A Practical Theory of Language-integrated Query

A Practical Theory of Language-Integrated Query

James Cheney
The University of Edinburgh
jcheney@inf.ed.ac.uk

Sam Lindley
University of Strathclyde
Sam.Lindley@ed.ac.uk

Philip Wadler
The University of Edinburgh
wadler@inf.ed.ac.uk

Abstract

Language-integrated query is receiving renewed attention, in part because of its support through Microsoft's LINQ framework. We present a practical theory of language-integrated query based on quotation and normalisation of quoted terms. Our technique supports join queries, abstraction over values and predicates, composition of queries, dynamic generation of queries, and queries with nested intermediate data. Higher-order features prove useful even for constructing first-order queries. We prove a theorem characterising when a host query is guaranteed to generate a single SQL query. We present experimental results confirming our technique works, even in situations where Microsoft's LINQ framework either fails to produce an SQL query or, in one case, produces an avalanche of SQL queries.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.1 [*Formal Definitions and Theory*]; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]; H.2.3 [*Languages*]: Query languages

Keywords lambda calculus; LINQ; F#; SQL; quotation; anti-quotation

1. Introduction

What is the difference between theory and practice?
In theory there is no difference, but in practice there is.[1]

A quarter-century ago, Copeland and Maier (1984) decried the "impedance mismatch" between database and conventional programming models, and Atkinson and Buneman (1987) spoke of "The need for a *uniform* language" (their emphasis), and observed that "Databases and programming languages have developed almost independently of one another for the past twenty years." Smooth integration of database queries with programming languages, also known as *language-integrated query*, remains an open problem. Language-integrated query is receiving renewed attention, in part because of its support through Microsoft's LINQ framework (Meijer et al. 2006; Syme 2006).

[1] Attributed variously to Yogi Berra, William T. Harbaugh, Karl Marx, Chuck Reid, and Jan L. A. van de Snepscheut.

ICFP '13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500586

The problem is simple: two languages are more than twice as difficult to use as one language. The host and query languages often use different notations for the same thing, and convenient abstractions such as higher-order functions and nesting may not be available in the query language. Interfacing between the two adds to the mental burden on the programmer and leads to complex code, bugs, and security holes such as SQL injection attacks. Wrapper libraries, such as JDBC, provide raw access to high-performance SQL, but the resulting code is difficult to maintain. Object-relational mapping frameworks, such as Hibernate, provide an object-oriented view of the data that makes code easier to maintain but sacrifices performance (Goldschmidt et al. 2008); and workarounds to recover performance, such as framework-specific query languages, reintroduce the drawbacks of the two-language approach.

We present a practical theory of language-integrated query, consisting of a theoretical model, T-LINQ, and a practical implementation, P-LINQ. Our approach is based on quotation and normalisation of quoted terms. Our approach can be used in off-the-shelf languages, such as F#, fits with existing frameworks, such as Microsoft LINQ, and does not require specialised type systems of the kind found in Ur (Chlipala 2010) or our own work on Links (Cooper et al. 2007; Cooper 2009). In this paper we focus on SQL queries, although we believe the techniques adapt to other targets, such as XQuery, and may even be applicable to DSLs for domains other than query.

Microsoft LINQ was released as a part of .NET Framework 3.5 in November 2007, and LINQ continues to evolve with new releases. LINQ operators construct queries in a host language, and LINQ providers translate these into a target language, such as SQL, XQuery, or GPU code. In this paper we focus on Microsoft's LINQ to SQL provider, which targets Microsoft SQL Server. There are variants of LINQ for C#, Visual Basic, and F#, among others. T-LINQ and P-LINQ correspond to LINQ for F#. Similar techniques may apply to C#, though it requires some tricks to support anti-quotation (Petricek 2007b). A few details of F# 3.0 hinder smooth integration with P-LINQ, and we are working with Microsoft to enable its wider use.

Our formal theory T-LINQ is similar in scope, power, and translation technique to NRC (Buneman et al. 1994; Wong 1996) and to our own previous work on Links (Cooper et al. 2007; Cooper 2009). NRC is a special-purpose, first-order query language, and does not address integration into a general-purpose language. Links supports higher-order queries and addresses integration, but requires a special-purpose type-and-effect system. Our achievement here is to transpose most of the benefits of Links for language-integrated querying to a widely-used practical language.

We had previously argued that Links offered significant benefits over LINQ. A precise comparison was difficult because LINQ, like many practical systems, is large and not precisely defined. To explore the relationship between Links and LINQ, we formulated a core calculus capturing essential features of LINQ. In the course of formulating T-LINQ, we realised that many of the benefits of

Links can in fact be achieved in LINQ, and we are pleased to offer P-LINQ as a widely-available alternative.

Through a series of examples, we show how our technique supports join queries, abstraction over values and predicates, composition of queries, dynamic generation of queries, and queries containing intermediate nested data. We believe our series of examples distills an important set of features, and may be of independent interest. Among other things, these examples demonstrate the value of higher-order functions and nesting in queries, even when the target query language, in this case SQL, is first-order and does not support nested data.

We require that each query in the host language generate exactly one SQL query. Alluding to twin perils Odysseus sought to skirt when navigating the straits of Medina, we seek to avoid *Scylla and Charybdis*. Scylla stands for the case where the system fails to generate a query, signalling an error. Charybdis stands for the case where the system generates multiple queries, hindering efficiency. The overhead of accessing a database is high, and to a first approximation cost is proportional to the number of queries. We particularly want to avoid a query avalanche, in the sense of Grust et al. (2010), where a single host query generates a number of SQL queries proportional to the size of the data.

Current LINQ implementations may founder on either of these perils. Scylla: many of our examples fail under either F# 2.0 or F# 3.0, and different examples fail in the two different systems. Charybdis: one of our examples causes a query avalanche in F# 3.0. In Grust et al. (2010) avalanches are caused by queries with nested results; interestingly, the avalanche is caused by a query with flat results, but with nested intermediate structure. The documentation for F# LINQ does not explain when host queries may fail or when multiple SQL queries may be generated; trial and error is required to determine what does or does not work.

Our work avoids these perils. For T-LINQ, we prove the Scylla and Charybdis theorem, characterising when a host query is guaranteed to generate a single SQL query. All our examples are easily seen to satisfy the characterisation in the theorem, and indeed our theory yields the same SQL query for each that one would write by hand. For P-LINQ, we verify that its run time on our examples is comparable to that of F# 2.0 and F# 3.0, in the cases where those systems generate a query, and significantly faster in the one case where F# 3.0 generates an avalanche—indeed, arbitrarily faster as the size of the data grows.

Our theory models only a fraction of the features of LINQ, but our practice applies to its entirety. T-LINQ supports queries formed from comprehensions, union, and existence tests, but excludes sorting, grouping, and aggregation; extending it to these other features is important future work. However, P-LINQ supports all features of LINQ, and we validate its coverage and performance, showing it has comparable runtime to F# 3.0 for all 62 database queries listed in the F# 3.0 documentation.

The language features we require are lambda abstraction, records and lists, comprehension notation, and quotation and anti-quotation. Quotation and anti-quotation first appear in Lisp, and are also found in Scheme, Racket, Haskell (via Template Haskell), OCaml (via Camlp4), and F#. Quotation comes in two forms, *closed* and *open*; in the former, quoted terms must contain no free quoted variables. All languages with quotation support closed quotation, while only some support open quotation; in particular, F# only supports closed quotation. Our approach requires only closed quotation.

We observe that one should abstract when possible in the quoted language rather than the host language, as this enables composition while restricting to closed quotation. Though it seems obvious in retrospect, that one should abstract when possible in the quoted language came as a surprise to us; previously, we had assumed one should abstract when possible in the host language, as is done when using open quotation.

people

name	age
"Alex"	60
"Bert"	55
"Cora"	33
"Drew"	31
"Edna"	21
"Fred"	60

couples

her	him
"Alex"	"Bert"
"Cora"	"Drew"
"Edna"	"Fred"

Figure 1. People as a Database

Many embedded domain specific languages (EDSLs) build expression trees as constructs of the host language. For instance, in Nikola (Mainland and Morrisett 2010) or Feldspar (Axelsson et al. 2010; Axelsson and Svenningsson 2012), addition is overloaded so that $a + b$ denotes either an integer (of type **int**) or a quotation of an expression that yields an integer (of type Expr< **int** >), depending on whether a and b denote either the former or latter. For technical reasons, overloading does not apply to all operations, so users have the minor inconvenience of writing $a < b$ as $a \;.<.\; b$ and **if** a **then** b **else** c as $a \;?\; (b, c)$. In our case, quotation avoids this inconvenience, allowing terms in the embedded language to use identical syntax to the host language. It is common for EDSLs to reuse the type system of the host language as the type system of the embedded language—here we have, in effect, extended the same idea to syntax.

The contributions of this paper are:

- We introduce our technique, and through a series of examples show how it supports join queries, abstraction over values and predicates, composition of queries, dynamic generation of queries, and nesting of intermediate data. We also present a larger example of translation from XPath to SQL, which illustrates the power resulting from combining higher-order queries with language integration. (Sections 2, 3, and 4).
- We develop our theoretical model, T-LINQ, and prove standard results, including a theorem that normalisation always succeeds in translating any T-LINQ query to SQL. (Section 5.)
- We observe that one should abstract when possible in the quoted language rather than the host language, in order to support composition in the presence of closed quotation, and we show how closed quotation can simulate open quotation. (Section 6 and 7.)
- We describe our practical implementation, P-LINQ, and present experimental results confirming our technique works in practice. We observe that F# 2.0 or F# 3.0 fail on key examples where we succeed; that when F# 2.0 or F# 3.0 succeed our performance is comparable; that when F# 3.0 avalanches our performance is arbitrarily better as the problem size grows; and that we handle with comparable performance all 62 database query examples in the F# 3.0 documentation. (Sections 8 and 9.)
- We discuss related work, and summarise our results in the form of a recipe that may apply to a wide variety of domain-specific languages. (Sections 10 and 11.)

Our implementation, examples, and experimental data are available online in the ACM Digital Library.

2. Fundamentals

We consider a simplified programming language, based loosely on F# (Syme et al. 2012), featuring records and list comprehensions.

```
{people =
   [{name = "Alex" ; age = 60};
    {name = "Bert" ; age = 55};
    {name = "Cora" ; age = 33};
    {name = "Drew"; age = 31};
    {name = "Edna" ; age = 21};
    {name = "Fred" ; age = 60}];
 couples =
   [{her = "Alex" ; him = "Bert" };
    {her = "Cora" ; him = "Drew"};
    {her = "Edna"; him = "Fred" }]}
```

Figure 2. People as Data

We review the relationship between comprehensions and database queries and then introduce the use of quotation to construct queries.

2.1 Comprehensions and Queries

For purposes of illustration, we consider a simple database containing two tables, shown in Figure 1. The first table, people, has columns for name and age, and the second table, couples, has columns for her and him. (Schema update will be required once equal marriage legislation passes in Scotland.) Here is an SQL query that finds the name of every woman that is older than her mate, paired with the difference in ages.

```
select w.name as name, w.age − m.age as diff
from couples as c, people as w, people as m
where c.her = w.name and c.him = m.name and
      w.age > m.age
```

It returns the following table:

name	diff
"Alex"	5
"Cora"	2

Assuming the people table is indexed with name as a key, this query can be answered in time $O(|\text{couples}|)$.

The database is represented in T-LINQ as a record of tables, where each table is represented as a list of rows, and each row is represented as a record of scalars.

```
type DB = {people :  {name : string; age : int} list;
           couples : {her : string; him : string} list}
```

Following F#, we use lists to represent tables, and will not consider the order of their elements as significant. We follow the notational conventions of F#, writing lists in square brackets and records in curly braces.

Our language includes a construct that takes the name of the database and returns its content as a data structure.

```
let db′ : DB = database("People")
```

If "People" is the name of the database in Figure 1, then db′ is bound to the value shown in Figure 2. We stick a prime on the name to warn that this is too naive: typically, the database will be too large to read into main memory. We consider a feasible alternative in the next section.

Many programming languages provide a comprehension notation offering operations over lists analogous to those provided by SQL over tables (Trinder and Wadler 1989; Buneman et al. 1994). F# also supports a form of comprehension notation called *computation expressions* (Petricek and Syme 2012). T-LINQ follows this approach, so the previous query can be written as follows.

```
let differences′ : {name : string; diff : int} list =
  for c in db′.couples do
  for w in db′.people do
  for m in db′.people do
  if c.her = w.name && c.him = m.name &&
     w.age > m.age then
  yield {name : w.name; diff : w.age − m.age}
```

Evaluating differences′ returns the value

```
[{name = "Alex"; diff = 5}; {name = "Cora"; diff = 2}]
```

which corresponds to the table returned by the previous SQL query. Again, we stick a prime on the name to warn that this technique is too naive: typically, in-memory evaluation of a comprehension does not take advantage of indexing, and so requires time $O(|\text{couples}| \cdot |\text{people}|^2)$. We consider a feasible alternative in the next section.

Here we use three constructs, similar to those supported in the sequence expressions of F#. The term **for** x **in** M **do** N binds x to each value in the list M and computes the list N, concatenating the results; in mathematical notation, we write $\biguplus\{N \mid x \in M\}$; note that x is free in M but bound in N. The term **if** L **then** M evaluates boolean L and returns list M if it is true and the empty list otherwise. The term **yield** M returns a singleton list containing the value of M.

Many languages support similar notation, including Haskell, Python, C#, and F#. The T-LINQ term

for x **in** L **do for** y **in** M **do if** P **then yield** N

is equivalent to the mathematical notation

$$\{N \mid x \in L,\, y \in M,\, P\}$$

or the F# sequence expression

seq {**for** x **in** L **do for** y **in** M **do if** P **then yield** N}.

The last is identical to T-LINQ, save it is preceded by the keyword **seq** and surrounded by braces.

2.2 Query via Quotation

T-LINQ allows programmers to access databases using a notation nearly identical to the naive approach of the previous section, but generating efficient queries in SQL. The recipe for conversion is as follows. First, we wrap the reference to the database inside quotation brackets, `<@ ··· @>`.

```
let db : Expr< DB > = <@ database("People") @>
```

Next, we wrap the query inside quotation brackets, `<@ ··· @>`, and wrap occurrences of any externally bound variable, such as db, in anti-quotation brackets, `(%···)`.

```
let differences : Expr< {name : string; diff : int} list > =
  <@ for c in (%db).couples do
     for w in (%db).people do
     for m in (%db).people do
     if c.her = w.name && c.him = m.name &&
        w.age > m.age then
     yield {name : w.name; diff : w.age − m.age} @>
```

Finally, to get the answer we evaluate the term

run(differences) (1)

Evaluating (1) takes the quoted expression, normalises it, translates the normalised expression to SQL, evaluates the SQL query on the database, and imports the resulting table as a data structure. In this case, the quoted expression is already in normal form, and it translates into the SQL in the previous section, and so returns the table and answer seen previously. We drop the warning primes, because the answer is computed feasibly by access to the database.

The notation `<@ ··· @>` indicates quotation, which views an expression of type A as a data structure of type Expr< A > that represents the expression as an abstract syntax tree. The notation (% · · ·) indicates anti-quotation, which splices a value of type Expr< A > into a quoted expression at a point expecting a quoted term of type A. Database access, indicated by the keyword **database**, denotes the value of the database viewed as a record of tables, where each table is a list of rows, and each row is a record of scalars. Database access is only permitted within quotation, as its use outside quotation would require reading the entire database into main memory, which is in general infeasible. Query evaluation, indicated by the keyword **run**, takes an expression of type Expr< A >, normalises it, translates the normalised expression to SQL, evaluates the SQL query on the database, and imports the result as a data structure of type A.

Some restrictions are required on the abstract syntax tree in a **run** expression in order to ensure that it may be successfully translated to SQL. First, all database literals within a given query must refer to a single database on which the query is to be evaluated. Second, the return type must be a *flat relation type*, that is, a list of records with fields of scalar type. Third, the expression must not contain operations that cannot be converted to SQL, such as recursion. (Technically, SQL supports some forms of recursion, such as transitive closure, but current LINQ systems do not.) Fourth, we restrict our attention to queries built from sequence comprehensions, emptiness tests, and sequence union. In T-LINQ, these restrictions are statically checked (in the case of the first restriction, by only supporting one database); in P-LINQ, these restrictions are dynamically checked.

T-LINQ captures the essence of query processing in Microsoft LINQ, particularly as it is expressed in F#. However, the details of Microsoft LINQ are more complicated, involving three types Expression<A>, IEnumerable<A> and IQueryable<A> that play overlapping roles, together with implicit type-based coercions including a type-based approach to quotation in C# and Visual Basic, plus special additional query notations in C#, Visual Basic, and F# 3.0. We relate our model to the pragmatics of LINQ in Section 8.

2.3 Abstracting over Values

Our quoted language supports abstraction. Here is a query that finds the names of all people with ages in a given range, inclusive of the lower bound but exclusive of the upper bound.

```
type Names = {name : string} list
let range : Expr< (int, int) → Names > =
   <@ fun(a, b) → for w in (%db).people do
                   if a ≤ w.age && w.age < b then
                   yield {name : w.name} @>
```

We insist that the answer type always corresponds to a table, so here we return a list of records with a name field, rather than just a list of strings.

Here we have abstracted in the quoted language rather than the host language. As we shall see, this is essential to being able to reuse queries flexibly in constructing other queries. As we explain in Section 6, we recommend abstracting in the quoted language rather that the host language whenever possible, because it supports composition.

Here we use the usual F# notation for function abstraction. Function applications inside queries normalise by beta reduction:

$$(\mathbf{fun}(\overline{x}) \rightarrow N)(\overline{M}) \rightsquigarrow N[\overline{x := M}],$$

where $N[\overline{x := M}]$ denotes the capture-avoiding substitution of terms $\overline{M}$ for variables $\overline{x}$ in term N.

We form a specific query by instantiating the parameters:

```
run(<@ (%range)(30, 40) @>)                    (2)
```

Evaluating (2) finds everyone in their thirties:

```
[{name = "Cora"}; {name = "Drew"}]
```

In this case, the term passed to **run** is not quite in normal form: it requires one step of beta-reduction, substituting the actuals 30 and 40 for the formals a and b.

More generally, we can instantiate arbitrary values by use of the **lift** operator, which lifts a value of some base type O into a quoted expression of type Expr< O >.

```
let a = 30
let b = 40
run(<@ (%range)(lift(a), lift(b)) @>)
```

This returns the same value as the previous query.

2.4 Abstracting over a Predicate

In general, we may abstract over an arbitrary predicate.

```
let satisfies : Expr< (int → bool) → Names > =
   <@ fun(p) → for w in (%db).people do
               if p(w.age) then
               yield {name : w.name} @>
```

A predicate over ages is denoted by a function from integers to booleans. We form a specific query by instantiating the predicate:

```
run(<@ (%satisfies)(fun(x) → 30 ≤ x && x < 40) @>)    (3)
```

Evaluating (3) yields the same query as (2). In this case, the term passed to **run** requires two steps of beta-reduction to normalise. The first step replaces p by the function, and enables the second step, which replaces x by w.age.

We can instantiate the query with any predicate, so long as it only contains operators available in SQL:

```
run(<@ (%satisfies)(fun(x) → x mod 2 = 0) @>)    (4)
```

Evaluating (4) finds everyone whose age is even. It would not work if, say, the predicate invoked recursion.

2.5 Composing Queries

Uniformly defining queries as quotations makes it easy to compose queries. Say that given two names, we wish to find the names of everyone at least as old as the first but no older than the second. To express this concisely, we define an auxiliary query that finds a person's age.

```
let getAge : Expr< string → int list > =
   <@ fun(s) → for u in (%db).people do
               if u.name = s then yield u.age @>
```

If names are keys, this will return at most one age. It returns a list of integers, not a list of records, so it is not suitable for use as a query on its own, but may be used inside other queries. We may now form our query by composing two uses of the auxiliary getAge with the query range.

```
let compose : Expr< (string, string) → Names > =
   <@ fun(s, t) → for a in (%getAge)(s) do
                  for b in (%getAge)(t) do
                  (%range)(a, b) @>
```

We form a specific query by instantiating the parameters.

```
run(<@ (%compose)("Edna", "Bert") @>)    (5)
```

Evaluating (5) yields the value:

```
[{name = "Cora"}; {name = "Drew"}; {name = "Edna"}]
```

Unlike the previous examples, normalisation of this query requires rules other than beta-reduction; it is described in detail in Section 5.4.

2.6 Dynamically Generated Queries

We now consider dynamically generated queries. The following algebraic datatype represents predicates over integers as abstract syntax trees.

```
type Predicate =
  | Above of int
  | Below of int
  | And of Predicate × Predicate
  | Or of Predicate × Predicate
  | Not of Predicate
```

We take Above(a) to denote all ages greater than or equal to a, and Below(a) to denote all ages strictly less than a, so each is the negation of the other.

For instance, the following trees both specify predicates that select everyone in their thirties:

```
let t₀ : Predicate = And(Above(30), Below(40))
let t₁ : Predicate = Not(Or(Below(30), Above(40)))
```

Given a tree representing a predicate we can compute the quotation of a function representing the predicate. We make use of the **lift** operator, which lifts a value of some base type O into a quoted expression of type Expr< O >. The definition is straightforward.

```
let rec P(t : Predicate) : Expr< int → bool > =
  match t with
  | Above(a)→ <@ fun(x) → (%lift(a)) ≤ x @>
  | Below(a)→ <@ fun(x) → x < (%lift(a)) @>
  | And(t, u)→ <@ fun(x) → (%P(t))(x) && (%P(u))(x) @>
  | Or(t, u)  → <@ fun(x) → (%P(t))(x) || (%P(u))(x) @>
  | Not(t)    → <@ fun(x) → not((%P(t))(x)) @>
```

For instance, P(t₀) returns

```
<@ fun(x) → (fun(x) → 30 ≤ x)(x) &&
            (fun(x) → x < 40)(x) @>
```

Applying normalisation to the above simplifies it to

```
<@ fun(x) → 30 ≤ x && x < 40 @>.
```

Note how normalisation enables modular construction of a dynamic query.

We can combine P with the previously defined satisfies to find all people that satisfy a given predicate:

```
run(<@ (%satisfies)(%P(t₀)) @>)          (6)
```

Evaluating (6) yields the same query as (2) and (3). We may also instantiate to a different predicate:

```
run(<@ (%satisfies)(%P(t₁)) @>)          (7)
```

Evaluating (7) yields the same answer as (6), though it normalises to a slightly different term, where the test 30 ≤ w.age && w.age < 40 is replaced by **not**(w.age < 30 || 40 ≤ w.age).

This series of examples illustrates our key result: including abstraction in the quoted language and normalising quoted terms supports abstraction over values, abstraction over predicates, composition of queries, and dynamic generation of queries.

3. Nesting

We now consider nested data, and show further advantages of the use of normalisation before execution of a query.

```
{departments =
   [{dpt = "Product"}; {dpt = "Quality"};
    {dpt = "Research"}; {dpt = "Sales"}];
 employees =
   [{dpt = "Product"; emp = "Alex"};
    {dpt = "Product"; emp = "Bert"};
    {dpt = "Research"; emp = "Cora"};
    {dpt = "Research"; emp = "Drew"};
    {dpt = "Research"; emp = "Edna"};
    {dpt = "Sales"; emp = "Fred"}];
 tasks =
   [{emp = "Alex"; tsk = "build"};
    {emp = "Bert"; tsk = "build"};
    {emp = "Cora"; tsk = "abstract"};
    {emp = "Cora"; tsk = "build"};
    {emp = "Cora"; tsk = "design"};
    {emp = "Drew"; tsk = "abstract"};
    {emp = "Drew"; tsk = "design"};
    {emp = "Edna"; tsk = "abstract"};
    {emp = "Edna"; tsk = "call"};
    {emp = "Edna"; tsk = "design"};
    {emp = "Fred"; tsk = "call"}]}
```

Figure 3. Organisation as Flat Data

For purposes of illustration, we consider a simplified database representing an organisation, with tables listing departments, employees belonging to each department, and tasks performed by each employee. Its type is Org, defined as follows.

```
type Org = {departments : {dpt : string} list;
            employees :   {dpt : string; emp : string} list;
            tasks :       {emp : string; tsk : string} list }
```

We bind a variable to a reference to the relevant database.

```
let org : Expr< Org > = <@ database("Org") @>
```

The corresponding data is shown in Figure 3.

The following parameterised query finds departments where *every* employee can perform a given task u.

```
let expertise′ : Expr< string → {dpt : string} list > =
  <@ fun(u) →
      for d in (%org).departments do
      if not(exists(
        for e in (%org).employees do
        if d.dpt = e.dpt && not(exists(
          for t in (%org).tasks do
          if e.emp = t.emp && t.tsk = u then yield { })
        )) then yield { })
      )) then yield {dpt = d.dpt} @>
```

Evaluating

```
run(<@ (%expertise′)("abstract") @>)          (8)
```

finds departments where every employee can abstract:

```
[{dpt = "Quality"}; {dpt = "Research"}]
```

There are no employees in the Quality department, so it will be contained in the result of this query regardless of the task specified.

Query expertise′ works as follows. The innermost **for** yields an empty record for each task t performed by employee e that is equal to u; the resulting list is empty if employee e cannot perform task u. The middle **for** yields an empty record for each employee e in department d that cannot perform task u; the resulting list is empty if every employee in department d can perform task u. Therefore,

```
[{dpt = "Product"; employees =
   [{emp = "Alex"; tasks = ["build"]}
    {emp = "Bert"; tasks = ["build"]}]};
 {dpt = "Quality"; employees = []};
 {dpt = "Research"; employees =
   [{emp = "Cora"; tasks = ["abstract"; "build"; "design"]};
    {emp = "Drew"; tasks = ["abstract"; "design"]};
    {emp = "Edna"; tasks = ["abstract"; "call"; "design"]}]};
 {dpt = "Sales"; employees =
   [{emp = "Fred"; tasks = ["call"]}]}]
```

Figure 4. Organisation as Nested Data

the outermost **for** yields departments where every employee can perform task u. We stick a prime on the name to warn that the query is hard to read. Using nested intermediate data structures will help us formulate a more readable equivalent.

3.1 Nested Structures

An alternative way to represent an organisation uses nesting, where each department record contains a list of employees and each employee record contains a list of tasks:

```
type NestedOrg =
  {dpt : string; employees :
     {emp : string; tasks : string list} list} list
```

We convert the first representation into the second as follows:

```
let nestedOrg : Expr< NestedOrg > =
  <@ for d in (%org).departments do
     yield {dpt = d.dpt; employees =
               for e in (%org).employees do
               if d.dpt = e.dpt then
               yield {emp = e.emp; tasks =
                         for t in (%org).tasks do
                         if e.emp = t.emp then
                         yield t.tsk}}} @>
```

If org is bound to the data in Figure 3, then nestedOrg is bound to the data in Figure 4. We cannot write **run**(nestedOrg) to compute this value directly, because **run** requires an argument that is flat, and the type of nestedOrg is nested. However, it can be convenient to use nestedOrg to formulate other queries, as we now show.

3.2 A Query over a Nested Structure

For convenience, we define several higher-order queries. The first takes a predicate and a list and returns **true** if the predicate holds for *any* item in the list.

```
let any : Expr< (A list, A → bool) → bool > =
  <@ fun(xs, p) →
       exists(for x in xs do if p(x) then yield { }) @>
```

The second takes a predicate and a list and returns **true** if the predicate holds for *all* items in the list. It is defined in terms of any using De Morgan duality.

```
let all : Expr< (A list, A → bool) → bool > =
  <@ fun(xs, p) →
       not((%any)(xs, fun(x) → not(p(x)))) @>
```

The third takes a value and a list and returns **true** if the value appears in the list. It is also defined in terms of any.

```
let contains : Expr< (A list, A) → bool > =
  <@ fun(xs, u) → (%any)(xs, fun(x) → x = u) @>
```

All three of these resemble well-known operators from functional programming, and similar operators with the same names are provided in Microsoft's LINQ framework. We define all three as quotations, so that they may be used in queries.

We define a query equivalent to expertise′ as follows:

```
let expertise : Expr< string → {dpt : string} list > =
  <@ fun(u) →
       for d in (%nestedOrg)
       if (%all)(d.employees,
            fun(e) → (%contains)(e.tasks, u) then
       yield {dpt = d.dpt} @>
```

Evaluating

```
run(<@ (%expertise)("abstract") @>)                (9)
```

yields the same query as the previous example, (8).

In order for this to work, normalisation must not only perform beta-reduction, but also perform various reductions on sequence expressions that are well known from the literature on conservativity results. The complete set of reductions that we require is discussed in Section 5.

4. From XPath to SQL

So far, all of our examples (except satisfies) could have been written in a higher-order variant of SQL, implemented using normalisation; they do not illustrate the full power of our approach. We now consider the problem of dynamic generation of SQL queries that simulate XPath queries over XML data represented as relations. This requires not only support for higher-order queries within SQL, but also the capability to construct queries dynamically using recursion and other host language features.

We represent tree-structured XML in a relation using "stretched" pre-order and post-order indexes; see, for example, Grust et al. (2004, sec. 4.2). Each node of the tree corresponds to a row in a table xml with schema:

```
type Node =
  {id : int, parent : int, name : string, pre : int, post : int}
```

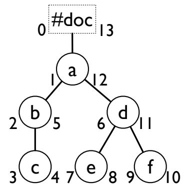
The id field uniquely identifies each node; the parent field refers to the identifier of the node's parent (or -1 for the root node); the name field stores the element tag name; and the pre and post fields store the position of the opening and closing brackets of the node in its serialisation. For example, Figure 5 shows an XML tree and its tabular representation.

The datatypes Axis and Path, defined below, represent the abstract syntax of a fragment of XPath.

```
type Axis =              type Path =
  | Self                   | Seq of Path × Path
  | Child                  | Axis of Axis
  | Descendant             | Name of string
  | DescendantOrSelf       | Filter of Path
  | Following
  | FollowingSibling
  | Rev of Axis
```

The Axis datatype defines the primitive forward axes and Rev to reverse the axes (the reverse of child is parent, the reverse of descendant is ancestor, and so on). The Path datatype defines Seq to concatenate two paths, Axis to define an axis step, Name to test whether an element's name is equal to a given string, and Filter to test whether a path expression is satisfiable from a given node.

Figure 6 gives the complete code of an evaluator for this fragment of XPath, which generates one SQL query per XPath query. The functions axis and path are defined by case analysis over the datatypes Axis and Path, respectively; they yield predicates that



xml

id	parent	name	pre	post
0	-1	"#doc"	0	13
1	0	"a"	1	12
2	1	"b"	2	5
3	2	"c"	3	4
4	1	"d"	6	11
5	4	"e"	7	8
6	4	"f"	9	10

Figure 5. XML Tree and Tabular Representation

```
let rec axis(ax : Axis) : Expr< (Node, Node) → bool > =
  match ax with
  | Self → <@ fun(s, t) → s.id = t.id @>
  | Child → <@ fun(s, t) → s.id = t.parent @>
  | Descendant → <@ fun(s, t) →
      s.pre < t.pre && t.post < s.post @>
  | DescendantOrSelf → <@ fun(s, t) →
      s.pre ≤ t.pre && t.post ≤ s.post @>
  | Following → <@ fun(s, t) → s.post < t.pre @>
  | FollowingSibling → <@ fun(s, t) →
      s.post < t.pre && s.parent = t.parent @>
  | Rev(axis) → <@ fun(s, t) → (%axis(ax))(t, s) @>
let rec path(p : Path) : Expr< (Node, Node) → bool > =
  match p with
  | Seq(p, q) → <@ fun(s, u) → (%any)((%db).xml,
      fun(t) → (%path(p))(s, t) && (%path(q))(t, u)) @>
  | Axis(ax) → axis(ax)
  | Name(name) → <@ fun(s, t) →
      s.id = t.id && s.name = name @>
  | Filter(p) → <@ fun(s, t) → s.id = t.id &&
      (%any)((%db).xml, fun(u) → (%path(p))(s, u)) @>
let xpath(p : Path) : Expr< int list > =
  <@ for root in (%db).xml do
     for s in (%db).xml do
     if root.parent = -1 && (%path(p))(root, s) then
     yield s.id @>
```

Figure 6. An Evaluator for XPath

hold when two nodes are related by the given axis or path. The function xpath translates a Path p to a query expression that returns each node in the table that matches p, starting from the root.

In our tests we consider the following example paths.

$$\begin{aligned} \mathsf{xp}_0 &= \texttt{/*/*} && (10)\\ \mathsf{xp}_1 &= \texttt{//*/parent::*} && (11)\\ \mathsf{xp}_2 &= \texttt{//*[following-sibling::d]} && (12)\\ \mathsf{xp}_3 &= \texttt{//f[ancestor::*/preceding::b]} && (13) \end{aligned}$$

Translation of XPath to Path is straightforward; for example, xp_0 is Seq(Axis(Child), Axis(Child)). The four queries yield the results [2,4], [1,2,4], [2], and [6], respectively on the example data in Figure 5.

While this is a small fragment of XPath, there is no obstacle to adding other features such as attributes, boolean operations on filters, or tests on text data.

5. Core Language

In this section we give a formal account of T-LINQ as a lambda calculus with comprehension, quotation, and constructs to access a database and run queries. Queries are constructed by quotation, and—crucially!—the quoted term is normalised as part of the process of issuing a query.

In practice, host and quoted languages are usually identical, but this is a design choice. In T-LINQ, there are small, but important, differences between the host and quoted languages.

$$\begin{array}{lll}
\text{(base)} & O & ::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{string} \\
\text{(type)} & A, B & ::= O \mid A \to B \mid \{\overline{\ell : A}\} \mid A\ \mathbf{list} \mid \mathsf{Expr}{<}\,A\,{>} \\
\text{(table)} & T & ::= \{\overline{\ell : O}\}\ \mathbf{list} \\
\text{(env't)} & \Gamma, \Delta & ::= \cdot \mid \Gamma, x : A \\
\text{(term)} & L, M, N & ::= c \mid op(\overline{M}) \mid \mathbf{lift}\ M \mid x \mid \mathbf{fun}(x) \to N \\
& & \quad\mid\ L\ M \mid \mathbf{rec}\ f(x) \to N \mid \{\overline{\ell = M}\} \mid L.\ell \\
& & \quad\mid\ \mathbf{yield}\ M \mid \texttt{[]} \mid M\ \texttt{@}\ N \mid \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N \\
& & \quad\mid\ \mathbf{exists}\ M \mid \mathbf{if}\ L\ \mathbf{then}\ M \mid \mathbf{run}\ M \\
& & \quad\mid\ \texttt{<@}\ M\ \texttt{@>} \mid \mathbf{database}(\mathsf{db}) \mid (\%M)
\end{array}$$

Figure 7. Syntax of T-LINQ

5.1 Typing Rules

The syntax of types and terms is given in Figure 7 and the typing rules are given in Figure 8. There are two typing judgements, one for host terms and one for quoted terms. Judgement $\Gamma \vdash M : A$ states that host term M has type A in type environment Γ, and judgement $\Gamma; \Delta \vdash M : A$ states that quoted term M has type A in host type environment Γ and quoted type environment Δ. For simplicity, we assume that all queries are on a single database db; in practice, we check dynamically that each query refers to a single database. We assume a signature Σ that maps each constant c, each primitive operator op, and the database db to its type.

Most of the typing rules are standard, and mirrored across both judgements; we list here the rules that differ. Recursion is only available in a host term (REC); other operations available to the host but not the database may be modelled similarly. The database is only available in a quoted term (DATABASE). A term of base type O can be lifted to a quoted term of type Expr< O > (LIFT). A quoted term of table type T can be evaluated as a query (RUN). The rule for quoting requires the quoted type environment to be empty, ensuring quoted terms are closed (QUOTE). The rule for splicing is the only place the host type environment Γ is referenced in the typing rules for quoted terms (ANTIQUOTE).

5.2 Operational Semantics

The syntax of values and evaluation contexts is given in Figure 9. Values are standard, save that we add values <@ Q @>, where Q is a *quotation value*, a quoted term in which all anti-quotes have been resolved. We write $[\overline{V}]$ to abbreviate **yield** V_1 @ $\cdots$ @ **yield** V_n @ [ ]. The semantics is parameterised by an interpretation δ for each primitive operation op, and an interpretation Ω for the database db, both of which respect types: if $\Sigma(op) = \overline{O} \to O$ and $\vdash \overline{V : O}$ and $V = \delta(op, \overline{V})$ then $\vdash V : O$, and $\vdash \Omega(\mathsf{db}) : \Sigma(\mathsf{db})$.

Reduction $M \longrightarrow N$ is the relation in Figure 10. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The rules are standard apart from those for quotation and query evaluation. All of the rules are interpreted relative to a fixed database; the *eval* function evaluates the query against this database. Evaluation contexts $\mathcal{E}$ enforce left-to-right call-by-value evaluation, and quotation contexts $\mathcal{Q}$ are contexts over quoted terms that have no anti-quotation to the left of the hole. Rule (lift) converts a constant into a quoted constant, and rule (splice) resolves an anti-quote once its body has been evaluated to a quotation. Rule (run) evaluates a quotation value Q by first *normalising* Q to yield an equivalent SQL query $S = norm(Q)$, and then *evaluating* S on the database to yield the

$\boxed{\Gamma \vdash M : A}$

$$\textsc{Const}\ \frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \textsc{Op}\ \frac{\Sigma(op) = (\overline{O}) \to O \qquad \overline{\Gamma \vdash M : O}}{\Gamma \vdash op(\overline{M}) : O} \qquad \textsc{Lift}\ \frac{\Gamma \vdash M : O}{\Gamma \vdash \textbf{lift}\ M : \mathsf{Expr}\texttt{<}O\texttt{>}} \qquad \textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Fun}\ \frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \textbf{fun}(x) \to N : A \to B}$$

$$\textsc{App}\ \frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B} \qquad \textsc{Rec}\ \frac{\Gamma, f : A \to B, x : A \vdash N : B}{\Gamma \vdash \textbf{rec}\ f(x) \to N : A \to B} \qquad \textsc{Record}\ \frac{\overline{\Gamma \vdash M : A}}{\Gamma \vdash \{\overline{\ell = M}\} : \{\overline{\ell : A}\}} \qquad \textsc{Project}\ \frac{\Gamma \vdash L : \{\overline{\ell : A}\}}{\Gamma \vdash L.\ell_i : A_i}$$

$$\textsc{Singleton}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{yield}\ M : A\ \textbf{list}} \qquad \textsc{Empty}\ \frac{}{\Gamma \vdash \texttt{[]} : A\ \textbf{list}} \qquad \textsc{Union}\ \frac{\Gamma \vdash M : A\ \textbf{list} \qquad \Gamma \vdash N : A\ \textbf{list}}{\Gamma \vdash M\ \texttt{@}\ N : A\ \textbf{list}} \qquad \textsc{For}\ \frac{\Gamma \vdash M : A\ \textbf{list} \qquad \Gamma, x : A \vdash N : B\ \textbf{list}}{\Gamma \vdash \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N : B\ \textbf{list}}$$

$$\textsc{Exists}\ \frac{\Gamma \vdash M : A\ \textbf{list}}{\Gamma \vdash \textbf{exists}\ M : \textbf{bool}} \qquad \textsc{If}\ \frac{\Gamma \vdash L : \textbf{bool} \qquad \Gamma \vdash M : A\ \textbf{list}}{\Gamma \vdash \textbf{if}\ L\ \textbf{then}\ M : A\ \textbf{list}} \qquad \textsc{Run}\ \frac{\Gamma \vdash M : \mathsf{Expr}\texttt{<}T\texttt{>}}{\Gamma \vdash \textbf{run}\ M : T} \qquad \textsc{Quote}\ \frac{\Gamma; \cdot \vdash M : A}{\Gamma \vdash \texttt{<@}\ M\ \texttt{@>} : \mathsf{Expr}\texttt{<}A\texttt{>}}$$

$\boxed{\Gamma; \Delta \vdash M : A}$

$$\textsc{ConstQ}\ \frac{\Sigma(c) = A}{\Gamma; \Delta \vdash c : A} \qquad \textsc{OpQ}\ \frac{\Sigma(op) = (\overline{O}) \to O \qquad \overline{\Gamma; \Delta \vdash M : O}}{\Gamma; \Delta \vdash op(\overline{M}) : O} \qquad \textsc{VarQ}\ \frac{x : A \in \Delta}{\Gamma; \Delta \vdash x : A} \qquad \textsc{FunQ}\ \frac{\Gamma; \Delta, x : A \vdash N : B}{\Gamma; \Delta \vdash \textbf{fun}(x) \to N : A \to B}$$

$$\textsc{AppQ}\ \frac{\Gamma; \Delta \vdash L : A \to B \qquad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash L\ M : B} \qquad \textsc{RecordQ}\ \frac{\overline{\Gamma; \Delta \vdash M : A}}{\Gamma; \Delta \vdash \{\overline{\ell = M}\} : \{\overline{\ell : A}\}} \qquad \textsc{ProjectQ}\ \frac{\Gamma; \Delta \vdash L : \{\overline{\ell : A}\}}{\Gamma; \Delta \vdash L.\ell_i : A_i} \qquad \textsc{SingletonQ}\ \frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \textbf{yield}\ M : A\ \textbf{list}}$$

$$\textsc{EmptyQ}\ \frac{}{\Gamma; \Delta \vdash \texttt{[]} : A\ \textbf{list}} \qquad \textsc{UnionQ}\ \frac{\Gamma; \Delta \vdash M : A\ \textbf{list} \qquad \Gamma; \Delta \vdash N : A\ \textbf{list}}{\Gamma; \Delta \vdash M\ \texttt{@}\ N : A\ \textbf{list}} \qquad \textsc{ForQ}\ \frac{\Gamma; \Delta \vdash M : A\ \textbf{list} \qquad \Gamma; \Delta, x : A \vdash N : B\ \textbf{list}}{\Gamma; \Delta \vdash \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N : B\ \textbf{list}}$$

$$\textsc{ExistsQ}\ \frac{\Gamma; \Delta \vdash M : A\ \textbf{list}}{\Gamma; \Delta \vdash \textbf{exists}\ M : \textbf{bool}} \qquad \textsc{IfQ}\ \frac{\Gamma; \Delta \vdash L : \textbf{bool} \qquad \Gamma; \Delta \vdash M : A\ \textbf{list}}{\Gamma; \Delta \vdash \textbf{if}\ L\ \textbf{then}\ M : A\ \textbf{list}} \qquad \textsc{Database}\ \frac{\Sigma(\mathsf{db}) = \{\overline{\ell : T}\}}{\Gamma; \Delta \vdash \textbf{database}(\mathsf{db}) : \{\overline{\ell : T}\}} \qquad \textsc{Antiquote}\ \frac{\Gamma \vdash M : \mathsf{Expr}\texttt{<}A\texttt{>}}{\Gamma; \Delta \vdash \texttt{(\%}M\texttt{)} : A}$$

Figure 8. Typing Rules for T-LINQ

value $V = eval(\Omega, S)$. Define $eval(\Omega, Q) = V$ when $\Omega(Q) \longrightarrow^* V$, where $\Omega(Q)$ replaces each occurrence of **database**(db) in Q by $\Omega(\mathsf{db})$. The following section defines $norm(Q)$, and shows that normalisation preserves types and meaning: if $\cdot; \cdot \vdash Q : T$ and $S = norm(Q)$ then $\vdash S : T$ and $eval(\Omega, S) = eval(\Omega, Q)$.

It is straightforward to show that type soundness holds, via the usual method of preservation and progress.

Proposition 1. *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow N$ *then* $\Gamma \vdash N : A$. *If* $\Gamma \vdash M : A$ *then either* M *is a value or* $M \longrightarrow N$ *for some* N.

5.3 Query Normalisation

In this section we establish the Scylla and Charybdis theorem: Every quoted term of flat type can be normalised to a term that is isomorphic to a single SQL query.

Query normalisation is central to our technique. Similar techniques go back to Wong (1996), and the work here is based directly on Cooper (2009) and Lindley and Cheney (2012). The query normalisation function $norm$ is based on two reduction relations, symbolic reduction, $P \rightsquigarrow Q$, and *ad-hoc* reduction, $P \hookrightarrow Q$. We write $\rightsquigarrow^*$ and $\hookrightarrow^*$ for the reflexive and transitive closure of $\rightsquigarrow$ and $\hookrightarrow$ respectively. Define $norm(P) = R$ when $P \rightsquigarrow^* Q$ and $Q \hookrightarrow^* R$, where Q and R are in normal form with respect to $\rightsquigarrow$ and $\hookrightarrow$, respectively.

Symbolic reduction $P \rightsquigarrow Q$ is the compatible closure of the rules in Figure 11. The rules are straightforward, including beta-reduction for functions, records, and booleans, plus the usual laws for monads with sums (Trinder 1991). Terms in normal form under this relation satisfy the subformula property: with the exception of predicates (such as $<$ or **exists**), the type of a subterm must be a subformula of either the type of a free variable or of the type of the term (Prawitz 1965). Hence, symbolic reduction eliminates nesting from a term that returns a value of table type.

Ad-hoc reduction, $P \hookrightarrow Q$, is the compatible closure of the rules in Figure 12. These reductions account for the lack of uniformity in SQL. Rule (for-@), which hoists a union out of a comprehension body, is the only rule that is not sound for a list semantics, since it changes the order in which elements are generated.

Rewriting preserves types and meaning.

Proposition 2. *If* $\vdash P : A$ *and* $P \rightsquigarrow Q$ *or* $P \hookrightarrow Q$ *then* $\vdash Q : A$ *and* $eval(\Omega, P) = eval(\Omega, Q)$.

The normal form of a query is easy to compute because rewrites may be applied in any order and rewriting always terminates.

$$
\begin{array}{lll}
\text{(value)} & & \\
\quad V, W & ::= & c \mid \mathbf{fun}(x) \to M \mid \mathbf{rec}\, f(x) \to M \mid \{\overline{\ell = V}\} \\
 & \mid & [\overline{V}] \mid \texttt{<@}\, Q \,\texttt{@>} \\
\text{(quotation value)} & & \\
\quad P, Q, R & ::= & c \mid op(\overline{Q}) \mid \mathbf{lift}\, Q \mid x \mid \mathbf{fun}(x) \to R \mid P\, Q \\
 & \mid & \{\overline{\ell = Q}\} \mid P.\ell \mid \mathbf{yield}\, Q \mid \texttt{[]} \mid Q \,\texttt{@}\, R \\
 & \mid & \mathbf{for}\, x\, \mathbf{in}\, Q\, \mathbf{do}\, R \mid \mathbf{exists}\, Q \mid \mathbf{if}\, P\, \mathbf{then}\, Q \\
 & \mid & \mathbf{database}(\mathsf{db}) \\
\text{(evaluation context)} & & \\
\quad \mathcal{E} & ::= & [\,] \mid op(\overline{V}, \mathcal{E}, \overline{M}) \mid \mathbf{lift}\, \mathcal{E} \mid \mathcal{E}\, M \mid V\, \mathcal{E} \\
 & \mid & \{\overline{\ell = V}, \ell' = \mathcal{E}, \overline{\ell'' = M}\} \mid \mathcal{E}.\ell \mid \mathbf{yield}\, \mathcal{E} \\
 & \mid & \mathcal{E} \,\texttt{@}\, M \mid V \,\texttt{@}\, \mathcal{E} \mid \mathbf{for}\, x\, \mathbf{in}\, \mathcal{E}\, \mathbf{do}\, N \mid \mathbf{exists}\, \mathcal{E} \\
 & \mid & \mathbf{if}\, \mathcal{E}\, \mathbf{then}\, M \mid \mathbf{run}\, \mathcal{E} \mid \texttt{<@}\, \mathcal{Q}[(\%\mathcal{E})] \,\texttt{@>} \\
\text{(quotation context)} & & \\
\quad \mathcal{Q} & ::= & [\,] \mid op(\overline{Q}, \mathcal{Q}, \overline{M}) \mid \mathbf{lift}\, \mathcal{Q} \mid \mathbf{fun}(x) \to \mathcal{Q} \\
 & \mid & \mathcal{Q}\, M \mid Q\, \mathcal{Q} \mid \{\overline{\ell = Q}, \ell' = \mathcal{Q}, \overline{\ell'' = M}\} \\
 & \mid & \mathcal{Q}.\ell \mid \mathbf{yield}\, \mathcal{Q} \mid \mathcal{Q} \,\texttt{@}\, M \mid Q \,\texttt{@}\, \mathcal{Q} \\
 & \mid & \mathbf{for}\, x\, \mathbf{in}\, \mathcal{Q}\, \mathbf{do}\, N \mid \mathbf{for}\, x\, \mathbf{in}\, Q\, \mathbf{do}\, \mathcal{Q} \\
 & \mid & \mathbf{exists}\, \mathcal{Q} \mid \mathbf{if}\, \mathcal{Q}\, \mathbf{then}\, M \mid \mathbf{if}\, Q\, \mathbf{then}\, \mathcal{Q} \mid \mathbf{run}\, \mathcal{Q}
\end{array}
$$

Figure 9. Values and Evaluation Contexts

$$
\begin{array}{rcll}
op(\overline{V}) & \longrightarrow & \delta(op, \overline{V}) & \\
(\mathbf{fun}(x) \to N)\, V & \longrightarrow & N[x := V] & \\
(\mathbf{rec}\, f(x) \to N)\, V & \longrightarrow & M[f := \mathbf{rec}\, f(x) \to N, x := V] & \\
\{\overline{\ell = V}\}.\ell_i & \longrightarrow & V_i & \\
\mathbf{if\ true\ then}\, M & \longrightarrow & M & \\
\mathbf{if\ false\ then}\, M & \longrightarrow & \texttt{[]} & \\
\mathbf{for}\, x\, \mathbf{in\ yield}\ V\, \mathbf{do}\, M & \longrightarrow & M[x := V] & \\
\mathbf{for}\, x\, \mathbf{in}\, \texttt{[]}\, \mathbf{do}\, N & \longrightarrow & \texttt{[]} & \\
\mathbf{for}\, x\, \mathbf{in}\, L \,\texttt{@}\, M\, \mathbf{do}\, N & \longrightarrow & & \\
 & & (\mathbf{for}\, x\, \mathbf{in}\, L\, \mathbf{do}\, N) \,\texttt{@}\, (\mathbf{for}\, x\, \mathbf{in}\, M\, \mathbf{do}\, N) & \\
\mathbf{exists}\, \texttt{[]} & \longrightarrow & \mathbf{false} & \\
\mathbf{exists}\, [\overline{V}] & \longrightarrow & \mathbf{true}, \quad |\overline{V}| > 0 & \\
\mathbf{run}\, \texttt{<@}\, Q \,\texttt{@>} & \longrightarrow & eval(norm(Q)) & \text{(run)} \\
\mathbf{lift}\, c & \longrightarrow & \texttt{<@}\, c \,\texttt{@>} & \text{(lift)} \\
\texttt{<@}\, \mathcal{Q}[(\%\texttt{<@}\, Q \,\texttt{@>})] \,\texttt{@>} & \longrightarrow & \texttt{<@}\, \mathcal{Q}[Q] \,\texttt{@>} & \text{(splice)}
\end{array}
$$

$$
\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}
$$

Figure 10. Operational Semantics for T-LINQ

PROPOSITION 3. *Both $\rightsquigarrow$ and $\hookrightarrow$ are confluent and strongly normalising for typed terms.*

The proof is straightforward. Factoring into two relations makes the strong normalisation proof easier than in Cooper (2009).

The grammar of normalised terms, given in Figure 13, is essentially isomorphic to a subset of SQL. The correspondence is not quite exact, because the SQL standard has no notation for empty records, and lacks constructs for an empty table or to access a table or table variable directly (the first constructs of S, X, and Y, respectively); but these idiosyncrasies are easy to work around, and are handled already by LINQ. It is straightforward to establish that if Q is a closed term of table type T, then its normalisation exists and matches the grammar of S.

PROPOSITION 4 (Scylla and Charybdis). *If $\vdash Q : T$ then there exists an S such that $S = norm(Q)$.*

$$
\begin{array}{rcll}
(\mathbf{fun}(x) \to R)\, Q & \rightsquigarrow & R[x := Q] & \\
\{\overline{\ell = Q}\}.\ell_i & \rightsquigarrow & Q_i & \\
\mathbf{for}\, x\, \mathbf{in}\, (\mathbf{yield}\, Q)\, \mathbf{do}\, R & \rightsquigarrow & R[x := Q] & \text{(for-yld)} \\
\mathbf{for}\, y\, \mathbf{in}\, (\mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, Q)\, \mathbf{do}\, R & \rightsquigarrow & & \\
 & & \mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, (\mathbf{for}\, y\, \mathbf{in}\, Q\, \mathbf{do}\, R) & \text{(for-for)} \\
\mathbf{for}\, x\, \mathbf{in}\, (\mathbf{if}\, P\, \mathbf{then}\, Q)\, \mathbf{do}\, R & \rightsquigarrow & & \\
 & & \mathbf{if}\, P\, \mathbf{then}\, (\mathbf{for}\, x\, \mathbf{in}\, Q\, \mathbf{do}\, R) & \text{(for-if)} \\
\mathbf{for}\, x\, \mathbf{in}\, \texttt{[]}\, \mathbf{do}\, N & \rightsquigarrow & \texttt{[]} & \\
\mathbf{for}\, x\, \mathbf{in}\, (P \,\texttt{@}\, Q)\, \mathbf{do}\, R & \rightsquigarrow & & \\
 & & (\mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, R) \,\texttt{@}\, (\mathbf{for}\, x\, \mathbf{in}\, Q\, \mathbf{do}\, R) & \\
\mathbf{if\ true\ then}\, Q & \rightsquigarrow & Q & \\
\mathbf{if\ false\ then}\, Q & \rightsquigarrow & \texttt{[]} &
\end{array}
$$

Figure 11. Normalisation Stage 1: symbolic reduction

$$
\begin{array}{rcll}
\mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, (Q \,\texttt{@}\, R) & \hookrightarrow & & \text{(for-@)} \\
 & & (\mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, Q) \,\texttt{@}\, (\mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, R) & \\
\mathbf{for}\, x\, \mathbf{in}\, P\, \mathbf{do}\, \texttt{[]} & \hookrightarrow & \texttt{[]} & \\
\mathbf{if}\, P\, \mathbf{then}\, (Q \,\texttt{@}\, R) & \hookrightarrow & (\mathbf{if}\, P\, \mathbf{then}\, Q) \,\texttt{@}\, (\mathbf{if}\, P\, \mathbf{then}\, R) & \\
\mathbf{if}\, P\, \mathbf{then}\, \texttt{[]} & \hookrightarrow & \texttt{[]} & \\
\mathbf{if}\, P\, \mathbf{then}\, (\mathbf{if}\, Q\, \mathbf{then}\, R) & \hookrightarrow & \mathbf{if}\, (P \,\texttt{\&\&}\, Q)\, \mathbf{then}\, R & \text{(if-if)} \\
\mathbf{if}\, P\, \mathbf{then}\, (\mathbf{for}\, x\, \mathbf{in}\, Q\, \mathbf{do}\, R) & \hookrightarrow & \mathbf{for}\, x\, \mathbf{in}\, Q\, \mathbf{do}\, (\mathbf{if}\, P\, \mathbf{then}\, R) & \\
 & & & \text{(if-for)}
\end{array}
$$

Figure 12. Normalisation Stage 2: *ad-hoc* reduction

$$
\begin{array}{llll}
\text{(SQL query)} & S & ::= & \texttt{[]} \mid X \mid X_1 \,\texttt{@}\, X_2 \\
\text{(collection)} & X & ::= & \mathbf{database}(\mathsf{db}) \mid \mathbf{yield}\, Y \mid \mathbf{if}\, Z\, \mathbf{then\ yield}\, Y \\
 & & \mid & \mathbf{for}\, x\, \mathbf{in\ database}(\mathsf{db}).\ell\, \mathbf{do}\, X \\
\text{(record)} & Y & ::= & x \mid \{\overline{\ell = Z}\} \\
\text{(base)} & Z & ::= & c \mid x.\ell \mid op(\overline{X}) \mid \mathbf{exists}\, S
\end{array}
$$

Figure 13. Syntax of Normalised Terms

5.4 An Example

As an example of normalisation, we consider evaluation of query (5) from Section 2.5.

```
run(<@ (%compose)("Edna", "Bert") @>)
```

After splicing, the quotation becomes:

```
(fun(s, t) →
   for a in (fun(s) →
               for u in database("People").people do
               if u.name = s then yield u.age)(s) do
   for b in (fun(s) →
               for u in database("People").people do
               if u.name = s then yield u.age)(t) do
   (fun(a, b) →
      for w in database("People").people do
      if a ≤ w.age && w.age < b then
      yield {name : w.name})(a, b))
("Edna", "Bert")
```

For stage 1 (Figure 11), applying four beta-reductions yields:

```
for a in (for u in database("People").people do
           if u.name = "Edna" then yield u.age) do
for b in (for u in database("People").people do
           if u.name = "Bert" then yield u.age) do
for w in database("People").people do
if a ≤ w.age && w.age < b then
yield {name : w.name}
```

Continuing stage 1, applying each of rules (for-for), (for-if), and (for-yld) twice, and renaming to avoid capture, yields:

```
for u in database("People").people do
if u.name = "Edna" then
for v in database("People").people do
if v.name = "Bert" then
for w in database("People").people do
if u.age ≤ w.age && w.age < v.age then
yield {name : w.name}
```

For stage 2 (Figure 12), applying rule (if-for) thrice and rule (if-if) twice yields:

```
for u in database("People").people do
for v in database("People").people do
for w in database("People").people do
if u.name = "Edna" && v.name = "Bert" &&
  u.age ≤ w.age && w.age < v.age then
yield {name : w.name}
```

This is in normal form, and easily converted to SQL. Running it yields the answer given previously.

6. Quoted Language vs. Host Language

We write in a style where we abstract in the quoted language whenever possible. As we shall see, this is essential if several functions are to be composed into a single query, as in Section 2.5. We abstract in the host language only when we need a feature not present in the quoted language, such as recursion, as used to construct dynamic queries in Sections 2.6 and 4.

Another style, which might at first appear appealing, is to always abstract in the host language. For instance, one might redefine range from Section 2.3 as follows.

```
let range′(a : Expr<int>, b : Expr<int>) : Expr<Names> =
  <@ for w in (%db).people do
     if (%a) ≤ w.age && w.age < (%b) then
     yield {name : w.name} @>
```

(Or one might define a variant where a and b have type **int** and lifting is used, which raises similar issues.) Before, we wrote an invocation like this:

```
run(<@ (%range)(30, 40) @>).
```

Now, we write an invocation like this:

```
run(range′(<@ 30 @>, <@ 40 @>)).
```

The latter may be slightly more efficient, as it directly yields a quotation in normal form, and no beta-reduction is required. For this reason, we had originally assumed that one should abstract in the host language, but late in the process of writing this paper we realised this is a mistake. We stick a prime on the name to warn that this form of definition hinders composition.

Let's see what goes wrong with composition. In Section 2.5 we used range to define compose. Attempting a revision using range′ yields the following.

```
let compose′ : Expr<(string, string) → Names> =
  <@ fun(s, t) → for a in (%getAge)(s) do
                 for b in (%getAge)(t) do
                 (%range′(<@ a @>, <@ b @>)) @>
```

Warning: the above is not legal in F# (or T-LINQ)! Previously, all the quotations we saw were *closed*, since every quoted variable is bound within the quotation; but the two quotations <@ a @> and <@ b @> passed to range′ are *open*, since they contain free quoted variables. In this case, the variables become bound after splicing into the surrounding quotation, but, in general, open quotations come with no guarantee that free variables meet their binding occurrences. For this reason, open quotations are illegal in F# and in T-LINQ, and there is no easy way to use range′ to define compose′.

Typing closed quotation is straightforward in current languages, and is supported in F# or in any language with GADTs (Cheney and Hinze 2003). In contrast, open quotation in a typed language requires some form of specialised type system; experimental languages that support open quotation include MetaML (Taha and Sheard 2000) or Ur (Chlipala 2010), and a recent experimental feature in GHC supports MetaML typing rules for quotations. We sketch a simple type system that supports open quotation in Section 7, and show how closed quotation can simulate that form of open quotation, which suggests how closed quotation may be adequate for situations where one might have thought open quotation was required.

While open quotation avoids the cost of some beta-reductions, it does not avoid the need for the other normalisation rules discussed in Section 5.3. Further, the cost of normalising a quoted term is low compared to the cost of evaluating the resulting SQL query against the database, as demonstrated in Section 9. Thus, we believe closed quotation and normalisation provide a practical alternative to more elaborate systems that support open quotation.

7. Open Quotation

Section 6 discussed the difference between closed and open quotation. T-LINQ and P-LINQ, like F#, support only closed quotation. Here we generalise T-LINQ to support a form of open quotation, and we show how this form may be simulated by closed quotation. Choi et al. (2011) give a similar result. Our formalisation of open quotation is not as powerful as that found in systems such as MetaML or Ur, but the translation is suggestive of why closed quotation may be adequate for dealing with cases (such as composition of queries or dynamic generation of queries) that one may previously have thought required open quotation.

For the extension, we add a type environment to the type of quoted terms, generalising $\mathsf{Expr}\langle A\rangle$ to $\mathsf{Expr}\langle \Delta; A\rangle$, where Δ specifies the types of the free variables in the quoted term. Only four typing rules need to change.

$$\textsc{Lift}\quad \frac{\Gamma \vdash M : O}{\Gamma \vdash \mathbf{lift}(M) : \mathsf{Expr}\langle \cdot; O\rangle} \qquad \textsc{Run}\quad \frac{\Gamma \vdash M : \mathsf{Expr}\langle \cdot; T\rangle}{\Gamma \vdash \mathbf{run}\, M : T}$$

$$\textsc{Quote}\quad \frac{\Gamma; \Delta \vdash M : A}{\Gamma \vdash \texttt{<@}\, M\, \texttt{@>}_\Delta : \mathsf{Expr}\langle \Delta; A\rangle} \qquad \textsc{Antiquote}\quad \frac{\Gamma \vdash M : \mathsf{Expr}\langle \Delta; A\rangle}{\Gamma; \Delta \vdash (\%M)_\Delta : A}$$

In order to allow syntax-directed typechecking, we add type environment annotations to quotation and antiquotation expressions.

To simulate the extended language in the original, we represent an open quotation of type $\mathsf{Expr}\langle \Delta; A\rangle$ by a closed quotation of type $\mathsf{Expr}\langle \Delta \to A\rangle$, explicitly abstracting over each of the free variables in the quoted type environment. We specify translations of types, host terms, and query terms from the extended language back into the original language. There is only one case of interest for each translation.

$$\begin{aligned}
[\![\mathsf{Expr}\langle \Delta; A\rangle]\!] &= \mathsf{Expr}\langle [\![\Delta]\!] \to [\![A]\!]\rangle \\
[\![\texttt{<@}\, M\, \texttt{@>}_\Delta]\!] &= \texttt{<@}\, \mathbf{fun}(\Delta) \to [\![M]\!]\, \texttt{@>} \\
[\![(\%M)_\Delta]\!] &= (\%M)\ \Delta
\end{aligned}$$

All of the other cases are defined homomorphically. Here Δ on the right-hand side stands in the first line for a tuple of the types in the environment; in the second line for a tuple of the bindings in the environment, over which the translation is abstracted; and in the third line for a tuple of the variables in the environment, to which the translation is applied. All tuples must be consistently ordered, say alphabetically on the names of the variables in Δ.

PROPOSITION 5. *The translation preserves types, and the extended language is simulated by the original language.*

- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!] \vdash [\![M]\!]$.
- *If* $\Gamma; \Delta \vdash M : A$ *then* $[\![\Gamma]\!]; [\![\Delta]\!] \vdash [\![M]\!] : [\![A]\!]$.
- *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow N$ *then* $[\![M]\!] \longrightarrow [\![N]\!]$.
- *If* $\Gamma; \Delta \vdash M : A$ *and* $M \longrightarrow N$ *then* $[\![M]\!] \longrightarrow [\![N]\!]$.

8. Comparison to Microsoft LINQ

T-LINQ abstracts from several distracting issues in the implementation of Microsoft LINQ for C#, Visual Basic, and F#.

Microsoft's LINQ library includes interfaces IEnumerable<A> and IQueryable<A> that provide standard query operators including selection, join, filtering, grouping, sorting, and aggregation. These query operators are defined to act both on sequences and on quotations that yield sequences. LINQ query expressions in C# or Visual Basic are translated to code that calls the methods in these interfaces. For example, a C# LINQ query

from x **in** e **where** p(x) **select** f(x)

translates to the sequence of calls

e.Where(x $\Rightarrow$ p(x)).Select(x $\Rightarrow$ f(x))

Depending on the context, lambda-abstractions in C# and Visual Basic are treated either as functions or as quoted functions.

Any external data source that can implement some of the query operations can be connected to LINQ using a *query provider*. Implementing a query provider can be difficult, in part because of the overhead of dealing with the Expression<A> type. Eini (2011) characterises the experience of writing a custom query provider as "doom, gloom with just a tad of despair". Microsoft supplies a LINQ to SQL query provider for SQL Server. Microsoft's query provider is proprietary, so its behaviour is a black box, but it does appear to perform some beta-reduction and other normalisation.

As we have already described, F# supports LINQ using syntactic sugar for comprehensions (called *computation expressions* (Petricek and Syme 2012)), quotations, and reflection. In the F# PowerPack library made available for F# 2.0, some LINQ capabilities are supported by a translator from the F# Expr< A > type to the LINQ Expression<A> type. This implementation has some bugs and limitations, for instance, it fails to translate arguments of **exists** in the test of a conditional.

F# 3.0 supports LINQ through an improved translation based on computation expressions (Petricek and Syme 2012). In F# 3.0, one can simply write **query**$\{e\}$ to indicate that a computation expression e should be interpreted as a query; the standard library class QueryBuilder translates e to a C# LINQ expression and evaluates it. This implementation also has some bugs and limitations, for instance, it forbids some uses of splicing, and does not correctly process some queries that start with a conditional.

9. Implementation and Results

To validate our design, we implemented a pre-processor that takes any quoted F# sequence expression over the standard query operators and normalises it as described in Section 5.3. In theory, our normaliser could be followed by either the F# 2.0 or F# 3.0 backend, but the bugs noted in the previous section prevent some of our sample queries from working with each. The F# 2.0 PowerPack is distributed as a separate library and easy to modify, while the F# 3.0 backend is built-in and difficult to modify. Hence, for most of our experiments, we opted to use F# 2.0 LINQ syntax and a modified version of the F# 2.0 backend with our pre-processor. To experiment with grouping, aggregation and other operations that are only supported in F# 3.0, we have also developed a prototype that provides subclasses of the F# 3.0 QueryBuilder library class, along with variants of the **query**$\{\cdots\}$ keyword, that perform normalisation before calling F# 3.0's LINQ implementation. The QueryBuilder implementation employs some subtle tricks to support type-directed dispatch so that **query**$\{\cdots\}$ works for both in-memory and external database calls. Unfortunately, these tricks make it difficult to smoothly override the QueryBuilder class to provide drop-in compatible behaviour; in our implementation, we provide different variants of the query keyword for use in different contexts. This limitation could easily be overcome by a change to the F# 3.0 QueryBuilder class, and we are discussing this possibility with Microsoft. We use the term P-LINQ to refer to both implementations, and specify the back-end F# LINQ implementation used when it is relevant.

Example		F# 2.0	F# 3.0	P-LINQ	*norm*
differences	(1)	17.6	20.6	18.1	0.5
range	(2)	$\times$	5.6	2.9	0.3
satisfies	(3)	2.6	$\times$	2.9	0.3
satisfies	(4)	4.4	$\times$	4.6	0.3
compose	(5)	$\times$	$\times$	4.0	0.8
P(t$_0$)	(6)	2.8	$\times$	3.3	0.3
P(t$_1$)	(7)	2.7	$\times$	3.0	0.3
expertise$'$	(8)	7.2*	9.2	8.0*	0.6
expertise	(9)	$\times$	66.7^{av}	8.3*	0.9
xp$_0$	(10)	$\times$	8.3	7.9	1.9
xp$_1$	(11)	$\times$	14.7	13.4	1.1
xp$_2$	(12)	$\times$	17.9	20.7*	2.2
xp$_3$	(13)	$\times$	3744.9	3768.6*	4.4

All times in milliseconds. $\times$ marks failures.
* marks cases requiring modified F# 2.0 PowerPack library.
^{av} marks the case where a query avalanche occurs.
$|$people$| = 10000$, $|$couples$| = 5000$, $|$employees$| = 5000$, $|$tasks$| = 4931$, $|$xml$| = 6527$

Table 1. Experimental Results.

All experiments were run on a Dell OptiPlex 790 with Intel Core i5-2400 CPU at 3.10 GHz, 4GB RAM and a 7200 RPM hard drive with 8MB cache, and using Microsoft .NET 4.0 runtime, Visual Studio 2012 v11.0.50727.1, and SQL Server 2012, all running on the same machine to avoid any network-related latency. All reported times are the medians of 21 trials. All source code for the examples, the data, and the modified F# 2.0 PowerPack library is available online (Cheney et al. 2012).

Table 1 summarises our experimental results. We wrote and ran versions of each example using the F# 2.0 PowerPack LINQ library, the F# 3.0 LINQ library, and P-LINQ (using the F# 2.0 backend). We randomly generated data for the couples and organisation databases, and used an existing repository of XML data, with sizes as listed in the table. Each entry in the table either indicates that the query failed ($\times$), or gives the total time in milliseconds for successful evaluation, including time to generate the SQL query (or queries, in the case of an avalanche), to evaluate the query, and to construct a value from the result. For P-LINQ, the total includes time to normalise the quoted expression; this is also shown separately in the column labelled *norm*.

F# 2.0 failed on seven examples, and F# 3.0 failed on five, though each succeeds on examples on which the other fails. The modified PowerPack library was required in one case by F# 2.0 and in four cases by P-LINQ. F# 3.0 generated an avalanche of SQL queries for query (9); this example query involves nested intermediate data but its result is flat, in contrast to cases of avalanche

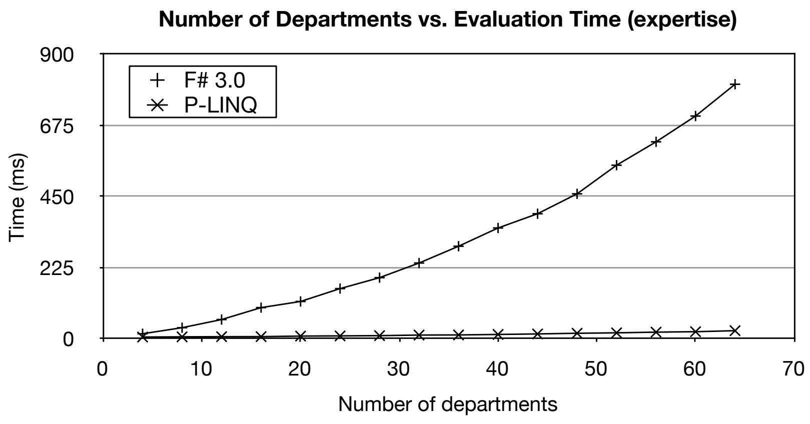


Figure 14. Query avalanche results. Each department is of size 100, with each employee assigned 0, 1, or 2 tasks at random.

Q#	F# 3.0	P-LINQ	*norm*
Q1	2.0	2.4	0.3
Q2	1.5	1.7	0.2
Q5	1.7	2.1	0.3
Q6	1.7	2.1	0.3
Q7	1.5	1.8	0.2
Q8	2.3	2.4	0.2
Q9	2.3	2.7	0.3
Q10	1.4	1.7	0.2
Q11	1.4	1.7	0.2
Q12	4.4	4.9	0.4
Q13	2.5	2.9	0.4
Q14	2.5	2.9	0.3
Q15	3.5	4.0	0.5
Q16	3.5	4.0	0.5
Q17	6.2	6.7	0.4
Q18	1.5	1.8	0.2
Q19	1.5	1.8	0.2
Q20	1.5	1.8	0.2
Q21	1.6	1.9	0.3
Q22	1.6	1.9	0.3
Q23	1.6	1.9	0.3

Q#	F# 3.0	P-LINQ	*norm*
Q24	1.8	2.0	0.3
Q25	1.4	1.6	0.2
Q27	1.8	2.1	0.2
Q29	1.5	1.7	0.2
Q30	1.8	2.0	0.2
Q32	2.7	3.1	0.3
Q33	2.8	3.1	0.3
Q34	3.1	3.6	0.5
Q35	3.1	3.6	0.4
Q36	2.2	2.4	0.2
Q37	1.3	1.6	0.2
Q38	4.2	4.9	0.6
Q39	4.2	4.7	0.4
Q40	4.1	4.6	0.4
Q41	6.3	7.3	0.6
Q42	4.7	5.5	0.5
Q43	7.2	6.9	0.7
Q44	5.4	6.2	0.7
Q45	2.2	2.6	0.3
Q46	2.3	2.7	0.4
Q47	2.1	2.5	0.3

Q#	F# 3.0	P-LINQ	*norm*
Q48	2.1	2.5	0.3
Q49	2.4	2.7	0.3
Q50	2.2	2.5	0.3
Q51	2.0	2.4	0.3
Q52	6.1	5.9	0.4
Q53	11.9	11.2	0.6
Q54	4.4	4.8	0.4
Q55	5.2	5.6	0.4
Q56	4.6	5.1	0.5
Q57	2.5	2.9	0.4
Q58	2.5	2.9	0.4
Q59	3.1	3.6	0.5
Q60	3.6	4.4	0.7
Q61	5.8	6.3	0.3
Q62	5.4	5.9	0.2
Q63	3.4	3.8	0.4
Q64	4.3	4.9	0.6
Q65	10.2	10.1	0.4
Q66	8.9	8.7	0.6
Q67	14.7	13.1	1.1

Table 2. Comparison of F# 3.0 and P-LINQ (using F# 3.0 as a back-end) on the 62 example database queries in the F# 3.0 documentation (Microsoft 2013). There are 67 examples in total; five query expressions (Q3, Q4, Q26, Q28, Q31) are excluded because they execute on in-memory lists rather than generate SQL.

reported by Grust et al. (2010), all of which return nested results. As guaranteed by Proposition 4, translation for P-LINQ always succeeds and never generates avalanches for queries with flat results.

Generally, normalisation time is dwarfed by query evaluation time, in some cases by several orders of magnitude. The F# type Expr< A > maintains information irrelevant to our application, so we elected to normalise by converting the F# type Expr< A > to our own custom representation, normalising that, and converting back to Expr< A >. Profiling suggests most of the time in our normaliser is spent converting to our custom representation. The only way to traverse Expr< A > expressions in F# is through active pattern matching, which appears to be expensive.

To evaluate the impact of query avalanches, we reran query (9) with F# 3.0 and P-LINQ with varying numbers of departments, ranging from 4 to 64. For F# 3.0, the number of queries performed is $d{+}1$ where d is the number of departments. The results are shown in Figure 14. Both approaches scale roughly linearly in the number of departments (and hence, total data size); we summarise the results in terms of the average time s to process each department. The value of s for F# 3.0 is 12.8 milliseconds per department, while that for P-LINQ is 0.3. These results confirm that P-LINQ's normalisation can reap significant savings by avoiding query avalanches.

T-LINQ does not include constructs such as sorting, grouping, or aggregation, which are important in practical use of LINQ. We have designed P-LINQ so that it rewrites any subterm it recognises, and carries through other constructs unchanged. Our results suggest this is a practical alternative: we tested this prototype on all 62 of the example database queries on the F# 3.0 Query Expressions documentation page (Microsoft 2013). (There are also five tests that do not generate SQL queries, which we excluded from the experiments.) All of these queries are concrete, that is, none involves abstraction, and they are evaluated on a small database of about 30 records. The results are shown in Table 2. We summarise the results in terms of the ratio r of P-LINQ to F# 3.0 evaluation time. The geometric mean of r over all tests is 1.13 (so on average P-LINQ is 13% slower), and the minimum and maximum values of r over all tests is 0.89 and 1.24, respectively (so at best P-LINQ is 11% faster and at worst 24% slower). All the translations succeeded, suggesting that normalisation does not interfere with F# 3.0's support for additional query operators. These results demonstrate that the overhead of normalisation is modest, even for small data sets, and occasionally normalisation improves query time, even for concrete queries.

At present, F# 3.0 does not allow overriding the default query builder, so we cannot yet provide our implementation as a drop-in replacement. We are discussing with the Microsoft F# team how best to make our techniques available in a future version of F#.

10. Related Work

LINQ has attracted considerable commercial interest, but has not been extensively documented in the research literature. Meijer et al. (2006) and Meijer (2011) give overviews of the foundations of LINQ. Syme (2006) presents an early version of F#'s quotation and reflection capabilities, illustrated via applications to LINQ, GPU code generation, and runtime F# code generation. Bierman et al. (2007) present a formalisation of several extensions to C#, including LINQ. Eini (2011) identifies obstacles to implementing LINQ providers for non-SQL databases. Beckman (2012) advocates LINQ as an interface to cloud computing platforms. Petricek and Syme (2012) and Syme et al. (2012) describe F# 3.0's sequence expressions and the related computation builder mechanism. Use of LINQ for abstraction over values and predicates and dynamic generation of queries has been discussed in blogs and online forums, such as Petricek (2007b,a), but has not, to our knowledge, previously been modelled formally.

Type-safe quotation and meta-programming is an active research area. Davies and Pfenning (2001) introduce a calculus $\lambda^{\Box}$ for closed multistage programming based on a modal logic, where each stage uses the same language. T-LINQ can be viewed as a variant with just two stages, each using a slightly different language. Rhiger (2012) presents a calculus for multistage programming with open quotations, noting that closed quotation leads to less efficient code due to administrative redexes. In our setting, such administrative redexes have negligible cost because we normalisation time is dominated by query execution time. Choi et al. (2011) present a translation from open to closed quotation similar to ours, aimed at supporting translation-based static analysis for staged computation. Van den Bussche et al. (2005) present a meta-querying system for SQL, but does not consider type safety or language integration.

Integrating queries into a general-purpose language is also an active research area. Ohori and Ueno (2011) introduces SML#, which offers direct support for SQL queries, including a type system that guarantees each query accesses only a single database. It does not normalise queries. Chlipala (2010) introduces Ur/Web, which uses open quotations with a sophisticated type system. It also does not normalise queries. Ur/Web can express most of the queries given here, though it relies on subqueries to express query composition, and it cannot express the nested query (9) of Section 3.2 (Adam Chlipala, personal communication, January 2013). Budiu et al. (2013) present a general framework for composing compilers based on LCF-style tactics, and apply this methodology to a subset of LINQ in C# called μLINQ. It would be interesting to see whether this approach can be used for T-LINQ.

Grust et al. (2009, 2010) describe Ferry, a functional query language that, like our work, supports higher-order functions and nested data, but goes beyond our work in also supporting queries that return nested results. The Ferry team have implemented several LINQ query providers, as well as interfacing Ferry with Links (Ulrich 2011) and Haskell (Giorgidze et al. 2010). Henglein and Larsen (2010) consider efficient in-memory evaluation of query-like constructs using lazy evaluation and generic discrimination. These approaches employ complementary techniques to our normalisation-based approach; combining our results with these systems appears possible, and should be explored in future work.

In our case the host language and quoted language are taken to be identical, but that is a design choice. An approach where host and quoted languages differ is described by Mainland (2012).

We indicate quotations syntactically (with `<@ ··· @>`). An alternative approach is to indicate quotation by type declaration, as is done in C# and with Lightweight Modular Staging in Scala (Rompf and Odersky 2012). Another alternative is not to support quotation in the language, but to use an embedded DSL to construct parse trees, an approach taken in Nikola (Mainland and Morrisett 2010) and Feldspar (Axelsson et al. 2010; Axelsson and Svenningsson 2012). We intend to team with Feldspar researchers to compare their approach with ours.

11. Conclusion

We presented a simple theory of language-integrated query based on quotation and normalisation. Through a series of examples, we demonstrated that our technique supports abstraction over values and predicates, composition of queries, dynamic generation of queries, and queries with nested intermediate data; and that higher-order features proved useful even for dynamic generation of first-order queries. We developed a formal theory, T-LINQ, and proved that normalisation always succeeds in translating any query of flat relation type to SQL. We presented an implementation in F# called P-LINQ, and experimental results confirming our technique works in practice as predicted. We observed that for several of our examples, Microsoft's LINQ framework either fails to produce an SQL query or produces an avalanche of SQL queries.

In essence, we have supplied a recipe for using a host language to generate code in a target language. The recipe involves three languages: the host language (in our case, F#), the target language (in our case, SQL), and a quoted language (in our case, essentially F# again).

- The host language should support quotation and anti-quotation of terms in the quoted language: in our case, we use F# quotation.
- The quoted language may need to add constructs not in the host language (so it is as expressive as the target language), and omit some constructs in the host language (so it is not more expressive than the target language after normalisation): in our case, the quoted language adds the **database** construct but omits recursion.
- The quoted language should at least support lambda abstraction and typing: support for lambda abstraction means it is sufficient to support closed quotations, which in turn makes it easier to support typing.
- Finally—and most importantly—one must identify an adequate normalisation procedure. Normalisation should at least perform beta-reduction: thus the quoted language may exploit the expressiveness of lambda abstraction even if the target language is first order. Normalisation may perform operations other than beta-reduction: in our case, additional rewrite rules support translation into SQL.

Applying the recipe to other domains is an important area for future work.

To conclude, let's compare the theoretical and practical aspects of our work. Regarding the host and quoted languages, the recipe above makes clear they may differ in theory, and this is the case in T-LINQ; in practice, they tend to be the same, and this is the case in P-LINQ. Regarding coverage, T-LINQ differs from full LINQ, and extending it to cover sorting, grouping, and aggregation remains important work for tomorrow. Meanwhile, P-LINQ supports the same features as LINQ, and it can be put to work today.

Our opening adage acknowledges that theory can fall short of the needs of practice. In contrast, our experience confirms that practice benefits from a judicious dose of theory, even when that theory is incomplete. We propose a reversal of the opening platitude.

What is the difference between theory and practice?
In theory there is a difference, but in practice there isn't.

Acknowledgments

We are grateful to Nada Amin, James McKinna, Tomas Petricek, Simon Peyton Jones, Tiark Rompf, and Don Syme for comments on this work. The second author would like to thank Conor McBride for the craic. This work is supported in part by a Google Research Award (Lindley), EPSRC grants EP/J014591/1 (Lindley) and EP/K034413/1 (Lindley, Wadler), and a Royal Society University Research Fellowship (Cheney).

References

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

E. Axelsson and J. Svenningsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.

E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar—an embedded language for digital signal processing. In J. Hage and M. T. Morazán, editors, *IFL*, volume 6647 of *LNCS*, pages 121–136. Springer, 2010.

B. Beckman. Why LINQ matters: cloud composability guaranteed. *Commun. ACM*, 55(4):38–44, Apr. 2012.

G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *OOPSLA*, pages 479–498. ACM, 2007.

M. Budiu, J. Galenson, and G. D. Plotkin. The compiler forest. In *ESOP*, number 7792 in LNCS, pages 21–40. Springer-Verlag, 2013.

P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23:87–96, 1994.

J. Cheney and R. Hinze. First-class phantom types. Computer and Information Science Technical Report TR2003-1901, Cornell University, 2003.

J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query (code supplement), 2012.
`http://homepages.inf.ed.ac.uk/jcheney/linq`.

A. J. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, pages 122–133. ACM, 2010.

W. Choi, B. Aktemur, K. Yi, and M. Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *POPL*, pages 81–92. ACM, 2011.

E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, number 5708 in LNCS, pages 36–51. Springer-Verlag, 2009.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296, 2007.

G. Copeland and D. Maier. Making Smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, 1984.

R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

O. Eini. The pain of implementing LINQ providers. *Commun. ACM*, 54(8): 55–61, 2011.

G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the Ferry - database-supported program execution for Haskell. In *IFL*, number 6647 in LNCS, pages 1–18. Springer-Verlag, 2010.

T. Goldschmidt, R. Reussner, and J. Winzen. A case study evaluation of maintainability and performance of persistency techniques. In *ICSE*, pages 401–410. ACM, 2008.

T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *SIGMOD*, pages 1063–1066. ACM, 2009.

T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1):162–172, 2010.

F. Henglein and K. F. Larsen. Generic multiset programming with discrimination-based joins and symbolic cartesian products. *Higher-Order and Symbolic Computation*, 23(3):337–370, 2010.

S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*, pages 91–102. ACM, 2012.

G. Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In P. Thiemann and R. B. Findler, editors, *ICFP*, pages 311–322. ACM, 2012. ISBN 978-1-4503-1054-3.

G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in haskell. In J. Gibbons, editor, *Haskell Symposium*, pages 67–78. ACM, 2010. ISBN 978-1-4503-0252-4.

E. Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, Oct. 2011.

E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, page 706. ACM, 2006.

Microsoft. Query expressions (F# 3.0 documentation), 2013. `http://msdn.microsoft.com/en-us/library/vstudio/-hh225374.aspx`, accessed March 18, 2013.

A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *ICFP*, pages 307–319. ACM, 2011.

T. Petricek. Building LINQ queries at runtime in (F#), 2007a.
`http://tomasp.net/blog/dynamic-flinq.aspx`.

T. Petricek. Building LINQ queries at runtime in (C#), 2007b.
`http://tomasp.net/blog/dynamic-linq-queries.aspx`.

T. Petricek and D. Syme. Syntax Matters: Writing abstract computations in F#. Pre-proceedings of TFP, 2012.
`http://www.cl.cam.ac.uk/~tp322/drafts/notations.pdf`.

D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almqvist and Wiksell, Stockholm, 1965.

M. Rhiger. Staged computation with staged lexical scope. In *ESOP*, number 7211 in LNCS, pages 559–578. Springer-Verlag, 2012.

T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML Workshop*, pages 43–54. ACM, 2006.

D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Apress, 2012. ISBN 978-1-4302-4650-3.

W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

P. Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages*, pages 49–62. Morgan Kaufmann, 1991.

P. Trinder and P. Wadler. Improving list comprehension database queries. In *TENCON '89.*, 1989.

A. Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011.

J. Van den Bussche, S. Vansummeren, and G. Vossen. Towards practical meta-querying. *Inf. Syst.*, 30(4):317–332, 2005.

L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.