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 program-mer. 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 compositibility comes at the expense of efficiency. We illustrate the hard trade-off between efficiency and compositibility/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 aggrega-
tion (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 fac-
tum.

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 opti-
nization 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 con-
struction

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') ap-
proach (TFA) [3] to embed and optimize a query language. That
approach proved convenient for embedding domain-specific lan-
guages (DSL): we can express not only the syntax and the de-
notational 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 implement-
ing 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 op-
timization 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 technol-
ogy, 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 technol-
ogy 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 lan-
guages and still be able to reuse old transformations as they are.

This paper is organized as follows: the background §2 intro-
duces the running example of sample queries and uses it to infor-
mally introduce our embedded query language, called QUEA, a su-
perset of T-LINQ of [4]. The section shows how the naive inter-
pretation of the language to query a SQL database leads to severe per-
formance problems, and how program transformation (some form
of meta-programming) may help. §3 formally introduces QUEA,
and its embedding to the host language OCaml in TFA. We briefly
illustrate the normalization (transformation) rules of QUEA pro-
grams in §4. §5 evaluates the performance. §6 shows off the exten-
sibility of our QUEA 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/quea/.

2. QUEA by Example

This section introduces our query language QUEA, 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 transforma-
tions may solve the performance problem. §4 will explain how to
actually program these transformations and be sure of their safety.

The language QUEA is a similar 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
very simple and self-explanatory examples. The examples use the
database in Fig. 1 with two tables: the products table is a bag (a mul-
siset) 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).

![Table 1: Sample database tables and the types of their records](image)

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.

\[
Q_1 = \lambda o. \quad \begin{array}{l}
\text{SELECT } o.p \text{ FROM orders AS o WHERE o.oid = oid}
\end{array}
\]

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

\[
\{(\text{oid} = 2, \text{pid} = 5, \text{qty} = 10), (\text{oid} = 2, \text{pid} = 6, \text{qty} = 20)\}
\]

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.

\[
Q_2 = \lambda o. \quad \begin{array}{l}
\text{SELECT p.pid AS pid, p.name AS name, p.price * o.qty AS sale}
\end{array}
\]

Then \( run \left( Q_2, (\text{oid} = 2, \text{pid} = 5, \text{qty} = 10) \right) \) returns

\[
\{(\text{pid} = 5, \text{name} = \text{HDD}, \text{sale} = 1000)\}
\]

We wish to compose \( Q_1 \) and \( Q_2 \) to obtain the query \( Q'_1 \), yielding
the bag of records for the given oid:

\[
Q'_1 = \lambda o. \quad \begin{array}{l}
\text{SELECT p.pid AS pid, p.name AS name, p.price * o.qty AS sale}
\end{array}
\]
Although very natural, \( Q_3 \)'s has a serious performance problem, known as Query Avalanche, or the \( N + 1 \) query problem [10]; running \( Q_3 \)'s 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, \( Q_3 \)'s 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

\[
\text{compose} = \lambda q. \lambda r. \lambda x. \text{for } (y \leftarrow q\ x) \ r y
\]

Then \( Q_3 \)'s could be interpreted in SQL as

\[
\text{SELECT} \ (Q_2 \ T_y \text{oid} \ Ty \text{pid} Ty \text{qty}) \text{ FROM} \ Q_1 \ \text{oid} \ AS Ty
\]

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 Opaley [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. 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.” 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

\[
Q_3'' = \lambda \text{oid}. \text{for } (y \leftarrow \ (o \leftarrow \text{orders})
\]

\[
\text{where} \ (o.\text{oid} = \text{oid}) \ \text{yield} \ (p.\text{pid} = o.\text{pid}, \text{qty} = o.\text{qty}))
\]

\[
\text{for } (p \leftarrow \text{products}) \text{ where } (p.\text{pid} = y.\text{pid})
\]

\[
\text{yield} \ (p.\text{pid} = p.\text{pid}, \text{name} = p.\text{name}, \text{sale} = p.\text{price} \ast y.\text{qty})
\]

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

\[
Q_3''' = \lambda \text{oid}. \text{for } (o \leftarrow \text{orders})
\]

\[
\text{where} \ (o.\text{oid} = \text{oid}) \ \text{and} \ (p.\text{pid} = o.\text{pid})
\]

\[
\text{yield} \ (p.\text{pid} = p.\text{pid}, \text{name} = p.\text{name}, \text{sale} = p.\text{price} \ast o.\text{qty})
\]

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):

\[
\text{SELECT } p.\text{pid} \text{ AS pid}, p.\text{name} \text{ AS name}, p.\text{price} \ast o.\text{qty} \text{ AS sale}
\]

\[
\text{FROM } \text{products AS p, orders AS o WHERE } p.\text{pid} = o.\text{pid} \text{ AND o.oid = oid}
\]

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 QUEA in more detail as well as its embedding as DSL in OCaml.

3. Language-Integrated Query

This section formally introduces the relational query language QUEA and embeds it in OCaml in the typed-final style. QUEA 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 QUEA, Its Syntax and Semantics

The following figure gives types and terms in QUEA. We use metavariables \( x, y \) for variables, \( c \) for constants, \( t \) for table names in the database, \( f \) for record labels, \( \odot \) for primitive operators (such as arithmetic, comparison, and also \texttt{exists}.) The sequence \( M_1, \ldots, M_n \) is abbreviated as \( M \). Types of \( Q \) are base types, function types, bag types (for multisets), and record types \( \langle f_1; \ldots; f_n \rangle \) where \( f_1, \ldots, f_n \) are field labels. We say a record type \( \langle f_1; \ldots; f_n \rangle \) is flat if all \( f_i \) are base types, and similarly for a bag type \( \langle f_1: M \rangle \).

<table>
<thead>
<tr>
<th>Base type</th>
<th>( O )</th>
<th>( A, B )</th>
<th>( Ty )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( O )</td>
<td>( A \rightarrow B )</td>
<td>( Bag )</td>
</tr>
<tr>
<td>Term</td>
<td>( L, M, N )</td>
<td>( e )</td>
<td>( L )</td>
</tr>
<tr>
<td></td>
<td>( \langle f_1; \ldots; f_n \rangle )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
</tbody>
</table>

QUEA 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; \( \text{table}(t) \) 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 \( \text{table}(t) \) has a flat bag type. We can still create and use non-flat records in QUEA, 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 QUEA-term of flat bag type into a term which does not use non-flat records and bags.

The operational semantics of QUEA 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 \text{yield} \ V_n \uplus \[ \] \) for \( n \geq 0 \). We assume that \( \uplus \) is associative and commutative.

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

\[
\text{table}(t) \rightarrow \Omega(t)
\]

\[
\text{where true } M \rightarrow M \quad \text{for false } M \rightarrow []\n\]

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 \( \Sigma \) maps constants to base types, primitive operators to functions on base types, and table names to flat bag types. We assume that \( \Omega \) and \( \delta \) are consistent with \( \Sigma \). We write \( x_1 : B_1, \ldots, x_n : B_n \vdash M : A \) if we can derive \( M : A \) under the assumptions \( x_1 : B_1, \ldots, x_n : B_n \).

<table>
<thead>
<tr>
<th>SINGLETON</th>
<th>( M : A )</th>
<th>( M : Bag A )</th>
</tr>
</thead>
</table>
| yield \( M : Bag A \) | \( \text{for } (x \leftarrow M) \ N : Bag B \)

<table>
<thead>
<tr>
<th>WHERE</th>
<th>( L : \text{Bool} )</th>
<th>( M : Bag A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L ) : Bag A</td>
<td>( where L : \text{Bag A} )</td>
<td></td>
</tr>
</tbody>
</table>
**Theorem 1** (Subject reduction). If $\Gamma \vdash M : A$ and $M \rightarrow^* N$, then $\Gamma \vdash N : A$.

The proof is not difficult but tedious, as we need a lemma: if $\Gamma_1 \vdash V : B$ and $\Gamma_1, \Gamma_2, x : x : B \vdash M : A$ 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 QUEA

This section describes the embedding of QUEA 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 `func` constructs the representation of QUEA 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 QUEA from §3.1 is given below.

```ml
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
  val where: (unit -> α list repr) -> (α -> β list repr) -> β list repr
  val yield: α repr, α list repr -> α list repr
  val nil: unit -> α list repr
  val (%) : α list repr, α list repr -> α list repr
  val (%) : α list repr -> α list repr
  val (%) : α -> β repr, α expr -> α expr
  val (%) : α expr -> bool repr
  (* abbreviated *)
  type α obs (* observation *)
  val observe: (unit -> α repr) -> α obs
end
```

The interface represents not just the syntax of QUEA (in the form close to BNF), but also its type system. The type of OCaml expressions that represent QUEA terms, typically called `repr`, is indexed by the QUEA’s type. (We take Bag to be a synonym for list.) The type of `foreach` 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 `@`, `-%` 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 only the typed embedded-language terms. This property holds for our QUEA embedding, as it is easy to see. Ill-typed QUEA terms cannot be encoded: their representation will not type check in OCaml. As a bonus, we use the OCaml type inferencer to infer QUEA types.

Formally the embedding of QUEA is defined in Appendix C. Here we give a few examples: QUEA’s term \( (\lambda x. 3 + x) \) 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 QUEA term is represented as an OCaml functor. For example, the query \( Q_1 \) from our running example §2 is represented in OCaml as follows:

```ml
module Q1(S:Symantics) = struct open S
  let res = fun oid ->
    foreach (fun () -> table_orders)@@ fun o ->
      where ((o %, oid) =>' oid) @@ fun () ->
        yield o
  end
end
```

Records of QUEA 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 \( Q_2 \) then takes the form

```ml
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 QUEA 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

```ml
module Q1Nice(S:Symantics) = struct open S
  let res = fun oid -> fun () ->
    let o = table_orders_gen () in
      where ((o %, oid) =>' oid); o
  end
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 \( Q_3 \) is similar. For the composed query \( Q_3 \) (producing total sales for the given oid) we write

```ml
module Q3(S:Symantics) = struct open S
  module M1 = Q1(S)
  module M2 = Q2(S)
  let res = fun x ->
    func (fun () ->
      app M1.res x)@@
      fun y ->
        M2.res y
  end
end
```

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

```ml
module R = struct
  type α repr = α
  let bool b = if b then true else false
  let string s = s
  let int n = n
  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 () = [ ]
end
```

OCaml’s `@@`, like Haskell’s `$`, is the low-precedence infix operator for applications.

---

1 We use `foreach` instead of `for`, since the latter is reserved in OCaml.
A QUEA term of type $\alpha$ is represented by an OCaml term of the same type, and the evaluation of QUEA maps to the evaluation of OCaml. Since OCaml type system (at least of the subset used here) is sound, the R-embedding of QUEA is type-sound. Since the representation type $\text{repr}$ is kept abstract, and since the typed-final encoding is tight, it follows that QUEA itself has the property of subject reduction.

To run the query $Q_{1.2}$ on an in-memory non-SQL database with the needed tables, we evaluate

\begin{verbatim}
let module M = Q1_2(R) in
M.observe (fun () -> M.res)
\end{verbatim}

obtaining the result shown in §2. Exactly the same $Q_{1.2}$ can be run against an SQL database, as we show below. We have used the so far neglected function $\text{observe}$ to observe the $\text{repr}$ value as a value of some observation type $\alpha \text{obs}$, which is also kept abstract. The choice of $\text{obs}$ hence is also left for a Symantics implementation. In the $\text{observe}$ implementation it is the same as $\text{repr}$ 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 $\text{F}$, pretty-prints QUEA terms: using it with $\text{F.observe}$

\begin{verbatim}
let module M = Q1(F) in
print_endline (\@\$ M.observe (fun () -> M.res))
\end{verbatim}

prints out the query:

\begin{verbatim}
fun x -> foreach (fun () -> table "orders") (fun y ->
  where (y.oid = x) (fun () -> yield y))
\end{verbatim}

### 3.3 Normal QUEA Programs

There is another implementation of the Symantics interface, $\text{GenSQL}$, which however works only on normal QUEA programs that produce flat record bags. Normal programs are closed irreducible QUEA 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**
  - $F ::= \{ x \mapsto \text{table}(t) \} \mid F \mid Z$

- **Body**
  - $Z ::= \text{where} \ B \mid \text{yield} \ R \mid \text{table}(t)$

- **Record**
  - $R ::= \{(I = B) \mid x\}$

- **Primitives**
  - $B ::= \oplus (B) \mid x.\{l \mid c\}$

A normal program can be straightforwardly converted to the single SQL query without nested subsequeries. For reference, this transformation is described in Appendix D. For example, the following code that is similar to $Q_{1.2}$ produces a flat record bag and is in the normal form. Evaluating it with the $\text{GenSQL}$ implementation

\begin{verbatim}
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)
\end{verbatim}

first converts the program to SQL – the same SELECT query we saw in §2 for $Q_1$. The function $\text{observe}$ in the $\text{GenSQL}$ 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 $\text{GenSQL}.\text{obs}$, which is order list). On the other hand, the program $Q_2$ 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 $Q_2$ and $Q_3$ 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 $[] \uplus N \sim N$. 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 $f.\text{repr}$ be the un-annotated representation type. The annotated type then takes the form of the GADT

\begin{verbatim}
type \alpha \text{annrepr} =
| \text{Empty} : \alpha \text{list} \text{annrepr}
| \text{Unknown} : \alpha \text{F.}\text{repr} -> \alpha \text{annrepr}
\end{verbatim}

The variant $\text{Unknown}$ represents the value about which nothing is statically known. The function

\begin{verbatim}
let dyn : \alpha \text{annrepr} -> \alpha \text{F.}\text{repr} = function
| \text{Empty} -> \text{F.nil} ()
| \text{Unknown} x -> x
\end{verbatim}

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

\begin{verbatim}
module LNIL_preliminary = struct
  type \alpha \text{repr} = \alpha \text{annrepr}
  let int n = Unknown(F.int n)
  let bool n = Unknown(F.bool n)
  let app x y = Unknown(F.append (fun x -> dyn (f (Unknown x))))
  let lam f = Unknown(F.lam)
  ...  
  let nil () = Empty
  let (\@\%) x y = match x with
    | Empty -> y
    | Unknown x -> Unknown ((f.\@\%) x (dyn y))

  type \alpha obs = \alpha \text{F.obs}
  let observe x = F.observe (fun () -> dyn (x ()))
end
\end{verbatim}

As expected, $\text{nil}$ 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 \( \alpha \) \( \text{annrepr} \) 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 rule takes the following form

\[
\text{module } \text{LNil}(F; \text{Symantics}) = \text{struct}
\]

\[
\text{type } \alpha \text{ annrepr} = \\
| \text{Empty} : \alpha \text{ list annrepr} \\
| \text{Unknown} : \alpha \text{ F.repr} \to \alpha \text{ annrepr} \\
\text{type } \alpha \text{ repr} = \alpha \text{ annrepr} \\
\text{... (as before *)}
\]

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 \( Q_1 \) and run the result, we do

\[
\text{let module } M = Q_1_2(\text{LNil}(\text{GenSQL})) \text{ in} \\
M.\text{observe}(\text{fun} () \to M.\text{res})
\]

One can read this code as doing the transformation \( \text{LNil} \) and interpreting the result with GenSQL. One can also read it as interpreting the original \( Q_1_2 \) term using the transformed interpreter \( \text{LNil}(\text{GenSQL}) \); the inner \( \text{GenSQL} \) will never see the terms of the form \( [\text{Nil}(\text{term})] \) 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 \( \text{LNil} \) 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, suppression rule had to produce the complete implementation of transformations on their interpreters.

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 \( \alpha \) \( \text{annrepr} \) 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 rule takes the following form

\[
\text{module } \text{LNil}(F; \text{Symantics}) = \text{struct}
\]

\[
\text{type } \alpha \text{ annrepr} = \\
| \text{Empty} : \alpha \text{ list annrepr} \\
| \text{Unknown} : \alpha \text{ F.repr} \to \alpha \text{ annrepr} \\
\text{type } \alpha \text{ repr} = \alpha \text{ annrepr} \\
\text{... (as before *)}
\]

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 \( Q_1 \) and run the result, we do

\[
\text{let module } M = Q_1_2(\text{LNil}(\text{GenSQL})) \text{ in} \\
M.\text{observe}(\text{fun} () \to M.\text{res})
\]

One can read this code as doing the transformation \( \text{LNil} \) and interpreting the result with GenSQL. One can also read it as interpreting the original \( Q_1 \) term using the transformed interpreter \( \text{LNil}(\text{GenSQL}) \); the inner \( \text{GenSQL} \) will never see the terms of the form \( [\text{Nil}(\text{term})] \) 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 \( \text{LNil} \) 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 QUEA expressions in the \( \text{annrepr} \) domain.

The ForFor optimization is implemented quite literally, in a couple 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

\[
\text{module } \text{MainPasses}(S; \text{Symantics}) = \text{AbsBeta}( \\
\quad \text{RecordBeta}(\text{ForFor}(\text{ForWhere}(\text{ForYield}(\text{ForFor}(\text{ForEmpty1}))))))
\]

Interpreting our running example \( Q_3 \) from \$2 \& \$3 using \( \text{MainPasses}(\text{GenSQL}) \) 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 QUEA 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 QUEA into Ocaml. The performance has several components: optimizing the user-entered QUEA 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 \( \text{MainPasses} \) in \$4 and the second is \( \text{AllPasses} \) defined as follows.

\[
\text{module } \text{AllPasses}(S; \text{Symantics}) = \text{AbsBeta}( \\
\quad \text{RecordBeta}(\text{ForYield}(\text{ForFor}(\text{ForEmpty1}))))
\]

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 transformations 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
can re-use, rather than re-write, all previously written interpreters. This section
shows off the extensibility, by adding one small extension.

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 QUEA 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 QUEA is as simple as it can get.

First we extend the syntax of QUEA with the new set-based union operation, to be called (∪):

```plaintext
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 old.

Next we have to extend the interpreters of QUEA, 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:

```plaintext
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 QUEA exactly as they were, with no changes. We do need to add new passes that concern the set-based union, for example

- for \( (x \leftarrow L \cup M) \) \( N \) \( \mapsto \)
- for \( (x \leftarrow L) \) \( N \cup \) 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 QUEA 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 we 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 QUEA as follows. The corresponding SQL query is shown on the right:

```plaintext
Q4 =
for (o ← orders)
for (p ← products)
where (o.pid = p.pid)
∧ p.price > 200)
group (gprice ← p.price)
having (sum(p.price * oqty) > 500)
gyield (gprice, sum (p.price * oqty))
```

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 total sales 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 | \( L, M, N ::= \cdots \mid \text{group } (q ← M) \) \( N \) \mid \text{having } L M \mid \text{gyield } M \mid (M, N) \mid (\oplus (M)) |

Fig. 2. Syntax of QUEA. The metavariable \( \oplus \) ranges over aggregate functions such as \text{sum} and \text{average}.

To implement \( Q_4 \) we extend QUEA with several new primitives, obtaining QUEA\(_G\); 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

\[
[<\text{name}=\text{Tablet}, \text{price}=500, \text{qty}=5>,
<\text{name}=\text{Laptop}, \text{price}=1000, \text{qty}=5>, \ldots]
\]

The group statement then builds a bag of bags

\[
[<\text{gprice}=500, [<\text{name}=\text{Tablet}, \text{price}=500, \text{qty}=5>,
<\text{name}=\text{SSD}, \text{price}=500, \text{qty}=20>],
<\text{gprice}=1000, [<\text{name}=\text{Laptop}, \text{price}=1000, \text{qty}=5>,
<\text{name}=\text{Tablet}, \text{price}=1000, \text{qty}=50>]]
\]

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 \( \text{sum aggregates} \) 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.

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:

\[
gprice, \text{sum(o.qty)}, \text{sum(p.price)}, gprice \ast \text{sum(o.qty)}, \text{sum(p.price)} \ast \text{weightfactor(p.name)}
\]

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; \text{weightfactor} is a user-defined function.

The type system for \( QUEAG \) uses two new judgments: \( \vdash G M : A \) 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.

```haskell
defining type SymanticsG = sig
defining include Symantical
defining type (α, β, ‘key’) gres

defining type (α, β, ‘groupkeys’, ‘reskeys’) coll

defining val group : ‘gk gb_sequence →
(‘gk gb_key_sequence → (α, β, ‘res) gres) →
(α, β,’gk,’res) coll list repr

defining val gint : int → (int, int, constk) grepr

defining val gpair: (α, β, ‘k1) grepr → (β, β2, ‘k2) grepr →
(α + β β2, ‘k1+ k2) grepr

defining val having : (bool, ‘k1) grepr →
(unit → (α, β2, ‘k2) gres) →
(α, β1 β2, ‘k1+ k2) gres

defining val gyield : (α, β,’key) grepr → (α, β,’key) gres

end

The type (α, β,’key) grepr corresponds to the second judgment \( \vdash G M : A \) in Fig. 3, where we ignore the arguments \( β \) and ‘key which are used for final observation only. The function \( \text{gint} \) implements the typing rule \( \text{CONST} \) when \( O \) is int, and \( \text{sum} \) and \( \text{gpair} \), resp., implement the typing rules \( \text{AGGREGATION} \) (when \( \odot \) is \( \text{sum} \)) and \( \text{pair} \), resp. Similarly, the types (α, β,’key) gres

```

Figure 3. Typing rules of \( QUEAG \)

\[
\begin{align*}
\vdash G M : A & \quad \text{GROUP} \\
\vdash G M : A & \quad \vdash G M : A \quad \text{AGGREGATION} \\
\vdash \text{group} (g \leftarrow M) N : \text{Bag} B
\end{align*}
\]

```

and \( (α, β,’\text{groupkeys,’reskeys}) \) coll, resp., correspond to the judgments \( \vdash G M : α \) and \( \vdash M : α \) resp.

We can write the query \( Q_4 \) in \( QUEAG \) as follows.

```haskell
module Q4 {def SYM_SCHEMA} = struct

open S

let products = table("products", products ()))

let orders = table("orders", orders ()))

let res =
  foreach (fun () → orders) ΩΩ fun c →
  foreach (fun () → products) ΩΩ fun p →
  where (\{o qty \% p qty \% p qty \% \}
  p price >\% int 100) ΩΩ fun r →
  group (seq_one (p qty) ΩΩ seq_decon
  (fun gprice →
    having (sum (p qty) price >\% o qty)
    +\$ gint 500) ΩΩ fun r →
    gyield (gpair gprice
      (sum (p qty) 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.
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# quotient 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, relying 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)]

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 QUEA, 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 (xs:x) | 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. 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 QUEA 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 [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 QUEA 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 typesclasses made the encoding lightweight compared to OCaml modules. On the other hand, in

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

6which 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 micro-management 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 QEUA’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 QUA

We define operational semantics in Fig. 4 and 5, and typing rules in Fig. 6. Recall that \([V_1, \ldots, V_n]\) is an abbreviated expression for yield \(V_1 \sqcup \cdots \sqcup V_n\) yield \(V_n \sqcup [\cdot]\).

\[
\begin{align*}
V & ::= c | \lambda x.M | (\ell = \mathcal{V}) | [V_1, \cdots, V_n] \\
\mathcal{E} & ::= [] | \oplus(\mathcal{V}), \mathcal{E}, M | \mathcal{E} M | \mathcal{V} \mathcal{E} \\
& \quad | \mathcal{E} \sqcup M | \mathcal{V} \sqcup \mathcal{E} | \text{for } (x \leftarrow \mathcal{E}) N \\
& \quad | \text{where } \mathcal{E} M
\end{align*}
\]

**Figure 4. Values and Evaluation Contexts**

\[
\begin{align*}
\oplus(\mathcal{V}) & \rightarrow \delta(\oplus)(\mathcal{V}) \\
table(t) & \rightarrow \Omega(t) \\
(\lambda x.M) V & \rightarrow M[x := V] \\
(\ell = \mathcal{V}), t_i & \rightarrow V_i \\
\text{where } true M & \rightarrow M \\
\text{where } false M & \rightarrow [\cdot] \\
\text{for } (x \leftarrow \text{yield } \mathcal{V}) M & \rightarrow M[x := V] \\
\text{for } (x \leftarrow [\cdot]) M & \rightarrow [\cdot] \\
\text{for } (x \leftarrow L \sqcup M) N & \rightarrow \\
\quad (\text{for } (x \leftarrow L) N) \sqcup (\text{for } (x \leftarrow M) N) \\
M & \rightarrow N \\
\mathcal{E}[M] & \rightarrow \mathcal{E}[N] \quad \text{(E-Context)}
\end{align*}
\]

**Figure 5. Operational Semantics of QUA**

The map \(\delta\) gives semantics to each primitive operator, and \(\Omega\) maps each table name to a value of a flat bag type.

We assume that \(\Omega\) and \(\delta\) in the previous subsection are consistent with \(\Sigma\); for each table \(\Omega(t)\) is assumed to be a value of type \(\Sigma(t)\), and \(\delta\) respect types: if \(\Sigma(\oplus) = O\) and \(\vdash \mathcal{V} : O\) and \(V = \delta(\oplus, \mathcal{V})\) 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 Syntactics with the schema as follows.

```ocaml
module type SCHEMA = sig
  type α repr
(* record constructors *)
val product : int repr → string repr → <pid:int; name:string; price:int> repr...
(* projection *)
val name : <pid:int; name:string; price:int> repr → int repr
val price : <pid:int; name:string; price:int> repr → string repr
end
```

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

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.

\(<\ell : \mathcal{A}\rangle\) is the type for objects in OCaml. The function \(\text{const}_O\) maps constants in the object language to OCaml constants of type

```ocaml
module type SYM_SCHEMA = sig
  include Symntactical.
  include SCHEMA with type α repr := α repr
end
```

In this paper, we implicitly assumed that we use this extended signature as a Syntactics. Interpreters are similarly extended.
\[ M[A] = A^* \text{ repr} \]
\[ \text{Int}^* = \text{int} \]
\[ \text{Bool}^* = \text{bool} \]
\[ \text{String}^* = \text{string} \]
\[ (A \to B)^* = A^* \to B^* \]
\[ (\text{Bag} A)^* = A^* \text{ list} \]
\[ ((T, A))^* = \langle T, A^* \rangle \]
\[ M[x] = x \]
\[ M[c] = \text{const}_O(c) \]
\[ M[(M)] = @M[M[M]] \]
\[ M[L M] = \text{app } M[L] M[M] \]
\[ M[\lambda x. N] = \text{lam } \langle \text{fun } x \to M[N] \rangle \]
\[ M[(L = M)] = \text{record}_O(M[M]) \]
\[ M[L.1] = M[L] \% \lambda \]
\[ M[\text{for } (x \leftarrow M) N] = \text{foreach } \langle \text{fun } \langle \rangle \to M[M] \rangle \]
\[ M[\text{where } L M] = \text{where } \langle M[L] \rangle \]
\[ M[\text{yield } M] = \text{yield } M[M] \]
\[ M[\text{null }] = \text{nil } () \]
\[ M[M \uplus N] = M[M] \% M[N] \]
\[ M[\text{table } (t, 0)] = \text{table } (t, 0()) \]

\[ S[U_1 \cup U_2] = S[U_1] \cup \text{ALL } S[U_2] \]
\[ S[\text{null }] = \text{SELECT } \text{null } \text{AS } x, y \text{ FROM } \phi \]
\[ \text{WHERE } \text{FALSE} \]
\[ S[F] = \text{SELECT } e \text{ AS } x, y \text{ FROM } s \text{ AS } x, y \text{ WHERE } B \]
\[ \text{WHERE } F = \text{for } (x \leftarrow \text{table } (a)) F' \]
\[ \text{where } F = \text{for } (x \leftarrow \text{table } (a)) F' \]
\[ \text{and } S[F'] = \text{SELECT } e \text{ AS } x, y \text{ FROM } t \text{ AS } x, y \text{ WHERE } B \]
\[ S[\text{where } B Z] = \text{SELECT } e \text{ AS } x \text{ FROM } \tilde{\tau} \text{ WHERE } B' \]
\[ \text{WHERE } B' = B' \land S[B] \]
\[ \text{where } S[Z] = \text{SELECT } e \text{ AS } x \text{ FROM } \tilde{\tau} \text{ WHERE } B' \]
\[ S[\text{table } (s)] = \text{SELECT } s.l \text{ AS } l \text{ FROM } s \text{ WHERE } \text{TRUE} \]
\[ S[\text{yield } R] = \text{SELECT } S[R] \text{ FROM } \phi \text{ WHERE } \text{TRUE} \]
\[ S[(l = B)] = S[B] \text{ AS } l \]
\[ S[\text{exists } U] = \text{EXISTS } S[U] \]
\[ S[B] = S_\text{sql } S[B] \]
\[ S[x.l] = x.l \]
\[ S[x] = x.* \]
\[ S[c] = c \]

\[ \ell_i = M_i \]
\[ (\lambda x. N) M \leadsto N[x := M] \]
\[ (\ell = M_i) \lambda x. N \leadsto M_i[x := M] \]
\[ (\ell_i = M_i) \lambda x. N \leadsto \ell_i[x := M] \]
\[ \text{FORTELL} \]
\[ \text{FORFOR} \]
\[ \text{WHEREUNION} \]
\[ \text{WHEREWHERE} \]
\[ \text{WHERETRUE} \]
\[ \text{WHEREFALSE} \]

\begin{figure}[h]
\centering
\include{figure8}
\caption{Embedding Quea into OCaml}
\end{figure}

\section*{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 \( \phi \) denotes an empty database table. We list it here for completeness.

\begin{figure}[h]
\centering
\include{figure9}
\caption{SQL Translation}
\end{figure}

\section*{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.