The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed

Ezra Cooper

University of Edinburgh

Abstract. We show how to translate expressions in a higher-order programming language into SQL queries. Somewhat surprisingly, we show that any suitable expression translates to a single SQL query, where the suitability is determined by a type-and-effect check. Thus, unlike in Hollywood where a script-writer can never be sure a movie sequel will be popular, we show how to be sure that your SQL—written in your own language—will succeed (in being translated).

Introduction

Language-integrated query is an approach to accessing relational databases from a general-purpose programming language, which reduces the infamous "impedance mismatch" [7], by allowing queries to be written with the full flexibility of the general language, and translating them to flat relational queries (SQL, for our purposes) when possible. The increased flexibility may include features like functional abstraction, permitting the refactoring of query fragments, or the ability to form nested intermediate data structures having no relational equivalent—all within the familiarity of the host language. Language-integrated query is exemplified by the systems Kleisli [23], LINQ [13], Links [6], and Ferry [10].

These existing systems vary in their support of *abstraction* and degree of *totality*. They do not always permit queries to be abstracted by refactoring query fragments into parameterized functions, nor permit using predefined functions in queries, even if those could be translated in their applied context—at least not with the full flexibility of λ-calculus. As to totality, they generally make a best effort to translate host-language expressions to queries, but may fail to do so at runtime—either with a run-time error or by executing a query outside the database; we call this *partiality*.

The increased flexibility of general programming languages makes it difficult to recognize query-translatable expressions. Besides the problem of nested data structures, programming languages normally have operations with no equivalent in the query language, including recursion, side-effecting statements, and primitive functions that simply aren't available. Existing versions of Kleisli, Links and LINQ resolve the tension through partiality: they may not always transform the expression completely to an SQL query. Instead they might give a run-time error (in the case of LINQ) or they might evaluate the query expression directly by the

host language in a naive and inefficient way (forgoing the benefit of indexes, for example). The consequence may be that the program behaves very inefficiently, with no indication until runtime that this happens. The Ferry system gives a total translation, but without supporting abstraction.

This paper contributes to the science of language-integrated query by extending query translation to higher-order functions (permitting *abstraction*) and offering a static test for translatability (ensuring *totality* at runtime, after passing the static check). Thus you can "write SQL" in your own native programming language, and be sure at compile time that it will succeed in translation.

Contributions The technical contributions of this paper include

1. a translation from any suitable expression of a typical impure, functional programming language to an equivalent SQL query,
2. a type-and-effect system for statically checking this suitability
3. a generalization of existing results for unnesting relational algebra by Wong [22] to a higher-order calculus, and
4. a description of a proof of totality for the translation, securing a result by Fegaras [8].

How it works The recipe for translating higher-order language-integrated queries can be summarized as follows:

1. At compile time,
   (a) Check that query expressions have a flat relation type.
   (b) Use a *type-and-effect system* to check that query expressions are pure.
   (c) Generate two representations of each query-translatable expression: one suitable for execution and one suitable for query generation.
2. At runtime, to execute an expression via SQL,
   (a) Insert the values (query representations) for any free variables and
   (b) Reduce it to eliminate intermediate structures (that is, functions and nested data structures). This produces a normal form directly translatable to SQL.

Example

Suppose Alice runs a local baseball league. First, she wants a list of the players with age at least 12. She might write this function (these examples use the syntax of Links, which is a general-purpose language but is specially adapted to look like a query language):

```
fun overAgePlayers() {
  query { for (p <- players)
          where (p.age > 12)
            [(name = p.name)] }
}
```

(The `for (x <- xs) e` construct is a *bag comprehension*, and works as follows: for each element in the bag denoted by `xs`, it evaluates `e` with `x` bound to that element, producing a new bag; the result is the union of all those bags resulting from the body. Comprehensions have a long history of use in programming languages from Haskell to Python and JavaScript. The construct `where (`$e1$`) `$e2$ is simply shorthand for `if `$e1$` then `$e2$` else []`.) The compiler can deduce that this expression is in fact equivalent to an SQL query, so it accepts the function. However, the following code would give a compiler error:

```
fun overAgePlayersReversed() {
 query { for (p <- players)
         where (p.age > 12)
           [(name = reverse(p.name))] }      # ERROR!
}
```

This is because the `reverse` function has no SQL equivalent, and so no query is equivalent to this expression.

Now, it takes nine players to make a baseball team, but some "teams" in Alice's league are short of players. She needs to generate a mailing list of players that belong to a full team. One way to do this is to collect, for each team, a team roster (list of players) and then filter for those with a roster of at least nine elements. She writes the following code:

```
fun teamRosters() {
  for (t <- teams)
    [(name = t.name,
      roster = for (p <- players)
               where (p.team == t.name)
                 [(playerName=p.name)])];
}
fun usablePlayers() {
  query {
    for (t <- teamRosters())
    where (length(t.roster) >= 9)
      t.roster
  }
}
```

Note the lack of brackets `[]` around the final `t.roster`: since the `for` comprehension takes the *union* of bags produced by the body, we here take the union of satisfying rosters. This expression is equivalent to an SQL query, although not in a direct way, since it uses an intermediate data structure that is nested (`teamRosters` has type `[(name:String, roster:[(playerName:String)])]`), and this is not supported by SQL. But since the final result is flat, our analysis accepts the query-bracketed expression and translates it into an equivalent SQL query, such as this one:

```
select p.name as playerName
  from players as p, teams as t
 where (select count(*) from players as p2
         where p2.team = t.name) < 9)
```

Also note that factoring out the part of the query which returns the bag of all rosters posed no problem: the query translator will simply inline the function and produce a single query.

Suppose now that Alice wishes to abstract the query condition on teams. That is, she wishes to write a function which accepts as argument a roster predicate, and produces a list of the player records belonging to those teams satisfying the predicate. With the following code, the query translator will still produce, in each invocation, a single SQL query:

```
fun playersBySelectedTeams(pred) {
  query {
    for (t <- teamRosters())
    where (pred(t.roster))
      t.roster
  }
}
```

The query translator will ensure that any argument passed as *pred* is itself a translatable function. If any call site tries to pass a untranslatable predicate, it produces a compiler error.

This type of abstraction is particularly difficult to achieve in SQL, since the team rosters themselves cannot be explicitly constructed in SQL as part of a query. SQL restricts how subqueries can be used (for example, in various contexts they must return just one column, just one row, or both) and the subquery itself must be changed if we apply an aggregate function to its single column.

For example, suppose we want to form a sub-league of "senior" teams: teams with enough players of age 15. We might define the following predicates:

```
fun fullTeam(list) { length(list) >= 9 }
fun seniorPlayers(list) { for (x <- list) where (x.age >= 15) [x] }
```

and use them as follows:

```
playersBySelectedTeams(fun(x) { fullTeam(seniorPlayers(x)) } )
```

Here we have freely used *length* and a filtering comprehension, despite the fact that in SQL these are represented very differently:

```
select p.name as playerName
  from players as p, teams as t
 where (select count(*) from players as p2
        where p2.team = t.name and p2.age >= 15) >= 9)
```

The application of `count` needs to be placed within the subquery, while the comparison `>= 9` is placed outside of it. Also the condition on `p2.age` must be placed within the `where` clause of the subquery. It would not be easy to write a "template" SQL query which would produce through string substitution all the queries that *playersBySelectedTeams* does.

Microsoft's LINQ system allows abstraction of query predicates, provided they are defined at a special type (the type of *expressions*). Because of the special type, these predicates cannot directly be reused as ordinary functions, and composition is not supported, so the last example would require explicitly

$$
\begin{array}{rrll}
\text{(terms)} & B, L, M, N & ::= & \mathsf{for}\,(x \leftarrow L)\,M \\
& & \mid & \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N \\
& & \mid & \mathsf{table}\,s : T \\
& & \mid & [M] \mid [] \mid M \uplus N \\
& & \mid & (\overrightarrow{l = M}) \mid M.l \\
& & \mid & LM \mid \lambda x.N \mid x \mid c \\
& & \mid & \oplus(\overrightarrow{M}) \\
& & \mid & \mathsf{empty}(M) \\
& & \mid & \mathsf{query}\,M \\
\text{(primitives)} & \oplus & & \\
\text{(table names)} & s, t & & \\
\text{(field names)} & l & & \\
\\
\text{(types)} & T & ::= & o \mid (\overrightarrow{l : T}) \mid [T] \mid S \xrightarrow{e} T \\
\text{(base types)} & o & ::= & \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \\
\text{(atomic effects)} & E & ::= & \mathsf{noqy} \mid \cdots \\
\text{(effect sets)} & e & & \text{a set of atomic effects}
\end{array}
$$

Fig. 1. Grammar of the source language.

rewriting the composed function `fun(x) { fullTeam(seniorPlayers(x)) }` as a new function. This paper shows how such a system might support composition.

Road map The next sections (1) define the core source language and its static analysis, (2) translate it into SQL, (3) characterize the correctness of the algorithm, and (4) extends it in two ways: with recursion and with the length operator.

1 The Source Language

We define a language which resembles the core of an ordinary impure functional programming language, and is also a conservative extension of the (higher-order) Nested Relational Calculus with side-effects and a query annotation. Its grammar is given in Figure 1.

The terms $[M]$, $[]$ and $M \uplus N$ represent bag (multiset) operations: singleton construction, the empty bag, and bag union. The bag comprehension, written $\mathsf{for}\,(x \leftarrow L)\,M$, computes the union of the bags produced by evaluating M in successive environments formed by binding x to the elements of L in turn. A table handle $\mathsf{table}\,s : T$ denotes a reference to a table, named s, in some active database connection; T designates the effective type of the table.

The conditional form $\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N$ evaluates to the value of either M or N depending on the value of B.

Records are constructed as a parenthesized sequence of field-name–term pairs $(\overrightarrow{l = M})$ and destructed with the field projection $M.l$. When speaking of a record

This work	NRC
$LM \quad \lambda x.N \quad x$	$LM \quad \lambda x.M \quad x$
$\mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N$	$\mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N$
$\mathsf{empty}\ M$	$\mathsf{empty}\ M$
c	c
$\mathsf{for}\ (x \leftarrow L)\ M$	$\bigcup\{M \mid x \in L\}$
$[] \quad [M] \quad M \uplus N$	$\{\} \quad \{M\} \quad M \cup N$
$\mathsf{table}\ s : T$	x
$(\overrightarrow{l = M}) \quad M.l$	$(L, M) \quad () \quad \pi_1 \quad \pi_2$
$\oplus(\overrightarrow{M})$	$M = N$
$\mathsf{query}\ M$	

Fig. 2. Nested Relational Calculus (NRC).

construction $(\overrightarrow{l = M})$ we will indicate the immediate subterms by subscripting the metavariable M with labels so that M_l is a field of $(\overrightarrow{l = M})$ for each $l \in \overrightarrow{l}$. Similarly for record types $(\overrightarrow{l : T})$ the field types will be indicated T_l when $l \in \overrightarrow{l}$.

Functional abstraction $\lambda x.N$ and application LM are as usual. Variables are ranged by x, y, z and other italic alphabetic identifiers, but c ranges over constants.

The language may contain primitive operations, ranged by $\oplus$, which must appear fully-applied (this is not a significant restriction since one may abstract over such expressions). The primitives should include boolean negation ($\neg$).

The form $\mathsf{empty}(M)$ evaluates to a boolean indicating whether the bag denoted by M is the empty bag or not.

The form $\mathsf{query} M$ evaluates to the same value as M but instructs the compiler that M must evaluate as an SQL query. An expression is *translatable* if it can be so evaluated.

Terms are assigned types which can be: *base* types ranged by o, *record* types $(\overrightarrow{l : T})$ where each field label l is given a type T_l, *bag* types $[T]$ and *function* types $S \xrightarrow{e} T$, where S is the function domain, T is the range, and e is a set of effects that the function needs permission to perform.

We consider effects abstractly; E ranges over some arbitrary set of effects, which includes at least an effect noqy and may include other runtime actions such as I/O or reference-cell mutations. Every effect should represent some kind of runtime behavior that has no SQL equivalent; we use the distinguished effect noqy to mark nondatabasable operations when no other effect presents itself.

For simplicity, this first calculus does not include the length operator; Section 4.2 extends to include it.

NRC Comparison Compare this language with NRC as given by Wong [22] (Figure 2). The two languages are nearly the same. Some apparent differences are only notational. NRC's comprehension form $\bigcup\{M \mid x \in L\}$ is identical to

ours, $\mathsf{for}\,(x \leftarrow L)\,M$. The NRC literature uses set notation, $\{\}$, $\{M\}$, and $M \cup N$, but they can refer to any of the extended monads for bags, sets or lists. (Here we treat only bags.) We write table handles explicitly as $\mathsf{table}\,s : T$, with a name s and required type annotation T, which facilitates local translation, while NRC uses free variables to refer to tables.

A few differences are not notational. NRC is defined with tuples while we use records, without loss of generality. We admit an arbitrary set of basic operations $\oplus$ while NRC, at its core, includes no operations; these are treated as extensions in the NRC literature. This formulation of NRC includes an equality test at each type; the equality at base types is treated as a basic operation (ranged by $\oplus$) in our calculus. Finally, our language extends NRC with the translatability assertion $\mathsf{query}\,M$.

(In fact Wong [22] gives the same grammar as reproduced here, but the paper goes on to treat λ-abstractions as though they can only appear in application position, as in $(\lambda x.N)M$, effectively restricting the language to a first-order form. Here we treat it in its full higher-order glory.)

Type-and-effect system The static semantics is defined by a type-and-effect system in Figure 3. It is close to standard systems along the lines of Talpin and Jouvelot [17]. The typing permits no recursion, and thus is analogous to simply-typed λ-calculus. We add recursion later using a fixpoint operator.

The system involves just one form of judgment, $\Gamma \vdash M : T \,!\, e$ which can be read, "In environment Γ, term M has type T and may take effects in the set e."

The type of each constant c is given as T_c, which should be a base type. Constant values at complex type can, of course, be constructed explicitly.

We demand that each primitive $\oplus$ either has an SQL equivalent, $\oplus_{\mathrm{sql}}$, or carries a nonempty effect annotation. We also insist that primitives have basic argument types and basic result type, or else have an effect annotation. The limitation to base-type arguments means that functions like empty and length, which anyway require special rewrite rules, cannot be defined as primitives.

For example, the primitives might include addition, $(+) : \mathsf{int} \times \mathsf{int} \xrightarrow{\varnothing} \mathsf{int}$, which has an SQL equivalent, as well as $\mathsf{print} : \mathsf{string} \xrightarrow{noqy} ()$, which prints to the terminal and has no SQL equivalent, and hence it carries an effect.

In keeping with the flatness of SQL tables, we require that each table must have flat relation type: in $\mathsf{table}\,s : T$ we require $T = [(\overrightarrow{l : o})]$ for some base-membered record type $(\overrightarrow{l : o})$.

An immediate type annotation is required on table expressions. This may seem a nuisance; as a practical matter, however, it provides a direct way for the programmer to check whether the usage type of a table agrees with the underlying table's schema type in the DBMS.

The type system is monomorphic, so for example each appearance of the empty bag $[]$ must be given some particular concrete type.

The main proposition we show is that if a term $\mathsf{query}\,M$ has a typing derivation under these rules, then when any closing, well-typed substitution σ is applied, it gives a term which can be translated to an equivalent SQL query.

$$\Gamma \vdash c : T_c \,!\, \varnothing \quad \text{(T-Const)}$$

$$\Gamma, x : T \vdash x : T \,!\, \varnothing \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : S \vdash N : T \,!\, e'}{\Gamma \vdash \lambda x.N : S \xrightarrow{e'} T \,!\, \varnothing} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash L : S \xrightarrow{e} T \,!\, e' \qquad \Gamma \vdash M : S \,!\, e''}{\Gamma \vdash LM : T \,!\, e \cup e' \cup e''} \quad \text{(T-App)}$$

$$\frac{\begin{array}{c}\oplus : S_1 \times \cdots \times S_n \xrightarrow{e} T \\ \Gamma \vdash M_i : S_i \,!\, e_i \text{ for each } 1 \le i \le n\end{array}}{\Gamma \vdash \oplus(\overrightarrow{M}) : T \,!\, e \cup \bigcup_i e_i} \quad \text{(T-Op)}$$

$$\frac{\Gamma \vdash M : [T] \,!\, e}{\Gamma \vdash \mathsf{empty}(M) : \mathsf{bool} \,!\, e} \quad \text{(T-Empty)}$$

$$\frac{\Gamma \vdash M : T \,!\, \varnothing \qquad T \text{ has the form } [(\overrightarrow{l : o})]}{\Gamma \vdash \mathsf{query}\ M : T \,!\, \varnothing} \quad \text{(T-Db)}$$

$$\frac{T \text{ has the form } [(\overrightarrow{l : o})]}{\Gamma \vdash (\mathsf{table}\, t : T) : T \,!\, \varnothing} \quad \text{(T-Table)}$$

$$\frac{\Gamma \vdash M : [S] \,!\, e \qquad \Gamma, x : S \vdash N : [T] \,!\, e'}{\Gamma \vdash \mathsf{for}\,(x \leftarrow M)\ N : [T] \,!\, e \cup e'} \quad \text{(T-For)}$$

$$\frac{\Gamma \vdash M_i : T_i \,!\, e_i \text{ for each } M_i : T_i \text{ in } \overrightarrow{M : T}}{\Gamma \vdash (\overrightarrow{l = M}) : (\overrightarrow{l : T}) \,!\, \bigcup_i e_i} \quad \text{(T-Record)}$$

$$\frac{\Gamma \vdash M : (\overrightarrow{l : T}) \,!\, e \qquad (l : T) \in (\overrightarrow{l : T})}{\Gamma \vdash M.l : T \,!\, e} \quad \text{(T-Project)}$$

$$\Gamma \vdash [] : [T] \,!\, \varnothing \quad \text{(T-Null)}$$

$$\frac{\Gamma \vdash M : T \,!\, e}{\Gamma \vdash [M] : [T] \,!\, e} \quad \text{(T-Singleton)}$$

$$\frac{\Gamma \vdash M : [T] \,!\, e \qquad \Gamma \vdash N : [T] \,!\, e'}{\Gamma \vdash M \uplus N : [T] \,!\, e \cup e'} \quad \text{(T-Union)}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : \mathsf{bool} \,!\, e \\ \Gamma \vdash M_1 : T \,!\, e' \qquad \Gamma \vdash M_2 : T \,!\, e''\end{array}}{\Gamma \vdash \mathsf{if}\ L\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : T \,!\, e \cup e' \cup e''} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash M : T \,!\, e \qquad e \subseteq e'}{\Gamma \vdash M : T \,!\, e'} \quad \text{(T-Subsump)}$$

Fig. 3. Type-and-effect system.

2 Making Queries

To make queries from the source language, we will rewrite source terms to a sublanguage which translates directly and syntactically into SQL.

We first examine the sublanguage and its relationship to our SQL fragment, then turn to the rewrite system.

SQL-like sublanguage The target sublanguage is given in Figure 4. Observe that the "normal form" expressions ranged by V all have *relation type*: bag of record of base type, or $[(\overrightarrow{l : o})]$. Types of the form $(\overrightarrow{l : o})$ are called "row types," after the database rows that they represent.

SQL The target SQL fragment is given in Figure 5. This includes all unions of queries on an inner join of zero or more tables, with result and query conditions taken from some given algebra of operations, including field projection, boolean conjunction, negation, the exists operator, and conditionals case ... end.

SQL translation The type-sensitive function $[\![-]\!]$ (Figure 6) translates each closed term in the sublanguage directly into a query. (A fine point: SQL has no way of selecting an empty set of result columns in a select clause; to translate $[()]$ we need to offer some dummy value, or *, as the result column. An implementation can supply any such dummy value, remembering to ignore the columns that the database actually returns.)

We assume, without loss of generality, that all bound variables in the source program are distinct, thus there can be no clashes among the tables' as aliases used in the from clause of the query.

Rewrite rules The translation of source terms into the SQL-isomorphic sublanguage is given as a rewrite system (Figure 7). We write $M[L/x]$ for the substitution of the term L for the free variable x in the term M.

A few rules beg explanation. The syntax of SQL permits conditional choices only at the level of individual fields, never at row or table level. Thus IF-SPLIT transforms a choice between two bags into the union of two oppositely-guarded bags. Similarly, IF-RECORD pushes conditional choices at the row level down to the level of fields. As such, both of these rules (and only these) are type-sensitive. The EMPTY-FLATTEN rule ensures that the argument to empty can be normalized to one of the query normal forms of Figure 4, which requires that it have relation type. Rules like APP-IF are common in higher-order rewrite systems: APP-IF pushes the deconstructor $(-)M$ past an interposing form, in order to bring it together with corresponding constructors (here $\lambda x.N$) thus exposing β-reductions.

Several rules may seem unnecessary if we were to use SQL subqueries. For example, why employ the FOR-ASSOC rule if we can write an SQL query that uses a nested select statement in its from clause? After all, we could more directly translate the expression

$$\mathsf{for}\ (y \leftarrow \mathsf{for}\ (x \leftarrow \mathsf{table}\, s : T)\ [(b = x.a)])\ [(c = y.b)]$$

into SQL as follows:

$$\mathsf{select}\ y.b\ \mathsf{as}\ c\ \mathsf{from}\ (\mathsf{select}\ x.a\ \mathsf{as}\ b\ \mathsf{from}\ s\ \mathsf{as}\ x)\ \mathsf{as}\ y.$$

The nested comprehension became a nested subquery. Why then FOR-ASSOC?

The answer is that such rules are critical to the unnesting. Watch how we rewrite this query, which internally constructs a nested bag-of-bag type, not an SQL-representable type:

$$\mathsf{for}\ (y \leftarrow (\mathsf{for}\ (x \leftarrow \mathsf{table}\, s)\ [[x]])\ y \rightsquigarrow \qquad \text{(FOR-ASSOC)}$$

$$\mathsf{for}\ (x \leftarrow \mathsf{table}\, s)\ (\mathsf{for}\ (y \leftarrow [[x]])\ y) \rightsquigarrow \qquad (\beta\text{-FOR})$$

$$\mathsf{for}\ (x \leftarrow \mathsf{table}\, s)\ [x]$$

$$
\begin{array}{lll}
\text{(query normal forms)} & V, U & ::= V \uplus U \mid [] \mid F \\
\text{(comprehension NFs)} & F & ::= \mathsf{for}\ (x \leftarrow \mathsf{table}\ s : T)\ F \mid Z \\
\text{(comprehension bodies)} & Z & ::= \mathsf{if}\ B\ \mathsf{then}\ Z\ \mathsf{else}\ [] \mid [R] \mid \mathsf{table}\ s : T \\
\text{(row forms)} & R & ::= (\overrightarrow{l = B}) \mid x \\
\text{(basic expressions)} & B & ::= \mathsf{if}\ B\ \mathsf{then}\ B'\ \mathsf{else}\ B'' \mid \mathsf{empty}(V) \mid \\
 & & \quad \oplus(\overrightarrow{B}) \mid x.l \mid c
\end{array}
$$

Fig. 4. SQL-like sublanguage.

$$
\begin{array}{rcl}
Q, R & ::= & Q\ \mathsf{union\ all}\ R \mid S \\
S & ::= & \mathsf{select}\ \overrightarrow{s}\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e \\
s & ::= & e\ \mathsf{as}\ l \mid x.* \\
e & ::= & \mathsf{case\ when}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''\ \mathsf{end} \mid \\
 & & \quad c \mid x.l \mid e \wedge e' \mid \neg e \mid \mathsf{exists}(Q) \mid \oplus(\overrightarrow{e})
\end{array}
$$

Fig. 5. Target SQL fragment.

$$
\begin{array}{rcl}
[\![V \uplus U]\!] & = & [\![V]\!]\ \mathsf{union\ all}\ [\![U]\!] \\
[\![[] : [(\overrightarrow{l : T})]]\!] & = & \mathsf{select}\ \overrightarrow{\mathsf{null}\ \mathsf{as}\ l}\ \mathsf{from}\ \varnothing\ \mathsf{where\ false} \\
[\![\mathsf{for}\ (x \leftarrow \mathsf{table}\ s : T)\ F]\!] & = & \mathsf{select}\ \overrightarrow{e\ \mathsf{as}\ l}\ \mathsf{from}\ s\ \mathsf{as}\ x, \overrightarrow{t\ \mathsf{as}\ y}\ \mathsf{where}\ e \\
 & & \qquad \mathsf{where}\ (\mathsf{select}\ \overrightarrow{e\ \mathsf{as}\ l}\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ y}\ \mathsf{where}\ e) = [\![F]\!] \\
[\![\mathsf{if}\ B\ \mathsf{then}\ Z\ \mathsf{else}\ []]\!] & = & \mathsf{select}\ \overrightarrow{e\ \mathsf{as}\ l}\ \mathsf{from}\ \overrightarrow{t}\ \mathsf{where}\ e' \wedge [\![B]\!] \\
 & & \qquad \mathsf{where}\ (\mathsf{select}\ \overrightarrow{e\ \mathsf{as}\ l}\ \mathsf{from}\ \overrightarrow{t}\ \mathsf{where}\ e') = [\![Z]\!] \\
[\![\mathsf{table}\ s : [(\overrightarrow{l : o})]]\!] & = & \mathsf{select}\ \overrightarrow{s.l\ \mathsf{as}\ l}\ \mathsf{from}\ s\ \mathsf{where\ true} \\
[\![[R]]\!] & = & \mathsf{select}\ [\![R]\!]\ \mathsf{from}\ \varnothing\ \mathsf{where\ true} \\
[\![(\overrightarrow{l = B})]\!] & = & \overrightarrow{[\![B]\!]\ \mathsf{as}\ l} \\
[\![x]\!] & = & x.* \\
[\![\mathsf{if}\ B\ \mathsf{then}\ B'\ \mathsf{else}\ B'']\!] & = & \mathsf{case\ when}\ [\![B]\!]\ \mathsf{then}\ [\![B']\!]\ \mathsf{else}\ [\![B'']\!]\ \mathsf{end} \\
[\![\mathsf{empty}(V)]\!] & = & \neg\mathsf{exists}([\![V]\!]) \\
[\![\oplus(\overrightarrow{B})]\!] & = & \oplus_{\mathrm{sql}}(\overrightarrow{[\![B]\!]}) \\
[\![x.l]\!] & = & x.l \\
[\![c]\!] & = & c
\end{array}
$$

Fig. 6. Translation from normalized terms to SQL.

$$\mathsf{for}\ (x \leftarrow [M])\ N : T \rightsquigarrow N[M/x] \qquad (\textsc{for}\text{-}\beta)$$

$$(\lambda x.N)M : T \rightsquigarrow N[M/x] \qquad (\textsc{abs}\text{-}\beta)$$

$$(\overrightarrow{l = M}).l : T \rightsquigarrow M_l \qquad (\textsc{record}\text{-}\beta)$$

$$\mathsf{for}\ (x \leftarrow [])\ M : T \rightsquigarrow [] \qquad (\textsc{for-zero-l})$$

$$\mathsf{for}\ (x \leftarrow N)\ [] : T \rightsquigarrow [] \qquad (\textsc{for-zero-r})$$

$$\mathsf{for}\ (x \leftarrow \mathsf{for}\ (y \leftarrow L)\ M)\ N : T \rightsquigarrow \mathsf{for}\ (y \leftarrow L)\ (\mathsf{for}\ (x \leftarrow M)\ N) \quad \text{if } y \notin \textsc{fv}(N) \qquad (\textsc{for-assoc})$$

$$\mathsf{for}\ (x \leftarrow M_1 \uplus M_2)\ N : T \rightsquigarrow \mathsf{for}\ (x \leftarrow M_1)\ N \uplus \mathsf{for}\ (x \leftarrow M_2)\ N \qquad (\textsc{for-union-src})$$

$$\mathsf{for}\ (x \leftarrow M)\ (N_1 \uplus N_2) : T \rightsquigarrow \mathsf{for}\ (x \leftarrow M)\ N_1 \uplus \mathsf{for}\ (x \leftarrow M)\ N_2 \qquad (\textsc{for-union-body})$$

$$\mathsf{for}\ (x \leftarrow \mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ [])\ N : T \rightsquigarrow \mathsf{if}\ B\ \mathsf{then}\ (\mathsf{for}\ (x \leftarrow M)\ N)\ \mathsf{else}\ [] \qquad (\textsc{for-if-src})$$

$$(\mathsf{if}\ B\ \mathsf{then}\ L\ \mathsf{else}\ L')M : T \rightsquigarrow \mathsf{if}\ B\ \mathsf{then}\ LM\ \mathsf{else}\ L'M \qquad (\textsc{app-if})$$

$$\mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N : (\overrightarrow{l : T}) \rightsquigarrow (\overrightarrow{l = L}) \quad \text{with } L_l = \mathsf{if}\ B\ \mathsf{then}\ M.l\ \mathsf{else}\ N.l \text{ for each } l \in \overrightarrow{l} \qquad (\textsc{if-record})$$

$$\mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N : [T] \rightsquigarrow \mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ [] \uplus \mathsf{if}\ \neg B\ \mathsf{then}\ N\ \mathsf{else}\ [] \quad \text{if } N \neq [] \qquad (\textsc{if-split})$$

$$\mathsf{if}\ B\ \mathsf{then}\ []\ \mathsf{else}\ [] : T \rightsquigarrow [] \qquad (\textsc{if-zero})$$

$$\mathsf{if}\ B\ \mathsf{then}\ (\mathsf{for}\ (x \leftarrow M)\ N)\ \mathsf{else}\ [] : T \rightsquigarrow \mathsf{for}\ (x \leftarrow M)\ (\mathsf{if}\ B\ \mathsf{then}\ N\ \mathsf{else}\ []) \qquad (\textsc{if-for})$$

$$\mathsf{if}\ B\ \mathsf{then}\ M \uplus N\ \mathsf{else}\ [] : T \rightsquigarrow \mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ [] \uplus \mathsf{if}\ B\ \mathsf{then}\ N\ \mathsf{else}\ [] \qquad (\textsc{if-union})$$

$$\mathsf{empty}(M) : T \rightsquigarrow \mathsf{empty}(\mathsf{for}\ (x \leftarrow M)\ [()]) \quad \text{if } M \text{ is not relation-typed} \qquad (\textsc{empty-flatten})$$

$$\mathsf{query}\ M : T \rightsquigarrow M \qquad (\textsc{ignore-db})$$

Fig. 7. The rewrite system for normalizing source-language terms.

And so the data is unnested. The FOR-ASSOC rule thus exposes β-reductions, which themselves eliminate other constructor/destructor pairs and so reduce the types of intermediate values.

3 Correctness

To show that the translation is correct, we show that the rewrite rules are sound with respect to the static semantics, the rewrite system is strongly normalizing, and its normal forms are the ones given earlier. We merely state the results here; full proofs can be found in an accompanying technical report [5].

Soundness

We use the rewrite rules only on pure terms, so the soundness proposition says that types and purity are preserved by rewriting.

Proposition 1 *If $M \rightsquigarrow M'$ and $\vdash M : T \mid \varnothing$ with T a relation type then $\vdash M' : T \mid \varnothing$. Furthermore $M \rightsquigarrow^* V$ gives then $\vdash V : T \mid \varnothing$.*

In fact, the rewrite rules fail to preserve effects in general, since they may relocate, remove, and duplicate subterms and their effects.

Totality

We want the translation really to give a query for every well-typed query expression. First we show that the rewrite rules strongly normalize—they admit no infinite reduction—then that the normal forms fall into the SQL target.

Proposition 2 *Any well-typed term strongly normalizes.*

The proof uses the reducibility with $\top\top$-lifting approach of Lindley and Stark [12]; our for construct is simply the let of the monadic metalanguage, and there is some additional work to deal with the rule IF-FOR, since it, unusually, pushes a context inside the binder of another.

Proposition 3 (Normal forms) *Closed, well-typed terms of effect-free relation type have normal forms that satisfy the grammar of Fig. 4.*

The proof goes in two steps; first, by induction on the structure of terms, we establish a grammar of all normal forms, where destructors are all pushed to the inside with constructors on the outside. Then, arguments based on the type and effect of query expressions narrow the grammar to the one of Fig. 4.

4 Extensions

4.1 Recursion

A general-purpose programming language without recursion would be severely hobbled; but in standard SQL, many recursive queries are not expressible. Thus we need to add recursion to our language but ban it from query expressions.

We add recursion by introducing a recursive λ-abstraction as follows. It introduces a recursive function of one argument and forces an effect on its type.

$$\frac{\Gamma, f : S \xrightarrow{e \cup \{\mathsf{noqy}\}} T, x : S \vdash M : T \,!\, e}{\Gamma \vdash \mathsf{recfun}\ f\ x = M : S \xrightarrow{e \cup \{\mathsf{noqy}\}} T \,!\, \varnothing} \qquad \text{(T-RecFun)}$$

This prohibition is strong; some recursive functions may be expressible in SQL, and it will be interesting to see if a stronger translation can be given for recursive functions.

4.2 Length operator

The examples in Section 1 used a function *length* which we have not yet studied. This section shows how to extend the system to support it. In the source, it is much like *empty*, but SQL's nonuniformity forces us to handle it specially.

First we extend the source language with length:

$$M ::= \cdots \mid \mathsf{length}(M)$$

The typing rule is as you would expect, resulting in type int.

Extend the SQL-like sublanguage (the normal forms) as follows:

$$B ::= \cdots \mid \mathsf{length}(F)$$

Note that we will normalize the argument to a comprehension normal form, F, so that it gives a select query and not, say, a union all query. Next, augment the SQL target:

$$e ::= \cdots \mid \mathsf{select\ count}(*)\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e$$

And hence we can translate it to SQL as follows:

$$[\![\mathsf{length}(F)]\!] = \mathsf{select\ count}(*)\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e$$
$$\text{where } \mathsf{select}\ \overrightarrow{s}\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e = [\![F]\!]$$

Now we add the following rewrite rules:

$$\mathsf{length}(M) : T \rightsquigarrow \mathsf{length}(\mathsf{for}\ (x \leftarrow M)\ [()]) \qquad \text{(LENGTH-FLATTEN)}$$
$$\text{if } M \text{ is not relation-typed}$$
$$\mathsf{length}([]) : T \rightsquigarrow 0 \qquad \text{(LENGTH-ZERO)}$$
$$\mathsf{length}(M \uplus N) : T \rightsquigarrow \mathsf{length}(M) + \mathsf{length}(N) \qquad \text{(LENGTH-UNION)}$$

Thus assumes, of course, that we have the constant 0 and the integer-addition operation (+) in our set of constants and primitives.

5 Related Work

The science of language-integrated query is young. It might begin with Thomas and Fischer [18], who first defined an algebra of nested relations, thus freeing queries from the flatland of the 1st-normal-form restriction.

Paradaens and Van Gucht [14] gave the first unnesting result, showing that nested relational algebra, when restricted to flat input and output relations, is equivalent in power to traditional flat relational algebra. Wong [22] soon extended this result, showing that any first-order nested relational algebra expression can be rewritten so that it produces no intermediate data structures deeper than the greatest of its input and output relations. Suciu [15] showed that the unnesting holds even in the presence of recursion, through a first-order fixed-point combinator. Suciu and Wong [16] show how first-order functions in a higher-order NRC can be implemented in first-order NRC, provided all the primitive functions have a semantic property called "internal," which requires that they do not output basic values not contained in their input.

Fegaras [8] also shows how to transform higher-order nested relational queries into flat ones using a rewrite system with similarities to ours; we extend this by offering a proof of normalization, taking the queries all the way to SQL, and showing how a type-and-effect system can separate the translatable and untranslatable fragments of a general-purpose language. Van den Bussche [20] extended Wong's first-order result to show that even nested-result-type expressions can be simulated with the flat relational algebra, using an interpretation function which assembles the flat results into a nested relation.

The effort to harmonize query languages with programming languages is nearly as old as the two fields themselves. Atkinson and Buneman [1] survey the early history of integration. The impedance mismatch problem between databases and programming languages is described by Copeland and Maier [7].

The use of comprehension syntax was a breakthrough for language integration, since it gave an iteration construct that was both powerful enough for much general-purpose programming and also explicit enough to admit a more direct translation to a query language; this connection is explored by many authors [19, 2, 4, 11]. Grust [9] summarizes much preceding work on monad comprehensions as a query language.

In a different setting, Wiedermann and Cook [21] present a technique, using *abstract interpretation*, for extracting structured queries from imperative, object-oriented program code, where method dispatch rather than first-class functions is the means of query abstraction.

Kleisli [3, 23] is a functional programming system that allows querying a variety of data sources—including flat relational databases—and allows the use of nested relations as intermediate values and as results. The Kleisli system heuristically pushes constructs inside an SQL query, perhaps leaving some code untranslated. At run-time, it always correctly executes an expression, whether or not it is translatable to SQL. The Kleisli approach has the advantage that all query translation can be done at compile-time.

The LINQ project [13] integrates queries into several programming languages, using an object-oriented interface for forming queries; a comprehension-like syntax is also provided. The interface works with many data sources, including relational DBMSes. In LINQ, one writes query fragments (e.g. conditions and transformations) using a language-level quoting mechanism which captures the code for use at runtime. Quoted and unquoted expressions give different types of values, and they are not interchangeable; in particular, quoted functions cannot be applied to produce new quoted functions, so compositionality is hampered. When translating an expression, LINQ may (a) produce a run-time error, (b) partition the expression into an SQL query and pre- and post-processing phases executed outside SQL, or (c) translate the expression completely to SQL.

The Links project [6] also offers language-integrated query. The original version of Links, like Kleisli, always correctly executes an expression but may not translate it to SQL; Links' execution may be very naive and perhaps ineffecient.

Ferry [10] is a new system which translates a first-order nested query with a nested result into multiple flat SQL queries (a fixed number based on the type), expanding on Van den Bussche and translating the queries all the way to SQL.

Conclusion and Future Work

Expanding on a rich history of theoretical results, we have shown how to extract SQL queries from a host programming language with higher-order functions, using comprehensions as a natural query primitive, as well as a static discipline for checking the translatability of expressions. This permits the first language-integrated-query system which supports functional abstraction and has a runtime SQL translator that is total—that is, once the static check passes, the runtime translator will succeed in translating the expression completely to SQL.

We prevented unlabelled recursion using monomorphic simple types, but polymorphism is essential in programming, so we aim to accomodate it. Sam Lindley has developed a calculus, Frak, which uses type kinds to constrain type variables in a polymorphic calculus, and incorporates effect inference.

We targeted a modest sublanguage of SQL, but would like to go further. For example, we might like to take advantage of grouping and aggregation (using the `group by` clause of SQL) and the nonstandard `limit` and `offset` clauses to fetch subrows. Also, supporting an ordered list type would be useful; the Ferry system handles ordered data, which we hope will inspire further results.

The `query` annotation proposed may be useful for asserting that an expression be SQL-translatable; but the programmer may instead want to allow the compiler more flexibility, e.g. to partition the expression into an SQL query and pre- and post- processing phases, along the lines of what is normally done in LINQ and Kleisli. A science of such partitioning may be called for; this might allow the programmer to write even more query expressions while retaining a high degree of confidence in their run-time efficiency.

References

1. Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.
2. Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *ICDT '92*. Springer, 1992.
3. P. Buneman, S. B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *VLDB '95*, pages 158–169, 1995.
4. Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23:87–96, 1994.
5. Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed (tech report). Technical Report EDI-INF-RR-1327, University of Edinburgh, May 2009.
6. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO '06*, 2006.
7. George Copeland and David Maier. Making smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, 1984.
8. Leonidas Fegaras. Query unnesting in object-oriented databases. In *SIGMOD '98*, pages 49–60, New York, NY, USA, 1998. ACM.
9. Torsten Grust. *The Functional Approach to Data Management*, chapter Monad comprehensions, a versatile representation for queries. Springer Verlag, 2003.
10. Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *SIGMOD '09*, June 2009.
11. Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.
12. Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *TLCA '05*, pages 262–277, 2005.
13. Microsoft Corporation. The LINQ project: .NET language integrated query. White paper, September 2005.
14. Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *PODS '88*, pages 29–38, New York, NY, USA, 1988. ACM.
15. Dan Suciu. Fixpoints and bounded fixpoints for complex objects. In *DBPL '93*, pages 263–281, 1993.
16. Dan Suciu and Limsoon Wong. On two forms of structural recursion. In *ICDT '95*, page 111. Springer, 1995.
17. Jean-pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Information and Computation*, pages 162–173, 1992.
18. S. J. Thomas and P. C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.
19. Phil Trinder. Comprehensions, a query notation for DBPLs. In *DBPL '91*, San Francisco, CA, USA, 1992.
20. Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1-2):363–377, 2001.
21. Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL '07*, 2007.
22. Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.
23. Limsoon Wong. Kleisli, a functional query system. *J. Functional Programming*, 10(1):19–56, January 2000.