Author's version.
To appear in the Proceedings of FLOPS 2020 (International Symposium on Functional and Logic Programming) as a volume in Springer LNCS series.

Language-Integrated Query with Nested Data Structures and Grouping

Rui Okura and Yukiyoshi Kameyama

University of Tsukuba, Japan
rui@logic.cs.tsukuba.ac.jp, kameyama@acm.org

Abstract. Language-integrated query adds to database query the power of high-level programming languages such as abstraction, compositionality, and nested data structures. Cheney et al. designed a two-level typed language for it and showed that any closed term of suitable type can be normalized to a single SQL query which does not have nested data structures nor nested SELECT clauses.
This paper extends their language to cover the GROUP BY clause in SQL to express grouping and aggregate functions. Although the GROUP BY clause is frequently used, it is not covered by existing studies on efficient implementation of language-integrated queries. In fact, it seems impossible to express composition of two aggregate functions by a single aggregate functions, therefore, there exists a query with nested GROUP BY clauses which has no equivalent query without nested one. However, since several SQL such as PostgreSQL allows nested queries, we can still ask if it is possible to convert an arbitrary query with grouping and aggregation to a single query in SQL which allows nested queries, but disallows nested data structures such as a table of tables.
This paper solves the latter question affirmatively. Our key observation is that the GROUP BY clause in SQL does two different kinds of things, manipulating input/output data and grouping with aggregation, the former can be transformed, but may have complex types, while the latter cannot be transformed, but has simple types. Hence, we decouple the GROUP BY clause and introduce primitives into our language-integrated query to obtain a calculus which can express GROUP BY. We then show our language has the normalization property that every query is converted to a single query which does not have nested data structures. We conduct simple benchmarks which show that queries in our language can be transformed to efficient SQL queries.

Keywords: database · language-integrated query · grouping · aggregation · normalization · type safety

1 Introduction

Language-integrated query is gaining an increasingly bigger attention by integrating a database query language such as SQL with a high-level programming language. Microsoft's LINQ is a typical example, used by various applications. Existing languages

for language-integrated query allow one to interact with a database management system, construct abstraction mechanisms and complex data structures, and compute over them.

A classic problem in language-integrated queries is the query-avalanche problem; composing two queries and executing the result of the composition may sometimes need a huge number of transactions with a database. Another classic problem is that nested data structures such as a list of lists are allowed in language-integrated queries, while they are not allowed in SQL, hence they are not directly implementable. Cooper [1] proposed normalization on queries to solve these problems. He showed that a closed query of non-nested type [1] can be always transformed ('normalized') to a form which does not use abstractions, nested data structures, or nested comprehensions, hence can be translated to SQL. Moreover, the result of normalization is a single SQL query, solving the query avalanche problem. Cheney et al. [2] formalized his idea in a two-level typed language with quotation and anti-quotation to give theoretical foundation for it, and Suzuki et al. [3] implemented it via tagless-final encoding as an extensible type-safe framework. However, none of the studies mentioned above targeted the features of grouping and aggregate functions, which are indispensable in many realistic queries.

This paper addresses the problem of introducing grouping and aggregate functions into language-integrated queries while keeping efficient implementation. Grouping (the GROUP BY clause in SQL) classifies data records based on the given *keys*, and computes aggregated values for each group of records. Aggregation is a reduction operation, which computes a single value from a list of values in the same group.

An example in SQL is given as follows:

```
SELECT p.orderId AS oid, SUM(p.quantity) AS sales
FROM products AS p
GROUP BY p.orderId
```

This query gets data records from the products table, classifies them into groups based on the orderId key, and returns, for each group, a record which consists of an orderId and the summation of the quantity. SUM is called an *aggregate* function, which computes the sum of all records for each group, and the GROUP BY clause specifies the key for this grouping. Following Cheney et al., we write $\mathbf{for}(x \leftarrow L)\ M$ for the following SQL query:

```
SELECT M
FROM L AS x
```

and we introduce a new construct $\mathbf{gfor}(x \leftarrow L;\ K)\ M$ for the SQL query with grouping:

```
SELECT M
FROM L AS x
GROUP BY K
```

[1] We say that a table type (a bag type of a record type) is not nested, if each component type of the record is a basic type such as string, integer, or floating-point number. All other table types are nested in our terminology.

While the GROUP BY clause (and the **gfor** clause) is simple to handle in SQL, it is problematic as a construct in language-integrated query, which will be explained as follows.

First, there seems no normalization rules for the combination of **gfor** and another control structure such as **for** or **gfor** itself. A normalization rule is a rule to transform a query to a normal form which directly corresponds to an SQL query, and Cheney et al. showed that **for** follows a number of normalization rules such as $\textbf{for}(x \leftarrow \textbf{for}(y \leftarrow L)\ M)\ N \rightsquigarrow \textbf{for}(y \leftarrow L)\ \textbf{for}(x \leftarrow M)\ N$ which are sufficient to flatten nested control structures [2]. On the other hand, **gfor** does not seem to have such normalization rules which work in general. For instance, $\textbf{gfor}(x \leftarrow \textbf{for}(y \leftarrow L)\ M;\ K)\ N$ cannot be normalized to a natural candidate $\textbf{for}(y \leftarrow L)\ (\textbf{gfor}(x \leftarrow M;\ K)\ N)$, which is semantically different. There are other combinations of constructors such as **gfor**-**gfor** which also suffer from the same problem. Informally it is explained by an example: suppose we are given a table for GPAs for all students in a university, which has several schools and each school has departments. and we want to determine, for each school, the department whose students' average GPA is the best among the departments in the school. Clearly, we need to compute average per department, and then take the maximum value per school, each of which corresponds to grouping and aggregate functions AVG and MAX. Therefore, we need to have nested **gfor** constructs to obtain the correct result.

It follows that our target language for SQL need to have *subqueries*, which means an input (or an output) of a query is the result of another query. Several SQL dialects including PostgreSQL indeed allow subqueries, hence we also assume that our SQL backend allows (an arbitrary nesting of) subqueries.

Although allowing subqueries in SQL solves the problem of nested control structures, we still have a problem of nested data structures. To see this, let Q_1 be $\textbf{gfor}(x \leftarrow L;\ K)\ M$ where M has a nested data structure such that each component type of the record is a bag of a basic type. Let Q_2 be a query $\textbf{for}(y \leftarrow Q_1)\ N$ whose type is not nested.We expect that Q_2 is normalized to a query which is translated to a single SQL, in particular, the resulting query has no nested data structures. Unfortunately, normalization does not work for this query, since it contains **gfor** which is a barrier for any normalization to occur. One may think that the **for** construct in Q_1 (at the outermost position of (Q_2) can be moved inward, and we can rewrite Q_2 into $\textbf{gfor}(x \leftarrow L;\ K)\ \textbf{for}(y \leftarrow M)\ N$, which can be further transformed. However, this rewriting has another problem; if N has an aggregate function which operates on grouping outside of Q_2, the aggregate function goes under this **gfor**, rather than grouping outside of Q_2, leading to an incorrect result. Namely, this rewriting may not be semantics-preserving.

In summary, having **gfor** in language-integrated query causes a big problem and most desirable properties which Cheney et al. proved for the language without **gfor** will be lost. Hence, it has been an open problem to have grouping and aggregate functions in language-integrated query, while keeping the desirable properties such as normalization to non-nested data structures.

[2] Queries where a **for** construct exists inside another **for** construct are called queries with nested control structures.

This paper solves the problem above at least partially. We start by observing that the **gfor** construct in language-integrated queries (or the GROUP BY clause in SQL) does too many things; it performs not only grouping, but also aggregation and construction of (possibly complicated) output. In the context of this study, these three processes have quite different nature, and should be treated separately.

- Grouping and Aggregation; examples are taking an average score among a department, and getting the maximum average score among a school. These operations, if combined with other control structures, cannot be transformed, while their input and output types are restricted to basic types. (Note that aggregate functions are provided in the target SQL, and they work on a collection of basic values and return a reduced value of the same type as input.)
- Output Construction; an example is constructing a nested list whose element is a student-score pair whose score is above the average in her department. This operation may have complicated types (arbitrarily nested types), however, they are standard operations made from **for**, UNION ALL , WHERE and so on, hence, they can be transformed by Cheney et al.'s normalization rules.

Since GROUP BY is a combination of the two, we cannot normalize it, nor its data type may be nested, and we got stuck. A lesson learned from here is that we should *decouple* the two, then we can normalize output construction to normal form, while grouping/aggregation have non-nested types, hence there are no obstacles to obtain a result which has no nested data structures. This is a rather simple idea but as far as we know, there is no similar research in this direction, and the present paper is a straightforward realization of this simple idea.

We design our source language based on Cooper's work and Cheney et al.'s work, and add the functionality of grouping and aggregate functions by a new construct instead of the **gfor** construct. The new construct captures only the first process of the above two, namely, grouping and aggregation, and the second process is represented by the existing constructs provided by Cheney et al. We argue that most functionality of GROUP BY in SQL can be recovered by combining the new constructs and existing ones by some sort of rewriting. We introduce a type system and prove several interesting properties. We then give a set of transformation rules which transform all the typable queries of an appropriate type [3] to a single SQL query, thus avoiding the Query Avalanche problem completely.

The rest of this paper is organized as follows. Section 2 informally discusses how to resolve nested clauses with grouping. Section 3 introduces our languages and transformation rules, and Section 4 introduces adding grouping to the language. And the performance measurements for our language and implementation is explained in Section 5. Finally, we state concluding remarks in Section 6.

[3] An SQL-convertible query must produce a collection of records whose fields are of base types. Hence, we restrict our query to have this class of types.

2 Example of Grouping and Aggregation

In this section, we consider several examples for language-integrated query, and informally discuss our new primitives for aggregation and grouping. Formal development will be presented in the next section. A database used in our example has two tables in Fig.1

products

pid	name	price
110	shirt	100
111	T-shirt	200
210	pants	500
310	suit	1000

orders

oid	pid	qty
1	110	2
1	111	3
2	110	5
2	210	10
2	310	15
3	310	20

$$\textbf{type}\ Ordering = \{Products : \mathsf{Bag}\ \{\mathrm{pid} : Int, \mathrm{name} : String, \mathrm{price} : Int\},\\ Orders : \mathsf{Bag}\ \{\mathrm{oid} : Int, \mathrm{pid} : Int, \mathrm{qty} : Int\}\}$$

Fig. 1. Sample database tables

The products table has (left) columns of product ID (pid), name, and price (price), while the orders table (right) has columns of order ID (oid), product ID (pid), and quantity (qty). Let us first introduce an example without grouping and aggregation as follows:

$$\begin{aligned} Q = \ & \textbf{for}(p \leftarrow \textbf{table}(\text{“products”}))\\ & \textbf{for}(o \leftarrow \textbf{table}(\text{“orders”}))\\ & \textbf{where}\ (p.\mathrm{pid} = o.\mathrm{pid})\\ & \textbf{yield}\ \{\mathrm{oid} = o.\mathrm{oid}, \mathrm{sales} = p.\mathrm{price} * o.\mathrm{qty}\} \end{aligned}$$

which corresponds to the following SQL query:

```
SELECT o.oid AS oid, p.price * o.qty AS sales
FROM products AS p, orders AS o
WHERE p.pid = o.pid
```

The collection is a multiset, or a *bag*. The above query scans tables, and returns a bag of records consisting of two fields oid and sales, whose values are order ID and qty multiplied by the price. In this paper, we do not consider the value NULL, and assume all fields have some values of appropriate type.

Next, we use an aggregate function without grouping, shown below:

$$
\begin{aligned}
Q_0 = \ & \textbf{yield}\ \mathcal{A}_\alpha(\textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
& \qquad \textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
& \qquad \textbf{where}\ (p.\text{pid} = o.\text{pid}) \\
& \qquad \textbf{yield}\ \{\text{sales} = p.\text{price} * o.\text{qty}\}) \\
& \text{Where } \alpha = \{(\text{sales}, \text{SUM}, \text{sales\_sum})\}
\end{aligned}
$$

which corresponds to the following SQL query:

```
SELECT SUM(p.price * o.qty) AS sales_sum
FROM products AS p, orders AS o
WHERE p.pid = o.pid
```

where SUM is an *aggregate* function, which computes the sum of p.price * o.qty for *all* collections in the constructed table, and $\mathcal{A}_\alpha$ is the new operator in our language, standing for aggregation application.

The role of the new operator $\mathcal{A}_\alpha$ is to apply aggregate functions to the components specified in α. The component values are taken from its argument. In the above example $\alpha = \{(\text{sales}, \text{SUM}, \text{sales\_sum})\}$ specifies that it should retrieve values from the sales field of the argument, convert to a field named sales_sum and returns the value. The resulting record of $\mathcal{A}_\alpha(L)$ has the record type which consists of a field sales. Thus, the above query in our language does exactly the same thing as the query in SQL.

Clearly the notation in our language is heavier than the corresponding expression in SQL, but it is justified by the following argument. The essential difference between the two is the position the function SUM appears at: in SQL (the lower query), SUM appears at deep inside of the query, while in our language (the upper query), it appears at the outermost position. Since SUM in our language appears remotely from its real argument, we need to specify which field and return field name it will pick up, and the heavier notation above is necessary. But our notation has a merit that the target table of aggregate functions is clearer. In the above example, the target table of SUM is the argument computed by the **for** expression, while the target of SUM in SQL is determined by its external context which is not always clear. In this example, there are no GROUP BY clauses in the query so the target table is the whole expression, but in general, there may be several GROUP BY clauses around SUM, and they form a sort of binder-bindee correspondence. But, since they are not really binders (no variables are used to make the correspondence explicit) so the correspondence is fragile under rewriting, or normalization. When designing normalization rules, we always need to consider if the binder-bindee correspondence is kept correctly, which is quite cumbersome, and sometimes impossible (note that our language has the standard lambda binding and function application, hence substitution for variable may occur at any time of computation).

We then add the functionality of grouping to the $\mathcal{A}_\alpha$ operator. The extended operator is denoted by $\mathcal{G}_{(\kappa,\alpha)}$ where α is the specification for aggregate functions as in $\mathcal{A}_\alpha$. The extra parameter κ is a list of field names and considered as grouping keys on which grouping takes place. To show an example, we perform grouping with the example

above.

$$
\begin{aligned}
Q_1 = \mathcal{G}_{(\mathrm{oid},\alpha)}(&\textbf{for}(p \leftarrow \textbf{table}(\text{“products”}))\\
&\textbf{for}(o \leftarrow \textbf{table}(\text{“orders”}))\\
&\textbf{where}\ (p.\mathrm{pid} = o.\mathrm{pid})\\
&\textbf{yield}\ \{\mathrm{oid} = o.\mathrm{oid}, \mathrm{sales} = p.\mathrm{price} * o.\mathrm{qty}\})\\
\text{Where } \alpha = \{&(\mathrm{sales}, \mathrm{SUM}, \mathrm{sales\_sum})\}
\end{aligned}
$$

which corresponds to the following SQL query:

```
SELECT o.oid AS oid, SUM(p.price * o.qty) AS sales_sum
FROM products AS p, orders AS o
WHERE p.pid = o.pid
GROUP BY o.oid
```

In the lower query, we have added the order ID field to the record created dynamically, which will be the grouping key as specified by the GROUP BY clause on the last line. In the upper query, we also do the same thing, and in addition to it, the grouping operator specifies not only α, but also order ID as the grouping key. When we execute the upper query, it groups the table create by the **for** clause based on the order ID field, computes the sum of qty multiplied by price for each group, and then returns a record consisting of the order ID field, and the summation.

The merit and demerit of expressing grouping and aggregate functions in terms of the $\mathcal{G}_{(\kappa,\alpha)}$ operator inherit those for the $\mathcal{A}_\alpha$ operator. In addition, one query in SQL may have more than one GROUP BY clauses, and then the correspondence between the GROUP BY clause and aggregate functions are even more complicated, and will be error prone. On the contrary, our notation has its scope (its argument) as the target table of grouping and aggregate functions, hence we seldom make any 'scope'-related issues. Note that $\mathcal{G}_{(\kappa,\alpha)}$ is a natural extension of $\mathcal{A}_\alpha$, but for technical reason, $\mathcal{G}_{(\cdot,\alpha)}$ (no grouping keys) is equivalent to **yield** $\mathcal{A}_\alpha$, which returns a singleton consisting of $\mathcal{A}_\alpha$. Modulo this small twist, the former extends the latter, and $\mathcal{A}_\alpha$ exists only for the purpose of explanation.

In SQL, we can group, aggregate values, and construct complicated data from them all in one query. As we discussed in the previous section, it is problematic to do all three things in a single primitive, therefore, our language does not have such a super operator. Instead, our operator $\mathcal{G}_{(\kappa,\alpha)}$ can do grouping and aggregation only. The resulting value of applying this operator to an expression is a bag of records consisting of the results of aggregate functions, whose types are not nested. Any operation after applying aggregate functions are disallowed by this primitive. For instance, the following query has no direct counterpart in our language:

```
SELECT o.oid AS oid,
       SUM(p.price * o.qty)/SUM(o.qty) AS average
FROM products AS p, orders AS o
WHERE p.pid = o.pid
GROUP BY o.oid
```

where we divide one aggregated value by another. It is still no problem to precompute values before aggregation such as SUM(p.price * o.qty).

We can recover the lost expressiveness by simple rewriting. A query which is equivalent to the above one may be written in our language as follows:

$$
\begin{aligned}
Q_2 = \ & \textbf{for}(q \leftarrow \mathcal{G}_{(\mathrm{oid},\alpha)}(\textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
& \qquad \textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
& \qquad \textbf{where}\ (p.\mathrm{pid} = o.\mathrm{pid}) \\
& \qquad \textbf{yield}\ \{\mathrm{oid} = o.\mathrm{oid}, \mathrm{sales} = p.\mathrm{price} * o.\mathrm{qty}, \\
& \qquad\qquad \mathrm{qty} = o.\mathrm{qty}\})) \\
& \textbf{yield}\ \{\mathrm{oid} = q.\mathrm{oid}, \mathrm{average} = q.\mathrm{sales\_sum}/q.\mathrm{qty\_sum}\} \\
& \text{Where } \alpha = \{(\mathrm{sales}, \mathrm{SUM}, \mathrm{sales\_sum}), (\mathrm{qty}, \mathrm{SUM}, \mathrm{qty\_sum})\}
\end{aligned}
$$

Thus we divide one big process performed by the GROUP BY clause into a combination of triple nested control structures **for**-$\mathcal{G}_{(\kappa,\alpha)}$-**for**. It is arguable that this decomposition (or 'decoupling') is beneficial for performance, but we believe that, as long as the nested data structures are concerned, our decomposition is the only way to normalize *all* queries systematically into a single SQL query which has subqueries but does not have nested data structures.

The above query in our language corresponds to the following query in SQL:

```
SELECT q.oid AS oid, q.sales_sum / q.qty_sum AS average
FROM (SELECT o.oid AS oid,
             SUM(p.price * o.qty) AS sales_sum,
             SUM(o.qty) AS qty_sum
      FROM products AS p, orders AS o
      WHERE p.pid = o.pid
      GROUP BY o.oid) AS q
```

which uses a subquery and performs badly if we compare it with the above single query. In this paper, we do not talk about optimization of queries, but a very clever query optimization engine for SQL will hopefully optimize the last one to the previous one.

3 The Language with Aggregate Functions

This section explains the base language for language-integrated query in the existing studies, and introduces our language with aggregate functions. Grouping will be added to the language in the next section.

3.1 Base Language

The base language Quel is essentially the same as Cooper's source language [1] without effects (which is 'nearly the same' as Nested Relational Calculus [4]), and Cheney et al.'s T-LINQ [2] without quotation and code generation. Fig. 2 gives the syntax of types and terms in Quel where t denotes a name of database tables, and l denotes a field name of a record.

$$
\begin{array}{lrl}
\text{Base types} & O & ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \\
\text{Types} & A, B & ::= O \mid A \to B \mid \mathsf{Bag}\ A \mid \{\overline{l : A}\} \\
\text{Typeenvironment} & \Gamma & ::= \phi \mid \Gamma, x : A \\
\text{Terms} & L, M, N & ::= \lambda x.\ M \mid M\ N \mid \oplus(\overline{M}) \mid M \uplus N \mid x \mid c \\
 & & \mid \mathbf{for}(x \leftarrow M)\ N \mid \mathbf{where}\ L\ M \mid \mathbf{yield}\ M \mid [\,] \\
 & & \mid \mathbf{exists}\ M \mid \mathbf{table}(t) \mid \{\overline{l = M}\} \mid L.l
\end{array}
$$

Fig. 2. Types and Terms of Quel

Types are either a basic type (integers, booleans, and strings), a function type $A \to B$, a bag type $\mathsf{Bag}\ A$, or a record type $\{\overline{l : A}\}$ where $\overline{l : A}$ is abbreviation of a sequence $l_1 : A_1, \cdots, l_k : A_k$ for some $k \geq 0$. The bag type is the type for multisets in which the order of elements is irrelevant and the number of elements matters. A record type $\{\overline{l : O}\}$ where O is a basic type is called a *flat* type. The bag type of a flat record type is also called a flat type. Flat types are important in the study of language-integrated query, since SQL allows only values of flat types as input and output tables.

Terms are either lambda terms augmented with a primitive operator $\oplus$, a variable x, a constant c, $\{\overline{l = M}\}$ (record), $L.l$ (selection), or constructed by database primitives such as $M \uplus N$ (multiset union), $\mathbf{for}(x \leftarrow M)\ N$ (bag comprehension) $\mathbf{where}\ L\ M$ (conditional), $\mathbf{yield}\ M$ (singleton), $\mathbf{exists}\ M$ (emptiness test), and $\mathbf{table}(t)$ (database table with name t). The term $\mathbf{for}(x \leftarrow M)\ N$ corresponds to the SELECT statement in SQL, which computes N for each element in (the value of) M, and returns their multiset union. The term $\mathbf{where}\ L\ M$ returns the value of M if L returns true, and returns the empty bag $[]$ otherwise. The term $\mathbf{yield}\ M$ creates a singleton multiset consisting of the value of M. The term $\mathbf{exists}\ M$ is emptiness test for a multiset M and returns a boolean value. The variable x in $\lambda x.\ M$ and $\mathbf{for}(x \leftarrow L)\ M$ are bound in M. As usual, we identify two terms which are α-equivalent.

3.2 The Language Quela

We add aggregate functions to Quel, and call the extended language Quela. Fig. 3 defines new syntax in Quela where a sequence of dots means the corresponding syntax in Quel.

$$
\begin{array}{lrl}
\text{Terms} & L, M, N & ::= \ldots \mid \mathcal{A}_\alpha(L) \\
\text{A-Spec} & \alpha & ::= \{\overline{(l, \circledcirc, l')}\}
\end{array}
$$

Fig. 3. Types and Terms of Quela

The term $\mathcal{A}_\alpha(L)$ applies aggregate functions to L as specified by α. Here, α is a finite collection of pairs of a field name and an aggregate function $\circledcirc$ such as MAX,

MIN, AVG, COUNT, and SUM.[4] An example of α is $\{(l_1, \text{MAX}, l'_1), (l_2, \text{SUM}, l'_2)\}$, which means that we apply MAX to the values of the l_1 field and SUM to the values of the l_2 field, and returns a record consisting of these data with new field names l'_1 and l'_2. For simplicity, we assume that all field names in α are mutually distinct, but this restriction can be easily removed.

Quela has the standard call-by-value, left-to-right semantics. Let us explain how the term $\mathcal{A}_\alpha(L)$ is evaluated where $\alpha = \{(l_1, \circledcirc_1, l'_1), \cdots, (l_k, \circledcirc_k, l'_k)\}$. L is an expression of record type whose fields are $l_1, \cdots, l_k$. For each $i \leq k$, we apply the aggregate function $\circledcirc_i$ to the l_i-component of L to get an aggregated value which we call v_i. Then we return a record $\{l'_1 = v_1, \cdots, l'_k = v_k\}$ as the result. For instance, suppose L is a bag with two elements $[\{l_1 = 10, l_2 = 20\},\ \{l_1 = 30, l_2 = 10\}]$. Then the term $\mathcal{A}_{\{(l_1,\text{SUM},l'_1),(l_2,\text{MAX},l'_2)\}}(L)$ is evaluated to $\{l'_1 = 40,\ l'_2 = 20\}$.

Quela is a statically typed language, and Fig. 4 lists a few interesting typing rules.

$$\frac{\Gamma \vdash M : \mathsf{Bag}\ A \quad \Gamma, x : A \vdash N : \mathsf{Bag}\ B}{\Gamma \vdash \textbf{for}(x \leftarrow M)\ N : \mathsf{Bag}\ B}\ \text{FOR} \qquad \frac{\Gamma \vdash M : \mathsf{Bag}\ \{\overline{l_i : O_i}\}}{\Gamma \vdash \textbf{exists}\ M : Bool}\ \text{EXISTS}$$

$$\frac{\Gamma \vdash L : \mathsf{Bag}\ \{\overline{l_i : O_i}\} \quad \alpha = \{(\overline{l_i, \circledcirc_i, l'_i})\} \quad \circledcirc_i : \mathsf{Bag}\ O_i \to O_i}{\Gamma \vdash \mathcal{A}_\alpha(L) : \{\overline{l'_i : O_i}\}}\ \text{AGGREGATION}$$

Fig. 4. Type System of Quela

The first typing rule represents the one for the **for**-construct, or bag comprehension. The term **for**$(x \leftarrow M)\ N$ computes a bag N for each element x in M, and takes the multi-set-union for all the results. Hence, M and N must have bag types, and x is bound in N. The second one is for the **exists** -construct. Here we need to constrain that the argument M must have a flat bag type (Notice that the type of each field is a basic type O_i). Otherwise, we cannot normalize such a term to an SQL query where nested data structures are not allowed. The third typing rule is for aggregate application $\mathcal{A}_\alpha(L)$. The A-Spec α specifies which aggregate function should be used for each field of the given record. The aggregate function $\circledcirc_i$ for the field l_i must have a function type $\mathsf{Bag}\ O_i \to O_i$, which must match the type of each field in L where O_i is a basic type. Note that the field names in α and those in the type of L must match, which means that we cannot throw away any fields by aggregation (no projection is allowed). This is again for the sake of guaranteeing the non-nested property of normal forms in Quela. This restriction does not affect the expressiveness of Quela, since we can always insert an explicit 'projections' as a normal term.

Other typing rules are standard in simply typed lambada calculus, and omitted here.

[4] In this study the set of aggregate functions is left unspecified, but we assume that they must operate on simple types. See the type system.

3.3 Normalization of Quela

Cooper has shown that any query of an appropriate type in his language can be normalized to a simple form which directly corresponds to a single SQL query, thus solving so called query avalanche problem.

We have the same property for Quela. More precisely, given a closed term in Quela which has a flat bag type (a bag-of-record type whose fields are of basic types) can be transformed to normal form, which is directly convertible to an SQL query. In the rest of this section, we explain how we can show this property. Note that we assume that the target SQL to allow subqueries (or nested queries), so nested control structures are not problematic. However, the normal form must not create or manipulate nested data structures (such as a record of records, or a table of tables), our goal is to eliminate the latter.

Fig. 11 in the appendix shows normalization rules essentially proposed by Cheney et al., after slight adjustment for Quela. For the newly added primitive $\mathcal{A}_\alpha(L)$ we do not have normalization rules[5] as explained in earlier section. (It is a 'barrier' for normalization.) Hence, we need to add the term $\mathcal{A}_\alpha(L)$ to the normal form of appropriate type. Fig.5 defines the normal form for Quela.

$$
\begin{array}{lll}
\text{Queries} & U & ::= U_1 \uplus U_2 \mid [\,] \mid F \\
\text{Comprehension} & F & ::= \textbf{for}(x \leftarrow H)\ F \mid H \mid Z \\
\text{Table} & H & ::= \textbf{table}(t) \\
\text{Body} & Z & ::= \textbf{where}\ B\ Z \mid \textbf{yield}\ R \\
\text{Record} & R & ::= \{\overline{l = B}\} \mid x \mid \mathcal{A}_\alpha(U) \\
\text{Primitives} & B & ::= \textbf{exists}\ U \mid \oplus(\overline{B}) \mid x.l \mid c \\
\text{A} - \text{Spec} & \alpha & ::= \{(\overline{l, \circledcirc, l'})\}
\end{array}
$$

Fig. 5. Normal form of Quela

Note that even if $\mathcal{A}_\alpha(U)$ is normal form (if U is), we only have to add it to R above. For this term, U must be of a flat bag type, and so is the whole term, hence no nested data structures are used in this term, which is the key of our proof of the non-nested property.

U in Fig.5 is a query expression of top-level and must be a bag with a flat type, that is U of $\mathcal{A}_\alpha(U)$ must be a bag type of flat record, and if U is not normal form, it is rewritten by normalization rules. This means that, although the return type is different, the existing syntax **exists** has the same properties as $\mathcal{A}_\alpha(U)$.

We will formally state the desirable properties on 'non-nestedness' of the result of our translation. Here, we take the minimalist approach, and we define flat types, instead of defining nested data types and non-nested data types in general. We call a record type whose components are basic types as flat record types. A flat bag type is the type $\mathsf{Bag}\ F$

[5] When L is an empty bag, we can transform the whole expression, but it is a special case which does not contribute general patterns.

where F is a flat record type. We sometimes say flat types for basic types, flat record types, or flat bag types. Flat types clearly do not contain nested structures, and provide data structures for our target SQL.

Theorem 1. *1. Normalization rules for Quela preserve typing, namely, $\Gamma \vdash L : A$ and $L \rightsquigarrow M$, then $\Gamma \vdash M : A$.*

2. For any typable term, normalization weakly terminates, namely, If $\Gamma \vdash L : A$, then there is a normal form N such that $L \rightsquigarrow N$.

3. Suppose N is normal form, and $\vdash N : F$ is derivable where F is a flat type. Then its type derivation contains only flat types.

The item 3 is crucial in our work, as it implies that all closed normal form of flat type does not use any nested data structures as intermediate data, which is necessary for it to be translated to a SQL query.

Let us briefly mention the proof of the theorem.

Item 1 can be proved by straightforward induction.

Item 2 is proved by making an analogy; Quela's aggregation is a 'barrier' for normalization since it has no transformation (normalization) rules. In Cheney et al.'s work, the **exists** primitive is the same, as it has no transformation rules other than the rare cases when its argument happens to be a value. Hence, as far as we are concerned with weak normalization property, we can treat our aggregation primitive just like the **exists** primitive, and the proof of Cheney et al.'s weak normalization theorem can be re-used without essential modification.

Item 3 is proved by induction on type derivation for a slightly stronger lemma: if $\Gamma \vdash N : \mathsf{Bag}\ \{\overline{l_i : O_i}\}$ is derivable for some $\Gamma = x_1 : F_1, \cdots, x_n : F_n$ where F_i are flat types, and N is normal form, then the typing derivation does not non-flat types. For this inductive proof, it is essential to have that our aggregation operator works over terms of flat types only. The rest of the proof is easy and omitted.

Note that, for this property to hold, we need to restrict the argument of **exists** primitive must have flat types; otherwise the item 3 does not hold in general.

The normal form is actually easy to translate to an SQL query. Fig.6 gives the translation for only one important case.

$$
\begin{array}{ll}
[\![\textbf{yield}\ \mathcal{A}_\alpha(U)]\!] & = \textbf{SELECT}\ \overline{\circledcirc(e)\ \textbf{AS}\ l'} \\
& \quad \textbf{FROM}\ \overline{t\ \textbf{AS}\ y}\ \ \textbf{WHERE}\ B \\
& \quad where\ [\![U]\!] = \textbf{SELECT}\ \overline{e\ \textbf{AS}\ l}\ \ \textbf{FROM}\ \overline{t\ \textbf{AS}\ y}\ \ \textbf{WHERE}\ B \\
& \quad and\ \alpha = \{(\overline{l, \circledcirc, l'})\}
\end{array}
$$

Fig. 6. Translation to SQL

For a normal form N, we write $[\![N]\!]$ for its translation to SQL. For $\mathcal{A}_\alpha(U)$, we first convert U to an SQL query, and then apply aggregate functions $\circledcirc_i$ for each e_i designated by the field l_i. We finally collect all fields and return the answer.

3.4 Comparison with Classic Results

The statement of Theorem 1 in the previous subsection resembles the 'conservativity' property studied in classical database theory. Among all, Libkin and Wong [5] formulated a simple calculus with aggregation, and proved that, for any query whose type has height n, there exists an equivalent query which has height n or lower. Here the height is the degree of nested data structures, and by taking n as 0, we have a statement which is very similar to the item 3 of Theorem 1.

However, there are several differences between Libkin and Wong's work and ours, and their result does not subsume ours (and vice versa).

The first difference is that the primitive data structure in their language is sets, whereas ours is bags (multi-sets). This different is minor, as we can adjust their theory to the one based on multi-sets.

The second, more essential difference is that they have more normalization rules than we have; one of their rules (in our syntax) is:

$$\mathcal{A}_{(\mathrm{l},\mathrm{SUM},\mathrm{l}')}(L_1 \cup L_2) = \mathcal{A}_{(\mathrm{l},\mathrm{SUM},\mathrm{l}')}(L_1) + \mathcal{A}_{(\mathrm{l},\mathrm{SUM},\mathrm{l}')}(L_2)$$

They regard this equation as a left-to-right rewrite rule. This rewriting often makes a query rather inefficient; it replaces an aggregation to normal addition, and if $L_1 \cup L_2$ in the above equation is replaced by a union of 100 bags, then the right-hand side will be sum of 100 elements, which is clearly slower to execute than a aggregate function.

The third difference is that their language does not have grouping, whereas ours has, as we will see in the next section. Adding grouping to the language would make the above efficiency problem even more serious; there is no simple way to express grouping and aggregation for $L_1 \cup L_2$ in terms of those for L_1 and those for L_2.

One may wonder why we successfully get the same (or very similar) theorem while our set of normalization rules is strictly smaller than theirs. The trick is that, our primitives for aggregation (and grouping) are finer than those primitives in existing studies.[6] The aggregation primitive in Libkin and Wong's study is:

$$\Sigma\{\{e_1 \mid x \in e_2\}\}$$

where e_1 and e_2 are expressions for queries and x is bound in e_1. The above primitive sums up the result of $e_1[a_i/x]$ for all $a_i \in e_2$. As the syntax reveals, their primitive can manipulate input by the expression e_1, which our primitive $(\mathrm{l}, \mathrm{SUM}, \mathrm{l}')$ cannot. In our language, constructing e_1 from $x \in e_2$ should be expressed by another expression (it can written as $\mathbf{for}(x \leftarrow e_2)\ [e_1]$ if we ignore labels and the difference of set and multi-set), and we can recover their aggregation primitive by combining these two. The bonus of this decomposition is Theorem 1.

In summary, while we obtained nothing new in theory (a very similar theorem is already known in old days, which can be adjusted to our setting), we claim that we have made solid progress towards a practical theory as advocated by Cheney et al., since much more efficient queries can be generated by our method. In the subsequent sections we will back up our claim by adding groping and showing performance.

[6] In the introduction of the present paper, we already explained it against SQL's GROUP BY.

4 Adding Grouping to the Language

The language Quela in the previous section does not have grouping, and this section extends it to the language Quelg, which has grouping.

One might think that this extension is a big step, however, surprising, the difference is quite small, since grouping with aggregation behaves quite similar to aggregation. It cannot be normalized but it works on terms of flat types, so it does not affect the important property that the normal form does not have nested data structures.

We briefly explain in this section the extended language and its properties.

The extended syntax is defined in Fig. 7. We introduce a new operator $\mathcal{G}_{(\kappa,\alpha)}(L)$ for grouping and aggregation, where κ is a list of field names, and represents the keys of this grouping, and α is the same as α in $\mathcal{A}_\alpha(L)$.

$$\text{Terms } L, M, N ::= ... \mid \mathcal{G}_{(\kappa,\alpha)}(L)$$

Fig. 7. Syntax of Quelg

Intuitively, $\mathcal{G}_{(\kappa,\alpha)}(L)$ gets an input table from (the result of computing) L, performs grouping based on the keys in κ, and then apply aggregate functions listed in κ. The result of this computation is a table whose element is a record consisting of the keys and the fields with the results of aggregate functions. As a simple example, the term $\mathcal{G}_{(\text{oid},\{(\text{qty},\text{MAX},\text{qty\_max})\})}(\textbf{table}(\text{“orders”}))$ evaluates to $[\{\text{oid} = 1, \text{qty\_sum} = 3\}, \{\text{oid} = 2, \text{qty\_sum} = 15\}, \{\text{oid} = 3, \text{qty\_sum} = 20\}]$.

Typing rules for Quelg are those for Quela plus the rule shown in Fig. 8.

$$\text{GROUPING}\quad \frac{\Gamma \vdash L : \mathsf{Bag}\ \{\overline{\kappa_i : O_i}, \overline{l_i : O'_i}\} \quad \kappa = \{\overline{l_i}\} \quad \alpha = \{(\overline{l_i, \circledcirc_i, l'_i})\} \quad \circledcirc_i : \mathsf{Bag}\ O'_i \to O'_i}{\Gamma \vdash \mathcal{G}_{(\kappa,\alpha)}(L) : \mathsf{Bag}\ \{\overline{\kappa_i : O_i}, \overline{l'_i : O'_i}\}}$$

Fig. 8. Type System of Quelg

The new typing rule is for grouping. To type $\mathcal{G}_{(\kappa,\alpha)}(L)$, we need to have L is a flat bag type $\mathsf{Bag}\ \{\overline{\kappa_i : O_i}, \overline{l_i : O'_i}\}$. The keys for grouping κ must appear in this list, and here we assume that κ_i appears in the first half of this sequence. The aggregate functions specified in α must be of function type from a bag of a basic type to the basic type. Finally, $\mathcal{G}_{(\kappa,\alpha)}(L)$ has the same type as L.

For the language Quelg, the normal form becomes a bit more complicated than those for Quela, because the new primitive for grouping returns a value of bag type, and it is still normal, hence each syntactic category of bag type must have the new primitive as normal form. Fig. 9 defines the normal form for Quelg.

$$
\begin{array}{lll}
\text{Queries} & U & ::= U_1 \uplus U_2 \mid [\,] \mid \mathcal{G}_{(\kappa,\alpha)}(U) \mid F \\
\text{Comprehension} & F & ::= \textbf{for}(x \leftarrow H)\ F \mid \mathcal{G}_{(\kappa,\alpha)}(U) \mid Z \\
 & H & ::= \textbf{table}(t) \mid \mathcal{G}_{(\kappa,\alpha)}(U) \\
\text{Body} & Z & ::= \textbf{where}\ B\ Z \mid \textbf{yield}\ R \mid \mathcal{G}_{(\kappa,\alpha)}(U) \\
\text{Record} & R & ::= \{\overline{l = B}\} \mid x \\
\text{Primitives} & B & ::= \textbf{exists}\ U \mid \oplus(\overline{B}) \mid x.l \mid c \\
 & \alpha & ::= \{(\overline{l, \circledcirc}, l')\} \\
 & \kappa & ::= \overline{l}
\end{array}
$$

Fig. 9. Normal form of Quelg

Finally, we define the translation from a normal form in Quelg to an SQL query. Fig. 10 defines the most interesting case.

$$
\begin{array}{ll}
[\![\mathcal{G}_{(\overline{\kappa},\alpha)}(U)]\!] & = \textbf{SELECT}\ \overline{\circledcirc(x.l)\ \textbf{AS}\ l'} \\
 & \quad \textbf{FROM}\ [\![U]\!]\ \textbf{AS}\ x \\
 & \quad \textbf{GROUP BY}\ \overline{x.\kappa} \\
 & \quad \textit{and}\ \alpha = \{(\overline{l, \circledcirc, l'})\}
\end{array}
$$

Fig. 10. Translation from Quelg to SQL

In the case of Quela, we analyzed the translation for U and added aggregate functions to them and we get a simple query (we do not have nested SELECT -statements). On the other hand, in Quelg, U in $\mathcal{G}_{(\overline{\kappa},\alpha)}$ may be translated to an SQL query with grouping, in which case we cannot translate the whole term to a non-nested SQL. Hence, we translate it to nested queries. In the right hand side of the definition in Fig. 10, $[\![U]\!]$ appears inside the FROM clause, which is a subquery. Note, however, that we can translate all normal form in Quelg to a single SQL query, thanks to the property that no nested data structures are used.

5 Implementation and Examples of Normalization

We have implemented normalization in Quelg, and translation to SQL. For this purpose, we embedded Quelg in the programming language OCaml using the tagless-final embedding following Suzuki et al.[3], and use PostgreSQL as the backend database server. The computing environment is Mac OS 10.13.2 with Intel Core i5-7360 CPU with memory 8GB RAM, and our programs run on OCaml 4.07.1, and generated SQL queries on PostgreSQL 11.5. The results of performance measurements will be shown after we explain example queries.

We prepare several concrete queries in Quelg which use aggregate functions and grouping in different ways. The database used here has two tables, the products table and the orders table in Section 2.

The first query Q'_3 accesses the orders table, and produces a bag of all orders whose oid matches the given value. The next query Q''_3 accesses the products table, and gets the $Orders$ record from Q'_3, finds all the products which has the same oid value as its input record o, and returns a bag of records with the oid and sales fields.

$$
\begin{aligned}
Q'_3 = \ & \lambda oid. \\
& \textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
& \textbf{where}\ (o.\text{oid} = oid) \\
& \textbf{yield}\ o
\end{aligned}
\qquad
\begin{aligned}
Q''_3 = \ & \lambda o. \\
& \textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
& \textbf{where}\ (p.\text{pid} = o.\text{pid}) \\
& \textbf{yield}\ \{\text{oid} = o.\text{oid}, \\
& \qquad \text{sales} = p.\text{price} * o.\text{qty}\}
\end{aligned}
$$

We want to compose these kinds of small queries to obtain a large, complicated query. It is easy to achieve in our language, since we can define a generic combinator for composition $compose$: for composition as follows:

$$compose = \lambda q.\ \lambda r.\ \lambda x.\ \textbf{for}(y \leftarrow q\ x)\ r\ y$$

Then we only have to apply it to Q'_3 and Q''_3 in this order to obtain a composed query. Here we also perform grouping and aggregation on the results, and we define a new query Q_3 as follows:

$$
\begin{aligned}
Q_3 = \ & \lambda x.\ \mathcal{G}_{(\text{oid},\alpha)}(compose\ Q'_3\ Q''_3\ x) \\
= \ & \lambda x.\ \mathcal{G}_{(\text{oid},\alpha)}(\textbf{for}(y \leftarrow Q'_3\ x)\ Q''_3\ y) \\
& \text{where } \alpha = \{(\text{sales}, \text{SUM}, \text{sales\_sum})\} \quad \ldots\ (1)
\end{aligned}
$$

We normalize Q_3 N (for a concrete value N) to obtain the following normal form:

$$
\begin{aligned}
Q_3 = \ & \mathcal{G}_{(\text{oid},\alpha)}(\textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
& \qquad \textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
& \qquad \textbf{where}\ (p.\text{pid} = o.\text{pid}\ \wedge\ o.\text{oid}\ >\ \text{N}) \\
& \qquad \textbf{yield}\ \{\text{oid} = o.\text{oid}, \text{sales} = p.\text{price} * o.\text{qty}\}) \\
& \text{Where } \alpha = \{(\text{sales}, \text{SUM}, \text{sales\_sum})\}
\end{aligned}
$$

which is immediately translated to SQL as:

```
SELECT o.oid AS oid, SUM(p.price * o.qty) AS sales_sum
FROM products AS p, orders AS o
WHERE p.pid = o.pid AND o.oid > N
GROUP BY o.oid                                    ... (2)
```

One can see how nested control structures are resolved to result in a flat-structured program.

After implementing our system, we have conducted performance measurement. We measured the execution time of program transformation and SQL generation in our implementation, and the execution time of generated SQL. We tested varying data size for the orders and products tables, ranging up to 10,000 records per table.

Table 1 shows the total execution time of the program transformations and SQL generation (from (1) to (2) in the above query). In addition to Q_1 to Q_3 in the previous chapters, we tested several more programs; Q_4 with lambda abstraction, Q_5 with a predicate, Q_6 to Q_8 with nested control structures, and Q_9 with a nested data structure. All queries of our system used in the experiment is available online at http://logic.cs.tsukuba.ac.jp/~rui/quelg/. In Table.1, we measured the time to iterate the transformation until the given term is in normal form, and we tested several cases where we allow only minimum required normalization rules, or all normalization rules. In Table.2, we measured the execution time of SQL.

Example	main rules	all rules
Q_1	0.028	0.029
Q_2	0.046	0.101
Q_3	0.11	1.15
Q_4	0.058	0.074
Q_5	0.019	0.19
Q_6	0.031	0.43
Q_7	0.052	0.043
Q_8	0.24	5.59
Q_9	0.32	9.31

All times in milliseconds.

Table 1. Time for Normalization and SQL generation

Example	time
Q_1	16.19
Q_2	14.07
Q_3	4.36
Q_4	0.95
Q_5	7.41
Q_6	14.14
Q_7	11.25
Q_8	18.04
Q_9	3732.62

| products | = 10000, | orders | = 10000

Table 2. Execution time for SQL

Table 1 confirms that such programs that require normalization takes longer times to generate SQL than those do not. However, compared to the time required to execute a query against a database, the time for optimizations and SQL generation is negligible, even if all normalization rules are used.

Table 2 shows the performance of subqueries (nested queries). For instance, Q_7 is a query which uses only one table, but has almost the same execution time as Q_1, which has two tables and has fewer subqueries than Q_7. It re-confirms the standard knowledge that executing subqueries in SQL takes a long time. The query Q_9, which calculates the average value in a subquery, takes about 3 seconds, and the execution time for subqueries is quite dependent on queries.

Although analyzing the execution time for different kinds of queries is beyond this work, we claim that our initial aim has been achieved, since all the queries used in this experiment have been converted to single SQL queries, which run on a common

database engine (PostgreSQL). It is an interesting future work to investigate how one can further optimize the generated SQL queries.

6 Conclusion

In this paper, we have given a tiny core language for language-integrated queries which has grouping and aggregate functions, while retaining the pleasant properties: any closed term of flat type (a bag-of-record type whose component types are basic types) can be normalized to a normal form, which corresponds to a single query in SQL where subqueries are allowed.

The key idea of this study is to decouple a complex job of SQL's GROUP BY clause into two pieces: One is grouping and aggregation which cannot be normalized but have simple types. The other is output construction which can be normalized but have complex types. By this decoupling, we have succeeded in getting the properties achieved in the earlier work for the language without grouping and aggregation. Our language is not as expressive as the language with the full GROUP BY clause, but by simply rewriting queries using such clauses, our language can host most such queries. To our knowledge, this work is the first success case of (subset of) language-integrated query which has the above pleasant property.

We have implemented our language by embedding our language Quelg in a host language OCaml. We have shown a concrete example and the result of simplistic performance test.

There are many directions for future work, among which the most important ones are performance evaluation against larger examples, optimization of generated SQL, and thorough comparison with other frameworks. Extending our language to cover other complicated database primitives is also an interesting next step.

Acknowledgements. We would like to thank Oleg Kiselyov and Kenichi Suzuki for development of Quel and its tagless-final implementation. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) No. 18H03218.

References

1. Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *Database Programming Languages - DBPL 2009, 12th International Symposium, Lyon, France, August 24, 2009. Proceedings*, pp. 36–51, 2009.
2. James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pp. 403–416, 2013.
3. Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 37–48, 2016.
4. Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, Vol. 52, No. 3, pp. 495–505, 1996.

5. Leonid Libkin and Limsoon Wong. Aggregate functions, conservative extensions, and linear orders. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pp. 282–294. Springer, 1993.
6. Simon L. Peyton Jones and Philip Wadler. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pp. 61–72, 2007.
7. Oleg Kiselyov and Tatsuya Katsushima. Sound and efficient language-integrated query - maintaining the ORDER. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, pp. 364–383, 2017.
8. Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Vol. 19, pp. 509–543, 2009.
9. Microsoft corporation. the linq project: .net language integrated query. September 2005.
10. Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, Vol. 3, No. 1, pp. 162–172, 2010.
11. Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *Database Theory - ICDT'92, 4th International Conference, Berlin, Germany, October 14-16, 1992, Proceedings*, pp. 140–154, 1992.
12. James Cheney, Sam Lindley, and Philip Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 1027–1038, 2014.
13. Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .net framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, p. 706, 2006.
14. Atsushi Ohori, Peter Buneman, and Val Tannen. Database programming in machiavelli - a polymorphic language with static type inference. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989.*, pp. 46–57, 1989.

A Normalization Rules of Quel

Normalization rules of Quel are given as follows:

$$
\begin{array}{rll}
\textbf{(Stage 1)} & & \\
(\lambda x.N)\ M & \rightsquigarrow N[x := M] & (\text{ABS}{-}\beta) \\
\{\overline{l = M}\}.l_i & \rightsquigarrow M_i & (\text{RECORD}{-}\beta) \\
\textbf{for}(x \leftarrow \textbf{yield}\ M)\ N & \rightsquigarrow N[x := M] & (\text{FORYIELD}) \\
\textbf{for}(x \leftarrow \textbf{for}(y \leftarrow L)\ M)\ N & \rightsquigarrow & \\
\quad \textbf{for}(y \leftarrow L)\ \textbf{for}(x \leftarrow M)\ N & \ (\text{if } y \notin FV(N)) & (\text{FORFOR}) \\
\textbf{for}(x \leftarrow \textbf{where}\ L\ M)\ N & \rightsquigarrow \textbf{where}\ L\ \textbf{for}(x \leftarrow M)\ N & (\text{FORWHERE}_1) \\
\textbf{for}(x \leftarrow [])\ N & \rightsquigarrow [] & (\text{FOREMPTY}_1) \\
\textbf{for}(x \leftarrow M_1 \uplus M_2)\ N & \rightsquigarrow & \\
\quad \textbf{for}(x \leftarrow M_1)\ N \uplus \textbf{for}(x \leftarrow M_2)\ N & & (\text{FORUNIONALL}_1) \\
\textbf{where true}\ M & \rightsquigarrow M & (\text{WHERETRUE}) \\
\textbf{where false}\ M & \rightsquigarrow [] & (\text{WHEREFALSE}) \\
 & & \\
\textbf{(Stage 2)} & & \\
\textbf{for}(x \leftarrow M)\ (N_1 \uplus N_2) & \hookrightarrow & \\
\quad \textbf{for}(x \leftarrow M)\ N_1 \uplus \textbf{for}(x \leftarrow M)\ N_2 & & (\text{FORUNIONALL}_2) \\
\textbf{for}(x \leftarrow M)\ [] & \hookrightarrow [] & (\text{FOREMPTY}_2) \\
\textbf{where}\ L\ M \uplus N & \hookrightarrow & \\
\quad (\textbf{where}\ L\ M) \uplus (\textbf{where}\ L\ N) & & (\text{WHEREUNION}) \\
\textbf{where}\ L\ \textbf{where}\ M\ N & \hookrightarrow \textbf{where}\ L \wedge M\ N & (\text{WHEREWHERE}) \\
\textbf{where}\ L\ \textbf{for}(x \leftarrow M)\ N & \hookrightarrow & \\
\quad \textbf{for}(x \leftarrow M)\ \textbf{where}\ L\ N & & (\text{WHEREFOR})
\end{array}
$$

Fig. 11. Normalization rules of Quela