Effective Quotation: Relating Approaches to Language-integrated Query

Citation for published version:

Digital Object Identifier (DOI):
10.1145/2543728.2543738

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Effective Quotation
Relating approaches to language-integrated query

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

Gabriel Radanne
ENS Cachan
gabriel.radanne@zoho.com

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

Abstract
Language-integrated query techniques have been explored in a number of different language designs. We consider two different, type-safe approaches employed by Links and F#. Both approaches provide rich dynamic query generation capabilities, and thus amount to a form of heterogeneous staged computation, but to date there has been no formal investigation of their relative expressiveness. We present two core calculi Eff and Quot, respectively capturing the essential aspects of language-integrated querying using effects in Links and quotation in LINQ. We show via translations from Eff to Quot and back that the two approaches are equivalent in expressiveness. Based on the translation from Eff to Quot, we extend a simple Links compiler to handle queries.

Categories and Subject Descriptors D.3.1 [Formal Definitions and Theory]: D.3.2 [Language Classifications]: Applicative (functional) languages; H.2.3 [Languages]: Query languages

Keywords language-integrated query; effects; quotation

1. Introduction
Increasingly, programming involves coordinating data and computation among several layers, such as server-side, client-side and database layers of a typical three-tier Web application. The interaction between the host programming language (e.g. Java, C#, F#, Haskell or some other general-purpose language) running on the server and the query language (e.g. SQL) running on the database is particularly important, because the relational model and query language provided by the database differ from the data structures of most host languages. Conventional approaches to embedding database queries within a general-purpose language, such as Java’s JDBC, provide the programmer with precise control over performance but are subject to typing errors and security vulnerabilities such as SQL injection attacks [35]. Object-relational mapping (ORM) tools and libraries, such as Java’s Hibernate, provide a popular alternative by wrapping database access and update in type-safe object-oriented interfaces, but this leads to a loss of control over the structure of generated queries, which makes it difficult to understand and improve performance [14].

To avoid these so-called impedance mismatch problems, a number of language-integrated query techniques for embedding queries into general-purpose programming languages have emerged, which seek to reconcile the goals of type-safety and programmer control. Two distinctive styles of language-integrated query have emerged:

- Employ some form of static analysis or type system to identify parts of programs that can be turned into queries (e.g. Kleisli [38], Links [8], Batches for Java [36]).
- Extend a conventional language with explicit facilities for quotation or manipulation of query code (e.g. LINQ [21], Ur/Web [5], Database-Supported Haskell [13]).

Links is an example of the first approach. It uses a type-and-effect system [32] to classify parts of programs as executable only on the database, executable only on the host programming language, or executable anywhere. For example, consider the employee and task data in tables Figure 1. The following code

```java
for (x <- employees)
  where(x.salary > 50000)
  [(name=x.name)]
```

retrieves the names of employees earning over $50,000, specifically [“Bert”, “Drew”, “Erik”, “Fred”, “Gina”]. In Links, the same code can be run either on the database (if employees and tasks are tables) or in the host language. If executed as a query, the interpreter generates a single (statically defined) SQL query that can take advantage of the database’s indexing or other query optimisation; if executed in-memory, the expression will by default be interpreted as a quadratic nested loop. (Efficient in-memory implementations of query expressions are also possible [16].)

In contrast, in Microsoft’s LINQ (supported in C#, F#, and some other .NET languages), the programming language is extended with query-like syntax. For example, the same query as above can be written in F# as:

```fsharp
query { for x in employees 
  where (x.salary > 50000)
  yield {name=x.name} }
```

Figure 1. Sample Data

<table>
<thead>
<tr>
<th>employees</th>
<th>tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>dpt</td>
<td>name</td>
</tr>
<tr>
<td>&quot;Product&quot;</td>
<td>&quot;Alex&quot;</td>
</tr>
<tr>
<td>&quot;Product&quot;</td>
<td>&quot;Bert&quot;</td>
</tr>
<tr>
<td>&quot;Research&quot;</td>
<td>&quot;Cora&quot;</td>
</tr>
<tr>
<td>&quot;Research&quot;</td>
<td>&quot;Drew&quot;</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>&quot;Erik&quot;</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>&quot;Fred&quot;</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>&quot;Gina&quot;</td>
</tr>
<tr>
<td>emp</td>
<td>tsk</td>
</tr>
<tr>
<td>&quot;Alex&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Bert&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;abstract&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;dissemble&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;enthuse&quot;</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>&quot;abstract&quot;</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>&quot;enthuse&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;enthuse&quot;</td>
</tr>
<tr>
<td>&quot;Fred&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Gina&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Gina&quot;</td>
<td>&quot;dissemble&quot;</td>
</tr>
</tbody>
</table>

CE"
This is just syntactic sugar for code that builds and manipulates quotations. In F#, this facility is built explicitly on top of language support for quotation [22, 29] and its computation expression syntax [26]. The above F# query expression is implemented by quoting the code inside the query{ ... } brackets and translating it (at run time) to C# values of type Expression<T>, which are converted to SQL by the .NET LINQ to SQL library.

The above example is rather simplistic: the query is static, that is, does not depend on any run-time data. Static queries can be handled easily even by libraries such as JDBC, and systems such as Links and LINQ provide the added benefit of type-safety. However, most queries are generated dynamically, depending on some run-time data. The ability to generate dynamic queries is essential for database programming. Libraries such as JDBC allow queries to be parameterized over base type values such as strings or integers, ensuring that values are correctly escaped to prevent SQL injection attacks. Both Links and LINQ go significantly further: they allow constructing dynamic queries using λ-abstraction and run-time normalization, while retaining type safety and preventing SQL injection. However, this capability comes with its own pitfalls: it can be difficult to predict when an expression can be turned into a single query.

To address this problem, Cooper [7] showed how to extend Links so that performance-critical code can be highlighted with the query keyword. Links will statically check that the enclosed expression will definitely translate to a single query (neither failing at run-time, nor generating multiple queries). We refer to this as the single-query guarantee. In database theory, conservativity results due to Wong [37] and others provided a single-query guarantee in the case of first-order queries: any query expression having flat input and output types can be turned into an SQL query. This idea provided the basis for the Kleisli system [38], which was a source of inspiration to Links; the single-query guarantee was generalised to the higher-order case by Cooper [7], who also gave a static type- and effect system that showed how to embed queries in a higher-order general-purpose language. Subsequent work on Links [19] generalised this to use row typing and effect polymorphism.

The possibility of generating LINQ queries dynamically in ad hoc cases was discussed by Syme [29] and Petricek [24, 25]. The F# and LINQ to SQL libraries in Microsoft .NET do not provide a single-query guarantee for dynamic queries; instead, they attempt to generate a single query but sometimes fail or generate multiple queries. In our recent paper [4] we showed that Cooper’s approach to normalisation for Links can be transferred to provide systematic support for abstraction in LINQ in F#, providing a single-query guarantee. In the rest of this paper, we consider the F# LINQ approach with this extension.

Nevertheless, there are still apparent differences between the approaches. For example, in F#, a query expression cannot be cloned (easily) reused as ordinary code. This potentially leads to the need to write (essentially) the same code twice, once for ordinary use and once for use on the database. Code duplication can interfere with the use of functional abstraction to construct queries. For example, the following Links code

```haskell
fun elem(x,xs) {
  not(empty(for (y <- xs) where (x == y) ()))
}
fun canDo(name,taks) {
  elem("build", for (t <- tasks) where (t.emp == name) [t.tsk])
}
query { for (x <- employees) where (canDo(x.name,"build")) [(name=x.name)] }
```

defines functions elem and canDo that test respectively whether a value is an element of a collection and whether an employee can do a certain task. The Links effect system correctly determines that elem can be run anywhere, and that canDo can be run on the database. When the query is to be executed, Links normalises the query by inlining elem and canDo and performing other transformations to generate a single SQL query [7]. In contrast, naively executing this code might involve loading all of the data from the employees table, and running one subquery to compute canDo for each employees row in-memory.

In F#, it is possible to do something similar, but only by explicitly quoting elem and canDo.

```haskell
let elem = @< fun x xs -> query { for y in xs 
    exists(y = x) } @>
let canDo = @< fun name tsk -> 
  (elem) tsk (for t in tasks where (t.emp = name) yield t.tsk) @>
query { for x in employees 
  where ((canDo) x.name "build") 
  yield {name=x.name} }
```

The quoted version of elem is spliced into the query using antiquotation (%elem). If we need the elem function in both query and non-query code, its code must be duplicated, or we need to evaluate or generate compiled code for it at runtime. F#’s quotation library does include Eval and Compile functions that can be used for this purpose, but it is not clear that these actually generate efficient code at runtime, nor is it convenient to write this boilerplate code.

It is important to note that SQL does not natively support general recursion or first-class functional abstraction (although there are recent proposals to support the latter [15]). Nonrecursive lambda-abstraction is supported in query expressions in Links and F#, but it is eliminated in the process of generating an SQL query. Recursive functions can also be used to construct queries from data in the host language, in both Links and LINQ, but care is needed to make the staging explicit. For example, in F# we can define a predicate that tests whether an employee can do all tasks in some list as follows:

```haskell
let rec canDoAll(taks) = match taks with [] -> <@ fun name -> true @> | tsk::taks' -> <@ fun name -> (canDo) name tsk && (canDoAll taks') name @>
query { for x in employees 
  where ((canDoAll ["build","call"])) x.name) 
  yield {name=x.name} }
```

This is also possible in Links, but we need to use function abstraction and hoist subcomputations to satisfy the effect type system:

```haskell
fun canDoAll(taks) {
  switch (taks) {
    case [] -> fun (name) {true} 
    case (tsk::taks') -> var p = canDoAll(taks'); 
      fun (name) { canDo(name,task) && p (name) } }
  }
query { for (x <- employees) 
  where (canDoAll(["build","call"]) (x.name) 
  [(name=x.name)]) }
```
We have to hoist the recursive call to canDoAll and name it so that it is clear to the type system that the recursive computation does not depend on values in the database that are not directly available to the host language interpreter. Arguably, in this case F#’s explicit quotation and antiquotation annotations clarify the distinction between staging and functional abstraction, whereas in Links the distinction is not as explicit in the program syntax.

Both techniques are essentially heterogeneous forms of staged computation, based on a common foundation of manipulating partially-evaluated query expressions (or query fragments) at run time in order to construct SQL queries. The single-query property [4, 7] guarantees each query expression succeeds in generating and only one query, even if lambda-abstraction or recursion is used to construct the queries. However, several natural questions about the relative strengths of the two approaches remain unanswered: Can we translate the Links effect-based approach to the (seemingly lower-level) LINQ quotation-based approach? If so, this might suggest a fruitful implementation strategy. Conversely, do we lose any expressiveness by providing the (seemingly higher-level) effect-based Links approach? Or can we always (in principle) translate LINQ-style quotation-based code to Links-style effect-based code?

In this paper, we consider the problem of relating the expressiveness of the two approaches to language-integrated query represented by Links and LINQ. Database query languages are often limited in expressiveness: for example, plain SQL conjunctive queries cannot express recursive properties such as transitive closure. Thus, understanding the relative expressiveness of different (Turing-incomplete) query languages, and the tradeoff with complexity of query evaluation or optimisation, are important issues in database theory [1]. We are interested in a dual question: what is the expressiveness of a programming language that generates queries? Given that both Links and LINQ approaches provide a measure of support for dynamic queries, can they express the same classes of dynamic queries?

To make this question precise, we introduce two core calculi: Eff ≤, representing the effect-based approach supported by Links, and Quot, representing the quotation-based approach adopted in LINQ in F#. The former is similar to Cooper’s core language [7]; the latter is essentially the same as the T-LINQ core language [4]. Both core languages make simplifying assumptions compared to Links and F# respectively, but we argue that they capture what is essential about the two approaches, making them suitable for a formal comparison that avoids preoccupation with other distracting details (e.g. Links’s support for client-side programming [9] or F#’s support for objects [30], or different facilities for polymorphism in both languages.)

In database theory, the expressiveness of a language is usually measured by the set of functions definable in it, according to a conventional denotational semantics of database queries. However, this notion of expressiveness is not very interesting for general-purpose languages: two Turing-complete programming languages are always (by definition) expressively equivalent in this sense. Felleisen [12] and Mitchell [23] proposed notions of expressiveness based on restricted forms of translation among different languages. However, neither of these notions seems appropriate for relating language-integrated query formalisms.

Programs interact with a database that may be concurrently updated by other programs, and we want a notion of equivalence that takes query behaviour into account while abstracting over the possible concurrent behaviours of the database. For example, we want to consider a program that issues query Q to the database inequivalent to another program that reads all the database tables into memory and executes Q in-memory. In addition, we wish to abstract as much as possible over the possible behaviours of the database: databases are typically concurrently accessed and updated by many applications, and we want our notion of expressiveness to minimise assumptions about the behaviour of the database. Thus, we define the semantics of both languages as labeled transitions, where labels are either silent transitions or pairs \((q, V)\) consisting of database queries and responses. We consider two programs query-equivalent if, given the same input, they have the same possible (finite and infinite) query/response traces \((q_1, V_1), \ldots, (q_n, V_n)\).

We show that Eff ≤ programs can be translated to query-equivalent Quot programs via a two-stage translation: first we eliminate subeffect by duplicating code in the doubling translation, then we introduce explicit quotation and antiquotation in the splicing translation. Perhaps more surprisingly, we can also give a converse translation from Quot to Eff ≤, which translates quoted code to thunks (functions with unit domain): thus, the two approaches are expressively equivalent up to query-equivalence.

The current version of Links is interpreted, and query normalisation depends on being able to inspect code at run time. The translation from Eff ≤ to Quot suggests a compilation strategy by translating Links-style code to explicitly quoted code.

In the rest of this paper, we present the following contributions:

- We propose an appropriate notion of dynamic query behaviour suitable for comparing the expressiveness of different language-integrated query techniques.
- We provide a detailed exploration of the relationship between implicit, effect-based (Links/Eff ≤) and explicit, quotation-based (LINQ/Quot) approaches, giving type- and semantics-preserving translations in each direction.
- We discuss an application of the translation from Eff ≤ to Quot to support compilation of Links programs with embedded queries, along with preliminary experimental results.

The rest of this paper is structured as follows. Section 2 presents necessary background material from prior work, and defines the desired notion of equivalence of programs with respect to observable query behaviour. Section 3 presents Eff ≤ and Quot, giving their syntax, type systems, and operational semantics. Section 4 presents translations between Eff ≤ and Quot. Section 5 presents a practical application of the translation from Eff ≤ to Quot, which serves as the basis for a prototype compiler for Links that supports run-time dynamic query generation. Section 6 provide additional discussion of related work and Section 7 concludes.

2. Background

The Nested Relational Calculus (NRC) is a widely-studied core language for database queries corresponding closely to monadic comprehension syntax [2, 3]. Previous work [4, 7, 19, 37] has shown how first- and higher-order variants of NRC can be used for language-integrated query. We give the syntax of first-order NRC in Figure 2. Extended examples of the use of NRC are presented in prior work [2, 4].

We let \(x\) range over variables, \(c\) range over constants, and \(op\) range over primitive operators. Records \((f \leftarrow \bar{q})\) and field projections \(q.f\) are standard. We write \(\emptyset\) for an empty bag, \(q\) for the singleton bag containing the element \(q\), and \(q \leftrightarrow q'\) for the union of bags \(q\) and \(q'\). We write for \((x^A \leftarrow q)\) a bag comprehension, which for each element \(x\) in \(q\) evaluates \(q'\), then computes the union of the resulting bags. We write table \(t\) for the relational database table \(t\). In order to keep normalisation as simple as possible we restrict ourselves to one sided conditionals over collections (equivalent to SQL where clauses). The expression if \(q\) then \(q'\) else \(q''\) evaluates to \(q\) if \(q\) evaluates to true and \(t\) if \(q\) evaluates to false. Lindley and Cheney [19] describe how to normalise in the presence of general conditionals; briefly, the idea is to push conditionals inside records.
and translate if \( q q' \) to \( (q q') + (\sim q q') \) where \( q' \) are of bag type.

The NRC types include base types (int, bool, string, etc.), record types \( (E : A) \), and bag types \( [A] \). Row types are flat record types restricted to contain base types (just like rows in SQL queries).

Conservativeness results (see e.g. Wong [37]) ensure that any NRC expression \( M \) has a flat return type and flat inputs can be normalised to a form that corresponds directly to SQL.

We will model the behaviour of a database server nondeterministically: whenever a query is posed, the response may be any value of the appropriate type. Thus, we fix a set \( \Omega \) of all pairs \((q, V)\) such that whenever \( \sim v : A \) we have \( \sim V : A \). Further constraints, reflecting the semantics of the query language or integrity constraints on the database tables, could be imposed. Our results concerning expressiveness are parametric in \( \Omega \) (provided it is at least type-safe and respects query equivalence).

**Definition 1.** Let \( L \) be a set of actions, including a “silent” action \( \tau \), and let \( \mu \) range over elements of \( L \). A labeled transition system over some set of labels \( L \) is a structure \((X, \to)\) where \( \to \subseteq X \times L \times X \). We write \( x \xrightarrow{\mu} y \) when \( (x, \mu, y) \in \to \) for some \( \mu \in L \) and \( x \to y \) when \( x \xrightarrow{\tau} y \). We write \( \xrightarrow{\tau} \) for \( \xrightarrow{\tau} \) and \( \xrightarrow{\mu} \) for \( \xrightarrow{\mu} \). Suppose that \( x_1 \equiv \mu_{1} \Rightarrow x_2 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} \cdots \) is a (finite or infinite) run starting from \( x_1 \). We say that the (finite or infinite) word \( \mu_1 \cdots \mu_n \) obtained from concatenating the non-\( \tau \) labels is the trace of the run, and we define \( \text{Tr}(x) \) to be the set of all traces starting from \( x \).

**Definition 2.** Let \((X, \rightarrow_X)\) and \((Y, \rightarrow_Y)\) be labeled transition systems over the same label set \( L \). We say that \( x \in X \) tracesimulates \( y \in Y \) (or \( x \preceq y \)) if \( \text{Tr}(x) \subseteq \text{Tr}(y) \), and \( x \in X \) and \( y \in Y \) are trace-equivalent (\( x = y \)) if \( \text{Tr}(x) = \text{Tr}(y) \).

In the rest of this paper, we are concerned with instances of trace-equivalence where the labels are either \( \tau \) or pairs \((q, V) \in \Omega \).

### 3. Formalisation

#### 3.1 Source Language: Eff$_\tau$

Eff$_\tau$ is a higher-order nested relational calculus over bags augmented with an effect type system for issuing flat NRC relational queries. It is similar to calculi considered previously in work on query compilation [7, 19]. A difference is that Eff$_\tau$ models two ground effects (pl and db), whereas the other calculi model just one (indicating code that cannot be run in the database). Like Cooper’s system, Eff$_\tau$ provides subeffecting. Our previous work [19], and its implementation in the Links web programming language [8] uses row-based effect polymorphism instead; while effect polymorphism is more flexible, we focus on the simpler subtype-based approach here.

The effects of Eff$_\tau$ are given by the following grammar.

\[
\begin{align*}
\text{(ground effects)} & \quad X, Y ::= \text{pl} \mid \text{db} \\
\text{(effects)} & \quad E ::= X \mid \text{any} \\
\end{align*}
\]

And that runs in the programming language has the \text{pl} effect. Code that runs in the database has the \text{db} effect. Code that can be run in either place, i.e. anywhere, has the \text{any} effect. The typing rules will enforce a subeffecting discipline, allowing a function that has the \text{any} effect to be applied with either the \text{pl} or the \text{db} effect.

The types of Eff$_\tau$ are given by the following grammar.

\[
\begin{align*}
\text{(types)} & \quad A, B ::= O \mid (E : A) \mid [A] \\
\end{align*}
\]

Types in Eff$_\tau$ extend the NRC base types, records, and collections with function types \( A \rightarrow B \) annotated by an effect \( E \).

The terms of Eff$_\tau$ are given by the following grammar.

\[
\begin{align*}
L, M, N ::= x \mid c \mid \text{op}(M) \mid \lambda x.A,M \mid M N \\
\mid \text{if } L M \mid (E : M) \mid M.ℓ \\
\mid [M] \mid M + N \mid f x.A,M \mid \text{fold } M N M \\
\mid \text{query } M \mid \text{table } t \\
\end{align*}
\]

We will often omit type and effect annotations on bindings. We include them so that later we can define translations on terms rather than on judgements (Section 4).

The syntax is in most cases similar to NRC or standard. Lambda-abstractions \( \lambda x.A,M \) are annotated with an effect \( E \). Recursive functions \( f x.M \) and folds are only available in the programming language. The fold operation is the only elimination form for bag values. The special form \text{query } M denotes a query returning results of type \([R]\), where \( M \) has type \([R]\).

A type environment ascribes types to variables.

\[ \Gamma, \Delta ::= \cdot \mid \Gamma, x : A \]

The typing rules are given in Figure 3. In this figure and elsewhere, notations such as \( \Sigma \) or \( (\Sigma : A) \) abbreviate lists of terms, judgments, etc. The typing judgment is of the form \( \Gamma \vdash M : A \mid E \), indicating that in context \( \Gamma \), term \( M \) has type \( A \) and effect \( E \). The effect \( E \) can be either \text{pl} or \text{db}, indicating that \( M \) can only be executed by the host programming language or by the database respectively, or it can be \text{any}, indicating that \( M \) can be executed in either context.

We again assume a signature \( \Sigma \) that maps each constant \( c \) to its underlying type, each primitive operator \( op \) to its type (e.g. \( \Sigma(\lambda) = (\text{Bool,Bool}) \rightarrow \text{Bool} \) and \( \Sigma(+) = (\text{Int,Int}) \rightarrow \text{Int} \)), and each table \( t \) to the type of its rows. (For simplicity we assume that the same base types, constants and primitive operations are available to both the host and database.) The rules are quite standard; most of them are parametric in the effect. The most interesting rules are those that do something non-trivial with effects, such as \text{TABLE}, \text{QUERY}, \text{REC}, \text{FOLD} and \text{APP}. The \text{TABLE} rule ensures that table references table \( t \) can only be used in a database context. The \text{QUERY} rule requires that the body of a query expression query \( M \) must be run in the database, and the query expression itself must be invoked from the programming language. This rule also implicitly requires that a query result type \([R]\) is a bag of records (note that \( R \) is a row type). Recursive functions (rule \text{REC}) and folds (rule \text{FOLD}) can only be applied in the programming language. The \text{APP} rule allows a function with the \text{any} effect to be applied in the programming language or the database.

We define a sub-language \text{Eff} by restriction of \text{Eff}$_\tau$ to programs in which the \text{any} effect is disallowed. The subeffecting constraint in the \text{APP} rule is superfluous in this sublanguage. We can translate any \text{Eff}$_\tau$ program to an equivalent, albeit longer, \text{Eff} program, by
\[
\Gamma \vdash M : A!E \quad E_1 \leq E_2
\]

<table>
<thead>
<tr>
<th>VAR</th>
<th>[ \Sigma(c) = A ]</th>
<th>[ \Sigma(op) = (\overline{O}) \rightarrow O ]</th>
<th>[ \Gamma \vdash M : O!E ]</th>
<th>[ \Gamma \vdash \text{op}(\overline{M}) : O!E ]</th>
<th>[ \Gamma \vdash M : O!E ]</th>
<th>[ \Gamma \vdash M : O!E ]</th>
<th>[ \Gamma \vdash \lambda x.M : A \rightarrow^t B \vdash E']</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP</td>
<td>[ \Gamma \vdash M \rightarrow^t B!E ]</td>
<td>[ \Gamma \vdash N : A!E ]</td>
<td>[ E' \leq E ]</td>
<td>[ \Gamma \vdash M N : B!E ]</td>
<td>[ \Gamma \vdash \text{if } \text{LM} : [E] ]</td>
<td>[ \Gamma \vdash \text{M} : A!E ]</td>
<td>[ \Gamma \vdash (\ell : M) : (\ell : A)!E ]</td>
</tr>
</tbody>
</table>

**Figure 3. Typing and Subeffecting Rules for Eff**

\[ \Gamma \vdash M : A \]

<table>
<thead>
<tr>
<th>VAR</th>
<th>[ \Sigma(c) = A ]</th>
<th>[ \Sigma(op) = (\overline{O}) \rightarrow O ]</th>
<th>[ \Gamma \vdash M_1 : O ]</th>
<th>[ \Gamma \vdash \text{op}(\overline{M}) : O ]</th>
<th>[ \Gamma \vdash M : O ]</th>
<th>[ \Gamma \vdash \text{lift } M : \text{Expr}O &gt; ]</th>
<th>[ \Gamma \vdash x^A : M \rightarrow B ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP</td>
<td>[ \Gamma \vdash M : A \rightarrow B ]</td>
<td>[ \Gamma \vdash N : A ]</td>
<td>[ \Gamma \vdash M : A ]</td>
<td>[ \Gamma \vdash \text{if } \text{LM} : [E] ]</td>
<td>[ \Gamma \vdash \text{M} : A ]</td>
<td>[ \Gamma \vdash (\ell : M) : (\ell : A)!A ]</td>
<td>[ \Gamma \vdash [\ell : A] ]</td>
</tr>
</tbody>
</table>

**Figure 4. Typing Rules for Quot**
(value)
\[ V, W : \vdash c \mid \lambda x. M \mid \text{rec } f \ x. M \mid \langle T = V \rangle \mid \langle V \rangle \]

(evaluation context)
\[ \mathcal{E} ::= \varepsilon \mid \text{op}(\mathcal{E}, \mathcal{E}, \mathcal{E}) \mid \mathcal{E} M \mid V \mathcal{E} \]
\[ \langle T = V, E', \mathcal{E}, \mathcal{E}, M \rangle \mid \langle E, E \rangle \mid \mathcal{E} \mathcal{E} \]
\[ \mathcal{E} + M \mid \langle V \rangle + \mathcal{E} \mid \text{for } (x \leftarrow E) N \]
\[ \text{if } \mathcal{E} M \]

--

**Figure 5.** Values and Evaluation Contexts for $\text{Eff}_5$

\[
\text{op}(\mathcal{V}) \rightarrow \delta(\text{op}, \mathcal{V}) \\
(\lambda x. M) V \rightarrow M[x := V] \\
(\text{rec } f \ x. M) V \rightarrow M[f := \text{rec } f \ x. M, x := V] \\
\langle T = V \rangle \ell_i \rightarrow V_i \\
\text{if } true M \rightarrow M \\
\text{if } false M \rightarrow \Box \\
\Box + V \rightarrow V \\
\text{for } (x \leftarrow \Box) N \rightarrow \Box \\
\text{for } (x \leftarrow [V] + V') N \rightarrow [V] + (V' + V') \\
\text{fold } \Box M N \rightarrow M \\
\text{fold } ([V] + W) M N \rightarrow N V \text{ (fold } W M N) \\
\text{query } M \rightarrow M[V] \rightarrow \langle M[V], V \rangle \notin \Omega \\
\mathcal{E}[M] \rightarrow \mathcal{E}[N]
\]

**Figure 6.** Operational Semantics for $\text{Eff}_5$

representing each any function as a pair of a pl function and a db function (Section 4.1).

3.2 Operational Semantics for $\text{Eff}_5$

We now present small-step operational semantics for $\text{Eff}_5$. The syntax of values and evaluation contexts is given in Figure 5. The values are standard. We write $\mathcal{V}$ for $[V_1] + \ldots + [V_n] + \Box$. The operational semantics is parameterised by reduction $M \xrightarrow{\mathcal{E}} N$ as shown in Figure 6. It is parameterised by an interpretation $\delta$ for each primitive operation $\text{op}$, and a set $\Omega$ of possible query request and response pairs $(q, V)$, both of which respect types: if $\Sigma(\text{op}) = \mathcal{O} \rightarrow O$ and $\vdash \forall V. O$ and $V = \delta(\text{op}, \mathcal{V})$ then $\vdash V : O$, and if $(q, V) \in \Omega$ and $\vdash q : [R]$ then $\vdash V : [R]$.

The rules are standard apart from the one for query evaluation. Evaluation contexts $\mathcal{E}$ enforce left-to-right call-by-value evaluation. Rule (query) evaluates a query $M$ by first normalising $M$ to yield an equivalent NRC query $q = [M]$, and then taking a transition labeled $(q, V)$ yielding result value $V$. The normalisation function $\varepsilon$ is the same as that of Cheney et al. [4]. It first applies standard symbolic reduction rules (Figure 7) and then further ad hoc reduction rules (Figure 8). The former eliminate all nesting and abstraction from a closed term of flat bag type, while the latter account for the lack of uniformity in SQL (see [4] for further details). Define $|L| = N$ when $L \rightarrow^* M$ and $M \rightarrow^* N$, where $M$ and $N$ are in normal form with respect to $\rightarrow^*$ and $\rightarrow^*$ respectively.

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

**Proposition 3.** If $\Gamma \vdash M : A \rightarrow E$ and $M \xrightarrow{\mathcal{E}} N$ then $\Gamma \vdash N : A \rightarrow E$. If $\Gamma \vdash M : A \rightarrow E$ then either $M$ is a value or $M \xrightarrow{\mathcal{E}} N$ for some $N$ and $\mu$.

3.3 Target Language: Quot

Quot is a higher-order nested relational calculus over bags augmented with a quotation mechanism for constructing NRC queries using quotation.

Modulo superficial differences Quot is essentially the same as the T-LINQ core language [4]. Specifically, the differences are: the lexical syntax; Quot includes fold; Quot includes extra type annotations (to aid the translation to $\text{Eff}$); T-LINQ has a database construct whereas Quot has a table construct.

Where $\text{Eff}_5$ uses effect types to distinguish the programming language from the database, Quot uses quotation to distinguish the host language from the query language. Host language terms may build and evaluate quoted query language terms. For convenience (and ease of comparison with $\text{Eff}_5$), we use the same syntax for the host and query languages, so that the query language is essentially a sublanguage of the host language (except for quotation and table $\ell$). In general, the two languages may be different, as already discussed elsewhere [4].

The types of Quot are given by the following grammar.

\[
\text{(types) } A, B ::= O \mid (E : A) \mid [A] \mid A \rightarrow B \mid \text{Expr} \langle A \rangle
\]

They are the same as for $\text{Eff}_5$, except function types are not annotated with effects, and types are extended to include quotation types $\text{Expr} \langle A \rangle$, which represent closed, quoted query terms of type $A$.

The terms of Quot are given by the following grammar.

\[
L, M, N ::= x \mid c \mid \text{op}(\langle M \rangle) \mid \lambda x^A. M \mid M N \\
\downarrow \text{if } L M \downarrow (\langle E = M \rangle) \mid M.\ell \\
\downarrow \text{if } [M] \mid M + N \downarrow (x^A \leftarrow M) N \\
\downarrow \text{rec } f^{A \rightarrow B} x^A. M \downarrow \text{fold } N L M N \\
\downarrow \text{query } M \downarrow \text{table } t \\
\downarrow \langle \Phi M \rangle \downarrow \langle \text{Expr} \rangle \downarrow \text{lift } M
\]

The grammar is largely the same as that of $\text{Eff}_5$. The key difference is that effects are replaced by quotation constructs. Lambda-abstractions are no longer annotated with effects. More importantly, Quot includes quotation: we write $\langle \Phi M \rangle$ for the quotation operation, where $M$ is a query term; we write $\langle \text{Expr} \rangle$ for the ant quo-

tation operation, which splits a quoted term $M$ into a query term; and we write $\text{lift } M$ for the operation that coerces a value of base type to a quoted value. (This is a limited form of cross-stage per-
sistence [31], which is otherwise unavailable because quotation expression must be closed.) Type environments \( \Gamma, \Delta \) are as follows.

\[
\Gamma, \Delta ::= \emptyset \ | \ \Gamma, x : A
\]

The typing rules are given in Figure 4. 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 \( \Gamma \). The judgement \( \Gamma; \Delta \vdash M : A \) states that query term \( M \) has type \( A \) in host type environment \( \Gamma \) and query type environment \( \Delta \).

Most of the typing rules are standard and similar in both judgements. The variable typing rule (VARQ) for the query judgement does not allow the use of variables from the \( \Gamma \) environment; hence variables from \( \Gamma \) must be explicitly lifted (rule LIFT) or used within an antiquotation (rule ANTIQUOTE) in order to be used within a quotation.

The interesting rules are those that involve quotation, namely QUOTE, QUERY, ANTIQUOTE, and LIFT. Query terms can only be quoted in the host language (QUOTE). In the host language, a closed quoted term of flat bag type can be evaluated as a query (QUERY). A quoted term can be spliced into a query term (ANTIQUOTE). A host term of base type can be lifted to a query term (LIFT). Note that variables of bag, record, or function type in \( \Gamma \) cannot be lifted using LIFT. As in Eff\(_s\), the table \( t \) construct is only available within a query; its use within queries is enabled by rule TABLE and its use elsewhere is forbidden because there is no similar rule for unquoted table references.

### 3.4 Operational Semantics for Quot

The operational semantics for Quot is standard, and is similar to that given in Cheney et al. [4] for the T-LINQ language. Values and evaluation contexts are given in Figure 9. The normalisation function \( \| \cdot \| \) is the same as \( \| \cdot \| \) but applied to Quot instead of Eff. Define \( \| L \| = N \) when \( L \sim^* M \) and \( M \sim^* N \), where \( M \) and \( N \) are in normal form with respect to \( \sim \) and \( \sim^* \) respectively. The semantics is given in Figure 10; most rules are the same as in Eff\(_s\), except those involving quotation and querying. The rules are parameterised by the same \( \delta \) and \( \mu \) as the semantics of Eff\(_s\).

**Proposition 4.** If \( \Gamma \vdash M : A \) and \( M \overset{\mu}{\rightarrow} N \) then \( \Gamma \vdash N : A \). If \( \Gamma \vdash M : A \) then either \( M \) is a value or \( M \overset{\mu}{\rightarrow} N \) for some \( N \) and \( \mu \).

### 4. Translations

In this section we present translations that show that Eff\(_s\) and Quot can simulate one another. The translations are summarised in Figure 11. We first (Section 4.1) show how to compile away the any-effect, translating arbitrary Eff\(_s\) programs to any-free Eff programs. Next (Section 4.2) we show how to translate Eff programs to Quot programs. In the reverse direction, we first (Section 4.3) give a straightforward translation hoisting computations out of antiquote in Quot, resulting in a normal form Quot' in which all antiquotations are of the form \( \langle x \rangle \). Finally (Section 4.4) we translate Quot' programs to Eff programs.

#### 4.1 From Eff\(_s\) to Eff

As a first step, we can translate away all subeffecting through a global doubling translation that simulates each any function as a pair of a pl function and a db function. The type translation (\( A \)) is shown in Figure 12. Type environments are translated pointwise:

\[
\{ x_1 : A_1, \ldots, x_n : A_n \} = x_1 : \{ A_1 \}, \ldots, x_n : \{ A_n \}
\]

Terms are translated as shown in Figure 13, where \( \langle M \rangle X \) stands for the doubling translation of \( M \) with respect to target effect \( X \). The type and term translations are structure-preserving except on any-function types, abstractions, and applications; these interesting cases are highlighted in grey boxes. Technically, this translation is defined by induction on the structure of typing derivations. However, only the cases where this matters are those for applications of any-functions. To avoid notational clutter, we write the translation in a syntax-directed style and only include type annotations in the cases for application.
or second element of the pair before applying the corresponding target effect parameter is used to determine whether to use the first. Each

The main correctness properties are as follows:

The term translation is parameterised by the concrete effect: \(\text{pl}\) or \(\text{db}\). The type translation is written \([A]_X\), where \(X\) is \(\text{pl}\) or \(\text{db}\) and \(A\) is an Eff-type, and is defined in Figure 14. In the programming language, \(\text{db}\) function types are interpreted as quoted functions. In the database, \(\text{pl}\) function types are interpreted as the unit type, which (as we will show) suffices because \(\text{pl}\) functions can never be called in the database.

One additional complication is how to deal with occurrences of variables bound within a \(\text{pl}\) context, that are also accessed by code within a \(\text{db}\) context. For example, consider the function \(\lambda^\text{pl}.x.(\lambda^\text{db}y.(x,y),x)\): to translate this, we need to be able to convert the value of \(x\) to a quoted term, but \(\text{Quot}\) only allows lifting at base type. We deal with this by adding a special shadow variable \(x_{\text{db}}\), for each ordinary variable \(x\), so that the value of \(x_{\text{db}}\) is a quoted version of \(x\). In this example, the translation is

\[
\lambda x.\text{let } x_{\text{db}} = \langle x \rangle \text{ in } \langle \lambda y. (\langle x\rangle_{\text{db}}, y) \Rightarrow x \rangle.
\]

The special variables \(x_{\text{db}}\) and reification operation \(\iota x (-)\) are explained in greater detail below.

To define the term translation, we need some auxiliary notation. The interpretations for terms and type environments are further parameterised by an effect environment \(\rho\), which tracks the provenance of bound variables. This is necessary for interpreting a value bound in a programming language context and used in a database context, or vice-versa. The environment \(\rho\) is a finite map from variable names \(x\) to pairs \(A \times X\) denoting the type \(A\) and effect \(X\) at which \(x\) was bound. We write \(e\) for the empty effect environment and \(\rho[x \mapsto A \times X]\) for the extension of effect environment \(\rho\) with the mapping \(x\) to \(A \times X\).

We define a judgement \(\Gamma \vdash \rho\) which indicates that effect environment \(\rho\) is compatible with type environment \(\Gamma\):

\[
\begin{array}{c}
\text{EMPTYENV} \\
\Gamma \vdash \epsilon
\end{array}
\quad
\begin{array}{c}
\text{EXTENDENV} \\
\Gamma \vdash \rho \\
\Gamma, x : A \vdash \rho[x \mapsto A \times X]
\end{array}
\]

The interpretation \([\rho]_X\) of an effect environment \(\rho\) at effect \(X\) is defined as:

\[
\begin{align*}
\lbrack \epsilon \rbrack_X & = \cdot \\
\lbrack \rho[x \mapsto A \times X] \rbrack_{\text{db}} & = \lbrack \rho \rbrack_{\text{db}}, x : [A]_{\text{db}} \\
\lbrack \rho[x \mapsto A \times X] \rbrack_{\text{pl}} & = \lbrack \rho \rbrack_{\text{pl}}, x : [A]_{\text{pl}}, x_{\text{db}} : [A]_{\text{db}} \\
\lbrack \rho[x \mapsto A \times X] \rbrack_X & = \lbrack \rho \rbrack_X
\end{align*}
\]

if \(X \neq Y\) which induces the interpretation of a type environment with respect to a compatible effect environment. Notice that the translation \([\rho]_{\text{pl}}\) introduces the shadow variables \(x_{\text{db}}\). If \(\Gamma \vdash \rho\), then we define \([\Gamma]_{X}^\rho\) as follows:

\[
[\Gamma]_{X}^\rho \equiv [\rho]_{X}
\]

The term translation is defined as \([M]_{X}^\rho\) in Figure 16, where \(M\) is an Eff expression, \(X\) is an effect \(\text{db}\) or \(\text{pl}\) called the target of the translation, and \(\rho\) is an effect environment mapping variables to their types and effects (in Eff). The type translation is structure-preserving except at function types, and similarly the term translation is structure-preserving in most cases. The interesting cases (highlighted in grey) are those for function types, lambda-abstractions, or variables where the translation’s target effect \(X\) does not match the actual effect. In addition, whenever a variable \(x\) is bound in the \(\text{pl}\) context, we bind an additional special variable \(x_{\text{db}}\), whose value is a quoted version of \(x\); intuitively, we need this quoted value to translate any occurrences of \(x\) within a \(\text{db}\) context. This affects all of the variable binding cases of \([\_]_{\text{pl}}\) and is explained in more detail below. Variables are coerced to effect \(X\) using the \(\mathfrak{z}_0\) operation described below. Following the interpretation on types, \(\lambda^\text{db}\)-abstractions are coerced to a \(\text{pl}\) target by trans-
Moreover, if $\rho(x) = A! X$, then an error term $\text{error}_A = (\text{rec } f \mapsto \lambda x. x \cdot f x) ()$

Reification is a type-directed operation that maps a term of type $[A]_{\text{pl}}$ to a corresponding term of type $[\text{reify}(A)]_{\text{db}}$. A term of base type is reified by lifting and splicing. Both (need be) because lifting coerces a value of base type to its quotation.) A $\text{db}$ function is reified as unit. $A$ function is reified as an antiquotation. Records and lists are reified by structural recursion on the type (which is why we chose to include fold in the core calculi).

**Theorem 8** (Type preservation). Assume $\Gamma \vdash \rho$.

1. If $\Gamma \vdash M : A! \text{pl}$, then $[\lambda x. x]_{\text{pl}} \sqsubseteq [A]_{\text{pl}} \sqsubseteq [\text{reify}(A)]_{\text{db}}$.
2. If $\Gamma \vdash M : A! \text{db}$, then $[\lambda x. x]_{\text{db}} \sqsubseteq [A]_{\text{db}} \sqsubseteq [\text{reify}(A)]_{\text{pl}}$.

Notice that the translation of the database portion of the type environment $[\Gamma]_{\text{pl}}$ appears in the programming language judgement $\Gamma_{\text{pl}} \vdash [\text{reify}(A)]_{\text{db}}$, in Theorem 8(1). This is sound as any such database variable in $[M]_{\text{pl}}$ is interpreted as $\text{error}$. Database variables can never appear in a closed program.

**Theorem 9** (Semantics preservation). Assume $\Gamma \vdash M : A! \text{pl}$.

1. $M$ is a value if and only if $[M]_{\text{pl}}$ is a value.
2. If $M \nvdash M$, then $[M]_{\text{pl}} \nvdash [N]_{\text{pl}}$.
3. If $[M]_{\text{pl}}$ is reducible then there exist $N$ and $\mu$ such that $[M]_{\text{pl}} \equiv_{\text{db}} [N]_{\text{pl}}$ and $M \nvdash N$.

Moreover, if $\Gamma \vdash M : A! \text{db}$ then $[M]_{\text{db}} = [M]_{\text{pl}}$.
\[
\langle \text{Int} \rangle_X = \text{Int} \\
\langle \text{Bool} \rangle_X = \text{Bool} \\
\langle A \to B \rangle_X = \langle B \rangle_X \to \langle A \rangle_X \\
\langle f \circ A \rangle_X = \langle f \rangle_X (\langle A \rangle_X) \\
\langle \text{Expr} < \lambda \rangle \rangle_X = \langle \lambda \rangle_{\text{db}}
\]

\textbf{Figure 17. Quot’ to Eff: types}

\[
\begin{align*}
\langle x \rangle_X &= x \\
\langle c \rangle_X &= c \\
\langle \text{op}(M) \rangle_X &= \text{op}(\langle M \rangle_X) \\
\langle \text{op}(\lambda x:A.M) \rangle_X &= \text{op}(\langle x \rangle_A \langle x \rangle_A \langle M \rangle_X) \\
\langle M \cdot N \rangle_X &= \langle M \rangle_X . \langle N \rangle_X \\
\langle \Gamma, M \rangle_X &= \langle \Gamma \rangle_X \mid \langle M \rangle_X \\
\langle M + N \rangle_X &= \langle M \rangle_X + \langle N \rangle_X \\
\langle \text{for}(\alpha \to M) \rangle_X &= \langle \text{for}(\alpha \to M) \rangle_X (\langle M \rangle_X) (\langle N \rangle_X) \\
\langle \text{let} M \phi \rangle_X &= \langle \lambda x:A.M \rangle_X \phi \\
\langle \text{query} M \rangle_X &= \langle \text{query} \rangle (\langle M \rangle_{\text{db}}) \\
\langle \text{rec} f^{M} \rangle_X &= \langle \text{rec} f^{M} \rangle_X (\langle A \rangle_{\text{db}}) \phi \\
\langle \text{fold} \Gamma, M \rangle_{\text{db}} &= \langle \text{fold} \phi \rangle (\langle M \rangle_{\text{db}}) (\langle N \rangle_{\text{db}}) \\
\langle \text{table} t \rangle_{\text{db}} &= \langle \text{table} t \rangle \\
\langle \langle\mathcal{M}\rangle \rangle_{\text{db}} &= \langle \langle\mathcal{M}\rangle \rangle_{\text{pl}}
\end{align*}
\]

\textbf{Figure 18. Quot’ to Eff: terms}

hoisted out of a quotation are still evaluated left-to-right with respect to their original positions in the quotation.

We omit (routine but tedious) proofs of type-preservation and semantics-preservation for this transformation.

\subsection{4.4 From Quot’ to Eff}

We now give a translation from Quot’ to Eff. The type translation is shown in Figure 17. It is structure-preserving except on closed quotation types, which are translated to \textbf{db} thunks.

Type environments are translated pointwise:

\[
\begin{align*}
\langle x_1 : A_1, \ldots, x_n : A_n \rangle_X &= x_1 : \langle x_1 \rangle_A, \ldots, x_n : \langle A_n \rangle_X
\end{align*}
\]

The term translation is shown in Figure 18. It is structure-preserving, except on lambda-abstractions, quotation, queries, antiquotation and lifting. Lambda-abstractions are annotated with the appropriate effect. Quoted and lifted terms are translated to \textbf{db} thunks \(\lambda x:A.M)_{\text{db}}\) and \(\lambda x:A.M)_{\text{pl}}\) respectively. For lifting, note that it does not actually matter whether we use \(\text{let} X \phi\) or \(\text{let} X \phi\) on \(M\), since in a Quot’ term, \(M\) will always be either a variable or constant of base type. Queries and antiquotations are translated to force the \textbf{db} thunks by applying to unit.

\textbf{REM 11. The \(\langle \_ \rangle_X\) translation uses general rules for translating quotation, lifting, and antiquotation of arbitrary types. However, this is only correct for source expressions in Quot’. The Quot’ terms resulting from the hoisting stage in the previous section will only have variables appearing as arguments to quotation, lifting, and antiquotation. However, this invariant is not preserved by evaluation, because variables may be replaced by values during evaluation. This is why Quot’ allows antiquotation, lifting, and query operations to be applied to values as well as variables.}

\textbf{Theorem 12 (Type preservation).}

1. If \(\Gamma \vdash M : A\), then \(\langle M \rangle_{\text{pl}} : \langle A \rangle_{\text{pl}} ! \text{pl}\).
2. If \(\Gamma; \Delta \vdash M : A\), then \(\langle M \rangle_{\text{pl}}, \langle \Delta \rangle_{\text{db}} \vdash \langle A \rangle_{\text{db}} ! \text{db}\).

\textbf{Theorem 13 (Semantics preservation). Assume \(v : M : A\).}

1. If \(M\) is a value then \(\langle M \rangle_{\text{pl}}\) is a value.
2. If \(\Gamma \vdash M : A\) and \(M \triangleright \to N\), then \(\langle M \rangle_{\text{pl}} (M \triangleright \to \gamma) (\langle N \rangle_{\text{pl}}\).
3. If \(\Gamma \vdash M : A\) and \(\langle M \rangle_{\text{pl}}\) is reducible then there exists \(N\) and \(\mu\) such that \(\langle M \rangle_{\text{pl}} \langle N \rangle_{\text{pl}} \langle N \rangle_{\text{pl}}\).

Moreover, if \(\cdot \vdash Q : A\) then \(\langle Q \rangle_{\text{pl}} = \langle \langle Q \rangle \rangle_{\text{pl}}\).

\textbf{Remark 14. Unlike the previous translations, these translations are not exact simulations of steps in Quot by steps in Eff. The reason is that the splicing rule in Figure 10 corresponds to \(\beta\)-value reduction under a \(\lambda\)-abstraction in Eff. Fortunately, \(\beta\)-value equivalence is valid for Eff modulo \(\equiv\).

CULORARY 3. If \(\Gamma \vdash M : A\) in Quot then \(M \approx \langle M \rangle_{\text{pl}}\).

\section{5. Application: Links query compilation}

Holtz [17] previously developed a compiler for plain Links. Holtz’ compiler translates Links programs (which include row polymorphism and other features not present in plain OCaml) to OCaml programs that manipulate tagged values. This typically improves the performance of computationally-intensive Links programs by 1-2 orders of magnitude. However, the compiler does not support Links’ query or client-side Web programming features. The translation from Eff to Quot shows how to compile effect-based Links code to quotation-based code. Moreover, quotation-based queries are relatively easy to translate to plain OCaml simply by translating quotation expressions to an explicit run-time abstract syntax tree representation, and extending the Links run-time library to include query normalisation. Accordingly, we have adapted Holtz’ compiler to support queries, by first translating the Links intermediate representation (IR) code to eliminate effect-polymorphic functions via doubling, then inserting appropriate splicing annotations, and finally translating the resulting IR code to OCaml code that explicitly manipulates quoted terms.

The Links IR differs from Eff\(_\text{c}\) in some important respects: in particular, it employs row typing and effect polymorphism instead of subtyping. In the current implementation, we handle a subset of Links and some polymorphic code is not handled. However, we believe the basic idea of the doubling translation can be adapted to handle polymorphism instead of subtyping. We view the current query compiler as a proof of concept demonstrating practical implications of the expressiveness results presented earlier, while it is not a mature compiler for Links, our experience with it reported here will help guide further development of such a compiler.

In the rest of this section we present some experiments showing the strengths and weaknesses of the doubling and splicing translations, which we hope will guide future work on compilation for language-integrated query. Figure 19 compares the interpreter (1), compiler without support for queries (C-Q) and query-enabled compiler (C+Q) on several examples.

We first consider programs that do not involve queries, and so can be handled by all three techniques. The map-any benchmark (Figure 19(a)) measures the time needed to map a simple any-function (which calls some other any-functions) over a list. The query-enabled compiler has a small, but measurable overhead compared to the plain compiler. The sumlist benchmark (Figure 19(b)) measures the time to construct a list of the first \(n\) natural numbers, and compute its sum. Here, both compilers provide similar results, though again C+Q is slightly slower. The quicksort benchmark (Figure 19(c)) sorts a decreasing \(n\)-element list, exercising the quadratic worst-case behaviour of quicksort. In this case, both compilers provide similar speedup.

We next consider examples that involve a mix of queries and ordinary execution. These examples can only be run using the
terpreter and query-enabled compiler. The first benchmark (Figure 19(d)) simply loops and generates \( n \) queries, with no other computation. There is no measurable difference between the two techniques. This is unsurprising since most of the time is spent in communicating with the database, and compilation does not affect this time, so speedup is bounded by Amdahl’s Law. Next, we consider a variant (Figure 19(e)) where in each iteration the program both issues a query and performs some computation, namely a quicksort of a 10-element list. As we saw before, the compiler yields a significant improvement on the quicksort code. Finally, we consider a similar benchmark (Figure 19(f)), where the query has a higher-order parameter, which takes one value for odd-numbered iterations and another for even-numbered iterations. The results of this benchmark show no appreciable difference from Figure 19(e).

These results show that while the translation from Eff to Quot introduces some overhead to ordinary code compared to the basic compiler, the query-enabled compiler still can realise significant gains for code that mixes queries and ordinary execution.

6. Related Work

There is a large and growing literature on approaches to language-integrated query, as well as language-based techniques for combining conventional execution with other execution models, such as MapReduce, GPU, and multicore-based data-parallelism. We discuss only work closely related to this paper; other recent papers [4, 19] compare our approach to language-integrated query with other work in greater depth.

Wadler [33] advocated monads as a technique for structuring programs, including list comprehensions and database queries. This approach was adopted in the nested relational calculus of Buneanu et al. [2] and Wong [37] gave rewriting-based normalisation techniques for translating complex nested relational queries over flat data to SQL queries, implemented in the Kleisli system [38]. Links [7, 8, 19] built on this work in several ways, particularly in introducing the ability to compose queries using nonrecursive lambda-abstractions and recursion in the host language.

The LINQ approach adopted in C#, F# and other .NET languages also draws upon monadic comprehensions and nested relational query languages [20, 21], but differs in its implementation strategy: query syntax in C# and F# is desugared to quoted abstract syntax trees which are manipulated and translated to SQL by a library. In F#, it is possible to write dynamic LINQ queries that fail at run time or generate unnecessarily large numbers of SQL queries. Our recent work [4] gave examples, and showed how Links’s normalisation algorithm can be adapted to F# to remedy this problem.

Our formalisation of LINQ-style Quot draws upon a long line of work on quotation and metaprogramming, starting with MetaML [27, 31]. Our approach is closest to the \( \lambda^2 \) calculus of Davies and Pfenning, which provides homogeneous closed quotation (the host and quoted languages coincide); for simplicity, we consider only one level of staging. As discussed elsewhere [4], open quotation can be simulated in Quot using lambda-abstraction, but better support for open quotation and multiple stages, possibly following the approach of Rhiger [27], may also be of interest. Our approach also has some similarities to Eckhardt et al.’s explicitly heterogeneous approach [11]. Reasoning about multi-stage programs is a well-known hard problem. Choi et al. [6] present translations from staged to unstaged programs that employ similar ideas to our translations, particularly hoisting pl-code out of db-code in the \( \langle \rangle \) translation. Inoue and Taha [15] present techniques for reasoning about call-by-value multi-stage programs.

Wadler and Thiemann demonstrate a close relationship between effect type systems and monads [34]. Our translation from Eff to Quot has some similarities to that work. However, the languages considered here are quite different from those in Wadler and Thiemann’s work; the latter employ reference types, effects that are sets of regions, and monads indexed by sets of regions, and there is nothing analogous to our doubling translation.

Felleisen [12] and Mitchell [23] presented different notions of expressiveness of programming languages, formulated in terms of different kinds of reductions preserving termination behaviour or observational equivalence. We adopt an ad hoc notion of equivalence based on preservation of query behaviour, which is inspired to some extent by the notion of (weak) bisimilarity familiar from concurrency theory [28]. However, in general nondeterministic labelled transition systems, bisimilarity is strictly stronger than trace equivalence, so it is possible that our translations do not preserve observable behaviour up to bisimulation. We intend to investigate whether our translations are (weak) bisimulations.

7. Conclusion

Combining database capabilities with general-purpose programming has been of interest for nearly thirty years [10]. Despite this long history, only within the last ten years have mature techniques begun to appear in mainstream languages, with the chief example being Microsoft’s LINQ, based on explicitly manipulating query code at run time using quotations. Over the same period, techniques developed in the Kleisli and Links languages have built on rigorous foundations of query rewriting to show how to type-safely embed nested relational queries in general-purpose languages, using type-and-effect systems.
In recent work [4], we started to bring these threads together, by showing that some techniques from Links, particularly query normalisation, can be adapted to LINQ in F# in order to improve the expressiveness of the latter. That work raised the question of the relative expressiveness of the two approaches with respect to dynamic query generation: Can Links express dynamic queries that LINQ in principle cannot, or vice versa? 

In this paper, we provided a partial answer to this question: we proposed core languages Eff, and QueF similar to those used in previous work on Links and LINQ respectively, and we gave semantics-preserving translations in both directions. This shows, surprisingly in our view, that the two approaches are equivalent in expressive power, at least relative to simple classes of queries: that is, while Links programs or LINQ programs may seem more convenient in different situations, in principle any program written using one approach can also be written using the other. In addition, we used one direction of the translation to extend a Links compiler with partial support for queries, demonstrating the effectiveness of quotation for compiling Links.

A number of areas for future work remain, including extending our translations to handle other query language features such as grouping, aggregation and nested results; extending the compiler to handle full Links including polymorphism; and completely eliminating the overhead of doubling, which we believe should be possible using closure conversion and storing each generated d b function in a lookup table indexed by the code pointer of the corresponding pl function.

Acknowledgments

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