Sound and Efficient Language-Integrated Query
Maintaining the `ORDER`

Oleg Kiselyov and Tatsuya Katsushima

Tohoku University, Japan
oleg@okmij.org

Abstract. As SQL moved from the English-like language for ad hoc queries by business users to its present status as the universal relational database access, the lack of abstractions and compositionality in the original design is felt more and more acute. Recently added subqueries and common table expressions compensate, albeit generally inefficiently. The inadequacies of SQL motivated language-integrated query systems such as (T-)LINQ, which offer an applicative, programming-like query language compiled to efficient SQL.
However, the seemingly straightforward ranking operations ORDER BY and LIMIT are not supported efficiently, consistently or at all in subqueries. The SQL standard defines their behavior only when applied to the whole query. Language-integrated query systems do not support them either: naively extending ranking to subexpressions breaks the distributivity laws of UNION ALL underlying optimizations and compilation.
We present the first compositional semantics of ORDER BY and LIMIT, which reproduces in the limit the standard-prescribed SQL behavior but also applies to arbitrarily composed query expressions and preserves the distributivity laws. We introduce the relational calculus SQUR that includes ordering and subranging and whose normal forms correspond to efficient, portable, subquery-free SQL. Treating these operations as effects, we describe a type-and-effect system for SQUR and prove its soundness. Our denotational semantics leads to the provably correctness-preserving normalization-by-evaluation. An implementation of SQUR thus becomes a sound and efficient language-integrated query system maintaining the `ORDER`.

1 Introduction

Language-integrated query is "smooth integration" of database queries with a conventional programming language [1, 4]. 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. Reflecting operations on database relations in the type system is the old problem with good solutions [3]. Query reuse and composition, detailed below, is more difficult: as we are about to see, the seemingly straightforward operation to sort the query results – patterned after ORDER BY of SQL – wreaks havoc with the existing approaches to query composition. It hence acts quite like a side effect. Seemingly simple, the more one (or stackoverflow users) thinks about it, the more problematic edge cases come to light. A formal treatment is called for, which we develop in this paper in terms of a denotational approach.

1.1 Query composition

Since query composition is the central, and subtle problem, we introduce it in more detail. The query language of this section is SQL, although the rest of the paper uses the calculus SQUR that is more regular and suited for formal analysis. However, an implementation of SQUR still eventually produces SQL, the *lingua franca* of relational databases. Therefore, understanding the SQL behavior is crucial. Besides, SQL compositions here and their problems in §1.2 occur in the wild, as seen, e.g., on `stackoverflow` and in training literature.

Consider a sample database table employee that lists, among others, employee names, their departments and the hourly wage. The following query then

$$\mathsf{QE} \overset{def}{=} \mathsf{SELECT\ E.{*}\ FROM\ employee\ \textbf{as}\ E\ WHERE\ E.wage} > 20$$

reports employees paid at twice the minimum wage. One may also think of the query as building a filtered table: a subset of well-paid employees. As typical, the department in the employee table is mentioned by its deptID and so not informative for people outside the company. Suppose there is a table department with the descriptive name for each department, along with other data. A more presentable employee report is then obtained by the so-called join query, which all relational database systems are meant to run well:

$$\begin{aligned}\mathsf{QD} \overset{def}{=}\ & \mathsf{SELECT\ E.{*},\ D.name\ FROM\ employee\ \textbf{as}\ E,\ department\ \textbf{as}\ D}\\ & \mathsf{WHERE\ E.deptID = D.deptID}\end{aligned}$$

Queries QE and QD are useful on their own; one may imagine them stored in a library. It is also useful to combine them, so to report well-paid employees with descriptive department names. We would like to reuse the queries without changing them except for substituting table names.

There have been three approaches to query composition, shown in the order of increasing performance[1].

temporary table One may store the results of QE in a (temporary) table subemp and then perform QD in which employee is replaced with the filtered subemp. The latter table can be 'virtual': a (non-materialized) view or a so-called common table expression shown below:

[1] To obtain an informative report one may, in principle, run QE first and for each resulting record, query department for the descriptive department name. Such 'composition' essentially executes as many department queries as there are well-paid employees and leads to the 'query avalanche' well explained in [4]. In this paper we consider only those compositions that result in a single query.

with subemp $\{QE\}$ $QD[employee := subemp]$

where QE and QD above are meant to be replaced with the respective SQL statements; $[old := new]$ stands for substitution. No full subemp is created; instead, QE runs incrementally.

subquery Another way to compose QE and QD is to consider the query QE as a table on its own and substitute it for the original employee table in QD:

$QD[employee := QE]$

Thus QE will run as a subquery of QD[2]. Subqueries have generally better performance: the optimizer may rewrite a subquery into a simple join (like the one shown below), which can then be further optimized. The rewriting is by no means guaranteed[3].

rewriting into flat SQL After composing QE and QD as above, rather than sending the result to the ('black-box') database engine we re-write it ourselves into a subquery-free (i.e., 'flat') SQL using equational laws:

```
SELECT E.*, D.name FROM employee as E, department as D
   WHERE E.deptID = D.deptID AND E.wage > 20
```

This is the most performant approach, and in fact recommended in vendor and training literature. The database user is supposed to do the rewriting. Language-integrated query, as an intermediary between the programmers and SQL, aims to isolate them from the vagaries of SQL – in particular, to perform such re-writing automatically. We demonstrate it in §2.

1.2 Ordering

Ordering of the query results at first glance appears straightforward: for example, the following modification to QE

QEO $\overset{def}{=}$ SELECT E.∗ FROM employee **as** E WHERE E.wage $>$ 20 ORDER BY E.wage

lists employees in the increasing order of their wages. As for instance, the Oracle documentation puts it, "Use the ORDER BY clause to order rows returned by the [SELECT] statement. Without an ORDER BY clause, no guarantee exists that the same query executed more than once will retrieve rows in the same order."[4] It takes time for the second part of the quote to sink in, however. We shall see the examples momentarily.

This seemingly simple ORDER BY feature wrecks all three approaches to compositionality described above. Consider the temporary table approach, with the ordered QEO in place of QE: **with** subemp $\{QEO\}$ $QD[employee := subemp]$.

[2] Such subqueries in the FROM SQL clause are sometimes called 'derived tables' or 'inlined views'.

[3] For example, MySQL 5.7 never re-writes and hence optimizes subqueries in the FROM clause: `http://dev.mysql.com/doc/refman/5.7/en/subquery-restrictions.html`.

[4] `https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_10002.htm\#sthref6708`

To order the filtered employee table, it first has to be fully constructed, and then sorted (at least that is what PostgreSQL does). Therefore, QEO cannot run incrementally; it requires space for the filtered data and sorting. Unfortunately, this expense is all for naught: the rows of the entire composed query are still reported in unpredictable order. Recall, unless ORDER BY is attached to the whole query (QD is our example), nothing certain can be said about the order of its results. Expensive useless operations are not the feature one wishes for in a practical system.

If we substitute QEO for employee as a (derived) table in QD, we end up with a subquery containing ORDER BY. According to the Microsoft SQL server documentation, "The ORDER BY clause is not valid in views, inline functions, derived tables, and subqueries, unless either the TOP or OFFSET and FETCH clauses are also specified. When ORDER BY is used in these objects, the clause is used only to determine the rows returned by the TOP clause or OFFSET and FETCH clauses. The ORDER BY clause does not guarantee ordered results when these constructs are queried, unless ORDER BY is also specified in the query itself."[5] Thus composing of ORDER BY queries is not just about poor performance. It could be no performance at all: dynamic error when trying to run the composed query.

Attempting to rewrite the ordered subquery into flat SQL is just as problematic: Re-writing relies on equational laws [6], many of which are no longer valid in the presence of ordering naively extended to subqueries. As an example, consider one of the equational laws (called ForUnionAll1 in [6, 13]), applied when a subquery is a UNION ALL of two smaller queries Q1 and Q2:

```
SELECT ∗ FROM (Q1 UNION ALL Q2) WHERE exp ≡
   (SELECT ∗ FROM Q1 WHERE exp) UNION ALL (SELECT ∗ FROM Q2 WHERE exp)
```

For the ordered query, the analogous distributivity no longer holds:

```
SELECT ∗ FROM (Q1 UNION ALL Q2) WHERE exp ORDER BY eo ≢
(SELECT ∗ FROM Q1 WHERE exp ORDER BY eo) UNION ALL
   (SELECT ∗ FROM Q2 WHERE exp ORDER BY eo)
```

Indeed, the left-hand-side query gives the results in a definite order whereas the right-hand-side does not. (With subranging, LIMIT... OFFSET, the failure of distributivity is even more evident.) Incidentally, in our semantics of ordering and subranging, distributivity does hold: see §4.

Composing queries with ORDER BY is not contrived: 'real-life' database programmers regularly do that and are regularly confused by the inconsistent, puzzling, vendor- and version–specific responses of database engines, as amply documented in stackoverflow and other fora. For example,

[5] https://msdn.microsoft.com/en-us/library/ms188385.aspx

> "Query with ORDER BY in a FROM subquery produces unordered result. Is this a bug? Below is an example of this:
>
> ```
> SELECT field1, field2
>   FROM ( SELECT field1, field2 FROM table₁ ORDER BY field2) alias
> ```
>
> returns a result set that is not necessarily ordered by field2."[6]
> "I am using an application (MapServer) that wraps SQL statements, so that the ORDER BY statement is in the inner query. E.g.
>
> ```
> SELECT * FROM (SELECT ID, GEOM, Name FROM t ORDER BY Name) as tbl
> ```
>
> The application has many different database drivers. I mainly use the MS SQL Server driver, and SQL Server 2008. This throws an error if an ORDER BY is found in a subquery."[7]

One may find many more such discussions[8] by googling "ORDER BY derived table". The semantics presented in the paper not only answers these questions but also shows how to transform the problematic queries into portable SQL.

1.3 Contributions

If we are to talk to a relational database and obtain sorted results, we have to send the SQL code with ORDER BY attached where the SQL standard expects to see it, to the whole SELECT statement. If we are to freely compose and reuse the already built queries, we quickly end up with ORDER BY appearing in subexpressions, often where the SQL standard does not expect to see it. It is not clear what such composed queries actually mean, let alone how to transform them to the ones that can be *portably* executed on various database systems. (Many systems, e.g., PostgreSQL, Oracle, DB2, and MySQL support nested ORDER BY, but *each* differently.) In short, the problem is that neither the SQL standard, nor its realizations, nor the existing language-integrated query systems supporting ordering give a compositional semantics to ORDER BY.

We present the first compositional treatment of ORDER BY and the related LIMIT...OFFSET – and its application for optimizing composed queries to yield efficient and *portable* SQL. Specifically,

- We present a new language-integrated query calculus SQUR[9] with the denotational, and hence, compositional semantics. The semantics is strikingly, ridiculously simple, requiring *no* domain calculus, no complete partial orders, no lifted types. As the price for simplicity, bare SQUR, like the old SQL, lacks abstractions and is cumbersome to use. Unlike SQL, however, SQUR

[6] `https://mariadb.com/kb/en/mariadb/why-is-order-by-in-a-from-subquery-ignored/`

[7] `http://dba.stackexchange.com/questions/82930/database-implementations-of-order-by-in-a-subquery`

[8] `http://stackoverflow.com/questions/18031421/the-order-by-clause-is-invalid-in-views-inline-functions-derived-tables-subqu`
`https://www.sqlservercentral.com/Forums/Topic1079476-391-1.aspx`

[9] The name, just like QUEΛ of [13], is a pun for old-timers

is composable and hence easily embeddable into a rich metalanguage, which provides modules, combinators, and other syntax sugar. To demonstrate[10], we have embedded SQUR into OCaml (see below).

- Using the non-standard domain of partially-known values we design a normalization-by-evaluation algorithm, which normalizes a SQUR query into the form easily convertible to the *portable* flat SQL. We hence re-implement QUEΛ [13] and reproduce its (and T-LINQ [4]) results. The earlier research relied on the (syntactic) normalization-by-rewriting and hence had to contend with confluence and termination. Our, semantic, normalization is deterministic by construction and is easily shown total;
- We extend the core SQUR with ordering and subranging, maintaining the denotational, compositional treatment and the normalization-by-evaluation. Our semantics of ranking coincides with the SQL semantics when applied to the whole query. Our semantics preserves the distributivity laws of UNION ALL whereas the naively understood ordering and subranging do not.
- We treat ordering and subranging as effects, and design the corresponding type-and-effect system. The types tell when subqueries can be eliminated and when they have to be turned into virtual tables (common-table expressions).

The calculus has been implemented by embedding into OCaml in the tagless-final style, resulting in a sound and efficient language-integrated–query system. The implementation is available at `http://okmij.org/ftp/meta-programming/Sqr/` but is not described here for the lack of space.

The structure of the paper is as follows. The next section introduces the core SQUR: first, by examples and then formally. We give the denotational semantics, the type system, and the semantic proof of the type soundness. §3 describes the normalization-by-evaluation for the core calculus, and the generation of flat SQL. §4 adds sorting and the limiting of the query output. We then review the related work and conclude.

2 Core SQUR

This section presents the core calculus, to be extended with ranking in §4.

Before the formal presentation (formal definition in §2.1, type system §2.2, denotational semantics §2.3), we introduce SQUR informally, on examples from §1.1. The query QE that listed high-paid employees looks in SQUR as follows:

```
defn Qe:
for(e ← table employee) where e.wage>20 yield e
```

The first line is not part of SQUR (hence the different font): it is mere a presentation tool to attach the name to a query for easy reference. One may think that SQUR is to be subjected to a C-like preprocessor (although we certainly

[10] A simpler example of embedding a bare, first-order calculus into a rich, functional (meta)language is described in `http://okmij.org/ftp/tagless-final/nondet-effect.html`.

have in mind something more refined: see the accompanying code). The query QD attaching meaningful department names to employees is

```
defn Q_d emp:
 for (e ← emp) for (d ← table department)
 where e.deptID = d.deptID yield <name=e.name, dep=d.name, wage=e.wage>
```

This definition is parameterized by the source of the employee records.

For the presentable report of high-paid employees we compose the two queries, simply as Q_d Q_e. After desugaring/preprocessing (i.e., inlining the definitions and substituting the parameters) we obtain

```
for (e ← for(e ← table employee) where e.wage>20 yield e)
for (d ← table department)
 where e.deptID = d.deptID yield <name=e.name, dep=d.name, wage=e.wage>
```

which can be interpreted as SQL with the nested SELECTs (derived tables), as described in §1.1. The paper [13] has explained in detail why subqueries are suboptimal. Normalizing that query gives

```
for (e ← table employee)
for (d ← table department)
 where e.deptID = d.deptID && e.wage>20
 yield <name=e.name, dep=d.name, wage=e.wage>
```

which corresponds to the simple and efficient flat query at the end of §1.1. §3 explains the normalization, and §3.2 SQL generation, in detail. But first we have to formally introduce SQUR.

2.1 SQUR, Formally

The basic SQUR is formally defined in Fig. 1. It is reminiscent of nested relational calculus [14]. One of the most glaring, although essentially minor differences, is the absence of lambda-abstractions and applications. SQUR hence is not (an extension of) lambda-calculus. Database systems generally do not support first-class functions in queries; hence the lack of lambda-abstractions in SQUR is not an expressiveness limitation in that respect. Furthermore, SQUR is designed for language-*integrated* queries: it is intended to be 'preprocessed', that is, embedded into a host language, and hence to take advantage of the host language's abstraction mechanisms such as first-class functions, modules, etc.

We use x,y for variables, c for integer, boolean, etc. and table constants, n and m for numeric literals, l for record labels. The sequence of items $\mathsf{e}_1,\ldots,\mathsf{e}_n$ is abbreviated as e,.... For clarity, Fig. 1 defines only one basic operation: addition. Others are analogous and are silently added when needed. Types of SQUR are base types b, record types <l:b,...> where $\mathsf{l}_1,\ldots,\mathsf{l}_n$ are field labels, and bag types t bag where t is a base or a record type. Bag types can be annotated with the set of effects ϵ; if empty, it is frequently elided. Effects come into play only in §4; for now we can assume them empty and ignore.

Besides constants, variable references and primitive operations, the language supports record construction $<\mathsf{l}_1=\mathsf{e}_1,\mathsf{l}_2=\mathsf{e}_2,\ldots>$ and record field l_i projection

Variables	x,y,z...
Constants	c (integers, booleans, tables, etc.)
Numeric Literals	n, m
Record Labels	l
Effect Annotations	ϵ
Base Types	b ::= int | bool | string
Flat Types	t ::= b | <l:b,...>
Types	s ::= t | t bag^ϵ | t tbl
Type Environment	Γ ::= x:t, y:t tbl, ...
Expressions	e ::= c | x | e + e | <l=e,...> | e.l | **for**(x←e) e | e ⊎ e | **where** e e | **yield** e | table e

Fig. 1. Syntax of core SQUR

e.l$_i$, bag comprehensions **for**(x←e$_1$) e$_2$, and the bag concatenation e$_1$ ⊎ e$_2$ patterned after SQL's UNION ALL. The body of **for** extends as far to the right as possible (similarly for **where** and **yield**). Intuitively, **where** e$_1$ e$_2$ evaluates to the empty bag if e$_1$ is false; **yield** e produces the singleton bag with the result of e. The language has table constants representing database tables (plus the special constant bag_empty) with their own type t tbl. The expression table e turns a table into a bag; the need to distinguish table constants comes during normalization and SQL conversion. Since the language has no first-class functions, we assume from the outset that all variable names (appearing in comprehensions) are unique.

2.2 Type System

The type system is presented in Fig. 2. The (Const) rule shows the typing of a single base-type constant **true** and the single table constant employee. Other constants are analogous. The type environment Γ contains only the bindings with flat types and table types (we see the latter bindings only in §4). As one may expect from the typing of empty containers in general, bag_empty can bear any element type and any effect annotation. From now on, we only deal with well-typed SQUR terms.

The next section presents the (denotational) dynamic semantics and proves the type system sound.

2.3 Denotational Semantics

The denotational semantics of SQUR is set-theoretic and Church-style (that is, only typed expressions are given meaning). Fig. 3 presents semantic domains and defines $\mathcal{T}$[s] that maps SQUR's type s to a semantic domain, which is an ordinary set. If A_1 and A_2 are sets, we write l$_1$:A_1 × l$_2$:A_2 for a labeled product: the set of pairs <l$_1$:a_1,l$_2$:a_2>, $a_1 \in A_1, a_2 \in A_2$. The components of the pair are identified by their labels l$_i$ rather than their position. If p is such a labeled pair, we write p.l$_i$ to access the l$_i$-th component, and p × l$_3$:a_3 to extend the

$$\frac{}{\Gamma \vdash \textbf{true}\text{: bool}}\ \text{Const} \qquad \frac{}{\Gamma \vdash \text{employee: <name:string, deptID:int , wage:int> tbl}}\ \text{Const}$$

$$\frac{}{\Gamma \vdash \text{bag\_empty: t bag}\hat{\ }\epsilon}\ \text{Empty} \qquad \frac{\text{x:t} \in \Gamma}{\Gamma \vdash \text{x: t}}\ \text{Var} \qquad \frac{\Gamma \vdash \text{e}_1\text{: int} \quad \Gamma \vdash \text{e}_2\text{: int}}{\Gamma \vdash \text{e}_1 + \text{e}_2\text{: int}}\ \text{Op}$$

$$\frac{\Gamma \vdash \text{e: t tbl}}{\Gamma \vdash \text{table e : t bag}\hat{\ }\phi}\ \text{Table} \qquad \frac{\Gamma \vdash \text{e: b} \ldots}{\Gamma \vdash \text{<l=e,}\ldots\text{>: <l:b,}\ldots\text{>}}\ \text{Rec} \qquad \frac{\Gamma \vdash \text{e: <l:b,}\ldots\text{>}}{\Gamma \vdash \text{e.l}_i\text{: b}_i}\ \text{Proj}$$

$$\frac{\Gamma \vdash \text{e}_1\text{: t bag}\hat{\ }\epsilon \quad \Gamma \vdash \text{e}_2\text{: t bag}\hat{\ }\epsilon}{\Gamma \vdash \text{e}_1 \uplus \text{e}_2\text{: t bag}\hat{\ }\epsilon}\ \text{UnionAll} \qquad \frac{\Gamma \vdash \text{e:t}}{\Gamma \vdash \textbf{yield}\text{ e: t bag}\hat{\ }\phi}\ \text{Yield}$$

$$\frac{\Gamma \vdash \text{e}_1\text{: bool} \quad \Gamma \vdash \text{e}_2\text{: t bag}\hat{\ }\epsilon}{\Gamma \vdash \textbf{where}\text{ e}_1\text{ e}_2\text{: t bag}\hat{\ }\epsilon}\ \text{Where} \qquad \frac{\Gamma \vdash \text{e}_1\text{: t}_1\text{ bag}\hat{\ }\phi \quad \Gamma\text{,x:t}_1 \vdash \text{e}_2\text{: t}_2\text{ bag}\hat{\ }\epsilon}{\Gamma \vdash \textbf{for}(\text{x}\leftarrow\text{e}_1)\text{ e}_2\text{: t}_2\text{ bag}\hat{\ }\epsilon}\ \text{For}$$

Fig. 2. Type system

pair with a new component. We write $\{\{a,\ldots\}\}$ for a multiset with elements a_i, and $\{\{A\}\}$ for the set of multisets whose elements come from the set A.

$\mathcal{T}$[int]	$= \mathbb{N}$	set of integers
$\mathcal{T}$[bool]	$= \{T, F\}$	booleans
$\mathcal{T}$[string]	$= \mathbb{S}$	set of strings
$\mathcal{T}$[<l$_1$:b$_1$,…,l$_n$:b$_n$>]	$=$ l$_1$:$\mathcal{T}$[b$_1$] $\times \cdots \times$ l$_n$:$\mathcal{T}$[b$_n$]	labeled product
$\mathcal{T}$[t tbl]	$= \{\{\mathcal{T}[\text{t}]\}\}$	set of multisets of elements of type $\mathcal{T}$[t]
$\mathcal{T}$[t bag]	$= \{\{\mathcal{T}[\text{t}]\}\}$	set of multisets of elements of type $\mathcal{T}$[t]
$\mathcal{T}$[x$_1$:t$_1$,…,x$_n$:t$_n$]	$=$ x$_1$:$\mathcal{T}$[t$_1$] $\times \cdots \times$ x$_n$:$\mathcal{T}$[t$_n$]	interpretation of the environment

Fig. 3. Semantic domains and the interpretation of types

The semantic function $\mathcal{E}[\Gamma \vdash \text{e:s}]\ \rho_\Gamma$ in Fig. 4 maps a type judgment and the environment to an element of $\mathcal{T}$[s]. Here ρ_Γ is an element of the labeled product $\mathcal{T}[\Gamma]$. We added the if-then-else conditional to our mathematical notation for writing denotations, overload $\cup$ to mean the set or multiset union, and write $\{\{\ldots \mid x\leftarrow A\}\}$ for a multiset comprehension.

It is clear from Fig. 4 that $\mathcal{E}[-]\rho$ is the total map. Hence

Theorem 1 (Type Soundness). *For any $\rho_\Gamma \in \mathcal{T}[\Gamma]$, $\mathcal{E}[\Gamma \vdash \text{e:s}]\ \rho_\Gamma \in \mathcal{T}[\text{s}]$.*

The semantics of e$_1$ $\uplus$ e$_2$ clearly shows that the UNION ALL operation is associative and commutative. Moreover, the following distributive laws hold (which is easy to verify by applying the semantic function to both sides of the equations). These laws, among others, underlie the normalization-by-rewriting of [4, 6].

Theorem 2 (Distributive Equational Laws of UNION ALL).

$$\begin{array}{lclcl}
\textbf{for}(x \leftarrow e_1 \uplus e_2)\ e & \equiv & (\textbf{for}(x\leftarrow e_1)\ e) & \uplus & (\textbf{for}(x\leftarrow e_2)\ e) \\
\textbf{for}(x \leftarrow e)\ e_1 \uplus e_2 & \equiv & (\textbf{for}(x\leftarrow e)\ e_1) & \uplus & (\textbf{for}(x\leftarrow e)\ e_2) \\
\textbf{where}\ e\ e_1 \uplus e_2 & \equiv & (\textbf{where}\ e\ e_1) & \uplus & (\textbf{where}\ e\ e_2)
\end{array}$$

$$
\begin{array}{lcl}
\mathcal{E}[\Gamma \vdash \mathsf{c\colon s}]\ \rho & \in & \mathcal{T}[\mathsf{s}] \\
\mathcal{E}[\Gamma \vdash \mathsf{bag\_empty\colon t\ bag}]\ \rho & = & \{\{\}\} \\
\mathcal{E}[\Gamma \vdash \mathsf{x\colon t}]\ \rho & = & \rho.\mathsf{x} \\
\mathcal{E}[\Gamma \vdash \mathsf{e_1 + e_2\colon int}]\ \rho & = & \mathcal{E}[\Gamma \vdash \mathsf{e_1\colon int}]\rho + \mathcal{E}[\Gamma \vdash \mathsf{e_2\colon int}]\rho \\
\mathcal{E}[\Gamma \vdash \mathsf{<l=e,\ldots>\colon <l{:}b,\ldots>}]\ \rho & = & \mathsf{<l{:}}\mathcal{E}[\Gamma \vdash \mathsf{e{:}b}]\rho,\ldots\mathsf{>} \\
\mathcal{E}[\Gamma \vdash \mathsf{e.l_\mathit{i}\colon b_\mathit{i}}]\ \rho & = & (\mathcal{E}[\Gamma \vdash \mathsf{e\colon <l{:}b,\ldots>}]\rho).\mathsf{l}_i \\
\mathcal{E}[\Gamma \vdash \mathsf{e_1 \uplus e_2\colon t\ bag}]\ \rho & = & \mathcal{E}[\Gamma \vdash \mathsf{e_1\colon t\ bag}]\rho \cup \mathcal{E}[\Gamma \vdash \mathsf{e_2\colon t\ bag}]\rho \\
\mathcal{E}[\Gamma \vdash \textbf{yield}\ \mathsf{e\colon t\ bag}]\ \rho & = & \{\{\ \mathcal{E}[\Gamma \vdash \mathsf{e\colon t}]\rho\ \}\} \\
\mathcal{E}[\Gamma \vdash \textbf{where}\ \mathsf{e_1\ e\colon t\ bag}]\ \rho & = & \textbf{if}\ \mathcal{E}[\Gamma \vdash \mathsf{e_1\colon bool}]\rho\ \textbf{then}\ \mathcal{E}[\Gamma \vdash \mathsf{e\colon t\ bag}]\rho\ \textbf{else}\ \{\{\}\} \\
\mathcal{E}[\Gamma \vdash \mathsf{table\ e\colon t\ bag}]\ \rho & = & \mathcal{E}[\Gamma \vdash \mathsf{e\colon t\ tbl}]\rho \\
\mathcal{E}[\Gamma \vdash \textbf{for}(\mathsf{x{\leftarrow}e_1})\ \mathsf{e\colon t\ bag}]\ \rho & = & \bigcup\{\{\ \mathcal{E}[\Gamma,\mathsf{x{:}t_1} \vdash \mathsf{e\colon t\ bag}]\ (\rho \times \mathsf{x} : x')\ \mid \\
 & & \qquad x'{\leftarrow}\mathcal{E}[\Gamma \vdash \mathsf{e_1\colon t_1\ bag}]\rho\ \}\}
\end{array}
$$

Fig. 4. Denotational semantics of Core SQUR

3 Normalization-by-Evaluation

The just described denotational semantics may be regarded as an interpreter of SQUR queries over an in-memory database of lists of records. Our motivation however is to run SQUR over external relational databases. Therefore, this section interprets SQUR expressions as SQL statements – i.e., gives a different semantics to SQUR, over the domain of SQL queries.

The problem is not trivial: as we saw in §2 earlier and see again later, only a subset of SQUR expressions can be easily translated to 'flat' SQL queries. A solution suggested in Cooper [6] is to re-write, if possible, the expressions outside the easily-translatable subset into that good subset. For the simple language corresponding to our Core SQUR, Cooper (later followed by [4]) introduced a set of rewriting rules, proved they are type- and semantic- preserving, confluent and terminating, and the resulting normal forms are easily-translatable to SQL. No such rules are known for the language extended with ordering, grouping, etc.

We take an approach radically different from [6] and its followers: semantic, rather than syntactic. We *start* with the 'normal form' for SQUR expressions, specifically designed to be easily convertible to SQL. We then show how to compute that normal form, through deterministic evaluation. The totality of evaluation proves that all SQUR expressions are translatable to SQL. The denotational approach makes it rather easy to show the type and semantics preservation of this normalization by evaluation.

We define the normalization-by-evaluation as giving another, non-standard interpretation of SQUR, into different semantic domains. This section defines new semantic functions $\mathcal{T}^n[-]$ and $\mathcal{E}^n[-]$, clause by clause. The environments and variables are handled as before:

$$
\begin{array}{ll}
\mathcal{T}^n[\mathsf{x_1{:}t_1,\ldots,x_\mathit{n}{:}t_\mathit{n}}] & = \mathsf{x_1}{:}\mathcal{T}^n[\mathsf{t_1}] \times \cdots \times \mathsf{x}_n{:}\mathcal{T}^n[\mathsf{t}_n] \\
\mathcal{E}^n[\Gamma \vdash \mathsf{x\colon t}]\ \rho & = \rho.\mathsf{x}
\end{array}
$$

The new semantic domains will include, in one form or another, sets of all SQUR terms of type s, which we denote as $\mathbb{E}_\mathsf{s}$. The terms are generally open, so, strictly speaking we have to index $\mathbb{E}$ not only by the type but also by the corresponding typing environment. To keep the notation readable (and writable), however, we make the typing environment implicit. For base types (we only show int), the new domain is the disjoint union (sum) of $\mathcal{T}[\mathsf{b}]$ and $\mathbb{E}_\mathsf{b}$. To simplify the notation, we explicitly write only the *inr* tag of the sum, and elide *inl*. We also show the relevant clauses of a new semantic function, $\mathcal{I}[-]\colon \mathcal{T}^n[\mathsf{s}] \to \mathbb{E}_\mathsf{s}$, called *reification*. It is in some sense (made precise later) an 'inverse' of evaluation, producing the SQUR expression with the given (non-standard) meaning.

$$
\begin{array}{ll}
\mathcal{T}^n[\mathsf{int}] & = \mathbb{N} \oplus \mathbb{E}_{\mathsf{int}} \\
\mathcal{E}^n[\Gamma \vdash \mathsf{0}\colon \mathsf{int}]\ \rho & = 0 \quad \text{(other integer constants are similar)} \\
\mathcal{E}^n[\Gamma \vdash \mathsf{e}_1 + \mathsf{e}_2\colon \mathsf{int}]\ \rho & = \mathit{add}\ (\mathcal{E}^n[\Gamma \vdash \mathsf{e}_1\colon \mathsf{int}]\ \rho)\ (\mathcal{E}^n[\Gamma \vdash \mathsf{e}_2\colon \mathsf{int}]\ \rho) \text{ where} \\
\quad \mathit{add}\ 0\ \mathrm{x} & = \mathrm{x} \\
\quad \mathit{add}\ \mathrm{x}\ 0 & = \mathrm{x} \\
\quad \mathit{add}\ n\ m & = n + m \quad n, m \in \mathbb{N} \\
\quad \mathit{add}\ \mathrm{x}\ \mathrm{y} & = \mathit{inr}\ (\mathcal{I}[\mathsf{x}] + \mathcal{I}[\mathsf{y}]) \\
\mathcal{I}[\mathsf{0}] & = \mathsf{0} \\
\mathcal{I}[\mathit{inr}\ \mathsf{e}] & = \mathsf{e}
\end{array}
$$

The standard and non-standard semantics thus differ in assigning meaning to open expressions: the former interprets, say, x+1 to mean a $\mathbb{N} \to \mathbb{N}$ function (viz., the increment). On the other hand, the non-standard semantics interprets the same expression as itself (plus the implicit typing environment associating x with int). The non-standard domain is hence the domain of partially-known values, familiar from partial evaluation, also known as a glued domain [7].

The interpretation of record types is similar:

$$
\begin{array}{ll}
\mathcal{T}^n[\mathsf{<l{:}b,\ldots>}] & = \mathsf{l}{:}\mathcal{T}^n[\mathsf{b}] \times \cdots \oplus \mathbb{E} \\
\mathcal{E}^n[\Gamma \vdash \mathsf{<l{=}e,\ldots>}\colon \mathsf{<l{:}b,\ldots>}]\ \rho & = \mathsf{<l{:}}\mathcal{E}^n[\Gamma \vdash \mathsf{e{:}b}]\rho,\ldots\mathsf{>} \\
\mathcal{E}^n[\Gamma \vdash \mathsf{e.l}_i\colon \mathsf{b}_i]\ \rho & = \mathit{prj}_\mathsf{l}\ (\mathcal{E}^n[\Gamma \vdash \mathsf{e}\colon \mathsf{<l{:}b,\ldots>}]\rho) \text{ where} \\
\quad \mathit{prj}_\mathsf{l}\ \mathsf{<l{:}}x,\ldots\mathsf{>} & = x \\
\quad \mathit{prj}_\mathsf{l}\ (\mathit{inr}\ \mathsf{e}) & = \mathit{inr}\ \mathsf{e.l} \\
\mathcal{I}[\mathsf{<l{:}}x,\ldots\mathsf{>}] & = \mathsf{<l{:}}\mathcal{I}[x],\ldots\mathsf{>} \\
\mathcal{I}[\mathit{inr}\ \mathsf{e}] & = \mathsf{e}
\end{array}
$$

The non-standard semantic domain for bag types is quite more complex. Formally, the meaning of a t bag expression is a multiset whose elements (to be called primitive comprehensions) are triples (ts,y,w) where w is the non-standard boolean representing the guard of the comprehension; y is the meaning of the (generic) comprehension element and ts is the set of pairs (x,m) where m is a table constant of type t' tbl for some t' and x: t' is a fresh variable. We will use a special notation for such triple: fors(x←m. . .) whr w yld y, which should remind one of SQUR's repeated comprehension expressions (with $n \geq 0$)

for(x$_1$← table m$_1$) . . . **for**(x$_n$← table m$_n$) **where** w **yield** y

We write $\mathbb{M}$ for the set of table constants. The reification clause should explain the intent behind our representation of bags. As we shall see later, the representation is also good for converting to SQL.

A primitive table and **yield** are straightforward to interpret as primitive comprehensions. The operation $\uplus$, like in the standard semantics, joins the multisets. The **where** operation pushes its boolean guard down to the whr w of a primitive comprehension. The **for** operation is interpreted as a nested comprehension. The

$$\begin{aligned}
&\mathcal{T}^n[\mathsf{t\ bag}] &&= \{\{\ \texttt{fors}(\mathsf{x}\leftarrow\mathbb{M}\ldots)\ \texttt{whr}\ \mathcal{T}^n[\mathsf{bool}]\ \texttt{yld}\ \mathcal{T}^n[\mathsf{t}]\ \}\}\\
&\mathcal{E}^n[\Gamma \vdash \mathsf{bag\_empty}\colon \mathsf{t\ bag}]\ \rho &&= \{\{\}\}\\
&\mathcal{E}^n[\Gamma \vdash \mathsf{e}_1 \uplus \mathsf{e}_2\colon \mathsf{t\ bag}]\ \rho &&= \mathcal{E}^n[\Gamma \vdash \mathsf{e}_1\colon \mathsf{t\ bag}]\rho \cup \mathcal{E}^n[\Gamma \vdash \mathsf{e}_2\colon \mathsf{t\ bag}]\rho\\
&\mathcal{E}^n[\Gamma \vdash \mathbf{yield}\ \mathsf{e}\colon \mathsf{t\ bag}]\ \rho &&= \{\{\ \texttt{fors}\ ()\ \texttt{whr}\ T\ \texttt{yld}\ \mathcal{E}^n[\Gamma \vdash \mathsf{e}\colon \mathsf{t}]\rho\ \}\}\\
&\mathcal{E}^n[\Gamma \vdash \mathsf{table\ m}\colon \mathsf{t\ bag}]\ \rho &&= \{\{\ \texttt{fors}\ (\mathsf{u}\leftarrow\mathsf{m})\ \texttt{whr}\ T\ \texttt{yld}\ \mathsf{u}\ \}\}\text{ and }\mathsf{u}\text{ is fresh}\\
&\mathcal{E}^n[\Gamma \vdash \mathbf{where}\ \mathsf{e}_1\ \mathsf{e}\colon \mathsf{t\ bag}]\ \rho &&= \mathit{where}'\ (\mathcal{E}^n[\Gamma \vdash \mathsf{e}_1\colon \mathsf{bool}]\rho)\ (\mathcal{E}^n[\Gamma \vdash \mathsf{e}\colon \mathsf{t\ bag}]\rho)\text{ where}\\
&\quad \mathit{where}'\ T\ \mathit{xs} &&= \mathit{xs}\\
&\quad \mathit{where}'\ F\ \mathit{xs} &&= \{\{\}\}\\
&\quad \mathit{where}'\ t\ \mathit{xs} &&= \{\{\ \texttt{fors}\ (\mathsf{x}\leftarrow\mathsf{m}\ldots)\ \texttt{whr}\ w \wedge t\ \texttt{yld}\ y\\
& && \qquad\quad \mid\ \texttt{fors}\ (\mathsf{x}\leftarrow\mathsf{m}\ldots)\ \texttt{whr}\ w\ \texttt{yld}\ y \leftarrow \mathit{xs}\}\}\\
&\mathcal{E}^n[\Gamma \vdash \mathbf{for}(\mathsf{x}\leftarrow\mathsf{e}_1)\ \mathsf{e}\colon \mathsf{t\ bag}]\ \rho =\\
&\qquad \{\{\ \texttt{fors}\ (\mathsf{x'}\leftarrow\mathsf{m'},\ldots\mathsf{x''}\leftarrow\mathsf{m''},\ldots)\ \texttt{whr}\ w' \wedge w''\ \texttt{yld}\ y''\ \mid\\
&\qquad\qquad \texttt{fors}\ (\mathsf{x'}\leftarrow\mathsf{m'}\ldots)\ \texttt{whr}\ w'\ \texttt{yld}\ y' \leftarrow \mathcal{E}^n[\Gamma \vdash \mathsf{e}_1\colon \mathsf{t}_1\ \mathsf{bag}]\rho,\\
&\qquad\qquad \texttt{fors}\ (\mathsf{x''}\leftarrow\mathsf{m''}\ldots)\ \texttt{whr}\ w''\ \texttt{yld}\ y'' \leftarrow \mathcal{E}^n[\Gamma,\mathsf{x}\colon\mathsf{t}_1 \vdash \mathsf{e}\colon \mathsf{t\ bag}](\rho\times\mathsf{x}\colon y')\ \}\}\\
&\mathcal{I}[\{\{\}\}] &&= \mathsf{bag\_empty}\\
&\mathcal{I}[\mathit{xs}] &&= \uplus\ \{\{\ \mathbf{for}(\mathsf{x}\leftarrow\mathsf{table\ m})\ \ldots\ \mathbf{where}\ \mathcal{I}[w]\ \mathbf{yield}\ \mathcal{I}[y]\ \mid\\
& && \qquad\qquad \texttt{fors}(\mathsf{x}\leftarrow\mathsf{m}\ldots)\ \texttt{whr}\ w\ \texttt{yld}\ y \leftarrow \mathit{xs}\}\}\\
& && \qquad (\text{the } \mathbf{where} \text{ clause is omitted if } w \text{ is } T)
\end{aligned}$$

SQUR implementation realizes the non-standard evaluation $\mathcal{E}^n[-]$ as another tagless-final interpreter of SQUR's expressions.

3.1 Formal properties of NBE

The mere inspection of $\mathcal{E}^n[-]$ shows it to be total and hence well-defined, giving another denotational semantics of SQUR. Recall, $\mathcal{I}[-]\colon \mathcal{T}^n[\mathsf{s}] \to \mathbb{E}_\mathsf{s}$ takes the non-standard interpretation of a SQUR expression ('semantics') and picks an expression ('syntax') with that meaning. Such an operation is typically called 'reify'; see the tutorial [7] for more discussion. Clearly, $\mathcal{I}[-]$ picks an expression of the same type as the original one. We have been implicit about the environments however. The following proposition, which follows from a more careful analysis of $\mathcal{E}^n[-]$ and $\mathcal{I}[-]$, recovers the environments.

Proposition 1 (Type Preservation). *For all $\Gamma \vdash \mathsf{e}\colon\mathsf{s}$ and $\rho \in \mathcal{T}^n[\Gamma]$, it holds $\Gamma' \vdash \mathcal{I}[\mathcal{E}^n[\mathsf{e}]\rho]$, where Γ' lists the variables in the domain of ρ and their types.*

As an example, consider

$$\mathsf{x}\colon\langle \mathsf{l}_1\colon\mathsf{int},\mathsf{l}_2\colon\mathsf{int},\mathsf{l}_3\colon\mathsf{int}\rangle,\ \mathsf{y}\colon\mathsf{int} \vdash \mathsf{x.l}_1 + \mathsf{x.l}_2 + \mathsf{x.l}_3 + \mathsf{y}\colon\mathsf{int}$$

Interpreting it in the environment $\rho = \langle\ \mathsf{x}\colon\langle \mathsf{l}_1\colon \mathit{inr}\ \mathsf{u}_1, \mathsf{l}_2\colon \mathit{inr}\ \mathsf{u}_2{+}2, \mathsf{l}_3{=}3\rangle,\ \mathsf{y}\colon 4\ \rangle$ and reifying gives $\mathsf{u}_1\colon\mathsf{int},\mathsf{u}_2\colon\mathsf{int} \vdash \mathsf{u}_1{+}(\mathsf{u}_2{+}2){+}7\colon\mathsf{int}$.

Theorem 3 (Soundness of NBE). *For all* SQUR *expressions $\Gamma \vdash \mathsf{e}\colon\mathsf{s}$, and environments ρ and ρ' of appropriate types, $\mathcal{E}[\ \mathcal{I}[\mathcal{E}^n[\mathsf{e}]\rho]\ ]\rho'$ is equal to $\mathcal{E}[\mathsf{e}](\mathcal{E}[\mathcal{I}[\rho]]\rho')$.*

The non-standard interpretation is thus consistent with the standard denotational semantics in §2.3. For closed e the theorem states that $\mathcal{I}[\mathcal{E}^n[\mathsf{e}]\langle\rangle]$ is equal (i.e., has the equal denotation) to e. Hence $\mathcal{I}[-]$ is the left inverse of $\mathcal{E}^n[-]$. The proof is straightforward and is outlined in the Appendix.

3.2 Normal Forms and SQL Generation

Definition 1 (Normal form). *We call $\mathcal{I}[\mathcal{E}^n[\mathsf{e}]\texttt{<>}]$ the normal form $\mathcal{N}[\mathsf{e}]$ of a closed term* e

Proposition 2 (Correctness of normal form). *If e is a closed term of the type* s*, then: (a)* $\mathcal{N}[\mathsf{e}]$ *exists; (b)* $\vdash \mathcal{N}[\mathsf{e}]{:}\mathsf{s}$*; (c)* $\mathcal{N}[\mathcal{N}[\mathsf{e}]] = \mathcal{N}[\mathsf{e}]$*; (d)* $\mathcal{E}[\mathsf{e}] = \mathcal{E}[\mathcal{N}[\mathsf{e}]]$

That is, the normalization is total, type-preserving, idempotent, and meaning preserving. The totality comes from the fact that $\mathcal{E}^n[-]$ and $\mathcal{I}[-]$ are total; (b) is Prop. 1; (d) is a corollary of Thm. 3, and (c) is easy to verify by inspection of $\mathcal{E}^n[-]$. The fact that $\mathcal{N}[\mathsf{table\ m}]$ is **for**(x←table m) **yield** x tells that our normal forms are eta-long.

We can now complete our program of interpreting every well-typed SQUR expression of a bag type as a SQL query. First we apply the normalization-by-evaluation to obtain the normal form $\mathcal{N}[\mathsf{e}]$, which is designed to be easily mapped to a flat SQL query. Actually, it is simpler use the result of $\mathcal{E}^n[\mathsf{e}]$ directly. Recall, if e is a closed expression of a bag type, $\mathcal{E}^n[\mathsf{e}]\texttt{<>}$ is a multiset of primitive comprehensions {{ `fors` (x←m. . .) `whr` w `yld` y }} where m are table constants. A primitive comprehension is straightforward to convert to a SELECT statement: the table constants m,. . . become the FROM list of the SELECT statement, w becomes the WHERE condition and y becomes the SELECT list. If the multiset of primitive comprehensions has more than one element, the resulting SELECTs are UNION ALL-ed together.

The approach presented so far handles the same language-integrated query language as QueΛ and T-LINQ [4], eventually producing the same SQL code as in those two previous approaches (SQUR in Fig. 1 does not include EXIST queries, but our implementation of SQUR does).

An example is given in §2, when describing the SQL code eventually obtained for the composition of the sample queries, Q_e and Q_d.

4 Ordering

SQL has operations to sort the results of a query or extract a particular range of rows, as discussed in §1.2. The query in the right-hand column is the example from §1.2, sorting the filtered list of employees.

defn Q_{eo}:
for(e ← table employee) **where** e.wage>20
ordering_wage e.wage **yield** e
	SELECT E.∗ FROM employee **as** E
	WHERE E.wage > 20 ORDER BY E.wage

whereas the following one returns only the first three rows of the sorted table starting from the second one.

defn Q_{el}:
for(e ← table employee) **where** e.wage>20
limit (3,1) **ordering**_wage e.wage **yield** e
	SELECT E.∗ FROM employee **as** E
	WHERE E.wage > 20 ORDER BY E.wage
	LIMIT 3 OFFSET 1

To write these queries in SQUR we add the operations ordering and limit, as illustrated in the left-hand column of the tables. Formally the operations are defined as

```
Ordering Effects o:[olabel,...], l:(n,m)
Ordering Labels  owage,...
Expressions      e +:= ordering_wage e1 e | limit (n,m) e | let table x = e in e
```

Fig. 5. Syntax of ordering and ranging operations

We assume a countable supply of ordering operations **ordering**_l, each with its own effect label ol; we show only one such operation **ordering**_wage and its effect label owage; the others are analogous. An implementation of SQUR is presumed to be able to declare ordering operations and the corresponding labels (e.g., using generative modules). Following the earlier conventions, expressions e in **ordering**_l e_1 e and **limit**(n,m) e continue as far right as possible, saving us parentheses. The ordering is ascending; this is not a limitation since the key is an arbitrary integer expression, which can always be adjusted for the desired ordering. The directive **limit** (n,m) e extracts n elements starting from m from the sequence obtained by sorting the *final* bag result; the type system ensures the presence of ordering defining the sorting keys. We also added **let** table, the let-expression specialized for bags, to be used shortly.

In SQL, ORDER BY and LIMIT ... OFFSET are to be applied at the end, to the results of the query – at the top-level, so to speak. When reusing previously written queries as part of new ones, the originally 'top-level' forms quickly become buried in subexpressions. Here is an example of a query composition, reusing Q_{eo}:

```
defn Q_eo2:
 for (e ← Q_eo) for (d ← table department)
 where e.deptID = d.deptID ordering_dept d.deptID
 yield <name=e.name, dep=d.name, wage=e.wage>
```

(assume Q_{eo} in the above expression is substituted with the corresponding query). Naively doing such composition in SQL results in

```
SELECT E.name, D.name, E.wage FROM (SELECT E.∗ FROM employee as E WHERE
       E.wage > 20 ORDER BY E.wage) AS E, department as D
WHERE E.deptID = D.deptID ORDER BY D.deptID
```

with ORDER BY in a subquery – which is either slow and wasteful, or even not allowed at all, as we saw in §1.2.

To allow compositionality we do not impose syntactic restrictions on ordering and limit: they may in principle appear anywhere within a query. We do however wish to preserve the intent of SQL of treating these operations as directives, to be applied to the end-result of a query. Thus, ordering and limiting are *effects*. Even if ordering and limit may be buried and duplicated, they have an effect, which is noticed, accumulated, and applied to the query results. In the type system the effects appear as the annotation on the bag type, as shown in the extended type system in Fig. 6. We have refined the (For) rule, whose justification will become clear from the dynamic semantics explained below. The ordering effect annotation o:[label,...] is parameterized by the list of ordering labels. The limit annotation l:(n,m) is indexed by the subranging parameters. They are statically known integers and do not require dependent types. They can be simply realized using OCaml modules.

$$\frac{\Gamma \vdash \mathsf{e_1{:}\ int} \qquad \Gamma \vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon \qquad \epsilon \subseteq \{\mathsf{o{:}[lb,\ldots]}\}}{\Gamma \vdash \textbf{ordering}\mathsf{\_wage\ e_1\ e\ :\ t\ bag\hat{}\{o{:}[owage,lb,\ldots]\}}}\ \text{Ordering}$$

$$\frac{\Gamma \vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon \qquad \epsilon = \{\mathsf{o{:}[lb,\ldots]}\}}{\Gamma \vdash \textbf{limit}\ \mathsf{(n,m)\ e\ :\ t\ bag\hat{}}(\epsilon \cup \{\mathsf{l{:}(n,m)}\})}\ \text{Limit}$$

$$\frac{\Gamma \vdash \mathsf{e_1{:}\ t_1\ bag\hat{}}\epsilon_1 \qquad \epsilon_1 \subseteq \{\mathsf{o{:}[lb,\ldots]}\} \qquad \Gamma,\mathsf{x{:}t_1} \vdash \mathsf{e_2{:}\ t_2\ bag\hat{}}\epsilon}{\Gamma \vdash \textbf{for}\mathsf{(x{\leftarrow}e_1)\ e_2{:}\ t_2\ bag\hat{}}\epsilon}\ \text{For}$$

$$\frac{\vdash \mathsf{e_1{:}\ t_1\ bag\hat{}}\epsilon_1 \qquad \Gamma,\mathsf{y{:}t_1\ tbl} \vdash \mathsf{e_2{:}\ t_2\ bag\hat{}}\epsilon}{\Gamma \vdash \textbf{let}\ \mathsf{table\ y{=}e_1}\ \textbf{in}\ \mathsf{e_2{:}\ t_2\ bag\hat{}}\epsilon}\ \text{Let}$$

Fig. 6. Type system with ordering operations

The type system reflects several arbitrary choices, as we explain later. For example, in the (Ordering) rule, **ordering**_wage e$_1$ e adds the corresponding ordering label owage before other ordering effect labels associated with e. As extensively discussed back in §1.2, ORDER BY in subexpressions, unless accompanied by LIMIT, makes no sense and we (along with several real database systems) ignore it, which is reflected in the denotational semantics below.

The extended normalization-by-evaluation normalizes Q_{eo2} into

```
defn Q^n_eo2:
 for (e ← table employee)
 for (d ← table department)
 where e.deptID = d.deptID && e.wage>20
 ordering_dept d.deptID
 yield <name=e.name, dep=d.name, wage=e.wage>
```

```
SELECT E.name, D.name, E.wage
FROM employee as E, department as D
WHERE E.deptID = D.deptID AND
  E.wage > 20
ORDER BY D.deptID
```

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

If we attempt to write Q_{eo2} with the subranged Q_{el} in place of Q_{eo}, it will not type check: whereas Q_{eo} has the effect annotation {o:[owage]}, Q_{el} has {l:(3,1), o:[owage]}. The (For) typing rule for **for**(x←e$_1$) e$_2$ does not permit the subranging l:(n,m) effect annotation on e$_1$. That is, ordering with the limit cannot be eliminated from a subquery, resulting in the performance hit. One has to use let-table, to make the performance implications explicit:

```
defn Q_eol2:
 let table t = Q_el in
 for (e ← table t) for (d ← table department)
 where e.deptID = d.deptID ordering_dept d.deptID
 yield <name=e.name, dep=d.name, wage=e.wage>
```

which translates to SQL with common table expressions

```
WITH t8 AS (SELECT E.∗ FROM employee as E
            WHERE E.wage > 20 ORDER BY E.wage LIMIT 3 OFFSET 1)
SELECT t9.name, t7.name, t9.wage FROM department AS t7, t8 AS t9
WHERE t9.deptID = t7.deptID ORDER BY t7.deptID
```

The denotational semantics of SQUR extended with ordering and subranging is subtle. It is rather surprising how little has changed: only the interpretation of **for** is updated, and only slightly.

$$
\begin{array}{ll}
\mathcal{T}[\mathsf{t\ bag\hat{}}\phi] & = \{\{\ \mathcal{T}[\mathsf{t}]\ \}\} \\
\mathcal{T}[\mathsf{t\ bag\hat{}}\{\mathsf{o{:}[lb,\ldots]}\}] & = \{\{\ \mathcal{T}[\mathsf{t}] \times \mathsf{o}{:}(\mathbb{N} \times \ldots)\ \}\} \\
\mathcal{T}[\mathsf{t\ bag\hat{}}\{\mathsf{o{:}[lb,\ldots],l{:}(n,m)}\}] & = \{\{\ \mathcal{T}[\mathsf{t}] \times \mathsf{o}{:}(\mathbb{N} \times \ldots) \times \mathsf{l}{:}(\mathbb{N} \times \mathbb{N})\ \}\} \\
\mathcal{E}[\Gamma \vdash \mathbf{for}(\mathsf{x}{\leftarrow}\mathsf{e}_1)\ \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\ \rho & = \\
\quad \bigcup\{\{\ \mathcal{E}[\Gamma,\mathsf{x{:}t}_1 \vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\ (\rho \times \mathsf{x} : x')\ \mid\ x' \times \mathsf{o}{:}\_ \leftarrow \mathcal{E}[\Gamma \vdash \mathsf{e}_1{:}\ \mathsf{t}_1\ \mathsf{bag\hat{}}\epsilon']\rho\ \}\} & \\
\mathcal{E}[\Gamma \vdash \mathbf{ordering\_}\mathsf{lb}\ \mathsf{e}_1\ \mathsf{e{:}\ t\ bag\hat{}}\{\mathsf{o{:}[lb]}\}]\ \rho & = \\
\quad \{\{\ x \times \mathsf{o}{:}[\mathcal{E}[\mathsf{e}_1]\rho]\ \mid\ x \leftarrow \mathcal{E}[\Gamma \vdash \mathsf{e{:}\ t\ bag\hat{}}\phi]\rho\ \}\} & \\
\mathcal{E}[\Gamma \vdash \mathbf{ordering\_}\mathsf{lb}\ \mathsf{e}_1\ \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\ \rho & = \\
\quad \{\{\ x \times \mathsf{o}{:}[\mathcal{E}[\mathsf{e}_1]\rho,\mathsf{lb'},\ldots]\ \mid\ x \times \mathsf{o{:}[lb',\ldots]} \leftarrow \mathcal{E}[\Gamma \vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon_1]\rho\ \}\} & \\
\quad \text{where } \epsilon_1{=}\{\mathsf{o{:}[lb',\ldots]}\} \text{ and } \epsilon{=}\{\mathsf{o{:}[lb,lb',\ldots]}\} & \\
\mathcal{E}[\Gamma \vdash \mathbf{limit}\ \mathsf{(n,m)}\ \mathsf{e{:}\ t\ bag\hat{}}\{\epsilon\ \mathsf{U}\ \mathsf{l{:}(n,m)}\}]\ \rho = & \\
\quad \{\{\ x \times \mathsf{l{:}(n,m)}\ \mid\ x \leftarrow \mathcal{E}[\Gamma \vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\rho\ \}\} & \\
\mathcal{E}[\Gamma \vdash \mathbf{let}\ \mathsf{table\ y{=}e}_1\ \mathbf{in}\ \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\ \rho & = \\
\quad \mathcal{E}[\Gamma,\mathsf{y{:}t}_1\ \mathsf{tbl} \vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\ (\rho \times \mathsf{y}{:}\mathcal{M}[\vdash \mathsf{e}_1{:}\ \mathsf{t}_1\ \mathsf{bag\hat{}}\epsilon_1]) & \\
\mathcal{M}[\vdash \mathsf{e{:}\ t\ bag\hat{}}\phi] & = \mathcal{E}[\vdash \mathsf{e{:}\ t\ bag\hat{}}\phi]\mathsf{<>} \\
\mathcal{M}[\vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon] & = \\
\quad \textit{subrange}\ \mathsf{(n,m)} \circ \textit{sort}\ \mathsf{keys}\ \{\{\ x\ \mid\ x \times \mathsf{o{:}keys} \times \mathsf{l{:}(n,m)} \leftarrow \mathcal{E}[\vdash \mathsf{e{:}\ t\ bag\hat{}}\epsilon]\mathsf{<>}\ \}\} & \\
\quad \text{(no subranging if the } \mathsf{l} \text{ annotation is absent)} &
\end{array}
$$

Fig. 7. Denotational semantics of SQUR with ordering and limit

As before, an expression of the t bag^ϵ type is interpreted as a multiset, whose elements are (typically records) $\mathcal{T}$[t]. If the effect annotation ϵ includes ordering o:[label,...], we add to $\mathcal{T}$[t] a new field o, a tuple of sorting keys (which SQUR takes, without loss of generality, to be integers). If the l:(n,m) annotation is also present, we add yet another field, l, with the pair of integers n and m. The type system ensures that the field has the same value across all elements of a multiset (and so it could be factored out, as our OCaml implementation actually does). The interpretation of **for** is changed to ignore the extra fields in the comprehended multiset (the l field should be absent to start with, according to the type system). The type system ensures that in $\mathsf{e}_1 \uplus \mathsf{e}_2$, both expressions have the same effect annotations: the same subranging and the same sorting keys. Therefore, taking, as before, the multiset union of $\mathcal{E}[\mathsf{e}_1]\rho$ and $\mathcal{E}[\mathsf{e}_2]\rho$ is meaningful. It is easy to verify that type soundness (Thm. 1) is preserved.

We now have to distinguish the denotation of an expression from the denotation of the whole program. The latter is computed by the semantic function $\mathcal{M}[-]$ that maps a closed term of a bag type to a *sequence* of elements. For an expression e without the ordering annotation, $\mathcal{M}[-]$ converts the multiset $\mathcal{E}$[e]<> to a sequence of some non-deterministic order. If the ordering annotation is present, however, $\mathcal{M}[-]$ uses the o:keys field to sort the elements (removing the field from the result). If the l:(n,m) annotation is also present, the (n,m) subsequence is extracted afterwards. $\mathcal{M}[-]$ hence corresponds to the semantics of ordering and subranging defined in SQL: sorting and limiting are applied at the end of query processing.

Since UNION ALL has the same denotation as in Core SQUR, it is still commutative and associative, and the distributivity laws (Thm. 2) still hold:

Theorem 4 (Distributive Equational Laws of UNION ALL).

$$\begin{array}{lcll}
\mathbf{for}\,(x \leftarrow e_1 \uplus e_2)\ e & \equiv & (\mathbf{for}\,(x{\leftarrow}e_1)\ e) & \uplus\ (\mathbf{for}\,(x{\leftarrow}e_2)\ e) \\
\mathbf{for}\,(x \leftarrow e)\ e_1 \uplus e_2 & \equiv & (\mathbf{for}\,(x{\leftarrow}e)\ e_1) & \uplus\ (\mathbf{for}\,(x{\leftarrow}e)\ e_2) \\
\mathbf{where}\ e\ e_1 \uplus e_2 & \equiv & (\mathbf{where}\ e\ e_1) & \uplus\ (\mathbf{where}\ e\ e_2) \\
\mathbf{ordering\_}lb\ \ e\ (e_1 \uplus e_2) & \equiv & (\mathbf{ordering\_}lb\ \ e\ e_1) & \uplus\ (\mathbf{ordering\_}lb\ \ e\ e_2) \\
\mathbf{limit}\ \ (n,m)\ (e_1 \uplus e_2) & \equiv & (\mathbf{limit}\ \ (n,m)\ e_1) & \uplus\ (\mathbf{limit}\ \ (n,m)\ e_2)
\end{array}$$

That is quite unexpected, if one were to take ordering and limit naively, as sorting and subranging of their argument expressions. One surely would not think of UNION ALL to be distributive, let alone symmetric. The distributivity across ordering and limit is particularly outrageous. It is worth stressing again that ordering and limit are directives. The expression ordering e_1 e does *not* immediately sort the bag e; neither does **limit** (1,0) e mean taking the first element of the bag e (which is meaningless since bags, as multisets, have no definite order).

In defining the semantics of ordering and limit, we had to resolve a number of essentially arbitrary choices:

- in a nested ordering e_1 ordering e_2 e, should e_1 be the major sort key, or e_2? We chose e_1.
- in a nested subranging **limit** (n_1,m_1) **limit** (n_2,m_2) e, should the outer limit take effect over the inner, or the inner overriding outer, or somehow be combined? Or the nesting of limit should be outlawed by the time system? We chose the latter. Multiple limits are still possible; on has to explicitly use **let** table to force subranging and other effects of an intermediary expression.
- should ordering e_1 **limit** (n,m) e be allowed (without the the explicit **let** table)? We chose against it, since e_1 as the major sort key affects the subranging.

One may argue for different choices: after all, SQL gives no guidance since the standard only talks about ORDER BY when attached to the top-most SELECT. The choices are up to us to make. However we – or the reader – choose, our denotational framework trivially accommodates any choice.

The presented denotational semantics can then be generalized to the normalization-by-evaluation semantics, similar to the approach illustrated in §3. The only significant change is to add to the triple `fors` (x←m. . .) `whr` w `yld` y two extra fields, for the ordering keys and subranging, depending on the effect annotations. For the lack of space, we refer to the SQUR implementation for details. The non-standard interpretation can again be converted to SQL. The example SQL code at the beginning of this section was the output of such conversion.

5 Related Work

Language-integrated query as a research area was established with the nested relational calculus (NRC) in [14] and [2]. Specifically, integration of relational algebra into a typed functional language was pioneered in [3]. Although some versions of NRC [10, Corollary 3.3] can express ordering (called 'rank assignment' in that paper), it is what §4 has called the 'naive' ordering, whose serious drawbacks have been explained in §1.2.

Our approach may superficially be seen as an extension of a long line of work starting from Cooper [6] and Microsoft LINQ, and continuing through T-LINQ [4] and [13]. None of them considered ordering and subranging. Our use of effects is notably different from Cooper's and our integration with the host language is less tight and more stylized. Compared to T-LINQ, SQUR has no quotation. As we have already emphasized, SQUR has no functions and is not an extension of lambda-calculus. Although SQUR is superficially similar to QueΛ of Suzuki et al., it has no first-class functions, its dynamic semantics is given denotationally rather than operationally, the soundness of its type system is proven semantically rather than syntactically. Finally, SQUR relies on the normalization-by-evaluation (NBE) rather than on repeated, and hopefully convergent, re-writing.

The language-integrated query systems SML# [12] and Haskell's Opaleye [8] and HRR [9] also deal with ordering. They do not present relational data as local arrays to iterate over. They have rather complicated type system. The published materials describe no compositional semantics. Mainly, these systems consider no query normalization and optimizations, relying on subqueries to achieve compositionality – hence exhibit the problems described in §1.2 and may generate broken queries.

We owe a great debt to Dybjer and Filinski's tutorial on normalization-by-evaluation [7]. The many similarities in our NBE approaches are not accidental. One notable difference is our use of the tagless-final approach, which let us index semantic domains by the object types without resorting to dependent types. Therefore, all our interpreters are patently total and hence easier to see correct. Also, we do not define any reduction-based semantics or the corresponding reduction equational theory.

Normalization in embedded languages is thoroughly discussed by Najd et al. in [11]. However, they consider quite a more complicated problem of so-called quoted DSLs, where quoted expressions may contain higher-order constructs such as function applications. The latter are to be eliminated in the course of normalization, and the subformula property sees to it. The normalization procedure of their QDSL operates on 'syntax', a representation of a QDSL expression. We, on the other hand, deal with 'semantics', with a representation of the meaning of a SQUR expression. Since our SQUR does not have functions, there are no β-redices or other higher-order constructs to eliminate.

6 Conclusions

We have presented the new, denotational approach to language-integrated query based on the calculus SQUR. We support query composition and reuse, and still are able to generate efficient flat SQL for interaction with external databases. The key feature is the normalization-by-evaluation (NBE), facilitated by the denotational semantics, which converts a query to the normal form from which flat SQL can be easily generated. Unlike the previous syntactic, rewriting-rule–based approaches, NBE is deterministic and can easily be proven total. Notably, it can be extended to support such SQL features as ordering and subranging of the eventual query results. The denotational approach goes hand-in-hand with the embedding SQUR in the tagless-final style.

In the future work we extend the approach to GROUP BY and aggregation. It is also interesting to further explicate the equational laws induced by the semantics of ordering, grouping and subranging, with [5] for inspiration.

Acknowledgments We thank anonymous reviewers for many very helpful comments and suggestions.

References

[1] Atkinson, M.P., Buneman, O.P.: Types and persistence in database programming languages. ACM Comput. Surv. 19(2), 105–170 (Jun 1987)

[2] Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theor. Comput. Sci. 149(1), 3–48 (Sep 1995)

[3] Buneman, P., Ohori, A.: Polymorphism and type inference in database programming. ACM Transactions on Database Systems 21(1), 30–76 (Mar 1996)

[4] Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. In: ICFP '13. pp. 403–416. ACM, New York, NY, USA (2013)

[5] Chu, S., Weitz, K., Cheung, A., Suciu, D.: HoTTSQL: Proving query rewrites with univalent SQL semantics. CoRR abs/1607.04822 (2016), `http://arxiv.org/abs/1607.04822`

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

[7] Dybjer, P., Filinski, A.: Normalization and partial evaluation. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000: International Summer School on Applied Semantics, Advanced Lectures. pp. 137–192. No. 2395 in Lecture Notes in Computer Science, Springer (2002)

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

[9] Hibino, K., Murayama, S., Yasutake, S., Kuroda, S., Yamamoto, K.: Haskell relational record. `http://khibino.github.io/haskell-relational-record/`, last visited: May 2017

[10] Libkin, L., Wong, L.: Conservativity of nested relational calculi with internal generic functions. Information Processing Letters 49(6), 273–280 (22 Mar 1994)

[11] Najd, S., Lindley, S., Svenningsson, J., Wadler, P.: Everything old is new again: quoted domain-specific languages. In: PEPM. pp. 25–36. ACM (2016)

[12] Ohori, A., Ueno, K.: Making Standard ML a practical database programming language. In: ICFP '11. pp. 307–319. ACM, New York, NY, USA (2011)

[13] Suzuki, K., Kiselyov, O., Kameyama, Y.: Finally, safely-extensible and efficient language-integrated query. In: Proc. PEPM. pp. 37–48. ACM (2016)

[14] Tannen, V., Buneman, P., Wong, L.: Naturally embedded query languages. In: ICDT '92. pp. 140–154. Springer-Verlag, London, UK, UK (1992)

A Proof of Soundness of NBE

This section outlines the proof of soundness of the non-standard evaluation, Thm. 3.

The proof is particularly straightforward for flat (base and record types), considering that record components may have only base types and a flat-type expression may have only flat-type subexpressions.

Indeed, the environment ρ may have only the following cases of bindings: (1) integers (or booleans, strings, etc); (2) a fully opaque expression (code), possibly containing variables; (3) a record whose components are either (1) or (2).

If e is a variable x, consider each of the cases for the corresponding binding in ρ.

1. $\mathcal{I}[\mathcal{E}^n[\mathsf{x}]\langle \mathsf{x}{:}\mathsf{n},\ldots\rangle]$ is n. Then $\mathcal{E}[\mathcal{I}[\langle \mathsf{x}{:}\mathsf{n},\ldots\rangle]]\rho'$ is $\langle \mathsf{x}{:}\mathsf{n},\ldots\rangle$. The theorem holds for any ρ'.
2. $\mathcal{I}[\mathcal{E}^n[\mathsf{x}]\langle \mathsf{x}{:}\mathit{inr}\ \mathsf{e'},\ldots\rangle]$ is e', $\mathcal{I}[\langle \mathsf{x}{:}\mathit{inr}\ \mathsf{e'},\ldots\rangle]$ is $\langle \mathsf{x}{:}\mathsf{e'},\ldots\rangle$, so the theorem statement becomes $\mathcal{E}[\mathsf{e'}]\rho' = \mathcal{E}[\mathsf{x}](\langle \mathsf{x}{:}\mathcal{E}[\mathsf{e'}]\rho',\ldots\rangle)$, which is obviously true.
3. The case of the binding of x to a record reduces to the previous two.

If e is an addition expression $\mathsf{e}_1 + \mathsf{e}_2$, record construction or projection, we use the induction hypothesis.

For bag types, the proof is also a simple induction on the structure of the expression (keeping in mind that the environment ρ has no bindings to expressions of bag type). Consider a few cases:

For **yield** e, $\mathcal{I}[\mathcal{E}^n[\textbf{yield}\ \mathsf{e}]\rho]$ is **yield** $\mathcal{I}[\mathcal{E}^n[\mathsf{e}]\rho]$ and the theorem holds given the proven flat-type case.

For primitive tables, $\mathcal{I}[\mathcal{E}^n[\mathsf{table\ m}]\rho]$ is **for**(x←table m) **yield** x, which is the eta-expanded table m. Hence our normal forms are eta-long.

For UNION ALL expressions, $\mathcal{I}[\mathcal{E}^n[\mathsf{e}_1 \uplus \mathsf{e}_2]\rho]$, the statement of the theorem follows from the inductive hypothesis and the symmetry and associativity of multiset union. The only slightly non-trivial case is **for**-expressions – which is also easy to see considering again the the symmetry and associativity of multiset union, manifesting in the multiset comprehension 'fusion law':

$$\bigcup \{\{\ \mathsf{f\ y} \mid \mathsf{y} \leftarrow \bigcup \{\{\ \mathsf{g\ x} \mid \mathsf{x} \leftarrow \mathsf{xs}\ \}\}\ \}\} = \bigcup \{\{\ \mathsf{f\ y} \mid \mathsf{x} \leftarrow \mathsf{xs},\ \mathsf{y} \leftarrow \mathsf{g\ x}\ \}\}$$