Rule Languages and Internal Algebras for Rule-Based Optimizers

Mitch Cherniack*
Department of Computer Science, Brown University
mfc@cs.brown.edu

Stanley B. Zdonik*
Department of Computer Science, Brown University
sbz@cs.brown.edu

Abstract

Rule-based optimizers and optimizer generators use rules to specify query transformations. Rules act directly on query representations, which typically are based on query algebras. But most algebras complicate rule formulation, and rules over these algebras must often resort to calling to externally defined bodies of code. Code makes rules difficult to formulate, prove correct and reason about, and therefore compromises the effectiveness of rule-based systems.

In this paper we present KOLA; a combinator-based algebra designed to simplify rule formulation. KOLA is not a user language, and KOLA's variable-free queries are difficult for humans to read. But KOLA is an effective internal algebra because its combinator-style makes queries manipulable and structurally revealing. As a result, rules over KOLA queries are easily expressed without the need for supplemental code. We illustrate this point, first by showing some transformations that despite their simplicity, require head and body routines when expressed over algebras that include variables. We show that these transformations are expressible without supplemental routines in KOLA. We then show complex transformations of a class of nested queries expressed over KOLA. Nested query optimizations, while having been studied before, have seriously challenged the rule-based paradigm.

1 Introduction

Rule-based optimizers and optimizer generators use rules to specify transformations of queries. Rules act directly on query representations, which typically are based on query algebras. In this paper, we argue that query algebras determine the success with which rules can express transformations. We make the argument by describing desirable transformations, and comparing rules that express these transformations over differing algebras.

*Partial support for this work was provided by the Advanced Research Projects Agency under contract N00014-91-J-4052 ARPA order 8225, and contract DAAB-07-91-C-Q518 under subcontract F41100.

1.1 The Problem

A rule describes a query transformation and the queries subject to the transformation. A *declarative* rule specifies transformations without committing to code that manipulates representations, and specifies the queries to which it applies without committing to code that makes the decision. Declarative rules grant optimizers and optimizer generators the freedom to make intelligent implementation decisions. But most existing systems limit themselves by permitting rule inputs that are not declaratively expressed.

The Starburst [20] optimizer and EXODUS [8] optimizer generator are example rule-based systems that permit rules to be supplemented with code. Code appears in two places:

- *Head Routines* (called "conditions" in [8] and "condition functions" in [20]) are invoked in the *heads* (left-hand sides) of rules and analyze query representations to decide if they should be transformed by rules.
- *Body Routines* (called "support functions" in [8] and "action routines" in [20]) are invoked in the *bodies* (right-hand sides) of rules and are used to transform query representations into alternative forms.

Code fragments are restrictive, making the quality and correctness of generated optimizers depend on the quality and correctness of included code. Code fragments also make rules difficult to formulate, prove correct and reason about. This helps to explain why for example, transformations of nested queries do not typically get implemented as instances of rules. Nested query optimization is particularly important and particularly difficult when nested queries are expressed over data with complex structure, as in *nested relational* [33], *complex object* [1] and *object-oriented* [27] databases. Such data models exacerbate both the classification and manipulation of nested queries by allowing tuples and objects to refer to sets and to each other. This potentially introduces data dependencies into queries, complicating their transformation as we later show. There has been much progress in nested query optimization [24], [17], [20], [30], [31], [12], [14]. But nested queries tend to be fairly large and rules for them tend to lack generality or have especially complex head and body routines. As a result, these transformations are rarely implemented as instances of rules, and when they are, the rules are complex and difficult to reason about.

In this paper, we show that it is the choice of query representation that determines if code fragments are required to build general and expressible rules. In particular, representations based on *variable-based* query algebras (algebras that use variables to name manipulated data) make code fragments necessary supplements to rules. Intuitively, this is because rule-based optimization has the flavor of unification [32]; a rule and a query *match* if the rule head unifies with a part of the query, and the *transformation* of a matched query is expressed by substituting for the variables that appear in the rule body. Unification has the benefit of being efficient in practice while facilitating the formulation of general rules. But unification demands that a query's representation be revealing (in how the query should be transformed) and manipulable. Variable-based representations complicate unification in both respects.

Our concern is with the expression of rules rather than the strategies for their use. The latter, while an important issue, [29] is not considered in this paper.

1.2 Our Contributions

We propose a variable-free (combinator-based) algebra, KOLA, that supports the formulation of expressive transformation rules. (A good tutorial on combinators can be found in Turner's paper [37]). Combinator-based internal algebras have been considered for queries before. In particular, [6] uses a combinator algebra to express the semantics for their query calculus and to allow category theory machinery to be used to reason about the correctness of transformations. We consider an alternative use, proposing combinators as the basis for query representations manipulated by an optimizer. We propose an alternative combinator set to that of [6] that permits smaller translations of queries [11] at the expense of allowing some redundancy. We have implemented translators into our combinator set from both OQL [9] and AQUA [25]. (See [11] for details.) This makes our work similar in spirit to the use of combinators in functional language compilers. But whereas the trend in this community is to generate a variable set of combinators on-the-fly (to improve the efficiency of graph reduction), our combinator set is fixed and therefore our queries are amenable to algebraic preprocessing.

We are able to formulate rules over KOLA-based representations of queries that are problematic to express over variable-based algebras. The impact of this is in the following areas:

Optimizer Generation: Declarative rule sets make optimizer generators more effective, as optimizers generated from input code are limited by the potential inefficiency or incorrectness of that code. Our work makes the goal of fully declarative rule sets realizable in two respects.

- *We can formulate rules without code.* We are able to formulate transformation rules that would require code if expressed over variable-based query representations. We present some rules that require code (over variable-based representations) in Section 2 and their code-free equivalents (over KOLA-based representations) in Section 3.
- *We can formulate rules for transformations that are not usually expressed with rules.* In Section 4, we present rules that transform a class of nested queries into joins. The transformation described is not new, but as far as we know has not been previously expressed in terms of a set of generally applicable and gradually transforming rules.

Optimizer Correctness: The famous "count bug" of [24] illustrates how difficult it can be to formulate correct transformations. Rule-based optimizations simplify correctness proofs of optimizations because rules are simpler to prove correct than algorithms. But this is exactly why rules should not include calls to code. As has been pointed out elsewhere ([3], [6]), combinator algebras make rule proofs easier because of the absence of variables. In fact, we have constructed a formal specification of KOLA using the Larch [19] specification tool LSL, and have constructed proofs of over 500 rules that form a pool from which a rule-based optimizer could draw. The proofs have been verified using the Larch theorem proving tool, LP. This work is described in [10].

Optimizer Extensibility: Rules that are expressed in a purely declarative fashion are easier to understand, reason about and manipulate than those that are not. Our work is a step forward in the direction of completely declarative rule sets. This means that rule sets should be easily augmented to extend the functionality of existing optimizers.

1.3 Outline of the Paper

In Section 2, we present the problems that variables introduce to the formulation of transformation rules. We use examples written in AQUA [25]. In Section 3 we present KOLA and show how KOLA simplifies the formulation of these same rules. In Section 4, we show that a KOLA-based rule language can express transformations that typically are not implemented as instances of rules (transformations of nested queries). We compare our work with related work on optimizer rule languages, combinator-based languages and nested query optimizations in Section 5, and then summarize in Section 6.

2 Variables Considered Harmful

Optimizers manipulate query representations. Effective representations facilitate manipulation, simplifying the optimizer with respect to both its implementation and formalization. Query algebras usually form the basis of query representations. Therefore it is crucial that the algebra facilitate manipulation of the representation and not be simply a means of expressing a query.

In this section, we explore the reasons why variable-based query algebras make it necessary for rules to include head and body routines. We will use AQUA [25] as a case study, although our remarks apply to other variable-based algebras (e.g. [18], [38], [36] and [23]).

2.1 Rules that Build New Functions

Anonymous functions are functions denoted without names. An expressive query algebra should permit anonymous functions to be used within a query to express what should be done with each object in a queried collection. Most object algebras provide anonymous function support using a notation borrowed from the λ-calculus[1]. But while λ-notation is straightforward for users to *write*, it is far less straightforward for optimizers to *manipulate*. We present two examples (in Sections 2.1 and 2.2 respectively) to illustrate.

Figure 1 presents two useful transformations of AQUA queries. The queries in this Figure (and Figure 2.2) use the AQUA set operators **app** and **sel** which have the semantics shown below. Note that `[` and `]` delimit object *pairs*.

$$\begin{array}{rcl} \textbf{app}\ (f)(A) & = & \{f(a) \mid a \in A\} \\ \textbf{sel}\ (p)(A) & = & \{a \mid a \in A, p(a)\} \\ \textbf{flatten}\ (A) & = & \{a \mid \exists B\ (B \in A, a \in B)\} \\ \textbf{join}\ (p,f)(\,\texttt{[}A,B\texttt{]}\,) & = & \{f(a,b) \mid a \in A, b \in B, p(a,b)\} \end{array}$$

The transformations are expressed with the notation $Q_1 \stackrel{\rightarrow}{=} Q_2$, where Q_1 and Q_2 denote equivalent queries. We assume a schema with an abstract data type (ADT), `Person`, whose interface includes `addr` (returning an `Address`), `age` (returning an integer), `child` (returning a set of `Persons` corresponding to the children of a given person), `cars` (returning a set of `Vehicles` corresponding to the cars owned by a particular person) and `grgs` (returning a set of `Addresses` corresponding to the set of garages kept by a person). The ADT, `Address` has an interface that includes `city` (returning a string). P and V denote sets of `Persons` and `Vehicles` respectively.

The transformations of Figure 1 require construction of **new** functions and predicates from the anonymous functions and predicates found in the original queries. T_1 constructs a new function, $\lambda\ (p)\ p.\texttt{addr.city}$ by *composing* the two functions used in the original query. T_2 constructs both a function ($\lambda\ (p)p.\texttt{age}$) and predicate ($\lambda\ (a)a > 25$) by *decomposing* the original predicate, $\lambda\ (p)\ p.\texttt{age} > 25$. These transformations are difficult to express over variable-based query representations because they require the optimizer to open the "black-boxes" that are λ-expressions. The manipulation must then be of the expressions that are the function and predicate bodies, requiring additional machinery to perform such tasks as

- *variable renaming* For example, "$\lambda\ (x)x.\texttt{age}$" of the first query of T_2 should be renamed to "$\lambda\ (p)p.\texttt{age}$" so that this function is recognized as a "subfunction" of "$\lambda\ (p)\ p.\texttt{age} > 25$"

[1]For example, λ-calculus notation is explicit in GOM, EQUAL and AQUA. The others use naming conventions (OFL) or variable declarations (Excess) as an alternative means of denoting a variable's meaning and scope.

T_1 : $\textbf{app}\ (\lambda\ (a)a.\texttt{city})(\textbf{app}\ (\lambda\ (p)p.\texttt{addr})(P)) \stackrel{\rightarrow}{=}$
$\textbf{app}\ (\lambda\ (p)p.\texttt{addr.city})(P)$
Return the cities inhabited by people in P.

T_2 : $\textbf{app}\ (\lambda\ (x)x.\texttt{age})(\textbf{sel}\ (\lambda\ (p)p.\texttt{age} > 25)(P)) \stackrel{\rightarrow}{=}$
$\textbf{sel}\ (\lambda\ (a)a > 25)(\textbf{app}\ (\lambda\ (p)p.\texttt{age})(P))$
Return the ages of people in P older than 25.

Figure 1: Building New Functions and Predicates

A_3 : $\textbf{app}\ (\lambda(p)\ \texttt{[}p, \textbf{sel}\ (\lambda(c)c.\texttt{age} > 25)(p.\texttt{child})\texttt{]})(P)$
Return persons in P, p paired with their children who are older than 25.

A_4 : $\textbf{app}\ (\lambda(p)\ \texttt{[}p, \textbf{sel}\ (\lambda(c)p.\texttt{age} > 25)(p.\texttt{child})\texttt{]})(P)$
Return persons in P, p paired with p's children (if p is older than 25) and with the $\oslash$ (otherwise).

Figure 2: Structurally Identical Nested Queries

- *expression composition* The transformation of T_1 requires building a new function by composing the expressions, "$a.\texttt{city}$" and "$p.\texttt{addr}$" from the original query. Expression composition requires substituting one expression for a free variable in the other expression (e.g., substituting $p.\texttt{addr}$ for a in $a.\texttt{city}$ gives $p.\texttt{addr.city}$). This substitution is not expressible using unification alone because expressions are not uniform (while some may be path expressions, others may involve pre-defined operators and functions (prefix, infix or postfix) or even queries).

This additional machinery complicates the optimizer's implementation and specification.

2.2 Rules that Manipulate Nested Queries

Queries are *nested* if they contain other queries. Figure 2 shows two nested queries expressed in AQUA. The queries are nested because the anonymous function inputs to **app** are queries (involving the operator, **sel**).

Within nested queries it is possible for subexpressions to reference free variables. Query A_4 includes the function "$\lambda\ (c)p.\texttt{age} > 25$" with a reference to the free variable p. Whether or not a variable appears free in a query can determine if a transformation is appropriate. For example, query 4 of Figure 2 is subject to a *code motion* transformation [2] which would move the predicate out of the inner query, resulting in the equivalent (but more efficient) form, $\textbf{app}\ (\lambda\ (p)f(p))(P)$ where

$$f = \textit{if}\ (p.\texttt{age} > 25)\ \textit{then}\ \texttt{[}p, p.\texttt{child}\texttt{]}\ \textit{else}\ \texttt{[}p, \oslash\texttt{]}.$$

Query A_3 of Figure 2 is structurally identical to A_4, differing only by the identifier appearing in the predicate. (A_3 checks that the age of the child c is greater than 25 rather than the age of the person p.) But A_3 is not subject to a code motion optimization. That variables appear in the query representation makes these queries structurally identical but subject to different transformations. Therefore, the rule that expresses this transformation must be supplemented with a head routine to perform *environmental analysis* to determine if variables that appear in the expression are free variables.

2.3 Problem Summary

The operation of rule-based optimizers typically resembles unification. The unification style supports the formulation of general rules (through the use of unification variables) and efficient performance. But a pure unification style requires that query representations have structure that is both revealing in how queries should be transformed, and easily manipulable. Variable-based representations do not have these properties.

- Variables are used in expressions that are function and predicate bodies. The function and predicate manipulation that is required in expressing the queries that result from transformation requires machinery above and beyond what one gets for free with unification. Therefore, this kind of manipulation requires rules with body routines.

- Variables cannot be distinguished by their structure. Because transformations can depend on which variables appear in various parts of the query (i.e., scoping), rules expressing these transformations must be supplemented with head routines.

We complete our argument by presenting a variable-free (combinator-based) algebra, and showing how the problems discussed in this section go away.

3 KOLA: A Combinator Algebra

KOLA's[2] *combinator-style* facilitates the kind of query manipulation that is difficult with variable-based algebras. KOLA has the flavor of Backus' FP [3] but unlike FP can build functions and predicates over sets. It provides for anonymous functions through *formers*; functionals that denote new functions in terms of existing ones. It also provides a set of *primitive* functions and predicates such as the identity (**id**) function and equality (**eq**) predicate, as well as functions and predicates found in ADT interfaces included in a schema (such as the `age`, `addr`, `child`, `cars` and `grgs` functions on `Person`). Variables and λ-notation are neither provided nor required to denote functions.

Tables 1 and 2 describe the operational semantics of some KOLA primitives and formers. (A formal specification of the entire algebra using Larch [19] is presented in [10]). The semantics equations show the results of invoking KOLA functions and predicates on their arguments. All functions are invoked via the infix operator, "!", while predicates are invoked with "?" (also infix). Within these equations and throughout the rest of the paper, we use variables to denote

[2]KOLA is an acronym for [K]ind [O]f [L]ike [A]QUA.

KOLA	Semantics		
id	**id** $! \; x$	$\equiv$	x
π_1	$\pi_1 \; ! \; [x, y]$	$\equiv$	x
π_2	$\pi_2 \; ! \; [x, y]$	$\equiv$	y
eq	**eq** $? \; [x, y]$	$\iff$	$x = y$
leq	**leq** $? \; [x, y]$	$\iff$	$x \leq y$
gt	**gt** $? \; [x, y]$	$\iff$	$x > y$
in	**in** $? \; [x, A]$	$\iff$	$x \in A$
$\circ$	$(f \circ g) \; ! \; x$	$\equiv$	$f \; ! \; (g \; ! \; x)$
$\langle \, \rangle$	$\langle f, g \rangle \; ! \; x$	$\equiv$	$[f \; ! \; x, g \; ! \; x]$
$\times$	$(f \times g) \; ! \; [x, y]$	$\equiv$	$[f \; ! \; x, g \; ! \; y]$
K_f	$\mathrm{K}_f \; (x) \; ! \; y$	$\equiv$	x
C_f	$\mathrm{C}_f \; (f, x) \; ! \; y$	$\equiv$	$f \; ! \; [x, y]$
con	**con** $(p, f, g) \; ! \; x$	$\equiv$	$\begin{cases} f \; ! \; x, & \text{if } p \; ? \; x \\ g \; ! \; x, & \text{else} \end{cases}$
$\oplus$	$(p \oplus f) \; ? \; x$	$\iff$	$p \; ? \; (f \; ! \; x)$
&	$(p \; \& \; q) \; ? \; x$	$\iff$	$(p \; ? \; x \; \wedge \; q \; ? \; x)$
$\mid$	$(p \mid q) \; ? \; x$	$\iff$	$(p \; ? \; x \; \vee \; q \; ? \; x)$
$^{-1}$	$(p^{-1}) \; ? \; [x, y]$	$\iff$	$p \; ? \; [y, x]$
K_p	$\mathrm{K}_p \; (b) \; ? \; x$	$\iff$	b
C_p	$\mathrm{C}_p \; (p, x) \; ? \; y$	$\iff$	$p \; ? \; [x, y]$

Table 1: Basic KOLA Combinators

arbitrary functions (f, g, h, j), predicates (p, q), objects (x, y), bools (b) and sets (A, B). Variables therefore indicate how a former is instantiated. Table 1 presents generally applicable KOLA primitives and formers, while Table 2 presents those that generate functions and predicates on sets (*queries*).

The semantics equations can be used to derive a query's "meaning". For example, the query below uses the primitive functions `city` and `addr`, $\circ$ (the composition function former), **iterate** (a set function former similar to OFL's "iterate" operator [18], and that captures both of AQUA's **app** and **sel** operators[3]) and the constant predicate former, K_p. (**iterate**'s semantics is given in Table 2 – all others are listed in Table 1). This query's "meaning" is derived by the reduction below.

$$\begin{aligned} &\textbf{iterate} \; (\mathrm{K}_p \; (T), \texttt{city} \circ \texttt{addr}) \; ! \; P \\ &\quad = \; \{(\texttt{city} \circ \texttt{addr}) \; ! \; e \mid e \in P, \; \mathrm{K}_p \; (T) \; ? \; e\} \quad (1) \\ &\quad = \; \{\texttt{city} \; ! \; (\texttt{addr} \; ! \; e) \mid e \in P, \; \mathrm{K}_p \; (T) \; ? \; e\} \quad (2) \\ &\quad = \; \{\texttt{city} \; ! \; (\texttt{addr} \; ! \; e) \mid e \in P\} \quad (3) \end{aligned}$$

Steps (1-3) of the reduction are justified by the definitions of **iterate**, $\circ$ and K_p respectively. This query is therefore a translation of the transformed query of transformation (1) of Figure 1, as the KOLA expression, "`city ! (addr !` e`)`" is equivalent to the path expression e`.addr.city`.

Table 1 is in divided into four sections. The first two sections present KOLA primitive functions and predicates respectively, which, besides **id** and **eq**, include the projection

[3]**app** (f) is equivalent to **iterate** $(\mathrm{K}_p \; (T), \; f)$ and **sel** (p) is equivalent to **iterate** $(p, \; \mathbf{id})$, where **id** and K_p are as defined in Table 1.

flat ! A	$\equiv \{x \mid x \in B, B \in A\}$	**join** (p, f) ! $[A, B]$	$\equiv \{f \,!\, [x, y] \mid x \in A, y \in B, p \,?\, [x, y]\}$
iterate (p, f) ! A	$\equiv \{f \,!\, x \mid x \in A, p \,?\, x\}$	**nest** (f, g) ! $[A, B]$	$\equiv \{[y, \{g \,!\, x \mid x \in A, f \,!\, x = y\}] \mid y \in B\}$
iter (p, f) ! $[x, B]$	$\equiv \{f \,!\, [x, y] \mid y \in B, p \,?\, [x, y]\}$	**unnest** (f, g) ! A	$\equiv \{[f \,!\, x, y] \mid x \in A, y \in (g \,!\, x)\}$

Table 2: KOLA Query Combinators

functions on pairs (π_1 and π_2), the "greater than" (**gt**) and "less than or equal" (**leq**) predicates, and the set membership predicate (**in**). (Not listed, but assumed are schema-based primitives such as those described in Section 2.1.) The third section of the table presents general purpose function formers. Besides $\circ$, these include $\langle\rangle$ (pairing functions), $\times$ (pairwise function application), K_f (constant functions), C_f (currying) and **con** (conditionals). The fourth section lists KOLA predicate formers which besides K_p, include $\oplus$ (predicate/function combiner), & and | (predicate conjunction and disjunction), $^{-1}$ (predicate inverse), K_p (constant predicates) and C_p (currying).

Table 2 presents KOLA's query formers. Besides **iterate**, these include **flat** (set flattening), **join**, **iter**, **nest** and **unnest**. **iter** is similar to **iterate**, but is invoked on pairs $[e, A]$ rather than on sets. (Binary functions and predicates are invoked over pair objects in KOLA). **iter** is suited for expressing nested queries as e can be a representation of the environment that would be implicit in a variable-based query representation. To illustrate, we trace the reduction of the "Garage Query" [28] (K_{G_1} of Figure 3); a query that associates each of a set of `Vehicles` with the set of `Addresses` where the `Vehicle` might be located. For notational simplicity, we adopt the convention that chains of function compositions ($f_1 \circ f_2 \circ \ldots \circ f_n$) are written without parentheses (exploiting associativity) and with each f_i on a separate line. Similarly, function pairs $\langle f, g\rangle$ are sometimes written with g directly below f. The semantics of K_{G_1} is shown by the reduction below, where f denotes the function, "**flat** $\circ$ **iter** $(\text{K}_p\ (T), \texttt{grgs} \circ \pi_2)$", g denotes the function "**iter** (**in** $\oplus \langle \pi_1,\ \texttt{cars} \circ \pi_2\rangle, \pi_2)$" and P_v (for `Vehicle`, v) denotes the set, $\{p \mid p \in P, v \in p.\texttt{cars}\}$.

$$\begin{aligned} &\textbf{iterate}\ (\text{K}_p\ (T),\ \langle \textbf{id},\ f \circ \langle \textbf{id},\ g \circ \langle \textbf{id},\ \text{K}_f\ (P)\rangle\rangle\rangle)\ !\ V \\ &\quad = \{[v, f\ !\ [v, g\ !\ [v, P]]] \mid v \in V,\ \text{K}_p\ (T)\ ?\ v\} && (1) \\ &\quad = \{[v, f\ !\ [v, P_v]] \mid v \in V\} && (2) \\ &\quad = \{[v, \{z \mid z \in p.\texttt{grgs},\ p \in P_v\}] \mid v \in V\} && (3) \end{aligned}$$

Step (1) follows from the definitions of **iterate**, $\langle\ \rangle$, **id**, $\circ$ and K_f. Step (2) follows as $g\ !\ [v, P]$

$$\begin{aligned} &= \textbf{iter}\ (\textbf{in} \oplus \langle \pi_1,\ \texttt{cars} \circ \pi_2\rangle, \pi_2)\ !\ [v, P] \\ &= \{\pi_2\ !\ [v, p] \mid p \in P, \textbf{in} \oplus \langle \pi_1,\ \texttt{cars} \circ \pi_2\rangle\ ?\ [v, p]\} \\ &= \{p \mid p \in P, \textbf{in}\ ?\ [\pi_1\ !\ [v, p], (\texttt{cars} \circ \pi_2)\ !\ [v, p]]\} \\ &= \{p \mid p \in P,\ v \in p.\texttt{cars}\} \ = \ P_v \end{aligned}$$

$$\begin{aligned} K_{G_1}:\ &\textbf{iterate}\ (\text{K}_p\ (T),\ \langle \textbf{id}, \\ &\qquad \textbf{flat} \circ \\ &\qquad \textbf{iter}\ (\text{K}_p\ (T), \texttt{grgs} \circ \pi_2) \circ \\ &\qquad \langle \textbf{id},\ \textbf{iter}\ (\textbf{in} \oplus \langle \pi_1,\ \texttt{cars} \circ \pi_2\rangle, \pi_2) \circ \\ &\qquad\quad \langle \textbf{id},\ \text{K}_f\ (P)\rangle\rangle\rangle)\ !\ V \end{aligned}$$

$$\begin{aligned} K_{G_2}:\ &\textbf{nest}\ (\pi_1,\ \pi_2) \circ \\ &(\textbf{unnest}\ (\pi_1,\ \pi_2) \times \textbf{id}) \circ \\ &\langle \textbf{join}\ (\textbf{in} \oplus (\textbf{id}\ \times \texttt{cars}),\ \textbf{id}\ \times \texttt{grgs}),\ \pi_1\rangle\ !\ [V, P] \end{aligned}$$

Figure 3: Two Equivalent Versions of the "Garage Query"

while step (3) follows as $f\ !\ [v, P_v]$

$$\begin{aligned} &= \textbf{flat} \circ \textbf{iter}\ (\text{K}_p\ (T), \texttt{grgs} \circ \pi_2)\ !\ [v, P_v] \\ &= \textbf{flat}\ !\ \{(\texttt{grgs} \circ \pi_2)\ !\ [v, p] \mid p \in P_v, \text{K}_p\ (T)\ ?\ [v, p]\}) \\ &= \textbf{flat}\ !\ \{p.\texttt{grgs} \mid p \in P_v\} \\ &= \{x \mid x \in p.\texttt{grgs},\ p \in P_v\}. \end{aligned}$$

Steps (2) and (3) of the reduction have g and f each being evaluated with respect to an explicit environment (pair) containing v. These environments are created when the identity function is invoked as part of the application of functions $\langle \textbf{id},\ \text{K}_f\ (P)\rangle$ (which creates an environment for g) and $\langle \textbf{id},\ g \circ \text{K}_f\ (P)\rangle$ (which creates an environment for f).

The combinator **nest** forms a function that is invoked on pairs of sets. A typical use of **nest** involves nesting a join of two sets A and B, pairing each element $a \in A$ with the subset of B containing all elements that satisfy the join predicate with a. To ensure that the cardinality of the result is the same as that of A, many algebras introduce an outer join operator that associates NULLs with elements of A which never satisfy the join predicate (e.g. [14]). That is, NULLs preserve values of A that are needed in the nesting but are lost by the join. Our version of **nest** allows us to avoid NULL values by instead making the nesting of a set (the first argument to nest) relative to a second set (the second argument to nest). Rather than associating NULLs with particular elements of A as a result of the join, we associate the empty set with these elements as a result of the nest. We avoid "losing" join values by making A a second argument to **nest**. as in

$$\textbf{nest}\ (\pi_1, \pi_2)\ !\ [\textbf{join}\ (p, \textbf{id})\ !\ [A, B], A].$$

The reader can verify, by reducing this expression according to the rules of Table 2 that every element of A is represented

$$\begin{array}{ll}
\textbf{iterate}\ (\mathtt{K}_p\ (T), \mathtt{city}) \circ \textbf{iterate}\ (\mathtt{K}_p\ (T), \mathtt{addr})\ !\ P & \stackrel{11}{\rightarrow} \\
\textbf{iterate}\ (\mathtt{K}_p\ (T)\ \&\ (\mathtt{K}_p\ (T) \oplus \mathtt{addr}), \mathtt{city} \circ \mathtt{addr})\ !\ P & \stackrel{6}{\rightarrow} \\
\textbf{iterate}\ (\mathtt{K}_p\ (T)\ \&\ \mathtt{K}_p\ (T), \mathtt{city} \circ \mathtt{addr})\ !\ P & \stackrel{5}{\rightarrow} \\
\textbf{iterate}\ (\mathtt{K}_p\ (T), \mathtt{city} \circ \mathtt{addr})\ !\ P &
\end{array}$$

$$\begin{array}{ll}
\textbf{iterate}\ (\mathtt{K}_p\ (T), \mathtt{age}) \circ & \\
\quad \textbf{iterate}\ (\textbf{gt} \oplus (\langle \mathtt{age},\ \mathtt{K}_f\ (25)\rangle), \textbf{id})\ !\ P & \stackrel{11,\ 6,\ 5,\ 1}{\rightarrow} \\
\textbf{iterate}\ (\textbf{gt} \oplus (\langle \mathtt{age},\ \mathtt{K}_f\ (25)\rangle), \mathtt{age})\ !\ P & \stackrel{13,\ 7}{\rightarrow} \\
\textbf{iterate}\ (\mathtt{C}_p\ (\textbf{leq}, 25) \oplus \mathtt{age}, \mathtt{age}))\ !\ P & \stackrel{12^{-1}}{\rightarrow} \\
\textbf{iterate}\ (\mathtt{C}_p\ (\textbf{leq}, 25), \textbf{id}) \circ \textbf{iterate}\ (\mathtt{K}_p\ (T), \mathtt{age})\ !\ P &
\end{array}$$

Figure 4: KOLA Transformations T_1^K and T_2^K

$$\begin{array}{rclcrclc}
f \circ \textbf{id} & \stackrel{\rightarrow}{=} & f & (1) & \textbf{id} \circ f & \stackrel{\rightarrow}{=} & f & (2) \\
p \oplus \textbf{id} & \stackrel{\rightarrow}{=} & p & (3) & \langle \pi_1,\ \pi_2 \rangle & \stackrel{\rightarrow}{=} & \textbf{id} & (4) \\
\mathtt{K}_p\ (T)\ \&\ p & \stackrel{\rightarrow}{=} & p & (5) & \mathtt{K}_p\ (b) \oplus f & \stackrel{\rightarrow}{=} & \mathtt{K}_p\ (b) & (6) \\
\textbf{gt}^{-1} & \stackrel{\rightarrow}{=} & \textbf{leq} & (7) & \mathtt{K}_f\ (k) \circ f & \stackrel{\rightarrow}{=} & \mathtt{K}_f\ (k) & (8) \\
\pi_1 \circ \langle f,\ g \rangle & \stackrel{\rightarrow}{=} & f & (9) & \pi_2 \circ \langle f,\ g \rangle & \stackrel{\rightarrow}{=} & g & (10)
\end{array}$$

$$\textbf{iterate}\ (p, f) \circ \textbf{iterate}\ (q, g) \stackrel{\rightarrow}{=} \textbf{iterate}\ (q\ \&\ (p \oplus g), f \circ g) \quad (11)$$
$$\textbf{iterate}\ (p, \textbf{id}) \circ \textbf{iterate}\ (\mathtt{K}_p\ (T), f) \stackrel{\rightarrow}{=} \textbf{iterate}\ (p \oplus f, f) \quad (12)$$

$$\begin{array}{rclc}
p \oplus \langle f,\ \mathtt{K}_f (k) \rangle & \stackrel{\rightarrow}{=} & \mathtt{C}_p\ (p^{-1}, k) \oplus f & (13) \\
p \oplus (f \circ g) & \stackrel{\rightarrow}{=} & (p \oplus f) \oplus g & (14) \\
\textbf{iter}\ (p \oplus \pi_1, \pi_2) & \stackrel{\rightarrow}{=} & \textbf{con}\ (p \oplus \pi_1, \pi_2, \mathtt{K}_f\ (\oslash)) & (15) \\
\textbf{con}\ (p, f, g) \circ h & \stackrel{\rightarrow}{=} & \textbf{con}\ (p \oplus h, f \circ h, g \circ h) & (16)
\end{array}$$

Figure 5: Rules for Figures 4 and 6

in the result. The second "garage query" (K_{G_2} of Figure 3) invokes **nest** on the result of **join**. We show its equivalence to K_{G_1} in Section 4.

3.1 Rules that Build New Functions

Figure 4 presents step-by-step transformations of KOLA queries equivalent to the AQUA transformations of Figure 1. Each step in the transformation is justified by a rule from Figure 5. We use the notation $\stackrel{i_1, \ldots, i_n}{\rightarrow}$ to indicate the rules used to justify a step in the transformation. Rule references of the form i^{-1} are "right-to-left" interpretations of rule i.

These transformations required head and body routines when expressed over AQUA queries because sophisticated manipulation of function and predicate bodies was required. KOLA's many function and predicate formers provide a catalog of ways to recognize and build complex functions. Transformation T_1^K uses the $\circ$ function former to combine two existing functions (rule 11). Transformation T_2^K decomposes the predicate and function subparts of a predicate by separating the arguments to the $\oplus$ predicate former (rule 12^{-1}). Formers simplify the optimizer's implementation (which requires no extra machinery such as variable renaming) and formalization (which can be based on a set of declarative rules such as those of Figure 5).

3.2 Rules that Manipulate Nested Queries

The nested AQUA queries of Figure 2 are structurally identical to one another but only one is transformable using code motion. The applicability of the code motion rule depends on the freeness of a variable appearing in a subexpression. Therefore, the routine that performs this transformation over a variable-based representation must perform environmental analysis.

The KOLA versions of these queries are both of the form,

$$\begin{array}{l}
\textbf{iterate}\ (\mathtt{K}_p\ (T), \langle \textbf{id}, \\
\qquad \textbf{iter}\ (\textbf{gt} \oplus \langle \mathtt{age} \circ \overline{f},\ \mathtt{K}_f\ (25)\rangle, \pi_2) \circ \\
\qquad\quad \langle \textbf{id},\ \mathtt{child} \rangle\rangle)\ !\ P
\end{array}$$

$$\begin{array}{ll}
\textbf{iter}\ (\textbf{gt} \oplus \langle \mathtt{age} \circ \pi_1,\ \mathtt{K}_f\ (25)\rangle, \pi_2) \circ \langle \textbf{id},\ \mathtt{child} \rangle & \stackrel{13,\ 7}{\rightarrow} \\
\textbf{iter}\ (\mathtt{C}_p\ (\textbf{leq}, 25) \oplus (\mathtt{age} \circ \pi_1), \pi_2) \circ \langle \textbf{id},\ \mathtt{child} \rangle & \stackrel{14}{\rightarrow} \\
\textbf{iter}\ ((\mathtt{C}_p\ (\textbf{leq}, 25) \oplus \mathtt{age}) \oplus \pi_1, \pi_2) \circ \langle \textbf{id},\ \mathtt{child} \rangle & \stackrel{15}{\rightarrow} \\
\textbf{con}\ ((\mathtt{C}_p\ (\textbf{leq}, 25) \oplus \mathtt{age}) \oplus \pi_1, \pi_2, \mathtt{K}_f\ (\oslash)) \circ \langle \textbf{id},\ \mathtt{child} \rangle & \\
\quad \ldots \stackrel{16}{\rightarrow}, \stackrel{10}{\rightarrow}, \stackrel{8}{\rightarrow}, \stackrel{14^{-1}}{\rightarrow}, \stackrel{9}{\rightarrow}, \stackrel{1}{\rightarrow} \ldots & \\
\textbf{con}\ (\mathtt{C}_p\ (\textbf{leq}, 25) \oplus \mathtt{age}, \mathtt{child}, \mathtt{K}_f\ (\oslash)) &
\end{array}$$

Figure 6: Rule-based Transformation of Query 4_k

but differ by what is $\overline{f}$: KOLA's version of A_3 (hereafter referred to as query K_3), has $\overline{f}$ as π_2, whereas K_4 has $\overline{f}$ as π_1. Thus, the KOLA queries are structurally *similar* to one another, but not identical. The difference is sufficient to determine that a code motion transformation only applies to the translation of K_4.

In Figure 6, we present the stepwise transformation of K_4. Specifically, we show how the function argument to **iterate**,

$$\textbf{iter}\ (\textbf{gt} \oplus \langle \mathtt{age} \circ \pi_1,\ \mathtt{K}_f\ (25)\rangle, \pi_2) \circ \langle \textbf{id},\ \mathtt{child} \rangle$$

is transformed to remove the unnecessary looping operator, **iter**. (We omit some steps in the interests of space.) K_3 would be transformed by similar steps, but after having been transformed according to rule (14), its predicate argument to **iter** would have the form, $p \oplus \pi_2$ ($p = \mathtt{C}_p\ (\textbf{leq}, 25) \oplus \mathtt{age}$) making it unaffected by rule (15).

To summarize, we have shown that the effectiveness of a rule-based optimizer depends in part on the underlying query algebra. An algebra is a means of **representing** queries and not just a medium for expressing them. As the basis of query representations, algebras should facilitate the query analysis and manipulation performed by an optimizer. But variables complicate transformation, demanding that additional machinery be available to build new functions and examine environments. We have introduced our combinator-based algebra, KOLA. KOLA is a useful basis for query representations because both analysis and manipulation of KOLA queries can be expressed in terms of declarative rules and without code. This simplifies an optimizer's implementation and formalization, helping to ensure that it is built correctly.

4 Transforming Hidden Join Queries

We have shown that KOLA's combinator-based denotations of functions and predicates make it possible to express optimization rules without the need for head and body routines. In this section, we consider rules for a class of nested query transformations that further demonstrate the expressive power we get from a KOLA-based rule language. A great deal of research has been done in nested query optimization, but typically this research makes it into practice with complex rules that are difficult to formalize and reason about (e.g., [12] and [20]) or with transformations expressed informally over query languages (e.g., [24, 17, 31]).

4.1 Hidden Join Queries

The class of queries we consider are *hidden joins*; nested queries that (like join queries) pair objects that are taken from two sets and that satisfy some relationship. Because of their potentially deep nesting, it is not immediately apparent that hidden joins can be transformed into explicit joins. We propose a five-step strategy for "untangling" hidden join queries into their join equivalents, complete with rule sets used at each step. The rules we use for these transformations are generally applicable and perform the optimization in gradual steps, unlike the monolithic and overly specific rules that sometimes appear in the literature [12]. We describe and illustrate our technique showing how to transform from one "Garage Query" (K_{G_1}) to the other (K_{G_2}).

AQUA's hidden join queries are of the form:

$$\textbf{app}\ (\lambda\ (a)\ [j\ (a),\ g_1\ (g_2\ (\ldots\ (g_n\ (B))\ \ \ldots))])\ (A)$$

where j is any function and each g_i is a function that invokes a query, as in, **app** $(\ldots)$, **sel** $(\ldots)$, **flatten** (**app** $(\ldots)$) or **flatten** (**sel** $(\ldots)$). KOLA translations of these queries are of the form shown in Figure 7, where j is any function, each h_i is either **flat** or **id**, and each g_i is **iter** (p_i, f_i) for some function f_i and predicate p_i. For K_{G_1} of Figure 3, we have $n = 2$, $j = \mathbf{id}$, $h_1 = $ **flat**, $p_1 = \mathtt{K}_p\ (T)$, $f_1 = \mathtt{grgs} \circ \pi_2$, $h_2 = \mathbf{id}^4$, $p_2 = $ **in** $\oplus \langle \pi_1,\ \mathtt{cars} \circ \pi_2 \rangle$, $f_2 = \pi_2$, $B\ =\ P$ and $A\ =\ V$.

The optimization of hidden joins involves transforming them into nestings of explicit joins, as in the KOLA query K_{G_2} of Figure 3. This kind of optimization may be advantageous because of the variety of implementation techniques known for performing nestings of joins [24]. But hidden joins are

$$\begin{aligned}\textbf{iterate}\ (\mathtt{K}_p\ (T),\ \langle j,\ h_1 \circ g_1 \circ\ & \\ &\langle \mathbf{id}, h_2 \circ g_2 \circ \ldots \circ \\ &\quad\langle \mathbf{id}, h_n \circ g_n \circ \\ &\qquad \langle \mathbf{id},\ \mathtt{K}_f\ (B) \rangle\rangle \ldots \rangle\rangle)\ \ !\ A\end{aligned}$$

Figure 7: KOLA Hidden Join Queries

difficult to transform with rules because nesting can occur to any degree (i.e., the value of n above is unbounded). Rules that express the optimization monolithically (as in [12]) must analyze the query using complex head routines that delve to any level of nesting, to see if the query is of the desired form. (The query is not of the desired form for example if the query that is the function instantiating **iterate** is invoked on a set derived from a rather than the globally named set B). Our techniques use multiple smaller rules to gradually transform the query to its desired form. As we will see, the rules chosen simplify queries to the point where it is straightforward to decide if the query is transformable into a nest of a join. In cases where this transformation is inapplicable, the query has still been simplified enough that other appropriate strategies can be simply considered.

Below we present a strategy and associated rule set for converting hidden join queries into queries with explicit joins. Our strategy consists of five steps, where each step uses a small rule set to guide the transformation of its input query. We summarize these steps in terms of the actions that are taken on parse tree representations of hidden join queries.

1. *Break up complex* **iterate** into multiple, smaller **iterate**'s.
2. *Bottom-Out* the parse tree with a nest of a join.
3. *Pull up* **nest** to the top of the query tree
4. *Pull up* **unnest***s* to the top of the query tree (below **nest**).
5. *Absorb into* **join**, the **iterate** operations above it.

We consider each step in detail below, by describing the general idea behind each step, the general form of the query that results from the transformation and the effect of each step on the "Garage query". The rules used at each stage are listed in Figures 5 and 8. All of these rules have been proven correct with proofs verified by the Larch theorem prover, LP [19].

Step 1: Break up complex iterate This step has the effect of breaking up the query from the monolithic form,

$$\textbf{iterate}\ (\mathtt{K}_p\ (T),\ \langle F,\ G \rangle)\ !\ A,$$

where G is potentially very large, into a composition chain of **iterate** operations. Rules 17, and 18 of Figure 8, and rule 4 of Figure 5 are used to reduce the initial query into a query of the form,

$$\begin{aligned}&\textbf{iterate}\ (\mathtt{K}_p\ (T),\ \langle j \circ \pi_1,\ \pi_2 \rangle) \circ \\ &\textbf{iterate}\ (\mathtt{K}_p\ (T),\ \overline{f_1})\ \circ\ \textbf{iterate}\ (\mathtt{K}_p\ (T),\ \overline{g_1})\ \circ\ \ldots \\ &\textbf{iterate}\ (\mathtt{K}_p\ (T),\ \langle \mathbf{id},\ \mathtt{K}_f\ (B) \rangle)\ \ !\ A\end{aligned}$$

[4] Note that the association of **id** with h_2 follows after applying rule 2^{-1} of Figure 5.

where $\overline{f_i} = \langle \pi_1,\ \textbf{flat} \circ \pi_2 \rangle$[5] and $\overline{g_i} = \langle \pi_1,\ \textbf{iter}\,(p_i,\ f_i)\rangle$. For example, applying these transformations to K_{G_1} of Figure 3 leaves $K_{G_{1a}}$ =

$$\begin{array}{l} \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{flat}\ \circ \pi_2\rangle) \circ \\ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(\mathtt{K}_p\ (T),\ \mathtt{grgs}\ \circ \pi_2)\rangle) \circ \\ \ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(\textbf{in} \oplus \langle \pi_1,\ \mathtt{cars}\ \circ \pi_2\rangle,\ \pi_2)\rangle) \circ \\ \ \ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \mathbf{id},\ \mathtt{K}_f\ (P)\rangle)\ !\ V \end{array}$$

Because in this example, j is **id**, the first function in the composition chain reduces to **id** (by rule 18) and then is eliminated (by rule 2).

Step 2: Bottom-Out In this step we convert the expression, **iterate** $(\mathtt{K}_p\ (T), \langle \mathbf{id},\ \mathtt{K}_f\ (B)\rangle)\ !\ A$, (which occurs at the bottom of the query tree) into a nest of a join. Rule 19 of Figure 8 is used to reduce the query resulting from the transformations of Step 1, into a query of the form,

$$\begin{array}{l} \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle j \circ \pi_1,\ \pi_2\rangle) \circ \\ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \overline{f_1})\ \circ\ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \overline{g_1})\ \circ\ \ldots \\ \ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \overline{f_n})\ \circ\ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \overline{g_n}) \circ \\ \ \ \ \textbf{nest}\,(\pi_1,\ \pi_2) \circ \\ \ \ \ \ \langle \textbf{join}\,(\mathtt{K}_p\ (T),\ \mathbf{id}),\ \pi_1\rangle\ !\ [A, B] \end{array}$$

Applied to $K_{G_{1a}}$, this transformation results in $K_{G_{1b}}$ =

$$\begin{array}{l} \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{flat}\ \circ \pi_2\rangle) \circ \\ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(\mathtt{K}_p\ (T),\ \mathtt{grgs}\ \circ \pi_2)\rangle) \circ \\ \ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(\textbf{in} \oplus \langle \pi_1,\ \mathtt{cars}\ \circ \pi_2\rangle,\ \pi_2)\rangle) \circ \\ \ \ \ \textbf{nest}\,(\pi_1,\ \pi_2) \circ \\ \ \ \ \ \langle \textbf{join}\,(\mathtt{K}_p\ (T),\ \mathbf{id}),\ \pi_1\rangle\ !\ [V, P] \end{array}$$

Step 3: Pull Up nest In this step, **nest** is pulled from the bottom of the query tree to the top. Rules 20 and 21 of Figure 8 reduce the query resulting from Step 2, into a query of the form,

$$\begin{array}{l} \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle j \circ \pi_1,\ \pi_2\rangle) \circ \\ \ \textbf{nest}\,(\pi_1,\ \pi_2) \circ \\ \ \ \overline{h_1}\ \circ\ (\textbf{iterate}\,(p_1,\ \langle \pi_1,\ f_1\rangle) \times \mathbf{id})\ \circ\ \ldots\ \circ \\ \ \ \ \overline{h_n}\ \circ\ (\textbf{iterate}\,(p_n,\ \langle \pi_1,\ f_n\rangle) \times \mathbf{id}) \circ \\ \ \ \ \ \langle \textbf{join}\,(\mathtt{K}_p\ (T),\ \mathbf{id}),\ \pi_1\rangle\ !\ [A, B] \end{array}$$

where each $\overline{h_i}$ is either **unnest** $(\pi_1,\ \pi_2)$ or **id** (in which case it "drops out" by rule 2 of Figure 5). (If j is **id**, **nest** will appear at the top of the query tree after this step.) Applied to $K_{G_{1b}}$, this transformation results in $K_{G_{1c}}$ =

$$\begin{array}{l} \textbf{nest}\,(\pi_1,\ \pi_2) \circ \\ \ (\textbf{unnest}\,(\pi_1,\ \pi_2) \times \mathbf{id}) \circ \\ \ \ (\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \mathtt{grgs}\ \circ \pi_2\rangle) \times \mathbf{id}) \circ \\ \ \ \ (\textbf{iterate}\,(\textbf{in} \oplus \langle \pi_1,\ \mathtt{cars}\ \circ \pi_2\rangle,\ \mathbf{id}) \times \mathbf{id}) \circ \\ \ \ \ \ \langle \textbf{join}\,(\mathtt{K}_p\ (T),\ \mathbf{id}),\ \pi_1\rangle\ !\ [V, P] \end{array}$$

[5] $\overline{f_i}$ could also be of the form, $\langle \pi_1,\ \textbf{id}\ \circ\ \pi_2\rangle\ =\ \textbf{id}$ (by rules 2 and 4 of Figure 5), in which case **iterate** $(\mathtt{K}_p\ (T),\ \overline{f_i})$ "drops out" of the query by rules 18 (Figure 8) and 2 (Figure 5).

17. $\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle j,\ (g \circ \textbf{iter}\,(p,\ f) \circ \langle \mathbf{id},\ h\rangle)\rangle)\ \stackrel{\rightarrow}{=}$
$$\begin{array}{l} \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle (j\ \circ\ \pi_1),\ \pi_2\rangle) \circ \\ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ (g\ \circ\ \pi_2)\rangle) \circ \\ \ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(p,\ f)\rangle) \circ \\ \ \ \ \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \mathbf{id},\ h\rangle) \end{array}$$

18. $\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \mathbf{id})\ \stackrel{\rightarrow}{=}\ \mathbf{id}$

19. $\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \mathbf{id},\ \mathtt{K}_f\ (B)\rangle)\ !\ A\ \stackrel{\rightarrow}{=}$
$\textbf{nest}\,(\pi_1,\ \pi_2)\ \circ\ \langle \textbf{join}\,(\mathtt{K}_p\ (T),\ \mathbf{id}),\ \pi_1\rangle\ !\ [A, B]$

20. $\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(p,\ f)\rangle) \circ \textbf{nest}\,(\pi_1,\ \pi_2)\ \stackrel{\rightarrow}{=}$
$\textbf{nest}\,(\pi_1,\ \pi_2)\ \circ\ (\textbf{iterate}\,(p,\ \langle \pi_1,\ f\rangle) \times \mathbf{id})$

21. $\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{flat}\ \circ \pi_2\rangle) \circ \textbf{nest}\,(\pi_1,\ \pi_2)\ \stackrel{\rightarrow}{=}$
$\textbf{nest}\,(\pi_1,\ \pi_2)\ \circ\ (\textbf{unnest}\,(\pi_1,\ \pi_2) \times \mathbf{id})$

22. $(\textbf{iterate}\,(p,\ \langle \pi_1,\ f\rangle) \times \mathbf{id}) \circ (\textbf{unnest}\,(\pi_1,\ \pi_2) \times \mathbf{id})\ \stackrel{\rightarrow}{=}$
$$\begin{array}{l} (\textbf{unnest}\,(\pi_1,\ \pi_2)\ \times\ \mathbf{id}) \circ \\ \ (\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{iter}\,(p,\ f)\rangle)\ \times\ \mathbf{id}) \end{array}$$

23. $(\textbf{unnest}\,(\pi_1,\ \pi_2) \times \mathbf{id}) \circ (\textbf{unnest}\,(\pi_1,\ \pi_2) \times \mathbf{id})\ \stackrel{\rightarrow}{=}$
$$\begin{array}{l} (\textbf{unnest}\,(\pi_1,\ \pi_2) \times \mathbf{id}) \circ \\ \ (\textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle \pi_1,\ \textbf{flat}\ \circ \pi_2\rangle) \times \mathbf{id}) \end{array}$$

24. $(\textbf{iterate}\,(p,\ f) \times \mathbf{id}) \circ \langle \textbf{join}\,(q,\ g),\ \pi_1\rangle\ \stackrel{\rightarrow}{=}$
$\langle \textbf{join}\,(q\ \&\ (p \oplus g),\ f \circ g),\ \pi_1\rangle$

Figure 8: Rules Used to Optimize Hidden Joins

Step 4: Pull Up unnest In this step, all **unnest** operations appearing in the parse tree are pulled up to the top, just below the **nest** operator. Naturally, if the only instance of **unnest** is situated immediately following **nest**, this step need not be performed. Rules 22 and 23 of Figure 8 are used to reduce the query resulting from the transformation of Step 3, into a query of the form,

$$\begin{array}{l} \textbf{iterate}\,(\mathtt{K}_p\ (T),\ \langle j \circ \pi_1,\ \pi_2\rangle)\ \circ\ \textbf{nest}\,(\pi_1,\ \pi_2) \circ \\ (j_0)\ \circ\ (\overline{j_1}\ \times\ \mathbf{id})\ \circ\ \ldots\ \circ\ (\overline{j_n}\ \times\ \mathbf{id}) \circ \\ \langle \textbf{join}\,(\mathtt{K}_p\ (T),\ \mathbf{id}),\ \pi_1\rangle\ !\ [A, B] \end{array}$$

where j_0 is (**unnest** $(\pi_1,\ \pi_2)\ \times\ \mathbf{id}$) or **id** (in which case it "drops out"), and j_i $(i > 0)$ is **iter** $(\mathtt{K}_p\ (T), \langle \pi_1,\ \overline{h_i}\rangle)$ where $\overline{h_i}$ is either **iterate** $(p,\ f)$ (for some predicate p and function f) or **flat** $\circ\ \pi_2$. Query $K_{G_{1c}}$ is unaffected by this step because **unnest** appears just once in the parse tree just following **nest**.

Step 5: Absorb into join In this step, the **join** operation found at the bottom of the query tree is combined with the **iterate** operations above it, thereby removing the **iterate** operations in favor of a join with a potentially complex function and predicate. Rule 24 of Figure 8 is used to reduce the query resulting from the transformation of Step 4, into a query of the form,

$$\textbf{iterate}\ (\mathrm{K}_p\ (T),\ \langle j \circ \pi_1,\ \pi_2\rangle) \circ$$
$$\textbf{nest}\ (\pi_1,\ \pi_2)\ \circ\ j_0\ \circ\ \langle \textbf{join}\ (\overline{p},\ \overline{h}),\ \pi_1\rangle\ \ !\ \ [A, B]$$

where $\overline{h}$ is any function and $\overline{p}$ is any predicate. Applied to $K_{G_{1c}}$ this transformation produces query K_{G_2} of Figure 3.

4.2 Discussion

The hidden join transformation illustrates the advantages of using a combinator-based query representation for developing rules. This transformation is far more difficult to express over variable-based query representations because the problems that variables introduce into the expression of rules are exaggerated when these rules must express complex transformations. To illustrate, we consider how the hidden join optimization might be expressed over an AQUA-based query representation. One could try and do so with multiple, simple rules as we have done, but this is complicated by the same problems described earlier. For example, Step 1 of our hidden join strategy requires recognizing that a query is of the form, "$\textbf{iterate}\ (\mathrm{K}_p\ (T),\ \langle f,\ g\rangle)\ !\ A$." The equivalent AQUA query is of the form, "$\textbf{app}\ (\lambda\ (a)\ [e_1,\ e_2])\ (A)$", where both e_1 and e_2 have occurences of a. Of course, e_1 and e_2 can be arbitrarily complex expressions. Recognizing that a occurs in these expressions would then require a complex head routine.

An alternative is to express the hidden join transformation in terms of a single complex monolithic rule. (This is the approach taken to express transformations in [12]). Such rules are problematic for two reasons.

Complex Rules Need Complex Head and Body Routines. This is not surprising, given the arguments presented in Section 2. However, we can appreciate how complex the routines can be by considering how a monolithic rule would express the hidden join optimization. In order to fire this rule on a query, matching must determine that the function which is applied to the elements of A is a query over a set, B. The reference to B can be arbitrarily deeply nested within the query, meaning that the *level* at which it appears in the representation parse tree is unbounded. Therefore, the structural matching provided by unification must be insufficient to decide that the hidden join rule is applicable to a query. Rather, a head routine is neccessary to perform the "dive" into the query tree, sinking as many levels as is required to decide whether or not the rule should be fired.

Complex Rules Do Not Simplify Queries. The complex head routines that would be required to express this rule monolithically are especially troublesome when one remembers that most often, rules are not applicable to queries. (The rule set used to transform the query will invariably be a small subset of the entire rule set used by the optimizer). In deciding that the monolithically expressed hidden join rule is not applicable to a given query, the query is *not* simplified in any way. Thus, the resources required in attempting to match a rule to a query do not bring the query much closer to being optimized (except that there is one less rule to try).

We believe the approach of using multiple simple rules to transform the query to be especially promising, because many of these rules *simplify* the query in such a way that alternative strategies are easily considered. (This was the case for queries K_3 and K_4 (from Section 3.2). Both queries were subject to the same initial transformations. These simplified the query to a point where it was possible to determine if code motion transformations were applicable.) Similarly, the first step of the "garage query" transformation simplified the initial query by breaking up the large function applied to each vehicle, and replacing the query with a composition chain of simpler functions. If the function applied to each vehicle, v had not been a query on P but instead a query on some set attribute of v (such as $v.\mathbf{drivers}$), this first step would still have simplified the query by breaking up its monolithic function into simpler subparts. Step 2 would then be quickly recognized as inapplicable, and an alternative strategy could be considered.

While the advantages of combinator algebras have been spelled out in some detail, their drawbacks must also be considered:

Expressibility: It is not obvious at first glance that a combinator algebra such as KOLA is expressive enough to serve as an intermediate form for such expressive query languages as OQL. However, we have designed, implemented and verified translators from both OQL and AQUA to KOLA, demonstrating KOLA's expressive power [11].[6] Translation (which proceeds in similar fashion to that described in [6] and [13]) relies on combinators that permit generation of explicit environments ($\mathbf{id}$ and $\langle\ \rangle$), and access to those environments (π_1, π_2 and $\circ$). For iteration, KOLA provides the environment accessing former, **iter** (which generalizes the "pairwith" combinator of [6]).

The other issue concerns the expressibility of rules, especially given our avoidance of head and tail routines. Some transformations are only valid or appropriate provided that certain conditions hold. We permit preconditions within the KOLA rule language (for details see [10]), but they are expressed as *attributes* whose values are determined not with code, but with annotations and additional rules. For example, a function is *injective* if it results in unequal results when invoked on unequal objects (as in a key). We permit rules such as

$$injective\ (f) ::$$
$$((\textbf{iterate}\ (\mathrm{K}_p\ (T),\ f)\ !\ A)\ \cap\ (\textbf{iterate}\ (\mathrm{K}_p\ (T),\ f)\ !\ B))\ \stackrel{\rightarrow}{=}$$
$$\textbf{iterate}\ (\mathrm{K}_p\ (T),\ f)\ !\ (A\ \cap\ B)$$

which says that provided a function is *injective* it can be applied before or after two sets are intersected. As well, rules

[6] Both translators are confined to queries on sets involving objects and tuples, as bags and lists are not yet accounted for in the algebra.

such as

$$injective\ (f) \wedge injective\ (g) \implies injective\ (f \circ g)$$

indicate (without code) how conditions can be inferred of complex functions and predicates. KOLA preconditions add expressibility to the rule language, as rules can depend on whether, for example, a function is a key or is functionally dependent on another function. These preconditions can be exploited and inferred without calls to code.

Complexity: Combinators make queries "larger". Intuitively, this is because variables, which occupy one node of a parse tree) must be replaced by functions, which can occupy several nodes. But we show in [11] that the complexity of translated queries are $O\ (mn)$ in the size of the input, where size is measured in parse tree nodes, n is the number of nodes in the original query, and m is the maximum number of variables appearing simultaneously in the original query's environment (i.e., the "degree of nesting"). m is typically small (e.g., < 10) as queries with large values of m are difficult to conceive and formulate. In our experience, we have found that translated queries are less than twice the size of the queries they translate.

In simplifying rules, we have also increased the size of the rule set. For example, we have introduced 24 KOLA rules to replace the four transformations presented in this paper. However, most of the rules introduced (e.g. 1-11, 13-14, 16 and 18) have general applicability beyond the transformations described here, and therefore we speculate that the rule set will not increase in size by a large factor. But to handle the still large set of rules, as well as to account for rules that are bidirectional (rules 2, 12 and 14 were all used in a "right-to-left" manner in this paper), we are developing a language, COKO[7] with which to express *rule blocks*; sets of rules that are used together, together with strategies for their firing. Rule blocks correspond to "conceptual transformations" which are transformations that are small enough to be thought of as individual transformations, but too complex to be expressed with a single rule. Example rule blocks include "push selects past joins" and "convert predicates to CNF", as well as each of the steps in the hidden join transformation described in Section 4.1. What is common to these transformations is the need to apply one or more rules in succession, and throughout a tree. Rule blocks reduce the number of transformations that an optimizer needs to consider without complicating proofs establishing the correctness of the transformation. COKO will be presented in a later paper.

5 Related Work

Rule-based optimization is a well-known approach to building extensible query optimizers. We mentioned EXODUS [8] and Starburst [20] as examples of rule-based systems. Both systems assume a graph or tree-based query representation annotated with variables. (Therefore rules over both representations can require head and tail routines.) Both also assume that nodes in the representation are based on query operators, and not on the anonymous functions or predicates that they use (e.g., `select a < 100` forms a node in the EXODUS representation, whereas entire `select from where` queries (minus subqueries) form nodes in Starburst's representation). This has the effect of making representations have fewer nodes at the expense of making nodes larger and more complex. As well, it means that transformations involving manipulation of anonymous functions (as in Figures 1 and 2) require construction of new **nodes** and not just new trees, and therefore are inexpressible with rule languages based solely on unification.

Many rule-based systems (e.g. [26]) use rules to map algebraic operators to plan-level implementations. The transformations addressed do not consider rewriting at the source level. [16] has similar motivations to ours in that they attempt to remove code fragments that appear in rules. Like [26] however, their work primarily addresses rules that express source-to-physical transformations. They replace head routines with declarative preconditions that test the values of *attributes* that annotate the call-graph formed with rules at the nodes. However, the values that these attributes take sometimes require calls of externally defined routines; in effect, head routines are replaced in their framework by attribute-generating body routines. The few algebraic transformations they show also include tail routines. For example, they use a rule that describes how join predicates must be adjusted with a join reordering. This rule invokes a routine that sorts the predicates appearing in joins into bins, according to which tuples the predicates reference. Predicate sorting of this kind is straightforward to express with KOLA rules, as predicates of the form $p \oplus \pi_1$ examine tuples only from the first set while $p \oplus \pi_2$ examine only those in the second.

Despite the wide-spread use of the rule-based approach, scant mention can be found discussing design issues for rule languages. Rather, rule languages are usually assumed to be by-products of algebra definitions and not considered in and of themselves. An exception is the work of Sciore and Sieg [35], who suggest ways to augment rule languages over variable-based algebras to ensure formulation of a wide variety of rules. Proposed extensions include rule *preconditions* (expressed in code), and *multivariables* (abstractions of variable lists) that allow optimization rules to be independent of function arity. Multivariables are declarative, but make matching inefficient. (This is pointed out by the authors). Also, because multivariables abstract away from variable names, they make certain transformations over variable-based query representations inexpressible. For example, the code motion rule that guides the optimization of AQUA query A_4 of Figure 2 cannot be expressed with multivariables because the precondition for this rules requires reasoning about the "freeness" of variables that can no longer be referenced. In short, multivariables and precondition code are intended to address the same problem that we do; variables in query algebras make rules over algebraic representations difficult to express without additional machinery. But whereas [35]

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

add the machinery, we instead remove the variables.

As we mentioned earlier, ours is not the first combinator-based algebra proposed in a database context. [15] and [5] propose an FP-style [3] query language. But combinator-style languages are difficult for users to master and thus ill-suited as query languages. [7], [6] and [4] use combinator-based algebras to present optimization rules. They do not consider the reasons why this style of algebra is useful for implementing rule-based optimizers.

Combinator representations are often used within functional language compilers as internal representations of λ-expressions. Combinator sets proposed in the functional language literature can be classified according to whether they are *fixed* or *variable*. Fixed combinator sets use the same finite set of combinators as the target for every program's translation. Variable combinator techniques produce new combinators specific to particular programs. Fixed combinator sets include the **SKI** combinator set of Schönfinkel [34] (and its many variations) as well as the Category Theory-inspired combinator set of Curien [13]. Variable-set combinators are produced by λ-lifting [22] and supercombinator techniques [21]. Variable sets of combinators keep the size of translated expressions reasonably small while still producing the desired effect of making graph reduction efficient (the combinators generated tend to be fairly complicated). We settled on a fixed set of combinators for KOLA for two reasons:

1. Algebraic query optimization must reference a known (i.e. fixed) set of operators.

2. A reasonable increase in query size resulting from translating queries into combinator form is tolerable because queries tend to be small (compared with functional programs for example).

Nested queries have been studied extensively in the relational context [24], and have recently been examined in the context of object-oriented models [12]. In this paper, we have seen how structured data in these models can lead to very complex nested queries. The optimizations described, while useful, are expressed monolithically. We believe our approach of using multiple, gradually transforming rules will make these optimizations more easily realized and verified.

6 Conclusions

Rule-based optimizers require an internal representation of queries and a rule language for expressing transformations. Because rules act directly on representations, the effectiveness of the rule language is dependent on the form of the representation. Rule-based optimizers and optimizer generators typically cannot "go it alone". Rather, rule languages and rule-driven optimizer modules must often be supplemented with extensions to support optimizations that are inexpressible otherwise.

Optimizer extensions typically contain bodies of code. Head and body routines are calls to code placed within rules to supplement the matching and transformation capabilities provided by unification. Transformations whose expression is beyond the capability of the rule language, require expression with procedural code. But code compromises the declarative nature of rules. Optimizer generators are forced to use provided code and adopt whatever inefficiencies and errors it contains. Rules are made more difficult to reason about and prove correct, making optimizers vulnerable to mistakes in their design and implementation. The success of the rule-based optimization paradigm therefore depends on the development of expressive rule languages that support the formulation of rules without the need for supplemental code.

In this paper, we have demonstrated that the choice of query representation is a crucial factor influencing the expressivity of a rule language. We showed that *variable-based* representations compromise the expressive capabilities of the rule language for two reasons. First, variables are structurally indistinguishable. Because the appropriateness of a transformation can depend on what variables appear in a query, rules must necessarily resort to techniques beyond the matching provided by unification. Secondly, variables make function manipulation occur at the level of the expression. Expressions are not easily manipulated in the manner required for many transformations. Rules therefore must invoke external routines.

Our proposed alternative is a combinator-based algebra, KOLA. KOLA query representations have revealing and manipulable structure. We showed in Section 3 that simple transformations that require head and body routines when expressed over variable-based representations, can be expressed with KOLA rules without code. We showed in Section 4 that KOLA representations permit the expression of transformations that are not typically expressed with rules (nested query optimizations). The class of nested query optimizations that we looked at are those that "untangle" hidden join queries. We showed a five-step strategy and associated rule set that could be used to convert these queries into expressions involving nests and joins.

Our current efforts are concentrated in two areas. First, we are extending KOLA to incorporate other bulk types besides sets, both to increase compatibility with languages such as OQL (which supports bags and lists also) and to permit expressions of optimizations that exploit these kinds of collections (e.g. optimizations that defer duplicate elimination can be expressed as transformations that produce bags as intermediate results). Our other work concerns COKO. We are in the process of implementing a generator of algebraic optimizer modules based on COKO inputs. As well, we are writing COKO rule blocks for a number of optimization strategies including those presented here as well as others related to nested query optimization, join optimization, predicate ordering and semantic optimization.

7 Acknowledgements

Special thanks are due to Gail Mitchell for helpful comments on an earlier draft of this paper.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, 1988.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[3] J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[4] 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.

[5] A. Bossi and C. Ghezzi. Using FP as a query language for relational data-bases. *Computer Languages*, 9(1):25–37, 1984.

[6] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *Database Theory - ICDT'92, 4th International Conference*, volume 646 of *LNCS*. Springer Verlag, 1992.

[7] P. Buneman and R. E. Frankel. FQL – a functional query language. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, 1979.

[8] 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.

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

[10] M. Cherniack. Form(ers) over function(s): The KOLA query algebra. Technical report, Brown University Department of Computer Science, May 1995. In preparation.

[11] M. Cherniack and S. B. Zdonik. Combinator translations of queries. Technical Report CS-95-40, Brown University Department of Computer Science, September 1995.

[12] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. 4th Int'l Workshop on Database Programming Languages*, NY, NY, August 1993. Springer-Verlag.

[13] P.-L. Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhäuser, 1993.

[14] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates and quantifiers. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings if the 13th International Conference on Very Large Databases*, pages 197–208, Brighton, England, September 1987. Morgan-Kaufman.

[15] M. Erwig and U. W. Lipeck. A functional DBPL revealing high level optimizations. In P. Kanellakis and J. W. Schmidt, editors, *Bulk Types & Persistent Data: The Third International Workshop on Database Programming Languages*, pages 306–, Nafplion, Greece, August 1991. Morgan Kaufmann Publishers, Inc.

[16] L. Fegaras, D. Maier, and T. Sheard. Specifying rule-based query optimizers in a reflective framework. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 146–168, 1993.

[17] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In U. Dayal and I. Traiger, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 23–33, San Francisco, California, May 1987. ACM Special Interest Group on Management of Data, ACM Press.

[18] G. Gardarin, F. Machuca, and P. Pucheral. OFL: A functional execution model for object query languages. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 59–70, 1995.

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

[20] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 377–388, 1989.

[21] R. J. M. Hughes. *The design and implementation of programming languages*. PhD thesis, University of Oxford, 1984.

[22] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Conference on Functional Programming Languages and Computer Architecture*, LNCS. Springer Verlag, 1985.

[23] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings if the 19th International Conference on Very Large Databases*, pages 543–554, Dublin, Ireland, August 1987. Morgan-Kaufman.

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

[25] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik. The AQUA data model and algebra. In *Proc. 4th Int'l Workshop on Database Programming Languages*, New York, New York, August 1993. Springer-Verlag.

[26] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of ACM SIGMOD*, June 1988.

[27] D. Maier and S. B. Zdonik. Fundamentals of object-oriented databases. Introduction to the book 'Readings in Object-Oriented Databases.

[28] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database*. PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910, May 1993.

[29] G. Mitchell, S. B. Zdonik, and U. Dayal. An architecture for query processing in persistent object stores. In *Proc. Hawaii Int'l Conference on System Scyences, Volume II*, pages 787–798, 1992.

[30] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings iof the 15th International Conference on Very Large Databases*, pages 77–85, Amsterdam, the Netherlands, August 1989. Morgan-Kaufman.

[31] M. Muralikrishna. Improving unnesting algorithms for join aggregate SQL queries. In Yuan, editor, *Proceedings of the 18th Int'l Conference on Very Large Databases*, Vancouver, Canada, August 1992.

[32] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[33] H. J. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[34] M. Schönfinkel. Über die bausteine der mathematischen logik. *Math. Annalen*, 92:305–316, 1924.

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

[36] D. D. Straube and M. T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Office Information Systems*, 8(4), 1990.

[37] D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.

[38] S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In J. Clifford and R. King, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 158–167, Denver, Colorado, May 1991. ACM Special Interest Group on Management of Data, ACM Press.