion of $\leq$ in MySQL and Oracle

No.	*Exp* in MySQL	result (MySQL)	result (Oracle)
1	1 && 1 ;	1	*error*
2	1 && 0 ;	0	*error*
3	1 && `"1"` ;	1	*error*
4	`"1"` && 1 ;	1	*error*
5	`"1"` && `"1"` ;	1	*error*
6	`"1"` && `"0"` ;	0	*error*
7	`TRUE` && 1 ;	1	*error*
8	`FALSE` && 1 ;	0	*error*
9	`TRUE` && `FALSE` ;	0	*error*
10	1 && `"a"` ;	0	*error*
11	1 && `"1a"` ;	1	*error*
12	1 && `"1a1"` ;	1	*error*
13	`"a"` && `"a"` ;	0	*error*

Table 4.3: Testing queries for coercion of && in MySQL and Oracle11

4.3 Interpretation of `NULL` value

SQL has a special value `NULL`, which is interprets differently among SQL dialects. There are choices on handling `NULL`:

- `NULL` is an undefined (unknown) value.
- `NULL` is an error.
- `NULL` is the empty string.
- `NULL` is `FALSE`.

For example, consider queries in Table 4.4.

`Student`

No.	Name
1	`NULL`
2	`Kim`
3	`Few`
4	`Nat`

When we execute `SELECT * FROM Student`, we cannot decide what is an answer for the row containing `NULL` value. One might have a question that what name of the student

No.	*Exp* in MySQL	result (MySQL)	result (Oracle)
1	1 + 1 ;	2	2
2	1 + `NULL` ;	`NULL`	`''`
3	`NULL` + 1 ;	`NULL`	`''`
4	`NULL` + `NULL` ;	`NULL`	`''`
5	`NULL` || `TRUE` ;	1	*error*
6	`TRUE` || `NULL` ;	1	*error*
7	`NULL` && `FALSE` ;	0	*error*
8	`FALSE` && `NULL` ;	0	*error*
9	`concat('1',NULL)` ;	`NULL`	`'1'`

Table 4.4: Testing queries for `NULL` (undefined) treatment in MySQL and Oracle

No.	*Exp* in MySQL	result (MySQL)	result (Oracle)
1	`NULL` ;	`NULL`	`''`
2	`CONCAT(NULL,1)` ;	`NULL`	`'1'`
3	`CONCAT(NULL,"1")` ;	`NULL`	`'1'`
4	`NULL IS NULL` ; (*BExp*)	1	1
5	`"" IS NULL` ; (*BExp*)	0	1

Table 4.5: Testing queries for `NULL` in string expressions in MySQL and Oracle11

number one is. If no answers, one might have a question that why the table has the number one.

MySQL regards `NULL` as:

- `NULL` means an unknown or undefined value.
- `NULL` means an error.

whereas Oracle11 regards it as the empty string. We observe them by examples. The expression of operator +, ||, &&, and `concat` (`OR`, `AND`, and || in Oracle11) to investigate how MySQL and Oracle11 treat (in table 4.4).

In MySQL, we observe that + is strict on `NULL` as well as other arithmetic and comparison operators. For boolean operations with `NULL`, MySQL treats it like Kleene's three value logic. They are shown in the queries 5-9 in table 4.4. In Oracle11, `NULL` is considered as the empty string. The query 9 in Table 4.4 and 4.5 show the contrast.

For MySQL, the queries in Table 4.5 show that even `NULL` in string behaves as an unknown value and is not equal to the empty string. As `NULL` is also used as failure of the

No.	*Exp* in MySQL	result (MySQL)	result (Oracle)
1	1 / 0 ;	`NULL`	*error*
2	1 % 0 ;	`NULL`	1

Table 4.6: Testing queries for `NULL` as errors in MySQL and Oracle11

No.	Query	result
1	`SELECT 1 FROM T JOIN T`	*error*
2	`SELECT 1 FROM T JOIN T AS T2`	1

Table 4.7: Testing queries for treatment conflict name space in MySQL

evaluation in Table 4.6. In Oracle11, an error is shown by the error messages ("*error*") and `NULL` is the empty string. Thus, 1 + `NULL` contains the coercion on 1 from *Int* to (null) string where MySQL treats it as the sum with an undefined value (then `NULL` is returned).

The execution of 1/0 and 1%0 (zero-divisor) cause errors, but MySQL returns `NULL` as a value, while Oracle11 returns an error message.

4.4 Name space

MySQL and Oracle11 have the name space depending on how to handle table names. For instance, an operation with the same table, like the self join and product, how to distinguish references to elements in two tables with the same name. Possible choices of the name space are:

- Name conflict is not allowed.
- Identifying an object by prefixes of the database name and the table name (as in MySQL).
- Identifying an object by prefixes of the account name and the table name (as in Oracle).
- Identifying an object by prefixes of the database name, the schema name, and the table name (as in PostgreSQL).

These designs of the name space are to make identification of each object unique.

We observe the behaviour in MySQL in Table 4.7. The query 1 shows that both MySQL and Oracle do not accept self join operation due to the ambiguity of names. However, they solve such a problem by allowing user to give alias as in the query 2.

T1	
A_{INT}	B_{TEXT}
1	"a"
2	"b"

T2	
A_{INT}	B_{INT}
1	1
2	0

What if we execute `SELECT A FROM T1 JOIN T2`? This query causes an error because both MySQL and Oracle11 cannot identify the attribute `A` whether it is in `T1` or `T2`. We can make the attribute `A` clear by providing a name space as `SELECT T1.A FROM T1 JOIN T2` or `SELECT D.T1.A FROM T1 JOIN T2`, where `D` for a database name. We call such multiple parts of an identifier as a qualifier. Oracle11 does similarly but with a different name space, `SELECT T1.A FROM T1 JOIN T2` or `SELECT U.T1.A FROM T1 JOIN T2`, where `U` for a username.

From our observation, we expect that MySQL accesses one object by identifying the database name and the object name, like `DatabaseName.TableName`. In contrast, Oracle11 accesses one object by identifying the user name and the object name, like `Owner.TableName`. PostgreSQL does with the database name, the schema name, and the table name, like `DatabaseName.SchemaName.TableName`.

Chapter 5

KSQL description of standard table operations

We present our formal description of database systems on K-framework, which we named *KSQL*. Typical SQL queries are creation, update, and retrieval of tables in a database. First we formalize tables. In KSQL, we prepare three sorts, *Int*, *Bool*, and *String*. *Int* literal value represents an integer. *Bool* values are represented by zero and one. Our semantics stays at the logical level and ignores the bit-length at the physical level. Here we have the subsort relation $Bool \sqsubseteq Int$. *String* values are represented by starting and ending with double quote ('`"`'), which contain text in between. We simply call *fields* for terms of sort *Field*, *values* for those of *Val*, *data types* for those of *DataType*, and *(table) identifiers* for terms of *Id*. In KSQL, we omit the physical level variations of data types, e.g., Int-16 and Int-32, and then we denote `INT` for the integer data type and `TEXT` for the string data type.

5.1 Syntax

Definition 5.1. *A* Field element *is a tuple of a field and a data type. A* Schema *is a list of field elements, and a* record *is a tuple of* values*. We denote a set of schemas by Schema and a set of lists of records by Record. We define a* table *as the triple* (T, S, R) *of an identifier* T*, a schema* S*, and a list* R *of records, denoted by* $T[S : R]$*.*

Example 5.2. *Consider the two tables* $\texttt{T1}[S_1 : R_1]$ *and* $\texttt{T2}[S_2 : R_2]$ *with*

$$S_1 = [(\texttt{A}, \textit{INT}), (\texttt{B}, \textit{TEXT})] \qquad S_2 = [(\texttt{A}, \textit{INT}), (\texttt{B}, \textit{INT})]$$

$$R_1 = [\,(1, \texttt{"a"}), (2, \texttt{"b"}), (3, \texttt{"c"})\,] \qquad R_2 = [\,(1, \texttt{TRUE}), (2, \texttt{TRUE}), (2, \texttt{FALSE})\,]$$

Note that TRUE *and* FALSE *are aliases of* 1 *and* 0*, respectively. These tables are displayed as follows:*

T1

$\texttt{A}_{INT}$	$\texttt{B}_{TEXT}$
1	"a"
2	"b"
3	"c"

T2

$\texttt{A}_{INT}$	$\texttt{C}_{INT}$
1	TRUE
2	TRUE
3	FALSE

A database is a set of tables. Additionally, SQL supports *aliases* for table identifiers in order to solve the name conflict problem.

We formalize the core part of syntax for SQL queries.

Definition 5.3. *Queries or statements in KSQL are defined as follows:*

$$\begin{aligned} \mathit{Query} \ ::= \ & \texttt{CREATE TABLE}\ \mathit{Id}\ \texttt{(}\ \mathit{FieldDcl}{*}\ \texttt{)}\ ; \\ \mid\ & \texttt{INSERT INTO}\ \mathit{Id}\ \texttt{(}\ \mathit{Field}{*}\ \texttt{)}\ \texttt{VALUES}\ \texttt{(}\ \texttt{Val}{*}\ \texttt{)}\ ; \\ \mid\ & \texttt{SELECT}\ \mathit{ProjectionExp}\ \texttt{FROM}\ \mathit{Id}\ \texttt{WHERE}\ \mathit{Exp}\ ; \\ \mid\ & \mathit{Query}\ \mathit{Query} \end{aligned}$$

In the creation of table semantics of KSQL, we omit data type checking of values, which will be done at the instruction level. We assume that expression Exp and ProjectionExp are defined as follows:

$$\begin{aligned} \mathit{Exp} &::= \mathit{Id} \mid \mathit{Int} \mid \mathit{String} \mid \mathit{Exp} \circ \mathit{Exp} \\ \mathit{ProjectionExp} &::= * \ \mid\ \mathit{Field}\ (,\ \mathit{Field}){*} \\ \mathit{FieldDcl} &::= \mathit{Field}\ \mathit{DataType} \\ \mathit{DataType} &::= \mathtt{INT} \ \mid\ \mathtt{TEXT} \end{aligned}$$

where $\circ \in \{+, -, =, <\}$. *Field* (, *Field*)∗ *stands for a non-empty list.*

As notational conversion, we use T for a table identifier, n for a natural number, S and S' for schemas, R for a list of records, r for a record, and f for a field. We start with the syntax of `store` definition, which is a function used in defining the semantics of `CREATE`.

Definition 5.4. *The syntax of* `store` *is given by the next grammar:*

$$K ::= \ldots \mid \texttt{store}\ \mathit{Table}$$

We define auxiliary functions `doGetTable`, `doCondition`, and `doProjection`, which are used in defining the semantics of `SELECT`.

Definition 5.5. *The syntax of* `doGetTable`, `doCondition`, *and* `doProjection` *are given by:*

$$
\begin{array}{rcl}
\mathit{Table} & ::= & \texttt{doGetTable(}\ \mathit{Id}\ \texttt{)} \\
 & | & \texttt{doCondition(}\ \mathit{Table}\ ,\ \mathit{Exp}\ \texttt{)} \\
 & | & \texttt{doProjection(}\ \mathit{Table}\ ,\ \mathit{ProjectionExp}\ \texttt{)}
\end{array}
$$

5.2 Configuration

Definition 5.6. *Let $\bar{S}$ be a set of schemas, and let $\bar{R}$ be a set of lists of records. A* (database) configuration *is a term of the form:*

$$\mathit{Configuration} ::= \Big\langle K \Big\rangle_{K} \Big\langle M_e \Big\rangle_{env} \Big\langle M_s \Big\rangle_{schema} \Big\langle M_r \Big\rangle_{records} \Big\langle Nat \Big\rangle_{loc}$$

where $M_e : Id \to Nat, M_s : Nat \to \bar{S}$, and $M_r : Nat \to \bar{R}$.

Example 5.7 (Continued from Example 5.2)**.** *We define a configuration for the database consisting of the tables* `T1` *and* `T2` *as below:*

$$
\begin{cases}
\Big\langle \epsilon_K \Big\rangle_{K} \\
\Big\langle [\texttt{T1} \mapsto 0, \texttt{T2} \mapsto 1] \Big\rangle_{env} \\
\Big\langle [0 \mapsto [(\texttt{A}, \textit{\texttt{INT}}), (\texttt{B}, \textit{\texttt{TEXT}})], 1 \mapsto [(\texttt{A}, \textit{\texttt{INT}}), (\texttt{C}, \textit{\texttt{INT}})]] \Big\rangle_{schema} \\
\Big\langle [0 \mapsto [(1, \texttt{"a"}), (2, \texttt{"b"}), (3, \texttt{"c"})], 1 \mapsto [(1, \texttt{1}), (2, \texttt{1}), (3, \texttt{0})]] \Big\rangle_{records} \\
\Big\langle 2 \Big\rangle_{loc}
\end{cases}
$$

5.3 Rewrite rules

Before we define the semantics of the table creation, we need to define the interpretation of `store`. The `store` function moves a table content from a cell of K into the cells of *env*, *schema*, and *records* in the configuration.

Definition 5.8. *The semantics of* `store` *is given by the following two rules:*

$$1: \left\langle \frac{\texttt{store}\ T[S:R]}{\epsilon_K}\ \cdots \right\rangle_K \left\langle \cdots\ T \mapsto n\ \cdots \right\rangle_{env} \left\langle \cdots\ n \mapsto \frac{-}{S}\ \cdots \right\rangle_{schema} \left\langle \cdots\ n \mapsto \frac{-}{R}\ \cdots \right\rangle_{records}$$

$$2: \left\langle \frac{\texttt{store}\ T[S:R]}{\epsilon_K}\ \cdots \right\rangle_K \left\langle \cdots\ \frac{\epsilon_M}{T \mapsto n} \right\rangle_{env} \left\langle \cdots\ \frac{\epsilon_M}{n \mapsto S} \right\rangle_{schema} \left\langle \cdots\ \frac{\epsilon_M}{n \mapsto R} \right\rangle_{records} \left\langle n \mapsto n+1 \right\rangle_{loc}$$

There are two rewrite rules for the `store` evaluation. The first rule is applied when we already have an identifier T in the *env* cell. The second rule is applied when we have no identifier T in the *env* cell. We will use `store` to define the semantics of `CREATE`. The `CREATE` query is used to create a table together with the declaration of a list of fields.

Definition 5.9. *The evaluation of* `CREATE` *is defined as follows:*

$$\left\langle \frac{\texttt{CREATE TABLE}\ T(Fd)\ ;}{T[\texttt{createSchema}(Fd) : \epsilon_L\ ] \curvearrowright \texttt{store}\ \Diamond}\ \cdots \right\rangle_K$$

where `createSchema` *is a function defined as below:*

$$\texttt{createSchema}(Fd) = [f \mid f\ d \in Fd \textit{ for some data type } d]$$

Here we use the comprehension notation $[\cdots \mid \cdots]$. The K framework does not support it but we can easily translate such a notation to corresponding recursive definitions in the K framework. We briefly explain the evaluation of the table creation. The evaluation of the table creation creates a structure of the table T containing fields from a list Fd of field declarations, and then use `store` to store the table in the configuration. An insertion inserts a new record to the configuration.

Definition 5.10. *The evaluation of* `INSERT` *is defined as follows:*

$$\left\langle \frac{\texttt{INSERT INTO}\ T(S)\ \texttt{VALUES}(Vs);}{\epsilon_K}\ \cdots \right\rangle_K \left\langle \cdots L \mapsto \frac{[R]}{[[Vs], R]} \cdots \right\rangle_{records} \left\langle \cdots T \mapsto L \cdots \right\rangle_{env} \left\langle \cdots L \mapsto S \cdots \right\rangle_{schema}$$

where we assume that list Vs of values and the schema S have the same number of elements.

When inserting a new record, KSQL first find the location in the *env* cell and then store a schema and a new record of values in *schema* and *records* cells respectively.

The selection is the most complicated. As its syntax indicates, it consists of three ingredients. It begins with the table retrieval part, followed by the condition and the projection part. Their semantics is defined by auxiliary functions `doGetTable`, `doCondition`, and `doProjection`.

Definition 5.11. *The semantics of* `doGetTable` *is defined as follows:*

$$\left\langle \frac{\texttt{doGetTable}(T)}{T[\ S : R\ ]} \ \cdots \right\rangle_K$$
$$\left\langle \cdots \ T \mapsto L \ \cdots \right\rangle_{env}$$
$$\left\langle \cdots \ L \mapsto S \ \cdots \right\rangle_{schema}$$
$$\left\langle \cdots \ L \mapsto R \ \cdots \right\rangle_{records}.$$

Before we define `doCondition`, we need two more auxiliary functions `eval` and `filter`. The function `eval` is to evaluate an expression using data in a record, which is specific to a schema. The function `filter` is to filter a list of records, which satisfies the condition for the schema .

Definition 5.12. *For an identifier I, integers n and m, a string s, and a value v, we inductively define the* `eval` *function as follows:*

$$\texttt{eval} : \mathit{Schema} \times \mathit{Record} \times \mathit{Exp} \to \mathit{Exp}$$

$$\begin{aligned}
\texttt{eval}(S, r, I) &= (S \otimes r)(I) \\
\texttt{eval}(S, r, n) &= n \\
\texttt{eval}(S, r, s) &= s \\
\texttt{eval}(S, r, n + m) &= \texttt{eval}(S, r, n) + \texttt{eval}(S, r, m) \\
\texttt{eval}(S, r, n - m) &= \texttt{eval}(S, r, n) - \texttt{eval}(S, r, m) \\
\texttt{eval}(S, r, n = m) &= \texttt{eval}(S, r, n) = \texttt{eval}(S, r, m) \\
\texttt{eval}(S, r, n < m) &= \texttt{eval}(S, r, n) < \texttt{eval}(S, r, m)
\end{aligned}$$

Definition 5.13. *The definition of* `filter` *for a schema S, a list R of records, and an expression E is defined by:*

$$\begin{aligned}
\texttt{filter} :&\ \mathit{Schema} \times \mathit{Records} \times \mathit{Exp} \to \mathit{Records} \\
\texttt{filter}(S, R, E) &= [\ r \in R \mid \texttt{eval}(S, r, E) = 1\ ]
\end{aligned}$$

Definition 5.14. *The semantics of* `doCondition` *is defined as follows:*

$$\left\langle \frac{\texttt{doCondition}(T[\ S : R\ ], E)}{T[\ S : \texttt{filter}(S, R, E)\ ]}\ \cdots \right\rangle_K$$

The last auxiliary function is `doProjection`, and we need one more auxiliary function `project`, which extracts attributes that we need.

Definition 5.15. *The definition of* `project`*, which generates a new schema S' from a schema S and a list R of records, is as follows:*

$$\begin{aligned} \texttt{project} :&\ \ \textit{Schema} \times \textit{Records} \times \textit{Schema} \to \textit{Records} \\ \texttt{project}(S, R, S') =&\ \ [[(S \otimes r)(f) \mid f \in S'] \mid r \in R] \end{aligned}$$

Definition 5.16. *The semantics of* `doProjection` *is defined as follows:*

$$1 :\ \left\langle \frac{\texttt{doProjection}(T[\ S : R\ ], *)}{T[\ S : R\ ]}\ \cdots \right\rangle_K \quad 2 :\ \left\langle \frac{\texttt{doProjection}(T[\ S : R\ ], S')}{\texttt{project}(S, R, S')}\ \cdots \right\rangle_K$$

The first rule contains $*$ (which means all of attributes) and returns a table without changes. The second rule projects only needed attributes from the table, and generates S'.

The selection consists of (1) get the table content from the table name (`doGetTable`), (2) filter its records that satisfy the conditions (`doCondition`), and (3) project needed attributes (`doProjection`).

Definition 5.17. *The semantics of* `SELECT` *is defined as follows:*

$$\left\langle \frac{\texttt{SELECT}\ P\ \texttt{FROM}\ T\ \texttt{WHERE}\ E\ ;}{\texttt{doGetTable}(T)\ \curvearrowright\ \texttt{doCondition}(\Diamond, E)\ \curvearrowright\ \texttt{doProjection}(\Diamond, P)}\ \cdots \right\rangle_K$$

In addition to the definition above, we need structural rules. Each of which instantiates a value to $\Diamond$ and changes the sequence of queries into the list of K terms.

Definition 5.18. *The structural rules for the instantiation of a table to $\Diamond$ are defined as follows:*

$$\left\langle \frac{T[S : R] \curvearrowright \texttt{store}\ \Diamond}{\texttt{store}\ T[S : R]}\ \cdots \right\rangle_K$$

$$\left\langle \frac{T[S : R] \curvearrowright \texttt{doCondition}(\Diamond, E)}{\texttt{doCondition}(T[S : R], E)}\ \cdots \right\rangle_K$$

$$\left\langle \frac{T[S : R] \curvearrowright \texttt{doProjection}(\Diamond, P)}{\texttt{doProjection}(T[S : R], P)}\ \cdots \right\rangle_K$$

Definition 5.19. *The structural rule to transform a term of a sequence of queries into a list of K terms is defined as follows:*

$$\left\langle \frac{Q_1\ Q_2}{Q_1 \curvearrowright Q_2} \ \cdots \right\rangle_K$$

The rule above gets a term formed by Q_1 and Q_2 and then rewrite it into a K term $Q_1 \curvearrowright Q_2$. The K framework will try to rewrite the first argument of K terms until it reaches normal form, then it unifies the first term to the next term.

Example 5.20. *Let $Q = S_1, \ldots, S_5$ with for convenience, we first introduce statement variables $S_1, \ldots, S_5$ where*

$$\begin{aligned}
S_1 &= \texttt{CREATE TABLE T1}(\texttt{A } \textit{INT}, \texttt{B } \textit{TEXT}) \ ; \\
S_2 &= \texttt{INSERT INTO T1}(\texttt{A}, \texttt{B}) \ \texttt{VALUES}(1, \texttt{"a"}) \ ; \\
S_3 &= \texttt{INSERT INTO T1}(\texttt{A}, \texttt{B}) \ \texttt{VALUES}(2, \texttt{"b"}) \ ; \\
S_4 &= \texttt{INSERT INTO T1}(\texttt{A}, \texttt{B}) \ \texttt{VALUES}(3, \texttt{"c"}) \ ; \\
S_5 &= \texttt{SELECT} \ * \ \texttt{FROM T1 WHERE A} > 1 \ ;
\end{aligned}$$

The query Q is evaluated as follows:

$$\left\langle S_1\ S_2\ S_3\ S_4\ S_5 \right\rangle_K C_1 \quad \textit{where } C_1 = \left\langle \epsilon_M \right\rangle_{env} \left\langle \epsilon_M \right\rangle_{schema} \left\langle \epsilon_M \right\rangle_{records} \left\langle 0 \right\rangle_{loc}$$

$$\to^* \left\langle S_1 \curvearrowright S_2 \curvearrowright S_3 \curvearrowright S_4 \curvearrowright S_5 \right\rangle_K C_1$$

$$\to^* \left\langle \mathtt{T_1} \curvearrowright \textit{store } \Diamond \curvearrowright S_2 \curvearrowright S_3 \curvearrowright S_4 \curvearrowright S_5 \right\rangle_K C_1$$

$$\to^* \left\langle \textit{store } \mathtt{T_1} \curvearrowright S_2 \curvearrowright S_3 \curvearrowright S_4 \curvearrowright S_5 \right\rangle_K C_1$$

$$\to^* \left\langle S_2 \curvearrowright S_3 \curvearrowright S_4 \curvearrowright S_5 \right\rangle_K C_2 \quad \textit{where } C_2 = D \left\langle 0 \mapsto \epsilon_L \right\rangle_{records}$$

$$\textit{and} \quad D = \left\langle [\mathtt{T1} \mapsto 0] \right\rangle_{env} \left\langle 0 \mapsto [(\mathtt{A}, \textit{INT}), (\mathtt{B}, \textit{TEXT})] \right\rangle_{schema} \left\langle 1 \right\rangle_{loc}$$

$$\to^* \left\langle S_3 \curvearrowright S_4 \curvearrowright S_5 \right\rangle_K C_3 \quad \textit{where } C_3 = D \left\langle 0 \mapsto [(1, \texttt{"a"})] \right\rangle_{records}$$

$$\to^* \left\langle S_4 \curvearrowright S_5 \right\rangle_K C_4 \quad \textit{where } C_4 = D \left\langle 0 \mapsto [(1, \texttt{"a"}), (2, \texttt{"b"})] \right\rangle_{records}$$

$$\to^* \left\langle S_5 \right\rangle_K C_5 \quad \textit{where } C_5 = D \left\langle 0 \mapsto [(1, \texttt{"a"}), (2, \texttt{"b"}), (3, \texttt{"c"})] \right\rangle_{records}$$

$$\to^* \left\langle \mathtt{doGetTable}(\mathtt{T1}) \curvearrowright \mathtt{doCondition}(\Diamond, \mathtt{A} > 1) \curvearrowright \mathtt{doProjection}(\Diamond, *) \right\rangle_K C_5$$

$$\to^* \left\langle \mathtt{T_2} \curvearrowright \mathtt{doCondition}(\Diamond, \mathtt{A} > 1) \curvearrowright \mathtt{doProjection}(\Diamond, *) \right\rangle_K C_5$$

$$\to^* \left\langle \mathtt{doCondition}(\mathtt{T_2}, \mathtt{A} > 1) \curvearrowright \mathtt{doProjection}(\Diamond, *) \right\rangle_K C_5$$

$$\to^* \left\langle \mathtt{T_2} \curvearrowright \mathtt{doProjection}(\Diamond, *) \right\rangle_K C_5$$

$$\to^* \left\langle \mathtt{T_3} \curvearrowright \mathtt{doProjection}(\Diamond, *) \right\rangle_K C_5$$

$$\to^* \left\langle \mathtt{doProjection}(T_3, *) \right\rangle_K C_5$$

$$\to^* \left\langle T_3 \right\rangle_K C_5$$

$\mathtt{T_1}$

$\mathtt{A}_{INT}$	$\mathtt{B}_{TEXT}$

$\mathtt{T_2}$

$\mathtt{A}_{INT}$	$\mathtt{B}_{TEXT}$
1	"a"
2	"b"
3	"c"

$\mathtt{T_3}$

$\mathtt{A}_{INT}$	$\mathtt{B}_{TEXT}$
2	"b"
3	"c"

Chapter 6

KSQL descriptions of coercion, NULL, and name space in MySQL

6.1 Coercion in arithmetic and boolean operations

We define the semantics of the coercion in the K framework as observed in the section 4. First we recall the structure of expressions.

$$\mathit{Exp} ::= \cdots \;\mid\; \mathit{Exp} \;\circ\; \mathit{Exp} \;\mid\; \texttt{NOT}\; \mathit{Exp}$$

where $\circ \in \{+, -, *, /, \%, =, >=, >, <=, <, != , ||, \&\&\}$ and *Exp* is an expression, which refers to an integer or a string value.

The K framework provides functions that take one literal value and convert it from some type to the string and vice versa.

Definition 6.1. *The K framework provides functions for types conversion as follow:*

$$\begin{aligned} \texttt{tokenToString} &: \mathit{Token} \to \mathit{String} \\ \texttt{parseToken} &: \tau \times \mathit{String} \to \mathit{Token} \end{aligned}$$

where Token is a set of literal values (string and integer). The function `tokenToString` *converts the value into the corresponded string and* `parseToken` *returns the literal value corresponds to the sort τ such that its value is described in the String.*

By combining the two functions above, we construct a new function as follows:

$$\mathcal{C} \;:\; \tau_1 \times \tau_2 \times \mathit{Token}_{\tau_1} \to \mathit{Token}_{\tau_2}$$

which convert the value v of the sort τ_1, corresponding to the value of the sort τ_2. We also denote $\mathcal{C}_{\tau_1 \to \tau_2}(v)$ for $\mathcal{C}(\tau_1, \tau_2, v)$. We call $\mathcal{C}$ a coercion function *and we denote the coercion function for an operator f by $\mathcal{C}^f$.*

In MySQL, `TRUE` and `FALSE` are aliases of 1 and 0, respectively.

Definition 6.2. *The coercion function from Int to Bool is defined as follows:*

$$\mathcal{C}^{\circ}_{Int \to Bool}(i) = \begin{cases} 0 \ \textit{if } i = 0 \\ 1 \ \textit{otherwise} \end{cases}$$

where i is an integer.

Conversion from *String* to *Int* is more complicated.

Definition 6.3. *The coercion function from String to Int is defined as follows:*

$$\mathcal{C}^{\circ}_{String \to Int}(s) = \begin{cases} n \ \textit{if } s \textit{ is } \overbrace{[+-]?[0-9]+}^{n}\texttt{\^{}}[0-9]?(.*) \\ 0 \ \textit{otherwise} \end{cases}$$

where n is an integer.

Now, we define the semantics of arithmetic and boolean operators with the coercion.

Definition 6.4. *Let i, i_1, and i_2 be integers, and let s, s_1, and s_2 be strings. The semantics of the arithmetic operator $\circ$ is defined as:*

$$1 : \frac{i_1 \circ i_2}{m} \qquad 2 : \frac{i \circ s}{i \circ \mathcal{C}^{\circ}_{String \to Int}(s)}$$

$$3 : \frac{s \circ i}{\mathcal{C}^{\circ}_{String \to Int}(s) \circ i} \qquad 4 : \frac{s_1 \circ s_2}{\mathcal{C}^{\circ}_{String \to Int}(s_1) \circ \mathcal{C}^{\circ}_{String \to Int}(s_2)}$$

where $\circ \in \{+, -, *, /, \%\}$ and m is the value after execution of the standard arithmetic on integers i_1 and i_2.

Definition 6.5. *Let b, b_1, and b_2 be boolean values, let i, i_1, and i_2 be integers, and let s, s_1, and s_2 be strings. The semantics of the logical operator $\circ$ is defined as:*

$$1 : \frac{b_1 \circ b_2}{m} \qquad 2 : \frac{b \circ i}{b \circ \mathcal{C}^{\circ}_{Int \to Bool}(i)}$$

$$3 : \frac{i \circ b}{\mathcal{C}^{\circ}_{Int \to Bool}(i) \circ b} \qquad 4 : \frac{i_1 \circ i_2}{\mathcal{C}^{\circ}_{Int \to Bool}(i_1) \circ \mathcal{C}^{\circ}_{Int \to Bool}(i_2)}$$

$$7 : \frac{i \circ s}{i \circ \mathcal{C}^{\circ}_{String \to Int}(s)} \qquad 5 : \frac{s \circ i}{\mathcal{C}^{\circ}_{String \to Int}(s) \circ i}$$

$$7 : \frac{s_1 \circ s_2}{\mathcal{C}^{\circ}_{String \to Int}(s_1) \circ \mathcal{C}^{\circ}_{String \to Int}(s_2)}$$

where $\circ \in \{\&\&, ||\}$ and m is the value after execution of the standard boolean operation on b_1 and b_2.

Next we formalize the semantics of comparison operators.

Definition 6.6. *Let i, i_1, and i_2 be integers and let s, s_1, and s_2 be strings. The semantics of comparison operator $\diamond$ is defined as follows:*

$$\frac{i_1 \ \diamond \ i_2}{m} \qquad \frac{i \ \diamond \ s}{i \ \diamond \ \mathcal{C}^{\diamond}_{String \to Int}(s)}$$

$$\frac{s \ \diamond \ i}{\mathcal{C}^{\diamond}_{String \to Int}(s) \ \diamond \ i} \qquad \frac{s_1 \ \diamond \ s_2}{s_1 \ \diamond_{lex} \ s_2}$$

where $\diamond \in \{=, <, \leq, >, \geq, \neq\}$, m is 1 if $i_1 \diamond i_2$ holds; 0, otherwise, and $\diamond_{\mathtt{lex}}$ is the string comparison corresponds to the lexicographical ordering on strings.

The type coercion function $\mathcal{C}^{\diamond}_{String \to Int}$ is defined as the same as $\mathcal{C}^{\circ}_{String \to Int}$. Table 6.1, 6.2, and 6.3 show testing results of KSQL and MySQL.

6.2 Treatment of NULL value

Definition 6.7. `NULL` *is a* constant *of the sort Val.*

As we have observed behaviour of `NULL` in Section 4. The `NULL` is of the bottom data type, and `NULL` follows to strict semantics for functions (except for boolean operations).

Definition 6.8. *For each operator $f : \tau_1 \times \cdots \times \tau_{\mathtt{NULL}} \times \cdots \times \tau_n \to \tau$, the semantics of* `NULL` *is defined as follows:*

$$\frac{f(e_1, \ldots, \mathtt{NULL}, \ldots, e_n)}{\mathtt{NULL}},$$

where $e_1, \ldots, e_n$ are expressions and $\tau_{\mathtt{NULL}}$.

No.	query	result (MySQL)	result (KSQL)
1	`SELECT` 1 + 1 ;	2	2
2	`SELECT` 1 + 2 ;	3	3
3	`SELECT` 1 + `"1"` ;	2	2
4	`SELECT` `"1"` + 1 ;	2	2
5	`SELECT` `"1"` + `"1"` ;	2	2
6	`SELECT` `"1"` + `"2"` ;	3	3
7	`SELECT` `TRUE` + 1 ;	2	2
8	`SELECT` `FALSE` + 1 ;	1	1
9	`SELECT` `TRUE` + `FALSE` ;	1	1
10	`SELECT` 0 + `"a"` ;	0	0
11	`SELECT` 1 + `"a"` ;	1	1
12	`SELECT` 0 + `"1a"` ;	1	1
13	`SELECT` 1 + `"2a"` ;	3	3
14	`SELECT` 1 + `"1a1"` ;	2	2
15	`SELECT` 0 + `"-1"` ;	-1	-1
16	`SELECT` 0 + `"-1a"` ;	-1	-1
17	`SELECT` `"-1a"` + `"-1a"` ;	-2	-2
18	`SELECT` `TRUE` + `"-1a"` ;	0	0

Table 6.1: Comparison of the coercion of + between MySQL and KSQL

Number	Query	result (MySQL)	result (KSQL)
1	`SELECT` 1 <= 1 ;	1	1
2	`SELECT` 1 <= 0 ;	0	0
3	`SELECT` 1 <= ”1” ;	1	1
4	`SELECT "1"` <= 1 ;	1	1
5	`SELECT` 1 <= `"0"` ;	0	0
6	`SELECT` 1 <= `"a"` ;	0	0
7	`SELECT` 1 <= `"1a"` ;	1	1
8	`SELECT TRUE` <= 0 ;	0	0
9	`SELECT TRUE` <= 1 ;	1	1
10	`SELECT` 2 <= `TRUE` ;	0	0
11	`SELECT FALSE` <= 1 ;	1	1
12	`SELECT TRUE` <= `FALSE` ;	0	0
13	`SELECT "1"` <= `"2"` ;	1	1
14	`SELECT "2"` <= `"1"` ;	0	0
15	`SELECT "-1"` <= `"1"` ;	1	1
16	`SELECT "-2"` <= `"-1"` ;	0	0
17	`SELECT "a"` <= `"b"` ;	1	1
18	`SELECT "b"` <= `"a"` ;	0	0
19	`SELECT "a"` <= `"ab"` ;	1	1
20	`SELECT "bb"` <= `"ac"` ;	0	0
21	`SELECT "-"` <= `"-2"` ;	1	1

Table 6.2: Comparison of the coercion of $\leq$ between MySQL and KSQL

No.	Query	result (MySQL)	result (KSQL)
1	`SELECT` 1 && 1 ;	1	1
2	`SELECT` 1 && 0 ;	0	0
3	`SELECT` 1 && `"1"` ;	1	1
4	`SELECT` `"1"` && 1 ;	1	1
5	`SELECT` `"1"` && `"1"` ;	1	1
6	`SELECT` `"1"` && `"0"` ;	0	0
7	`SELECT` `TRUE` && 1 ;	1	1
8	`SELECT` `FALSE` && 1 ;	0	0
9	`SELECT` `TRUE` && `FALSE` ;	0	0
10	`SELECT` 1 && `"a"` ;	0	0
11	`SELECT` 1 && `"1a"` ;	1	1
12	`SELECT` 1 && `"1a1"` ;	1	1
13	`SELECT` `"a"` && `"a"` ;	0	0

Table 6.3: Comparison of the coercion of && between MySQL and KSQL

The semantics of `NULL` in MySQL is two fold.

- Description of an error, which obeys to the strict semantics (Definition 6.8)
- The bottom constant in a three-valued logic (Definition 6.9)

In MySQL, if a function is a boolean operation, `NULL` behaves as the bottom constant. Otherwise, `NULL` is an error constant.

Definition 6.9. *The semantics of boolean operators*

$$\texttt{NOT} : Bool \qquad \&\& : Bool \times Bool \to Bool \qquad || : Bool \times Bool \to Bool$$

are defined as follows:

&&	TRUE	NULL	FALSE
TRUE	TRUE	NULL	FALSE
NULL	NULL	NULL	FALSE
FALSE	FALSE	FALSE	FALSE

||	TRUE	NULL	FALSE
TRUE	TRUE	TRUE	TRUE
NULL	TRUE	NULL	NULL
FALSE	TRUE	NULL	FALSE

	NOT
TRUE	FALSE
NULL	NULL
FALSE	TRUE

Lastly we have two special rules for zero-divisor in / and % operators.

Definition 6.10. *The semantics of zero-divisors of operators / and % are defined as*

$$e \;/\; 0 \to \texttt{NULL} \qquad\qquad e \;\%\; 0 \to \texttt{NULL}$$

The other errors are defined similarly using `NULL` representation. However, we cannot generalize `NULL` semantics for each operator, because of conflicts of the specification above.

The tables 6.4, 6.5, and 6.6 show testing results on `NULL` of MySQL and KSQL.

No.	Query	result (MySQL)	result (KSQL)
1	`SELECT` 1 + 1 ;	2	2
2	`SELECT` 1 + `NULL` ;	`NULL`	`NULL`
3	`SELECT NULL` + 1 ;	`NULL`	`NULL`
4	`SELECT NULL` + `NULL` ;	`NULL`	`NULL`
5	`SELECT NULL` || `TRUE` ;	1	1
6	`SELECT TRUE` || `NULL` ;	1	1
7	`SELECT NULL` && `FALSE` ;	0	0
8	`SELECT FALSE` && `NULL` ;	0	0

Table 6.4: Comparison of `NULL` (undefined) treatment between MySQL and KSQL

No.	MySQL query	result (MySQL)	result (KSQL)
1	`SELECT NULL` ;	`NULL`	`NULL`
2	`SELECT CONCAT(NULL,` 1) ;	`NULL`	`NULL`
3	`SELECT CONCAT(NULL, "1")` ;	`NULL`	`NULL`
4	`SELECT NULL IS NULL` ;	1	1
5	`SELECT "" IS NULL` ;	0	0

Table 6.5: Comparison of `NULL` in string expressions between MySQL and KSQL

No.	Query	result (MySQL)	result (KSQL)
1	`SELECT` 1 / 0 ;	`NULL`	`NULL`
2	`SELECT` 1 % 0 ;	`NULL`	`NULL`

Table 6.6: Comparison of `NULL` as errors between MySQL and KSQL

6.3 Name space

KSQL supports the name space. We keep a field name in the form *TableName.FieldName* and the schema is the set of those form. The semantics supports the field name extension.

Definition 6.11. *The evaluation of* `CREATE` *is defined as follows:*

$$\left\langle \frac{\texttt{CREATE TABLE } T(Fd)\ ;}{T[\texttt{createSchema}(T, Fd) : \epsilon_L\ ] \curvearrowright \texttt{store}\ \Diamond}\ \cdots \right\rangle_K$$

where T *is a table identifier and* `createSchema` *is a function defined below:*

$$\texttt{createSchema}(T, Fd) = [T.f \mid f\ \ d \in Fd \textit{ for some data type } d]$$

MySQL allows a user to identify the field by either by a field name or by a table name with the field name. We define a function to unify them to the latter form, i.e., *TableName.FieldName*.

Definition 6.12. *The function* `uniform` *is defined as follows:*

$$\texttt{uniform}(T, S) = [T.f \mid f \in S \vee T.f \in S]$$

Corresponding to this extension, each operation is redefined.

Definition 6.13. *The semantics of* `SELECT` *is redefined as follows:*

$$\left\langle \frac{\texttt{SELECT } P \texttt{ FROM } T \texttt{ WHERE } E\ ;}{\texttt{doGetTable}(T)\ \curvearrowright\ \texttt{doCondition}(\Diamond, E)\ \curvearrowright\ \texttt{doProjection}(\Diamond, \texttt{uniform}(T, P))}\ \cdots \right\rangle_K$$

Other rules are redefined in a similar way.

6.4 Limits and difficulties in KSQL

Coercion

To support the type coercion, we need to define many rewrite rules. For instance the semantics of $+$ and $\leq$ operators require four rewrite rules for each. This is to support the coercion from string to integer.

Our implementation of the coercion in KSQL is based on extensive case analysis. If f has n-arguments, the direct encoding of the coercion from sort τ_i to τ_i' ($1 \leq i \leq n$) is to prepare

$$f(t_1, \ldots, t_n) \to f(\mathcal{C}^f_{\tau_{t_1} \to \tau_{u_1}}(t_1), \ldots, \mathcal{C}^f_{\tau_1' \to \tau_n'}(t_n))$$

(if $\mathcal{C}^f_{\tau_{t_i} \to \tau_i'}(t_i) \neq t_i$ for some i), and becomes $2^n - 1$ rules.

Furthermore, if we have m different sorts, Each combination requires such rewrite rules and the total number of rewrite rules is $(m-1)(2^n-1)$.

Furthermore, if there are p operators $f_1, \ldots, f_p$ in a language, each of them has $n_1, \ldots, n_p$ arguments respectively. Then the number of rewrite rules becomes

$$\Sigma_{i=1}^{p}(m-1)(2^{n_i}-1).$$

For the coercion of current KSQL, there are 13 operators and 2 sorts. Each operator has 2 arguments, thus we need 39 rules (instead of 13 rules).

This exponential growth prohibits to apply our current method to a practical language. There are several possibility to reduce the number of rules. The first idea is to separate operators into groups of operators, which have the same type of argument. For example, arithmetic operators and string operators have certain similar structure in each category. By defining new structure of expressions as below:

$$\begin{aligned} \mathit{Op} &::= + \mid - \mid * \mid / \mid \% \\ \mathit{AExp} &::= \mathit{AExp}\ \ \mathit{Op}\ \ \mathit{AExp} \end{aligned}$$

The type of these operators' arguments is the integer type. Then, we can simply define semantics of the coercion as the rules 2, 3, 4 in Definition 6.4 by replacing $\circ$ with Op. This can reduce the number of rules from 15 to 3.

The second idea would be to use inheritance of the ordered sort, which we have not investigate so far.

Treatment of `NULL`

As we have seen that the semantics to support `NULL` requires many rewrite rules.

Some operators `IS NULL`, `IS NOT NULL`, or `IFNULL`, explicitly accept `NULL` as an argument. Except for such operators, the n-arguments require n-rules. Suppose that there are p operators $f_1, \ldots, f_p$ with $n_1, \ldots, n_p$ arguments, respectively. Then the number of rewrite rules is

$$\Sigma_{i=1}^{p} n_i.$$

Additionally, we have to define rules for error handling. It is difficult to exactly observe all behaviours of the error handling.

Although the number of rewrite rules for `NULL` treatment affect less (since it is not exponential). Still it is difficult because MySQL has many error-handling solutions. Further investigation will be required.

Logical and physical model of data types

For integer and string data types, KSQL supports unbounded bits of integers and strings. In fact, MySQL has variation of data types. For example, the integer data type varies TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT for 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integers, respectively. Current implementation of KSQL does not support such detailed variation of data types. This is because when we insert a datum into a field of integer, MySQL solves overflow and underflow of such an integer by bounding its value, which is not supported by current KSQL.

Uniqueness of primary key

Oracle11 and MySQL do not allow the duplication of the primary key. When the primary key has duplicated values, Oracle11 and MySQL return errors. In current implementation of KSQL, each record has a product type and thus allows duplicated records. This can be simply solved by defining a function type. For instance, consider a record that contains three integer columns, `A`, `B`, and `C`. Suppose that the primary key contains only the attribute `A`. Then, a record $(\mathtt{1}, \mathtt{2}, \mathtt{3})$ is a relation represented by $(\mathtt{1}, [\mathtt{2}, \mathtt{3}])$ instead of a list $[\mathtt{1}, \mathtt{2}, \mathtt{3}]$. We can define a type of records as $\mathit{Int} \rightarrow (\mathit{Int} \times \mathit{Int})$ instead of $\mathit{Int} \times \mathit{Int} \times \mathit{Int}$.

Chapter 7

Related work

There are several works to give formal semantics of practical languages. We overview such works in three views.

- Formal semantics of SQL [8]
- Formal semantics of other programming languages: PHP [11], DATALOG [1], self-modifying x86 [2].
- Executable formal semantics on K framework: JAVA [4], Verilog [6], Scheme [5], and C [3].

7.1 Formal semantics of SQL language

The semantics is based on first-order-logic (FOL) and this leads to the problem of the `NULL` definition.

Negri, Pelagatti, and Sbattella (1991) gave a formal semantics of SQL queries [8]. The work starts with the translation of SQL to a formal model consisting of a set of rules in first order logic. The formal model is called *Extended Three Valued Predicate Calculus* (E3VPC). However, their semantics is still missing the real semantics, e.g. `NULL`. First, an undefined value `NULL` in MySQL is in Kleene's three valued logic beyond the standard FOL. Secondly, their semantics focuses on the selection query only, while ours semantics additionally define creation, deletion, update, and insertion.

7.2 Other formal semantics

PHP Tozawa, Tatsubori, Onodera and Minamide (2009) gave the definition of a copy-on-write semantics of PHP language [11]. The semantics is used to solve copy-on-assignment, which causes copy overhead. Copy-on-write model is formalized by using graph rewriting.

DATALOG Alpuente, Feliu, Joubert, and Villanueva (2010) formalized the definition of DATALOG in a rewriting logic using Maude [1]. DATALOG is also a relational query language, beyond RDB, like SQL. It is a subset of Prolog (Prolog with only predicates and constants) and allow recursive expressions. Their focus is more on standard operations, but not on boundary cases, like error handling and coercion.

x86 Bonfante, Marion, and Reynaud-Plantey (2009) gave the formal semantics for self-modifying x86 programs [2]. A self-modifying binary program can be constructively rewritten to a non-modifying program.

7.3 Executable formal semantics on K framework

There are several programming languages being defined on the K framework. They motivated us to describe a core SQL language in the K framework.

Java Farzan, Chen, Meseguer, and Rosu (2004) gave the semantic of a program analysis framework for Java [4]. The semantics can be applied to model-checker provided by K framework.

Scheme Meredith, Hills, and Rosu (2007) defined an equational semantics of Scheme [5]. The semantics includes the support for macros.

Verilog Meredith, Katelman, Meseguer, and Rosu (2010) gave a formal executable semantics of Verilog [6]. Their semantics is used to emulate programs and search on its behaviours.

C Ellison and Rosu (2012) gave of an executable formal semantics of C [3]. Contributions of this work are not only the formal semantics itself, but also illustrating a way to discover bugs and the use of K framework for defining non-deterministic behaviours of a C program.

Chapter 8

Conclusion

In this thesis, we investigated the formal semantics of the core of SQL dialects, specifically MySQL and Oracle11. The former is implemented in the K framework as KSQL.

We found semantics differences on the coercion of types, the interpretation of `NULL`, the name space management, the error handling, and the logical/physical model of data types. Among the differences, we focus on the first three parts. We observe differences between MySQL and Oracle11:

- Consider the coercion from a string to an integer. Oracle11 returns an error if a string contains non-digit characters, while MySQL tries to conceal this problem.

- MySQL interprets `NULL` as an undefined value for boolean operations. However, `NULL` is also used as the return of an error like zero divisor. Surprisingly, MySQL further accepts `NULL` as an explicit argument, e.g., `NULL IS NULL`, `ISNULL`, `IS NOT NULL`, and `IFNULL`. We feel that these behaviour would lead inconsistency. Oracle11 has more reasonable behaviour. It interprets `NULL` as the empty string. An error is reported by an error message, not resulting `NULL` as an output.

- Oracle11 and MySQL adopt different name space conversion. The name space is the matter when the table with the same name appears repeatedly. For instance, self-join and self-product are such cases. Both MySQL and Oracle refuse them.

Our work is just to open possibility of formal semantics of SQL. There are several obstructions as future work.

- Current implementation of coercion is based on an exhaustive case analysis, which explodes with an exponentially many number of rewrite rules. (`NULL` requires similar case analysis, but the number of rules does not grow exponentially.) It can be reduced by unifying similar cases. Other possibility is to use the inheritance among ordered sorts.

- Currently, a table is defined to have a product type, and the uniqueness of the primary key is not guaranteed. This can be solved by applying a function type for a table, instead of the product type.
- Currently, KSQL ignores physical models (bit-length) of data types, which affects the semantics of JOIN operator. This is the reason why current KSQL does not implement JOIN.

Appendices

Appendix A

SQL semantics in K-framework

A.1 Expression Syntax

```
module EXP-SYNTAX
   syntax Boolean ::= "TRUE" | "FALSE"
   syntax Null ::= "NULL"
   syntax Val ::= Int | Bool | Float | String | Null | Boolean
   syntax Vals ::= List{Val, ","}
   syntax Exp ::= Val | Id
   syntax KResult ::=   #Int | #Bool | Id | #String | #Float | Null
   syntax Exp ::= "(" Exp ")" [bracket]
            > "-" Exp       [strict]
            > Exp "*" Exp  [strict, left]
            | Exp "/" Exp  [strict, left]
            | Exp "DIV" Exp   [strict, left]
            | Exp "MOD" Exp   [strict, left]
            | Exp "%" Exp  [strict, left]
            > Exp "+" Exp  [strict, left]
            | Exp "-" Exp  [strict, left]
   syntax Exp ::= "(" Exp ")" [bracket]
            > "!" Exp       [strict]
            > Exp "=" Exp  [strict, non-assoc]
            | Exp ">=" Exp [strict, non-assoc]
            | Exp ">" Exp  [strict, non-assoc]
            | Exp "<=" Exp [strict, non-assoc]
            | Exp "<" Exp  [strict, non-assoc]
            | Exp "!=" Exp [strict, non-assoc]
            | Exp "<>" Exp [strict, non-assoc]
            | Exp "IS" "TRUE"    [strict]
            | Exp "IS" "FALSE"   [strict]
            | Exp "IS" "NULL"    [strict]
            | Exp "IS" "UNKNOWN"    [strict]
            | Exp "IS" "NOT" "TRUE" [strict]
            | Exp "IS" "NOT" "FALSE"   [strict]
            | Exp "IS" "NOT" "NULL"    [strict]
            | Exp "IS" "NOT" "UNKNOWN" [strict]
            > "NOT" Exp        [strict]
            > Exp "&&" Exp     [strict(1), left]
            | Exp "AND" Exp    [strict(1), left]
            > Exp "||" Exp     [strict, left]
            | Exp "OR" Exp     [strict, left]
```

```
  syntax Exp ::= "CONCAT" "(" Vals ")"          [strict]
           | "ELT" "(" Int "," Strings ")"      [strict]
           | "FIELD" "(" String "," Strings ")"   [strict]
           | "INSERT" "(" String "," Int "," Int "," Strings ")" [strict]
           | "INSTR" "(" String "," String ")" [strict]
           | "LENGTH" "(" String ")"        [strict]
           | "LOCATE" "(" String "," String ")"          [strict]
           | "LOCATE" "(" String "," String "," Int ")" [strict]
           | "TRIM" "(" String ")"          [strict]
           | "LTRIM" "(" String ")"         [strict]
           | "RTRIM" "(" String ")"         [strict]
           | "POSITION" "(" String "IN" String ")"      [strict]
           | "REPEAT" "(" String "," Int ")"            [strict]
           | "REPLACE" "(" String "," String "," String ")"   [strict]
           | "LEFT" "(" String "," Int ")"      [strict]
           | "RIGHT" "(" String "," Int ")"     [strict]
           | "SPACE" "(" Int ")"                [strict]
           | "SUBSTRING" "(" String "," Int ")"          [strict]
           | "SUBSTRING" "(" String "FROM" Int ")"      [strict]
           | "SUBSTRING" "(" String "," Int "," Int ")"       [strict]
           | "SUBSTRING" "(" String "FROM" Int "FOR" Int ")"  [strict]
  syntax Ints ::= List{Int,","}          [strict]
  syntax Strings ::= List{String,","}    [strict]
  syntax Exps ::= List{Exp,","}          [strict]
  syntax Ids ::= List{Id,","}
  syntax Bool ::= in( String , Strings)
  syntax Int ::= getOrd(String)          [strict]
  syntax String ::= intToChar(Int)       [function]
  syntax Int ::= charToInt(String)       [function]
  syntax Int ::= strcmp(String,String)   [function]
  syntax Int ::= cmpChar(String,String)  [function]
  syntax Int ::= cmpChar(String,String)  [function]
  syntax Int ::= convertsi(String)       [function]
  syntax Int ::= firstLetterAt(String)   [function]
endmodule
```

A.2 Expression Semantics

```
module EXP
  imports EXP-SYNTAX
  rule TRUE:Boolean => 1     [anywhere]
  rule FALSE:Boolean => 0       [anywhere]
  rule true:Boolean => 1
  rule false:Boolean => 0
  rule ! 0 => 1
  rule ! I:Int => 0 when I =Int 0
  rule ! NULL => NULL
  rule - I:Int => 0 -Int I
  rule I1:Int * I2:Int => I1 *Int I2
  rule I1:Int / I2:Int => I1 /Int I2 when I2 =/=K 0
  rule I1:Int / I2:Int => NULL when I2 ==K 0
  rule I1:Int DIV I2:Int => I1 /Int I2 when I2 =/=K 0
  rule I1:Int DIV I2:Int => NULL when I2 ==K 0
  rule I1:Int MOD I2:Int => I1 %Int I2 when I2 =/=K 0
  rule I1:Int MOD I2:Int => NULL when I2 ==K 0
  rule I1:Int % I2:Int => I1 %Int I2 when I2 =/=K 0
  rule I1:Int % I2:Int => NULL when I2 ==K 0
  rule I1:Int + I2:Int => I1 +Int I2
  rule I1:Int - I2:Int => I1 -Int I2
```

```
rule I1:Int >= I2:Int => 1 when I1 >=Int I2
rule I1:Int >= I2:Int => 0 when I1 <Int I2
rule I1:Int > I2:Int => 1 when I1 >Int I2
rule I1:Int > I2:Int => 0 when I1 <=Int I2
rule I1:Int <= I2:Int => 1 when I1 <=Int I2
rule I1:Int <= I2:Int => 0 when I1 >Int I2
rule I1:Int < I2:Int => 1 when I1 <Int I2
rule I1:Int < I2:Int => 0 when I1 >=Int I2

rule S:String + I:Int => convertsi(S) + I    [anywhere]
rule I:Int + S:String => I + convertsi(S)    [anywhere]
rule S1:String + S2:String => convertsi(S1) + convertsi(S2) [anywhere]
rule S:String - I:Int => convertsi(S) - I    [anywhere]
rule I:Int - S:String => I - convertsi(S)    [anywhere]
rule S1:String - S2:String => convertsi(S1) - convertsi(S2) [anywhere]
rule S:String * I:Int => convertsi(S) * I    [anywhere]
rule I:Int * S:String => I * convertsi(S)    [anywhere]
rule S1:String * S2:String => convertsi(S1) * convertsi(S2) [anywhere]
rule S:String / I:Int => convertsi(S) / I    [anywhere]
rule I:Int / S:String => I / convertsi(S)    [anywhere]
rule S1:String / S2:String => convertsi(S1) / convertsi(S2) [anywhere]
rule S:String DIV I:Int => convertsi(S) DIV I   [anywhere]
rule I:Int DIV S:String => I DIV convertsi(S)   [anywhere]
rule S1:String DIV S2:String => convertsi(S1) DIV convertsi(S2)   [anywhere]
rule S:String % I:Int => convertsi(S) % I    [anywhere]
rule I:Int % S:String => I % convertsi(S)    [anywhere]
rule S1:String % S2:String => convertsi(S1) % convertsi(S2) [anywhere]

rule F1:Float * F2:Float => F1 *Float F2
rule I1:Int * F2:Float => Int2Float(I1) * F2
rule F1:Float * I2:Int => F1 * Int2Float(I2)

rule F1:Float / F2:Float => F1 /Float F2 when F2 =/=K 0
rule F1:Float / F2:Float => NULL when F2 ==K 0
rule F1:Float DIV F2:Float => F1 /Float F2 when F2 =/=K 0
rule F1:Float DIV F2:Float => NULL when F2 ==K 0

rule I1:Int / F2:Float => Int2Float(I1) / F2
rule F1:Float / I2:Int => F1 / Int2Float(I2)
rule I1:Int DIV F2:Float => Int2Float(I1) DIV F2
rule F1:Float DIV I2:Int => F1 DIV Int2Float(I2)

rule F1:Float MOD F2:Float => F1 %Float F2 when F2 =/=K 0
rule F1:Float MOD F2:Float => NULL when F2 ==K 0
rule F1:Float % F2:Float => F1 %Float F2 when F2 =/=K 0
rule F1:Float % F2:Float => NULL when F2 ==K 0

rule I1:Int MOD F2:Float => Int2Float(I1) MOD F2
rule F1:Float MOD I2:Int => F1 MOD Int2Float(I2)
rule I1:Int % F2:Float => Int2Float(I1) % F2
rule F1:Float % I2:Int => F1 % Int2Float(I2)

rule F1:Float + F2:Float => F1 +Float F2
rule I1:Int + F2:Float => Int2Float(I1) + F2
rule F1:Float + I2:Int => F1 + Int2Float(I2)
rule F1:Float - F2:Float => F1 -Float F2
rule I1:Int - F2:Float => Int2Float(I1) - F2
rule F1:Float - I2:Int => F1 - Int2Float(I2)
rule F1:Float >= F2:Float => 1 when F1 >=Float F2
rule F1:Float >= F2:Float => 0 when F1 <Float F2
rule I1:Int >= F2:Float => Int2Float(I1) >= F2
```

```
rule F1:Float >= I2:Int => F1 >= Int2Float(I2)
rule F1:Float > F2:Float => 1 when F1 >Float F2
rule F1:Float > F2:Float => 0 when F1 <=Float F2
rule I1:Int > F2:Float => Int2Float(I1) > F2
rule F1:Float > I2:Int => F1 > Int2Float(I2)
rule F1:Float <= F2:Float => 1 when F1 <=Float F2
rule F1:Float <= F2:Float => 0 when F1 >Float F2
rule I1:Int <= F2:Float => Int2Float(I1) <= F2
rule F1:Float <= I2:Int => F1 <= Int2Float(I2)
rule F1:Float < F2:Float => 1 when F1 <Float F2
rule F1:Float < F2:Float => 0 when F1 >=Float F2
rule I1:Int < F2:Float => Int2Float(I1) < F2
rule F1:Float < I2:Int => F1 < Int2Float(I2)

rule S:String = I:Int => convertsi(S) = I [anywhere]
rule I:Int = S:String => I = convertsi(S) [anywhere]
rule S:String < I:Int => convertsi(S) < I [anywhere]
rule I:Int < S:String => I < convertsi(S) [anywhere]
rule S:String <= I:Int => convertsi(S) <= I  [anywhere]
rule I:Int <= S:String => I <= convertsi(S)  [anywhere]
rule S:String > I:Int => convertsi(S) > I [anywhere]
rule I:Int > S:String => I > convertsi(S)    [anywhere]
rule S:String >= I:Int => convertsi(S) >= I  [anywhere]
rule I:Int >= S:String => I >= convertsi(S)  [anywhere]
rule S:String != I:Int => convertsi(S) != I  [anywhere]
rule I:Int != S:String => I != convertsi(S)  [anywhere]
rule S:String <> I:Int => convertsi(S) <> I  [anywhere]
rule I:Int <> S:String => I <> convertsi(S)  [anywhere]

rule I1:Int = I2:Int => 1 when I1 =Int I2          [anywhere]
rule I1:Int = I2:Int => 0 when I1 =/=Int I2        [anywhere]
rule I1:Int != I2:Int => 1 when I1 =/=Int I2       [anywhere]
rule I1:Int != I2:Int => 0 when I1 =Int I2         [anywhere]
rule F1:Float = F2:Float => 1 when F1 ==Float F2   [anywhere]
rule F1:Float = F2:Float => 0 when F1 =/=Float F2  [anywhere]
rule F1:Float != F2:Float => 1 when F1 =/=Float F2 [anywhere]
rule F1:Float != F2:Float => 0 when F1 ==Float F2  [anywhere]
rule S1:String >= S2:String => 1 when strcmp(S1,S2) >=Int 0    [anywhere]
rule S1:String >= S2:String => 0 when strcmp(S1,S2) =Int -1    [anywhere]
rule S1:String > S2:String => 1 when strcmp(S1,S2) =Int 1      [anywhere]
rule S1:String > S2:String => 0 when strcmp(S1,S2) <=Int 0     [anywhere]
rule S1:String < S2:String => 1 when strcmp(S1,S2) =Int -1     [anywhere]
rule S1:String < S2:String => 0 when strcmp(S1,S2) >=Int 0     [anywhere]
rule S1:String <= S2:String => 1 when strcmp(S1,S2) <=Int 0    [anywhere]
rule S1:String <= S2:String => 0 when strcmp(S1,S2) =Int 1     [anywhere]
rule S1:String = S2:String => 1 when strcmp(S1,S2) =Int 0      [anywhere]
rule S1:String = S2:String => 0 when strcmp(S1,S2) =/=Int 0    [anywhere]
rule S1:String != S2:String => 1 when strcmp(S1,S2) =/=Int 0   [anywhere]
rule S1:String != S2:String => 0 when strcmp(S1,S2) =Int 0     [anywhere]
rule (N1 <> N2) => N1 != N2   [anywhere]

rule NULL + _ => NULL
rule _ + NULL => NULL
rule NULL - _ => NULL
rule _ - NULL => NULL
rule NULL * _ => NULL
rule _ * NULL => NULL
rule NULL / _ => NULL
rule _ / NULL => NULL
rule NULL DIV _ => NULL
rule _ DIV NULL => NULL
```

```
  rule NULL MOD _ => NULL
  rule _ MOD NULL => NULL
  rule NULL % _ => NULL
  rule _ % NULL => NULL
  rule NULL <= _ => NULL
  rule _ <= NULL => NULL
  rule NULL < _ => NULL
  rule _ < NULL => NULL
  rule NULL = _ => NULL
  rule _ = NULL => NULL
  rule NULL != _ => NULL
  rule _ != NULL => NULL
  rule NULL <> _ => NULL
  rule _ <> NULL => NULL
  rule NULL >= _ => NULL
  rule _ >= NULL => NULL
  rule NULL > _ => NULL
  rule _ > NULL => NULL

  rule I:Int IS TRUE => 1 when I =/=K 0
  rule 0 IS TRUE => 0
  rule NULL IS TRUE => 0
  rule A IS TRUE => 0 when A ==Int 0
  rule I:Int IS FALSE => 0 when I =/=K 0
  rule 0 IS FALSE => 1
  rule NULL IS FALSE => 0
  rule A IS FALSE => 1 when A ==Int 0
  rule I:Int IS UNKNOWN => 0 when I =/=K 0
  rule 0 IS UNKNOWN => 0
  rule NULL IS UNKNOWN => 1
  rule A IS UNKNOWN => 1 when A ==K NULL
  rule I:Int IS NULL => 0 when I =/=K 0
  rule 0 IS NULL => 0
  rule NULL IS NULL => 1
  rule A IS NULL => 0 when A ==K NULL

  rule I:Int IS NOT TRUE => 0 when I =/=K 0
  rule 0 IS NOT TRUE => 1
  rule NULL IS NOT TRUE => 1
  rule A IS NOT TRUE => 0 when A ==Int 0
  rule I:Int IS NOT FALSE => 1 when I =/=K 0
  rule 0 IS NOT FALSE => 0
  rule NULL IS NOT FALSE => 1
  rule A IS NOT FALSE => 1 when A ==Int 0
  rule I:Int IS NOT UNKNOWN => 1 when I =/=K 0
  rule 0 IS NOT UNKNOWN => 1
  rule NULL IS NOT UNKNOWN => 0
  rule A IS NOT UNKNOWN => 0 when A ==K NULL
  rule I:Int IS NOT NULL => 1 when I =/=K 0
  rule 0 IS NOT NULL => 1
  rule NULL IS NOT NULL => 0
  rule A IS NOT NULL => 0 when A ==K NULL

  rule 0 && 0 => 0
  rule 0 && 1 => 0
  rule 0 && NULL => 0
  rule I:Int && 0 => 0 when I =/=K 0
  rule I:Int && 1 => 1 when I =/=K 0
  rule I:Int && NULL => NULL when I =/=K 0
  rule NULL && 0 => 0
  rule NULL && I:Int => NULL when I =/=K 0
```

```
rule NULL && NULL => NULL
rule I1:Int && S2:String => I1 && convertsi(S2)
rule S1:String && I2:Int => convertsi(S1) && I2
rule S1:String && S2:String => convertsi(S1) && convertsi(S2)

rule 0 || 0 => 0
rule 0 || I:Int => 1 when I =/=K 0
rule 0 || NULL => NULL
rule I:Int || 0 => 1 when I =/=K 0
rule I:Int || I => 1 when I =/=K 0
rule I:Int || NULL => 1
rule NULL || 0 => NULL
rule NULL || I:Int => 1 when I =/=K 0
rule NULL || NULL => NULL
rule I1:Int || S2:String => I1 || convertsi(S2)
rule S1:String || I2:Int => convertsi(S1) || I2
rule S1:String || S2:String => convertsi(S1) || convertsi(S2)

rule I:Int && B => B when I =/=K 0
rule 0 && B => 0
rule I:Int || B => 1 when I =/=K 0
rule 0 || B => B
rule I:Int AND B => B when I =/=K 0
rule 0 AND B => 0
rule I:Int OR B => 1 when I =/=K 0
rule 0 OR B => B
rule I1 AND I2 => I1 && I2
rule I1 OR I2 => I1 || I2

rule NOT 0 => 1
rule NOT I:Int => 0 when I =/=Int 0
rule NOT NULL => NULL


rule B1:Bool && B2:Bool => B1 andBool B2
rule B1:Bool AND B2:Bool => B1 andBool B2
rule B1:Bool || B2:Bool => B1 orBool B2
rule B1:Bool OR B2:Bool => B1 orBool B2

rule CONCAT(.Vals) => ""
rule CONCAT(S:String) => S
rule CONCAT(S:String, Vs:Vals) => CONCAT(Vs) ~> S +String HOLE
rule S2:String ~> S1 +String HOLE => S1 +String S2       [structural]
rule NULL ~> S1 +String HOLE => NULL                     [structural]

rule CONCAT(NULL, Vs:Vals) => NULL
rule CONCAT(I:Int, Vs:Vals) => CONCAT(Int2String(I),Vs)
rule CONCAT(F:Float, Vs:Vals) => CONCAT(Float2String(F),Vs)


rule ELT(1, S:String , Ss:Strings) => S
rule ELT(N:Int, S:String, Ss:Strings) => ELT(N -Int 1, Ss)

rule FIELD(S:String, Ss:Strings) => 0 when notBool in(S,Ss)
rule FIELD(S:String, .Strings) => 0
rule FIELD(S:String, S2:String, Ss:Strings) => 1 when S ==String S2
rule FIELD(S:String, S2:String, Ss:Strings) => 1 + FIELD(S,Ss) when notBool (S
 ==String S2)

rule INSERT(S1:String, P:Int, L:Int, S2:String) => (substrString(S1, 0, P -Int 1)
 +String S2) +String (substrString(S1, (P -Int 1) +Int L, lengthString(S1)))
```

```
rule INSTR(S1:String, S2:String) => findString(S1,S2,3) +Int 1
rule LENGTH(S:String) => lengthString(S)
rule LOCATE(Sub:String, S:String) => findString(S,Sub,0) +Int 1
rule LOCATE(Sub:String, S:String, P:Int) => findString(S,Sub,P) +Int 1
rule POSITION( Sub:String IN S:String) => findString(S,Sub,0) +Int 1
rule TRIM(S:String) => trim(S)
rule LTRIM(S:String) => ltrim(S)
rule RTRIM(S:String) => rtrim(S)
rule REPEAT(S:String,0) => ""
rule REPEAT(S:String, N:Int) => REPEAT(S, (N -Int 1)) ~> S +String HOLE
rule S2:String ~> S +String HOLE => S +String S2 [structural]
rule REPLACE(S:String, FromS:String, ToS:String) => replaceAll(S,FromS,ToS)
rule LEFT(S:String, L:Int) => substrString(S,0,L)
rule RIGHT(S:String, L:Int) => substrString(S,lengthString(S) -Int L, lengthString(S))
rule SPACE(0) => ""
rule SPACE(N:Int) =>  SPACE(N -Int 1) ~> HOLE +String " "
rule S:String ~> HOLE +String S2 => S +String S2        [structural]
rule SUBSTRING(S:String, StartP:Int) => substrString(S, StartP -Int 1,
 lengthString(S))
rule SUBSTRING(S:String FROM StartP:Int) => substrString(S, StartP -Int 1,
 lengthString(S))
rule SUBSTRING(S:String , StartP:Int , Len:Int) => substrString(S, StartP -Int 1,
 (StartP -Int 1) +Int Len)
rule SUBSTRING(S:String FROM StartP:Int FOR Len:Int) => substrString(S, StartP -Int
 1, (StartP -Int 1) +Int Len)

rule in(S:String , .Strings ) => false [anywhere]
rule in(S:String , S2:String , Ss:Strings) => true when S ==String S2 [anywhere]
rule in(S:String , S2:String , Ss:Strings) => in(S, Ss) when S =/=String S2 [anywhere]

rule strcmp("","") => 0
rule strcmp("",S:String) => -1 when lengthString(S) >Int 0
rule strcmp(S:String,"") => 1 when lengthString(S) >Int 0
rule strcmp(S1:String,S2:String) => strcmp(substrString(S1,1,lengthString(S1)),
 substrString(S2,1,lengthString(S2))) when ordChar(substrString(S1,0,1)) ==Int
 ordChar(substrString(S2,0,1))
rule strcmp(S1:String,S2:String) => cmpChar(substrString(S1,0,1),
 substrString(S2,0,1)) when ordChar(substrString(S1,0,1)) =/=Int
 ordChar(substrString(S2,0,1))

rule cmpChar(S1:String,S2:String) => -1 when ordChar(S1) <Int ordChar(S2)
rule cmpChar(S1:String,S2:String) => 0 when ordChar(S1) ==Int ordChar(S2)
rule cmpChar(S1:String,S2:String) => 1 when ordChar(S1) >Int ordChar(S2)

rule intToChar(I:Int) => chrChar(I)
rule charToInt(S:String) => ordChar(S)

rule convertsi(S:String) => 0 when  notBool (#isDigit(substrString(S,0,1)) orBool
 (substrString(S,0,1) ==String "+" orBool substrString(S,0,1) ==String "-"))
rule convertsi(S:String) => 0 when (substrString(S,0,1) ==String "+" orBool
 substrString(S,0,1) ==String "-") andBool (notBool #isDigit(substrString(S,1,2)))
rule convertsi(S:String) => String2Int(substrString(S,1,(1 +Int
 firstLetterAt(substrString(S,1,lengthString(S)))))) when substrString(S,0,1)
 ==String "+" andBool #isDigit(substrString(S,1,2))
rule convertsi(S:String) => String2Int(substrString(S,0,(1 +Int
 firstLetterAt(substrString(S,1,lengthString(S)))))) when  substrString(S,0,1)
 ==String "-" andBool #isDigit(substrString(S,1,2))
rule convertsi(S:String) => String2Int(substrString(S,0,1 +Int
 firstLetterAt(substrString(S,1,lengthString(S))))) when #isDigit(substrString(S,0,1))

rule firstLetterAt("") => 0
```

```
  rule firstLetterAt(S:String) =>  (1 +Int
   firstLetterAt(substrString(S,1,lengthString(S)))) when #isDigit(substrString(S,0,1))

  rule firstLetterAt(S:String) => 0 when notBool #isDigit(substrString(S,0,1))

endmodule
```

A.3 Table Syntax

```

module TABLE-SYNTAX
  imports EXP
  syntax DataType ::= "INT" | "BOOL" | "TEXT" | "FLOAT"
////**** Representation ****/////
  syntax #TableElement ::= "e(" Vals ")"        [strict]
  syntax TableElement ::= #TableElement
  syntax TableElements ::= List{TableElement,","} [strict]
  syntax #Record ::= "r[" TableElements "]"    [strict]
  syntax Record ::= #Record
  syntax #Field ::= "f(" String "," DataType "," Bool "," Bool ")"
  syntax Field ::= #Field
  syntax Fields ::= List{Field,","}          [strict]
  syntax #Schema ::= "s[" Fields "]"         [strict]
  syntax Schema ::= #Schema
  syntax #Table ::= Id "[" Schema ":" Record "]"  [strict]
  syntax Val ::= FieldRep
  syntax FieldRep ::= FieldRep1 | FieldRep2
  syntax FieldRep1 ::= Id | "`" Id "`"
  syntax FieldRep1s ::= List{FieldRep1,","}
  syntax FieldRep2 ::= Id "." Id
  syntax Collumn ::= FieldRep
  syntax Collumns ::= List{FieldRep,","}
  syntax KResult ::= #Record | #Field | #Schema | #Table | #TableElement

////**** Main function ****/////
  syntax Table ::=  #Table
                  | Table "union" Table      [strict]
                  | Table "intersect" Table  [strict]
         | Table "difference" Table    [strict]
         | Table "cartesian" Table     [strict]
         | Table "x" Table             [strict]

  syntax Table ::= union(Table,Table)          [strict]
  syntax Table ::= difference(Table,Table)     [strict]
  syntax Table ::= intersect(Table,Table)      [strict]
  syntax Table ::= catesian(Table,Table)       [strict]
  syntax Record ::= union(Record,Record)       [strict]
  syntax Record ::= difference(Record,Record)  [strict]
  syntax Record ::= intersect(Record,Record)   [strict]
  syntax Record ::= catesian(Record,Record)    [strict]
  syntax Table ::= join(Table,Table)           [strict]
  syntax Table ::= join(Table,Table,Exp)       [strict(1,2)]
  syntax Table ::= leftJoin(Table,Table,Exp)   [strict(1,2)]
  syntax Table ::= rightJoin(Table,Table,Exp)  [strict(1,2)]
  syntax Table ::= select(Table,Exp)           [strict(1)]
  syntax Record ::= leftJoin2(Schema,Schema,Record,Record,Exp)
  syntax Record ::= unionLeftJoin(Schema,Schema,TableElement,Record)
  syntax Record ::= rightJoin2(Schema,Schema,Record,Record,Exp)
  syntax Record ::= unionRightJoin(Schema,Schema,TableElement,Record)
```

```

////**** Auxiliary function ****/////
   syntax Id ::= "tmp"
   syntax Int ::= getIndex(Schema,String)          [strict]
   syntax Int ::= getIndex2(Schema,String)         [strict]
   syntax Exp ::= getValue(TableElement,Exp)       [strict]
   syntax Record ::= filter(Schema,Record,Exp)     [strict(1,2)]
   syntax SelectElement ::= s(Int,TableElement)    [strict]
   syntax KResult ::= SelectElement
   syntax Exp ::= eval(Schema,TableElement,Exp)    [strict(1,2)]
   syntax Field ::= getFieldFromSchema(Collumn,Schema)   [strict]
   syntax Table ::= project(Table,Schema) [strict]
   syntax Record ::= project2(Record,Schema,Schema)      [strict]
   syntax TableElement ::= project3(TableElement,Schema,Schema)   [strict]
   syntax Val ::= getValue(Schema,TableElement,String)            [strict]
   syntax Table ::= rename(Table,Id)               [strict]
   syntax Table ::= changeFieldNameCorrespondToTable(Table)       [strict]
   syntax Field ::= rename(Field,String)           [strict]
   syntax Fields ::= concatFieldName(Schema,Id)    [strict]
   syntax String ::= changeFieldRepIntoStringName( FieldRep )  [strict]
   syntax String ::= changeFieldRepIntoStringName( Id , Id )   [strict]
   syntax Schema ::= excludeFields(Schema,FieldRep1s)
   syntax Schema ::= excludeField(Schema,FieldRep1)

   syntax TableElement ::= addNullElementOnBottom(TableElement,Int)  [strict]
   syntax TableElement ::= addNullElementOnTop(TableElement,Int)     [strict]
   syntax Int ::= numberOfElementInRecord(Record)           [strict]
   syntax TableElement ::= addElementOnBottom(TableElement,Val)      [strict]
   syntax TableElement ::= addElementOnTop(TableElement,Val)         [strict]
   syntax K ::= if(Int,K,K)
   syntax Schema ::= addElement(Field,Schema)         [strict]
   syntax Schema ::= concat(Schema, Schema)           [strict]
   syntax Record ::= concat(Record, Record)           [strict]
   syntax TableElement ::= append(TableElement,TableElement)   [strict]
   syntax TableElement ::= addTopElement(Val , TableElement)   [strict]
   syntax Record ::= addElement(TableElement,Record)           [strict]
   syntax Record ::= appendElementToRecord(TableElement,Record)[strict]
   syntax Int ::= numberOfFields(Schema)              [strict]
   syntax Bool ::= checkUnionCompatible( Schema , Schema)      [strict]
   syntax Bool ::= isTableElementEqual( TableElement,TableElement)   [strict]
   syntax Bool ::= consistOf(Record,TableElement)     [strict]
   syntax Bool ::= in( String , Ids)
   syntax K ::= num(Ids)
endmodule
```

A.4 Table Semantics

```
module TABLE
      imports TABLE-SYNTAX

////**** Main function ****/////
   rule T1:Table cartesian T2:Table => catesian(T1, T2) [anywhere]
   rule T1:Table union T2:Table => union(T1,T2) [anywhere]
   rule T1:Table difference T2:Table => difference(T1,T2) [anywhere]
   rule T1:Table intersect T2:Table => intersect(T1,T2) [anywhere]

   // Union
   rule union(Id1:Id[S1:Schema : R1:Record] , Id2:Id[S2:Schema : R2:Record]) =>
    union(R1,R2) ~> tmp[S1 : HOLE] when checkUnionCompatible(S1,S2) [structural]
```

```
rule union(R,r[.TableElements]) => R [anywhere,structural]
rule union(r[.TableElements],R) => R [anywhere,structural]
rule union(r[T:TableElement,Ts:TableElements],R) => union(r[Ts],R) when
 consistOf(R,T) [anywhere]
rule union(r[T1:TableElement,
 Ts1:TableElements],r[T2:TableElement,Ts2:TableElements]) => union(r[Ts1],r[T2,Ts2])
 ~> addElement(T1,HOLE) when notBool
 consistOf(r[T2:TableElement,Ts2:TableElements],T1) [anywhere]

// Difference
rule difference(Id1:Id[S1:Schema : R1:Record] , Id2:Id[S2:Schema : R2:Record]) =>
 difference(R1,R2) ~> tmp[S1 : HOLE ] when checkUnionCompatible(S1,S2)

rule difference(r[.TableElements],R) => r[.TableElements] [structural]
rule difference(r[T:TableElement,Ts:TableElements],R) => difference(r[Ts],R) when
 consistOf(R,T)
rule difference(r[T:TableElement,Ts:TableElements],R) => difference(r[Ts],R) ~>
 addElement(T, HOLE ) when notBool consistOf(R,T)

// Intersect
rule intersect(Id1:Id[S1:Schema : R1:Record] , Id2:Id[S2:Schema : R2:Record]) =>
 intersect(R1,R2) ~> tmp[S1: HOLE ] when checkUnionCompatible(S1,S2)
rule intersect(r[.TableElements],R) => r[.TableElements]  [structural]
rule intersect(r[T:TableElement,Ts:TableElements],R) => intersect(r[Ts],R) when
 notBool consistOf(R,T)
rule intersect(r[T1:TableElement,
 Ts1:TableElements],r[T2:TableElement,Ts2:TableElements]) => intersect(r[Ts1],r[Ts2])
 ~> addElement(T1, HOLE ) when consistOf(r[T2:TableElement,Ts2:TableElements],T1)

// Catesian
rule catesian(r[.TableElements],_) => r[.TableElements] [structural]
rule catesian(r[E1:TableElement, Es1:TableElements],r[Es2:TableElements]) =>
 concat(appendElementToRecord(E1,r[Es2]),catesian(r[Es1],r[Es2]))
rule catesian(T1:Id[S1:Schema : R1:Record],T2:Id[S2:Schema : R2:Record]) =>
 tmp[concat(S1,S2) : catesian(R1,R2)]
rule T1:Table cartesian T2:Table =>
 catesian(changeFieldNameCorrespondToTable(T1),changeFieldNameCorrespondToTable(T2))

// Renaming
rule rename(T1:Id[S:Schema : R:Record], T2:Id) => T2[S : R] [anywhere]
rule rename(f(_,T,B1,B2),S2:String) => f(S2,T,B1,B2)  [anywhere]
rule concatFieldName(s[.Fields],T:Id) => .Fields      [anywhere]
rule concatFieldName(s[f(F:String,DT:DataType,B1:Bool,B2:Bool),Fs:Fields],T:Id) =>
 rename(f(F,DT,B1,B2),(#tokenToString(T) +String ".") +String F),
 concatFieldName(s[Fs],T)  [anywhere]

// cross join
rule join(T1:Table, T2:Table) => T1 cartesian T2
rule join(T1:Table, T2:Table, E:Exp) => T1 cartesian T2 ~> select( HOLE , E)
rule T:#Table ~> select(HOLE , E) => select(T,E)                 [structural]

// left join
rule   leftJoin(T1:Id[ S1:Schema : R1] , T2:Id[ S2:Schema : R2:Record ], E:Exp) =>
      tmp[ concat(S1,S2) : leftJoin2(S1,S2, R1,R2,E) ]
rule   leftJoin2(S1:Schema,S2:Schema, r[ .TableElements], R:Record, E:Exp) => r[
 .TableElements ]
rule   leftJoin2(S1:Schema,S2:Schema, r[TE1:TableElement, TEs:TableElements],
 R:Record, E:Exp) => filter(concat(S1,S2),catesian( r[ TE1 ], R), E) ~>
 unionLeftJoin(S1,S2,TE1,HOLE) ~> union(HOLE,leftJoin2(S1,S2,r[TEs],R,E))
rule   Result:#Record ~> unionLeftJoin(S1:Schema,S2:Schema,T:TableElement,HOLE) =>
```

```
    unionLeftJoin(S1,S2,T,Result) [structural]
  rule   Result:#Record ~>
    union(HOLE,leftJoin2(S1:Schema,S2:Schema,R1:Record,R2:Record,E:Exp)) =>
    union(Result,leftJoin2(S1,S2,R1,R2,E)) [structural]
  rule   unionLeftJoin(S1:Schema,S2:Schema,T:TableElement,Result:Record) =>
    numberOfElementInRecord(Result) = 0 ~> if(HOLE, r[
    addNullElementOnBottom(T,numberOfFields(S2)) ], Result)

  // rightjoin
  rule   rightJoin(T1:Id[ S1:Schema : R1] , T2:Id[ S2:Schema : R2:Record ], E:Exp) =>
        tmp[ concat(S1,S2) : rightJoin2(S1,S2, R1,R2,E) ]
  rule   rightJoin2(S1:Schema,S2:Schema,  R:Record ,r[ .TableElements], E:Exp) => r[
    .TableElements ]
  rule   rightJoin2(S1:Schema,S2:Schema,  R:Record, r[TE1:TableElement,
    TEs:TableElements], E:Exp) => filter(concat(S1,S2),catesian( R, r[ TE1 ]), E) ~>
    unionRightJoin(S1,S2,TE1,HOLE) ~> union(HOLE,rightJoin2(S1,S2,R,r[TEs],E))
  rule   Result:#Record ~> unionRightJoin(S1:Schema,S2:Schema,T:TableElement,HOLE) =>
    unionRightJoin(S1,S2,T,Result) [structural]
  rule   Result:#Record ~>
    union(HOLE,rightJoin2(S1:Schema,S2:Schema,R1:Record,R2:Record,E:Exp)) =>
    union(Result,rightJoin2(S1,S2,R1,R2,E)) [structural]
  rule   unionRightJoin(S1:Schema,S2:Schema,T:TableElement,Result:Record) =>
    numberOfElementInRecord(Result) = 0 ~> if(HOLE, r[
    addNullElementOnTop(T,numberOfFields(S2)) ], Result)

////**** Auxiliary function ****/////

  rule getIndex(s[ .Fields ], S2:String) => NULL [anywhere]
  rule getIndex(s[f(S1:String,_,_,_) , Fs:Fields] , S2:String) => 0 when S1 ==String S2
    [anywhere]
  rule getIndex(s[f(S1:String,_,_,_) , Fs:Fields] , S2:String) => (getIndex(s[Fs],S2) +
    1) when notBool S1 ==String S2 [anywhere]
  rule getIndex2(s[ .Fields ], S2:String) => NULL [anywhere]
  rule getIndex2(s[f(S1:String,_,_,_) , Fs:Fields] , S2:String) => 0 when
    substrString(S1,(findChar(S1,".",0) +Int 1),lengthString(S1)) ==String S2 [anywhere]
  rule getIndex2(s[f(S1:String,_,_,_) , Fs:Fields] , S2:String) => (getIndex2(s[Fs],S2)
    + 1) when substrString(S1,(findChar(S1,".",0) +Int 1),lengthString(S1)) =/=String S2
    [anywhere]

  rule excludeFields(S:Schema, .FieldRep1s) => S  [anywhere]
  rule excludeFields(S:Schema, F:FieldRep1,Fs:FieldRep1s) =>
    excludeFields(excludeField(S,F),Fs)  [anywhere]
  rule changeFieldRepIntoStringName(  I1:Id . I2:Id  )  => ((#tokenToString(I1) +String
    ".") +String #tokenToString(I2)) [anywhere]
  rule changeFieldRepIntoStringName(  T:Id , F:Id  )  => ((#tokenToString(T) +String
    ".") +String #tokenToString(F)) [anywhere]

  rule excludeField(s[.Fields] , ` I:Id `) => s[ .Fields ]    [anywhere]
  rule excludeField(s[ f(FN:String,D:DataType,B1:Bool,B2:Bool) , Fs:Fields] , ` I:Id `)
    => excludeField(s[Fs], ` I `) when substrString(FN,(findChar(FN,".",0) +Int
    1),lengthString(FN)) ==String #tokenToString(I) [anywhere]
  rule excludeField(s[ f(FN:String,D:DataType,B1:Bool,B2:Bool) , Fs:Fields] , ` I:Id `)
    => addElement(f(FN,D,B1,B2),excludeField(s[Fs], ` I `)) when
    substrString(FN,(findChar(FN,".",0) +Int 1),lengthString(FN)) =/=String
    #tokenToString(I) [anywhere]
  rule excludeField(s[.Fields] ,  I:Id ) => s[ .Fields ]      [anywhere]
  rule excludeField(s[ f(FN:String,D:DataType,B1:Bool,B2:Bool) , Fs:Fields] ,  I:Id )
    => excludeField(s[Fs], ` I `) when substrString(FN,(findChar(FN,".",0) +Int
    1),lengthString(FN)) ==String #tokenToString(I) [anywhere]
  rule excludeField(s[ f(FN:String,D:DataType,B1:Bool,B2:Bool) , Fs:Fields] , I:Id ) =>
    addElement(f(FN,D,B1,B2),excludeField(s[Fs], ` I `)) when
```

```
 substrString(FN,(findChar(FN,".",0) +Int 1),lengthString(FN)) =/=String
 #tokenToString(I) [anywhere]

rule getValue(_, NULL) => NULL
rule getValue(e(V:Val, _), 0) => V
rule getValue(e(_,Vs:Vals),I:Int) => getValue(e(Vs),I -Int 1)
rule getValue(s[.Fields],e(Vs:Vals),S2:String) => NULL [structural]
rule getValue(s[f(S1:String,D:DataType,B1,B2),Fs:Fields],e(V:Val,Vs:Vals),S2:String)
 => V when S1 ==String S2
rule getValue(s[f(S1:String,D:DataType,B1,B2),Fs:Fields],e(V:Val,Vs:Vals),S2:String)
 => getValue(s[Fs],e(Vs),S2) when S1 =/=String S2
rule select(I:Id[ S:Schema : R:Record ] , E:Exp ) => filter(S,R,E) ~> I[ S : HOLE ]
rule R:Record ~> T[ S : HOLE ] => T[ S : R ]       [structural]
rule  r[ Ts:TableElements ] ~> s(0 , T:TableElement) => r[Ts]
rule  r[ Ts:TableElements ] ~> s(I:Int , T:TableElement) => r[T,Ts] when I =/=Int 0
rule  r[ .TableElements ] ~> s(I:Int, T:TableElement) => r[T] when I =/=Int 0
rule  r[ .TableElements ] ~> s(0, T:TableElement) => r[ .TableElements ]

rule filter( S, r[ .TableElements ] , E ) => r[ .TableElements ]  [structural]
rule filter( S:Schema, r[ T1:TableElement , Ts:TableElements ], E:Exp ) =>
 eval(S,T1,E) ~> s(HOLE,T1) ~> filter(S,r[Ts],E)
rule I:Int ~> s(HOLE,T:TableElement) => s(I,T)
rule S:SelectElement ~> R:Record => R ~> S         [structural]

rule project(T:Id[ S1:Schema : R:Record], S2:Schema) => T[ S2 : project2( R,S1,S2)]
rule project2(r[ .TableElements], S1,S2) => r[ .TableElements]
rule project2(r[ T:TableElement , Ts:TableElements], S1:Schema , S2:Schema) =>
 addElement(project3(T,S1,S2),project2(r[Ts],S1,S2))
rule project3(T:TableElement,S:Schema,s[ .Fields]) => e(.Vals)
rule project3(T:TableElement,S:Schema,s[ f(FN:String,_,_,_), Fs:Fields]) =>
 getValue(T,getIndex(S,FN)) ~> addElementOnTop(project3(T,S,s[Fs]),HOLE)
rule NULL ~> addElementOnTop(T,HOLE) => addElementOnTop(T,NULL)
rule V:Val ~> addElementOnTop(T,HOLE) => addElementOnTop(T,V)

rule eval(_,T,NULL) => NULL
rule eval(_,T,B:Bool) => B
rule eval(_,T,I:Int) => I
rule eval(_,T,S:String) => S
rule eval(S:Schema,T:TableElement,I:Id) => getValue(T,getIndex2(S,#tokenToString(I)))
 ~> eval(S,T,HOLE)
rule eval(S:Schema,T:TableElement,‘ I:Id ‘) =>
 getValue(T,getIndex2(S,#tokenToString(I))) ~> eval(S,T,HOLE)
rule eval(S:Schema,T:TableElement,F:FieldRep) =>
 getValue(T,getIndex(S,changeFieldRepIntoStringName(F))) ~> eval(S,T,HOLE)
rule V:Val ~> eval(S,T,HOLE) => eval(S,T,V) [structural]
rule eval(S,T, - E:Exp) =>  - eval(S,T,E)
rule eval(S,T, E1:Exp * E2:Exp) => eval(S,T, E1) * eval(S,T,E2)
rule eval(S,T, E1:Exp / E2:Exp) => eval(S,T, E1) / eval(S,T,E2)
rule eval(S,T, E1:Exp DIV E2:Exp) => eval(S,T, E1) DIV eval(S,T,E2)
rule eval(S,T, E1:Exp MOD E2:Exp) => eval(S,T, E1) MOD eval(S,T,E2)
rule eval(S,T, E1:Exp % E2:Exp) => eval(S,T, E1) % eval(S,T,E2)
rule eval(S,T, E1:Exp + E2:Exp) => eval(S,T, E1) + eval(S,T,E2)
rule eval(S,T, E1:Exp - E2:Exp) => eval(S,T, E1) - eval(S,T,E2)
rule eval(S,T, ! E:Exp) => ! eval(S,T,E)
rule eval(S,T, E1:Exp = E2:Exp) => eval(S,T, E1) = eval(S,T,E2)
rule eval(S,T, E1:Exp >= E2:Exp) => eval(S,T, E1) >= eval(S,T,E2)
rule eval(S,T, E1:Exp > E2:Exp) => eval(S,T, E1) > eval(S,T,E2)
rule eval(S,T, E1:Exp <= E2:Exp) => eval(S,T, E1) < eval(S,T,E2)
rule eval(S,T, E1:Exp < E2:Exp) => eval(S,T, E1) < eval(S,T,E2)
rule eval(S,T, E1:Exp != E2:Exp) => eval(S,T, E1) != eval(S,T,E2)
rule eval(S,T, E1:Exp <> E2:Exp) => eval(S,T, E1) <> eval(S,T,E2)
```

```
rule eval(S,T, E:Exp IS TRUE) => eval(S,T, E) IS TRUE
rule eval(S,T, E:Exp IS FALSE) => eval(S,T, E) IS FALSE
rule eval(S,T, E:Exp IS NULL) => eval(S,T, E) IS NULL
rule eval(S,T, E:Exp IS UNKNOWN) => eval(S,T, E) IS UNKNOWN
rule eval(S,T, E:Exp IS NOT TRUE) => eval(S,T, E) IS NOT TRUE
rule eval(S,T, E:Exp IS NOT FALSE) => eval(S,T, E) IS NOT FALSE
rule eval(S,T, E:Exp IS NOT NULL) => eval(S,T, E) IS NOT NULL
rule eval(S,T, E:Exp IS NOT UNKNOWN) => eval(S,T, E) IS NOT UNKNOWN
rule eval(S,T, NOT E:Exp ) => NOT eval(S,T, E)
rule eval(S,T, E1:Exp && E2:Exp) => eval(S,T, E1) && eval(S,T,E2)
rule eval(S,T, E1:Exp AND E2:Exp) => eval(S,T, E1) AND eval(S,T,E2)
rule eval(S,T, E1:Exp || E2:Exp) => eval(S,T, E1) || eval(S,T,E2)
rule eval(S,T, E1:Exp OR E2:Exp) => eval(S,T, E1) OR eval(S,T,E2)
rule eval(S,T, CONCAT(Vs:Vals)) => CONCAT(Vs)
rule eval(S,T, ELT(I:Int,Ss:Strings)) => ELT(I,Ss)
rule eval(S,T, FIELD(S1:String,Ss:Strings)) => FIELD(S1,Ss)
rule eval(S,T, INSERT(S1:String,I1:Int,I2:Int,Ss:Strings)) => INSERT(S1,I1,I2,Ss)
rule eval(S,T, INSTR(S1:String,S2:String)) => INSTR(S1,S2)
rule eval(S,T, LENGTH(S1:String)) => LENGTH(S1)
rule eval(S,T, LOCATE(S1:String,S2:String)) => LOCATE(S1,S2)
rule eval(S,T, LOCATE(S1:String,S2:String,I:Int)) => LOCATE(S1,S2,I)
rule eval(S,T, TRIM(S1:String)) => TRIM(S1)
rule eval(S,T, LTRIM(S1:String)) => LTRIM(S1)
rule eval(S,T, RTRIM(S1:String)) => RTRIM(S1)
rule eval(S,T, POSITION(S1:String IN S2:String)) => POSITION(S1 IN S2)
rule eval(S,T, REPEAT(S1:String, I:Int)) => REPEAT(S1,I)
rule eval(S,T, LEFT(S1:String, I:Int)) => LEFT(S1,I)
rule eval(S,T, RIGHT(S1:String, I:Int)) => RIGHT(S1,I)
rule eval(S,T, SPACE(I:Int)) => SPACE(I)
rule eval(S,T, SUBSTRING(S1:String,I:Int)) => SUBSTRING(S1,I)
rule eval(S,T, SUBSTRING(S1:String FROM I:Int)) => SUBSTRING(S1 FROM I)
rule eval(S,T, SUBSTRING(S1:String, I1:Int, I2:Int)) => SUBSTRING(S1,I1,I2)
rule eval(S,T, SUBSTRING(S1:String FROM I1:Int FOR I2:Int)) => SUBSTRING(S1 FROM I1
 FOR I2)

rule getFieldFromSchema(I:Id,s[ f(FN2,T:DataType,B1:Bool,B2:Bool), Fs:Fields]) =>
 f(FN2,T:DataType,B1:Bool,B2:Bool) when #tokenToString(I) ==String
 substrString(FN2,(findChar(FN2,".",0) +Int 1),lengthString(FN2)) [anywhere]
rule getFieldFromSchema(` I:Id `,s[ f(FN2,T:DataType,B1:Bool,B2:Bool), Fs:Fields]) =>
 f(FN2,T:DataType,B1:Bool,B2:Bool) when  #tokenToString(I) ==String
 substrString(FN2,(findChar(FN2,".",0) +Int 1),lengthString(FN2)) [anywhere]
rule getFieldFromSchema(F:FieldRep1,s[ f(FN2,T:DataType,B1:Bool,B2:Bool), Fs:Fields])
 => f(FN2,T:DataType,B1:Bool,B2:Bool) when  changeFieldRepIntoStringName(F) ==String
 FN2  [anywhere]
rule getFieldFromSchema(C:Collumn,s[ f(FN2,T:DataType,B1:Bool,B2:Bool), Fs:Fields])
 => getFieldFromSchema(C,s[Fs]) [anywhere]

rule changeFieldNameCorrespondToTable(T:Id[S:Schema : R:Record]) =>
 T:Id[s[concatFieldName(S,T)] : R] when T =/=K tmp [anywhere]
rule changeFieldNameCorrespondToTable(T:Id[S:Schema : R:Record]) => T:Id[S : R] when
 T ==K tmp [anywhere]

rule   addNullElementOnBottom(T:TableElement,0) => T [anywhere]
rule   addNullElementOnBottom(T:TableElement,I:Int) =>
 addNullElementOnBottom(append(T,e(NULL)), I -Int 1) [anywhere]

rule   addNullElementOnTop(T:TableElement,0) => T [anywhere]
rule   addNullElementOnTop(T:TableElement,I:Int) =>
 addNullElementOnTop(addTopElement(NULL,T), I -Int 1)  [anywhere]

rule   numberOfElementInRecord(r[ .TableElements ]) => 0 [anywhere]
```

```
  rule   numberOfElementInRecord(r[ T:TableElement , Ts:TableElements]) => 1 +
   numberOfElementInRecord(r[Ts]) [anywhere]
  rule   addElementOnBottom(T:TableElement,V:Val) => append(T,e(V)) [anywhere]
  rule   addElementOnTop(e( Vs:Vals ),V:Val) => e(V,Vs) [anywhere]
  rule if(I:Int,K1,K2) => K1 when I =/=Int 0 [anywhere]
  rule if(0,K1,K2) => K2             [anywhere]
  rule I:#Int ~> if(HOLE,K1:K,K2:K) => if(I,K1,K2)  [anywhere]
  rule addElement(T:TableElement,r[Ts:TableElements]) => r[T,Ts] [anywhere]
  rule addElement(F:Field,s[Fs:Fields]) => s[F,Fs]    [anywhere]
  rule concat(s[.Fields], S:Schema) => S          [anywhere]
  rule concat(s[F1:Field,Fs1:Fields],s[Fs2:Fields]) => concat(s[Fs1],s[Fs2]) ~>
   addElement(F1, HOLE ) [anywhere]
rule concat(r[.TableElements], R:Record) => R         [anywhere]
  rule concat(r[T1:TableElement,Tb1:TableElements],r[Tb2:TableElements]) =>
   concat(r[Tb1],r[Tb2]) ~> addElement(T1, HOLE ) [structural]  [anywhere]
  rule S:#Schema ~> addElement(F:Field , HOLE ) => addElement( F, S)
   [anywhere,structural]
  rule R:#Record ~> tmp [S : HOLE ] => tmp [S : R]             [anywhere,structural]
  rule R:#Record ~> addElement( T1:TableElement, HOLE ) => addElement(T1,R)
   [anywhere,structural]
  rule append(e(.Vals),e(Vs:Vals)) => e(Vs)  [anywhere]
  rule append(e(V:Val, Vs1:Vals) ,e(Vs2:Vals)) => append(e(Vs1),e(Vs2)) ~>
   addTopElement(V , HOLE )  [anywhere]
  rule addTopElement(V:Val,e(.Vals)) => e(V)          [anywhere]
  rule addTopElement(V:Val,e(Vs:Vals)) => e(V,Vs) [anywhere]
  rule T:#TableElement ~> addTopElement(V:Val , HOLE) => addTopElement( V , T)
   [structural]
  rule addElement(E:TableElement,r[Es:TableElements]) => r[E,Es]
  rule appendElementToRecord(E1,r[.TableElements]) => r[.TableElements] [structural]
  rule appendElementToRecord(E1:TableElement,r[E2:TableElement,Es:TableElements]) =>
   addElement(append(E1,E2),appendElementToRecord(E1,r[Es]))
  rule numberOfFields(s[.Fields]) => 0 [anywhere,structural]
  rule numberOfFields(s[_, Fs:Fields]) => 1 +Int numberOfFields(s[Fs]) [anywhere]
  rule checkUnionCompatible(s[.Fields] , s[.Fields]) => true [anywhere,structural]
  rule checkUnionCompatible(s[.Fields] , s[_]) => false [anywhere,structural]
  rule checkUnionCompatible(s[_] , s[.Fields]) => false [anywhere,structural]
  rule checkUnionCompatible(s[f(S1:String,T1:DataType,_,_),
   Fs1:Fields],s[f(S2:String,T2:DataType,_,_), Fs2:Fields]) =>
   checkUnionCompatible(s[Fs1],s[Fs2]) when S1 ==K S2 andBool T1 ==K T2 [anywhere]
  rule checkUnionCompatible(s[f(S1:String,T1:DataType,_,_),
   _],s[f(S2:String,T2:DataType,_,_), _]) => false when S1 =/=K S2 orBool T1 =/=K T2
   [anywhere]
  rule isTableElementEqual(e(.Vals),e(.Vals)) => true      [anywhere,structural]
  rule isTableElementEqual(e(V1:Val,Vs1:Vals),e(V2:Val,Vs2:Vals)) => false when V1 =/=K
   V2  [anywhere]
  rule isTableElementEqual(e(V1:Val,Vs1:Vals),e(V2:Val,Vs2:Vals)) =>
   isTableElementEqual(e(Vs1),e(Vs2)) when V1 ==K V2 [anywhere]
  rule consistOf(r[.TableElements] , E2) => false [anywhere,structural]
  rule consistOf(r[E1:TableElement ,Ts:TableElements] , E2) => true when
   isTableElementEqual(E1,E2) [anywhere]
  rule consistOf(r[E1:TableElement ,Ts:TableElements] , E2) =>
   consistOf(r[Ts:TableElements] , E2) when notBool isTableElementEqual(E1,E2)
   [anywhere]
  rule in( S:String , .Ids ) => false [anywhere]
  rule in(S:String , I1:Id , Is:Ids) => true when S ==String #tokenToString(I1)
   [anywhere]
  rule in(S:String , I1:Id , Is:Ids) => in(S, Is) when S =/=String #tokenToString(I1)
   [anywhere]
  rule num(.Ids) => 0
  rule num(_, Xs:Ids) => num(Xs) +Int 1
endmodule
```

A.5 SQL Syntax

```
module SQL-SYNTAX
   imports TABLE
   syntax Table ::= Stm
   syntax Stm ::= CreateStm | InsertStm | SelectStm | UpdateStm | DeleteStm | DropStm
   syntax Stms ::= Stm | Stms Stms           [left,structural]

   syntax Table ::= doGetTableExp(TableExp)              [strict]
   syntax Table ::= doConditionExp(Table,ConditionExp)   [strict]
   syntax Table ::= doProjectionExp(Table,ProjectionExp) [strict]

   // Store
   syntax K ::= "store" Table

   // Create
   syntax CreateStm ::= "CREATE" "TABLE" Id "(" FieldDcls ")" ";"
             | "CREATE" "TABLE" Id "(" FieldDcls "," CreateOptionList ")" ";"
   syntax ProjectionExp ::= "*" | AsClauseOrCollumns
   syntax AsClauseOrCollumn ::= Collumn | AsClause
   syntax AsClauseOrCollumns ::= List{AsClauseOrCollumn,","}
   syntax AsClause ::= Collumn "AS" Collumn
   syntax ConditionExp ::= "WHERE" Exp

   syntax FieldDcl ::= Id DataType
   syntax FieldDcls ::= List{FieldDcl,","}
   syntax CreateOption ::= "PRIMARY" "KEY" "(" Ids ")"
   syntax CreateOptionList ::= List{CreateOption,","}
   syntax K ::= doCreateOption( CreateOptionList , Table)
   syntax Schema ::= createSchemaFromCollumns(AsClauseOrCollumns,Schema)   [strict]
   syntax Fields ::= makeField( FieldDcls )        [strict]
   syntax Table ::= setPrimaryKey( Ids, Table )    [strict]
   syntax Schema ::=  setPrimaryKey( Ids , Schema )      [strict]

   // Select
   syntax SelectStm ::= "SELECT" Exp ";"
              | "SELECT" ProjectionExp TableExp ";"
              | "SELECT" ProjectionExp TableExp ConditionExp ";"

   // Insert
   syntax InsertStm ::= "INSERT" "INTO" Id "(" Ids ")" "VALUES" "(" Vals ")" ";"

   // Delete
   syntax DeleteStm ::= "DELETE" TableExp ConditionExp ";"
   syntax Table ::= doDeleteRecords(Table,Exp)           [strict(1)]
   syntax Record ::= deleteAllWhere(Schema,Record,Exp)   [strict(1,2)]
   syntax Record ::= delete(TableElement,Record)         [strict]

   // Drop
   syntax DropStm ::= "DROP" "TABLE" Ids ";"
   syntax K ::= dropTable(Ids) [strict]

   // Join
   syntax Table ::= getTableFromId(Id)
   syntax Table ::= getTableFromIds(Ids)

   syntax TableExp ::= "FROM" Ids                        [strict]
              | "FROM" JoinExp                           [strict]

   syntax JoinExp ::= Ids "JOIN" Ids                     [strict(1,2)]
```

```
                | Ids "JOIN" Ids JoinCondition            [strict(1,2)]
                | Ids "INNER" "JOIN" Ids                  [strict(1,2)]
                | Ids "INNER" "JOIN" Ids JoinCondition    [strict(1,2)]
                | Ids "CROSS" "JOIN" Ids                  [strict]
                | Ids "CROSS" "JOIN" Ids JoinCondition    [strict]
                | Ids "LEFT" "JOIN" Ids JoinCondition     [strict(1,2)]
                | Ids "LEFT" "OUTER" "JOIN" Ids JoinCondition  [strict(1,2)]
                | Ids "RIGHT" "JOIN" Ids JoinCondition    [strict(1,2)]
                | Ids "RIGHT" "OUTER" "JOIN" Ids JoinCondition [strict(1,2)]
                | Ids "NATURAL" "JOIN" Ids                [strict(1,2)]
                | Ids "NATURAL" "LEFT" "JOIN" Ids         [strict(1,2)]
                | Ids "NATURAL" "LEFT" "OUTER" "JOIN" Ids    [strict(1,2)]
                | Ids "NATURAL" "RIGHT" "JOIN" Ids        [strict(1,2)]
                | Ids "NATURAL" "RIGHT" "OUTER" "JOIN" Ids   [strict(1,2)]
   syntax Table ::= joinUsing(Table,Table,Exp,FieldRep1s)      [strict(1,2,4)]
   syntax Table ::= leftJoinUsing(Table,Table,Exp,FieldRep1s)  [strict(1,2,4)]
   syntax Table ::= rightJoinUsing(Table,Table,Exp,FieldRep1s) [strict(1,2,4)]
   syntax Table ::= naturalJoin(Table,Table)       [strict]
   syntax Table ::= naturalLeftJoin(Table,Table)   [strict]
   syntax JoinCondition ::= OnClause | UsingClause
   syntax Table ::= naturalLeftJoin(Table,Table)   [strict]
   syntax Table ::= naturalRightJoin(Table,Table)  [strict]

   syntax OnClause ::= "ON" Exp
   syntax UsingClause ::= "USING" "(" Collumns ")"
   syntax Exp ::= changeCommonCollumnToEqualExp(Id, Id, FieldRep1s)
   syntax KResult ::= FieldRep | FieldEquality
   syntax FieldEquality ::=  FieldRep "=" FieldRep
   syntax Ids ::= commonField(Schema,Schema) [strict]
   syntax Bool ::= hasCommonField(Schema,Field) [strict]

//     Update
   syntax UpdateStm ::= "UPDATE" Id "SET" AssignValues "WHERE" Exp ";"  [strict(1,2)]
   syntax #AssignValue ::= Id "=" Val
               | String "=" Val
   syntax AssignValue ::=  #AssignValue
                | Id "=" Exp [strict]
                | String "=" Exp [strict]
   syntax KResult ::= #AssignValue | AsClause
   syntax AssignValues ::= List{AssignValue,","}
   syntax Table ::= doUpdateValues(Table,Exp,AssignValues)              [strict(1,3)]
   syntax Record ::= updateAllWhere(Schema, Record, Exp, AssignValues)  [strict(1,2,4)]
   syntax TableElement ::= update(Schema, TableElement, AssignValues )  [strict]
   syntax TableElement ::= update2(Schema, TableElement, AssignValue )  [strict]
   syntax Field ::= changeFieldNameTo(Field,String)  [strict]
endmodule
```

A.6 SQL Semantics

```
module SQL
   imports SQL-SYNTAX

   // Configuration
   configuration  <T color = "red">
                      <k> $PGM:K </k>
                      <env color = "blue"> .Map </env>
                      <store color = "green">
                         <schema color = "pink"> .Map </schema>
                         <record color = "orange"> .Map </record>
```

```
                      </store>
                      <nextloc color = "yellow"> 0 </nextloc>
                   </T>

////**** Main function ****////

   // Delete
   rule DELETE FROM I:Id WHERE E:Exp ; => getTableFromId(I) ~> doDeleteRecords(HOLE,E)
    ~> store HOLE
   rule T:#Table ~> doDeleteRecords(HOLE,E) => doDeleteRecords(T,E)
   rule doDeleteRecords(I:Id[ S:Schema : R:Record ], E:Exp) => I[ S :
    deleteAllWhere(S,R,E)]
   rule deleteAllWhere(S,r[ .TableElements ], E) => r[ .TableElements ] [structural]
   rule deleteAllWhere(S,r[ T:TableElement, Ts:TableElements ], E:Exp) => eval(S,T,E) ~>
    delete(T,deleteAllWhere(S,r[ Ts ], E))
   rule true ~> delete(T,D) => D
   rule false ~> delete(T,D) => addElement(T,D)

   // Update
   rule UPDATE I:Id SET A:AssignValue WHERE E:Exp ; => getTableFromId(I) ~>
    doUpdateValues(HOLE, E, A) ~> store HOLE
   rule T:#Table ~> doUpdateValues(HOLE,E,A) => doUpdateValues(T,E,A)
   rule doUpdateValues(I:Id[ S:Schema : R:Record ],E:Exp,As:AssignValues) => I[ S :
    updateAllWhere(S,R,E,As)]
   rule updateAllWhere(S,r[ .TableElements ],E,As:AssignValues) => r[ .TableElements ]
    [structural]
   rule updateAllWhere(S,r[ T:TableElement, Ts:TableElements],E:Exp,As:AssignValues) =>
    eval(S,T,E) ~> update(S,T,As) ~> addElement(HOLE, updateAllWhere(S,r[Ts],E,As))
    [structural]
   rule true ~> update(S,T,As) ~>  addElement(HOLE, U ) => addElement(update(S,T,As),U)
    [structural]
   rule false ~> update(S,T,As) ~>  addElement(HOLE, U ) => addElement(T,U)  [structural]
   rule update(S:Schema, T:TableElement, .AssignValues) => T
   rule update(S:Schema, T:TableElement,  A:AssignValue , As:AssignValues) =>
    update(S,update2(S,T,A),As)
   rule update2( _, e(.Vals), A ) => e(.Vals) [structural]
   rule update2(s[ f(FName:String,_,_,_), Fs:Fields], e(V:Val , Vs:Vals), F2Name:Id =
    VNew:Val) => e(VNew,Vs) when FName ==String #tokenToString(F2Name)
   rule update2(s[ f(FName:String,_,_,_), Fs:Fields], e(V:Val , Vs:Vals), F2Name:Id =
    VNew:Val) => addTopElement(V,update2(s[Fs], e(Vs), F2Name = VNew)) when FName
    =/=String #tokenToString(F2Name)

   // Drop
   rule DROP TABLE Ts:Ids ; => dropTable(Ts)
   rule dropTable( .Ids ) => .
   rule <k> dropTable(I1:Id, Ids) => dropTable(Ids) </k>
         <env> ... ((I1 => NULL) |-> L:Int) ... </env>
         <store>
            <schema> ... (L |-> (S => NULL)) ... </schema>
            <record> ... (L |-> (R => NULL)) ... </record>
         </store>
   // Get Table
   rule <k> getTableFromId(I:Id) => I[ S : R] ... </k>
         <env> ... (I |-> L:Int) ... </env>
         <store>
            <schema> ... (L |-> S) ... </schema>
            <record> ... (L |-> R) ... </record>
         </store>
   rule getTableFromIds( I:Id ) => getTableFromId(I)        [structural,anywhere]
   rule getTableFromIds(I1:Id, Is:Ids) => join(getTableFromId(I1),getTableFromIds(Is))
    [anywhere]
```

```
  // Store
  rule <k> I:Id[ S:#Schema : R:#Record] ~> store HOLE => . ... </k>
        <env>  ... I |-> L ... </env>
        <store>
           <schema>  ... L |-> (_ => S) ... </schema>
           <record>  ... L |-> (_ => R) ... </record>
        </store>
  rule <k> I:Id[ S:#Schema : R:#Record] ~> store HOLE => . ... </k>
        <env> ... . => I |-> L </env>
        <store>
           <schema> ... . => L |-> S </schema>
           <record> ... . => L |-> R </record>
        </store>
        <nextloc> L:Int => L +Int 1 </nextloc>

  rule store T:Table => T ~> store HOLE [structural]

  // Create
  rule S1:Stms S2:Stms => S1 ~> S2 [structural]
  rule CREATE TABLE TNAME:Id ( FDcls:FieldDcls ) ;  =>
   changeFieldNameCorrespondToTable(TNAME[s[ makeField( FDcls )] : r[ .TableElements ]
   ]) ~> store HOLE
  rule CREATE TABLE Id ( FDcls:FieldDcls , Opt:CreateOptionList ) ;  =>
   changeFieldNameCorrespondToTable(doCreateOption(Opt,Id[s[ makeField( FDcls )] :
   r[.TableElements ]])) ~> store HOLE

  rule makeField(.FieldDcls) => .Fields [structural,anywhere]
  rule makeField(I:Id T:DataType, Dcls:FieldDcls) => f(#tokenToString(I) , T
   ,false,false) , makeField(Dcls) [anywhere]

  rule doCreateOption( .CreateOptionList , T) => T
  rule doCreateOption(PRIMARY KEY (KIds:Ids), Opt:CreateOptionList , T:Table) =>
   doCreateOption(Opt, setPrimaryKey(KIds,T))

  // Insert
  rule <k> (INSERT INTO I:Id(Fs:Ids) VALUES (Vs:Vals) ; => .) ... </k>
        <env> ... I |-> L ... </env>
        <store>
           <schema> ... L |-> S ... </schema>
           <record> ... (L |-> (r[ Es ] => concat(r[Es],r[e(Vs)]))) ... </record>
        </store>

  // Select
  rule SELECT E:Exp ; => E
  rule <k> SELECT P:ProjectionExp T:TableExp ;
               => doGetTableExp(T) ~> doProjectionExp( HOLE , P ) ... </k>
  rule <k> SELECT P:ProjectionExp T:TableExp C:ConditionExp ; => doGetTableExp(T) ~>
   doConditionExp( HOLE , C ) ~> doProjectionExp( HOLE , P ) ... </k>
  rule T:#Table ~> doConditionExp(HOLE,C) => doConditionExp( T , C) [structural]
  rule T:#Table ~> doProjectionExp( HOLE , P) => doProjectionExp(T,P) [structural]
  rule doConditionExp(T:Table , WHERE E:Exp ) => select(T,E)     [anywhere]
  rule doProjectionExp(T:Table, * ) => T              [anywhere]
  rule doProjectionExp(T:Id[ S:Schema : R:Record], As:AsClauseOrCollumns) =>
   project((T[ S : R ]),createSchemaFromCollumns( As,S ))      [anywhere]

////**** Auxiliary function ****////
  rule changeFieldNameTo(f(FN1,D:DataType,B1:Bool,B2:Bool),FN2:String) =>
   f(substrString(FN1,0,(findChar(FN1,".",0) +Int 1)) +String FN2,D,B1,B2) [anywhere]
  rule createSchemaFromCollumns( .AsClauseOrCollumns , S:Schema) => s[ .Fields ]
   [anywhere,structural]
```

```
rule createSchemaFromCollumns(C:Collumn,Cs:AsClauseOrCollumns,S:Schema) =>
     addElement( getFieldFromSchema(C,S) , createSchemaFromCollumns(Cs,S)) [anywhere]
rule createSchemaFromCollumns(C1:Collumn AS I:Id, Cs:AsClauseOrCollumns,S:Schema) =>
     addElement( changeFieldNameTo(getFieldFromSchema(C1,S), #tokenToString(I) ) ,
 createSchemaFromCollumns(Cs,S))                    [anywhere]
rule createSchemaFromCollumns(C1:Collumn AS ‘ I:Id ‘, Cs:AsClauseOrCollumns,S:Schema)
 =>  addElement( changeFieldNameTo(getFieldFromSchema(C1,S), #tokenToString(I) ) ,
 createSchemaFromCollumns(Cs,S))  [anywhere]

rule setPrimaryKey( KIds:Ids, TName:Id[ S : R]) => TName[setPrimaryKey(KIds, S) :
 R]   [anywhere]
rule setPrimaryKey( KIds:Ids , s[.Fields]) => s[.Fields]           [anywhere]
rule setPrimaryKey( KIds:Ids , s[f( S:String , T:DataType , B1, B2 ) , Fs:Fields] )
 =>  addElement( f(S,T,true,B2) , setPrimaryKey(KIds, s[Fs]))  when in(S,KIds)
 [anywhere]
rule setPrimaryKey( KIds:Ids , s[f( S:String , T:DataType , B1, B2 ) , Fs:Fields] )
 =>  addElement( f(S,T,B1,B2) , setPrimaryKey(KIds, s[Fs]))  when notBool in(S,KIds)
 [anywhere]

rule doGetTableExp(FROM Is:Ids) => getTableFromIds(Is)
rule doGetTableExp(FROM Is1:Ids JOIN Is2:Ids) =>
 join(getTableFromIds(Is1),getTableFromIds(Is2))
rule doGetTableExp(FROM Is1:Ids JOIN Is2:Ids ON E:Exp) =>
 join(getTableFromIds(Is1),getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id JOIN Is2:Id USING(Fs:FieldRep1s)) =>
 joinUsing(getTableFromIds(Is1),getTableFromIds(Is2),
 changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)

rule doGetTableExp(FROM Is1:Ids INNER JOIN Is2:Ids) =>
 join(getTableFromIds(Is1),getTableFromIds(Is2))
rule doGetTableExp(FROM Is1:Ids INNER JOIN Is2:Ids ON E:Exp) =>
 join(getTableFromIds(Is1),getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id INNER JOIN Is2:Id USING(Fs:FieldRep1s)) =>
 joinUsing(getTableFromIds(Is1),getTableFromIds(Is2),
 changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)

rule joinUsing(T1:Id[S1:Schema : R1:Record] , T2:Id[S2:Schema : R2:Record] , E:Exp ,
 Fs:FieldRep1s) => join(T1:Id[S1:Schema : R1:Record] , T2:Id[S2:Schema : R2:Record] ,
 E:Exp) ~> project(HOLE,concat(S1,excludeFields(S2,Fs)))
rule T:#Table ~> project(HOLE,S) => project(T,S)         [structural]

rule doGetTableExp(FROM Is1:Ids LEFT JOIN Is2:Ids ON E:Exp) => leftJoin(
 getTableFromIds(Is1), getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id LEFT JOIN Is2:Id USING(Fs:FieldRep1s)) =>
 leftJoinUsing( getTableFromIds(Is1),
 getTableFromIds(Is2),changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)
rule doGetTableExp(FROM Is1:Ids LEFT OUTER JOIN Is2:Ids ON E:Exp) => leftJoin(
 getTableFromIds(Is1), getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id LEFT OUTER JOIN Is2:Id USING(Fs:FieldRep1s)) =>
 leftJoinUsing( getTableFromIds(Is1),
 getTableFromIds(Is2),changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)

rule leftJoinUsing(T1:Id[S1:Schema : R1:Record] , T2:Id[S2:Schema : R2:Record] ,
 E:Exp , Fs:FieldRep1s) => leftJoin(T1:Id[S1:Schema : R1:Record] , T2:Id[S2:Schema :
 R2:Record] , E:Exp) ~> project(HOLE,concat(S1,excludeFields(S2,Fs)))
rule doGetTableExp(FROM Is1:Ids RIGHT JOIN Is2:Ids ON E:Exp) => rightJoin(
 getTableFromIds(Is1),getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id RIGHT JOIN Is2:Id USING(Fs:FieldRep1s)) =>
 rightJoinUsing(
 getTableFromIds(Is1),getTableFromIds(Is2),changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)
rule doGetTableExp(FROM Is1:Ids RIGHT OUTER JOIN Is2:Ids ON E:Exp) => rightJoin(
```

```
 getTableFromIds(Is1),getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id RIGHT OUTER JOIN Is2:Id USING(Fs:FieldRep1s)) =>
 rightJoinUsing(
 getTableFromIds(Is1),getTableFromIds(Is2),changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)
rule rightJoinUsing(T1:Id[S1:Schema : R1:Record] , T2:Id[S2:Schema : R2:Record] ,
 E:Exp , Fs:FieldRep1s) => rightJoin(T1:Id[S1:Schema : R1:Record] , T2:Id[S2:Schema :
 R2:Record] , E:Exp) ~> project(HOLE,concat(excludeFields(S1,Fs),S2))
rule doGetTableExp(FROM Is1:Ids CROSS JOIN Is2:Ids) => catesian(
 getTableFromIds(Is1), getTableFromIds(Is2))
rule doGetTableExp(FROM Is1:Ids CROSS JOIN Is2:Ids ON E:Exp) => join(
 getTableFromIds(Is1), getTableFromIds(Is2),E)
rule doGetTableExp(FROM Is1:Id CROSS JOIN Is2:Id USING(Fs:FieldRep1s)) => joinUsing(
 getTableFromIds(Is1), getTableFromIds(Is2),
 changeCommonCollumnToEqualExp(Is1,Is2,Fs),Fs)
rule doGetTableExp(FROM Is1:Id NATURAL JOIN Is2:Id) =>
 naturalJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule   naturalJoin(T1:Id[ S1:Schema : R1:Record ], T2:Id[ S2:Schema : R2:Record ]) =>
 joinUsing(T1[ S1 : R1],T2[ S2 : R2
 ],changeCommonCollumnToEqualExp(T1,T2,commonField(S1,S2)),commonField(S1,S2))
rule doGetTableExp(FROM Is1:Id NATURAL LEFT JOIN Is2:Id) =>
 naturalLeftJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule doGetTableExp(FROM Is1:Id NATURAL LEFT OUTER JOIN Is2:Id) =>
 naturalLeftJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule   naturalLeftJoin(T1:Id[ S1:Schema : R1:Record ], T2:Id[ S2:Schema : R2:Record
 ]) => leftJoinUsing(T1[ S1 : R1],T2[ S2 : R2
 ],changeCommonCollumnToEqualExp(T1,T2,commonField(S1,S2)),commonField(S1,S2))
rule doGetTableExp(FROM Is1:Id NATURAL RIGHT JOIN Is2:Id) =>
 naturalRightJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule doGetTableExp(FROM Is1:Id NATURAL RIGHT OUTER JOIN Is2:Id) =>
 naturalRightJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule   changeCommonCollumnToEqualExp( T1:Id, T2:Id, .FieldRep1s ) => true [anywhere]
rule   changeCommonCollumnToEqualExp( T1:Id, T2:Id, F1:Id , Fs:FieldRep1s ) =>
 (((T1.F1 = T2.F1):Exp) && (changeCommonCollumnToEqualExp(T1:Id,T2:Id,
 Fs:FieldRep1s))) [anywhere]
rule   changeCommonCollumnToEqualExp( T1:Id, T2:Id , ` F1:Id ` , Fs:FieldRep1s ) =>
 (((T1.F1 = T2.F1):Exp) && (changeCommonCollumnToEqualExp(T1:Id,T2:Id,
 Fs:FieldRep1s))) [anywhere]
rule   naturalLeftJoin(T1:Id[ S1:Schema : R1:Record ], T2:Id[ S2:Schema : R2:Record
 ]) => leftJoinUsing(T1[ S1 : R1],T2[ S2 : R2
 ],changeCommonCollumnToEqualExp(T1,T2,commonField(S1,S2)),commonField(S1,S2))
rule doGetTableExp(FROM Is1:Id NATURAL RIGHT JOIN Is2:Id) =>
 naturalRightJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule doGetTableExp(FROM Is1:Id NATURAL RIGHT OUTER JOIN Is2:Id) =>
 naturalRightJoin(getTableFromIds(Is1),getTableFromIds(Is2))
rule   changeCommonCollumnToEqualExp( T1:Id, T2:Id , ` F1:Id ` , Fs:FieldRep1s ) =>
 (((T1.F1 = T2.F1):Exp) && (changeCommonCollumnToEqualExp(T1:Id,T2:Id,
 Fs:FieldRep1s))) [anywhere]
rule commonField(s[.Fields],S2:Schema) => .Ids  [anywhere]
rule commonField(s[f(FN1:String,D,B1,B2), Fs:Fields],S2:Schema) =>
```

```
    hasCommonField(S2,f(FN1:String,D,B1,B2)) ~>
    if(HOLE,String2Id(substrString(FN1,(findChar(FN1,".",0) +Int 1),lengthString(FN1)))
    , commonField(s[Fs],S2),commonField(s[Fs],S2))  [anywhere]

   rule hasCommonField(s[ .Fields ], _ ) => false [anywhere]
   rule hasCommonField(s[f(FN1:String,D,B1,B2), Fs:Fields],f(FN2:String,_,_,_)) => true
    when substrString(FN1,(findChar(FN1,".",0) +Int 1),lengthString(FN1)) ==String
    substrString(FN2,(findChar(FN2,".",0) +Int 1),lengthString(FN2)) [anywhere]
   rule hasCommonField(s[f(FN1:String,D,B1,B2), Fs:Fields],f(FN2:String,D2,B3,B4)) =>
    hasCommonField(s[Fs],f(FN2,D2,B3,B4)) when substrString(FN1,(findChar(FN1,".",0)
    +Int 1),lengthString(FN1)) =/=String substrString(FN2,(findChar(FN2,".",0) +Int
    1),lengthString(FN2)) [anywhere]
endmodule
```

Bibliography

[1] Mara Alpuente, Marco Antonio Feli, Christophe Joubert, and Alicia Villanueva. Defining datalog in rewriting logic. In Danny De Schreye, editor, *Logic-Based Program Synthesis and Transformation*, volume 6037 of *Lecture Notes in Computer Science*, pages 188–204. Springer Berlin Heidelberg, 2010.

[2] Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey. A computability perspective on self-modifying programs. In *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 231–239, 2009.

[3] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.

[4] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of java programs in javafan. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

[5] Patrick Meredith, Mark Hills, and Grigore Roşu. An Executable Rewriting Logic Semantics of K-Scheme. In Danny Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701*, pages 91–103. Laval University, 2007.

[6] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. Technical Report http://hdl.handle.net/2142/17079, University of Illinois at Urbana Champaign, July 2010.

[7] Yasuhiko Minamide and Shunsuke Mori. Reachability analysis of the html5 parser specification and its application to compatibility testing. In Dimitra Giannakopoulou and Dominique Mry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 293–307. Springer Berlin Heidelberg, 2012.

[8] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Formal semantics of sql queries. *ACM Trans. Database Syst.*, 16(3):513–534, September 1991.

[9] Grigore Rosu. Specifying languages and verifying programs with k. In *Proceedings of 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'13)*, IEEE/CPS. IEEE, September 2013. Invited talk. To appear.

[10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[11] Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 200–212, 2009.