Finally, Safely-extensible and Efficient Language-integrated Query

Kenichi Suzuki
University of Tsukuba, Japan
ken@logic.cs.tsukuba.ac.jp

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Yukiyoshi Kameyama
University of Tsukuba, Japan
kameyama@acm.org

Abstract

Language-integrated query is an embedding of database queries into a host language to code queries at a higher level than the all-to-common concatenation of strings of SQL fragments. The eventually produced SQL is ensured to be well-formed and well-typed, and hence free from the embarrassing (security) problems. Language-integrated query takes advantage of the host language's functional and modular abstractions to compose and reuse queries and build query libraries. Furthermore, language-integrated query systems like T-LINQ generate efficient SQL, by applying a number of program transformations to the embedded query. Alas, the set of transformation rules is not designed to be extensible.

We demonstrate a new technique of integrating database queries into a typed functional programming language, so to write *well-typed*, *composable* queries and execute them *efficiently* on any SQL back-end as well as on an in-memory noSQL store. A distinct feature of our framework is that both the query language as well as the transformation rules needed to generate efficient SQL are *safely* user-extensible, to account for many variations in the SQL back-ends, as well for domain-specific knowledge. The transformation rules are guaranteed to be type-preserving and hygienic by their very construction. They can be built from separately developed and reusable parts and arbitrarily composed into optimization pipelines.

With this technique we have embedded into OCaml a relational query language that supports a very large subset of SQL including grouping and aggregation. Its types cover the complete set of intricate SQL behaviors.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code Generation; H.2.3 [*Database Management*]: Languages—Query Languages; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

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

1. Introduction

Writing efficient database queries in a composable, extensible and safe way is a yet-to-be-achieved dream of every database programmer. For better or for worse, SQL still stands as the unique standard query language in widely used database management systems. Mature database systems can execute queries in the classical SQL (SQL-92 Intermediate level) very efficiently by now. The lack of functional abstractions and nested data structures in that language however makes it hard or impossible to build efficient complex queries from simpler, previously written and tested ones, and to compile query libraries. Various ways of embedding SQL into (functional) programming languages – so called language-integrated queries – let us use the abstraction facilities and the type system of the host language to build queries safely and modularly. Yet this composability comes at the expense of efficiency. We illustrate the hard trade-off between efficiency and composability/reuse in §2.

As in numeric computing, metaprogramming comes to the rescue and once again lets us have the "abstraction without guilt:" the performance problem of language-integrated queries is cured by program transformation. Cooper, in the well-titled paper "The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed" [6], proposes transformation rules. One notable implementation of the rules is the T-LINQ system by Cheney et al. [4], which is an embedding of a typed relational query language T-LINQ into F#. Although the system was designed primarily for F#, it could be ported to other languages with typed quotation. An important design choice was that the set of the transformation rules was not user-extensible. Since the implementation is not expected to change, one could afford one-time thorough verification of the code.

In contrast, the starting point for our system is to make both the embedded query language and the set of optimization rules for generating efficient SQL *open*, moreover, *user-extensible*. There are many subtly different SQL implementations with the host of extensions and restrictions, which require adjustments to the embedded query language and its rules. In addition, a programmer may add custom rules to express domain-specific knowledge unavailable to a general-purpose SQL optimizer. The consequence of letting users extend the language and its optimizations is the obligation to make it easy and safe to do so. Any extension should be incremental and reuse as much of the already written code as possible. We should strive to automatically prevent classes of mistakes and ensure some degree of correctness *by construction*.

We have attained the desiderata. Our contribution hence is a practical, *safely-extensible* language-integrated query system:

- It is the system to write *composable* database queries and execute them *efficiently* using any SQL back-end as well as an in-memory noSQL store.
- The user may add domain-specific optimizations to compensate for the deficiencies of the back-end (e.g., MySQL) or to better exploit domain-specific knowledge that may not be available to

the standard SQL query optimizer. The optimizations are type-safe and type-preserving by construction.

- The system is unique in supporting grouping and aggregation (see §7) to the letter of the ANSI SQL standard as well as PostgreSQL extensions: GROUPBY and HAVING clauses may contain arbitrary expressions with the arbitrary mixture of grouped and ungrouped columns. An expression with at least one ungrouped column must be (a part of) an argument of an aggregate function. Our type system accepts all and the only queries that satisfy the complex of SQL grouping rules.
- The query language itself is extensible; as an illustration we show how aggregation and GROUPBY can be added post factum.

At first glance, our system looks like an re-implementation of T-LINQ. However, it is

extensible letting the users extend both the language and the optimization rules;

modular letting extensions re-use as much of the old code as possible and preventing from breaking it;

safe making optimizations type-preserving and hygienic by construction

with full grouping and aggregation in the language type system.

Our system can be easily extended to other back-ends, other SQL features and peculiarities, and other host languages.

Our framework uses the typed final (aka, 'tagless-final') approach (TFA) [3] to embed and optimize a query language. That approach proved convenient for embedding domain-specific languages (DSL): we can express not only the syntax and the denotational semantics of the DSL but also its typing rules, getting the type system of the host language to ensure the type safety of the embedded one. We are hence spared the trouble of implementing a type checker/inferencer for the DSL. An evaluator, a pretty printer, and a code generator can be uniformly implemented as type-respecting interpretations of the terms in DSL. We can also optimize the embedded language, safely and modularly. The optimization technique has been introduced previously [15, 17], but only on small examples. Hence *another contribution* of the present paper is to demonstrate:

- The typed final optimization technique scales up.

One may wonder what fundamentally is being done that could not be done straightforwardly using traditional compiler technology, or the deep embedding (that is, embedding of SQL as a data type). First of all, all our transformations are type- and scope-preserving *by the very construction*. It is not possible to apply a transformation that makes the result ill-typed or leaves unbound variables. It is not even possible to write such a bad transformation and getting it past the type checker. Traditional compiler technology works on an untyped AST and hence makes no mechanically verified promises about the result. Deep embedding, via a datatype, can make some guarantees (using GADTs) but such embeddings are difficult to make extensible, that is, add new forms to the languages and still be able to reuse old transformations as they are.

This paper is organized as follows: the background §2 introduces the running example of sample queries and uses it to informally introduce our embedded query language, called QUEΛ, a superset of T-LINQ of [4]. The section shows how the naive interpretation of the language to query a SQL database leads to severe performance problems, and how program transformation (some form of meta-programming) may help. §3 formally introduces QUEΛ, and its embedding to the host language OCaml in TFA. We briefly illustrate the normalization (transformation) rules of QUEΛ programs in §4. §5 evaluates the performance. §6 shows off the extensibility of our QUEΛ embedding: we add to the language set-based union and re-use, rather than re-write, all the previous interpreters and the transformers. To normalize the extended language we only need to add to the optimization pipeline a couple of transformation rules specifically dealing with the just added operation. §7 extends the target language in the substantial way, with aggregate functions and the GROUPBY clauses. We then discuss the related work and conclude in §9.

The complete code of our system is available online at `http://logic.cs.tsukuba.ac.jp/~ken/quel/`.

2. QUEΛ by Example

This section introduces our query language QUEΛ, modeled after T-LINQ of [4], on several sample queries that serve as our running example. We describe the interpretation of the language in terms of SQL, and see how composing queries leads to very inefficient database interactions. We then outline how program transformations may solve the performance problem. §4 will explain how to actually program these transformations and be sure of their safety.

The language QUEΛ is a simply typed functional language with the primitives for accessing and manipulating data in database tables. Its syntax is formally defined in §3.1; here we illustrate it on very simple and self-explanatory examples. The examples use the database in Fig. 1 with two tables: the products table is a bag (a multiset) of records with the fields for product id (pid), name and price; the orders table has columns for order id (oid), pid, and qty (quantity).

products

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

orders

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

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

Figure 1. Sample database tables and the types of their records

The first sample query Q_1 produces all orders with the given oid. To explain it, we show the corresponding SQL query on the right.

$Q_1 = \lambda oid.$
for $(o \leftarrow$ **table**("orders")$)$
where $(o.\mathsf{oid} = oid)$ **yield** o

```
SELECT o.* FROM orders AS o
WHERE o.oid = oid
```

On the sample database, $run\ (Q_1\ 2)$ returns the following bag:

$$[\langle \mathsf{oid} = 2, \mathsf{pid} = 5, \mathsf{qty} = 10 \rangle, \langle \mathsf{oid} = 2, \mathsf{pid} = 6, \mathsf{qty} = 20 \rangle]$$

where run is a primitive to translate the query into SQL and execute it. The next query Q_2 gets the $Order$ record, finds the product(s) with the pid mentioned in the order and returns the bag of records with the pid, the name, and the sale amount.

$Q_2 = \lambda o.$
for $(p \leftarrow$ **table**("products")$)$
where $(p.\mathsf{pid} = o.\mathsf{pid})$
yield $\langle \mathsf{pid} = p.\mathsf{pid}, \mathsf{name} = p.\mathsf{name}, \mathsf{sale} = p.\mathsf{price} * o.\mathsf{qty} \rangle$

```
SELECT p.pid AS pid,
  p.name AS name,
  p.price * o.qty AS sale
FROM products AS p
WHERE p.pid = o.pid
```

Then $run\ (Q_2\ \langle \mathsf{oid} = 2, \mathsf{pid} = 5, \mathsf{qty} = 10 \rangle)$ returns

$$[\langle \mathsf{pid} = 5, \mathsf{name} = \mathsf{HDD}, \mathsf{sale} = 1000 \rangle]$$

We wish to compose Q_1 and Q_2 to obtain the query Q_3' yielding the bag of records with the total sales for the given oid:

$Q_3{}' = \lambda oid.\, map\ (\lambda o.\, run(Q_2\ o))\ (run(Q_1\ oid))$

Although very natural, $Q_3{}'$ has a serious performance problem, known as Query Avalanche, or the $N+1$ query problem [10]; running $Q_3{}'\ 2$ would issue 3 SQL queries: first Q_1 is run, returning a bag with 2 records. Then Q_2 is run on each record. In general, $Q_3{}'\ oid$ would issue $N+1$ SQL queries, if Q_1 returns a bag of N records. Running a database query has a very high overhead: setting up the communication session with the database server, optimizing the query, setting up and tearing down the transaction, etc. The key to high-performance is to obtain all results in a single database query.

A different way to compose the two queries is by a higher-order function

$compose = \lambda q.\, \lambda r.\, \lambda x.\, \textsf{for}\ (y \leftarrow q\ x)\ \ r\ y$
$Q_3 = \lambda x.\, compose\ Q_1\ Q_2\ x$

Then $Q_3\ oid$ could be interpreted in SQL as

SELECT (Q_2 Ty.oid Ty.pid Ty.qty) **FROM** $Q_1\ oid$ **AS** Ty

Recall however that both Q_1 and Q_2 are interpreted as SQL SELECT statements; therefore Q_3 has nested SELECTs (nested subqueries). Many language-integrated query systems used in practice such as Opaleye [9] and HRR [11] indeed generate nested subqueries. However, the performance problem is only partly solved: Google search for "nested subquery performance" brings the abundance of recommendations to avoid subqueries whenever possible, even for such mature databases as Oracle[1]. MySQL 5.7 documentation states "The optimizer is more mature for joins than for subqueries, so in many cases a statement that uses a subquery can be executed more efficiently if you rewrite it as a join."[2]. The quote hints at the solution: re-writing, which is the approach taken by Cooper [6] and T-LINQ [4]. For example, Q_3, after inlining Q_1 and Q_2 and several beta-reductions, becomes

$Q_3'' = \lambda oid.$
for $(y \leftarrow ($**for** $(o \leftarrow orders)$
  where $(o.\mathsf{oid} = oid)$ **yield** $\langle \mathsf{pid} = o.\mathsf{oid}, \mathsf{qty} = o.\mathsf{qty}\rangle))$
 for $(p \leftarrow products)$
  where $(p.\mathsf{pid} = y.\mathsf{pid})$
  yield $\langle \mathsf{pid} = p.\mathsf{pid}, \mathsf{name} = p.\mathsf{name}, \mathsf{sale} = p.\mathsf{price} * y.\mathsf{qty}\rangle$

A number of transformation steps, described in §4 (for example, lifting **for** and combining adjacent **where**) result in

$Q_3^n = \lambda oid.$ **for** $(o \leftarrow orders)$
  for $(p \leftarrow products)$
  where $(o.\mathsf{oid} = oid) \wedge (p.\mathsf{pid} = o.\mathsf{pid})$
  yield $\langle \mathsf{pid} = p.\mathsf{pid}, \mathsf{name} = p.\mathsf{name}, \mathsf{sale} = p.\mathsf{price} * o.\mathsf{qty}\rangle$

which can be straightforwardly interpreted as the following SQL statement (keeping in mind that nested consecutive **for** compute the Cartesian product, which in SQL is expressed by enumerating the tables in the FROM clause):

SELECT p.pid **AS** pid, p.name **AS** name, p.price ∗ o.qty **AS** sale
FROM products **AS** p, orders **AS** o
WHERE p.pid = o.pid **AND** o.oid = oid

This single SQL query can be efficiently executed in all relational database systems. In the rest of the paper we describe how to program such transformations. But first we have to describe QUEΛ in more detail as well as its embedding as DSL in OCaml.

[1] http://www.remote-dba.net/t_op_sql_tuning_subqueries.htm

[2] http://dev.mysql.com/doc/refman/5.7/en/subquery-restrictions.html

3. Language-integrated query

This section formally introduces the relational query language QUEΛ and embeds it in OCaml in the typed-final style. QUEΛ is deliberately not original: it is Cooper's language without effects typing, and Cheney et al.'s T-LINQ without quotation. We extend it in §7 with aggregates and grouping.

3.1 QUEΛ, its syntax and semantics

The following figure gives types and terms in QUEΛ. We use metavariables x, y for variables, c for constants, t for table names in the database, l for record labels, $\oplus$ for primitive operators (such as arithmetic, comparison, and also $exists$.) The sequence $M_1, \ldots, M_n$ is abbreviated as $\overline{M}$. Types of QUEΛ are base types, function types, bag types (for multisets), and record types $\langle \overline{l:A} \rangle$ where $l_1, \ldots, l_n$ are field labels. We say a record type $\langle \overline{l:A} \rangle$ is *flat* if all A_i are base types, and similarly for a bag type $\mathsf{Bag}\ \langle \overline{l:A} \rangle$.

Base type	O	::=	$Int \mid Bool \mid String$
Type	A, B	::=	$O \mid A \to B \mid \mathsf{Bag}\ A \mid \langle \overline{l:A} \rangle$
Term	L, M, N	::=	$c \mid \oplus(\overline{M}) \mid x \mid \lambda x.\, N \mid L\ M$
		|	$\langle \overline{l = M} \rangle \mid L.l$
		|	**for** $(x \leftarrow M)\ N \mid M \uplus N$
		|	**where** $L\ M$ | **yield** M | $[\,]$ | **table**(t)

QUEΛ terms are the standard lambda terms with primitive operators and records, plus several primitives: **for** $(x \leftarrow M)\ N$ for bag comprehension, $M \uplus N$ for bag union, **yield** M for the singleton bag, $[\,]$ for the empty bag, **where** $L\ M$ for the conditional; **table**(t) denotes the table with the name t.

In many databases, tables may only store the values of basic types; therefore, we assume that the input table **table**(t) has a flat bag type. We can still create and use non-flat records in QUEΛ, which do not have direct counterparts in SQL queries. However, the transformation in §4 always eliminates non-flat records and bags; namely, we can always transform a QUEΛ-term of flat bag type into a term which does not use non-flat records and bags.

The operational semantics of QUEΛ is standard call-by-value with the left-to-right evaluation order for function applications. Values V are constants, abstractions, records with the value components and $[V_1, \ldots, V_n]$, which is the abbreviation for **yield** $V_1 \uplus \ldots \uplus$ **yield** $V_n \uplus [\,]$ for $n \geq 0$. We assume that $\uplus$ is associative and commutative.

The reduction relation $\longrightarrow$ is the same as those of T-LINQ modulo notational difference. The map δ gives semantics to each primitive operator, and Ω maps each table name to a value of a flat bag type. Let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$. Shown below is a sample of primitive reduction rules. Appendix A lists the complete definition.

$$\textsf{table}(t) \longrightarrow \Omega(t)$$
$$\textsf{where true}\ M \longrightarrow M \qquad \textsf{where false}\ M \longrightarrow [\,]$$
$$\textsf{for}\ (x \leftarrow \textsf{yield}\ V)\ M \longrightarrow M[x := V] \qquad \textsf{for}\ (x \leftarrow [\,])\ M \longrightarrow [\,]$$

Typing rules, given in the natural-deduction style, are standard; shown below is a representative sample. The complete presentation is in Appendix A. The typing judgment is of the form $M : A$ for a term M and a type A. The signature Σ maps constants to base types, primitive operators to functions on base types, and table names to flat bag types. We assume that Ω and δ are consistent with Σ. We write $x_1 : B_1, \ldots, x_n : B_n \vdash M : A$ if we can derive $M : A$ under the assumptions $x_1 : B_1, \ldots, x_n : B_n$.

SINGLETON
$$\frac{M : A}{\textsf{yield}\ M : \mathsf{Bag}\ A}$$

FOR
$$\frac{M : \mathsf{Bag}\ A \qquad \begin{array}{c}[x:A] \\ \vdots \\ N : \mathsf{Bag}\ B\end{array}}{\textsf{for}\ (x \leftarrow M)\ N : \mathsf{Bag}\ B}$$

WHERE
$$\frac{L : Bool \qquad M : \mathsf{Bag}\ A}{\textsf{where}\ L\ M : \mathsf{Bag}\ A}$$

THEOREM 1 (Subject reduction). *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow^* N$, *then* $\Gamma \vdash N : A$.

The proof is not difficult but tedious, as we need a lemma: if $\Gamma_1 \vdash V : B$ and $\Gamma_1, \Gamma_2, x : B \vdash M : A$ hold, then $\Gamma_1, \Gamma_2 \vdash M[x := V]$ holds. Our typed-final embedding can be taken as indication or even an automatic proof of this property: see the next section for detail.

3.2 Typed Final Embedding of QUEΛ

This section describes the embedding of QUEΛ in the typed final (a.k.a. 'tagless-final') approach [16]. We will be using OCaml as the host language; Haskell or Scala, etc. may be used as well (see §8 for more discussion).

The typed final approach does a shallow embedding: for each syntactic form of the embedded language we define an OCaml function that will construct the representation of that form. For example, the function `app` constructs the representation of QUEΛ applications. These constructor functions are collected in the module, whose interface is often called *Symantics* since it essentially defines the syntax of the embedded language, and its implementations define the semantics. The interface for QUEΛ from §3.1 is given below.

```
module type Symantics = sig
type α repr (* representation type *)
val int:      int  → int repr
val bool:     bool → bool repr
val string:   string → string repr
val lam:      (α repr → β repr) → (α→β) repr
val app:      (α → β) repr → α repr → β repr
val foreach: (unit→ α list repr) →
            (α repr → β list repr) → β list repr
val where:    bool repr →
            (unit→ α list repr) → α list repr
val yield:    α repr →  α list repr
val nil:      unit → α list repr
val (@%):   α list repr → α list repr →
              α list repr (* bag union *)
val (=%):   α repr → α repr → bool repr
...  (* abbreviated *)
type α obs              (* observation *)
val observe: (unit → α repr) → α obs
end
```

The interface represents not just the syntax of QUEΛ (in the form close to BNF), but also its type system. The type of OCaml expressions that represent QUEΛ terms, typically called `repr`, is indexed by the QUEΛ's type. (We take Bag to be a synonym for `list`.) The type of `foreach`[3] encodes the typing rule FOR, and similarly for the other constructor functions. The embedding has a bit of noise in the form of extra arguments of type unit, to delay the evaluation of conditional branches and to get around the value restriction. We use the symbol `@%` for $\uplus$, `=%` for equality test, and `%.` for projection, whose typing is elided for brevity. See our code for more details. The typed final embedding is tight and faithful, representing all and the only typed embedded-language terms. This property holds for our QUEΛ embedding, as it is easy to see. Ill-typed QUEΛ terms cannot be encoded: their representation will not type check in OCaml. As a bonus, we use the OCaml type inferencer to infer QUEΛ types.

Formally the embedding of QUEΛ is defined in Appendix C. Here we give a few examples: QUEΛ's term $(\lambda x.\ 3 + x)\ 5$ of type Int is represented as `app (fun x → add (int 3) x) (int 5)` of type `int repr`. We use Higher-order Abstract Syntax (HOAS) [5, 13, 19] to represent variable bindings, which lets us use OCaml syntax for bindings and, mainly, saves us from programming α-conversions and worrying about variable name clashes. HOAS lets us faithfully encode the natural-deduction style of typing rules. HOAS also has a seemingly fatal drawback of preventing any inspection or optimization of functions bodies, which is overcome in the typed final approach.

Characteristically for the typed-final approach, the representation type `repr` is kept abstract. Different implementations of the Symantics interface will define `repr` in their own ways, but the encoded term cannot know it. The implementation of Symantics is abstracted away: therefore, a QUEΛ term is represented as an OCaml functor. For example, the query Q_1 from our running example §2 is represented in OCaml as follows:[4]

```
module Q1(S:Symantics) = struct open S
 let res = fun xoid →
   foreach (fun () → table_orders) @@ fun o →
    where ((o %. oid) %= xoid) @@ fun () →
      yield o
end
```

Records of QUEΛ can be implemented in several ways, and we use OCaml objects. Please see Appendix B for details. We rely on the extensibility of the embedding to add the descriptions of tables as constants. The application Q_1 2 then takes the form

```
module Q1_2 (S:Symantics) = struct open S
 module M = Q1(S)
 let res = app M.res (int 2) end
```

It is possible to embed QUEΛ in OCaml more conveniently, without many spurious `fun ()` and using the familiar operator names like = rather than `=%`. For example, the above query might then look as

```
module Q1Nicer(S:Symantics) = struct open S
 let res = fun xoid → fun () →
    let o = table_orders_gen () in
    where ((o %. oid) = xoid); o
end
```

Another possible improvement is to use first-class modules, as demonstrated in [17]. For clarity, we use the (painfully) explicit approach for the time being. We plan to investigate the more convenient embedding in future work.

The representation for Q_2 is similar. For the composed query Q_3 2 (producing total sales for the given oid) we write

```
module Q3(S:Symantics) = struct  open S
 module M1 = Q1(S)  module M2 = Q2(S)
 let q3 x = foreach (fun () → app M1.res x) @@
           fun y → M2.res y
 let res = app q3 (int 2)
end
```

Once written, a QUEΛ representation can be interpreted using any implementation of the Symantics signature. There are several. First is a meta-circular interpreter, typically called `R`:

```
module R = struct
  type α repr = α
  let int n = n  let bool b = b  let string s = s
  let lam f = f
  let app e1 e2 = e1 e2
  let rec foreach tbl f = match tbl () with
    | [] → []
    | t::rest → f t @ (foreach (fun ()→ rest) f)
  let where p e = if p then e () else []
  let yield e = [e]
  let nil () = []
```

[3] We use `foreach` instead of `for`, since the latter is reserved in OCaml.

[4] OCaml's `@@`, like Haskell's `$`, is the low-precedence infix operator for applications.

```
  let (@%) e1 e2 = e1 @ e2
  let (=%) e1 e2 = e1 = e2
  ...
  type α obs = α
  let observe f = f ()
end
```

A QUEΛ term of type α is represented by an OCaml term of the same type, and the evaluation of QUEΛ maps to the evaluation of OCaml. Since OCaml type system (at least of the subset used here) is sound, the R-embedding of QUEΛ is type-sound. Since the representation type `repr` is kept abstract, and since the typed-final encoding is tight, it follows that QUEΛ itself has the property of subject reduction.

To run the query `Q1_2` on an in-memory non-SQL database with the needed tables, we evaluate

```
let module M = Q1_2(R) in
  M.observe (fun () → M.res)
```

obtaining the result shown in §2. Exactly the same `Q1_2` can be run against an SQL database, as we show below. We have used the so far neglected function `observe` to observe the α `repr` value as a value of some observation type α `obs`, which is also kept abstract. The choice of `obs` hence is also left for a Symantics implementation. In the `R` implementation it is the same as `repr` but they generally differ as some post-processing is often needed to observe the result (as we will soon see).

Another implementation of the Symantics interface, called `P`, pretty-prints QUEΛ terms: using it with `Q1`:

```
let module M = Q1(P) in
  print_endline @@ M.observe (fun () → M.res)
```

prints out the query:

```
fun x → foreach (fun ()→ table "orders") (fun y →
              where (y.oid = x) (fun () → yield y))
```

3.3 Normal QUEΛ programs

There is another implementation of the Symantics interface, `GenSQL`, which however works only on normal QUEΛ programs that produce flat record bags. Normal programs are closed irreducible QUEΛ terms under the transformation rules in the next section, and its syntax is defined as follows.

Queries	U	$::=$	$U_1 \uplus U_2 \mid [\,] \mid F$
Comprehensions	F	$::=$	$\textsf{for}\,(x \leftarrow \textsf{table}(t))\; F \mid Z$
Body	Z	$::=$	$\textsf{where}\; B\; Z \mid \textsf{yield}\; R \mid \textsf{table}(t)$
Record	R	$::=$	$\langle \overline{l = B} \rangle \mid x$
Primitives	B	$::=$	$\oplus(\overline{B}) \mid x.l \mid c$

A normal program can be straightforwardly converted to the single SQL query without nested subqueries. For reference, this transformation is described in Appendix D.

For example, the following code that is similar to `Q1_2` produces a flat record bag and is in the normal form. Evaluating it with the `GenSQL` implementation

```
module Q1'(S:Symantics) = struct open S
  let res =
      foreach (fun () → table_orders) @@ fun o →
      where ((oid o) =% (int 2)) @@ fun () →
      yield o
end
let module M = Q1'(GenSQL) in M.observe (fun () →
     M.res)
```

first converts the program to SQL – the same SELECT query we saw in §2 for Q_1. The function `observe` in the GenSQL interpretation does the non-trivial work of sending the generated SQL query to the database server and receiving the result (of the type `order list GenSQL.obs`, which is `order list`). On the other hand, the program `Q3` is not in the normal form and hence cannot be converted to the efficient SQL as it is. It has to be normalized first. That process is described next.

4. Typed-Final Program Transformations

This section briefly describes program transformations in the typed-final style, on an example of bringing in our sample `Q2` and `Q3` programs to the normal form. For clarity and to save space we describe only a couple of transformations, but we have implemented all the normalization rules of Cheney et al. [4] (listed in Appendix E for reference) as well as extensions in §6, all of which preserve well-typedness and well-scopedness *by construction*. §5 demonstrates the good performance of the transformations. The general idea of typed-final transformations, independent of the application domain and applicable to Haskell, OCaml, Scala, etc., has been presented elsewhere [15, 17]. The application to language-integrated query described here is the largest so far application of the typed-final optimization method, demonstrating its expressivity.

In the typed-final approach used in the paper, an term of the embedded DSL (EDSL) is represented as a function (functor) that takes an implementation of the Symantics interface. The only thing to do with such a representation is to pass an implementation of Symantics. Therefore, a term transformation has also to be expressed as a Symantics interpreter.

For a running example we use a simple program transformation $[\,] \uplus N \rightsquigarrow N$. It is the standard constant folding, well-familiar from partial evaluation, which suggests the implementation: annotate each expression with available static information – in our case, if it is statically known to be the empty bag. Let `F` be some implementation of Symantics and hence α `F.repr` be the un-annotated representation type. The annotated type then takes the form of the GADT

```
  type α annrepr =
   | Empty   : α list annrepr
   | Unknown : α F.repr → α annrepr
```

The variant `Unknown` represents the value about which nothing is statically known. The function

```
 let dyn : α annrepr → α F.repr = function
   | Empty → F.nil ()
   | Unknown x → x
```

forgets the statically known information and returns the un-annotated term. We now write the implementation of Symantics that interprets QUEΛ terms in the domain of annotated `F` representation:

```
module LNil_preliminary = struct
  type α repr = α annrepr
  let int  n  = Unknown(F.int n)
  let bool n  = Unknown(F.bool n)
  let app x y = Unknown(F.app (dyn x) (dyn y))
  let lam f   = Unknown(F.lam
                  (fun x → dyn (f (Unknown x))))
  ...
  let nil () = Empty
  let (@%) x y = match x with
    | Empty → y
    | Unknown x → Unknown ((F.@%) x (dyn y))
  type α obs = α F.obs
  let observe x = F.observe (fun () → dyn (x ()))
end
```

As expected, `nil` creates the statically known empty bag; the interpretation of union looks at the annotation and does constant folding.

The observation function extracts the un-annotated term and observes it. The `LNil_preliminary` transformation looks very similar to the simple optimization in [12] (whose development also followed the tagless-final style); one can even relate it to the Kleene's pairing trick of encoding the predecessor in the lambda-calculus. Our particular form of the annotated term α `annrepr` however enables generalization to the extensible optimization framework [15], illustrated below.

Our sample `F` implementation is truly arbitrary, therefore, we abstract it. Thus the nil-suppression transformer takes the following form

```
module LNil(F:Symantics) = struct
  type α annrepr =
    | Empty   : α list annrepr
    | Unknown : α F.repr → α annrepr
  type α repr = α annrepr
  ... (* as before *)
```

That is, the transformation has the form of a functor which takes a Symantics interpreter and produces another Symantics interpreter. To apply it to the sample query `Q1_2` and run the result, we do

```
let module M = Q1_2(LNil(GenSQL)) in
  M.observe (fun () → M.res)
```

One can read this code as doing the transformation `LNil` and interpreting the result with `GenSQL`. One can also read it as interpreting the original `Q1_2` term using the transformed interpreter `LNil (GenSQL)`; the inner `GenSQL` will never see the terms of the form $[\,] \uplus N$ since they will be normalized away. Thus in the typed-final style, *transformations on EDSL terms are written as transformations on their interpreters*.

The above `LNil` functor that implemented the trivial nil-suppression rule had to produce the complete implementation of Symantics. Therefore, it had to define how to interpret booleans, integers and all other QUEΛ expressions in the `annrepr` domain. Only the interpretations of `nil` and union did something for normalization; the rest was the boilerplate. One can eliminate the boilerplate arriving at the general transformation framework [15], which is used in the present paper. Incidentally, [15] discuss at length the differences between deep-embedding optimizations (transforming the data type representation) from the typed-final optimizations. One of the main advantages of the latter is modularity: when new expression forms are added to the language, the previously written optimization passes, if they apply, can be used as they are.

To reiterate the pattern we demonstrate another optimization, the ForFor rule:

$$\textbf{for}\,(x \leftarrow \textbf{for}\,(y \leftarrow L)\,M)\,N \quad \rightsquigarrow$$
$$\textbf{for}\,(y \leftarrow L)\,(\textbf{for}\,(x \leftarrow M)\,N) \qquad (\text{if } y \notin FV(N))$$

where $FV(N)$ denotes the set of free variables in N.

Again we introduce the data type that adds an annotation to the unadorned α `F.repr` type. The annotation relevant for the ForFor transformation is whether a term has the form ($\textbf{for}\,(y \leftarrow L)\,M$) or not:

```
type α annrepr =
  | For     : (unit → α list annrepr) *
              (α annrepr → β list annrepr) →
              β list annrepr
  | Unknown : α F.repr → α annrepr
```

The ever-present `Unknown` represents the value about which nothing is statically known. The annotations can always be forgotten:

```
let rec dyn : α annrepr → α F.repr = function
  | Unknown e → e
  | For (s,b) →
      F.foreach (fun () → dyn @@ s ())
                (fun x → dyn @@ b (Unknown x))
```

The ForFor optimization is implemented quite literally, in a couple of lines of code, as the new interpretation of `foreach` in the interpreter of the annotated terms:

```
let foreach s b' = match s () with
  | Unknown e → For (s, b')
  | For (s,b) → Unknown @@
     F.foreach (fun () → dyn @@ s ()) (fun y →
        F.foreach (fun () → dyn @@ b (Unknown y))
          (fun x → dyn @@ b' (Unknown x)))
```

In our optimization framework, this is essentially all what the programmer has to write to program the optimization. One may wonder however about the side-condition of the ForFor rule: in the result of the transformation, the index variable `y` of the outer loop must not occur free in the body `N` of the inner loop. We have done nothing to satisfy that side condition. We did not have to: the higher-order abstract syntax used for representing loop bodies ensures the condition holds at all times, automatically. This is one more example of assuring safety by the very construction.

Composing program transformations is achieved by simply composing the correspondent functors. For example, to compose the five main transformations of [4] we do

```
module MainPasses(S:Symantics) = AbsBeta(
    RecordBeta(ForFor(ForWhere(ForYield(WhereFor(
    WhereWhere(S)))))))
```

Interpreting our running example `Q3` from §2 and §3 using `MainPasses (GenSQL)` normalizes Q_3 to the form shown at the end of §2, from which the efficient, subquery-free SQL is produced.

Since we have implemented all of the transformations of [4], our system has the property guaranteed by Prop. 4 of Cheney et al.'s paper, that is: Applying a sequence of our transformations to a QUEΛ program of the flat record type eventually produces normal form from which a single, flat SQL statement can be generated.

Our system thus meets its goal. Its performance is addressed next.

5. Performance

This section describes performance of our language-integrated query system, embedding of QUEΛ into OCaml. The performance has several components: optimizing the user-entered QUEΛ expression, generating the SQL code, sending it to the database server and executing. First we evaluate the former two contributions. Tab. 1 summarizes the results. The execution environment is MacBook Pro 11,1 with Intel Core i5-4288U CPU; we used bytecode OCaml 4.01.0. The sample query was `compose` of Cheney et al.[4]. We implemented its transformation in two ways, the first of which is `MainPasses` in §4 and the second is `AllPasses` defined as follows.

```
module AllPasses(S:Symantics) = AbsBeta(
    RecordBeta(ForYield(ForFor(ForWhere(ForEmpty1
    (ForUnionAll1(WhereTrue(WhereFalse(
    ForUnionAll2(ForEmpty2(WhereEmpty(WhereWhere(
    WhereFor(S))))))))))))))
```

The latter applies all transformation rules, while the former applies only rules necessary to normalize the given query.

We also implemented two different iteration policies. The first one iterates the transformation for a given number, set to 10 in this experiment. The other is to iterate the transformation until the given term is in the normal form (NF).

In total, we have four difference cases (two for the order of primitive transformations, two for the policy of iterations). In Tab. 1 the first four lines show the total execution time of the program transformations and SQL generation in our implementation. For the

AllPasses for 10 times	324.20
AllPasses until NF	6.68
MainPasses for 10 times	0.49
MainPasses until NF	0.18
P-LINQ	0.8

Time in milliseconds.

Table 1. Execution time for the `compose` query

purpose of comparison, the last line shows the execution time for the program transformation in [4].

Although our typed-final embedding is a proof-of-concept implementation, it runs within a modest time-bound even if we apply all possible transformation rules to the given query, provided we stop the iteration as soon as the target term becomes a normal form (the second line). If we selectively apply the necessary transformation rules only, it even outperforms the efficient implementation in the literature (the last three lines). We stress that the shown running time is not just the transformation time; it includes the time for generating SQL code.

Compared to the time of executing the query against a database, which typically runs for seconds if not minutes, all optimization and SQL generation steps clearly take negligible time.

The most interesting question is how fast the generated SQL code executes. This question is also difficult to answer as benchmarking database performance is very hard due to a large number of contributing factors. Fortunately, we can reduce this question to an already solved problem. Our QUEΛ system has the same input language as T-LINQ (modulo notational differences between F# and our encoding) – and it generates exactly the same SQL code as T-LINQ does. Therefore, all the extensive empirical evaluation results of running the queries done in the T-LINQ paper [4, §9] apply as they are. We refer the reader to that paper for all detail.

6. Extension

This section shows off the extensibility, by adding one small extension: set-based union. The union operation `(@%)` used before was the multiset union, or UNION ALL in SQL terms. It turns out we can re-use, rather than re-write, all previously written interpreters and transformers. Extending QUEΛ is as simple as it can get.

First we extend the syntax of QUEΛ with the new set-based union operation, to be called `(@^)`:

```
module type SymanticsS = sig
  include Symantics
  val (@^) : α list repr → α list repr →
             α list repr
end
```

The code literally adds a new declaration to the existing set of declarations `Symantics`, fully reusing the latter.

Next we have to extend the interpreters of QUEΛ, to handle the newly added form. The extensions are just as straightforward, fully reusing the existing interpreters as they were. For example, for the `R` interpreter, we write

```
module RS = struct
  include R
  let (@^) xs ys =
     List.fold_right (fun x l →
       if List.mem x l then l else x::l) xs ys
end
```

The existing optimization passes apply to the extended QUEΛ exactly as they were, with no changes. We do need to add new passes that concern the set-based union, for example

$$\textbf{for}\,(x \leftarrow L \,@\hat{}\, M)\, N \quad\leadsto\quad \textbf{for}\,(x \leftarrow L)\, N \,@\hat{}\, \textbf{for}\,(x \leftarrow M)\, N$$

This rule is programmed similarly to other optimization rules, as described in §4.

That is all we need to use the new feature and normalize queries with its feature.

7. Grouping and Aggregation

This section extends QUEΛ post-factum by the group-by clause and aggregate functions, which are frequently used in practice. Adding them to the language-integrated query in a type-safe and efficient way has been a challenge (which we review in §8). We meet the challenge here, also demonstrating that our system is truly extensible: we reuse, *as they are*, all previously written interpreters and optimization passes.

Our sample query runs against the database in §2 and counts the total sales classified by categories. It can be written in the extended QUEΛ as follows. The corresponding SQL query is shown on the right.

$Q_4 =$
for $(o \leftarrow orders)$
for $(p \leftarrow products)$
where $(o.pid = p.pid$
$\quad \wedge\ p.price > 200)$
group $(gprice \leftarrow p.price)$
having $(\textbf{sum}(p.price * o.qty)$
$\quad > 500)$
gyield $\langle gprice,$
$\quad \textbf{sum}\ (p.price * o.qty)\rangle$

```
SELECT p.price,
  SUM(p.price * o.qty)
FROM orders AS o,
     products AS p
WHERE o.pid = p.pid
 AND p.price > 200
GROUP BY p.price
HAVING
  SUM(p.price * o.qty) > 500
```

The query Q_4 joins the tables `orders` and `products`, selects the products with the price higher than 200, groups by price, keeps only the groups with sales total higher than 500, and finally lists the price and the sales total. Its result is

```
[<1000, 55000>, <500, 12500>]
```

Such a query is very common in practice.

Types	$A, B ::= \cdots \mid A \times B$
Terms	$L, M, N ::= \cdots \mid \textbf{group}\ (\overline{g \leftarrow M})\ N \mid \textbf{having}\ L\ M$
	$\mid\ \textbf{gyield}\ M \mid \langle M, N\rangle \mid \circledcirc(M)$

Figure 2. Syntax of QUEΛ_G. The metavariable $\circledcirc$ ranges over aggregate functions such as `sum` and `average`.

To implement Q_4 we extend QUEΛ with several new primitives, obtaining QUEΛ_G; see Fig. 2. The **group** keyword corresponds to the `GROUP BY` clause in SQL. In the example above, `group (gprice ← p.price)` groups the bag of records generated by the outer **for** and **where** clauses by the key `p.price`. For the sample database in Fig. 1, these clauses produce the bag

```
[<name=Tablet,price=500,qty=5>,
 <name=Laptop,price=1000,qty=5>,...]
```

The **group** statement then builds a bag of bags

```
[<gprice=500,  [<name=Tablet,price=500,qty=5>,
                <name=SSD,price=500,qty=20>]>,
 <gprice=1000,[<name=Laptop,price=1000,qty=5>,
                <name=Laptop,price=1000,qty=50>]>]
```

which then enumerates. The variable `gprice` is bound to the price value within each bag. Since this is a grouping key, all records within the inner bag have the same value of `price`. The **having** and **gyield** primitives are similar to **where** and **yield**, respectively, and are used only with a **group** clause, that is, within the bag-of-bag enumeration. The former filters whereas the latter selects from the tuples of the outer bag. In the first tuple, `price` has the value 500, whereas `qty`, not the grouping key, does not have the fixed value: it is `5` for the first inner record and `20` for the second. Likewise, the product of the price and quantity does not have the fixed value. The aggregate function `sum` *aggregates* over the inner bag; in our case, it sums up the total sales within each inner bag to the single scalar value. The semantics just explained is intuitive semantics but not compositional: when explaining **group** we had to refer to the bag produced by the outer **for** clauses. Describing the semantics in terms of building and enumerating bags also gives a poor guidance to implementors (since it is hard to implement efficiently).

We have designed compositional semantics of grouping and aggregation, which we implemented as an `R` interpreter and used to run Q_4. We cannot describe this semantics here for the lack of space and have to refer the reader to the accompanying source code. In the rest of the section we explain the type system of QUEΛ$_G$.

The goal of the type system is to statically enforce the restriction on expressions that may appear with **having** and **yield** clauses: such expressions may only contain constants, aggregated values, and group keys, and have the fixed value for all records within one group. Here are examples of acceptable expressions:

$$gprice,\quad \textbf{sum}(o.qty),\quad \textbf{sum}(p.price),\quad gprice * \textbf{sum}(o.qty),\quad \textbf{sum}(p.price * o.qty * weightfactor(p.name))$$

An argument of an aggregate function on the other hand may freely mix fixed-value expressions such as group keys with references to ungrouped columns. The last expression above is an example; $weightfactor$ is a user-defined function.

The type system for QUEΛ$_G$ uses two new judgments: $\vdash_a M : A$ and $\vdash_G M : A$ in addition to the previously introduced one $M : A$ (here we write $\vdash M : A$ to distinguish it from others). The former defines expressions that have the fixed value within a group and are hence acceptable for **having** and **yield** clauses. The judgment $\vdash_G M : A$ is used for typing those clauses, which can only be used within grouping. Fig. 3 shows the new typing rules with the new judgments.

It is now easy to extend the definition of Symantics to accommodate grouping and aggregation. In fact, we developed the other way round; we first implemented Symantics and then formulated the typing rules in Fig. 3. We note that the type system of the host language is very helpful in designing and developing correct typing rules, which is not possible in other formulations. The typed final approach truly helps.

The following code shows the extended Symantics corresponding to the typing rules in Fig. 3.

```
module type SymanticsG = sig
 include SymanticsL
 type (α,β,'key) grepr
 type (α,β,'key) gres
 type (α,β,'groupkeys,'reskeys) coll

 val group : 'gk gb_sequence →
   ('gk gb_key_sequence → (α,β,'res) gres) →
     (α,β,'gk,'res) coll list repr
 val gint : int   → (int,int,constk) grepr
 val sum  : int repr → (int,int,sumk) grepr
 val gpair:(α,β1,'k1) grepr → (β,β2,'k2) grepr
           → (α * β,β1*β2,'k1*'k2) grepr
 val having : (bool,β1,'k1) grepr →
              (unit → (α,β2,'k2) gres) →
              (α,β1*β2,'k1*'k2) gres
 val gyield : (α,β,'key) grepr → (α,β,'key) gres
 (* ... abbreviated ... *)
end
```

The type `(α,β,'key) grepr` corresponds to the second judgment $\vdash_a M : \alpha$ in Fig. 3, where we ignore the arguments β and `'key` which are used for final observation only. The function `gint` implements the typing rule `CONST` when O is `int`, and `sum` and `gpair`, resp., implement the typing rules `AGGREGATION` (when ⊚ is `sum`) and `pair`, resp. Similarly, the types `(α,β,'key) gres` and `(α,β,'groupkeys,'reskeys) coll`, resp., correspond to the judgments $\vdash_G M : \alpha$ and $\vdash M : \alpha$, resp.

We can write the query Q_4 in QUEΛ$_G$ as follows.

```
module Q4(S:SYM_SCHEMA) = struct
 open S
 let products = table("products", products ())
 let orders   = table("orders", orders ())

 let res =
  foreach (fun () → orders) @@ fun o →
  foreach (fun () → products) @@ fun p →
  where ((o %. opid =% p %. pid) &%
          (p %. price >% int 100)) @@ fun () →
  group (seq_one (p %. price)) @@ seq_decon
    (fun gprice _ →
      having (sum (p %. price *% o %. qty)
              >$ gint 500) @@ fun () →
      gyield (gpair gprice
            (sum ((p %. price) *% (o %. qty)))))
end
```

This query is complicated, but the type system again helps us to formulate it correctly. Due to lack of space, we refer the reader to the accompanying code for further details.

To repeat, our implementation stands out in supporting grouping and aggregation to the letter of the ANSI SQL standard as well as PostgreSQL extensions: GROUPBY and HAVING clauses may contain arbitrary expressions with the arbitrary mixture of grouped and ungrouped columns. An expression with at least one ungrouped column must be (a part of) an argument of an aggregate function. Our type system accepts all and the only queries that satisfy the complex of SQL grouping rules.

$\boxed{\vdash M : A}$

GROUP

$$\frac{\vdash \overline{M : A} \qquad \vdash_G N : \mathsf{Bag}\ B \quad [\vdash_a \overline{g : A}]}{\vdash \textbf{group}\ (\overline{g \leftarrow M})\ N : \mathsf{Bag}\ B}$$

$\boxed{\vdash_a M : A}$

CONST

$$\frac{\Sigma(c) = O}{\vdash_a\ c : O}$$

OP

$$\frac{\Sigma(\oplus) = O_1 \times \cdots \times O_n \to O \qquad \vdash_a\ M_i : O_i\ (\text{for each } 1 \leq\ i \leq\ n)}{\vdash_a\ \ \oplus(\overline{M}) : O}$$

AGGREGATION

$$\frac{\Sigma(\circledcirc) = O_1 \to O_2 \qquad \vdash M : O_1}{\vdash_a \circledcirc(M) : O_2}$$

PAIR

$$\frac{\vdash_a M_1 : A_1 \qquad \vdash_a M_2 : A_2}{\vdash_a \langle M_1, M_2\rangle : A_1 \times A_2}$$

$\boxed{\vdash_G M : A}$

HAVING

$$\frac{\vdash_a L : Bool \qquad \vdash_G M : \mathsf{Bag}\ A}{\vdash_G \textbf{having}\ L\ M : \mathsf{Bag}\ A}$$

SINGLETON

$$\frac{\vdash_a M : A}{\vdash_G \textbf{gyield}\ M : \mathsf{Bag}\ A}$$

Figure 3. Typing rules of QUEΛ$_G$

8. Related Work

Language-integrated query is an old topic, dating back to [22]. The Nested Relational Calculus (NRC) of Buneman et al. [2] provides the foundation for query languages on comprehension. Cooper presents a strongly normalizing rewriting system for NRC [6]. Cheney et al.'s T-LINQ [4] refines Cooper's normalization. We use the T-LINQ rules as they are in our system.

The language-integrated query of T-LINQ was designed to understand and improve SQL extensions of F#, and so naturally took advantage of the F# quotation mechanism and Microsoft F# LINQ library. Normalization transformations are implemented on the untyped, first-order representation of their query language. One has to be careful: some rules involve beta-reductions under binders, so one has to be sure to do alpha-conversion to avoid variable capture. Although theoretically trivial, such operations are tedious and a source of subtle bugs. Correctness, including type preservation, confluence and termination, have all been proven – offline. Since nothing is assured by construction, upon modification of the rules or the language, the proofs have to be reworked.

Our approach does not use any language-specific meta-programming facilities (quotation, etc). Although implemented in OCaml, it can easily be ported to Haskell or Scala, for example.

HRR [11] and Opaleye [9] (and the older and less capable HaskellDB [18] and others) are meant for industrial-scale applications and support a very large subset of SQL. (It is not clear if they support all of the GROUPBY facility as mandated by the SQL standard.) Queries are composable; however, they result in SQL with (sometimes, very many) nested SELECT statements, which is known to be suboptimal. The author of Opaleye does consider this to be weakness and intends to redress it at some point.

Some functional language systems natively support type-safe SQL queries: for example, Ohori et al.'s SML# [20] and Cooper et al.'s early version of Links [7]. Composing SQL statements is either not allowed (SML#) or leads to the query avalanche problem.

Rompf and Amin [21] describe compiling simple SQL queries to the very efficient C by a sequence of small transformations, ultimately realizing one of the Holy Grails of partial evaluation: obtaining an efficient compiler by specializing an interpreter to the program. Their paper is another good illustration of Lightweight Modular Staging (LMS). Whereas in Rompf and Amin's approach, SQL is the input, we take SQL as the output of our transformation chain, relaying on the off-the-shelf data base engines for its execution.

Peyton Jones and Wadler [14] proposed an extension of list comprehensions of Haskell to be called ComCom, inspired by SQL's GROUP BY and ORDER BY. It has an elegant design going beyond SQL in generality (for example, getting group-by to compute running average). Also, Haskell list comprehensions do not have to produce flat list of base-type tuples; nested lists are easily possible. For these reasons, ComCom comprehensions cannot be translated to SQL. Although more general than SQL, ComCom is also, unexpectedly, less expressive than SQL. Furthermore, the errors that database systems typically catch when compiling a SQL statement and we prevent using types, are detected by ComCom only at run-time. Concomitant with the weak typing is the loss of efficiency.

Peyton Jones and Wadler's design has a surprising and unique (as the paper itself calls) feature – implicit rebinding of variables, which changes their types. For example, the following SQL query against the table `employees :: [(Name,Dept,Float)]`

```
SELECT dept, SUM(salary) FROM employees
  GROUP BY dept
```

is represented by the following comprehension:

```
   [ (the dept, sum salary)
   | (name, dept, salary) <— employees,
     group by dept ]
```

On the second line, the variable `dept` (explicitly bound by the pattern) has the type `Dept`. Yet on the first line, the same variable has the type `[Dept]`. This feature has even more unpleasant surprises. Consider a slightly changed query

```
SELECT dept, salary FROM employees GROUP BY dept
```

with the following comprehension:

```
[ (the dept, the salary)
| (name, dept, salary) <— employees ,
  group by dept ]
```

The new query is invalid according to the SQL standard: the column `salary` is not grouped by and it appears in the `SELECT` list by itself rather than within an aggregate. SQL engines will reject the query when compiling it. In our QUEΛ_G, it is ill-typed. Yet the Haskell comprehension is well-typed and can even be run, with a run-time error. The function `the`

```
the :: Eq a => [a] → a
the (x:xs) | all (x ==) xs = x
```

is partial. The expression `the dept` succeeds since `dept` is a list of identical values because it is a group key. However, `the salary` fails since the ungrouped `salary` is in general the list of distinct values. Because of this typing flaw, the function `the` has to compare all list values, every time it is applied.

The implicit rebinding also prevents realizing valid SQL statements such as

```
SELECT dept, SUM(salary) FROM employees
GROUP BY dept
HAVING SUM (deptweight(dept) * salary) > 2000
```

Assume `deptweight` is a SQL function that assigns a weight factor to a `dept`[5]. Its straightforward translation to the comprehension

```
   [ (the dept, the salary)
   | (name, dept, salary) <— employees
   , group by dept
   , sum (deptweight(dept) * salary) > 2000 ]
```

is ill-typed since `group` implicitly rebound `dept` to the type `[Dept]`, which cannot then be passed to the `deptweight :: Dept → Float` function.

Our QUEΛ is unique in fully matching the SQL behavior, accepting compilable SQL statements and statically rejecting invalid statements with a type error.

Typed final, or tagless-final style is introduced in [3] and further described in [16]. The former paper has the extensive comparison with the related work in this area, to which the latter adds the discussion of Böhm-Berarducci encodings of data types [6] [1]. Unlike the tagless-final representation, Böhm-Berarducci encodings are defined only for strictly positive data types, and, mainly, are not extensible. Tagless-final style was recently applied to OO languages, under the name of object algebras [8].

This paper described the tagless-final embedding of QUEΛ into OCaml. We could have just as well used Haskell or Scala as host languages. In fact, we have performed a small experiment with the Haskell embedding [15]. Haskell typeclasses made the encoding lightweight compared to OCaml modules. On the other hand, in

[5] The example is patterned after the one in the PostgreSQL documentation `http://www.postgresql.org/docs/9.4/interactive/queries-table-expressions.html#QUERIES-GROUP`

[6] which are frequently confused with Church encodings; see `http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html` for the explanation of the differences.

OCaml we relied on the `include` mechanism to program optimizations by reusing the code for the identity transformation and overriding a couple of definitions. Haskell does not support that sort of code reuse among type classes. Therefore, programming tagless-final transformation in Haskell has quite a bit of boilerplate.

9. Conclusion and Future work

Building high-performance, practical applications and tools is hard because performance is often at odds with features that make programming manageable: early error detection, abstracting away micromanagement and boilerplate, reusing previously developed and tested components in new and bigger ones. We faced two such hard trade-offs in this paper: First, tagless-final approach with higher-order abstract syntax makes embedding of higher-order typed languages as simple as it can probably be, offering extensibility, ensuring well-typedness and hygiene by construction and taking advantage of the host language type checker to check and even infer embedded language types. On the other hand, such embedding was considered to be impossible to optimize. On the application side, language-integrated queries are composable and reusable, with good error detection, thanks to the host language type system. On the other hand, it is very hard to generate efficient SQL to communicate with mature high-performance database engines.

In this paper we overcome both trade-offs. We show that optimizing tagless-final embedded languages is not only possible but advantageous. Optimization rules are also well-typed and type- and scope-preserving by construction. The rules can be reused even in face of extensions to the language. Optimizations rules may be arbitrarily assembled into optimization pipelines; a programmer may easily reassemble the pipeline or add previously or newly written passes to account for peculiarities of database engines or domain-specific knowledge (e.g., presence or importance of NULLs).

We have not only shown the general framework for embedding and optimizing DSL, but also applied it to language-integrated queries. We implemented all optimization rules proposed in the earlier work by Cooper and Cheney et al. and thus are able to generate efficient SQL. Taking advantage of the extensibility of the framework we extended the language and the optimizations to the complete SQL grouping and aggregation behavior. Our implementation of the meta-circular interpreter can be seen a first formal semantics of this facility.

In the future work, we plan to investigate more convenient, comprehension-like surface syntax, demonstrated in §3. We would also like to automatically generate QUEΛ's table types from database schema.

Acknowledgments.

We are immensely grateful to James Cheney for the explanations of the T-LINQ implementation and great many helpful comments. We thank Kazu Yamamoto for many helpful discussions, and anonymous reviewers for constructive comments.

The third author is supported in part by JSPS Grant-in-Aid for Scientific Research No. 25280020.

References

A. Operational Semantics and Typing of QUEΛ

We define operational semantics in Fig. 4 and 5, and typing rules in Fig. 6. Recall that $[V_1, \cdots, V_n]$ is an abbreviated expression for $\textsf{yield}\ V_1 \uplus \ldots \uplus \textsf{yield}\ V_n \uplus [\,]$.

Value

$$V \quad ::= \quad c \mid \lambda x.\, M \mid \langle \overline{l = V} \rangle \mid [V_1, \cdots, V_n]$$

Evaluation context

$$\begin{array}{lcl} \mathcal{E} & ::= & [\,] \mid \oplus(\overline{V}, \mathcal{E}, \overline{M}) \mid \mathcal{E}\ M \mid V\ \mathcal{E} \\ & \mid & \langle \overline{l = V}, l' = \mathcal{E}, \overline{l'' = M} \rangle \mid \mathcal{E}.l \mid \textsf{yield}\ \mathcal{E} \\ & \mid & \mathcal{E} \uplus M \mid V \uplus \mathcal{E} \mid \textsf{for}\ (x \leftarrow \mathcal{E})\ N \\ & \mid & \textsf{where}\ \mathcal{E}\ M \end{array}$$

Figure 4. Values and Evaluation Contexts

$$\begin{array}{rcl} \oplus(\overline{V}) & \longrightarrow & \delta(\oplus)(\overline{V}) \\ \textsf{table}(t) & \longrightarrow & \Omega(t) \\ (\lambda x.\, M)\ V & \longrightarrow & M[x := V] \\ \langle \overline{l = V} \rangle.l_i & \longrightarrow & V_i \\ \textsf{where true}\ M & \longrightarrow & M \\ \textsf{where false}\ M & \longrightarrow & [\,] \\ \textsf{for}\ (x \leftarrow \textsf{yield}\ V)\ M & \longrightarrow & M[x := V] \\ \textsf{for}\ (x \leftarrow [\,])\ M & \longrightarrow & [\,] \\ \textsf{for}\ (x \leftarrow L \uplus M)\ N & \longrightarrow & \end{array}$$
$$(\textsf{for}\ (x \leftarrow L)\ N) \uplus (\textsf{for}\ (x \leftarrow M)\ N)$$

$$\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}\ (\textrm{E-CONTEXT})$$

Figure 5. Operational Semantics of QUEΛ

The map δ gives semantics to each primitive operator, and Ω maps each table name to a value of a flat bag type.

We assume that Ω and δ in the previous subsection are consistent with Σ: for each table $\Omega(t)$ is assumed to be a value of type $\Sigma(t)$, and δ respect types: if $\Sigma(\oplus) = \overline{O} \to O$ and $\vdash \overline{V : O}$ and $V = \delta(\oplus, \overline{V})$ then $\vdash V : O$.

B. Encoding Schemata and Records

We briefly mention how we encoded database schemata and records.

To encode records in the object language, we use objects in OCaml. Record construction is encoded as a function which creates an object, and record projection is encoded as a function that takes a record (encoded as an object) and returns its field value. A database schema is then encoded as a signature for these functions. Fig. 7 is an example schema encoded as a signature.

To use the signature for a schema, we extend a standard Symantics with the schema as follows.

```
module type SYM_SCHEMA = sig
  include SymanticsL
  include SCHEMA  with type α repr := α repr
end
```

In this paper, we implicitly assumed that we use this extended signature as a Symantics. Interpreters are similarly extended.

$$\textsc{Const}\ \frac{\Sigma(c) = O}{c : O} \qquad \textsc{Op}\ \frac{\Sigma(\oplus) = O_1 \times \cdots \times O_n \to O \quad M_i : O_i\ (\text{for each } 1 \le i \le n)}{\oplus(\overline{M}) : O}$$

$$\textsc{Abs}\ \frac{\begin{array}{c}[x : A] \\ \mid \\ M : B\end{array}}{\lambda x.\, M : A \to B} \qquad \textsc{App}\ \frac{L : A \to B \qquad M : A}{L\ M : B}$$

$$\textsc{Record}\ \frac{M_i : A_i\ (\text{for each} 1 \le i \le n)}{\langle \overline{l = M} \rangle : \langle \overline{l : A} \rangle} \qquad \textsc{Project}\ \frac{M : \langle \overline{l : A} \rangle}{M.l_i : A_i}$$

$$\textsc{Singleton}\ \frac{M : A}{\textsf{yield}\ M : \textsf{Bag}\ A} \qquad \textsc{Empty}\ \frac{}{[\,] : \textsf{Bag}\ A}$$

$$\textsc{UnionAll}\ \frac{M : \textsf{Bag}\ A \qquad N : \textsf{Bag}\ A}{M \uplus N : \textsf{Bag}\ A} \qquad \textsc{Table}\ \frac{\Sigma(t) = \textsf{Bag}\ \langle \overline{l : O} \rangle}{\textsf{table}(t) : \textsf{Bag}\ \langle \overline{l : O} \rangle}$$

$$\textsc{For}\ \frac{M : \textsf{Bag}\ A \qquad \begin{array}{c}[x : A] \\ \mid \\ N : \textsf{Bag}\ B\end{array}}{\textsf{for}\ (x \leftarrow M)\ N : \textsf{Bag}\ B} \qquad \textsc{Where}\ \frac{L : Bool \qquad M : \textsf{Bag}\ A}{\textsf{where}\ L\ M : \textsf{Bag}\ A}$$

Figure 6. Typing rules

```
module type SCHEMA = sig
  type α repr

  (* record constructors *)
  val product : int repr → string repr →
    int repr → <pid:int; name:string; price:int>
    repr
  ...

  (* projection *)
  val pid   : <pid:int; name:string; price:int>
    repr → int repr
  val name  : <pid:int; name:string; price:int>
    repr → string repr
  val price: <pid:int; name:string; price:int>
    repr → int repr
  ...

  (* data sources *)
  val products : unit → <pid:int; name:string;
    price:int> list
  ...
end
```

Figure 7. An Example Schema as a Signature

C. Embedding the Object Language

Fig. 8 defines the embedding function $\mathcal{M}$ that maps the terms and types in the object language into those in the metalanguage.

$<\overline{l : A}*>$ is the type for objects in OCaml. The function $const_O$ maps constants in the object language to OCaml constants of type

$$
\begin{array}{rcl}
\mathcal{M}[\![A]\!] &=& A^*\ \texttt{repr}\\
Int^* &=& \texttt{int}\\
Bool^* &=& \texttt{bool}\\
String^* &=& \texttt{string}\\
(A \to B)^* &=& A^*\ \to\ B^*\\
(\mathsf{Bag}\ A)^* &=& A^*\ \texttt{list}\\
(\langle \overline{l : A}\rangle)^* &=& \texttt{<}\overline{l : A^*}\texttt{>}\\
\\
\mathcal{M}[\![x]\!] &=& x\\
\mathcal{M}[\![c]\!] &=& const_O(c)\\
\mathcal{M}[\![\oplus(\overline{M})]\!] &=& \oplus_{\mathcal{M}}(\overline{\mathcal{M}[\![M]\!]})\\
\mathcal{M}[\![L\ M]\!] &=& \texttt{app}\ \mathcal{M}[\![L]\!]\ \mathcal{M}[\![M]\!]\\
\mathcal{M}[\![\lambda x.\, N]\!] &=& \texttt{lam}\ (\textbf{fun}\ x\ \to\ \mathcal{M}[\![N]\!])\\
\\
\mathcal{M}[\![\langle \overline{l = M}\rangle]\!] &=& record_{\bar{l}}(\overline{\mathcal{M}[\![M]\!]})\\
\mathcal{M}[\![L.l]\!] &=& \mathcal{M}[\![L]\!]\ \texttt{\%.}\ l\\
\mathcal{M}[\![\textbf{for}\ (x \leftarrow M)\ N]\!] &=& \texttt{foreach}\ (\textbf{fun}\ ()\ \to\ \mathcal{M}[\![M]\!])\\
&& \quad (\textbf{fun}\ x\ \to\ \mathcal{M}[\![N]\!])\\
\mathcal{M}[\![\textbf{where}\ L\ M]\!] &=& \texttt{where}\ (\mathcal{M}[\![L]\!])\\
&& \quad (\textbf{fun}\ ()\ \to\ \mathcal{M}[\![M]\!])\\
\mathcal{M}[\![\textbf{yield}\ M]\!] &=& \texttt{yield}\ \mathcal{M}[\![M]\!]\\
\mathcal{M}[\![[\,]]\!] &=& \texttt{nil}\ ()\\
\mathcal{M}[\![M \uplus N]\!] &=& \mathcal{M}[\![M]\!]\ \texttt{@\%}\ \mathcal{M}[\![N]\!]\\
\mathcal{M}[\![\textbf{table}(t)]\!] &=& \texttt{table}(t, \Omega(t))
\end{array}
$$

Figure 8. Embedding QUEΛ into OCaml

O. We assume that $record_{\bar{l}}$ builds OCaml objects with labels $\bar{l}$, such as `products` in Fig. 7. `%.` is record projection for the label l.

D. Generating SQL

After applying transformations to a given closed query of flat bag type, we will get its normal form, which is then translated to a SQL query. Fig. 9 shows Cooper's translation for this last step where ϕ denotes an empty database table. We list it here for completeness.

$$
\begin{array}{l}
\mathcal{S}[\![U_1 \uplus U_2]\!] = \mathcal{S}[\![U_1]\!]\ \textsf{UNION ALL}\ \mathcal{S}[\![U_2]\!]\\
\mathcal{S}[\![[\,]]\!] = \textsf{SELECT}\ \overline{null\ \textsf{AS}\ l}\ \textsf{FROM}\ \phi\ \textsf{WHERE FALSE}\\
\\
\mathcal{S}[\![F]\!] = \textsf{SELECT}\ \overline{e\ \textsf{AS}\ l}\ \textsf{FROM}\ s\ \textsf{AS}\ x, \overline{t\ \textsf{AS}\ y}\ \textsf{WHERE}\ B\\
\quad where\ F = \textbf{for}\ (x \leftarrow \textbf{table}(s))\ F'\\
\quad and\ \mathcal{S}[\![F']\!] = (\textsf{SELECT}\ \overline{e\ \textsf{AS}\ l}\ \textsf{FROM}\ \overline{t\ \textsf{AS}\ y}\ \textsf{WHERE}\ B)\\
\\
\mathcal{S}[\![\textbf{where}\ B\ Z]\!] = \textsf{SELECT}\ \overline{e\ \textsf{AS}\ l}\ \textsf{FROM}\ \bar{t}\ \textsf{WHERE}\ B' \wedge \mathcal{S}[\![B]\!]\\
\quad where\ \mathcal{S}[\![Z]\!] = (\textsf{SELECT}\ \overline{e\ \textsf{AS}\ l}\ \textsf{FROM}\ \bar{t}\ \textsf{WHERE}\ B')\\
\\
\mathcal{S}[\![\textbf{table}(s)]\!] = \textsf{SELECT}\ \overline{s.l\ \textsf{AS}\ l}\ \textsf{FROM}\ s\ \textsf{WHERE TRUE}\\
\mathcal{S}[\![\textbf{yield}\ R]\!] = \textsf{SELECT}\ \mathcal{S}[\![R]\!]\ \textsf{FROM}\ \phi\ \textsf{WHERE TRUE}\\
\mathcal{S}[\![\langle \overline{l = B}\rangle]\!] = \overline{\mathcal{S}[\![B]\!]\ \textsf{AS}\ l}\\
\mathcal{S}[\![\textbf{exists}\ U]\!] = \textsf{EXISTS}(\mathcal{S}[\![U]\!])\\
\mathcal{S}[\![\oplus(\overline{B})]\!] = \oplus_{sql}(\overline{\mathcal{S}[\![B]\!]})\\
\mathcal{S}[\![x.l]\!] = x.l\\
\mathcal{S}[\![x]\!] = x.*\\
\mathcal{S}[\![c]\!] = c
\end{array}
$$

Figure 9. SQL Translation

Stage 1:

$$
\begin{array}{rcll}
(\lambda x.\, N)\ M & \leadsto & N[x := M] & (\textsc{Abs-}\beta)\\
\langle \overline{l = M}\rangle.l_i & \leadsto & M_i & (\textsc{Record-}\beta)\\
\textbf{for}\ (x \leftarrow \textbf{yield}\ M)\ N & \leadsto & N[x := M] & (\textsc{ForYield})\\
\textbf{for}\ (x \leftarrow \textbf{for}\ (y \leftarrow L)\ M)\ N & \leadsto & & \\
\multicolumn{3}{r}{\textbf{for}\ (y \leftarrow L)\ (\textbf{for}\ (x \leftarrow M)\ N)} & (\textsc{ForFor})\\
\multicolumn{3}{r}{\text{if } y \notin FV(N)} & \\
\textbf{for}\ (x \leftarrow \textbf{where}\ L\ M)\ N & \leadsto & & \\
\multicolumn{3}{r}{\textbf{where}\ L\ (\textbf{for}\ (x \leftarrow M)\ N)} & (\textsc{ForWhere}_1)\\
\textbf{for}\ (x \leftarrow [\,])\ N & \leadsto & [\,] & (\textsc{ForEmpty}_1)\\
\textbf{for}\ (x \leftarrow L \uplus M)\ N & \leadsto & & \\
\multicolumn{3}{r}{\textbf{for}\ (x \leftarrow L)\ N \uplus \textbf{for}\ (x \leftarrow M)\ N} & (\textsc{ForUnionAll}_1)\\
\textbf{where true}\ M & \leadsto & M & (\textsc{WhereTrue})\\
\textbf{where false}\ M & \leadsto & [\,] & (\textsc{WhereFlase})
\end{array}
$$

Stage 2:

$$
\begin{array}{rcll}
\textbf{for}\ (x \leftarrow L)\ (M \uplus N) & \hookrightarrow & & \\
\multicolumn{3}{l}{\textbf{for}\ (x \leftarrow L)\ M \uplus \textbf{for}\ (x \leftarrow L)\ N} & (\textsc{ForUnionAll}_2)\\
\textbf{for}\ (x \leftarrow M)\ [\,] & \hookrightarrow & [\,] & (\textsc{ForEmpty}_2)\\
\textbf{where}\ L\ (M \uplus N) & \hookrightarrow & & \\
\multicolumn{3}{l}{(\textbf{where}\ L\ M) \uplus (\textbf{where}\ L\ N)} & (\textsc{WhereUnion})\\
\textbf{where}\ L\ [\,] & \hookrightarrow & [\,] & (\textsc{WhereEmpty})\\
\textbf{where}\ L\ (\textbf{where}\ M\ N) & \hookrightarrow & \textbf{where}\ (L \wedge M)\ N & (\textsc{WhereWhere})\\
\textbf{where}\ L\ (\textbf{for}\ (x \leftarrow M)\ N) & \hookrightarrow & & \\
\multicolumn{3}{l}{\textbf{for}\ (x \leftarrow M)\ (\textbf{where}\ L\ N)} & (\textsc{WhereFor})
\end{array}
$$

Figure 10. Normalization rules

E. Normalization rules

Fig. 10 lists the normalization rules by Cheney et al.'s T-LINQ. We have implemented all these rules as typed final program transformations.