Language-Integrated Query with Ordering, Grouping and Outer Joins (Poster Paper)

Tatsuya Katsushima
Tohoku University, Japan
katsushima@sf.ecei.tohoku.ac.jp

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Abstract

Language-integrated query systems like T-LINQ or QUEΛ make relational operations on (generally external) data feel like the ordinary iteration over native arrays. As ordinary programs, queries are type-checked, can be abstracted over and composed. To access relational database systems, queries are eventually translated into well-formed, well-typed and efficient SQL. However, most existing language-integrated query systems implement only a small subset of relational operations supported by modern databases.

To make QUEΛ full-featured, we add to it the operations corresponding to SQL's ORDER BY, LIMIT, OUTER JOIN, GROUP BY and HAVING. We describe the type system and the normalization rules to produce the efficient SQL code. The type system not only ensures by construction the intricate SQL validity constraints. It also prevents the accidental composition of hard-to-optimize queries.

Our extended QUEΛ is embedded in OCaml in the tagless-final style.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code Generation; H.2.3 [*Database Management*]: Languages—Query Languages

Keywords SQL, tagless-final, language-integrated query, LINQ, EDSL

1. Introduction

Language-integrated query is "smooth integration" of database queries with a conventional programming language. Not only do we access (generally external) relational data as if they were local arrays of records. Not only do we type-check queries as ordinary programs. Mainly, we use the abstraction facilities of the programming language – functions, modules, classes, etc. – to parameterize queries, reuse old queries as parts of new ones, and compile query libraries.

Alas, the completely smooth integration is still an open problem. For one, the *lingua franca* of relational databases, SQL, was not designed as a programming language and is not compositional. The later added subqueries help, yet the full compositionality is still lacking: e.g., the SQL standard and many databases disallow ORDER BY in subqueries. Mainly, subqueries are optimized less well. Even the vendor literature recommends subqueries be re-written into a flat SQL for performance.

The re-writing approach has been worked out in detail in [2], and implemented, in particular, in T-LINQ [1]. QUEΛ [5] is a re-implementation of T-LINQ using the tagless-final embedding into OCaml, which makes the rewriting rules user-definable, extensible and typeable. The rules convert the query to a normal form from which the 'flat' SQL (without subqueries) can be easily generated.

T-LINQ and the original QUEΛ used only the core relational operations of selections, cartesian products, projections and unions. QUEΛ also considered the type system (but not the re-writing rules) for grouping. The practically used SQL however has more operations: specifically, sorting, selection of a subset of sorted rows (LIMIT), and outer joins. It is not known how to optimize composite queries that involve these facilities .

We report on the ongoing work to make language-integrated queries (QUEΛ in particular) support the full range of practically significant relational operations of the modern database engines:

- We smoothly integrate into programming language the operations corresponding to SQL's ORDER BY, LIMIT, OUTER JOIN, GROUP BY and HAVING;
- We give the semantics of these operations independent of SQL;
- We assign types to those operations so to ensure, by construction, that the generated SQL code passes the intricate validity checks imposed by the SQL Standard. We also prevent the accidental composition of hard-to-execute queries, such as composing GROUP BY with other queries. Our type system makes the programmer aware of the performance implications and forces the explicit use of common-table-expressions if such compositions are really desirable.
- We design re-writing rules for queries with nested ORDER BY and outer joins;
- We cast the query normalization as an effectful normalization-by-evaluation.

The closely related to us Haskell's Opaleye [3] and HRR [4] also deal with ordering, grouping and outer joins. They do not present relational data as local arrays to iterate over. They have rather complicated type system. The published materials offer no semantics other than the re-writing into SQL. Mainly, these systems consider no query normalization and optimizations, relying on subqueries to achieve compositionality.

2. Extended QUEΛ

We describe the extended QUEΛ by examples. The examples rely on the following sample database (the same as in [5]) of two tables below. The first query sorts the products table by price (we show

PEPM'17, January 16–17, 2017, Paris, France
ACM. 978-1-4503-4721-1/17/01...$15.00
http://dx.doi.org/10.1145/3018882.3018893

products

pid	name	price
1	Tablet	500
2	Laptop	1,000
3	Desktop	1,000
4	Router	150
5	HDD	100
6	SSD	500

orders

oid	pid	qty
1	1	5
1	2	5
1	4	2
2	5	10
2	6	20
3	2	50

$type\ Product = \langle \mathsf{pid} : Int, \mathsf{name} : String, \mathsf{price} : Int \rangle$

$type\ Order = \langle \mathsf{oid} : Int, \mathsf{pid} : Int, \mathsf{qty} : Int \rangle$

QUEΛ alongside the corresponding SQL):

```
Q_o1 =
  for (p ← table("products"))
    order [(price, DESC)]
      yield p
```

```
SELECT p.*
FROM products AS p
ORDER BY price DESC
```

The second query returns only the first two rows of the sorted table

```
Q_ol1 =
  for (p ← table("products"))
   orderlimit [(price, DESC)] (0, 2)
    yield p
```

```
SELECT p.*
FROM products AS p
ORDER BY
 price DESC
LIMIT 2 OFFSET 0
```

The next example demonstrate query composition, reusing already built queries like Q_{o1} to define new ones:

```
Q_o2 = for (x ← Q_o1) for (o ← table("orders"))
       where (x.pid = o.pid) order [(sale, ASC)]
       yield ⟨pid=x.pid, name=x.name, sale=x.price * o.qty⟩
```

Naively doing the similar composition in SQL results in

```
SELECT x.pid, x.name, x.price * o.qty AS sale
FROM
   (SELECT * FROM products ORDER BY price DESC) AS x,
   orders AS o
WHERE x.pid = o.pid ORDER BY sale
```

with ORDER BY in a subquery – which is not allowed in many databases (e.g., Oracle). PostgreSQL does allow nested ORDER BY but performs unnecessary sorting, costing performance.

QUEΛ normalizes queries (see the next section), producing

```
Q^n_o2 =
  order [(sale, ASC)]
  for (p ← table("products"))
  for (o ← table("orders"))
  where (p.pid = o.pid)
  yield ⟨pid=p.pid,
   name=p.name,
   sale=p.price * o.qty⟩
```

```
SELECT p.pid, p.name,
   p.price * o.qty as sale
FROM products AS p,
        orders AS o
WHERE p.pid = o.pid
ORDER BY sale ASC
```

eliminating the nested ORDER BY (as well as the subquery). The result is easily convertible to a flat SQL (shown on the right).

If we attempt to write Q_{o2} with the limited Q_{ol1} in place of Q_{o1}, it will not type check in QUEΛ: ordering with the limit cannot be eliminated and such a composition will have poor performance (if even allowed in some databases). The user has to use the let-table–expression (which corresponds to with-expression of SQL), to make the performance implications explicit:

```
Q_o2 = let table t = Q_ol1 in
       for (x ← t) for (o ← table("orders"))
         where (x.pid = o.pid) order [(sale, ASC)]
         yield ⟨pid=x.pid, name=x.name, sale=x.price * o.qty⟩
```

For outer joins, we added the 'binary version' of the for-iterator

```
Q_left =
  leftjoin (p ← table("products"))
       (o ← table("orders"))
       (p.pid = o.pid)
   where (p.price > 700)
   yield ⟨pid=p.pid,
    price=p.price, qty=o.qty⟩
```

```
SELECT p.pid, p.price, o.qty
FROM products AS p
   LEFT JOIN orders AS o
   ON p.pid = o.pid
WHERE p.price > 700
```

which is however treated as a macro, expanding into a UNION ALL of an inner join, and a traversal of the products table selecting pids with no corresponding orders (In table' below, each field is of the option type.) The two queries can then be normalized as usual.

```
Q_left =
  for (p ← table("products"))
  for (o ← table'("orders"))
  where (p.pid = o.pid)
  where (p.price > 700)
  yield ⟨pid=p.pid, price=p.price,
   qty=o.qty⟩ ⊎
  for (p ← table("products"))
  for (o ← yield ⟨oid = None,
        pid = None, qty = None⟩)
  where (¬exists
  (for (o' ← table("orders"))
        where (p.pid = o'.pid) yield 1))
  where (p.price > 700)
  yield ⟨pid=p.pid, price=p.price,
   qty=o.qty⟩
```

```
SELECT p.pid,
   p.price , o.qty
FROM products AS p,
        orders AS o
WHERE p.pid=o.pid
AND p.price > 700
UNION ALL
SELECT p.pid,
 p.price , null
FROM products AS p
WHERE NOT EXISTS
(SELECT 1
 FROM orders AS o
 WHERE p.pid=o.pid)
 AND p.price > 700
```

3. Normalization and the Type System

To normalize the extended QUEΛ we added the following rules (where $L \mathbin{@} M$ means concatenation).

$$\text{where } L \text{ (order } M\ N) \rightsquigarrow \text{order } M \text{ (where } L\ N) \quad (\textsc{WhereOrder})$$

$$\text{for } (x \leftarrow \text{order } L\ M)\ N \rightsquigarrow (\text{for } (x \leftarrow M)\ N) \quad (\textsc{ForOrder}_1)$$

$$\text{for } (x \leftarrow L) \text{ (order } M\ N) \rightsquigarrow \text{order } M \text{ (for } (x \leftarrow L)\ N) \quad (\textsc{ForOrder}_2)$$

$$\text{order } L \text{ (order } M\ N) \rightsquigarrow \text{order } (L \mathbin{@} M)\ N \quad (\textsc{OrderOrder})$$

$$\text{order } [\,]\ M \rightsquigarrow M \quad (\textsc{OrderEmpty})$$

$$\text{orderlimit } L\ M \text{ (order } N\ K) \rightsquigarrow \text{orderlimit } (L \mathbin{@} N)\ M\ K \quad (\textsc{OrderlimitOrder})$$

The rules make it clear that ordering is regarded as an effect, to be applied to the finished complete query. This is the meaning given to ORDER BY in SQL.

We omit the type system for the lack of space, only mentioning the typing judgment $\Gamma \vdash L : t; \tau$ that an expression L has the type t (like $\langle \mathsf{pid} : Int \rangle\ Bag$) with effect annotations τ. The latter describe the effects of L such as ordering or grouping.

4. Conclusions

We have described the extension of QUEΛ with ordering and outer joins, giving a short summary of the normalization process and the type system. We are investigating the implementation of the normalization as normalization-by-evaluation, with control effects like ordering.

References

[1] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP '13*, pages 403–416, New York, NY, USA, 2013. ACM.

[2] E. Cooper. The script-writer's dream: How to write great sql in your own language, and be sure it will succeed. In *DBPL '09*, pages 36–51, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] T. Ellis. Opaleye. `https://github.com/tomjaguarpaw/haskell-opaleye`. last visited: Dec. 2014.

[4] K. Hibino, S. Murayama, S. Yasutake, S. Kuroda, and K. Yamamoto. Haskell relational record. `http://khibino.github.io/haskell-relational-record/`. last visited: Jun. 2015.

[5] K. Suzuki, O. Kiselyov, and Y. Kameyama. Finally, safely-extensible and efficient language-integrated query. In *Proc. PEPM*, pages 37–48. ACM, 2016. .