Experiences with Query Translation: Object Queries meet DB2*

Mitch Cherniack†
Dept of Computer Science
Brown University
Providence, RI 02912
mfc@cs.brown.edu
Phone: (401) 863 7673
Fax: (401) 863 7657

Ashok Malhotra
IBM T. J. Watson Research Center,
P.O. Box 704,
Yorktown Heights, NY 10598
petsa@us.ibm.com

Stan Zdonik
Dept of Computer Science
Brown University
Providence, RI 02912
sbz@cs.brown.edu
Phone: (401) 863 7648
Fax: (401) 863 7657

Abstract

COKO-KOLA is a two-tiered framework for building efficient and correct query rewriters. KOLA is a combinator-based (i.e., variable-free) query algebra and representation that permits the expression of basic query rewrites in terms of rewrite rules that can be verified with a theorem prover. COKO is a programming language for expressing complex query rewrites using sets of KOLA rewrite rules supplemented with a (firing) algorithm that controls their application. This paper describes our experiences using COKO-KOLA in an industrial setting. The goal of this work was to build a *translating query rewriter* for an object-oriented database built on top of the relational database DB2, that would translate object queries posed in a subset of OQL into equivalent SQL queries over DB2.

To assess the difficulty of using COKO-KOLA in a practical setting, we reflect on the effort that was required to specify normalizations specific to translation, and the rewrites to do the "actual work" of making object queries relational. Interestingly, the normalization of translated queries proved most difficult to write. But the experience revealed deficiencies in the COKO language and compiler implementation that can be addressed in future versions. Most encouraging was the ease with which we were able to write the transformations that did the actual work of the query rewrite. We were able to express these useful query rewrites with only a few rules and few lines of firing algorithm code.

1 Introduction

Query rewriting [14] is a heuristic phase of query optimization that performs source-to-source rewrites of queries to prepare them for plan generation. Rewrites typically are of one of two kinds:

- *Heuristic improvements* apply heuristics that make queries more efficient to evaluate, regardless of the physical representations or contents of the queried collections (e.g., a query rewrite to remove unnecessary duplicate elimination).
- *Normalizations* rewrite queries that have been posed in a way that forces an optimizer to choose a poor evaluation plan (e.g., nested queries which force nested loop evaluations) into a form that the plan generator is better equipped to process (e.g., into join queries)

In this paper, we present our experiences using COKO-KOLA [3, 4, 5] to develop a query rewriter for San Francisco [15]: an object-oriented database (OODB) being developed at IBM. San Francisco is built on top of the relational database, DB2 [6]. Our rewriter *translates* queries posed in a subset of OQL [1] into equivalent SQL queries over the DB2 implementation. Specifically, object queries with path expressions in their `WHERE` clause get rewritten into relational join queries over the underlying representation. (This is illustrated in Figure 1). Such rewrites are similar to the object $\rightarrow$ relational query mappings defined in [10].

*DB2 is a trademark of the IBM Corporation.
†Contact Author. Some of Mitch Cherniack's work was funded by a grant from the IBM Thomas. J. Watson Research Center

```
SELECT s
FROM s IN Sens
WHERE (s.bornin.popn > 1M) AND (s.pty == "GOP") AND
      (s.reps.capt.sistercity.popn > 1M) AND NOT (s.bornin.popn > 2M) AND
      (s.reps.capt.popn > 10M) AND (s.age > 50) AND
      NOT (s.reps.capt.sistercity.name == "Providence")
```

(a)

```
SELECT s1
FROM s1 IN Sens, s2 IN Sts, c1 IN Caps, c2 IN Cits, c3 IN Cits
WHERE (c3.popn > 1M) AND NOT (c3.popn > 2M) AND NOT (c2.name == "Providence") AND
      (c2.popn > 1M) AND (c1.popn > 1M) AND (c1.sistercity == c2.OID) AND
      (s2.capt == c1.OID) AND (s1.pty == "GOP") AND (s1.reps == s2.OID) AND
      (s1.age > 50) AND (s1.bornin == c3.OID)
```

(b)

Figure 1: A Query Lite Query (a) And Its Translation Into SQL (b)

Our experience provides insight into how query rewriting can be used to translate between query languages and data models. Translation is similar to normalization — rewriting a query into a form that an existing software component (for our work, the query processor for DB2 and not just a plan generator) is better prepared to accept. This kind of translation is becoming increasingly important as many commercial systems attempt to provide additional functionality in their query facilities while building upon existing components. The most common example of this is the addition of object-oriented capabilities to relational DBMS (producing hybrid object-relational systems). Typically, the goal of such extensions is to reuse the relational query optimizer as much as possible, both to save in development costs and to exploit the maturity of relational optimizer technology. Other examples include:

- Object-Oriented databases that attempt to provide support for OQL: the emerging query language standard for OODB's. Many OODB's (e.g., ObjectStore [12]) were developed before OQL was defined, and now are trying to reuse their existing query processor components to process OQL queries

- Databases that attempt to provide support for foreign data models (e.g., XML [16]). XML exhibits a hierarchical data model. If collections of XML documents are stored as fragments in a database, then queries on XML documents will have to be translated into queries on the underlying repository.

Translation of data models and queries from one form to another has been considered in the context of heterogenous multidatabase systems for many years [9, 13]. Solutions typically involve *wrappers* [17]. A wrapper provides an interface that the global layers of the multidatabase expects and translates this interface onto the interface that the actual database can support. In our case, the external interface is an object-based model supporting an object query language, and the internal interface is the relational model (i.e., DB2) supporting SQL.

The work reported here can be thought of as experiences using query rewriting technology to build wrappers. The novel part of this technique is that it is based on a formalism that has been shown in previous work to be well-suited to verification. COKO-KOLA uses formal methods to support the development of query rewriters whose correctness can be verified with the theorem prover LP [8].[1] Therefore, this work not only gave us experience with translation, but also provided a testbed within which we could assess the usefulness of the COKO-KOLA framework for expressing "real" query rewrites.

[1] A query rewriter is *correct* if it preserves the meaning of the queries it transforms.

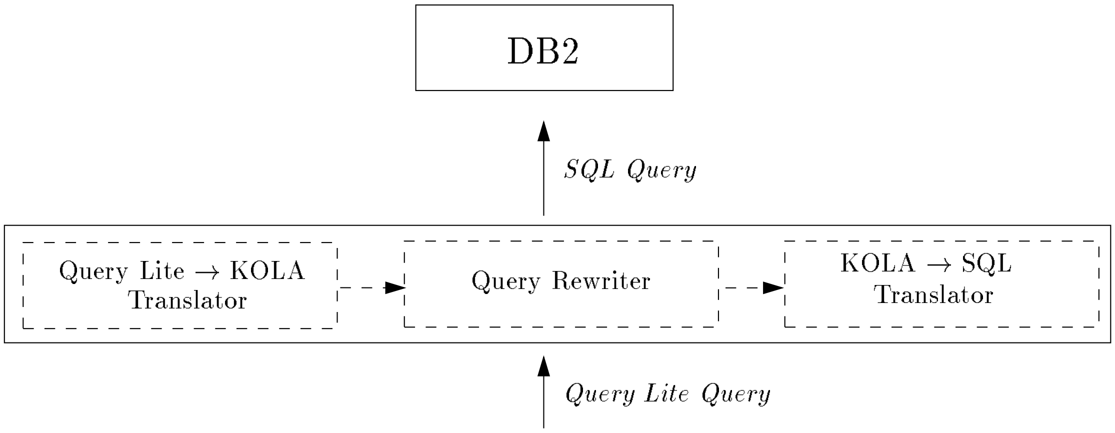

Figure 2: An Architecture for the Query Lite Query Rewriter

The paper begins with an introduction to COKO-KOLA followed by background on the San Francisco project [15] and Query Lite in Section 2. Sections 3 and 5 describe additions to the COKO-KOLA implementation required to complete this project, including translators to translate Query Lite queries (and more generally, OQL's set-and-bag queries) into KOLA (Section 3), and select KOLA queries into SQL (Section 5). Section 4 describes the COKO routines (*transformations*) developed for our translating query rewriter. These include a normalization transformation described in Section 4.1 whose purpose is to normalize KOLA queries resulting from translation (from Query Lite or OQL), and a transformation (presented in Section 4.2) to rewrite these normalized queries into a form that can be translated into SQL. Section 6 considers feasibility issues in light of these transformations. Section 7 summarizes the paper.

2 Background

The architecture for our translating query rewriter is illustrated in Figure 2. San Francisco queries (defined in a subset of OQL informally called "Query Lite") get translated into SQL queries in three stages. First, Query Lite queries are translated into KOLA: our *combinator-based* (i.e., variable-free) query representation [3]. Next, KOLA queries are rewritten by transformations expressed in COKO [4]: our specialized language for expressing complex query rewrites. Finally, the queries produced by our rewriter are translated into SQL. Alternatively, we could have translated Query Lite queries directly into SQL rather than translating them into and out of KOLA. But we chose to use KOLA as an intermediate query representation for reasons of:

- *Extensibility:* If we later decided to process OQL queries and not just Query Lite queries, then our architecture would allow us simply to replace the Query Lite → KOLA translator with an OQL → KOLA translator and reuse the other components of Figure 2. Similarly, if we later decided to process Query Lite queries over a different backbone than DB2, this would require us only to replace the KOLA → SQL translator and not every component illustrated in Figure 2.

- *Correctness:* Query rewriting is an error-prone process [11]. COKO-KOLA defines a framework by which query rewrites can be verified with an automated theorem prover.

In this section, we provide background on COKO-KOLA and San Francisco required for this paper.

2.1 COKO-KOLA

COKO-KOLA is a two-tiered framework for the building of correct and efficient rule-based query rewriters. At its foundation lies KOLA [3] : a combinator-based (i.e., variable-free) algebra of functions and predicates.

Basic query rewrites can be expressed with declarative (i.e., code-free) rewrite rules. Complex query rewrites get expressed in the programming language, COKO [4]. And semantic query rewrites get expressed in a semantic extension of this framework [5]. All rewrites expressed in this framework can be verified with the theorem prover LP [8].

For the examples appearing in this paper, we assume an object database schema modeling the political structure of the United States. This defines collections: `Sens` (with objects of type *Senator*), `Mays` (with objects of type *Mayor*), `Sts` (with objects of type *State*), and `Caps` and `Cits` (with objects of type *City* such that `Caps` is the subcollection of `Cits` containing state capitals). A Senator has a name (`name`: *String*), a state that he/she represents (`reps`: *State*), a party (`pty`: *String*), a city where he/she was born (`bornin`: *City*), a number of terms served (`terms`: *Int*) and an age (`age`: *Int*). A mayor has a name (`name`: *String*), a city that he/she serves (`city`: *City*) and a city where he/she was born (`bornin`: *City*). A state has a name (`name`: *String*), a capital city (`capt`: *City*) and a population (`popn`: *Int*). A city has a name (`name`: *String*), a sister city (`sistercity`: *City*), a mayor (`mayor`: *Mayor*) and a population (`popn`: *Int*).

KOLA: KOLA functions and predicates are explicitly invoked ($f \ ! \ x$ and $p \ ? \ x$ denote the respective invocations of function f and predicate p on some object x) and are either *primitives* with a predefined semantics, or *formed* from simpler functions and predicates.

In Tables 1 and 2, we show the operational semantics of the KOLA primitives and formers respectively that appear in this paper. The semantics of these operators are defined in terms of arbitrary objects x, y and z, integers or floats i and j, boolean b, functions f and g, predicates p and q, and bags A and B. We assume a bag notation that resembles set comprehensions. The expression, $x^i \in A$ indicates that there are i copies of x in bag A (such that $i > 0$). The expression,

$$\{\!|(f \ ! \ x)^i \mid x^i \in A, \ p \ ? \ x|\!\}$$

denotes the bag consisting of i copies of $f \ ! \ x$ for every object x that satisfies p and that has i copies in A (again such that $i > 0$).

The KOLA primitives presented in Table 1 include the function primitives: **id** (identity), π_1 and π_2 (projection), **shl** and **shr** (shifting), **singleton** (singleton constructor), database-specific attributes (e.g., `terms`) and **add**, **sub**, **mul** and **div** (arithmetic operators). Also listed in Table 1 are predicate primitives: **eq** (equality), **neq** (inequality) and **lt**, **leq**, **gt**, **geq** (order predicates). The formers of Table 2 include function formers: $\circ$ (composition), $\langle \ \rangle$ (pairing), $\mathtt{K}_f$ (constant function), and $\mathtt{C}_f$ (curried function). Also listed in Table 2 are predicate formers: $\oplus$ (combination), `&` (conjunction), $|$ (disjunction), $\sim$ (negation), $\mathtt{K}_p$ (constant predicate), and $\mathtt{C}_p$ (curried predicate), and query function formers: **iterate** (which corresponds to SQL's and OQL's "`SELECT FROM WHERE`" construct), **unnest** (unnesting) and **join** (the join former).

The reduction of a KOLA query follows from the definitions of Tables 1 and 2. For example, the KOLA query of Figure 3c reduces to the expression,

$$\{\!|(\mathbf{id} \ ! \ s)^i \mid s^i \in \texttt{Sens}, \ (\mathbf{gt} \oplus \langle \mathbf{id}, \mathtt{K}_f \ (5)\rangle \oplus \texttt{terms}) \ ? \ s|\!\}$$

by the definition of **iterate**. By the definition of **id**, ($\mathbf{id} \ ! \ s$) reduces to "s". (($\mathbf{gt} \oplus \langle \mathbf{id}, \mathtt{K}_f \ (5)\rangle \oplus \texttt{terms}) \ ? \ s$) reduces to "$s.\texttt{terms} \ > \ 5$" as shown below:

$$\begin{array}{lcll}
(\mathbf{gt} \oplus \langle \mathbf{id}, \mathtt{K}_f \ (5)\rangle \oplus \texttt{terms}) \ ? \ s & \stackrel{1}{\to} & (\mathbf{gt} \oplus \langle \mathbf{id}, \mathtt{K}_f \ (5)\rangle) \ ? \ (\texttt{terms} \ ! \ s) & \text{(By the def. of } \oplus) \\
 & \stackrel{2}{\to} & (\mathbf{gt} \oplus \langle \mathbf{id}, \mathtt{K}_f \ (5)\rangle) \ ? \ (s.\texttt{terms}) & \text{(By the def. of } \texttt{terms} \text{ (attributes))} \\
 & \stackrel{3}{\to} & \mathbf{gt} \ ? \ [\mathbf{id} \ ! \ (s.\texttt{terms}), \mathtt{K}_f \ (5) \ ! \ (s.\texttt{terms})] & \text{(By the defs. of } \oplus \text{ and } \langle _, _\rangle) \\
 & \stackrel{4}{\to} & \mathbf{gt} \ ? \ [s.\texttt{terms}, 5] & \text{(By the defs. of } \mathbf{id} \text{ and } \mathtt{K}_f) \\
 & \stackrel{5}{\to} & s.\texttt{terms} \ > \ 5 & \text{(By the def. of } \mathbf{gt})
\end{array}$$

Functions			
identity	$\textbf{id} \; ! \; x$	$=$	x
projection (1)	$\pi_1 \; ! \; [x,\, y]$	$=$	x
projection (2)	$\pi_2 \; ! \; [x,\, y]$	$=$	y
shift left	$\textbf{shl} \; ! \; [x,\, [y,\, z]]$	$=$	$[[x,\, y],\, z]$
shift right	$\textbf{shr} \; ! \; [[x,\, y],\, z]$	$=$	$[x,\, [y,\, z]]$
singleton	$\textbf{singleton} \; ! \; x$	$=$	$\{\!\mid x \mid\!\}$
attributes	$<\texttt{att}> \; ! \; x$	$=$	$x.\texttt{att}$
absolute value	$\textbf{abs} \; ! \; x$	$=$	$\|x\|$
addition	$\textbf{add} \; ! \; [i,\, j]$	$=$	$i + j$ (also **sub**, **mul**, **div** and **mod**)
Predicates			
equality	$\textbf{eq} \; ? \; [x,\, y]$	$=$	$x == y$ (also **neq**)
less than	$\textbf{lt} \; ? \; [x,\, y]$	$=$	$x < y$ (also **leq**, **gt**, **geq**)
null comparison (1)	$\textbf{isnull} \; ? \; x$	$=$	x IS NULL
null comparison (2)	$\textbf{isnotnull} \; ? \; x$	$=$	x IS NOT NULL

Table 1: KOLA Primitives

Functions			
composition	$(f \circ g) \; ! \; x$	$=$	$f \; ! \; (g \; ! \; x)$
pairing	$\langle f,\, g\rangle \; ! \; x$	$=$	$[f \; ! \; x,\, g \; ! \; x]$
constant function	$\texttt{K}_f \; (x) \; ! \; y$	$=$	x
curried function	$\texttt{C}_f \; (f,\, x) \; ! \; y$	$=$	$f \; ! \; [x,\, y]$
Predicates			
combination	$(p \oplus f) \; ? \; x$	$=$	$p \; ? \; (f \; ! \; x)$
conjunction	$(p \;\&\; q) \; ? \; x$	$=$	$(p \; ? \; x) \wedge (q \; ? \; x)$
disjunction	$(p \mid q) \; ? \; x$	$=$	$(p \; ? \; x) \vee (q \; ? \; x)$
negation	$\sim (p) \; ? \; x$	$=$	$\neg \, (p \; ? \; x)$
constant predicate	$\texttt{K}_p \; (b) \; ? \; x$	$=$	b
curried predicate	$\texttt{C}_p \; (p,\, x) \; ? \; y$	$=$	$p \; ? \; [x,\, y]$
Query Functions			
iteration	$\textbf{iterate} \; (p,\, f) \; ! \; A$	$=$	$\{\!\mid (f \; ! \; x)^i \mid x^i \in A,\, p \; ? \; x \mid\!\}$
unnest	$\textbf{unnest} \; (f,\, g) \; ! \; A$	$=$	$\{\!\mid (f \; ! \; [x,\, y])^{ij} \mid x^i \in A,\, y^j \in (g \; ! \; x) \mid\!\}$
join	$\textbf{join} \; (p,\, f) \; ! \; [A,\, B]$	$=$	$\{\!\mid (f \; ! \; [x,\, y])^{ij} \mid x^i \in A,\, y^j \in B,\, p \; ? \; [x,\, y] \mid\!\}$

Table 2: KOLA Formers

Therefore, the entire query reduces to $\{\!\mid s^i \mid s^i \in \texttt{Sens},\, s.\texttt{terms} \; > \; 5 \mid\!\}$ which is also the semantics of the OQL query of Figure 3a.

KOLA's combinator-style makes basic rewrite rules expressible without code. This males it possible to verify KOLA rules with a theorem prover, as we have done with several hundred rules using LP [8].

COKO: Rewrite rules are inherently simple, and cannot capture complex query rewrites that require multiple rule firings, perhaps over several subexpressions of a query and perhaps over several passes. For this reason, we developed the programming language COKO [4] for expressing complex query rewrites in terms of sets of KOLA rewrite rules supplemented with *firing algorithms* that control their firing. Only rewrite rules can transform queries. Therefore, COKO transformations can be verified with LP just like the KOLA rules they fire.

To illustrate COKO, Figure 4 shows a transformation (`CNF`) that rewrites KOLA predicates into conjunc-

```
SELECT x
FROM x IN Sens
WHERE x.terms > 5
```
a. A Query Lite Query

$(\mathbf{iterate}\ (\mathbf{gt} \oplus \langle \mathtt{terms} \circ \pi_2,\ \mathtt{K}_f\ (5)\rangle,\ \pi_2) \circ$
$\mathbf{unnest}\ (\mathbf{id},\ \mathtt{K}_f\ (\mathtt{Sens})) \circ$
$\mathbf{singleton})\ !\ \mathtt{NULL}$

b. Translation Into KOLA

$\mathbf{iterate}\ (\mathbf{gt} \oplus \langle \mathbf{id},\ \mathtt{K}_f\ (5)\rangle \oplus \mathtt{terms},\ \mathbf{id})\ !\ \mathtt{Sens}$

c. After Normalization

Figure 3: A Query Lite Query (a), its Translation (b) and its Normalization (c)

```
TRANSFORMATION CNF
 USES
  dist1: (p & q) | r  ⇉=  (p | r) & (q | r)
  dist2: r | (p & q)  ⇉=  (p | r) & (q | r)
 BEGIN
  BU CNFAux
 END
```

```
TRANSFORMATION CNFAux
 USES
  dist1: (p & q) | r  ⇉=  (p | r) & (q | r)
  dist2: r | (p & q)  ⇉=  (p | r) & (q | r)
 BEGIN
  {dist1 || dist2} →
    GIVEN p1 & p2 DO {CNFAux (p1); CNFAux (p2)}
 END
```

Figure 4: A CNF Transformation Expressed in COKO

tive normal form (CNF).[2] The firing algorithm for this transformation is delimited by keywords `BEGIN` and `END`. All statements that appear in a firing algorithm have an associated *success value* (i.e., *true* or *false*) as a result of execution. For example, a rewrite rule that gets fired has a success value of *true* if the left-hand side of the rule matched the expression on which it was fired. Complex statements also have success values. For example, for any two statements S and S', "$S \mid\mid S'$" succeeds if either S or S' succeed. Success values control the execution flow of a COKO firing algorithm. For example, "$S \mid\mid S'$" only executes S' if S fails. For example, "$S \rightarrow S'$" only executes S' if S succeeds.

The firing algorithm for `CNF` indicates that a bottom-up pass is performed over a KOLA predicate expression (i.e., its *parse tree*) firing the auxiliary transformation, `CNFAux` on each visited subtree. The firing algorithm for `CNFAux` begins with the statement "`{dist1 || dist2}`", which instructs the transformation to fire KOLA rules `dist1` and `dist2` (if `dist1` fails) on a given KOLA predicate. The "→" following this statement indicates that if it *succeeds* (i.e., if either `dist1` or `dist2` successfully fire on the predicate), the statement that follows (`GIVEN p1 & p2 DO {CNFAux (p1); CNFAux (p2)}`) is to be executed. The `GIVEN` statement first *unifies* pattern `p1 & p2` with the predicate that resulted from firing `dist1` and `dist2`, and then fires `CNFAux` recursively with the subpredicates bound to `p1` and `p2` as a result of unification. The effect of this is to make `CNFAux` fire `dist1` and `dist2` on all subtrees visited in a top-down pass of a predicate until some subtree is visited on which both rules fail to fire. A proof showing that this results in a predicate that is in CNF is given in [2].

[2] Analagous to standard CNF definitions for boolean expressions, a KOLA predicate is in CNF if it consists of conjunctions (`&`) of disjunctions (`|`) of possibly negated (∼) literals, such that a literal is a KOLA predicate with no subpredicate that is a conjunction, disjunction or negation.

$$\texttt{is\_injective (f), is\_set (A) ::}$$
$$\texttt{set ! (iterate (p, f) ! A)} \stackrel{\rightarrow}{=} \texttt{iterate (p, f) ! A}$$
(a)

$$\texttt{is\_key (f)} \implies \texttt{is\_injective (f)} \qquad (1)$$
$$\texttt{is\_injective (f)} \wedge \texttt{is\_injective (g)} \implies \texttt{is\_injective (f} \circ \texttt{g)} \qquad (2)$$
(b)

Figure 5: A Rewrite Rule Conditioned on Injectivity (a) and Rules for Inferring Injectivity (b)

Semantic Extensions to COKO-KOLA: The KOLA rules fired by COKO transformations can be *conditional*, as in the conditional rewrite rule of Figure 5a. This rule rewrites a query that performs duplicate elimination (**set**) into a query that performs no duplicate elimination. This rule is conditioned on function f being injective (is_injective (f)) and collection A being a set (is_set (A)). When fired, conditional rules use inference rules such as those of Figure 5b to determine if these conditions hold. For example, the query "**set** ! (**iterate** (K_p (true), capt ∘ reps) ! Sens)" which returns the capital cities of all states represented by senators would be rewritten to "**iterate** (K_p (true), capt ∘ reps) ! Sens" (performing no duplicate elimination), as "capt ∘ reps" would be recognized as an injective function by inference rules (1) and (2) of Figure 5b, provided that the schema identified capt and reps each as keys.

2.2 San Francisco and Query Lite

The object model for San Francisco is similar to the ODMG object model [1]. The relational implementation of objects is straightforward and similar to schemes described elsewhere (e.g., [10]). Any collection of objects of a particular type (including a type extent) is represented by a relation. The structure of this relation is derived from the interface of the associated object type in the following way:

- A column (which for simplicity we will name OID) is dedicated to denote each object's unique identifier. Such identifiers are typically strings.
- Columns are reserved for each attribute-based method (except those that are collection-valued.) Atribute-based methods that return values of some basic type (e.g., integers) are represented by columns with values of that type. Attribute-based methods that return other objects are represented by columns whose values are strings denoting object identifiers.

To illustrate, collection Sts of *State* objects would be represented within this implementation with a relation (Sts) with columns: name of type *String*, capt of type *String* (OID), and popn of type *Integer*.

Presently, San Francisco is queried using a limited subset of OQL informally known as "Query Lite". Query Lite restricts queries to those of the form,

SELECT x FROM x IN A [WHERE *BoolExp*] [ORDER BY *OrderExp*]

such that *BoolExp* denotes an optional boolean expression consisting of conjunctions, disjunctions and negations of simple comparison expressions, and *OrderExp* denotes an optional ordering expression that is either the name of an externally defined comparison function or a list of unary methods defined on the objects in A. Like OQL, Query Lite queries can contain invocations of methods with multiple arguments and path expressions (although only in their WHERE clause). Query Lite supports neither join queries nor embedded collections.

$$
\begin{array}{l}
(\textbf{iterate}\ (p,\ f) \circ \\
\ \ \textbf{unnest}\ (\textbf{id},\ f_n) \circ \ldots \circ \textbf{unnest}\ (\textbf{id},\ f_0) \circ \\
\ \ \textbf{singleton})
\end{array}
$$

$$\textit{such that for all } 0 \leq i \leq n,\ f_i = \mathbf{T}\ [\![E_i]\!]$$

Figure 6: KOLA Functions Returned As a Result of Translating OQL `SELECT-FROM-WHERE` Queries

3 Translating Object Queries into KOLA

Our approach to translation is similar to how Query Lite or OQL queries would be given a denotational semantics. Because arbitrary query expressions can have occurences of free variables, (e.g., x in "x `> 5`") the semantics of such expressions is dependent on an *environment* that defines what is referred to by the free variables appearing within the expression. More precisely, an *environment* is a list of (variable, value) pairs $((v_1, d_1), (v_2, d_2), \ldots, (v_n, d_n))$ providing the bindings of all free variables, $v_1, \ldots, v_n$ appearing in a given expression. (If an environment ρ gives bindings to every free variable in an expression, e, we say that e is *well-formed* wrt to ρ.) The denotational semantics of a Query Lite or OQL expression, e is a function ($\mathbf{Eval}\ [\![e]\!]$) that maps all environments for which it is well-formed to values. For example, $\mathbf{Eval}\ [\![x > 5]\!]\ \rho$ is *true* if $\rho = ((x, 6))$ but *false* if $\rho = ((x, 4))$. Queries that do not appear as subexpressions of other queries will have no free variables, and therefore are well-formed wrt the empty environment, (). The denotational semantics of these expressions are constant functions.

The idea behind translation is to map every expression e to a KOLA function $\mathbf{T}\ [\![e]\!]$ whose operational semantics coincides with e's denotational semantics. More precisely, if e is well-formed with respect to some environment ρ, then $\mathbf{T}\ [\![e]\!]$ is a KOLA function satisfying the constraint that

$$\mathbf{T}\ [\![e]\!]\ !\ (\overline{\rho}) \quad = \quad \mathbf{Eval}\ [\![e]\!]\ \rho$$

such that $\overline{\rho}$ is the KOLA (nested) pair equivalent of ρ.[3] Queries that are well-formed wrt the empty environment get mapped to constant KOLA functions. The choice of `NULL` as an argument to these functions can be thought of as reflecting how KOLA represents the empty environment. (i.e., $\overline{()}$ = `NULL`).

The Query Lite $\rightarrow$ KOLA translation function ($\mathbf{T}$) is defined in Table 3. For any Query Lite expression e, $\mathbf{T}\ [\![e]\!]$ is its KOLA translation. If e is well-formed wrt some environment ρ, then ($\mathbf{T}\ [\![e]\!]\ !\ \overline{\rho}$) reduces by the definitions in Tables 1 and 2 to an expression that is equivalent to e wrt ρ.[4] Translation maps constants into constant predicates or functions; attribute names and other unary functions and predicates into primitive functions and predicates; path expressions into compositions of primitive functions; and conjunctions, disjunctions and negations into KOLA conjunction, disjunction and negation predicates. Translation also maps binary functions into compositions of the form ($f \circ \langle g_1,\ g_2 \rangle$), and binary predicates into combinations of the form ($p \oplus \langle g_1,\ g_2 \rangle$), such that f and p are the KOLA primitive equivalents of the binary functions and predicates, and g_1 and g_2 are the KOLA translations of the arguments to the binary function or predicate.

Translation of Query Lite `SELECT − FROM − WHERE` queries produces somewhat unintuitive looking functions that get invoked on the constant `NULL`. These functions are compositions of functions formed by **iterate**, **unnest** and **singleton**. (To illustrate, the translation of the Query Lite query of Figure 3a is shown in Figure 3b). Instead, Query Lite queries could have been translated into more intuitive **iterate** queries (e.g., the KOLA query of Figure 3c). However, the latter translation strategy (which must distinguish between subexpressions of the Query Lite query that correspond to KOLA functions and subexpressions that correspomd

[3] If ρ is $((v_1, d_1), \ldots, (v_n, d_n))$, then $\overline{\rho}$ is `[... [NULL,` d_1`],` $\ldots d_n$`]`.
[4] This is proved formally with respect to an OQL $\rightarrow$ KOLA translation function that subsumes $\mathbf{T}$ in [2].

Constants

$$
\begin{aligned}
\mathbf{T}\,[\![c]\!] &= \mathtt{K}_f\ (c)\ (c \text{ a non-Bool constant/global name})\\
\mathbf{T}\,[\![b]\!] &= \mathtt{K}_p\ (b)\ (b \text{ equal to } \mathtt{TRUE} \text{ or } \mathtt{FALSE})
\end{aligned}
$$

Path Expressions

$$
\begin{aligned}
\mathbf{T}\,[\![Exp\,.\,\mathtt{IDENT}]\!] &= \mathtt{IDENT} \circ \mathbf{T}\,[\![Exp]\!]\\
\mathbf{T}\,[\![Exp\,.\,\mathtt{IDENT}\ (Exp_1,\ Exp_2,\ \ldots,\ Exp_n)]\!] &= \\
\mathtt{IDENT} \circ \langle \mathbf{T}\,[\![Exp]\!],\ &\langle\langle \ldots \langle \mathbf{T}\,[\![Exp_1]\!],\ \mathbf{T}\,[\![Exp_2]\!]\rangle,\ \ldots\rangle,\ \mathbf{T}\,[\![Exp_n]\!]\rangle\rangle
\end{aligned}
$$

Basic Expressions

$$
\begin{aligned}
\mathbf{T}\,[\![-\ (Exp)]\!] &= \mathtt{C}_f\ (\mathbf{mul},\ \mathtt{-1}) \circ \mathbf{T}\,[\![Exp]\!]\\
\mathbf{T}\,[\![\mathtt{ABS}\ (Exp)]\!] &= \mathbf{abs} \circ \mathbf{T}\,[\![Exp]\!]\\
\mathbf{T}\,[\![Exp_0\ \mathtt{+}\ Exp_1]\!] &= \mathbf{add} \circ \langle \mathbf{T}\,[\![Exp_0]\!],\ \mathbf{T}\,[\![Exp_1]\!]\rangle\\
& \quad \text{similarly for } \mathtt{-}\ (\mathbf{sub}),\ \mathtt{*}\ (\mathbf{mul}) \text{ and } \mathtt{/}\ (\mathbf{div})\\
\\
\mathbf{T}\,[\![Exp\ \mathtt{IS\ NULL}]\!] &= \mathbf{isnull} \oplus \mathbf{T}\,[\![Exp]\!]\\
\mathbf{T}\,[\![Exp\ \mathtt{IS\ NOT\ NULL}]\!] &= \mathbf{isnotnull} \oplus \mathbf{T}\,[\![Exp]\!]\\
\mathbf{T}\,[\![Exp_0\ \mathtt{==}\ Exp_1]\!] &= \mathbf{eq} \oplus \langle \mathbf{T}\,[\![Exp_0]\!],\ \mathbf{T}\,[\![Exp_1]\!]\rangle\\
& \quad \text{similarly for } \mathtt{!=}\ (\mathbf{neq})\\
\\
\mathbf{T}\,[\![Exp_0\ \mathtt{<}\ Exp_1]\!] &= \mathbf{lt} \oplus \langle \mathbf{T}\,[\![Exp_0]\!],\ \mathbf{T}\,[\![Exp_1]\!]\rangle\\
& \quad \text{similarly for } >\ (\mathbf{gt}),\ <=\ (\mathbf{leq}) \text{ and } >=\ (\mathbf{geq})\\
\\
\mathbf{T}\,[\![\mathtt{NOT}\ (BoolExp)]\!] &= \sim (\mathbf{T}\,[\![BoolExp]\!])\\
\mathbf{T}\,[\![BoolExp_0\ \mathtt{AND}\ BoolExp_1]\!] &= \mathbf{T}\,[\![BoolExp_0]\!]\ \mathtt{\&}\ \mathbf{T}\,[\![BoolExp_1]\!]\\
\mathbf{T}\,[\![BoolExp_0\ \mathtt{OR}\ BoolExp_1]\!] &= \mathbf{T}\,[\![BoolExp_0]\!]\ |\ \mathbf{T}\,[\![BoolExp_1]\!]
\end{aligned}
$$

Query Expressions

$$
\mathbf{T}\left[\!\!\left[\begin{array}{l}\mathtt{SELECT}\ x\\ \mathtt{FROM}\ x\ \mathtt{IN}\ \mathtt{IDENT}\\ \mathtt{WHERE}\ BoolExp\end{array}\right]\!\!\right] = \begin{array}{l}\mathbf{iterate}\ (\mathbf{T}\,[\![BoolExp]\!],\ \pi_2) \circ\\ \ \mathbf{unnest}\ (\mathbf{id},\ \mathtt{K}_f\ (\mathtt{IDENT})) \circ\\ \ \ \mathbf{singleton}\end{array}
$$

Table 3: **T**: The Query Lite → KOLA Translation Function

to KOLA objects) does not generalize well to OQL, where expressions appearing in `FROM`-clauses can be path expressions (which correspond to KOLA functions) and not just named collections (which correspond to KOLA objects). On the other hand, the translation strategy we chose generalizes well, as OQL queries of the form,

$$
\begin{array}{l}
\mathtt{SELECT}\ f(x_0, \ldots, x_n)\\
\mathtt{FROM}\ x_0\ \mathtt{IN}\ E_0, \ldots, x_n\ \mathtt{IN}\ E_n\\
\mathtt{WHERE}\ p(x_0, \ldots, x_n)
\end{array}
$$

get mapped to KOLA functions of the form shown in Figure 6. To simplify the eventual migration from Query Lite to OQL, we map all Query Lite expressions to KOLA functions or predicates. The results of translation (e.g., Figure 3b) are unintuitive but correct as can be seen by reducing these queries according to the definitions of Tables 1 and 2. In Section 4.1 we describe COKO transformations that rewrite these queries into their more intuitive forms (e.g., Figure 3c).

4 Query Rewriting

The second component of our Query Lite → SQL translator rewrites KOLA queries generated by translation from Query Lite, into KOLA queries that perform joins. The COKO transformations that perform this rewriting operate in two stages:

1. *Normalization:* First, the unintuitive results of translation (that invoke functions or predicates on NULL) are normalized into more intuitive forms (i.e., query functions that get invoked on collections).

2. *Conversion into relational queries:* Next, queries that include path expressions (in KOLA, compositions of attribute primitives) get converted into join queries.

In this section, we briefly describe the COKO transformations that perform normalization, and describe more throughly the COKO transformations that convert path expressions into joins.

4.1 Normalization

The purpose of the COKO transformations described in this section is to rewrite KOLA query functions of the form shown in Figure 6 into more intuitive forms. Specifically, the purpose of this transformation is to rewrite query functions that get invoked on NULL into query functions that get invoked on a collection. A full description of these transformations can be found in [2]. Note that this normalization can be fired on translations of OQL queries, and not just translations of Query Lite queries.

When invoked on NULL, the query function of Figure 6 returns:

$$\{\!|(f \;\; ! \;\; \vec{x})^{i_0,\ldots,i_n} \mid p \;\; ? \;\; \vec{x},\; q(\vec{x})|\!\}$$

such that

$$\vec{x} = \texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots x_{n-1}\texttt{]},\; x_n\texttt{]}$$

and

$$q(\vec{x}) = \bigwedge_{0 \leq j \leq n} x_j^{i_j} \in (f_j \;\; ! \;\; \texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots x_{j-2}\texttt{]},\; x_{j-1}\texttt{]}).$$

Put another way, this query invokes f on every "tuple" ($\texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots x_{n-1}\texttt{]},\; x_n\texttt{]}$) that satisfies p in the cartesian product of collections generated by functions $f_0, \ldots, f_n$.[5]

Every function f_i "ignores" (i.e., does not extract) the NULL that appears in its argument. Because f_0 is invoked on NULL and returns a collection, it must be of the form $\texttt{K}_f\ (A)$ for some collection A. For each function f_i such that $i > 0$, f_i ignores the NULL that appears in its argument, and therefore f_i can be rewritten into a function of the form,

$$f_i' \circ \pi_2 \circ \mathbf{shr}^{i-1}$$

for some function f_i'.[6] Invoked on a "tuple", this function reduces as shown below:

$$\begin{array}{cl}
\multicolumn{2}{l}{(f_i' \circ \pi_2 \circ \mathbf{shr}^{i-1}) \;\; ! \;\; \texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots x_{i-2}\texttt{]},\; x_{i-1}\texttt{]}} \\
= & f_i' \;\; ! \;\; (\pi_2 \;\; ! \;\; (\mathbf{shr}^{i-1} \;\; ! \;\; \texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots x_{i-2}\texttt{]},\; x_{i-1}\texttt{]})) \\
= & f_i' \;\; ! \;\; (\pi_2 \;\; ! \;\; (\underbrace{\mathbf{shr} \;\; ! \;\; (\mathbf{shr} \;\; ! \;\; (\ldots \;\; ! \;\; (\mathbf{shr}}_{i-1} \;\; ! \;\; \texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots x_{i-2}\texttt{]},\; x_{i-1}\texttt{]})))) \\
= & f_i' \;\; ! \;\; (\pi_2 \;\; ! \;\; (\underbrace{\mathbf{shr} \;\; ! \;\; (\mathbf{shr} \;\; ! \;\; (\ldots \;\; ! \;\; (\mathbf{shr}}_{i-2} \;\; ! \;\; \texttt{[[}\ldots\texttt{[NULL, } x_0\texttt{]}, \ldots\texttt{]},\; \texttt{[}x_{i-2},\; x_{i-1}\texttt{]]})))) \\
= & \ldots \\
= & f_i' \;\; ! \;\; (\pi_2 \;\; ! \;\; \texttt{[NULL, [}x_0,\; \texttt{[}\ldots,\; \texttt{[}x_{i-2},\; x_{i-1}\texttt{]]]]}) \\
= & f_i' \;\; ! \;\; (\texttt{[}x_0,\; \texttt{[}\ldots,\; \texttt{[}x_{i-2},\; x_{i-1}\texttt{]]]})
\end{array}$$

[5] Strictly speaking, this is not a cartesian product as a collection generated by some function f_i may vary according to elements $x_0, \ldots, x_{i-1}$.

[6] For any function g, "g^j" is shorthand for $\underbrace{g \circ \ldots \circ g}_{j}$ and $g^0 = \mathbf{id}$.

$$
\begin{array}{l}
(\textbf{iterate}\ (\overline{p},\ \overline{f}) \circ \\
\quad \textbf{unnest}\ (\textbf{id},\ \overline{f_n}) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_{n-1}}) \circ \ \ldots\ \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_1}) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_0}) \circ \\
\quad \textbf{singleton})\ !\ \texttt{NULL}
\end{array}
$$

such that

$$
\begin{array}{rcl}
\overline{p} & = & p' \oplus (\pi_2 \circ \textbf{shr}^n) \\
\overline{f} & = & f' \circ \pi_2 \circ \textbf{shr}^n \\
\overline{f_i} & = & f_i' \circ \pi_2 \circ \textbf{shr}^{i-1} \ \ \text{(for } i > 0\text{), and} \\
\overline{f_0} & = & \texttt{K}_f\ (A)
\end{array}
$$

Figure 7: Translated OQL Queries Following Rewriting of their Data Functions

The normalization that rewrites the query of Figure 6 first rewrites predicate p and each of the functions $f, f_n, \ldots, f_0$ in the manner described above, leaving a query of the form shown in Figure 7, for some functions f' and f_i' $(1 \leq i \leq n)$, predicate p' and collection A.

The rest of the normalization proceeds in five steps and uses the rules of Figure 8. In this figure and in the text, we use the notation "δ_i" to denote a KOLA function defined for $i \geq 0$ as follows:

$$
\begin{array}{rcl}
\delta_0 & = & \textbf{shr} \\
\delta_i & = & \langle \pi_1,\ \delta_{i-1} \circ \pi_2 \rangle, \quad \text{for } i > 0
\end{array}
$$

Substituting for δ in rules (11) and (12) of Figure 8 reveals these rules to be strictly unnecessary: rule (11) follows from rule (7), and rule (12) follows from rule (10) (fired n times). In fact the COKO implementations of these transformations cannot express rules (11) and (12) (and instead would fire rules (7) and (10) as many times as necessary) because the unification of these rules to queries would require counting and not just pattern matching. (Note however that we can fire rule (2) once we substitute $\langle \pi_1,\ \textbf{shr} \circ \pi_2 \rangle$ for δ_1.) We include these "macro rules" here to simplify the description of this transformation.

The normalization firing algorithm proceeds by visiting each subfunction of the form, $f \circ g$ of the query shown in Figure 7. Therefore, the first subfunction visited is **iterate** $(p,\ f) \circ$ **unnest** $(\textbf{id},\ f_n)$, the second is **unnest** $(\textbf{id},\ f_n) \circ$ **unnest** $(\textbf{id},\ f_{n-1})$ (modulo changes made to **unnest** $(\textbf{id},\ f_n)$ after visiting the first subfunction) and so on. These visits are described below.

Step 1: Substituting for p and f, the first visited subfunction is

$$\textbf{iterate}\ (p' \oplus (\pi_2 \circ \textbf{shr}^n),\ f' \circ \pi_2 \circ \textbf{shr}^n) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_n}).$$

Rule (1) of Figure 8 fires on this function $n+1$ times, and rule (10) fires once, leaving a query of the form,

$$
\begin{array}{l}
(\textbf{iterate}\ (p',\ f') \circ \\
\quad \textbf{unnest}\ (\pi_2 \circ \textbf{shr}^n,\ \overline{f_n}) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_{n-1}}) \circ \ \ldots\ \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_1}) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_0}) \circ \\
\quad \textbf{singleton})\ !\ \texttt{NULL}
\end{array}
$$

such that p', f' and $\overline{f_i}$ are as defined in Figure 7.

Step 2: In this step, each pair of adjacent **unnest** subfunctions in the composition chain,

$$\textbf{unnest}\ (\pi_2 \circ \textbf{shr}^n,\ \overline{f_n}) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_{n-1}}) \circ \ \ldots\ \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_0})$$

is visited in turn. The purpose of this is to "push" the data function "$\pi_2 \circ \textbf{shr}^n$" from the left-most **unnest** function into the **unnest** functions to its right. Each "push" loses one of the "**shr**" subfunctions, and so the effect of this step is to rewrite the above composition chain into

$$\textbf{unnest}\ (g_n,\ \overline{f_n}) \circ \textbf{unnest}\ (g_{n-1},\ \overline{f_{n-1}}) \circ \ \ldots\ \circ \textbf{unnest}\ (g_2,\ \overline{f_2}) \circ \textbf{unnest}\ (\pi_2 \circ \textbf{shr},\ \overline{f_1}) \circ \textbf{unnest}\ (\textbf{id},\ \overline{f_0})$$

$$
\begin{array}{ll}
(1) & \textbf{iterate}\ (\mathtt{p} \oplus (\mathtt{f} \circ \mathtt{h}),\ \mathtt{g} \circ \mathtt{h}) \circ \textbf{unnest}\ (\mathtt{h1},\ \mathtt{h2})\ \stackrel{\rightarrow}{=} \\
 & \quad \textbf{iterate}\ (\mathtt{p} \oplus \mathtt{f},\ \mathtt{g}) \circ \textbf{unnest}\ (\mathtt{h} \circ \mathtt{h1},\ \mathtt{h2}) \\
(2) & \textbf{unnest}\ (\mathtt{f} \circ \textbf{shr} \circ \textbf{shr},\ \mathtt{g} \circ \textbf{shr}) \circ \textbf{unnest}\ (\mathtt{h1},\ \mathtt{h2})\ \stackrel{\rightarrow}{=} \\
 & \quad \textbf{unnest}\ (\mathtt{f} \circ \delta_1 \circ \textbf{shr},\ \mathtt{g}) \circ \textbf{unnest}\ (\textbf{shr} \circ \mathtt{h1},\ \mathtt{h2}) \\
(3) & \textbf{unnest}\ (\mathtt{f} \circ \pi_2 \circ \textbf{shr},\ \mathtt{g} \circ \pi_2) \circ \textbf{unnest}\ (\mathtt{h1},\ \mathtt{h2})\ \stackrel{\rightarrow}{=} \\
 & \quad \textbf{unnest}\ (\mathtt{f},\ \mathtt{g}) \circ \textbf{unnest}\ (\pi_2 \circ \mathtt{h1},\ \mathtt{h2}) \\
(4) & \textbf{unnest}\ (\pi_2 \circ \textbf{shr},\ \mathtt{f} \circ \pi_2) \circ \textbf{unnest}\ (\textbf{id},\ \mathtt{K}_f\ (\mathtt{A})) \circ \textbf{singleton}\ \stackrel{\rightarrow}{=} \\
 & \quad \mathtt{K}_f\ (\textbf{unnest}\ (\textbf{id},\ \mathtt{f})\ \mathtt{!}\ \mathtt{A})
\end{array}
$$

$$
\begin{array}{lllllll}
(5) & \mathtt{f} \circ \mathtt{K}_f\ (\mathtt{x}) & \stackrel{\rightarrow}{=} & \mathtt{K}_f\ (\mathtt{f}\ \mathtt{!}\ \mathtt{x}) & (6) & \mathtt{f} \ \stackrel{\rightarrow}{=} & \mathtt{f} \circ \textbf{shl} \circ \textbf{shr} \\
(7) & \pi_2 \circ \langle \mathtt{f},\ \mathtt{g} \rangle & \stackrel{\rightarrow}{=} & \mathtt{g} & (8) & \mathtt{f} \circ \textbf{id} \ \stackrel{\rightarrow}{=} & \mathtt{f} \\
(9) & \mathtt{K}_f\ (x)\ \mathtt{!}\ y & \stackrel{\rightarrow}{=} & x & & & \\
(10) & \multicolumn{6}{l}{\textbf{shr} \circ \langle \pi_1,\ \mathtt{f} \circ \pi_2 \rangle \circ \textbf{shl}\ \stackrel{\rightarrow}{=}\ \langle \pi_1,\ \langle \pi_1,\ \mathtt{f} \circ \pi_2 \rangle \circ \pi_2 \rangle} \\
(11) & \pi_2 \circ \delta_i & \stackrel{\rightarrow}{=} & \delta_{i-1} \circ \pi_2 & (12) & \textbf{shr}^i \circ \delta_1 \circ \textbf{shl}^i \ \stackrel{\rightarrow}{=} & \delta_{i+1}
\end{array}
$$

Figure 8: Rewrite Rules Used to Normalize the Results of Translation

for some functions $g_2, \ldots, g_n$.

Substituting for $\overline{f_i}$, the visit of the $(n-i+1)^{th}$ pair of **unnest** functions,

$$\textbf{unnest}\ (\pi_2 \circ \textbf{shr}^i,\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-1}) \circ \textbf{unnest}\ (\textbf{id},\ f_{i-1})$$

pushes the data function "$\pi_2 \circ \textbf{shr}^{i-1}$" out of the left **unnest** function and into the right. This is captured by the "macro rule",

$$
\begin{array}{l}
\textbf{unnest}\ (\pi_2 \circ \textbf{shr}^i,\ \mathtt{f} \circ \pi_2 \circ \textbf{shr}^{i-1}) \circ \textbf{unnest}\ (\textbf{id},\ \mathtt{g} \circ \pi_2 \circ \textbf{shr}^{i-2})\ \stackrel{\rightarrow}{=} \\
\quad \textbf{unnest}\ (F,\ \mathtt{f}) \circ \textbf{unnest}\ (\pi_2 \circ \textbf{shr}^{i-1},\ \mathtt{g} \circ \pi_2 \circ \textbf{shr}^{i-2})
\end{array}
$$

for some function F. Again, this rule is not expressible in COKO and instead we specify a COKO transformation that achieves the same result with the KOLA rules of Figure 8. In visiting the $(n-i+1)^{th}$ pair of **unnest** functions (for $i \geq 2$),

$$\textbf{unnest}\ (\pi_2 \circ \textbf{shr}^i,\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-1}) \circ \textbf{unnest}\ (\textbf{id},\ f_{i-1}),$$

this transformation first fires rule (2) and rule (8), leaving:

$$\textbf{unnest}\ (\pi_2 \circ \textbf{shr}^{i-2} \circ \delta_1 \circ \textbf{shr},\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-2}) \circ \textbf{unnest}\ (\textbf{shr},\ f_{i-1}).$$

Next, rule (6) fires $i-2$ times, leaving:

$$\textbf{unnest}\ (\pi_2 \circ \gamma \circ \textbf{shr}^{i-1},\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-2}) \circ \textbf{unnest}\ (\textbf{shr},\ f_{i-1}).$$

such that

$$\gamma = \textbf{shr}^{i-2} \circ \delta_1 \circ \textbf{shl}^{i-2}.$$

The firing of "macro rule" (12) (equivalently, $(i-2)$ firings of rule (10)) on γ leaves:

$$\textbf{unnest}\ (\pi_2 \circ \delta_{i-1} \circ \textbf{shr}^{i-1},\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-2}) \circ \textbf{unnest}\ (\textbf{shr},\ f_{i-1}).$$

Then, the firing of "macro rule" (11) (equivalently, rule (7)) leaves:

$$\textbf{unnest}\ (\delta_{i-2} \circ \pi_2 \circ \textbf{shr}^{i-1},\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-2}) \circ \textbf{unnest}\ (\textbf{shr},\ f_{i-1}).$$

This pass is repeated for every occurence of "**shr**" that needs to be passed from right to left. In this case, $(i-1)$ passes are completed in all, leaving

$$\textbf{unnest}\ (\delta_{i-2} \circ \ \ldots \ \circ \delta_0 \circ \pi_2 \circ \textbf{shr},\ f_i' \circ \pi_2) \circ \textbf{unnest}\ (\textbf{shr}^{i-1},\ f_{i-1}).$$

Subsequently, rule (3) fires, leaving:

$$\textbf{unnest}\ (\delta_{i-2} \circ \ \ldots \ \circ \delta_0,\ f_i') \circ \textbf{unnest}\ (\pi_2 \circ \textbf{shr}^{i-1},\ f_{i-1})$$

and, substituting for $\overline{f_{i-1}}$ the next adjacent pair of **unnest** functions,

$$\textbf{unnest}\ (\pi_2 \circ \textbf{shr}^{i-1},\ f_i' \circ \pi_2 \circ \textbf{shr}^{i-2}) \circ \textbf{unnest}\ (\textbf{id},\ f_{i-2})$$

is visited. In all, $n-1$ visits are made of adjacent **unnest** functions, leaving a query of the form,

$$\begin{array}{l}
(\textbf{iterate}\ (p',\ f') \circ \\
\ \ \textbf{unnest}\ (\delta_{n-2} \circ \ \ldots \ \circ \delta_0,\ f_n') \circ \textbf{unnest}\ (\delta_{n-3} \circ \ \ldots \ \circ \delta_0,\ f_{n-1}') \circ \\
\ \ \ \ldots \circ \\
\ \ \ \textbf{unnest}\ (\delta_0,\ f_2') \\
\ \ \ \ \textbf{unnest}\ (\pi_2 \circ \textbf{shr},\ f_1' \circ \pi_2) \circ \textbf{unnest}\ (\textbf{id},\ \mathtt{K}_f\ (A)) \circ \textbf{singleton})\ \mathtt{!}\ \mathtt{NULL}.
\end{array}$$

Steps 3, 4 and 5: In the next step,

$$\textbf{unnest}\ (\pi_2 \circ \textbf{shr},\ f_1' \circ \pi_2) \circ \textbf{unnest}\ (\textbf{id},\ \mathtt{K}_f\ (A)) \circ \textbf{singleton}$$

is rewritten by rule (4) leaving

$$\begin{array}{l}
(\textbf{iterate}\ (p',\ f') \circ \\
\ \ \textbf{unnest}\ (\delta_{n-2} \circ \ \ldots \ \circ \delta_0,\ f_n') \circ \textbf{unnest}\ (\delta_{n-3} \circ \ \ldots \ \circ \delta_0,\ f_{n-1}') \circ \\
\ \ \ \ldots \circ \\
\ \ \ \textbf{unnest}\ (\delta_0,\ f_2') \circ \mathtt{K}_f\ (\textbf{unnest}\ (\textbf{id},\ f_1')\ \mathtt{!}\ A))\ \mathtt{!}\ \mathtt{NULL}.
\end{array}$$

Then, rule (5) is fired in bottom-up fashion, leaving

$$\begin{array}{l}
\mathtt{K}_f\ (\textbf{iterate}\ (p',\ f')\ \mathtt{!} \\
\ \ \ \ (\textbf{unnest}\ (\delta_{n-2} \circ \ \ldots \ \circ \delta_0,\ f_n')\ \mathtt{!}\ (\textbf{unnest}\ (\delta_{n-3} \circ \ \ldots \ \circ \delta_0,\ f_{n-1}')\ \mathtt{!}\ \ldots\ \mathtt{!} \\
\ \ \ \ \ (\textbf{unnest}\ (\delta_0,\ f_2')\ \mathtt{!}\ (\textbf{unnest}\ (\textbf{id},\ f_1')\ \mathtt{!}\ A))))))\ \mathtt{!}\ \mathtt{NULL}.
\end{array}$$

Finally, rule (9) eliminates `NULL` from this query, leaving:

$$\begin{array}{l}
\textbf{iterate}\ (p',\ f')\ \mathtt{!} \\
\ (\textbf{unnest}\ (\delta_{n-2} \circ \ \ldots \ \circ \delta_0,\ f_n')\ \mathtt{!}\ (\textbf{unnest}\ (\delta_{n-3} \circ \ \ldots \ \circ \delta_0,\ f_{n-1}')\ \mathtt{!}\ \ldots\ \mathtt{!} \\
\ \ (\textbf{unnest}\ (\delta_0,\ f_2')\ \mathtt{!}\ (\textbf{unnest}\ (\textbf{id},\ f_1')\ \mathtt{!}\ A)))).
\end{array}$$

In all, the normalization transformation described above required 46 rewrite rules (including rules 1–10 of Figure 8) and fewer than 100 lines of firing algorithm code (including white space, lines only with delimiters `BEGIN` or `END` and comments). Note that this count includes general purpose transformations such as a transformation to simplify functions (`SimpFunc`), and the CNF transformation presented in Section 2. Not including these general-purpose transformations, 60 lines of firing algorithm code was required along with 24 rewrite rules.

4.2 Transforming Path Expressions to Joins

In this section, we describe a query rewrite that rewrites Query Lite queries with path expressions in their `WHERE` clauses into SQL join queries over their underlying DB2 implementations. Specifically, this rewrite affects *conjunctive* Query Lite queries of the form,

$$\texttt{SELECT}\ x\ \texttt{FROM}\ x\ \texttt{IN}\ A\ \texttt{WHERE}\ Comp_1\ \texttt{AND}\ \ldots\ \texttt{AND}\ Comp_m$$

such that each $Comp_i$ is a simple comparison predicate involving a path expression such as, "$x.\texttt{m}_0.\ldots.\texttt{m}_\texttt{n}\ op\ k$" or "NOT $(x.\texttt{m}_0.\ldots.\texttt{m}_\texttt{n}\ op\ k)$". Once translated into KOLA and normalized (as described in Section 4.1), conjunctive Query Lite queries have the form,

$$\textbf{iterate}\ (p_0\ \texttt{\&}\ \ldots\ \texttt{\&}\ p_m,\ \textbf{id})\ \texttt{!}\ A$$

such that each p_i is of the form

$$(op \oplus \langle \textbf{id},\ \texttt{K}_f\ \ (k)\rangle \oplus \texttt{m}_{\texttt{i}_\texttt{n}} \circ\ \ldots\ \circ \texttt{m}_0)\quad \text{or}\quad (\sim\ (op) \oplus \langle \textbf{id},\ \texttt{K}_f\ \ (k)\rangle \oplus \texttt{m}_{\texttt{i}_\texttt{n}} \oplus\ \ldots\ \oplus \texttt{m}_0),$$

and such that op is a KOLA binary predicate primitive (such as **eq**), each $\texttt{m}_j$ is a method primitive and k is a constant.

4.2.1 Illustrating the Effects of the Query Rewrite

The effects of the Query Lite → SQL rewrite are demonstrated in Figures 9 (a and d) and 1. Figure 9a shows a simple Query Lite query that returns all senators in `Sens` who have served more than 5 terms, and who represent a state whose capital city 1) has a population of over a million people and 2) has a mayor who was born in a city with a population of over a million people. The SQL equivalent of this query (Figure 9d) performs a join of the relations representing collections `Sens`, `Sts`, `Caps`, `Mays` and `Cits`. These collections appear in the join because attribute `reps` of `Sens` is a foreign key for `Sts` (i.e., column `reps` of the relational implementation of `Sens` contains OID values contained in a key column (`OID`) of `Sts`), attribute `capt` of `Sts` is a foreign key for `Caps`, attribute `mayor` of `Caps` is a foreign key for `Mays`, and attribute `bornin` of `Mays` is a foreign key for `Cits`. The join query filters `Caps` (x.`popn` $>$ $1M$) and `Caps` (c.`popn` $>$ $1M$) to identify capitals and cities with a population of over a million people, and filters `Sens` (s.`terms` $>$ 5) to find which have served more than 5 terms. The resulting collection of large cities is joined with `Mays` (m.`bornin == `x.`OID`) to identify which cities are birthplaces. Mayors are joined with the collection of large capitals (c.`mayor == `m.`OID`) to identify which mayors serve capital cities. Large capital cities are joined with `Sts` (r.`capt == `c.`OID`) to determine the states for which they are capitals. And states are joined with senior senators in `Sens` (s.`reps == `r.`OID`) to determine the senators that represent them. (Figures 9b and 9c show the KOLA representations of the Query Lite and SQL queries.)

Figures 1a and 1b show a more complex Query Lite query and the result of its rewrite into an SQL join query. (In the interests of space, we omit the KOLA representations of these queries here. The KOLA representations are shown in [2] however.) This example shows that rewriting handles queries with multiple subpredicates containing path expressions (including negations) and recognizes *shared* path expressions. The query of Figure 1a has subpredicates with various degrees of path expression sharing such as:

- "s.`bornin.popn` $>$ $1M$", which shares an entire path expression ("s.`bornin.popn`") with predicate "`NOT` (s.`bornin.popn` $>$ $2M$)"
- "s.`age` $>$ 50", which does not share a path expression with any other predicate, and

- "s.reps.capt.sistercity.popn $> 1M$", which shares part of its path expression ("s.reps.capt.sistercity)" with "NOT (s.reps.capt.sistercity.name == "Providence")", and a different part of its path expression ("s.reps.capt") with "s.reps.capt.popn $> 10M$ ".

If sharing of these and other path expressions were not recognized, the translation of this query would not be the five way join of Figure 1b, but the more expensive eleven way join shown below:

```
SELECT s1
FROM s1 IN Sens, s2 IN Sts, s3 IN Sts, s4 IN Sts, c1 IN Caps, c2 IN Cits,
     c3 IN Cits, c4 IN Cits, c5 IN Caps,, c6 IN Caps, c7 IN Cits
WHERE (c3.popn > 1M) AND NOT (c4.popn > 2M) AND NOT (c7.name == "Providence") AND
       (c2.popn > 1M) AND (c5.popn > 1M) AND (c1.sistercity == c3.OID) AND
       (s2.capt == c1.OID) AND (s1.pty == "GOP") AND (s1.reps == s3.OID) AND
       (s1.age > 50) AND (s1.bornin == c4.OID) AND (s1.bornin == c2.OID) AND
       (s1.reps == s2.OID) AND (s3.capt == c5.OID) AND (s1.reps == s4.OID) AND
       (s4.capt == c5.OID) AND (c6.sistercity == c7.OID).
```

Our rewriter recognizes path expression sharing for any number of path expressions in a WHERE clause, any degree of sharing, and any number of path expressions sharing a given subexpression (even when path expressions with shared subexpressions are not adjacent in the query).

4.2.2 The COKO Implementation of this Rewrite

In this section we describe the COKO implementation of the query rewrite to convert conjunctive Query Lite queries with path expressions in their WHERE clauses into SQL join queries. The "main" COKO transformation that performs this rewrite is PEWhere, shown in Figure 10 with its auxiliary transformation, PEWAux.[7] We will describe PEWhere by tracing the effects of its firing on the KOLA query of Figure 9b (the result of translating and normalizing the query of Figure 9a). Figure 9c shows the result of firing transformation PEWhere on this query.

Auxiliary Transformations: PEWhere fires a number of auxiliary COKO transformations whose code is given and described in [2]. These include:

- LBJoin: a transformation to transform multiple joins into "left-bushy" form. Though strictly not required, this transformation simplifies the translation of KOLA queries into SQL as we will show in Section 5.

- PullComFunc: a transformation to rewrite predicates of the form, "$(p_1 \oplus f_1)$ & ... & $(p_n \oplus f_n)$" into the form, "$(p'_1 \oplus f_1)$ & ... & $(p'_m \oplus f_m)$" such that $m \leq n$ and no two functions, f_i and f_j $(i \neq j)$ are equal. Thus, the effect of this transformation is to collapse all distinct conjuncts sharing the same subfunction, $(p_i \oplus f)$ and $(p_j \oplus f)$, into a predicate in which the subfunction appears once $((p_i$ & $p_j) \oplus f)$.

- SimpFunc and SimpPred: simplification predicates that reduce the (parse tree) sizes of functions and predicates respectively by eliminating redundancies etc.

Scope – **A Semantic Condition:** The transformations presented in this section exploit the semantic extensions to COKO-KOLA that were presented in [5] and described briefly in Section 2.1. Auxiliary

[7] The only aspects of these transformations not discussed in Section 2 are "don't care" variables (_) which are used like variables during unification but do not subsequently return variable bindings.

```
SELECT s
FROM s IN Sens
WHERE (s.reps.capt.popn > 1M) AND
      (s.terms > 5) AND
      (s.reps.capt.mayor.bornin.popn > 1M)
```
(a)

$$\textbf{iterate}\ ((\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{capt} \oplus \mathtt{reps})\ \texttt{\&}\ (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (5)\rangle \oplus \mathtt{terms})\ \texttt{\&}\ (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{bornin} \oplus \mathtt{mayor} \oplus \mathtt{capt} \oplus \mathtt{reps}),\ \textbf{id})\ !\ \mathtt{Sens}$$

(b)

$$\begin{array}{l}\textbf{join}\ ((((((((((((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2) \oplus \pi_2)\ \texttt{\&}\ p_3\ \texttt{\&}\ (p_4 \oplus \pi_1)) \oplus \pi_2)\ \texttt{\&}\ p_5) \oplus \pi_2)\ \texttt{\&} \\ \quad (((p_6 \oplus \pi_1)\ \texttt{\&}\ p_7) \oplus \langle \pi_1,\ \pi_1 \circ \pi_2\rangle)) \oplus \langle \pi_1,\ \langle \pi_1 \circ \pi_1,\ \textbf{shr} \circ \langle \pi_2 \circ \pi_1,\ \pi_2\rangle\rangle \circ \pi_2\rangle) \oplus \textbf{shr}) \oplus \\ \quad \langle\langle \pi_1 \circ \pi_1 \circ \pi_1,\ \textbf{shr} \circ \langle\langle \pi_2 \circ \pi_1,\ \pi_2\rangle \circ \pi_1,\ \pi_2\rangle\rangle \circ \pi_1,\ \pi_2\rangle), \\ \quad \pi_1 \circ \langle \pi_1 \circ \pi_1 \circ \pi_1,\ \textbf{shr} \circ \langle\langle \pi_2 \circ \pi_1,\ \pi_2\rangle \circ \pi_1,\ \pi_2\rangle\rangle \circ \pi_1)\ ! \\ \quad [\textbf{join}\ (\mathtt{K}_p\ (\mathtt{true}),\ \textbf{id})\ ! \\ \quad [\textbf{join}\ (\mathtt{K}_p\ (\mathtt{true}),\ \textbf{id})\ ! \\ \quad [\textbf{join}\ (\mathtt{K}_p\ (\mathtt{true}),\ \textbf{id})\ !\ [\mathtt{Sens},\ \mathtt{Sts}],\ \mathtt{Caps}],\ \mathtt{Mays}],\ \mathtt{Cits}]\end{array}$$

such that

$$\begin{array}{lll} p_1 = \textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} & & p_2 = \textbf{eq} \oplus \langle \mathtt{bornin} \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle \\ p_3 = \textbf{eq} \oplus \langle \mathtt{mayor} \circ \pi_1,\ \mathtt{OID} \circ \pi_1 \circ \pi_2\rangle & & p_4 = \textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \circ \mathtt{popn} \\ p_5 = \textbf{eq} \oplus \langle \mathtt{capt} \circ \pi_1,\ \mathtt{OID} \circ \pi_1 \circ \pi_2\rangle & & p_6 = \textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (5)\rangle \oplus \mathtt{terms} \\ p_7 = \textbf{eq} \oplus \langle \mathtt{reps} \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle & & \end{array}$$

(c)

```
SELECT s
FROM s IN Sens, r IN Sts, c IN Caps, m IN Mays, x IN Cits
WHERE (x.popn > 1M) AND (m.bornin == x.OID) AND
      (c.mayor == m.OID) AND (c.popn > 1M) AND
      (r.capt == c.OID) AND (s.terms > 5) AND (s.reps == r.OID)
```
(d)

Figure 9: A Query Lite Query (a), Its Normalization (b), Its Rewrite by `PEWhere` (c) Expressed In SQL (d)

transformation `PEWAux` of Figure 10 fires the conditional rewrite rule (`pe2j`) conditioned on the semantic property `scope`. `Scope` identifies foreign key relations. That is, the condition

`scope (B, f, A)`

states that `f` is an attribute of elements of `A` and is a foreign key for `B`. For our example schema, we assume that the scope relations shown in Table 4 have been declared.

Figure 11 shows a property definition for `Scope` that lists three inference rules for inferring this `scope` relations that are not explicitly declared. The first of these rules,

$$\mathtt{scope\ (B,\ f,\ A)} \Longrightarrow \mathtt{scope\ (B,\ f,\ }\textbf{iterate}\ \mathtt{(p,\ }\textbf{id}\mathtt{)\ !\ A)}$$

says that if `f` is a foreign key for `B` that is defined on `A`, then it is also a foreign key for `B` that is defined on any selection on `A`. The second and third rules,

$$\mathtt{scope\ (B,\ f,\ A)} \Longrightarrow \mathtt{scope\ (B,\ f,\ }\textbf{join}\ \mathtt{(p,\ }\pi_1\mathtt{)\ !\ [A,\ \_]).}$$
$$\mathtt{scope\ (B,\ f,\ A)} \Longrightarrow \mathtt{scope\ (B,\ f,\ }\textbf{join}\ \mathtt{(p,\ }\pi_2\mathtt{)\ !\ [\_,\ A]).}$$

```
TRANSFORMATION PEWhere
 -- Transforms QL queries with where predicates of the form,
 -- x.a1()...an() op k, or k op x.a1()...an()
 -- into join queries using scoping rules for attributes
 USES
  LBJoin,
  PEWAux,
  PullComFunc,
  SimpFunc,
  SimpPred
 BEGIN
   GIVEN iterate (p, _) ! _ DO PullComFunc (p);   % Step 1
   PEWAux;                                        % Step 2
   LBJoin                                         % Step 3
 END
```

```
TRANSFORMATION PEWAux
 USES
  pe2j:  scope (B, f, A) ::
          iterate (p ⊕ f, id) ! A ⟶
           join (eq ⊕ <f ∘ π1, OID ∘ π2⟩, π1) ! [A, iterate (p, id) ! B],
  splcon:  iterate (p & q, id) ! A ⟶ iterate (p, id) ! (iterate (q, id) ! A),
  SimpPred
 INFERS
  Scope
 BEGIN
  splcon → GIVEN _ ! x DO {PEWAux (x); PEWAux};
  pe2j → GIVEN join (p, _) ! [_, B] DO PEWAux (B)
 END
```

Figure 10: Transformation `PEWhere` and its Auxiliary Transformation

say that if `f` is a foreign key for `B` that is defined on `A`, then it is a foreign key for `B` that is defined on (left and right) semi-joins on `A`. Our COKO compiler compiles these inference rules into a decision-making procedure that gets executed when conditional rewrite rules are fired.

`PEWhere`: The firing algorithm for `PEWhere` consists of three steps. When invoked on a KOLA query,

$$\textbf{iterate}\ (p_0\ \texttt{\&}\ \ldots\ \texttt{\&}\ p_n,\ f)\ \texttt{!}\ A,$$

`PEWhere` first fires `PullComFunc` on $(p_0\ \texttt{\&}\ \ldots\ \texttt{\&}\ p_n)$ to extract common subexpressions from distinct path expressions. When applied to the query of Figure 9b, this step results in the expression, Q:

$$\textbf{iterate}\ (((\texttt{q}_{1a}\ \texttt{\&}\ \texttt{q}_{1b}) \oplus \texttt{capt} \oplus \texttt{reps})\ \texttt{\&}\ \texttt{q}_2,\ \textbf{id})\ \texttt{!}\ \texttt{Sens}$$

`scope (Sts, reps, Sens)`	`scope (Cits, mayor, Mays)`
`scope (Cits, bornin, Mays)`	`scope (Cits, bornin, Sens)`
`scope (Caps, capt, Sts)`	`scope (Mays, mayor, Cits)`
`scope (Mays, mayor, Caps)`	

Table 4: `Scope` relationships assumed declared

```
PROPERTY Scope
 BEGIN
  scope (B, f, A) ⟹ scope (B, f, iterate (p, id) ! A).
  scope (B, f, A) ⟹ scope (B, f, join (p, π1) ! [A, _]).
  scope (B, f, A) ⟹ scope (B, f, join (p, π2) ! [_, A]).
 END
```

Figure 11: Property `Scope` and Sample Metadata Infomation Regarding Scope

such that

$$\begin{aligned} \mathtt{q}_{1a} &= \mathbf{gt} \oplus \langle \mathbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn}, \\ \mathtt{q}_{1b} &= \mathbf{gt} \oplus \langle \mathbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{bornin} \oplus \mathtt{mayor} \text{ and} \\ \mathtt{q}_{2} &= \mathbf{gt} \oplus \langle \mathbf{id},\ \mathtt{K}_f\ (5)\rangle \oplus \mathtt{terms}. \end{aligned}$$

The second step of `PEWhere` fires auxiliary transformation, `PEWAux` on the expression resulting from the previous step. `PEWAux` has two key rules. The first is `splcon`, which splits an **iterate** function instantiated with a conjunction predicate into two. When fired on queries of the form,

$$\mathbf{iterate}\ (p'_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ p'_m,\ \mathbf{id})\ \mathtt{!}\ A$$

resulting from the previous step, this rule returns

$$\mathbf{iterate}\ (p'_1,\ \mathbf{id})\ \mathtt{!}\ (\mathbf{iterate}\ (p'_2\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ p'_m,\ \mathbf{id})\ \mathtt{!}\ A.$$

The effect of the recursive firings of `PEWAux` that follow the successful firing of `splcon` is to fire `PEWAux` successively on subqueries: (1) "**iterate** $(q_m,$ **id**$)$! A", (2) "**iterate** $(q_{m-1},$ **id**$)$! (**iterate** $(q_m,$ **id**$)$! $A)$", (3) "**iterate** $(q_{m-2},$ **id**$)$! (**iterate** $(q_{m-1},$ **id**$)$! (**iterate** $(q_m,$ **id**$)$! $A)$)", ... The second rule in `PEWAux` is the conditional rewrite rule, `pej2`. Provided that `scope` is inferred to be true for collection B, function f and collection A, the query,

$$\mathbf{iterate}\ (p \oplus f,\ \mathbf{id})\ \mathtt{!}\ A$$

gets rewritten by this rule to a join of A and the subcollection of B satisfying p:

$$\mathbf{join}\ (\mathbf{eq} \oplus \langle f \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle,\ \pi_1)\ \mathtt{!}\ \mathtt{[}A,\ \mathbf{iterate}\ (p,\ \mathbf{id})\ \mathtt{!}\ B\mathtt{]}.$$

This query screens B for those elements satisfying p, and then joins the result with A to return those elements of A whose values for f are represented on the subcollection of B. In terms of Query Lite queries, the left-hand side of the rule denotes queries of the form,

$$\mathtt{SELECT}\ x\ \mathtt{FROM}\ x\ \mathtt{IN}\ A\ \mathtt{WHERE}\ p\ (f\ (x)),$$

and in terms of SQL queries, the right-hand side of the rule denotes queries of the form,

$$\mathtt{SELECT}\ x\ \mathtt{FROM}\ x\ \mathtt{IN}\ A,\ y\ \mathtt{IN}\ B\ \mathtt{WHERE}\ ((f\ (x)\ \mathtt{==}\ y)\ \mathtt{AND}\ (p\ (y)).$$

Each time `pe2j` is successfully fired, it returns a query of the form,

$$\mathbf{join}\ (\mathbf{eq} \oplus \langle f \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle,\ \pi_1)\ \mathtt{!}\ \mathtt{[}A,\ \mathbf{iterate}\ (p,\ \mathbf{id})\ \mathtt{!}\ B\mathtt{]},$$

and `PEWAux` is fired recursively on the second argument to the join,

$$\mathbf{iterate}\ (p,\ \mathbf{id})\ \mathtt{!}\ B.$$

This recursive firing ensures that all methods appearing in a path expression contribute to the rewrite into a join query.

When PEWAux is fired on query Q (the result of the previous step on the query of Figure 9b) splcon successfully fires and triggers a recursive firing of PEWAux on

$$\textbf{iterate}\ (\mathtt{q}_2,\ \textbf{id})\ \mathtt{!\ Sens}.$$

Rule pe2j does not successfully fire on this expression (because there is no collection B such that

$$\mathtt{scope}\ (B,\ \mathtt{terms},\ \mathtt{Sens})$$

holds.) Therefore, a second recursive firing of PEWAux is made on

$$\textbf{iterate}\ ((\mathtt{q}_{1a}\ \&\ \mathtt{q}_{1b}) \oplus \mathtt{capt} \oplus \mathtt{reps},\ \textbf{id})\ \mathtt{!}\ (\textbf{iterate}\ (\mathtt{q}_2,\ \textbf{id})\ \mathtt{!\ Sens}).$$

The firing of PEWAux on this query fails to fire splcon, but succeeds in firing pe2j. The first inference rule for Scope in Figure 11 is used to infer scope (Sts, reps, **iterate** ($\mathtt{q}_2$, **id**) ! Sens) given the declared relationship, scope (Sts, reps, Sens). Therefore pej2 successfully fires, resulting in the query,

$$\begin{array}{l}\textbf{join}\ (\textbf{eq} \oplus \langle \mathtt{reps} \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle,\ \pi_1)\ \mathtt{!} \\ \ [\textbf{iterate}\ (\mathtt{q}_2,\ \textbf{id})\ \mathtt{!\ Sens},\ \textbf{iterate}\ ((\mathtt{q}_{1a}\ \&\ \mathtt{q}_{1b}) \oplus \mathtt{capt},\ \textbf{id})\ \mathtt{!\ Sts}].\end{array}$$

The successful firing of rule pe2j triggers another recursive firing of PEWAux, this time on

$$\textbf{iterate}\ ((\mathtt{q}_{1a}\ \&\ \mathtt{q}_{1b}) \oplus \mathtt{capt},\ \textbf{id})\ \mathtt{!\ Sts}.$$

Again, pe2j successfully fires on this subquery because of the declared relationship, scope (Caps, capt, Sts). This leaves the subexpression,

$$\begin{array}{l}\textbf{join}\ (\textbf{eq} \oplus \langle \mathtt{capt} \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle,\ \pi_1)\ \mathtt{!} \\ \ [\mathtt{Sts},\ \textbf{iterate}\ (\mathtt{q}_{1a}\ \&\ \mathtt{q}_{1b},\ \textbf{id})\ \mathtt{!\ Caps}].\end{array}$$

Next, PEWAux is recursively fired on

$$\textbf{iterate}\ (\mathtt{q}_{1a}\ \&\ \mathtt{q}_{1b},\ \textbf{id})\ \mathtt{!\ Caps},$$

or equivalently,

$$\begin{array}{l}\textbf{iterate}\ ((\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn})\ \& \\ \qquad\qquad (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{bornin} \oplus \mathtt{mayor}),\ \textbf{id})\ \mathtt{!\ Caps}.\end{array}$$

This first fires splcon leaving,

$$\begin{array}{l}\textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn},\ \textbf{id})\ \mathtt{!} \\ \ (\textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{bornin} \oplus \mathtt{mayor},\ \textbf{id})\ \mathtt{!\ Caps}).\end{array}$$

Then PEWAux is fired on

$$\textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{bornin} \oplus \mathtt{mayor},\ \textbf{id})\ \mathtt{!\ Caps}.$$

The declared relationship scope (Mays, mayor, Caps) leads to a successful firing of pej2, leaving

$$\begin{array}{l}\textbf{join}\ (\textbf{eq} \oplus \langle \mathtt{mayor} \circ \pi_1,\ \mathtt{OID} \circ \pi_2\rangle,\ \pi_1)\ \mathtt{!} \\ \ [\mathtt{Caps},\ \textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \mathtt{K}_f\ (1M)\rangle \oplus \mathtt{popn} \oplus \mathtt{bornin},\ \textbf{id})\ \mathtt{!\ Mays}].\end{array}$$

$$\mathbf{T}^{-1}\ [\![\textbf{iterate}\ (p,\ f)\ \texttt{!}\ A]\!] \quad = \quad \begin{array}{l}\texttt{SELECT}\ (\mathbf{T}^{-1}\ [\![f\ \texttt{!}\ x]\!])\\ \texttt{FROM}\ x\ \texttt{IN}\ A\\ \texttt{WHERE}\ (\mathbf{T}^{-1}\ [\![p\ \texttt{?}\ x]\!])\end{array}$$

$$\mathbf{T}^{-1}\left[\!\!\left[\begin{array}{l}\textbf{join}\ (p,\ f)\ \texttt{!}\\ \ \texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\\ \ \ \texttt{[...}\\ \ \ \ \texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\\ \ \ \ \ \texttt{[}A_1,\ A_2\texttt{]},\texttt{...}\ \texttt{]},\ A_n\texttt{]}\end{array}\right]\!\!\right] \quad = \quad \begin{array}{l}\texttt{SELECT}\ (\mathbf{T}^{-1}\ [\![f\ \texttt{!}\ \texttt{[...[}x_1,\ x_2\texttt{]},\ \texttt{...]},\ x_n\texttt{]}]\!])\\ \texttt{FROM}\ x_1\ \texttt{IN}\ A_1,\ \ldots,\ x_n\ \texttt{IN}\ A_n\\ \texttt{WHERE}\ (\mathbf{T}^{-1}\ [\![p\ \texttt{?}\ \texttt{[...[}x_1,\ x_2\texttt{]},\ \texttt{...]},\ x_n\texttt{]}]\!])\end{array}$$

$$\mathbf{T}^{-1}\ [\![\textbf{join}\ (p,\ f)\ \texttt{!}\ \texttt{[}A_1,\ A_2\texttt{]}]\!] \quad = \quad \begin{array}{l}\texttt{SELECT}\ (\mathbf{T}^{-1}\ [\![f\ \texttt{!}\ \texttt{[}x_1,\ x_2\texttt{]}]\!])\\ \texttt{FROM}\ x_1\ \texttt{IN}\ A_1,\ x_2\ \texttt{IN}\ A_2\\ \texttt{WHERE}\ (\mathbf{T}^{-1}\ [\![p\ \texttt{?}\ \texttt{[}x_1,\ x_2\texttt{]}]\!])\end{array}$$

Table 5: $\mathbf{T}^{-1}$: Applied to KOLA Formers

Next, `PEWAux` is again fired recursively on

$$\textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \texttt{K}_f\ (1M)\rangle \oplus \texttt{popn} \oplus \texttt{bornin},\ \textbf{id})\ \texttt{!}\ \texttt{Mays}.$$

The fact, `scope (Cits, bornin, Mays)` leads `pej2` to fire, leaving,

$$\begin{array}{l}\textbf{join}\ (\textbf{eq} \oplus \langle \texttt{bornin} \circ \pi_1,\ \texttt{OID} \circ \pi_2\rangle,\ \pi_1)\ \texttt{!}\\ \ \texttt{[Mays},\ \textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \texttt{K}_f\ (1M)\rangle \oplus \texttt{popn},\ \textbf{id})\ \texttt{!}\ \texttt{Cits]}.\end{array}$$

The subsequent recursive firing of `PEWAux` on

$$\textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \texttt{K}_f\ (1M)\rangle \oplus \texttt{popn},\ \textbf{id})\ \texttt{!}\ \texttt{Cits}$$

fails (as there is no collection B such that `scope (`B`, popn, Cits`) and the recursion terminates at all levels. Finally, the transformation is finished leaving the query,

$$\textbf{join}\ (\textbf{eq} \oplus \langle \texttt{reps} \circ \pi_1,\ \texttt{OID} \circ \pi_2\rangle,\ \pi_1)\ \texttt{!}\ \ \texttt{[}\textbf{iterate}\ (\texttt{q}_2,\ \textbf{id})\ \texttt{!}\ \texttt{Sens},\ B\texttt{]}$$

such that B is:

$$\begin{array}{l}\textbf{join}\ (\textbf{eq} \oplus \langle \texttt{capt} \circ \pi_1,\ \texttt{OID} \circ \pi_2\rangle,\ \pi_1)\ \texttt{!}\\ \ \texttt{[Sts},\ \textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \texttt{K}_f\ (1M)\rangle \oplus \texttt{popn},\ \textbf{id})\ \texttt{!}\\ \qquad \textbf{join}\ (\textbf{eq} \oplus \langle \texttt{mayor} \circ \pi_1,\ \texttt{OID} \circ \pi_2\rangle,\ \pi_1)\ \texttt{!}\\ \qquad \ \texttt{[Caps},\ \textbf{join}\ (\textbf{eq} \oplus \langle \texttt{bornin} \circ \pi_1,\ \texttt{OID} \circ \pi_2\rangle,\ \pi_1)\ \texttt{!}\\ \qquad \qquad \texttt{[Mays},\ \textbf{iterate}\ (\textbf{gt} \oplus \langle \textbf{id},\ \texttt{K}_f\ (1M)\rangle \oplus \texttt{popn},\ \textbf{id})\ \texttt{!}\ \texttt{Cits]]]]}.\end{array}$$

The final step of this transformation converts the query resulting from step 2 into a left-bushy join. This step is strictly not required, but simplifies the task of converting the resulting query into SQL as we show in Section 5. Applied to the query above, this step returns the query of Figure 9c.

The path expression $\rightarrow$ join transformations and property presented in this section required expression with 3 rewrite rules, 2 inference rules and fewer than 20 lines of firing algorithm code.

5 Translating KOLA Queries into SQL

Transformation `PEWhere` returns a left-bushy join query of the form,

$$\begin{array}{l}\textbf{join}\ (p_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ p_n,\ f)\ \texttt{!}\\ \ \texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\\ \ \ \texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\\ \ \ \ \texttt{[...}\\ \ \ \ \ \texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\ \texttt{[}A_1,\ A_2\texttt{]},\ \texttt{...]},\ A_{m-2}\texttt{]},\ A_{m-1}\texttt{]},\ A_m\texttt{]}\end{array}$$

$$\textbf{join}\ ((((((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2) \oplus \pi_2)\ \texttt{\&}\ p_3) \oplus \textbf{shr}),\ \pi_1 \circ \pi_1)\ \texttt{!}$$
$$\texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\ \texttt{[Sens, Sts], Caps]}$$

such that

$$\begin{array}{rcl} p_1 & = & \textbf{gt} \oplus \langle \textbf{id},\ \texttt{K}_f\ (1M)\rangle \oplus \texttt{popn} \\ p_2 & = & \textbf{eq} \oplus \langle \texttt{capt} \circ \pi_1,\ \texttt{OID} \circ \pi_2\rangle \\ p_3 & = & \textbf{eq} \oplus \langle \texttt{reps} \circ \pi_1,\ \texttt{OID} \circ \pi_1 \circ \pi_2\rangle \end{array}$$

Figure 12: A Simplified Version of Figure 9c

or a query over a single collection, **iterate** $(p,\ f)\ \texttt{!}\ A$. Table 5 defines a translation function ($\mathbf{T}^{-1}$) to translate these KOLA queries into SQL. Note that Table 5 includes two translation definitions for **join**. The first of these is for translating n-ary left-bushy joins ($n > 2$). The second of these is for translating binary joins only. All KOLA expressions not accounted for in this table (e.g., $\textbf{id}\ \texttt{!}\ x$) get translated into expressions corresponding to their operational semantics as defined in Tables 1 and 2.

To illustrate translation of KOLA into SQL, we trace the translation of a simplified version of the KOLA query of Figure 9c. This query, shown in Figure 12, results from first translating, then normalizing and finally firing `PEWhere` on the Query Lite query,

$$\texttt{SELECT}\ s\ \texttt{FROM}\ s\ \texttt{IN Sens WHERE}\ (s.\texttt{reps.capt.popn}\ >\ 1M).$$

Translation of this query into SQL first generates a `SELECT-FROM-WHERE` expression with KOLA terms:

$$\mathbf{T}^{-1} \left[\!\!\left[\begin{array}{l} \textbf{join}\ ((((((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2) \oplus \pi_2)\ \texttt{\&}\ p_3) \oplus \textbf{shr}),\ \pi_1 \circ \pi_1)\ \texttt{!} \\ \ \texttt{[}\textbf{join}\ (\texttt{K}_p\ (\texttt{true}),\ \textbf{id})\ \texttt{!}\ \texttt{[Sens, Sts], Caps]} \end{array} \right]\!\!\right]$$

$$= \begin{array}{l} \texttt{SELECT}\ \mathbf{T}^{-1}\ [\![(\pi_1 \circ \pi_1)\ \texttt{!}\ \texttt{[[}s,\ r\texttt{]},\ c\texttt{]}]\!] \\ \texttt{FROM}\ s\ \texttt{IN Sens},\ r\ \texttt{IN Sts},\ c\ \texttt{IN Caps} \\ \texttt{WHERE}\ \mathbf{T}^{-1}\ [\![(((((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2) \oplus \pi_2)\ \texttt{\&}\ p_3) \oplus \textbf{shr})\ \texttt{?}\ \texttt{[[}s,\ r\texttt{]},\ c\texttt{]}]\!] \end{array}$$

The expression in the `FROM` clause reduces as follows:

$$\begin{array}{rcl} \mathbf{T}^{-1}\ [\![(\pi_1 \circ \pi_1)\ \texttt{!}\ \texttt{[[}s,\ r\texttt{]},\ c\texttt{]}]\!] & = & \pi_1\ \texttt{!}\ (\pi_1\ \texttt{!}\ \texttt{[[}s,\ r\texttt{]},\ c\texttt{]}) \\ & = & \pi_1\ \texttt{!}\ \texttt{[}s,\ r\texttt{]} \\ & = & s \end{array} .$$

The expression in the `WHERE` clause reduces as follows:

$$\begin{array}{l} \mathbf{T}^{-1}\ [\![((((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2) \oplus \pi_2)\ \texttt{\&}\ p_3)\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]}]\!] \\ \quad = \ ((((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2) \oplus \pi_2)\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]})\ \texttt{AND}\ (p_3\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]}) \\ \quad = \ (((p_1 \oplus \pi_2)\ \texttt{\&}\ p_2)\ \texttt{?}\ \texttt{[}r,\ c\texttt{]})\ \texttt{AND}\ (p_3\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]}) \\ \quad = \ ((p_1 \oplus \pi_2)\ \texttt{?}\ \texttt{[}r,\ c\texttt{]})\ \texttt{AND}\ (p_2\ \texttt{?}\ \texttt{[}r,\ c\texttt{]})\ \texttt{AND}\ (p_3\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]}) \\ \quad = \ (p_1\ \texttt{?}\ c)\ \texttt{AND}\ (p_2\ \texttt{?}\ \texttt{[}r,\ c\texttt{]})\ \texttt{AND}\ (p_3\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]}) \end{array}$$

Subsequent simplification of the latter expressions reduce "$p_1\ \texttt{?}\ c$" to "$c.\texttt{popn}\ >\ 1M$", "$p_2\ \texttt{?}\ \texttt{[}r,\ c\texttt{]}$" to "$r.\texttt{capt == }c.\texttt{OID}$" and "$p_3\ \texttt{?}\ \texttt{[}s,\ \texttt{[}r,\ c\texttt{]]}$" to "$s.\texttt{reps == }r.\texttt{OID}$", leaving the SQL join query,

$$\begin{array}{l} \texttt{SELECT}\ s \\ \texttt{FROM}\ s\ \texttt{IN Sens},\ r\ \texttt{IN Sts},\ c\ \texttt{IN Caps} \\ \texttt{WHERE}\ (c.\texttt{popn}\ >\ 1M)\ \texttt{AND}\ (r.\texttt{capt == }c.\texttt{OID})\ \texttt{AND}\ (s.\texttt{reps == }r.\texttt{OID}), \end{array}$$

6 Discussion

6.1 Integration Capabilities of COKO-KOLA

The integration of the COKO-KOLA framework within the San Francisco project required the development of two translators. The first (described in Section 3) translates Query Lite queries (and more generally, all set and bag-based OQL queries) into KOLA. The second (described in Section 5) translates a subset of KOLA queries into SQL. Of these, the first was by far the easier to design and implement. This translator has a similar flavor to the many combinator translations of the lambda calculus (e.g., [7]).

Translation from KOLA to SQL was more tricky. The problem here was specifying *exactly* the sublanguage of KOLA that *could* be translated into SQL. For example, queries of the form, **iterate** $(p,\ f)$ `!` A, can be translated into KOLA, but only in certain cases. For example,

- f cannot be another **iterate** or **join** query,
- f can be a composition of some functions (e.g., **add** $\circ$ $\langle$`age`, $\mathtt{K}_f$ $(1)\rangle$) but not others (e.g., `capt` $\circ$ `reps`) that translate into path expressions, ...

Our approach to this issue is unsatisfying in the long term. In defining our KOLA $\rightarrow$ SQL translation function, $\mathbf{T}^{-1}$, we assumed that inputs were generated from the pipeline of (1) the Query Lite $\rightarrow$ KOLA translator, (2) the normalization query rewrite described in Section 4.1, and (3) the `PEWhere` rewrite described in Section 4.2. In fact, $\mathbf{T}^{-1}$ is not defined for all of KOLA, and worse, would translate some queries that were not produced by the pipeline into queries not recognized by SQL.

The long term solution is to somehow characterize exactly the sublanguage of KOLA that can be translated into SQL. We consider this to be extremely important. Query rewriting could prove useful for translating queries written for one data model to another. San Francisco provides one example of this, extending the capabilities of the DB2 query processor to handle object-oriented queries. But we foresee other applications: queries over sets of XML documents stored in a relational database, object database products that want to provide query support for OQL or SQL3 while using existing optimizer technology, and heterogeneous databases that might want to rewrite queries expressed over an entire federation of databases into separate queries specific to the individual databases included within the federation. We believe KOLA to be an ideal intermediate representation for these efforts, both because of its expressive power and because KOLA rewrites are verifiable with a theorem prover. But this will require far more precision in specifying sublanguages of KOLA that will serve as targets for query rewriting (e.g., the sublanguage of KOLA that can be translated into SQL.)

6.2 Ease-Of-Use of COKO-KOLA

For the San Francisco project, we used COKO-KOLA to generate:

1. a library of general-purpose normalization and simplification transformations (presented in Section 3),
2. a normalization to make the KOLA queries resulting from translation from OQL or Query Lite more intuitive (presented in Section 4), and
3. a transformation to rewrite Query Lite queries with path expressions in their `WHERE` clauses to SQL join queries (presented in Section 4.2).

Of these generated rewrites, the last by far was the easiest to write. This transformation rewrites queries with any number of path expressions, of any length, and with any degree of sharing. Yet, the transformation itself required fewer than 20 lines of firing algorithm code. The ease with which this transformation was written is encouraging, as it is exactly these kinds of transformations that are most likely to be developed for optimizers.

On the other hand, by far the most difficult transformations to write were the normalization transformations. This transformation required close to 50 rewrite rules and 100 lines of firing algorithm code. Of course, it should be remembered that this normalization need only be written once and users of the COKO-KOLA framework would most likely find it in a library of rewrites at their disposal, rather than programming it themselves. But nonetheless, our experience writing this transformation revealed certain deficiencies of COKO as a programming language and development environment. Such insights will motivate improvements in future versions of the language and compiler. Amongst our observations are the following:

COKO's Control Language This exercise provided quite a bit of experience with COKO's language for firing algorithms, and suggested some improvements. One aspect of the language that requires some additional thought is the association of success values with all statements in the language. For some statements (e.g., rule firings and complex statements), success values are quite natural and make certain common firing algorithms idioms straightforward to express. For example, the exhaustive firing of a set of rules, `ru1, ...,` `run` can be expressed as:

$$\texttt{BU}\ \{\texttt{ru1} \,||\, \ldots \,||\, \texttt{run}\} \ \rightarrow \ \textit{recursive call.}$$

The execution of a statement (S_2) if another statement (S_1) succeeds and a different statement (S_3) if it fails can be expressed as:

$$S_1 \ \rightarrow \ S_2 \,||\, S_3.$$

But for other statements, the success values associated with the statements sometimes ran counter to the desired flow of control. The most obvious example of this concerns the `GIVEN` statement. `GIVEN` statements have the form,

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

and return a success value of *true* if all equations, eqn_i succeed in unifying and *false* otherwise. While this success result is desirable sometimes, at other times the desired result is to return *true* if all equations succeed in unifying **and** statement S succeeds. To address this problem, we could change the default success value for `GIVEN` statements to return *true* if both equations and follow-up statement succeed. Then, one could simulate the old behavior of `GIVEN` (i.e., return *true* if all equations succeed regardless of the success value of the follow-up statement) by writing:

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

This may require us to rethink the entire notion of success values, perhaps restricting success values to rule and transformation invocations and adding `if-then-else` statements to the firing algorithm language to express desired control.

Pattern Matching Associative Formers One of the most time-consuming programming tasks is to account for the various ways that associative functions and predicates (e.g., $(f \circ g \circ h)$ or $(p \ \& \ q \ \& \ r)$) can be associated. Presently, we address this problem by normalizing functions and predicates in advance so that a certain association can be assumed (e.g., `LBComp` and `LBConj` make compositions and conjunctions left-associative (or "left-bushy")). But determining the associative structure guaranteed of transformation results (when such guarantees are possible) is difficult, and frequently we find ourselves normalizing perhaps unnecessarily, to ensure that functions and predicates are structured in a particular way.

In the long term, we believe that a better approach to this problem will be to adapt our matching algorithm to account for associative function and predicate formers. That is, under this scheme the pattern, "$f \circ g \circ h$" would match either possible association of compositions: "$f \circ (g \circ h)$" or "$(f \circ g) \circ h$". This approach is taken by theorem provers such as LP [8].

Speed of Generated Code The query rewrites that are generated from COKO transformations sometimes have poor performance. For example, transformation `PEWhere` performs well on "small queries" such as that of Figure 9a, it is slower with larger queries such as that of Figure 1a. (Executed on a 200Mz Sparcstation 10, the query of Figure 1a required roughly 15 seconds to translate, normalize and rewrite by `PEWhere`.) Possible contributing factors to poor performance are:

- The COKO compiler was designed for simplicity rather than performance. The code it produces generates a parse tree for the compiled transformations, and then invokes a method (`exec`) on its root (triggering subsequent calls to `exec` in descendants of the tree.) This approach is simple and makes it easy to extend the language with new statements (as is done with the definition of every new COKO transformation.) However, the resulting code frequently has inadequate performance. For example, the COKO statement, "$S_1; \ldots; S_n$" gets compiled into code that builds a binary parse tree of minimum height n, as each ";" appears as a node in the tree with the two statements it separates as its immediate children. This design results in a great deal of information passing (e.g., environments of unification variables) must be passed between parse tree nodes. Further, the generated parse trees can be so large that the C++ compiler cannot generate the code that builds them. We have often been forced to "break up" COKO transformations into separate subparts in order to get around this.

- As described in [2], our implementation of semantic query rewrites invokes a Prolog interpreter during rewriting to issue a semantic query. As in the case of `PEWAux`, semantic queries can be posed numerous times during the course of rewriting. (For example, in rewriting the relatively simple query of Figure 9a, the conditional rewrite rule, `pe2j` was fired 9 times (succeeding 4 times)). Each call to the Prolog interpreter incurs a tremendous amount of overhead from loading and initializing the interpreter, and converting to and from Prolog representations of KOLA expressions. We believe that performance of generated rewrites will improve greatly once our Prolog-based implementation of semantic query rewrites is replaced by specialized unification routines for KOLA trees.

In short, the message from our experience is that the COKO-KOLA framework potentially has much to offer in the development of "real-world" query rewriters given its formal foundation. But for this potential to be realized, its implementation must grow beyond its present prototype status. We were especially encouraged by the concise and elegant manner with which we were able to express the path expression-to-join query rewrite described in Section 4.2.2. This rewrite exploited all facets of the COKO-KOLA framework, including the semantic rewrite facility in order to infer knowledge of foreign keys, and recursive firing algorithms to ensure the handling of an unbounded number of path expressions of unbounded length and unbounded degree of subexpression sharing. But while our implementation is adequate as a proof of concept, it still is only a research prototype.

7 Conclusions

Many persistent systems are being built on top of existing database systems that assume a different underlying data model. Many object databases built before the definition of OQL are trying to add OQL support after the fact. Many relational databases are adding object capabilities. And relational storage engines are being considered as repositories for non-relational data such as that expressed in XML. In all of these cases, it is desirable that a user be able to write queries in terms of one language and data model, and have some system component translate these queries into the language supported by the storage engine.

Query rewriters could be used for such translations. In previous work, we introduced the COKO-KOLA framework which applies formal methods to the development of query rewriters. To test the practicality of this framework in an industrial setting, we implemented a translating query rewriter for the IBM San Francisco system, which offers object database and object query facilities on top of data stored in in DB2. This rewriter translates queries specified in an object query language into SQL.

The results were encouraging. It was simple to implement the translations from OQL to KOLA. These took only a few lines of code and little effort. The translation from KOLA to SQL was more difficult because the exact sub-language of KOLA that could be translated into SQL needed to be defined. Rewrites themselves were straightforward, although normalizing the results of translating from OQL required more effort than the rewrite that removes path expressions. Performance of generated rewrites was also a problem.

COKO-KOLA was created as a research prototype to test some ideas about query optimization and the use of formal methods. That it has matured to the point where it can tackle a real problem is a source of satisfaction to us. The experiment discussed in this paper showed that it is not an industrial-strength query optimizer but it also showed us some directions it must follow to become one.

References

[1] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan-Kaufman, 1993.

[2] Mitch Cherniack. *Building Query Optimizers with Combinators.* PhD thesis, Brown University, November 1998.

[3] Mitch Cherniack and Stanley 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.

[4] Mitch Cherniack and Stanley B. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, Seattle, WA, June 1998.

[5] Mitch Cherniack and Stanley B. Zdonik. Inferring function semantics to optimize queries. In *Proc. 24th Int'l Conference on Very Large Data Bases*, New York, NY, August 1998.

[6] IBM Corporation. *IBM DB2 Universal Database 1.0.* IBM Corporation, S10J-8165-00, 1998.

[7] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Birkhäuser, 1993.

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

[9] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yemeni, M. Breunig, and V. Vassalos. Template-based Wrappers in the TSIMMIS System. In *Proceedings of the ACM SIGMOD Conference*, Tucson, AZ, June 1997.

[10] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 364–374, 1990.

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

[12] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[13] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *International Conference on Deductive and Object-Oriented Databases*, pages 97–107, August 1995.

[14] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar 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.

[15] The IBM San Francisco Project. *IBM Systems Journal*, 37(2), 1998. Special Issue.

[16] C.M. Sperberg-McQueen T. Bray, J. Paoli. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, Located at `http://www.w3c.org/TR/REC-xml`, 1998.

[17] D. Wells. Wrappers. Survey paper. Available at http://www.objs.com/survey/wrap.htm.