Inferring Function Semantics to Optimize Queries

Mitch Cherniack
Brown University
Providence, RI 02912
mfc@cs.brown.edu

Stan Zdonik
Brown University
Providence, RI 02912
sbz@cs.brown.edu

Abstract

The goal of the COKO-KOLA project [10, 9] is to express rules of rule-based optimizers in a manner permitting verification with a theorem prover. In [10], we considered query transformations that were too *general* to be expressed with rewrite rules. In this paper, we consider the complementary issue of expressing query transformations that are too *specifc* for rewrite rules. Such transformations require rewrite rules to be supplemented with semantic conditions to guard rule firing. This work considers the expression of such transformations using *conditional rewrite rules*, and the expression of *inference rules* to guide the optimizer in deciding if semantic conditions hold. This work differs from existing work in semantic query optimization in that semantic transformations in our framework are verifiable with a theorem prover. Further, our use of inference rules to guide semantic reasoning makes our optimizer extensible in a manner that is complementary to the extensibility benefits of existing rule-based technology.

1 Introduction

Query optimizers are hard to build. In the past, relational optimizers have proved to be brittle and error-prone [17]. The added complexity of objects and hence object queries makes the task of building *object* (i.e., object-oriented and object-relational) database optimizers that much more difficult.

It is now accepted practice to use software engineering techniques when building optimizers. For example, many optimizers are now *rule-based* [3, 11], and therefore express the query-to-query or query-to-plan *transformations* that take place during optimization incrementally with *rules*. This approach makes optimizers *extensible* as the behavior of an optimizer can be altered by modifying its rule set. Further, this approach can make optimizers *verifiable* as the correctness of a rule-based optimizer follows from the correctness of the rules it fires.[1]

Optimizer verification becomes difficult when rules get expressed with code. Code is hard to reason about, and therefore rules are best expressed *declaratively*, as in the *rewrite rules* of term rewriting systems. A rewrite rule consists of a pair of patterns: a *lhs* (left-hand side) pattern that matches expressions that should be transformed, and a *rhs* (right-hand side) pattern that specifies the transformation of the expression. Rewrite rules are straightforward to verify with theorem provers as we showed in [10]. But rewrite rules lack the expressive power required to express many real query transformations.

1.1 General Transformations

Some query transformations are too general to be expressed with rewrite rules. Consider transformations that rewrite query expressions into syntactically characterizable forms (*normalizations*). Typically, normalizations affect large classes of syntactically varied expressions. For example, a normalization to transform Boolean expressions into conjunctive normal form (CNF) must transform expressions that are conjunctions, disjunctions, negations, quantifications and simple predicates. CNF cannot be expressed as a rewrite rule because no pair of patterns is both general enough to match this variety of expressions and specific enough to express their CNF equivalents. In short, rewrite rules depend on syntactic uniformity amongst the expressions they affect. Because normalizations apply to classes of expressions that lack syntactic uniformity, they defy expression with rules.

1.2 Semantic Transformations

Some query transformations are too specific to be expressed with rewrite rules. Consider a relational transformation to eliminate unnecessary duplicate removal from the processing a query that projects on a key attribute. This transformation (used in many relational systems such as Starburst [20]) is captured by the rewrite rule below, such that the patterns shown are SQL patterns, and pattern variables x, f, A and p match arbitrary SQL variables, attribute names, relation names and boolean expressions re-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

[1] A query optimizer is *correct* if it preserves the semantics of queries that it transforms. The *verification* of a query optimizer is with respect to this interpretation of correctness. Note that this says nothing about whether a query gets evaluated in an appropriate way.

spectively.

$$\begin{array}{lcl}\texttt{SELECT DISTINCT } x.f & & \texttt{SELECT } x.f \\ \texttt{FROM } x \texttt{ IN } A & \stackrel{\rightarrow}{=} & \texttt{FROM } x \texttt{ IN } A \\ \texttt{WHERE } p & & \texttt{WHERE } p\end{array}$$

The rewrite rule above must be additionally qualified by the restrictions that any attribute matching f be a key, and any collection matching A be a set. Any query that matches the initial pattern but does not satisfy these additional conditions would have its semantics (specifically, its element counts) changed as a result of rewriting. Semantic conditions such as those identifying f as a key and A as a set cannot be expressed with patterns. Therefore, transformations whose validity depends on semantic conditions such as these cannot be expressed solely with rewrite rules.

Existing rule-based systems address the expressivity issues above by replacing or supplementing rewrite rules with code. General transformations are expressed as rules supplemented with code in the rule's rhs to manipulate matched expressions in ways that cannot be expressed with patterns. For example, CNF would be expressed in Cascades [12] as a "function rule" whose firing invokes user-defined code. Semantic transformations are expressed with rules supplemented with code in the rule's lhs to test semantic conditions of expressions that successfully matched the head pattern. For example, the SQL transformation above that eliminates duplicate removal would be expressed in Starburst [20] with C code that examined annotations of the underlying query representation (QGM) to decide if a matched attribute was a key and if a matched collection was a set.

As we argued in [10], code-based rules are difficult to verify. We seek an alternative means of expressing both general and semantic transformations that enables their verification with a theorem prover. In [9], we proposed a new language (COKO) for expressing general transformations in terms of sets of declarative rewrite rules and an algorithm to control their firing (a *firing algorithm*). This paper proposes a complementary technique for expressing semantic transformations. As with COKO, this work builds upon our KOLA [10] foundation which used a combinator-based (i.e., variable-free) query algebra to express rewrite rules without code. To express semantic transformations, we propose the addition of two alternative kinds of rules for rule-based optimizers:

- *Conditional rewrite rules*, and
- *Inference rules.*

Conditional rewrite rules are like (unconditional) rewrite rules, except that when they are fired, the matching of the rule's head pattern to a query expression is followed by analysis to see if certain conditions hold of identified subexpressions. Inference rules tell the optimizer how to decide if the conditions of conditional rewrite rules hold.

The contributions of this work are as follows:

1. *Verifiable Semantic Transformations:* In keeping with our goal, all transformations specifiable with inference and conditional rewrite rules are verifiable with a theorem prover.

2. *Use of Inference Rules to Infer Query Semantics:* Our work is unique in its use of inference rules to specify semantic conditions. This technique separates the rules that depend on semantic conditions (conditional rewrite rules) from the decision making process that decides if these semantic conditions hold. This distinguishes our approach from that of existing rule-based systems that embed decision making code within the rule that may or may not fire as a result.

The use of inference rules to specify semantic conditions makes optimizers extensible in ways that standard rule-based optimizers are not. By modifying the set of inference rules defining a given semantic condition, one changes the circumstances under which that rule (and any other rule using the same condition) gets fired. This provides a complementary form of extensibility to that which comes from expressing query transformations with rewrite rules, the modification of which changes the set of possible queries and plans that can be output.

Section 2 provides a review of KOLA and COKO, giving the context for the work introduced in this paper. In Section 3, we demonstrate how conditional rewrite rules and inference rules are integrated within the COKO-KOLA framework by presenting the specification of two example transformations. We also discuss the extensibility and verifiability benefits of our approach with respect to these examples. In Section 4 we present the implementation of our optimizer that processes inference and conditional rewrite rules. We compare our work with related work in Section 5 and conclude in Section 6.

2 Background

This section reviews the COKO-KOLA approach to expressing query transformations. In the interest of space, we review just what is required for this paper. More in-depth treatments can be found in [10] (for KOLA) and [9] (for COKO).

2.1 KOLA

KOLA is a combinator-based query representation. As such, queries are built out of other functions using special combinators or *formers*. The expression of a query (or any other function or predicate)[2] contains no variables. This makes it straightforward to use a theorem prover to verify rewrite rules.

KOLA is designed to be easy to manipulate by the optimizer as opposed to being easy to read. Users write queries in a language such as OQL, and those queries are translated into KOLA. We have a working OQL to KOLA translator described in [7].

[2] In KOLA, functions and predicates are separated and invoked differently ("!" for functions; "?" for predicates).

KOLA	Semantics
id	$\textbf{id} \; ! \; x \;=\; x$
π_1	$\pi_1 \; ! \; [x,\, y] \;=\; x$
π_2	$\pi_2 \; ! \; [x,\, y] \;=\; y$
$<\texttt{att}>$	$<\texttt{att}> \; ! \; x \;=\; x.<\texttt{att}>$
$\circ$	$(f \circ g) \; ! \; x \;=\; f \; ! \; (g \; ! \; x)$
$\langle\ \rangle$	$\langle f,\, g\rangle \; ! \; x \;=\; [f \; ! \; x,\, g \; ! \; x]$
$\times$	$(f \times g) \; ! \; [x,\, y] \;=\; [f \; ! \; x,\, g \; ! \; y]$
$\mathtt{K}_f$	$\mathtt{K}_f \; (x) \; ! \; y \;=\; x$
$\mathtt{C}_f$	$\mathtt{C}_f \; (f,\, x) \; ! \; y \;=\; f \; ! \; [x,\, y]$
eq	$\textbf{eq} \; ? \; [x,\, y] \;=\; x \;==\; y$
lt	$\textbf{lt} \; ? \; [x,\, y] \;=\; x \;<\; y$
$\oplus$	$(p \oplus f) \; ? \; x \;=\; p \; ? \; (f \; ! \; x)$
&	$(p \;\&\; q) \; ? \; x \;=\; (p \; ? \; x) \;\wedge\; (q \; ? \; x)$
$\mid$	$(p \mid q) \; ? \; x \;=\; (p \; ? \; x) \;\vee\; (q \; ? \; x)$
$\sim$	$\sim (p) \; ? \; x \;=\; \neg \; (p \; ? \; x)$
$\mathbf{K}_p$	$\mathtt{K}_p \; (b) \; ? \; x \;=\; b$
$\mathbf{C}_p$	$\mathtt{C}_p \; (p,\, x) \; ? \; y \;=\; p \; ? \; [x,\, y]$

Table 1: KOLA Function and Predicate Semantics

$$\textbf{set} \; ! \; A \;=\; \{x \mid x^i \in A\}$$
$$\textbf{iterate} \; (p,\, f) \; ! \; A \;=\; \{\!\{(f \; ! \; x)^i \mid x^i \in A,\, p \; ? \; x\}\!\}$$

$$\textbf{join} \; (p,\, f) \; ! \; [A,\, B] \;=\; \{\!\{(f \; ! \; [x,\, y])^{ij} \mid x^i \in A,\, y^j \in B,\, p \; ? \; [x,\, y]\}\!\}$$

$$\textbf{exists} \; (p) \; ? \; A \;=\; \exists \, x,\, j \; (x^j \in A \;\wedge\; p \; ? \; x)$$
$$\textbf{forall} \; (p) \; ? \; A \;=\; \forall \, x,\, j \; (x^j \in A \;\Rightarrow\; p \; ? \; x)$$

Table 2: KOLA Query Formers

KOLA's operators are defined in Tables 1 and 2. These definitions assume that f and g denote arbitrary functions, p and q denote arbitrary predicates, A and B denote arbitrary *collections* (i.e., sets and multisets), and x and y denote arbitrary values or objects. The primitive functions presented in this table include identity (**id**) and projection functions on pairs (π_1 and π_2), as well as schema-based primitive *attributes* (denoted by $<\texttt{att}>$) such as an `age` attribute for `People`. KOLA's general function formers include composition ($\circ$), function pairing ($\langle\ \rangle$), pairwise function application ($\times$), constant functions ($\mathtt{K}_f$) and curried functions ($\mathtt{C}_f$). Primitive predicates include equality (**eq**), and ordering predicates such as "less than" (**lt**). KOLA's predicate formers include predicate/function combination ($\oplus$), the logic-inspired formers conjunction (&), disjunction ($\mid$) and negation ($\sim$), and the constant ($\mathbf{K}_p$) and curried predicate formers ($\mathbf{C}_p$).

Table 2 shows KOLA primitives and formers for generating functions or predicates on collections (queries). We use set comprehensions in describing set semantics, and the following notation in describing multiset semantics:

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

$$k = \sum_{x^i \in A, p(x), f(x) \,==\, v} g(i).$$

KOLA's query primitives and formers include:

- **set**: a function on collections to remove duplicates,
- **iterate** $(p,\, f)$: a query former resembling SQL's `select-from-where` construct. Generated functions apply function f to every element of the argument collection that satisfies predicate p.
- **join** $(p,\, f)$: a query former that accepts a binary predicate p and a binary function f to produce a function on pairs of collections, $[A,\, B]$. This function joins A and B by applying f to pairs of elements, $[a,\, b]$, that are drawn respectively from A and B and that satisfy p.
- **exists** (p)/**forall** (p): query formers that accept a predicate p to produce existential (universal) quantifier predicates on collections A that return `true` if some (all) elements of A satisfy p.

There is no conceptual difference between the formers of Table 1 and those of Table 2; all create complex functions and predicates from simpler ones. That queries are first-class functions and predicates facilitates reasoning about their meaning, as was demonstrated in [8] by formally specifying KOLA with the Larch algebraic specification language LSL [15] and by verifying well over 300 KOLA rewrite rules with the Larch theorem prover (LP).

2.2 COKO

COKO [9] is our language for expressing general transformations. A COKO transformation consists of a set of KOLA rewrite rules and a firing algorithm to control their firing. Because all query modification performed by a COKO transformation is by rule firing, a COKO transformation can be verified by verifying the rules it fires. Therefore like KOLA rewrite rules, COKO transformations can be verified with a theorem prover.

3 Adding Semantic Capabilities to COKO

In this section, we motivate and illustrate our semantic extensions to COKO. The example transformations used to demonstrate these extensions are not new. But by comparing their expression in other systems with their expression in our framework, we demonstrate the verification and extensibility benefits of our approach. This section concludes with a discussion of these benefits as well as the advantage of using KOLA to implement these extensions.

Type	Attributes		Key?
State	`name`	*String*	Primary
	`senator`	*Senator*	Secondary
	`capital`	*City*	Secondary
	`cities`	$\{City\}$	Secondary
City	`name`	*String*	Primary
	`mayor`	*Mayor*	Secondary
	`popn`	*Integer*	No
Senator	`name`	*String*	Primary
	`reps`	*State*	Secondary
	`party`	*Party*	No
	`terms`	*Integer*	No
Mayor	`name`	*String*	Primary
	`reps`	*City*	Secondary
	`party`	*Party*	No
Party	`name`	*String*	Primary
	`leader`	*Senator*	Secondary
	`mayors`	$\{Mayor\}$	Secondary

Table 3: An Object Database Schema

```
select distinct x.reps.capital
from x in S
```

(a)

```
select distinct (select d.mayor
                  from d in x.reps.cities)
from x in S
```

(b)

Figure 1: The "Capitals" (a) and "Mayors" (b) Queries

3.1 A Motivating Example

Table 3 shows an object database schema assumed for the examples in this paper. This schema models a country's political structure and includes type definitions for *States*, *Cities*, *Senators*, *Mayors* and (Political) *Parties*. A state's attributes include its name (`name`), senator (`senator`), capital city (`capital`) and the set of cities it includes (`cities`). A city's attributes include its name (`name`), mayor (`mayor`) and population (`popn`). Senators and mayors have names (`name`), a state/city that he/she represents (`reps`) and a party affiliation (`party`). A senator also has served some number of terms (`terms`). A party has a name (`name`), a leader (`leader`) and a set of mayors who belong to the party (`mayors`). For simplicity, we assume that names form primary keys for all types listed. However, each type has its own secondary keys that include `senator`, `capital` and `cities` (for states), `mayor` (for cities), `reps` (for senators and mayors), and `leader` and `mayors` (for parties).

Figures 1a and 1b show object (specifically, OQL [4]) queries over this political database. The "Capitals Query" (Figure 1a) queries a set of senators (`S`) applying the *path expression*, `x.reps.capital`, to each. The result of this query is the collection of capital cities of states represented by the senators in `S` (with duplicate cities removed). The

```
select x.reps.capital
from x in S
```

(a)

```
select (select d.mayor
         from d in x.reps.cities)
from x in S
```

(b)

Figure 2: The Queries of Figure 1 After Transformation

"Mayors Query" (Figure 1b) also queries a set of senators. For each senator, this query returns the mayors of cities in the state that the senator represents (this time, with duplicate mayor collections removed from the result).

The "Capitals Query" and the "Mayors Query" can be evaluated in similar ways: by first retrieving senators in `S`, applying their *data functions* (i.e., the expressions in their `SELECT` clauses) to each, storing the results in an intermediate collection and then eliminating duplicates from this stored collection. Duplicate elimination requires an initial sort or hash of the contents of these collections followed by a scan of the result for equal, consecutive elements. For the "Capitals Query" duplicates are cities with the same name. For the "Mayors Query" duplicates are collections with the same members.

Duplicate elimination is expensive but unnecessary in the case of both queries. Because of the *semantics* of their data functions, both queries generate intermediate collections that already are free of duplicates. No state is represented by more than one senator, and no city is a capital for more than one state. Therefore, the "Capitals Query" inserts a distinct city into its intermediate result for each distinct senator. As `S` has no duplicates, neither will this collection of cities. Similarly, every state has a unique collection of cities and every city has a unique mayor. Therefore, the collections of mayors generated as an intermediate result of the "Mayors Query" also will not require duplicate elimination.

Transformed versions of both queries that avoid duplicate elimination are shown in Figure 2a and 2b. The transformation is similar to the relational query transformation presented in Section 1.2. However, these transformations are more general in that they can be applied to queries that cannot be expressed as relational queries (such as object queries with path expressions or subqueries as data functions). This requires more sophisticated analysis of query semantics than was required in the relational case. Specifically, for this transformation to be valid for an object query, its data functions need not be key attributes but any *injective* function (of which keys comprise a special case). A relational query optimizer need only consult metadata files (e.g., the database schema) to determine if the query's data function is a key. But there can be far more cases to consider in the case of object queries. The number of injective path expressions alone might be very large and even infinite. (Aside from `x.reps.capital`, other injective path expressions on sen-

1a. **set** ! (**iterate** (K_p (true), σ) ! S)
(σ = capital ∘ reps)

1b. **set** ! (**iterate** (K_p (true), σ) ! S)
(σ = **iterate** (K_p (true), mayor) ∘ cities ∘ reps)

2a. **iterate** (K_p (true), σ) ! S
(σ = capital ∘ reps)

2b. **iterate** (K_p (true), σ) ! S
(σ = **iterate** (K_p (true), mayor) ∘ cities ∘ reps)

Figure 3: KOLA Translations of Figures 1a, 1b, 2a and 2b

ators include `x.name`, `x.reps`, `x.reps.capital.name`, `x.reps.capital.mayor.party.leader` and so on.) Thus, it is unlikely that metadata files can be scaled to keep track of all data functions that make this transformation valid, and inference of conditions that make the transformation valid is required instead.

3.2 Conditional Rules and Inference Rules in COKO

In keeping with the spirit established by KOLA and preserved by COKO, we express semantic transformations with two kinds of declarative rules:

Conditional Rewrite Rules resemble (unconditional) rewrite rules, but can include semantic preconditions on subexpressions of matched query expressions. Such conditions can, for example, indicate that a given KOLA function must be injective or that a given KOLA collection must be a set. Like unconditional rules, conditional rewrite rules can be fired by COKO transformations.

Inference Rules specify how semantic conditions can be inferred of KOLA functions, predicates, objects and collections. The inference rules used to define semantic conditions are compiled by our COKO compiler into decision making algorithms invoked during rule firing.

3.2.1 Inference Rules in COKO

To perform the query transformation described earlier, an optimizer must determine that a query's data function is injective and that a collection is a set. Like most semantic properties of functions, injectivity is undecidable in general. But, inferring injectivity in some cases is better than not inferring it at all for at least in those cases optimization might improve the evaluation of the query. Therefore, we care about soundness and not about completeness in inferring semantic properties.

Figure 3 shows KOLA equivalents of the "Capitals Query" and "Mayors Query" both before and after the application of the transformation to remove duplicate elimination (**set**). The KOLA translations of the data functions (σ) for these two queries are:

- capital ∘ reps, equivalent to the "Capitals Query" path expression, `x.reps.capital`, and
- **iterate** (K_p (true), mayor) ∘ cities ∘ reps, equivalent to the "Mayors Query" subquery,

```
select d.mayor
from d in x.reps.cities.
```

An optimizer constructed within our framework could infer that these functions are injective according to the specifications of the *inference rules* of Figure 4a. These rules have the form,

$$\textit{body} \implies \textit{head}$$

(or just *head* which states a *fact* that is unconditionally true). The *head* of a rule names a condition (e.g., inj (f)) to infer. A condition is an uninterpreted logical relation whose arguments can be either KOLA expressions or pattern variables (such as f) that implicitly are universally quantified.

The *body* of a rule is a logical sentence (i.e., consisting of conjunctions ($\wedge$), disjunctions ($\vee$) and/or negations ($\neg$) of terms), whose terms are conditions that must be satisfied to infer the head condition. To illustrate, the rules of Figure 4a should be interpreted as follows:

1. The identity (**id**) function is injective,
2. A KOLA function is injective if it is a key,
3. KOLA function $f \circ g$ is injective if both f and g are injective,
4. KOLA function $\langle f,\ g \rangle$ is injective if either f and g are injective, and
5. KOLA query function **iterate** (K_p (true), f) is injective if f is injective.

Provided that an optimizer can discern from metadata that reps and capital are keys, and the type of S is set (T) for some type, T, rules 2, 3 and 5 of Figure 4a are sufficient to decide that the "Capitals Query" and "Mayors Query" can be transformed safely.

Figure 4b shows some of the inference rules an optimizer might use to decide if collections are sets. These rules state that (1) the result of invoking the function, **set**, on any collection is a set (the "don't care" expression (_) indicates that the argument to **set** is irrelevant), (2) a collection is a set if its declared type is a set, (3) the cartesian product of two sets is a set, (4) the intersection of any two collections (of which one is a set) is a set, and (5) taking the difference of any collection from a set returns a set.

3.2.2 Conditional Rewrite Rules in COKO

Conditional rewrite rules have the form,

$$C\ ::\quad L \stackrel{\rightarrow}{=} R$$

such that L and R are patterns of KOLA expressions (i.e., $L \stackrel{\rightarrow}{=} R$ is an unconditional rewrite rule), and C is a set of semantic conditions that must hold of various subexpressions of query expressions that match L. A conditional rewrite

$$\begin{array}{lr} \texttt{is\_inj}\,(\textbf{id}). & (1)\\ \texttt{is\_key}\,(f) \implies \texttt{is\_inj}\,(f) & (2)\\ \texttt{is\_inj}\,(f) \wedge \texttt{is\_inj}\,(g) \implies \texttt{is\_inj}\,(f \circ g) & (3)\\ \texttt{is\_inj}\,(f) \vee \texttt{is\_inj}\,(g) \implies \texttt{is\_inj}\,(\langle f,\, g\rangle). & (4)\\ \texttt{is\_inj}\,(f) \implies \texttt{is\_inj}\,(\textbf{iterate}\ (\texttt{K}_p\ (\texttt{true}),\ f)) & (5) \end{array}$$

(a)

$$\begin{array}{lr} \texttt{is\_set}\,(\textbf{set}\ !\ \_). & (1)\\ \texttt{is\_type}\,(A,\, \texttt{set}\,(\_\_)) \implies \texttt{is\_set}\,(A). & (2)\\ \texttt{is\_set}\,(A) \wedge \texttt{is\_set}\,(B) \implies \texttt{is\_set}\,(A \times B). & (3)\\ \texttt{is\_set}\,(A) \vee \texttt{is\_set}\,(B) \implies \texttt{is\_set}\,(A \cap B). & (4)\\ \texttt{is\_set}\,(A) \implies \texttt{is\_set}\,(A - B). & (5) \end{array}$$

(b)

Figure 4: Inference Rules for Inferring that Functions are Injective *(a)* and Collections are Sets *(b)*

rule that specifies the transformation used for the "Capitals Query" and the "Mayors Query" is shown below.

$$\texttt{is\_inj}\,(f),\ \texttt{is\_set}\,(A) ::\\ \textbf{set}\ !\ (\textbf{iterate}\ (p,\ f)\ !\ A) \stackrel{\rightarrow}{\leftarrow} \textbf{iterate}\ (p,\ f)\ !\ A.$$

The lhs rule pattern matches queries that remove duplicates (with **set**) from the results of select-project (**iterate**) queries. The rhs rule pattern shows the same query as the lhs but with the invocation of **set** removed. The conditions (which use the same names as the conditions defined by inference rules) state that this rule is valid provided that the data function, f, is injective and the collection, A, is a set. Therefore, conditional rewrite rules specify transformations that should only be fired if certain conditions hold.

A second conditional rewrite rule conditioned on the injectivity of a function is shown below:

$$\texttt{is\_inj}\,(f) ::\\ \textbf{iterate}\ (p,\ f)\ !\ (A \cap B) \stackrel{\rightarrow}{\leftarrow}\\ (\textbf{iterate}\ (p,\ f)\ !\ A) \cap (\textbf{iterate}\ (p,\ f)\ !\ B).$$

Intersection is typically implemented with joins. Thus, this rule effectively pushes selections (p) and projections (f) past joins. f must be injective for the rewrite to be correct for if it is not, then the query that results from firing this rule might return more answers than the original query. (For example, if f is the noninjective squaring function, A contains 3 but not -3, and B contains -3 but not 3, then then the query resulting from firing this rule may include 9 in its result whereas the original query will not.)

If S and S′ are collections of senators, then this rule could be used with the inference rules described earlier to transform a query that returns the capital cities of all states represented by senators in both S and S′ who have served at least 5 terms,

$$\textbf{iterate}\ (\texttt{C}_p\ (\textbf{lt},\ \texttt{5}) \oplus \texttt{terms},\ \texttt{capital} \circ \texttt{reps})\ !\ (\texttt{S} \cap \texttt{S}')$$

into the equivalent query,

$$(\textbf{iterate}\ (\texttt{C}_p\ (\textbf{lt},\ \texttt{5}) \oplus \texttt{terms},\ \texttt{capital} \circ \texttt{reps})\ !\ \texttt{S})\\ \cap\\ (\textbf{iterate}\ (\texttt{C}_p\ (\textbf{lt},\ \texttt{5}) \oplus \texttt{terms},\ \texttt{capital} \circ \texttt{reps})\ !\ \texttt{S}').$$

The initial query first takes a potentially expensive intersection of collections of senators, before filtering the result for those who have served more than 5 terms. The transformed version of this query filters the collections of senators for their senior members first, before performing the intersection of the presumably smaller collections that result.

3.3 Another Example: Predicate Strength

Predicate strength is unlike injectivity in that it holds of two predicates rather than of individual functions. As with the previous example, the transformations discussed here are not new — many are implemented in commercial database systems and some were proposed in the context of relations by Levy et. al. in [19]. What is new is their expression with declarative rules that simplifies their verification and extension.

A predicate p is "stronger" than a predicate q ($\texttt{is\_stronger}\,(p,\ q)$) if p always implies q. More formally, for any predicates p and q over objects of type T,

$$\texttt{is\_stronger(p, q)} \iff\\ \forall\ \texttt{x}{:}T\ \ (\texttt{p ? x} \Rightarrow \texttt{q ? x}).$$

Predicate strength is used as a condition for two kinds of rewrite rules:

- If p is stronger than q and a query requires that both p and q be invoked on some object, x, then the query can be transformed to only invoke p. This is advantageous in certain circumstances because it saves the cost of invoking q.
- If p is stronger than q and a query requires that p be invoked on some object, x, then the query can be transformed to invoke both p and q. This is advantageous in cases where q is cheaper to invoke than p, and therefore invoking q before invoking p limits the objects on which p must be invoked.

3.3.1 Rules for Predicate Strength

Figure 5 shows inference rules for inferring predicate strength. Rule (1) states that any predicate is stronger than itself. Rules (2) and (3) state that if $f\ !\ x == g\ !\ y$, and predicate p is known to be true of $f\ !\ x$ ($g\ !\ y$), then it must also be true of $g\ !\ y$ ($f\ !\ x$).

must also hold of the other. Rule (4) states that equality of partial path expressions implies equality on full path expressions. That is:

$$\texttt{x.a}_1.\ldots.\texttt{a}_i\ =\ \texttt{y.a}_1.\ldots.\texttt{a}_i\ \implies\\ \texttt{x.a}_1.\ldots.\texttt{a}_i.\ldots.\texttt{a}_n\ =\ \texttt{y.a}_1.\ldots.\texttt{a}_i.\ldots.\texttt{a}_n.$$

Rules (5) and (6) show how predicate strength can be inferred of predicate conjuncts. Rule (7) uses the injectivity property described earlier to say that equality of keys implies equality of all other attributes.

$$\texttt{is\_stronger}\,(p,\, p). \tag{1}$$
$$\texttt{is\_stronger}\,((\mathbf{eq} \oplus (f \times g))\ \&\ (p \oplus f \oplus \pi_1),\, p \oplus g \oplus \pi_2). \tag{2}$$
$$\texttt{is\_stronger}\,((\mathbf{eq} \oplus (f \times g))\ \&\ (p \oplus g \oplus \pi_2),\, p \oplus f \oplus \pi_1). \tag{3}$$
$$\texttt{is\_stronger}\,(\mathbf{eq} \oplus (f \times f),\, \mathbf{eq} \oplus ((g \circ f) \times (g \circ f))). \tag{4}$$
$$\texttt{is\_stronger}\,(p,\, q) \wedge \texttt{is\_stronger}\,(p',\, q') \implies \texttt{is\_stronger}\,(p\ \&\ p',\, q\ \&\ q'). \tag{5}$$
$$\texttt{is\_stronger}\,(p,\, r) \vee \texttt{is\_stronger}\,(q,\, r) \implies \texttt{is\_stronger}\,(p\ \&\ q,\, r). \tag{6}$$
$$\texttt{is\_inj}\,(f) \implies \texttt{is\_stronger}\,(\mathbf{eq} \oplus (f \times f),\, \mathbf{eq} \oplus (g \times g)). \tag{7}$$

Figure 5: Inference Rules for Inferring Predicate Strength

Figure 6 shows rewrite rules conditioned on predicate strength. Rule (1) says that if p is stronger than q, then the conjunction of p and q can be rewritten to p. It is sometimes advantageous to add a predicate to an existing query rather than remove one (we show an example of this later) and therefore rule (2) is the *inverse* of rule (1). Rules (3), (4) and (5) state that quantification with a weaker predicate over the result of filtering a collection with a stronger predicate can be simplified to avoid traversing the collection at all (rules (3) and (5)) or to quantify over an unfiltered collection (rule (4)).

3.3.2 Example Uses of Predicate Strength

Example 1: The OQL query expression below returns true if a universally quantified predicate holds of a result of a subquery. The subquery performs a join of senator collections, S and S′ returning a collection of pairs that agree on their party affiliations. `For all` returns true if the senators paired by this subquery all agree on the leaders of their party.

```
for all y in
  (select struct (one:  s1, two:  s2)
   from s1 in S, s2 in S′
   where s1.party == s2.party):
 y.one.party.leader == y.two.party.leader
```

Because all pairs of senators resulting from the subquery agree on their party affiliations, all pairs will also agree on the leaders of the parties with which they are affiliated. Therefore, this complex expression can be transformed into the constant, `true`.

The inference rules of Figure 5 and conditional rewrite rules of Figure 6 justify the transformation of the KOLA equivalent of the expression above,

$$\mathbf{forall}\ (\mathbf{eq} \oplus ((\texttt{leader} \circ \texttt{party}) \times (\texttt{leader} \circ \texttt{party})))\ ?\ \mathbf{join}\ (\mathbf{eq} \oplus (\texttt{party} \times \texttt{party}),\ \mathbf{id})\ !\ [\texttt{S},\, \texttt{S}']$$

into the constant, `true`. By setting f to `party` and g to `leader`, rule (4) of Figure 5 establishes $p = \mathbf{eq} \oplus (\texttt{party} \times \texttt{party})$ to be stronger than $q = \mathbf{eq} \oplus ((\texttt{leader} \circ \texttt{party}) \times (\texttt{leader} \circ \texttt{party}))$. Rule (5) of Figure 6 then uses these bindings of p and q to rewrite this expression to `true`.

Example 2: Whereas the previous example used predicate strength to avoid invoking predicates unnecessarily, the following examples *add* predicates to queries to make them more efficient to evaluate. These examples evoke the spirit of the "predicate move-around" transformations of [19].

The OQL query below joins senators from collections S and S′ who have served the same number of terms such that the senator from S has served more than 5 terms. As this query stands, it likely would be evaluated by first filtering senators in S to include only those who have served more than 5 terms, and then joining this result with S′.

```
select *
from s1 in S, s2 in S′
where s1.terms > 5 AND
      s1.terms == s2.terms
```

A better form of this query introduces a new predicate (`s2.terms > 5`) on senators in S′:

```
select *
from s1 in S, s2 in S′
where s1.terms > 5 AND
      s1.terms == s2.terms AND
      s2.terms > 5
```

The addition of this predicate does not change the semantics of the query, as any senators from S′ that appear in the original query result will have served more than 5 terms because they will have served the same number of terms as some senator in S who has served more than 5 terms. Put another way, this transformation is justified because the predicate,

```
s1.terms > 5 AND s1.terms == s2.terms
```

is stronger than the predicate, `s2.terms > 5` . This transformation is advantageous as it makes it likely that *both* S and S′ will be filtered for their senior senators, before submitted as inputs to the join.

The KOLA equivalents of these two queries are shown below. The first query would be expressed in KOLA as, $\mathbf{join}\ (\rho,\ \mathbf{id})\ !\ [\texttt{S},\, \texttt{S}']$ such that ρ is:

$$(\mathbf{eq} \oplus (\texttt{terms} \times \texttt{terms}))\ \&\ (\texttt{C}_p\ (\mathbf{lt},\ 5) \oplus \texttt{terms} \oplus \pi_1).$$

The second query is $\mathbf{join}\ (\rho\ \&\ \tau,\ \mathbf{id})\ !\ [\texttt{S},\, \texttt{S}']$ such that τ is: $\texttt{C}_p\ (\mathbf{lt},\ 5) \oplus \texttt{terms} \oplus \pi_2$. The transformation of ρ to $\rho\ \&\ \tau$ is justified by rewrite rule (2) of Figure 6 with p set to ρ and q set to τ. That p is stronger than q is justified

$$\texttt{is\_stronger}\,(p,q) \;::\; (p \;\&\; q) \;\stackrel{\rightrightarrows}{=}\; p \qquad (1)$$
$$\texttt{is\_stronger}\,(p,q) \;::\; p \;\stackrel{\rightrightarrows}{=}\; (p \;\&\; q) \qquad (2)$$
$$\texttt{is\_stronger}\,(p,q) \;::$$
$$\textbf{forall}\;(q)\;?\;(\textbf{iterate}\;(p,\;\textbf{id})\;!\;\_)\;\stackrel{\rightrightarrows}{=}\;\texttt{true} \qquad (3)$$
$$\texttt{is\_stronger}\,(p,q) \;::$$
$$\textbf{exists}\;(q)\;?\;(\textbf{iterate}\;(p,\;\textbf{id})\;!\;A)\;\stackrel{\rightrightarrows}{=}\;\textbf{exists}\;(p)\;?\;A \qquad (4)$$
$$\texttt{is\_stronger}\,(p,q) \;::$$
$$\textbf{forall}\;(q)\;?\;(\textbf{join}\;(p,\;\textbf{id})\;!\;[\_,\,\_])\;\stackrel{\rightrightarrows}{=}\;\texttt{true} \qquad (5)$$

Figure 6: Rewrite Rules Conditioned on Predicate Strength

by inference rule (2) of Figure 5 (setting p to $\mathtt{C}_p\;(\mathbf{lt},\;\mathtt{5})$ and f and g to `terms`.)

Example 3: Predicate strength rules can be used to generate *new* predicates and not just to duplicate existing ones, as the following example shows. The query below pairs senators in `S` who represent states whose capital cities have more than 100 000 people, with mayors in `M` who are mayors of those cities:

```
select *
from s in S, m in M
where s.reps.capital.popn > 100K AND
      s.reps.capital == m.reps
```

The KOLA equivalent to this query is, $\textbf{join}\;((\rho \oplus \gamma \oplus \pi_1)\;\&\;\tau,\;\textbf{id})\;!\;[\mathtt{S},\,\mathtt{M}]$ such that

$$\begin{aligned}\rho &= \mathtt{C}_p\;(\mathbf{lt},\;\mathtt{100K}) \oplus \mathtt{popn},\\ \gamma &= \mathtt{capital} \circ \mathtt{reps},\;\text{ and}\\ \tau &= \mathbf{eq} \oplus (\gamma \times \mathtt{reps}).\end{aligned}$$

Rule (2) of Figure 5 can be used to generate a **new** predicate that can filter the mayors that participate in the join. Specifically, by setting f to γ, g to `reps` and p to ρ, the new predicate,

$$\mathtt{C}_p\;(\mathbf{lt},\;\mathtt{100K}) \oplus \mathtt{popn} \oplus \mathtt{reps} \oplus \pi_2$$

can be determined to be weaker than $(\rho \oplus \gamma \oplus \pi_1)\;\&\;\tau$. Thus, applying rewrite rule (2) of Figure 6 leaves a query that would be expressed in OQL as:

```
select *
from s in S, m in M
where s.reps.capital.popn > 100K AND
      s.reps.capital == m.reps AND
      m.reps.popn > 100K
```

such that `m.reps.popn > 100K` is a *new* predicate and not just a copy of a predicate that appeared elsewhere in the original query. If the number of mayors who serve cities with populations over 100 000 is small, or if mayors are indexed on the populations of their cities, then this transformation is likely to make the query more efficient to evaluate.

3.4 Discussion

3.4.1 Benefits Of Our Approach

The examples of the previous section demonstrate our approach to *expressing* semantic query transformations. The transformation of Section 3.1 that recognizes when duplicate elimination is unnecessary is used in many commercial relational database systems. It is also presented as one of the Starburst transformations in [20] performed during the *query rewriting* phase of query processing. In Starburst, this transformation is used as a normalization step before view merging. Subqueries that perform duplicate elimination make view merging impossible because duplicate semantics are lost as a result of the merge. Starburst uses this transformation in order to recognize subqueries that can be transformed into equivalent queries that perform no duplicate elimination so that view merging can take place thereafter. The transformations of Section 3.3 that use predicate strength have also been considered elsewhere. Those that remove quantification from complex predicates (Example 1 of Section 3.3.2) are standard techniques that one can find in many textbooks. Those that introduce new predicates (Examples 2 and 3 of Section 3.3.2) are similar to the "predicate move-around" techniques for transforming relational queries proposed in [19].

What is unique in our work is the use of declarative conditional rewrite rules and inference rules to express these complex transformations. With our approach we can verify all of the rules presented in these sections with a theorem prover. (See Appendix A for LP theorem prover scripts for these rules.) Verification of conditional rewrite rules establishes that query semantics are preserved when these rules are fired on queries satisfying the rules' semantic preconditions. Verification of inference rules establishes that semantic conditions are inferred only when appropriate (soundness).

The other contribution of this approach concerns extensibility. [20] and [19] present the transformations discussed in Sections 3.1 and 3.3 in the context of relational databases. To simulate their results, we do not need all of the inference rules of Figure 4a that infer injectivity, nor do we need all of the inference rules of Figure 5 that infer predicate strength. For example, to capture the duplicate elimination transformation presented in [20] for relational queries, we only need inference rules that establish an attribute to be injective if it is a key (Figure 4a, rule (2)) and if it is a pair (equivalently, a relational tuple[3]) containing a key (Figure 4a, rule (4)). Rule (3) of Figure 4a is not needed as there is no notion of a composed data function in the relational data model. But if the relational version of this transformation were expressed in our framework, it would be straightforward to **extend** this transformation (to work for example, in an object database setting) simply by adding a verified inference rules such as rule (3) of Figure 4a. Note that the addition of this one inference

[3] Our translator translates all tuple expressions into (potentially nested) KOLA pairs.

rule makes the rewrite rules conditioned on injectivity fired in a greater variety of contexts (e.g., when queries include path expressions with keys, or tuples with fields containing path expressions with keys etc.). By similar reasoning, not all of the predicate strength inference rules of Figure 5 are required to express the transformations of [19] when confined to relations (e.g., rule (4) of Figure 5 is unnecessary because of its use of function composition). Again, rules such as this one can be added to simply extend a relational optimizer to work robustly in an object setting.

3.4.2 The Advantage of KOLA

In [10], we showed that the KOLA's combinator style makes it easier to formulate declarative (unconditional) rewrite rules to express query transformations. Query representations that include variables make it difficult to express rewrite rules without code supplements because representations can include subexpressions with free variables. The meaning of these subexpressions is context-dependent (dependent for example, on how free variables were declared in their surrounding expressions). Because rewrite rules *identify* and *move* query subexpressions into new expressions, code supplements to rules are required to analyze the context of subexpressions to determine if they should be identified, and to massage subexpressions so that their meaning does not change as a result of their being moved into a new query.

The combinator flavor of KOLA makes declarative rewrite rules easier to express because the meaning of a KOLA subexpression is context-independent. Therefore, code supplements are not required to distinguish between subexpressions that look the same but have different meanings. Nor are code supplements required to massage subexpressions so that their meaning is preserved when transplanted into a different expression.

The advantage of combinators extends to the formulation of conditional rewrite rules and inference rules. Conditional rewrite rules must identify subexpressions upon which to express conditions. Inference rules must identify subexpressions because conditions tend to be inferred from conditions held of subexpressions (e.g., the injectivity of complex functions can be inferred from the injectivity of their subfunctions.) Again, variables in a query representation complicate the identification of these subexpressions.

Consider as an example, the data functions appearing in the "Capitals Query" and the "Mayors Query". The KOLA forms of these functions are:

$$\texttt{capital} \circ \texttt{reps}, \text{ and}$$
$$\textbf{iterate}\ (\texttt{K}_p\ (\texttt{true}),\ \texttt{mayor}) \circ \texttt{cities} \circ \texttt{reps}.$$

These two functions are both injective by similar reasoning: they are compositions of other functions which are also injective. Inferring injectivity of the OQL forms of these data functions,

```
x.reps.capital, and

select distinct d.mayor
from d in x.reps.cities
```

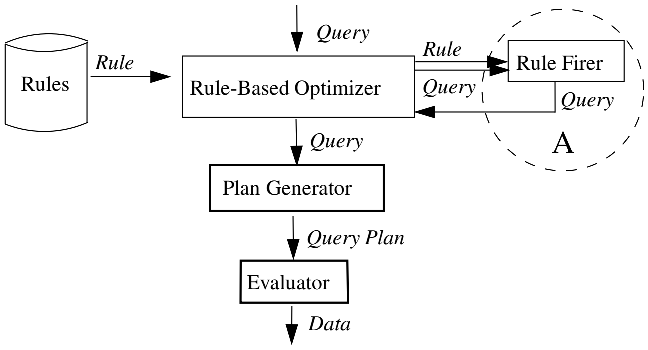

Figure 7: A Typical Rule-Based Optimizer Architecture

is more complicated. The identification of both of these data functions as being compositions of other functions requires machinery beyond what can be expressed with rewrite rules. Specifically, determining exactly what are the subfunctions of these functions requires reversing the process of substituting expressions for variables. This requires factoring the complex expressions denoting the functions into two expressions for which the substitution of one for a variable in the other reproduces the original expression. For the path expression, `x.reps.capital`, these subfunctions are `x.reps` and `x.capital` (as substituting the first of these expressions for `x` in the second expression reproduces the original path expression). For the subquery, the subfunctions are, `x.reps.cities` and

```
select distinct d.mayor
from d in x
```

as again, substituting the first expression for `x` in the second expression results in the original expression. The decomposition required to identify subfunctions is inexpressible with declarative rewrite rules and instead requires calls to supplemental code.

4 Implementation

In the previous section, we showed that declarative inference rules and conditional rewrite rules could be used to guide a query optimizer in how to infer semantic conditions that hold of functions and how to exploit these semantic conditions to perform query transformations. In this section, we show how these rules are processed in our optimizer implementation.

4.1 Implementation Overview

A rule-based optimizer receives a query to process, and then selects rules to fire on the query. Rule selection and firing is performed repeatedly until an equivalent query is constructed that is amenable to efficient plan generation. This query is then submitted to the plan generator (which can be rule-based also). This architecture is illustrated in Figure 7.

One of the key components of the rule-based optimizer is the *rule firer*, which accepts representations of a query

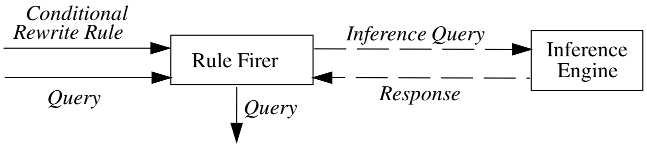

Figure 8: A Conditional Rewrite Rule Firer

and a rule as inputs and produces a new query representation (resulting from firing the rule) as a result. This component is labeled (A) in Figure 7. We implemented the ideas discussed in this paper by extending the operation of the rule firer. The new rule firer, illustrated in Figure 8, extends the original firer in two ways:

Inference: The new rule firer can consult an inference engine to infer conditions relevant to the firing of conditional rewrite rules. The rule firer makes a request of the inference engine by issuing *inference queries* such as:

- *is the function,* $\mathtt{capital} \circ \mathtt{reps}$ *injective*?, or
- *can any predicate be generated that is weaker than:*

$(\mathtt{C}_p\ (\mathbf{lt},\ 5) \oplus \mathtt{terms} \oplus \pi_1)\ \&\ (\mathbf{eq} \oplus (\mathtt{terms} \times \mathtt{terms}))$?

The inference engine answers queries with a simple yes or no (as in the first inference query above) or with KOLA expressions that satisfy the inference query (as in the second inference query above)

Conditional Rule Firing: The new rule firer accepts conditional rewrite rules (as well as unconditional rules) as inputs. When such rules are fired, inference queries are posed to the inference engine and the answers interpreted.

Section 4.2 presents the inference engine component of our optimizer. Section 4.3 describes the operation of our rule firer in the presence of conditional functions.

4.2 The Inference Engine

Our inference engine is the Sicstus Prolog interpreter [21]. Using Prolog as an inference engine makes our implementation a prototype rather than one of commercial quality. We envision replacing the Prolog interpreter with specialized unification routines that operate directly on KOLA queries as future work.

The interpreter's inputs are Prolog programs that are:

- built-in facts and rules describing aspects of the data model that are invariant (e.g., rules for inferring subtyping, type information for KOLA operators etc.),
- facts and rules generated from inference rules, and
- facts generated from metadata information specific to a given database instance, such as types contained in the schema, signatures of attributes, types of persistent data, and attributes that are keys.

Presently, metadata based rules are generated manually. However, Prolog facts and rules are generated automatically by compiling sets of inference rules. The mapping of inference rules into Prolog facts and rules is straightforward. To illustrate, inference rules (1), (3) and (5) of Figure 4a are translated respectively into the Prolog rules:

```
pis_inj (id).
pis_inj (compose (Vf, Vg)) :-
  pis_inj (Vf), pis_inj (Vg).
pis_inj (pairing (Vf, Vg)) :-
  pis_inj (Vf) ; pis_inj (Vg).
```

The following observations about translation can be made from this example:

- KOLA primitives (e.g., **id**) are translated into unique Prolog constants (e.g., `id`).
- Variables that denote arbitrary KOLA expressions (e.g., f and g) are translated into Prolog variables by prepending a capital '`V`' (`Vf` and `Vg`). (Prolog variables are required to be capitalized).
- Formed KOLA expressions (e.g., $f \circ g$) are translated into prefix notation (e.g., `compose (Vf, Vg)`). (Prolog terms must be expressed in prefix notation.)
- Names of conditions (e.g., `is_inj`) are prepended with a lower-case '`p`' (`pis_inj`). (Prolog relations must begin with lower-case letters.)
- Inference facts (e.g., rule (1) of Figure 4a) are translated into Prolog facts.
- Inference rules are translated into Prolog rules. Tails of inference rules that include conjunctions (disjunctions) translate into Prolog rules whose body terms are separated by commas (semi-colons).

4.3 Integrating Inference and Rule Firing

Below we illustrate the steps that are performed when a conditional rewrite rule is fired on a query. We demonstrate these steps by tracing the firing of the rule,

$$\mathtt{is\_inj}\,(f), \mathtt{is\_set}\,(A)\ ::\ \mathbf{set}\ !\ (\mathbf{iterate}\ (p,\ f)\ !\ A) \stackrel{\rightarrow}{=} \mathbf{iterate}\ (p,\ f)\ !\ A$$

with pattern variables p, f and A, on the KOLA version of the "Capitals Query" (Figure 3: 1a).

1. The lhs pattern of the rule above is matched with the "Capitals Query" generating an *environment* of bindings for pattern variables: p (bound to $\mathtt{K}_p\ (\mathtt{true})$), f (bound to $\mathtt{capital} \circ \mathtt{reps}$) and A (bound to `S`).
2. A Prolog query is generated. First, Prolog subqueries of the form, $\mathtt{V}_i = \mathtt{T}_i$ are generated for each variable, $\mathtt{V}_i$ appearing in the lhs pattern (p,

f and A). For a given variable $\texttt{V}_i$, $\texttt{T}_i$ is the Prolog translation of the subexpression bound to $\texttt{V}_i$. In the case of the "Capitals Query", the generated subqueries are: `Vp = const (true)`, `Vf = compose (fcapital, freps)`, and `VA = oS`. (Note that global names (such as `S`) are prepended with a lower-case '`o`' in their translation into Prolog, and attributes (such as **`reps`**) are similarly prepended with a lower-case '`f`'.) These Prolog subqueries are then added to a Prolog subquery generated by translating the conditions of the conditional rewrite rule. The Prolog query generated from firing the conditional rewrite rule on the "Capitals Query" is

```
?- Vp = const(true), VA = oS,
   Vf = compose (fcapital, freps),
   pis_inj (Vf), pis_set (VA).
```

3. The generated Prolog query is issued to the Prolog interpreter with the built-in rules described earlier, and the relevant Prolog facts and rules generated from inference rules and metadata. In the case of the "Capitals Query", the relevant Prolog rules would be those resulting from the compilation of the inference rules of Figure 4a and 4b, and the metadata facts: `pis_key (freps)`, `pis_key (fcapital)` and `ptype (oS, set (senator))`.

4. The Prolog query is posed to the Prolog interpreter and the results interpreted. If the results include new variable bindings to KOLA expressions expressed as Prolog terms, these terms are parsed back into KOLA expressions and added to the environment of (variable, subexpression) bindings generated in Step (1). For the "Capitals Query", the Prolog interpreter uses the translations of inference rules (2) and (3) (Figure 4a) and (2) (Figure 4b) to reduce the inference query generated in step (2) to the simpler queries, `pis_key (fcapital)`, `pis_key (freps)` and `pis_type (oS, set (_))`, all of which succeed.

5. The environment generated in steps (1) and (4) is used to instantiate the pattern variables appearing in the rhs pattern of the conditional rewrite rule. The instantiated pattern is then returned as the output of rule firing. In the case of the "Capitals Query", the returned query is: **iterate** (K_p (**`true`**), **`capital`** $\circ$ **`reps`**) `! S`.

5 Related Work

The contribution of our work is in the expression of semantic transformations in a manner supporting verification and extensibility. This work contrasts with existing rule-based systems (e.g., [20], [3, 13, 12]) that use code to express semantic conditions (thereby compromising verifiability) and that embed this code within rewrite rules (thereby compromising extensibility).

Our approach could be used to express, verify and extend semantic transformations from many sources. This includes semantic transformations used in relational systems (aside from those mentioned here, these primarily involve the use of integrity constraints as discussed in [16], [18] and [5]). But our approach also permits expression of semantic transformations that depend on the semantics of queries (i.e., functions) and not just data.

The semantic transformations that appear in the literature can be classified into three categories: (1) those that neither do inference nor have conditional rules but that express operator-specific rules, (2) those that have conditional rules but that perform no inference, and (3) those that have both conditional rules and inference. As we move from the first to the third category, we get more general making the approach more scalable.

Chaudhuri and Shim's work [6] involves optimizations of SQL queries that contain foreign functions. They incorporate rewrite rules over foreign functions to express equivalent expressions. Each equivalence must be captured in a separate rule. These rules are always valid given that they are specific to particular operators. Therefore they perform no inference nor conditional rewriting (Category (1)).

More recent work in the context of object models has looked at semantic optimization in the presence of methods. [1] considers semantic optimization over methods based on equivalences derived from method semantics. This semantics comes from the schema and is then translated by hand into rules that can be applied to expressions written in their algebra. As with [6], these rewrites are unconditional and there is no inference (Category (1)).

Grant et al [14] employ a Datalog-based scheme for object-oriented databases that infers new integrity constraints from old explicitly declared constraints plus schema-related information about functions (methods). This work employs inference, but only of new integrity constraints (i.e., conditions that hold of data) and not conditions that hold of functions. Function conditions can be stated but not inferred, and can be used only to infer new integrity constraints (and not for example, to guard the firing of a rewrite rule). Therefore, this work comes close to falling in Category (3) because it includes both conditional rules and inference, but falls short because inference is not over function conditions.

Beeri and Kornatsky [2] describe a combinator-based algebra for representing queries and they even present several rewrite rules that are conditioned on function conditions. For example, they have several rules that only apply to expressions that contain an idempotent function. But whereas they introduce the idea of having rewrites that are conditioned on function conditions, they do not include any mechanism to reason about how these conditions can be inferred (Category (2)).

6 Conclusions

Query optimizers are difficult to build. The emergence of object databases has further complicated the task, introducing more complex data and hence more complex queries and making optimizers even more error-prone than before.

Rule-based optimizers apply software engineering techniques to query optimizer development. By modularizing the process that maps queries to plans, rules make optimizers extensible and verifiable. But rules that get expressed with code are difficult to verify. On the other hand, rewrite rules without code have limited expressive power.

This paper presents our approach to extending the expressive power of rewrite rules without compromising the ease with which they can be verified. The techniques proposed here target the expression of query transformations that are too specific to be captured with rewrite rules. (Expression of transformations that are too general to be expressed with rewrite rules is addressed in [9].) This work builds upon the foundation laid with the combinator-based algebra, KOLA. We extend KOLA's rewrite rules by introducing *conditional rewrite rules*: rewrite rules whose firing depends on the satisfaction of semantic conditions of matched expressions. We then use *inference rules* to instruct the optimizer on how to decide if these semantic conditions hold. In the spirit of KOLA, both conditional rewrite rules and inference rules are expressed without code.

This work contributes to the extensibility and verifiability of rule-based optimizers. With respect to verification, the declarative flavor of both forms of rules makes them amenable to verification with a theorem prover. This is in stark contrast to the code-based rules of existing rule-based systems such as Starburst [20] and Cascades [12] which express conditions and condition-checking with code. With respect to extensibility, the separation of a condition's inference rules from the rewrite rules that depend on them achieves a different form of extensibility than was provided by rewrite rules alone. Whereas rewrite rules make optimizers extensible by making it simple to change the potential actions taken by an optimizer, inference rules make optimizers extensible by making it simple to change the contexts under which these actions take place.

A Rewrite and Inference Rule Proof Scripts

See `ftp://ftp.cs.brown.edu/u/mfc/vldbscripts.lp`.

References

[1] K. Aberer and G. Fischer. Semantic query optimization for methods in object-oriented database systems. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the 11th International Conference on Data Engineering*, pages 70–79, Taipei, Taiwan, 1995.

[2] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In S. Abiteboul and P. C. Kanellakis, editors, *Proceedings of the Third International Conference on Database Theory*, number 470 in Lecture Notes in Computer Science, pages 72–88, Paris, France, December 1990. EATCS, Springer-Verlag.

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

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

[5] U. Chakravathy, J. Grant, and J. Minker. Semantic query optimization: Additional constraints and control strategies. In *Proceedings of Expert Database Systems Conference*, pages 259–269, Charleston, SC, April 1986.

[6] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proceedings of the 19th VLDB Conference*, pages 529–542, Dublin, Ireland, August 1993.

[7] M. Cherniack. Translating queries into combinators. Technical Report CS-96-40, Brown University Department of Computer Science, September 1996.

[8] M. Cherniack. Building query optimizers with combinators. Technical report, December 1997. Dissertation proposal.

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

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

[11] J. C. Freytag. A rule-based view of query optimization. In U. Dayal and I. Traiger, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 173–180, San Francisco, California, May 1987. ACM Special Interest Group on Management of Data, ACM Press.

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

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

[14] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proceedings of the 13th ICDE Conference*, pages 444–454, Birmingham, UK, April 1997.

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

[16] M. Hammer and S. B. Zdonik. Knowledge-based query processing. In *Proceedings of the 6th International Conference on Very Large Databases*, Montreal, Canada, October 1980. Morgan-Kaufman.

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

[18] J. King. A system for semantic query optimization in relational databases. In *Proceedings of the 7th International Conference on Very Large Databases*, pages 510–517, September 1981.

[19] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th VLDB Conference*, pages 96–107, Santiago, Chile, September 1994.

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

[21] Swedish Institute Of Computer Science. SICStus prolog user's manual. Release 3, # 5, 1996.