The essence of language-integrated query

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

Sam Lindley
The University of Strathclyde
sam.lindley@strath.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 simple theory of language-integrated query based on quotation and normalisation of quoted terms. Our technique supports abstraction over queries, dynamic generation of queries, and queries with nested intermediate data. Higher-order features prove useful even for dynamic generation of first-order queries. We prove that normalisation always succeeds in translating any query of flat relation type to SQL. 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.

1. INTRODUCTION

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).

The problem is simple: two languages are more than twice as difficult to use as one language. Most programming languages support nested data and data abstraction, while most relational databases only support flat tables over concrete data. Any task involving both requires that the programmer keep in mind two representations of the same underlying data, converting between them and synchronising updates to either. This pervasive bookkeeping adds to the mental burden on the programmer and leads to complex code, bugs, and security holes such as SQL injection attacks.

Most database developers work in two languages. Wrapper libraries, such as JDBC, provide raw access to high-performance SQL, but the resulting code is difficult to maintain. Object-Relational Mapping (ORM) 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). Workarounds to recover performance, such as framework-specific query languages, reintroduce the drawbacks of the two-language approach.

We present a simple theory of language-integrated query based on quotation and normalisation of quoted terms, called *Idealised LINQ*. Our technique supports abstraction over queries, dynamic generation of queries, and queries with nested intermediate data. Higher-order features prove useful even for dynamic generation of first-order queries. We prove that normalisation always succeeds in translating any query of flat relation type to a single SQL query, avoiding query avalanches in the sense of Grust et al. (2010), where the number of queries depends on the database size.

Microsoft LINQ was released as a part of .NET Framework 3.5 in November 2007, and LINQ continues to evolve with new releases. LINQ translates query expressions in the source language into queries in a target language such as SQL or XQuery. In this paper, we focus on SQL, though we believe the ideas may adapt to other targets. We often write Microsoft LINQ as shorthand to stand for Microsoft's LINQ to SQL provider, which targets Microsoft SQL Server.

There are variants of LINQ for C#, Visual Basic, and F#, among others. Idealised LINQ corresponds most closely to LINQ for F#. We choose F# as a basis because it supports the features we require: records and lists, comprehension notation for list processing, and quotation and anti-quotation. Idealised LINQ can easily be adapted to any language with these features. For instance, we believe similar ideas could be adapted to C#, though more clumsily. C# supports a form of quotation (a lambda-term is treated as quoted in some contexts), but has poor support for anti-quotation, so one may be forced to manipulate the C# Expression type directly (Petricek 2007b).

Our concern is not with the myriad details of a specific framework, but with presenting a simple theory of the key underlying ideas. Nonetheless, our theory can be applied directly to Microsoft LINQ, as we demonstrate by implementing our normalisation algorithm in F# as a preprocessing step to the F# PowerPack LINQ implementation. 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,

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

```
{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

produces an avalanche of SQL queries. Our experiments revealed bugs and limitations in the F# LINQ library, which we have communicated to the F# team.

SQL corresponds closely to the comprehension notation found in many functional languages, as noted by Trinder and Wadler (1989) and Buneman et al. (1994). Conservativity results for nested relational algebra prove that if a comprehension term has a flat type, then it has a normal form that corresponds directly to SQL, as shown by Wong (1996). The normalisation rules we present are based on those formulated for our web programming language, Links (Cooper 2009, Lindley and Cheney 2012).

While Microsoft's LINQ is best explained in terms of quotation, our own Links system is better explained in terms of an effect-based theory. Both depend on normalisation. One motivation behind this paper is to better establish the theory of quotation-based language-integrated query, as a preliminary to describing the relation between the quotation-based and effect-based approaches.

To focus on the central ideas, we limit our attention to conjunctive queries. We ignore features such as grouping and aggregation, and we only run flat queries, those that return a list of records of scalars. Related results for grouping and aggregation have been studied by Libkin and Wong (1997) and Grust et al. (2009), and queries with nested results have been studied by (Grust et al. 2010).

The title of this paper and the name Idealised LINQ tip a hat to Reynolds (1981).

The remainder of this paper is organised as follows. Section 2 introduces our approach to language-integrated query. Section 3 considers nesting. Section 4 applies the approach to a relational representation of XML trees. Section 5 formalises our work, giving a complete description of the necessary reduction rules. Section 6 compares Idealised LINQ to Microsoft's LINQ. Section 7 describes our implementation and gives our results. Section 8 explains why it is preferable to express queries as quotations rather than as functions. Section 9 discusses related work. Section 10 concludes.

2. FUNDAMENTALS

We consider a simplified programming language, based loosely on F# (Syme et al. 2012), featuring lists and records and sequence comprehensions. 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. 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 may be represented in Idealised LINQ as a record of tables, 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}
```

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.

We imagine augmenting our language with a construct that takes the name of the database and returns its content as the corresponding 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 over lists offering operations analogous to those provided by SQL over tables (Trinder and Wadler 1989, Buneman et al. 1994). In Idealised LINQ the analogue of the previous SQL query is 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

$$[\{\mathsf{name} = \text{“Alex”};\ \mathsf{diff} = 5\}; \{\mathsf{name} = \text{“Cora”};\ \mathsf{diff} = 2\}]$$

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. In-memory evaluation of a comprehension may not take advantage of indexing, so may require $\Theta(|\mathsf{couples}| \cdot |\mathsf{people}|^2)$ time. 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 Idealised LINQ term

$$\textbf{for } x \textbf{ in } L \textbf{ do for } y \textbf{ in } M \textbf{ do if } P \textbf{ then yield } N$$

is equivalent to the mathematical notation

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

or the F# sequence expression

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

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

2.2 Query via quotation

Idealised LINQ allows programmers to access databases using a notation almost identical to the naive approach of the previous section, but generating efficient queries in an appropriate query language such as 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 in anti-quotation brackets; in this case the only externally bound variable is db.

```
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)
```

This 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 data. 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 may now drop the warning primes, because the answer is computed feasibly by access to the database.

The notation <@ ··· @> indicates quotation, which converts an expression of type A into a data structure of type Expr< A > that represents the expression as an abstract syntax tree. The notation (% ···) indicates anti-quotation, which splices a quoted expression of type Expr< A > inside a quoted expression at a point expecting a value 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 infeasible for very large databases. Query evaluation, indicated by the keyword **run**, takes an expression of type Expr< A >, normalises the expression, translates the normalised expression to SQL, evaluates the SQL, and imports the result as a value of type A. The type A must correspond to an SQL table type, that is, a list of rows, where each row is a record of scalars.

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 file I/O or recursion. (Technically SQL does support some forms of recursion, such as transitive closure, but current LINQ systems do not.) In Idealised LINQ, the first condition is dynamically checked, and the other two are statically checked. In Microsoft LINQ, all checks are dynamic.

Idealised 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 specifics of LINQ in Section 7.

2.3 Abstracting over values

An advantage of language-integrated query is that one may exploit the abstraction mechanisms of the host language to formulate queries. Let's begin with 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} @>
```

To keep things simple, 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.

As before, we have defined the query as a quoted term, this time one that contains a function abstraction. We shall see that this is essential to being able to reuse queries flexibly in constructing other queries.

Here we use the usual F# notation for function abstraction. Function applications normalise by substitution:

$$(\textbf{fun}(\bar{x}) \to N)(\bar{M}) \rightsquigarrow N[\overline{x := M}],$$

where $N[\overline{x := M}]$ denotes the capture-avoiding substitution of terms $\bar{M}$ for variables $\bar{x}$ in term N. Theorists write $\mathbf{fun}(\bar{x}) \to N$ as $\lambda\bar{x}.N$, and call it a lambda-abstraction, and they call the rewriting rule above beta-reduction.

We form a specific query by instantiating the parameters:

$$\mathbf{run}(\texttt{<@ (\%range)(30, 40) @>}) \tag{2}$$

Evaluating (2) finds everyone in their thirties:

$$[\{\mathsf{name} = \text{“Cora”}\};\ \{\mathsf{name} = \text{“Drew”}\}]$$

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.

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} @>
```

Predicates over ages are denoted by functions from integers to booleans. We form a specific query by instantiating the predicate. Evaluating

$$\mathbf{run}(\texttt{<@ (\%satisfies)(}\mathbf{fun}\texttt{(x) → 30 ≤ x \&\& x < 40) @>}) \tag{3}$$

is equivalent to the previous example, (2). In this case, the term passed to **run** requires two steps of beta-reduction to normalise. The first replaces p by the function, and enables the second, which replaces x by w.age.

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

$$\mathbf{run}(\texttt{<@ (\%satisfies)(}\mathbf{fun}\texttt{(x) → x mod 2 = 0) @>}) \tag{4}$$

finds everyone whose age is even. It would not work if, say, the predicate invoked recursion. For Idealised LINQ, we statically check that quoted terms can be translated to SQL; in Microsoft LINQ, query translation fails at run-time on quotations containing operations with no SQL equivalent.

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 ageFromName : Expr< string → int> =
  <@ 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 ageFromName with the query range.

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

We form a specific query by instantiating the parameter.

$$\mathbf{run}(\texttt{<@ (\%nameRange)(“Edna”, “Bert”) @>}) \tag{5}$$

This yields the value

$$[\{\mathsf{name} = \text{“Cora”}\};\ \{\mathsf{name} = \text{“Drew”}\};\ \{\mathsf{name} = \text{“Edna”}\}]$$

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. Assume our program declares the following algebraic type, the values of which are trees representing predicates over integers.

```
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)))
```

We can write a function that given a tree representing a predicate returns the quotation of a function representing the predicate. We will 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 @>.
```

Again, notice how normalisation is necessary if we are to write P using closed rather than open quotation.

We can combine P with the previously defined satisfies to find all people that satisfy a given predicate. For example,

$$\mathbf{run}(\texttt{<@ (\%satisfies)(\%P(t0)) @>}) \tag{6}$$

is equivalent to the previous examples, (2) and (3). It generates the same SQL and yields the same answer. Evaluating

$$\mathbf{run}(\texttt{<@ (\%satisfies)(\%P(t1)) @>}) \tag{7}$$

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

An important point of this paper is that incorporating a normalisation step means it is straightforward to generate dynamic queries.

```
{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

3. NESTING

In this section, we consider the application of language-integrated query to nested data, and show further advantages of the use of normalisation before execution of a query.

For purposes of illustration, we consider a simplified database representing an organisation, with tables for departments, employees, and tasks. 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, called "Org".

```
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"}]
```

```
[{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

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

Query expertise′ works as follows. The innermost **for** returns an (empty) record for each task t performed by employee e that is equal to u; it will contain no elements if employee e cannot perform task u. The middle **for** returns an (empty) record for each employee e in department d that cannot perform task u; it will contain no elements if every employee in department d can perform task u. Therefore, the outermost **for** returns departments where every employee can perform task u.

We stick a prime on the name to warn that this query is hard to read. Using nested 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]}]}]
```

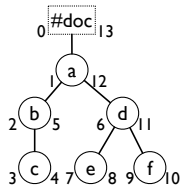
We compute the latter from the former as follows:

```
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 the above binds nestedOrg to the data in Figure 4. We cannot write **run**(nestedOrg) to compute this value directly, because **run** requires an argument of type Expr< A >, where A is a list of records of scalars, and the type of nestedOrg is a list of records of lists of records of scalars. However, it can be convenient to use nestedOrg to formulate other queries, as we show in the next section.

3.2 Higher-order queries

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



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

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) @>
```

Here we have used the trick of *currying*, standard from functional programming, where a function from two arguments to a result is expressed as a function from the first argument to a function from the second argument to the result. 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)
```

is equivalent to the previous example, (8). It generates the same SQL and yields the same answer.

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

As a final example of the power of our approach to type-safe dynamic query generation, we consider an implementation of tree-structured XML data in a relation using

```
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.pre < 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)
  | NameTest(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< Node list > =
  <@ for root in (%db).xml do
     for s in (%db).xml do
     if root.parent = −1 && (%path(p))(root, s) then
     yield s @>
```

Figure 6: An evaluator for XPath

"stretched" pre-order and post-order indexes; see, for example, Grust et al. (2004; sec. 4.2). Range indexing on the pre and post fields ensures efficient execution of a wide range of XPath queries (Grust et al. 2004). Each node of the tree corresponds to a row in a table db.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 if 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                 | NameTest of string
  | DescendantOrSelf           | Filter of Path
  | Following
  | FollowingSibling
  | Rev of Axis
```

In the Axis datatype, we define the primitive forward axes and the Rev case handles the reverse steps (parent, ancestor, ancestor-or-self, preceding, and preceding-sibling). In the Path datatype, Seq concatenates two paths, Axis defines an axis step, NameTest tests whether an element's name is equal to a given string, and Filter tests whether a path expression is satisfiable from a given node.

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

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

$$\textsc{App}\ \frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B} \qquad \textsc{Record}\ \frac{\overline{\Gamma \vdash M : A}}{\Gamma \vdash \{\overline{\ell = M}\} : \{\overline{\ell : A}\}} \qquad \textsc{Project}\ \frac{\Gamma \vdash M : \{\overline{\ell : A}\}}{\Gamma \vdash M.\ell_i : A_i} \qquad \textsc{Singleton}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{yield}\ M : A\ \mathbf{list}}$$

$$\textsc{For}\ \frac{\Gamma \vdash M : A\ \mathbf{list} \qquad \Gamma, x : A \vdash N : B\ \mathbf{list}}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N : B\ \mathbf{list}} \qquad \textsc{If}\ \frac{\Gamma \vdash L : \mathbf{bool} \qquad \Gamma \vdash M : A\ \mathbf{list}}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M : A\ \mathbf{list}} \qquad \textsc{Exists}\ \frac{\Gamma \vdash M : A\ \mathbf{list}}{\Gamma \vdash \mathbf{exists}\ M : \mathbf{bool}} \qquad \textsc{Empty}\ \frac{}{\Gamma \vdash \texttt{[]} : A\ \mathbf{list}}$$

$$\textsc{Union}\ \frac{\Gamma \vdash M : A\ \mathbf{list} \qquad \Gamma \vdash N : A\ \mathbf{list}}{\Gamma \vdash M\ \texttt{@}\ N : A\ \mathbf{list}} \qquad \textsc{Rec}\ \frac{\Gamma, f : A \to B, x : A \vdash N : B}{\Gamma \vdash \mathbf{rec}\ f(x) \to N : A \to B} \qquad \textsc{Run}\ \frac{\Gamma \vdash M : \mathsf{Expr}\langle T \rangle}{\Gamma \vdash \mathbf{run}\ M : T} \qquad \textsc{Quote}\ \frac{\Gamma; \cdot \vdash M : A}{\Gamma \vdash \texttt{<@}\ M\ \texttt{@>} : \mathsf{Expr}\langle A \rangle}$$

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

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

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

$$\textsc{ForQ}\ \frac{\Gamma; \Delta \vdash M : A\ \mathbf{list} \qquad \Gamma; \Delta, x : A \vdash N : B\ \mathbf{list}}{\Gamma; \Delta \vdash \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N : B\ \mathbf{list}} \qquad \textsc{IfQ}\ \frac{\Gamma; \Delta \vdash L : \mathbf{bool} \qquad \Gamma; \Delta \vdash M : A\ \mathbf{list}}{\Gamma; \Delta \vdash \mathbf{if}\ L\ \mathbf{then}\ M : A\ \mathbf{list}} \qquad \textsc{ExistsQ}\ \frac{\Gamma; \Delta \vdash M : A\ \mathbf{list}}{\Gamma; \Delta \vdash \mathbf{exists}\ M : \mathbf{bool}}$$

$$\textsc{EmptyQ}\ \frac{}{\Gamma; \Delta \vdash \texttt{[]} : A\ \mathbf{list}} \qquad \textsc{UnionQ}\ \frac{\Gamma; \Delta \vdash M : A\ \mathbf{list} \qquad \Gamma; \Delta \vdash N : A\ \mathbf{list}}{\Gamma; \Delta \vdash M\ \texttt{@}\ N : A\ \mathbf{list}} \qquad \textsc{Database}\ \frac{\Sigma(\mathsf{db}) = \{\overline{\ell : T}\}}{\Gamma; \Delta \vdash \mathbf{database}(\mathsf{db}) : \{\overline{\ell : T}\}} \qquad \textsc{Antiquote}\ \frac{\Gamma \vdash M : \mathsf{Expr}\langle A \rangle}{\Gamma; \Delta \vdash \texttt{(\%}M\texttt{)} : A}$$

Figure 7: Typing rules for Idealised LINQ

Figure 6 gives the complete code of an evaluator for this fragment of XPath, which generates one SQL query per XPath query. For each axis step, the predicate axis matches pairs of rows in the data table if and only if the corresponding rows are related by the axis. It is then straightforward to define the translation path from path expressions to predicates.

Finally, the xpath function evaluates a given path expression starting from the root. It is straightforward to translate conventional XPath strings to Path expressions, so that we can run the following queries:

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

yielding results $\{2, 4\}$, $\{1, 2, 4\}$, $\{2\}$ and $\{6\}$ respectively.

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

5. CORE LANGUAGE

In this section we give a formal account of Idealised LINQ as a higher-order nested relational calculus over bags augmented with a quotation mechanism for issuing flat relational queries to an SQL database.

The types of Idealised LINQ are as follows:

$$\begin{array}{lll} \text{(base type)} & O & ::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{string} \\ \text{(type)} & A, B & ::= O \mid A \to B \mid \{\overline{\ell : A}\} \mid A\ \mathbf{list} \mid \mathsf{Expr}\langle A \rangle \\ \text{(table type)} & T & ::= \{\overline{\ell : O}\}\ \mathbf{list} \end{array}$$

The terms of Idealised LINQ are as follows:

$$\begin{array}{rcl} L, M, N & ::= & c \mid op(\overline{M}) \mid \mathbf{lift}(M) \mid x \mid \mathbf{fun}(x) \to N \mid L\ M \\ & \mid & \{\overline{\ell = M}\} \mid M.\ell \mid \mathbf{yield}\ M \mid \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N \\ & \mid & \mathbf{if}\ L\ \mathbf{then}\ M \mid \mathbf{exists}\ M \mid \texttt{[]} \mid M\ \texttt{@}\ N \\ & \mid & \mathbf{rec}\ f(x) \to N \\ & \mid & \mathbf{run}\ M \mid \texttt{<@}\ M\ \texttt{@>} \mid \mathbf{database}(\mathsf{db}) \mid \texttt{(\%}M\texttt{)} \end{array}$$

For convenience, we use F# list-notation, but we are not concerned with the order of elements. For instance, we treat @ as bag union. In F# values of base type are implicitly coerced to quoted terms when referenced inside quoted terms.

For our core language we make this coercion explicit, writing **lift**(M) for the operation that lifts a host term M of base type to a quoted term of type Expr< M >. We write **rec** f(x) $\to N$ for a recursive function definition. This is equivalent to writing **let rec** f(x) $= N$ **in** f in F#.

For simplicity, we assume that all queries are on a single database db. In practice, one can check either dynamically or statically (Ohori and Ueno 2011) to ensure that a given query only refers to a single database. We do not include a general **if then else** construct, as doing so slightly complicates the normalisation procedure (Lindley and Cheney 2012). Nevertheless, it is straightforward to implement, and is supported in our implementation. Similarly, we do not include polymorphism in the formalisation as it is an orthogonal feature, but we get it for free in our implementation.

Type environments are as follows:

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A$$

The typing rules are given in Figure 7. There are two typing judgements: one for host terms, and the other for query terms. The judgement $\Gamma \vdash M : A$ states that host term M has type A in type environment Γ. The judgement $\Gamma; \Delta \vdash M : A$ states that query term M has type A in host type environment Γ and query type environment Δ.

Most of the typing rules are standard and mirrored across both judgements. The core rules for the query judgement do not use the Γ environment. The interesting rules are those that involve quotation. A query term can be quoted (Quote). A quoted term of flat relation type can be evaluated as a query (Run). A database term can be used inside a query term (Database). A quoted term can be spliced into a query term (Antiquote). A term of base type O can be lifted to a quoted term of type Expr< O > (Lift). Recursion is only available in a host term (Rec).

We assume a signature Σ that maps each constant c to its underlying type, each primitive operator op to its type (e.g. $\Sigma(\texttt{\&\&}) = (\mathbf{bool}, \mathbf{bool}) \to \mathbf{bool}$ and $\Sigma(+) = (\mathbf{int}, \mathbf{int}) \to \mathbf{int}$), and the database db to its type, a record of flat relation types with a field for the type of each table.

5.1 Operational semantics

We now present a small-step operational semantics for Idealised LINQ. The values are as follows:

$$\begin{array}{rcl} V, W & ::= & c \mid \mathbf{fun}(x) \to M \mid \mathbf{rec}\ f(x) \to M \\ & \mid & \{\overline{\ell = V}\} \mid [V_1, \ldots, V_n] \mid \texttt{<@}\ U\ \texttt{@>} \\ U & ::= & c \mid op(\overline{U}) \mid x \mid \mathbf{fun}(x) \to U \mid U\ U' \\ & \mid & \{\overline{\ell = U}\} \mid U.\ell \mid \mathbf{yield}\ U \mid \mathbf{for}\ x\ \mathbf{in}\ U\ \mathbf{do}\ U' \\ & \mid & \mathbf{if}\ U\ \mathbf{then}\ U' \mid [\,] \mid U \texttt{@} U' \mid \mathbf{database}(\texttt{db}).\ell \end{array}$$

They are mostly standard, except for quotation values <@ U @>, which depend on an auxiliary class of *query values* ranged over by U. These are just quoted terms in which all anti-quoting has been resolved.

We quotient bag values by bag equivalence, writing $[V_1, \ldots, V_n]$ for any term equivalent to $[\,] \texttt{@} \mathbf{yield}\ V_1 \texttt{@} \ldots \texttt{@} \mathbf{yield}\ V_n$ modulo identity, associativity and commutativity on bag union (@).

The semantics is parameterised by an interpretation DB that maps each database db in the schema Σ to its underlying data of type $\Sigma(db)$ and an interpretation δ that defines the semantics of each primitive operation op, such that:

$$\frac{\Sigma(op) = (\overline{O}) \to O \qquad \vdash \overline{V : O}}{\vdash \delta(op, \overline{V}) : O}$$

The rules are given in Figure 8.

The semantics is standard apart from the rules for quotation and query evaluation.

Evaluation can take place at the top-level or inside *evaluation contexts* ($\mathcal{E}$), which enforce left-to-right call-by-value evaluation, except inside quoted terms.

$$\begin{array}{rcl} \mathcal{E} & ::= & [\,] \mid op(\overline{V}, \mathcal{E}, \overline{M}) \mid \mathcal{E}\ M \mid V\ \mathcal{E} \mid \mathcal{E}.\ell \\ & \mid & \mathbf{for}\ x\ \mathbf{in}\ \mathcal{E}\ \mathbf{do}\ N \mid \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ M \mid \mathcal{E} \texttt{@} M \mid V \texttt{@} \mathcal{E} \\ & \mid & \mathbf{run}\ \mathcal{E} \mid \texttt{<@}\ \mathcal{Q}[\texttt{(\%}\mathcal{E}\texttt{)}]\ \texttt{@>} \mid \mathbf{lift}([\,]) \end{array}$$

$$\begin{array}{rcl} \mathcal{Q} & ::= & [\,] \mid op(\overline{M}, \mathcal{Q}, \overline{N}) \mid \mathbf{fun}(x) \to \mathcal{Q} \mid \mathcal{Q}\ N \mid M\ \mathcal{Q} \\ & \mid & \{\overline{\ell' = M}, \ell = \mathcal{Q}, \overline{\ell'' = N}\} \mid \mathcal{Q}.\ell \\ & \mid & \mathbf{yield}\ \mathcal{Q} \mid \mathbf{for}\ x\ \mathbf{in}\ \mathcal{Q}\ \mathbf{do}\ N \mid \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ \mathcal{Q} \\ & \mid & \mathbf{if}\ \mathcal{Q}\ \mathbf{then}\ M \mid \mathbf{if}\ L\ \mathbf{then}\ \mathcal{Q} \mid \mathcal{Q} \texttt{@} M \mid N \texttt{@} \mathcal{Q} \\ & \mid & \mathbf{exists}\ \mathcal{Q} \end{array}$$

Query contexts ($\mathcal{Q}$), are one-hole contexts over query terms that are free from anti-quotation. They allow evaluation to proceed inside anti-quotes. The (splice) rule resolves an anti-quote once its body has been evaluated to a quotation. The (lift) rule converts a constant into a quoted constant.

The idea underlying the (run) rule is that a query M of flat relation type is evaluated by first *normalising* M to yield an equivalent query Q, and then evaluating Q with respect to the database. The important feature of the language of normal forms is that it is isomorphic to a subset of SQL. Thus Q is evaluated simply by issuing an SQL query to the database, which we model abstractly by $eval_{DB}(Q)$. The only assumptions we make here are that $eval_{DB}$ is total on normalised queries and for any <@ Q @> of type Expr< T >, $eval_{DB}(Q)$ returns a result V of type T.

The property that guarantees that M can be normalised to a flat query is that M has flat relation type. The key idea of using normalisation to convert higher-order queries into SQL was formalised by Cooper (2009) building on ideas of

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

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

Figure 8: Operational semantics for Idealised LINQ

$$
\begin{array}{rcl}
(\mathbf{fun}(x) \to N)\ M & \rightsquigarrow & N[x := M] \\
\{\overline{\ell = M}\}.\ell_i & \rightsquigarrow & M_i \\
\mathbf{for}\ x\ \mathbf{in}\ (\mathbf{yield}\ M)\ \mathbf{do}\ N & \rightsquigarrow & N[x := M] \quad \text{(for-yld)} \\
\mathbf{for}\ y\ \mathbf{in}\ (\mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ M)\ \mathbf{do}\ N & \rightsquigarrow & \\
\quad \mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ (\mathbf{for}\ y\ \mathbf{in}\ M\ \mathbf{do}\ N) & & \text{(for-for)} \\
\mathbf{for}\ x\ \mathbf{in}\ (\mathbf{if}\ L\ \mathbf{then}\ M)\ \mathbf{do}\ N & \rightsquigarrow & \\
\quad \mathbf{if}\ L\ \mathbf{then}\ (\mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N) & & \text{(for-if)} \\
\mathbf{for}\ x\ \mathbf{in}\ [\,]\ \mathbf{do}\ N & \rightsquigarrow & [\,] \\
\mathbf{for}\ x\ \mathbf{in}\ (L \mathbin{@} M)\ \mathbf{do}\ N & \rightsquigarrow & \\
\quad (\mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ N) \mathbin{@} (\mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N) & & \\
\mathbf{if\ true\ then}\ M & \rightsquigarrow & M \\
\mathbf{if\ false\ then}\ M & \rightsquigarrow & [\,]
\end{array}
$$

$$
\frac{M \rightsquigarrow N}{\mathcal{Q}[M] \rightsquigarrow \mathcal{Q}[N]}
$$

Figure 9: Normalisation stage 1: symbolic reduction

$$
\begin{array}{rcl}
\mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ (M \mathbin{@} N) & \hookrightarrow & \\
\quad (\mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ M) \mathbin{@} (\mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ N) & & \\
\mathbf{for}\ x\ \mathbf{in}\ L\ \mathbf{do}\ [\,] & \hookrightarrow & [\,] \\
\mathbf{if}\ L\ \mathbf{then}\ (M \mathbin{@} N) & \hookrightarrow & \\
\quad (\mathbf{if}\ L\ \mathbf{then}\ M) \mathbin{@} (\mathbf{if}\ L\ \mathbf{then}\ N) & & \\
\mathbf{if}\ L\ \mathbf{then}\ [\,] & \hookrightarrow & [\,] \\
\mathbf{if}\ L\ \mathbf{then}\ (\mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N) & \hookrightarrow & \\
\quad \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ (\mathbf{if}\ L\ \mathbf{then}\ N) & & \text{(if-for)} \\
\mathbf{if}\ L\ \mathbf{then}\ (\mathbf{if}\ M\ \mathbf{then}\ N) & \hookrightarrow & \\
\quad \mathbf{if}\ (L\ \&\&\ M)\ \mathbf{then}\ N & & \text{(if-if)} \\
\mathbf{yield}\ x & \hookrightarrow & \mathbf{yield}\ \{\overline{\ell = x.\ell}\} \\
\mathbf{database}(\mathsf{db}).\ell & \hookrightarrow & \\
\quad \mathbf{for}\ x\ \mathbf{in}\ \mathbf{database}(\mathsf{db}).\ell\ \mathbf{do}\ \mathbf{yield}\ x & &
\end{array}
$$

Figure 10: Normalisation stage 2: ad hoc reduction

Wong (1996). The particular algorithm we present here is based on our earlier work (Lindley and Cheney 2012).

An important property from the latter paper is that the semantics is invariant with respect to normalisation, that is, if we replace $eval_{DB}$ by a function that loads all of the DB database into memory before running the query in memory, then the observable behaviour is unchanged.

5.2 Query normalisation

We define the query normalisation function *norm* in two stages. The first stage normalises M with respect to the rewrite relation $\rightsquigarrow$ defined in Figure 9, which performs standard symbolic reduction. Note the similarity with the evaluation relation $\longrightarrow$ of Figure 8. The effect of the first stage is to eliminate all higher-order and nested sub-terms. This is guaranteed by M having flat relation type (Cooper 2009).

The second stage translates a flat query returned by the first stage into a form isomorphic to a subset of SQL. The relation $\hookrightarrow$ is defined by taking the compatible closure of all of the rules in Figure 10 but the last, which must only be applied at the top level or immediately inside an `@` operation. The second stage can be viewed as a way of accounting for flaws in the syntax of SQL. The first rule, which hoists a union out of a comprehension body, is the only rule that is not sound for a list semantics. It is unsound precisely because it changes the order of the generated elements.

To define *norm*, we combine both reduction relations: if L is a closed term of flat relation type, $L \rightsquigarrow$-normalises to M, and $M \hookrightarrow$-normalises to N, then $norm(L) = N$. Termination of normalisation can be proved as in Cooper (2009) or Lindley and Cheney (2012). The resulting normal forms are given by the following grammar:

$$
\begin{array}{lrcl}
\text{(normal form)} & Q & ::= & [\,] \mid C \mid C \mathbin{@} D \\
\text{(clause)} & C, D & ::= & \mathbf{yield}\ R \mid \mathbf{if}\ X\ \mathbf{then\ yield}\ R \\
 & & \mid & \mathbf{for}\ x\ \mathbf{in\ database}(\mathsf{db}).\ell\ \mathbf{do}\ C \\
\text{(record)} & R & ::= & \{\overline{\ell = X}\} \\
\text{(base term)} & X & ::= & c \mid x.\ell \mid op(\overline{X}) \mid \mathbf{exists}\ Q
\end{array}
$$

Strictly speaking, this is not quite isomorphic to a subset of SQL as SQL cannot handle rows with no fields, or empty unions. These idiosyncrasies are easy to work around in practice.

5.3 Type soundness

Type soundness consists of two properties: *preservation* states that the evaluation relation preserves well-typing; and *progress* states that if a term is well-typed, then either it is a value or an evaluation rule applies.

Proposition 1 (Preservation).
- *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow N$ *then* $\Gamma \vdash N : A$.
- *If* $\Gamma; \Delta \vdash M : A$ *and* $M \longrightarrow N$ *then* $\Gamma; \Delta \vdash N : A$.

Proposition 2 (Progress). *If* $\vdash M : A$, *then either* M *is a value, or there exists a term* N *such that* $M \longrightarrow N$.

The main result is that Idealised LINQ programs never encounter run-time errors while generating queries: each **run** expression generates a single, type-safe SQL query, avoiding query avalanches where the number of queries depends on the database size.

5.4 An example

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

```
run(<@ (%nameRange)("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 9), 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 10), 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. COMPARISON TO MICROSOFT LINQ

Our model abstracts from several distracting issues in the LINQ implementation. In this section we compare Idealised LINQ to Microsoft's LINQ features in C# and F#, supported by the LINQ to SQL query provider library.

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

$$\textbf{from } x \textbf{ in } e \textbf{ where } p(x) \textbf{ select } f(x)$$

translates to the sequence of calls

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

Depending on the context, lambda-abstractions 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 C#'s Expression<A> type. Eini (2011) characterises writing a custom query provider as "doom, gloom with just a tad of despair".

Microsoft supplies a LINQ to SQL query provider that maps to SQL Server syntax. Microsoft's query provider is proprietary, so its behaviour is a black box, but it does appear to perform some normalisation. If the query expression involves nesting, however, Microsoft's provider can suffer from what Grust et al. (2010) call a "query avalanche"—where the number of queries generated is proportional to the size of the data. As we shall see, this can happen even for queries such as expertise whose result is flat.

As we have already described, F# supports LINQ using syntactic sugar for comprehensions (called "computation expressions"), 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 translation also performs some beta-reduction, but it is complex and incomplete. In particular, it fails to translate occurrences of **exists** in the test of a conditional, or occurrences of **yield** outside of a **for** expression.

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**$\{\cdots\}$ to indicate that a computation expression should be interpreted as a query. Unfortunately, although it is more mature, this implementation also has some bugs and limitations: it forbids some uses of splicing, and does not correctly process some queries that start with a conditional.

Example	F# 2.0	F# 3.0	Our system
differences (1)	Success	Success	Success
range (2)	Failure	Success	Success
satisfies (3)	Success	Failure	Success
satisfies (4)	Success	Failure	Success
rangeFromNames (5)	Failure	Failure	Success
$P(t_0)$ (6)	Success	Failure	Success
$P(t_1)$ (7)	Success	Failure	Success
expertise′ (8)	Success*	Success	Success*
expertise (9)	Failure	Avalanche	Success*
xp_0 (10)	Failure	Success	Success
xp_1 (11)	Failure	Success	Success
xp_2 (12)	Failure	Failure	Success*
xp_3 (13)	Failure	Failure	Success*

*marks the cases where our modifications to the F# 2.0 PowerPack library were required.

Table 1: Experimental results.

7. IMPLEMENTATION AND RESULTS

To validate our design, we implemented a pre-processor norm : Expr<A> → Expr<A> that takes any quoted F# sequence expression over the standard query operators and normalises it following the strategy defined in Section 5. Although it is possible to implement normalisation directly on the Expr<A> type, and this may be more efficient, we elected to introduce a simpler type Exp that corresponds exactly to the query expressions of Idealised LINQ. We first traverse the F# Expr<A> data structure and translate it into an Exp. Type information available through the F# reflection mechanism is used to annotate parts of the query appropriately. These expressions are normalised using an algorithm based on the query normalisation reductions of Section 5, using the type annotations to perform type-directed normalisation. The Idealised LINQ query expression is translated back to an F# Expr<A> data structure. This data structure is passed to the F# 2.0 PowerPack library which translates it to a LINQ Expression<A>. In turn Microsoft's LINQ to SQL query provider translates this to SQL and executes the query in SQL Server.

Table 1 summarises our experimental results. We wrote each example in the paper using the F# 2.0 and F# 3.0 LINQ libraries and our library. Each entry in the table lists whether the result was failure, success by generating a single query, or a "query avalanche" of multiple queries. In the course of running the experiments we encountered the bugs described in the previous section. We modified the

source code of the F# 2.0 PowerPack library to fix some of these problems. The modified source code is available online, along with the source code of all examples (Cheney et al. 2012). The experiments that rely upon our modifications are marked with an asterisk ($\star$) in Table 1.

Each of F# 2.0 and F# 3.0 failed on seven examples, though not the same seven. In one case, F# 2.0 required our modified PowerPack library. Also, F# 3.0 generated an avalanche of SQL queries for query (9), which uses an intermediate nested structure. Our normaliser (using the modified PowerPack library) succeeded on all our examples.

In all cases where more than one technique succeeded by generating a single query, the queries generated are equivalent. Incidentally, we note that since all three approaches ultimately generate C# Expression<A> terms, all of the examples can be implemented directly in C# or F# by writing programs that construct the appropriate Expression<A> terms using reflection. However, this typically involves a code size increase of a factor of at least 2, and can be difficult to debug because the dynamic expression construction is not typechecked until run-time.

8. QUOTATIONS VS. FUNCTIONS

We have adopted a style where all queries are represented as quotations. Another style which might at first appear appealing is to represent queries as functions, which take quotations as arguments and return quotations as results. For instance, one might redefine range from Section 2.3 as follows.

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

Before, we wrote an invocation like this:

$$\mathbf{run}(\texttt{<@ (\%range)(30, 40) @>}).$$

Now, we write an invocation like this:

$$\mathbf{run}(\mathsf{range}'(\texttt{<@ 30 @>}, \texttt{<@ 40 @>})).$$

The latter is slightly more efficient, as the application directly yields a quotation in normal form, and no beta-reduction is required.

However, the price paid for saving a few beta-reductions is high, because the variant form hinders composition. In Section 2.5 we used range to define rangeFromNames. Attempting a revision using range′ yields the following.

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

Something odd happens here! 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 variables will ever meet their binding occurrences. In contrast, all the other quotations we have seen are *closed*, since all quoted variables are bound within the quotation. While closed quotations are well understood, open quotations are more subtle, and a subject of active research (see e.g. Rhiger (2012)). In particular, open quotations are illegal in F#, so there is no easy way to use range′ to define rangeFromNames′.

The one advantage of the style described is that by splicing in directly it can avoid the need for beta-reductions in normalising quotations. However it does not avoid the need for all the other normalisation rules discussed in Section 5.2, and it hinders composition. As a guideline, we recommend that whenever possible, queries should be defined queries as quotations of functions, not functions over quotations.

9. RELATED WORK

LINQ (Meijer et al. 2006) has attracted considerable commercial interest, but has not been extensively documented in the research literature. Bierman et al. (2007) present a formalisation of several extensions to C#, including LINQ. Meijer (2011) gives an overview of the foundations of LINQ, while Beckman (2012) advocates LINQ as an interface to cloud computing platforms and Eini (2011) identifies obstacles to implementing LINQ providers for non-SQL databases. The problem of abstracting over parts of queries or constructing dynamic queries has been discussed widely on blogs and online forums but to our knowledge has not previously been formally modelled.

Syme (2006) presents an early version of F#'s quotation and reflection capabilities, illustrating via applications to LINQ, GPU code generation, and runtime F# code generation. Petricek (2007b;a) discusses early techniques for supporting some dynamic queries in C# and F#, including a clever technique for simulating certain forms antiquoting in C# LINQ expressions. F# 3.0's computation builder mechanism is presented by Petricek and Syme (2012) and covered further by Syme et al. (2012).

Type-safe quotation and metaprogramming is an active research area, with most work focusing on homogeneous multistage programming, where the embedded language is the same as the host language. Davies and Pfenning (2001) introduce a calculus $\lambda^{\Box}$ for closed homogeneous multistage programming based on a modal logic. Idealised LINQ can be viewed as a heterogeneous variant of $\lambda^{\Box}$ restricted to one level of embedding. Rhiger (2012) presents a calculus for homogeneous multistage programming with open quotations. Rhiger notes that closed quotation leads to less efficient code due to administrative redexes. In our heterogeneous setting such administrative redexes have negligible cost because we normalise quoted terms, and normalisation time is dominated by query execution time. Van den Bussche et al. (2005) present a meta-querying system for SQL; however, they do not consider type-safety or language integration issues.

Our quotation-based theory abstracts the practice of language-integrated query as found in Microsoft's LINQ. Other examples of type-safe language-integrated query include SML# (Ohori and Ueno 2011), Ur/Web (Chlipala 2010), and our own work on the Web programming language Links (Cooper et al. 2007, Lindley and Cheney 2012). SML# and Ur/Web do not perform normalisation, while Links supports it via a type-and-effect system (Cooper 2009, Lindley and Cheney 2012). Although Links' treatment avoids the need for explicit quotation and antiquotation, its type system would be nontrivial to incorporate into F#.

Ferry (Grust et al. 2009; 2010) is a functional query language that provides both higher-order functions and nested

data, and the Ferry team have implemented several LINQ query providers, as well as interfacing Ferry with Links (Ulrich 2011). Combining our results with Ferry's handling of nested queries is another topic for future work.

10. 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 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, and proved that normalisation always succeeds in translating any query of flat relation type to SQL. We presented 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.

We believe future LINQ providers could and should incorporate our normalisation technique, supporting many useful forms of abstraction in manner that is predictable and practical. We hope this paper will spur wider appreciation of the value of normalisation, and wider application of these techniques in practice.

Code supplement. Our implementation, and the source code of all examples, is available online (Cheney et al. 2012).

References

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

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*, New York, NY, USA, 2007. ACM.

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

J. Cheney, S. Lindley, and P. Wadler. The essence 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*, 2010.

E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, 2009.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, 2007.

G. Copeland and D. Maier. Making Smalltalk a database system. *SIGMOD Rec.*, 14(2), 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.

T. Goldschmidt, R. Reussner, and J. Winzen. A case study evaluation of maintainability and performance of persistency techniques. In *ICSE*, 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*, June 2009.

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

L. Libkin and L. Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2), 1997.

S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*, 2012.

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*, 2006.

A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *ICFP*, pages 307–319, 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. URL `http://www.cl.cam.ac.uk/~tp322/drafts/notations.pdf`.

J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North Holland, October 1981.

M. Rhiger. Staged computation with staged lexical scope. In *ESOP*, pages 559–578, 2012.

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

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

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), 1996.