Changing the Rules: Transformations for Rule-Based Optimizers*

Mitch Cherniack
Department of Computer Science, Brown University
Providence, RI 02912-1910
mfc@cs.brown.edu

Stan Zdonik
Department of Computer Science, Brown University
Providence, RI 02912-1910
sbz@cs.brown.edu

Nobody realizes that some people expend tremendous energy merely to be normal. – Albert Camus

Abstract

Rule-based optimizers are extensible because they consist of modifiable sets of rules. For modification to be straightforward, rules must be easily reasoned about (i.e., understood and verified). At the same time, rules must be expressive and efficient (to fire) for rule-based optimizers to be practical. Production-style rules (as in [15]) are expressed with code and are hard to reason about. Pure rewrite rules (as in [1]) lack code, but cannot atomically express complex transformations (e.g., normalizations). Some systems allow rules to be grouped, but sacrifice efficiency by providing limited control over their firing. Therefore, none of these approaches succeeds in making rules expressive, efficient and understandable.

We propose a language (COKO) for expressing an alternative form of input to a rule-based optimizer. A COKO *transformation* consists of a set of declarative (KOLA) rewrite rules and a (firing) algorithm that specifies their firing. It is straightforward to reason about COKO transformations because all query modification is expressed with declarative rewrite rules. Firing is specified algorithmically with an expressive language that provides direct control over how query representations are traversed, and under what conditions rules are fired. Therefore, COKO achieves a delicate balance of understandability, efficiency and expressivity.

1 Introduction

Rule-based optimizers express *query rewrites* with rules. Rule-based optimizers are *extensible* because an optimizer's behavior can be modified by changing its rule set. But this is difficult unless rules are straightforward to understand and reason about. Therefore, extensibility is best achieved by expressing rules *declaratively*, rather than with code. (e.g., with rewrite rules).

In [5], we showed that the choice of query representation determines the effectiveness with which rewrite rules can express query rewrites. Rewrite rules use pairs of patterns to specify rewrites that *identify* and *rearrange* subexpressions. *Variable-based* query representations (i.e., representations formed from the parse structure of a query language or algebra with variables) complicate both identification and rearrangement. The problem is that their subexpressions can include *free variables*, which makes their meaning context-dependent. Code supplements to rewrite rules are required to analyze context when identifying subexpressions, and to massage identified subexpressions so that their meaning is preserved when they are rearranged. By removing variables from a query's representation (and instead using a *combinator-based* query representation such as KOLA), the need for these code supplements is eliminated.

But our KOLA work only tells part of the story about why query rewrites sometimes get expressed with code. Rewrite rules are inherently "small" in their operation. They are well-suited for expressing simple rewrites, such as ones that change the order of arguments to a join or push selections past joins. But some query rewrites are too complex to be expressed with rewrite rules, regardless of the underlying query representation. For example, a rewrite to convert Boolean expressions to *conjunctive normal form* (CNF) cannot be expressed with a rewrite rule because patterns are too constraining to capture its generality. (All boolean expressions can be transformed into CNF.) Instead, this rewrite is more appropriately described algorithmically.

The CNF rewrite is a *normalization*: a query rewrite that converts expressions into syntactically characterizable (i.e., *normal*) forms. Normalizations epitomize the kinds of complex query rewrites that cannot be expressed as rewrite rules. In this paper, we propose a language (COKO[1]) for specifying complex query rewrites such as these. Such rewrites require expression with code, but our objective is to limit where code appears so that complex rewrites can still be understood and reasoned about. A COKO *transformation* consists of a set of KOLA rewrite rules accompanied by a *firing algorithm* that specifies their firing. All query modification occurs by firing rewrite rules. Code can only be used in the firing algorithm and is limited to specifying how to traverse a query's representation and conditions for firing rules. The separation of rewrite rules from firing algorithms achieves a balance of understandability, expressivity and performance. A COKO transformation is understandable; it can be decomposed into its rewrite rules and even shown to preserve the semantics of the queries it transforms by proving the same property of these rules. As we showed in [5], this can be done with help from a theorem prover [10]. At the same time, COKO firing algorithms make transformations capable of expressing a wide variety of efficient query rewrites, as we later show.

This work generalizes and extends our KOLA work. COKO transformations behave like rewrite rules; they can be *fired* and can succeed or fail as a result. Therefore, the set of "rules" maintained by a rule-based optimizer can include KOLA rules (to express basic rewrites such as those that commute the arguments to a join) and COKO transformations (to express complex rewrites such as normalizations). At the same time, COKO transformations are built from KOLA rules. The modular approach of expressing complex transformations in terms of simpler rewrite rules simplifies reasoning about the meanings of transformations, just as expressing complex KOLA queries in terms of simpler functions simplified reasoning about the meanings of queries.

1.1 Related Work

Expressing Complex Query Rewrites with Code: In most rule-based systems, complex query rewrites are expressed with rules.

*This work was partially supported by NSF grant IRI 9632629.

[1] COKO is an acronym for [C]ontrol [O]f [K]OLA [O]ptimizations.

However, the rules of these systems are not declarative rewrite rules. Starburst [15] performs query rewrites during the *query rewriting* phase of optimization, firing *production rules* as in expert systems. These rules consist of two code routines (loosely corresponding to the lhs (left-hand side) and rhs (right-hand side) of a rewrite rule) programmed in C. Because they are programmed with a general purpose programming language, Starburst rules are able to express a wide variety of transformations including view merging and query unnesting (both discussed in [15]) and magic sets transformations ([14]). However, Starburst rules are difficult to understand and reason about, requiring a detailed understanding of the underlying graph-based query representation (QGM).

Expressing Complex Query Rewrites with Extended Rewrite Rules: Exodus [2] (and its successors, Volcano [9] and Cascades [8]) and ESL [7] use rules that resemble rewrite rules, but that can have code supplements. As with production rules, code-supplemented rules are expressive, but at the expense of understandability.

Of the rewrite rule-based systems, Gral [1] comes closest to ours in its effort to make rules declarative by avoiding code. Gral expands the expressive power of a rewrite rule without adding code supplements. Instead, Gral rules can have more than one pattern appear in the rhs, with each pattern associated with a declarative condition on the queried data. A query that matches the rule's lhs and satisfies the conditions associated with one of these patterns is then transformed according to that pattern.

The conditions associated with a rule analyze the data that is queried such as its representation, the existence of indices, cardinality etc. Therefore, Gral rules are more expressive than traditional declarative rewrite rules because they rely on more than pattern matching to decide whether or not a rule successfully fires. However, this is not the kind of expressive power that is required to express complex query rewrites such as normalizations. As we shall see, normalizations require the ability to fire rules repeatedly and to control the manner in which a query representation is traversed. Such control is not provided with Gral rules and instead must be expressed using the Gral meta-rule language as discussed below.

Expressing Complex Query Rewrites with Rule Groups: Many systems (e.g. [15], [1], [7], [13] [17]) provide some form of meta-control language for rules that includes rule grouping and sometimes sequencing. Rule groups can be associated with *search strategies* that indicate how the rules in a group should be fired. Search strategies are generic (i.e., they do not refer to the rules in any one rule group) and hence can be used with multiple rule groups.

It is tempting to view COKO as a meta-rule language in this style, and to compare COKO transformations with rule groups and firing algorithms with search strategies and sequencing control. But this analogy is misleading, as the "rules" that are grouped by these systems are more analagous to COKO transformations than KOLA rules. The purpose of grouping in this context is different. Other systems group to control search. Different rules generate different alternatives, and a "best" is chosen according to some criteria that considers the operators and data involved. Thus, whereas COKO groups rules to cooperatively specify a query rewrite, other systems group rules to "face-off" against one another.

The search strategies that are provided by these systems reflect this objective. Most (e.g., [15], [1], [7]) include *exhaustive search* (fire everything everywhere until firing no longer has any effect), with the goal of generating all possible alternatives before comparing them. Some systems permit some variations on exhaustive search (such as prioritizing rules [15] or limiting the number of passes over the query in advance [7]). Other systems provide pruning strategies so that only some alternatives are generated (e.g, [17] provide such strategies as branch-and-bound and simulated annealing). Some systems try to avoid generating poor alternatives by ranking rules (e.g., [15], [13] and [2]) or algebraic operators (e.g., [1]) to choose a rule to apply next.

COKO groups KOLA rules, not to generate alternatives, but to modularize the expression of a single complex query rewrite. There is no search involved. Rewriting is blind to the data and is concerned with the syntax of the result rather than its expected performance. Therefore, firing algorithms operate differently from search strategies. Of the search strategies listed above, only *exhaustive* firing can work as a firing algorithm, but only in restricted cases (rewrite rules must form a confluent set) and usually sacrificing performance in the process. (Exhaustive firing typically fires more rules than are needed to perform a transformation. We will see an example of this in the next section.) Firing algorithms needn't be generic nor exhaustive and instead can be customized to specific, fixed sets of rules. This can speed up query rewriting and free up more time for an optimizer to conduct its search.

1.2 Paper Outline

The paper proceeds as follows. We introduce COKO in Section 2 with a simple normalization transformation that converts KOLA predicates into CNF. We describe COKO fully in Section 3 and present our implemented compiler. In Section 4, we demonstrate the expressivity of COKO by showing a complex predicate normalization that partitions subpredicates of binary predicates according to the arguments they use. We then show that this transformation can be used in within other COKO transformations including ones that perform predicate pushdown and magic sets transformations. In Section 5, we consider issues that arose during the design and use of COKO. We give our conclusions and propose future work in Section 6.

2 A Simple Example: Expressing CNF in COKO

In this section, we introduce COKO by presenting a transformation that rewrites predicates into CNF. This example shows the potential performance benefits from being able to express customized firing algorithms. We describe what CNF means for KOLA predicates, and show a COKO transformation that rewrites predicates to CNF, tracing its execution on a sample KOLA predicate. We then compare this transformation with one that uses exhaustive firing and show that the non-exhaustive version exhibits far better performance.

2.1 Predicates and CNF

An SQL *predicate* is a boolean expression that appears in an SQL query's `where` or `having` clause. A predicate is in *CNF* if it is a conjunct (`AND`) of disjuncts (`OR`) of (possibly negated (`NOT`)) *literals* (i.e., predicates lacking conjuncts, disjuncts and negations). A query rewrite to convert predicates into CNF is a necessary preprocessing step to many other query rewrites such as one that changes the order of selection predicates.

Figure 1a shows a predicate that could appear in an SQL query's `where` clause. This predicate assumes the existence of a base table `Emp` (employees) with attributes `Eno` (employee number), `Sal` (salary), `Bon` (bonus), `Dno` (department number) and `Job`, and a view `Dept` (departments) with attributes `Dno` and `ASal` (average salary).[2] This predicate holds of an employee e and a department d if e is highly paid or if e is a manager for d and e's salary is more than the average for d. Figure 1b shows an equivalent predicate that has been normalized into CNF.

[2] A similar schema is used in papers from the Starburst group such as [14] and [18].

$$\texttt{(P AND Q AND R) OR S}$$
(a)

$$\texttt{(P OR S) AND (Q OR S) AND (R OR S)}$$
(b)

$$\begin{aligned}\texttt{P} &= e.\texttt{Dno} == d.\texttt{Dno}\\ \texttt{Q} &= e.\texttt{Sal} > d.\texttt{ASal}\\ \texttt{R} &= e.\texttt{Job} == \text{``Mgr''}\\ \texttt{S} &= (e.\texttt{Sal} + e.\texttt{Bon}) > 100K\end{aligned}$$

Figure 1: A Predicate (a) and its CNF Equivalent (b)

2.2 KOLA and CNF

KOLA is the language of queries and rewrite rules assumed by COKO transformations. Because they are expressed with combinators, KOLA queries can be difficult to read. But KOLA is intended to be an internal query representation and not a query language. Translation into KOLA from user-friendly query languages such as OQL is discussed elsewhere [3]. More comprehensive presentations of KOLA can be found in [5] and [4].

KOLA functions and predicates have no variables. They are either predefined *primitives* (e.g., **eq**) or are constructed from other functions and predicates with *formers*. To illustrate, the KOLA predicate, $\texttt{P}_\texttt{k} = \textbf{eq} \oplus \langle \texttt{Dno} \circ \pi_1, \texttt{Dno} \circ \pi_2 \rangle$ has the same semantics as the SQL predicate, P. To see this, it is necessary to know the semantics of the primitives (**eq**, Dno, π_1, π_2) and formers ($\oplus$, $\langle\ \rangle$, $\circ$) from which this predicate is built. The operational semantics of these and other primitives and formers are listed in Table 1. Functions (f) are defined by equations that show the result of invoking them on arbitrary objects (x) or object pairs ($[x,\ y]$)[3]. The invocation of function f on object or value x is expressed as $f\ !\ x$. Predicates (p) are also defined by equations showing their invocation ($p\ ?\ x$). In this table and throughout this paper, we denote arbitrary functions with variables f, g, h and j, predicates with p, q, r and s, objects and values with x and y, and collections with A and B.

Using the definitions of Table 1, the meaning of $\texttt{P}_\texttt{k}$ is revealed by invoking it on arbitrary employee, department pair ($[e,\ d]$), as illustrated by the derivation that reduces $\texttt{P}_\texttt{k}\ ?\ [e,\ d]$ to P:

$$\begin{aligned}&(\textbf{eq} \oplus \langle \texttt{Dno} \circ \pi_1, \texttt{Dno} \circ \pi_2 \rangle)\ ?\ [e,\ d]\\ =\ &\textbf{eq}\ ?\ (\langle \texttt{Dno} \circ \pi_1, \texttt{Dno} \circ \pi_2 \rangle\ !\ [e,\ d]) && (1)\\ =\ &\textbf{eq}\ ?\ [(\texttt{Dno} \circ \pi_1)\ !\ [e,\ d],\ (\texttt{Dno} \circ \pi_2)\ !\ [e,\ d]] && (2)\\ =\ &\textbf{eq}\ ?\ [\texttt{Dno}\ !\ e,\ \texttt{Dno}\ !\ d] && (3)\\ =\ &\textbf{eq}\ ?\ [e.\texttt{Dno},\ d.\texttt{Dno}] && (4)\\ =\ &e.\texttt{Dno} == d.\texttt{Dno} && (5)\end{aligned}$$

Step (1) of this reduction follows by the definition of $\oplus$. Step (2) follows by the definition of $\langle \_,\ \_ \rangle$. Step (3) follows by the definitions of $\_ \circ \_$, and the projection primitives π_1 and π_2. Step (4) follows from the definitions of attributes (in this case Dno). And step (5) follows from the definition of **eq**. The reader is encouraged to use the definitions of Table 1 to verify that the KOLA predicates $\texttt{Q}_\texttt{k}$, $\texttt{R}_\texttt{k}$ and $\texttt{S}_\texttt{k}$ from Figure 2 also have the same meaning as their SQL counterparts in Figure 1. From these derivations, it is straightforward to show that the KOLA predicates of Figure 2a and Figure 2b are equivalent to the SQL predicates of Figure 1a and 1b respectively.

KOLA's primitive functions include identity (**id**), projection functions on pairs (π_1 and π_2), and schema-dependent attributes ($<$ **att** $>$) such as Dno. KOLA's general function formers include

KOLA	Semantics
id	$\textbf{id}\ !\ x\ =\ x$
π_1	$\pi_1\ !\ [x,\ y]\ =\ x$
π_2	$\pi_2\ !\ [x,\ y]\ =\ y$
$<\textbf{att}>$	$<\textbf{att}>\ !\ x\ =\ x.<\textbf{att}>$
$\circ$	$(f \circ g)\ !\ x\ =\ f\ !\ (g\ !\ x)$
$\langle\ \rangle$	$\langle f,\ g \rangle\ !\ x\ =\ [f\ !\ x,\ g\ !\ x]$
$\times$	$(f \times g)\ !\ [x,\ y]\ =\ [f\ !\ x,\ g\ !\ y]$
$\texttt{K}_f$	$\texttt{K}_f\ (x)\ !\ y\ =\ x$
$\texttt{C}_f$	$\texttt{C}_f\ (f,\ x)\ !\ y\ =\ f\ !\ [x,\ y]$
eq	$\textbf{eq}\ ?\ [x,\ y]\ =\ x == y$ (**lt**, **gt**, **leq**, **geq**)
$\oplus$	$(p \oplus f)\ ?\ x\ =\ p\ ?\ (f\ !\ x)$
&	$(p\ \&\ q)\ ?\ x\ =\ (p\ ?\ x)$ AND $(q\ ?\ x)$
$\mid$	$(p \mid q)\ ?\ x\ =\ (p\ ?\ x)$ OR $(q\ ?\ x)$
$\sim$	$\sim (p)\ ?\ x\ =$ NOT $(p\ ?\ x)$
$^{-1}$	$p^{-1}\ ?\ [x,\ y]\ =\ p\ ?\ [y,\ x]$
$\texttt{K}_p$	$\texttt{K}_p\ (b)\ ?\ x\ =\ b$
$\texttt{C}_p$	$\texttt{C}_p\ (p,\ x)\ ?\ y\ =\ p\ ?\ [x,\ y]$

Table 1: KOLA Function and Predicate Semantics

$$((\texttt{P}_\texttt{k}\ \&\ \texttt{Q}_\texttt{k}\ \&\ \texttt{R}_\texttt{k}) \mid \texttt{S}_\texttt{k})\ ?\ [e,\ d]$$
(a)

$$((\texttt{P}_\texttt{k} \mid \texttt{S}_\texttt{k})\ \&\ (\texttt{Q}_\texttt{k} \mid \texttt{S}_\texttt{k})\ \&\ (\texttt{R}_\texttt{k} \mid \texttt{S}_\texttt{k}))\ ?\ [e,\ d]$$
(b)

$$\begin{aligned}\texttt{P}_\texttt{k} &= \textbf{eq} \oplus \langle \texttt{Dno} \circ \pi_1, \texttt{Dno} \circ \pi_2 \rangle\\ \texttt{Q}_\texttt{k} &= \textbf{gt} \oplus \langle \texttt{Sal} \circ \pi_1, \texttt{AvgSal} \circ \pi_2 \rangle\\ \texttt{R}_\texttt{k} &= \textbf{eq} \oplus \langle \texttt{Job} \circ \pi_1, \texttt{K}_f\ (\text{``Mgr''}) \rangle\\ \texttt{S}_\texttt{k} &= \textbf{gt} \oplus \langle \textbf{add} \circ \langle \texttt{Sal} \circ \pi_1, \texttt{Bon} \circ \pi_1 \rangle, \texttt{K}_f\ (\texttt{100K}) \rangle\end{aligned}$$

Figure 2: KOLA Versions of Predicates of Figure 1

composition ($\circ$), function pairing ($\langle\ \rangle$), pairwise function application ($\times$), constant functions ($\texttt{K}_f$) and curried functions ($\texttt{C}_f$). Primitive predicates include equality (**eq**), and ordering predicates $<$ (**lt**), $>$ (**gt**), $\leq$ (**leq**) and $\geq$ (**geq**). KOLA's predicate formers include predicate/function combination ($\oplus$), predicate inverses ($^{-1}$), constant predicates ($\texttt{K}_p$) and curried predicates ($\texttt{C}_p$). Binary functions and predicates (e.g., **eq**) are invoked on pairs ($[\_,\ \_]$).

The formers relevant to CNF are the logic-inspired formers, $\_\ \&\ \_$, $\_ \mid \_$ and $\sim (\_)$. A KOLA predicate p is in CNF if for any argument x, $p\ ?\ x$ reduces by the definitions of these formers to an SQL predicate that is in CNF. That is, p is in CNF if it is a conjunction (&) of disjunctions (|) of (possibly negated ($\sim$)) literals. Thus, the predicate of Figure 2b is in CNF.

2.3 A CNF Query Rewrite Expressed in COKO

We begin by considering a simplified query rewrite that converts any KOLA predicate lacking negations (subpredicates of the form, $\sim (p)$) into CNF. We first describe the rewrite rules and firing algorithm that perform the rewrite, and then trace the firing algorithm on the KOLA predicate of Figure 2a. We then show how this is rewrite is specified in COKO.

2.3.1 An Overview of the CNF Firing Algorithm

The two rewrite rules required for this rewrite are listed below:

$$\begin{aligned}\texttt{d1}:&\quad (\texttt{p}\ \&\ \texttt{q}) \mid \texttt{r} \ \stackrel{\rightarrow}{=}\ (\texttt{p} \mid \texttt{r})\ \&\ (\texttt{q} \mid \texttt{r})\\ \texttt{d2}:&\quad \texttt{r} \mid (\texttt{p}\ \&\ \texttt{q}) \ \stackrel{\rightarrow}{=}\ (\texttt{p} \mid \texttt{r})\ \&\ (\texttt{q} \mid \texttt{r})\end{aligned}$$

Each rule consists of two KOLA predicate patterns separated by the *rewrites to* symbol "$\stackrel{\rightarrow}{=}$". p, q and r are *pattern variables* denoting

[3] n-ary functions and predicates are modeled as unary functions and predicates that are applied to potentially nested pairs. Nested pairs also allow us to translate relational tuples of any length.

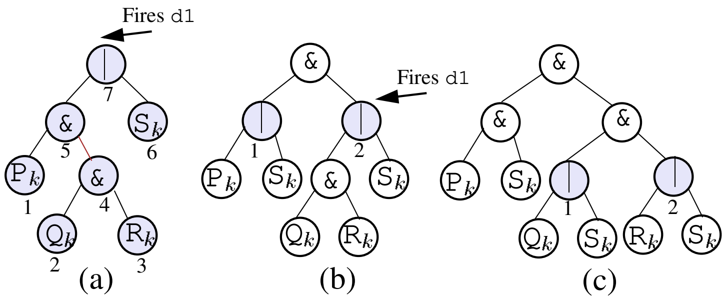

Figure 3: Converting the Predicate of Figure 2a to CNF

arbitrary KOLA predicates. The effect of firing either rule is to distribute a predicate (r) across a conjunction (p & q).

Rules are fired on parse tree representations of KOLA queries. (Example KOLA parse trees are shown in Figure 3.) The firing algorithm performs a single bottom-up pass over a KOLA predicate's parse tree. For each visited node, n:

1. Rules d1 and d2 are fired on the subtree rooted at n. If both rules fail (i.e., the lhs patterns fail to match the subtree), the firing algorithm proceeds to the next node.
2. If either rule succeeds, the subtree rooted at n gets transformed to match the rhs of the successfully fired rule. Regardless of which rule succeeds, the transformed subtree will be of the form, P & Q. Top-down passes are then initiated on both P and Q. These passes proceed as long as either d1 or d2 fire successfully on the subtrees rooted at visited nodes.

The effects of this rewrite on the KOLA predicate of Figure 2a are illustrated in Figure 3. Figure 3a shows the parse tree representation of this predicate before it is transformed. The bottom-up pass visits the shaded nodes of the initial tree in the order indicated beneath each. Attempts to fire d1 and d2 fail on the subtrees rooted at every node except the root (node 7). Firing d1 on the root results in the predicate tree of Figure 3b. Because d1 successfully fired, a top-down pass is initiated on the two disjuncts produced from firing (the subtrees rooted by the shaded nodes of Figure 3b). Firing d1 and d2 on the first of these subtrees fails, and therefore terminates the top-down pass of this subtree. But d1 fires successfully on the second subtree, producing the predicate tree of Figure 3c. Again, successful firing initiates top-down passes of the two newly constructed disjuncts. However, d1 and d2 fail to fire on both disjuncts, and the tree of Figure 3c is returned.

2.3.2 Expressing the CNF Rewrite in COKO

Figure 4 shows COKO transformations that rewrite KOLA predicates to CNF using the algorithm and rules presented in the previous section. As noted earlier, a COKO transformation consists of a set of rewrite rules and a firing algorithm. Rewrite rules are listed with any auxiliary COKO transformations in the transformation's USES section. The firing algorithm is delimited by the keywords BEGIN and END.

A COKO transformation can be fired (i.e., invoked) on any KOLA parse tree, and may transform this tree as a result. In presenting the CNF transformations of Figure 4, we will assume that the "main" transformation CNF, has been fired on some KOLA predicate, p via the invocation, CNF (p).

As a result of firing CNF, the statement, BU CNFAux, is executed. This performs a bottom-up (and specifically, a preorder) pass of p, firing CNFAux on every subtree of its parse tree. Let n denote some node of p's parse tree representation. Eventually, CNFAux gets fired on the subtree rooted at n (hereafter just referred

```
TRANSFORMATION CNF
 USES
  CNFAux
 BEGIN
  BU CNFAux
 END

TRANSFORMATION CNFAux
 USES
  d1 : (p & q) | r =⃗ (p | r) & (q | r)
  d2 : r | (p & q) =⃗ (p | r) & (q | r)
 BEGIN
  {d1 || d2}  →
    {GIVEN p & q DO {CNFAux(p); CNFAux(q)}}
 END
```

Figure 4: The CNF Transformation Expressed in COKO

to as n). The complex statement that is the firing algorithm for CNFAux then gets executed on this tree, resulting in the following steps:

- {d1 || d2} fires d1 on n, and then fires d2 on n if d1 fails.
- If either d1 or d2 successfully fired on n, n gets transformed to a predicate of the form, P & Q. Then, the statement {GIVEN p & q DO {CNFAux (p); CNFAux(q)}} identifies P and Q by matching n with the pattern, p & q, and then fires CNFAux recursively on the subtrees that get bound to variables p and q.
- "→" connects the statements above. This makes the execution of the second statement conditional on the success of the first. That is, CNFAux is fired recursively only if either d1 or d2 successfully fire on n.

In short, CNF performs a bottom-up (BU) pass of p and fires the auxiliary transformation, CNFAux, on the subtrees rooted at every visited node. CNFAux fires rewrite rules d1 and d2 and if either succeeds, initiates top-down passes on resulting subtrees.

This transformation exhibits the direct control of rule and transformation firing supported in COKO. This includes:

Explicit Firing: Rewrite rules used within a transformation are named (e.g., d1 and d2) and explicitly fired by the firing algorithm.

Traversal Control: Both bottom-up (in CNF) and top-down (in CNFAux) passes can be performed in the course of rewriting predicates into CNF.

Selective firing: CNFAux is fired recursively on the two disjuncts that result from successful firings of either d1 or d2. CNFAux is not fired on the conjunct resulting from these firings because both d1 and d2 can only succeed on trees rooted by "|".

Conditional firing: Some firings are conditioned on the success or failure of previous firings. For example, CNFAux fires d2 only if d1 fails and CNFAux is only fired recursively if one of the rules d1 or d2 succeeds.

Control of rule firings makes it possible to express efficient transformations. This is demonstrated in Section 2.5.

2.4 Correctness of CNF

Theorem 2.1 CNF *preserves the semantics of all queries it transforms*

Proof: All query modification performed by `CNF` occurs as result of firing `CNFAux`, which in turn occurs as result of firing `d1` and `d2`. Therefore, `CNF` is semantics-preserving if both rewrite rules are semantics-preserving. `d1` and `d2` are proved to be semantics preserving by execution of the theorem prover scripts of Appendix A using the theorem prover, LP [10]. □

Lemma 2.1 *Let p be a KOLA predicate tree lacking negations, and whose child subtrees are in CNF. Then* `CNFAux` *(p) is in CNF.*

Proof: (By structural induction on the height, $h(p)$ of the highest &-node in p.) For the base case ($h(p) = 0$) p must contain no &-nodes and therefore is in CNF and is returned untouched by `CNFAux`. For the inductive case, either p is already in CNF and is returned untouched, or p is not in CNF and is a disjunction, $Q \mid R$ such that Q and R are in CNF and at least one of Q or R is a conjunction ($x_0 \mid x_1$). For the case where exactly one of Q and R is a conjunction, assume w.l.o.g. that this is Q. Then, $h(Q)$ is larger than both $h(x_0)$ and $h(x_1)$ because both x_0 and x_1 are children of a &-node. Further, because R is in CNF it has no &-nodes and therefore $h(p) = h(Q)$. Firing `d1` on $Q \mid R$ returns S & T such that $S = (x_0 \mid R)$ and $T = (x_1 \mid R)$, and `CNFAux` is subsequently called on S and T. But $h(S) = h(x_0)$ and $h(T) = h(x_1)$ and therefore by induction, these firings result in trees that are in CNF. Therefore, the tree returned by `CNFAux` (p) is in CNF.

For the case where both Q (x_0 & x_1) and R (y_0 & y_1) are conjunctions, $h(p)$ is larger than $h(x_0)$, $h(x_1)$, $h(y_0)$ and $h(y_0)$ as all of the latter subtrees are children of &-nodes. A call of `CNFAux` on p fires `d1` once, and `d2` on each of the resulting disjuncts, leaving (S_1 & S_2) & (T_1 & T_2) such that $S_1 = (x_0 \mid y_0)$, $S_2 = (x_1 \mid y_0)$, $T_1 = (x_0 \mid y_1)$ and $T_2 = (x_0 \mid y_1)$. `CNFAux` is fired on each of S_1, S_2, T_1 and T_2 and by induction, each of these firings return a predicate in CNF. Therefore, `CNFAux` (p) is in CNF. □

Theorem 2.2 *Let p be any KOLA predicate lacking negations. Then* `CNF` *(p) is in CNF.*

Proof: Note that `CNF` (p) calls `CNFAux` on every node visited during a bottom-up traversal of p. By **Lemma 2.1**, each call leaves a subtree in CNF, and therefore p is left in CNF. □

2.5 Expressing CNF Exhaustively

`CNF_BU` (not shown) is an exhaustive, *bottom-up* version of the CNF transformation. This transformation fires the same rules as `CNF`, but with a firing algorithm consisting solely of the statement,

$$\texttt{BU}\ \{\texttt{d1}\ \|\ \texttt{d2}\}\ \rightarrow\ \texttt{CNF\_BU}.$$

`CNF_TD` is similarly defined to exhaustively apply rules `d1` and `d2`, but in a top-down (`TD`) fashion rather than bottom-up. Both `BU` and `TD` statements succeed if the statement they fire succeeds on one of the subtrees visited during the bottom-up or top-down traversal. Therefore, `BU` {`d1` || `d2`} fires `d1` and (if it fails) `d2` on the subtrees rooted at every node visited in a single bottom-up pass. Any successful firing during this pass triggers a recursive firing of the transformation making the transformation exhaustive. Note that it is possible to express the CNF transformation exhaustively because rules `d1` and `d2` happen to form a confluent set (i.e., changing the order in which rules are fired or nodes are visited does not affect the result).

Table 2 shows a performance comparison of `CNF_TD`, `CNF_BU` and `CNF`. The transformations were compiled with our COKO compiler (described in Section 3.3) into C++ code which in turn was compiled on Sparcstation 10's using the Sun C++ compiler. For each height class, the three transformations were run on the same 25 randomly generated queries. The average elapsed times for each class of queries are shown.

Height	`CNF_TD`	`CNF_BU`	`CNF`
4	0.17	0.17	0.15
5	0.42	0.48	0.20
6	1.19	2.05	0.35
7	3.24	4.18	0.59
8	7.71	12.95	1.35

Table 2: Average Times (sec) for `CNF_TD`, `CNF_BU` and `CNF`

$$\begin{array}{lcl}
\textbf{set}\ !\ A & = & \{x \mid x^i \in A\} \\
\textbf{sum}\ !\ A & = & \sum_{x_i \in A} xi \quad \text{(similarly for \textbf{avg}, \textbf{count})} \\
\textbf{max}\ !\ A & = & \max_{x_i \in A} x \quad \text{(similarly for \textbf{min})}
\end{array}$$

Table 4: KOLA Query Primitives

`CNF` exhibits far better performance than both exhaustive transformations. Intuitively, this is because `CNF` is discriminating in how it fires rules:

- Successful firing of either `d1` and `d2` requires both exhaustive transformations to perform additional passes over the entire query tree. On the other hand, successful firing of either rule requires `CNF` to perform passes over only selected parts of the query tree.
- The exhaustive transformations require a complete pass of failed rule firings in order to terminate. This pass is not required by transformation `CNF`.

The savings in rule firings is illustrated by considering how all three transformations transform the KOLA predicate,

$$((\texttt{P}_\texttt{k}\ \&\ \texttt{Q}_\texttt{k}\ \&\ \texttt{R}_\texttt{k}) \mid \texttt{S}_\texttt{k})\ ?\ [e,\ d].$$

As shown before, `CNF` performs one complete and two partial passes of this predicate's representation. In all, `CNF` fires rules 20 times with two firings succeeding. On the other hand, transformation `CNF_BU` performs three complete passes for a total of 52 rule firings, with two succeeding. `CNF_TD` does a little better than `CNF_BU`, requiring only two passes and 42 rule firings, again with two succeeding.

In this section, we introduced COKO with a practical yet simple example. The normalization of predicates into CNF is a necessary preprocessing step for many useful rewrites such as ordering the application of selection predicates. We showed a COKO transformation (`CNF`) that performs this rewrite for predicates lacking negations. (A transformation that also processes negations is not much more complex and is presented in [4].) Because its rules are confluent, `CNF` could also use an exhaustive firing algorithm as it likely would be in other systems that provide rule grouping. But exhaustive firing is an inefficient means of performing this transformation. Therefore, this example demonstrates the potential benefits from using customized firing algorithms to express complex query rewrites.

3 COKO-KOLA

COKO transformations are expressed with sets of rewrite rules and firing algorithms that control their firing. In this section, we review KOLA queries (Section 3.1), present COKO (Section 3.2) and the COKO compiler (Section 3.3).

3.1 Rewrite Rules and KOLA Queries

KOLA primitives and formers can not only express functions and predicates over simple objects and values but also functions and

$$\begin{aligned}
\textbf{iterate}\ (p,\ f)\ !\ A &= \{\!\!\{(f\ !\ x)^i \mid x^i \in A,\ p\ ?\ x\}\!\!\} \\
\textbf{join}\ (p,\ f)\ !\ [A,\ B] &= \{\!\!\{(f\ !\ [x,\ y])^{ij} \mid x^i \in A,\ y^j \in B,\ p\ ?\ [x,\ y]\}\!\!\} \\
\textbf{njoin}\ (p,\ f,\ g)\ !\ [A,\ B] &= \{[x,\ g\ !\ \{\!\!\{(f\ !\ y) \mid y^j \in B,\ p\ ?\ [x,\ y]\}\!\!\}] \mid x^i \in A\}
\end{aligned}$$

Table 3: KOLA Query Formers

predicates over collections (i.e., query functions and predicates). Table 3 gives the semantics of the KOLA query operators used in this paper. [5] and [4] provide more comprehensive descriptions of KOLA. Of these, the latter is more up-to-date. Since [5], we have defined KOLA over multisets and not just sets, so as to be able to capture the semantics of OQL and SQL.

KOLA collections are (finite) multisets and sets. Sets are special cases of multisets whose element counts are 0 or 1, and therefore can be substituted for multisets whenever multisets are expected. We use set comprehensions to denote sets, and the following notation in discussing multisets:

- "$x^i \in A$" (for $i > 0$) indicates that there are exactly i copies of x in the multiset, A.
- "$\{\!\!\{f(x)^{g(i)} \mid x^i \in A,\ p(x)\}\!\!\}$" denotes a multiset that is formed by inserting $g(i)$ copies of $f(x)$ for every x that satisfies p and that has i copies in A. More precisely, for any element v and $k > 0$, $v^k \in \{\!\!\{f(x)^{g(i)} \mid x^i \in A,\ p(x)\}\!\!\}$ iff
$$k = \sum_{x^i \in A, p(x), f(x)\ ==\ v} g(i).$$

Tables 3 and 4 show the KOLA query primitives and formers used in this paper. Table 4 shows primitives to remove duplicates (**set**) and aggregate (**avg**, **max**, **min**, **count** and **sum**). Table 3 shows formers **iterate** (based on SQL's `select ... from ... where` construct), **join**, and **njoin** (short for *nested join* and based on SQL's `group by`). The meanings of KOLA queries can be derived in the same manner as basic predicates and functions using the definitions of these tables. To illustrate, below we trace the meaning of the KOLA query of Figure 5b by showing the meanings of (1) $\texttt{Emp}'$, (2) $\texttt{Dept}_\texttt{k}$ and (3) the query as a whole.

1. By the definition of **iterate**, **iterate** $(\mathrm{K}_p\ (\texttt{T}),\ \texttt{Dno})\ !\ \texttt{Emp}$ reduces to
$$\texttt{Emp}' = \{\!\!\{(e.\texttt{Dno})^i \mid e^i \in \texttt{Emp}\}\!\!\}$$
and therefore is the result of projecting `Dno` on `Emp`.

2. By the definition of **njoin**, $\texttt{Dept}_\texttt{k}$ reduces to

$$\{\!\!\{[dno,\ \textbf{avg}\ !\ \{\!\!\{(e.\texttt{Sal})^j \mid e^j \in \texttt{Emp},\ dno\ ==\ e.\texttt{Dno}\}\!\!\}] \mid dno^i \in \texttt{Emp}'\}\!\!\}$$

which pairs department numbers of employees in `Emp` with the average salaries of employees in those departments. (Therefore, $\texttt{Dept}_\texttt{k}$ has the same semantics as `Dept`, except that resulting fields are not named.)

3. The join of `Emp` and $\texttt{Dept}_\texttt{k}$ reduces to

$$\begin{aligned}\{\!\!\{(e.\texttt{Eno})^{ij} \mid e^i \in \texttt{Emp},\ [dno,\ asal]^j \in \texttt{Dept}_\texttt{k}, \\ (\texttt{P}'_\texttt{k}\ \&\ \texttt{Q}'_\texttt{k}\ \&\ \texttt{R}_\texttt{k})\ ?\ [e,\ [dno,\ asal]]\}\!\!\}\end{aligned}$$

The reader can verify that this expression also describes the semantics of the SQL query of Figure 5a by reducing

$$\begin{aligned}
\texttt{P}'_\texttt{k}\ ?\ [e,\ [dno,\ asal]] &\quad \textit{to} \quad e.\texttt{Dno} == dno, \\
\texttt{Q}'_\texttt{k}\ ?\ [e,\ [dno,\ asal]] &\quad \textit{to} \quad e.\texttt{Sal} > asal \text{ and} \\
\texttt{R}_\texttt{k}\ ?\ [e,\ [dno,\ asal]] &\quad \textit{to} \quad e.\texttt{Job} == \text{``Mgr''}.
\end{aligned}$$

3.2 Firing Algorithms and COKO

The semantics of statements that appear in COKO transformation firing algorithms have two parts:

- their *operation* (what they do when executed), and
- their *success value* (what they return as a result of execution).

Success values are truth values that indicate if a statement succeeds when invoked. For example, invoked rules return success values of *true* if their lhs match the expressions on which they are fired.

Below we present COKO's statements, categorizing them by the kinds of firing control they provide. Each is presented with its *operation* and *success value* semantics. For the purposes of discussion, we assume that statements are contained in some transformation that has been fired on some KOLA query tree, p.

3.2.1 Explicit Firing

Rules and transformations declared in the `USES` section of a transformation can be fired as if they were procedures invoked in a programming language such as C. Rules and transformations can be fired on subtrees of p named by pattern variables (see Section 3.2.3) or can be fired on p directly in which case, no argument needs to be named in the call. As well, rules that use the same pattern variables in their heads and their bodies can be fired *inversely*. For example, `d1` of Figure 4 could be fired inversely (`d1 INV`) to factor a common subexpression (`r`) from a conjunction of disjuncts. Rule (and inverse rule) invocations return *true* as their success values if they successfully fire. Transformation invocations succeed if the complex statements which are their main bodies succeed (see Section 3.2.4).

3.2.2 Traversal Control

Query trees can be traversed in bottom-up (postorder) or top-down (preorder) fashion. For any statement S, "`BU` S" performs a bottom-up pass of p executing S on every subtree. (Analagously, "`TD` S" executes S on every subtree during a top-down pass of p.) Both traversal statements return a success value of *true* if S succeeds when fired on some subtree visited during the traversal. Whereas "`TD` S" and "`BU` S" execute S on every subtree rooted by a node visited during a pass over p, "`REPEAT` S" fires S repeatedly on p until S no longer succeeds.

3.2.3 Selective Firing

Rules and transformations need not be fired on p and can instead be fired on isolated subtrees of p. These subtrees are identified by matching *patterns* to query trees using COKO's `GIVEN` statement. A pattern is half of a rewrite rule. It resembles a KOLA expression, but can include pattern variables and "don't care variables" ("_": as in Prolog) that get bound by matching. By naming variables as arguments in subsequent rule or transformation firings, the subtrees bound to these variables are selectively transformed.

The COKO matching statement, `GIVEN`, identifies patterns with variables, producing environments for use in subsequent statements. A `GIVEN` statement has the form,

$$\texttt{GIVEN}\ eqn_1, \ldots, eqn_n\ \texttt{DO}\ S.$$

```
select e.Eno                      join (P'_k & Q'_k & R_k, Eno ∘ π1) !
from Emp e, Dept d                 [Emp, Dept_k]
where P AND Q AND R

Dept (Dno, ASal) =                Dept_k =
 select Dno, AVG (Sal)            njoin (eq ⊕ ⟨π1, Dno ∘ π2⟩, Sal, avg) !
 from Emp                          [iterate (K_p (T), Dno) ! Emp, Emp]
 group by Dno

P = e.Dno == d.Dno                P'_k = eq ⊕ ⟨Dno ∘ π1, π1 ∘ π2⟩
Q = e.Sal > d.ASal                Q'_k = gt ⊕ ⟨Sal ∘ π1, π2 ∘ π2⟩
R = e.Job == "Mgr"                R_k = eq ⊕ ⟨Job ∘ π1, K_f ("Mgr")⟩
        (a)                                  (b)
```

Figure 5: An SQL Join Query (a) and Its KOLA Equivalent (b)

ρ AND σ AND τ *such that*

$$\begin{array}{rcl} \rho & = & (\mathtt{R\ OR\ S}) \\ \sigma & = & \mathtt{TRUE} \\ \tau & = & ((\mathtt{P\ OR\ S})\ \mathtt{AND}\ (\mathtt{Q\ OR\ S})) \end{array}$$

(a)

$(\rho_k \oplus \pi_1)$ & $(\sigma_k \oplus \pi_2)$ & τ_k *such that*

$$\begin{array}{rcl} \rho_k & = & (\mathtt{R2_k} \mid \mathtt{S2_k}) \\ \sigma_k & = & \mathtt{K}_p\ (\mathtt{T}) \\ \tau_k & = & ((\mathtt{P_k} \mid (\mathtt{S2_k} \oplus \pi_1))\ \&\ (\mathtt{Q_k} \mid (\mathtt{S2_k} \oplus \pi_1))) \\ & & \\ \mathtt{R2_k} & = & \mathbf{eq} \oplus (\langle \mathbf{id},\ \mathtt{K}_f\ (\text{"Mgr"})\rangle \circ \mathtt{Job}) \\ \mathtt{S2_k} & = & \mathbf{gt} \oplus (\langle \mathbf{id},\ \mathtt{K}_f\ (\mathtt{100K})\rangle \circ \mathbf{add} \circ \langle \mathtt{Sal},\ \mathtt{Bon}\rangle) \end{array}$$

(b)

Figure 6: The SQL/KOLA Predicates of Figs 1 (a) and 2 (b) in SNF

such that S is any COKO statement, and each "equation", eqn_i is of the form, "$<$*variable*$_i$ $>$ = $<$*pattern*$_i$ $>$" (or just *pattern*$_i$ if this pattern is to be matched with all of p). The processing of eqn_i results in an attempted match of *pattern*$_i$ with the tree previously bound to *variable*$_i$. Successful matching adds the variables appearing in *pattern*$_i$ (and the subtrees that they match with) to an environment that is then visible to equations appearing after eqn_i and S. The success value for the entire GIVEN statement is *true* if all n equations successfully match and S succeeds.

3.2.4 Conditional Firing

One can condition the execution of a COKO statement, S' on the result of a previous statement, S in three ways:

- $S \rightarrow S'$ executes S and then executes S' if S succeeds. This statement succeeds if S succeeds.
- $S \mid\mid S'$ executes S and then executes S' if S fails. This statement succeeds if S or S' succeed.
- S ; S' executes S and then executes S'. This statement succeeds if either S or S' succeed.

"$\rightarrow$" is right-associative. "|" and ";" are left-associative. One can override default associativities using braces ({}).

3.3 The COKO Compiler

We have implemented a compiler for COKO ([12]) that generates C++ classes from COKO transformations. Objects of these generated classes manipulate KOLA trees according to the firing algorithm of the compiled COKO transformation. The compiler has an object-oriented design. Every COKO statement is implemented with its own C++ class. Each of these classes is a subclass of the virtual class, `Statement`, and is obligated to define a method `exec` which takes an environment of variable-to-KOLA tree bindings as input and produces a transformed version of this environment as output. These environments include entries for the trees on which each statement is executed.

The compilation of a COKO transformation generates a new subclass of `Statement`, complete with an implementation of an `exec` method. The `exec` method definition for a compiled transformation simply constructs a tree of COKO `Statement` objects corresponding to the parse structure of the COKO code, and then invokes `exec` on the root. Thus, the definition of a new COKO transformation extends the language, even at the level of its implementation.

4 Another Normalization: "Separated Normal Form"

In this section we describe a novel normalization and show its expression in COKO. The normalization is of binary predicates, (in KOLA, predicates on pairs) and involves isolating those subpredicates that act as unary predicates on just one argument. This normalization is a useful preprocessing step when unary predicates need to be moved to other parts of the query, as in predicate pushdown and magic sets rewrites. Because this normalization "separates" unary and binary subpredicates, we characterize the predicates that result as being in "*Separated Normal Form*" or SNF.

The point of this section is to show that COKO transformations are expressive enough to to be used in place of rules in existing rule-based systems. We make this point in several ways:

- The SNF normalization is more complex than CNF, even firing CNF as part of its firing algorithm. Therefore, COKO is expressive enough to capture complex normalizations.
- SNF is itself fired in the firing algorithms of many transformations that are not normalizations. These include predicate pushdown (Section 4.3.1), and Magic Sets rewrites (Section 4.3.2). These rewrites are expressible in COKO also, showing that COKO can express a wide variety of complex rewrites in modular fashion.
- SNF is not usually thought of as a normalization, and as far as we know has not been expressed before with declarative rewrite rules. Thus, COKO can express rewrites that are usually expressed only with code.

4.1 Definition

A binary SQL predicate over arguments x and y is in SNF if it consists of three conjuncts, ρ, σ and τ such that ρ requires only its first argument, σ requires only its second argument, and τ requires both of its arguments. For example, Figure 6a shows the SNF equivalent of the SQL predicates of Figure 1.

For KOLA predicates, the goal of the SNF normalization is to transform binary predicates into the form,

$$(\rho_k \oplus \pi_1)\ \&\ (\sigma_k \oplus \pi_2)\ \&\ \tau.$$

Given any pair argument $[x,\ y]$, $(\rho_k \oplus \pi_1)\ ?\ [x,\ y] = \rho_k\ ?\ x$ and therefore ρ_k denotes a predicate that requires only the first of its arguments. (Similarly, σ_k denotes a predicate requiring the second argument.) Figure 6b shows the SNF equivalent of the KOLA predicates of Figure 2.

4.2 The COKO Transformation for SNF

In this section, we show a COKO transformation (SNF) that converts KOLA predicates into SNF. The transformation has been simplified for presentation, but in its presented form is able to isolate conjuncts that name attributes from only one relation (such as $(e.\mathtt{Sal} + e.\mathtt{Bon}) > \mathtt{100K}$).

4.2.1 Tracing the Execution

Transformation SNF of Figure 8 consists of the six steps described below. We demonstrate these steps by showing how this transformation rewrites predicate $p = (\mathtt{P_k}\ \&\ \mathtt{Q_k}\ \&\ \mathtt{R_k}) \mid \mathtt{S_k}$ of Figure 2a. into the predicate of Figure 6b. Figure 7 shows the parse tree for this predicate at various stages during the execution of the transformation. The original predicate tree is shown in Figure 7a.

Step 1: The first step of `SNF` fires the transformation `SimpLits` (Simplify Literals) shown in Figure 9. This reduces subpredicates of the form $p \oplus f$ to the forms, $p \oplus \pi_1$, or $p \oplus \pi_2$, if possible. (Appendix B gives an induction proof showing that `SimpLits` accomplishes this goal for a class of predicates that includes $\mathtt{P_k}$, $\mathtt{Q_k}$, $\mathtt{R_k}$, and $\mathtt{S_k}$.) More precisely, `SimpLits` transforms any *qualification* predicate [16] lacking subfunctions of the form, $f \circ \pi_1$, to either of the forms, $p \oplus \pi_2$, or $\mathrm{K}_p\ (x)$ (and similarly for predicates lacking $f \circ \pi_2$ as a subfunction).)

`SimpLits` performs a single bottom-up pass of the input predicate, firing rules `sl1`, ..., `sl7` on the subtrees rooted at each visited node. Rules `sl1`, ..., `sl5` only fire successfully on function subtrees while `sl6` and `sl7` only fire successfully on predicate subtrees. Because the pass is bottom-up, a predicate of the form, $op \oplus \langle F,\ G\rangle$ is visited after its subfunctions, F, G and $\langle F,\ G\rangle$.

`SimpLits` transforms $\mathtt{R_k}$ (to $\mathtt{R2_k}$) and $\mathtt{S_k}$ (to $\mathtt{S2_k}$) but has no effect on $\mathtt{P_k}$ or $\mathtt{Q_k}$. In transforming $\mathtt{R_k}$, `SimpLits` first fires rule `sl3` on its subfunction, $\langle \mathtt{Job} \circ \pi_1,\ \mathrm{K}_f\ (\text{"Mgr"})\rangle$. generating $\langle \mathbf{id},\ \mathrm{K}_f\ (\text{"Mgr"})\rangle \circ (\mathtt{Job} \circ \pi_1)$. The successful firing of `sl3` triggers the firing of `sft` leaving $(\langle \mathbf{id},\ \mathrm{K}_f\ (\text{"Mgr"})\rangle \circ \mathtt{Job}) \circ \pi_1$. Finally, rule `sl6` fires on the entire predicate leaving $\mathtt{R2_k} \oplus \pi_1$. When fired on $\mathtt{S_k}$, `SimpLits` first fires `sl5` on its subfunction

$$\langle \mathtt{Sal} \circ \pi_1,\ \mathtt{Bon} \circ \pi_1\rangle$$

leaving $\langle \mathtt{Sal},\ \mathtt{Bon}\rangle \circ \pi_1$. Then, `sl3`, `sft` and `sl6` fire resulting in $\mathtt{S2_k} \oplus \pi_1$. Therefore, the result of firing `SimpLits` on p is

$$p_1 = (\mathtt{P_k}\ \&\ \mathtt{Q_k}\ \&\ (\mathtt{R2_k} \oplus \pi_1)) \mid (\mathtt{S2_k} \oplus \pi_1).$$

Step 2: Next, transformation `CNF` of Figure 4 is fired. Applied to p_1, this results in

$$p_2 = \begin{array}{l} (\mathtt{P_k} \mid (\mathtt{S2_k} \oplus \pi_1))\ \&\ (\mathtt{Q_k} \mid (\mathtt{S2_k} \oplus \pi_1))\ \& \\ ((\mathtt{R2_k} \oplus \pi_1) \mid (\mathtt{S2_k} \oplus \pi_1)) \end{array}$$

as illustrated in Figure 7b.

Step 3: Next, rule `pushneg` is fired in top-down fashion. As p_2 has no negations, this step has no effect in this example.

Step 4: Rule `init` is fired, appending trivial conjuncts to the current predicate to trivially convert it to `SNF`. This sets up **Steps 4** and **5** which merges the conjuncts generated in **Step 2** with these trivial conjuncts. Fired on p_2, this step results in

$$p_3 = (\mathrm{K}_p\ (\mathtt{T}) \oplus \pi_1)\ \&\ (\mathrm{K}_p\ (\mathtt{T}) \oplus \pi_2)\ \&\ \mathrm{K}_p\ (\mathtt{T})\ \&\ p_2.$$

Step 5: This step executes the COKO statement,

$$\mathtt{BU}\ \{\mathtt{pull} \mid\mid \mathtt{REPEAT}\ \{\mathtt{sftp}\}\},$$

such that `pull` and `sftp` are rewrite rules. `pull` can only succeed on disjuncts, while `sftp` can only succeed on conjuncts. The effect of calling `pull` in bottom-up fashion is to "pull" common functions out of disjuncts. For example, the result of executing `pull` on p_3's subpredicate,

$$(\mathtt{R2_k} \oplus \pi_1) \mid (\mathtt{S2_k} \oplus \pi_1)$$

is $(\mathtt{R2_k} \mid \mathtt{S2_k}) \oplus \pi_1$. Therefore, this step converts disjuncts to the form $p \oplus \pi_1$, or, $p \oplus \pi_2$, where possible. The effect of calling `REPEAT{sft}` in bottom-up fashion is to make the resulting predicate tree "left-bushy" in preparation for the following step that orders conjuncts. When applied to p_3, this results in:

$$p_4 = \begin{array}{l} (\mathrm{K}_p\ (\mathtt{T}) \oplus \pi_1)\ \&\ (\mathrm{K}_p\ (\mathtt{T}) \oplus \pi_2)\ \&\ \mathrm{K}_p\ (\mathtt{T})\ \& \\ (\mathtt{P_k} \mid (\mathtt{S2_k} \oplus \pi_1))\ \&\ (\mathtt{Q_k} \mid (\mathtt{S2_k} \oplus \pi_1))\ \& \\ ((\mathtt{R2_k} \mid \mathtt{S2_k}) \oplus \pi_1) \end{array}$$

as illustrated in Figure 7c.

Step 6: This step fires the transformation `OrderConjs` shown in Figure 9. This transformation performs a bottom-up pass, firing rules `oc1`, `oc2` and `oc3` on subtrees of the form,

$$(p \oplus \pi_1)\ \&\ (q \oplus \pi_2)\ \&\ r\ \&\ S$$

for predicates p, q, r and S. Because of steps **3** and **4**, the first subtree visited with this form has p, q and r equal to $\mathrm{K}_p\ (\mathtt{T})$, and S equal to a conjunct from the original predicate. The structure of of S determines which rule gets fired:

- if S is of the form, $s \oplus \pi_1$, then s is merged with p by rule `oc1` to form $(p\ \&\ s) \oplus \pi_1$.
- if S is of the form, $s \oplus \pi_2$, then s is merged with q by rule `oc2` to form $(q\ \&\ s) \oplus \pi_2$.
- if S is of any other form, then it is combined with r by rule `oc3` to form $r\ \&\ s$.

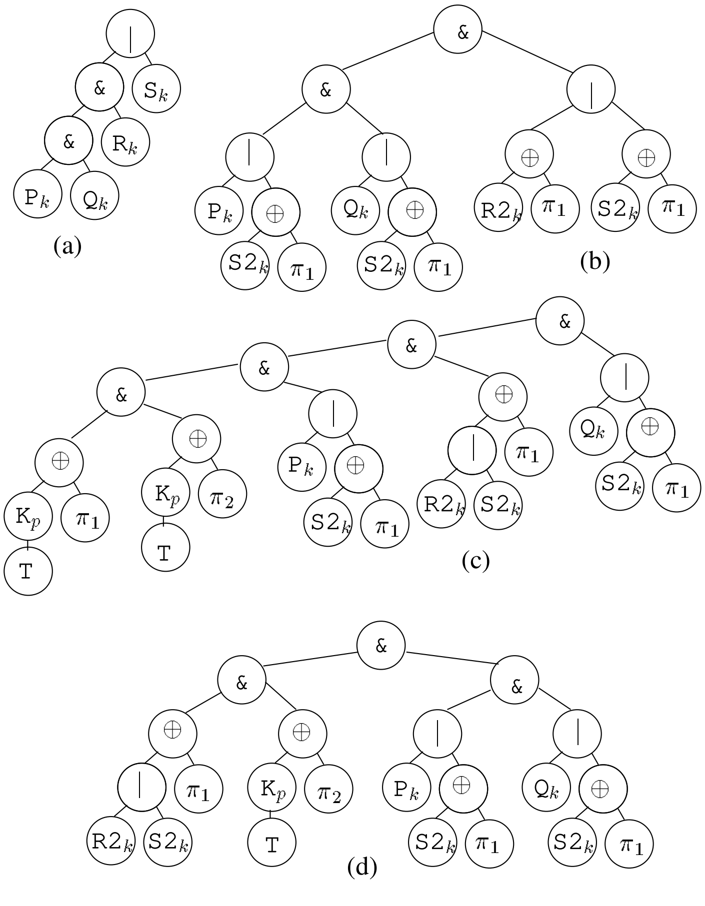

Figure 7: Tracing the effects of `SNF` on the Predicate p of Fig 2a

```
TRANSFORMATION SNF
 USES
  SimpLits, CNF, OrderConjs,
  pushneg :  ~ ((p ⊕ f))          ⇉   (~ (p)) ⊕ f,
  init:      p                    ⇉   (Kp (T) ⊕ π1) &
                                      (Kp (T) ⊕ π2) &
                                       Kp (T) & p,
  pull :     (p ⊕ f) | (q ⊕ f)    ⇉   (p | q) ⊕ f,
  sftp :     p & (q & r)          ⇉   (p & r) & q,
  simp :     Kp (T) & p           ⇉   p
 BEGIN
   SimpLits;                           (1)
   CNF;                                (2)
   TD {pushneg};                       (3)
   init;                               (4)
   BU {pull || {REPEAT sftp}};         (5)
   OrderConjs;                         (6)
   GIVEN (p ⊕ π1) & (q ⊕ π2) & r DO    (7)
     {simp (p); simp (q); simp (r)}
 END
```

Figure 8: The SNF Normalization Expressed in COKO

```
TRANSFORMATION SimpLits
 USES
  sft :  f ∘ (g ∘ h)           ⇉   (f ∘ g) ∘ h
  sl1 :  ⟨Kf (x), Kf (y)⟩      ⇉   Kf ([x, y]),
  sl2 :  f ∘ Kf (x)            ⇉   Kf (f ! x),
  sl3 :  ⟨f, Kf (x)⟩           ⇉   ⟨id, Kf (x)⟩ ∘ f,
  sl4 :  ⟨Kf (x), f⟩           ⇉   ⟨Kf (x), id⟩ ∘ f,
  sl5 :  ⟨f ∘ h, g ∘ h⟩        ⇉   ⟨f, g⟩ ∘ h,
  sl6 :  p ⊕ (f ∘ g)           ⇉   p ⊕ f ⊕ g,
  sl7 :  p ⊕ Kf (x)            ⇉   Kp (p ? x)
 BEGIN
  BU {sl1 || sl2 || {{sl3 || sl4} → {REPEAT sft}} ||
      sl5 || sl6 || sl7 || {REPEAT sft}}
 END

TRANSFORMATION OrderConjs
 USES
  oc1 :  (p ⊕ π1) & (q ⊕ π2) & r & (s ⊕ π1)   ⇉
         ((p & s) ⊕ π1) & (q ⊕ π2) & r,
  oc2 :  (p ⊕ π1) & (q ⊕ π2) & r & (s ⊕ π2)   ⇉
         (p ⊕ π1) & ((q & s) ⊕ π2) & r,
  oc3 :  (p ⊕ π1) & (q ⊕ π2) & r & s          ⇉
         (p ⊕ π1) & (q ⊕ π2) & (r & s)
 BEGIN
  BU {oc1 || oc2 || oc3}
 END
```

Figure 9: Auxiliary Transformations Used by SNF

Therefore, the effect of this transformation is to order the conjuncts of the predicate produced in **Step 4** into a predicate that is in SNF. The effect of this step on p_4 is to transform it to

$$p_5 = ((\mathrm{K}_p\ (\mathtt{T})\ \&\ \rho_k) \oplus \pi_1)\ \&\ (\sigma_k \oplus \pi_2)\ \&\ \mathrm{K}_p\ (\mathtt{T})\ \&\ \tau_k$$

such that ρ_k, σ_k and τ_k are as defined in Figure 6b.

Step 7: Finally, this step fires the rule, `simp` to get rid of the $\mathrm{K}_p\ (\mathtt{T})$ predicates that were added in **Step 3**. (Note that σ_k of p_5 above is not affected by this step.) The effect of this step on p_5 is to produce the final KOLA predicate of Figure 6b. (This is illustrated in Figure 7d.)

$$\textbf{join}\ ((\mathtt{p} \oplus \pi_1)\ \&\ (\mathtt{q} \oplus \pi_2)\ \&\ \mathtt{r},\ \mathtt{f})\ !\ [\mathtt{A},\ \mathtt{B}]$$
$$\stackrel{\rightarrow}{=}$$
$$\textbf{join}\ (\mathtt{r},\ \mathtt{f})\ !\ [\textbf{iterate}\ (\mathtt{p},\ \textbf{id})\ !\ \mathtt{A},\ \textbf{iterate}\ (\mathtt{q},\ \textbf{id})\ !\ \mathtt{B}]$$

Figure 10: `push` : A KOLA Rule to Push Predicates Past Joins

4.3 Two Applications of SNF

In this section, we present two COKO transformations that depend on prior normalization by `SNF`: predicate pushdown (Section 4.3.1) and Magic Sets (Section 4.3.2). Space does not permit a full presentation of these transformations. Instead, we describe the firing algorithms and relevant rewrite rules of these transformations and refer interested readers to more complete descriptions in [4].

4.3.1 Predicate Pushdown

Figure 10 shows a KOLA rewrite rule that pushes predicates past joins. This rule identifies predicates in join queries that apply only to one argument (`p` and `q`) and "pushes" them out of the join and onto the join inputs. This is a useful heuristic as it will usually result in a join of smaller collections.

This rule will not succeed on every join query for the predicate used in the join may not be in a form that makes "pushable" subpredicates recognizable. For example, if this rule were fired on the query, $\textbf{join}\ (p,\ \mathtt{Eno} \circ \pi_1)\ !\ [\mathtt{Emp},\ \mathtt{Dept_k}]$ such that p were the predicate of Figure 2a, it would fail because this predicate does not match the pattern, $(\mathtt{p} \oplus \pi_1)\ \&\ (\mathtt{q} \oplus \pi_2)\ \&\ \mathtt{r}$. Before this rule is fired, the predicate argument to **join** must be normalized into SNF so that "pushable" subpredicates can be identified. In the case of the predicate of Figure 2a, normalization into SNF results in the query,

$$\textbf{join}\ ((\rho_k \oplus \pi_1)\ \&\ (\sigma_k \oplus \pi_2)\ \&\ \tau_k,\ \mathtt{Emp} \circ \pi_1)\ !\ [\mathtt{Emp},\ \mathtt{Dept_k}]$$

such that ρ_k, σ_k and τ_k are as defined in Figure 6b. Once in this form, firing `push` results in,

$$\textbf{join}\ (\tau_k,\ \mathtt{Eno} \circ \pi_1)\ !\ [\textbf{iterate}\ (\rho_k,\ \textbf{id})\ !\ \mathtt{Emp},\ \textbf{iterate}\ (\sigma_k,\ \textbf{id})\ !\ \mathtt{Dept_k}].$$

Note that in this case, σ_k is $\mathrm{K}_p\ (\mathtt{T})$. Therefore firing a subsequent rule,

$$\textbf{iterate}\ (\mathrm{K}_p\ (\mathtt{T}),\ \textbf{id})\ !\ \mathtt{A} \stackrel{\rightarrow}{=} \mathtt{A}$$

would result in the query,

$$\textbf{join}\ (\tau_k,\ \mathtt{Eno} \circ \pi_1)\ !\ [\textbf{iterate}\ (\rho_k,\ \textbf{id})\ !\ \mathtt{Emp},\ \mathtt{Dept_k}].$$

4.3.2 Magic Sets

The idea behind the Magic Sets transformation for relational queries [14] is to restrict inputs to joins by filtering those that cannot possibly satisfy the join predicate. Therefore, this transformation is very much in the spirit of predicate pushdown, but passing filter predicates "sideways" from one join input to another, rather than "down" from the join predicate.

Figure 11 shows the results of applying magic sets transformations to the SQL and KOLA queries of Figure 5. The SQL query (Figure 11a) introduces new view definitions, `MEmp`, `MDept` and `DSet`. (The KOLA analogs to these are $\mathtt{MEmp_k}$, $\mathtt{MDept_k}$ and $\mathtt{DSet_k}$.) `MEmp` and `MDept` replace `Emp` and `Dept` in the original query respectively. `MEmp` filters employees in `Emp` for those that are managers (`R`). Like $\mathtt{Dept_k}$, `MDept` groups employess in `Emp` by their departments before finding the average salary for each department. But `MDept` is potentially far more efficient to evaluate than `Dept` because `MDept` only computes average salaries for departments that have managers in Emp, whereas Dept computed average salaries for all departments. The filtered set of departments used by MDept is defined by the (*magic*) view DSet.

We have deviated from the Magic Sets transformation presented in [14] in two respects. First, the Magic Sets transformation of [14] would express DSet as a view over MEmp rather than over Emp to avoid the redundant work of computing which employees are managers. Second, MEmp would return a relation with attributes, Dno, Eno, and Sal rather than every attribute of Emp, as the other attributes are not necessary to compute the rest of the query. We express DSet as a view over Emp to simplify presentation; [4] presents a COKO transformation that generates DSet properly. The second deviation is more problematic to address, for it may require a COKO transformation to examine an entire query to see which attributes of Emp are used elsewhere. COKO is not designed to express this kind of "global" transformation. We have yet to determine whether transformations such as this can be expressed incrementally (i.e., by iteratively applying "local" rewrite rules). We believe that they can, and note that SimpLits performed a similar "global" transformation (finding all references in a binary predicate to one of its inputs). However, we have not yet explored this to any depth. If such a transformation cannot be expressed with COKO, we intend to extend the language of firing algorithms so that it can.

The key rewrite rule of the COKO transformation that performs magic sets is magic, shown in Figure 12. magic assumes that a predicate argument to a join query has first been normalized into the form,

$$(\texttt{q} \oplus (\textbf{id} \times \pi_1)) \ \& \ \texttt{r}.$$

This isolates the subpredicate, q which relates elements of A and B. magic then rewrites the query so that q is used to "filter" elements of B so that only those that relate to some element of A will be involved in the grouping/aggregate computation performed by **njoin**. This filtering is expressed in the rhs of magic with the subexpression, $\textbf{join}\ (\texttt{q},\ \pi_2)\ !\ [\texttt{A},\ \texttt{B}]$ which performs a right semi-join of A and B with respect to q. The elements of B that are collected from this query are then freed of duplicates via a subsequent invocation of **set**. In many cases, the semi-join expression can be further simplified (for example, if A = B). Thus, the COKO version of the Magic Sets transformation is quite similar to Pushdown and many other transformations that implement heuristics, in that it consists of three high-level steps: (1) **Normalize**, (2) **Apply a "main rule"** (e.g., magic or push), and (3) **Simplify**.

```
select e.Eno
from MEmp e, MDept d
where P AND Q
```

```
MEmp =
  select *
  from Emp e
  where R
```

```
MDept (Dno, ASal) =
  select e.Dno, AVG (e.Sal)
  from DSet d, Emp e
  where d.Dno == e.Dno
  group by d.Dno
```

```
DSet (Dno) =
  select distinct e.Dno
  from Emp e
  where R
```

```
P = e.Dno == d.Dno
Q = e.Sal > d.ASal
R = e.Job == "Mgr"
```

(a)

$$\textbf{join}\ (\texttt{P}'_\texttt{k}\ \&\ \texttt{Q}'_\texttt{k},\ \texttt{Eno} \circ \pi_1)\ !\ [\texttt{MEmp}_\texttt{k},\ \texttt{MDept}_\texttt{k}]$$

$$\begin{array}{rcl}
\texttt{MEmp}_\texttt{k} & = & \textbf{iterate}\ (\texttt{R}'_\texttt{k},\ \textbf{id})\ !\ \texttt{Emp} \\
\texttt{MDept}_\texttt{k} & = & \textbf{njoin}\ (\textbf{eq} \oplus \langle \pi_1,\ \texttt{Dno} \circ \pi_2 \rangle,\ \texttt{Sal},\ \textbf{avg})\ ! \\
 & & [\texttt{DSet}_\texttt{k},\ \texttt{Emp}] \\
\texttt{DSet}_\texttt{k} & = & \textbf{set}\ !\ (\textbf{iterate}\ (\texttt{R}'_\texttt{k},\ \texttt{Dno})\ !\ \texttt{Emp}) \\
\texttt{P}'_\texttt{k} & = & \textbf{eq} \oplus (\texttt{Dno} \times \pi_1) \\
\texttt{Q}'_\texttt{k} & = & \textbf{gt} \oplus (\texttt{Sal} \times \pi_2) \\
\texttt{R}'_\texttt{k} & = & \textbf{eq} \oplus (\langle \textbf{id},\ K_f\ (\text{"Mgr"}) \rangle \circ \texttt{Job})
\end{array}$$

(b)

Figure 11: Queries of Fig 5a and 5b After Rewriting by Magic Sets

$$\begin{array}{l}
\textbf{join}\ ((\texttt{q} \oplus (\textbf{id} \times \pi_1))\ \&\ \texttt{r},\ \texttt{f})\ \ \ ! \\
\quad [\texttt{A},\ \textbf{njoin}\ (\texttt{s},\ \texttt{g},\ \texttt{h})\ !\ [\texttt{B},\ \texttt{C}]] \\
\qquad\qquad\qquad \rightrightarrows \\
\textbf{join}\ ((\texttt{q} \oplus (\textbf{id} \times \pi_1))\ \&\ \texttt{r},\ \texttt{f})\ \ \ ! \\
\quad [\texttt{A},\ \textbf{njoin}\ (\texttt{s},\ \texttt{g},\ \texttt{h})\ !\ [\textbf{set}\ !\ (\textbf{join}\ (\texttt{q},\ \pi_2)\ !\ [\texttt{A},\ \texttt{B}]),\ \texttt{C}]]
\end{array}$$

Figure 12: magic: A KOLA Rule to Perform Magic Sets

Below we describe the steps performed by this transformation, demonstrating their effects on the KOLA query of Figure 5b, and showing how they result in a rewrite of this query to that shown in Figure 11b. In practice, translation into KOLA performs view merging, and therefore the query of Figure 5b would be presented to the COKO Magic Sets transformation as Q_0:

$$\begin{array}{l}
\textbf{join}\ (\texttt{P}'_\texttt{k}\ \&\ \texttt{Q}'_\texttt{k}\ \&\ \texttt{R}_\texttt{k},\ \texttt{Eno} \circ \pi_1)\ ! \\
\quad [\texttt{Emp},\ \textbf{njoin}\ (\textbf{eq} \oplus \langle \pi_1,\ \texttt{Dno} \circ \pi_2 \rangle,\ \texttt{Sal},\ \textbf{avg})\ ! \\
\qquad [\textbf{iterate}\ (K_p\ (\texttt{T}),\ \texttt{Dno})\ !\ \texttt{Emp},\ \texttt{Emp}]]
\end{array}$$

such that $\texttt{P}'_\texttt{k}$, $\texttt{Q}'_\texttt{k}$ and $\texttt{R}_\texttt{k}$ are as defined in Figure 5b.

Step 1: Normalization and Pushdown The first step of the transformation normalizes the predicate appearing inside **join** into the form,

$$(\texttt{q} \oplus (\textbf{id} \times \pi_1))\ \&\ \texttt{r}$$

to prepare it for the magic rule (Figure 12). This normalization requires three steps:

1. Firing SNF on the join predicate to convert it to the form,

$$(\rho_k \oplus \pi_1)\ \&\ (\sigma_k \oplus \pi_2)\ \&\ \tau_k.$$

2. Firing Pushdown on the entire query to push ρ_k and σ_k out of the join (and then simplifying).
3. Firing the transformation SNF2 (not shown) on τ_k. SNF2 is similar to SNF, but rewrites τ_k into the form,

$$(\alpha_k \oplus (\textbf{id} \times \pi_1))\ \&\ (\beta_k \oplus (\textbf{id} \times \pi_2))\ \&\ \gamma_k.$$

After these steps, the predicate is easily put into the desired form by setting $\texttt{q} = \alpha_k$ and $\texttt{r} = (\beta_k \oplus (\textbf{id} \times \pi_2))\ \&\ \gamma_k$. Fired on Q_0, this step returns Q_1:

$$\begin{array}{l}
\textbf{join}\ ((\texttt{q} \oplus (\textbf{id} \times \pi_1))\ \&\ \texttt{Q}'_\texttt{k},\ \texttt{Eno} \circ \pi_1)\ ! \\
\quad [\texttt{MEmp}_\texttt{k},\ \textbf{njoin}\ (\textbf{eq} \oplus \langle \pi_1,\ \texttt{Dno} \circ \pi_2 \rangle,\ \texttt{Sal},\ \textbf{avg})\ ! \\
\qquad [\textbf{iterate}\ (K_p\ (\texttt{T}),\ \texttt{Dno})\ !\ \texttt{Emp},\ \texttt{Emp}]]
\end{array}$$

such that

$$\begin{array}{rcl}
\texttt{MEmp}_\texttt{k} & = & \textbf{iterate}\ (\texttt{R}'_\texttt{k},\ \textbf{id})\ !\ \texttt{Emp}, \\
\texttt{q} & = & \textbf{eq} \oplus (\texttt{Dno} \times \textbf{id}),\ \text{and} \\
\texttt{R}'_\texttt{k} & = & \textbf{eq} \oplus (\langle \textbf{id},\ K_f\ (\text{"Mgr"}) \rangle \circ \texttt{Job}).
\end{array}$$

(Note that $\texttt{P}'_\texttt{k} = \texttt{q} \oplus (\textbf{id} \times \pi_1)$.)

1.	$(\mathbf{id} \times \mathbf{id})$	$\stackrel{\rightarrow}{=}$	$\mathbf{id}$
2.	$\mathtt{p} \oplus \mathbf{id}$	$\stackrel{\rightarrow}{=}$	$\mathtt{p}$
3.	$\mathtt{K}_p\ (\mathtt{T}) \oplus \mathtt{f}$	$\stackrel{\rightarrow}{=}$	$\mathtt{K}_p\ (\mathtt{T})$
4.	$\mathtt{p}\ \&\ \mathtt{K}_p\ (\mathtt{T})$	$\stackrel{\rightarrow}{=}$	$\mathtt{p}$
5.	$\mathtt{f} \circ \mathbf{id}$	$\stackrel{\rightarrow}{=}$	$\mathtt{f}$
6.	$(\mathtt{f} \times \mathtt{g}) \circ (\mathtt{h} \times \mathtt{j})$	$\stackrel{\rightarrow}{=}$	$(\mathtt{f} \circ \mathtt{h}) \times (\mathtt{g} \circ \mathtt{j})$
7.	$\mathtt{p} \oplus \mathtt{f} \oplus \mathtt{g}$	$\stackrel{\rightarrow}{=}$	$\mathtt{p} \oplus (\mathtt{f} \circ \mathtt{g})$

8. $\mathbf{set}\ !\ (\mathbf{join}\ (\mathbf{eq},\ \pi_2)\ !\ [\mathtt{A},\ \mathtt{B}]) \stackrel{\rightarrow}{=} \mathbf{set}\ !\ (\mathtt{A}\ \cap\ \mathtt{B})$

9. $\mathbf{join}\ (\mathtt{p} \oplus (\mathtt{f} \times \mathtt{g}),\ \pi_2)\ !\ [\mathtt{A},\ \mathtt{B}] \stackrel{\rightarrow}{=}$
$\mathbf{join}\ (\mathtt{p} \oplus (\mathbf{id} \times \mathtt{g}),\ \pi_2)\ !\ [\mathbf{iterate}\ (\mathtt{K}_p\ (\mathtt{T}),\ \mathtt{f})\ !\ \mathtt{A},\ \mathtt{B}]$

10. $\mathbf{iterate}\ (\mathtt{p},\ \mathtt{f})\ !\ (\mathbf{iterate}\ (\mathtt{q},\ \mathtt{g})\ !\ \mathtt{A}) \stackrel{\rightarrow}{=}$
$\mathbf{iterate}\ (\mathtt{q}\ \&\ (\mathtt{p} \oplus \mathtt{g}),\ \mathtt{f} \circ \mathtt{g})\ !\ \mathtt{A}$

11. $(\mathbf{iterate}\ (\mathtt{p},\ \mathtt{f})\ !\ \mathtt{A})\ \cap\ (\mathbf{iterate}\ (\mathtt{K}_p\ (\mathtt{T}),\ \mathtt{f})\ !\ \mathtt{A}) \stackrel{\rightarrow}{=}$
$\mathbf{iterate}\ (\mathtt{p},\ \mathtt{f})\ !\ \mathtt{A}$

12. $\mathbf{njoin}\ (\mathtt{p},\ \mathtt{f},\ \mathtt{g})\ !\ [\mathbf{set}\ !\ \mathtt{A},\ \mathtt{B}] \stackrel{\rightarrow}{=}$
$\mathbf{njoin}\ (\mathtt{p},\ \mathtt{f},\ \mathtt{g})\ !\ [\mathtt{A},\ \mathtt{B}]$

Table 5: Simplification Rules Used After Invoking `magic`

Step 2: Magic Next rewrite rule `magic` (Figure 12) is fired. This rule introduces the left join argument (A) into the right-hand side of the join. This makes it possible to restrict the input (B) to **njoin** to elements that are related by q to A. Fired on Q_1, this returns Q_2:

$$\mathbf{join}\ ((\mathtt{q} \oplus (\mathbf{id} \times \pi_1))\ \&\ \mathtt{Q}'_\mathtt{k},\ \mathtt{Eno} \circ \pi_1)\ !\ [\mathtt{MEmp_k},\ \mathbf{njoin}\ (\mathbf{eq} \oplus \langle \pi_1,\ \mathtt{Dno} \circ \pi_2 \rangle,\ \mathtt{Sal},\ \mathbf{avg})\ !\ [\mathtt{DSet_{1k}},\ \mathtt{Emp}]]$$

such that $\mathtt{DSet_{1k}}$ is:

$$\mathbf{set}\ !\ (\mathbf{join}\ (\mathtt{q},\ \pi_2)\ !\ [\mathtt{MEmp_k},\ \mathbf{iterate}\ (\mathtt{K}_p\ (\mathtt{T}),\ \mathtt{Dno})\ !\ \mathtt{Emp}]).$$

Step 3: Simplify The last step uses the rules of Table 5 to simplify $\mathtt{DSet_{1k}}$ to construct $\mathtt{DSet_k}$ as defined in Figure 11b. As a result of firing rule 9, followed by rules 1, 2 and 8, $\mathtt{DSet_{1k}}$ gets transformed into $\mathtt{DSet_{2k}}$:

$$\mathbf{set}\,!\ ((\mathbf{iterate}\ (\mathtt{K}_p\ (\mathtt{T}),\ \mathtt{Dno})\ !\ \mathtt{MEmp_k})\ \cap\ (\mathbf{iterate}\ (\mathtt{K}_p\ (\mathtt{T}),\ \mathtt{Dno})\ !\ \mathtt{Emp})).$$

Expansion for $\mathtt{MEmp_k}$ reveals that rule 10 can be used to combine **iterate** functions. Subsequent simplification by rules 3, 4 and 5 leaves $\mathtt{DSet_{3k}}$:

$$\mathbf{set}\ !\ (\mathbf{iterate}\ (\mathtt{R}'_\mathtt{k},\ \mathtt{Dno})\ !\ \mathtt{Emp}\ \cap\ \mathbf{iterate}\ (\mathtt{K}_p\ (\mathtt{T}),\ \mathtt{Dno})\ !\ \mathtt{Emp}).$$

Rule 11 makes the intersection unnecessary leaving

$$\mathtt{DSet_k} = \mathbf{set}\ !\ (\mathbf{iterate}\ (\mathtt{R}'_\mathtt{k},\ \mathtt{Dno})\ !\ \mathtt{Emp}).$$

Application of rules 12 and 6 then leave the query (modulo view merging) of Figure 11b.

5 Discussion

5.1 On the Design and Expressivity of COKO

COKO is still evolving. Our goal is to reach a point with it where we are able to express all of the useful transformations that can be expressed in a programming language, without compromising the separation of query modification (rewrite rules) from firing algorithms. To this end, our design process alternates between modifying the language and using it to express complex transformations. In this paper, we have presented some of the transformations that we have generated with COKO including `CNF` (exhaustive and non-exhaustive versions), `SNF`, `Pushdown` and `MagicSets`. Aside from these, we have also implemented (and generalized) the query unnesting transformations of [11] (modulo their bug fixes) and the nested object query transformation presented in [5]. Details can be found in [4].

Though COKO is evolving, this is not to say that it is immature. We believe that COKO already provides most of the useful idioms required to express query rewriting. By combining COKO statements in varying ways, we are able to express such common firing algorithms as

- `BU` $\{$`r`$_1$ || ... || `r`$_n\}$: for each node of a tree, fire every rule `r`$_1$, ..., `r`$_n$ on the node,
- `BU r`$_1$`; ... BU r`$_n$: for each rule `r`$_1$, ..., `r`$_n$, fire `r`$_i$ on every node,
- `BU` $\{\{$`REPEAT r`$_1\}$ || ... || $\{$`REPEAT r`$_n\}\}$: for each node of a tree, fire every rule `r`$_1$, ..., `r`$_n$ repeatedly on the node until it fails,
- $S \rightarrow S' \ ||\ S''$: perform S and then S' if S succeeded and S'' if it failed, and
- $S \rightarrow$ *recursive fire*: perform any statement S exhaustively.

COKO may yet be extended. We spoke in Section 4 about how we were unsure if certain global "optimizations" could be expressed within COKO in its present form. As well, we have found that many transformations expressed in COKO are similar, perhaps differing by minor points. For example, transformations `SNF` and `SNF2` are similar except that the former generates predicates of the form,

$$(\rho_k \oplus \pi_1)\ \&\ (\sigma_k \oplus \pi_2)\ \&\ \tau_k$$

whereas the other generates predicates of the form,

$$(\rho_k \oplus (\mathbf{id} \times \pi_1))\ \&\ (\sigma_k \oplus (\mathbf{id} \times \pi_2))\ \&\ \tau_k$$

This suggests that a parameterized version of COKO could be a future direction.

As before, attempts to express transformations drive the design of the language. Provided that we are able to maintain the separation of rewrite rules from firing algorithms, we are prepared to let COKO evolve into a steady and usable state.

5.2 Why COKO?

Why is a new language for expressing transformations necessary? Why not use C to express transformations as in Starburst?

Languages such as C were designed for much a broader application than expressing query transformations. Thus, they provide expressivity needed for other applications at the expense of understanding and reasoning. For example, COKO transformations are semantics-preserving if the rewrite rules they fire are semantics-preserving. To accomplish this, COKO has no *assignment statements* and instead permits query modification only with rule firing. Transformations expressed in C might use assignment statements both to modify a query and to change the position of a cursor within the query tree. This makes it difficult to identify what parts of the transformation make changes to query representation, and therefore what parts require verification to ensure transformations preserve query semantics. Also, assignment statements are much finer-grained modification primitives than are rule firings. The changes made to a query by one rule may have to be expressed with several assignment statements, each but the last leaving the query in an inconsistent state. Thus, verification is also complicated by the need to consider the flow of control of the transformation to determine if consistent states are revisited once left.

All of this is not to say that correct transformations cannot be written in C. Of course they can, but incorrect transformations can be written in C also and discerning between them is difficult. Writers of COKO transformations have a far easier task as they are using a language that permits easy identification of proof obligations (i.e., rules), and can use a theorem prover to help with the task of proving the rules correct.

5.3 Future Directions

Aside from extending COKO and implementing new transformations, we are interested in two other directions for this work. First, we have argued the expressivity of COKO in an informal way (i.e., with examples). We would like to explore more formal expressivity metrics for transformation languages. (Note that expressivity metrics for query languages are not the same!) We believe that general-purpose programming languages have more expressivity than is needed to express transformation. On the other hand, the meta-rule languages of other rule-based systems such as [15], [1], and [6] do not give us *enough* expressive power to implement transformations efficiently. A more theoretical analysis of expressivity in this context is called for.

A second interesting direction concerns normalization. Normalization gets little attention in the optimizer literature. And yet, normalization can be more complex, expensive and error-prone than the optimizations they preceed. (Consider the many erroneous unnesting normalizations for nested queries that have been proposed over the years.) In the context of object-oriented and object-relational databases, normalization assumes even greater importance. Firstly, nested object queries are far more prevalent and can be more deeply nested than relational queries making their normalization more difficult and more urgent. Secondly, because many object databases are built as extensions of existing systems (e.g., object-relational extensions of relational systems), normalization affords the opportunity to rewrite complex (e.g., object) queries into a series of simpler (e.g., relational) queries that can be posed of the original query engine.

6 Conclusions

In this paper, we proposed a language (COKO) for defining *transformations*: query rewrites for rule-based optimizers. COKO transformations generalize rewrite rules. Like rules, they can be fired and can succeed or fail as a result. But because they are expressed algorithmically, they are able to express many rewrites (such as normalizations) that rules cannot.

A COKO transformation consists of a set of rewrite rules and a firing algorithm specifying how they are fired. The separation of query modification (expressed with rewrite rules) and firing control (expressed with firing algorithms) achieves a delicate balance of understandability, efficiency and expressivity. Rules and transformations must be understandable and able to be reasoned about if extensibility is to be straightforward. At the same time, they must be efficient and expressive if rule-based optimizers are to be practical.

COKO transformations are easily understood because they are modular; they are built from simpler transformations and declarative rules. Moreover, COKO transformations can be verified as being semantics-preserving simply by proving the same property of the rewrite rules are fired. As we showed in [5], KOLA rules can be verified with a theorem prover. Because of COKO's language for expressing firing algorithms, COKO transformations can be made efficient and expressive. The language for firing algorithms includes the kinds of operators that are most useful for describing rewrites. These include explicit rule firing, traversal control, selective firing and conditional firing. We demonstrated the efficiency benefits of firing algorithms with an example COKO transformation that converts predicates to CNF. We demonstrated the expressivity of COKO with numerous examples that typically do not get expressed with declarative rewrite rules. These examples included a complex normalization (SNF) and two heuristics that depended on it (predicate pushdown and Magic Sets transformations).

This work extends and generalizes our KOLA work. KOLA laid the foundation showing how the choice of query representation determined the ease with which one could reason about queries. Whereas KOLA used a modular approach to build queries from functions, COKO uses a modular approach to build transformations from rewrite rules. The result is a conceptual hierarchy that facilitates the understanding and building of optimizers.

Acknowledgments

We would like to thank Blossom Sumulong and Joachim Kröger and especially, Gail Mitchell for their observations and comments on earlier drafts of this paper. We'd also like to thank Joe Hellerstein for his many useful suggestions.

References

[1] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2):247–303, June 1992.

[2] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS extensible DBMS project: An overview. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1990.

[3] M. Cherniack. Translating queries into combinators. September 1996.

[4] M. Cherniack. Building query optimizers with combinators. Ph.D. Dissertation Proposal. Brown University, 1997.

[5] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, Montréal, Québec, Canada, June 1996.

[6] B. Finance and G. Gardarin. A rule-based query rewriter in an extensible dbms. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 248–256, Kobe, Japan, April 1991. IEEE.

[7] B. Finance and G. Gardarin. A rule-based query optimizer with multiple search strategies. *Data and Knowledge Engineering*, 13:1–29, 1994.

[8] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3):19–29, September 1995.

[9] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Vienna, Austria, April 1993. IEEE.

[10] J. Guttag, J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specifications*. Springer-Verlag, 1992.

[11] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[12] J.-S. Lee, K.-E. Kim, and M. Cherniack. A COKO compiler. Available at http://www.cs.brown.edu/software/cokokola/coko.tar.Z, 1996.

[13] G. Mitchell, U. Dayal, and S. B. Zdonik. Control of and extensible query optimizer: A planning-based approach. In *Proc. 19th Int'l Conference on Very Large Data Bases*, August 1993.

[14] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 247–258, 1990.

[15] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 39–48, San Diego, CA, June 1992.

[16] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1996.

[17] E. Sciore and J. Sieg Jr. A modular query optimizer generator. In *Proceedings of the 6th International Conference on Data Engineering*, pages 146–153, Los Angeles, USA, 1990.

[18] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, Montréal, Québec, Canada, June 1996.

A All Transformation Proof Scripts

Available at ftp://ftp.cs.brown.edu/u/mfc/SIGMOD98_Scripts.lp.

B Proof of Correctness for SimpLits (Figure 9)

Available at ftp://ftp.cs.brown.edu/u/mfc/simplits.ps.Z.