HOMES: A Higher-Order Mapping Evaluation System

Huy Vu
Oxford University
Department of Computer Science
huy.vu@cs.ox.ac.uk

Michael Benedikt
Oxford University
Department of Computer Science
michael.benedikt@cs.ox.ac.uk

ABSTRACT

We describe a system that integrates querying and query transformation in a single higher-order query language. The system allows users to write queries that integrate and combine query transformations. The power of higher-order functions also allows one to succinctly write complex relational queries. Our demonstration shows the utility of the system, explains the implementation architecture on top of a relational DBMS, and explains optimizations that combine subquery caching techniques from relational databases with sharing detection schemes from functional programming.

1. INTRODUCTION

Higher-order functions play a fundamental role in computer science; they are critical to functional programs, and in object-oriented programming they play a key role in encapsulation. In database systems they have appeared in isolation at several points: query-transformation plays a role in numerous aspects of databases, including data integration [7], access control [6], and privacy [9].

EXAMPLE 1. *Consider the situation where we need an interface to control the access of a query over a relation instance. For example, given a source R_1 and a query Q, accesses to R_1 via Q are transformed for security reasons, returning the result of Q on only a selection of R_1 and returning only two of the columns, a and b, in the output of Q. This could be implemented via the following higher-order query.*

$\tau_0 := \lambda Q.\lambda R_1.\ SELECT\ a, b\ FROM$

$$Q(SELECT\ *\ FROM\ R_1\ WHERE\ a = 5)$$

The subterm $SELECT\ *\ FROM\ R_1\ WHERE\ a = 5$ in the example filters the input data; for modularity we may wish to develop the query without a particular filter in mind. We can thus create a more "generic" higher-order filtering query, with a query variable Fil representing a filter:

$\tau_0' := \lambda Fil.\lambda Q.\lambda R_1.\ SELECT\ a, b\ FROM\ Q(Fil(R_1))$

Later when we are ready to commit to using fil_0, we can reclaim τ_0 as $\tau_0'(fil_0)$ with

$fil_0 = \lambda R_2.(SELECT\ *\ FROM\ R_2\ WHERE\ a = 5)$.

A very common example of higher-order transformations in database management is *query rewriting* [5]. For example, given a query Q and a set of views $V_1, \ldots, V_n$ both over $D_1, \ldots, D_m$, we may be interested in generating the maximally contained rewriting of Q over $V_1, \ldots, V_n$. There are many algorithms for obtaining these rewritings (e.g. Bucket [8], MiniCon [10]). They can be encapsulated as an operator, called RW, that takes a query and a set of views as its input, and returns a rewriting of the query. In a higher-order querying system, users can freely make use of RW in building more complex queries.

EXAMPLE 2. *A user could write a higher-order term that takes queries Q, Fil, V_1, V_2 and first rewrites Q with respect to V_1 and V_2 and then post-filters the result using Fil.*

$$\tau_1 := \lambda V_1.\lambda V_2.\lambda Q.\lambda Fil.\tau_0'(Fil)(\mathsf{RW}(Q, V_1, V_2))$$

where τ_0' is the "post-filtering transform" of Example 1. Later they can instantiate the views, forming τ_2 by applying τ_1 to

$$V_1 = \lambda R_2.\ SELECT\ *\ FROM\ R_2\ WHERE\ R_2.a > 3$$

$$V_2 = \lambda R_2.\ SELECT\ *\ FROM\ R_2\ WHERE\ R_2.b < 5$$

Still later they can instantiate Q, forming τ_3 by applying τ_2 to:

$$Q = \lambda R_2.\ SELECT\ *\ FROM\ R_2\ WHERE\ R_2.b = R_2.a$$

and finally they can instantiate Fil by forming $\tau_3(fil_0)$, where fil_0 is the filter from Example 1.

Note that particular higher-order functions, such as query-rewriting transformations, or even flexible rule-based query-rewriting frameworks, have been implemented stand-alone for decades. But their implementation is not part of a system that integrates higher-order transformations with ordinary data transformations, as in functional programs. Functional databases allow the definition of higher-order terms, but do not support query transformation. In [4, 11], a framework for combining relational algebra with higher-order functional languages is defined, which we refer to as *λ-embedded query languages*: it is exactly the simply-typed λ-calculus with database operators as "constants" (that is, as built-in functions). In this work we will demonstrate HOMES (Higher-Order Mapping Evaluation System), an evaluation system for higher-order queries defined in a variant of the framework of [11].

Proceedings of the VLDB Endowment, Vol. 4, No. 12

A strength of the language is that it is extremely expressive, allowing users to write queries very succinctly. But this is also a weakness, since evaluation of the language is expensive.

EXAMPLE 3. *A relation with integer attributes (a, b) can code a graph. Let τ_p^2 be an ordinary conjunctive query checking for for the existence of a path of length 2 in such a relation: such a query is easily written as a self-join.*

Consider the term.

$$\tau_p^{16} = (\lambda \mathbf{Q_1}.\lambda \mathbf{R_1}.(\mathbf{Q_1}(\mathbf{Q_1}(\mathbf{R_1}))))((\lambda \mathbf{Q}.\lambda \mathbf{R}.(\mathbf{Q}(\mathbf{Q}(\mathbf{R}))))\tau_p^2)$$

One can check that τ_p^{16} takes as input a graph and returns a graph containing all the pairs of nodes having a path of length 16 between them. In general, using query variables, we can write queries of length n checking for paths of length doubly exponential in n.

The worst-case complexity of evaluation of the language is non-elementary [11], and so no evaluation strategy can be efficient on every query. The implementation of HOMES uses special techniques which combine *graph reduction* from functional programming with *selective materialization* to increase efficiency.

2. SYSTEM OVERVIEW

2.1 The Higher-order language

We review the syntax of the query language, given in [4, 11].

Types: We fix an infinite linearly-ordered set of *attribute names* (or *attributes*). The basic types are the *relational types* each given by a (possibly empty) set of attribute names, $\mathcal{T} = (a_1, \ldots, a_m)$. The order of any relational type is 0.

We define *higher-order types* by using the functional type constructor: if $\mathcal{T}, \mathcal{T}'$ are types, then $\mathcal{T} \to \mathcal{T}'$ is a type whose order is $\mathtt{order}(\mathcal{T} \to \mathcal{T}') = \max(\mathtt{order}(\mathcal{T}) + 1, \mathtt{order}(\mathcal{T}'))$. Order 1 types are often called *query types*.

Constants: We will fix a set of constants of each type. Database instances are constants of relational type. The operators and expressions of Relational Algebra are constants of query type. For convenience, we accept SQL syntax for those constants. In addition, we have constants accepting queries as their inputs, such as RW.

Simply typed terms: Higher-order *terms* are built up from constants and variables by using the operations of abstraction and application:

- every constant or variable is a term of the same type as the constant or the variable;
- if X is a variable of type $\mathcal{T}$ and ρ is a term of type $\mathcal{T}'$, then $\lambda X.\, \rho$ is a term of type $\mathcal{T} \to \mathcal{T}'$;
- τ is a term of type $\mathcal{T} \to \mathcal{T}'$ and ρ is a term of type $\mathcal{T}$, then $\tau(\rho)$ is a term of type $\mathcal{T}'$.

The semantics of terms can be found in [4, 11].

The *order* of a term τ is the order of its type. For example, the terms τ_0 and RW in Section 1 are of order 2.

Every closed term in our language is expressible in SQL, supplemented with recursion in case Inflationary Fixed Point is used. Thus we have a naive method to evaluate higher-order terms:

Apply β-reduction until no higher-order abstractions are present, then convert the term to an equivalent SQL query and evaluate using a standard relational engine.

2.2 System Architecture

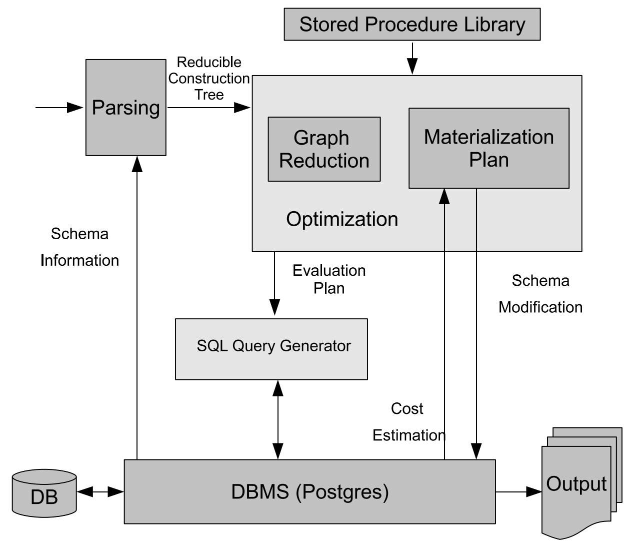

Figure 1: System Architecture.

We will explain the components of the system architecture, which is shown in Figure 1 through a running example. Let the input higher-order term be defined as:
$\tau := \lambda Q.\lambda R_1.\tau_p^2\{\sigma_{b>5}(Q(R_1))\}$

$$\lambda R_2.(\text{SELECT } * \text{ FROM } R_2 \text{ WHERE } a = 3)$$

with τ_p^2 in Example 3. That is, we first form a query transform that filters the query and then performs a self-join; then we apply the transform to a particular selection query. Given D_0 a database instance, we wish to evaluate $\tau(D_0)$.

The *Parsing* component reads the input in different forms from users, parses the input query, and validates its type; it produces an internal representation, a *construction tree*, represented in the upper part of Figure 2. In the construction tree, relational algebra operators are employed to represent the term. For example, τ_p^2 is represented as $\lambda R.\pi_{a,b}(\rho_{b,c}(R) \bowtie \rho_{a,c}(R))$.

The construction tree in Figure 2 is then input to the *Optimization* component, which also takes information from *Stored Procedure Library* to evaluate higher-order constants in the term. The library contains processing methods of higher-order operators, which are either built-in or user defined. Examples of built-in operators currently supported are Inflationary Fixed Point (`ifp`) and Query Rewriting (RW).

The output of the optimization is an evaluation plan, which contains information about the materialization tables and the order to process the subterms of the input term. The evaluation plan is presented by a set of equations of which one side is a table name, the other side is the equivalent subterm. The Optimization component will be explained in detail in Section 2.3.

The SQL Query Generator component implements the evaluation plan, generating queries at runtime to a Relational Database Management System – in our implementation we use PostgreSQL 8.4.

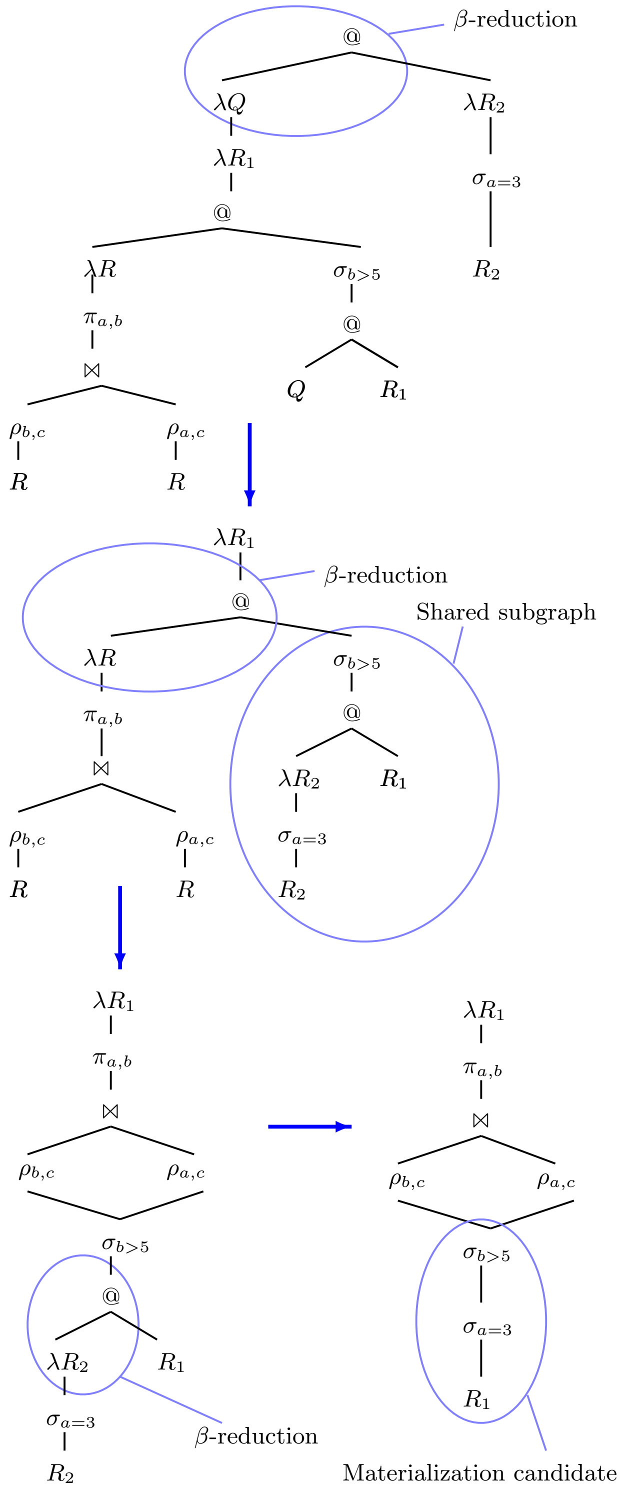

Figure 2: The construction tree and the graph reduction stage of τ.

2.3 Optimization

In a naive evaluation, we might repeat many parts of a term when converting to an SQL query, especially when there are higher-order variables which support repetitions of subterms. Many of these copying operations do not need to be carried out during the reduction, since later reductions eliminate them. This is a common problem in the evaluation of functional programs, and we adapt the technique of "graph reduction" to decide which subterms need not be copied. Note that this is not as simple as detecting common subqueries within a collection of views, since the shared subterm may contain variables of high order, and sharing may not be present in the original term but may emerge in the process of reduction. Sharing subterms is useful even when the shared subterms return higher-order objects; but it is particularly helpful for subterms returning relations, since there we can calculate the result and store it in an auxiliary table. That is, we perform (online) *materialization*, analogous to materialization of views in standard relational query processing.

There are thus two stages in the optimization: Graph reduction and generation of an evaluation plan.

Stage 1 (Graph reduction): Graph reduction not only reduces lambda variables but also calculates how many times a subtree is called, which can be used to decide if the subtree can be shared. We use a variant of the graph reduction algorithm described in [12, 2] to reduce variables. Instead of always copying subtrees, we attempt to share subtrees as much as possible. When the graph reduction process finishes, we have a graph, the subgraphs of which contain information about the number of times they are called.

Our running example τ from Section 2.2 shows the process at its simplest. From the construction tree in the upper part of Figure 2, we reduce to the DAG shown at the bottom of the figure: variable R is reduced by replacing its occurrences by pointers to the substituted subgraph, thus introducing sharing. The shared subgraph contains a variable and it can be reduced. The reduction of the shared subgraph produces multiple simplifications in the construction tree. At the end, the shared subterm is a selection over R_1.

The output of this stage is a set of candidates for materialization. In our running example, the shared subgraph of the last graph is a candidate for materialization because it is called twice.

Stage 2 (Generation of an evaluation plan): Based on the reduced graph, we can decide which subgraphs should be materialized. Our optimizer produces a materialization plan for the graph. The decisions about materialization are based on the graph and a cost function, roughly similar to the strategies used to decide materialization in a cost-based way in XML query processing over relational stores (see, e.g., [3]).

The materialization plan contains a set of tables to be materialized and a set of equations over input relations and materialized relations. In our example, a possible plan is:

$R_0 \Leftarrow$ SELECT $*$ FROM R_1 WHERE $a = 3$ AND $b > 5$

$Output \Leftarrow$ SELECT $R_0^1.a, R_0^2.b$
FROM R_0 AS R_0^1, R_0 AS R_0^2 WHERE $R_0^1.b = R_0^2.a$

Where R_0 is a new table name for materialization.

	Query 1	Query 2	Query 3	Query 4	Query 5
Query Description	1 join	8 joins	16 joins	4 joins + `ifp`	4 joins + RW
Time (Improved , scale = 1)	2.3s	0.9s	8.0s	3.4s	1.2s
Time (Naive, scale = 1)	2.9s	160s	18930s (5.3h)	11.3s	7.1s
Time (Improved , scale = 5)	5.5s	5.0s	10.1s	9.7s	4.6s
Time (Naive, scale = 5)	7.5s	930s	> 12h	91.8s	61.8s

Table 1: Empirical evaluation results.

2.4 Empirical Evaluation

We consider the performance of our implementation on a set of queries based on the TPC-H schema [1]. The test database contains the following tables: *customer*, *lineitem*, *nation*, *orders*, *part*, *partsupp*, *region*, and *supplier*. TCP-H gives default sizes for each table, and in our experiments instances are generated by uniformly multiplying the default size by a scaling factor.

Table 1 presents information on evaluation of some typical queries over instances of different scales, on an Intel ®Core i3 2.27GHz machine with 4 GB RAM. The first three queries show the improvement due to graph reduction and sharing, especially for the more expensive queries. Queries 4 and 5 show that optimization also helps when the Fixed Point and Query Rewriting operators are present.

Due to limited space, we give only hints of the nature of each query in Table 1 (in the description field). We describe the types of variables and the syntax of Query 3 in detail below.

Three relational variables R, R_1 and R_2 have schema of the table *partsupp*:

$\mathcal{S}$ = (partkey,suppkey,ps_availqty,ps_supplycost,ps_comment)

A query variable Q has schema $\mathcal{S} \to \mathcal{S}$.

The syntax of Query 3 is defined as follows.

Let $\rho_{comp} = \lambda Q.\lambda R.(Q(Q(R)))$

Let ρ_{join} =
λR_2.((SELECT $*$ FROM R_2 WHERE $partkey < 10$)
JOIN (SELECT $*$ FROM R_2 WHERE $partkey > 5$))

Let ρ_{sel} = SELECT $*$ FROM R_1 WHERE $suppkey < 200$

Then Query 3 is defined as $\lambda R_1.\{\rho_{comp}(\rho_{comp}(\rho_{join}))\}\rho_{sel}$. That is: our query takes a relation, filters it by *suppkey*, and then applies a query to it that is built up by composing a self-join query twice. In the table we consider the evaluation of Query 3 on instance *partsupp*.

Overall, we have seen significant performance improvement by applying the graph reduction plan, especially when the materialized tables are accessed many times.

3. DEMONSTRATION DETAILS

The demonstration of HOMES shows the following two main aspects. First, we show the usefulness of integrating higher-order queries into a relational DBMS. The integration allows users to create and reuse higher-order queries, including variables representing queries and query transformations. Secondly, we show how to extend traditional query optimization techniques to higher-order query languages. We show how our techniques yield acceptable running time even for complex queries.

To highlight the aspects above, during the demonstration users are guided through query development and evaluation using the system's GUI. They can begin with a number of pre-rewritten higher-order queries, applying them to several sample databases. Once the users are familiar with the syntax, they can modify pre-existing queries or create new ones from scratch. The evaluation of the queries can be done live, showing both the output and the execution time for several variants of the evaluation algorithm. The system also provides facilities to illustrate the graph reduction and materialization process in action.

4. REFERENCES

[1] Transaction processing performance council. *TPC-H Benchmark*. `http://www.tpc.org`.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[3] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *PVLDB*, pages 838–849, 2002.

[4] M. Benedikt, G. Puppis, and H. Vu. Positive Higher Order Queries. In *PODS*, pages 27–38, 2010.

[5] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. What is query rewriting? In *KRDB*, pages 17–27, 2000.

[6] W. Fan, F. Geerts, and X. J. A. Kementsietsidis. Rewriting Regular XPath Queries on XML Views. In *ICDE*, pages 666 – 675, 2007.

[7] A. Levy, A. Rajaraman, and J. Ullman. Answering Queries using Limited External Query Processors . In *PODS*, pages 227–237, 1996.

[8] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996.

[9] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *PODS*, pages 123–134, 2010.

[10] R. Pottinger and A. Halevy. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10:182–198, 2001.

[11] H. Vu and M. Benedikt. Complexity of Higher-Order Queries. In *ICDT*, pages 208–219, 2011.

[12] C. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. PhD thesis, University of Oxford, 1971.