Language-integrated queries: a BOLDR approach

Anonymous Author(s)

ABSTRACT

We present BOLDR, a modular framework that enables the evaluation in databases of queries containing application logic and, in particular, user-defined functions. BOLDR also allows the nesting of queries for different databases of possibly different data models. The framework detects the boundaries of queries present in an application, translates them into an intermediate representation together with the relevant language environment, rewrites them in order to avoid query avalanches and to make the most out of database optimizations, and converts the results back to the application. Our experiments show that the techniques we implemented are applicable to real-world database applications, successfully handling a variety of language-integrated queries with good performances.

1 INTRODUCTION

The increasing need for sophisticated data analysis encourages the use of programming languages that either better fit a specific task (e.g., R or Python for statistical analysis and data mining) or that manipulate specific data formats (e.g., JavaScript for JSON). It is therefore crucial for databases to support data analysis methods written in these languages. Currently, two opposite and incompatible solutions exist. One is to use a particular language supported by a specific database. For example, Oracle R Enterprise (Oracle 2017a), and PostgreSQL's PL/R for R; PL/Python (PostgreSQL 2017), its derivative Amazon Redshift (Amazon 2017), Hive (Apache 2017a), and SPARK (Apache 2017b) for Python; or MongoDB (MongoDB 2017) and Cassandra's CQL (Apache 2017c) for JavaScript. But this implies using a low-level and ad hoc API in which data is accessed by custom operations and yields code that is not portable. Contrary to this ad hoc approach, language-integrated querying, popularized with Microsoft's LINQ framework (Microsoft 2017), proposes to extend programming languages with builtin querying syntax and to represent externally-stored data in the data model of the language, thus shielding programmers from having to learn the specific syntax or data model aspects of databases. To that end, LINQ exposes the language to a set of *standard query operators* that external data providers must implement. However, LINQ suffers from a key limitation: queries can only execute if they can be translated into the set of query operators. For instance, the LINQ query

```
db.Employee.Where(x => x.sal >= 2000 * getRate("USD", x.cur))   (1)
```

which is intended to return the set of all employees having a salary (stored in some currency) greater than `2000 USD`, will throw an error at runtime since LINQ fails to translate the function `getRate` to an equivalent expression in the database. One solution is to mirror the definition of `getRate` in the database, but this hinders portability and may not be possible at all if the function references values present in the runtime of the language. A more common workaround is to rewrite the code as follows:

```
db.Employee.AsEnumerable()
           .Where(x => x.sal >= 2000 * getRate("USD", x.cur))
```

But this seemingly innocuous piece of code hides huge performance issues: all the data is imported in the runtime of the language, potentially causing important network delays and out-of-memory errors, and the filtering is evaluated in main memory thus neglecting all possible database optimizations.

In this work, we introduce BOLDR (**B**reaking boundaries **O**f **L**anguage and **D**ata **R**epresentations), a language-integrated query framework that allows arbitrary expressions from the *host language* (language from which the query comes from) to occur in queries and be evaluated in a database, thus lifting a key limitation of the existing solutions. Additionally, BOLDR is tied neither to a particular combination of database and programming language, nor to querying only one database at a time: for instance, BOLDR allows a NoSQL query targeting a HBase server to be nested in a SQL query targeting a relational database. BOLDR translates these queries into a **Q**uery **I**ntermediate **R**epresentation (or QIR for short), an untyped λ-calculus with data-manipulation builtin operators. Then, it applies a normalization process that may perform a partial evaluation of the QIR expression. This partial evaluation composes distinct queries that may occur separated in the code of the host language into larger queries that, after a further step of translation, are shipped to the database. The composition of distinct queries into a larger one not only reduces the communication overhead between the client runtime and the database but, above all, allows the database to perform whole query optimizations. Consider again our initial query (1) containing the call to `getRate`. In BOLDR, the translation of (1) produces a QIR expression according to four different scenarios: (*i*) if `getRate` can be translated in the query language of the targeted database, then the whole expression is translated into a single query expressed in the query language of the targeted database; (*ii*) if `getRate` cannot be entirely translated but contains one or several queries that can be translated, then the normalization process attempts to rewrite the QIR expression so it can be translated into fewer queries to be sent to the database; (*iii*) if `getRate` contains subqueries to one or several databases different from the targeted database of the outer query, then BOLDR produces the corresponding translated subqueries and send them to their respective databases, and combines the results at QIR level; (*iv*) if `getRate` cannot be translated at all, then BOLDR sends a query containing the serialized host language abstract syntax tree of `getRate` to be potentially executed on the database side.

Our implementation of BOLDR uses Truffle (Würthinger et al. 2013), a framework developed by Oracle Labs to implement programming languages. Several features make Truffle appealing to BOLDR: first, Truffle implementations of languages must compile to an executable abstract syntax tree (AST) that BOLDR can directly manipulate; second, languages implemented with Truffle can be executed on off-the-shelf JVMs, making their addition as an external language effortless in databases written in Java (e.g., Cassandra, HBase, ...), and relatively simple in others such as PostgreSQL. Third, the work done for one Truffle language can easily be transposed to other Truffle languages.

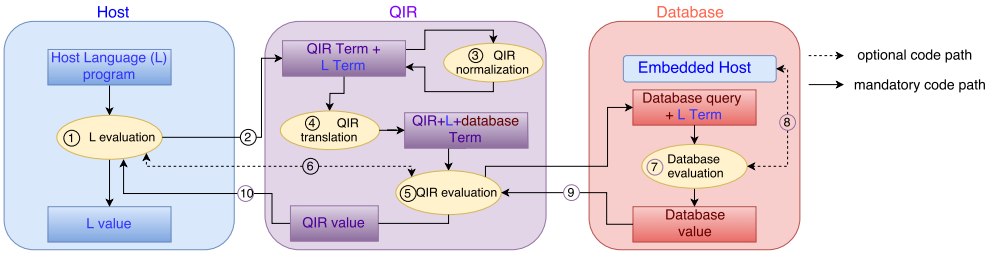


Figure 1: Evaluation of a BOLDR host language program

Our implementation currently supports the *PostgreSQL*, *HBase* and *Hive* databases, as well as *FastR* (Oracle 2017b) (Truffle implementation of the R language) and Oracle's *SimpleLanguage* (a simple dynamic language with syntax and features inspired by JavaScript). The following R program illustrates the key aspects of BOLDR:

```
1  # Exchange rate between rfrom and rto
2  getRate = function(rfrom, rto) {
3    # table change has three columns: cfrom, cto, rate
4    t = tableRef("change", "PostgreSQL")
5    if (rfrom == rto) 1
6       else subset(t, cfrom == rfrom && cto == rto, c(rate))
7  }
8
9  # Employees earning at least minSalary in the cur currency
10 atLeast = function(minSalary, cur) {
11   # table employee has two columns: name, sal
12   t = tableRef("employee", "PostgreSQL")
13   subset(t, sal >= minSalary * getRate("USD", cur), c(name))
14 }
15
16 richUSPeople = atLeast(2000, "USD")
17 richEURPeople = atLeast(2000, "EUR")
18 print(executeQuery(richUSPeople))
19 print(executeQuery(richEURPeople))
```

This example is a pure R program with two exceptions: the function `tableRef` (Line 4 and 12) referencing an external source in lieu of creating a data frame (R implementation of tables) from a text file; and the function `executeQuery` (Line 18 and 19) that forces the evaluation of queries. We recall that in R, the `c` function creates a vector, the `subset` function accepts a table, a logical expression as a filter, and optionally a vector of names of columns to extract. The first function `getRate` takes the code of two currencies and queries a table using the `subset` function to get their exchange rate. The second function `atLeast` takes as input a minimum salary and a currency code and retrieves the names of the employees earning at least the minimal salary. Since the salary is stored in dollars in the database, the `getRate` function is used to perform the conversion.

In BOLDR, `subset` is overloaded to build an intermediate query representation if applied on an external source reference. The first call to `atLeast(2000, "USD")` builds a query and captures the variables in the local scope. When `executeQuery` is called, then (*i*) the intermediate query is *normalized*, inlining all bound variables with their values; (*ii*) the normalized query is translated into a target database query (here SQL); and (*iii*) the resulting query is evaluated in the database and the results sent back. After normalization, the query generated for the first call on Line 16 is:

```
SELECT name FROM employee WHERE sal >= 2000 * 1
```

which is optimal, in the sense that a single SQL query is generated. The code generated for the second call is also optimal thanks to the interplay between lazy building of the query and normalization:

```
SELECT name FROM employee WHERE sal >= 2000 *
       (SELECT rate FROM change WHERE rfrom = "USD" AND rto = "EUR")
```

Thus, BOLDR not only allows the database to execute user-defined functions (UDFs) written in the syntax of the host language (the `getRate` call at Line 13), but also it creates fewer and larger queries, thus allowing a better exploitation of database optimizations and avoiding the "query avalanche" phenomenon (Grust et al. 2010).

While similar approaches exist (see Section 8 on related work), BOLDR outperforms them on UDFs that cannot be completely translated. For instance, consider:

```
1 getRate = function(rfrom, rto) {
2   cfrom = c("EUR", "EUR", "USD", "USD", "JPY", "JPY")
3   cto = c("USD", "JPY", "EUR", "JPY", "EUR", "USD")
4   rate = c(1.44, 129, 0.88, 114, 0.0077, 0.0088)
5   t = data.frame(cfrom, cto, rate)
6   if (rfrom == rto) 1
7     else subset(t, cfrom == rfrom && cto == rto, c(rate))
8 }
```

This function builds an in-memory data frame using the builtin function `data.frame`. BOLDR cannot translate it to QIR since it calls the underlying runtime, so instead it generates the following query:

```
SELECT name FROM table_employee WHERE
       sal >= 2000 * R.eval("@...", array("USD", "EUR"))
```

where the string `"@..."` is a reference to a closure for `getRate`.

Mixing different data sources is supported, although less efficiently. For instance, we can refer to an HBase table in the function `getRate` by replacing the second argument of `tableRef` in Line 4 of our example by `"HBase"`. BOLDR is still able to evaluate the query by sending a query to both the HBase and PostgreSQL database, and by executing in main memory what cannot be translated.

The general flow of query evaluation in BOLDR is described in Figure 1. During the evaluation ① of a host program, QIR nodes are lazily accumulated. Once the host runtime reaches a point where it needs to evaluate a QIR term, it requests this evaluation to the QIR runtime ②. The QIR term is normalized ③ in order to reduce the fragmentation of queries. Next, the QIR term is translated ④ in a new QIR term which contains native queries (e.g., in SQL) to send to databases. Each piece of this final term is then evaluated where it belongs, either in main-memory ⑤ or in a database ⑦. "Frozen" host language expressions occurring in these queries are evaluated either by the runtime of the host language that called the QIR evaluation ⑥ or in the runtime embedded in the target database ⑧, thus repeating the whole process (this happens to the subquery in Line 6 of our example). Results are then translated from the database to QIR ⑨, then from QIR to the host language ⑩.

Overview and Contributions. In this work, we introduce BOLDR,

a multi-language framework for integrated queries with a unique combination of features such as the possibility of executing user-defined functions in databases, of partially evaluating and merging distinct query fragments, and of defining single queries that operate on data from different data sources. Our technical developments are organized as follows. We first give a formal definition of QIR (Section 3). We then present the translation from QIR to query languages and focus on a translation from QIR to SQL, and a type system that statically ensures that well-typed queries translate into SQL and are avalanche-free (Section 4). We continue by presenting a normalization procedure on the QIR to optimize the translation of a query (Section 5). We finally describe the translation from a host language (namely R) to QIR (Section 6). We discuss experimental results (Section 7) of our implementation that handles the languages R and SimpleLanguage and the databases PostgreSQL, HBase and Hive. We show that queries generated by BOLDR perform on a par with hand-written ones, and that UDFs can be efficiently executed in an embedded runtime whenever they cannot be natively translated in the target database.

2 DEFINITIONS

We give some basic definitions we use throughout the presentation.

Definition 2.1 (Host language). A *host language* $\mathcal{H}$ is a 4-tuple $(\mathsf{E}_\mathcal{H}, \mathsf{I}_\mathcal{H}, \mathsf{V}_\mathcal{H}, \xrightarrow{\mathcal{H}})$ where:

- $\mathsf{E}_\mathcal{H}$ is a set of *syntactic expressions*
- $\mathsf{I}_\mathcal{H}$ is a set of *variables*, $\mathsf{I}_\mathcal{H} \subset \mathsf{E}_\mathcal{H}$
- $\mathsf{V}_\mathcal{H}$ is a set of *values*
- $\xrightarrow{\mathcal{H}} : 2^{\mathsf{I}_\mathcal{H} \times \mathsf{V}_\mathcal{H}} \times \mathsf{E}_\mathcal{H} \to 2^{\mathsf{I}_\mathcal{H} \times \mathsf{V}_\mathcal{H}} \times \mathsf{V}_\mathcal{H}$, is the *evaluation function*

We abstract a host language $\mathcal{H}$ by reducing it to its bare components: a syntax given by a set of expressions $\mathsf{E}_\mathcal{H}$, a set of variables $\mathsf{I}_\mathcal{H}$, and a set of values $\mathsf{V}_\mathcal{H}$. Lastly we assume that the semantics of $\mathcal{H}$ is given by a partial evaluation function $\xrightarrow{\mathcal{H}}$. This function takes as input an evaluation *environment* (a set of pairs of variables and values, ranged over by σ) and an expression and returns a new environment and a value resulting from the evaluation of the input expression. This entails that to integrate a host language we need to be able to manipulate syntactic expressions of the language, inspect and build *environments*, and have access to an *interpreter* for the language. We write $dom(\sigma)$ for the set of variables bound in σ and FV(e) for the set of free variables in an expression e.

Definition 2.2 (Database language). Given a host language $\mathcal{H}$, a *database language* $\mathcal{D}$ with support for $\mathcal{H}$ is a 4-tuple $(\mathsf{E}_\mathcal{D}, \mathsf{V}_\mathcal{D}, \mathsf{O}_\mathcal{D}, \xrightarrow{\mathcal{D}})$ where:

- $\mathsf{E}_\mathcal{D}$ is a set of *syntactic expressions*
- $\mathsf{V}_\mathcal{D}$ is a set of *values*
- $\mathsf{O}_\mathcal{D}$ is a set of *supported operator names*, equipped with an *arity* function $|\_| : \mathsf{O}_\mathcal{D} \to \mathbb{N}$.
- $\xrightarrow{\mathcal{D}} : 2^{\mathsf{I}_\mathcal{H} \times \mathsf{V}_\mathcal{H}} \times \mathsf{E}_\mathcal{D} \to 2^{\mathsf{I}_\mathcal{H} \times \mathsf{V}_\mathcal{H}} \times \mathsf{V}_\mathcal{D}$ is the *evaluation function*

Similarly to host languages, we abstract databases by identifying them with their query language $\mathcal{D}$ composed of a syntax $\mathsf{E}_\mathcal{D}$, a set of values $\mathsf{V}_\mathcal{D}$, and an evaluation function $\xrightarrow{\mathcal{D}}$ which takes as input a host language environment and a database expression and returns a new host language environment and a database expression. Such an evaluation function allows us to abstract the behavior of modern databases that support queries containing foreign function calls (e.g., calling R code from a SQL query). Last, but not least, a database language exposes the set $\mathsf{O}_\mathcal{D}$ of operators it supports. This set of operators will play a crucial role in building queries that can be efficiently executed by a database back-end.

3 QUERY INTERMEDIATE REPRESENTATION

3.1 Core calculus

In this section, we define our Query Intermediate Representation, a λ-calculus with recursive functions, constants, basic operations, data structures, data operators, and foreign language expressions.

Definition 3.1. Given a host language $\mathcal{H}$, a set of database languages $\mathbb{D}$, and a countable set of variables $\mathsf{I}_{\mathtt{QIR}}$, we define the set of QIR *expressions*, denoted by $\mathsf{E}_{\mathtt{QIR}}$ and ranged over by q, as the set of finite productions of the following grammar:

$$
\begin{aligned}
q ::= {} & x \mid \mathbf{fun}^x(x)\mathbf{\to} q \mid q\ q \mid c \mid op(q, \ldots, q) \mid \mathbf{if}\ q\ \mathbf{then}\ q\ \mathbf{else}\ q \\
& \mid \{\, l : q, \ldots, l : q \,\} \mid [\ ] \mid q :: q \mid q\ \mathbf{@}\ q \\
& \mid q \cdot l \mid q\ \mathbf{as}\ x :: x\ ?\ q : q \mid o\langle q, \ldots, q \mid q, \ldots, q \rangle \mid \blacksquare(\sigma, e)
\end{aligned}
$$

Besides lambda-terms, QIR expressions include constants (such as integers, strings, ...), and some builtin operations (arithmetic operations, ...). The data model consists of records and sequences. Records can be deconstructed through field projections. Sequences are constructed as usual and deconstructed by the *list matching* destructor whose four arguments are: the list to destruct, a simple pattern that binds the head and the tail of the list to variables, the term to evaluate (with the bound variables in scope) when the list is not empty, and the term to return when the list is empty. The new additions to these mundane constructs are *database operators* and *host language expressions*. A database operator $o\langle q_1 \ldots, q_n \mid q'_1, \ldots, q'_m \rangle$ is similar to the notion of operator in the relational algebra. We divide its arguments in two groups: the q_i expressions are called *configurations* and influence the behavior of the operator; the q'_i expressions are the sub-collections that are operated on. Finally, a host expression $\blacksquare(\sigma, e)$ is an opaque construct that contains an evaluation environment σ and an expression e of the host language. We use the following syntactic shortcuts:

- $[\, q_1, \ldots, q_n \,]$ stands for $q_1 :: \ldots :: q_n :: [\ ]$
- $\mathbf{fun}^f(x_1, \ldots, x_n)\mathbf{\to} q$ stands for $\mathbf{fun}^f(x_1)\mathbf{\to}(\ldots(\mathbf{fun}(x_n)\mathbf{\to} q))$
- $q\ (q_1, \ldots, q_n)$ stands for $(\ldots(q\ q_1)\ldots)\ q_n$

Functions can be defined recursively by using the recursion variable that indexes the **fun** keyword, that we omit when useless.

Definition 3.2 (Reduction rules). Let $\to^\delta \subset \mathsf{E}_{\mathtt{QIR}} \times \mathsf{E}_{\mathtt{QIR}}$ be a reduction relation for basic operators and $\to \subset \mathsf{E}_{\mathtt{QIR}} \times \mathsf{E}_{\mathtt{QIR}}$ be the reduction relation defined by:

$$
\begin{aligned}
(\mathbf{fun}^f(x)\mathbf{\to} q_1)\ q_2 &\to q_1\{f/\mathbf{fun}^f(x)\mathbf{\to} q_1,\ x/q_2\} \\
\mathbf{if\ true\ then}\ q_1\ \mathbf{else}\ q_2 &\to q_1 \\
\mathbf{if\ false\ then}\ q_1\ \mathbf{else}\ q_2 &\to q_2 \\
\{\ldots, l : q, \ldots\} \cdot l &\to q \\
[\ ]\ \mathbf{as}\ x :: y\ ?\ q_{list} : q_{empty} &\to q_{empty} \\
q_{head} :: q_{tail}\ \mathbf{as}\ x :: y\ ?\ q_{list} : q_{empty} &\to q_{list}\{x/q_{head},\ y/q_{tail}\}
\end{aligned}
$$

$$
\begin{aligned}
[\ ]\ \mathbf{@}\ q &\to q & (q_1 :: q_2)\ \mathbf{@}\ q_3 &\to q_1 :: (q_2\ \mathbf{@}\ q_3) \\
q\ \mathbf{@}\ [\ ] &\to q & (q_1 :: q_2)\ \mathbf{@}\ q_3 &\to q_1 :: (q_2\ \mathbf{@}\ q_3)
\end{aligned}
$$

where $q\{x_1/q_1, \ldots, x_n/q_n\}$ denotes the standard capture avoiding substitution. We define the reduction relation of QIR expressions as the context closure of the relation $\to^\delta \cup \to$.

Crucially, embedded host expressions as well as database operator applications whose arguments are all reduced are irreducible.

3.2 Extended semantics

We next define how to interface host languages and databases with QIR. We introduce the notion of *driver*, a set of functions that translate values from one world to another.

Definition 3.3 (Language driver). Let $\mathcal{H}$ be a host language. A *language driver* for $\mathcal{H}$ is a 3-tuple $({}^{\mathcal{H}}\overrightarrow{\mathsf{EXP}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{H}}, {}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}})$ of total functions such that:

- ${}^{\mathcal{H}}\overrightarrow{\mathsf{EXP}} : 2^{\mathsf{I}_{\mathcal{H}} \times \mathsf{V}_{\mathcal{H}}} \times \mathsf{E}_{\mathcal{H}} \to \mathsf{E}_{\mathsf{QIR}} \cup \{\Omega\}$ takes an environment and an $\mathcal{H}$ expression and translates the expression into a QIR expression.
- $\overrightarrow{\mathsf{VAL}}^{\mathcal{H}} : \mathsf{V}_{\mathsf{QIR}} \to \mathsf{V}_{\mathcal{H}} \cup \{\Omega\}$ translates QIR values to $\mathcal{H}$ values
- ${}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}} : \mathsf{V}_{\mathcal{H}} \to \mathsf{V}_{\mathsf{QIR}} \cup \{\Omega\}$ translates $\mathcal{H}$ values to QIR values.

where the special value Ω denotes failure to translate.

A host language driver must therefore be able to translate values from the host language to QIR (and vice versa), and provide a translation function to embed expressions of the host language into QIR terms. Likewise, we define a database driver as:

Definition 3.4. (Database driver) Let $\mathcal{D}$ be a database language. A *driver* for $\mathcal{D}$ is a 3-tuple $(\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{D}}, {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}})$ of total functions such that:

- $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}} : \mathsf{E}_{\mathsf{QIR}} \to \mathsf{E}_{\mathcal{D}}$ translates a QIR expression into a $\mathcal{D}$ expression.
- $\overrightarrow{\mathsf{VAL}}^{\mathcal{D}} : \mathsf{V}_{\mathsf{QIR}} \to \mathsf{V}_{\mathcal{D}} \cup \{\Omega\}$ translates QIR values to $\mathcal{D}$ values.
- ${}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}} : \mathsf{V}_{\mathcal{D}} \to \mathsf{V}_{\mathsf{QIR}} \cup \{\Omega\}$ translates $\mathcal{D}$ values to QIR values.

where the special value Ω denotes a failure to translate.

We are now equipped to define the semantics of QIR terms, extended to host expressions and database operators.

Definition 3.5 (Extended QIR semantics). Let $\mathcal{H}$ be a host language, $({}^{\mathcal{H}}\overrightarrow{\mathsf{EXP}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{H}}, {}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}})$ a driver for $\mathcal{H}$, $\mathcal{D}$ a database language, and $(\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{D}}, {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}})$ a driver for $\mathcal{D}$. We define the extended semantics $\sigma, q \twoheadrightarrow \sigma', q'$ of QIR by the following set of rules:

$$\frac{q \to q'}{\sigma, q \twoheadrightarrow \sigma, q'} \qquad \frac{\sigma \cup \sigma', e \xrightarrow{\mathcal{H}} \sigma'', w}{\sigma, \blacksquare(\sigma', e) \twoheadrightarrow \sigma'', {}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}}(w)}$$

$$\frac{\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}(o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle) = e \quad \sigma, e \xrightarrow{\mathcal{D}} \sigma', w \quad o \in \mathsf{O}_{\mathcal{D}} \quad e \neq \Omega \quad {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}}(w) \neq \Omega}{\sigma, o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle \twoheadrightarrow \sigma', {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}}(w)}$$

Since QIR acts as an intermediate language from a host language to a database language, the evaluation of QIR terms will always be initiated from the host language runtime. It is therefore natural for the extended semantics to evaluate a QIR term in a given host language environment. If this QIR term is pure, meaning it is neither a database operator nor a host language expression, then the simple semantics of Definition 3.2 is used to evaluate the term. The extended semantics of Definition 3.5 is instead used to evaluate impure terms. Host expressions are evaluated using the evaluation relation of the host language in the environment formed by the union of the current running environment and the captured environment. While this rule may seem strange from the point of view of statically typed and scoped languages, it allows us to simulate the behavior of most dynamic languages we encountered (in particular R, Python, and JavaScript) that allow a function to reference an undefined global variable as long as it is defined when the function is called. Last, but not least, the evaluation of a database operator consists in *(i)* finding a database language that supports this operator, *(ii)* use the database driver for that language to translate the QIR term into a native query and *(iii)* use the evaluation function of the database to evaluate the query. The results are then translated back into a QIR value.

At this stage, we have defined a perfectly viable Query Intermediate Representation in the form of a λ-calculus extended with data operators. When a QIR term is evaluated, subterms that consist of data operators are evaluated by the appropriate database driver and their results are returned to the host runtime. We next address the two following problems:

(1) How to create database drivers in practice?
(2) How to avoid query avalanches as much as possible?

4 DATABASE TRANSLATION

In this section, we describe how the implementer of a database driver can translate a QIR expression into a representation understandable by a target database. Here the difficulty is twofold: not only do we need to translate a QIR expression into an efficient query of a given database language, but we also need to do so in a context where QIR subterms targeting different database languages may co-exist in the same program. Also, we want to design the translation process so it can be seamlessly extended with new database drivers. To that end, we separate the translation in two phases. First, a *generic* translation that traverses QIR expressions recursively to determine the targeted query language, second, a *specific* translation that makes use of database drivers.

4.1 Generic translation

The goal of the generic translation is to produce a QIR term where some subexpressions have been translated into native database queries. Ideally, we want the whole QIR expression to be translated into a single database query, but this is not always possible and sometimes part of the query has to be evaluated in the client side (where the QIR runtime resides). The QIR evaluator therefore relies on two components. First, a "fallback" implementation of QIR operators using the QIR itself, that we dub MEM for in-memory evaluation. The MEM language is a trivial database language where the translations of values to and from the QIR are the identity function, and where some operators (namely, `Filter`, `Project`, and `Join`) are defined as plain QIR recursive functions. The full definition of MEM is straightforward and given in Appendix A. Second, to allow the QIR evaluator to send queries to a database and translate the results back into QIR values, we assume that for each supported database language $\mathcal{D} \in \mathbb{D}$, we have a basic QIR operator, $\mathtt{eval}^{\mathcal{D}}$ defined as:

$$\frac{\sigma, e \xrightarrow{\mathcal{D}} \sigma', v}{\sigma, \mathtt{eval}^{\mathcal{D}}(e) \twoheadrightarrow \sigma', {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}}(v)}$$

Notice that in the case of the MEM language, the operator $\mathtt{eval}^{\mathsf{MEM}}$ is simply the reduction of a QIR term.

$$\text{(db-op)}\ \frac{q'_i \rightsquigarrow e_i, \mathcal{D}, i \in 1..m \quad \overrightarrow{\mathsf{EXP}}^{\mathcal{D}}(o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle) = e \quad \mathcal{D} \in \mathbb{D} \setminus \mathsf{MEM} \quad e \neq \Omega}{o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle \rightsquigarrow e, \mathcal{D}}$$

$$\text{(fun)}\ \frac{q \rightsquigarrow e, \mathcal{D}}{\mathbf{fun}^f(x)\to q \rightsquigarrow \mathbf{fun}^f(x)\to \mathsf{eval}^{\mathcal{D}}(e), \mathsf{MEM}}$$

$$\text{(mem-op)}\ \frac{q'_i \rightsquigarrow e_i, \mathcal{D}^i, \in 1..m \qquad \begin{array}{c} o \in \mathsf{O}_{\mathsf{MEM}} \\ e_i \neq \Omega \end{array}}{o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle \rightsquigarrow \overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(o, q_1, \ldots, q_n, \mathsf{eval}^{\mathcal{D}^1}(e_1), \ldots, \mathsf{eval}^{\mathcal{D}^m}(e_m)), \mathsf{MEM}}$$

$$\text{(app)}\ \frac{q_1 \rightsquigarrow e_1, \mathcal{D}^1 \quad q_2 \rightsquigarrow e_2, \mathcal{D}^2}{q_1\ q_2 \rightsquigarrow (\mathsf{eval}^{\mathcal{D}^1}(e_1))\ (\mathsf{eval}^{\mathcal{D}^2}(e_2)), \mathsf{MEM}}$$

Figure 2: Some rules of the generic translation

The generic translation is given by the judgment $q \rightsquigarrow e, \mathcal{D}$ where $q \in \mathsf{E}_{\mathsf{QIR}}$ and $e \in \mathsf{E}_{\mathcal{D}} \cup \{\Omega\}$, whose meaning is that a QIR expression q can either be rewritten into an expression e of the language $\mathsf{E}_{\mathcal{D}}$ of the database $\mathcal{D}$ or fail when $e = \Omega$. An excerpt of the set of inference rules used to derive this judgment is given in Figure 2. We focus on the rules dealing with database operators. The main one, rule (db-op), states that given a database operator, if there exists a database language $\mathcal{D}$ distinct from the fallback language MEM such that all data arguments can be translated into expressions of $\mathcal{D}$, then if the specific translation $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}$ called on the operator yields a fully translated $\mathcal{D}$ expression e, then e is returned as a translation in $\mathcal{D}$. This rule may fail in two cases. First case, the specific translation for $\mathcal{D}$ could yield an error Ω even if all data arguments of the operator have been successfully translated into expressions of the same language $\mathcal{D}$. This can happen, for instance, when the operator is not supported by $\mathcal{D}$ or when the specific translation of a configuration q_i fails. Second case, the data arguments of the operator could have been translated to more than one database language. If the operator o at issue is one of the supported operators of MEM, then both cases are handled by the rule (mem-op): each translated subexpression e_i is wrapped in a call to the $\mathsf{eval}^{\mathcal{D}_i}$ operator and o is evaluated with its MEM semantics. All the other rules are bureaucratic and propagate the translation recursively to subterms.

4.2 Specific translation: SQL

We document how to define specific database translations using SQL as an example of a database language. QIR to SQL is an important translation as it allows BOLDR to target not only relational databases but also some distributed databases such as Apache Hive (Apache 2017a), Apache Spark (Apache 2017b), or Cassandra's CQL (Apache 2017c). We assume that the set of values for SQL, namely $\mathsf{V}_{\mathsf{SQL}}$, only contains basic constants (strings, numbers, Booleans, ...) and tables. A table is a pair of a schema, that is a n-tuple of column names associated to column types, and a multi-set of n-tuples of column names associated to values. The set of expressions $\mathsf{E}_{\mathsf{SQL}}$ is the set of syntactically valid SQL queries (sql 2016). The set of supported operators $\mathsf{O}_{\mathsf{SQL}}$ we consider is { `Filter`, `Project`, `Join`, `From`, `GroupBy`, `Sort` }. Due to space constraints, we describe these operators and the full translation from QIR to SQL in Appendix A and B. The translation from QIR to SQL is mostly straightforward. However, ensuring that it does not fail is challenging. Indeed, SQL is *not* Turing complete and relies on a flat data model: a SQL query should only deal with sequences of records whose fields have basic types. Another important aspect of this translation is that we want to avoid query avalanche by translating as many QIR expressions as possible.

We obtain both strong guarantees using an ad hoc SQL type system for QIR terms described in Figure 3. This type system is straightforward, but in accordance with the semantics of SQL we require applications of basic operators and conditional expressions to take as arguments and return expressions that have basic types B, and data operators to take as sources flat record lists. We also use a rule to type a flat record list as a base type since SQL allows the use of a table containing only one value (one line of one column) as a value. For instance, `(SELECT 1)+ 1` is allowed and returns `2`.

Note that we *do not* require the host language to be statically typed. Given a QIR term q, we ensure the following:

(1) if q can be typed with a type t in the SQL type system, then the reduction relation of Definition 3.2 terminates and yields a term q' that has type t.
(2) if a term q has type t in the SQL type system and is in normal form, then the generic translation of Figure 2 yields a single, syntactically correct SQL expression (using the translation of Appendix B).

We restrict the set of expressions of QIR by restricting functions to non-recursive functions, refusing untranslatable terms (such as list destructors) as well as host expressions since we limit ourselves to pure queries, and by restricting data operators to `Project`, `From`, `Filter`, `Join`, `GroupBy`, and `Sort`. What we obtain is a simply typed λ-calculus extended with records and sequences without recursive functions, which entails strong normalization. We also state an expected subject reduction theorem

Theorem 4.1 (Subject reduction). *Let $q \in \mathsf{E}_{\mathsf{QIR}}$ and Γ an environment from QIR variables to QIR types. If $\Gamma \vdash q : T$, and $q \to q'$, then $\Gamma \vdash q' : T$.*

and are now equipped to state our soundness of translation theorem

Theorem 4.2 (Soundness of translation). *Let $q \in \mathsf{E}_{\mathsf{QIR}}$ such that $\emptyset \vdash q : T$, $q \to^* v$, and v is in normal form. If $T \equiv B$ or $T \equiv R$ or $T \equiv R$ **list** then $v \rightsquigarrow \mathtt{s}$, SQL.*

On the technical side, we show that typable QIR terms have a very particular normal form imposed by their type, and that these normal forms can be translated into syntactically correct SQL expressions (see Appendix B).

5 QIR HEURISTIC NORMALIZATION

Our guarantees only hold for the translation of a QIR query targeting only a SQL database. However, in general, a QIR term may mix several databases or use features that escape the hypotheses

$$B ::= \texttt{string} \mid \texttt{int} \mid \texttt{bool} \mid \ldots$$
$$T ::= B \mid T \to T \mid T\ \textbf{list} \mid \{l : T, \ldots, l : T\}$$
$$R ::= \{l : B, \ldots, l : B\}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : T_1 \vdash q : T_2}{\Gamma \vdash \textbf{fun}(x) \to q : T_1 \to T_2} \qquad \frac{\Gamma \vdash q_1 : T_1 \to T_2 \quad \Gamma \vdash q_2 : T_1}{\Gamma \vdash q_1\ q_2 : T_2} \qquad \frac{}{\Gamma \vdash c : \texttt{typeof}(c)}$$

$$\frac{\Gamma \vdash q : T}{\Gamma \vdash q :: [\,] : T\ \textbf{list}} \qquad \frac{\Gamma \vdash q_1 : T \quad \Gamma \vdash q_2 : T\ \textbf{list}}{\Gamma \vdash q_1 :: q_2 : T\ \textbf{list}}\ (q_2 \neq [\,]) \qquad \frac{\Gamma \vdash q_1 : T\ \textbf{list} \quad \Gamma \vdash q_2 : T\ \textbf{list}}{\Gamma \vdash q_1\ \textbf{@}\ q_2 : T\ \textbf{list}}$$

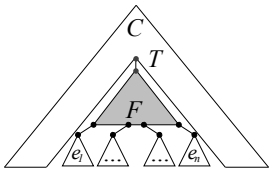
$$\frac{\Gamma \vdash op : B_1 \to \ldots \to B_n \to B \quad \Gamma \vdash b_i : B_i \quad i \in 1..n}{\Gamma \vdash op(b_1, \ldots, b_n) : B} \qquad \frac{\Gamma \vdash b_1 : \texttt{bool} \quad \Gamma \vdash b_2 : B \quad \Gamma \vdash b_3 : B}{\Gamma \vdash \textbf{if}\ b_1\ \textbf{then}\ b_2\ \textbf{else}\ b_3 : B} \qquad \frac{\Gamma \vdash q_i : T_i \quad i \in 1..n}{\Gamma \vdash \{\, l_1 : q_1, \ldots, l_n : q_n \,\} : \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\frac{\Gamma \vdash q_1 : R_2 \to R_1\ \Gamma \vdash q_2 : R_2\ \textbf{list}}{\Gamma \vdash \texttt{Project}\langle q_1 \mid q_2 \rangle : R_1\ \textbf{list}} \qquad \frac{\Gamma \vdash n : \texttt{string}}{\Gamma \vdash \texttt{From}\langle n \rangle : R\ \textbf{list}} \qquad \frac{\Gamma \vdash q_1 : R \to \texttt{bool} \quad \Gamma \vdash q_2 : R\ \textbf{list}}{\Gamma \vdash \texttt{Filter}\langle q_1 \mid q_2 \rangle : R\ \textbf{list}}$$

$$\frac{\begin{array}{l}\Gamma \vdash q_1 : R_3 \to R_4 \to R_1 \quad \Gamma \vdash q_3 : R_3\ \textbf{list} \\ \Gamma \vdash q_2 : R_3 \to R_4 \to \texttt{bool}\ \Gamma \vdash q_4 : R_4\ \textbf{list}\end{array}}{\Gamma \vdash \texttt{Join}\langle q_1, q_2 \mid q_3, q_4 \rangle : R_1\ \textbf{list}}$$

$$\frac{\Gamma \vdash q_1 : R_3 \to R_1\ \textbf{list} \quad \Gamma \vdash q_2 : R_1\ \textbf{list} \to R_2 \quad \Gamma \vdash q_3 : R_3\ \textbf{list}}{\Gamma \vdash \texttt{GroupBy}\langle q_1, q_2 \mid q_3 \rangle : R_2\ \textbf{list}} \qquad \frac{\Gamma \vdash q_1 : R_2 \to R_1 \quad \Gamma \vdash q_2 : R_2\ \textbf{list}}{\Gamma \vdash \texttt{Sort}\langle q_1 \mid q_2 \rangle : R_2\ \textbf{list}} \qquad \frac{\Gamma \vdash q : \{l : B, \ldots\}\ \textbf{list}}{\Gamma \vdash q : B} \qquad \frac{\Gamma \vdash q : \{l : T, \ldots\}}{\Gamma \vdash q \cdot l : T}$$

Figure 3: QIR type system for SQL



C: surrounding context

F: fragment

T: non-compatible expression

$e_1, \ldots, e_n$: non-compatible expressions

Figure 4: A fragment within a larger QIR term

of Theorem 4.2. In particular, outside these hypotheses, we cannot guarantee the termination of the normalization of a term. We are therefore stuck between two unsatisfactory options: either (*i*) trying to normalize the term (to fully reduce all applications) and yield the best possible term w.r.t. query translation but risk diverging in the process, or (*ii*) translate the term as-is at the risk of introducing query avalanches. We tackle this problem with a heuristic normalization procedure that tries to reduce QIR terms enough to produce a good translation by combining database query subtrees.

To that end, we define a measure of "good" QIR terms, and ask that each reduction step taken yields a term with a smaller measure. To formally define this measure, we first introduce a few concepts.

Definition 5.1 (Compatible data operator application). Let $\mathbb{D}$ be the set of database languages. A QIR data operator $o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m \rangle$ is a *compatible operator application* if and only if:

$\exists \mathcal{D} \in \mathbb{D}, e_1, \ldots, e_m \in \mathsf{E}_{\mathcal{D}}$ s.t. $\overrightarrow{\mathrm{EXP}}^{\mathcal{D}}(o, q_1, \ldots, q_n, e_1, \ldots, e_m) \neq \Omega$

Intuitively, a compatible data operator application is one where the configuration arguments are in a form that is accepted by the specific translation of the database language $\mathcal{D}$. We now define the related notion of *fragment*.

Definition 5.2 (Fragment). A fragment F is a subterm of a QIR term q such that $q = C[T(q_1, \ldots, q_{i-1}, F[e_1, \ldots, e_n], q_{i+1}, \ldots, q_j)]$ where C is a one-hole context made of arbitrary expressions; T is a non-compatible j-ary expression; $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_j$ and F are the children of T; F is an n-hole context made only of compatible operators applications of the same database language $\mathcal{D}$; and all $e_1, \ldots, e_n$ have head expressions that are not compatible.

Figure 4 gives a graphical representation of a fragment. We are now equipped to define a measure of "good" QIR terms.

Definition 5.3 (measure). Let $q \in \mathsf{E}_{\mathrm{QIR}}$ be a QIR expression, we define the measure of q as the pair

$$M(q) = (\mathrm{Op}(q) - \mathrm{Comp}(q), \mathrm{Frag}(q))$$

where $\mathrm{Op}(q)$ is the number of occurrences of data operators in q, $\mathrm{Comp}(q)$ is the number of occurrences of *compatible* data operator applications in q and $\mathrm{Frag}(q)$ is the number of fragments in q. The order associated with M is the lexicographic order on pairs.

This measure works as follows. During a step of reduction of a term q into a term q', q' is considered a better term either if the number of operators decreases, or if q' possesses more occurrences of *compatible* operator applications, meaning less cycles between QIR and the databases, or lastly, if the number of data operators does not change but the number of fragments decreases, meaning that some data operators were combined into a larger fragment.

Our heuristic-based normalization procedure uses this measure as a guide through the reduction of a QIR term: it applies all possible combinations of reduction steps to the term as long as its measure decreases after a number of steps fixed by heuristic. This allows us to generate a more efficient translation while ensuring termination.

Some practical choices impact the effectiveness of the QIR normalization such as the choice of which reduction rules to apply among all the possible ones (e.g., choosing those with more arguments), or which maximum number of steps to use. Extensive experiments for both points are detailed in a technical report [ANONYMIZED]. In particular, we measure that the normalization represents a negligible fraction of the execution time of the whole process, since it is dominated by those of other tasks (parsing, exchanges on the network with databases, …).

6 FROM A HOST LANGUAGE TO QIR

In this section we outline how to interface a general purpose programming language with BOLDR. Our aim is to allow programmers to write queries using constructs of the language they already master. In particular, we do not want to extend the syntax of the language and only allow ourselves to extend its runtime with some builtin functions. The full details of our treatment to the R language can be found in Appendix C.

We choose the R programming language as example of a host language and give an overview on how to implement a language driver for it. R programs include first-class functions; side effects ("=" being the assignment operator as well as the variable definition operator); sequences of expressions separated by ";" or a newline; structured data types such as vectors and tables with named

columns (called *data frames* in R's lingo); and static scoping as it is usually implemented in dynamic languages (e.g., as in Python or JavaScript) where identifiers that are not in the current static scope are assumed to be *global* identifiers even if they are undefined when the scope is created. For instance, the R program:

```
f = function (x) { x + y }; y = 3; z = f(2);
```

is well-defined and stores 5 in z (but calling f before defining y yields an error). We next define the core syntax of R.

Definition 6.1. The set E_R of expressions (e) and values (v) of R are generated by the following grammars:

$$\begin{aligned} e &::= c \mid \mathsf{x} \mid \mathsf{function}(x,\ldots,x)\{e\} \mid e(e,\ldots,e) \mid \mathsf{op}\ e\ \ldots\ e \\ &\quad \mid \mathsf{x} = e \mid e; e \mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \\ v &::= c \mid \mathsf{function}_\sigma(\mathsf{x},\ldots,\mathsf{x})\ \{e\} \mid \mathsf{c}(v,\ldots,v) \end{aligned}$$

where c represents constants, $\mathsf{x} \in I_R$, and $\sigma \in 2^{I_R \times V_R}$ is the environment of the closure.

We recall that $\mathsf{c}(e_1,\ldots,e_n)$ is used in R to build vectors. Definition 6.1 only defines expressions that can be translated to QIR. Expressions not listed in the definition are translated into host expression nodes.

We now highlight how queries are performed in (plain) R. While there are several ways to filter a data frame, the preferred one is the `subset` function we already used in the example in the introduction:

```
13  subset(t, sal >= minSalary * getRate("USD", cur), c(name))
```

This function returns the data frame given as first argument, filtered by the predicate given as second argument, and restricted to the columns listed in the third argument. Note that before resolving its second and third arguments, and for every row of the first argument, `subset` binds the values of each column of the row to a variable of the corresponding name. This is why in our example the variables `sal` and `name` occur free: they represent columns of the data frame `t`.

The join between two data frames is implemented with the function `merge`. We recall that the join operation returns the set of all combinations of rows in two tables that satisfy a given predicate.

To integrate R with BOLDR, we define two builtin functions:

- `tableRef` takes the name of a table and the name of the database the table belongs to, and returns a reference to the table
- `executeQuery` takes a QIR expression, closes it by binding its free variables to the translation to QIR of their value from the current R environment, sends it to the QIR runtime for evaluation, and translate the results into R values

We also extend the set of values V_R:

$$v ::= \ldots \mid \texttt{tableRef}(v,\ldots,v) \mid q_\sigma$$

where q_σ are *QIR closure values* representing queries associated with the R environment σ used at their definition.

The functions `subset` and `merge` (and other functions used to manipulate data frames) are overloaded to call the translation ${}^R\overrightarrow{\mathsf{EXP}}$ on themselves if their first argument is a reference to a database table created by `tableRef`, yielding a QIR term q to which the current scope is affixed, creating a "query closure" q_σ. At this stage, free variables in q that are not in σ can only be global identifiers whose bindings are to be resolved when the query is executed (when q_σ is given as argument to `executeQuery`).

We now illustrate the whole process on the introductory example of Section 1.

Evaluation of the query expression: When an expression recognized as a query is evaluated, it is translated to QIR (using Definition C.2). In the introductory example, the function call

```
16  richUSPeople = atLeast(2000, "USD")
```

triggers the evaluation of the function `atLeast`:

```
10  atLeast = function(minSalary, cur) {
11    # table employee has two columns: name, sal
12    t = tableRef("employee", "PostgreSQL")
13    subset(t, sal >= minSalary * getRate("USD", cur), c(name))
14  }
```

in which the function `subset` (Line 13) is evaluated with a table reference as first argument, and is therefore translated to a QIR expression. `richUSPeople` is then bound to the QIR closure value:

$$\begin{aligned} &\texttt{Project}\langle \textbf{fun}(t) \rightarrow \{\, name : t \cdot name \,\} \mid \\ &\texttt{Filter}\langle \textbf{fun}(e) \rightarrow e \cdot sal \geq minSalary * (getRate\ \texttt{"USD"}\ cur) \rangle \mid \\ &\texttt{From}\langle employee \rangle \rangle \rangle \\ &\{minSalary \mapsto 2000,\ getRate \mapsto \mathsf{function}_\sigma(\mathsf{rfrom},\mathsf{rto})\{\ldots\}, cur \mapsto \text{"USD"}\} \end{aligned}$$

Query execution: As mentioned earlier in this section, a query is executed when the function `executeQuery` is called with the corresponding QIR closure as argument. In our running example, this happens at Line 18 and 19:

```
18  print(executeQuery(richUSPeople))
19  print(executeQuery(richEURPeople))
```

At that point, the only free variables in the QIR closure are global variables. These are resolved by applying each variable to the translation to QIR of their value in the R environment:

$$\begin{aligned} &(\textbf{fun}(getRate) \rightarrow \\ &\quad (\textbf{fun}(minSalary, cur) \rightarrow \\ &\qquad \texttt{Project}\langle \textbf{fun}(t) \rightarrow \{\, name : t \cdot name \,\} \mid \\ &\qquad \texttt{Filter}\langle \textbf{fun}(e) \rightarrow \geq (e \cdot sal, *(minSalary, getRate\ \texttt{"USD"}\ cur)) \mid \\ &\qquad \texttt{From}\langle employee \rangle \rangle \rangle \\ &\quad)(2000, \texttt{"USD"}) \\ &)(\textbf{fun}(rfrom, rto) \rightarrow \ldots) \end{aligned}$$

Then, the evaluation engine for QIR terms is called, the query is normalized to:

$$\begin{aligned} &\texttt{Project}\langle \textbf{fun}(t) \rightarrow \{\, name : t \cdot name \,\} \mid \\ &\texttt{Filter}\langle \textbf{fun}(e) \rightarrow \geq (e \cdot sal, 2000) \mid \\ &\texttt{From}\langle employee \rangle \rangle \rangle \end{aligned}$$

and translated to SQL as:

```
SELECT T.name AS name FROM (
  SELECT * FROM (SELECT * FROM employee) AS E WHERE E.sal >= 2000
) AS T
```

This query is sent to the targeted database, and the results are translated back to QIR values (Steps ③ to ⑨ in Figure 1). Finally, the results are translated back to R using $\overrightarrow{\mathsf{VAL}}^{\mathcal{R}}$, then cached in the QIR closure to avoid evaluating the same query twice.

7 IMPLEMENTATION AND RESULTS

Implementation. The BOLDR framework consists of the QIR runtime, host languages, and target databases. To evaluate our approach, we have implemented the full stack, using R and SimpleLanguage as host languages and PostgreSQL, HBase and Hive as database back-ends. Table 1 gives the number of lines of Java code of each component to gauge the relative development effort needed to interface a Truffle-based host language or a database back-end to BOLDR. All developments are done in Java using the Truffle framework.

Component	l.o.c.	Remark
FastR / SimpleLanguage	*173000 / 12000*	not part of the framework
Detection of queries (in R and SL)	600	modification of builtins/operators
R to QIR / SL to QIR	750 / 1000	the translation of Definition C.2
QIR	4000	norm/generic translation/evaluator
QIR to SQL / HBase language	500 / 400	the translation $\overset{SQL}{\rightsquigarrow}$ / $\overset{HBase}{\rightsquigarrow}$
PostgreSQL / Hbase / Hive binding	150 / 100 / 100	low-level interface

Table 1: BOLDR components and their sizes in lines of code.

As expected, the bulk of our development work lies in the proper QIR component (its definition and normalization) which is completely shared between all languages and database backends. Compared to its 4000 l.o.c., the development cost of languages or database drivers, including translations to and from QIR is modest (between 400 and 1000 l.o.c.).

Even though our main focus is on Truffle-based languages, on which we have full control over their interpreters, all our requirements are also met by the introspection capabilities of modern dynamic languages. For instance, in standard R, the `environment` function allows to retrieve the environment affixed to a closure as a modifiable R value (and even replace the environment of a closure by a new one!), the `body` function returns the body of a closure as a manipulable abstract syntax tree, and the `formal` function returns the name of the formal parameters of a function. Using these introspection capabilities would be a way to achieve an even more seamless integration.

Experiments. The results of our evaluation[1] are reported in Table 2. Queries named TPCH-*n* are SQL queries taken from the TPC-H performance benchmark (TPC 2017). These queries feature joins, nested queries, grouping, ordering and various arithmetic subexpressions. The goal of this benchmark is to see how well a database engine optimizes these queries when generated by BOLDR. Table 2.**A** and 2.**B** illustrate how our approach fare against hand-written SQL queries. Each row reports the expected cost (in disk page fetches as reported by the `EXPLAIN ANALYZE` commands) as well as the actual execution time on a 1GB dataset. Row SQL represents the hand-written SQL queries from the benchmark, Row SQL+UDFs represents the same SQL queries where some subexpressions are expressed as function calls of stored functions written in PL/SQL. Row R represents the SQL queries from the benchmark generated by BOLDR from an equivalent R expression, and Row R+UDFs represents the same SQL queries as in Row SQL+UDFs generated by BOLDR from an equivalent R expression using R functions. Lastly, for row R+■, we added untranslatable subexpressions kept as host language nodes to impose a call to the database embedded R runtime. The results show that we can successfully match the performances of Row SQL with Row R, and that BOLDR outperforms PostgreSQL in Row R+UDFs against Row SQL+UDFs. This last result comes from the fact that PostgreSQL is not always able to inline function calls, even for simple functions written in PL/SQL. In stark contrast, no overhead is introduced for a SQL query generated from an R program, since the normalization is able to inline function calls properly, yielding a query as efficient as a hand-written one. As an example, the TPCH-15 query was written in R+UDFs as:

[1]The test machine was a PC with Ubuntu 16.04.2 LTS, kernel 4.4.0-83, with the latest master from the Truffle/Graal framework and PostgreSQL 9.5, Hive 2.1.1, and HBase 1.2.6 all with default parameters.

```
supplier = tableRef("supplier", "PostgreSQL", "postg.conf", "tpch")
revenue = tableRef("revenue", "PostgreSQL", "postg.conf", "tpch")
max_rev = function() max(subset(revenue, TRUE, c(total_revenue)))
q = subset(merge(supplier, revenue, function(x, y) x$s_suppkey ==
      y$supplier_no),
          total_revenue == max_rev(),
          c(s_suppkey, s_name, s_address, s_phone, total_revenue)
         )[order(s_suppkey), ]
print(executeQuery(q))
```

BOLDR was able to inline this query, whereas the equivalent in SQL+UDFs could not be inlined by the optimizer of PostgreSQL.

Table 2.B illustrates the overhead of calling the host language evaluator from PostgreSQL by comparing the cost of a *non-inlined* pure PL/SQL function with the cost of the same function embedded in a host expression within the query. While it incurs a high overhead, it remains reasonable even for expensive queries (such as TPCH-1) compared to the cost of network delays that would happen otherwise since host expressions represent expressions that are impossible to inline or to translate in the database language.

Table 2.C illustrates the overhead of calling the host language evaluator from Hive against a pure inlined Hive query. For instance

```
SELECT * FROM movie WHERE year > 1974 ORDER BY title
```

against

```
SELECT * FROM movie WHERE R.APPLY('@...', array(year))ORDER BY title
```

where `'@...'` is the serialization of an R closure, and `R.APPLY` is a function we defined that applies an R closure to an array of values from Hive (including the necessary translations between Hive, QIR, and R). The results are that with one (Query 1/2) or two (Query 3) calls to the external language runtime, the overhead is negligible compared to the execution of the query in Map/Reduce.

We give for completeness the performances of queries mixing two data sources between a PostgreSQL, a HBase, and an Hive database in Table 2.D. We executed the code of the example in the Introduction and varied the data sources for the functions `getRate` and `atLeast`. In the current implementation, the join between tables from different databases is made on the client side (see our future work in the Conclusion), therefore the queries in which the two functions target the same database perform better, since they are entirely evaluated in one database implying less network delays and less work on the client side.

8 RELATED WORK

The work in the literature closest to BOLDR is T-LINQ and P-LINQ by Cheney et al. (2013) which subsumes previous work on LINQ and Links and gives a comprehensive "practical theory of language integrated queries". In particular, it gives the strongest results to date for a language-integrated queries framework. Among their contributions stand out: (*i*) a quotation language (a λ-calculus with list comprehensions) used to express queries in a host language, (*ii*) a normalization procedure ensuring that no query can cause a query avalanche when translated in the target database, (*iii*) a type system which guarantees that well-typed queries can be normalized, (*iv*) a general recipe to implement language-integrated queries and (*v*) a practical implementation that outperforms Microsoft's LINQ standard. Some parts of our work are strikingly similar: our intermediate representation is a λ-calculus using reduction as a normalization procedure. However, our work diverges radically from their approach because we target a different kind

A	TPCH-1		TPCH-2		TPCH-3		TPCH-4		TPCH-5		TPCH-6		TPCH-9		TPCH-10		TPCH-11		TPCH-12		TPCH-13		TPCH-14		TPCH-15		TPCH-16		TPCH-18		TPCH-19	
	10^3 page fetches	time (s)																														
SQL	424	9.44	99	0.42	353	0.99	161	0.49	200	0.59	248	1.03	336	5.54	270	1.72	65	0.33	320	1.61	258	2.07	217	1.05	412	2.06	45	0.94	1462	4.33	306	1.31
SQL+UDFs	1753	15.50	655	0.43	426	2.21	424	0.79	617	1.69	1719	1.90	677	5.34	869	2.90	43	∞	1798	1.95	493	3.20	1766	5.26	207∗	∞	133	206.11	1118∗	∞	216	1.25
R	424	9.37	52	0.66	359	0.97	49939	0.53	200	0.76	248	1.02	300	3.20	272	1.73	65	0.31	334	1.37	258	2.03	217	0.97	412	2.00	39	0.41	2069	4.45	338	1.54
R+UDFs	424	9.51	52	0.68	359	0.95	49939	0.53	200	0.71	248	0.95	300	3.09	272	1.85	65	0.31	334	1.41	258	1.99	217	0.95	412	1.95	39	0.40	2069	4.35	338	1.58

B	TPCH-1		TPCH-3		TPCH-5		Ex. 1	
SQL+UDFs	1753	15.50	426	2.21	617	1.69	0.9	0.05
R+■	1720	37.56	97	2.01	471	8.99	0.9	0.11

C (in s)	Query 1	Query 2	Query 3
Hive	1.24	6.27	6.27
Hive+R	1.25	6.31	6.33

D (in s)	PostgreSQL (`atLeast`)	HBase (`atLeast`)	Hive (`atLeast`)
PostgreSQL (`getRate`)	0.34	1.47	1.17
HBase (`getRate`)	1.44	1.33	2.07
Hive (`getRate`)	0.74	1.78	0.66

∞: evaluation took more than 5 minutes. ∗ wrong cost estimation due to complex PL/SQL function

Table 2: Evaluation of BOLDR's performances

of host languages. T-LINQ requires a pure host language, with quotation and anti-quotation support and a type-system. Further, T-LINQ only supports one (type of) database per query and a very limited set of operators (essentially, selection, projection, and join, expressed as comprehensions). While definitely possible, extending T-LINQ with other operators (e.g., "group by" or "sort") or other data models (e.g., graph databases) seems challenging since their normalization procedure hard-codes in several places the semantics of SQL. The host languages we target do not lend themselves as easily to formal treatment, as they are highly dynamic, untyped, and impure programming languages. We designed BOLDR to be target databases agnostic, and to be easily extendable to support new languages and databases. We also endeavored to lessen the work of driver implementers (adding support for a new language or database) through the use of embedded host language expressions, which take advantage of the capability of modern databases to execute foreign code. This contrasts with LINQ where adding new back-ends is known to be a difficult task (Eini 2011). Last, we obtained formal results corresponding to those of T/P-LINQ by grafting a specific SQL type system on our framework.

QIR is not the first intermediate language of its kind. While LINQ proposes the most used intermediate query representation, recent work by Ong et al. (2014) introduced SQL++, an intermediary query representation whose goal is to subsume SQL and NoSQL query languages. In this work, a carefully chosen set of operators is shown to be sufficient to express relational queries as well as NoSQL queries (e.g., queries over JSON native databases). Each operator supports configuration options to account for the subtle differences in semantics for distinct query languages and data models (treatment of the special value `NULL`, semantics of basic operators such as equality, ...). In opposite, we chose to let the database expose the operators it supports in a driver.

Grust et al. (2010) present an alternative compilation scheme for LINQ, where SQL and XML queries are compiled into an intermediate *table algebra* expression that can be efficiently executed in any modern relational database. While this algebra supports diverse querying primitives, it is designed to specifically target SQL databases, making it unfit for other back-ends.

Our current implementation of the BOLDR framework is at an early stage and, as such, it suffers several shortcomings. Some are already addressed in existing literature. First, since we target dynamic programming languages, some forms of error cannot be detected until query evaluation. This problem has been widely studied and, besides the already cited T-LINQ, there are works such as SML# (Ohori and Ueno 2011) or ScalaDB (Garcia et al. 2010) where the static type system of the host language is used to ensure the absence of a large class of runtime errors in generated queries. Second, our treatment of effects is rather crude. Local side effects, such as updating mutable references scoped inside a query, work as expected while observable effects, such as reading from a file on host machine memory, is unspecified behavior. The work of Cook and Wiedermann (2011) shows how client-side effects can be re-ordered (to some extent) and split apart from queries. Third, at the moment, when two subqueries target different databases, their aggregation is done in the QIR runtime. Costa Seco et al. (2015) present a language which allows manipulation of data coming from different sources, abstracting their nature and localization. A drawback of their work is the limitation in the set of expressions that can be handled. Our use of arbitrary host expressions would allow us to circumvent this problem.

9 CONCLUSION AND FUTURE WORK

We presented BOLDR, a framework that allows programming languages to express complex queries containing application logic such as user-defined functions. These queries can then target any source of data as long as it is interfaced with the framework, more precisely, with our intermediate language QIR. We provided methods for programming languages and databases to interface with QIR, as well as an implementation of the framework and interfaces for R, SimpleLanguage, PostgreSQL, HBase, and Hive. We described how QIR reduces and partially evaluates queries in order to take the most of database optimizations, and showed that BOLDR generates queries performing on a par with hand-written SQL queries.

Future work includes the definition and implementation of a domain-specific language to define the translation from QIR to a database query representation, allowing implementers of an interface between QIR and a database to focus on the translation and leave the implementation details to the language itself, with the associated gains of speed, clarity, and concision. Currently, queries targeting more than one data sources (e.g., a join between tables from different databases) are partially executed in the host language runtime. We plan to determine when such queries could be executed efficiently in one of the targeted data sources instead (e.g., in a join between two distinct data sources, determine when it is convenient to send data from one data source to the other that will complete the join). ORMs and LINQ can type queries since they know the type of the data source. BOLDR cannot do it yet since QIR queries may contain highly dynamic code (in host expression nodes). While we cannot foresee any general solution (and our goal is *not* to type-check the whole host language), we believe that to exploit any type information available one should use *gradual typing* (Siek and Taha 2006), a recent technique blending static and dynamic typing in the same language. In particular, we would be able to use type information from database schemas (when available) to infer types for the queries.

REFERENCES

2016. ISO/IEC 9075-2:2016, Information technology-Database languages–SQL–Part 2: Foundation (SQL/Foundation), 2016. (2016).

Amazon 2017. Amazon Redshift Documentation - Python Language Support for UDFs. (2017). http://docs.aws.amazon.com/redshift/latest/dg/udf-python-language-support.html

Apache 2017a. Hive Manual - MapReduce scripts. (2017). https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Transform

Apache 2017b. PySpark documentation - pyspark.sql.functions. (2017). http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html

Apache 2017c. User Defined Functions in Cassandra 3.0. (2017). http://www.datastax.com/dev/blog/user-defined-functions-in-cassandra-3-0

J. Cheney, S. Lindley, and P. Wadler. 2013. A Practical Theory of Language-Integrated Query. In *ICFP 2013*. ACM, New York, NY, USA, 403–416.

W. R. Cook and B. Wiedermann. 2011. Remote Batch Invocation for SQL Databases. In *DBPL 2011, 13th International Symposium, Seattle, Washington, USA, August 29, 2011. Proceedings*, Nate Foster and Anastasios Kementsietsidis (Eds.). DBLP, Trier, Germany. http://www.cs.cornell.edu/conferences/dbpl2011/papers/dbpl11-cook.pdf

João Costa Seco, Hugo Lourenço, and Paulo Ferreira. 2015. A common data manipulation language for nested data in heterogeneous environments. In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, James Cheney and Thomas Neumann 0001 (Eds.). ACM, New York, NY, USA, 11–20. http://dl.acm.org/citation.cfm?id=2815072

O. Eini. 2011. The Pain of Implementing LINQ Providers. *Commun. ACM* 54, 8 (Aug. 2011), 55–61.

M. Garcia, A. Izmaylova, and S. Schupp. 2010. Extending Scala with Database Query Capability. In *Journal of Object Technology (JOT)*, Vol. 9, no. 4. DBLP, Trier, Germany, 45–68.

T. Grust, J. Rittinger, and T. Schreiber. 2010. Avalanche-Safe LINQ Compilation. In *Proceedings of the VLDB Endowment, Volume 3, September 2010 (VLDB 2010)*. VLDB Endowment, Singapore, 162–172.

Microsoft 2017. LINQ (Language-Integrated Query). (2017). https://msdn.microsoft.com/en-us/library/bb397926.aspx

MongoDB 2017. MongoDB User Manual - Server-side JavaScript. (2017). https://docs.mongodb.com/manual/core/server-side-javascript/

A. Ohori and K. Ueno. 2011. Making standard ML a practical database programming language. In *ICFP*. ACM, New York, NY, USA, 307–319.

K. W. Ong, Y. Papakonstantinou, and R. Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). http://arxiv.org/abs/1405.3631

Oracle 2017b. FastR. (2017). https://github.com/graalvm/fastr

Oracle 2017a. Oracle R Enterprise. (2017). http://www.oracle.com/technetwork/database/database-technologies/r

PostgreSQL 2017. PL/Python - Python Procedural Language. (2017). https://www.postgresql.org/docs/9.5/static/plpython.html

J. G. Siek and W. Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings, Scheme and Functional Programming Workshop 2006*. University of Chicago TR-2006-06, Chicago, USA, 81–92.

TPC. 2017. The TPC-H benchmark. (2017). http://www.tpc.org/tpch/

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 2013. One VM to Rule Them All. In *Onward! 2013 Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, New York, NY, USA, 187–204.

Appendices

A QIR AS A DATABASE LANGUAGE

An important design decision of any intermediate query representation is the set of *supported* database operators. The choice might prove to be difficult due to the following conflicting aspects:

- an operator may be specific to a particular data model (e.g., computing the transitive closure in a graph database);
- an operator may be generic enough but not natively supported by some back-ends (NoSQL databases usually do not support join operations).

The set of operators exposed by the intermediate representation should be broad enough so that the end users do not have to re-implement operators in host languages, and generic enough so that translating queries from the intermediate representation to a database representation stays manageable.

To experiment with this vast design space, we equip the QIR with a particular database language, dubbed MEM. MEM shares its values and expressions with QIR, and its data operators constitute a first attempt at defining a core set of operators that should be implemented by BOLDR even though some back-ends may not support them natively.

Definition A.1 (MEM database language). The *MEM language* $\mathsf{MEM} = (\mathsf{E}_{\mathsf{MEM}}, \mathsf{V}_{\mathsf{MEM}}, \mathsf{O}_{\mathsf{MEM}}, \overset{\mathsf{MEM}}{\longrightarrow})$ is defined by:

- $\mathsf{E}_{\mathsf{MEM}} = \mathsf{E}_{\mathsf{QIR}}$
- $\mathsf{O}_{\mathsf{MEM}} = \{\texttt{Filter}, \texttt{Project}, \texttt{Join}\}$
- $\mathsf{V}_{\mathsf{MEM}}$ is the set of finite terms generated by the grammar
  $v ::= \mathbf{fun}^x(x)\mathbf{\rightarrow} q \mid c \mid \{\, l : v, \ldots, l : v \,\} \mid [\,] \mid v :: v \mid \blacksquare(\sigma, e)$
- $\overset{\mathsf{MEM}}{\longrightarrow}$ is the operational semantics of the QIR (relation $\longrightarrow$ in Definition 3.2).

Of course the semantics of the MEM target language would be incomplete without the definition of a driver for MEM:

Definition A.2 (MEM driver). The driver for the MEM database language is the 3-tuple $(\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(\_), \overrightarrow{\mathsf{VAL}}^{\mathsf{MEM}}(\_), {}^{\mathsf{MEM}}\overrightarrow{\mathsf{VAL}}(\_))$ of total functions such that:

- $\overrightarrow{\mathsf{VAL}}^{\mathsf{MEM}}(\_) : \mathsf{V}_{\mathsf{QIR}} \to \mathsf{V}_{\mathsf{MEM}} \cup \{\Omega\}$ is the identity function.
- ${}^{\mathsf{MEM}}\overrightarrow{\mathsf{VAL}}(\_) : \mathsf{V}_{\mathcal{D}} \to \mathsf{V}_{\mathsf{QIR}} \cup \{\Omega\}$ is the identity function.
- $\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(\_) : \mathsf{E}_{\mathsf{QIR}} \to \mathsf{E}_{\mathsf{MEM}}$ is defined by case as

$\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(\texttt{Filter}\langle f \mid l\rangle) =$
$(\mathbf{fun}^{\mathsf{filter}}(l)\mathbf{\rightarrow} l\ \mathbf{as}\ h :: t\ ?\ \mathbf{if}\ f\ h\ \mathbf{then}\ h :: (\mathsf{filter}\ t)\ \mathbf{else}\ (\mathsf{filter}\ t) : [\,])\ l$

$\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(\texttt{Project}\langle f \mid l\rangle) =$
$(\mathbf{fun}^{\mathsf{project}}(l)\mathbf{\rightarrow} l\ \mathbf{as}\ h :: t\ ?\ (f\ h) :: (\mathsf{project}\ t) : [\,])\ l$

$\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(\texttt{Join}\langle f_1, f_2 \mid l_1, l_2\rangle) =$
$(\mathbf{fun}^{\mathsf{join}}(l)\mathbf{\rightarrow}\texttt{Project}\langle f_1 \mid l\ \mathbf{as}\ h_1 :: t_1 ?$
$\quad (\texttt{Project}\langle \mathbf{fun}(h_2)\mathbf{\rightarrow} h_1 \bowtie h_2 \mid \texttt{Filter}\langle f_2\ h_1 \mid l_2\rangle\rangle)\ @\ (\mathsf{join}\ t_1)$
$\quad : [\,])\ l_1\rangle$

The definition of the operators supported by MEM is straightforward and well-known to functional programmers. $\texttt{Filter}\langle f \mid l\rangle$ is implemented as a recursive function that iterates through an input list l and keeps elements for which the input predicate f returns **true**. $\texttt{Project}\langle f \mid l\rangle$ (also known as *map*) applies the function f to every element of l and returns the list of the outputs of f. Lastly the $\texttt{Join}\langle f_1, f_2 \mid l_1, l_2\rangle$ operator is defined as a double iteration which tests for each record element h_1 of l_1 and each record element h_2 of l_2 if the pair h_1, h_2 satisfies the join condition given by the function f_2, then the two records are concatenated and added to the result. Finally, the function f_1 is applied to every element to obtain the final result. For simplicity, we express `Join` in terms of `Project` and `Filter`, but we could have given a direct definition.

B QIR TO SQL

In this section, we give technical details of our translation from QIR to SQL. The set of supported operators of the translation is:

$$\mathsf{O}_{\mathsf{SQL}} = \{\texttt{Project}, \texttt{From}, \texttt{Filter}, \texttt{GroupBy}, \texttt{Sort}, \texttt{Join}, \texttt{Limit}\}$$

The semantics of `Filter`, `Project`, and `Join` was described in Section A. $\texttt{From}\langle n\rangle$ loads the contents of a table from its name, $\texttt{GroupBy}\langle f, agg \mid l\rangle$ partitions elements of l for which f returns the same value into groups, and returns the list of the results of applying agg to each group. $\texttt{Sort}\langle f \mid l\rangle$ sorts a collection l using a sorting key returned by applying f to each element of the list. The specific translation $\overrightarrow{\mathsf{EXP}}^{\mathsf{SQL}}$ is defined by the judgment $q \overset{\mathsf{SQL}}{\rightsquigarrow} e$ stating that a QIR expression q can be translated into a SQL expression e. The derivation of this judgment is given by the rules in Figure 5.

Data operators are translated into their SQL equivalent. Constants are translated using the translation function $\overrightarrow{\mathsf{VAL}}^{\mathsf{SQL}}$ provided by the driver, identifiers are translated as they are. Field access is handled by (SQL-field-simp) if the first argument is syntactically an identifier, and by (SQL-field-cplx) otherwise. Basic operators are translated into their SQL counterpart by rule (SQL-basic-op). Conditional expressions are translated into the corresponding `CASE` construct. Host language expressions are evaluated using a BOLDR-provided function that calls the evaluator of the host language in the database. Lastly, the (SQL-error) rule propagates errors and ensures that the whole translation fails if one of the sub-cases fails.

Next we show that the type-system defined in Section 4.2 statically detects QIR terms that can be soundly translated into SQL, that is, QIR terms in which all subterms can be handled by our specific translation. This property is formally stated by Theorem 4.2:

Let $q \in \mathsf{E}_{\mathsf{QIR}}$ such that $\emptyset \vdash q : T$, $q \longrightarrow^* v$, and v is in normal form. If $T \equiv B$ or $T \equiv R$ or $T \equiv R$ **list** then $v \rightsquigarrow \mathsf{s}, \mathsf{SQL}$.

To prove this theorem, we proceed in three steps. First, we show through the usual property of subject reduction that the normal form v has the same type T as q. Second, we show that normal forms of a given type have a particular shape. Lastly, we show by case analysis on the translation rules that the translation can never fail.

SUBJECT-REDUCTION. Let $q \in \mathsf{E}_{\mathsf{QIR}}$ and Γ an environment from QIR variables to QIR types. If $\Gamma \vdash q : T$, and $q \longrightarrow q'$, then $\Gamma \vdash q' : T$.

We prove the property by induction on the typing derivation.

- $(\mathbf{fun}^f(x)\mathbf{\rightarrow} q_1)\ q_2 \longrightarrow q_1\{f/\mathbf{fun}^f(x)\mathbf{\rightarrow} q_1,\ x/q_2\}$:

$$\frac{\dfrac{\Gamma, f : T' \to T, x : T' \vdash q_1 : T}{\Gamma \vdash \mathbf{fun}^f(x)\mathbf{\rightarrow} q_1 : T' \to T} \qquad \Gamma \vdash q_2 : T'}{\Gamma \vdash (\mathbf{fun}^f(x)\mathbf{\rightarrow} q_1)\ q_2 : T}$$

(SQL-var)
$$\frac{}{x \overset{\text{SQL}}{\rightsquigarrow} \texttt{X}}$$

(SQL-apply)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..2}{q_1\ q_2 \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT F(}e_2\texttt{) FROM (}e_1\texttt{) AS F}}$$

(SQL-cst)
$$\frac{}{c \overset{\text{SQL}}{\rightsquigarrow} \overrightarrow{\text{VAL}}^{\text{SQL}}(c)}$$

(SQL-basic-op)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..n}{o(q_1, \ldots, q_n) \overset{\text{SQL}}{\rightsquigarrow} o_{\text{SQL}}((e_1), \ldots, (e_n))}$$

(SQL-if)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..3}{\textbf{if } q_1 \textbf{ then } q_2 \textbf{ else } q_3 \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT CASE WHEN (}e_1\texttt{) THEN (}e_2\texttt{) ELSE (}e_3\texttt{) END}}$$

(SQL-lcons-empty)
$$\frac{q \overset{\text{SQL}}{\rightsquigarrow} e \quad e \neq \Omega \quad \text{TMP fresh}}{q :: [\,] \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_1\texttt{) AS TMP}}$$

(SQL-record)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..n}{\{\, l_1 : q_1, \ldots, l_n : q_n \,\} \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT (}e_1\texttt{) AS } X_1\texttt{, . . . , (}e_n\texttt{) AS } X_n}$$

(SQL-tdestr-cplx)
$$\frac{q \overset{\text{SQL}}{\rightsquigarrow} e \quad e \neq \Omega \quad \text{TMP fresh}}{q \cdot l \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT TMP.L FROM (SELECT (}e\texttt{)) AS TMP}}$$

(SQL-lcons)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..2 \quad \texttt{TMP, TMP2 fresh}}{q_1 :: q_2 \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_1\texttt{) AS TMP UNION ALL (}e_2\texttt{) AS TMP2}}$$

(SQL-lconcat)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..2 \quad \texttt{TMP, TMP2 fresh}}{q_1 \,@\, q_2 \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_1\texttt{) AS TMP UNION ALL (}e_2\texttt{) AS TMP2}}$$

(SQL-tdestr-simpl)
$$\frac{q \overset{\text{SQL}}{\rightsquigarrow} X}{q \cdot l \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT X.L}}$$

(SQL-project)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad i \in 1..2 \quad e \neq \Omega}{\texttt{Project}\langle \textbf{fun}(x) \rightarrow q_1 \mid q_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } e_1 \texttt{ FROM (}e_2\texttt{) AS X}}$$

(SQL-from)
$$\frac{}{\texttt{From}\langle\texttt{"table"}\rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM table}}$$

(SQL-filter)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad i \in 1..2 \quad e_i \neq \Omega}{\texttt{Filter}\langle \textbf{fun}(x) \rightarrow q_1 \mid q_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_2\texttt{) AS X WHERE (}e_1\texttt{)}}$$

(SQL-limit)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad i \in 1..2 \quad e_i \neq \Omega}{\texttt{Limit}\langle q_1 \mid q_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_2\texttt{) AS X LIMIT (}e_1\texttt{)}}$$

(SQL-group-by)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..3}{\texttt{GroupBy}\langle \textbf{fun}(x) \rightarrow q_1, \textbf{fun}(x) \rightarrow q_2 \mid q_3 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT (}e_2\texttt{) FROM (}e_3\texttt{) AS X GROUP BY (}e_1\texttt{)}}$$

(SQL-join)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..4}{\texttt{Join}\langle \textbf{fun}(x,y) \rightarrow q_1, \textbf{fun}(x,y) \rightarrow q_2 \mid q_3, q_4 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT (}e_2\texttt{) FROM (}e_3\texttt{) AS X, (}e_4\texttt{) AS Y WHERE (}e_1\texttt{)}}$$

(SQL-sort)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad i \in 1..2 \quad e_i \neq \Omega}{\texttt{Sort}\langle \textbf{fun}(x) \rightarrow q_1 \mid q_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_2\texttt{) AS X ORDER BY (}e_1\texttt{)}}$$

(SQL-host-expr)
$$\frac{}{\blacksquare(\sigma, e) \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT BOLDR.EVAL(}\blacksquare(\sigma, e)\texttt{)}}$$

Figure 5: Translation from QIR to SQL

- **if true then** q_1 **else** $q_2 \rightarrow q_1$:

$$\frac{\Gamma \vdash \textbf{true} : \texttt{bool} \quad \Gamma \vdash q_1 : T \quad \Gamma \vdash q_2 : T}{\Gamma \vdash \textbf{if true then } q_1 \textbf{ else } q_2 : T}$$

- **if false then** q_1 **else** $q_2 \rightarrow q_2$:

$$\frac{\Gamma \vdash \textbf{true} : \texttt{bool} \quad \Gamma \vdash q_1 : T \quad \Gamma \vdash q_2 : T}{\Gamma \vdash \textbf{if false then } q_1 \textbf{ else } q_2 : T}$$

- $[\,] \,@\, q \rightarrow q$:

$$\frac{\Gamma \vdash [\,] : T \quad \Gamma \vdash q : T}{\Gamma \vdash [\,] \,@\, q : T}$$

- $q \,@\, [\,] \rightarrow q$:

$$\frac{\Gamma \vdash q : T \quad \Gamma \vdash [\,] : T}{\Gamma \vdash q \,@\, [\,] : T}$$

- $(q_1 :: q_2) \mathbin{@} q_3 \longrightarrow q_1 :: (q_2 \mathbin{@} q_3)$:

$$\frac{\dfrac{\Gamma \vdash q_1 : T \quad \Gamma \vdash q_2 : T\ \mathbf{list}}{\Gamma \vdash q_1 :: q_2 : T\ \mathbf{list}} \quad \Gamma \vdash q_3 : T\ \mathbf{list}}{\Gamma \vdash (q_1 :: q_2) \mathbin{@} q_3 : T\ \mathbf{list}}$$

so:

$$\frac{\Gamma \vdash q_1 : T \quad \dfrac{\Gamma \vdash q_2 : T\ \mathbf{list} \quad \Gamma \vdash q_3 : T\ \mathbf{list}}{\Gamma \vdash q_2 \mathbin{@} q_3 : T\ \mathbf{list}}}{\Gamma \vdash q_1 :: (q_2 \mathbin{@} q_3) : T\ \mathbf{list}}$$

- $\{\ldots, l : q, \ldots\} \cdot l \longrightarrow q$:

$$\frac{\Gamma \vdash \{\ldots, l : q, \ldots\} : \{\ldots, l : T, \ldots\}}{\Gamma \vdash \{\ldots, l : q, \ldots\} \cdot l : T}$$

□

The next step is to show that a normal form of QIR has a particular shape depending on its type.

Definition B.1. A normal form v of QIR is a finite production of the following grammar:

v	::=	x	
	|	$\mathbf{fun}^x(x) \rightarrow v$	
	|	$v\ v$	first $v \not\equiv \mathbf{fun}^x(x) \rightarrow v$
	|	c	
	|	$op(v, \ldots, v)$	
	|	$\mathbf{if}\ v\ \mathbf{then}\ v\ \mathbf{else}\ v$	first $v \not\equiv \texttt{true}$ and first $v \not\equiv \texttt{false}$
	|	$\{l : v, \ldots, l : v\}$	
	|	$[\,]$	
	|	$v :: v$	
	|	$v \mathbin{@} v$	$v \not\equiv [\,]$ and first $v \not\equiv v :: v$
	|	$v \cdot l$	$v \not\equiv \{l : v, \ldots, l : v\}$
	|	$o\langle v, \ldots, v \mid v, \ldots, v\rangle$	

We now isolate a subset of normal forms that are translatable into SQL, that is a set of normal forms for which the translation succeeds.

Definition B.2 (Translatable normal forms). We define translatable normal forms as the finite terms produced by the following grammar:

s	::=	$r :: [\,] \mid r :: s \mid s \mathbin{@} s \mid \texttt{Project}\langle \mathbf{fun}^x(x) \rightarrow r \mid s\rangle$	b	::=	$\mathbf{true} \mid \mathbf{false} \mid 0 \mid 1 \mid \ldots$
	|	$\texttt{From}\langle b\rangle \mid \texttt{Filter}\langle \mathbf{fun}^x(x) \rightarrow b \mid s\rangle$		|	$\mathbf{if}\ b\ \mathbf{then}\ b\ \mathbf{else}\ b$
	|	$\texttt{Join}\langle \mathbf{fun}^x(x, x) \rightarrow r, \mathbf{fun}^x(x, x) \rightarrow b \mid s, s\rangle$		|	$x \cdot l$
	|	$\texttt{GroupBy}\langle \mathbf{fun}^x(x) \rightarrow s, \mathbf{fun}^x(x) \rightarrow r \mid s\rangle$		|	$op(b, \ldots, b)$
	|	$\texttt{Sort}\langle \mathbf{fun}^x(x) \rightarrow r \mid s\rangle$			
r	::=	$x \mid \{l : b, \ldots, l : b\}$			

Lemma B.3. *Let v be a normal form of QIR and Γ an environment from QIR variables to QIR types such that $\forall x \in \mathsf{dom}(\Gamma).\Gamma(x) \equiv R$, and $\Gamma \vdash v : T$, then:*

- *If $T \equiv B$ then $v \equiv b$ or $v \equiv s$*
- *If $T \equiv R$ then $v \equiv r$*
- *If $T \equiv R\ \mathbf{list}$ then $v \equiv s$*
- *If $T \equiv R \rightarrow B$ then $v \equiv \mathbf{fun}^x(x) \rightarrow b$*
- *If $T \equiv R \rightarrow R$ then $v \equiv \mathbf{fun}^x(x) \rightarrow r$*
- *If $T \equiv R \rightarrow R\ \mathbf{list}$ then $v \equiv \mathbf{fun}^x(x) \rightarrow s$*
- *If $T \equiv R\ \mathbf{list} \rightarrow R$ then $v \equiv \mathbf{fun}^x(x) \rightarrow r$*
- *If $T \equiv R \rightarrow R \rightarrow B$ then $v \equiv \mathbf{fun}^x(x) \rightarrow \mathbf{fun}^x(x) \rightarrow b$*
- *If $T \equiv R \rightarrow R \rightarrow R$ then $v \equiv \mathbf{fun}^x(x) \rightarrow \mathbf{fun}^x(x) \rightarrow r$*
- *If $T \equiv T \rightarrow T$ then $v \equiv \mathbf{fun}^x(x) \rightarrow v$*
- *If $T \equiv \{l : T, \ldots, l : T\}$ then $v \equiv x$ or $v \equiv \{l : v, \ldots, l : v\}$*

Proof.

Hypothesis 1 (H1). *v is in normal form*

Hypothesis 2 (H2). $\forall x \in \mathsf{dom}(\Gamma).\Gamma(x) \equiv R$

We prove the property by structural induction on the typing derivation of $\Gamma \vdash v : T$. We proceed by case analysis on T:

- If $T \equiv B$ then if the last typing rule used in the proof of $\Gamma \vdash v : B$ is the coercion rule, then $\Gamma \vdash v : \{l : B\}\ \mathbf{list}$, so by induction hypothesis $v \equiv s$ else
  - If $v = x$ then impossible since $\Gamma \vdash x : \Gamma(x) \equiv R$ by Hypothesis H2
  - If $v = \mathbf{fun}^f(x) \rightarrow v'$ then impossible since $\Gamma \vdash \mathbf{fun}^f(x) \rightarrow v' : T_1 \rightarrow T_2$
  - If $v = v_1\ v_2$ then by the typing rule of the application: $\Gamma \vdash v_1 : T_1 \rightarrow T_2$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \rightarrow v$, which is impossible by Hypothesis H1
  - If $v = c$ then $v \equiv b$
  - If $v = op(v_1, \ldots, v_n)$ then by the typing rule of operators: $\forall i \in 1..n, \Gamma \vdash v_i : B_i$, so by induction hypothesis $\forall i \in 1..n, v_i \equiv b$, so $v = op(v_1, \ldots, v_n) \equiv op(b, \ldots, b) \equiv b$
  - If $v = \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3$ then by the typing rule of the conditional expression: $\forall i \in 1..3, \Gamma \vdash v_i : B_i$, so by induction hypothesis $\forall i \in 1..3, v_i \equiv b$, so $v = \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3 \equiv \mathbf{if}\ b\ \mathbf{then}\ b\ \mathbf{else}\ b \equiv b$
  - If $v = \{l_1 : v_1, \ldots, l_n : v_n\}$ then impossible since $\Gamma \vdash \{l_1 : v_1, \ldots, l_n : v_n\} : \{l_1 : T_1, \ldots, l_n : T_n\}$
  - If $v = [\,]$ then impossible since $[\,]$ cannot be typed
  - If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash v_1 :: v_2 : R\ \mathbf{list}$
  - If $v = v_1 \mathbin{@} v_2$ then impossible since $\Gamma \vdash v_1 \mathbin{@} v_2 : R\ \mathbf{list}$
  - If $v = v' \cdot l$ then by the typing rule of the record destructor: $\Gamma \vdash v' : \{l_1 : T_1, \ldots, l_n : T_n\}$, so by induction hypothesis either $v' \equiv \{l : v, \ldots, l : v\}$, which is impossible by Hypothesis H1, or $v' \equiv x$, then $v = v' \cdot l \equiv x \cdot l \equiv b$
  - If $v = \texttt{Project}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Project}\langle v_1 \mid v_2\rangle : R\ \mathbf{list}$
  - If $v = \texttt{From}\langle v'\rangle$ then impossible since $\Gamma \vdash \texttt{From}\langle v'\rangle : R\ \mathbf{list}$
  - If $v = \texttt{Filter}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Filter}\langle v_1 \mid v_2\rangle : R\ \mathbf{list}$
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4\rangle$ then impossible since $\Gamma \vdash \texttt{Join}\langle v_1, v_2 \mid v_3, v_4\rangle : R\ \mathbf{list}$
  - If $v = \texttt{GroupBy}\langle v_1, v_2 \mid v_3\rangle$ then impossible since $\Gamma \vdash \texttt{GroupBy}\langle v_1, v_2 \mid v_3\rangle : R\ \mathbf{list}$
  - If $v = \texttt{Sort}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Sort}\langle v_1 \mid v_2\rangle : R\ \mathbf{list}$

- If $T \equiv R$ then
  - If $v = x$ then $v \equiv r$
  - If $v = \mathbf{fun}^f(x)\rightarrow v'$ then impossible since $\Gamma \vdash \mathbf{fun}^f(x)\rightarrow v' : T_1 \rightarrow T_2$
  - If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv B$
  - If $v = c$ then impossible since $\Gamma \vdash c : \texttt{typeof}(c) \equiv B$
  - If $v = op(v_1, \ldots, v_n)$ then impossible since $\Gamma \vdash op(v_1, \ldots, v_n) : B$
  - If $v = \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3$ then impossible since $\Gamma \vdash \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3 : B$
  - If $v = \{\, l_1 : v_1, \ldots, l_n : v_n \,\}$ then by the typing rule of the record constructor $\forall i \in 1..n, \Gamma \vdash v_i : B_i$, so by induction hypothesis $\forall i \in 1..n, v_i \equiv b$, so $v = \{\, l_1 : v_1, \ldots, l_n : v_n \,\} \equiv \{\, l : b, \ldots, l : b \,\} \equiv r$
  - If $v = [\ ]$ then impossible since $[\ ]$ cannot be typed
  - If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash v_1 :: v_2 : R\ \mathbf{list}$
  - If $v = v_1\ \mathbf{@}\ v_2$ then impossible since $\Gamma \vdash v_1\ \mathbf{@}\ v_2 : R\ \mathbf{list}$
  - If $v = v' \cdot l$ then by the typing rule of the record destructor: $\Gamma \vdash v' : \{l_1 : T_1, \ldots, l_n : T_n\}$, so by induction hypothesis either $v' \equiv \{\, l : v, \ldots, l : v \,\}$, which is impossible by Hypothesis H1, or $v' \equiv x$, but then by Hypothesis H2 $\Gamma \vdash v' \equiv x : \Gamma(x) \equiv R'$, so impossible since by the typing rule of the record destructor $\Gamma \vdash v = v' \cdot l : B$
  - If $v = \texttt{Project}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash \texttt{Project}\langle v_1 \mid v_2 \rangle : R\ \mathbf{list}$
  - If $v = \texttt{From}\langle v' \rangle$ then impossible since $\Gamma \vdash \texttt{From}\langle v' \rangle : R\ \mathbf{list}$
  - If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash \texttt{Filter}\langle v_1 \mid v_2 \rangle : R\ \mathbf{list}$
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then impossible since $\Gamma \vdash \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle : R\ \mathbf{list}$
  - If $v = \texttt{GroupBy}\langle v_1, v_2 \mid v_3 \rangle$ then impossible since $\Gamma \vdash \texttt{GroupBy}\langle v_1, v_2 \mid v_3 \rangle : R\ \mathbf{list}$
  - If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash \texttt{Sort}\langle v_1 \mid v_2 \rangle : R\ \mathbf{list}$
- If $T \equiv R\ \mathbf{list}$ then
  - If $v = x$ then impossible since $\Gamma \vdash x : \Gamma(x) \equiv R$ by Hypothesis H2
  - If $v = \mathbf{fun}^f(x)\rightarrow v'$ then impossible since $\Gamma \vdash \mathbf{fun}^f(x)\rightarrow v' : T_1 \rightarrow T_2$
  - If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv B$
  - If $v = c$ then impossible since $\Gamma \vdash c : \texttt{typeof}(c) \equiv B$
  - If $v = op(v_1, \ldots, v_n)$ then impossible since $\Gamma \vdash op(v_1, \ldots, v_n) : B$
  - If $v = \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3$ then impossible since $\Gamma \vdash \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3 : B$
  - If $v = \{\, l_1 : v_1, \ldots, l_n : v_n \,\}$ then impossible since $\Gamma \vdash \{\, l_1 : v_1, \ldots, l_n : v_n \,\} : \{l_1 : T_1, \ldots, l_n : T_n\}$
  - If $v = [\ ]$ then impossible since $[\ ]$ cannot be typed
  - If $v = v_1 :: v_2$ then by the typing rule of the list constructor: $\Gamma \vdash v_1 : R$ and $\Gamma \vdash v_2 : R\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv r$ and $v_2 \equiv s$, so $v = v_1 :: v_2 \equiv r :: s \equiv s$
  - If $v = v_1\ \mathbf{@}\ v_2$ then by the typing rule of the list concatenation: $\Gamma \vdash v_1 : R\ \mathbf{list}$ and $\Gamma \vdash v_2 : R\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv s$ and $v_2 \equiv s$, so $v = v_1\ \mathbf{@}\ v_2 \equiv s\ \mathbf{@}\ s \equiv s$
  - If $v = v' \cdot l$ then impossible for the same argument as for $T \equiv R$
  - If $v = \texttt{Project}\langle v_1 \mid v_2 \rangle$ then by the typing rule of Project: $\Gamma \vdash v_1 : R' \rightarrow R$ and $\Gamma \vdash v_2 : R'\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x)\rightarrow r$ and $v_2 \equiv s$, so $v = \texttt{Project}\langle v_1 \mid v_2 \rangle \equiv \texttt{Project}\langle \mathbf{fun}^x(x)\rightarrow r \mid s \rangle \equiv s$
  - If $v = \texttt{From}\langle v' \rangle$ then by the typing rule of From: $\Gamma \vdash v' : \texttt{string} \equiv B$, so by induction hypothesis $v' \equiv b$, so $v = \texttt{From}\langle v' \rangle \equiv \texttt{From}\langle b \rangle \equiv s$
  - If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then by the typing rule of Filter: $\Gamma \vdash v_1 : R' \rightarrow \texttt{bool}$ and $\Gamma \vdash v_2 : R'\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x)\rightarrow b$ and $v_2 \equiv s$, so $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle \equiv \texttt{Filter}\langle \mathbf{fun}^x(x)\rightarrow b \mid s \rangle \equiv s$
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then by the typing rule of Join: $\Gamma \vdash v_1 : R' \rightarrow R'' \rightarrow R$, $\Gamma \vdash v_2 : R' \rightarrow R'' \rightarrow \texttt{bool}$, $\Gamma \vdash v_3 : R'\ \mathbf{list}$ and $\Gamma \vdash v_4 : R''\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x, x)\rightarrow r$, $v_2 \equiv \mathbf{fun}^x(x, x)\rightarrow b$, $v_3 \equiv s$ and $v_4 \equiv s$, so $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle \equiv \texttt{Join}\langle \mathbf{fun}^x(x, x)\rightarrow r, \mathbf{fun}^x(x, x)\rightarrow b \mid s, s \rangle \equiv s$
  - If $v = \texttt{GroupBy}\langle v_1, v_2 \mid v_3 \rangle$ then by the typing rule of GroupBy: $\Gamma \vdash v_1 : R'' \rightarrow R'\ \mathbf{list}$, $\Gamma \vdash v_2 : R'\ \mathbf{list} \rightarrow R$ and $\Gamma \vdash v_3 : R''\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x)\rightarrow s$, $v_2 \equiv \mathbf{fun}^x(x)\rightarrow r$ and $v_3 \equiv s$, so $v = \texttt{GroupBy}\langle v_1, v_2 \mid v_3 \rangle \equiv \texttt{GroupBy}\langle \mathbf{fun}^x(x)\rightarrow s, \mathbf{fun}^x(x)\rightarrow r \mid s \rangle \equiv s$
  - If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then by the typing rule of Sort: $\Gamma \vdash v_1 : R \rightarrow R'$ and $\Gamma \vdash v_2 : R\ \mathbf{list}$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x)\rightarrow r$ and $v_2 \equiv s$, so $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle \equiv \texttt{Sort}\langle \mathbf{fun}^x(x)\rightarrow r \mid s \rangle \equiv s$
- If $T \equiv R \rightarrow B$ then
  - If $v = x$ then impossible since $\Gamma \vdash x : \Gamma(x) \equiv R$ by Hypothesis H2
  - If $v = \mathbf{fun}^f(x)\rightarrow v'$ then by typing rule of the function: $\Gamma, x : R \vdash v' : B$, so by induction hypothesis $v' \equiv b$, so $v = \mathbf{fun}^f(x)\rightarrow v' \equiv \mathbf{fun}^x(x)\rightarrow b$
  - If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv B$
  - If $v = c$ then impossible since $\Gamma \vdash c : \texttt{typeof}(c) \equiv B$
  - If $v = op(v_1, \ldots, v_n)$ then impossible since $\Gamma \vdash op(v_1, \ldots, v_n) : B$
  - If $v = \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3$ then impossible since $\Gamma \vdash \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3 : B$
  - If $v = \{\, l_1 : v_1, \ldots, l_n : v_n \,\}$ then impossible since $\Gamma \vdash \{\, l_1 : v_1, \ldots, l_n : v_n \,\} : \{l_1 : T_1, \ldots, l_n : T_n\}$
  - If $v = [\ ]$ then impossible since $[\ ]$ cannot be typed
  - If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash v_1 :: v_2 : R\ \mathbf{list}$
  - If $v = v_1\ \mathbf{@}\ v_2$ then impossible since $\Gamma \vdash v_1\ \mathbf{@}\ v_2 : R\ \mathbf{list}$
  - If $v = v' \cdot l$ then impossible for the same argument as for $T \equiv R$
  - If $v = \texttt{Project}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash \texttt{Project}\langle v_1 \mid v_2 \rangle : R\ \mathbf{list}$
  - If $v = \texttt{From}\langle v' \rangle$ then impossible since $\Gamma \vdash \texttt{From}\langle v' \rangle : R\ \mathbf{list}$

  - If $v = \texttt{Filter}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Filter}\langle v_1 \mid v_2\rangle : R$ **list**
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4\rangle$ then impossible since $\Gamma \vdash \texttt{Join}\langle v_1, v_2 \mid v_3, v_4\rangle : R$ **list**
  - If $v = \texttt{GroupBy}\langle v_1, v_2 \mid v_3\rangle$ then impossible since $\Gamma \vdash \texttt{GroupBy}\langle v_1, v_2 \mid v_3\rangle : R$ **list**
  - If $v = \texttt{Sort}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Sort}\langle v_1 \mid v_2\rangle : R$ **list**
- If $T \equiv R \to R$ then
  - If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R \vdash v' : R$, so by induction hypothesis $v' \equiv r$, so $v = \mathbf{fun}^x(x) \to v' \equiv \mathbf{fun}^x(x) \to r$
  - all other cases are impossible for the same arguments as for $T \equiv R \to B$
- If $T \equiv R \to R$ **list** then
  - If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R \vdash v' : R$ **list**, so by induction hypothesis $v' \equiv s$, so $v = \mathbf{fun}^x(x) \to v' \equiv \mathbf{fun}^x(x) \to s$
  - all other cases are impossible for the same arguments as for $T \equiv R \to B$
- If $T \equiv R$ **list** $\to R$ then
  - If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R$ **list** $\vdash v' : R$, so by induction hypothesis $v' \equiv r$, so $v = \mathbf{fun}^x(x) \to v' \equiv \mathbf{fun}^x(x) \to r$
  - all other cases are impossible for the same arguments as for $T \equiv R \to B$
- If $T \equiv R \to R \to B$ then
  - If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R \vdash v' : R \to B$, so by induction hypothesis $v' \equiv \mathbf{fun}^x(x) \to b$, so $v = \mathbf{fun}^x(x) \to v' \equiv \mathbf{fun}^x(x, x) \to b$
  - all other cases are impossible for the same arguments as for $T \equiv R \to B$
- If $T \equiv R \to R \to R$ then
  - If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R \vdash v' : R \to R$, so by induction hypothesis $v' \equiv \mathbf{fun}^x(x) \to r$, so $v = \mathbf{fun}^x(x) \to v' \equiv \mathbf{fun}^x(x, x) \to r$
  - all other cases are impossible for the same arguments as for $T \equiv R \to B$
- If $T \equiv T_1 \to T_2$ then
  - If $v = \mathbf{fun}^x(x) \to v'$ then the property is true
  - all other cases are impossible for the same arguments as for $T \equiv R \to B$
- If $T \equiv \{l_1 : T_1, \ldots, l_n : T_n\}$
  - If $v = x$ then the property is true
  - If $v = \mathbf{fun}^f(x) \to v'$ then impossible since $\Gamma \vdash \mathbf{fun}^f(x) \to v' : T_1 \to T_2$
  - If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv B$
  - If $v = c$ then impossible since $\Gamma \vdash c : \texttt{typeof}(c) \equiv B$
  - If $v = op(v_1, \ldots, v_n)$ then impossible since $\Gamma \vdash op(v_1, \ldots, v_n) : B$
  - If $v = \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3$ then impossible since $\Gamma \vdash \mathbf{if}\ v_1\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3 : B$
  - If $v = \{\, l_1 : v_1, \ldots, l_n : v_n \,\}$ then the property is true
  - If $v = [\ ]$ then impossible since $[\ ]$ cannot be typed
  - If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash v_1 :: v_2 : R$ **list**
  - If $v = v_1 \mathbin{@} v_2$ then impossible since $\Gamma \vdash v_1 \mathbin{@} v_2 : R$ **list**
  - If $v = v' \cdot l$ then impossible for the same argument as for $T \equiv R$
  - If $v = \texttt{Project}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Project}\langle v_1 \mid v_2\rangle : R$ **list**
  - If $v = \texttt{From}\langle v'\rangle$ then impossible since $\Gamma \vdash \texttt{From}\langle v'\rangle : R$ **list**
  - If $v = \texttt{Filter}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Filter}\langle v_1 \mid v_2\rangle : R$ **list**
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4\rangle$ then impossible since $\Gamma \vdash \texttt{Join}\langle v_1, v_2 \mid v_3, v_4\rangle : R$ **list**
  - If $v = \texttt{GroupBy}\langle v_1, v_2 \mid v_3\rangle$ then impossible since $\Gamma \vdash \texttt{GroupBy}\langle v_1, v_2 \mid v_3\rangle : R$ **list**
  - If $v = \texttt{Sort}\langle v_1 \mid v_2\rangle$ then impossible since $\Gamma \vdash \texttt{Sort}\langle v_1 \mid v_2\rangle : R$ **list**

□

We have shown that well-typedness of a QIR term restricts its syntactic form. We can finally show that terms that have a relational type can be translated into SQL by our specific translation.

Lemma B.4. *Let v be a normal form of QIR such that $v \equiv b$, or $v \equiv r$, or $v \equiv s$, then $\exists! e \in \mathsf{E}_{\mathsf{SQL}}$ such that $v \overset{SQL}{\rightsquigarrow} e$.*

Proof. The rules of Figure 5 used to derive the judgment $v \overset{\text{SQL}}{\rightsquigarrow} e$ are syntax-directed (at most one rule applies), and terminate since the premises are always applied on a strict syntactic subterm of the conclusion. Thus, since v is finite by definition of a QIR term, the translation derivation is finite and unique. We can therefore prove our lemma by induction on the translation derivation. We will use HI (Hypothesis Induction) to denote the induction hypothesis and proceed by case analysis.

- If $v \equiv b$ then
  - If $v = c$ then:

$$\frac{}{c \overset{\text{SQL}}{\rightsquigarrow} \overrightarrow{\mathsf{VAL}}^{\mathsf{SQL}}(c)}\ \text{(SQL-cst)}$$

  - If $v = \mathbf{if}\ b_1\ \mathbf{then}\ b_2\ \mathbf{else}\ b_3$ then:

$$\frac{\dfrac{}{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i}\ \text{(HI)} \qquad e_i \neq \Omega \quad i \in 1..3}{\mathbf{if}\ b_1\ \mathbf{then}\ b_2\ \mathbf{else}\ b_3 \overset{\text{SQL}}{\rightsquigarrow} \begin{array}{l}\texttt{SELECT CASE WHEN (}e_1\texttt{) THEN (}e_2\texttt{)}\\ \texttt{ELSE (}e_3\texttt{) END}\end{array}}\ \text{(SQL-if)}$$

  - If $v = x \cdot l$ then:

$$\frac{\dfrac{\dfrac{}{x \overset{\text{SQL}}{\rightsquigarrow} \texttt{X}}\ \text{(SQL-var)}}{x \cdot l \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT X.L}}\ \text{(SQL-tdestr-simpl)}}$$

– If $v = op(b_1, \ldots, b_n)$ then:

(SQL-basic-op)

$$\frac{(\text{HI}) \quad \overline{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \qquad e_i \neq \Omega \quad i \in 1..n}{o(b_1, \ldots, b_n) \overset{\text{SQL}}{\rightsquigarrow} o_{\text{SQL}}((e_1), \ldots, (e_n))}$$

- If $v \equiv r$ then
  – If $v = x$ then:

$$\frac{(\text{SQL-var})}{x \overset{\text{SQL}}{\rightsquigarrow} \texttt{X}}$$

  – If $v = \{ l_1 : b_1, \ldots, l_n : b_n \}$ then:

(SQL-record)

$$\frac{(\text{HI}) \quad \overline{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \qquad e_i \neq \Omega \quad i \in 1..n}{\{ l_1 : b_1, \ldots, l_n : b_n \} \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } (e_1) \texttt{ AS } X_1, \ \ldots, \ (e_n) \texttt{ AS } X_n}$$

- If $v \equiv s$ then:
  – If $v = r :: [\,]$ then:

(SQL-lcons-empty)

$$\frac{(\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e} \qquad e \neq \Omega \quad \text{TMP fresh}}{r :: [\,] \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM } (e_1) \texttt{ AS TMP}}$$

  – If $v = r :: s$ then:

(SQL-lcons)

$$\frac{(\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \qquad (\text{HI}) \quad \overline{s \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad e_i \neq \Omega \ \ i \in 1..2 \qquad \text{TMP, TMP2 fresh}}{r :: s \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM } (e_1) \texttt{ AS TMP UNION ALL } (e_2) \texttt{ AS TMP2}}$$

  – If $v = \texttt{Project}\langle \mathbf{fun}^f(x) \rightarrow r \mid s \rangle$ then:

(SQL-project)

$$\frac{(\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \qquad (\text{HI}) \quad \overline{s \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad e_i \neq \Omega \quad i \in 1..2}{\texttt{Project}\langle \mathbf{fun}^f(x) \rightarrow r \mid s \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } e_1 \texttt{ FROM } (e_2) \texttt{ AS X}}$$

  – If $q = \texttt{From}\langle n \rangle$ then:

(SQL-from)

$$\frac{}{\texttt{From}\langle \texttt{"table"} \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM table}}$$

  – If $q = \texttt{Filter}\langle \mathbf{fun}^f(x) \rightarrow r \mid s \rangle$ then:

(SQL-filter)

$$\frac{(\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \qquad (\text{HI}) \quad \overline{s \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad e_i \neq \Omega \quad i \in 1..2}{\texttt{Filter}\langle \mathbf{fun}^f(x) \rightarrow r \mid s \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM } (e_2) \texttt{ AS X WHERE } (e_1)}$$

  – If $q = \texttt{Join}\langle \mathbf{fun}^f(x, y) \rightarrow r, \mathbf{fun}^f(x, y) \rightarrow b \mid s_1, s_2 \rangle$ then:

(SQL-join)

$$\frac{(\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \qquad (\text{HI}) \quad \overline{b \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad (\text{HI}) \quad \overline{s_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \qquad e_i \neq \Omega \ \ i \in 3..4}{\texttt{Join}\langle \mathbf{fun}^f(x, y) \rightarrow r, \mathbf{fun}^g(x, y) \rightarrow b \mid s_3, s_4 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } (e_2) \texttt{ FROM } (e_3) \texttt{ AS X, } (e_4) \texttt{ AS Y WHERE } (e_1)}$$

  – If $q = \texttt{GroupBy}\langle \mathbf{fun}^f(x) \rightarrow s, \mathbf{fun}^f(x) \rightarrow r \mid s \rangle$ then:

(SQL-group-by)

$$\frac{(\text{HI}) \quad \overline{s_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \qquad (\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e_3} \qquad e_i \neq \Omega \quad i \in 1..2 \quad e_3 \neq \Omega}{\texttt{GroupBy}\langle \mathbf{fun}(x) \rightarrow s_1, \mathbf{fun}(x) \rightarrow r \mid s_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } (e_1) \texttt{ FROM } (e_2) \texttt{ AS X GROUP BY } (e_3)}$$

  – If $q = \texttt{Sort}\langle \mathbf{fun}^f(x) \rightarrow r \mid s \rangle$ then:

(SQL-sort)

$$\frac{(\text{HI}) \quad \overline{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \qquad (\text{HI}) \quad \overline{s \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad e_i \neq \Omega \quad i \in 1..2}{\texttt{Sort}\langle \mathbf{fun}(x) \rightarrow r \mid s \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM } (e_2) \texttt{ AS X ORDER BY } (e_1)}$$

□

Last, but not least, we can prove our sound translation theorem as a direct corollary of Theorem 4.1, and Lemmas B.3 and B.4.

Translation. Let $q \in \mathsf{E}_{\text{QIR}}$ such that $\emptyset \vdash q : T, q \rightarrow^* v$, and v is in normal form. If $T \equiv B$ or $T \equiv R$ or $T \equiv R$ **list** then $v \rightsquigarrow$ s, SQL.

By Theorem 4.1, we have: $\emptyset \vdash v : T$. Thus, we can apply Lemma B.3 and deduce: $v \equiv b$ or $v \equiv r$ or $v \equiv s$. Consequently, by Lemma B.4, we obtain: $v \overset{\text{SQL}}{\rightsquigarrow}$ s.Finally, we apply the rule (db-op) of our generic translation, and since the specific translation was able to translate v to SQL, it is able to translate the sources of the query (which are syntactically strict subterms of v) to SQL. Therefore, the rule (db-op) succeeds at translating v to a term of SQL. □

C FROM R TO QIR

We give some technical details of our translation from R to QIR. Even though we do not modify the parsing of R programs, we still need to translate R closures into QIR λ-expressions. For instance, given the R program:

```
less2000 = function (x) { x <= 2000 }
t = tableRef("employee", "PostgreSQL")
subset(t, less2000(sal))
```

we want to end up with the QIR term (before normalization):

$$\begin{array}{l}(\mathbf{fun}(\mathit{less2000}, t) \rightarrow \\ \quad \texttt{Filter}\langle \mathbf{fun}(r) \rightarrow \mathit{less2000}\, r \cdot \mathit{sal} \mid t \rangle \\ )(\mathbf{fun}(x) \rightarrow \leq (x, 2000), \texttt{From}\langle \mathit{employee} \rangle)\end{array}$$

which becomes, after normalization:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow \leq (r \cdot \mathit{sal}, 2000) \mid \texttt{From}\langle \mathit{employee} \rangle \rangle$$

While it seems obvious from this example that the function `less2000` should be translated into the λ-term $\mathbf{fun}(x) \rightarrow \leq (x, 2000)$, it is not always sound to do so. Indeed, a variable x can be soundly

$$\frac{}{\gamma, \sigma \vdash c \overset{R}{\rightsquigarrow} \overset{R}{\overrightarrow{\mathsf{VAL}}}(c)} \qquad \frac{x \notin \gamma}{\gamma, \sigma \vdash \mathsf{x} \overset{R}{\rightsquigarrow} x}$$

$$\frac{\gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \gamma, \sigma \vdash e_2 \overset{R}{\rightsquigarrow} q_2 \qquad \begin{array}{l} t \text{ fresh} \\ \{y_1, \ldots, y_m\} = FV(e_2) \setminus \mathrm{dom}(\sigma) \\ q_2' = q_2\{y_1/t \cdot y_1, \ldots, y_m/t \cdot y_m\} \end{array}}{\gamma, \sigma \vdash \mathtt{subset}(e_1, e_2, \mathsf{c}(\mathsf{x}_1, \ldots, \mathsf{x}_n)) \overset{R}{\rightsquigarrow} \mathtt{Project}\langle \mathbf{fun}(t) \rightarrow \{\, l_i : t \cdot l_i \,\} \mid \mathtt{Filter}\langle \mathbf{fun}(t) \rightarrow q_2' \mid q_1 \rangle\rangle}$$

$$\frac{\gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \gamma, \sigma \vdash e_2 \overset{R}{\rightsquigarrow} q_2}{\gamma, \sigma \vdash \mathtt{merge}(e_1, e_2, \mathsf{c}(\mathsf{x}_1, \ldots, \mathsf{x}_n)) \overset{R}{\rightsquigarrow} \begin{array}{l} \mathtt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \\ \mathbf{fun}(a, b) \rightarrow \bigwedge_i a \cdot x_i = b \cdot x_i \mid q_1, q_2 \rangle \end{array}} \qquad \frac{\gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \ldots \quad \gamma, \sigma \vdash e_n \overset{R}{\rightsquigarrow} q_n}{\gamma, \sigma \vdash \mathsf{c}(e_1, \ldots, e_n) \overset{R}{\rightsquigarrow} [\, q_1, \ldots, q_n \,]}$$

$$\frac{\gamma \cup \{x_1, \ldots, x_n\}, \sigma \vdash e \overset{R}{\rightsquigarrow} q}{\gamma, \sigma \vdash \mathtt{function}(\mathsf{x}_1, \ldots, \mathsf{x}_n)\ \{e\} \overset{R}{\rightsquigarrow} \mathbf{fun}(x_1, \ldots, x_n) \rightarrow q} \qquad \frac{\gamma, \sigma \vdash e \overset{R}{\rightsquigarrow} q \quad \gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \ldots \quad \gamma, \sigma \vdash e_n \overset{R}{\rightsquigarrow} q_n}{\gamma, \sigma \vdash e(e_1, \ldots, e_n) \overset{R}{\rightsquigarrow} q\ q_1\ \ldots\ q_n}$$

$$\frac{\gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \ldots \quad \gamma, \sigma \vdash e_n \overset{R}{\rightsquigarrow} q_n}{\gamma, \sigma \vdash \mathsf{op}\ e_1\ \ldots\ e_n \overset{R}{\rightsquigarrow} op(q_1, \ldots, q_n)} \qquad \frac{\gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \gamma \setminus \{x\}, \sigma \vdash e_2 \overset{R}{\rightsquigarrow} q_2 \quad \mathsf{x} \notin Mod(\sigma, e_2)}{\gamma, \sigma \vdash (\mathsf{x} = e_1); e_2 \overset{R}{\rightsquigarrow} (\mathbf{fun}(x) \rightarrow q_2)\ (q_1)}$$

$$\frac{\gamma, \sigma \vdash e_1 \overset{R}{\rightsquigarrow} q_1 \quad \gamma, \sigma \vdash e_2 \overset{R}{\rightsquigarrow} q_2 \quad \gamma, \sigma \vdash e_3 \overset{R}{\rightsquigarrow} q_3}{\gamma, \sigma \vdash \mathsf{if}\ (e_1)\ e_2\ \mathsf{else}\ e_3 \overset{R}{\rightsquigarrow} \mathbf{if}\ q_1\ \mathbf{then}\ q_2\ \mathbf{else}\ q_3} \qquad \frac{}{\gamma, \sigma \vdash e \overset{R}{\rightsquigarrow} \blacksquare(\sigma, e)}\ \text{otherwise}$$

Figure 6: Translation from R to QIR terms

translated into a QIR variable x if it is not the subject of side effects (in particular for function parameters), otherwise accesses to x must be nested inside host language expressions $\blacksquare(\sigma, \mathsf{x})$ so that the correct value for x can be retrieved.

The set of modified variables can be approximated by the *Mod* function defined as such:

Definition C.1 (Approximation of modified variables). Let $e \in \mathsf{E}_\mathsf{R}$ be an expression and σ an evaluation environment for R. The set $Mod(\sigma, e)$ of modified variables in e is inductively defined as:

$$\begin{array}{lcl}
Mod(\sigma, \mathsf{x}) & = & \{\} \text{ if } \mathsf{x} \notin \mathrm{dom}(\sigma) \\
Mod(\sigma, \mathsf{x}\ =\ e) & = & \{\mathsf{x}\} \cup Mod(\sigma, e) \\
Mod(\sigma, \mathsf{x}) & = & \{\} \text{ if } \sigma(\mathsf{x}) \neq \mathtt{function}_{\sigma'}(\ldots)\ldots \\
Mod(\sigma, \mathsf{x}) & = & Mod(\sigma' \cup \sigma, e') \\
 & & \text{if } \sigma(\mathsf{x}) = \mathtt{function}_{\sigma'}(\ldots)e' \\
Mod(\sigma, \mathtt{function(...)e}) & = & Mod(\sigma, e) \\
Mod(\sigma, c) & = & \{\} \\
Mod(\sigma, e_1\,; e_2) & = & Mod(\sigma, e_1) \cup Mod(\sigma, e_2) \\
Mod(\sigma, e(e_1, \ldots, e_n)) & = & Mod(\sigma, e) \cup \bigcup_{i=1}^{n} Mod(\sigma, e_i) \\
 & \vdots &
\end{array}$$

The first five cases of the *Mod* function are the most interesting ones (the others being only bureaucratic subterm calls). First, if a variable is used, but is not in the current scope, it is not marked as modified. If the variable is being assigned to, then it is added to the set of modified variables. If the variable is bound in the current scope, to a value that is not closure, then it is also marked as unmodified. However, if a variable is bound to a closure, then the body of the latter is traversed, in an environment augmented with the closure environment. Lastly, the body of anonymous functions are recursively explored to collect modified variables. We can now tackle the translation from R expressions to QIR terms.

Definition C.2. We define the judgment $\gamma, \sigma \vdash e \overset{R}{\rightsquigarrow} q$, which means that given a set of modified variables γ and an R environment σ, the R expression e can be translated into a QIR expression q. The derivation of this judgment is given by the rules in Figure 6. We define the translation ${}^{R}\overrightarrow{\mathsf{EXP}}(\sigma, e) = q$ as $Mod(\sigma, e), \sigma \vdash e \overset{R}{\rightsquigarrow} q$.

This translation is straightforward and kept as simple as possible. Constants are translated into QIR equivalents. Identifiers that are not modified are also translated into their QIR counterpart. Anonymous functions are translated into QIR lambdas. More interesting is the translation of the builtin function `subset`. Its first two arguments are recursively translated, but the second one requires some post-processing. Recall that in the case of `subset`, the second argument e_2 contains free variables bound to column names. We simulate this behavior by introducing a lambda abstraction whose argument is a fresh name t and replace all occurrences of a free variable x in the translation by $t \cdot x$. The last argument is expected to be a list of column names we use to build a lambda abstraction to project over these names. The `merge` function is similarly translated into a `Join` operator. The last interesting case is when a local variable is defined in a sequence of expressions. If this variable is not modified in the subsequent expression, then we can translate this definition into a lambda application. Expressions that are not handled (in particular modified variables) are kept in host expression nodes that will be evaluated either locally, in a QIR term that is not shipped to a database, or remotely, using the R runtime embedded in a database.

Now that we have defined the translation of expressions in a given scope, we can easily define the translation of values from R

to QIR. The translation of constants, sequences and data frames is straightforward. The translation of a closure $\texttt{function}(\texttt{x}_1, \ldots, \texttt{x}_n)_\sigma\{e\}$ is simply the translation of the body wrapped in a lambda: $\textbf{fun}(x_1, \ldots, x_n) \rightarrow {}^{\mathsf{R}}\overrightarrow{\mathsf{EXP}}(\sigma, e)$.