On Verifying a Query Optimizer: A Correctness Proof for a Real-World Query Rewrite Rule

by

Bryan Yongbing Feng

An essay
in fulfilment of the
course requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1994

Abstract

It is generally accepted that rule-based query optimization is a more flexible approach to supporting high-level query languages. However, current practice involves very limited consideration of the issue of rule validity (or correctness). Consequently, the reliability of rule-based query optimization will tend to diminish as both the expressiveness of query languages and complexity of the underlying data models increase.

This has motivated members of the Advanced Database Systems Laboratory at our institution to develop a refinement calculus that enables a formal specification of rewrite rules used in current relational and object-oriented optimization technology. This essay reports on an experiment to apply this calculus to capture the intentions underlying a rule used in an optimizer for an experimental object-oriented database system, also developed in our laboratory, and then to attempt proving the validity of this rule. Perhaps most significantly, we learned from this experiment:

(1) that our original informal understanding of the intentions underlying the rule was incorrect, and

(2) that our first few attempts at a formal specification of this rule were not valid.

We believe that this constitutes clear evidence that the issue of rewrite rule validity for existing query optimization technology has now become crucial in achieving essential levels of reliability in database systems.

Acknowledgements

Thanks go to my supervisor Grant Weddell. Without his patience, support and guidance, the work would never have been done. His careful reading through all the proofs makes the work reliable.

Also, thanks go to my friend Martin van Bommel. His careful proofreading of the earlier drafts of this essay make the final version more readable.

Bryan Yongbing Feng

Contents

List of Tables

1 Introduction

Query optimization continues to be one of the great challenges in post-relational database systems. Since an *ad hoc* query facility is still one of the mandatory features in next generation of database systems [ABD$^+$89], lots of research effort has been devoted to it, among which is work on query rewriting techniques using rule-based query optimization models [BG92, FG91, FG86, Fre87, GD87, GM93, PHH92]. However, unfortunately, two important issues adhering to the rule-based systems - rule formalization (logic) and rule control - are rarely considered in these works.

In the paper [CW93], two leading members of the Advanced Database Systems Laboratory at our institution presented the development of a wide-spectrum algebra and a refinement calculus that enable a formal specification of rewrite rules used in current relational and object-oriented optimization technology. In this essay, I am reporting an experiment to apply this calculus to capture the intentions underlying a rule (see Table 1) used in the optimizer for an experimental object-oriented database system, also developed in our laboratory, and then to prove the validity of this rule. The motivation for the work is as follows: *complex semantic query rewrite rules, instead of simple syntax query rewrite rules, are playing the central role in rule-based query optimizers, and the correctness of semantic rewrite rules are not trivial*.

In the following sections, I will briefly review the wide-spectrum algebra and the refinement calculus over it, introduce some basic concepts on the correctness of the query rewrite rules, prove some basic axioms for the refinement calculus, and finally derive the correctness of the real-world query rewrite rule.

Table 1: A Real-World Query Rewrite Rule (in its original form)

```
(SelectCondSpecialize
    (Find > QInfo > Type
        > ∗ FEList1
        (AndHeap > ∗ PList1)
        (Scan > V > ScanSpec)
        (AndHeap > ∗ PList2 > Pred
            where (Free <q Pred '(< V)) > ∗ PList3)
        > ∗ FEList2)
    (Find < QInfo < Type
        << FEList1
        (AndHeap < Pred << PList1)
        (Scan < V < ScanSpec)
        (AndHeap << PList2 << PList3)
        << FEList2))
```

2 Preliminary

A query language is called wide-spectrum if it can be used uniformly to express both user level queries as well as low-level evaluation strategies or physical access plans. A wide-spectrum query language would uniformly express a query in any stage during query optimization and evaluation, so it provides a logical basis for the query rewrite rule formalization. In the paper [CW93], Dr. Weddell and Dr. Coburn defined such a language (algebra), and further formally defined a refinement calculus over the language. The following is a brief summary of the paper.

In the syntax aspect, the algebra and the calculus are defined in terms of an indexed non-terminal grammer, a context-free grammer with indexed non-terminals (refer to Table 2), which allows high-order logic behaviors. In the semantics aspect, they are based on a simple graph-based data model, following a general trend [Bee89,

Table 2: A Grammar for the Wide-Spectrum Algebra (modified version)

		productions	*description*
R	$\rightarrow$	"(" Clause R R* ")"	Inference constraint.
	|	"(" Ref E E ")"	Refinement constraint.
	|	"(" Neq S S ")"	Syntactic inequality constraint.
	|	"(" Empty F ")"	Syntactic empty set constraint.
	|	. . .	
E	$\rightarrow$	S	Object quantification.
	|	S ":=" Pd	Navigation.
	|	S ":=" S "{" S* "}"	Index scan.
	|	S ":=" S "(" S* ")"	Method call.
	|	S ":" S	Type checking predicate.
	|	S "=" S	Equality predicate.
	|	S "<" S	Ordering predicate.
	|	"(" Keep S* E ")"	Keep.
	|	"(" Filter S* E ")"	Filter.
	|	"(" Not E ")"	Simple compliment.
	|	"(" Cross E* ")"	Cross product.
	|	"(" Perm E ")"	Permutation.
	|	"(" Nest E E ")"	Nested cross product.
	|	"(" Inter E* ")"	Interleave.
	|	"(" Cat E E ")"	Concatenation.
	|	"(" Sort O* E ")"	Sort.
	|	. . .	
F	$\rightarrow$	"(" Bound E ")"	Bounded variable set.
	|	"(" Var E ")"	All variable set.
	|	"(" Union F F ")"	Set union operation.
	|	"(" Intersect F F ")"	Set intersection.
	|	. . .	
Pd	$\rightarrow$	S | Pd "." S	Path description.
O	$\rightarrow$	S D	Sort condition.
D	$\rightarrow$	asc | desc	Sort direction.
S	$\rightarrow$	(a symbol in **S**)	A class, attribute, function or variable name.
$X*$	$\rightarrow$	$\epsilon \mid X \mid X* \; X*$	(where X is one of {R, S, E, O})

Day89, GPvG90], and a possible result semantics in terms of a list of tuples is proposed to the algebra, and further extended to the calculus.

Based on the possible results semantics, a set of possible results is defined for each of the operators in the algebra with respect to the underlying database and a complete tuple. For a query expression E, the interpretation is denoted as: $[\![E]\!]^t_{db}$. For example, the interpretation for the Nest operator is defined as follows:

$$[\![(\text{Nest } E_1\ E_2)]\!]^t_{db} \overset{def}{=} \bigcup_{l \in [\![E_1]\!]^t_{db}} \mathcal{J}(db, E_2, l)$$

where

$$\mathcal{J}(db, E, l_1) \overset{def}{=} \begin{cases} \{()\},\ \text{if } len(l_1) = 0, \\ \{app(l_2, l_3) | l_2 \in [\![E]\!]^{hd(l_1)}_{db} \text{ and } l_3 \in \mathcal{J}(db, E, tl(l_1))\}, \\ \qquad \text{otherwise} \end{cases}$$

For better expressing the query, we have modified the Cross operator by splitting it into two operators: one is still called Cross, and the other one is called Perm. Here we give semantics of the new Cross and Perm operators as follows:
Cross Product "(Cross $E_1, \ldots, E_n$)" adds to its output the union of all the lists of tuples. Each tuple in a given list consists of each possible combination of a tuple from the ith argument list with tuples from the remaining argument lists.

$$[\![(\text{Cross } E_1, \ldots, E_2)]\!]^t_{db} \overset{def}{=} \begin{cases} \{(t)\},\ \text{if } n = 0, \\ \bigcup_{i=1}^{n} [\![(\text{Nest } E_i\ (\text{Cross } E_1, \ldots, E_{i-1}, E_{i+1}, E_n))]\!]^t_{db}, \\ \qquad \text{otherwise.} \end{cases}$$

Perm "(Perm E)" returns all permutation of lists of tuples from its arguments.

$$[\![(\text{Perm } E)]\!]^t_{db} \overset{def}{=} \{l' \in perm(l) | l \in [\![E]\!]^t_{db}\}$$

Further, the semantics of the refinement calculus is defined through the model theory and the proof theory:

Model Theory A database *db* is a model for a ground rule R, written $db \models R$, if and only if one of the three conditions holds.

1. If R is the syntactic constraint "(Neq S_1 S_2)", then S_1 and S_2 are distinct symbols.
2. If R is the refinement constraint "(Ref E_1 E_2)", then $[\![E_2]\!]^t_{db} \subseteq [\![E_1]\!]^t_{db}$ for any complete tuple t over $\mathbf{V}_{db}$.
3. If R is the inference constraint "(Clause R $R_1, \ldots, R_n$)", then $db \models R$ if $db \models R_i$ for each $1 \leq i \leq n$.

A database *db* is a model for a non-ground rule R, also written $db \models R$, if and only if $db \models R'$ for each ground rule R' where $R \overset{*}{\Rightarrow} R'$ (substitution). R is a *logical consequence* of rules $R_1, \ldots, R_n$, written $\{R_1, \ldots, R_n\} \models R$, if and only if, for any database *db*, $db \models R$ if $db \models R_i$ for each $1 \leq i \leq n$. If $\{\} \models R$, R is called an axiom in the calculus.

Proof Theory Let $\{R, R_1, \ldots, R_n\}$ denote an arbitrary collection of rules. There is a derivation of R from $\{R_1, \ldots, R_n\}$, written

$$\{R_1, \ldots, R_n\} \vdash R$$

if and only if $R = R_i$ for some $1 \leq i \leq n$, or can be derived from $\{R_1, \ldots, R_n\}$ using the inference axioms in Table 3.

Table 3: Inference Axioms for the Refinement Calculus

name	*definition*
(substitution)	$\dfrac{R}{R\psi}$
(modus ponens)	$\dfrac{R_1, \ldots, R_n, (\text{Clause } R\ R_1, \ldots, R_n)}{R}$
(known axiom)	$\dfrac{\text{There exists a proof that } \{\} \models R}{R}$

Although the semantics given by the proof theory is sound, but not complete, the calculus could be used in storage definition, generic optimization, database dependency, etc..

For better expressing the rewrite rule, we extend the calculus with an *Empty Set Constraints*, which is:

"(" Empty F ")"

where F denotes a set of symbols, and the semantics for the empty set constraint is straight-forward and simple: a database *db* is a model for an empty set constraint "(Empty F)" if and only if F is an empty set. The union and intersection of sets are defined as usual, and in the next section, I will discuss the two special symbol sets generated by (Bound E) and (Var E).

In general, the definition of the semantics of the refinement calculus provides two ways to prove whether or not a given rule is an axiom: One way is to use the model theory to prove an axiom by referring to the semantics of the wide-spectrum algebra, and the other way is to use the proof theory to derive an axiom from a set of known axioms. The first way is mandatory to prove the basic axioms, however is not the proper way to prove a complex axiom, since it seldom use existing axioms, and the proof principle is irregular, which results that the proof is out of control in both size and complexity. On the other hand, although the proof theory is not complete, the second way is more preferable if we could classify a set of basic axioms, because the proof process could be done in a mechanical manner. In this essay, I will use the first way to prove some basic axioms, and then show the second way by deriving the correctness of a real-world rewrite rule.

3 Read/Write Sets for A Query

In order to explore the proof process of the axioms in the refinement calculus, it is necessary to define some basic concepts for the wide-spectrum algebra, and to examine the related properties.

Given a ground query, we are interested in two kinds of symbol sets, as defined below.

DEFINITION **3.1** *A set of variables $\{S_1, \ldots, S_n\}$ is called a write set for a ground query E if and only if for any database db and any complete tuple t over $\mathbf{V}_{db}$, for each $l \in [\![E]\!]^t_{db}$*

$$nth(l, i)[\alpha(t) - \{S_1, \ldots, S_n\}] = t[\alpha(t) - \{S_1, \ldots, S_n\}],$$

for $1 \leq i \leq len(l)$.

$\square$

Table 4: Bound Set and Var Set for the Wide-Spectrum Algebra

E	*(Bound E)*	*(Var E)*
S	$\{S\}$	$\{S\}$
$S := Pd$	$\{S\}$	$\{S\} \cup head(Pd)^*$
$S := S'\{S_1, \ldots, S_n\}$	$\{S\}$	$\{S, S', S_1, \ldots, S_n\}$
$S := S'(S_1, \ldots, S_n)$	$\{S\}$	$\{S, S', S_1, \ldots, S_n\}$
$Pd : \{S_1, \ldots, S_n\}$	$\{\}$	$\{S_1, \ldots, S_n\} \cup head(Pd)$
$Pd = Pd'$	$\{\}$	$head(Pd) \cup head(Pd')$
$Pd < Pd'$	$\{\}$	$head(Pd) \cup head(Pd')$
(Keep $S_1, \ldots, S_n$ E)	(Bound E) $\cap \{S_1, \ldots, S_n\}$	(Var E) $\cup \{S_1, \ldots, S_n\}$
(Filter $S_1, \ldots, S_n$ E)	(Bound E)	(Var E) $\cup \{S_1, \ldots, S_n\}$
(Not E)	(Bound E)	(Var E)
(Cross $E_1, \ldots, E_n$)	$\bigcup_{1 \leq i \leq n}$(Bound E_i)	$\bigcup_{1 \leq i \leq n}$(Var E_i)
(Nest E_1 E_2)	(Bound E_1) $\cup$ (Bound E_2)	(Var E_1) $\cup$ (Var E_2)
(Inter $E_1, \ldots, E_n$)	$\bigcup_{1 \leq i \leq n}$(Bound E_i)	$\bigcup_{1 \leq i \leq n}$(Var E_i)
(Cat E_1 E_2)	(Bound E_1) $\cup$ (Bound E_2)	(Var E_1) $\cup$ (Var E_2)
(Sort S_1 $D_1, \ldots, S_n$ D_n E)	(Bound E)	(Var E) $\cup \{S_1, \ldots, S_n\}$

DEFINITION **3.2** *A set of variables $\{S_1, \ldots, S_n\}$ is called a read set for a ground query E if and only if for any database db and any pair of complete tuples t_1 and t_2 over $\mathbf{V}_{db}$, for which $t_1[\{S_1, \ldots, S_n\}] = t_2[\{S_1, \ldots, S_n\}]$, the following statement holds:*

$$\{l_1[\{S_1, \ldots, S_n\}] | l_1 \in [\![E]\!]^{t_1}_{db}\} = \{l_2[\{S_1, \ldots, S_n\}] | l_2 \in [\![E]\!]^{t_2}_{db}\}.$$

$\square$

*Only the first symbol in the path sequence is bounded to a vertex. Here $head(Pd)$ denotes the singleton set of the first symbol in Pd.

Also, for a given ground query, there are two syntax symbol sets associated with it, which are the *Bound* set and the *Var* set, respectively.

DEFINITION **3.3** *Given a ground query E,* (Bound E) *and* (Var E) *are defined recursively in Table 4, respectively.*

□

The following theorem shows that the two special sets fall into the two interesting catalogs, respectively.

THEOREM **3.1** *For any ground query E,* (Bound E) *is a write set, and* (Var E) *is both a write set and a read set.*

PROOF:

First, we prove that (Bound E) *and* (Var E) *are a write set and a read set, respectively, by the induction on the depth of the query (tree).*

BASIS. The depth of the query tree is 1, and from the definition table, it is easy to see there are seven kinds of queries falling into this category. Due to the space limit, we only take one kind of query as an example to prove. The others can be proved similarly.

Suppose that the query is $S := Pd$, then according to the definition, for any database db and any complete tuple t over $\mathbf{V}_{db}$,

$$[\![S := Pd]\!]^t_{db} = perm(\mathcal{F}(S, list(\{v \in \mathbf{V}_{db} | (db, t, Pd) \rightsquigarrow v\})) \prec t).$$

So, for each $l \in [\![S := Pd]\!]^t_{db}$, by definition of $\mathcal{F}$ and $\prec$,

$$nth(l, i)[\alpha(t) - \{S\}] = t[\alpha(t) - \{S\}],$$

for $1 \leq i \leq len(l)$. Because (Bound $S := Pd$) $= \{S\}$, it is a write set.

For any database db and any pair of complete tuples t_1 and t_2 over $\mathbf{V}_{db}$, if

$$t_1[head(Pd)] = t_2[head(Pd)],$$

then

$$\{v_1 \in \mathbf{V}_{db} | (db, t_1, Pd) \rightsquigarrow v_1\} = \{v_2 \in \mathbf{V}_{db} | (db, t_2, Pd) \rightsquigarrow v_2\}.$$

So, according to the definition,

$$\mathcal{F}(S, list(\{v_1 \in \mathbf{V}_{db} | (db, t_1, Pd) \rightsquigarrow v_1\})) = \mathcal{F}(S, list(\{v_2 \in \mathbf{V}_{db} | (db, t_2, Pd) \rightsquigarrow v_2\}))$$
$$\{l_1[\{S\}] | l_1 \in [\![S := Pd]\!]^{t_1}_{db}\} = \{l_2[\{S\}] | l_2 \in [\![S := Pd]\!]^{t_2}_{db}\}.$$

Now, let's consider the projection on the $head(Pd)$ set. If $head(Pd) = \{S\}$, (Var $S := Pd$) $= \{S\}$. Therefore, (Var $S := Pd$) is a read set by definition. Otherwise, $head(Pd) \cap$ (Bound $S := Pd$) $= \emptyset$, so $head(Pd) \subseteq (\alpha(t) -$ (Bound $S := Pd$)). For any complete tuple t over $\mathbf{V}_{db}$, and for each $l \in [\![S := Pd]\!]^{t}_{db}$,

$$nth(l, i)[head(Pd)] = t[head(Pd)], \text{ for all } 1 \leq i \leq len(l).$$

Therefore,

$$\{l_1[\{S\} \cup head(Pd)] | l_1 \in [\![S := Pd]\!]^{t_1}_{db}\} = \{l_2[\{S\} \cup head(Pd)] | l_2 \in [\![S := Pd]\!]^{t_2}_{db}\}.$$

Because $t_1[(\text{Var } S := Pd)] = t_2[(\text{Var } S := Pd)]$ implies $t_1[head(Pd)] = t_2[head(Pd)]$, and (Var $S := Pd$) $= \{S\} \cup head(Pd)$, therefore,

$$\{l_1[(\text{Var } S := Pd)] | l_1 \in [\![S := Pd]\!]^{t_1}_{db}\} = \{l_2[(\text{Var } S := Pd)] | l_2 \in [\![S := Pd]\!]^{t_2}_{db}\}.$$

The basis holds.

Induction. Assume the conclusion is true for all the query trees whose depth are less than n, and now we consider the case that the query tree has depth of $n, n > 1$.

Now, the query tree may be one of eight kinds. Due to the space limit, we also only prove one kind as an example, and leave others to interested readers. Here, we consider the Keep operator as the root of the query tree, i.e., the query E' is of form:

$$(\text{Keep } S_1, \ldots, S_n\ E),$$

and the corresponding Bound and Var sets are,

$$(\text{Bound } E') = (\text{Bound } E) \cap \{S_1, \ldots, S_n\},$$

$$(\text{Var } E') = (\text{Var } E) \cup \{S_1, \ldots, S_n\}.$$

According to the definition, for any database db and any complete tuple t over $\mathbf{V}_{db}$,

$$[\![(\text{Keep } S_1, \ldots, S_n \ E)]\!]^t_{db} = \{l' | \exists l \in [\![E]\!]^t_{db} \text{ such that } l' = (l[S_1, \ldots, S_n] \prec t)\},$$

therefore, for each $l' \in [\![(\text{Keep } S_1, \ldots, S_n \ E)]\!]^t_{db}$,

$$nth(l', i)[\alpha(t) - \{S_1, \ldots, S_n\}] = t[\alpha(t) - \{S_1, \ldots, S_n\}].$$

Because the depth of E is less than n, by assumption, (Bound E) is a write set, i.e., for any database db and any complete tuple t over $\mathbf{V}_{db}$, for each $l \in [\![E]\!]^t_{db}$,

$$nth(l, i)[\alpha(t) - (\text{Bound } E)] = t[\alpha(t) - (\text{Bound } E)].$$

For each $l' \in [\![(\text{Keep } S_1, \ldots, S_n \ E)]\!]^t_{db}$, there exists $l \in [\![E]\!]^t_{db}$ such that

$$l' = (l[S_1, \ldots, S_n] \prec t),$$

therefore,

$$nth(l', i)[\alpha(t) - (\text{Bound } E)] = t[\alpha(t) - (\text{Bound } E)].$$

Thus, by set theory,

$$nth(l', i)[\alpha(t) - (\text{Bound } E) \cap \{S_1, \ldots, S_n\}] = t[\alpha(t) - (\text{Bound } E) \cap \{S_1, \ldots, S_n\}],$$

i.e., (Bound E') is a write set.

Now, we try to prove that (Var E') is a read set. For any database db, let t_1 and t_2 are two complete tuples over $\mathbf{V}_{db}$ such that $t_1[(\text{Var } E')] = t_2[(\text{Var } E')]$, i.e.,

$$t_1[(\text{Var } E) \cup \{S_1, \ldots, S_n\}] = t_2[(\text{Var } E) \cup \{S_1, \ldots, S_n\}],$$

which implies

$$t_1[(\text{Var } E)] = t_2[(\text{Var } E)].$$

By assumption,

$$\{l_1[(\text{Var } E)] | l_1 \in [\![E]\!]^{t_1}_{db}\} = \{l_2[(\text{Var } E)] | l_2 \in [\![E]\!]^{t_2}_{db}\}.$$

Further,

$$\{l_1[(\text{Var } E) \cup \{S_1, \ldots, S_n\}] | l_1 \in [\![E]\!]^{t_1}_{db}\} = \{l_2[(\text{Var } E) \cup \{S_1, \ldots, S_n\}] | l_2 \in [\![E]\!]^{t_2}_{db}\}.$$

By the definition of the Keep operator,

$$[\![E']\!]_{db}^{t_1} = [\![E']\!]_{db}^{t_2},$$

so,

$$\{l_1[(\text{Var } E) \cup \{S_1, \ldots, S_n\}] | l_1 \in [\![E']\!]_{db}^{t_1}\} = \{l_2[(\text{Var } E) \cup \{S_1, \ldots, S_n\}] | l_2 \in [\![E']\!]_{db}^{t_2}\},$$

which means that (Var E') is a read set.

By the basis and induction part, we can draw the conclusion that for a ground query E, (Bound E) and (Var E) are a write set and a read set, respectively.

*Now, we prove that (*Var *E) is a write set, too.*

By the definition of the Bound and Var set, for any ground query E,

$$(\text{Bound } E) \subseteq (\text{Var } E),$$

so, for any database db and any complete tuple t over $\mathbf{V}_{db}$,

$$(\alpha(t) - (\text{Var } E)) \subseteq (\alpha(t) - (\text{Bound } E)).$$

(Bound E) is a write set, so for any database db and any complete tuple t over $\mathbf{V}_{db}$, for each $l \in [\![E]\!]_{db}^{t}$,

$$nth(l, i)[\alpha(t) - (\text{Bound } E)] = t[\alpha(t) - (\text{Bound } E)],$$

for each $1 \leq i \leq len(l)$, which implies

$$nth(l, i)[\alpha(t) - (\text{Var } E)] = t[\alpha(t) - (\text{Var } E)],$$

for each $1 \leq i \leq len(l)$.

Thus, (Var E) is a write set, too.

$\square$

4 Two Important Axioms for the Nest Operator

Since the query algebra is operator-oriented, to explore the axioms for the refinement calculus, and finally to analyze the correctness for rule-based query optimization, it is beneficial to examine the basic properties of each operator. In this section, we focus on one important operator, **Nest**, and examine its associativity and commutativity.

4.1 Associativity for the Nest Operator

LEMMA **4.1** *For any database db, any query E, and any pair of lists l_1 and l_2,*

$$\mathcal{J}(db, E, app(l_1, l_2)) = \{app(l'_1, l'_2) | l'_1 \in \mathcal{J}(db, E, l_1) \textit{ and } l'_2 \in \mathcal{J}(db, E, l_2)\}.$$

PROOF: By induction of length of list l_1.

BASIS. Trivial for $len(l_1) = 0$.

INDUCTION. True for $len(l_1) < n$. Consider when $len(l_1) = n$.

By definition,

$$\mathcal{J}(db, E, app(l_1, l_2)) = \{app(l_3, l_4) | l_3 \in [\![E]\!]_{db}^{hd(l_1)} \text{ and } l_4 \in \mathcal{J}(db, E, app(tl(l_1), l_2)\}.$$

Applying the assumption,

$$l_4 \in \{app(l_5, l'_2) | l_5 \in \mathcal{J}(db, E, tl(l_1)) \text{ and } l'_2 \in \mathcal{J}(db, E, l_2)\},$$

therefore,

$$app(l_3, l_4) = app(l_3, app(l_5, l'_2)) = app(app(l_3, l_5), l'_2).$$

Also by definition,

$$app(l_3, l_5) \in \mathcal{J}(db, E, l_1).$$

Let it be l'_1, then the lemma is true for $len(l) = n$. □

LEMMA **4.2** *For any database db, any pair of queries E_1 and E_2, and any list l,*

$$\mathcal{J}(db, (Nest\ E_1\ E_2), l) = \bigcup_{l' \in \mathcal{J}(db,E_1,l)} \mathcal{J}(db, E_2, l').$$

Proof: By induction of length of list l.

basis. Trivial for $len(l) = 0$.

induction. True for $len(l) < n$. Consider when $len(l) = n$.

By definition,

$$\begin{aligned} & \mathcal{J}(db, (\text{Nest } E_1\ E_2), l) \\ = \quad & \{app(l_1, l_2) | l_1 \in [\![(\text{Nest } E_1\ E_2)]\!]_{db}^{hd(l)} \text{ and } l_2 \in \mathcal{J}(db, (\text{Nest } E_1\ E_2), tl(l))\}. \end{aligned}$$

Further by definition,

$$[\![(\text{Nest } E_1\ E_2)]\!]_{db}^{hd(l)} = \bigcup_{l_3 \in [\![E_1]\!]_{db}^{hd(l)}} \mathcal{J}(db, E_2, l_3).$$

Applying the assumption,

$$\mathcal{J}(db, (\text{Nest } E_1\ E_2), tl(l)) = \bigcup_{l_4 \in \mathcal{J}(db,E_1,tl(l))} \mathcal{J}(db, E_2, l_4).$$

So, by set theory,

$$\begin{aligned} & \mathcal{J}(db, (\text{Nest } E_1\ E_2), l) = \\ & \bigcup_{l_3 \in [\![E_1]\!]_{db}^{hd(l)} \text{ and } l_4 \in \mathcal{J}(db,E_1,tl(l))} \{app(l_1, l_2) | l_1 \in \mathcal{J}(db, E_2, l_3) \text{ and } l_2 \in \mathcal{J}(db, E_2, l_4)\}. \end{aligned}$$

By Lemma 4.1,

$$\mathcal{J}(db, (\text{Nest } E_1\ E_2), l) = \bigcup_{l_3 \in [\![E_1]\!]_{db}^{hd(l)} \text{ and } l_4 \in \mathcal{J}(db,E_1,tl(l))} \mathcal{J}(db, E_2, app(l_3, l_4)).$$

Let l' be $app(l_3, l_4)$, then we get

$$\mathcal{J}(db, (\text{Nest } E_1\ E_2), l) = \bigcup_{l' \in \mathcal{J}(db,E_1,l)} \mathcal{J}(db, E_2, l')).$$

Therefore, the lemma is true for $len(l) = n$. $\square$

Theorem **4.1** *Associativity holds for the operator* Nest*, i.e.,*

(Ref (Nest (Nest $E1$ $E2$) $E3$) (Nest $E1$ (Nest $E2$ $E3$)))

(Ref (Nest $E1$ (Nest $E2$ $E3$)) (Nest (Nest $E1$ $E2$) $E3$))

both are axioms.

PROOF: By the definition of axiom.

For any database db and any complete tuple t over $\mathbf{V}_{db}$, by definition,

$$\begin{aligned}
[\![(\text{Nest }(\text{Nest } E_1\ E_2)\ E_3)]\!]^t_{db} &= \bigcup_{l\in[\![(\text{Nest } E_1\ E_2)]\!]^t_{db}} \mathcal{J}(db, E_3, l) \\
&= \bigcup_{l\in\bigcup_{l'\in[\![E_1]\!]^t_{db}} \mathcal{J}(db,E_2,l')}(db, E_3, l) \\
&= \bigcup_{l'\in[\![E_1]\!]^t_{db}} (\bigcup_{l\in\mathcal{J}(db,E_2,l')}(db, E_3, l))
\end{aligned}$$

$$[\![(\text{Nest } E_1\ (\text{Nest } E_2\ E_3))]\!]^t_{db} = \bigcup_{l'\in[\![E_1]\!]^t_{db}} \mathcal{J}(db, (\text{Nest } E_2\ E_3), l')$$

By Lemma 4.2,

$$[\![(\text{Nest }(\text{Nest } E_1\ E_2)\ E_3)]\!]^t_{db} = [\![(\text{Nest } E_1\ (\text{Nest } E_2\ E_3))]\!]^t_{db}$$

Therefore, the two reference rules are axioms.

□

4.2 Commutativity for the Nest Operator

Unlike associativity, in general, commutativity does not hold for the Nest operator. However, a nice property holds under certain conditions.

LEMMA **4.3** *Given two expressions E_1 and E_2 such that*

$$(\text{Bound } E_1) \cap (\text{Var } E_2) = \emptyset\ ,$$

then for any database db and any complete tuple t over $\mathbf{V}_{db}$, for each $l \in [\![E_1]\!]^t_{db}$,

$$\{l'[(\text{Var } E_2)] \prec t | l' \in [\![E_2]\!]_{db}^{nth(l,i)}\} = [\![E_2]\!]_{db}^{t}$$

for $1 \leq i \leq len(l)$.

Proof: According to Theorem 3.1, for each $l \in [\![E_1]\!]_{db}^{t}$, for any $1 \leq i \leq len(l)$,

$$nth(l,i)[\alpha(t) - (\text{Bound } E_1)] = t[\alpha(t) - (\text{Bound } E_1)].$$

Because E_1 and E_2 satisfy the *emptyset* condition

$$(\text{Bound } E_1) \cap (\text{Var } E_2) = \emptyset,$$

we have

$$(\text{Var } E_2) \subseteq \alpha(t) - (\text{Bound } E_1),$$

so,

$$nth(l,i)[(\text{Var } E_2)] = t[(\text{Var } E_2)].$$

Also by Theorem 3.1,

$$\{l'[(\text{Var } E_2)] \prec t | l' \in [\![E_2]\!]_{db}^{nth(l,i)}\} = \{l''[(\text{Var } E_2)] \prec t | l'' \in [\![E_2]\!]_{db}^{t}\}.$$

Because $l'' \in [\![E_2]\!]_{db}^{t}$, we know that

$$l''[(\text{Var } E_2)] \prec t = l'',$$

thus

$$\{l'[(\text{Var } E_2)] \prec t | l' \in [\![E_2]\!]_{db}^{nth(l,i)}\} = [\![E_2]\!]_{db}^{t}.$$

□

Lemma **4.4** *Given two ground queries E_1 and E_2 such that*

$$(\text{Bound } E_1) \cap (\text{Var } E_2) = \emptyset,$$

for any database db and any complete tuple t over $\mathbf{V}_{db}$, for a list l satisfying that there exists l' and $app(l', l) \in [\![E_1]\!]_{db}^{t}$,

$$\mathcal{J}(db, E_2, l) \neq \{()\} \textit{ if and only if } [\![E_2]\!]_{db}^{nth(l,i)} \neq \{()\},$$

for each $1 \leq i \leq len(l)$.

Proof:

if. For each $1 \leq i \leq len(l)$, $[\![E_2]\!]_{db}^{nth(l,i)} \neq \{()\}$.

And by the definition of $\mathcal{J}$,

$$\mathcal{J}(db, E_2, l) = \begin{cases} \{()\} & \text{if } len(l) = 0 \\ \{app(l_1, l_2) | l_1 \in [\![E_2]\!]_{db}^{hd(l)} \text{ and } l_2 \in \mathcal{J}(db, E_2, tl(l))\} & \text{otherwise} \end{cases}$$

so, it is trivial to conclude $\mathcal{J}(db, E_2, l) \neq \{()\}$.

only if. $\mathcal{J}(db, E_2, l) \neq \{()\}$.

By definition, there must exist a tuple in l, say $nth(l, j)$ for some $1 \leq j \leq len(l)$, which satisfies

$$[\![E_2]\!]_{db}^{nth(l,j)} \neq \{()\}.$$

By Theorem 3.1, and the condition for l, there exists l' such that $app(l', l) \in [\![E_1]\!]_{db}^{t}$,

$$nth(l, i)[\alpha(t) - (\text{Bound } E_1)] = t[\alpha(t) - (\text{Bound } E_1)],$$

for $1 \leq i \leq len(l)$.

Because of the empty condition,

$$(\text{Bound } E_1) \cap (\text{Var } E_2) = \emptyset,$$

the above statement can be written as,

$$nth(l, i)[(\text{Var } E_2)] = t[(\text{Var } E_2)],$$

for $1 \leq i \leq len(l)$. Further, by Lemma 4.3,

$$\{l'[(\text{Var } E_2)] \prec t | l' \in [\![E_2]\!]_{db}^{nth(l,i)}\} = [\![E_2]\!]_{db}^{t},$$

for $1 \leq i \leq len(l)$.

Therefore, $[\![E_2]\!]_{db}^{nth(l,j)} \neq \{()\}$, implies $[\![E_2]\!]_{db}^{nth(l,i)} \neq \{()\}$, for for $1 \leq i \leq len(l)$.

$\square$

LEMMA **4.5** *Let E_1, E_2, db, t and l_1 denote a pair of ground queries, a database, a complete tuple and a list of tuples, respectively, which satisfy the following conditions:*

C1. $(\text{Bound } E_1) \cap (\text{Var } E_2) = \emptyset$,

C2. $db \models (\mathit{Ref}\, E_2\ (\mathit{Filter}\ E_2))$,

C3. $l_1 \in \{l'_1 | \exists l''_1 \ s.t. \ app(l''_1, l'_1) \in [\![E_1]\!]^t_{db}\}$,

If $\mathcal{J}(db, E_2, l_1) \neq \{()\}$, then for any non-empty $l_2 \in [\![E_2]\!]^t_{db}$, there exists $l \in \mathcal{J}(db, E_2, l_1)$ such that

C4. $l[\alpha(t) - (\text{Bound } E_2)] \prec t = l_1$,

C5. $nth(l, i)[(\text{Var } E_2)] = hd(l_2)[(\text{Var } E_2)]$, *for all* $1 \leq i \leq len(l)$.

PROOF: By induction on the length of l_1.

BASIS. There are two basic cases here. $len(l_1) = 0$ and $len(l_1) = 1$. If $len(l_1) = 0$, the lemma is trivial by the definition of the $\mathcal{J}$ function. Now, consider $len(l_1) = 1$, i.e., l_1 consists of a single tuple, and we have $tl(l_1) = ()$.

By definition,

$$\mathcal{J}(db, E_2, l_1) = [\![E_2]\!]^{hd(l_1)}_{db}.$$

Based on C2, we have, for any complete tuple t over $\mathbf{V}_{db}$, for each $l_2 \in [\![E_2]\!]^t_{db}$,

$$len(l_2) = 0, \text{ or } len(l_2) = 1,$$

which means that if $\mathcal{J}(db, E_2, l_1) \neq \{()\}$, there exists at least a non-empty list $l \in \mathcal{J}(db, E_2, l_1)$, and for any non-empty list $l \in \mathcal{J}(db, E_2, l_1)$, $len(l) = 1$.

By Theorem 3.1 and C3, for any complete tuple t over $\mathbf{V}_{db}$,

$$hd(l_1)[\alpha(t) - (\text{Bound } E_1)] = t[\alpha(t) - (\text{Bound } E_1)],$$

and adding C1,

$$hd(l_1)[(\text{Var } E_2)] = t[(\text{Var } E_2)],$$

then by Lemma 4.3,

$$\{l[(\text{Var } E_2)] \prec t | l \in [\![E_2]\!]_{db}^{hd(l_1)}\} = [\![E_2]\!]_{db}^{t}.$$

Now, for any non-empty list $l_2 \in [\![E_2]\!]_{db}^{t}$, there exists a non-empty list $l \in [\![E_2]\!]_{db}^{hd(l_1)}$ such that

$$hd(l)[(\text{Var } E_2)] = hd(l_2)[(\text{Var } E_2)],$$

and by Theorem 3.1,

$$hd(l)[\alpha(t) - (\text{Bound } E_2)] \prec t = hd(l_1).$$

Because $len(l) = 1$, conclusions C4 and C5 hold.

So, the basis is true.

Induction. Assume the lemma is true for $len(l_1) < n$, and now consider the case when $len(l_1) = n$.

If $\mathcal{J}(db, E_2, l_1) \neq \{()\}$, by Lemma 4.4,

$$[\![E_2]\!]_{db}^{hd(l_1)} \neq \{()\}, \text{ and } \mathcal{J}(db, E_2, tl(l_1)) \neq \{()\}.$$

Because $tl(l_1)$ still satisfies C3, by the assumption, if $\mathcal{J}(db, E_2, tl(l_1)) \neq \{()\}$, then for any non-empty $l_2 \in [\![E_2]\!]_{db}^{t}$, there exists $l'' \in \mathcal{J}(db, E_2, tl(l_1))$ such that

$$l''[\alpha(t) - (\text{Bound } E_2)] \prec t = tl(l_1),$$
$$nth(l'', i)[(\text{Var } E_2)] = hd(l_2)[(\text{Var } E_2)],$$

for all $1 \leq i \leq len(l'')$.

And similar to the basis for $len(l) = 1$, if $[\![E_2]\!]_{db}^{hd(l_1)} \neq \{()\}$, then

$$\{l'[(\text{Var } E_2)] \prec t | l' \in [\![E_2]\!]_{db}^{hd(l_1)}\} = [\![E_2]\!]_{db}^{t}.$$

Now, for any non-empty list $l_2 \in [\![E_2]\!]_{db}^{t}$, there exists a non-empty list $l' \in [\![E_2]\!]_{db}^{hd(l_1)}$ such that

$$len(l') = 1,$$

$$hd(l')[(\text{Var } E_2)] = hd(l_2)[(\text{Var } E_2)],$$

and by Theorem 3.1,

$$hd(l')[\alpha(t) - (\text{Bound } E_2)] \prec t = hd(l_1).$$

Now, we fix l_2, and let $l = app(l', l'')$. By the definition of $\mathcal{J}$, $l \in \mathcal{J}(db, E_2, l_1)$, and $len(l) \neq 0$. Further, it is easy to see that l satisfies C4 and C5. So, the induction is true.

By the basis and the induction, we can draw the conclusion that the lemma is true.

□

Now, it is time to state and prove the commutative property for the Nest operator.

Theorem **4.2** *The inference rule*

```
( Clause   (Ref (Nest E1 E2) (Nest E2 E1))
           (Ref E2 (Filter E2))
           (Empty (Intersect (Bound E1) (Var E2)))
           (Empty (Intersect (Var E1) (Bound E2))))
```

is an axiom.

Proof: By contraction.

If the rewrite rule is not an axiom, there exist a database db and a ground rule R, which is of the form:

(Clause (Ref (Nest E_1 E_2) (Nest E_2 E_1))
(Ref E_2 (Filter E_2))
(Empty (Intersect (Bound E_1) (Var E_2)))
(Empty (Intersect (Var E_1) (Bound E_2)))),

such that for some complete tuple t,

1. $db \models$ (Ref E_2 (Filter E_2))
2. $db \models$ (Empty (Intersect (Bound E_1) (Var E_2)))
3. $db \models$ (Empty (Intersect (Var E_1) (Bound E_2))), and
4. $[\![(\text{Nest } E_2\ E_1)]\!]^t_{db} \not\subseteq [\![(\text{Nest } E_1\ E_2)]\!]^t_{db}$.

Now, we try to contradict condition 4, based on condition 1, 2 and 3. Here, the empty list and the non-empty lists are considered separately.

Empty list. For any database db and any complete tuple t over $\mathbf{V}_{db}$, if there exists an empty list in $[\![(\text{Nest } E_2\ E_1)]\!]^t_{db}$, according to the definition of operator Nest,

$$[\![(\text{Nest } E_2\ E_1)]\!]^t_{db} = \bigcup_{l_2 \in [\![E_2]\!]^t_{db}} \mathcal{J}(db, E_1, l_2),$$

there must exist $l_2 \in [\![E_2]\!]^t_{db}$ such that $() \in \mathcal{J}(db, E_1, l_2)$. Further, by function $\mathcal{J}$, l_2 could be an empty list or a non-empty list.

l_2 is an empty list. By Lemma 4.3, there exists a list $l_1 \in [\![E_1]\!]^t_{db}$,

$$\{l'[(\text{Var } E_2)] \prec t | l' \in [\![E_2]\!]^{nth(l_1,i)}_{db}\} = [\![E_2]\!]^t_{db},$$

which implies

$$() \in [\![E_2]\!]^{nth(l,i)}_{db},$$

for $1 \leq i \leq len(l_1)$, so, by the definition of function $\mathcal{J}$, $() \in \mathcal{J}(db, E_2, l_1)$, which indicates that $() \in [\![(\text{Nest } E_1\ E_2)]\!]^t_{db}$.

l_2 is a non-empty list. By the key condition, we know that $len(l_2) = 1$, i.e., l_2 is a list consisting of a single tuple $hd(l_2)$, so

$$\mathcal{J}(db, E_1, l_2) = [\![E_1]\!]^{hd(l_2)}_{db}.$$

By Lemma 4.3 (with condition 3),

$$\{l'[(\text{Var } E_1)] \prec t | l' \in [\![E_1]\!]_{db}^{hd(l_2)}\} = [\![E_1]\!]_{db}^{t},$$

so we have $() \in [\![E_1]\!]_{db}^{t}$, which means $() \in [\![(\text{Nest } E_1\ E_2)]\!]_{db}^{t}$.

Non-empty list. For any database db and any complete tuple t over $\mathbf{V}_{db}$, if there exists a non-empty list $l \in [\![(\text{Nest } E_2\ E_1)]\!]_{db}^{t}$, there must be a non-empty list $l' \in [\![E_2]\!]_{db}^{t}$ such that $l \in \mathcal{J}(db, E_1, l')$. By the key condition, $l \in [\![E_1]\!]_{db}^{hd(l')}$.

According to Theorem 3.1,

$$hd(l')[(\text{Var } E_1)] = t[(\text{Var } E_1)],$$

and further by Lemma 4.3 (with condition 3),

$$\{l[(\text{Var } E_1)] \prec t | l \in [\![E_1]\!]_{db}^{hd(l')}\} = [\![E_1]\!]_{db}^{t},$$

so, there exists a corresponding list $l_1 \in [\![E_1]\!]_{db}^{t}$ such that

$$l[(\text{Var } E_1)] = l_1[(\text{Var } E_1)].$$

Two possibilities are considered here.

Empty possibility. $\mathcal{J}(db, E_2, l_1) = \{()\}$. By definition

$$[\![E_2]\!]_{db}^{hd(l_1)} = \{()\},$$

and by Theorem 3.1,

$$hd(l_1)[(\text{Var } E_2)] = t[(\text{Var } E_2)],$$

so, finally by Lemma 4.3,

$$[\![E_2]\!]_{db}^{t} = \{()\}.$$

By definitions,

$$[\![(\text{Nest } E_2\ E_1)]\!]_{db}^{t} = \{()\}.$$

Because we have l is a non-empty list in $[\![(\text{Nest } E_2\ E_1)]\!]_{db}^{t}$, so there is no such a possibility.

Non-empty possibility. $\mathcal{J}(db, E_2, l_1) \neq \{()\}$. By Lemma 4.5, there is $l_2 \in \mathcal{J}(db, E_2, l_1)$ such that

$$l_2[\alpha(t) - (\text{Bound } E_2)] \prec t = l_1,$$

$$nth(l_2, i)[(\text{Var } E_2)] = hd(l')[(\text{Var } E_2)],$$

which implies

$$l_2[(\text{Var } E_1)] = l[(\text{Var } E_1)],$$

$$nth(l_2, i)[(\text{Var } E_2)] = nth(l, i)[(\text{Var } E_2)],$$

for all $1 \leq i \leq len(l_2)$. So, we have

$$l[(\text{Var } E_1) \cup (\text{Var } E_2)] = l_2[(\text{Var } E_1) \cup (\text{Var } E_2)].$$

Finally, by Theorem 3.1, $l = l_2$, which means

$$l \in [\![(\text{Nest } E_1 \; E_2)]\!]^t_{db}.$$

Because of the generality of l, we can say

$$[\![(\text{Nest } E_2 \; E_1)]\!]^t_{db} \subseteq [\![(\text{Nest } E_1 \; E_2)]\!]^t_{db},$$

which contradicts to the assumption, therefore the rewrite rule is an axiom.

□

From this theorem, an interesting axiom could be drawn directly.

Corollary **4.2.1** *The inference rule*

```
( Clause   (Ref (Nest E1 E2) (Cross E1 E2))
           (Ref E2 (Filter E2))
           (Empty (Intersect (Bound E1) (Var E2)))
           (Empty (Intersect (Var E1) (Bound E2))))
```

is an axiom.

Proof: The proof is conducted by contradiction. Let's assume that the given rule is not an axiom, so there must exist a database db and a ground rule R (generated from the given rule by substitution), which is of the form:

$$\begin{array}{ll}(\text{Clause} & (\text{Ref } (\text{Nest } E_1\ E_2)\ (\text{Cross } E_1\ E_2)) \\ & (\text{Ref } E_2\ (\text{Filter } E_2)) \\ & (\text{Empty } (\text{Intersect } (\text{Bound } E_1)\ (\text{Var } E_2))) \\ & (\text{Empty } (\text{Intersect } (\text{Var } E_1)\ (\text{Bound } E_2)))),\end{array}$$

such that for some complete tuple t,

1. $db \models$ (Ref E_2 (Filter E_2))
2. $db \models$ (Empty (Intersect (Bound E_1) (Var E_2)))
3. $db \models$ (Empty (Intersect (Var E_1) (Bound E_2))), and
4. $[\![(\text{Cross } E_1\ E_2)]\!]^t_{db} \not\subseteq [\![(\text{Nest } E_1\ E_2)]\!]^t_{db}$.

However, by the theorem 4.2, under the condition 1, 2 and 3, we have

$$[\![(\text{Nest } E_2\ E_1)]\!]^t_{db} \subseteq [\![(\text{Nest } E_1\ E_2)]\!]^t_{db}.$$

and by the definition of the Cross operator,

$$[\![(\text{Cross E1 E2})]\!]^t_{db} = [\![(\text{Nest E1 E2})]\!]^t_{db} \cup [\![(\text{Nest E2 E1})]\!]^t_{db}$$

so for any database db and complete tuple t,

$$[\![(\text{Cross } E_1\ E_2)]\!]^t_{db} \subseteq [\![(\text{Nest } E_1\ E_2)]\!]^t_{db}.$$

which is contradictory to the assumption.

Therefore, the given rule is an axiom.

□

5 Correctness of A Real-World Rewrite Rule

A *query rewrite rule* is an inference rule defined in the refinement calculus with the following fixed format:

$$\text{(Clause (Ref } E_1\ E_2\text{) } R_1, \ldots, R_n\text{)}$$

We say a query rewrite rule is *correct* if and only if it is an axiom.

In this section, we use the refinement calculus inference axioms (given in Table 3) to derive the correctness of a real-world rewrite rule from a set of axioms in the refinement calculus. This derivation shows the proof theory for the calculus, although the theory is not complete.

5.1 More Related Axioms

Before we get into the derivation of the correctness of the real-world rewrite rule, we need to prove four more related basic axioms to be used in the derivation.

THEOREM **5.1** *The following rules are axioms.*

1. (Ref (Cross E*1 E1 E*2) (Nest E1 (Cross E*1 E*2)))

2. (Clause
```
        (Ref (Nest E1 (Cross E*1)) (Cross E1 E*1))
        (Ref E1 (Filter E1))
        (Empty (Intersect (Var (Cross E*1)) (Bound E1)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E1))))
```

3. (Clause
```
        (Clause (Ref (Nest E1 E2) (Nest E3 E4)) R*1)
        (Clause (Ref E1 E2) R*1)
        (Clause (Ref E3 E4) R*1))
```

4. (Clause
    (Clause (Ref E1 E2) R*1)
    (Clause (Ref E1 E3) R*1)
    (Clause (Ref E3 E2) R*1))

Proof: We prove them one by one, based on the model theory of the refinement calculus.

Rule 1. It is trivial by the definition of the Cross operator.

Rule 2. As usual, the proof is by contradiction. If the rule is not an axiom, then for some ground rule substituted from the rule, which is of the form

(Clause
    (Ref (Nest E_1 (Cross $E_2, \ldots, E_n$)) (Cross E_1 $E_2, \ldots, E_n$))
    (Ref E_1 (Filter E_1))
    (Empty (Intersect (Var (Cross $E_2, \ldots, E_n$)) (Bound E_1)))
    (Empty (Intersect (Bound (Cross $E_2, \ldots, E_n$)) (Var E_1)))),

there must exist a database db, which satisfies,

1. $db \models$ (Ref E_1 (Filter E_1)),
2. $db \models$ (Empty (Intersect (Var (Cross $E_2, \ldots, E_n$)) (Bound E_1))),
3. $db \models$ (Empty (Intersect (Bound (Cross $E_2, \ldots, E_n$)) (Var E_1))), and
4. $[\![(\text{Cross } E_1, E_2, \ldots, E_n)]\!]^t_{db} \not\subseteq [\![(\text{Nest } E_1 \text{ (Cross } E_2, \ldots, E_n))]\!]^t_{db}$, for some complete tuple t.

However, by the definition of the Cross operator,

$$\begin{aligned}&[\![(\text{Cross } E_1, \ldots, E_n)]\!]^t_{db}\\ &= \bigcup_{1 \le i \le n} [\![(\text{Nest } E_i \text{ (Cross } E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n))]\!]^t_{db}\\ &= \bigcup_{1 \le q_1 \le n, \ldots, 1 \le q_n \le n} [\![(\text{Nest } E_{q_1} \text{ (Nest } \ldots \text{(Nest } E_{q_n})))]\!]^t_{db}\end{aligned}$$

where $q_1, \ldots, q_n$ is a permutation of the number $1, \ldots, n$, therefore every list $l \in [\![(\text{Cross } E_1, \ldots, E_n)]\!]^t_{db}$ must belong to some

$$[\![(\text{Nest } E_{q_1} \text{ (Nest } \ldots \text{(Nest } E_{q_n})))]\!]^t_{db} .$$

Without losing generality, let's assume q_t be 1 $(1/leqt/leqn)$. By the commutativity and the associativity of the Nest operator (Theorem 4.1 and 4.2), we can exchange subexpressions E_{q_1} and E_{q_t} inside the last expression, and have the equation below:

$$[\![(\text{Nest } E_{q_1} \text{ (Nest } \ldots \text{(Nest } E_{q_n})))]\!]^t_{db} =$$
$$[\![(\text{Nest } E_{q_t} \text{ (Nest } \ldots \text{(Nest } E_{q_{t-1}} \text{ (Nest } E_{q_1} \text{ (Nest } E_{q_{t+1}} \ldots \text{(Nest } E_{q_n})))))]\!]^t_{db}.$$

It is easy to see that $q_1, \ldots, q_{t-1}, q_{t+1}, \ldots, q_n$ is a permutation of $2, \ldots, n$, and also by the definition of the Cross operator, any list l' in

$$[\![(\text{Nest } E_{q_2} \text{ (Nest } \ldots \text{(Nest } E_{q_{t-1}} \text{ (Nest } E_{q_1} \text{ (Nest } E_{q_{t+1}} \ldots \text{(Nest } E_{q_n})))))]\!]^t_{db},$$

belongs to $[\![(\text{Cross } E_2, \ldots, E_n)]\!]^t_{db}$. Therefore, back to the definition of the Nest operator, we have

$$l \in [\![(\text{Nest } E_1 \text{ (Cross } E_2, \ldots, E_n))]\!]^t_{db}.$$

Since the arbitrariness of l, the assumption is wrong, so Rule 2 is an axiom.

Rule 3. By contradiction. If the rule is not an axiom, there exists a ground rule of the form:

$$\begin{array}{l}(\text{Clause} \\ \quad (\text{Clause (Ref (Nest } E_1\ E_2) \text{ (Nest } E_3\ E_4))\ R_1, \ldots, R_n) \\ \quad (\text{Clause (Ref } E_1\ E_3)\ R_1, \ldots, R_n) \\ \quad (\text{Clause (Ref } E_2\ E_4)\ R_1, \ldots, R_n))\end{array}$$

which satisfies:

1. $db \models (\text{Clause (Ref } E_1\ E_3)\ R_1, \ldots, R_n),$

2. $db \models$ (Clause (Ref E_2 E_4) $R_1, \ldots, R_n$), and
3. $db \not\models$ (Clause (Ref (Nest E_1 E_2) (Nest E_3 E_4)) $R_1, \ldots, R_n$)

$db \models$ (Clause (Ref E_1 E_3) $R_1, \ldots, R_n$) means:

$$db \models \text{(Ref } E_1\ E_3\text{))} \text{ if } db \models R_i, \text{ for } 1 \leq i \leq n.$$

$db \models$ (Clause (Ref E_2 E_4) $R_1, \ldots, R_n$) means:

$$db \models \text{(Ref } E_2\ E_4\text{))} \text{ if } db \models R_i, \text{ for } 1 \leq i \leq n.$$

Since (Clause (Ref (Nest E1 E2) (Nest E3 E4)) (Ref E1 E3) (Ref E2 E4)) is an axiom (refer to [CW93]), we have

$$db \models \text{(Ref (Nest } E_1\ E_2\text{) (Nest } E_3 E_4\text{)) if } \begin{cases} db \models \text{(Ref E1 E3)} \\ db \models \text{(Ref E2 E4)} \end{cases}$$

Therefore,

$$db \models \text{(Ref (Nest } E_1\ E_2\text{) (Nest } E_3\ E_4\text{)) if if } db \models R_i, \text{ for } 1 \leq i \leq n.$$

which implies

$$db \models \text{(Clause (Ref (Nest } E_1\ E_2\text{) (Nest } E_3\ E_4\text{))} R_1, \ldots, R_n\text{)},$$

so the assumption is wrong, and Rule 3 is an axiom.

Rule 4. Can be proved in the same way as the last one. We leave the proof for the interested readers.

□

5.2 The Derivation of the Correctness

In general, the real-world rewrite rules are much more complex than the axioms that have been proved so far. Although the rewrite rules can be proved from the model

Table 5: A Real-World Query Rewrite Rule (in the refinement calculus)

```
(Clause
   (Ref
      (Nest E1
         (Nest (Cross E*1)
            (Nest E2 (Nest (Cross E*2 E3 E*3) E4))))
      (Nest E1
         (Nest (Cross E3 E*1)
            (Nest E2 (Nest (Cross E*2 E*3) E4)))))
   (Ref E3 (Filter E3))
   (Empty (Intersect (Bound E2) (Var E3)))
   (Empty (Intersect (Bound (Cross E*1)) (Var E3)))
   (Empty (Intersect (Var E2) (Bound E3)))
   (Empty (Intersect (Var (Cross E*1)) (Bound E3))))
```

theory (like the proofs we have gone through so far), it would be much easier to derive them from a set of basic and simple axioms. Actually, we are very interested in proving (i.e., derivation) from the proof theory because of its mechanical nature. Due to the incompleteness of the proof theory, one of future work is to find out a base set of axioms which could be used in general to prove the correctness of a rule-based query optimizer.

With the power of the refinement calculus, we apply it to capture the intentions underlying a real-world rule. In Table 5, we shows a formal specification in the refinement calculus for the real-world rewrite rule mentioned earlier (referring to Table 1). The rewrite rule was originally represented in a Lisp-like language, and is translated into the refinement calculus for the analysis of its correctness. The function of this rewrite rule is to generate a canonical form of a query expression by moving all the predicates together, so that more semantic query optimizations would be applied to the canonical form later on.

Now, let's prove the correctness of the real-world rewrite rule by derivation (the proof theory).

1. known axiom, refer to Theorem 5.1

```
(Ref (Cross E*1 E1 E*2)
        (Nest E1 (Cross E*1 E*2)))
```

2. known axiom, Associativity of Nest, refer to Theorem 4.1

```
(Ref (Nest (Nest E1 E2) E3)
        (Nest E1 (Nest E2 E3)))
```

3. substitution of 2 by { E1/E3, E2/(Cross E*2 E*3), E3/E4 }

```
(Ref (Nest (Nest E3 (Cross E*2 E*3)) E4)
        (Nest E3 (Nest (Cross E*2 E*3) E4)))
```

4. known axiom, refer to the paper [CW93]

```
(Clause
        (Ref (Nest E1 E2) (Nest E3 E4))
        (Ref E1 E3)
        (Ref E2 E4))
```

5. substitution of 1 by { E*1/E*2, E1/E3, E*2/E*3 }

```
(Ref (Cross E*2 E3 E*3)
        (Nest E3 (Cross E*2 E*3)))
```

6. substitution of 4 by
{ E1/(Cross E*2 E3 E*3), E2/E4, E3/(Nest E3 (Cross E*2 E*3)) }

```
(Clause
        (Ref (Nest (Cross E*2 E3 E*3) E4)
               (Nest (Nest E3 (Cross E*2 E*3)) E4))
        (Ref (Cross E*2 E3 E*3) (Nest E3 (Cross E*2 E*3))))†
```

7. modus ponens 5 and 6

```
(Ref (Nest (Cross E*2 E3 E*3) E4)
        (Nest (Nest E3 (Cross E*2 E*3)) E4))
```

8. substitution of 4 by { E1/E2, E2/(Nest (Cross E*2 E3 E*3) E4), E3/E2, E4/(Nest (Nest E3 (Cross E*2 E*3)) E4) }

```
(Clause
        (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4))
               (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4)))
        (Ref (Nest (Cross E*2 E3 E*3) E4)
               (Nest (Nest E3 (Cross E*2 E*3)) E4)))
```

9. modus ponens 7 and 8

```
(Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4))
        (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4)))
```

10. substitution of 4 by
{ E1/E2, E2/(Nest (Nest E3 (Cross E*2 E*3)) E4),
E3/E2, E4/(Nest E3 (Nest (Cross E*2 E*3) E4)) }

```
(Clause
        (Ref (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4))
               (Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4))))
        (Ref (Nest (Nest E3 (Cross E*2 E*3)) E4)
               (Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4))))
```

11. modus ponens 3 and 10

```
(Ref (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4))
        (Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4))))
```

12. known axiom, Associativity of Nest, refer to Theorem 4.1

```
(Ref (Nest E1 (Nest E2 E3))
        (Nest (Nest E1 E2) E3))
```

13. substitution of 12 by
{ E1/E2, E2/E3, E3/(Nest (Cross E*2 E*3) E4) }

```
(Ref (Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4)))
        (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)))
```

14. known axiom, refer to the paper [CW93]

```
(Clause
        (Ref E1 E2)
        (Ref E1 E3)
        (Ref E3 E2))
```

15. substitution of 14 by
{ E1/(Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4)),
E2/(Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)),
E3/(Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4))) }

```
(Clause
        (Ref (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4))
              (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)))
        (Ref (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4))
              (Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4))))
        (Ref (Nest E2 (Nest E3 (Nest (Cross E*2 E*3) E4)))
              (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4))))
```

16. modus ponens 11, 13 and 15

```
(Ref (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4))
        (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)))
```

17. substitution of 14 by
{ E1/(Nest E2 (Nest (Cross E*2 E3 E*3) E4))),
E2/(Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)),
E3/(Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4)) }

```
(Clause
        (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
              (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)))
        (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
              (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4)))
        (Ref (Nest E2 (Nest (Nest E3 (Cross E*2 E*3)) E4))
              (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4))))
```

18. modus ponens 9, 16 and 17

```
(Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
        (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)))
```

19. known axiom, Commutativity of Nest, refer to Theorem 4.2

```
(Clause
        (Ref (Nest E1 E2) (Nest E2 E1))
        (Ref E2 (Filter E2))
        (Empty (Intersect (Var E1) (Bound E2)))
        (Empty (Intersect (Bound E1) (Var E2))))
```

20. substitution of 19 by { E1/E2, E2/E3 }

```
(Clause
        (Ref (Nest E2 E3) (Nest E3 E2))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
```

21. known axiom, refer to Theorem 5.1

```
(Clause
        (Clause (Ref (Nest E1 E2) (Nest E3 E4)) R*1)
        (Clause (Ref E1 E2) R*1)
        (Clause (Ref E3 E4) R*1))
```

22. substitution of 21 by
    { E1/(Nest E2 E3), E2/(Nest (Cross E*2 E*3) E4), E3/(Nest E3 E2), E4/(Nest (Cross E*2 E*3) E4)), R*1/R_1 R_2 R_3 }
    where R_1 represents (Ref E3 (Filter E3)), R_2 indicates (Empty (Intersect (Var E2) (Bound E3))), and R_3 means (Empty (Intersect (Bound E2) (Var E3)))[‡]

[‡]For simplicity, we use these symbols in the following substitutions.

```
(Clause
        (Clause
               (Ref (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4))
                      (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var E2) (Bound E3)))
               (Empty (Intersect (Bound E2) (Var E3))))
        (Clause
               (Ref (Nest E2 E3) (Nest E3 E2))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var E2) (Bound E3)))
               (Empty (Intersect (Bound E2) (Var E3)))))
```

23. modus ponens 20 and 22

```
(Clause
        (Ref (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4))
               (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
```

24. known axiom, refer to Theorem 5.1

```
(Clause
        (Clause (Ref E1 E2) R*1)
        (Clause (Ref E1 E3) R*1)
        (Clause (Ref E3 E2) R*1))
```

25. substitution of 24 by
{ E1/(Nest E2 (Nest (Cross E*2 E3 E*3) E4))),
E2/(Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)),
E3/(Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)),
R*1/R_1 R_2 R_3 }

```
(Clause
    (Clause (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
    (Clause (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4)))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
    (Clause (Ref (Nest (Nest E2 E3) (Nest (Cross E*2 E*3) E4))
                (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))))
```

26. modus ponens 18, 23 and 25

```
(Clause
    (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
        (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)))
    (Ref E3 (Filter E3))
    (Empty (Intersect (Var E2) (Bound E3)))
    (Empty (Intersect (Bound E2) (Var E3))))
```

27. substitution of 2 by { E1/E2, E3/(Nest (Cross E*2 E*3) E4) }

```
(Ref (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4))
    (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
```

28. substitution of 24 by
{ E1/(Nest E2 (Nest (Cross E*2 E3 E*3) E4))),
E2/(Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))),
E3/(Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)),
R*1/R_1 R_2 R_3 }

```
(Clause
        (Clause (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                      (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var E2) (Bound E3)))
               (Empty (Intersect (Bound E2) (Var E3))))
        (Clause (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                      (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4)))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var E2) (Bound E3)))
               (Empty (Intersect (Bound E2) (Var E3))))
        (Clause (Ref (Nest (Nest E3 E2) (Nest (Cross E*2 E*3) E4))
                      (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var E2) (Bound E3)))
               (Empty (Intersect (Bound E2) (Var E3)))))
```

29. modus ponens 26, 27 and 28

```
(Clause
        (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
               (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
```

30. substitution of 21 by
{ E1/(Cross E*1), E2/(Nest E2 (Nest (Cross E*2 E3 E*3) E4)),
E3/(Cross E*1), E4/(Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))),
R*1/R_1 R_2 R_3 }

```
(Clause
    (Clause
        (Ref (Nest (Cross E*1)
                    (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
            (Nest (Cross E*1)
                    (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4)))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
    (Clause
        (Ref (Nest E2 (Nest (Cross E*2 E3 E*3) E4))
            (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))))
```

31. modus ponens 29 and 30

```
(Clause
    (Ref (Nest (Cross E*1)
                (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
        (Nest (Cross E*1)
                (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4)))))
    (Ref E3 (Filter E3))
    (Empty (Intersect (Var E2) (Bound E3)))
    (Empty (Intersect (Bound E2) (Var E3))))
```

32. substitution of 12 by
{ E1/(Cross E*1), E2/E3, E3/(Nest E2 (Nest (Cross E*2 E*3) E4)) }

```
(Ref (Nest (Cross E*1)
            (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
    (Nest (Nest (Cross E*1) E3)
        (Nest E2 (Nest (Cross E*2 E*3) E4))))
```

33. substitution of 24 by
{ E1/(Nest (Cross E*1) (Nest E2 (Nest (Cross E*2 E3 E*3) E4))),
E2/(Nest (Nest (Cross E*1) E3) (Nest E2 (Nest (Cross E*2 E*3) E4))),
E3/(Nest (Cross E*1) (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4)))),
R*1/R_1 R_2 R_3 }

```
(Clause
        (Clause
                (Ref (Nest (Cross E*1)
                                (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                        (Nest (Nest (Cross E*1) E3)
                                (Nest E2 (Nest (Cross E*2 E*3) E4))))
                (Ref E3 (Filter E3))
                (Empty (Intersect (Var E2) (Bound E3)))
                (Empty (Intersect (Bound E2) (Var E3))))
        (Clause
                (Ref (Nest (Cross E*1)
                                (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                        (Nest (Cross E*1)
                                (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4)))))
                (Ref E3 (Filter E3))
                (Empty (Intersect (Var E2) (Bound E3)))
                (Empty (Intersect (Bound E2) (Var E3))))
        (Clause
                (Ref (Nest (Cross E*1)
                                (Nest E3 (Nest E2 (Nest (Cross E*2 E*3) E4))))
                        (Nest (Nest (Cross E*1) E3)
                                (Nest E2 (Nest (Cross E*2 E*3) E4))))
                (Ref E3 (Filter E3))
                (Empty (Intersect (Var E2) (Bound E3)))
                (Empty (Intersect (Bound E2) (Var E3)))))
```

34. modus ponens 31, 32 and 33

```
(Clause
        (Ref (Nest (Cross E*1)
                        (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                (Nest (Nest (Cross E*1) E3)
                        (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3))))
```

35. known axiom, refer to Theorem 5.1

```
(Clause
        (Ref (Nest (Cross E*1) E1) (Cross E1 E*1)))
        (Ref E1 (Filter E1))
        (Empty (Intersect (Var (Cross E*1)) (Bound E1)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E1))))
```

36. substitution of 35 by { E1/E3 }

```
(Clause
        (Ref (Nest (Cross E*1) E3) (Cross E3 E*1)))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
```

37. substitution of 21 by
{E1/(Nest (Cross E*1) E3), E2/(Nest E2 (Nest (Cross E*2 E*3) E4)),
E3/(Cross E3 E*1), E4/(Nest E2 (Nest (Cross E*2 E*3) E4))),
R*1/R_1 R_4 R_5 }
where R_4 indicates (Empty (Intersect (Var (Cross E*1)) (Bound E3))), and R_5 means (Empty (Intersect (Bound (Cross E*1)) (Var E3)))

```
(Clause
        (Clause
               (Ref (Nest (Nest (Cross E*1) E3)
                              (Nest E2 (Nest (Cross E*2 E*3) E4)))
                     (Nest (Cross E3 E*1)
                              (Nest E2 (Nest (Cross E*2 E*3) E4))))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
               (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
        (Clause
               (Ref (Nest (Cross E*1) E3) (Cross E3 E*1)))
               (Ref E3 (Filter E3))
               (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
               (Empty (Intersect (Bound (Cross E*1)) (Var E3)))))
```

38. modus ponens 36 and 37

```
(Clause
        (Ref (Nest (Nest (Cross E*1) E3)
                   (Nest E2 (Nest (Cross E*2 E*3) E4)))
             (Nest (Cross E3 E*1)
                   (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
```

39. substitution of 24 by
{ E1/(Nest (Cross E*1) (Nest E2 (Nest (Cross E*2 E3 E*3) E4))),
E2/(Nest (Cross E3 E*1) (Nest E2 (Nest (Cross E*2 E*3) E4))),
E3/(Nest (Nest (Cross E*1) E3) (Nest E2 (Nest (Cross E*2 E*3) E4))),
R*1/R_1 R_2 R_3 R_4 R_5 }

```
(Clause
    (Clause
        (Ref (Nest (Cross E*1)
                    (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
              (Nest (Cross E3 E*1)
                    (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))
        (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
    (Clause
        (Ref (Nest (Cross E*1)
                    (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
              (Nest (Nest (Cross E*1) E3)
                    (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))
        (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
    (Clause
        (Ref (Nest (Nest (Cross E*1) E3)
                    (Nest E2 (Nest (Cross E*2 E*3) E4)))
              (Nest (Cross E3 E*1)
                    (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))
        (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E3)))))
```

40. modus ponens 34, 38 and 39

```
(Clause
        (Ref (Nest (Cross E*1)
                     (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
              (Nest (Cross E3 E*1)
                     (Nest E2 (Nest (Cross E*2 E*3) E4))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))
        (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
```

41. substitution of 21 by
    { E2/(Nest (Cross E*1) (Nest E2 (Nest (Cross E*2 E3 E*3) E4))),
    E3/E1,
    E4/(Nest (Cross E3 E*1) (Nest E2 (Nest (Cross E*2 E*3) E4))),
    R*1/R_1 R_2 R_3 R_4 R_5 }

```
(Clause
        (Clause
              (Ref (Nest E1
                          (Nest (Cross E*1)
                                (Nest E2
                                      (Nest (Cross E*2 E3 E*3) E4))))
                    (Nest E1
                          (Nest (Cross E3 E*1)
                                (Nest E2 (Nest (Cross E*2 E*3) E4)))))
              (Ref E3 (Filter E3))
              (Empty (Intersect (Var E2) (Bound E3)))
              (Empty (Intersect (Bound E2) (Var E3)))
              (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
              (Empty (Intersect (Bound (Cross E*1)) (Var E3))))
        (Clause
              (Ref (Nest (Cross E*1)
                          (Nest E2 (Nest (Cross E*2 E3 E*3) E4)))
                    (Nest (Cross E3 E*1)
                          (Nest E2 (Nest (Cross E*2 E*3) E4))))
              (Ref E3 (Filter E3))
              (Empty (Intersect (Var E2) (Bound E3)))
              (Empty (Intersect (Bound E2) (Var E3)))
              (Empty (Intersect (Var (Cross E*1)) (Bound E3)))
              (Empty (Intersect (Bound (Cross E*1)) (Var E3)))))
```

42. modus ponens 40 and 41

```
(Clause
        (Ref (Nest E1
                    (Nest (Cross E*1)
                          (Nest E2
                                (Nest (Cross E*2 E3 E*3) E4))))
              (Nest E1
                    (Nest (Cross E3 E*1)
                          (Nest E2 (Nest (Cross E*2 E*3) E4)))))
        (Ref E3 (Filter E3))
        (Empty (Intersect (Var E2) (Bound E3)))
        (Empty (Intersect (Bound E2) (Var E3)))
        (Empty (Intersect (Var (Cross E*1)) (Bound E1)))
        (Empty (Intersect (Bound (Cross E*1)) (Var E1))))
```

6 Conclusion

It is generally accepted that rule-based query optimization is a more flexible approach to supporting high-level query languages. However, current practice involves very limited consideration of the issue of rule validity (or correctness). Consequently, the reliability of rule-based query optimization will tend to diminish as both the expressiveness of query languages and complexity of the underlying data models increase.

In the Advanced Database Systems Laboratory at our institution, with the desire of increasing the expressive power of query languages and with the need of coping with the complex objects, a wide-spectrum query algebra and a refinement calculus over it was proposed [CW93]. In this essay, based on the calculus, I look into the issue of the analysis of the correctness of query rewrite rules for rule-based query optimizers.

In my work, I extended the calculus to better express query rewrite rules, proved additional basic axioms for the calculus in its model theory, and exhibited a derivation of a real-world query rewrite rule in its proof theory. Most significantly, we learned from this work:

(1) that our original informal understanding of the intentions underlying the rule was incorrect,

(2) that our first few attempts at a formal specification of this rule were not valid,

(3) that proofs in the model theory are sometimes needed for basic axioms (but that knowledge of existing axioms can simplify this),

(4) that there is reasonable prospect that the proof theory will suffice to establish the correctness of the real-world query rewrite rules, and

(5) that the nature of our derivations in the proof theory suggests that the process can be automated.

Future directions in this line of research include:

(1) establishing correctness for the remaining rewrite rules that comprises the rule-based query optimizer in our research database system, and

(2) further extending and refining the algebra and the calculus to facilitate expressing other queries and rewrite rules.

In regard to the first direction, we anticipate that completing this process will produce a base set of axioms that are likely to be sufficient for deriving rewrite rules used in other existing and future query optimizers.

In regard to the second direction, the following shows an example: by introducing a new syntactic constraint “(Pred E*)” with the semantics that a database is a model for it if and only if each query in E* is a predicate, the real-world rewrite rule (refer to Table 1) would be translated to a more readable (albeit less general) form:

```
(Clause
  (Ref
    (Nest E1
      (Nest (Cross E*1)
        (Nest E2 (Nest (Cross E*2 E3 E*3) E4))))
    (Nest E1
      (Nest (Cross E3 E*1)
        (Nest E2 (Nest (Cross E*2 E*3) E4)))))
  (Pred E*1)
  (Pred E3)
  (Empty (Intersect (Bound E2) (Var E3))
```

References

[ABD$^+$89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, December 1989.

[Bee89] C. Beeri. Formal models for object-oriented databases. In *Proceedings of the First International Conference on Deductive aand Object-Oriented Databases*, December 1989.

[BG92] Ludger Becker and Ralf Hartmut Guting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2), June 1992.

[CW93] Neil Coburn and Grant E. Weddell. A logic for rule-based query optimization in graph-based data model. In *Proceedings of the Third International Conference on Deductive aand Object-Oriented Databases*, 1993.

[Day89] U. Dayal. Queries and views in an object-oriented data model. In *Proceedings of the Second International Workshop on Database Programming Languages*, June 1989.

[FG86] J. C. Freytag and N. Goodman. Rule-based translation of relational queries into iterative programs. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1986.

[FG91] Béatrice Finance and Georges Gardarin. A rule-based query rewriter in an extensible dbms. In *IEEE Conference on Data Engineering*, 1991.

[Fre87] J. C. Freytag. A rule-based view of query optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1987.

[GD87] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1987.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *IEEE Conference on Data Engineering*, 1993.

[GPvG90] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proceeding of the Third ACM Symposium on Principles of Database Systems*, 1990.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1992.