Effective Quotation: Relating Approaches to Language-integrated Query

Cheney, J., Lindley, S., Radanne, G., & Wadler, P. (2014). Effective Quotation: Relating Approaches to Language-integrated Query. In Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. (pp. 15-26). (PEPM '14). New York, NY, USA: ACM. doi: 10.1145/2543728.2543738

Effective Quotation

Relating approaches to language-integrated query

James Cheney Sam Lindley
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]).

employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

tasks

emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"call"
"Cora"	"dissemble"
"Cora"	"enthuse"
"Drew"	"abstract"
"Drew"	"enthuse"
"Erik"	"call"
"Erik"	"enthuse"
"Fred"	"call"
"Gina"	"call"
"Gina"	"dissemble"

Figure 1. Sample Data

- 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 in Figure 1. The following code

```
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:

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

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 normalisation, 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 LINQ, a query expression cannot be (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

```
fun elem(x,xs) {
  not(empty(for (y <- xs) where (x == y) [()]))
}
fun canDo(name,tsk) {
  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`.

```
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:

```
let rec canDoAll(tsks) =
  match tsks with
    [] -> <@ fun name -> true @>
  | tsk::tsks' -> <@ fun name ->
      (%canDo) name tsk && (%canDoAll tsks') 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:

```
fun canDoAll(tsks) {
  switch (tsks) {
    case [] -> fun (name) {true}
    case (tsk::tsks') ->
      var p = canDoAll(tsks');
      fun (name) { canDo(name,tsk) && 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 one 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: $\mathsf{Eff}_{\leq}$, 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 $(q_1, V_1), \ldots, (q_n, V_n), \ldots$.

We show that $\mathsf{Eff}_{\leq}$ programs can be translated to query-equivalent Quot programs via a two-stage translation: first we eliminate subeffecting 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 $\mathsf{Eff}_{\leq}$, 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 $\mathsf{Eff}_{\leq}$ 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/$\mathsf{Eff}_{\leq}$) and explicit, quotation-based (LINQ/Quot) approaches, giving type- and semantics-preserving translations in each direction.
- We discuss an application of the translation from $\mathsf{Eff}_{\leq}$ 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 $\mathsf{Eff}_{\leq}$ and Quot, giving their syntax, type systems, and operational semantics. Section 4 presents translations between $\mathsf{Eff}_{\leq}$ and Quot. Section 5 presents a practical application of the translation from $\mathsf{Eff}_{\leq}$ 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 $\langle \overline{\ell = q} \rangle$ and field projections $q.\ell$ are standard. We write $[]$ for an empty bag, $[q]$ for the singleton bag containing the element q, and $q \uplus q'$ for the union of bags q and q'. We write $\mathsf{for}\,(x^A \leftarrow q)\; q'$ for a bag comprehension, which for each element x in q evaluates q', then computes the union of the resulting bags. We write $\mathsf{table}\; t$ 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 $\mathsf{if}\; q\; q'$ evaluates to q' if q evaluates to true and $[]$ 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

$$
\begin{array}{lll}
\text{(Query)} & q & ::= x \mid c \mid op(\overline{q}) \mid \mathsf{if}\ q\ q' \mid \langle \overline{\ell = q} \rangle \mid q.\ell \\
 & & \mid\ [] \mid [q] \mid q_1 +\!\!+ q_2 \mid \mathsf{for}\,(x \leftarrow q)\ q' \\
 & & \mid\ \mathsf{table}\ t \\
\text{(Base type)} & O & ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \cdots \\
\text{(Type)} & A, B & ::= O \mid \langle \overline{\ell = A} \rangle \mid [A] \\
\text{(Row type)} & R & ::= \langle \overline{\ell : O} \rangle
\end{array}
$$

Figure 2. Syntax of NRC

and translate $\mathsf{if}\ q\ q'\ q''$ to $(\mathsf{if}\ q\ q') +\!\!+ (\mathsf{if}\ (\neg q)\ q'')$ when q', q'' are of bag type.

The NRC types include base types (Int, Bool, String, etc.), record types $\langle \overline{\ell : A} \rangle$, and bag types $[A]$. Row types are *flat* record types restricted to contain base types (just like rows in SQL queries).

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

We assume a fixed signature Σ mapping constants c to base types, operators op to functions on base types, and table references to flat bag types $[R]$. We omit typing or evaluation rules for queries; these are standard and implicit in the typing and operational semantics rules of $\mathsf{Eff}_\leq$ and Quot given later.

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 Ω of all pairs (q, V) such that whenever $\vdash q : A$ we have $\vdash V : A$. 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 Ω (provided it is at least type-safe and respects query equivalence).

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

DEFINITION 2. *Let $(X, \longrightarrow_X)$ and $(Y, \longrightarrow_Y)$ be labeled transition systems over the same label set L. We say that $x \in X$* trace-simulates *$y \in Y$ (or $x \lesssim y$) if $Tr(x) \subseteq Tr(y)$, and $x \in X$ and $y \in Y$ are* trace-equivalent *($x \approx y$) if $Tr(x) = Tr(y)$.*

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

3. Formalisation

3.1 Source Language: $\mathsf{Eff}_\leq$

$\mathsf{Eff}_\leq$ 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 in previous work on query compilation [7, 19]. A difference is that $\mathsf{Eff}_\leq$ models two ground effects ($\mathbf{pl}$ and $\mathbf{db}$), whereas the other calculi model just one (indicating code that cannot be run in the database). Like Cooper's system, $\mathsf{Eff}_\leq$ 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 $\mathsf{Eff}_\leq$ are given by the following grammar.

$$
\begin{array}{lll}
\text{(ground effects)} & X, Y & ::= \mathbf{pl} \mid \mathbf{db} \\
\text{(effects)} & E & ::= X \mid \mathbf{any}
\end{array}
$$

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

The types of $\mathsf{Eff}_\leq$ are given by the following grammar.

$$\text{(types)}\quad A, B ::= O \mid \langle \overline{\ell : A} \rangle \mid [A] \mid A \to^E B$$

Types in $\mathsf{Eff}_\leq$ extend the NRC base types, records, and collections with function types $A \to^E B$ annotated by an effect E.

The terms of $\mathsf{Eff}_\leq$ are given by the following grammar.

$$
\begin{array}{rcl}
L, M, N & ::= & x \mid c \mid op(\overline{M}) \mid \lambda^E x^A.M \mid M\ N \\
 & \mid & \mathsf{if}\ L\ M \mid \langle \overline{\ell = M} \rangle \mid M.\ell \\
 & \mid & [] \mid [M] \mid M +\!\!+ N \mid \mathsf{for}\,(x^A \leftarrow M)\ N \\
 & \mid & \mathsf{rec}\ f^{A \to^{\mathbf{pl}} B}\ x^A.M \mid \mathsf{fold}\ L\ M\ N \\
 & \mid & \mathsf{query}\ M \mid \mathsf{table}\ t
\end{array}
$$

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 $\lambda^E x^A.M$ are annotated with an effect E. Recursive functions $\mathsf{rec}\ f\ x.M$ and folds are only available in the programming language. The fold operation is the only elimination form for bag values. The special form $\mathsf{query}\ M$ 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 $\overline{M}$ or $\overline{\Gamma \vdash M : A\,!\,E}$ abbreviate lists of terms, judgments, etc. The typing judgment is of the form $\Gamma \vdash M : A\,!\,E$, indicating that in context Γ, term M has type A and effect E. The effect E can be either $\mathbf{pl}$ or $\mathbf{db}$, indicating that M can only be executed by the host programming language or by the database respectively, or it can be $\mathbf{any}$, indicating that M can be executed in either context.

We again assume a signature Σ that maps each constant c to its underlying type, each primitive operator op to its type (e.g. $\Sigma(\wedge) = (\mathit{Bool}, \mathit{Bool}) \to \mathit{Bool}$ and $\Sigma(+) = (\mathit{Int}, \mathit{Int}) \to \mathit{Int}$), 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 TABLE, QUERY, REC, FOLD and APP. The TABLE rule ensures that table references $\mathsf{table}\ t$ can only be used in a database context. The QUERY rule requires that the body of a query expression $\mathsf{query}\ M$ 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 REC) and folds (rule FOLD) can only be applied in the programming language. The APP rule allows a function with the $\mathbf{any}$ effect to be applied in the programming language or the database.

We define a sub-language Eff by restriction of $\mathsf{Eff}_\leq$ to programs in which the $\mathbf{any}$ effect is disallowed. The subeffecting constraint in the APP rule is superfluous in this sublanguage. We can translate any $\mathsf{Eff}_\leq$ program to an equivalent, albeit longer, Eff program, by

$\boxed{\Gamma \vdash M : A \,!\, E}$ $\boxed{E_1 \le E_2}$

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash x : A \,!\, E} \qquad \textsc{Const}\ \frac{\Sigma(c) = A}{\Gamma \vdash c : A \,!\, E} \qquad \textsc{Op}\ \frac{\Sigma(op) = (\overline{O}) \to O \qquad \overline{\Gamma \vdash M : O \,!\, E}}{\Gamma \vdash op(\overline{M}) : O \,!\, E} \qquad \textsc{Lam}\ \frac{\Gamma, x : A \vdash M : B \,!\, E}{\Gamma \vdash \lambda^E x^A.M : A \to^E B \,!\, E'}$$

$$\textsc{App}\ \frac{\Gamma \vdash M : A \to^{E'} B \,!\, E \qquad \Gamma \vdash N : A \,!\, E \qquad E' \le E}{\Gamma \vdash M\ N : B \,!\, E} \qquad \textsc{If}\ \frac{\Gamma \vdash L : Bool \,!\, E \qquad \Gamma \vdash M : [A] \,!\, E}{\Gamma \vdash \mathsf{if}\ L\ M : [A] \,!\, E} \qquad \textsc{Record}\ \frac{\overline{\Gamma \vdash M : A \,!\, E}}{\Gamma \vdash \langle \overline{\ell = M} \rangle : \langle \overline{\ell : A} \rangle \,!\, E}$$

$$\textsc{Project}\ \frac{\Gamma \vdash M : \langle \overline{\ell : A} \rangle \,!\, E}{\Gamma \vdash M.\ell_i : A_i \,!\, E} \qquad \textsc{Empty}\ \frac{}{\Gamma \vdash [] : [A] \,!\, E} \qquad \textsc{Singleton}\ \frac{\Gamma \vdash M : A \,!\, E}{\Gamma \vdash [M] : [A] \,!\, E} \qquad \textsc{Union}\ \frac{\Gamma \vdash M : [A] \,!\, E \qquad \Gamma \vdash N : [A] \,!\, E}{\Gamma \vdash M +\!\!+ N : [A] \,!\, E}$$

$$\textsc{For}\ \frac{\Gamma \vdash M : [A] \,!\, E \qquad \Gamma, x : A \vdash N : [B] \,!\, E}{\Gamma \vdash \mathsf{for}\ (x^A \leftarrow M)\ N : [B] \,!\, E} \qquad \textsc{Table}\ \frac{\Sigma(t) = [A]}{\Gamma \vdash \mathsf{table}\ t : [A] \,!\, \mathbf{db}} \qquad \textsc{Query}\ \frac{\Gamma \vdash M : [R] \,!\, \mathbf{db}}{\Gamma \vdash \mathsf{query}\ M : [R] \,!\, \mathbf{pl}} \qquad \textsc{Rec}\ \frac{\Gamma, f : A \to^{\mathbf{pl}} B, x : A \vdash M : B \,!\, \mathbf{pl}}{\Gamma \vdash \mathsf{rec}\ f^{A \to^{\mathbf{pl}} B}\ x^A.M : A \to^{\mathbf{pl}} B \,!\, E}$$

$$\textsc{Fold}\ \frac{\Gamma \vdash L : [A] \,!\, \mathbf{pl} \qquad \Gamma \vdash M : B \,!\, \mathbf{pl} \qquad \Gamma \vdash N : A \to^{\mathbf{pl}} B \to^{\mathbf{pl}} B \,!\, \mathbf{pl}}{\Gamma \vdash \mathsf{fold}\ L\ M\ N : B \,!\, \mathbf{pl}} \qquad \textsc{Reflexivity}\ \frac{}{E \le E} \qquad \textsc{AnyPl}\ \frac{}{\mathbf{any} \le \mathbf{pl}} \qquad \textsc{AnyDb}\ \frac{}{\mathbf{any} \le \mathbf{db}}$$

Figure 3. Typing and Subeffecting Rules for $\mathsf{Eff}_{\le}$

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

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash x : A} \qquad \textsc{Const}\ \frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \textsc{Op}\ \frac{\Sigma(op) = (\overline{O}) \to O \qquad \overline{\Gamma \vdash M_i : O_i}}{\Gamma \vdash op(\overline{M}) : O} \qquad \textsc{Lift}\ \frac{\Gamma \vdash M : O}{\Gamma \vdash \mathsf{lift}\ M : \mathsf{Expr}\langle O \rangle} \qquad \textsc{Lam}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B}$$

$$\textsc{App}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \qquad \textsc{If}\ \frac{\Gamma \vdash L : Bool \qquad \Gamma \vdash M : [A]}{\Gamma \vdash \mathsf{if}\ L\ M : [A]} \qquad \textsc{Record}\ \frac{\overline{\Gamma \vdash M : A}}{\Gamma \vdash \langle \overline{\ell = M} \rangle : \langle \overline{\ell : A} \rangle} \qquad \textsc{Project}\ \frac{\Gamma \vdash M : \langle \overline{\ell : A} \rangle}{\Gamma \vdash M.\ell_i : A_i} \qquad \textsc{Empty}\ \frac{}{\Gamma \vdash [] : [A]}$$

$$\textsc{Singleton}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : [A]} \qquad \textsc{Union}\ \frac{\Gamma \vdash M : [A] \qquad \Gamma \vdash N : [A]}{\Gamma \vdash M +\!\!+ N : [A]} \qquad \textsc{For}\ \frac{\Gamma \vdash M : [A] \qquad \Gamma, x : A \vdash N : [B]}{\Gamma \vdash \mathsf{for}\ (x \leftarrow M)\ N : [B]} \qquad \textsc{Quote}\ \frac{\Gamma; \cdot \vdash M : A}{\Gamma \vdash \texttt{<@}\ M\ \texttt{@>} : \mathsf{Expr}\langle A \rangle}$$

$$\textsc{Query}\ \frac{\Gamma \vdash M : \mathsf{Expr}\langle [R] \rangle}{\Gamma \vdash \mathsf{query}\ M : [R]} \qquad \textsc{Rec}\ \frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \mathsf{rec}\ f^{A \to B}\ x^A.M : A \to B} \qquad \textsc{Fold}\ \frac{\Gamma \vdash L : [A] \qquad \Gamma \vdash M : B \qquad \Gamma \vdash N : A \to B \to B}{\Gamma \vdash \mathsf{fold}\ L\ M\ N : B}$$

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

$$\textsc{VarQ}\ \frac{}{\Gamma; \Delta, x : A \vdash x : A} \qquad \textsc{ConstQ}\ \frac{\Sigma(c) = A}{\Gamma; \Delta \vdash c : A} \qquad \textsc{OpQ}\ \frac{\Sigma(op) = (\overline{O}) \to O \qquad \overline{\Gamma; \Delta \vdash M : O}}{\Gamma; \Delta \vdash op(\overline{M}) : O} \qquad \textsc{LamQ}\ \frac{\Gamma; \Delta, x : A \vdash M : B}{\Gamma; \Delta \vdash \lambda x^A.M : A \to B}$$

$$\textsc{AppQ}\ \frac{\Gamma; \Delta \vdash M : A \to B \qquad \Gamma; \Delta \vdash N : A}{\Gamma; \Delta \vdash M\ N : B} \qquad \textsc{IfQ}\ \frac{\Gamma; \Delta \vdash L : Bool \qquad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \mathsf{if}\ L\ M : A} \qquad \textsc{RecordQ}\ \frac{\overline{\Gamma; \Delta \vdash M : A}}{\Gamma; \Delta \vdash \langle \overline{\ell = M} \rangle : \langle \overline{\ell : A} \rangle} \qquad \textsc{ProjectQ}\ \frac{\Gamma; \Delta \vdash M : \langle \overline{\ell : A} \rangle}{\Gamma; \Delta \vdash M.\ell_i : A_i}$$

$$\textsc{EmptyQ}\ \frac{}{\Gamma; \Delta \vdash []^A : [A]} \qquad \textsc{SingletonQ}\ \frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash [M] : [A]} \qquad \textsc{UnionQ}\ \frac{\Gamma; \Delta \vdash M : [A] \qquad \Gamma; \Delta \vdash N : [A]}{\Gamma; \Delta \vdash M +\!\!+ N : [A]} \qquad \textsc{ForQ}\ \frac{\Gamma; \Delta \vdash M : [A] \qquad \Gamma; \Delta, x : A \vdash N : [B]}{\Gamma; \Delta \vdash \mathsf{for}\ (x \leftarrow M)\ N : [B]}$$

$$\textsc{Table}\ \frac{\Sigma(t) = [R]}{\Gamma; \Delta \vdash \mathsf{table}\ t : [R]} \qquad \textsc{Antiquote}\ \frac{\Gamma \vdash M : \mathsf{Expr}\langle A \rangle}{\Gamma; \Delta \vdash (\%M) : A}$$

Figure 4. Typing Rules for Quot

(value)
$$V, W ::= c \mid \lambda x.M \mid \mathsf{rec}\ f\ x.M \mid \langle \overline{\ell = V} \rangle \mid [\overline{V}]$$
(evaluation context)
$$\begin{array}{rcl} \mathcal{E} & ::= & [\,] \mid op(\overline{V}, \mathcal{E}, \overline{M}) \mid \mathcal{E}\ M \mid V\ \mathcal{E} \\ & \mid & \langle \overline{\ell = V}, \ell' = \mathcal{E}, \overline{\ell'' = M} \rangle \mid \mathcal{E}.\ell \mid [\mathcal{E}] \\ & \mid & \mathcal{E} +\!\!+ M \mid [V] +\!\!+ \mathcal{E} \mid \mathsf{for}\,(x \leftarrow \mathcal{E})\ N \\ & \mid & \mathsf{if}\ \mathcal{E}\ M \end{array}$$

Figure 5. Values and Evaluation Contexts for Eff

$$\begin{array}{rcl} op(\overline{V}) & \longrightarrow & \delta(op, \overline{V}) \\ (\lambda x.M)\ V & \longrightarrow & M[x := V] \\ (\mathsf{rec}\ f\ x.M)\ V & \longrightarrow & M[f := \mathsf{rec}\ f\ x.M, x := V] \\ \langle \overline{\ell = V} \rangle.\ell_i & \longrightarrow & V_i \\ \mathsf{if}\ \mathsf{true}\ M & \longrightarrow & M \\ \mathsf{if}\ \mathsf{false}\ M & \longrightarrow & [] \\ [] +\!\!+ V & \longrightarrow & V \\ ([V] +\!\!+ V') +\!\!+ V'' & \longrightarrow & [V] +\!\!+ (V' +\!\!+ V'') \\ \mathsf{for}\,(x \leftarrow [])\ N & \longrightarrow & [] \\ \mathsf{for}\,(x \leftarrow [V] +\!\!+ W)\ N & \longrightarrow & N[x := V] +\!\!+ \\ & & \quad (\mathsf{for}\,(x \leftarrow W)\ N) \\ \mathsf{fold}\ []\ M\ N & \longrightarrow & M \\ \mathsf{fold}\ (([V]) +\!\!+ W)\ M\ N & \longrightarrow & N\ V\ (\mathsf{fold}\ W\ M\ N) \\ \mathsf{query}\ M & \xrightarrow{(|M|, V)} & V \qquad ((|M|, V) \in \Omega) \end{array}$$

$$\frac{M \xrightarrow{\mu} N}{\mathcal{E}[M] \xrightarrow{\mu} \mathcal{E}[N]}$$

Figure 6. Operational Semantics for $\mathsf{Eff}_{\leq}$

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

3.2 Operational Semantics for $\mathsf{Eff}_{\leq}$

We now present small-step operational semantics for $\mathsf{Eff}_{\leq}$. The syntax of values and evaluation contexts is given in Figure 5. The values are standard. We write $[\overline{V}]$ for $[V_1] +\!\!+ \ldots +\!\!+ [V_n] +\!\!+ []$. The operational semantics is defined by reduction relation $M \xrightarrow{\mu} N$ as shown in Figure 6. It is parameterised by an interpretation δ for each primitive operation op, and a set Ω of possible query request and response pairs (q, V), both of which respect types: if $\Sigma(op) = \overline{O} \to O$ and $\vdash \overline{V : O}$ and $V = \delta(op, \overline{V})$ then $\vdash V : O$, and 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 $|-|$ 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 $L \rightsquigarrow^* M$ and $M \hookrightarrow^* N$, where M and N are in normal form with respect to $\rightsquigarrow$ and $\hookrightarrow$ respectively.

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

PROPOSITION 3. *If $\Gamma \vdash M : A\,!\,E$ and $M \xrightarrow{\mu} N$ then $\Gamma \vdash N : A\,!\,E$. If $\Gamma \vdash M : A\,!\,E$ then either M is a value or $M \xrightarrow{\mu} N$ for some N and μ.*

$$\begin{array}{rcl} (\lambda x.M)\ N & \rightsquigarrow & M[x := N] \\ \langle \overline{\ell = M} \rangle.\ell_i & \rightsquigarrow & M_i \\ \mathsf{for}\,(x \leftarrow [M])\ N & \rightsquigarrow & N[x := M] \\ \mathsf{for}\,(y \leftarrow \mathsf{for}\,(x \leftarrow L)\ M)\ N & \rightsquigarrow & \mathsf{for}\,(x \leftarrow L)\ \mathsf{for}\,(y \leftarrow M)\ N \\ \mathsf{for}\,(x \leftarrow \mathsf{if}\ L\ M)\ N & \rightsquigarrow & \mathsf{if}\ L\ (\mathsf{for}\,(x \leftarrow M)\ N) \\ \mathsf{for}\,(x \leftarrow [])\ N & \rightsquigarrow & [] \\ \mathsf{for}\,(x \leftarrow (L +\!\!+ M))\ N & \rightsquigarrow & \\ & & (\mathsf{for}\,(x \leftarrow L)\ N) +\!\!+ (\mathsf{for}\,(x \leftarrow M)\ N) \\ \mathsf{if}\ \mathsf{true}\ M & \rightsquigarrow & M \\ \mathsf{if}\ \mathsf{false}\ M & \rightsquigarrow & [] \end{array}$$

Figure 7. Normalisation Stage 1: symbolic reduction

$$\begin{array}{rcl} \mathsf{for}\,(x \leftarrow L)\,(M +\!\!+ N) & \hookrightarrow & (\mathsf{for}(x \leftarrow L)\ M) +\!\!+ (\mathsf{for}\,(x \leftarrow L)\ N) \\ \mathsf{for}\,(x \leftarrow L)\ [] & \hookrightarrow & [] \\ \mathsf{if}\ L\ (M +\!\!+ N) & \hookrightarrow & (\mathsf{if}\ L\ M) +\!\!+ (\mathsf{if}\ L\ N) \\ \mathsf{if}\ L\ [] & \hookrightarrow & [] \\ \mathsf{if}\ L\ (\mathsf{if}\ M\ N) & \hookrightarrow & \mathsf{if}\ (L\ \&\ M)\ N \\ \mathsf{if}\ L\ (\mathsf{for}\,(x \leftarrow M)\ N) & \hookrightarrow & \mathsf{for}\,(x \leftarrow M)\ (\mathsf{if}\ L\ N) \end{array}$$

Figure 8. Normalisation Stage 2: ad hoc reduction

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 Eff); T-LINQ has a database construct whereas Quot has a table construct.

Where $\mathsf{Eff}_{\leq}$ 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 $\mathsf{Eff}_{\leq}$), 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 antiquotation and table t). In general, the two languages may be different, as already discussed elsewhere [4].

The types of Quot are given by the following grammar.

$$\text{(types)}\quad A, B ::= O \mid \langle \overline{\ell : A} \rangle \mid [A] \mid A \to B \mid \mathsf{Expr}\langle A \rangle$$

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

The terms of Quot are given by the following grammar.

$$\begin{array}{rcl} L, M, N & ::= & x \mid c \mid op(\overline{M}) \mid \lambda x^A.M \mid M\ N \\ & \mid & \mathsf{if}\ L\ M \mid \langle \overline{\ell = M} \rangle \mid M.\ell \\ & \mid & [] \mid [M] \mid M +\!\!+ N \mid \mathsf{for}\,(x^A \leftarrow M)\ N \\ & \mid & \mathsf{rec}\ f^{A \to B}\ x^A.M \mid \mathsf{fold}\ L\ M\ N \\ & \mid & \mathsf{query}\ M \mid \mathsf{table}\ t \\ & \mid & \texttt{<@}\ M\ \texttt{@>} \mid \texttt{(\%}M\texttt{)} \mid \mathsf{lift}\ M \end{array}$$

The grammar is largely the same as that of $\mathsf{Eff}_{\leq}$. 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 `<@` M `@>` for the *quotation* operation, where M is a query term; we write `(%`M`)` for the *antiquotation* operation, which splices a quoted term M into a query term; and we write 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 expressions must be closed.)

Type environments Γ, Δ are as follows.

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

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 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 similar in both judgements. The variable typing rule (VarQ) for the query judgement does not allow the use of variables from the Γ environment; hence variables from Γ 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 Γ cannot be lifted using Lift. As in $\mathsf{Eff}_{\leq}$, 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 $\| - \|$ is the same as $| - |$ but applied to Quot instead of Eff. Define $\|L\| = N$ when $L \rightsquigarrow^* M$ and $M \hookrightarrow^* N$, where M and N are in normal form with respect to $\rightsquigarrow$ and $\hookrightarrow$ respectively. The semantics is given in Figure 10; most rules are the same as in $\mathsf{Eff}_{\leq}$, except those involving quotation and querying. The rules are parameterised by the same δ and Ω as the semantics of $\mathsf{Eff}_{\leq}$.

Proposition 4. *If $\Gamma \vdash M : A$ and $M \xrightarrow{\mu} N$ then $\Gamma \vdash N : A$. If $\Gamma \vdash M : A$ then either M is a value or $M \xrightarrow{\mu} N$ for some N and μ.*

4. Translations

In this section we present translations that show that $\mathsf{Eff}_{\leq}$ 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 $\mathsf{Eff}_{\leq}$ 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 antiquotations in Quot, resulting in a normal form Quot′ in which all antiquotations are of the form $(\%x)$. Finally (Section 4.4) we translate Quot′ programs to Eff programs.

4.1 From $\mathsf{Eff}_{\leq}$ 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 $\langle A \rangle$ is shown in Figure 12. Type environments are translated pointwise:

$$\langle x_1 : A_1, \ldots, x_n : A_n \rangle = x_1 : \langle A_1 \rangle, \ldots, x_n : \langle A_n \rangle$$

(value)
$V, W ::= c \mid \lambda x.M \mid \mathsf{rec}\ f\ x.M \mid \langle \overline{\ell = V} \rangle \mid [\overline{V}] \mid \texttt{<@}\ Q\ \texttt{@>}$
(quotation value)
$Q ::= x \mid c \mid op(\overline{Q}) \mid \lambda x.Q \mid Q\ Q'$
$\mid \mathsf{if}\ Q\ Q' \mid \langle \overline{\ell = Q} \rangle \mid Q.\ell$
$\mid [\,] \mid [Q] \mid Q +\!\!+ Q' \mid \mathsf{for}\ (x \leftarrow Q)\ Q' \mid \mathsf{table}\ t$
(evaluation context)
$\mathcal{E} ::= [\ ] \mid op(\overline{V}, \mathcal{E}, \overline{M}) \mid \mathsf{lift}\ \mathcal{E} \mid \mathcal{E}\ M \mid V\ \mathcal{E}$
$\mid \langle \overline{\ell = V}, \ell' = \mathcal{E}, \overline{\ell'' = M} \rangle \mid \mathcal{E}.\ell \mid [\mathcal{E}]$
$\mid \mathcal{E} +\!\!+ M \mid [V] +\!\!+ \mathcal{E} \mid \mathsf{for}\ (x \leftarrow \mathcal{E})\ N$
$\mid \mathsf{if}\ \mathcal{E}\ M \mid \mathsf{query}\ \mathcal{E} \mid \texttt{<@}\ \mathcal{Q}[(\%\mathcal{E})]\ \texttt{@>}$
(quotation context)
$\mathcal{Q} ::= [\ ] \mid op(\overline{Q}, \mathcal{Q}, \overline{M}) \mid \lambda x.\mathcal{Q}$
$\mid \mathcal{Q}\ M \mid Q\ \mathcal{Q} \mid \langle \overline{\ell = Q}, \ell' = \mathcal{Q}, \overline{\ell'' = M} \rangle$
$\mid \mathcal{Q}.\ell \mid [\mathcal{Q}] \mid \mathcal{Q} +\!\!+ M \mid Q +\!\!+ \mathcal{Q}$
$\mid \mathsf{for}\ (x \leftarrow \mathcal{Q})\ N \mid \mathsf{for}\ (x \leftarrow Q)\ \mathcal{Q}$
$\mid \mathsf{if}\ \mathcal{Q}\ M \mid \mathsf{if}\ Q\ \mathcal{Q}$

Figure 9. Values and Evaluation Contexts for Quot

$$\begin{array}{rcll}
op(\overline{V}) & \longrightarrow & \delta(op, \overline{V}) & \\
(\lambda x.M)\ V & \longrightarrow & M[x := V] & \\
(\mathsf{rec}\ f\ x.M)\ V & \longrightarrow & M[f := \mathsf{rec}\ f\ x.M, x := V] & \\
\langle \overline{\ell = V} \rangle.\ell_i & \longrightarrow & V_i & \\
\mathsf{if}\ \mathsf{true}\ M & \longrightarrow & M & \\
\mathsf{if}\ \mathsf{false}\ M & \longrightarrow & [\,] & \\
[\,] +\!\!+ V & \longrightarrow & V & \\
([V] +\!\!+ V') +\!\!+ V'' & \longrightarrow & [V] +\!\!+ (V' +\!\!+ V'') & \\
\mathsf{fold}\ [\,]\ M\ N & \longrightarrow & M & \\
\mathsf{fold}\ ([V]) +\!\!+ W)\ M\ N & \longrightarrow & N\ V\ (\mathsf{fold}\ W\ M\ N) & \\
\mathsf{for}\ (x \leftarrow [\,])\ N & \longrightarrow & [\,] & \\
\mathsf{for}\ (x \leftarrow [V] +\!\!+ W)\ N & \longrightarrow & N[x := V] +\!\!+ & \\
 & & (\mathsf{for}\ (x \leftarrow W)\ N) & \\
\mathsf{query}\ \texttt{<@}\ Q\ \texttt{@>} & \xrightarrow{(\|Q\|, V)} & V & ((\|Q\|, V) \in \Omega) \\
\mathsf{lift}\ c & \longrightarrow & \texttt{<@}\ c\ \texttt{@>} & (\text{lift}) \\
\texttt{<@}\ \mathcal{Q}[(\%\texttt{<@}\ Q\ \texttt{@>})]\ \texttt{@>} & \longrightarrow & \texttt{<@}\ \mathcal{Q}[Q]\ \texttt{@>} & (\text{splice})
\end{array}$$

$$\frac{M \xrightarrow{\mu} N}{\mathcal{E}[M] \xrightarrow{\mu} \mathcal{E}[N]}$$

Figure 10. Operational Semantics for Quot

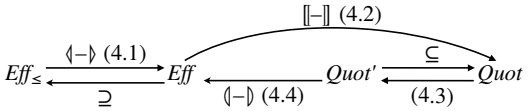

Figure 11. Summary of the Translations

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, the only 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.

$$
\begin{array}{rcl}
\langle Int \rangle & = & Int \\
\langle Bool \rangle & = & Bool \\
\langle String \rangle & = & String \\
\langle A \to^{X} B \rangle & = & \langle A \rangle \to^{X} \langle B \rangle \quad X \in \{\mathbf{db}, \mathbf{pl}\} \\
\langle A \to^{\mathbf{any}} B \rangle & = & \langle \langle A \rangle \to^{\mathbf{pl}} \langle B \rangle, \langle A \rangle \to^{\mathbf{db}} \langle B \rangle \rangle \\
\langle \langle \overline{\ell : A} \rangle \rangle & = & \langle \overline{\ell : \langle A \rangle} \rangle \\
\langle [A] \rangle & = & [\langle A \rangle]
\end{array}
$$

Figure 12. Doubling Translation: types

$$
\begin{array}{rcl}
\langle x \rangle_X & = & x \\
\langle c \rangle_X & = & c \\
\langle op(\overline{M}) \rangle_X & = & op(\langle \overline{M} \rangle_X) \\
\langle \lambda^{Y} x^{A}.M \rangle_X & = & \lambda^{Y} x^{\langle A \rangle}.\langle M \rangle_Y \quad Y \in \{\mathbf{db}, \mathbf{pl}\} \\
\langle \lambda^{\mathbf{any}} x^{A}.M \rangle_X & = & \langle \lambda^{\mathbf{pl}} x^{\langle A \rangle}.\langle M \rangle_{\mathbf{pl}}, \lambda^{\mathbf{db}} x^{\langle A \rangle}.\langle M \rangle_{\mathbf{db}} \rangle \\
\langle M^{A \to^{X} B}\ N \rangle_X & = & \langle M \rangle_X\ \langle N \rangle_X \\
\langle M^{A \to^{\mathbf{any}} B}\ N \rangle_{\mathbf{pl}} & = & \langle M \rangle_{\mathbf{pl}}.1\ \langle N \rangle_{\mathbf{pl}} \\
\langle M^{A \to^{\mathbf{any}} B}\ N \rangle_{\mathbf{db}} & = & \langle M \rangle_{\mathbf{db}}.2\ \langle N \rangle_{\mathbf{db}} \\
\langle \mathsf{if}\ L\ M \rangle_X & = & \mathsf{if}\ \langle L \rangle_X\ \langle M \rangle_X \\
\langle \langle \overline{\ell = M} \rangle \rangle_X & = & \langle \overline{\ell = \langle M \rangle_X} \rangle \\
\langle M.\ell \rangle_X & = & \langle M \rangle_X.\ell \\
\langle [] \rangle_X & = & [] \\
\langle [M] \rangle_X & = & [\langle M \rangle_X] \\
\langle M +\!\!+ N \rangle_X & = & \langle M \rangle_X +\!\!+ \langle N \rangle_X \\
\langle \mathsf{for}\ (x^{A} \leftarrow M)\ N \rangle_X & = & \mathsf{for}\ (x^{\langle A \rangle} \leftarrow \langle M \rangle_X)\ \langle N \rangle_X \\
\langle \mathsf{rec}\ f^{A \to^{\mathbf{pl}} B}\ x^{A}.M \rangle_{\mathbf{pl}} & = & \mathsf{rec}\ f^{\langle A \to^{\mathbf{pl}} B \rangle}\ x^{\langle A \rangle}.\langle M \rangle_{\mathbf{pl}} \\
\langle \mathsf{fold}\ L\ M\ N \rangle_{\mathbf{pl}} & = & \mathsf{fold}\ \langle L \rangle_{\mathbf{pl}}\ \langle M \rangle_{\mathbf{pl}}\ \langle N \rangle_{\mathbf{pl}} \\
\langle \mathsf{query}\ M \rangle_{\mathbf{pl}} & = & \mathsf{query}\ \langle M \rangle_{\mathbf{db}} \\
\langle \mathsf{table}\ t \rangle_{\mathbf{db}} & = & \mathsf{table}\ t
\end{array}
$$

Figure 13. Doubling Translation: terms

The term translation is parameterised by the concrete effect: **pl** or **db**. It is a straightforward structure-preserving traversal, except on lambda-abstractions and applications of **any** functions. Each **any**-function is translated to a pair of a **pl**-function and a **db**-function. To translate an application of an **any**-function, the target effect parameter is used to determine whether to use the first or second element of the pair before applying the corresponding function.

The main correctness properties are as follows:

THEOREM 5 (Type preservation). *If* $\Gamma \vdash M : A \,!\, E$ *and* $E \leq X$, *then* $\langle \Gamma \rangle \vdash \langle M \rangle_X : \langle A \rangle \,!\, X$.

THEOREM 6 (**any**-freedom). *If* $\Gamma \vdash M : A \,!\, E$ *and* $E \leq X$, *then judgement* $\langle \Gamma \rangle \vdash \langle M \rangle_X : \langle A \rangle \,!\, X$ *is derivable without* **any**.

The type preservation property is straightforward by structural induction on derivations. The **any**-freedom property clearly holds because the **any** effect appears nowhere on the right-hand-side of the definitions of the $\langle - \rangle_X$ functions. An appropriate semantic correctness property can also be shown, relating the operational semantics of $\mathsf{Eff}_{\leq}$ programs and their Eff translations.

THEOREM 7 (Semantics preservation). *Assume* $\vdash M : A \,!\, \mathbf{pl}$.

1. M *is a value if and only if* $\langle M \rangle_{\mathbf{pl}}$ *is a value.*
2. *If* $M \xrightarrow{\mu} N$ *then* $\langle M \rangle_{\mathbf{pl}} \overset{\mu}{\Longrightarrow} \langle N \rangle_{\mathbf{pl}}$.
3. *If* $\langle M \rangle_{\mathbf{pl}}$ *is reducible then there exists* N *such that* $\langle M \rangle_{\mathbf{pl}} \overset{\mu}{\Longrightarrow} \langle N \rangle_{\mathbf{pl}}$ *and* $M \xrightarrow{\mu} N$.

Moreover, if $\vdash M : A \,!\, \mathbf{db}$ *then* $|\langle M \rangle_{\mathbf{db}}| = \langle |M| \rangle_{\mathbf{db}}$.

COROLLARY 1. *For any* $\vdash M : A \,!\, \mathbf{pl}$ *in* $\mathsf{Eff}_{\leq}$ *we have* $M \approx \langle M \rangle_{\mathbf{pl}}$.

4.2 From Eff to Quot

We give an effect-directed translation from Eff to Quot. The interpretations of types, terms and typing environments are parameterised by the concrete effect: **pl** or **db**. The type translation is written $[\![A]\!]_X$, where X is **pl** or **db** and A is an Eff-type, and is defined in Figure 14. In the programming language, **db** function types are interpreted as quoted functions. In the database, **pl** function types are interpreted as the unit type, which (as we will show) suffices because **pl** functions can never be called in the database.

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

$$\lambda x.\mathsf{let}\ x_{\mathbf{db}} = \texttt{<@}\downarrow_A(x)\ \texttt{@>}\ \mathsf{in}\ \langle \texttt{<@}\ \lambda y.\langle (\texttt{\%}x_{\mathbf{db}}), y \rangle\ \texttt{@>}, x \rangle\ .$$

The special variables $x_{\mathbf{db}}$ and reification operation $\downarrow_A(-)$ 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 ρ, 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 ρ is a finite map from variable names x to pairs $A \,!\, X$ denoting the type A and effect X at which x was bound. We write ε for the empty effect environment and $\rho[x \mapsto A \,!\, X]$ for the extension of effect environment ρ with the mapping x to $A \,!\, X$.

We define a judgement $\Gamma \vdash \rho$ which states that effect environment ρ is compatible with type environment Γ.

$$\textsc{EmptyEnv}\ \frac{}{\cdot \vdash \varepsilon} \qquad \textsc{ExtendEnv}\ \frac{\Gamma \vdash \rho}{\Gamma, x : A \vdash \rho[x \mapsto A \,!\, X]}$$

The interpretation $[\![\rho]\!]_X$ of an effect environment ρ at effect X is defined as:

$$
\begin{array}{rcll}
[\![\varepsilon]\!]_X & = & \cdot & \\
[\![\rho[x \mapsto A \,!\, \mathbf{db}]]\!]_{\mathbf{db}} & = & [\![\rho]\!]_{\mathbf{db}}, x : [\![A]\!]_{\mathbf{db}} & \\
[\![\rho[x \mapsto A \,!\, \mathbf{pl}]]\!]_{\mathbf{pl}} & = & [\![\rho]\!]_{\mathbf{pl}}, x : [\![A]\!]_{\mathbf{pl}}, x_{\mathbf{db}} : [\![A]\!]_{\mathbf{db}} & \\
[\![\rho[x \mapsto A \,!\, Y]]\!]_X & = & [\![\rho]\!]_X & \text{if } X \neq Y
\end{array}
$$

which induces the interpretation of a type environment with respect to a compatible effect environment. Notice that the translation $[\![\rho]\!]_{\mathbf{pl}}$ introduces the shadow variables $x_{\mathbf{db}}$. If $\Gamma \vdash \rho$, then we define $[\![\Gamma]\!]^{\rho}_X$ as follows:

$$[\![\Gamma]\!]^{\rho}_X = [\![\rho]\!]_X$$

The term translation is defined as $[\![M]\!]^{\rho}_X$ in Figure 16, where M is an Eff expression, X is an effect **db** or **pl** called the *target* of the translation, and ρ 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 **pl** context, we bind an additional special variable $x_{\mathbf{db}}$ whose value is a quoted version of x; intuitively, we need this quoted value to translate any occurrences of x within a **db** context. This affects all of the variable binding cases of $[\![-]\!]^{\rho}_{\mathbf{pl}}$ and is explained in more detail below. Variables are coerced to effect X using the $x@_{\rho}X$ operation described below. Following the interpretation on types, $\lambda^{\mathbf{db}}$-abstractions are coerced to a **pl** target by trans-

$$
\begin{array}{rcl}
[\![Int]\!]_X &=& Int \\
[\![Bool]\!]_X &=& Bool \\
[\![String]\!]_X &=& String \\
[\![A \to^X B]\!]_X &=& [\![A]\!]_X \to [\![B]\!]_X \\
[\![A \to^{\mathbf{db}} B]\!]_{\mathbf{pl}} &=& \mathsf{Expr}\langle [\![A \to^{\mathbf{db}} B]\!]_{\mathbf{db}} \rangle \\
[\![A \to^{\mathbf{pl}} B]\!]_{\mathbf{db}} &=& \langle\rangle \\
[\![\langle \overline{\ell : A} \rangle]\!]_X &=& \langle \overline{\ell : [\![A]\!]_X} \rangle \\
[\![[A]]\!]_X &=& [[\![A]\!]_X]
\end{array}
$$

Figure 14. Splicing Translation: types

$$
\begin{array}{rcll}
x \mathbin{@_\rho} X &=& x & \text{if } \rho(x) = A\,!\,X \\
x \mathbin{@_\rho} \mathbf{pl} &=& error_{[\![A]\!]_{\mathbf{pl}}} & \text{if } \rho(x) = A\,!\,\mathbf{db} \\
x \mathbin{@_\rho} \mathbf{db} &=& (\%x_{\mathbf{db}}) & \text{if } \rho(x) = A\,!\,\mathbf{pl}
\end{array}
$$

$$
error_A = (\mathsf{rec}\ f^{\langle\rangle \to A}\ x^{\langle\rangle}.f\ x)\ \langle\rangle
$$

$$
\begin{array}{rcl}
\downarrow_O(M) &=& (\%\mathsf{lift}\ M) \\
\downarrow_{A \to^{\mathbf{pl}} B}(M) &=& \langle\rangle \\
\downarrow_{A \to^{\mathbf{db}} B}(M) &=& (\%M) \\
\downarrow_{\langle \overline{\ell : A} \rangle}(M) &=& \langle \overline{\ell = \downarrow_A(M.\ell)} \rangle \\
\downarrow_{[A]}(M) &=& \\
\multicolumn{3}{r}{(\%\mathsf{fold}\ M\ \texttt{<@}\ []\ \texttt{@>}\ (\lambda x.\lambda y.\texttt{<@}\ ([\downarrow_A(x)])\ +\!\!+\ (\%y)\ \texttt{@>}))}
\end{array}
$$

Figure 15. Splicing Translation: auxiliary functions

lating them to quoted lambda-abstractions, and $\lambda^{\mathbf{pl}}$-abstractions are coerced to a $\mathbf{db}$ target by translating them to unit values.

The coercion operation, written $x \mathbin{@_\rho} Y$, is defined in Figure 15. Given variable x of type $[\![A]\!]_X$ (where $\rho(x) = A\,!\,X$), it yields a corresponding term in Quot of type $[\![A]\!]_Y$. If $X = Y$ then coercion leaves x unchanged. If $X = \mathbf{db}$ and $Y = \mathbf{pl}$, then an error term of type $[\![A]\!]_{\mathbf{pl}}$ (implemented as a diverging term) results. This is sound (but not strictly necessary) because in a closed program a $\mathbf{db}$ variable can only be coerced to $\mathbf{pl}$ inside the body of a $\mathbf{pl}$ function bound in a $\mathbf{db}$ context, and such a function can never be applied. If $X = \mathbf{pl}$ and $Y = \mathbf{db}$, then we splice in the value of $x_{\mathbf{db}}$, the special variable bound to the *reified value* of x as a query term $\texttt{<@}\downarrow_A(x)\ \texttt{@>}$.

Reification is a type-directed operation that maps a term of type $[\![A]\!]_{\mathbf{pl}}$ to a corresponding term of type $[\![A]\!]_{\mathbf{db}}$. A term of base type is reified by lifting and splicing. (Both are needed because lifting coerces a value of base type to its quotation.) A $\mathbf{pl}$ function is reified as unit. A $\mathbf{db}$ 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* $\Gamma \vdash M : A\,!\,\mathbf{pl}$, *then* $[\![\Gamma]\!]^\rho_{\mathbf{pl}}, [\![\Gamma]\!]^\rho_{\mathbf{db}} \vdash [\![M]\!]^\rho_{\mathbf{pl}} : [\![A]\!]_{\mathbf{pl}}$.
2. *If* $\Gamma \vdash M : A\,!\,\mathbf{db}$, *then* $[\![\Gamma]\!]^\rho_{\mathbf{pl}}; [\![\Gamma]\!]^\rho_{\mathbf{db}} \vdash [\![M]\!]^\rho_{\mathbf{db}} : [\![A]\!]_{\mathbf{db}}$.

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

THEOREM 9 (Semantics preservation). *Assume* $\vdash M : A\,!\,\mathbf{pl}$.

1. *M is a value if and only if $[\![M]\!]^\varepsilon_X$ is a value.*
2. *If* $M \xrightarrow{\mu} N$, *then* $[\![M]\!]^\varepsilon_{\mathbf{pl}} \overset{\mu}{\Longrightarrow} [\![N]\!]^\varepsilon_{\mathbf{pl}}$.
3. *If $[\![M]\!]^\varepsilon_{\mathbf{pl}}$ is reducible then there exist N and μ such that* $[\![M]\!]^\varepsilon_{\mathbf{pl}} \overset{\mu}{\Longrightarrow} [\![N]\!]^\varepsilon_{\mathbf{pl}}$ *and* $M \xrightarrow{\mu} N$.

Moreover, if $\vdash M : A\,!\,\mathbf{db}$ *then* $\|[\![M]\!]^\varepsilon_{\mathbf{db}}\| = [\![|M|]\!]^\varepsilon_{\mathbf{db}}$.

$$
\begin{array}{rcl}
[\![x]\!]^\rho_X &=& x \mathbin{@_\rho} X \\
[\![c]\!]^\rho_X &=& c \\
[\![op(\overline{M})]\!]^\rho_X &=& op(\overline{[\![M]\!]^\rho_X}) \\
[\![\lambda^{\mathbf{pl}} x^A.M]\!]^\rho_{\mathbf{pl}} &=& \lambda x^{[\![A]\!]_{\mathbf{pl}}}. \\
&& \quad \mathsf{let}\ x_{\mathbf{db}} = \texttt{<@}\downarrow_A(x)\ \texttt{@>}\ \mathsf{in} \\
&& \quad [\![M]\!]^{\rho[x \mapsto A!\mathbf{pl}]}_{\mathbf{pl}} \\
[\![\lambda^{\mathbf{db}} x^A.M]\!]^\rho_{\mathbf{pl}} &=& \texttt{<@}\ \lambda x^{[\![A]\!]_{\mathbf{db}}}.[\![M]\!]^{\rho[x \mapsto A!\mathbf{db}]}_{\mathbf{db}}\ \texttt{@>} \\
[\![\lambda^{\mathbf{pl}} x^A.M]\!]^\rho_{\mathbf{db}} &=& \langle\rangle \\
[\![\lambda^{\mathbf{db}} x^A.M]\!]^\rho_{\mathbf{db}} &=& \lambda x^{[\![A]\!]_{\mathbf{db}}}.[\![M]\!]^{\rho[x \mapsto A!\mathbf{db}]}_{\mathbf{db}} \\
[\![M\ N]\!]^\rho_X &=& [\![M]\!]^\rho_X\ [\![N]\!]^\rho_X \\
[\![\mathsf{if}\ L\ M]\!]^\rho_X &=& \mathsf{if}\ [\![L]\!]^\rho_X\ [\![M]\!]^\rho_X \\
[\![\langle \overline{\ell = M} \rangle]\!]^\rho_X &=& \langle \overline{\ell = [\![M]\!]^\rho_X} \rangle \\
[\![M.\ell]\!]^\rho_X &=& [\![M]\!]^\rho_X.\ell \\
[\![[]]\!]^\rho_X &=& [] \\
[\![[M]]\!]^\rho_X &=& [[\![M]\!]^\rho_X] \\
[\![M +\!\!+ N]\!]^\rho_X &=& [\![M]\!]^\rho_X +\!\!+ [\![N]\!]^\rho_X \\
[\![\mathsf{for}\ (x^A \leftarrow M)\ N]\!]^\rho_{\mathbf{pl}} &=& \mathsf{for}\ (x^{[\![A]\!]_{\mathbf{pl}}} \leftarrow [\![M]\!]^\rho_{\mathbf{pl}}) \\
&& \quad (\mathsf{let}\ x_{\mathbf{db}} = \texttt{<@}\downarrow_A(x)\ \texttt{@>}\ \mathsf{in}\ [\![N]\!]^{\rho[x \mapsto A!\mathbf{pl}]}_{\mathbf{pl}}) \\
[\![\mathsf{for}\ (x^A \leftarrow M)\ N]\!]^\rho_{\mathbf{db}} &=& \mathsf{for}\ (x^{[\![A]\!]_{\mathbf{db}}} \leftarrow [\![M]\!]^\rho_{\mathbf{db}})[\![N]\!]^{\rho[x \mapsto A!\mathbf{db}]}_{\mathbf{db}} \\
[\![\mathsf{rec}\ f^{A \to^{\mathbf{pl}} B}\ x^A.M]\!]^\rho_{\mathbf{pl}} &=& \mathsf{rec}\ f^{[\![A \to^{\mathbf{pl}} B]\!]}\ x^{[\![A]\!]_{\mathbf{pl}}}. \\
&& \quad \mathsf{let}\ f_{\mathbf{db}} = \texttt{<@}\downarrow_{A \to^{\mathbf{pl}} B}(f)\ \texttt{@>}\ \mathsf{in} \\
&& \quad \mathsf{let}\ x_{\mathbf{db}} = \texttt{<@}\downarrow_A(x)\ \texttt{@>}\ \mathsf{in} \\
&& \quad [\![M]\!]^{\rho[f \mapsto (A \to^{\mathbf{pl}} B)!\mathbf{pl}, x \mapsto A!\mathbf{pl}]}_{\mathbf{pl}} \\
[\![\mathsf{rec}\ f^{A \to^{\mathbf{pl}} B}\ x^A.M]\!]^\rho_{\mathbf{db}} &=& \langle\rangle \\
[\![\mathsf{fold}\ L\ M\ N]\!]^\rho_{\mathbf{pl}} &=& \mathsf{fold}\ [\![L]\!]^\rho_{\mathbf{pl}}\ [\![M]\!]^\rho_{\mathbf{pl}}\ [\![N]\!]^\rho_{\mathbf{pl}} \\
[\![\mathsf{query}\ M]\!]^\rho_{\mathbf{pl}} &=& \mathsf{query}\ [\![M]\!]^\rho_{\mathbf{db}} \\
[\![\mathsf{table}\ t]\!]^\rho_{\mathbf{db}} &=& \mathsf{table}\ t
\end{array}
$$

Figure 16. Splicing Translation: terms

COROLLARY 2. *If* $\vdash M : A\,!\,\mathbf{pl}$ *in* Eff *then* $M \approx [\![M]\!]^\varepsilon_{\mathbf{pl}}$.

REMARK 10. *The translation could be simplified by optimising away or inlining unnecessary* let*-bindings of $x_{\mathbf{db}}$ variables. However, these simplifications complicate the correctness proof.*

4.3 From Quot to Quot′

In translating Quot to Eff, the translation of an antiquote presents a potential difficulty. It seems natural to translate an antiquoted Quot term $(\%M)$ into a corresponding Eff term with effect $\mathbf{db}$. The problem is that M can perform arbitrary computation including recursion. The solution is to hoist any spliced computation out of the containing quotation. We can always soundly hoist antiquoted computations out of quotations as they never depend on the inner Δ environment. Thus as a preprocessing step we replace all antiquoted terms with variables bound outside the scope of the containing quotation. For convenience, we also perform similar hoisting for lift and query expressions. The target language of this step, Quot′, is the restriction of Quot such that each antiquotation, query, or lift may only be applied to a variable or value.

The hoisting translation uses the let form (as usual) as syntactic sugar for a lambda application:

$$
\mathsf{let}\ x^A = M\ \mathsf{in}\ N \equiv (\lambda x^A.N)\ M
$$

Hoisting is defined by repeatedly applying the following rules

$$
\begin{array}{rcl}
\texttt{<@}\ \mathcal{Q}[\,(\%M)\,]\ \texttt{@>} &\longrightarrow& \mathsf{let}\ x = M\ \mathsf{in}\ \texttt{<@}\ \mathcal{Q}[\,(\%x)\,]\ \texttt{@>} \\
\mathsf{lift}\ M &\longrightarrow& \mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{lift}\ x \\
\mathsf{query}\ M &\longrightarrow& \mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{query}\ x
\end{array}
$$

where M is not a variable or a value, and x is a fresh variable. Note that the structure of quotation contexts ensures that the terms

$$
\begin{array}{rcl}
(\!|Int|\!)_X & = & Int \\
(\!|Bool|\!)_X & = & Bool \\
(\!|A \to B|\!)_X & = & (\!|A|\!)_X \to^X (\!|B|\!)_X \\
(\!|\langle \overline{\ell : A} \rangle|\!)_X & = & \langle \overline{\ell : (\!|A|\!)_X} \rangle \\
(\!|[A]|\!)_X & = & [(\!|A|\!)_X] \\
(\!|\mathsf{Expr}\langle A \rangle|\!)_X & = & \langle\rangle \to^{\mathbf{db}} (\!|A|\!)_{\mathbf{db}}
\end{array}
$$

Figure 17. Quot′ to Eff: types

$$
\begin{array}{rcl}
(\!|x|\!)_X & = & x \\
(\!|c|\!)_X & = & c \\
(\!|op(\overline{M})|\!)_X & = & op(\overline{(\!|M|\!)_X}) \\
(\!|\lambda x^A.M|\!)_X & = & \lambda^X x^{(\!|A|\!)_X}.(\!|M|\!)_X \\
(\!|M\ N|\!)_X & = & (\!|M|\!)_X\ (\!|N|\!)_X \\
(\!|\mathsf{if}\ L\ M|\!)_X & = & \mathsf{if}\ (\!|L|\!)_X\ (\!|M|\!)_X \\
(\!|\langle \overline{\ell = M} \rangle|\!)_X & = & \langle \overline{\ell = (\!|M|\!)_X} \rangle \\
(\!|M.\ell|\!)_X & = & (\!|M|\!)_X.\ell \\
(\!|[]|\!)_X & = & [] \\
(\!|[M]|\!)_X & = & [(\!|M|\!)_X] \\
(\!|M +\!\!+ N|\!)_X & = & (\!|M|\!)_X +\!\!+ (\!|N|\!)_X \\
(\!|\mathsf{for}\ (x^A \leftarrow M)\ N|\!)_X & = & \mathsf{for}\ (x^{(\!|A|\!)_X} \leftarrow (\!|M|\!)_X)\ (\!|N|\!)_X \\
(\!|\mathsf{lift}\ M|\!)_{\mathbf{pl}} & = & \lambda^{\mathbf{db}} x^{\langle\rangle}.(\!|M|\!)_{\mathbf{pl}} \\
(\!|\texttt{<@}\ M\ \texttt{@>}|\!)_{\mathbf{pl}} & = & \lambda^{\mathbf{db}} x^{\langle\rangle}.(\!|M|\!)_{\mathbf{db}} \\
(\!|\mathsf{query}\ M|\!)_{\mathbf{pl}} & = & \mathsf{query}\ ((\!|M|\!)_{\mathbf{pl}}\ \langle\rangle) \\
(\!|\mathsf{rec}\ f^{A \to B}\ x^A.M|\!)_{\mathbf{pl}} & = & \mathsf{rec}\ f^{(\!|A \to B|\!)_{\mathbf{pl}}}\ x^{(\!|A|\!)_{\mathbf{pl}}}.(\!|M|\!)_{\mathbf{pl}} \\
(\!|\mathsf{fold}\ L\ M\ N|\!)_{\mathbf{pl}} & = & \mathsf{fold}\ (\!|L|\!)_{\mathbf{pl}}\ (\!|M|\!)_{\mathbf{pl}}\ (\!|N|\!)_{\mathbf{pl}} \\
(\!|\mathsf{table}\ t|\!)_{\mathbf{db}} & = & \mathsf{table}\ t \\
(\!|\texttt{(\%}M\texttt{)}|\!)_{\mathbf{db}} & = & (\!|M|\!)_{\mathbf{pl}}\ \langle\rangle
\end{array}
$$

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.

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 **db** thunks.

Type environments are translated pointwise:

$$(\!|x_1 : A_1, \ldots, x_n : A_n|\!)_X = x_1 : (\!|A_1|\!)_X, \ldots, x_n : (\!|A_n|\!)_X$$

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 **db** thunks $\lambda^{\mathbf{db}} x^{\langle\rangle}.(\!|M|\!)_{\mathbf{db}}$ or $\lambda^{\mathbf{db}} x^{\langle\rangle}.(\!|M|\!)_{\mathbf{pl}}$ respectively. For lifting, note that it does not actually matter whether we use $(\!|-|\!)_{\mathbf{pl}}$ or $(\!|-|\!)_{\mathbf{db}}$ 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 **db** thunks by applying to unit.

REMARK 11. *The $(\!|-|\!)_X$ translation uses general rules for translating quotation, lifting, and antiquotation of arbitrary terms, but 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.*

THEOREM 12 (Type preservation).

1. *If $\Gamma \vdash M : A$, then $(\!|\Gamma|\!)_{\mathbf{pl}} \vdash (\!|M|\!)_{\mathbf{pl}} : (\!|A|\!)_{\mathbf{pl}}\ !\ \mathbf{pl}$.*
2. *If $\Gamma; \Delta \vdash M : A$, then $(\!|\Gamma|\!)_{\mathbf{pl}}, (\!|\Delta|\!)_{\mathbf{db}} \vdash (\!|M|\!)_{\mathbf{db}} : (\!|A|\!)_{\mathbf{db}}\ !\ \mathbf{db}$.*

THEOREM 13 (Semantics preservation). *Assume $\vdash M : A$.*

1. *If M is a value then $(\!|M|\!)_{\mathbf{pl}}$ is a value.*
2. *If $\Gamma \vdash M : A$ and $M \xrightarrow{\mu} N$, then $(\!|M|\!)_{\mathbf{pl}} (\overset{\mu}{\Longrightarrow} \cup \approx) (\!|N|\!)_{\mathbf{pl}}$.*
3. *If $\Gamma \vdash M : A$ and $(\!|M|\!)_{\mathbf{pl}}$ is reducible then there exists N and μ such that $(\!|M|\!)_{\mathbf{pl}} (\overset{\mu}{\Longrightarrow} \cup \approx) (\!|N|\!)_{\mathbf{pl}}$ and $M \xrightarrow{\mu} N$.*

Moreover, if $\cdot;\cdot \vdash Q : A$ then $|(\!|Q|\!)_{\mathbf{db}}| = (\!|\|Q\|\!)_{\mathbf{db}}$.

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 β-value reduction under a λ-abstraction in* Eff. *Fortunately, β-value equivalence is valid for* Eff *modulo $\approx$.*

COROLLARY 3. *If $\Gamma \vdash M : A$ in* Quot *then $M \approx (\!|M|\!)_{\mathbf{pl}}$.*

5. Application: Links query compilation

Holmes [17] previously developed a compiler for plain Links. Holmes' 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 Holmes' 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 $\mathsf{Eff}_{\leq}$ in some important respects: in particular, it employs row typing and effect polymorphism instead of subeffecting. 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 (I), 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 in-

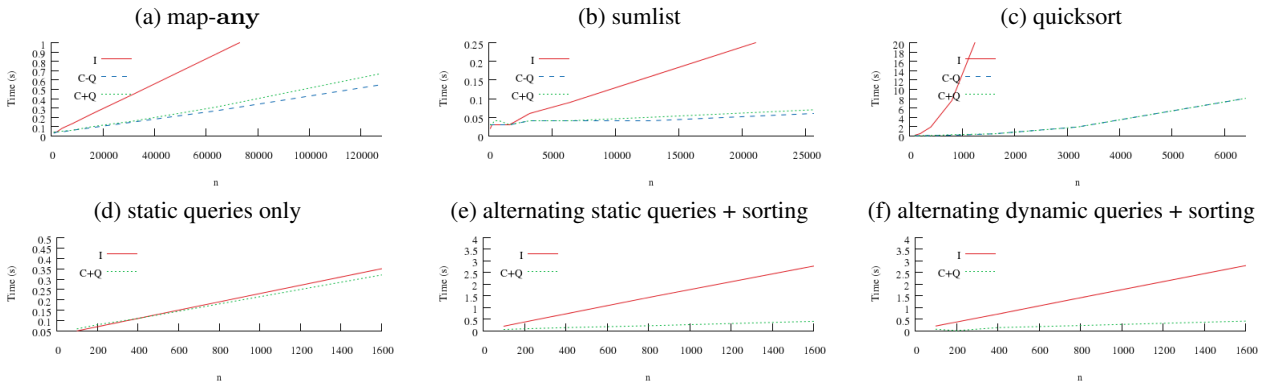

Figure 19. Experimental Results

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 Buneman 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 $\lambda^{\Box}$ 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 $(\!|-|\!)$ translation. Inoue and Taha [18] 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 labeled 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 $\mathsf{Eff}_{\leq}$ and Quot 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 $\mathbf{db}$ function in a lookup table indexed by the code pointer of the corresponding $\mathbf{pl}$ 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

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[3] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1), 1995.

[4] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.

[5] A. J. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010.

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

[7] 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.

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

[9] E. Cooper and P. Wadler. The RPC calculus. In *PPDP*, 2009.

[10] G. Copeland and D. Maier. Making Smalltalk a database system. *SIGMOD Rec.*, 14(2), 1984.

[11] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. N. Swadi, and W. Taha. Implicitly heterogeneous multi-stage programming. *New Generation Comput.*, 25(3):305–336, 2007.

[12] M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Programming*, 17:35–75, 1991.

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

[14] T. Goldschmidt, R. Reussner, and J. Winzen. A case study evaluation of maintainability and performance of persistency techniques. In *ICSE*, 2008.

[15] T. Grust and A. Ulrich. First-class functions for first-order database engines. In *DBPL*, 2013. `http://arxiv.org/abs/1308.0158`.

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

[17] S. Holmes. Compiling Links server-side code. Bachelor thesis, The University of Edinburgh, 2009.

[18] J. Inoue and W. Taha. Reasoning about multi-stage programs. In *ESOP*, pages 357–376, 2012.

[19] S. Lindley and J. Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, 2012.

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

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

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

[23] J. Mitchell. On abstraction and the expressive power of programming languages. *Sci. Comput. Programming*, 21:141–163, 1993.

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

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

[26] T. Petricek and D. Syme. The F# computation expression zoo. In *PADL*, 2014. To appear.

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

[28] D. Sangiorgi. *Introduction to bisimulation and coinduction*. Cambrudge University Press, 2012.

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

[30] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Apress, 2012.

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

[32] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. and Comput.*, 111(2), 1994.

[33] P. Wadler. Comprehending monads. *Math. Struct. in Comp. Sci.*, 2(4), 1992.

[34] P. Wadler and P. Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1), 2003.

[35] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16, September 2007.

[36] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL*, 2007.

[37] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996.

[38] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1), 2000.