Breaking boundaries between programming languages and databases

Julien Lopez

NNT : 2019SACLS235

Thèse de doctorat

Au-delà des frontières entre langages de programmation et bases de données

Thèse de doctorat de l'Université Paris-Saclay
préparée à Université Paris-Sud

Ecole doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Orsay, le 13/09/2019, par

Julien Lopez

Composition du Jury :

James Cheney Reader, Université d'Édimbourg	Rapporteur
Emmanuel Chailloux Professeur, Sorbonne Université (UMR 7606)	Rapporteur
Alan Schmitt Directeur de recherche, INRIA Rennes (UMR 6074)	Examinateur
Jérôme Siméon Chief Scientist, Clause Inc.	Examinateur
Sarah Cohen Boulakia Professeur, Université Paris-Sud (UMR 8623)	Présidente du jury
Véronique Benzaken Professeur, Université Paris-Sud (UMR 8623)	Directeur de thèse
Kim Nguyen Maitre de Conférences, Université Paris-Sud (UMR 8623)	Co-encadrant de thèse
Giuseppe Castagna Directeur de Recherche, CNRS et Université de Paris (UMR 8243)	Invité

Remerciements

Tout d'abord, je souhaite remercier mes encadrants de thèse Véronique Benzaken et Kim Nguyen. Véronique pour avoir toujours accepté de me recevoir lorsque j'en avais besoin, pour son énergie, son enthousiasme contagieux pour l'informatique, et pour m'avoir initié aux plus grandes abérrations syntaxiques d'SQL. Kim pour avoir partagé avec moi ses connaissances scientifiques du lambda calcul aux magies les plus noires de JavaScript, pour les expériences d'enseignement enrichissantes qui ont élargi mes connaissances générales, pour sa patience à m'expliquer un concept de manières différentes jusqu'à ce que mon esprit têtu finisse par comprendre, son aide précieuse dans tous les aspects de la thèse en particulier pour sa rédaction, et sa disponibilité sans faille malgré sa moyenne d'environ cent cours par jour. Merci beaucoup à tous les deux pour votre encadrement de qualité, pour m'avoir donné ma chance, et enfin pour m'avoir traité avec humanité dans les moments difficiles. Ce doctorat a été pour moi une expérience très enrichissante, et je n'oublierai jamais ce que vous avez fait pour moi.

Un grand merci également à James Cheney et Émmanuel Chailloux pour avoir accepté d'être les rapporteurs de ma thèse. Je suis honoré que vous ayez pris le temps d'examiner mon travail avec autant d'attention, et vous remercie pour vos corrections et remarques pertinentes. Merci à Alan Schmitt, Jérôme Siméon, et Sarah Cohen Boulakia d'avoir accepté de faire partie de mon jury de thèse.

Je suis très reconnaissant à Giuseppe Castagna pour avoir non seulement encadré mon stage de fin d'études en école d'ingénieur, et m'avoir ainsi apporté ma première expérience dans la recherche, mais aussi pour m'avoir proposé cette thèse, pour m'avoir soutenu durant mon année au MPRI, et pour son aide précieuse au cours de la thèse. Un grand merci à Beppe sans qui ce travail n'aurait pas été possible.

J'aimerais également remercier Laurent Daynes, mon encadrant de stage MPRI à Oracle Labs, qui m'a permis de travailler dans son équipe et ainsi de commencer à travailler sur mon sujet de thèse en collaborant avec les ingénieurs d'Oracle. Merci à Romain Vernoux dont les travaux de stage ont servi de bases solides pour mes recherches, ainsi qu'à Alban Petit et Romain Liautaud qui ont travaillé en stage sur ma solution.

Merci ensuite à mes collègues du LRI, et à l'équipe VALS pour m'avoir permis de travailler dans une ambiance chaleureuse. Merci à Sylvain Conchon avec qui

j'ai renforcé mon OCaml en enseignant notamment comment réaliser un jeu vidéo avec des `match`, et qui m'a appris le model checking. Merci à Guillaume Melquiond de m'avoir appris à faire du C++ correctement (sans *), et à Thibaut Balabonski grâce à qui j'ai dépoussiéré mes connaissances en compilation, tout cela me sera utile très prochainement. Merci à Jean-Christophe Filliâtre pour m'avoir partagé ses connaissances sur OCaml, et pour m'avoir prêté son exemplaire de "Sixty Million Frenchmen Can't Be Wrong". Merci à Frédéric Voisin pour avoir trouvé le code d'accès à la salle de reprographie pour un thésard en panique et pour les blagues du vendredi (du lundi au vendredi) au coin café. Merci à Sylvie Boldo pour m'avoir fait confiance pour la review d'un article au JFLA, pour son aide précieuse avec Véronique sur le recrutement académique, et sur le fonctionnement de la machine magique à faire des posters. Merci à Hai et Stefania pour m'avoir montré à l'avance dans quel état je serai en dernière année, à Robin pour avoir écouté mes tirades inintéressantes sur le speedrun de Mario 64, à Albin pour le vin à l'orange, à Bruno pour la promenade en vélo, à Alexandrina pour m'avoir fait rire quand j'en avais bien besoin et pour son amitié. Ah, et merci à Mattias pour les pauses, m'avoir presque donné envie de jouer à un MOBA, et pour m'avoir appris la quantité de crème pour des pâtes à la carbonara.

Merci à mes vieux amis, Pierre-Alain, Pierre-Alain, Frédéric, Romain qui m'ont soutenu malgré la distance, et Bertrand, Solenne, Thibaut et Pierre-Jean pour nos beuveries et parties de Borderlands.

Un grand merci à ma famille, tout particulièrement mes parents, ma grand-mère, mon frérot, et ma petite sœur. Sans votre amour et votre soutien, je ne serais jamais allé aussi loin. Enfin, merci à toi Jenny. Merci pour ton soutien, tes corrections sur mon anglais beaucoup trop français, et pour ces merveilleuses années que nous avons passées ensemble.

Abstract

Several classes of solutions allow programming languages to express queries: specific APIs such as JDBC, Object-Relational Mappings (ORMs) such as Hibernate, and language-integrated query frameworks such as Microsoft's LINQ. However, most of these solutions do not allow for efficient cross-databases queries, and none allow the use of complex application logic from the programming language in queries.

This thesis studies the design of a new language-integrated query framework called BOLDR that allows the evaluation in databases of queries written in general-purpose programming languages containing application logic, and targeting several databases following different data models. In this framework, application queries are translated to an intermediate representation. Then, they are typed with a type system extensible by databases in order to detect which database language each subexpression should be translated to. This type system also allows us to detect a class of errors before execution. Next, they are rewritten in order to avoid query avalanches and make the most out of database optimizations. Finally, queries are sent for evaluation to the corresponding databases and the results are converted 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.

Résumé

Plusieurs classes de solutions permettent d'exprimer des requêtes dans des langages de programmation: les interfaces spécifiques telles que JDBC, les mappings objet-relationnel ou object-relational mapping en anglais (ORMs) comme Hibernate, et les frameworks de requêtes intégrées au langage comme le framework LINQ de Microsoft. Cependant, la plupart de ces solutions ne permettent

pas d'écrire des requêtes visant plusieurs bases de données en même temps, et aucune ne permet l'utilisation de logique d'application complexe dans des requêtes aux bases de données.

Cette thèse présente un nouveau framework de requêtes intégrées au langage nommé BOLDR qui permet d'écrire des requêtes dans des langages de programmation généralistes et qui contiennent de la logique d'application, et de les évaluer dans des bases de données hétérogènes. Dans ce framework, les requêtes d'une application sont traduites vers une représentation intermédiaire de requêtes. Puis, elles sont typées en utilisant un système de type extensible par les bases de données pour détecter dans quel langage de données chaque sous-expression doit être traduite. Cette phase de typage permet également de détecter certaines erreurs avant l'exécution. Ensuite, les requêtes sont réécrites pour éviter le phénomène "d'avalanche de requêtes" et pour profiter au maximum des capacités d'optimisation des bases de données. Enfin, les requêtes sont envoyées aux bases de données ciblées pour évaluation et les résultats obtenus sont convertis dans le langage de programmation de l'application. Nos expériences montrent que les techniques implémentées dans ce framework sont applicables pour de véritables applications centrées données, et permettent de gérer efficacement un vaste champ de requêtes intégrées à des langages de programmation généralistes.

Note : Afin d'en assurer une plus large diffusion, et en accord avec l'école doctorale STIC Paris-Saclay, cette thèse est rédigée en anglais. Pour un résumé étendu des travaux rédigé en français, voir l'Annexe A page 163.

Contents

List of symbols

Chapter 1

Introduction

Note. This thesis is at the intersection between the fields of programming languages and databases. Therefore, this introduction describes some basic notions coming from both fields to help unfamiliar readers.

1.1 Context

Storing, accessing, and manipulating data is unavoidable and critical for most *applications*. Web, statistical and artificial intelligence applications, Internet of Things, all require access to large quantities of information stored in heterogeneous data sources.

Applications are written in *general-purpose programming languages* often chosen depending on their support for common operations in particular fields (e.g., R or Python for statistical analysis and data mining, JavaScript for Web programming). These programming languages are often *imperative*, meaning users must describe how to access and process information using sequences of statements that modify the state of the program.

The information required by applications is stored in *databases*, which are managed by *DataBase Management Systems* (DBMS). These systems handle storage, fast access to data using a *query language*, fault tolerance, scalability, confidentiality, and more. An expression written in a query language, called a *query*, describes the requested data, rather than describing in imperative fashion how to retrieve the data, letting the DBMS choose the best way to fetch the requested information.

To access data stored in a database, an application sends a query to the database written in its query language. A typical data-oriented application includes components which interface the application with the different databases it targets. For example, a recent technique called polyglot persistence [JSF12] consists of accessing different types of databases in one application to take advantage of the capabilities of the different database models for the different parts

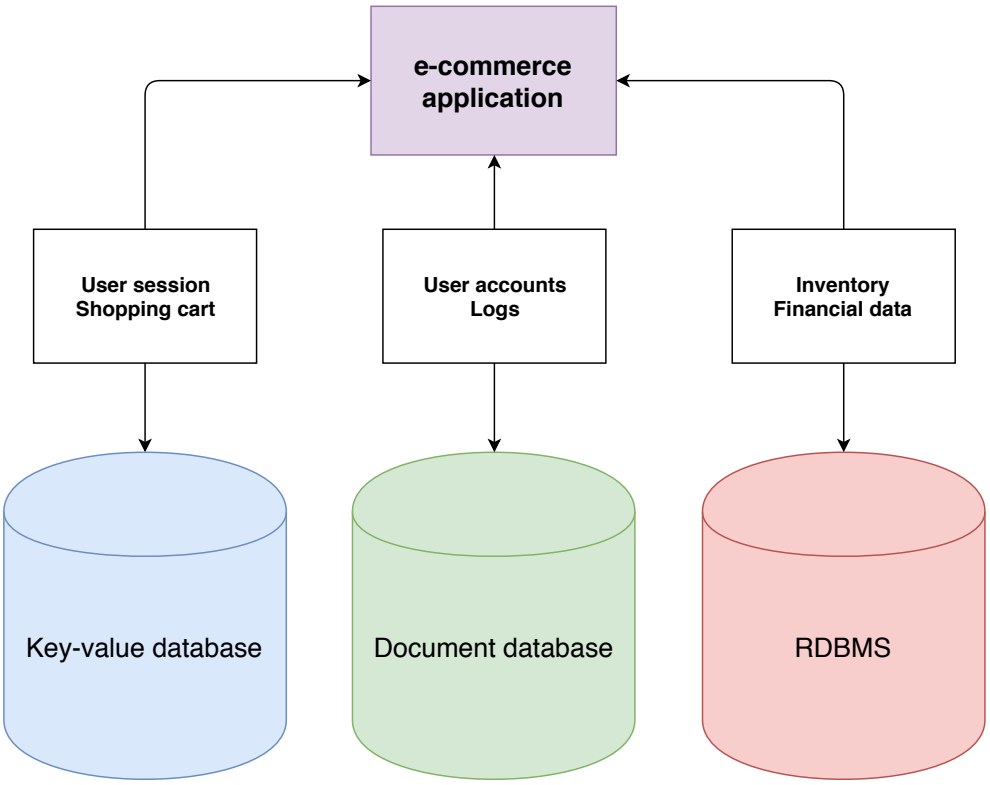


Figure 1.1 – Example of an application using different types of databases

of the application. Figure 1.1 shows an example of such application.

This thesis is a study aiming to create a solution that allows application developers to write safe and efficient queries targeting databases without forcing them to become experts in the data models and query languages of their target databases.

In this introduction, we first give an overview of database query languages, programming languages used in the development of applications, and the existing solutions to interface these two worlds and the problems encountered in the process. Then we describe a new solution in the form of a new language-integrated query framework called BOLDR.

1.2 SQL

SQL (Structured Query Language) is the most popular query language. It is a domain-specific language based on *relational algebra*.

id	name	salary	teamid
1	Lily Pond	5200	2
2	Daniel Rogers	4700	1
3	Olivia Sinclair	6000	1

(a) Table Employee

teamid	teamname	bonus
1	R&D	500
2	Sales	600

(b) Table Team

Figure 1.2 – An example of data organized as tables

1.2.1 Relational algebra

Relational algebra, first designed by Edgar F. Codd [Cod70], defines operations on data represented as a set of *n-tuples* where every element of the tuple corresponds to an attribute denoted by a name. Relational databases call these constructions *tables*, composed of *lines* and *columns*. Figure 1.2 gives an example of tables.

Relational algebra is the basis of most database query languages [AHV95]. The most common operations of relational algebra are the *projection*, which restricts the tuples to a set of attributes; the *selection* (or filter), which keeps only the tuples that satisfy a condition; and the *join*, which returns the set of all combinations of tuples from two tables that are equal on their common attributes. Figure 1.3 shows examples of applications of those operations on tables. Figure 1.3a shows the projection of table *Employee* on attributes *name* and *salary*, Figure 1.3b shows the selection in the table *Employee* of the tuples for which the value of the attribute *salary* is greater than 5000, and Figure 1.3c shows the result of the join between *Employee* and *Team*.

name	salary
Lily Pond	5200
Daniel Rogers	4700
Olivia Sinclair	6000

(a) Projection on *Employee*

id	name	salary	teamid
1	Lily Pond	5200	2
3	Olivia Sinclair	6000	1

(b) Selection on *Employee*

id	name	salary	teamname	bonus
1	Lily Pond	5200	Sales	600
2	Daniel Rogers	4700	Sales	600
3	Olivia Sinclair	6000	R&D	500

(c) Join between *Employee* and *Team*

Figure 1.3 – Example of applications of relational algebra

1.2.2 Expressing queries in SQL

SQL gives access to operations of relational algebra as a *declarative* programming language. In other words, instead of describing step-by-step how the computation must be done to achieve the desired result, programming in SQL involves describing the desired result. To achieve this, SQL adopted a syntax similar to a natural language. For instance, the projection of Figure 1.3a can be written in SQL as such:

```
SELECT name, salary FROM Employee
```

where `SELECT` represents the projection, and `FROM` represents a data operator `From` that returns the content of a table from its name.

The selection of Figure 1.3b can be written:

```
SELECT * FROM Employee WHERE salary > 5000
```

where `*` means all columns. Finally, the join of Figure 1.3c can be written:

```
SELECT * FROM Employee NATURAL JOIN Team
```

or similarly:

```
SELECT * FROM Employee, Team WHERE Employee.deptno = Team.deptno
```

As shown in this last query, the names of the tables can be used as the names of the current line of the corresponding table to lift the ambiguity on which row is to be accessed for the value of a column. SQL also allows the creation of *aliases* to give temporary names to tables using the keyword `AS`. For instance, this last query could be written:

```
SELECT * FROM Employee AS e, Team AS t WHERE e.deptno = t.deptno
```

These aliases are not directly useful in such a query where the table names already discriminate the rows, but it is necessary in some queries such as self joins which apply a `Join` operator between a table and itself, or to give a name to the result of a *subquery*:

```
SELECT * FROM Employee e,
  (SELECT 1 AS teamid, 'R&D' AS teamname, 500 AS bonus UNION ALL
   SELECT 2 AS teamid, 'Sales' AS teamname, 600 AS bonus) AS t
WHERE e.deptno = t.deptno
```

In this last query, the `UNION ALL` subquery creates the following anonymous table:

teamid	teamname	bonus
1	R&D	500
2	Sales	600

which is then bound to the name `t` using an alias, and this name is then used in the `WHERE` clause to refer to the table.

The simple syntax of SQL is one of the reasons why it is so popular, and the most commonly used query language. Most databases support SQL, even those that do not have a data model directly suited for relational algebra. Therefore, SQL is an unavoidable database language to study for solutions aiming to allow programmers to send queries to databases.

1.3 Application programming languages

The majority of data-driven applications are written in *imperative* programming languages. *Python* is used in particular for Web applications and for machine learning. It is a very popular programming language because of its simple syntax and numerous field-specific libraries, such as for machine learning, general algorithms, and statistics. *JavaScript* is widely used for Web applications. *R* is a language natively designed for statistical applications and data analysis. *Java* is a widely used general-purpose programming language with numerous libraries for Web development, machine learning, text processing, and more.

Contrary to declarative programming in languages like SQL, imperative programming involves describing step-by-step the control flow of a program, which requires programmers in these languages to describe *how* to get to the desired result. For instance, filtering a table in an imperative language would typically be written as such in Python:

```
filteredTable = []
for employee in employees:
  if (employee['salary'] > 5000):
    filteredTable.append(employee)
```

However, modern programming languages have made an effort to support some aspects of *functional* programming, making data-oriented applications less technically detailed. For instance, we can write the example above in Python using list comprehensions [Kuh11]:

```
[employee for employee in employees if employee['salary'] > 5000]
```

Application programs can use imperative and functional features, and usually contain a mix of both. Even so, most application languages are originally imperative, and in particular are more efficient at evaluating imperative code.

Additionally, most application languages (Python, R, Ruby, JavaScript, ...) are *dynamically typed*, meaning that the type-safety of a program is checked during its execution. For instance, a program such as

```
(function (x) { return x; })(2, 3)
```

which applies the identity function to two arguments would be recognized as an error during its execution (or should, but JavaScript just ignores the second argument in this case ...).

1.4 Sending queries from application languages

As stated earlier, most applications are written in general-purpose programming languages. These languages do not have native ways to query databases, and the vast majority of them are imperative languages, so their syntax is very different from those of query languages. Various solutions have been designed to enable programmers to send queries to databases from their programming languages. In this section, we take a look at some existing solutions, and discuss their pros and cons.

1.4.1 JDBC

Java Database Connectivity (JDBC) [Cor16] is an *application programming interface* (API) which provides data access from the Java programming language to data sources, including databases.

Example 1.1 is an example of use of JDBC in a Java program to retrieve data from a database.

Example 1.1.

```
final Connection conn = ...
final Statement stmt = conn.createStatement();
final String query =
  "SELECT id, name, salary FROM employee WHERE salary > 2500";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  System.out.println(rs.getInt("ID") + " "
  + rs.getString("NAME") + " " + rs.getFloat("SALARY"));
}
```

In this example, the program queries the identifier, name, and salary of employees which salary is greater than 2500 from a table `employee` stored in a database.

In JDBC, the user must first create a `Connection` object using the correct credentials to access the targeted database, then create a `Statement` object from the connection object to send a query. The query itself is a *string* in the query language of the database (SQL in the example). The results are represented

by a `ResultSet` object which contains a cursor starting at the beginning of the result set of rows. `ResultSet` exposes the method `next()` to move the cursor to the next row of data, and access methods such as `getInt()` that returns the data in the column given as argument in the current row pointed by the cursor. For example, `rs.getInt("ID")` accesses the integer stored in the column named `"ID"` of the current row in the result set `rs`.

Although JDBC is popular and easy-to-use for programmers who are experts in the query language of their targeted database, this very common type of solution has numerous flaws:

- Programmers must learn the query language of each database they target.
- Integration of application logic is very limited as the conversion from expressions of the application language to the database language is restricted to basic values (strings, integers, ...), thus forcing programmers to decompose complex queries into simpler ones to send to databases and combine the results in the application, which entails more work for the programmers, code duplication, and potentially disastrous performances.
- Errors in queries, even syntactical, are detected only at runtime since programming language tools such as type systems are unable to detect problems in a query written as a string.
- Special care must be taken when inserting user input into queries to avoid code injection attacks [HVO06].
- Explicit type coercions must be used to translate values from the database to the application language (in JDBC, by using methods such as `getInt`).
- Changing a targeted database to one that does not have the same query language implies rewriting all the queries in the application.
- From a software engineering stand point, this solution entails the development of a completely new API for every connection between an application language and a database. This explains the multiplication and diversity of competing solutions for a same application language.

This set of problems has been referred to in the literature as an *impedance mismatch* between the database and the application language [CM84]. Other solutions have been proposed to solve these difficulties.

1.4.2 ORMs

Object-Relational Mappers (ORMs) and equivalents such as *Object-Document Mappers* (ODMs) are design patterns allowing the conversion and manipulation of data between incompatible type systems in object-oriented programming

languages. By representing the data source with an object, these patterns allow abstraction from the data source and manipulation of information directly in the programming language itself. Examples of ORM libraries are Hibernate [KBA+09] for Java, ActiveRecord [ct17] for Ruby, Doctrine [WV10] for PHP (which is also an ODM library), or Django [KMH07] for Python.

Although most of these libraries rely on queries written as strings in SQL, or close cousins such as the OQLs (HQL, DQL, JPQL, ...) [ASL89], efforts have been made to improve the integration of queries in the application language. For example, rather than writing queries in plain text, Criteria for Hibernate allows a user to build a `CriteriaQuery` object on which one can apply operations such as filters using methods. Using Criteria and Hibernate, Example 1.1 would be written as shown in Example 1.2 in the language Java.

Example 1.2.

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cr = cb.createQuery(Employee.class);
Root<Item> r = cr.from(Employee.class);
cr.multiselect(r.get("id"), r.get("name"), r.get("salary"))
  .where(cb.gt(r.get("salary"), 2500));

Query<Employee> query = session.createQuery(cr);
List<Employee> results = query.getResultList();
```

The query is described using high-level expressions, therefore it is abstracted from a particular database language, such as SQL, and its syntax is verified using the type system of the language. However, this solution is still verbose; it requires a different library for every link between a language and a database; and its expressiveness is heavily restricted to the API. Additionally, this solution requires the programmer to replicate the schemas of database tables in the application using classes.

1.4.3 LINQ

A breakthrough in the domain came with the Microsoft *LINQ* [Mic] framework, a component of the *.NET* framework which adds native data querying capabilities to .NET languages. LINQ defines queries as a first-class concept within the language semantics, thus allowing .NET programmers to define queries for databases in the syntax of their language. Example 1.3 is the equivalent of Example 1.1 written in LINQ in the language C#.

Example 1.3.

```
var results =
  from e in db.Employee
  where e.salary > 2500
  select new { id = e.id, name = e.name, salary = e.salary };
foreach (var e in results) {
  Console.WriteLine(e.id + " " + e.name + " " + e.salary);
}
```

Although LINQ adds new syntactic constructs for queries, such as the keywords `from`, `where`, and `select`, expressions in queries are native C# expressions. For instance, the `new {...}` expression showcased in Example 1.3 is the native way in C# to create a new anonymous object. Additionally to the query syntax, LINQ also provides an object-oriented way to express a query. For instance, the query of Example 1.3 can also be written:

```
db.Employee
.Where(e => e.salary > 2500)
.Select(e => new { id = e.id, name = e.name, salary = e.salary })
```

where `x => e` denotes the anonymous function with parameter x and body e.

The queries in LINQ are therefore integrated to the programming language and type-safe. In addition, the use of the .NET framework and of an intermediate language makes it possible for any language or database to interface with LINQ independently. Indeed, as shown by Figure 1.4, instead of creating an interface between every application language expressing queries, called *host language*, and every database, this approach requires host languages and databases to only interface with the intermediate language. Thus, language and database implementors only need to be an expert in their language and the intermediate language to interface with the framework.

However, not all queries are executed successfully in LINQ, as only the code that can be translated into the intermediate language of LINQ is accepted. Therefore, the expressiveness of the queries is limited in this framework. For instance, queries in LINQ cannot include arbitrary *user-defined functions* (functions defined using the syntax of the programming language, UDFs for short). For instance, Example 1.4 throws an error at runtime since LINQ attempts to translate the function `dolToEuro` to an equivalent in the database, and fails to do so. This is not limited to user-defined functions: any expression that cannot be translated is rejected. It is the responsibility of the implementer of the *LINQ provider*, the part of the LINQ architecture that translates C# expressions into a specific query language, to handle as much of the expression language as possible. LINQ offers

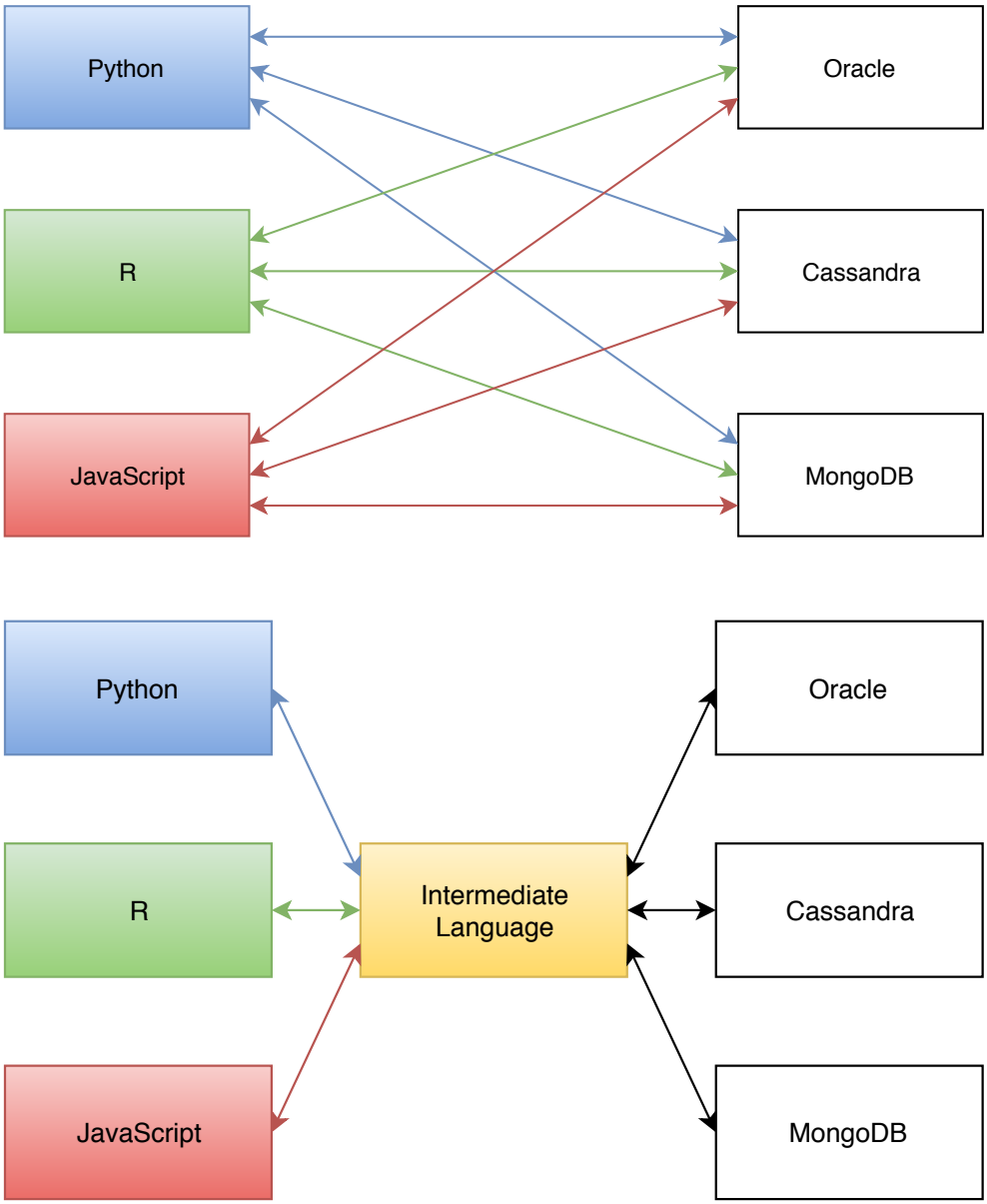


Figure 1.4 – High-level benefits of an intermediate representation

little support in that respect and this translation is considered a major pain point of writing providers [Ein11].

Example 1.4.

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee
.Where(e => e.salary > dolToEuro(2500));
```

There are two workarounds for this problem, but both are unsatisfactory. One solution is to manually mirror the definition of `dolToEuro` on the database side, as a stored procedure. This solution is particularly attractive now that databases are working on supporting application languages: Oracle R Enterprise [Orad], and PL/R [PL/a] for R; PL/Python [Pos], Amazon Redshift [Ama], Hive [Apab], and SPARK [Apac] for Python; or MongoDB [Mon] and Cassandra's CQL [Apad] for JavaScript. However, this results in the duplication of code performing application logic on the database side, causing substantial maintenance problems, especially in queries targeting several databases. Worse, such a function might not even be writable on the database side, since it may use features not supported by the database, or require access to values present in the runtime of the application language that the programmer would then have to send explicitly to the function at runtime, cluttering its definition with extra parameters.

Another solution is to fetch the data in the application, and then apply the operations. This solution seems to be preferred by developers, since it is syntactically very light in LINQ:

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee
.AsEnumerable()
.Where(e => e.salary > dolToEuro(2500));
```

This program is successfully executable by LINQ. But this seemingly innocuous addition of the call to `AsEnumerable()` hides huge performance problems: all data is transferred into the runtime of the application language by the method `Enumerable.AsEnumerable()`, which may result in terrible performances because of the network delay, and potentially causing out of memory errors. In addition, data operations are then executed in the application, thus ignoring all optimizations that the database can perform (e.g. using indexes). The same problem occurs with cross-database queries, as the solution in LINQ would also be to perform most of the queries in the application runtime with explicit calls to `AsEnumerable()`.

A partial solution to this problem is brought by *T-LINQ* [CLW13], which gives theoretical foundations to language-integrated queries based on quotations and a

normalization of queries. This solution allows the use of user-defined functions in queries as long as it is possible to translate them and inline them in the queries. However, T-LINQ is restricted by design to the data model of SQL, as well as to a few data operations. Implementations of LINQ for languages such as C# make a best effort to normalize queries containing features not handled by T-LINQ.

1.4.4 Apache Calcite

Apache Calcite [BCRH+18] is a query compiler framework that provides data-agnostic query processing and customizable optimization for queries targeting different data models and stores. Calcite provides database implementors with a unifying framework, including support for query languages such as SQL, and query optimizations. Additionally, Calcite allows queries between heterogeneous data sources by providing a unifying relational abstraction, and by selecting the most efficient plans to perform the queries, in particular using data migration to run the queries entirely in database engines if possible. Calcite takes as input SQL and JDBC, and is therefore limited in the expressiveness of queries. A language-integrated query syntax similar to LINQ is being implemented for the Java programming language, but this work is preliminary and only addresses the syntactical aspects.

1.4.5 Other interfaces

In the programming language R, *RODBC* allows programmers to send queries to databases using SQL in a similar way as JDBC. *Dplyr* is a library for data manipulation for R. *SparkR* gives an interface for *Apache Spark*. In Python, there are various libraries such as *pyodbc* or *PySpark* to access databases, and *NumPy* to manipulate of large collections of data.

All of these interfaces are similar to the other solutions we presented in this section and share their lot of shortcomings. We talk about more existing solutions in the related work, Section 8.1.

1.5 A new solution: BOLDR

As we showed in Section 1.4, existing solutions available to data-oriented applications all have their set of issues. Additionally, applications may need several of those solutions to access different databases. We need a solution allowing programmers to write queries in their programming languages, able to use as many language functionalities as possible, with a unified interface to access all databases.

In this thesis, we define a new solution called BOLDR (**B**reaking boundaries **O**f **L**anguage and **D**ata **R**epresentations), a language-integrated query framework

Features	BOLDR	T-LINQ	LINQ	Calcite	ORMs	JDBC
Specify queries, dispatch, get results	✓	✓	✓	✓	✓	✓
Language-integrated queries	✓	✓	✓	✓	✓	✗
Translate some UDF into queries	✓	✓	✗	✗	✗	✗
Export host language environment	✓	✓/✗(1)	✓/✗(1)	✗	✗	✗
Different sources of same data model	✓	✓/✗(2)	✗	✓	✗	✗
Different data models	✓	✗	✓	✓	✗	✓
Multiple data source drivers available	✓	✗	✓	✓	✓	✓
Execute UDF in the database	✓	✓/✗(3)	✓/✗(3)	✗	✗	✗
Query merge and normalization	✓	✓	✗	✗	✗	✗
Single queries on different data sources	✓	✗	✗	✓	✗	✗
Early detection of errors in queries	✓	✓	✓	✓	✓	✗
Theoretical foundations	✓	✓	✗	✗	✗	✗

(1). Only for basic type identifiers (2). Not in the same query (3). Only for inlined UDFs

Figure 1.5 – Features of the different solutions

allowing application developers to write safe, complex, and efficient database-agnostic queries in their programming language of choice.

1.5.1 Features

Figure 1.5 gives a summarized comparison of the features of existing solutions. In a modern language-integrated query framework, we want all of the features listed in the figure. Queries should be: expressed in the language of the application; able to contain complex application logic; able to target several databases at once; optimized to be evaluated in the databases as much as possible; and checked for correctness before evaluation.

Just as LINQ does, BOLDR relies on an intermediate representation called QIR for **Q**uery **I**ntermediate **R**epresentation. In addition to the benefits of having independent interfaces for every host language and database, QIR rewrites the queries to make them easier to translate into database languages.

BOLDR *does not apply query plan optimizations*. More generally, BOLDR optimizes QIR queries to make the most of the databases optimizations using information unusable by databases, and so does not substitute itself for their optimization engines. The goal is to generate queries that can be as optimally handled by databases as possible.

The QIR allows BOLDR to perform type-checking on queries to detect errors before their evaluation. For instance, consider this query in R using BOLDR:

```
t = tableRef("people", "PostgreSQL")
q = query.filter(function (x) x$name > 5000, t)
```

Note that the name is compared to an integer. This query, that targets a *PostgreSQL* database, is syntactically correct, but returns an error during the evaluation because of this erroneous comparison. By type-checking the QIR version of the query, BOLDR can detect the problem even before the translation of the query into query languages, thus avoiding the process of sending an invalid query to a database via the network and its lot of performance issues.

Furthermore, BOLDR gives us guarantees on its processing of queries, such as the termination of the optimization phases, and the guarantee that a well-typed query can be translated into query languages.

BOLDR defines the interfaces between a host language and the framework, as well as between a database language and the framework. BOLDR is tied neither to a particular combination of a database language and a programming language, nor to querying only one database at a time. For instance, this query:

```
t1 = tableRef("people", "PostgreSQL")
t2 = tableRef("team", "HBase")
q = query.join(function (t1, t2) t1.teamid = t2.teamid, t1, t2)
```

is perfectly valid in BOLDR and performs the *join* operation of relational algebra on two tables coming from different databases, without the need for an explicit operation to import data in the runtime of the application. As described earlier, BOLDR automatically translates the subqueries into the different query languages of the different targeted databases, send them to the correct databases, retrieve the results and translate them back into the application language.

The framework also allows arbitrary expressions from the host language to occur in queries. Therefore, our problematic LINQ Example 1.4:

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee.Where(e => e.salary > dolToEuro(2500));
```

is also valid in BOLDR. The framework inlines the function in the query if it is possible and efficient. Otherwise it is kept as an application language function to be executed later in the database or in the application runtime itself. Either way, this query is evaluated successfully. This process is entirely automatized, programmers do not need to migrate their application code into databases.

1.5.2 Detailed description

The general flow of query evaluation in BOLDR is described in Figure 1.6. During the evaluation ① of a host program, queries are translated to QIR terms then sent to the QIR runtime for evaluation ②. These two steps do not need to be contiguous. Typically, the queries are translated at their creation, but evaluated only when the program needs to access the results. The QIR runtime then takes

over and attempts to type QIR terms. If it succeeds, it normalizes ③ the QIR terms to defragment them using a strategy that is guaranteed to succeed. This step is essential to allow our translators into database languages to function optimally. If the typing failed, a strategy based on the syntactical structure of QIR expressions is used for the normalization ④ which may fail. QIR terms are then typed again ⑤ to provide information for the translation as to where subterms should be executed, but also to check for errors before execution and to give us other interesting formal properties. If it succeeds, BOLDR translates the QIR terms to new QIR terms that contain database language queries (e.g., in SQL) using a translation strategy that is guaranteed to succeed as well. Otherwise, once again, it is a syntactic strategy that is used ⑥ which might fail. Next, the pieces of these terms are evaluated where they belong, either in main-memory ⑦ or in a database ⑧. Host language expressions occurring in these terms are evaluated either by the runtime of the host language that called the QIR evaluation ⑨, or in the runtime embedded in a target database ⑩. Results are then translated from the database into QIR ⑪, then from QIR into the host language ⑫.

Example 1.5 illustrates the key aspects of BOLDR. Our Example 1.5 is a standard R program with two exceptions: the function `tableRef` that returns a reference to a table from an external source; and the function `executeQuery` that evaluates a query. We recall that in R, the `c` function creates a vector, and the `subset` function filters a table using a predicate and optionally keeps only the specified columns. The first function `getRate` takes the code of two currencies and queries a table using `subset` to get their exchange rate. The second function `atLeast` takes 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.

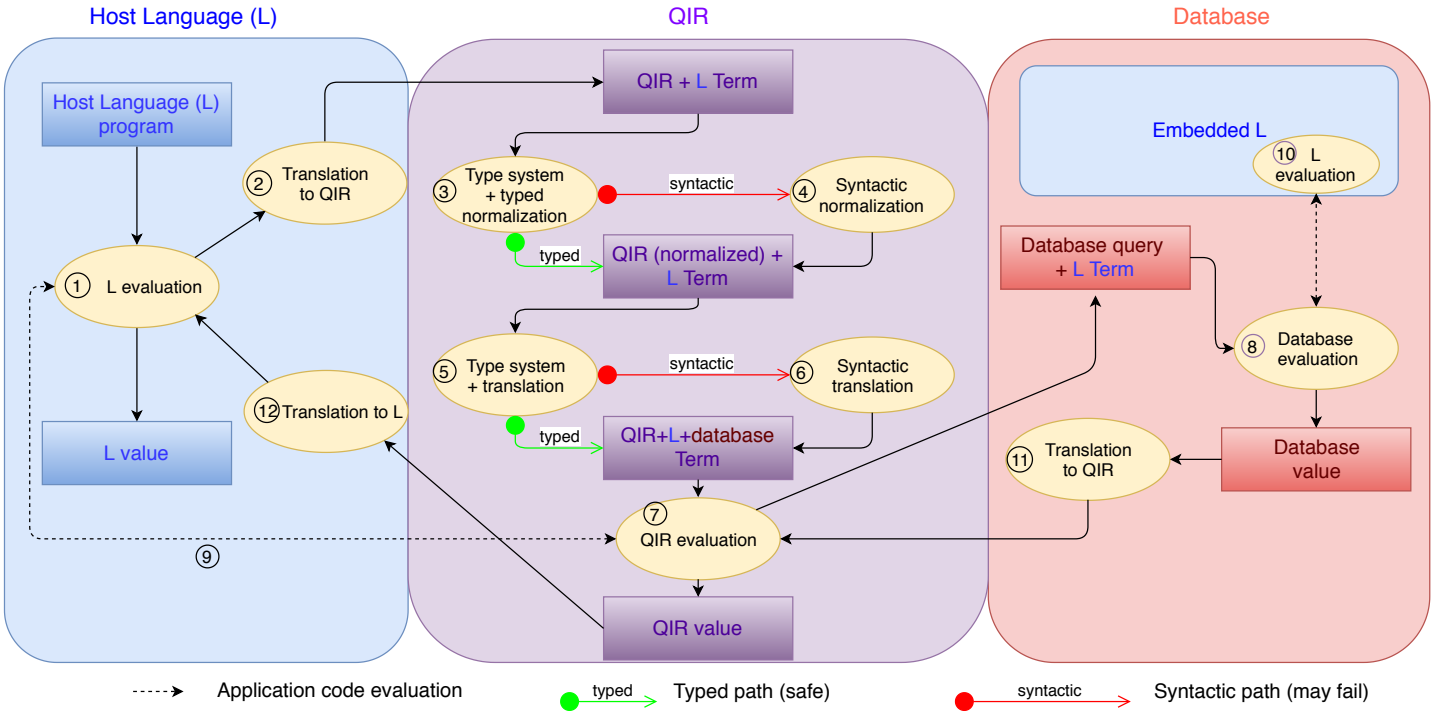


Figure 1.6 – Evaluation of a BOLDR host language program

Example 1.5.

```
# Exchange rate between rfrom and rto
getRate = function(rfrom, rto) {
  # table change has three columns: cfrom, cto, rate
  change = tableRef("change", "PostgreSQL")
  if (rfrom == rto) 1
  else subset(change, cfrom == rfrom && cto == rto, c(rate))
}

# Employees earning at least min in the cur currency
atLeast = function(min, cur) {
  # table employee has two columns: name, salary
  emp = tableRef("employee", "PostgreSQL")
  subset(emp, salary >= min * getRate("USD", cur), c(name))
}

richUSPeople = atLeast(2500, "USD")
richEURPeople = atLeast(2500, "EUR")
print(executeQuery(richUSPeople))
print(executeQuery(richEURPeople))
```

In BOLDR, `subset` is overloaded to build an intermediate query representation if applied on an external source reference. The evaluation of the first call to the function `atLeast` (`atLeast(2500, "USD")` found in Line 16) results in the creation of a query obtained by translation of the R expression into QIR. When `executeQuery` is called on the query, then (*i*) the runtime values linked to the free variables in the query are translated into QIR, then bound to these variables in the query, thus creating a *closed* QIR query; (*ii*) the query is *normalized*, process which in particular inlines bound variables with their values; (*iii*) the normalized query is translated into the target database language (here SQL); and (*iv*) the resulting query is evaluated in the database and the results are sent back. After normalization and translation, the query generated for the execution of `richUSPeople` is:

```
SELECT name FROM employee WHERE sal >= 2500 * 1
```

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

```
SELECT name FROM employee WHERE sal >= 2500 *
  (SELECT rate FROM change WHERE rfrom = "USD" AND rto = "EUR")
```

In this case, BOLDR merges subqueries together to create fewer and larger queries, thus benefiting from database optimizations as much as possible and avoiding the "query avalanche" phenomenon [GRS10].

User-defined functions that cannot be completely translated are also supported in BOLDR. For instance, consider Example 1.6.

Example 1.6.

```
getRate = function(rfrom, rto) {
  print(rto)
  change = tableRef("change", "PostgreSQL")
  if (rfrom == rto) 1
  else subset(change, cfrom == rfrom && cto == rto, c(rate))
}
```

This version of the function `getRate` calls the `print` function which cannot be translated into QIR, so instead BOLDR generates the following query:

```
SELECT name FROM employee
WHERE sal >= 2500 * 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 could refer to an HBase [Apaa] table in the function `getRate`. BOLDR would still be able to evaluate the query by sending a subquery to both the HBase and PostgreSQL databases, and by executing in main memory what could not be translated.

1.5.3 Implementation

Our implementation of BOLDR uses *Truffle* [WWW+13, Wim14], a framework developed by Oracle Labs to implement programming languages. Truffle allows language developers to implement abstract syntax tree (AST) interpreters with speculative runtime specialization. Language implementors typically write a parser for the target language that produces an AST composed of *Truffle nodes*. These nodes implement the basic operations of the AST interpreter (control-flow, typed operation on primitive types, object model operations such as method dispatch, ...), and use the Truffle API to implement runtime specialization and inform the JIT compiler of various key optimization aspects, such as runtime profiles on values, types, branches, or to implement runtime rewriting of the AST on de-optimization path when a speculative optimization failed.

Several features make Truffle appealing to BOLDR. First, Truffle implementations of languages must compile to an executable abstract syntax tree that BOLDR can directly manipulate, which, in particular, gives a simple way to translate queries into QIR. Second, languages implemented with Truffle can be executed on any Java Virtual Machine (JVM), although greater performance can be achieved when the JVM uses the Graal JIT-compiler [DWM14], which makes their addition as an external language effortless in databases written in Java (e.g., Cassandra, HBase, ...), and relatively simple in others such as PostgreSQL. Thus, it gives us the ability to execute any expression from any host language implemented by Truffle in databases. Third, several programming languages are already implemented, with varying degrees of maturity, on top of the framework, such as Zippy for Python [Orae]; JRuby for Ruby [Orac]; FastR for R [Oraa]; or Graal.js for JavaScript [Orab], and work on one Truffle language can easily be reused in these implementations.

Our implementation supports the *PostgreSQL*, *HBase* and *Hive* databases, as well as *FastR*, the Truffle implementation of the R language, and Oracle's *SimpleLanguage*, a core experimental dynamic language with syntax and features inspired by JavaScript (dynamically typed, prototype-based with high-order functions and a type system with just three primitive types: number, string and boolean). SimpleLanguage is developed by Oracle Labs to demonstrate the capabilities of Truffle. A detailed description of our implementation can be found in Chapter 7.

1.6 Contributions

This thesis studies the design of a framework for language-integrated queries with the formal definition and implementation of BOLDR and its different components. Chapter 2 gives some notations and definitions used throughout the document, and Chapter 8 concludes by discussing possible extensions and improvements. The chapters of this thesis each correspond to parts of the framework illustrated in Figure 1.6 and are described in the following subsections.

1.6.1 Query Intermediate Representation

Chapter 3, pages 27-53

The central point of BOLDR is its intermediate representation of queries called QIR. As stated earlier, a query is first translated into this representation before being translated to a database query. In this chapter, we define the language and its semantics ⑦⑨, including the semantics of data operators implemented in databases; a default database implementing important data operators to support queries that cannot be entirely translated into database languages; and the optimization applied on the queries before translation called the *QIR normalization* ④ which transforms a query to make it easier to translate into a database language. Indeed, our user-defined function application example:

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee.Where(e => e.salary > dolToEuro(2500));
```

is not an expression that can be translated as is into most databases languages because these languages usually do not easily allow the definition and application of user-defined functions. In particular, standard SQL does not support this feature (although it is possible to define routines which bodies are strictly limited to queries). Some databases support extensions of SQL (Oracle's PL/SQL [PL/b], Microsoft's T-SQL [T-S], ...) that allows the definition and application of user-defined functions, but this feature is not very optimized. Therefore, translating this query directly would either result in an error forcing the QIR runtime to handle most of its execution, or an inefficient query. For these reasons, we want to apply `dolToEuro` in the QIR before translation to generate an efficient query. Additionally, we define *drivers* which role is to interface QIR to a host language or to a database by providing translation functions from and into their language to QIR. To summarize, the contributions of this chapter are:

- A syntax and semantics for the QIR
- A reduction relation for the core calculus of QIR

- A reduction relation for the complete QIR making use of interfaces to host languages and databases
- A default database including default implementations of some data operators
- A best-effort normalization procedure that is guaranteed to terminate but with no formal properties

1.6.2 QIR type system

Chapter 4, pages 55-79

The evaluation of queries involves exchanging information with databases. This process can be very costly, depending on the amount of data involved, because of processing time and network delays. Thus, avoiding to send queries to databases when it is not needed, in particular when the queries are erroneous, is a major performance gain. Type systems are an efficient and classic way to detect in advance errors in programs. However, since BOLDR mostly targets dynamic host languages, expressions translated into QIR are untyped. Therefore, it makes sense to define a strong type system for the QIR to detect as many errors as possible before evaluation instead of relying entirely on the error detection of databases. Additionally, BOLDR supports queries targeting different databases, and different semantics for data operators depending on the database that evaluates them. Supporting these features requires being able to establish in which database each subexpression of a query should be evaluated.

In this chapter we define a compositional type system for QIR ③⑤ that we call the *generic* type system. Our generic type system is extendable with type systems provided by databases. These type systems, that we call *specific* type systems, allow database implementors to describe what expressions they support. Because of the unknown number of databases interfaced with BOLDR, and because queries might target several of those databases at the same time, this pattern of a generic process making specific components provided by databases work together is common in this thesis. To showcase how a database can provide a specific type system, we also define a type system for SQL in this chapter, as well as a type system for our default database, and we prove safety of execution properties obtained using our type systems.

1.6.3 QIR type inference

Chapter 5, pages 81-107

Type systems from Chapter 4 are designed for formal developments and presentation. However, these types systems are not algorithmic, and thus not directly suited for implementation.

In this chapter, we create typing algorithms ③⑤ using constraint solving of types, and prove that our typing algorithms are equivalent to the type systems of Chapter 4. We also define a constraint resolution algorithm, and prove it solves the constraints generated by our typing algorithms.

1.6.4 Type-oriented evaluation

Chapter 6, pages 109-128

In this chapter, we make use of our type systems to define the translation of a QIR expressions into database languages. Just as for the design of our type system, our translation from QIR to database languages is composed of specific translations provided by databases, and a generic translation that makes use of those specific translations. Our translation also makes use of our type system to translate as much of queries as possible into database languages, and leaves the rest to be evaluated by our default database. We define a syntactic translation ⑥ that triggers if the type system fails. Additionally, we define a translation for SQL and show that if our type system could type a QIR expression using our type system for SQL then it is translatable into SQL using our translation. Finally, we define a typed-oriented normalization ③.

1.6.5 Implementation and experiments

Chapter 7, pages 129-151

In this chapter, we interface the programming language R to BOLDR by defining a translation from R to QIR ②. We also describe our prototype implementation of BOLDR and present our results which show that BOLDR is able to inline most queries with user-defined functions, thus obtaining results at least as good as manually defined queries, and evaluates cross-databases queries and queries containing untranslatable expressions with decent performances.

Publications

The syntax and semantics of QIR described in Chapter 3, as well as parts of Chapter 4, 6, and 7 are presented in [BCD+18].

Chapter 2

Definitions

This chapter gives notations and definitions used in the rest of the thesis.

2.1 Basic notations

Definition 2.1 (Notations)**.**

- $\equiv$: syntactical equivalence. We denote its negation by $\not\equiv$.
- $=$: equality. Both terms are equal modulo some theory that is clear from the context. We denote its negation by $\neq$.
- $\emptyset$: the empty set.
- $\subseteq$: the subset inclusion.
- $\cup$, $\cap$, $\setminus$: respectively the union, intersection and difference of sets.
- $\texttt{dom}(f)$: the domain of a function f.
- $\texttt{img}(f)$: the image of a function f.
- $i..j$: the set of integers $\{i, i+1, \ldots, j\}$
- `_` : placeholder meaning any possible valid construct

In this thesis, we often substitute variables in a term with other terms. Our next definitions cover this operation.

Definition 2.2 (Substitution)**.** Let V be a set of variables and T a set of terms. A *substitution* is a partial function from V to T. We use the notation $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ when the domain is finite.

We use the notation $\{x \mapsto t'\}t$ to denote the term t into which every occurrence of the variable x has been replaced by the term t'.

Definition 2.3 (Type substitution)**.** A *type substitution* is a substitution from type variables to types $\{\alpha_1 \mapsto T_1, \ldots, \alpha_n \mapsto T_n\}$. We use σ to range over type substitutions.

Definition 2.4 (Composition of two type substitutions)**.** The *composition of two type substitutions* σ_1 and σ_2 noted $\sigma_1 \circ \sigma_2$ is defined as:

$$\sigma_1 \circ \sigma_2 = \left\{ \begin{array}{ll} \alpha \mapsto \sigma_1 T & \text{for each } \alpha \mapsto T \in \sigma_2 \\ \alpha \mapsto T & \text{for each } \alpha \mapsto T \in \sigma_1 \text{ with } \alpha \notin \texttt{dom}(\sigma_2) \end{array} \right\}$$

The composition of two type substitutions behaves, as expected, just like the composition of two functions: we first apply the second type substitution, then the first one.

Definition 2.5 (Typing environment)**.** A typing environment is a substitution from variables to types. We use Γ to range over type environments.

Definition 2.6 (Evaluation environment)**.** An evaluation environment is a substitution from variables to values. We use γ to range over evaluation environments.

2.2 Languages

To talk about interfaces between host languages and database languages with QIR, we first give a definition of these languages by listing what we expect them to define.

Definition 2.7 (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* (I for identifiers), $\mathsf{I}_\mathcal{H} \subset \mathsf{E}_\mathcal{H}$
- $\mathsf{V}_\mathcal{H}$ 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 an evaluation environment from variables to values and an expression and returns a new environment and a value resulting from the evaluation of the input expression.

Definition 2.8 (Database language)**.** A *database language* $\mathcal{D}$ with support for a host language $\mathcal{H}$ is a 3-tuple $(\mathsf{E}_{\mathcal{D}}, \mathsf{V}_{\mathcal{D}}, \xrightarrow{\mathcal{D}})$ where:

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

Similarly to host languages, we abstract a database language $\mathcal{D}$ as a syntax $\mathsf{E}_{\mathcal{D}}$, a set of values $\mathsf{V}_{\mathcal{D}}$, and an evaluation function $\xrightarrow{\mathcal{D}}$ which takes an $\mathcal{H}$ environment and a database expression and returns a new $\mathcal{H}$ environment and a database value. Such an evaluation function allows us to abstract the behavior of modern databases that support queries containing foreign function calls.

Note: We call an $\mathcal{L}$ environment an environment from $\mathsf{I}_{\mathcal{L}}$ to $\mathsf{V}_{\mathcal{L}}$.

2.3 Inference systems

Inference systems are a common way to describe recursive functions such as type systems or evaluation functions. Inference systems are composed of *inference rules*, themselves composed of zero or more *premisses* and a *conclusion*. An inference rule states that if its premisses are true, then so is the conclusion. A proof of a final conclusion is then created by chaining application of inference rules together until only axioms were reached, or until all premisses are true under hypotheses. Various works use these inference systems to build proofs [Pie02, Rey99]. The definitions of this section are taken from [LG09] which follows the presentation in [Acz77].

Chapter 4 and Chapter 5 define modular type systems that can be extended with other type systems provided by databases. In order to prove properties on these systems, we manipulate inference systems as syntactic objects. Therefore, we next define inference systems as syntactic objects.

Definition 2.9 (Inference rule)**.** Let $\mathcal{U}$ be a set of judgments. An *inference rule* is an ordered pair $(\mathcal{A}, c)$ where c is the *conclusion* of the inference rule and $\mathcal{A} \subseteq \mathcal{U}$ is the set of its *premises* or *antecedents*. If $\mathcal{A}$ is empty, the inference rule is called an *axiom*. An inference rule is usually noted:

$$\frac{\mathcal{A}}{c}$$

Intuitively, the conclusion c is true if the premises $\mathcal{A}$ are true.

Definition 2.10 (Inference system)**.** An inference system ranged over by Φ is a set of inference rules.

Definition 2.11 (Derivation)**.** A *derivation* (or *proof tree*) of a judgment c within an inference system is a tree of root node c whose nodes are labeled with judgments $j_n \in \mathcal{U}$ and such that for every node n, its label j_n and the labels $\mathcal{A}$ of its children correspond to an inference rule $(\mathcal{A}, c)$. A derivation is usually noted as a combination of inference rules:

$$\dfrac{\dfrac{b}{a} \quad d \quad \dfrac{\dfrac{g \quad h}{f}}{e}}{c}$$

Chapter 3

Query Intermediate Representation

As explained in Chapter 1, the Query Intermediate Representation (QIR) is a representation of queries that BOLDR uses as an intermediate between queries written in application languages and their translations into database languages. Using an intermediate representation of queries simplifies interfacing with BOLDR significantly, since the many different programming languages and databases simply need to interface with QIR, instead of creating numerous one-to-one interfaces. This also allows the framework to apply generic optimizations and verifications on the queries before translation. This chapter formally defines the QIR and its semantics.

We want the QIR to be able to represent:

- Data and data structures from data sources, and operations to retrieve data from the data structures
- Data operations such as the operations of relational algebra
- Basic programming language features such as functions
- Host language expressions

Data operations cannot all be supported by the QIR. An important number of different operators are implemented efficiently by data sources as a result of decades of expertise. Some operators may be very similar but present variations in their semantics in the different implementations given by databases. Similar approaches such as T-LINQ [CLW13] limit themselves to several important operators, and effort must be made to extend them to more operators. Instead, we design the QIR to allow databases to bring their own operators by interfacing them with the QIR via *drivers*. Thus, the QIR itself does not define data operations, and the choice of which operators to support in BOLDR in case the databases do not is a distinct issue that we also cover in this chapter.

3.1 Syntax

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

Definition 3.1 (QIR expressions)**.** Given a countable set of variables $\mathsf{I}_{\texttt{QIR}}$, we define the set of QIR *expressions*, denoted by $\mathsf{E}_{\mathit{QIR}}$ and ranged over by q, as the set of finite productions of the following grammar:

$$
\begin{array}{rcll}
q & ::= & x & \text{(variable)} \\
& | & \mathbf{fun}^{x}(x){\rightarrow}q & \text{(recursive function)} \\
& | & q\ q & \text{(application)} \\
& | & c & \text{(constant)} \\
& | & op & \text{(basic operation)} \\
& | & \mathbf{if}\, q\, \mathbf{then}\, q\, \mathbf{else}\, q & \text{(conditional expression)} \\
& | & \{\, l : q, \ldots, l : q \,\} & \text{(record)} \\
& | & q \bowtie q & \text{(record concatenation)} \\
& | & [\,] & \text{(empty list)} \\
& | & q :: q & \text{(list constructor)} \\
& | & q \mathbin{@} q & \text{(list concatenation)} \\
& | & q \cdot l & \text{(record destructor)} \\
& | & q\ \mathbf{as}\ x :: x\ ?\ q\ :\ q & \text{(list destructor)} \\
& | & o\langle q, \ldots, q \mid q, \ldots, q \rangle & \text{(data operator)} \\
& | & \blacksquare_{\mathcal{H}}(\gamma, e) & \text{(host language expression)}
\end{array}
$$

where $\mathcal{H}$ is a host language as described in Definition 2.7.

Notation 3.1. We use the following syntactic shortcuts:

- $\mathbf{fun}^{f}(x_1, \ldots, x_n){\rightarrow}q$ stands for $\mathbf{fun}^{f}(x_1){\rightarrow}(\ldots(\mathbf{fun}^{f}(x_n){\rightarrow}q))$
- $q\ (q_1, \ldots, q_n)$ stands for $(\ldots(q\ q_1)\ldots)\ q_n$
- $[\,q_1, \ldots, q_n\,]$ stands for $q_1 :: \ldots :: q_n :: [\,]$
- $\{\, l_i : q_i \,\}_{i=1..n}$ stands for $\{\, l_1 : q_1, \ldots, l_n : q_n \,\}$
- $q_1 = q_2$ stands for $= q_1, q_2$, and we use the same notation for all infix operators

QIR expressions include the terms of the λ-calculus, with functions optionally named. For instance $\mathbf{fun}(x){\rightarrow}x$ represents the anonymous identity function,

$\mathbf{fun}^{getRandomNumber}(x)\rightarrow 4$ represents a function named `getRandomNumber` that always returns 4. Obviously, recursive functions have to be named, for instance $\mathbf{fun}^{f}(x)\rightarrow f\ x$ represents a function named f that applies itself on its argument recursively. QIR functions are unary to simplify proofs, but functions with several arguments in an application program can classically be translated to functions returning functions, process known as *curryfication*. For instance, the anonymous function `function(f,x) f(x)` written in R that takes two arguments `f` and `x` and returns the application could be represented in QIR as $\mathbf{fun}(f)\rightarrow\mathbf{fun}(x)\rightarrow f\ x$ which represents an anonymous function that takes a function f and returns a function that takes an argument x and applies f to x. Similarly, most programming languages support functions with no arguments. These functions can easily be represented as functions taking one useless argument. For instance, `function() 2`, which is an anonymous R function with no argument and returning 2, could be represented in QIR as $\mathbf{fun}(x)\rightarrow 2$ where x could be any variable that is free in the body of the function, and the function call `(function() 2)()` could be represented in QIR as $(\mathbf{fun}(x)\rightarrow 2)\ 0$, where 0 could be any pure value. In practice, our implementation simply defines functions that can take zero, one, or several arguments.

QIR also has constants (integers, strings, ...), and builtin operations (arithmetic operations, ...). The data model consists of unordered records and ordered lists. For instance, the records $\{x:1, y:2\}$ and $\{y:2, x:1\}$ are equivalent, but the lists $[1, 2]$ and $[2, 1]$ are not. Records are deconstructed through field projections. For instance, $\{\, id:1 \,\}\cdot id = 1$. Lists are deconstructed by the *list matching* destructor whose four arguments are: the list to destruct, a 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. For instance, $[1, 2, 3]\ \mathbf{as}\ h :: t\ \mathbf{?}\ h\ \mathbf{:}\ 0$ represents the expression extracting the head of the list $[1, 2, 3]$, and $\mathbf{fun}(l)\rightarrow l\ \mathbf{as}\ h :: t\ \mathbf{?}\ h\ \mathbf{:}\ 0$ represents the function that returns the head of the list given as argument if it is not empty, and 0 otherwise.

The important additions to these mundane constructs are *data operators* and *host language expressions*. Data operators $o\langle q_1 \ldots, q_n \mid q'_1, \ldots, q'_m\rangle$ represent data operations to be evaluated by a database. Their arguments are divided 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. For instance, a `Filter` operator would have the filter function as configuration, the collection to be filtered as data argument. Finally, a host language expression $\blacksquare_{\mathcal{H}}(\gamma, e)$ is an opaque construct that contains an evaluation environment γ and an expression e of the host language $\mathcal{H}$. Their behavior is exactly the same as closures: during the translation from host language terms to QIR terms, a host language expression is closed together with the current host environment to be later executed, possibly by a remote runtime. The syntax of QIR is similar to the one of T-LINQ, but this addition of host language expressions simplifies

the translation of host programs to QIR since it can default to translating any expression of a host program to a host language expression of QIR. Optimally, the entire query should be translated to a QIR expression so BOLDR can apply optimizations, but even queries containing host language expressions will be successfully evaluated, although with consequences on performances as shown in Section 7.4.

Now we go through a few examples to understand the syntax of QIR for queries. We use data operators such as `Filter` in the following examples as simple symbols with no associated semantics. They simply represent data operations.

Example 3.1.

$$\texttt{Filter}\langle \textbf{fun}(x) \rightarrow x \leq 2 \mid [\,1, 2, 3\,] \rangle$$

Example 3.1 is a simple application of a `Filter` data operator. The configuration of this `Filter` is an anonymous function that returns true only if its argument is a number lower than or equal to 2. Its data argument is a QIR list of integers.

Example 3.2.

$$\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id \leq 2 \mid [\,\{id : 1\}, \{id : 2\}, \{id : 3\}\,] \rangle$$

Example 3.2 is another application of a `Filter` data operator. This time, the configuration of this `Filter` is an anonymous function that returns true only if the record field id of its argument is a number lower than or equal to 2, and its data argument is a QIR list of records with a field id associated to numbers.

Example 3.3.

$$\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot salary < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \mid \rangle \rangle$$

Example 3.3 is similar to Example 3.2, but the data argument of `Filter` is an application of a `From` operator instead of a QIR list. In this example, `From` is applied to a database $\mathcal{D}$ and a table name `"employee"`.

Notation 3.2. For readability reasons, we will write $\texttt{From}\langle \mathcal{D}, n \rangle$ instead of $\texttt{From}\langle \mathcal{D}, n \mid \rangle$ from now on, omitting the vertical bar, since `From` has two configurations and no sub-collections.

Example 3.4.

$$\texttt{Project}\langle \textbf{fun}(r) \rightarrow \{\, id : r \cdot id, name : r \cdot name, salary : r \cdot salary \,\} \mid \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot salary < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

Example 3.4 shows the equivalent of the query in Example 1.3 written in QIR. An operator `Project` has for configuration an anonymous function that returns a record containing the fields *id*, *name* and *salary* associated to their respected values in the record given as argument, and its data argument is the query of Example 3.3.

Example 3.5.

$$\texttt{Project}\langle \textbf{fun}(r) \rightarrow \{\, id : r \cdot id, name : r \cdot name, salary : (\blacksquare_R(\gamma, f))\ (r \cdot salary) \,\} \mid \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot salary < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

Example 3.5 is the same query as Example 3.4, except that the value associated with the label *salary* in the configuration of `Project` is the application of a host language expression that contains a function f written in the programming language R to the value associated with the label *salary* in the record given as argument of the configuration.

Example 3.6.

$$op\langle \textbf{fun}(r) \rightarrow r \cdot id, \texttt{true} \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Example 3.6 shows a QIR query using a specific database operator named *op* applied to a function and a boolean as configuration, and to a data argument.

Example 3.7.

$$\texttt{Filter}\langle (\textbf{fun}(x) \rightarrow (\textbf{fun}(r) \rightarrow r \cdot salary < x))\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Example 3.7 shows a `Filter` with a configuration that is an application returning a function.

Definition 3.2 (Children of a QIR expression)**.** The set of children of a QIR expression q, noted $\mathfrak{C}(q)$, is defined as:

- $\mathfrak{C}(x) = \mathfrak{C}(c) = \mathfrak{C}(op) = \mathfrak{C}([\,]) = \mathfrak{C}(\blacksquare_{\mathcal{H}}(\gamma, e)) = \emptyset$
- $\mathfrak{C}(\mathbf{fun}^{f}(x) \rightarrow q) = \{q\}$
- $\mathfrak{C}(q_1\ q_2) = \mathfrak{C}(q_1 \bowtie q_2) = \mathfrak{C}(q_1 :: q_2) = \mathfrak{C}(q_1 \mathbin{@} q_2) = \{q_1, q_2\}$
- $\mathfrak{C}(\mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3) = \mathfrak{C}(q_1\ \mathbf{as}\ h :: t\ ?\ q_2\ :\ q_3) = \{q_1, q_2, q_3\}$
- $\mathfrak{C}(\{\, l_i : q_i \,\}_{i=1..n}) = \mathfrak{C}(o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n \rangle) = \{q_1, \ldots, q_n\}$

Definition 3.3 (Pure QIR expression)**.** A QIR expression q is *pure* if $q \not\equiv \blacksquare_{\mathcal{H}}(\gamma, e)$ and $\forall q' \in \mathfrak{C}(q).q'$ is pure.

Thus, a QIR expression is pure if it does not contain any host language expression.

Definition 3.4 (Targeted databases)**.** We say that a database $\mathcal{D}$ is *targeted* by a QIR expression q if there is at least one occurrence of $\texttt{From}\langle\mathcal{D}, \_\rangle$ in the subexpressions of q. We note $\mathfrak{T}(q)$ the set of databases targeted by q.

The notion of targeted database is purely syntactic. A database is targeted by an expression if it is the configuration of a `From` in the expression.

3.2 Basic semantics

In this section, we define the semantics of the core calculus of QIR, which is the QIR without data operators and host language expressions. That is, we define the semantics that can be defined independently of host languages and databases.

Definition 3.5 (Values of QIR)**.** We define the set of *values of QIR*, noted $\mathsf{V}_{\mathsf{QIR}}$, as the set of finite productions of the following grammar:

$$v ::= \mathbf{fun}^{x}(x) \rightarrow q \mid c \mid op \mid \{\, l : v, \ldots, l : v \,\} \mid [\,] \mid v :: v$$

Definition 3.6 (Basic QIR semantics)**.** Let $\rightarrow^{\delta} \subseteq \mathsf{E}_{\mathsf{QIR}} \times \mathsf{E}_{\mathsf{QIR}}$ be a reduction relation for basic operators. We define the reduction relation of QIR expressions $\rightarrow\ \subseteq \mathsf{E}_{\mathsf{QIR}} \times \mathsf{E}_{\mathsf{QIR}}$. The set of rules used to derive $\rightarrow$ are given in Figure 3.1.

$$\text{(app-red1)}\ \frac{q_1 \to q_1'}{q_1\ q_2 \to q_1'\ q_2} \qquad \text{(app-red2)}\ \frac{q_2 \to q_2'}{v_1\ q_2 \to v_1\ q_2'} \qquad \text{(app-}\beta\text{)}\ \frac{}{(\mathbf{fun}^f(x){\to}q_1)\ v_2 \to \{f \mapsto \mathbf{fun}^f(x){\to}q_1, x \mapsto v_2\}q_1} \qquad \text{(app-op)}\ \frac{op\ v \to^\delta q}{op\ v \to q}$$

$$\text{(if-red)}\ \frac{q_1 \to q_1'}{\mathbf{if}\ q_1\ \mathbf{then}\ q_2\ \mathbf{else}\ q_3 \to \mathbf{if}\ q_1'\ \mathbf{then}\ q_2\ \mathbf{else}\ q_3} \qquad \text{(if-true)}\ \frac{}{\mathbf{if\,true\ then}\ q_2\ \mathbf{else}\ q_3 \to q_2} \qquad \text{(if-false)}\ \frac{}{\mathbf{if\,false\ then}\ q_2\ \mathbf{else}\ q_3 \to q_3}$$

$$\text{(rec-red)}\ \frac{q_m \to q_m'}{\{\, l_1 : v_1, \ldots, l_{m-1} : v_{m-1}, l_m : q_m, \ldots, l_n : q_n \,\} \to \{\, l_1 : v_1, \ldots, l_{m-1} : v_{m-1}, l_m : q_m', \ldots, l_n : q_n \,\}}$$

$$\text{(rconcat-red1)}\ \frac{q_1 \to q_1'}{q_1 \bowtie q_2 \to q_1' \bowtie q_2} \qquad \text{(rconcat-red2)}\ \frac{q_2 \to q_2'}{v_1 \bowtie q_2 \to v_1 \bowtie q_2'}$$

$$\text{(rconcat-rec)}\ \frac{}{\{\, l_i : q_i \,\}_{i=1..m} \bowtie \{\, l_i : q_i \,\}_{i=m+1..n} \to \{\, l_i : q_i \,\}_{i=1..n}}$$

$$\text{(lcons-red1)}\ \frac{q_1 \to q_1'}{q_1 :: q_2 \to q_1' :: q_2} \qquad \text{(lcons-red2)}\ \frac{q_2 \to q_2'}{v_1 :: q_2 \to v_1 :: q_2'} \qquad \text{(lconcat-red1)}\ \frac{q_1 \to q_1'}{q_1 \mathbin{@} q_2 \to q_1' \mathbin{@} q_2} \qquad \text{(lconcat-red2)}\ \frac{q_2 \to q_2'}{v_1 \mathbin{@} q_2 \to v_1 \mathbin{@} q_2'}$$

$$\text{(lconcat-lempty)}\ \frac{}{[\,] \mathbin{@} v \to v} \qquad \text{(lconcat-rempty)}\ \frac{}{v \mathbin{@} [\,] \to v}$$

$$\text{(lconcat-lcons)}\ \frac{}{(v_1 :: v_2) \mathbin{@} v_3 \to v_1 :: (v_2 \mathbin{@} v_3)} \qquad \text{(rdestr-red)}\ \frac{q \to q'}{q \cdot l \to q' \cdot l} \qquad \text{(rdestr-rec)}\ \frac{}{\{\ldots, l : v, \ldots\} \cdot l \to v}$$

$$\text{(ldestr-red)}\ \frac{q_1 \to q_1'}{q_1\ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 \to q_1'\ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3} \qquad \text{(ldestr-empty)}\ \frac{}{[\,]\ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 \to q_3}$$

$$\text{(ldestr-nonempty)}\ \frac{}{v_1 :: v_1'\ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 \to q_2\ (v_1, v_1')}$$

$$\text{(dataop-conf)}\ \frac{q_k \to q_k'}{o\langle v_1, \ldots, v_{k-1}, q_k, \ldots, q_m \mid q_{m+1}, \ldots, q_n\rangle \to o\langle v_1, \ldots, v_{k-1}, q_k', \ldots, q_m \mid q_{m+1}, \ldots, q_n\rangle}$$

$$\text{(dataop-data)}\ \frac{q_k \to q_k'}{o\langle v_1, \ldots, v_m \mid v_{m+1}, \ldots, v_{k-1}, q_k, \ldots, q_n\rangle \to o\langle v_1, \ldots, v_m \mid v_{m+1}, \ldots, v_{k-1}, q_k', \ldots, q_n\rangle}$$

Figure 3.1 – QIR reduction rules

The result of record concatenation is a record containing all the labels of both records. Note that QIR only supports record concatenation between records which labels are strictly distinct.

Note also that the basic semantics of QIR is small-step semantics.

As an example, the expression $(\mathbf{fun}(x) \to \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, 2)\ (not\ \texttt{false})$ would be reduced as:

$$\begin{array}{ll} & (\mathbf{fun}(x) \to \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, 2)\ (not\ \texttt{false}) \\ \to & (\mathbf{fun}(x) \to \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, 2)\ \texttt{true} \\ \to & \mathbf{if}\, \texttt{true}\, \mathbf{then}\, 1\, \mathbf{else}\, 2 \\ \to & 1 \end{array}$$

Crucially, embedded host expressions as well as database operator applications whose arguments are all reduced are $\to$ -irreducible. It is the job of databases and application languages to evaluate these expressions.

Thus, our examples of Section 3.1 are all irreducible since they are made of data operators and functions except for Example 3.7 which would be reduced as such:

$$\begin{array}{ll} & \texttt{Filter}\langle(\mathbf{fun}(x) \to (\mathbf{fun}(r) \to r \cdot id < x))\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \\ \to & \texttt{Filter}\langle \mathbf{fun}(r) \to r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \end{array}$$

using the (dataop-conf) rule with the (app-β) rule as premise. Therefore, the basic semantics does not reduce the data operators themselves, but it does reduce the arguments of data operators, thus allowing complex expressions in the arguments of data operators such as applications or conditional expressions.

We defined the semantics of the core calculus of QIR. Now, we define its complete semantics including data operators and host language expressions.

3.3 Extended semantics

In Section 3.2, we gave the semantics of QIR, but omitting the semantics of data operators and host language expressions. In this section, we complete those semantics, and to do so we first define how to interface host languages and databases with QIR.

The external component that wants to interface with BOLDR has to be able to perform translations from QIR into their language, and from their language into QIR. To be precise, host languages must translate

- their values to QIR values
- their expressions along with their associated runtime environment to closed QIR expressions
- QIR values to their values

and databases must translate

- QIR values to values of their query language
- QIR expressions to expressions of their query language
- values of their query language to QIR values

Since we target dynamic host languages, expressions from host languages to be translated are associated with a runtime environment. For instance, take the R query of the example in Chapter 1:

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

This query does not make sense outside the proper environment since the variables `emp`, `minSalary`, `getRate`, and `cur` are all free in this expression. It can only have a meaning within an environment that associates these free variables to values. If the original program is correct, as it is in our example, then the runtime environment would indeed include those free variables in its domain when the translation of the query is triggered. The host language must then translate the expression using the runtime environment to a closed QIR term. For instance, the result of the call:

```
16 richUSPeople = atLeast(2500, "USD")
```

could be translated to:

$$
\begin{array}{l}
(\textbf{fun}(emp, minSalary, getRate, cur) \rightarrow \\
\quad \texttt{Project}\langle \textbf{fun}(x) \rightarrow \{name : x \cdot name\} \mid \\
\quad \texttt{Filter}\langle \textbf{fun}(x) \rightarrow x \cdot salary \geq minSalary * (getRate\ \texttt{"USD"}\ cur) \mid \\
\quad emp \rangle \rangle) \\
(\texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle, 2500, \textbf{fun}(rfrom, rto) \rightarrow \ldots, \texttt{"USD"})
\end{array}
$$

The translation of our query is wrapped in functions binding the free variables, then applied to the translation of the values associated to these variables, thus creating a closed QIR expression. As we can see in this example, this process creates convoluted queries that can be difficult to translate as is, we will see later in this chapter how BOLDR simplifies these queries to make them easier to translate into database languages.

We now introduce the notion of *driver*, which defines a translation interface between a host language or a database and QIR.

Definition 3.7 (Host language driver)**.** Let $\mathcal{H}$ be a host language. A *host language driver* for $\mathcal{H}$ is a 3-tuple $({}^{\mathcal{H}}\overrightarrow{\texttt{EXP}}, \overrightarrow{\texttt{VAL}}^{\mathcal{H}}, {}^{\mathcal{H}}\overrightarrow{\texttt{VAL}})$ of total functions such that:

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

where the special value Ω denotes a failure to translate.

Definition 3.8. (Database driver) Let $\mathcal{D}$ be a database language. A *database driver* for $\mathcal{D}$ is a 3-tuple $(\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}, \overrightarrow{\texttt{VAL}}^{\mathcal{D}}, {}^{\mathcal{D}}\overrightarrow{\texttt{VAL}})$ of total functions such that:

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

where the special value Ω denotes a failure to translate.

A host language or database is required to define a driver to interface with QIR. From this point, we refer to $\mathbb{H}$ and $\mathbb{D}$ respectively as the set of host languages and databases that are interfaced with QIR, so for which there exists a corresponding host language driver or database driver.

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

Definition 3.9 (Extended QIR semantics)**.** We define the extended semantics $\gamma, q \twoheadrightarrow \gamma', q'$ of QIR by the following set of rules:

$$\frac{\gamma, e \xrightarrow{\mathcal{D}} \gamma', v}{\gamma, \texttt{eval}^{\mathcal{D}}(e) \twoheadrightarrow \gamma', {}^{\mathcal{D}}\overrightarrow{\texttt{VAL}}(v)}\ {\scriptstyle v \neq \Omega}\ \text{(ext-eval)} \qquad \frac{\gamma \cup \gamma', e \xrightarrow{\mathcal{H}} \gamma'', v}{\gamma, \blacksquare_{\mathcal{H}}(\gamma', e) \twoheadrightarrow \gamma'', {}^{\mathcal{H}}\overrightarrow{\texttt{VAL}}(v)}\ {\scriptstyle v \neq \Omega}\ \text{(ext-host)}$$

$$\frac{\overrightarrow{\texttt{EXP}}^{\mathcal{D}}(q) = e \quad \gamma, \texttt{eval}^{\mathcal{D}}(e) \twoheadrightarrow \gamma', v}{\gamma, q \twoheadrightarrow \gamma', v}\ \begin{array}{l}\mathcal{D} \in \mathbb{D} \\ e \neq \Omega \\ v \neq \Omega\end{array}\ \text{(ext-database)} \qquad \frac{q \to q'}{\gamma, q \twoheadrightarrow \gamma, q'}\ \text{(ext-basic)}$$

where (ext-database) is always prioritized over (ext-basic).

Since QIR is an intermediate language from a host language to a database language, the evaluation of QIR terms is always 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. To allow the QIR evaluator to send queries to a database and translate the results back to QIR values, we define a basic data operator $\texttt{eval}^{\mathcal{D}}$ for each supported database language $\mathcal{D} \in \mathbb{D}$. This operator represents the evaluation of an expression from a database language into the corresponding database, and abstracts the low-level processing required to access that database. BOLDR makes use of $\texttt{eval}^{\mathcal{D}}$ internally as we will see in Chapter 6, but it can also be used to express queries directly written in a database language, either for debugging or optimization purposes. The evaluation of an expression by rule (ext-database) consists in *(i)* finding a database language in which this expression can be translated, *(ii)* use the database driver for that language to translate the QIR term to a native query, *(iii)* use the evaluation function of the database to evaluate the expression, and *(iv)* translate the results back into QIR. Note that the evaluation of an expression in a database could return a different host language environment, since host language expressions might appear in its subexpressions. For instance, recall our Example 3.5:

$$\texttt{Project}\langle \textbf{fun}(r) \to \{\, id : r \cdot id, name : r \cdot name, salary : (\blacksquare_R(\gamma, f))\ (r \cdot salary) \,\} \mid \texttt{Filter}\langle \textbf{fun}(r) \to r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

If the database accepts host language expressions, then $\blacksquare_R(\gamma, f)$ is evaluated in the database runtime and returns a new host language environment γ' that may be different from γ if side effects occur. The rule (ext-eval) bypasses the research of the correct database by evaluating directly the database expression e in $\texttt{eval}^{\mathcal{D}}(e)$. We use this construct and this rule in Chapter 6. Host language expressions are evaluated by rule (ext-host) using the evaluation relation of the host language in the environment formed by the union of the current running

environment and the captured environment. This allows us to simulate the behavior of most dynamic languages (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. Finally, if the QIR term is neither a database operator nor a host language expression, then the simple semantics of Definition 3.6 is used to evaluate the term with the rule (ext-basic).

With this extended semantics, we can now fully evaluate our Example 3.3:

$$\texttt{Filter}\langle \textbf{fun}\,(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Supposing the database $\mathcal{D}$ supports `Filter` and `From`, we are able to directly apply the data operator rule of $\twoheadrightarrow$.

As stated in Definition 3.9, the rules (ext-database) and (ext-host) are always prioritized over rule (ext-basic). For instance, recall Example 3.7:

$$\texttt{Filter}\langle (\textbf{fun}\,(x) \rightarrow (\textbf{fun}\,(r) \rightarrow r \cdot id < x))\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Two rules can be applied here: we can attempt to use (ext-database) to directly translate the entire expression into a database language and execute it in the corresponding database, or we can use (ext-basic) to reduce the configuration of `Filter`. The goal being to execute as much as possible in databases, we attempt (ext-database) first, then if it fails we do one step of reduction using (ext-basic), then attempt (ext-database) again. This semantics is the most naive way to try to execute as much as possible in databases. For instance, in our Example 3.7, if we assume we cannot translate the `Filter` because we cannot translate the configuration, the rule (ext-database) would fail, and we would apply (ext-basic) to get

$$\texttt{Filter}\langle \textbf{fun}\,(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

and then apply (ext-database) again, this time successfully. We can also reach a point where both (ext-basic) and (ext-database) could be applied. For instance, consider the following query:

$$\begin{aligned}\texttt{Filter}\langle \textbf{fun}\,(r) \rightarrow r \cdot salary > (\textbf{if}\, true\, \textbf{then}\, 2500\, \textbf{else}\, 2000) \mid \\ \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle\end{aligned}$$

which returns the employees whose salary is greater than 2500. We could either use (ext-database) to translate the entire query to $\mathcal{D}$, assuming the database supports everything in this query, or (ext-basic) to reduce the conditional expression. Choosing (ext-database) is the correct choice here, since this entire query can be translated to a single $\mathcal{D}$ query, it would therefore be suboptimal to evaluate the conditional expression of the configuration of `Filter` in the runtime of QIR.

Giving priority to the rule (ext-host) is a best-effort addition which allows us to avoid problems in semantics by evaluating host language expressions as soon as possible. For instance:

$$(\mathbf{fun}(x) \rightarrow x + x)\ (\blacksquare_R(\gamma, print(2); 2))$$

would be reduced using rule (app-β) to:

$$(\blacksquare_R(\gamma, print(2); 2)) + (\blacksquare_R(\gamma, print(2); 2))$$

in which the host language expression has been duplicated which we absolutely do not want, since we changed the semantics by executing the side effects (here printing a value) twice. But if we execute the host language expression first, then we get the desired results of printing 2 once, then return 4.

3.4 A default database language: MEM

Ideally, we will translate all QIR expressions to expressions of database languages. However, parts of QIR expressions might be impossible and/or inefficient to translate and evaluate in databases. This can happen for different reasons. For instance, consider the query of Example 3.8.

Example 3.8.

$$\texttt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \mathbf{fun}(x, y) \rightarrow \texttt{true} \mid \texttt{From}\langle \mathsf{HBase}, \texttt{"employee"} \rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"} \rangle \rangle$$

This query uses the `Join` operation of relational algebra described in Section 1.2, and applies it on two tables provided by an HBase database. Since HBase does not support `Join`, this query is not translatable to a HBase query. Only the two `From` subexpressions are translatable. Even a query referring only to data operators supported by its targeted database may contain subexpressions impossible to translate, as we can see in Example 3.9.

Example 3.9.

$$\texttt{Project}\langle \mathbf{fun}(r) \rightarrow r \bowtie \{treated : \texttt{true}\} \mid \texttt{From}\langle \mathsf{HBase}, \texttt{"employee"} \rangle \rangle$$

Although HBase supports `Project`, it can only apply it to simple predicates, so the `Project` of Example 3.9 is not translatable into the query language of HBase.

Another reason that would make an expression untranslatable is if several databases are targeted. For instance, consider the query of Example 3.10.

Example 3.10.

$$\texttt{Join}\langle \textbf{fun}\,(x, y) \to x \bowtie y, \textbf{fun}\,(x, y) \to \texttt{true} \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathcal{D}', \texttt{"team"}\rangle\rangle$$

This query uses `Join` on two tables provided by two different databases $\mathcal{D}$ and $\mathcal{D}'$. Neither $\mathcal{D}$ nor $\mathcal{D}'$ is able to translate this expression since they are unable to access the data stored in the other database.

The solution to this problem is to have a default implementation of data operators. To that end, we define a default database that supports some important data operators. This database is dubbed MEM for in-memory evaluation. MEM directly uses QIR as its database language. MEM supports the operators `Filter`, `Project`, and `Join` defined as plain QIR recursive functions. This constitutes a first attempt at defining a core set of operators that are always supported by BOLDR, whether or not there are drivers interfaced with the framework that support these operators.

The definition of the set of *supported* database operators is an important design choice. It should be broad enough so host languages users do not have to re-implement operators, and generic enough so they can write generic queries and so that translating queries from QIR into a database language stays manageable. Thus, the choice to support an operator or not can be difficult to make, as an operator may be specific to a particular data model (e.g., computing the transitive closure in a graph database); or generic enough but not natively supported by some back-ends (NoSQL databases usually do not support join operations). `Project`, `Filter`, and `Join` are very common operations that we use as a starting point in our formal developments.

Definition 3.10 (MEM database language)**.** The *MEM language* $\mathsf{MEM} = (\mathsf{E}_{\mathsf{MEM}}, \mathsf{V}_{\mathsf{MEM}}, \overset{\mathsf{MEM}}{\Rightarrow})$ is defined by:

- $\mathsf{E}_{\mathsf{MEM}} = \mathsf{E}_{\mathsf{QIR}}$
- $\mathsf{V}_{\mathsf{MEM}} = \mathsf{V}_{\mathsf{QIR}}$
- $\overset{\mathsf{MEM}}{\Rightarrow}$ is the extended semantics of the QIR (relation $\twoheadrightarrow$ in Definition 3.9)

The values of MEM do not include data operators, since they must be evaluated or return an error if unsupported, nor does it include host language expressions that must be evaluated using the evaluation function of the corresponding host language. We now complete the semantics of the MEM database language with the definition of a driver for MEM.

Definition 3.11 (MEM driver)**.** The driver for the MEM database language is the 3-tuple $(\overrightarrow{\texttt{EXP}}^{\mathsf{MEM}}, \overrightarrow{\texttt{VAL}}^{\mathsf{MEM}}, {}^{\mathsf{MEM}}\overrightarrow{\texttt{VAL}})$ of total functions such that:

- $\overrightarrow{\texttt{VAL}}^{\mathsf{MEM}} : \mathsf{V}_{\mathsf{QIR}} \to \mathsf{V}_{\mathsf{MEM}} \cup \{\Omega\}$ is the identity function.
- ${}^{\mathsf{MEM}}\overrightarrow{\texttt{VAL}} : \mathsf{V}_{\mathcal{D}} \to \mathsf{V}_{\mathsf{QIR}} \cup \{\Omega\}$ is the identity function.
- $\overrightarrow{\texttt{EXP}}^{\mathsf{MEM}}(q) : \mathsf{E}_{\mathsf{QIR}} \to \mathsf{E}_{\mathsf{MEM}}$ is defined by case as
  - If $q = \texttt{Filter}\langle f \mid l\rangle$, then

    $(\mathbf{fun}^{\mathsf{filter}}(l) \to l \ \mathbf{as}\ h :: t\ \mathbf{?}\ \mathbf{if}\ f\ h\ \mathbf{then}\ h :: (\mathsf{filter}\ t)\ \mathbf{else}\ (\mathsf{filter}\ t)\ \mathbf{:}\ [\,])\ l$
  - Else if $q = \texttt{Project}\langle f \mid l\rangle$, then

    $$(\mathbf{fun}^{\mathsf{project}}(l) \to l \ \mathbf{as}\ h :: t\ \mathbf{?}\ (f\ h) :: (\mathsf{project}\ t)\ \mathbf{:}\ [\,])\ l$$
  - Else if $q = \texttt{Join}\langle f_1, f_2 \mid l_1, l_2\rangle$, then

    $(\mathbf{fun}^{\mathsf{join}}(l) \to \texttt{Project}\langle f_1 \mid l\ \mathbf{as}\ h_1 :: t_1\ \mathbf{?}$
    $(\texttt{Project}\langle \mathbf{fun}\,(h_2) \to h_1 \bowtie h_2 \mid \texttt{Filter}\langle f_2\ h_1 \mid l_2\rangle\rangle)\ @\ (\mathsf{join}\ t_1)\ \mathbf{:}$
    $[\,]\rangle)\ (l_1)$
  - Else if $q = o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n\rangle$, then Ω
  - Else q

The definition of the operators supported by MEM is a generalization of the relational algebra semantics described in Section 1.2. $\texttt{Filter}\langle f \mid l\rangle$ 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* to functional programmers) 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.

3.5 QIR normalization

3.5.1 Motivation

Translation from QIR into database languages might lead to suboptimal query generation. In particular, if some parts of a query cannot be translated, their results have to be transferred into the QIR runtime, and processed there.

The usual answer to this problem is to *normalize* the query, by applying rewriting rules on the intermediate representation. For instance, consider the term from Example 3.11.

Example 3.11.

$$(\mathbf{fun}(a,b) \to \texttt{Join}\langle \mathbf{fun}(x,y) \to x \bowtie y, \mathbf{fun}(x,y) \to x \cdot id = y \cdot id \mid a, b\rangle)$$
$$(\texttt{From}\langle \text{PostgreSQL}, \texttt{"people"}\rangle, \texttt{From}\langle \text{PostgreSQL}, \texttt{"dept"}\rangle)$$

Using directly a translation into SQL that does not handle anonymous functions, we would obtain the term:

$$(\mathbf{fun}(a,b) \to \texttt{Join}\langle \mathbf{fun}(x,y) \to x \bowtie y, \mathbf{fun}(x,y) \to x \cdot id = y \cdot id \mid a, b\rangle)$$
$$(\texttt{eval}^{\text{PostgreSQL}}(\texttt{SELECT * FROM PEOPLE}), \texttt{eval}^{\text{PostgreSQL}}(\texttt{SELECT * FROM DEPT}))$$

which is suboptimal. Indeed, although they target the same database, the two `From` subqueries are evaluated separately and worse, the `Join` is performed in main memory. Therefore, not only do we attempt to transfer the entire data from the tables `PEOPLE` and `DEPT` in the application, we use a less efficient version of `Join` to perform the query even though PostgreSQL could evaluate the entire query efficiently by itself. However, if we apply the β-reduction before translating, the query becomes:

$$\texttt{Join}\langle \mathbf{fun}(x,y) \to x \bowtie y, \mathbf{fun}(x,y) \to x \cdot id = y \cdot id \mid$$
$$\texttt{From}\langle \text{PostgreSQL}, \texttt{"people"}\rangle, \texttt{From}\langle \text{PostgreSQL}, \texttt{"dept"}\rangle\rangle$$

which can then be translated to a single SQL query:

```
SELECT * FROM PEOPLE AS x INNER JOIN DEPT AS y ON x.id = y.id
```

A naive solution implemented by the semantics of Definition 3.9 is to alternate between translating the expression and applying one step of reduction to the term. Obviously, this solution is not ideal as it makes several attempts at translating. Another naive solution is to attempt to reduce the term as much as possible, which leads to two problems. First, a QIR term may diverge, as we cannot guarantee the termination of the reduction, in particular if the expression contains the application of a recursive function. Second, a reduction may duplicate some data

operators, thus making the query less efficient. For instance, consider the QIR function:

$$\mathbf{fun}(a) \rightarrow \texttt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \mathbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid a, a \rangle$$

Given a collection a, the function performs a self join on a (finding pairs of elements of a such that the second one is older than the first). Applying the beta-reduction gives us two different outcome depending on the targeted database. If we apply this function to a table from a database that supports the `Join` operator, such as $\texttt{From}\langle \text{PostgreSQL}, \texttt{"people"} \rangle$, then reducing is beneficial, since it produces a whole `Join` query that can be sent to the database:

```
SELECT * FROM PEOPLE AS x INNER JOIN PEOPLE AS y ON x.age < y.age
```

instead of:

$$(\mathbf{fun}(a) \rightarrow \texttt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \mathbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid a, a \rangle)$$
$$(\texttt{eval}^{\text{PostgreSQL}}(\texttt{SELECT * FROM PEOPLE}))$$

However, if we apply this function to a table from a database that does not support the `Join` operator, such as $\texttt{From}\langle \text{HBase}, \texttt{"people"} \rangle$, then it is better not to apply the beta-reduction that would duplicate the argument, which would send two queries to HBase, since the join has to be evaluated in-memory anyway:

$$(\mathbf{fun}(a) \rightarrow \texttt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \mathbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid a, a \rangle)$$
$$(\texttt{eval}^{\text{HBase}}(\texttt{scan 'people'}))$$

instead of:

$$\texttt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \mathbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid$$
$$\texttt{eval}^{\text{HBase}}(\texttt{scan 'people'}), \texttt{eval}^{\text{HBase}}(\texttt{scan 'people'}) \rangle$$

Therefore, we have to find a middle ground between trying to fully reduce the term and yield the most translatable term but risk diverging, and translating the term without any preliminary reduction at the risk of introducing query avalanches.

3.5.2 Reduction relation for the normalization

As stated earlier, we want to use the reduction relation $\rightarrow$ of Definition 3.6 as the reduction relation for the normalization. We cannot use the relation $\twoheadrightarrow$ since we do not want to reduce data operators or host language expressions that must be evaluated by the databases.

However, as explained in Section 3.3, using $\rightarrow$ directly would cause us problems in QIR expressions that contain applications to host language expressions,

as it could lead to a duplication of side effects. Therefore, we restrict the β-reductions our normalization can perform to applications that does not contain host language expressions in their argument, thus avoiding the problematic cases.

Additionally, we want to add a rule to reduce the body of functions, which allows for more queries to be translatable into database languages. For instance, consider this QIR expression:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow (\mathbf{fun}(x) \rightarrow r \cdot \mathit{salary} > x)\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Again, this is not translatable in most databases languages because of the application of an anonymous function. We cannot reduce this expression using $\rightarrow$. Reducing the body of QIR functions allows us to reduce this expression to:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Therefore, we define a new reduction relation for the normalization, and its normal forms.

Definition 3.12 (QIR normal form)**.** A QIR expression q is a *normal form* of the QIR, ranged over by v, is denoted by the judgment $\vdash_{\text{NF}} q$ which is inferred by the rules:

$$\frac{}{\vdash_{\text{NF}} x} \qquad \frac{\vdash_{\text{NF}} v}{\vdash_{\text{NF}} \mathbf{fun}^f(x) \rightarrow v} \qquad \frac{\vdash_{\text{NF}} v_1 \quad \vdash_{\text{NF}} v_2}{\vdash_{\text{NF}} v_1\ v_2}\ \begin{array}{l} v_1 \not\equiv \mathbf{fun}^f(x) \rightarrow v \\ v_1 \not\equiv op \end{array}$$

$$\frac{\vdash_{\text{NF}} v_1 \quad \vdash_{\text{NF}} v_2}{\vdash_{\text{NF}} (\mathbf{fun}^f(x) \rightarrow v_1)\ v_2}\ v_2 \text{ not pure} \qquad \frac{}{\vdash_{\text{NF}} c} \qquad \frac{}{\vdash_{\text{NF}} op} \qquad \frac{}{\vdash_{\text{NF}} \blacksquare_{\mathcal{H}}(\gamma, e)}$$

$$\frac{\vdash_{\text{NF}} v_1}{\vdash_{\text{NF}} \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3}\ \begin{array}{l} v_1 \neq \texttt{true} \\ v_1 \neq \texttt{false} \end{array} \qquad \frac{}{\vdash_{\text{NF}} \{\, l_i : v_i \,\}_{i=1..n}}$$

$$\frac{\vdash_{\text{NF}} v_1 \quad \vdash_{\text{NF}} v_2}{\vdash_{\text{NF}} v_1 \bowtie v_2}\ v_1 \not\equiv \{\, l'_i : v'_i \,\}_{i=1..n} \text{ or } v_2 \not\equiv \{\, l'_i : v'_i \,\}_{i=1..n} \qquad \frac{}{\vdash_{\text{NF}} [\,]} \qquad \frac{\vdash_{\text{NF}} v_1 \quad \vdash_{\text{NF}} v_2}{\vdash_{\text{NF}} v_1 :: v_2}$$

$$\frac{\vdash_{\text{NF}} v_1 \quad \vdash_{\text{NF}} v_2}{\vdash_{\text{NF}} v_1 \mathbin{@} v_2}\ \begin{array}{l} v_1 \not\equiv [\, v'_1, \ldots, v'_n \,] \\ v_2 \not\equiv [\,] \end{array} \qquad \frac{\vdash_{\text{NF}} v}{\vdash_{\text{NF}} v \cdot l}\ v \not\equiv \{\, l'_i : v'_i \,\}_{i=1..n}$$

$$\frac{\vdash_{\text{NF}} v_1}{\vdash_{\text{NF}} v_1\ \mathbf{as}\ h :: t\ ?\ v_2\ :\ v_3}\ v_1 \not\equiv [\, v'_1, \ldots, v'_n \,] \qquad \frac{\vdash_{\text{NF}} v_1 \quad \ldots \quad \vdash_{\text{NF}} v_n}{\vdash_{\text{NF}} o\langle v_1, \ldots, v_m \mid v_{m+1}, \ldots, v_n \rangle}$$

For instance, 2, $\mathbf{fun}(x) \rightarrow 2$, $\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, 2$, $x \cdot \mathit{name}$, and $\texttt{Filter}\langle \mathbf{fun}(x) \rightarrow x \cdot \mathit{teamid} = 2 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$ are in normal form, but $\mathbf{fun}(x) \rightarrow \mathbf{if}\, \texttt{true}\, \mathbf{then}\, x\, \mathbf{else}\, x$, and $\texttt{From}\langle \mathcal{D}, \mathit{concat}\ (\text{"}\mathit{mySchema.}\text{"}, \text{"}\mathit{employee}\text{"}) \rangle$ are not.

We use Definition 3.12 for the definition of a normal form rather than the usual definition from the λ-calculus to handle cases such as:

$$\mathbf{fun}(x)\to((\mathbf{if}\,\texttt{Filter}\langle\ldots\mid\ldots\rangle\,\mathbf{then}\,\mathbf{fun}(y)\to y\,\mathbf{else}\,\mathbf{fun}(y)\to y\ *10)\ (x))$$

Indeed, since data operators are values for the normalization, normalized expressions can include data operators which could return any type of value.

Definition 3.13 (Normalization reduction relation)**.** The reduction relation of QIR expressions $\hookrightarrow\ \subseteq \mathsf{E}_{\mathsf{QIR}} \times \mathsf{E}_{\mathsf{QIR}}$ is defined as:

$$\frac{\vdash_{\texttt{NF}} v_2}{(\mathbf{fun}^f(x)\to q_1)\ v_2 \hookrightarrow \{f \mapsto \mathbf{fun}^f(x)\to q_1, x \mapsto v_2\}q_1}\ \text{(norm-app-}\beta\text{)}\ v_2 \text{ pure}$$

$$\frac{q \hookrightarrow q'}{\mathbf{fun}^f(x)\to q \hookrightarrow \mathbf{fun}^f(x)\to q'}\ \text{(norm-fun-red)} \qquad \frac{q \to q'}{q \hookrightarrow q'}\ \text{(norm-qred)}\ \ q \not\equiv (\mathbf{fun}^f(x)\to q_1)\ v_2,\ v_2 \text{ pure normal form}$$

where the rules are prioritized following their order of definition.

As required for the reasons stated earlier, $\hookrightarrow$ only applies a β-reduction if its argument is not a host language expression. This is done by replacing the (app-β) rule with a (norm-app-β) rule. Additionally, it includes a rule (norm-fun-red) that allow us to apply reductions in function bodies.

Our normalization relation has the property of *progress*.

Lemma 3.1. *Let $q \in \mathsf{E}_{QIR}$. Either q is in normal form, or $\exists q'.q \hookrightarrow q'$.*

Proof. By case analysis on q:

- If $q = x$, then q is in normal form.
- If $q = \mathbf{fun}^x(x)\to q_1$, then either
  - q_1 is in normal form, in which case q is in normal form;
  - or $q_1 \hookrightarrow q_1'$, in which case rule (norm-fun-red) applies.
- If $q = q_1\ q_2$, then either
  - $q_1 \equiv \mathbf{fun}^x(x)\to q_3$ and q_2 is in normal form and pure, in which case rule (norm-app-β) applies;
  - $q_1 \equiv \mathbf{fun}^x(x)\to v$ and q_2 are in normal form and q_2 is not pure, in which case q is a normal form;

- $q_1 \equiv op$ and q_2 are in normal form, in which case rule (app-op) applies;
- or $q_1 \not\equiv \mathbf{fun}^x(x) \rightarrow v$ or op and q_2 are in normal form, in which case q is in normal form;
- or either q_1 or q_2 is not in normal form, in which case rules (app-red1) or (app-red2) apply.

The complete proof can be found in Appendix B, page 187. □

As usual, since our normalization relation is deterministic, it immediately follows that the normal form of a QIR expression is unique when it exists. However, a normal form does not necessarily exist, with the classic example of $(\mathbf{fun}(x) \rightarrow x\ x)\ (\mathbf{fun}(x) \rightarrow x\ x)$.

Now that we have a reduction relation for the normalization, we next see how to apply it in a way that guarantees termination.

3.5.3 A measure for good queries

To use our $\hookrightarrow$ relation in a way that ensures termination, we define a measure that indicates how much a query is translatable to database languages. This measure allows us to guide the normalization by verifying our reduction steps are actually useful in making the query more translatable.

To know if a query is better suited for translation than another, we count data operators of an expression that can be translated into a database language.

Definition 3.14 (Compatible data operator application)**.** Let $\mathcal{D}$ be a database language. A QIR data operator application $o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m \rangle$ is *compatible with* $\mathcal{D}$ if $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}(o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m \rangle) \neq \Omega$.

For instance, recall the query from Example 3.7:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow (\mathbf{fun}(x) \rightarrow r \cdot \mathit{salary} > x)\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

This query uses an application of the operator `Filter` that is not compatible with the database $\mathcal{D}$ if it cannot translate the application of an anonymous function. In this case, reducing the application in the normalization and therefore making the `Filter` translatable, would reduce the number of incompatible data operator applications in the query, which is a measurable quantity showing that the reduction is beneficial.

Definition 3.15 (Measure)**.** Let $q \in \mathsf{E}_{\mathsf{QIR}}$ be a QIR expression, we define the *measure of q by the database $\mathcal{D}$* as

$$M_{\mathcal{D}}(q) = |\mathsf{Op}(q) \setminus \mathsf{Comp}_{\mathcal{D}}(q)|$$

where $\mathsf{Op}(q)$ is the set of data operator applications in q and $\mathsf{Comp}_{\mathcal{D}}(q)$ is the set of data operator applications in q that are compatible with $\mathcal{D}$.

This measure works as follows. During a step of reduction of a term q into a term q', q' is considered a better term if the number of incompatible data operator applications strictly decreases. We now go through a few examples.

Consider the query q from Example 3.7:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow (\mathbf{fun}(x) \rightarrow r \cdot salary > x)\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

that can reduce to q':

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot salary > 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

Depending on $\mathcal{D}$, we have three possible cases:

1. $\mathcal{D}$ does not support `Filter`, in which case reducing is useless, and indeed we have $M_{\mathcal{D}}(q) = 2 - 1 = 1$ and $M_{\mathcal{D}}(q') = 2 - 1 = 1$
2. $\mathcal{D}$ supports `Filter`, but does not support the configuration of `Filter` (e.g. it does not support the creation and/or application of an anonymous user-defined function), in which case reducing is beneficial, and indeed we have $M_{\mathcal{D}}(q) = 2 - 1 = 1$ and $M_{\mathcal{D}}(q') = 2 - 2 = 0$
3. $\mathcal{D}$ supports both `Filter` and its configuration, in which case reducing is useless, and indeed we have $M_{\mathcal{D}}(q) = 2 - 2 = 0$ and $M_{\mathcal{D}}(q') = 2 - 2 = 0$

Recall the query q from the beginning of Section 3.3:

$$\begin{array}{l}
(\mathbf{fun}(emp, minSalary, getRate, cur) \rightarrow \\
\quad \texttt{Project}\langle \mathbf{fun}(x) \rightarrow \{name : x \cdot name\} \mid \\
\quad \texttt{Filter}\langle \mathbf{fun}(x) \rightarrow x \cdot salary \geq minSalary * (getRate\ \texttt{"USD"}\ cur) \mid \\
\quad emp \rangle \rangle) \\
(\texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle, 2500, \mathbf{fun}(rfrom, rto) \rightarrow \mathrm{BODY}, \texttt{"USD"})
\end{array}$$

$$\begin{array}{l}
\mathrm{BODY} = \\
(\mathbf{fun}(change) \rightarrow \mathbf{if}\ rfrom = rto\ \mathbf{then}\ 1\ \mathbf{else} \\
\quad \texttt{Project}\langle \mathbf{fun}(x) \rightarrow \{rate : x \cdot rate\} \mid \\
\quad \texttt{Filter}\langle \mathbf{fun}(x) \rightarrow x \cdot cfrom = rfrom\ \&\&\ x \cdot cto = rto \mid \\
\quad change \rangle \rangle) \\
(\texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"change"} \rangle)
\end{array}$$

that is the translation of our R query from Example 1.5:

```
16  richUSPeople = atLeast(2500, "USD")
```

for which $M_{\mathcal{D}}(q) = 6 - 0 = 6$ counting the query in `getRate` because even the `From`s contain free variables.

After reductions we get q':

$$\texttt{Project}\langle\textbf{fun}(x)\rightarrow\{name : x \cdot name\} \mid \texttt{Filter}\langle\textbf{fun}(x)\rightarrow x \cdot salary \geq 2500 \mid \texttt{From}\langle\text{PostgreSQL}, \texttt{"employee"}\rangle\rangle\rangle$$

for which $M_{\mathcal{D}}(q') = 3 - 3 = 0$. The operators of the main query are now compatible with PostgreSQL and the reduction of the conditional expression in the function `getRate` has removed the subquery.

Similarly, our other R query from Example 1.5:

```
17  richEURPeople = atLeast(2500, "EUR")
```

is translated to:

$$\begin{aligned}
&(\textbf{fun}(emp, minSalary, getRate, cur)\rightarrow \\
&\quad \texttt{Project}\langle\textbf{fun}(x)\rightarrow\{name : x \cdot name\} \mid \\
&\quad \texttt{Filter}\langle\textbf{fun}(x)\rightarrow x \cdot salary \geq minSalary * (getRate\ \texttt{"USD"}\ cur) \mid \\
&\quad emp\rangle\rangle) \\
&(\texttt{From}\langle\text{PostgreSQL}, \texttt{"employee"}\rangle, 2500, \textbf{fun}(rfrom, rto)\rightarrow \text{BODY}, \texttt{"EUR"})
\end{aligned}$$

$$\begin{aligned}
&\text{BODY} = \\
&(\textbf{fun}(change)\rightarrow \textbf{if}\ rfrom = rto\ \textbf{then}\ 1\ \textbf{else} \\
&\quad \texttt{Project}\langle\textbf{fun}(x)\rightarrow\{rate : x \cdot rate\} \mid \\
&\quad \texttt{Filter}\langle\textbf{fun}(x)\rightarrow x \cdot cfrom = rfrom\ \&\&\ x \cdot cto = rto \mid \\
&\quad change\rangle\rangle) \\
&(\texttt{From}\langle\text{PostgreSQL}, \texttt{"change"}\rangle)
\end{aligned}$$

for which also $M_{\mathcal{D}}(q) = 6 - 0 = 6$. And the reduction q':

$$\begin{aligned}
&\texttt{Project}\langle\textbf{fun}(x)\rightarrow\{name : x \cdot name\} \mid \\
&\quad \texttt{Filter}\langle\textbf{fun}(x)\rightarrow x \cdot salary \geq 2500 * \\
&\quad\quad (\texttt{Project}\langle\textbf{fun}(x)\rightarrow\{rate : x \cdot rate\} \mid \\
&\quad\quad\ \texttt{Filter}\langle\textbf{fun}(x)\rightarrow x \cdot cfrom = \texttt{"USD"}\ \&\&\ x \cdot cto = \texttt{"EUR"} \mid \\
&\quad\quad\ \texttt{From}\langle\text{PostgreSQL}, \texttt{"change"}\rangle\rangle\rangle) \mid \\
&\quad \texttt{From}\langle\text{PostgreSQL}, \texttt{"employee"}\rangle\rangle\rangle
\end{aligned}$$

for which $M_{\mathcal{D}}(q') = 6 - 6 = 0$.

Consider now the query q from Example 3.12.

Example 3.12.

$$\texttt{Project}\langle \textbf{fun}(r) \rightarrow (\textbf{fun}(x) \rightarrow \{result : x\})\ (r \cdot id) \mid \\ \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

Suppose that $\mathcal{D}$ supports all three operators but not anonymous functions, then $M_{\mathcal{D}}(q) = 3 - 2 = 1$ since only `Project` is not compatible. However, if $\mathcal{D}$ does not support `Filter`, then $M_{\mathcal{D}}(q) = 3 - 1 = 2$. After reduction, we get q':

$$\texttt{Project}\langle \textbf{fun}(r) \rightarrow \{result : r \cdot id\} \mid \\ \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

for which $M_{\mathcal{D}}(q') = 3 - 3 = 0$ if `Filter` is supported, making the reduction useful, and $M_{\mathcal{D}}(q') = 3 - 1 = 2$ otherwise, making the reduction useless. Indeed, making the configuration of `Project` compatible does not help here, since the `Filter` and thus the `Project` would still be untranslatable.

We have yet to define how the normalization knows which database measure to choose for the normalization of a QIR term, especially in the case where multiple databases are referenced in the QIR term.

3.5.4 Generic measure

The measure we defined in Definition 3.15 is relative to a database $\mathcal{D}$. Therefore, using it on a simple query that targets only one database $\mathcal{D}$ and is completely translatable into the language of the database is straightforward as the measure to use is then obviously $M_{\mathcal{D}}()$. In other cases however, we can have several different databases executing different parts of the query, and in that case we want to make the different measures cooperate with one another.

Definition 3.16 (Generic measure)**.** Let $q \in \mathsf{E}_{\mathsf{QIR}}$, we define the *generic measure of* q as:

$$\begin{array}{lcll} M(q) & = & M_{\mathcal{D}}(q) & \text{if } \mathfrak{T}(q) = \{\mathcal{D}\}, \mathcal{D} \neq \mathsf{MEM} \\ M(q) & = & 1 + \sum_{q_i \in \mathfrak{C}(q)} M(q_i) & \text{if } q \equiv o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n \rangle \\ M(q) & = & \sum_{q_i \in \mathfrak{C}(q)} M(q_i) & \text{otherwise} \end{array}$$

Thus, the generic measure of an expression q is the number of data operator applications that are not compatible with a database different of MEM.

For instance, on our query q from Subsection 3.5.3:

$$(\textbf{fun}(a) \rightarrow \texttt{Join}\langle \textbf{fun}(x, y) \rightarrow x \bowtie y, \textbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid a, a \rangle)$$
$$(\texttt{From}\langle \mathcal{D}, \texttt{"people"} \rangle)$$

we get $M(q) = 1$ since the `Join` has no targeted database. As for its reduction q':

$$\texttt{Join}\langle \textbf{fun}(x, y) \rightarrow x \bowtie y, \textbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid$$
$$\texttt{From}\langle \mathcal{D}, \texttt{"people"} \rangle, \texttt{From}\langle \mathcal{D}, \texttt{"people"} \rangle \rangle$$

we would get $M(q') = M_{\mathcal{D}}(q') = 0$ if `Join` is supported by $\mathcal{D}$, which signifies the reduction is useful as expected, or if `Join` is not supported, then $M(q') = M_{\mathcal{D}}(q') = 1$ which indicates that the reduction is not useful as expected.

Recall the query q of Example 3.12:

$$\texttt{Project}\langle \textbf{fun}(r) \rightarrow (\textbf{fun}(x) \rightarrow \{result : x\})\ (r \cdot id) \mid$$
$$\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

Suppose that $\mathcal{D}$ supports all three operators but not anonymous functions, then $M(q) = M_{\mathcal{D}}(q) = 1$ since only `Project` cannot be translated. However, if $\mathcal{D}$ does not support `Filter`, then $M(q) = M_{\mathcal{D}}(q) = 2$. After reduction, we get q':

$$\texttt{Project}\langle \textbf{fun}(r) \rightarrow \{result : r \cdot id\} \mid$$
$$\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle$$

for which $M(q') = M_{\mathcal{D}}(q') = 0$ if `Filter` is supported, making the reduction useful, and $M(q') = M_{\mathcal{D}}(q') = 2$ otherwise, making the reduction useless. Indeed, making the configuration of `Project` compatible does not help here, since the `Filter` and thus the `Project` would still not be compatible.

As a last example, consider the following query:

$$\texttt{Project}\langle \textbf{fun}(x) \rightarrow \{r : A\} \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

which applies a `Project` on table `employee` where A is the following QIR expression:

$$A = (\textbf{fun}(y) \rightarrow \texttt{Filter}\langle \textbf{fun}(z) \rightarrow z \cdot teamid = y \mid \texttt{From}\langle \mathcal{D}, \texttt{"team"} \rangle \rangle)\ (x \cdot teamid)$$

If we assume that the database can handle every operator, but cannot translate the application of an anonymous function, then the `Project` would be executed in MEM since it could not be translated to the language of the database, which involves, in this case, the translation then evaluation in the database $\mathcal{D}$ of the `Filter` for every row of table `employee`. In other words, if table `employee` contains a million rows, we would send a million queries to the database. This problem is known in the literature as *query avalanche* [GRS10].

However, this problem disappears if we are able to merge our subqueries together. After a step of reduction, the query would become:

$$\texttt{Project}\langle \textbf{fun}\,(x) \rightarrow \{r : \texttt{Filter}\langle \textbf{fun}\,(z) \rightarrow z \cdot teamid = x \cdot teamid \mid \texttt{From}\langle \mathcal{D}, \texttt{"team"} \rangle\rangle\} \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle\rangle$$

which can be completely translated to the language of $\mathcal{D}$ since the application disappeared.

The good news is that our measure finds the normalization useful. We get $M(q) = M_{\mathcal{D}}(q) = 4 - 3 = 1$ for the query before reduction since only `Project` is not compatible, and $M(q') = M_{\mathcal{D}}(q') = 4 - 4 = 0$ for the query after reduction, thus marking the reduction as indeed useful.

Therefore, in some cases, the normalization allows us to evaluate queries very efficiently by avoiding query avalanches altogether. Obviously, the normalization would not be able to save us in every case, in particular if the two `From` target different databases, or if the database does not support `Project`.

3.5.5 Heuristic-based normalization

A reduction might not have a positive impact right away on the translation, but lead to a better query after more steps of reduction. As mentioned before, we cannot just explore indefinitely every path of reduction, as this process might not terminate. Therefore, to generate a more efficient translation while ensuring termination, we create a heuristic-based normalization procedure which uses the generic measure of Definition 3.16 as a guide through the reduction of a QIR term.

To define our heuristic, we first define the set of possible reductions of a QIR expressions:

Definition 3.17 (Set of possible reductions of a QIR expression)**.** The *set of possible reductions of a QIR expression* q, noted $\mathsf{Reds}\,(q)$, is defined as the set of expressions q' such as $q \hookrightarrow q'$.

Figure 3.2 describes our heuristic-based normalization in pseudo-code. It applies all possible combinations of reduction steps to the term as long as its measure decreases after a number of steps, called *fuel* (ϕ), fixed by heuristic. This normalization always terminates, either because it has applied a sequence of reductions to the QIR term and reached a normal form, or because it has run out of fuel in every possible reduction path.

For instance, our query from Example 3.7:

$$\texttt{Filter}\langle \textbf{fun}\,(r) \rightarrow (\textbf{fun}\,(x) \rightarrow r \cdot salary > x)\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle\rangle$$

is reduced to a normal form for the normalization in only one step.

```
function hnorm(q, φ) {
  if φ = 0 then return error else {
    for each q' in Reds(q) do {
      if M(q') < M(q) then return hnorm(q', φ_max) else {
        q'' <- hnorm(q', φ − 1)
        if q'' ≠ error then return q''
      }
    }
  }
  return q
}
```

Figure 3.2 – Heuristic-based normalization

Now, if we consider a query for which the reduction does not terminate such as:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow (\mathbf{fun}^f(x) \rightarrow f\ x)\ (2500) \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

The heuristic-based normalization would attempt a finite number of times (the value of its fuel) to reduce the application in the configuration with no improvement on the measure $M_{\mathcal{D}}()$, therefore the normalization terminates and returns the query after no reduction step.

Some practical choices impact the effectiveness of the heuristic such as choosing which reduction rule to apply at each step (e.g., choosing those with more arguments), or which maximum number of steps to use. Experiments for both points can be found in [Ver16], where a similar measure of good QIR terms as our measure from Definition 3.15 is defined to create a heuristic-based normalization, and where it is showed that the normalization represents a negligible fraction of the execution time of the whole process compared to tasks such as parsing, or exchanges on the network with databases. However, in that work, queries are limited to one targeted database, and the definition of a compatible data operator application is based on syntactic considerations on the name and the shape of the configurations of the data operator application.

There is one case where our measure makes the normalization apply reductions on fully translatable queries: because it is based on data operators, if an expression does not have a data operator at its root, the measure may consider a reduction useful even though it is unnecessary in the viewpoint of the translation to database languages.

For instance, take the last query q from Section 3.5.1:

$$(\mathbf{fun}(a) \rightarrow \texttt{Join}\langle \mathbf{fun}(x, y) \rightarrow x \bowtie y, \mathbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid a, a \rangle)\ (\texttt{From}\langle \mathcal{D}, \texttt{"people"} \rangle)$$

and its reduction q':

$$\texttt{Join}\langle \textbf{fun}(x, y) \rightarrow x \bowtie y, \textbf{fun}(x, y) \rightarrow x \cdot age < y \cdot age \mid \texttt{From}\langle \mathcal{D}, \texttt{"people"} \rangle, \texttt{From}\langle \mathcal{D}, \texttt{"people"} \rangle \rangle$$

$M(q) = M_{\mathcal{D}}(q) = 2 - 1 = 1$. Indeed, there are two operators `Join` and `From`, and only `From` is compatible. As for $M_{\mathcal{D}}(q')$, if `Join` is compatible, then $M(q') = M_{\mathcal{D}}(q') = 3 - 3 = 0$ thus making the reduction useful, but if `Join` is not compatible, then $M(q') = M_{\mathcal{D}}(q') = 3 - 2 = 1$ thus making the reduction useless. This result does not depend on whether or not the database $\mathcal{D}$ can translate the application. Indeed, if the database supports the application and `Join`, then the entire query can be translated into the language of $\mathcal{D}$ without the help of the normalization. However, the measure considers the reduction useful if the database supports `Join` with no consideration for the application. Thankfully, this specific case of false positive is not an issue in practice: even though some databases may allow the creation of user-defined functions in their language, they are much more efficient at handling UDF-free queries as we show in our results in Chapter 7.

Additionally, our definition of compatible data operator application calls the translation of expressions defined in the driver of the database on every operator, and to repeat this after every reduction step could become costly. But in practice, using a cache mechanism makes these translations mostly trivial as our experiments and those of Vernoux [Ver16] confirm. We show in Section 6.4 that we can avoid these calls to translations altogether under some conditions.

Chapter 4

QIR type system

Creating a type system for the QIR would be straightforward if it was not for data operators and host language expressions since all of our other constructions are well-known and type systems have already been designed for languages that includes them [Chu40, Bar92, Oho95]. Host language expressions can contain any type of expression, including any kind of side effects, which makes them difficult to classify into types. Although it would be possible to type some class of host language expressions that contain only some types of side effects [Wad95, NN99], we do not type host language expressions in this thesis. As for data operators, as explained in Chapter 3, we made the choice to let the databases provide their own operators. In other words, the behavior of a data operator depends on the database executing it. Therefore, to create a type system for QIR, we have to give a QIR expression a type that makes sense *for a database*. To achieve this, we define *specific* type systems for databases which give a type to QIR expressions that are compatible with the corresponding database, and a *generic* type system which uses the specific type systems to type as much as possible of QIR expressions for databases (other than MEM). This design also allows for extensions to new databases by integrating new specific type systems to the generic type system.

Which brings us to our most important reason to design a type system for QIR, it tells us which query languages each subexpression should be translated into: if an expression can be typed using a specific type system, then it can be translated into the query language of the corresponding database. This property is very useful to us since it gives BOLDR a way to know which database should take care of which parts of the query. In particular, we put this information to use in Chapter 6 for the translation from QIR into query languages.

Our second reason is to detect errors in QIR expressions before their evaluation which is usually the main motivation for a type system in a programming language. In our case, this early detection is very interesting since the evaluation of a query comes with, in the worst-case scenario, the costs of optimizing, translating, sending the different subqueries to the databases, waiting for the different

subqueries to complete, translating and sending back the results to QIR, and finally realize that there is an error at the execution of the remains of the query in MEM. Detecting errors before the translation of queries into QIR allows us to inform the programmer of the error immediately, even if the query originates from a dynamically typed programming language.

A third reason is that it allows us to prove interesting properties on our evaluation of queries. In particular, we show in this chapter that the normalization of Section 3.5 preserves the type of the QIR expression it is applied to, and that its reduction always terminates on well-typed expressions that do not contain recursive functions. Additionally, we discuss the possibility of detecting where the normalization is guaranteed to be useful in Section 6.4.

In this chapter, we define a *generic* type system for QIR which gives a type to any QIR expression. To achieve this, we also define *specific* type systems for databases which give a type to QIR expressions that are compatible with the corresponding database. We then define specific type systems for MEM and SQL, and deduce useful properties for typeable QIR expressions.

4.1 QIR types

First, we define types for all the constructs of our QIR.

Definition 4.1 (Basic QIR types)**.** A *basic QIR type* is a type B that represents basic data constructs: `bool`, `int`, `string`, ... We will note $\mathcal{B}$ the set of basic QIR types.

Definition 4.2 (QIR types)**.** A *QIR type* is a finite term of the following grammar:

$$\begin{array}{rcl} T & ::= & B \\ & | & T \to T \\ & | & T \ \textbf{list} \\ & | & \{l : T, \ldots, l : T\} \end{array}$$

Definition 4.3 (Domain of a record type)**.** The *domain of a record type* $R = \{l_1 : T_1, \ldots, l_n : T_n\}$ noted `dom(`R`)` is the set of its labels $\{l_1, \ldots, l_n\}$.

QIR expressions either have a basic type such as `bool` or `int`, a $\to$ type that represents function types, a list type, or a record type. For instance, the function $\textbf{fun}(x) \to not\ x$ that takes a boolean and returns its negation should be given the type `bool` $\to$ `bool`, and the type $\{id : int\}$ **list** is the type of QIR expressions

that represent lists of records containing *exactly* one field named *id* associated to an expression of type `int`.

Notation 4.1. We use the following syntactic shortcuts:

- $\{l_i : T_i\}_{i=1..n}$ stands for $\{l_1 : T_1, \ldots, l_n : T_n\}$
- $T_1 \to T_2 \to T_3$ stands for $T_1 \to (T_2 \to T_3)$

Classically, $\to$ is right-associative which allows us, as explained in Section 3.1, to talk about functions with multiple arguments easily. For instance, $\texttt{int} \to \texttt{int} \to \texttt{int}$ is equivalent to $\texttt{int} \to (\texttt{int} \to \texttt{int})$.

A very important feature of databases is flexible operations on records. When a data operator is applied, it allows data to contain more information than needed. For instance, consider this query in SQL:

```
SELECT e.name FROM employee
```

This query applies the `Project` operator on a table `employee` and for each row returns a row containing only the `name`. However, a row from the table `employee` may contain more than just the column `name`. We want to create a type system that reflects this feature.

For instance, a record $\{\, id : 1, name : \texttt{"Maggie"} \,\}$ should be given the type $\{id : \texttt{int}, name : \texttt{string}\}$, however we want to be able to apply this record to $\texttt{Project}\langle \textbf{fun}(r) \to \{id : r \cdot id\} \mid \ldots \rangle$ which configuration could have the type $\{id : \texttt{int}, name : \texttt{string}\} \to \texttt{int}$, but its most natural type would be $\{id : \texttt{int}\} \to \texttt{int}$. Therefore, our type systems have to be able to talk about the relation between these record types. One solution to this problem is to extend our record types to polymorphic record types [Wan87, Rém89]. This type of solution would give the expression $\textbf{fun}(r) \to r \cdot id$ the type $\{id : \texttt{int}, \rho\}$, where ρ is a *row variable* that represents the fact that the record type is extensible to more labels and can be instantiated to the type $\{id : \texttt{int}\}$, or $\{id : \texttt{int}, name : \texttt{string}\}$, or any other record type that contains at least the label *id* associated to the type `int`. However, this solution complexifies the type system substantially. Instead, we apply another solution which is to define a *subtyping* relation between our types [Car84, Car88].

Definition 4.4 (Subtyping relation)**.** The *subtyping relation* for QIR, noted $\preceq$, is defined as:

- $B_1 \preceq B_2$ iff $B_1 = B_2$
- $T_1 \to T_2 \preceq T_3 \to T_4$ iff $T_3 \preceq T_1$ and $T_2 \preceq T_4$
- T_1 **list** $\preceq T_2$ **list** iff $T_1 \preceq T_2$
- $\{l_i : T_i\}_{i=1..n} \preceq \{l'_j : T'_j\}_{j=1..m}$ iff for all $j \in 1..m$ there exists $i \in 1..n$ such that $l_i = l'_j$ and $T_i \preceq T'_j$

We say that a type T_1 is a *subtype* of a type T_2 if $T_1 \preceq T_2$, and that it is a *supertype* of a type T_3 if $T_3 \preceq T_1$.

Definition 4.5 (Strict subtype)**.** A QIR type T_1 is a *strict subtype* of a QIR type T_2, noted $T_1 \prec T_2$, if and only if $T_1 \preceq T_2$ and $T_1 \neq T_2$.

Our definition of the subtyping relation is standard. It is covariant in list types and in the output type of function types, and contravariant in the input type of function types.

A record type R_1 is a subtype of another record type R_2 if every label of R_2 is present in R_1, and if types associated to those labels in R_1 are themselves subtypes of the ones in R_2. For example, we have $\{id : \texttt{int}, name : \texttt{string}\} \preceq \{id : \texttt{int}\}$, and $\{id : \texttt{int}, rest : \{name : \texttt{string}\}\} \preceq \{id : \texttt{int}, rest : \{\}\}$. The idea is that we want to be able to give records their natural type, but also all of its supertypes. For instance, we want to be able to give our example $\{\, id\texttt{:}1, name\texttt{:"Maggie"} \,\}$ the type $\{id : \texttt{int}, name : \texttt{string}\}$, but also $\{id : \texttt{int}\}$, and even $\{\}$. This idea, that comes from object-oriented programming, reflects the intuition that if some expression is expected to be of a certain type, then any subtype should work as well. Going back to our example $\textbf{fun}(r) \to r \cdot id$, we can now give this function the type $\{id : int\} \to int$, and use the subtyping relation on the argument to make the application correct.

As usual, the subtyping relation is reflexive and transitive, properties that will be useful later.

Property 4.1 (Reflexivity of the subtyping relation)**.** $T \preceq T$.

Proof. By induction on the structure of T. □

Property 4.2 (Transitivity of the subtyping relation)**.** *If* $T_1 \preceq T_2$ *and* $T_2 \preceq T_3$ *then* $T_1 \preceq T_3$*.*

Proof. By induction on the structure of T_1:

- If $T_1 \in \mathcal{B}$ then $T_1 = T_2$ and $T_2 = T_3$.
- If $T_1 = T_1' \to T_1''$ then $T_2 = T_2' \to T_2''$ and $T_3 = T_3' \to T_3''$, and $T_2' \preceq T_1'$, $T_1'' \preceq T_2''$, $T_3' \preceq T_2'$, $T_2'' \preceq T_3''$, so by induction hypothesis $T_3' \preceq T_1'$ and $T_1'' \preceq T_3''$ which gives us $T_1' \to T_1'' \preceq T_3' \to T_3''$.
- If $T_1 = T_1'$ **list** then $T_2 = T_2'$ **list** and $T_3 = T_3'$ **list**, and $T_1' \preceq T_2'$ and $T_2' \preceq T_3'$, so by induction hypothesis $T_1' \preceq T_3'$ which gives us T_1' **list** $\preceq T_3'$ **list**.
- If $T_1 = \{l_i : T_i\}_{i=1..n}$ then $T_2 = \{l_j' : T_j'\}_{j=1..m}$ and $T_3 = \{l_k'' : T_k''\}_{k=1..l}$, and for all $k \in 1..l$ there exists $j \in 1..m$ such that $l_j' = l_k''$ and $T_j' \preceq T_k''$, and for all $j \in 1..m$ there exists $i \in 1..n$ such that $l_i = l_j'$ and $T_i \preceq T_j'$. Therefore, by induction hypothesis, for all $k \in 1..l$ there exists $i \in 1..n$ such that $l_i = l_k''$ and $T_i \preceq T_k''$.

□

This completes our definition of QIR types. In the next section, we define our QIR type systems.

4.2 QIR type systems

As explained earlier, BOLDR supports queries that target several databases at once, and allows databases to propose their own data operators with their semantics. Additionally, BOLDR has to be seamlessly extendable to new databases. For these reasons, we require databases to define a type system that gives a type to QIR expressions they support. This gives BOLDR information on which expressions can be translated into the query language of a database, and in particular which data operators are supported by the database. We call these type systems *specific type systems*.

Definition 4.6 (Specific type system)**.** A specific type system of a database $\mathcal{D}$ denoted by the judgment $\Gamma \vdash_{\mathcal{D}} q : T$ is a type system relation between a QIR typing environment Γ, a QIR term q, and a QIR type T.

Now that every database interfaced with BOLDR provides its type system for QIR, we define a global type system that makes use of specific type systems. Its

goal is to type as much of the QIR expressions as possible using the specific type systems of databases other than MEM to achieve our goal explained in Chapter 1 to execute as much of the queries in databases. We call this global type system our *generic QIR type system*.

Definition 4.7 (Generic QIR type system)**.** A QIR term q has a type T for the database $\mathcal{D}$ derivable from a QIR type environment Γ in the generic QIR type system, noted $\Gamma \vdash q : T, \mathcal{D}$. The set of inference rules used to derive this judgment is:

$$\frac{\forall q_i \in \mathfrak{C}(q).\Gamma \vdash q_i : T_i, \mathcal{D}_i \quad \Gamma \vdash_{\mathcal{D}} q : T}{\Gamma \vdash q : T, \mathcal{D}} \begin{array}{l}\{\mathcal{D}_i\} = \{\mathcal{D}, \mathsf{MEM}\} \\ \mathcal{D} \neq \mathsf{MEM}\end{array}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} \texttt{From}\langle \mathcal{D}, \_\rangle : T}{\Gamma \vdash \texttt{From}\langle \mathcal{D}, \_\rangle : T, \mathcal{D}}\ \mathcal{D}\neq\mathsf{MEM} \qquad \frac{\Gamma \vdash_{\mathsf{MEM}} q : T}{\Gamma \vdash q : T, \mathsf{MEM}}$$

We make two important design choices in our generic type system. First, the only case in which we initiate the process of calling the specific type systems is if we encounter a `From` operator. The rationale for this is that only this operator designates a database as the only possible target of the query.

Our second design choice can be seen in the first rule of the generic type system: we call the specific type system of a database $\mathcal{D} \neq \mathsf{MEM}$ on an expression *only if* its children have a type for $\mathcal{D}$ or MEM. Indeed, in that case, the only two databases that could be the target database for the expression are $\mathcal{D}$ or MEM. The default case, if the expression could not be entirely typed by the specific type system of $\mathcal{D}$, represented by the last rule, attempts to type the expression using the specific type system for MEM that we describe later in this section. The reason why we do not require all the children to have a type for $\mathcal{D}$ is that, because of our first design choice explained earlier, a subexpression typed for MEM does not necessarily imply that the whole expression cannot be typed for other databases. In fact, most configurations of data operators are typed for MEM. To illustrate this, recall Example 3.3 from Chapter 3:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle\rangle$$

Using the first rule of our generic type system, the configuration $\mathbf{fun}(r) \rightarrow r \cdot id < 2500$ is typed for MEM, and the data argument $\texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle$ is typed for $\mathcal{D}$. Finally, if the database supports `Filter`, the specific type system of $\mathcal{D}$ can give a type to the entire expression, and thus the generic type system can type the expression for $\mathcal{D}$ as desired.

Our generic type system can type queries targeting multiple databases. For

instance, recall the query of Example 3.10:

$$\texttt{Join}\langle \textbf{fun}(x,y) \rightarrow x \bowtie y, \textbf{fun}(x,y) \rightarrow \texttt{true} \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathcal{D}', \texttt{"team"}\rangle\rangle$$

where $\mathcal{D}$ and $\mathcal{D}'$ are two distinct databases that are not MEM. The generic type system then types the two different data arguments for $\mathcal{D}$ and $\mathcal{D}'$, which means the only applicable rule is then the last one which types the `Join` for MEM.

to complete the generic type system, we define a specific type system for MEM.

Definition 4.8 (Specific MEM type system)**.** The specific type system of MEM noted $\vdash_{\text{MEM}}$ is derived by the rules of Figure 4.1.

The specific type system of MEM is very broad, as all QIR expressions except data operators can be evaluated in MEM. As seen in Section 3.4, the data operators supported by MEM are `Project`, `Filter`, and `Join`.

We assume the existence of a function `typeofC` which gives the type of a constant c. We also assume the existence of a function `typeofOP` which returns all the possible types for a basic operator. For instance, $\texttt{typeofOP}(=) = \{\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{bool}, \texttt{string} \rightarrow \texttt{string} \rightarrow \texttt{bool}, \ldots\}$.

The record concatenation is typed successfully only if applied to two records which labels are strictly distinct, following the semantics of QIR. Thus, $\{\texttt{x} : 2, \texttt{y} : 3\} \bowtie \{\texttt{x} : \texttt{true}\}$ is an invalid expression in QIR. This is the only way to define a usable record concatenation with no loss of information which respects the label names. There are other ways to define record concatenation, for instance *asymmetric* record concatenation usually preferred by general-purpose programming languages, which keeps the values of the second record in case of conflict: $\{\texttt{x} : 2, \texttt{y} : 3\} \bowtie \{\texttt{x} : \texttt{true}\} = \{\texttt{x} : \texttt{true}, \texttt{y} : 3\}$. This type of concatenation has the virtue of being more flexible for programming, but the loss of information that ensues makes typing problematic when combined with subtyping: $\{\texttt{x} : \texttt{true}\}$ can be given the type $\{\texttt{x} : \texttt{bool}\}$, in which case $\{\texttt{x} : 2, \texttt{y} : 3\} \bowtie \{\texttt{x} : \texttt{true}\}$ is given the type $\{\texttt{x} : \texttt{bool}, \texttt{y} : \texttt{int}\}$, but $\{\texttt{x} : \texttt{true}\}$ can also be given the type $\{\}$, in which case $\{\texttt{x} : 2, \texttt{y} : 3\} \bowtie \{\texttt{x} : \texttt{true}\}$ is given the erroneous type $\{\texttt{x} : \texttt{int}, \texttt{y} : \texttt{int}\}$. Another type of record concatenation provided by SQL keeps all the information in a record that may contain multi-value labels: $\{\texttt{x} : 2, \texttt{y} : 3\} \bowtie \{\texttt{x} : \texttt{true}\} = \{\texttt{x} : 2, \texttt{y} : 3, \texttt{x} : \texttt{true}\}$, but the information then becomes impossible to access since there is no way to know which value to return on an access to the label `x`. Databases such as *MySQL* automatically rename the conflicting labels so the information can be accessed: $\{\texttt{x} : 2, \texttt{y} : 3\} \bowtie \{\texttt{x} : \texttt{true}\} = \{\texttt{x}_1 : 2, \texttt{y} : 3, \texttt{x}_2 : \texttt{true}\}$, but the query then has to be aware of that specific renaming process.

Crucially, the specific type system of MEM always calls the *generic* type system on the children in the premises of its rules, where _ denotes any database in

$$\frac{}{\Gamma, x : T \vdash_{\mathsf{MEM}} x : T} \qquad \frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash q : T_2, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \to q : T_1 \to T_2} \qquad \frac{\Gamma \vdash q_1 : T_1 \to T_2, \_ \quad \Gamma \vdash q_2 : T_1, \_}{\Gamma \vdash_{\mathsf{MEM}} q_1\ q_2 : T_2}$$

$$\frac{}{\Gamma \vdash_{\mathsf{MEM}} c : \mathtt{typeofC}(c)} \qquad \frac{T \in \mathtt{typeofOP}(op)}{\Gamma \vdash_{\mathsf{MEM}} op : T} \qquad \frac{\Gamma \vdash q_1 : \mathtt{bool}, \_ \quad \Gamma \vdash q_2 : T, \_ \quad \Gamma \vdash q_3 : T, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : T}$$

$$\frac{\Gamma \vdash q_i : T_i, \_ \quad i \in 1..n}{\Gamma \vdash_{\mathsf{MEM}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}} \qquad \frac{\Gamma \vdash q_1 : \{l_i : T_i\}_{i=1..m}, \_ \quad \Gamma \vdash q_2 : \{l_i : T_i\}_{i=m+1..n}, \_}{\Gamma \vdash_{\mathsf{MEM}} q_1 \bowtie q_2 : \{l_i : T_i\}_{i=1..n}}$$

$$\frac{}{\Gamma \vdash_{\mathsf{MEM}} [\,] : T\ \mathbf{list}} \qquad \frac{\Gamma \vdash q_1 : T, \_ \quad \Gamma \vdash q_2 : T\ \mathbf{list}, \_}{\Gamma \vdash_{\mathsf{MEM}} q_1 :: q_2 : T\ \mathbf{list}} \qquad \frac{\Gamma \vdash q_1 : T\ \mathbf{list}, \_ \quad \Gamma \vdash q_2 : T\ \mathbf{list}, \_}{\Gamma \vdash_{\mathsf{MEM}} q_1 \mathbin{@} q_2 : T\ \mathbf{list}}$$

$$\frac{\Gamma \vdash q : \{\ldots, l : T, \ldots\}, \_}{\Gamma \vdash_{\mathsf{MEM}} q \cdot l : T} \qquad \frac{\Gamma \vdash q_1 : T_2\ \mathbf{list}, \_ \quad \Gamma \vdash q_2 : T_2 \to T_2\ \mathbf{list} \to T_1, \_ \quad \Gamma \vdash q_3 : T_1, \_}{\Gamma \vdash_{\mathsf{MEM}} q_1\ \mathbf{as}\ h :: t\ ?\ q_2\ :\ q_3 : T_1}$$

$$\frac{\Gamma \vdash q_1 : T_2 \to T_1, \_ \quad \Gamma \vdash q_2 : T_2\ \mathbf{list}, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathtt{Project}\langle q_1 \mid q_2 \rangle : T_1\ \mathbf{list}} \qquad \frac{\Gamma \vdash q_1 : T \to \mathtt{bool}, \_ \quad \Gamma \vdash q_2 : T\ \mathbf{list}, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathtt{Filter}\langle q_1 \mid q_2 \rangle : T\ \mathbf{list}}$$

$$\frac{\begin{array}{ll}\Gamma \vdash q_1 : T_3 \to T_4 \to T_1, \_ & \Gamma \vdash q_3 : T_3\ \mathbf{list}, \_ \\ \Gamma \vdash q_2 : T_3 \to T_4 \to \mathtt{bool}, \_ & \Gamma \vdash q_4 : T_4\ \mathbf{list}, \_\end{array}}{\Gamma \vdash_{\mathsf{MEM}} \mathtt{Join}\langle q_1, q_2 \mid q_3, q_4 \rangle : T_1\ \mathbf{list}} \qquad \frac{\Gamma \vdash q : T_1, \mathcal{D} \quad T_1 \prec T_2}{\Gamma \vdash_{\mathsf{MEM}} q : T_2}$$

Figure 4.1 – The specific type system of MEM

$\mathbb{D}$ including MEM. This allows the specific type system of MEM to type expressions which children are typeable in other databases. For instance, as explained earlier, in the query of Example 3.10, the `Join` operator has to be typed by the specific type system of MEM since only MEM can perform the operation between the two tables coming from different databases. However, MEM cannot type `From` which is not a supported operator. Therefore, calling the generic type system on the children allows MEM to let the specific type systems of $\mathcal{D}$ and $\mathcal{D}'$ handle the children:

$$\dfrac{\dots \quad \dfrac{\dfrac{\dots}{\Gamma \vdash_{\mathcal{D}} \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle : T_3\ \textbf{list}}}{\Gamma \vdash \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle : T_3\ \textbf{list}, \mathcal{D}} \quad \dfrac{\dfrac{\dots}{\Gamma \vdash_{\mathcal{D}'} \texttt{From}\langle \mathcal{D}', \texttt{"team"}\rangle : T_4\ \textbf{list}}}{\Gamma \vdash \texttt{From}\langle \mathcal{D}', \texttt{"team"}\rangle : T_4\ \textbf{list}, \mathcal{D}'}}{\emptyset \vdash_{\text{MEM}} \begin{array}{l}\texttt{Join}\langle \textbf{fun}(x,y) \rightarrow x \bowtie y, \textbf{fun}(x,y) \rightarrow \texttt{true} \mid \\ \quad \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathcal{D}', \texttt{"team"}\rangle\rangle\end{array} : T_1\ \textbf{list}}$$

$$\emptyset \vdash \begin{array}{l}\texttt{Join}\langle \textbf{fun}(x,y) \rightarrow x \bowtie y, \textbf{fun}(x,y) \rightarrow \texttt{true} \mid \\ \quad \texttt{From}\langle \mathcal{D}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathcal{D}', \texttt{"team"}\rangle\rangle\end{array} : T_1\ \textbf{list}, \text{MEM}$$

As we will see in Section 4.4 where we define a specific type system for SQL, only the specific type system of MEM calls the generic type system. The specific type systems of the database simply call themselves recursively as they do not receive data from another source via the QIR. Therefore, this process of alternating between the generic type system and the specific type systems occurs only if we have no other choice than to execute part of the query in the runtime of MEM.

A *subsumption rule* is added to the specific type system of MEM to solve the issue discussed above about the record types. Indeed, consider Example 4.1.

Example 4.1.

$$(\textbf{fun}(r) \rightarrow r \cdot id)\ \{\, id : 1, name : \texttt{"Maggie"}\,\}$$

This query is typed as:

$$\dfrac{\dfrac{\emptyset \vdash_{\text{MEM}} \textbf{fun}(r) \rightarrow r \cdot id : \{id : \texttt{int}\} \rightarrow \texttt{int}}{\emptyset \vdash \textbf{fun}(r) \rightarrow r \cdot id : \{id : \texttt{int}\} \rightarrow \texttt{int}, \text{MEM}}\ A}{\emptyset \vdash_{\text{MEM}} (\textbf{fun}(r) \rightarrow r \cdot id)\ \{\, id : 1, name : \texttt{"Maggie"}\,\} : T}$$

$$A = \dfrac{\emptyset \vdash_{\text{MEM}} \{\, id : 1, name : \texttt{"Maggie"}\,\} : \{id : \texttt{int}, name : \texttt{string}\} \quad \{id : \texttt{int}, name : \texttt{string}\} \preceq \{id : \texttt{int}\}}{\emptyset \vdash_{\text{MEM}} \{\, id : 1, name : \texttt{"Maggie"}\,\} : \{id : \texttt{int}\}}$$

Additionally, the subsumption excludes the case where the supertype is equal to the subtype to avoid infinite derivations, as one could apply the subsumption rule on the same type as the premise since the subtyping relation is reflexive:

$$\frac{\dots \quad T \preceq T}{\Gamma \vdash_{\mathsf{MEM}} q : T} \quad T \preceq T \qquad \Gamma \vdash_{\mathsf{MEM}} q : T$$

As usual, the subsumption rule is not syntax-directed. This is one of the reasons why the specific type system of MEM is not algorithmic. As explained in [Oho95] and as we will see in Chapter 5, it is possible to define an equivalent type system for MEM that is algorithmic, in particular by removing the subsumption rule. However, the type system presented in Definition 4.8 being easier to understand and work with, we use this definition throughout the paper and prove the equivalence in Chapter 5.

As another example, recall the query from Example 3.2:

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot id \leq 2 \mid [\,\{id : 1\}, \{id : 2\}, \{id : 3\}\,] \rangle$$

This query is typed as:

$$\dfrac{\dfrac{\dfrac{\dots}{\{r : \{id : \texttt{int}\}\} \vdash r \cdot id \leq 2 : \texttt{bool}, \mathsf{MEM}}}{\emptyset \vdash_{\mathsf{MEM}} \mathbf{fun}(r) \rightarrow r \cdot id \leq 2 : \{id : \texttt{int}\} \rightarrow \texttt{bool}} \qquad \dfrac{\dots}{\emptyset \vdash [\,\{id : 1\}, \dots\,] : \{id : \texttt{int}\}\ \mathbf{list}, \mathsf{MEM}}}{\emptyset \vdash \mathbf{fun}(r) \rightarrow r \cdot id \leq 2 : \{id : \texttt{int}\} \rightarrow \texttt{bool}, \mathsf{MEM}}$$

$$\emptyset \vdash_{\mathsf{MEM}} \texttt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot id \leq 2 \mid [\,\{id : 1\}, \{id : 2\}, \{id : 3\}\,] \rangle : \{id : \texttt{int}\}\ \mathbf{list}$$

For the QIR type system to be consistent, we restrict specific type systems to prevent them from conflicting with each other. For example, we can have a database stating that a record cannot have lists as elements, but we cannot have a database expect a list as first element for the record destructor. The only exceptions being data operators, that can be freely typed by the databases.

Definition 4.9 (Specialization of a type inference judgment)**.** A type inference judgment $j = \Gamma \vdash_{\mathcal{D}} q : T$ is a *specialization of another type inference judgment* $j' = \Gamma' \vdash_{\mathcal{D}'} q' : T'$ or $j' = \Gamma' \vdash q' : T', \mathcal{D}'$ noted $j \subseteq j'$ if and only if:

1. $\Gamma = \Gamma'$
2. $q = q'$
3. $T \preceq T'$

Definition 4.10 (Specialization of a type inference rule)**.** A type inference rule $(\mathcal{A}, c)$ is a *specialization of another type inference rule* $(\mathcal{A}', c')$ noted $(\mathcal{A}, c) \subseteq (\mathcal{A}', c')$ if and only if, for any instance of the inference rules:

1. $c \subseteq c'$
2. $\forall j' \in \mathcal{A}'. \exists j \in \mathcal{A} \mid j \subseteq j'$

Intuitively, an inference rule is a specialization if it does not contradict the premises and conclusion of the other inference rule. For instance:

$$\frac{}{\Gamma \vdash_{\mathcal{D}} [\,] : B \textbf{ list}}\ {\scriptstyle B \in \mathcal{B}}$$

is a specialization of the empty list rule of the specific type system of MEM, but

$$\frac{}{\Gamma \vdash_{\mathcal{D}} q_1\ q_2 : \texttt{bool}}$$

is not a specialization of the application rule of the specific type system of MEM.

Definition 4.11 (Target of a type inference rule)**.** The *target of a type inference rule* $\Gamma \vdash_{\mathcal{D}} q : T$ or $\Gamma \vdash q : T, \mathcal{D}$ is q.

Definition 4.12 (Coherence of a specific type system)**.** A specific type system $\vdash_{\mathcal{D}}$ is *coherent* if for every type inference rule $(\mathcal{A}, c)$ of $\vdash_{\mathcal{D}}$, either c targets a data operator, or there exists a type inference rule $(\mathcal{A}', c')$ of $\vdash_{\text{MEM}}$ such that $(\mathcal{A}, c) \subseteq (\mathcal{A}', c')$.

Property 4.3 (Coherence of the specific type systems)**.** *We assume that all the specific type systems linked to QIR are coherent with the specific type system for MEM.*

In some cases, it could be possible and more efficient to migrate data from one database to another. For instance, in our Example 3.10, if $\mathcal{D}$ has `Join` as a compatible data operator and if the data in table `"team"` is transferable to $\mathcal{D}$, then we could transfer this information to $\mathcal{D}$ to have this database perform the `Join` instead of MEM.

Another possibility of data transfer would be to use specific type systems on QIR expressions that do not refer to data stored in databases. For instance, this query:

$$o_{\mathcal{D}}\langle [1, 2, 3] \rangle$$

where $o_{\mathcal{D}}$ is an operator specific to the database $\mathcal{D}$, cannot be successfully typed by our generic type system as is, but it could be extended to send the data in QIR form to the database $\mathcal{D}$. We do not explore this possibility in this thesis, but this could be done by transferring data from one database to QIR, and then from QIR to the other database, or by adding migration operators to the supported operators of a database that transfer data directly from one database to another if that feature is supported by a database.

4.3 Type safety

We now have a fully functional type system for QIR. In this section, we make use of this type system to prove interesting properties on the evaluation and normalization of QIR expressions.

4.3.1 Progress and preservation of types

First, we want to prove that if a QIR expression is typed by the generic type system with a type T, then the application of $\hookrightarrow$ to this QIR expression is also typed by the generic type system with a type T. This property is especially interesting to us, since the normalization bases itself on $\hookrightarrow$. Therefore, this property gives us the guarantee that the normalization preserves the type of the QIR expression, thus guaranteeing that our normalization does not change the semantics of a QIR expression. We proceed in a standard way by first proving the substitution lemma.

Definition 4.13 (Substitution lemma for $\mathcal{D}$)**.** If $\Gamma, x : T' \vdash q : T, \mathcal{D}$ and $\Gamma \vdash q' : T', \mathcal{D}$ then $\Gamma \vdash q\{x/q'\} : T, \mathcal{D}$.

Classically, we assume that x is not bound by a function in q, using α-conversion if need be. Since MEM can call any other type system using the generic type system, our property being true for MEM requires the property being true for the other specific type systems.

Lemma 4.1 (Substitution lemma for MEM)**.** *Let $q \in \mathsf{E}_{QIR}$ and Γ a QIR typing environment. Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus$ MEM, the substitution lemma holds. Then, the substitution lemma holds for MEM.*

Proof. If $\mathcal{D} \neq$ MEM, then the property is true by hypothesis. Suppose now that $\mathcal{D} =$ MEM. We prove the property by induction on the derivation of $\Gamma, x : T' \vdash_{\mathsf{MEM}} q : T$, since it is the only possible step after $\Gamma, x : T' \vdash q : T, \mathsf{MEM}$. If the last rule used is the subsumption rule, then it is immediately

true by induction hypothesis, otherwise:

- $q \equiv x_1$:

  If $x_1 = x$, and so $T' = T$, then the property is true since $x_1\{x/q'\} = q'$. Otherwise, $x_1\{x/q'\} = x_1$, and so the property is obviously true.

- $q \equiv \mathbf{fun}^f(x_1) \rightarrow q_1$:

$$\frac{\Gamma, x : T', f : T_1 \rightarrow T_2, x_1 : T_1 \vdash q_1 : T_2, \_}{\Gamma, x : T' \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x_1) \rightarrow q_1 : T_1 \rightarrow T_2}$$

  By induction hypothesis, we have $\Gamma, f : T_1 \rightarrow T_2, x_1 : T_1 \vdash q_1\{x/q'\} : T_2, \_$. Therefore, we have $\Gamma \vdash \mathbf{fun}^f(x_1) \rightarrow q_1\{x/q'\} : T_1 \rightarrow T_2, \mathsf{MEM}$, so by definition of the substitution: $\Gamma \vdash (\mathbf{fun}^f(x_1) \rightarrow q_1)\{x/q'\} : T_1 \rightarrow T_2, \mathsf{MEM}$.

- $q \equiv q_1\ q_2$:

$$\frac{\Gamma, x : T' \vdash q_1 : T_1 \rightarrow T_2, \_ \quad \Gamma, x : T' \vdash q_2 : T_1, \_}{\Gamma, x : T' \vdash_{\mathsf{MEM}} q_1\ q_2 : T_2}$$

  By induction hypothesis, we have $\Gamma \vdash q_1\{x/q'\} : T_1 \rightarrow T_2, \_$ and $\Gamma \vdash q_2\{x/q'\} : T_1, \_$. Therefore, we have $\Gamma \vdash q_1\{x/q'\}\ q_2\{x/q'\} : T_2, \mathsf{MEM}$, so by definition of the substitution: $\Gamma \vdash (q_1\ q_2)\{x/q'\} : T_1 \rightarrow T_2, \mathsf{MEM}$.

- $q \equiv c$, $q \equiv op$, $q \equiv [\ ]$: Immediate since $q\{x/q'\} = q$.

For all other cases, the lemma is true for the same argument as for $q_1\ q_2$, by applying the induction hypothesis on every child expression. □

We can now state our type preservation theorem.

Definition 4.14 (Subject reduction for $\mathcal{D}$)**.** We have subject reduction for a database $\mathcal{D}$ on a reduction relation R if $\Gamma \vdash q : T, \mathcal{D}$ and $q\ R\ q'$ implies $\Gamma \vdash q' : T, \mathcal{D}'$.

Theorem 4.1 (Subject reduction for MEM)**.** *Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus$ MEM, we have subject reduction on $\hookrightarrow$. Then, we have subject reduction for MEM on $\hookrightarrow$.*

Proof. If $\mathcal{D} \neq \mathsf{MEM}$, then the property is true by hypothesis. Suppose now that $\mathcal{D} = \mathsf{MEM}$. We prove the property by induction on the derivation of $\Gamma \vdash_{\mathsf{MEM}} q : T$, since it is the only possible step after $\Gamma \vdash q : T, \mathsf{MEM}$. We use L4.1 to denote Lemma 4.1, and P4.3 to denote Property 4.3.

- For all the (*-red*) and (dataop-*) rules, the property is immediately true by applying the induction hypothesis and the hypothesis that we have subject reduction for $\mathcal{D} \neq \mathsf{MEM}$.
- $(\mathbf{fun}^f(x) \rightarrow q_1)\ v_2 \hookrightarrow \{f \mapsto \mathbf{fun}^f(x) \rightarrow q_1, x \mapsto v_2\}q_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \{f \mapsto \mathbf{fun}^f(x) \rightarrow q_1, x \mapsto v_2\}q_1 : T_2, \_}{\Gamma, f : T_1 \rightarrow T_2, x : T_1 \vdash_{\mathsf{MEM}} q_1 : T_2}\ \text{L4.1}}{\Gamma \vdash_{\mathcal{D}} \mathbf{fun}^f(x) \rightarrow q_1 : T_1 \rightarrow T_2}\ \text{P4.3}}{\Gamma \vdash \mathbf{fun}^f(x) \rightarrow q_1 : T_1 \rightarrow T_2, \mathcal{D}} \qquad \Gamma \vdash v_2 : T_1, \_}{\Gamma \vdash_{\mathsf{MEM}} (\mathbf{fun}^f(x) \rightarrow q_1)\ v_2 : T_2}$$

  We used the substitution lemma twice here: once for f and once for x.

The complete proof can be found in Appendix B, page 189. □

We can also show properties of safety on typeable QIR expressions this time for the reduction relation $\twoheadrightarrow$ defined in Definition 3.9. We prove the properties of progress and subject reduction basing ourselves on our proofs for $\hookrightarrow$.

Theorem 4.2 (Progress)**.** *Let $q \in \mathsf{E}_{QIR}$, and $\mathcal{D}$ a database language. If $\emptyset \vdash q : T, \mathcal{D}$ and all data operators in q are translatable into a database language, then either q is a QIR value, or $\exists q'.q \twoheadrightarrow q'$.*

Proof. By induction on typing derivations:

- If $q = x$, then impossible since the rule

$$\dfrac{}{\Gamma, x : T \vdash_{\mathsf{MEM}} x : T}$$

  is not applicable since our environment is the empty set. And by Property 4.3, any typing rule for variables in other specific type systems cannot be applied either.
- If $q = \mathbf{fun}^x(x) \rightarrow q'$, q is a value.

- If $q = q_1\ q_2$, then either
  - $q_1 \equiv \mathbf{fun}^x(x) \rightarrow q_1'$ and q_2 is a value, in which case rule (app-β) applies;
  - $q_1 \equiv op$ and q_2 is a value, in which case rule (app-op) applies;
  - or $q_1 \not\equiv \mathbf{fun}^x(x) \rightarrow q_1'$ or op, in which case q_1 is not a value by typing and can be reduced by induction hypothesis.
- If $q = o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n \rangle$, then either
  - all q_i are values, in which case (ext-database) applies by hypothesis;
  - or at least one q_i is not a value, in which case either rule (dataop-conf) or (dataop-data) apply.

The complete proof can be found in Appendix B, page 192. □

We made an extra hypothesis in our theorem of progress: all data operators present in a query must be translatable into a database language. This is coherent with our first motivation of the chapter for a type system: if the type system of a database can type an expression, then this expression should be translatable. We will see in Section 4.4 how to construct such a type system for SQL.

Theorem 4.3 (Subject reduction). *Let $q \in \mathsf{E}_{QIR}$ and Γ a QIR typing environment. Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus \mathit{MEM}$, we have subject reduction on $\twoheadrightarrow$. Then, we have subject reduction on $\twoheadrightarrow$ for all $\mathcal{D} \in \mathbb{D}$.*

Proof. Since $\mathtt{eval}^{\mathcal{D}}(e)$ and $\blacksquare_{\mathcal{H}}(\gamma, e)$ are not typeable by the generic type system by Property 4.3, and since we have subject reduction for $\rightarrow$, we have subject reduction for $\twoheadrightarrow$ if $\xrightarrow{\mathcal{D}}$ and $\overrightarrow{\mathtt{VAL}}^{\mathcal{D}}(v)$ preserve types. □

4.3.2 Strong normalization

Lemma 3.1 and Theorem 4.1 guarantee that a term cannot get reduced into a term that is neither reducible nor a normal form, and that the normalization preserves types. The property we want to prove next is the *strong normalization* of expressions successfully typed by our generic type system. A QIR expression is strongly normalizing if any path of reduction of that expression using the relation $\hookrightarrow$ terminates. Associated with our safety properties, the strong normalization property gives us the guarantee that an expression typed in the generic type system can be normalized, and that the process terminates.

However, we first have to restrict $\mathsf{E}_{\mathsf{QIR}}$ to non-recursive functions. Indeed, if recursive functions were allowed, then the strong normalization would not hold. For instance, consider the following expression:

$$(\mathbf{fun}^f(x) \rightarrow f\ x)\ 2$$

Using the (norm-app-β) rule of $\hookrightarrow$, we would get:

$$(f\ x)\{f/\mathbf{fun}^f(x) \rightarrow f\ x,\ x/2\}$$

which is equivalent to:

$$(\mathbf{fun}^f(x) \rightarrow f\ x)\ 2$$

Therefore, we have:

$$(\mathbf{fun}^f(x) \rightarrow f\ x)\ 2 \hookrightarrow (\mathbf{fun}^f(x) \rightarrow f\ x)\ 2$$

So the reduction of this expression does not terminate in this example, it is not strongly normalizing. But the generic translation can type this expression:

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow f\ x : \mathtt{int} \rightarrow \mathtt{int}}{\emptyset \vdash \mathbf{fun}^f(x) \rightarrow f\ x : \mathtt{int} \rightarrow \mathtt{int}, \mathsf{MEM}} \quad \dfrac{\emptyset \vdash_{\mathsf{MEM}} 2 : \mathtt{int}}{\emptyset \vdash 2 : \mathtt{int}, \mathsf{MEM}}}{\emptyset \vdash_{\mathsf{MEM}} (\mathbf{fun}^f(x) \rightarrow f\ x)\ 2 : \mathtt{int}}}{\emptyset \vdash (\mathbf{fun}^f(x) \rightarrow f\ x)\ 2 : \mathtt{int}, \mathsf{MEM}}$$

Thus, we do not have strong normalization with recursive functions. We do not need to remove host language expressions since they are values for $\hookrightarrow$ and are therefore not reduced. So even if a host language expression containing an infinite loop were to appear in an expression, $\hookrightarrow$ would still terminate.

Notation 4.2 (Set of QIR expressions without recursive functions)**.** We note E^1_{QIR} the set $\mathsf{E}_{\mathtt{QIR}}$ without recursive functions.

To prove this property of strong normalization on our restricted QIR, we follow the method described in [Pie02]. First, we define a set of closed terms of type T.

Definition 4.15. Let T be a QIR type, R_T is the set of closed terms $q \in \mathsf{E}^1_{\mathtt{QIR}}$ such that:

- There exists a QIR typing environment Γ and a database language $\mathcal{D}$ such that $\Gamma \vdash q : T, \mathcal{D}$;
- $q \in R_c$, $c \in \mathcal{B}$ if and only if q is strongly normalizing;
- $q \in R_{T_1 \to T_2}$ if and only if q is strongly normalizing and if $q' \in R_{T_1} \implies q\ q' \in R_{T_2}$;
- $q \in R_{T\mathbf{list}}$ if and only if q is strongly normalizing;
- $q \in R_{\{l_i:T_i\}_{i=1..n}}$ if and only if q is strongly normalizing.

As explained in [Pie02], the idea of the proof is to show that every element of every set R_T is strongly normalizing, then to prove that every well-typed expression of type T is an element of R_T. Note that the only destructor that involves a substitution is still only the application. Therefore, only expressions with an arrow type require an extra condition to ensure the evaluation is terminating.

Lemma 4.2. *Let $q \in \mathsf{E}^1_{QIR}$. If $q \in R_T$ then q is strongly normalizing.*

Proof. Immediate by Definition 4.15. □

Lemma 4.3. *Let $q \in \mathsf{E}^1_{QIR}$. If $q \hookrightarrow q'$ and there exists a QIR typing environment Γ and a database language $\mathcal{D}$ such that $\Gamma \vdash q : T, \mathcal{D}$, then $q \in R_T$ if and only if $q' \in R_T$.*

Proof. We prove the lemma by induction on the structure of the type T. First, note that it is immediate that if q halts, then q' halts. If $T \neq T_1 \to T_2$, it is immediate by definition of R_T. Otherwise, for the direction $q \in R_T \Rightarrow q' \in R_T$, suppose that $q \in R_{T_1 \to T_2}$, and $q_1 \in R_{T_1}$. By definition, we have $q\ q_1 \in R_{T_2}$. But $q\ q_1 \hookrightarrow q'\ q_1$, from which the induction hypothesis on type T_2 gives us $q'\ q_1 \in R_{T_2}$. Since q_1 is arbitrary, the definition of $R_{T_1 \to T_2}$ gives us $q' \in R_{T_1 \to T_2}$. The argument for the direction $q \in R_T \Leftarrow q' \in R_T$ is analogous, since with subject reduction we have $\Gamma \vdash q' : T, \mathcal{D}$. □

In QIR, functions are not the only expressions that can have a function type. Therefore, we prove an additional lemma.

Lemma 4.4. *Let $v \in \mathsf{E}^1_{QIR}$ such as v is in normal form, $v \not\equiv \boldsymbol{fun}^x(x)\rightarrow v$, and there exists a QIR typing environment Γ and a database language $\mathcal{D}$ such that $\Gamma \vdash v : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T, \mathcal{D}$. Then, $v \in R_{T_1\rightarrow\ldots\rightarrow T_n\rightarrow T}$.*

Proof. By induction on n.

If $n = 0$, then the property is trivially true by Definition 4.15.

Assume the property true for n, let us prove it for $n+1$: Let $q' \in R_{T_1}$. We have $q' \hookrightarrow^* v'$, and, by Lemma 4.3, $v' \in R_{T_1}$. By Lemma 4.3 again, we have $v\ q' \in R_T$ if and only if $v\ v' \in R_T$, and by induction hypothesis, we have $v\ v' \in R_{T_2\rightarrow\ldots\rightarrow T_n\rightarrow T_{n+1}}$. Therefore, by Definition 4.15, $v \in R_{T_1\rightarrow T_2\rightarrow\ldots\rightarrow T_n\rightarrow T_{n+1}}$. □

Lemma 4.5. *Let $q \in \mathsf{E}^1_{QIR}$ and $v_1, \ldots, v_m \in \mathsf{V}_{QIR}$ be closed QIR values. If there exists QIR typing environments $\Gamma_1, \ldots, \Gamma_m$ and database languages $\mathcal{D}$, $\mathcal{D}_1, \ldots, \mathcal{D}_m$ such that $x_1 : T_1, \ldots, x_m : T_m \vdash q : T, \mathcal{D}$ and $\forall j \in 1..m.\Gamma_j \vdash v_j : T_j, \mathcal{D}_j$ and $v_j \in R_{T_j}$, then $q\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$.*

Proof. We note $\Gamma = \{x_j : T_j\}_{j=1..m}$. By induction on the derivation of $\Gamma \vdash q : T, \mathcal{D}$:

- If $q = x$, then $q = x_j$ and $T = T_j$, in which case the property is obviously true as if $v_j \in R_{T_j}$ then $x_j\{x_j/v_j\} = v_j \in R_{T_j}$.
- If $q = \mathbf{fun}(x)\rightarrow q_1$, then:

$$\dfrac{\dfrac{\Gamma, x : T' \vdash_{\mathsf{MEM}} q_1 : T''}{\Gamma \vdash_{\mathcal{D}} \mathbf{fun}(x)\rightarrow q_1 : T' \rightarrow T''}\ \text{P4.3}}{\Gamma \vdash \mathbf{fun}(x)\rightarrow q_1 : T' \rightarrow T'', \mathcal{D}}$$

  Let $q' \in R_{T'}$. By Lemma 4.2, we have $q' \hookrightarrow^* v$ for some v. By Lemma 4.3, $v \in R_{T'}$. By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m, x/v\} \in R_{T''}$. But, $(\mathbf{fun}(x)\rightarrow q_1\{x_1/v_1, \ldots, x_m/v_m\})\ (q') \hookrightarrow^* q_1\{x_1/v_1, \ldots, x_m/v_m, x/v\}$, which by Lemma 4.3 gives us $(\mathbf{fun}(x)\rightarrow q_1\{x_1/v_1, \ldots, x_m/v_m\})\ (q') \in R_{T''}$. Then, by definition of $R_{T'\rightarrow T''}$, since q' was chosen arbitrarily, we have: $(\mathbf{fun}(x)\rightarrow q_1)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'\rightarrow T''}$.

- If $q = q_1\ q_2$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : T' \to T'' \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T'}{\Gamma \vdash_{\mathcal{D}} q_1\ q_2 : T' \to T''}\ \text{P4.3}}{\Gamma \vdash q_1\ q_2 : T' \to T'', \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T' \to T''}$ and $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'}$. And by definition of $R_{T' \to T''}$, we have $(q_1\ q_2)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T''}$.

- If $q = c$, then $T = \texttt{typeofC}(c) \in \mathcal{B}$, in which case the property is obviously true as c is obviously strongly normalizing therefore $c \in R_{\texttt{typeofC}(c)}$.

- If $q = op$, true by Lemma 4.4.

- If $q = \textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : \texttt{bool} \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T \quad \Gamma \vdash_{\mathsf{MEM}} q_3 : T}{\Gamma \vdash_{\mathcal{D}} \textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3 : T}\ \text{P4.3}}{\Gamma \vdash \textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3 : T, \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\texttt{bool}}$, $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$, and $q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$. Therefore, by Lemma 4.2, we have $q_i\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* v_i'$ for $i \in 1..3$, and so $(\textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3)\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* \textbf{if}\, v_1'\, \textbf{then}\, v_2'\, \textbf{else}\, v_3'$. If $v_1' = \texttt{true}$ or $\texttt{false}$, we have $\textbf{if}\, v_1'\, \textbf{then}\, v_2'\, \textbf{else}\, v_3' \hookrightarrow^* v_2'$ or v_3'. But, by Lemma 4.3, since $q_i\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$, we have $v_i' \in R_T$ for $i \in 2..3$, and so by Lemma 4.3 again $\textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$. Otherwise, either $T \not\equiv T_1 \to T_2$, in which case the property is trivially true by Definition 4.15, or $T = T_1 \to T_2$, in which case the property is true by Lemma 4.4.

The complete proof can be found in Appendix B, page 194. □

Theorem 4.4 (Strong normalization)**.** *Let $q \in \mathsf{E}^1_{QIR}$. If there exists a QIR typing environment Γ and a database language $\mathcal{D}$ such that $\Gamma \vdash q : T, \mathcal{D}$, then q is strongly normalizing.*

Proof. By Lemma 4.5, we have $q \in R_T$. Therefore, by Lemma 4.2, we have that q is strongly normalizing. □

4.4 Specific type system for SQL

As stated in Chapter 1, QIR to SQL is an important translation as it allows BOLDR to target relational databases and a non-negligible number of distributed databases such as Hive or Cassandra. In this section, we create a specific type system for SQL and prove the necessary safety properties.

We assume that the set of values for SQL only contains basic constants (strings, numbers, booleans, ...) and tables. The set of expressions $\mathsf{E}_{\mathsf{SQL}}$ is the set of syntactically valid SQL queries [ISO16]. The supported operators we consider are `Filter`, `Project`, `Join`, `From`, `Group`, `Sort`, `Limit`, and `Exists`.

We next describe the semantics of these operators for SQL. The semantics of `Filter`, `Project`, and `Join` are described in Definition 3.11.
`Group`$\langle \mathbf{fun}(x) \to \{l_i : q_i\}_{i \in 1..n}, f \mid l \rangle$ partitions elements of l in groups using the values they associate to the column names l_i (q_i are not used). The second configuration f of `Group` shares the same purpose as the configuration of `Project`, that is it takes a record as argument and returns another record which will be part of the final result. For instance:

$$\begin{aligned}&\texttt{Group}\langle \mathbf{fun}(x) \to \{\mathit{teamid} : \texttt{true}\},\\ &\qquad \mathbf{fun}(x) \to \{\mathit{teamid} : x \cdot \mathit{teamid}, n : \mathit{count}\ (x \cdot \mathit{id})\} \mid\\ &\texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle\rangle\end{aligned}$$

is the equivalent of the SQL query:

```
SELECT x.teamid AS teamid, count(x.id) AS n
FROM employee AS x
GROUP BY x.teamid
```

and groups employees by their `teamid`, and returns a table describing how many employees there is in each group as shown in Figure 4.2.

id	name	salary	teamid
1	Lily Pond	5200	2
2	Daniel Rogers	4700	1
3	Olivia Sinclair	6000	1

(a) Table Employee

teamid	n
1	2
2	1

(b) Result of `Group` on `teamid`

Figure 4.2 – An example of `Group`

`Sort`$\langle \mathbf{fun}(x) \to \{l_i : q_i\}_{i \in 1..n} \mid l \rangle$ sorts the rows of collection l by the values they associate to the column names l_i and using a natural comparison defined in SQL. The q_i expressions are expected to be booleans. They describe, for each column name to use for sorting, whether or not the natural comparison should be used in ascending order (e.g., increasing order for numbers, alphabetical order

for strings) or in descending order (e.g., decreasing order for numbers, reverse alphabetical order for strings). For instance:

$$\texttt{Sort}\langle \textbf{fun}(x) \rightarrow \{\mathit{teamid} : \texttt{false}, \mathit{name} : \texttt{true}\} \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

is the equivalent of the SQL query:

```
SELECT * FROM employee ORDER BY teamid DESC, name
```

which returns all rows from table `employee` sorted by its values in column `teamid` in decreasing order (note the use of the SQL keyword `DESC`), then by its values in column `name` in alphabetic order.

Finally, $\texttt{Limit}\langle n \mid l \rangle$ returns the n first rows of collection l, and $\texttt{Exists}\langle l \rangle$ returns `false` if the collection l is empty, and `true` otherwise.

Technically, `Sort` and `Limit` are not part of the SQL standard, but they are supported by most SQL databases and commonly used. Additionally, `Limit` gives us the opportunity to show how to handle a case of an operator which configuration is not a function, and `Exists` is an operator that returns a basic type instead of a new collection.

Notation 4.3. We use R to denote types of the form $\{l : B, \ldots, l : B\}$ with $B \in \mathcal{B}$.

Definition 4.16 (Specific SQL type system)**.** The specific type system of SQL is derived by the rules of Figure 4.3.

The type system of Definition 4.16 is designed to type all expressions that can be reduced by the normalization into expressions translatable to SQL.

The SQL notation in $\texttt{From}\langle \text{SQL}, n \rangle$ is used to represent any SQL database.

SQL relies on a flat data model: records can only contain values with base types (*flat records*), and lists can only contain flat records (*flat lists*). Therefore, the type system for SQL only types those kinds of data structures, in particular, the data operators all take as data arguments and return flat lists.

Basic operators and conditional expressions are only valid for basic types in SQL. In SQL, the configurations of `Group` and `Sort` are also expected to return flat records.

Finally, notice there are two rules for `Join`. The second one gives a type to the `Join` operation where the first configuration returns the record concatenation between the records of the two tables, corresponding to a `Join` that returns all fields from the two tables which is a common operation in SQL. However, since SQL does not support record concatenation, we add a custom rule for this particular case.

$$\frac{}{\Gamma, x : T \vdash_{\mathsf{SQL}} x : T} \qquad \frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash_{\mathsf{SQL}} q : T_2}{\Gamma \vdash_{\mathsf{SQL}} \mathbf{fun}^f(x) \to q : T_1 \to T_2} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : T_1 \to T_2 \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : T_1}{\Gamma \vdash_{\mathsf{SQL}} q_1\ q_2 : T_2}$$

$$\frac{}{\Gamma \vdash_{\mathsf{SQL}} c : \mathtt{typeofC}(c)} \qquad \frac{B_1 \to \ldots \to B_n \in \mathtt{typeofOP}(op)}{\Gamma \vdash_{\mathsf{MEM}} op : B_1 \to \ldots \to B_n}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} b_1 : \mathtt{bool} \quad \Gamma \vdash_{\mathsf{SQL}} b_2 : B \quad \Gamma \vdash_{\mathsf{SQL}} b_3 : B}{\Gamma \vdash_{\mathsf{SQL}} \mathbf{if}\, b_1\, \mathbf{then}\, b_2\, \mathbf{else}\, b_3 : B} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} q_i : T_i \quad i \in 1..n}{\Gamma \vdash_{\mathsf{SQL}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : T \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : T\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} q_1 :: q_2 : T\ \mathbf{list}} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : T\ \mathbf{list} \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : T\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} q_1 \mathbin{@} q_2 : T\ \mathbf{list}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q : \{\ldots, l : T, \ldots\}}{\Gamma \vdash_{\mathsf{SQL}} q \cdot l : T} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : R_2 \to R_1 \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : R_2\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Project}\langle q_1 \mid q_2 \rangle : R_1\ \mathbf{list}} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} n : \mathtt{string}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{From}\langle \mathsf{SQL}, n \rangle : R\ \mathbf{list}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : R \to \mathtt{bool} \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : R\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Filter}\langle q_1 \mid q_2 \rangle : R\ \mathbf{list}} \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{SQL}} q_1 : R_3 \to R_4 \to R_1 \qquad \Gamma \vdash_{\mathsf{SQL}} q_3 : R_3\ \mathbf{list} \\ \Gamma \vdash_{\mathsf{SQL}} q_2 : R_3 \to R_4 \to \mathtt{bool} \quad \Gamma \vdash_{\mathsf{SQL}} q_4 : R_4\ \mathbf{list}\end{array}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Join}\langle q_1, q_2 \mid q_3, q_4 \rangle : R_1\ \mathbf{list}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q_2 : \{l_i : T_i\}_{i \in 1..m} \to \{l_i : T_i\}_{i \in m+1..n} \to \mathtt{bool} \qquad \begin{array}{l}\Gamma \vdash_{\mathsf{SQL}} q_3 : \{l_i : T_i\}_{i \in 1..m}\ \mathbf{list} \\ \Gamma \vdash_{\mathsf{SQL}} q_4 : \{l_i : T_i\}_{i \in m+1..n}\ \mathbf{list}\end{array}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Join}\langle \mathbf{fun}(x, y) \to x \bowtie y, q_2 \mid q_3, q_4 \rangle : \{l_i : T_i\}_{i \in 1..n}\ \mathbf{list}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : R_3 \to R_1 \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : R_3 \to R_2 \quad \Gamma \vdash_{\mathsf{SQL}} q_3 : R_3\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Group}\langle q_1, q_2 \mid q_3 \rangle : R_2\ \mathbf{list}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : R_2 \to R_1 \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : R_2\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Sort}\langle q_1 \mid q_2 \rangle : R_2\ \mathbf{list}} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : \mathtt{int} \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : R\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Limit}\langle q_1 \mid q_2 \rangle : R\ \mathbf{list}}$$

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q : R\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \mathtt{Exists}\langle q \rangle : \mathtt{bool}} \qquad \frac{\Gamma \vdash_{\mathsf{SQL}} q : T_1 \quad T_1 \prec T_2}{\Gamma \vdash_{\mathsf{SQL}} q : T_2}$$

Figure 4.3 – The specific type system of SQL

Let us now go through a few examples. Recall Example 3.3 from Chapter 3:

$$\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot salary < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle$$

This query is typed as:

$$\dfrac{\dfrac{\dfrac{\{x : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} r : \{id : \texttt{int}\}}{\{x : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} r \cdot id : \texttt{int}} \quad \dfrac{}{\{x : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} 2500 : \texttt{int}}}{\{x : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} r \cdot id < 2500 : \texttt{bool}} \quad A}{\emptyset \vdash_{\textsf{SQL}} \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle : \{id : \texttt{int}\}\ \textbf{list}}$$

$$A = \dfrac{\emptyset \vdash_{\textsf{SQL}} \texttt{"employee"} : \texttt{string}}{\emptyset \vdash_{\textsf{SQL}} \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle : \{id : \texttt{int}\}\ \textbf{list}}$$

Note that the record type deduced by the type system in this derivation is $\{id : \texttt{int}\}$, but any record type containing the association from id to $\texttt{int}$ can be deduced as well.

Recall the query from Example 3.12:

$$\begin{aligned}&\texttt{Project}\langle \textbf{fun}(r) \rightarrow (\textbf{fun}(x) \rightarrow \{result : x\})\ (r \cdot id) \mid \\ &\qquad \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle\end{aligned}$$

This query is typed as:

$$B = \dfrac{\dfrac{\dfrac{}{\{r : \{id : \texttt{int}\}, x : \texttt{int}\} \vdash_{\textsf{SQL}} x : \texttt{int}}}{\{r : \{id : \texttt{int}\}, x : \texttt{int}\} \vdash_{\textsf{SQL}} \{result : x\} : \{result : \texttt{int}\}}}{\{r : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} \textbf{fun}(x) \rightarrow \{result : x\} : \texttt{int} \rightarrow \{result : \texttt{int}\}}$$

$$A = \dfrac{\dfrac{\dfrac{B \quad \dfrac{\dfrac{}{\{r : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} r : \{id : \texttt{int}\}}}{\{r : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} r \cdot id : \texttt{int}}}{\{r : \{id : \texttt{int}\}\} \vdash_{\textsf{SQL}} (\textbf{fun}(x) \rightarrow \{result : x\})\ (r \cdot id) : \{result : \texttt{int}\}}}{\emptyset \vdash_{\textsf{SQL}} \textbf{fun}(r) \rightarrow (\textbf{fun}(x) \rightarrow \{result : x\})\ (r \cdot id) : \{id : \texttt{int}\} \rightarrow \{result : \texttt{int}\}} \quad \ldots}{A \quad \emptyset \vdash_{\textsf{SQL}} \begin{array}{l}\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \\ \qquad \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle\end{array} : \{id : \texttt{int}\}\ \textbf{list}}$$

$$\emptyset \vdash_{\textsf{SQL}} \begin{array}{l}\texttt{Project}\langle \textbf{fun}(r) \rightarrow (\textbf{fun}(x) \rightarrow \{result : x\})\ (r \cdot id) \mid \\ \qquad \texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot id < 2500 \mid \texttt{From}\langle \mathcal{D}, \texttt{"employee"} \rangle \rangle \rangle\end{array} : \{result : \texttt{int}\}\ \textbf{list}$$

Next, we prove the Property 4.3 of coherence for our type system for SQL.

Lemma 4.6 (Coherence of the specific type system for SQL). *The specific type system for SQL is coherent with the specific type system for MEM.*

Proof. By direct case analysis on the rules of the specific type system for SQL. □

There is no difficulty in the proof of that property, since the only differences between the two type systems are flat record types or basic types instead of any types in some places, some missing rules, and some extra rules for data operators.

Finally, to complete the integration of SQL, we prove the substitution lemma and subject reduction theorem for the specific SQL type system.

Lemma 4.7 (Substitution lemma for SQL). *Let $q \in \mathsf{E}_{QIR}$ and Γ a QIR typing environment. If $\Gamma, x : T' \vdash q : T, \mathsf{SQL}$, then for all $q' \in \mathsf{E}_{QIR}$ such that $\Gamma \vdash q' : T', \mathsf{SQL}$, we have $\Gamma \vdash q\{x/q'\} : T, \mathsf{SQL}$.*

Proof. By induction on the derivation of $\Gamma, x : T' \vdash_{\mathsf{SQL}} q : T$, since it is the only possible step after $\Gamma, x : T' \vdash q : T, \mathsf{SQL}$. The arguments are the same as for the proof of Lemma 4.1. □

Theorem 4.5 (Subject reduction for SQL). *Let $q \in \mathsf{E}_{QIR}$ and Γ a QIR typing environment. If $\Gamma \vdash q : T, \mathsf{SQL}$, and $q \hookrightarrow q'$, then $\Gamma \vdash q' : T, \mathsf{SQL}$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathsf{SQL}} q : T$, since it is the only possible step after $\Gamma \vdash q : T, \mathsf{SQL}$. We use L4.1 to denote Lemma 4.1.

- For all the (*-red*) and (dataop-*) rules, either no typing rule applies, or the property is immediately true by induction hypothesis.
- $(\mathbf{fun}^f(x) \rightarrow q_1)\ v_2 \hookrightarrow \{f \mapsto \mathbf{fun}^f(x) \rightarrow q_1, x \mapsto v_2\}q_1$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{SQL}} \{f \mapsto \mathbf{fun}^f(x) \rightarrow q_1, x \mapsto v_2\}q_1 : T_2}{\Gamma, f : T_1 \rightarrow T_2, x : T_1 \vdash_{\mathsf{SQL}} q_1 : T_2}\ \text{L4.1}}{\Gamma \vdash_{\mathsf{SQL}} \mathbf{fun}^f(x) \rightarrow q_1 : T_1 \rightarrow T_2} \qquad \Gamma \vdash_{\mathsf{SQL}} v_2 : T_1}{\Gamma \vdash_{\mathsf{SQL}} (\mathbf{fun}^f(x) \rightarrow q_1)\ v_2 : T_2}$$

- $op\ v \rightarrow q$: true supposing $\rightarrow^\delta$ preserves types.

- **if true then** v_1 **else** $v_2 \hookrightarrow v_1$:

$$\frac{\Gamma \vdash_{\mathsf{SQL}} \textbf{true} : \texttt{bool} \quad \Gamma \vdash_{\mathsf{SQL}} v_1 : B \quad \Gamma \vdash_{\mathsf{SQL}} v_2 : B}{\Gamma \vdash_{\mathsf{SQL}} \textbf{if true then } v_1 \textbf{ else } v_2 : B}$$

- **if false then** v_1 **else** $v_2 \hookrightarrow v_2$:

$$\frac{\Gamma \vdash_{\mathsf{SQL}} \textbf{false} : \texttt{bool} \quad \Gamma \vdash_{\mathsf{SQL}} v_1 : B \quad \Gamma \vdash_{\mathsf{SQL}} v_2 : B}{\Gamma \vdash_{\mathsf{SQL}} \textbf{if false then } v_1 \textbf{ else } v_2 : B}$$

- $\{ l_i : v_i \}_{i=1..m} \bowtie \{ l_i : v_i \}_{i=m+1..n} \hookrightarrow \{ l_i : v_i \}_{i=1..n}$: No typing rule.
- $[\,] \mathbin{@} v \hookrightarrow v$: No typing rule.
- $v \mathbin{@} [\,] \hookrightarrow v$: No typing rule.
- $(v_1 :: v_2) \mathbin{@} v_3 \hookrightarrow v_1 :: (v_2 \mathbin{@} v_3)$:

$$\frac{\dfrac{\Gamma \vdash_{\mathsf{SQL}} v_1 : T \quad \Gamma \vdash_{\mathsf{SQL}} v_2 : T \textbf{ list}}{\Gamma \vdash_{\mathsf{SQL}} v_1 :: v_2 : T \textbf{ list}} \quad \Gamma \vdash_{\mathsf{SQL}} v_3 : T \textbf{ list}}{\Gamma \vdash_{\mathsf{SQL}} (v_1 :: v_2) \mathbin{@} v_3 : T \textbf{ list}}$$

  so:

$$\frac{\Gamma \vdash_{\mathsf{SQL}} v_1 : T \quad \dfrac{\Gamma \vdash_{\mathsf{SQL}} v_2 : T \textbf{ list} \quad \Gamma \vdash_{\mathsf{SQL}} v_3 : T \textbf{ list}}{\Gamma \vdash_{\mathsf{SQL}} v_2 \mathbin{@} v_3 : T \textbf{ list}}}{\Gamma \vdash_{\mathsf{SQL}} v_1 :: (v_2 \mathbin{@} v_3) : T \textbf{ list}}$$

- $\{ \ldots, l : v, \ldots \} \cdot l \hookrightarrow v$:

$$\frac{\Gamma \vdash_{\mathsf{SQL}} \{ \ldots, l : v, \ldots \} : \{\ldots, l : T, \ldots\}}{\Gamma \vdash_{\mathsf{SQL}} \{ \ldots, l : v, \ldots \} \cdot l : T}$$

- $[\,]$ **as** $h :: t$ **?** q_2 **:** $q_3 \hookrightarrow v_3$: No typing rule.
- $v_1 :: v_1'$ **as** $h :: t$ **?** q_2 **:** $q_3 \hookrightarrow q_2\ (v_1, v_1')$: No typing rule.

□

We now have a functional type system for QIR in which we can also type queries targeting SQL databases. In the next chapter, we create typing algorithms that are equivalent to our type systems but are also suitable for implementation.

Chapter 5

QIR type inference

QIR expressions do not include type annotations in their syntax since they originate from dynamically typed programming languages. Therefore, to give a type to a QIR expression, BOLDR has to generate it. A well-known technique in statically typed programming languages is to use a *type inference* algorithm such as the Hindley–Milner type inference algorithm [Mil78] which generates a type for an expression of the lambda calculus.

In Chapter 4, we presented *specific* type systems which give a type to QIR expressions that are compatible with a database, and a *generic* type system for QIR which makes use of specific type systems to give a type to a QIR expression that targets one or more databases. The generic type system is perfectly useable as-is as an algorithm since it simply applies specific type systems in a deterministic way. However, the specific type systems defined in Chapter 4, useful for presentation and formal developments, are not algorithmic. In particular, we cannot directly use them to create a type inference algorithm for QIR expressions.

Indeed, if we recall the MEM type system of Definition 4.8, there are four problematic rules which are the function rule; the basic operator rule; the empty list rule; and the subsumption rule.

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash q : T_2, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \to q : T_1 \to T_2} \qquad \frac{T \in \texttt{typeofOP}(op)}{\Gamma \vdash_{\mathsf{MEM}} op : T}$$

$$\frac{}{\Gamma \vdash_{\mathsf{MEM}} [\,] : T\ \mathbf{list}} \qquad \frac{\Gamma \vdash_{\mathsf{MEM}} q : T_1 \quad T_1 \prec T_2}{\Gamma \vdash_{\mathsf{MEM}} q : T_2}$$

The reasons why these rules are problematic are standard and explained in [Pie02]. First, notice that T_1 and T_2 are mentioned in the premise of the function rule to type the body of the function. However, these types where not generated by the type system. T_1 and T_2 appear out of nowhere, meaning that the rule guesses T_1 and T_2, then merely checks that its guess was correct. This is obviously not viable for an algorithm since the number of possible types is infinite. The same problem appears in the empty list rule, for which the type

T is guessed. The basic operator rule picks a type in the set of possibly infinite types returned by `typeofOP`. As for the subsumption rule, the issue is that it can be applied to any QIR term. Therefore, at every step, a typing algorithm would have to choose whether to use this rule or a syntax-directed rule.

The standard solution to creating a typing algorithm based on a type system that can deduce infinitely many typing derivations for the same term is to use *type schemes*. A type scheme uses *polymorphic type variables* to denote an infinite number of similar types, called *instances*, which can be obtained by substituting variables of the type scheme with types. This solution is used in ML, in which the identity function is given the polymorphic type scheme $\forall\alpha.\alpha \to \alpha$.

To infer a type scheme for a term, we apply the technique of constraint solving [PR05]: we define typing algorithms that return a unique type for an expression by generating type variables and accumulating type constraints on these variables. Using type variables and constraints allow us to remove the subsumption rule by shifting the use of subtyping to the constraint resolution, and to generate type variables instead of guessing types.

However, our QIR also features record concatenation, which is notoriously hard to type when combined with a subtyping relation. ML-style type schemes are not expressive enough to describe the set of acceptable types for a calculus supporting these features. Solutions exist in the literature, such as bounded quantification [Pot98] for subtyping, and row variables [Rém89, Rém92] for record concatenation. Alternatively, we could also use semantic subtyping [CNXA15].

These approaches significantly complexify types and constraints, thus making algorithms manipulating types more complex and less efficient. We choose a more pragmatic solution based on two observations. First, the subtyping relation introduced in Chapter 4 only exists to deal with record types. Second, our goal is to precisely type expressions representing queries. We do not need to give an exact type to all QIR terms. We only want types that give us enough information to perform operations on queries such as normalization and translations. For instance, the expression

$$\mathbf{fun}(x, y) \to x \bowtie y$$

is not an expression that we want to send to a database, and thus we do not have much use for an exact type on this kind of QIR expressions.

Therefore, in this chapter, we define typing algorithms for the QIR following the approach of Ohori [Oho95]. To that end, we first add type variables to QIR types, then define *constraints* on types allowing us to restrict the set of valid *type substitutions* from Definition 2.3 that can be applied to a type. We separate our constraints in two categories: equality constraints between types, and *kind* constraints on record types. This allows us to treat subtyping constraints separately which simplifies our proofs and our constraint system. We extend the kinds of [Oho95] to account for record concatenation. This solution is a good compromise that keeps our typing algorithms simple but precise enough for our

needs. Note that, contrary to the QIR, the calculus used in [Oho95] is annotated with types. Because of this, we diverge slightly from this approach in the developments of this chapter, since we start from untyped expressions of dynamic programming languages, and thus we want to be able to manipulate untyped QIR expressions. Our typing algorithms return a type and sets of constraints that restrain the possible types that can replace the type variables present in the type. Finally, we prove our typing algorithms to be equivalent with the type systems defined in Chapter 4, and define a constraint solving algorithm that generates a valid substitution given sets of constraints generated by our typing algorithms.

5.1 Typing algorithms

First, we extend QIR types to type variables. We will use the notation T to denote algorithmic types.

Definition 5.1 (QIR types for typing algorithm)**.** An algorithmic QIR type is a finite term of the following grammar:

$$\begin{array}{rcl} \mathsf{T} & ::= & \mathsf{B} \\ & | & \mathsf{T} \to \mathsf{T} \\ & | & \mathsf{T}\ \textbf{list} \\ & | & \{l : \mathsf{T}, \ldots, l : \mathsf{T}\} \\ & | & \alpha \end{array}$$

We use a classic ML-style annotation for type variables: $\alpha \to \texttt{int}$ represents the type of functions that for any type α returns a value of type `int`. A free type variable is implicitly bound to a universal quantifier in a prenex form. For instance, $\alpha \to \alpha\ \textbf{list} \to \alpha\ \textbf{list}$ is actually a syntactic shortcut for $\forall\alpha : \alpha \to \alpha\ \textbf{list} \to \alpha\ \textbf{list}$.

Definition 5.2 (Set of type constraints)**.** A *set of type constraints* $\mathbb{C}$ is a finite set of equations between types noted $\{\mathsf{T}_i = \mathsf{T}'_i\}_{i \in 1..n}$. A type substitution σ *satisfies a set of type constraints* $\mathbb{C} = \{\mathsf{T}_1 = \mathsf{T}'_1, \ldots, \mathsf{T}_n = \mathsf{T}'_n\}$ noted $\sigma \models \mathbb{C}$ if $\sigma\mathsf{T}_i = \sigma\mathsf{T}'_i$ for $i \in 1..n$.

For instance:

- $\{\alpha \mapsto \texttt{int}\}$ satisfies $\{\alpha = \texttt{int}\}$
- $\{\alpha \mapsto \texttt{int}\}$ satisfies $\{\texttt{int} = \texttt{int}\}$
- $\{\alpha \mapsto \texttt{int}, \alpha' \mapsto \texttt{int}\}$ satisfies $\{\alpha = \texttt{int}, \alpha' = \alpha\}$

- $\{\alpha \mapsto \texttt{int}, \alpha' \mapsto \texttt{string}\}$ does not satisfy $\{\alpha = \texttt{int}, \alpha' = \alpha\}$

We now define a *kind*. Kinds allow us to give a description of a record type.

Definition 5.3 (Kind)**.** A *kind* k is either:

- a *record kind* $\{\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\}\}$
- a *record union kind* $(\mathsf{T}_1, \mathsf{T}_2)$
- or a *universal kind* $\mathtt{U}$

Definition 5.4 (Set of kind constraints)**.** A *set of kind constraints* $\mathbb{K}$ is a finite mapping between types noted $\{\alpha_i \overset{\text{k}}{=} k_i\}_{i \in 1..n}$. A type substitution σ *satisfies a set of kind constraints* $\mathbb{K} = \{\alpha_i \overset{\text{k}}{=} \{\{l_{j,i} : \mathsf{T}_{j,i}\}\}\}_{i=1..n} \cup \{\alpha_{i'} \overset{\text{k}}{=} (\mathsf{T}_{i',1}, \mathsf{T}_{i',2})\}_{i'=1..n'} \cup \{\alpha_{i''} \overset{\text{k}}{=} \mathtt{U}\}_{i''=1..n''}$ noted $\sigma \overset{\text{k}}{\models} \mathbb{K}$ if:

- $\sigma\alpha_i \preceq \sigma\{l_{j,i} : \mathsf{T}_{j,i}\}$ for $i \in 1..n$
- $\sigma\mathsf{T}_{i',1} = \{l_i : \mathsf{T}_i\}_{i=1..m}$
- $\sigma\mathsf{T}_{i',2} = \{l_i : \mathsf{T}_i\}_{i=m+1..n}$
- $\sigma\alpha_{i'} = \{l_i : \mathsf{T}_i\}_{i=1..n}$

Thus, a record kind restricts substitutions of a type variable to record types that are subtypes of the record type described by the kind, a record union kind restricts substitutions to *the* record type that represents the concatenation of two record types, and the universal kind does not add any restriction. For instance:

- $\{\alpha \mapsto \texttt{int}\}$ satisfies $\{\alpha \overset{\text{k}}{=} \mathtt{U}\}$
- $\{\alpha \mapsto \{l : \texttt{int}\}\}$ satisfies $\{\alpha \overset{\text{k}}{=} \{\{l : \texttt{int}\}\}\}$
- $\{\alpha \mapsto \{l : \texttt{int}\}\}$ satisfies $\{\alpha \overset{\text{k}}{=} \mathtt{U}\}$
- $\{\alpha \mapsto \{\}\}$ does not satisfy $\{\alpha \overset{\text{k}}{=} \{l : \texttt{int}\}\}$
- $\{\alpha \mapsto \{l_1 : \texttt{int}, l_2 : \texttt{string}\}\}$ satisfies $\{\alpha \overset{\text{k}}{=} \{\{l_1 : \texttt{int}\}\}\}$
- $\{\alpha \mapsto \{l_1 : \texttt{int}, l_2 : \texttt{bool}\}\}$ satisfies $\{\alpha \overset{\text{k}}{=} (\{l_1 : \texttt{int}, l_2 : \texttt{int}\}, \{l_2 : \texttt{bool}\})\}$

Note that since a type constraint corresponds to an equality relation between types, which is a symmetrical relation, the constraints $\alpha = \mathsf{T}$ and $\mathsf{T} = \alpha$ are equivalent. In opposite, a kind constraint corresponds to a subtyping relation between types, which is not symmetric. Thus, order matters in a kind constraint.

We can now give a definition of our typing algorithms. Similarly to Chapter 4, specific typing algorithms give types to QIR expressions that are compatible with corresponding databases, and the generic typing algorithm makes specific typing algorithms work together.

Definition 5.5 (Specific typing algorithm)**.** A *specific typing algorithm* for a database $\mathcal{D}$ denoted by the judgment $\Gamma \vdash^{\mathcal{A}}_{\mathcal{D}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})$ is a type system relation between a QIR typing environment Γ, a QIR term q, and a QIR type T constrained by $\mathbb{C}$ and $\mathbb{K}$.

The generic type system being already suitable for implementation, we can define the QIR generic typing algorithm in a similar way as in Definition 4.7, but we use the specific typing algorithms instead.

Definition 5.6 (QIR generic typing algorithm)**.** A QIR term q has a type T for the database $\mathcal{D}$ derivable from a QIR type environment Γ in the QIR generic typing algorithm, noted $\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \mathcal{D}$. The set of inference rules used to derive this judgment is:

$$\frac{\forall q_i \in \mathfrak{C}(q).\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}_i, (\mathbb{C}_i, \mathbb{K}_i), \mathcal{D}_i \quad \Gamma \vdash^{\mathcal{A}}_{\mathcal{D}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})}{\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \mathcal{D}} \begin{array}{l}\{\mathcal{D}_i\} = \{\mathcal{D}, \mathsf{MEM}\} \\ \mathcal{D} \neq \mathsf{MEM}\end{array}$$

$$\frac{\Gamma \vdash^{\mathcal{A}}_{\mathcal{D}} \texttt{From}\langle \mathcal{D}, \_ \rangle : \mathsf{T}, (\mathbb{C}, \mathbb{K})}{\Gamma \vdash^{\mathcal{A}} \texttt{From}\langle \mathcal{D}, \_ \rangle : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \mathcal{D}} \; \mathcal{D} \neq \mathsf{MEM} \qquad \frac{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})}{\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \mathsf{MEM}}$$

Now that we have a definition of generic and specific typing algorithms, we can define specific typing algorithms for databases, starting with one for MEM.

5.2 Specific typing algorithm for MEM

In this section, we define a specific typing algorithm for MEM, and prove that it is equivalent to the specific type system for MEM of Definition 4.8.

Definition 5.7 (Specific typing algorithm for MEM)**.** The specific typing algorithm of MEM noted $\vdash_{\mathsf{MEM}}$ is derived by the rules of Figure 5.1.

$$\frac{}{\Gamma, x : \mathsf{T} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \mathsf{T}, (\emptyset, \emptyset)} \qquad \frac{\alpha_1 \text{ and } \alpha_2 \text{ fresh} \quad \Gamma, f : \alpha_1 \to \alpha_2, x : \alpha_1 \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \_}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}^f(x) \to q : \alpha_1 \to \mathsf{T}, (\mathbb{C} \cup \{\alpha_2 = \mathsf{T}\}, \mathbb{K} \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1\ q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \to \alpha\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})} \qquad \frac{}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} c : \mathtt{typeofC}(c), (\emptyset, \emptyset)}$$

$$\frac{\{\alpha_1, \ldots, \alpha_n\} = TV(\mathtt{typeofOP}(op))}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} op : \mathtt{polytypeofOP}(op), (\emptyset, \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \ldots, \alpha_n \stackrel{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \Gamma \vdash^{\mathcal{A}} q_3 : \mathsf{T}_3, (\mathbb{C}_3, \mathbb{K}_3), \_ \quad \alpha_1 \text{ and } \alpha_2 \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \alpha_2, \left(\begin{array}{l} \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}, \\ \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\} \end{array} \right)}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}_i, (\mathbb{C}_i, \mathbb{K}_i), \_ \quad i \in 1..n}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : \mathsf{T}_i\}_{i=1..n}, (\bigcup_{i=1}^{n} \mathbb{C}_i, \bigcup_{i=1}^{n} \mathbb{K}_i)}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \bowtie q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} (\mathsf{T}_1, \mathsf{T}_2)\})} \qquad \frac{\alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} [\,] : \alpha\ \mathbf{list}, (\emptyset, \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 :: q_2 : \alpha\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 @ q_2 : \alpha\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha\ \mathbf{list} = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})}$$

Figure 5.1 – The specific typing algorithm for MEM (part 1 of 2)

$$\frac{\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \_ \quad \alpha_1, \alpha_2 \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q \cdot l : \alpha_2, (\mathbb{C} \cup \{\alpha_1 = \mathsf{T}\}, \mathbb{K} \cup \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}_i, (\mathbb{C}_i, \mathbb{K}_i), \_ \quad i \in 1..3 \quad \alpha_1 \text{ and } \alpha_2 \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 : \alpha_2, \left(\begin{array}{l} \bigcup_i \mathbb{C}_i \cup \{\alpha_1\ \mathbf{list} = \mathsf{T}_1, \alpha_1 \to \alpha_1\ \mathbf{list} \to \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}, \\ \bigcup_i \mathbb{K}_i \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\} \end{array} \right)}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha_1 \text{ and } \alpha_2 \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Project}\langle q_1 \mid q_2 \rangle : \alpha_2\ \mathbf{list}, \left(\begin{array}{l} \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \alpha_1 \to \alpha_2, \alpha_1\ \mathbf{list} = \mathsf{T}_2\}, \\ \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\} \end{array} \right)}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Filter}\langle q_1 \mid q_2 \rangle : \alpha\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \alpha \to \mathtt{bool}, \alpha\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\frac{\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}_i, (\mathbb{C}_i, \mathbb{K}_i), \_ \quad i \in 1..4 \quad \alpha_1, \alpha_2 \text{ and } \alpha_3 \text{ fresh}}{\begin{array}{l} \Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Join}\langle q_1, q_2 \mid q_3, q_4 \rangle : \alpha_3\ \mathbf{list}, (\bigcup_i \mathbb{C}_i \cup \{\mathsf{T}_1 = \alpha_1 \to \alpha_2 \to \alpha_3, \mathsf{T}_2 = \alpha_1 \to \alpha_2 \to \mathtt{bool}, \\ \qquad \alpha_1\ \mathbf{list} = \mathsf{T}_3, \alpha_2\ \mathbf{list} = \mathsf{T}_4\}, \bigcup_i \mathbb{K}_i \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathrm{k}}{=} \mathtt{U}\}) \end{array}}$$

Figure 5.1 – The specific typing algorithm for MEM (part 2 of 2)

There are several points to discuss about this typing algorithm. First, notice that the function rule, instead of guessing the argument and return types, generates two type variables to denote them. As a consequence, restrictions on types in the premises have all disappeared, and have been replaced by type constraints. This is because at any point in the premises the typing algorithm could (and probably will) return a type variable, which forces us to handle all type restrictions using constraints. For instance, an access to a variable that is bound as a function argument is always typed with a type variable in this type system.

Second, as expected, the only rules that add kind constraints are the record concatenation and the record destructor rules. All other constraints are equality constraints between types.

Third, our problems with the subsumption rule and the rules for the function and for the empty list disappeared as desired. Indeed, the function rule generates the two type variables that are used to type the function, the empty list rule generates the type of the elements of the list, and the subsumption rule disappeared altogether.

Fourth, notice that we use a function called `polytypeofOP` for basic operators. This function takes a basic operator and returns a type that can contain type variables which represent all of the types returned by `typeofOP` called on the same operator. This is represented by the following property:

Property 5.1 (Coherence of `polytypeofOP`)**.** *Let op be a basic operator. Then $T \in \texttt{typeofOP}(op)$ iff there exists σ such that $\sigma(\texttt{polytypeofOP}(op)) = T$ with $TV(T) = \emptyset$.*

In other words, the types returned by `typeofOP` for a basic operator are exactly the possible types obtained by substituting all of the type variables present in the type returned by `polytypeofOP`. For instance, $\texttt{polytypeofOP}(=) = \alpha \to \alpha \to \texttt{bool}$. This solves our problem with the basic operators rule being non-algorithmic in the specific type system for MEM.

Fifth, the subtyping relation does not appear anywhere in our typing algorithm. Classically, rules for destructors such as the application or the list destructor would have a subtyping condition. For instance, for the application, we would have a rule such as:

$$\frac{\vdash q_1 : T_1 \to T_2 \quad \vdash q_2 : T_1' \quad T_1' \preceq T_1}{\vdash q_1\ q_2 : T_2}$$

In our typing algorithm, we move all subtyping restrictions to type constraints, or to kind constraints in the cases of record concatenation and record destruction.

As an example of how the typing algorithm for MEM works, recall Example 4.1 from Chapter 4:

$$(\mathbf{fun}(r) \to r \cdot id)\ \{\, id : 1, name : \texttt{"Maggie"} \,\}$$

Typing the function with the specific typing algorithm of MEM (omitting the calls to the generic typing algorithm for presentation) gives us:

$$\dfrac{\dfrac{\dfrac{}{\{r : \alpha_2\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} r : \alpha_2, (\emptyset, \emptyset)}}{\{r : \alpha_2\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} r \cdot id : \alpha_4, (\{\alpha_3 = \alpha_2\}, \{\alpha_3 \overset{\mathrm{k}}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{\mathrm{k}}{=} \mathtt{U}\})}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}(r) \rightarrow r \cdot id : \alpha_2 \to \alpha_4, (\{\alpha_3 = \alpha_2\}, \{\alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathrm{k}}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{\mathrm{k}}{=} \mathtt{U}\})}$$

The typing algorithm gives this function the type $\alpha_2 \to \alpha_4$ along with the constraints that α_2 is equal to α_3, and that α_3 can only be substituted to a record type containing at least the association from the label id to the type α_4. In other words, the type of this function after applying a type substitution that satisfies all constraints generated by the typing algorithm can be $\{id : \mathtt{int}\} \to \mathtt{int}$, $\{id : \mathtt{bool}\} \to \mathtt{bool}$, $\{id : \mathtt{int}, name : \mathtt{string}\} \to \mathtt{int}$, ... or any other function type taking a record type containing an association for the label id and returning the type associated to id in its argument type.

The application is then typed as:

$$\dfrac{\ldots \quad \dfrac{}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{\, id : 1, name : \texttt{"Maggie"} \,\} : \{id : \mathtt{int}, name : \mathtt{string}\}, (\emptyset, \emptyset)}}{\begin{array}{l}\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} (\mathbf{fun}(r) \rightarrow r \cdot id)\ \{\, id : 1, name : \texttt{"Maggie"} \,\} : \\ \alpha_1, (\{\alpha_2 \to \alpha_4 = \{id : \mathtt{int}, name : \mathtt{string}\} \to \alpha_1, \alpha_3 = \alpha_2\}, \\ \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathrm{k}}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{\mathrm{k}}{=} \mathtt{U}\})\end{array}}$$

The type system gives this application the type α_1 along with the extra constraint that the function type we obtained earlier is equal to the function type $\{id : \mathtt{int}, name : \mathtt{string}\} \to \alpha_1$. Combining the constraints, we get that the type of the function must be $\{id : \mathtt{int}, name : \mathtt{string}\} \to \mathtt{int}$, and that the type of the application must be $\mathtt{int}$ as expected. We will see later in this chapter how to solve those constraints to generate the substitution that give us the correct result.

As another example, recall the query from Example 3.2:

$$\mathtt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot id \leq 2 \mid [\,\{id : 1\}, \{id : 2\}, \{id : 3\}\,]\rangle$$

Even though this example is quite simple, generating a type for this query using the typing algorithm involves building a derivation of a respectable size. Therefore, we proceed in several steps. Let us first type the configuration (omitting the calls to the generic type system, universal kinds, and reducing the of the data argument for presentation):

$$C = \dfrac{\dfrac{}{\{r : \alpha_1\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} r : \alpha_1, (\emptyset, \emptyset)}}{\{r : \alpha_1\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} r \cdot id : \alpha_6, (\{\alpha_5 = \alpha_1\}, \{\alpha_5 \overset{\mathrm{k}}{=} \{\{id : \alpha_6\}\}\})}$$

$$B = \frac{\dfrac{}{\{r : \alpha_1\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} \leq : \alpha_4 \to \alpha_4 \to \texttt{bool}, (\emptyset, \emptyset)} \quad C}{\{r : \alpha_1\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} \leq \; r \cdot id : \alpha_3, (\{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \alpha_3\}, \{\alpha_5 \overset{\text{k}}{=} \{\{id : \alpha_6\}\}\})}$$

$$A = \frac{\dfrac{B \quad \dfrac{}{\{r : \alpha_1\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} 2 : \texttt{int}, (\emptyset, \emptyset)}}{\{r : \alpha_1\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} r \cdot id \leq 2 : \alpha_2, (\{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \alpha_3, \alpha_3 = \texttt{int} \to \alpha_2\}, \{\alpha_5 \overset{\text{k}}{=} \{\{id : \alpha_6\}\}\})}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \textbf{fun}(r) \to r \cdot id \leq 2 : \alpha_1 \to \alpha_2, (\{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \alpha_3, \alpha_3 = \texttt{int} \to \alpha_2\}, \{\alpha_5 \overset{\text{k}}{=} \{\{id : \alpha_6\}\}\})}$$

The constraints tell us that the configuration is a function which argument is a record containing an association for label id by $\alpha_5 = \alpha_1$ and $\alpha_5 \overset{\text{k}}{=} \{\{id : \alpha_6\}\}$. Then by $\alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \alpha_3$ and $\alpha_3 = \texttt{int} \to \alpha_2$, we get that $\alpha_6 = \alpha_4 = \texttt{int}$ and $\alpha_2 = \texttt{bool}$. Therefore, according to the constraints, our configuration should have type $\{id : \texttt{int}, \ldots\} \to \texttt{bool}$, which is what we expected.

The entire query is then be typed as:

$$\frac{A \quad \dfrac{\dfrac{\dfrac{}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} 1 : \texttt{int}, (\emptyset, \emptyset)}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{id : 1\} : \{id : \texttt{int}\}, (\emptyset, \emptyset)} \quad \dfrac{}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} [\,] : \alpha_9 \; \textbf{list}, (\emptyset, \emptyset)}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} [\,\{id : 1\}\,] : \alpha_8 \; \textbf{list}, (\{\alpha_8 = \{id : \texttt{int}\}, \alpha_8 \; \textbf{list} = \alpha_9 \; \textbf{list}\}, \emptyset)}}{\begin{array}{l}\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \texttt{Filter}\langle \textbf{fun}(r) \to r \cdot id \leq 2 \mid [\,\{id : 1\}\,] \rangle : \alpha_7 \; \textbf{list}, (\\ \quad \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7 \; \textbf{list} = \alpha_8 \; \textbf{list}\}, \{\alpha_5 \overset{\text{k}}{=} \{\{id : \alpha_6\}\}\})\end{array}}$$

These constraints restrict the argument type of the configuration to $\{id : \texttt{int}\}$ by $\alpha_7 \to \texttt{bool} = \alpha_1 \to \alpha_2$, $\alpha_7 \; \textbf{list} = \alpha_8 \; \textbf{list}$, and $\alpha_8 = \{id : \texttt{int}\}$, and tell us that the query returns a list of type $\{id : \texttt{int}\} \; \textbf{list}$ which is what we expected as well. Note that, just like in the previous example, if the records had more fields then the type in the equality constraint would have changed to $\alpha_8 = \{id : \texttt{int}, \ldots\}$ which would still have solutions along with the kind constraint since it only forces the result type to be substituted to a record type containing the association label id to $\texttt{int}$.

One last example to illustrate record union kinds:

$$(\{\, x : 1, y : 2 \,\} \bowtie \{\, z : \texttt{true} \,\}) \cdot x$$

Typing it gives us:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} 1 : \mathtt{int}, (\emptyset, \emptyset)} \quad \dfrac{}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} 2 : \mathtt{int}, (\emptyset, \emptyset)}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{ x : 1, y : 2 \} : \{x : \mathtt{int}, y : \mathtt{int}\}, (\emptyset, \emptyset)} \quad \dfrac{\dfrac{}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{true} : \mathtt{bool}, (\emptyset, \emptyset)}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{ z : \mathtt{true} \} : \{z : \mathtt{bool}\}, (\emptyset, \emptyset)}}{\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{ x : 1, y : 2 \} \bowtie \{ z : \mathtt{true} \} : \alpha_1, (\emptyset, \{\alpha_1 \overset{\mathrm{k}}{=} (\{x : \mathtt{int}, y : \mathtt{int}\}, \{z : \mathtt{bool}\})\})}$$

$$\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} (\{ x : 1, y : 2 \} \bowtie \{ z : \mathtt{true} \}) \cdot x : \alpha_3, (\{\alpha_2 = \alpha_1\}, \{\alpha_1 \overset{\mathrm{k}}{=} (\{x : \mathtt{int}, y : \mathtt{int}\}, \{z : \mathtt{bool}\}), \alpha_2 \overset{\mathrm{k}}{=} \{\{x : \alpha_3\}\}, \alpha_3 \overset{\mathrm{k}}{=} \mathtt{U}\})$$

These constraints restrict the result type to the type associated to the label x in the record restricted by the record union kind of the types of the two records.

We now want to prove that the typing algorithm for MEM is equivalent to the specific type system for MEM. This equivalence allows us to transfer the properties of Chapter 4 for the specific type system for MEM to the typing algorithm for MEM.

We first prove that if the typing algorithm for MEM applied on a QIR expression gives a type T and sets of constraints, then applying a type substitution that satisfies the constraints to T should give us a type that is derivable in the specific type system for MEM. In other words, there should be no way for the typing algorithm to give to an expression a type that cannot be deduced by the specific type system.

This property of soundness is crucial since it guarantees us that *for any* σ the specific type system can deduce the corresponding type. Therefore, all the properties deduced in Chapter 4 are carried over to QIR expressions typed by the typing algorithm.

Definition 5.8 (Soundness of a specific typing algorithm)**.** A specific typing algorithm $\vdash^{\mathcal{A}}_{\mathcal{D}}$ is *sound* if $\forall \Gamma, q, \sigma : (\Gamma \vdash^{\mathcal{A}}_{\mathcal{D}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}) \wedge TV(\sigma\mathsf{T}) = \emptyset \wedge \sigma \models \mathbb{C} \wedge \sigma \overset{\mathrm{k}}{\models} \mathbb{K}) \implies \sigma\Gamma \vdash_{\mathcal{D}} q : \sigma T$.

Theorem 5.1 (Soundness of the typing algorithm for MEM)**.** *Let* $q \in \mathsf{E}_{QIR}$ *and* Γ *a QIR typing environment. Suppose that for all* $\mathcal{D} \in \mathbb{D} \setminus \mathit{MEM}$, $\vdash^{\mathcal{A}}_{\mathcal{D}}$ *is sound. Then,* $\vdash^{\mathcal{A}}_{\mathit{MEM}}$ *is sound.*

Proof. By induction on the typing derivation of $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})$:

- $\Gamma, x : \mathsf{T} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \mathsf{T}, (\emptyset, \emptyset)$

  The property is immediately true since we have $\sigma\Gamma, x : \sigma T \vdash_{\mathsf{MEM}} x : \sigma T$ which is true for any σ.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow q : \alpha_1 \rightarrow \mathsf{T}, (\mathbb{C} \cup \{\alpha_2 = \mathsf{T}\}, \mathbb{K} \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C} \cup \{\alpha_2 = \mathsf{T}\}$ and for $\mathbb{K} \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}$ and $\mathbb{K}$ and $\sigma\alpha_2 = \sigma\mathsf{T}$. Since we have $\Gamma, f : \alpha_1 \rightarrow \alpha_2, x : \alpha_1 \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \_$, by induction hypothesis we get $\sigma\Gamma, f : \sigma\alpha_1 \rightarrow \sigma\alpha_2, x : \sigma\alpha_1 \vdash q : \sigma T, \_$, so $\sigma\Gamma, f : \sigma\alpha_1 \rightarrow \sigma T, x : \sigma\alpha_1 \vdash_{\mathsf{MEM}} q : \sigma T$. Therefore, we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow q : \sigma\alpha_1 \rightarrow \sigma T$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1\ q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \rightarrow \alpha\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \rightarrow \alpha\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$ and $\sigma\mathsf{T}_1 = \sigma\mathsf{T}_2 \rightarrow \sigma\alpha$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \sigma T_2 \rightarrow \sigma\alpha, \_$. Therefore, we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1\ q_2 : \sigma\alpha$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} c : \mathtt{typeofC}(c), (\emptyset, \emptyset)$

  The property is immediately true since we have $\sigma\Gamma \vdash_{\mathsf{MEM}} c : \mathtt{typeofC}(c) = \sigma\mathtt{typeofC}(c)$ for any σ.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \alpha_2, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\mathbb{K}_3$ and $\sigma\alpha_1 = \sigma\mathsf{T}_1 = \mathtt{bool}$, $\sigma\alpha_2 = \sigma\mathsf{T}_2$, $\sigma\alpha_2 = \sigma\mathsf{T}_3$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$, $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, and $\Gamma \vdash^{\mathcal{A}} q_3 : \mathsf{T}_3, (\mathbb{C}_3, \mathbb{K}_3), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$, $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, and $\sigma\Gamma \vdash q_3 : \sigma T_3, \_$, so $\sigma\Gamma \vdash q_1 : \mathtt{bool}, \_$, $\sigma\Gamma \vdash q_2 : \sigma\alpha_2, \_$, and $\sigma\Gamma \vdash q_3 : \sigma\alpha_2, \_$. Therefore, we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \sigma\alpha_2$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \bowtie q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} (\mathsf{T}_1, \mathsf{T}_2)\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} (\mathsf{T}_1, \mathsf{T}_2)\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, and $\sigma\mathsf{T}_1 = \{l_i : \mathsf{T}'_i\}_{i=1..m}$, $\sigma\mathsf{T}_2 = \{l_i : \mathsf{T}'_i\}_{i=m+1..n}$, $\sigma\alpha = \{l_i : \mathsf{T}'_i\}_{i=1..n}$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \{l_i : T'_i\}_{i=1..m}, \_$, $\sigma\Gamma \vdash q_2 : \{l_i : T'_i\}_{i=m+1..n}, \_$. Therefore, we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1 \bowtie q_2 : \{l_i : T'_i\}_{i=1..n} = \sigma\alpha$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q \cdot l : \alpha_2, (\mathbb{C} \cup \{\alpha_1 = \mathsf{T}\}, \mathbb{K} \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

Let σ be a solution for $\mathbb{C} \cup \{\alpha_1 = \mathsf{T}\}$ and $\mathbb{K} \cup \{\alpha_1 \overset{k}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{k}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}$ and $\mathbb{K}$ and $\sigma\alpha_1 = \sigma\mathsf{T}$, $\sigma\alpha_1 \preceq \{l : \sigma\alpha_2\}$, so $\sigma\alpha_1 = \{\ldots, l : \sigma\alpha_2, \ldots\}$. Since we have $\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q : \sigma T, \_$, so $\sigma\Gamma \vdash q : \{\ldots, l : \sigma\alpha_2, \ldots\}, \_$. Therefore, we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q \cdot l : \sigma\alpha_2$.

The complete proof can be found in Appendix B, page 197. □

We now want to prove that our typing algorithm is complete. Namely, if the specific type system for MEM applied to a QIR expression returns a type T, then the corresponding typing algorithm returns a type T' and constraints such that there exists a type substitution satisfying the constraints that applied to T' gives T.

The completeness property gives us a measure of precision of our typing algorithm, since the typing algorithm that does not type anything is sound, but not very useful. The property of completeness shows that the typing algorithm gives us the same properties than the type system on the same set of QIR expressions.

As explained at the beginning of this chapter, our types cannot express polymorphic record concatenation precisely. This loss in precision is not reflected in our specific typing algorithm since our constraints are powerful enough to express those types, it is instead exposed in our algorithm of constraints resolution, presented in Section 5.3.

Note that the order of the composition of type substitutions matters. For instance, we have:

- $\{\alpha \mapsto \mathtt{int}\} \circ \{\alpha' \mapsto \mathtt{string}\} = \{\alpha \mapsto \mathtt{int}, \alpha' \mapsto \mathtt{string}\}$
- $\{\alpha \mapsto \mathtt{int}\} \circ \{\alpha \mapsto \mathtt{string}\} = \{\alpha \mapsto \mathtt{string}\}$
- $\{\alpha' \mapsto \mathtt{int}\} \circ \{\alpha \mapsto \alpha'\} = \{\alpha \mapsto \mathtt{int}, \alpha' \mapsto \mathtt{int}\}$

This last example gives us a substitution that successfully replaces two type variables with an actual type, but we have to be careful with the order of the composition. Indeed, doing the composition the other way gives us:

$$\{\alpha \mapsto \alpha'\} \circ \{\alpha' \mapsto \mathtt{int}\} = \{\alpha \mapsto \alpha', \alpha' \mapsto \mathtt{int}\}$$

which does not replace α':

- $\{\alpha \mapsto \mathtt{int}, \alpha' \mapsto \mathtt{int}\}(\alpha \to \alpha') = \mathtt{int} \to \mathtt{int}$
- $\{\alpha \mapsto \alpha', \alpha' \mapsto \mathtt{int}\}(\alpha \to \alpha') = \alpha' \to \mathtt{int}$

The completeness proof of the typing algorithm of MEM including the following lemma involves the composition of type substitutions, but the problem shown

above appears only when a type variable in the domain of the second substitution appears in the image of the first substitution. However, our typing algorithm always adds constraints that have fresh variables in their domains, which allows the proofs to avoid the problem by composing substitutions in the correct order which is that the substitution with fresh variables in its domain must be the second substitution in the composition.

First, since we do not have any type annotation to guide us for the QIR function, we need a preliminary property of coherence on all specific typing algorithms.

Definition 5.9 (Coherence of a specific typing algorithm)**.** If $\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2, x : \mathsf{T}_1 \vdash^{\mathcal{A}}_{\mathcal{D}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})$ then $\Gamma, f : \alpha_1 \to \alpha_2, x : \alpha_1 \vdash^{\mathcal{A}}_{\mathcal{D}} q : \mathsf{T}', (\mathbb{C}', \mathbb{K}')$ with α_1 and α_2 fresh variables, and for all σ that satisfies $\mathbb{C}$ and $\mathbb{K}$, $\sigma' = \sigma \circ \{\alpha_1 \mapsto \mathsf{T}_1, \alpha_2 \mapsto \mathsf{T}_2\}$ satisfies $\mathbb{C}'$ and $\mathbb{K}'$ and $\sigma\mathsf{T} = \sigma'\mathsf{T}'$.

Property 5.2 (Coherence of the specific typing algorithms)**.** *We assume that all the specific typing algorithms linked to QIR are coherent.*

Of course, we next prove the coherence of our specific typing algorithm for MEM.

Lemma 5.1 (Coherence of the specific typing algorithm for MEM)**.** *The specific typing algorithm for MEM is coherent.*

Proof. By case analysis on $\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2, x : \mathsf{T}_1 \vdash^{\mathcal{A}}_{\mathsf{MEM}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})$. Since all the premises accept any type, using type variables in Γ would simply result in replacing T_1 and T_2 everywhere by α_1 and α_2. So $\{\alpha_1 \mapsto \mathsf{T}_1, \alpha_2 \mapsto \mathsf{T}_2\}\mathsf{T}' = \mathsf{T}$, $\{\alpha_1 \mapsto \mathsf{T}_1, \alpha_2 \mapsto \mathsf{T}_2\}\mathbb{C}' = \mathbb{C}$, and $\{\alpha_1 \mapsto \mathsf{T}_1, \alpha_2 \mapsto \mathsf{T}_2\}\mathbb{K}' = \mathbb{K}$. □

We can now define the completeness of a specific typing algorithm, and prove it for the typing algorithm of MEM.

Definition 5.10 (Completeness of a specific typing algorithm)**.** A specific typing algorithm $\vdash^{\mathcal{A}}_{\mathcal{D}}$ is *complete* if $\Gamma \vdash_{\mathcal{D}} q : T \implies (\Gamma \vdash^{\mathcal{A}}_{\mathcal{D}} q : \mathsf{T}', (\mathbb{C}, \mathbb{K}) \wedge (\exists \sigma \mid \sigma \models \mathbb{C} \wedge \sigma \overset{\mathrm{k}}{\models} \mathbb{K} \wedge \sigma\mathsf{T}' = \mathsf{T}))$.

Theorem 5.2 (Completeness of the typing algorithm of MEM)**.** *Let $q \in \mathsf{E}_{QIR}$ and Γ a QIR typing environment. Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus$ MEM, $\vdash^{\mathcal{A}}_{\mathcal{D}}$ is complete. Then, $\vdash^{\mathcal{A}}_{MEM}$ is complete.*

Proof. By induction on the typing derivation of $\Gamma \vdash_{\mathsf{MEM}} q : T$:

- $\Gamma, x : T \vdash_{\mathsf{MEM}} x : T$

  Immediate since $\Gamma, x : \mathsf{T} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \mathsf{T}, (\emptyset, \emptyset)$ by taking $\sigma = \emptyset$.

- $\Gamma \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow q : T_1 \to T_2$

  We have $\Gamma, f : T_1 \to T_2, x : T_1 \vdash q : T_2, \_$. By induction hypothesis, we get $\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2, x : \mathsf{T}_1 \vdash^{\mathcal{A}} q : \mathsf{T}', (\mathbb{C}, \mathbb{K}), \_$, and there exists σ' that satisfies $\mathbb{C}$ and $\mathbb{K}$ such that $\sigma'\mathsf{T}' = \mathsf{T}_2$. So by Property 5.2 $\Gamma, f : \alpha_1 \to \alpha_2, x : \alpha_1 \vdash^{\mathcal{A}} q : \mathsf{T}'', (\mathbb{C}', \mathbb{K}'), \_$, and $\sigma = \sigma' \circ \{\alpha_1 \mapsto \mathsf{T}_1, \alpha_2 \mapsto \mathsf{T}_2\}$ satisfies $\mathbb{C}'$ and $\mathbb{K}'$ and $\sigma\mathsf{T}'' = \sigma'\mathsf{T}' = \mathsf{T}_2$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow q : \alpha_1 \to \mathsf{T}'', (\mathbb{C}' \cup \{\alpha_2 = \mathsf{T}''\}, \mathbb{K}' \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$, and σ satisfies $\mathbb{C}' \cup \{\alpha_2 = \mathsf{T}''\}$ and $\mathbb{K}' \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\}$. And we have $\sigma(\alpha_1 \to \mathsf{T}'') = \mathsf{T}_1 \to \mathsf{T}_2$.

- $\Gamma \vdash_{\mathsf{MEM}} q_1\ q_2 : T_2$

  We have $\Gamma \vdash q_1 : T_1 \to T_2, \_$ and $\Gamma \vdash q_2 : T_1, \_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}_1 \to \mathsf{T}_2$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}_1$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1\ q_2 : \alpha, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}'_1 = \mathsf{T}'_2 \to \alpha\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \mathsf{T}_2\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, $\mathbb{K}_2$, so σ satisfies $\mathbb{K}$ and since α is fresh $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}_1$ and $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T}_1 \to \mathsf{T}_2 = \sigma\mathsf{T}'_2 \to \sigma\alpha$, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha = \mathsf{T}_2$.

- $\Gamma \vdash_{\mathsf{MEM}} c : \mathtt{typeofC}(c)$

  Immediate since $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{c} : \mathtt{typeofC}(c), (\emptyset, \emptyset)$ by taking $\sigma = \emptyset$.

- $\Gamma \vdash_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : T$

  We have $\Gamma \vdash q_1 : \mathtt{bool}, \_$, $\Gamma \vdash q_2 : T, \_$, and $\Gamma \vdash q_3 : T, \_$. By induction hypothesis, we get, for $i \in 1..3$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathtt{bool}$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}$, $\sigma_3\mathsf{T}'_3 = \mathsf{T}$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \alpha_2, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}'_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}'_2, \alpha_2 = \mathsf{T}'_3\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 \circ \{\alpha_1 \mapsto \mathtt{bool}, \alpha_2 \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\mathbb{K}_3$, so σ satisfies $\mathbb{K}$ and since α_1 and α_2 are fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathtt{bool} = \sigma\alpha_1$, $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T} = \sigma\alpha_2$, $\sigma\mathsf{T}'_3 = \sigma_3\mathsf{T}'_3 = \mathsf{T} = \sigma\alpha_2$, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha_2 = \mathsf{T}$.

- $\Gamma \vdash_{\mathsf{MEM}} q_1 \bowtie q_2 : \{l_i : T''_i\}_{i=1..n}$

We have $\Gamma \vdash q_1 : \{l_i : T''_i\}_{i=1..m}, \_$ and $\Gamma \vdash q_2 : \{l_i : T''_i\}_{i=m+1..n}, \_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i)$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1 \mathsf{T}'_1 = \{l_i : \mathsf{T}''_i\}_{i=1..m}$, $\sigma_2 \mathsf{T}'_2 = \{l_i : \mathsf{T}''_i\}_{i=m+1..n}$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \bowtie q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} (\mathsf{T}'_1, \mathsf{T}'_2)\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \{l_i : \mathsf{T}''_i\}_{i=1..n}\}$ satisfies $\mathbb{C}$ and $\mathbb{K}$ and since α is fresh $\sigma \mathsf{T}'_1 = \sigma_1 \mathsf{T}'_1 = \{l_i : \mathsf{T}''_i\}_{i=1..m}$ and $\sigma \mathsf{T}'_2 = \sigma_2 \mathsf{T}'_2 = \{l_i : \mathsf{T}''_i\}_{i=m+1..n}$. And we have $\sigma\alpha = \{l_i : \mathsf{T}''_i\}_{i=1..n}$.

- $\Gamma \vdash_{\mathsf{MEM}} q \cdot l : T$

  We have $\Gamma \vdash q : \{\ldots, l : T, \ldots\}, \_$. By induction hypothesis, we get $\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}', (\mathbb{C}, \mathbb{K}), \_$, and there exists σ' that satisfies $\mathbb{C}$ and $\mathbb{K}$ such that $\sigma' \mathsf{T}' = \{\ldots, l : \mathsf{T}, \ldots\}$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q \cdot l : \alpha_2, (\mathbb{C} = \mathbb{C}' \cup \{\alpha_1 = \mathsf{T}'\}, \mathbb{K} = \mathbb{K}' \cup \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma' \circ \{\alpha_1 \mapsto \{\ldots, l : \mathsf{T}, \ldots\}, \alpha_2 \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}'$ and $\mathbb{K}'$, and since α is fresh $\sigma \mathsf{T}' = \sigma' \mathsf{T}' = \{\ldots, l : \mathsf{T}, \ldots\} = \{\ldots, l : \sigma\alpha_2, \ldots\} = \sigma\alpha_1$, so since $\{\ldots, l : \sigma\alpha_2, \ldots\} \preceq \{l : \sigma\alpha_2\}$, σ satisfies $\mathbb{C}$ and $\mathbb{K}$. And we have $\sigma\alpha_2 = \mathsf{T}$.

The complete proof can be found in Appendix B, page 201. □

Crucially, our completeness theorem does not directly give us a substitution to apply to the type returned by the generic typing algorithm applied to a QIR expression. Indeed, our proof generates a substitution *if it is already provided a type*, a type for the QIR expression is provided and the completeness builds the corresponding substitution. For instance, consider the example:

Example 5.1.

$$\mathbf{fun}(x) \rightarrow \{\, l : x \cdot l, orig : x \,\}$$

can be given the types $\{l : \mathtt{int}\} \to \{l : \mathtt{int}, orig : \{l : \mathtt{int}\}\}$, or $\{l : \mathtt{bool}\} \to \{l : \mathtt{bool}, orig : \{l : \mathtt{bool}\}\}$, or even $\{id : \mathtt{int}, name : \mathtt{string}\} \to \{id : \mathtt{int}, orig : \{id : \mathtt{int}, name : \mathtt{string}\}\}$ by the generic type system.

The generic typing algorithm gives us

$$\dfrac{\dfrac{\dfrac{}{\{x : \alpha_3\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \alpha_3, (\emptyset, \emptyset)}}{\{x : \alpha_3\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x \cdot l : \alpha_2, (\{\alpha_1 = \alpha_3\}, \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})} \quad \dfrac{}{\{x : \alpha_3\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \alpha_3, (\emptyset, \emptyset)}}{\{x : \alpha_3\} \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{\, l : x \cdot l, orig : x \,\} : \{l : \alpha_2, orig : \alpha_3\}, (\{\alpha_1 = \alpha_3\}, \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})}$$

$$\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}(x) \rightarrow \{\, l : x \cdot l, orig : x \,\} : \alpha_3 \to \{l : \alpha_2, orig : \alpha_3\}, (\{\alpha_1 = \alpha_3\}, \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathrm{k}}{=} \mathtt{U}\})$$

If we follow the proof of completeness taking $\{l : \texttt{int}\} \to \{l : \texttt{int}, orig : \{l : \texttt{int}\}\}$ as the type given by the specific type system, we get as expected the final substitution $\{\alpha_1 \mapsto \{l : \texttt{int}\}, \alpha_2 \mapsto \texttt{int}, \alpha_3 \mapsto \{l : \texttt{int}\}\}$ which gives us indeed the same type. An equivalent result is obtained if we replace $\{l : \texttt{int}\}$ by $\{l : \texttt{bool}\}$ or $\{id : \texttt{int}, name : \texttt{string}\}$.

In the next section, we present an algorithm to generate a substitution directly from the result of the generic typing algorithm to infer a type for a QIR expression with no help from the specific type system.

5.3 Constraint resolution

The last missing piece of our type inference algorithm is the generation of a solution for our sets of constraints. We adapt the unification algorithm of [Oho95]. We have additional rules: one to handle the record union kinds fully applied, three to handle polymorphic record union kinds, and one to handle equality constraints between list types.

Definition 5.11 (Unification algorithm)**.** The *unification of constraints* $\mathbb{C}$, $\mathbb{K}$, noted $\texttt{unify}(\mathbb{C}, \mathbb{K})$, applies the following rules on $(\mathbb{C}, \mathbb{K}, \emptyset, \emptyset)$ respecting the order until none of them apply:

1. $(\mathbb{C} \cup \{\mathsf{T} = \mathsf{T}\}, \mathbb{K}, \sigma, \mathbb{SK}) \Rightarrow (\mathbb{C}, \mathbb{K}, \sigma, \mathbb{SK})$

2. $(\mathbb{C} \cup \{\alpha = \mathsf{T}\}, \mathbb{K} \cup \{\alpha \overset{\text{k}}{=} \mathtt{U}\}, \sigma, \mathbb{SK}) \Rightarrow$
   $(\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \overset{\text{k}}{=} \mathtt{U}\})$
   if $\alpha \notin TV(\mathsf{T})$
   where $\sigma' = \{\alpha \mapsto \mathsf{T}\}$

3. $(\mathbb{C} \cup \{\alpha_1 = \alpha_2\}, \mathbb{K} \cup \{\alpha_1 \overset{\text{k}}{=} \{\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}\},$
   $\alpha_2 \overset{\text{k}}{=} \{\{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\}\}\}, \sigma, \mathbb{SK}) \Rightarrow$
   $(\sigma'(\mathbb{C} \cup \{\mathsf{T}'_1 = \mathsf{T}''_1, \ldots, \mathsf{T}'_o = \mathsf{T}''_o\}),$
   $\sigma'(\mathbb{K} \cup \{\alpha_2 \overset{\text{k}}{=} \{\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}\}\}),$
   $\sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha_1 \overset{\text{k}}{=} \{\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}\}\})$
   where $\sigma' = \{\alpha_1 \mapsto \alpha_2\}$

4. $(\mathbb{C} \cup \{\alpha = \{l_i : \mathsf{T}_i\}_{i=1..m..n}\}, \mathbb{K} \cup \{\alpha \overset{\text{k}}{=} \{\{l_j : \mathsf{T}'_j\}\}_{j=1..m}\}, \sigma, \mathbb{SK}) \Rightarrow$
   $(\sigma'(\mathbb{C} \cup \{\mathsf{T}'_j = \mathsf{T}_j \mid j \in 1..m\}), \sigma'\mathbb{K},$
   $\sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \overset{\text{k}}{=} \{\{l_j : \mathsf{T}'_j\}\}_{j=1..m}\})$
   if $\alpha \notin TV(\{l_i : \mathsf{T}_i\}_{i=1..m..n})$
   where $\sigma' = \{\alpha \mapsto \{l_i : \mathsf{T}_i\}_{i=1..m..n}\}$

5. $(\mathbb{C} \cup \{\{l_i : \mathsf{T}_i\}_{i=1..n} = \{l_i : \mathsf{T}'_i\}_{i=1..n}\}, \mathbb{K}, \sigma, \mathbb{SK}) \Rightarrow$
$(\mathbb{C} \cup \{\mathsf{T}_i = \mathsf{T}'_i \mid i \in 1..n\}, \mathbb{K}, \sigma, \mathbb{SK})$

6. $(\mathbb{C} \cup \{\mathsf{T}_1 \to \mathsf{T}_2 = \mathsf{T}'_1 \to \mathsf{T}'_2\}, \mathbb{K}, \sigma, \mathbb{SK}) \Rightarrow$
$(\mathbb{C} \cup \{\mathsf{T}_1 = \mathsf{T}'_1, \mathsf{T}_2 = \mathsf{T}'_2\}, \mathbb{K}, \sigma, \mathbb{SK})$

7. $(\mathbb{C} \cup \{\mathsf{T}_1\ \textbf{list} = \mathsf{T}_2\ \textbf{list}\}, \mathbb{K}, \sigma, \mathbb{SK}) \Rightarrow (\mathbb{C} \cup \{\mathsf{T}_1 = \mathsf{T}_2\}, \mathbb{K}, \sigma, \mathbb{SK})$

8a. $(\mathbb{C}, \mathbb{K} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\}, \sigma, \mathbb{SK}) \Rightarrow$
$(\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\})$
if $\forall i \in 1..n.\alpha \notin TV(T_i)$ **and** $\forall j \in 1..o.\alpha \notin TV(T''_j)$
where $\sigma' = \{\alpha \mapsto \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\}\}$

8b. $(\mathbb{C}, \mathbb{K} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\alpha_1, \alpha_2)\}, \sigma, \mathbb{SK}) \Rightarrow (\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\alpha_1, \alpha_2)\})$ **where** $\sigma' = \{\alpha \mapsto \{\}, \alpha_1 \mapsto \{\}, \alpha_2 \mapsto \{\}\}$

8c. $(\mathbb{C}, \mathbb{K} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\alpha', \{l_i : \mathsf{T}_i\}_{i=1..n})\}, \sigma, \mathbb{SK}) \Rightarrow (\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\alpha', \{l_i : \mathsf{T}_i\}_{i=1..n})\})$ **where** $\sigma' = \{\alpha \mapsto \{l_i : \mathsf{T}_i\}_{i=1..n}, \alpha' \mapsto \{\}\}$

8d. $(\mathbb{C}, \mathbb{K} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\{l_i : \mathsf{T}_i\}_{i=1..n}, \alpha')\}, \sigma, \mathbb{SK}) \Rightarrow (\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \stackrel{\mathrm{k}}{=} (\{l_i : \mathsf{T}_i\}_{i=1..n}, \alpha')\})$ **where** $\sigma' = \{\alpha \mapsto \{l_i : \mathsf{T}_i\}_{i=1..n}, \alpha' \mapsto \{\}\}$

where $\mathbb{SK}$ is a set of solved kind constraints, and $TV(\mathsf{T})$ is the set of type variables in T. This gives a result $(\mathbb{C}', \mathbb{K}', \sigma, \mathbb{SK})$. If $\mathbb{C}' = \emptyset$, then the algorithm returns σ, otherwise it returns an error.

Recall that $\mathsf{T} = \alpha$ is equivalent to $\alpha = \mathsf{T}$, so rule 2 also applies on $\mathbb{C} \cup \{\mathsf{T} = \alpha\}$.

Rule 1 states that if we have a type constraint that is trivially true such as `int` = `int` then we eliminate it. Rule 2 states that if a type variable α is linked to a type T by a type constraint and to a universal kind by a kind constraint, then we apply the substitution of α to T. Rule 3 states that if two type variables are linked together by a type constraint and both are linked to a record kind by kind constraints, then substitute one type variable by the other and create type constraints between the types linked to the common labels. Rule 4 states that if a type variable is linked to a record type by a type constraint and to a record kind by a kind constraint, then substitute the type variable with the record type and create type constraints between the types linked to the common labels. Rule 5 states that if we have a type constraint between two record types with the same labels then create type constraints between the types linked to the labels. Rule 6 states that if we have a type constraint between two function types, then create type constraints between the argument types and the return types. Rule 7 states that if we have a type constraint between two list types, then create a type

constraint between the element types. Rule 8a states that if a type variable is linked to a record union kind of record types, then we substitute the type variable with the record type corresponding to the concatenation of two records that have the corresponding types. Rules 8b, 8c, and 8d handle polymorphic record union kinds, and state that if a type variable α is linked to a record union kind of type variables and record types, then we substitute those type variables to the empty record type first, and then we substitute α to the record type corresponding to the union. As explained at the beginning of this section, this is sufficient for us since we do not need precise types for polymorphic record concatenation. These rules are last in the priority list since we only want to lose precision as a last resort, after we make sure no information on that record concatenation is available. For instance, the expression

$$\mathbf{fun}(x, y) \rightarrow x \bowtie y$$

is typed $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha$ with a kind constraint $\alpha \overset{\text{k}}{=} (\alpha_1, \alpha_2)$ where α_1 and α_2 are the types given to the variables x and y respectively. Then, our unification algorithm uses rule 8b to substitute α_1 and α_2 to the type $\{\}$, and then rule 8a to substitute α to the type $\{\}$. Thus, this expression is given the type $\{\} \rightarrow \{\} \rightarrow \{\}$ which is not the most general type, but is good enough for us since this QIR function that is not applied to anything is not of much interest to a framework that aims to send queries to databases.

Our rules 1 to 5 of our algorithm are exactly the same as the rules (I) to (V) in [Oho95]. Our rule 6 is the same as their rule (IX). Our rule 7 is similar to rule 6. and does not bring any new difficulty for proving properties on our algorithm. Their rules (VI) to (VIII) are absent as they do not apply to us. Only our rules 8a, 8b, 8c, and 8d dealing with record union kinds require extra work to prove our unification algorithm correct.

We now go through a few examples (skipping $\mathbb{SK}$ for presentation). Recall Example 4.1 from Chapter 4:

$$(\mathbf{fun}(r) \rightarrow r \cdot id)\ \{\, id : 1, name : \texttt{"Maggie"} \,\}$$

For which the typing algorithm for MEM gave us:

$$\begin{array}{l}\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} (\mathbf{fun}(r) \rightarrow r \cdot id)\ \{\, id : 1, name : \texttt{"Maggie"} \,\} : \\ \alpha_1, (\{\alpha_2 \rightarrow \alpha_4 = \{id : \texttt{int}, name : \texttt{string}\} \rightarrow \alpha_1, \alpha_3 = \alpha_2\}, \\ \{\alpha_1 \overset{\text{k}}{=} \texttt{U}, \alpha_2 \overset{\text{k}}{=} \texttt{U}, \alpha_3 \overset{\text{k}}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{\text{k}}{=} \texttt{U}\})\end{array}$$

We can apply our unification algorithm to those constraints in order to generate a substitution to apply to the type α_1 returned by our typing algorithm. $\texttt{unify}(\{\alpha_2 \rightarrow \alpha_4 = \{id : \texttt{int}, name : \texttt{string}\} \rightarrow \alpha_1, \alpha_3 = \alpha_2\}, \{\alpha_1 \overset{\text{k}}{=} \texttt{U}, \alpha_2 \overset{\text{k}}{=}$

$\mathtt{U}, \alpha_3 \overset{k}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{k}{=} \mathtt{U}\})$ gives us:

$$
\begin{array}{ll}
 & (\{\alpha_2 \to \alpha_4 = \{id : \mathtt{int}, name : \mathtt{string}\} \to \alpha_1, \alpha_3 = \alpha_2\}, \\
 & \{\alpha_1 \overset{k}{=} \mathtt{U}, \alpha_2 \overset{k}{=} \mathtt{U}, \alpha_3 \overset{k}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{k}{=} \mathtt{U}\}, \emptyset) \\
 & \\
\Rightarrow^6 & (\{\alpha_2 = \{id : \mathtt{int}, name : \mathtt{string}\}, \alpha_4 = \alpha_1, \alpha_3 = \alpha_2\}, \\
 & \{\alpha_1 \overset{k}{=} \mathtt{U}, \alpha_2 \overset{k}{=} \mathtt{U}, \alpha_3 \overset{k}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{k}{=} \mathtt{U}\}, \emptyset) \\
 & \\
 & (\{\alpha_4 = \alpha_1, \alpha_3 = \{id : \mathtt{int}, name : \mathtt{string}\}\}, \\
\Rightarrow^2 & \{\alpha_1 \overset{k}{=} \mathtt{U}, \alpha_3 \overset{k}{=} \{\{id : \alpha_4\}\}, \alpha_4 \overset{k}{=} \mathtt{U}\}, \\
 & \{\alpha_2 \mapsto \{id : \mathtt{int}, name : \mathtt{string}\}\}) \\
 & \\
 & (\{\alpha_3 = \{id : \mathtt{int}, name : \mathtt{string}\}\}, \\
\Rightarrow^2 & \{\alpha_1 \overset{k}{=} \mathtt{U}, \alpha_3 \overset{k}{=} \{\{id : \alpha_1\}\}\}, \\
 & \{\alpha_2 \mapsto \{id : \mathtt{int}, name : \mathtt{string}\}, \alpha_4 \mapsto \alpha_1\}) \\
 & \\
 & (\{\alpha_1 = \mathtt{int}\}, \\
\Rightarrow^4 & \{\alpha_1 \overset{k}{=} \mathtt{U}\}, \\
 & \{\alpha_2 \mapsto \{id : \mathtt{int}, name : \mathtt{string}\}, \alpha_4 \mapsto \alpha_1, \\
 & \ \alpha_3 \mapsto \{id : \mathtt{int}, name : \mathtt{string}\}\}) \\
 & \\
 & (\emptyset, \\
\Rightarrow^2 & \emptyset, \\
 & \{\alpha_2 \mapsto \{id : \mathtt{int}, name : \mathtt{string}\}, \alpha_4 \mapsto \alpha_1, \\
 & \ \alpha_3 \mapsto \{id : \mathtt{int}, name : \mathtt{string}\}, \alpha_1 \mapsto \mathtt{int}\})
\end{array}
$$

which gives us the type $\{\ldots, \alpha_1 \mapsto \mathtt{int}\}\alpha_1 = \mathtt{int}$ as expected.

Our second example:

$$\mathtt{Filter}\langle \mathbf{fun}(r) \to r \cdot id \leq 2 \mid [\,\{id : 1\}, \{id : 2\}, \{id : 3\}\,]\rangle$$

gave us the type and constraints:

$$
\begin{array}{l}
\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Filter}\langle \mathbf{fun}(r) \to r \cdot id \leq 2 \mid [\,\{id : 1\}\,]\rangle : \alpha_7\ \mathbf{list}, (\\
\quad \{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \mathtt{bool} = \alpha_6 \to \alpha_3, \alpha_3 = \mathtt{int} \to \alpha_2, \alpha_8 = \{id : \mathtt{int}\}, \\
\qquad \alpha_8\ \mathbf{list} = \alpha_9\ \mathbf{list}, \alpha_1 \to \alpha_2 = \alpha_7 \to \mathtt{bool}, \alpha_7\ \mathbf{list} = \alpha_8\ \mathbf{list}\}, \\
\quad \{\alpha_1 \overset{k}{=} \mathtt{U}, \alpha_2 \overset{k}{=} \mathtt{U}, \alpha_3 \overset{k}{=} \mathtt{U}, \alpha_4 \overset{k}{=} \mathtt{U}, \alpha_5 \overset{k}{=} \{\{id : \alpha_6\}\}, \alpha_6 \overset{k}{=} \mathtt{U}, \alpha_7 \overset{k}{=} \mathtt{U}, \\
\qquad \alpha_8 \overset{k}{=} \mathtt{U}, \alpha_9 \overset{k}{=} \mathtt{U}\} \\
)
\end{array}
$$

for which $\texttt{unify}(\mathbb{C}, \mathbb{K})$ gives us:

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \alpha_3, \alpha_3 = \texttt{int} \to \alpha_2, \alpha_8 = \{id : \texttt{int}\}, \\
 & \quad \alpha_8\ \textbf{list} = \alpha_9\ \textbf{list}, \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \alpha_8\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_3 \stackrel{\text{k}}{=} \texttt{U}, \alpha_4 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \alpha_6\}\}, \alpha_6 \stackrel{\text{k}}{=} \texttt{U}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \\
 & \quad \alpha_8 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \emptyset) \\
 & \\
 & (\{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \texttt{int} \to \alpha_2, \alpha_8 = \{id : \texttt{int}\}, \\
 & \quad \alpha_8\ \textbf{list} = \alpha_9\ \textbf{list}, \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \alpha_8\ \textbf{list}\}, \\
\Rightarrow^2 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_4 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \alpha_6\}\}, \alpha_6 \stackrel{\text{k}}{=} \texttt{U}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \\
 & \quad \alpha_8 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \alpha_2\}) \\
 & \\
 & (\{\alpha_5 = \alpha_1, \alpha_4 \to \alpha_4 \to \texttt{bool} = \alpha_6 \to \texttt{int} \to \alpha_2, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^2 & \quad \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_4 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \alpha_6\}\}, \alpha_6 \stackrel{\text{k}}{=} \texttt{U}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \alpha_2, \alpha_8 \mapsto \{id : \texttt{int}\}\}) \\
 & \\
 & (\{\alpha_5 = \alpha_1, \alpha_4 = \alpha_6, \alpha_4 \to \texttt{bool} = \texttt{int} \to \alpha_2, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^6 & \quad \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_4 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \alpha_6\}\}, \alpha_6 \stackrel{\text{k}}{=} \texttt{U}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \alpha_2, \alpha_8 \mapsto \{id : \texttt{int}\}\}) \\
 & \\
 & (\{\alpha_5 = \alpha_1, \alpha_6 \to \texttt{bool} = \texttt{int} \to \alpha_2, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^2 & \quad \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \alpha_6\}\}, \alpha_6 \stackrel{\text{k}}{=} \texttt{U}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \alpha_2, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \alpha_6\}) \\
 & \\
 & (\{\alpha_5 = \alpha_1, \alpha_6 = \texttt{int}, \texttt{bool} = \alpha_2, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^6 & \quad \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \alpha_6\}\}, \alpha_6 \stackrel{\text{k}}{=} \texttt{U}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \alpha_2, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \alpha_6\}) \\
 & \\
 & (\{\alpha_5 = \alpha_1, \texttt{bool} = \alpha_2, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^2 & \quad \alpha_1 \to \alpha_2 = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_2 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \alpha_2, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_1, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^2 & \quad \alpha_1 \to \texttt{bool} = \alpha_7 \to \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_1, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \\
\Rightarrow^6 & \quad \alpha_1 = \alpha_7, \texttt{bool} = \texttt{bool}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_1, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \alpha_1 = \alpha_7, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
\Rightarrow^1 & \{\alpha_1 \stackrel{\text{k}}{=} \texttt{U}, \alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_7, \{id : \texttt{int}\}\ \textbf{list} = \alpha_9\ \textbf{list}, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
\Rightarrow^2 & \{\alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \alpha_7\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_7, \{id : \texttt{int}\} = \alpha_9, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
\Rightarrow^7 & \{\alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}, \alpha_9 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \alpha_7\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_7, \alpha_7\ \textbf{list} = \{id : \texttt{int}\}\ \textbf{list}\}, \\
\Rightarrow^2 & \{\alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \alpha_7, \alpha_9 \mapsto \{id : \texttt{int}\}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \alpha_7, \alpha_7 = \{id : \texttt{int}\}\}, \\
\Rightarrow^7 & \{\alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}, \alpha_7 \stackrel{\text{k}}{=} \texttt{U}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \alpha_7, \alpha_9 \mapsto \{id : \texttt{int}\}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_5 = \{id : \texttt{int}\}\}, \\
\Rightarrow^2 & \{\alpha_5 \stackrel{\text{k}}{=} \{\{id : \texttt{int}\}\}\}, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \{id : \texttt{int}\}, \alpha_9 \mapsto \{id : \texttt{int}\}, \alpha_7 \mapsto \{id : \texttt{int}\}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\texttt{int} = \texttt{int}\}, \\
\Rightarrow^2 & \emptyset, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \{id : \texttt{int}\}, \alpha_9 \mapsto \{id : \texttt{int}\}, \alpha_7 \mapsto \{id : \texttt{int}\}, \alpha_5 \mapsto \{id : \texttt{int}\}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\emptyset, \\
\Rightarrow^1 & \emptyset, \\
 & \{\alpha_3 \mapsto \texttt{int} \to \texttt{bool}, \alpha_8 \mapsto \{id : \texttt{int}\}, \alpha_4 \mapsto \texttt{int}, \alpha_6 \mapsto \texttt{int}, \alpha_2 \mapsto \texttt{bool}, \\
 & \quad \alpha_1 \mapsto \{id : \texttt{int}\}, \alpha_9 \mapsto \{id : \texttt{int}\}, \alpha_7 \mapsto \{id : \texttt{int}\}, \alpha_5 \mapsto \{id : \texttt{int}\}\})
\end{array}
$$

which gives us the type $\{\ldots, \alpha_7 \mapsto \{id : \texttt{int}\}, \ldots\}\alpha_7 = \{id : \texttt{int}\}$ as expected.

Finally, our last example:

$$(\{\, x : 1, y : 2 \,\} \bowtie \{\, z : \texttt{true} \,\}) \cdot x$$

was typed as:

$$
\begin{array}{l}
\emptyset \vdash^{\mathcal{A}}_{\mathsf{MEM}} (\{\, x : 1, y : 2 \,\} \bowtie \{\, z : \texttt{true} \,\}) \cdot x : \alpha_3, (\{\alpha_2 = \alpha_1\}, \\
\{\alpha_1 \stackrel{\texttt{k}}{=} (\{x : \texttt{int}, y : \texttt{int}\}, \{z : \texttt{bool}\}), \alpha_2 \stackrel{\texttt{k}}{=} \{\{x : \alpha_3\}\}, \alpha_3 \stackrel{\texttt{k}}{=} \texttt{U}\})
\end{array}
$$

for which $\texttt{unify}(\mathbb{C}, \mathbb{K})$ gives us:

$$
\begin{array}{ll}
 & (\{\alpha_2 = \alpha_1\}, \\
 & \{\alpha_1 \stackrel{\texttt{k}}{=} (\{x : \texttt{int}, y : \texttt{int}\}, \{z : \texttt{bool}\}), \alpha_2 \stackrel{\texttt{k}}{=} \{\{x : \alpha_3\}\}, \alpha_3 \stackrel{\texttt{k}}{=} \texttt{U}\}, \emptyset)
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_2 = \{x : \texttt{int}, y : \texttt{int}, z : \texttt{bool}\}\}, \\
\Rightarrow^{8a} & \{\alpha_2 \stackrel{\texttt{k}}{=} \{\{x : \alpha_3\}\}, \alpha_3 \stackrel{\texttt{k}}{=} \texttt{U}\}, \\
 & \{\alpha_1 \mapsto \{x : \texttt{int}, y : \texttt{int}, z : \texttt{bool}\}\})
\end{array}
$$

$$
\begin{array}{ll}
 & (\{\alpha_3 = \texttt{int}\}, \\
\Rightarrow^4 & \{\alpha_3 \stackrel{\texttt{k}}{=} \texttt{U}\}, \\
 & \{\alpha_1 \mapsto \{x : \texttt{int}, y : \texttt{int}, z : \texttt{bool}\}, \alpha_2 \mapsto \{x : \texttt{int}, y : \texttt{int}, z : \texttt{bool}\}\})
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow^2 & (\emptyset, \emptyset, \\
 & \{\alpha_1 \mapsto \{x : \texttt{int}, y : \texttt{int}, z : \texttt{bool}\}, \alpha_2 \mapsto \{x : \texttt{int}, y : \texttt{int}, z : \texttt{bool}\}, \alpha_3 \mapsto \texttt{int}\})
\end{array}
$$

which gives us the type $\{\ldots, \alpha_3 \mapsto \texttt{int}\}\alpha_3 = \texttt{int}$ as expected.

Note that the type inference problem has been proven to be NP-complete for the relational algebra [Van05, VdBW02] and for the Nested Relational Calculus [BV07]. Thus, we know that our algorithm is NP.

We next give some definitions adapted from [Oho95] before proving the validity of our unification algorithm.

Definition 5.12 (Well-formed kind constraint)**.** A kind constraint $\mathbb{K}$ is *well-formed* if and only if $TV(\texttt{img}(\mathbb{K})) \subseteq \texttt{dom}(\mathbb{K})$.

For instance, $\{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \{\{l : \alpha_1\}\}\}$ is well-formed, but $\{\alpha_2 \overset{\mathrm{k}}{=} \{\{l : \alpha_1\}\}\}$ is not.

Definition 5.13 (Kinded substitution)**.** A *kinded substitution* is a pair $(\mathbb{K}, \sigma)$ of a kind constraint $\mathbb{K}$ and a substitution σ such that $TV(\sigma) \subseteq \mathtt{dom}(\mathbb{K})$.

Definition 5.14 (Kind of a QIR type)**.** A QIR type T has a kind k under $\mathbb{K}$ if the judgment $\mathbb{K} \overset{\mathrm{k}}{\vdash} \mathsf{T} : k$ is derivable from the following rules:

- $\mathbb{K} \overset{\mathrm{k}}{\vdash} \mathsf{T} : \mathtt{U}$ if $TV(\mathsf{T}) \subseteq \mathtt{dom}(\mathbb{K})$
- $\mathbb{K} \cup \{\alpha \overset{\mathrm{k}}{=} k\} \overset{\mathrm{k}}{\vdash} \alpha : k$
- $\mathbb{K} \cup \{\alpha \overset{\mathrm{k}}{=} \{\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, \ldots\}\}\} \overset{\mathrm{k}}{\vdash} \alpha : \{\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\}\}$
- $\mathbb{K} \overset{\mathrm{k}}{\vdash} \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, \ldots\} : \{\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\}\}$
  if $TV(\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, \ldots\}) \subseteq \mathtt{dom}(\mathbb{K})$
- $\mathbb{K} \overset{\mathrm{k}}{\vdash} \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\} : (\{l_i : \mathsf{T}_i\}, \{l_j : \mathsf{T}_j\})$
  where $\{l_i\} \cup \{l_j\} = \{l_1, \ldots, l_n\}$ and $\{l_i\} \cap \{l_j\} = \emptyset$,
  and if $TV(\{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n\}) \subseteq \mathtt{dom}(\mathbb{K})$

Definition 5.15 (Respect of a kind constraint)**.** A kinded substitution $(\mathbb{K}, \sigma)$ *respects a kind constraint* $\mathbb{K}'$ if $\forall \alpha \in \mathtt{dom}(\mathbb{K}').\mathbb{K} \overset{\mathrm{k}}{\vdash} \sigma\alpha : \sigma(\mathbb{K}'(\alpha))$.

Definition 5.16 (Unifier)**.** A kinded substitution $(\mathbb{K}', \sigma)$ is a *unifier* of a pair of sets of constraints $(\mathbb{C}, \mathbb{K})$ if it respects $\mathbb{K}$ and if σ satisfies $\mathbb{C}$.

Definition 5.17 (Most general unifier)**.** A kinded substitution $(\mathbb{K}', \sigma)$ is a *most general unifier* of a pair of sets of constraints $(\mathbb{C}, \mathbb{K})$ if it is a unifier of $(\mathbb{C}, \mathbb{K})$ and if for any unifier $(\mathbb{K}'', \sigma')$ of $(\mathbb{C}, \mathbb{K})$ there exists some substitution σ'' such that $(\mathbb{K}'', \sigma'')$ respects $\mathbb{K}'$ and $\sigma' = \sigma'' \circ \sigma$.

We next prove that our unification algorithm terminates, and doing so it returns a unifier or fails. Rule 8a, 8b, 8c, and 8d prevents our algorithm to return a most general unifier as a direct consequence of our design choice to not have precise types to express polymorphic record concatenation, but our property of soundness is still valid on any substitution.

Theorem 5.3. *The algorithm* $\mathtt{unify}(\mathbb{C}, \mathbb{K})$ *terminates.*

Proof. Let $\mathcal{N}(\mathbb{C})$ be the number of type constructors in $\mathbb{C}$. The measure of the lexicographical pair $(|\mathtt{dom}(\mathbb{K})|, \mathcal{N}(\mathbb{C}))$ decreases with each rule. Indeed, the domain of $\mathbb{K}$ strictly decreases in rules 2, 3, 4, 8a, 8b, 8c, and 8d, and it stays the same while the number of type constructors in $\mathbb{C}$ strictly decreases in rules 1, 5, 6, and 7. □

Theorem 5.4. *The algorithm* $\mathtt{unify}(\mathbb{C}, \mathbb{K})$ *computes a unifier for the set of constraints* $\mathbb{C}$ *and* $\mathbb{K}$ *if one exists, and fails otherwise.*

Proof. We follow the proof of Theorem 3.4.1 in [Oho95].

The main part of this proof is to show that if $\mathtt{unify}(\mathbb{C}, \mathbb{K})$ returns a substitution then it is a most general unifier for $\mathbb{C}$ and $\mathbb{K}$.

It is easily verified that each transformation rule preserves the following property on the result of its application to $(\mathbb{C}, \mathbb{K}, \sigma, \mathbb{SK})$:

(1) $\mathbb{K}$ and $\mathbb{K} \cup \mathbb{SK}$ are well-formed kind constraints; $TV(\mathbb{C}) \subseteq \mathtt{dom}(\mathbb{K})$; $\mathtt{dom}(\mathbb{K}) \cap \mathtt{dom}(\mathbb{SK}) = \emptyset$; and $\mathtt{dom}(\mathbb{SK}) = \mathtt{dom}(\sigma)$.

We establish that if property (1) holds for $(\mathbb{C}, \mathbb{K}, \sigma, \mathbb{SK})$, then each transformation rule also preserves the following properties on its result:

(2) For any kinded substitution $(\mathbb{K}_0, \sigma_0)$, if $(\mathbb{K}_0, \sigma_0)$ respects $\mathbb{K}$ and σ_0 satisfies $\mathbb{C} \cup \sigma$, then $(\mathbb{K}_0, \sigma_0)$ respects $\mathbb{SK}$.

(3) The set of unifiers of $(\mathbb{K}' \cup \mathbb{SK}', \mathbb{C}' \cup \sigma')$ is included in the set of unifiers of $(\mathbb{K} \cup \mathbb{SK}, \mathbb{C} \cup \sigma)$.

In [Oho95], property (3) states that the sets of unifier are equal, but this property is false in our case because of the rules 6., 8a., 8b., 8c., and 8d.

As stated before, only our rules on record union kinds add a significant difference to the algorithm presented in [Oho95]. Therefore, we refer the reader to the cited paper for most of the proof, and we only discuss this particular rule which is proven in a roughly similar way as for rule 2.

Rule 8a: $(\mathbb{C}, \mathbb{K} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\}, \sigma, \mathbb{SK}) \Rightarrow$
$(\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\})$

where $\sigma' = \{\alpha \mapsto \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\}\}$

Property (2). Let $(\mathbb{K}_0, \sigma_0)$ be a kinded substitution such that $(\mathbb{K}_0, \sigma_0)$ respects $\sigma'\mathbb{K}$ and σ_0 satisfies $\sigma'\mathbb{C} \cup (\sigma'\sigma \circ \sigma')$. Then $\sigma_0 = \sigma'_0 \circ \sigma'$, and σ'_0 satisfies $\mathbb{C} \cup \sigma$. Since $TV(\mathsf{T} = \{l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\}) \subseteq \mathbb{K}$, then $TV(\sigma'_0\alpha) \subseteq \mathbb{K}_0$ and $\mathbb{K}_0 \overset{\mathrm{k}}{\vdash} \sigma'_0\alpha : (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})$. Therefore, $(\mathbb{K}_0, \sigma_0)$ respects $\mathbb{K} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\}$. Then by property (2) of the premise of the rule, $(\mathbb{K}_0, \sigma_0)$ respects $\mathbb{SK}$, and thus $(\mathbb{K}_0, \sigma_0)$ respects $\sigma'\mathbb{SK} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\}$.

Property (3). Let σ_0 be any substitution. If σ_0 satisfies $\sigma'\mathbb{C} \cup (\sigma'\sigma \circ \sigma')$ then it satisfies $\mathbb{C} \cup \sigma$. Let $\mathbb{K}_0$ be any kind assignment such that $(\mathbb{K}_0, \sigma_0)$ is a kinded substitution. Suppose σ_0 satisfies $\sigma'\mathbb{C} \cup (\sigma'\sigma \circ \sigma')$. Then since $\sigma_0 = \sigma'_0 \circ \sigma'$, if $(\mathbb{K}_0, \sigma_0)$ respects $\sigma'\mathbb{K} \cup \sigma'\mathbb{SK} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\}$, then it respects $\mathbb{K} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\} \cup \mathbb{SK}$. Thus, if $(\mathbb{K}_0, \sigma_0)$ is a unifier for $(\sigma'\mathbb{K} \cup \sigma'\mathbb{SK} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\}, \sigma'\mathbb{C} \cup (\sigma'\sigma \circ \sigma'))$ it is a unifier for $(\mathbb{K} \cup \{\alpha \overset{\mathrm{k}}{=} (\{l_1 : \mathsf{T}_1, \ldots, l_m : \mathsf{T}_m, l'_1 : \mathsf{T}'_1, \ldots, l'_o : \mathsf{T}'_o\}, \{l_{m+1} : \mathsf{T}_{m+1}, \ldots, l_n : \mathsf{T}_n, l'_1 : \mathsf{T}''_1, \ldots, l'_o : \mathsf{T}''_o\})\} \cup \mathbb{SK}, \mathbb{C} \cup \sigma)$.

Rule 8b, 8c, 8d: By the same arguments as for rule 8a.

We can then conclude the proof following [Oho95]. □

Without our rules for record concatenation however, our unification algorithm returns a most general unifier.

Theorem 5.5. *The algorithm* $\mathtt{unify}(\mathbb{C}, \mathbb{K})$ *without rules 8a, 8b, 8c, and 8d computes a most general unifier for the set of constraints* $\mathbb{C}$ *and* $\mathbb{K}$ *if one exists, and fails otherwise.*

Proof. Directly using the proof in [Oho95]. □

5.4 Specific typing algorithm for SQL

The specific type system for SQL is very similar to the specific type system for MEM. The only non-algorithmic rules are the function rule; the basic operators rule; and the subsumption rule. We can apply the same logic as for the specific typing algorithm for MEM to solve the issues brought by these rules. However,

the rules of the specific type system for SQL also restrict the types of the premises to relational types. For instance:

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q_1 : R \to \texttt{bool} \quad \Gamma \vdash_{\mathsf{SQL}} q_2 : R\ \textbf{list}}{\Gamma \vdash_{\mathsf{SQL}} \texttt{Filter}\langle q_1 \mid q_2\rangle : R\ \textbf{list}}$$

This `Filter` rule for SQL requires the argument of the configuration to be of flat record type, and the data argument to be of flat record list type.

to represent those constraints, we add two new types of kinds: a *basic type kind* `B` which indicates that a type is a basic type, and a *flat record kind* `R` which indicates that a type is a record type that contains only basic types. For instance, the conditional rule is then written the exact same way as for MEM, but the kind constraint on α_2 is written as $\alpha_2 \overset{\text{k}}{=} \texttt{B}$ since α_2 represents the type of both branches of the conditional expression and, in the type system of SQL, these expressions must have a basic type:

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \Gamma \vdash^{\mathcal{A}} q_3 : \mathsf{T}_3, (\mathbb{C}_3, \mathbb{K}_3), \_ \quad \alpha_1 \text{ and } \alpha_2 \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3 : \alpha_2, \left(\begin{array}{l} \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}_1, \alpha_1 = \texttt{bool}, \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}, \\ \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\text{k}}{=} \texttt{U}, \alpha_2 \overset{\text{k}}{=} \texttt{B}\} \end{array} \right)}$$

Similarly, we can use the kind constraint $\alpha \overset{\text{k}}{=} \texttt{R}$ to write the `Filter` rule, since α represents both the type of the argument type of the configuration and the type of the elements of the data argument:

$$\frac{\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_ \quad \Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_ \quad \alpha \text{ fresh}}{\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \texttt{Filter}\langle q_1 \mid q_2\rangle : \alpha\ \textbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \alpha \to \texttt{bool}, \alpha\ \textbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\text{k}}{=} \texttt{R}\})}$$

We can then add the following rules to our algorithm of unification of constraints:

2a. $(\mathbb{C} \cup \{\alpha = \mathsf{B}\}, \mathbb{K} \cup \{\alpha \overset{\text{k}}{=} \texttt{B}\}, \sigma, \mathbb{SK}) \Rightarrow$
$(\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \overset{\text{k}}{=} \texttt{B}\})$
if $\alpha \notin TV(\mathsf{T})$
where $\sigma' = \{\alpha \mapsto \mathsf{B}\}$

2b. $(\mathbb{C} \cup \{\alpha = \mathsf{R}\}, \mathbb{K} \cup \{\alpha \overset{\text{k}}{=} \texttt{R}\}, \sigma, \mathbb{SK}) \Rightarrow$
$(\sigma'\mathbb{C}, \sigma'\mathbb{K}, \sigma'\sigma \circ \sigma', \sigma'\mathbb{SK} \cup \{\alpha \overset{\text{k}}{=} \texttt{R}\})$
if $\alpha \notin TV(\mathsf{T})$
where $\sigma' = \{\alpha \mapsto \mathsf{R}\}$

This completes our design of typing algorithms for QIR. In the next chapter, we show how to use our type systems to translate QIR expressions to query languages, to obtain formal guarantees on these translations, and to safely normalize QIR queries.

Chapter 6

Type-oriented evaluation

In this chapter, we make use of the type systems defined in Chapter 4 and 5 for the translation and evaluation of queries. We use the information on queries provided by those type systems to guide the normalization and the translation of queries, as well as prove some useful properties in particular a safety guarantee for the translation of QIR expressions into SQL.

6.1 Translation into database languages

As we saw in Section 3.3, every database must define a driver that allows it to interface with QIR, and this driver includes a translation from QIR into the database language. When the QIR expression to be translated is a query targeting a single database, then the translation process is straightforward, as we simply have to call the translation defined by the corresponding database driver. For instance,

$$\texttt{Filter}\langle \mathbf{fun}(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle \rangle$$

can be directly translated to

```
SELECT * FROM employee AS r WHERE r.salary > 2500
```

However, as explained in Section 3.4, a QIR expression might not be translatable directly to one database expression. This can happen for a number of reasons such as targeting several databases or using a feature unsupported by the targeted database such as an unsupported operator. In this case, we have to translate as much as we can into the languages of the targeted databases, and leave the untranslatable parts to the MEM database. For instance, consider this QIR query:

$$\begin{aligned}&\texttt{Join}\langle \mathbf{fun}(e,t) \rightarrow e \bowtie t, \mathbf{fun}(e,t) \rightarrow e \cdot \mathit{teamid} = t \cdot \mathit{teamid} \mid \\ &\quad \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"} \rangle \rangle\end{aligned}$$

which applies the `Join` operator between the table `"employee"` stored in a PostgreSQL database with the table `"team"` stored in a HBase database. The simplest translation for this query is:

$$\texttt{Join}\langle \mathbf{fun}(o,p) \rightarrow \{oid : oid, pname : p.name\}, \mathbf{fun}(o,p) \rightarrow o \cdot pid = p \cdot pid \mid \texttt{eval}^{\text{PostgreSQL}}(\texttt{SELECT * FROM EMPLOYEE}), \texttt{eval}^{\text{HBase}}(\texttt{scan 'team'})\rangle$$

in which the two `From` subqueries are correctly translated to the respective languages of the targeted databases, then leaves the `Join` operation itself in QIR form to be evaluated in MEM.

Similarly to this example, our translation must translate as much of the QIR expressions as possible using the different database language translations available, and default to MEM if none of these translations succeed. Additionally, our translation must be seamlessly extendable with new translations from database drivers.

In this section, we define a *generic* translation from QIR to database expressions by making use of the database drivers translations $\overrightarrow{\text{EXP}}$ that we call *specific* translations. We also define a specific translation for SQL.

6.1.1 Specific and generic translations

Let us first define specific translations.

Definition 6.1 (Specific translation)**.** The specific translation $\overrightarrow{\text{EXP}}^{\mathcal{D}}$ from QIR into a database language $\mathcal{D}$ is defined by the judgment $q \overset{\mathcal{D}}{\rightsquigarrow} e$ where $q \in \mathsf{E}_{\text{QIR}}$, $e \in \mathsf{E}_{\mathcal{D}} \cup \{\Omega\}$, and Ω is an error.

This definition is similar to the definition of the translation from Definition 3.8, adding the judgment $q \overset{\mathcal{D}}{\rightsquigarrow} e$ that we use to define specific translations with inference rules. Next, we define the generic translation of QIR.

Definition 6.2 (Generic translation)**.** The generic translation of QIR is defined by the judgment $q \rightsquigarrow e$ where $q, e \in \mathsf{E}_{\mathsf{QIR}}$. The set of inference rules used to derive this judgment are:

$$\text{(direct)}\quad \frac{\Gamma \vdash q : T, \mathcal{D} \quad q \overset{\mathcal{D}}{\rightsquigarrow} e}{q \rightsquigarrow \texttt{eval}^{\mathcal{D}}(e)}\ \mathcal{D} \neq \mathsf{MEM}$$

$$\text{(propagate)}\quad \frac{\Gamma \vdash q : T, \mathsf{MEM} \quad q_i \rightsquigarrow \texttt{eval}^{\mathcal{D}_i}(e_i) \quad q_j \rightsquigarrow \texttt{eval}^{\mathsf{MEM}}(e_j)}{q \rightsquigarrow \texttt{eval}^{\mathsf{MEM}}(q\{q_i/\texttt{eval}^{\mathcal{D}_i}(e_i), q_j/e_j\})}\ \begin{array}{l}\forall i.\mathcal{D}_i \neq \mathsf{MEM}\\ \{q_i\} \cup \{q_j\} = \mathfrak{C}(q)\\ \{q_i\} \cap \{q_j\} = \emptyset\end{array}$$

$$\text{(best-effort-direct)}\quad \frac{\forall q_i \in \mathfrak{C}(q).q_i \rightsquigarrow \texttt{eval}^{\mathcal{D}_i}(e_i) \quad q \overset{\mathcal{D}}{\rightsquigarrow} e}{q \rightsquigarrow \texttt{eval}^{\mathcal{D}}(e)}\ \begin{array}{l}\{\mathcal{D}_i\} = \{\mathcal{D}, \mathsf{MEM}\}\\ \mathcal{D} \neq \mathsf{MEM}\\ e \neq \Omega\end{array}$$

$$\text{(best-effort-propagate)}\quad \frac{q_i \rightsquigarrow \texttt{eval}^{\mathcal{D}_i}(e_i) \quad q_j \rightsquigarrow \texttt{eval}^{\mathsf{MEM}}(e_j)}{q \rightsquigarrow \texttt{eval}^{\mathsf{MEM}}(q\{q_i/\texttt{eval}^{\mathcal{D}_i}(e_i), q_j/e_j\})}\ \begin{array}{l}\forall i.\mathcal{D}_i \neq \mathsf{MEM}\\ \{q_i\} \cup \{q_j\} = \mathfrak{C}(q)\\ \{q_i\} \cap \{q_j\} = \emptyset\end{array}$$

where the order of priority of application of the rules is from top to bottom.

The goal of the generic translation is to produce a QIR expression where as many subterms as possible have been translated to native database queries. It makes use of the $\texttt{eval}^{\mathcal{D}}$ special operators, mentioned in the extended semantics of QIR in Section 3.3, to mark a translated query for evaluation in a database $\mathcal{D}$.

Rule (direct) states that given a QIR expression, if there exists a database $\mathcal{D}$ distinct from MEM such that the expression can be typed for $\mathcal{D}$ by the generic type system, then the generic translation returns the translation of the expression by $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}$ from the driver of $\mathcal{D}$. In Section 6.2, we will show that our type system for SQL gives us the guarantee that under some hypotheses the translation $\overset{\mathsf{SQL}}{\rightsquigarrow}$ will always succeed, in which case we can omit to check that $e \neq \Omega$ in this rule. Rule (propagate) states that if the QIR expression could only be typed for MEM by the generic type system, then the generic translation applies itself recursively to all children of the QIR expression, thus marking their translations for evaluation to the targeted databases and translation of the results back into QIR, then it integrates them back in place of the children in the original expression. Finally, the generic translation returns the translation of the new expression by $\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}$.

These two rules can only be applied on typeable expressions. For non-typeable expressions, the generic translation has to fallback on calling the translations $\overrightarrow{\mathsf{EXP}}$ without being entirely guided by types. Rule (best-effort-direct) calls the generic translation on the children, then if they were all translated to be evaluated by the same database $\mathcal{D} \neq \mathsf{MEM}$ or by MEM, it attempts to translate the entire

expression using $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}$. The downside of this rule is that it calls the translation for every child, and uses the results only to check there was no error, as it then calls the translation on the entire expression without using the results. Thankfully, if the translation $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}$ is compositional, which as we will see in Section 6.1.2 is possible even for SQL, it can implement a cache of its translations to avoid translating the same expression more than once. If even this fails, then rule (best-effort-propagate) applies the same treatment as (propagate) but without the call to the type system. The difference between those two very similar rules is that (propagate) can proceed with recursive calls with guaranteed success thanks to the type system, while (best-effort-propagate) might fail on one of its subexpressions.

We now go through a few examples. In our first query from the beginning of the chapter:

$$\texttt{Filter}\langle \mathbf{fun}\,(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle \rangle$$

the From is typed by our SQL type system as:

$$A = \frac{\emptyset \vdash_{\mathsf{SQL}} \texttt{"employee"} : \texttt{string}}{\emptyset \vdash_{\mathsf{SQL}} \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle : \{\mathit{salary} : \texttt{int}, \ldots\}\ \mathbf{list}}$$

and the whole query as:

$$\dfrac{\dfrac{\dfrac{\ldots}{\{r : \{\mathit{salary} : \texttt{int}, \ldots\}\} \vdash_{\mathsf{SQL}} r \cdot \mathit{salary} > 2500 : \texttt{bool}} \; A}{\emptyset \vdash_{\mathsf{SQL}} \begin{array}{l}\texttt{Filter}\langle \mathbf{fun}\,(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \\ \qquad \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle \rangle\end{array} : \{\mathit{salary} : \texttt{int}, \ldots\}\ \mathbf{list}}}{\emptyset \vdash \begin{array}{l}\texttt{Filter}\langle \mathbf{fun}\,(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \\ \qquad \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle \rangle\end{array} : \{\mathit{salary} : \texttt{int}, \ldots\}\ \mathbf{list}, \mathsf{SQL}}$$

Therefore, we can apply our (direct) rule:

$$\dfrac{\overset{\text{(direct)}}{\emptyset \vdash \begin{array}{l}\texttt{Filter}\langle \mathbf{fun}\,(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \\ \qquad \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle \rangle\end{array} : \{\mathit{salary} : \texttt{int}, \ldots\}\ \mathbf{list}, \mathsf{SQL}}}{q \rightsquigarrow \texttt{eval}^{\mathsf{PostgreSQL}}(\overrightarrow{\mathsf{EXP}}^{\mathsf{PostgreSQL}}(\begin{array}{l}\texttt{Filter}\langle \mathbf{fun}\,(r) \rightarrow r \cdot \mathit{salary} > 2500 \mid \\ \qquad \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle \rangle\end{array}))$$

As for our second query:

$$\begin{array}{l}\texttt{Join}\langle \mathbf{fun}\,(e,t) \rightarrow e \bowtie t, \mathbf{fun}\,(e,t) \rightarrow = \ (e \cdot \mathit{teamid}, t \cdot \mathit{teamid}) \mid \\ \quad \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"} \rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"} \rangle \rangle\end{array}$$

it is typed by MEM and the Froms are typed by their targeted databases:

$$\dfrac{\dfrac{\cdots \quad \begin{array}{l}\emptyset \vdash \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle : \{teamid : \texttt{int}, \ldots\}, \mathsf{PostgreSQL} \\ \emptyset \vdash \texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle : \{teamid : \texttt{int}, \ldots\}, \mathsf{HBase}\end{array}}{\emptyset \vdash_{\mathsf{MEM}} \begin{array}{l}\texttt{Join}\langle \mathbf{fun}(e,t) \to e \bowtie t, \mathbf{fun}(e,t) \to = \ (e \cdot teamid, t \cdot teamid) \mid \\ \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle\rangle\end{array} : \{\ldots\}\ \mathbf{list}}}{\emptyset \vdash \begin{array}{l}\texttt{Join}\langle \mathbf{fun}(e,t) \to e \bowtie t, \mathbf{fun}(e,t) \to = \ (e \cdot teamid, t \cdot teamid) \mid \\ \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle\rangle\end{array} : \{\ldots\}\ \mathbf{list}, \mathsf{MEM}}$$

Therefore, we can apply our (propagate) rule:

$$\dfrac{\begin{array}{l}\text{(propagate)} \\ \emptyset \vdash \begin{array}{l}\texttt{Join}\langle \mathbf{fun}(e,t) \to e \bowtie t, \mathbf{fun}(e,t) \to = \ (e \cdot teamid, t \cdot teamid) \mid \\ \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle\rangle\end{array} : \{\ldots\}\ \mathbf{list}, \mathsf{MEM}\end{array}}{\begin{array}{l}\texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle \rightsquigarrow \texttt{eval}^{\mathsf{PostgreSQL}}(\overrightarrow{\mathsf{EXP}}^{\mathsf{PostgreSQL}}(\texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle)) \\ \texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle \rightsquigarrow \texttt{eval}^{\mathsf{HBase}}(\overrightarrow{\mathsf{EXP}}^{\mathsf{HBase}}(\texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle))\end{array}}$$

$$q \rightsquigarrow \texttt{eval}^{\mathsf{MEM}}(\overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(\begin{array}{l}\texttt{Join}\langle \mathbf{fun}(e,t) \to e \bowtie t, \mathbf{fun}(e,t) \to = \ (e \cdot teamid, t \cdot teamid) \mid \\ \texttt{eval}^{\mathsf{PostgreSQL}}(\overrightarrow{\mathsf{EXP}}^{\mathsf{PostgreSQL}}(\texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle)), \\ \texttt{eval}^{\mathsf{HBase}}(\overrightarrow{\mathsf{EXP}}^{\mathsf{HBase}}(\texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle))\rangle\end{array}))$$

which, as required, translates the Froms into the database languages that correspond to their targeted database, and leaves the evaluation of Join to MEM.

Let us now see examples where the (best-effort) rules are used. For the (best-effort) rules to be necessary, the query must be impossible to type even by MEM, but it must also be translatable. Typically, queries that contain host language expressions have to be translated using the (best-effort) rules. For instance:

$$\texttt{Filter}\langle \mathbf{fun}(r) \to \blacksquare_{\mathsf{Python}}(\gamma, e) \mid \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle\rangle$$

Because of Property 4.3, this query cannot be typed as it contains a host language expression. However, if we assume that the targeted PostgreSQL database can execute Python code (for instance using PL/Python [Pos]), then this query can be translated by $\overrightarrow{\mathsf{EXP}}^{\mathsf{PostgreSQL}}$. Therefore, we can apply our (best-effort-direct) rule:

$$\dfrac{\begin{array}{l}\text{(best-effort-direct)} \\ \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle \rightsquigarrow \texttt{eval}^{\mathsf{PostgreSQL}}(\texttt{SELECT * FROM EMPLOYEE}) \\ q \overset{\mathcal{D}}{\rightsquigarrow} \texttt{SELECT * FROM EMPLOYEE WHERE (SELECT EVAL(}\blacksquare_{\mathsf{Python}}(\gamma, e)\texttt{))}\end{array}}{q \rightsquigarrow \texttt{eval}^{\mathcal{D}}(\texttt{SELECT * FROM EMPLOYEE WHERE (SELECT EVAL(}\blacksquare_{\mathsf{Python}}(\gamma, e)\texttt{)))}}$$

This query is therefore executed entirely by PostgreSQL as desired.

(best-effort-propagate) is used for the same kind of queries as (propagate), but that cannot be typed. For instance, we can add a host language expression to our (propagate) example:

$$\texttt{Join}\langle \mathbf{fun}(e,t) \to e \bowtie t, \mathbf{fun}(e,t) \to \blacksquare_{\mathsf{Python}}(\gamma, e) \mid \\ \texttt{From}\langle \mathsf{PostgreSQL}, \texttt{"employee"}\rangle, \texttt{From}\langle \mathsf{HBase}, \texttt{"team"}\rangle\rangle$$

which makes the `Join` impossible to type, but executed in MEM using (best-effort-propagate).

6.1.2 A specific translation for SQL

As an example of a specific translation, we document how to define a specific translation for SQL.

Definition 6.3 (Specific translation for SQL)**.** 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 to a SQL expression e. The derivation of this judgment is given by the rules in Figure 6.1.

Data operators are translated to their SQL equivalent, for example `Sort` is translated to an `ORDER BY` clause. `Project` returns e_1 `FROM` (e_2) `AS X` since e_1 is expected to be a record that translates to a `SELECT` clause, and basic SQL does not support queries such as `SELECT (SELECT name)FROM (...)`.

Constants are translated using the translation function $\overrightarrow{\mathtt{VAL}}^{\mathsf{SQL}}$ provided by the driver of the SQL database. Conditional expressions are translated to the corresponding `CASE` construct. Host language expressions are evaluated using a function provided by BOLDR that calls the evaluator of the host language in the database.

Although most of these rules look rather straightforward, they contain interesting particularities that come from the fact that SQL is an interesting and particular query language.

Applications are restricted to basic operators supported by SQL applied to their exact expected number of arguments since SQL does not support currying. Figure 6.1 shows examples for $+$ and *sum*. Additionally, basic operators can only appear in applications since they are not first-class expressions. For instance, it is invalid to write `SELECT +` in SQL.

Notice how most rules with children q_i translate them recursively to e_i and return constructions of the form (e_i) `AS X`. There are two reasons for this. First, we need the alias when we translate an operator. For instance, using the rule for `Filter`, the query $\texttt{Filter}\langle \mathbf{fun}(r) \to r \cdot age < 30 \mid \ldots\rangle$ would be translated to

```
SELECT * FROM (...) AS R WHERE (R.age < 30)
```

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

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

(SQL-sum)
$$\frac{q \overset{\text{SQL}}{\rightsquigarrow} e \quad e \neq \Omega}{sum\, q \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT sum}(e)}$$

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

(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-record)
$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 1..n}{\{\, l_i : q_i \,\}_{i=1..n} \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } (e_1) \texttt{ AS L}_1\texttt{, } \ldots\texttt{, } (e_n) \texttt{ AS L}_n}$$

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

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

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

(SQL-rdestr-cplx)
$$\frac{q \overset{\text{SQL}}{\rightsquigarrow} e \quad q \not\equiv x \quad e \neq \Omega \quad \texttt{R fresh}}{q \cdot l \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT R.L FROM } (e) \texttt{ AS R}}$$

Figure 6.1 – Translation from QIR to SQL (part 1 of 2)

(SQL-project)

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

(SQL-from)

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

(SQL-filter)

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

(SQL-join)

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

(SQL-join-noproject)

$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i \neq \Omega \quad i \in 2..4}{\texttt{Join}\langle \textbf{fun}(x,y) \rightarrow x \bowtie y, \textbf{fun}(x,y) \rightarrow q_2 \mid q_3, q_4 \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_3\texttt{) AS X, (}e_4\texttt{) AS Y WHERE (}e_2\texttt{)}}$$

(SQL-group)

$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad q \overset{\text{SQL}}{\rightsquigarrow} e \quad q' \overset{\text{SQL}}{\rightsquigarrow} e' \quad e_i \neq \Omega \quad e \neq \Omega \quad e' \neq \Omega \quad i \in 1..n}{\texttt{Group}\langle \textbf{fun}(x) \rightarrow \{\, l_i : q_i \,\}_{i=1..n}, \textbf{fun}(x) \rightarrow q \mid q' \rangle \overset{\text{SQL}}{\rightsquigarrow} e \texttt{ FROM (}e'\texttt{) AS X GROUP BY L}_1\texttt{, ..., L}_n}$$

(SQL-sort)

$$\frac{q_i \overset{\text{SQL}}{\rightsquigarrow} e_i \quad e_i = \texttt{true or false} \quad i \in 1..n \quad e'_i = \texttt{DESC if } e_i = \texttt{false} \quad q \overset{\text{SQL}}{\rightsquigarrow} e \quad e \neq \Omega}{\texttt{Sort}\langle \textbf{fun}(x) \rightarrow \{\, l_i : q_i \,\}_{i=1..n} \mid q \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e\texttt{) AS X ORDER BY L}_1\ e'_1\texttt{, ..., L}_n\ e'_n}$$

(SQL-limit)

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

(SQL-exists)

$$\frac{q \overset{\text{SQL}}{\rightsquigarrow} e \quad e \neq \Omega}{\texttt{Exists}\langle q \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT EXISTS(}e\texttt{)}}$$

(SQL-host-expr)

$$\frac{}{\blacksquare_{\mathcal{H}}(\gamma, e) \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT BOLDR.EVAL(}\blacksquare_{\mathcal{H}}(\gamma, e)\texttt{)}}$$

Figure 6.1 – Translation from QIR to SQL (part 2 of 2)

which is the correct translation in SQL. But without the alias we get an error since the variable `R` in the `WHERE` clause would then be undefined. The second reason is that SQL imposes this syntax for compositional queries even if the alias is not useful. For example:

```
SELECT * FROM (SELECT 1 AS id)
```

which could be a valid translation of $\{ id : 1 \} :: [\,]$, is not a valid SQL query because subqueries must have an alias, so a syntactically correct version of this query would be

```
SELECT * FROM (SELECT 1 AS id) AS X
```

where `X` is here useless. This is not just true for this example, all of the `TMP` variables in our rules are only there to respect the syntax of SQL. Another syntactic issue of SQL is that only queries can be put in parentheses. Variables and table names put in parentheses are *syntactically* incorrect. Therefore, we duplicate some rules to account for the possibility that children could be variables or table names. Thus, the (SQL-rdestr-simpl) rule deals with variables, and the (SQL-rdestr-cplx) rule deals with all other cases. Additionally, it is not possible to directly represent an empty list, record construction in SQL, or generic list construction in SQL. However, for convenience, we create a (SQL-record) rule that translates to the creation of a table with one row containing the record, as well as a (SQL-lcons) rules that allows adding a record to a list. (SQL-lcons-record) works using the same trick as (SQL-record) to create a list containing one row, then applies the `UNION ALL` operation of SQL that concatenates two tables. This operation is obviously also used for the translation of the list concatenation in (SQL-lconcat).

6.2 Type-safe SQL translation

In this section, we prove that a QIR expression typeable for SQL by the generic type system can always be translated into SQL after normalization. First, we isolate a subset of normal forms which are translatable into SQL.

Definition 6.4 (SQL-compatible normal forms)**.** We define SQL-compatible normal forms v_{SQL} as the restriction of normal forms of Definition 3.12 produced by the following grammar:

$$
\begin{array}{rcl}
s & ::= & r :: [\,] \;\mid\; r :: s \;\mid\; s \mathbin{@} s \\
& \mid & \texttt{Project}\langle \mathbf{fun}^x(x) \rightarrow r \mid s \rangle \\
& \mid & \texttt{From}\langle \mathcal{D}, b \rangle \\
& \mid & \texttt{Filter}\langle \mathbf{fun}^x(x) \rightarrow b \mid s \rangle \\
& \mid & \texttt{Join}\langle \mathbf{fun}^x(x,x) \rightarrow r, \mathbf{fun}^x(x,x) \rightarrow b \mid s, s \rangle \\
& \mid & \texttt{Join}\langle \mathbf{fun}^{\mathtt{f}}(\mathtt{x},\mathtt{y}) \rightarrow \mathtt{x} \bowtie \mathtt{y}, \mathbf{fun}^x(x,x) \rightarrow b \mid s, s \rangle \\
& \mid & \texttt{Group}\langle \mathbf{fun}^x(x) \rightarrow r, \mathbf{fun}^x(x) \rightarrow r \mid s \rangle \\
& \mid & \texttt{Sort}\langle \mathbf{fun}^x(x) \rightarrow r \mid s \rangle \\
& \mid & \texttt{Limit}\langle b \mid s \rangle
\end{array}
$$

$$
\begin{array}{rcl}
r & ::= & x \\
& \mid & \{\, l : b, \ldots, l : b \,\}
\end{array}
\qquad
\begin{array}{rcl}
b & ::= & c \\
& \mid & \mathbf{if}\, b\, \mathbf{then}\, b\, \mathbf{else}\, b \\
& \mid & x \cdot l \\
& \mid & op\ (b, \ldots, b) \\
& \mid & \texttt{Exists}\langle s \rangle
\end{array}
$$

Note that the second `Join` form requires the first configuration to have a precise form, although the variables `f`, `x`, and `y` can have any name, the configuration has to be a function returning the record concatenation of the first variable to the second one. Also, the `Exists` operator is included as a b form since it returns a boolean and not a collection like the other data operators. Now, we prove that if the generic type system returns a type for SQL given a QIR normal form, then this normal form must have a specific syntactic form.

Lemma 6.1 (Relation between SQL types and syntactic forms)**.** Let v be a normal form of QIR and Γ a QIR typing environment such that $\forall x \in \mathtt{dom}(\Gamma).\Gamma(x) \equiv R$, and $\Gamma \vdash v : T, \mathsf{SQL}$, then:

- If $T \equiv B$ then $v \equiv b$
- If $T \equiv R$ then $v \equiv r$
- If $T \equiv R$ **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 $T \equiv R \rightarrow R \rightarrow B$ then $v \equiv \mathbf{fun}^x(x) \rightarrow \mathbf{fun}^x(x) \rightarrow b$

- If $T \equiv R \to R \to R$ then $v \equiv \mathbf{fun}^x(x) \to \mathbf{fun}^x(x) \to r$
- If $T \equiv T \to T$ then $v \equiv \mathbf{fun}^x(x) \to v$ or $v \equiv op$
- If $T \equiv \{l : T, \ldots, l : T\}$ then $v \equiv x$ or $v \equiv \{\, l : v, \ldots, l : v \,\}$

Proof. The only valid rule for $\Gamma \vdash v : T, \mathsf{SQL}$ being the first one where $\mathcal{D} = \mathsf{SQL}$, we have to prove the property for $\Gamma \vdash_{\mathsf{SQL}} v : T$.

We prove the property by structural induction on the typing derivation of $\Gamma \vdash_{\mathsf{SQL}} v : T$. If the last rule used is the subsumption rule, then it is immediately true by induction hypothesis, otherwise we proceed by case analysis on T:

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

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

Note: We only show the interesting cases here. The rest of the proof can be found in Appendix B, page 205.

- If $T \equiv B$ then
  - If $v = v_1\ v_2$ then by the typing rule of the application: $\Gamma \vdash_{\mathsf{SQL}} v_1 : T_1 \to T_2$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to v_3$ which is impossible by Hypothesis H1, or $v_1 \equiv op$, then by the typing rule of operators: $\Gamma \vdash_{\mathsf{SQL}} v_2 : B$, so by induction hypothesis $v_2 \equiv b$ so $v = op\ v_2 \equiv op\ b \equiv b$
  - If $v = c$ then $v \equiv b$
  - If $v = \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3$ then by the typing rule of the conditional expression:
    $\forall i \in 1..3, \Gamma \vdash_{\mathsf{SQL}} v_i : B_i$, 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 $v = v' \cdot l$ then by the typing rule of the record destructor: $\Gamma \vdash_{\mathsf{SQL}} v' : \{l_i : T_i\}_{i=1..n}$, 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 $v = \texttt{Exists}\langle v' \rangle$ then by the typing rule of `Exists`: $\Gamma \vdash_{\mathsf{SQL}} v' : R\ \mathbf{list}$, so by induction hypothesis $v' \equiv s$, so $v = \texttt{Exists}\langle v' \rangle \equiv \texttt{Exists}\langle s \rangle \equiv b$
- If $T \equiv R$ then
  - If $v = x$ then $v \equiv r$

- If $v = v_1\ v_2$ then impossible since by the typing rule of the application: $\Gamma \vdash_{\mathsf{SQL}} v_1 : T_1 \to T_2$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to v_3$ which is impossible by Hypothesis H1, or $v_1 \equiv op$, which is impossible as $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \to \ldots \to B_n \to B$
- If $v = \{\, l_i : v_i \,\}_{i=1..n}$ then by the typing rule of the record constructor $\forall i \in 1..n, \Gamma \vdash_{\mathsf{SQL}} v_i : B_i$, so by induction hypothesis $\forall i \in 1..n, v_i \equiv b$, so $v = \{\, l_i : v_i \,\}_{i=1..n} \equiv \{\, l : b, \ldots, l : b \,\} \equiv r$
- If $v = v' \cdot l$ then by the typing rule of the record destructor: $\Gamma \vdash_{\mathsf{SQL}} v' : \{l_1 : T_1, \ldots, l_n : T_n\}$, 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 $\Gamma, x : T \vdash_{\mathsf{SQL}} v' \equiv x : T \equiv R'$, so impossible since by the typing rule of the record destructor $\Gamma \vdash_{\mathsf{SQL}} v = v' \cdot l : B$

- If $T \equiv R$ **list** then
  - If $v = [\,]$ then impossible since $[\,]$ cannot be typed
  - If $v = v_1 :: v_2$ then by the typing rule of the list constructor: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **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 $v = v_1 \mathbin{@} v_2$ then by the typing rule of the list concatenation: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R$ **list** and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **list**, so by induction hypothesis $v_1 \equiv s$ and $v_2 \equiv s$, so $v = v_1 \mathbin{@} v_2 \equiv s \mathbin{@} s \equiv s$
  - If $v = \texttt{Project}\langle v_1 \mid v_2 \rangle$ then by the typing rule of `Project`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R' \to R$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R'$ **list**, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to r$ and $v_2 \equiv s$, so $v = \texttt{Project}\langle v_1 \mid v_2 \rangle \equiv \texttt{Project}\langle \mathbf{fun}^x(x) \to r \mid s \rangle \equiv s$
  - If $v = \texttt{From}\langle \mathcal{D}, v' \rangle$ then by the typing rule of `From`: $\Gamma \vdash_{\mathsf{SQL}} v' : \texttt{string} \equiv B$, so by induction hypothesis $v' \equiv b$, so $v = \texttt{From}\langle \mathcal{D}, v' \rangle \equiv \texttt{From}\langle \mathcal{D}, b \rangle \equiv s$
  - If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then by the typing rule of `Filter`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R' \to \texttt{bool}$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R'$ **list**, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to b$ and $v_2 \equiv s$, so $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle \equiv \texttt{Filter}\langle \mathbf{fun}^x(x) \to b \mid s \rangle \equiv s$
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then by the typing rules of `Join`: $v_1 = \mathbf{fun}(x, y) \to x \bowtie y$ or $\Gamma \vdash_{\mathsf{SQL}} v_1 : R' \to R'' \to R$, $\Gamma \vdash_{\mathsf{SQL}} v_2 : R' \to R'' \to \texttt{bool}$, $\Gamma \vdash_{\mathsf{SQL}} v_3 : R'$ **list** and $\Gamma \vdash_{\mathsf{SQL}} v_4 : R''$ **list**, so $v_1 \equiv \mathbf{fun}^x(x, x) \to x \bowtie x$ or by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x, x) \to r$, $v_2 \equiv \mathbf{fun}^x(x, x) \to b$, $v_3 \equiv s$ 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 $v = \texttt{Group}\langle v_1, v_2 \mid v_3\rangle$ then by the typing rule of `Group`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R'' \to R$, $\Gamma \vdash_{\mathsf{SQL}} v_2 : R'' \to R'$ and $\Gamma \vdash_{\mathsf{SQL}} v_3 : R''$ **list**, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x){\rightarrow}r$, $v_2 \equiv \mathbf{fun}^x(x){\rightarrow}r$ and $v_3 \equiv s$, so
  $v = \texttt{Group}\langle v_1, v_2 \mid v_3\rangle$
  $\equiv \texttt{Group}\langle \mathbf{fun}^x(x){\rightarrow}r, \mathbf{fun}^x(x){\rightarrow}r \mid s\rangle \equiv s$
- If $v = \texttt{Sort}\langle v_1 \mid v_2\rangle$ then by the typing rule of `Sort`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R \to R'$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **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 $v = \texttt{Limit}\langle v_1 \mid v_2\rangle$ then by the typing rule of `Limit`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : \texttt{int}$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **list**, so by induction hypothesis $v_1 \equiv b$ and $v_2 \equiv s$, so $v = \texttt{Limit}\langle v_1 \mid v_2\rangle \equiv \texttt{Limit}\langle b \mid s\rangle \equiv s$

□

We have shown that well-typedness of a QIR normal form restricts its syntactic form. We can then show that SQL-compatible normal forms can be translated into SQL by our specific translation.

Lemma 6.2. *Let v be a SQL-compatible normal form, then there exists a unique $e \in \mathsf{E}_{\mathsf{SQL}}$ such that $v \overset{\mathsf{SQL}}{\rightsquigarrow} e$.*

Proof. By induction on the structure of v. Note that the rules of Figure 6.1 used to derive the judgment $q \overset{\mathsf{SQL}}{\rightsquigarrow} e$ are syntax-directed (at most one rule applies), and terminate since the premises are always applied on a strict syntactic subexpression of the conclusion. Thus, since v is finite by definition of a QIR term, the translation derivation is finite and unique. We use (IH) to denote the induction hypothesis.

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

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

– If $v = \textbf{if}\, b_1\, \textbf{then}\, b_2\, \textbf{else}\, b_3$ then:

(SQL-if)

$$\frac{\dfrac{}{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i}\,(\text{IH}) \quad e_i \neq \Omega \quad i \in 1..3}{\textbf{if}\, b_1\, \textbf{then}\, b_2\, \textbf{else}\, b_3 \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT CASE WHEN (}e_1\texttt{) THEN (}e_2\texttt{) ELSE (}e_3\texttt{) END}}$$

– If $v = x \cdot l$ then:

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

– If $v = op\ (b_1, \ldots, b_n)$ then:

$$\frac{\dfrac{}{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i}\,(\text{IH}) \quad e_i \neq \Omega \quad i \in 1..2}{+\,(q_1, q_2) \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT (}e_1\texttt{) + (}e_2\texttt{)}}\,(\text{SQL-plus}) \qquad \frac{\dfrac{}{b \overset{\text{SQL}}{\rightsquigarrow} e}\,(\text{IH}) \quad e \neq \Omega}{sum\ q \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT sum(}e\texttt{)}}\,(\text{SQL-sum}) \quad \ldots$$

– If $v = \texttt{Exists}\langle s \rangle$ then:

$$\frac{\dfrac{}{s \overset{\text{SQL}}{\rightsquigarrow} e}\,(\text{IH}) \quad e \neq \Omega}{\texttt{Exists}\langle s \rangle \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT EXISTS(}e\texttt{)}}\,(\text{SQL-exists})$$

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

$$\frac{}{x \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT } x.*}\,(\text{SQL-var})$$

– If $v = \{\, l_i : b_i \,\}_{i=1..n}$ then:

$$\frac{\text{(SQL-record)}\quad \dfrac{\text{(IH)}}{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \quad e_i \neq \Omega \quad i \in 1..n}{\{\, l_i : b_i \,\}_{i=1..n} \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT (}e_1\texttt{) AS } X_1\texttt{, ..., (}e_n\texttt{) AS } X_n}$$

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

$$\frac{\text{(SQL-lcons)}\quad \dfrac{\text{(IH)}}{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \quad \dfrac{\text{(IH)}}{s \overset{\text{SQL}}{\rightsquigarrow} e_2} \quad \begin{matrix} e_i \neq \Omega \\ i \in 1..2 \end{matrix} \quad \texttt{TMP fresh}}{r :: s \overset{\text{SQL}}{\rightsquigarrow} e_1\ \texttt{UNION ALL (}e_2\texttt{) AS TMP}}$$

  – If $v = s_1 \mathbin{@} s_2$ then:

$$\frac{\text{(SQL-lconcat)}\quad \dfrac{\text{(IH)}}{s_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \quad \begin{matrix} e_i \neq \Omega \\ i \in 1..2 \end{matrix} \quad \texttt{TMP, TMP2 fresh}}{s_1 \mathbin{@} s_2 \overset{\text{SQL}}{\rightsquigarrow} \texttt{SELECT * FROM (}e_1\texttt{) AS TMP UNION ALL (}e_2\texttt{) AS TMP2}}$$

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

$$\frac{\text{(SQL-project)}\quad \dfrac{\text{(IH)}}{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \quad \dfrac{\text{(IH)}}{s \overset{\text{SQL}}{\rightsquigarrow} e_2} \quad e_i \neq \Omega \quad i \in 1..2}{\texttt{Project}\langle \mathbf{fun}(x) \rightarrow r \mid s \rangle \overset{\text{SQL}}{\rightsquigarrow} e_1\ \texttt{FROM (}e_2\texttt{) AS X}}$$

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

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

– If $q = \texttt{Filter}\langle \mathbf{fun}(x) \to b \mid s \rangle$ then:

(SQL-filter)

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

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

(SQL-join)

$$\frac{\overset{\text{(IH)}}{r \overset{\text{SQL}}{\rightsquigarrow} e_1} \qquad \overset{\text{(IH)}}{b \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad \overset{\text{(IH)}}{s_1 \overset{\text{SQL}}{\rightsquigarrow} e_3} \qquad \overset{\text{(IH)}}{s_2 \overset{\text{SQL}}{\rightsquigarrow} e_4} \qquad e_i \neq \Omega \quad i \in 1..4}{\texttt{Join}\langle \mathbf{fun}(x, y) \to r, \mathbf{fun}(x, y) \to b \mid s_1, s_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \begin{array}{l} e_1 \texttt{ FROM } (e_3) \texttt{ AS X,} \\ (e_4) \texttt{ AS Y WHERE } (e_2) \end{array}}$$

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

(SQL-join-noproject)

$$\frac{\overset{\text{(IH)}}{b \overset{\text{SQL}}{\rightsquigarrow} e_2} \qquad \overset{\text{(IH)}}{s_1 \overset{\text{SQL}}{\rightsquigarrow} e_3} \qquad \overset{\text{(IH)}}{s_2 \overset{\text{SQL}}{\rightsquigarrow} e_4} \qquad e_i \neq \Omega \quad i \in 2..4}{\texttt{Join}\langle \mathbf{fun}(x, y) \to x \bowtie y, \mathbf{fun}(x, y) \to b \mid s_1, s_2 \rangle \overset{\text{SQL}}{\rightsquigarrow} \begin{array}{l} \texttt{SELECT * FROM } (e_3) \texttt{ AS X,} \\ (e_4) \texttt{ AS Y WHERE } (e_2) \end{array}}$$

– If $q = \texttt{Group}\langle \mathbf{fun}(x) \to \{\, l_i : b_i \,\}_{i=1..n}, \mathbf{fun}(x) \to r \mid s \rangle$ then:

(SQL-group)

$$\frac{\overset{\text{(IH)}}{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \qquad \overset{\text{(IH)}}{r \overset{\text{SQL}}{\rightsquigarrow} e} \qquad \overset{\text{(IH)}}{s \overset{\text{SQL}}{\rightsquigarrow} e'} \qquad e_i \neq \Omega \quad e \neq \Omega \quad e' \neq \Omega \quad i \in 1..n}{\texttt{Group}\langle \mathbf{fun}(x) \to \{\, l_i : b_i \,\}_{i=1..n}, \mathbf{fun}(x) \to r \mid s \rangle \overset{\text{SQL}}{\rightsquigarrow} \begin{array}{l} e \texttt{ FROM } (e') \texttt{ AS X} \\ \texttt{GROUP BY } l_1, \ \ldots, \ l_n \end{array}}$$

– If $q = \texttt{Sort}\langle \textbf{fun}(x) \rightarrow \{\, l_i : b_i \,\}_{i=1..n} \mid s \rangle$ then:

$$\frac{\dfrac{(\text{IH})}{b_i \overset{\text{SQL}}{\rightsquigarrow} e_i} \qquad \dfrac{(\text{IH})}{s \overset{\text{SQL}}{\rightsquigarrow} e} \qquad \begin{array}{l} e_i = \texttt{true} \text{ or } \texttt{false} \\ e'_i = \texttt{DESC if } e_i = \texttt{false} \end{array} \qquad e_i \neq \Omega \quad e \neq \Omega \quad i \in 1..n}{\texttt{Sort}\langle \textbf{fun}(x) \rightarrow \{\, l_i : b_i \,\}_{i=1..n} \mid s \rangle \overset{\text{SQL}}{\rightsquigarrow} \begin{array}{l} \texttt{SELECT * FROM (}e\texttt{) AS X} \\ \texttt{ORDER BY } l_1\ e'_1,\ \ldots,\ l_n\ e'_n \end{array}} \ (\text{SQL-sort})$$

– If $q = \texttt{Limit}\langle b \mid s \rangle$ then:

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

□

Finally, we can state our soundness of translation theorem and prove it as a direct corollary of Lemmas 6.1 and 6.2.

Theorem 6.1 (Soundness of translation)**.** *Let $v \in \mathsf{E}_{QIR}$ such that v is in normal form, and $\emptyset \vdash v : T, \mathsf{SQL}$ where $T \equiv B$ or $T \equiv R$ or $T \equiv R$ **list**, then $v \rightsquigarrow \mathit{eval}^{\mathsf{SQL}}(s)$.*

Proof. By Lemma 6.1: $v \equiv b$ or $v \equiv r$ or $v \equiv s$. Thus, by Lemma 6.2, we obtain: $v \overset{\text{SQL}}{\rightsquigarrow} \texttt{s}$. The property is then true by the rule (direct) of the generic translation. □

As desired, this theorem tells us that if the specific type system of SQL gave a type that is compatible with SQL to a QIR normal form, then this normal form is translatable to SQL. Combined with our theorems from Chapter 4, we get the property that any QIR expression typed with the specific type system for SQL can be fully normalized and then translated to SQL.

Corollary 6.1. *Let $q \in \mathsf{E}_{QIR}$ such that $\emptyset \vdash q : T, \mathsf{SQL}$ where $T \equiv B$ or $T \equiv R$ or $T \equiv R$ **list**, then $\exists v \mid q \hookrightarrow^* v$, v is in normal form, and $v \rightsquigarrow \mathit{eval}^{\mathsf{SQL}}(s)$.*

Proof. By Theorem 4.4, we have $q \hookrightarrow {}^{*}v$ and v is in normal form. By Theorem 4.1 and Theorem 4.5, we have $\emptyset \vdash v : T, \mathsf{SQL}$. Therefore, by Theorem 6.1, we have $v \rightsquigarrow \texttt{eval}^{\mathsf{SQL}}(\texttt{s})$. □

6.3 Extension to scalar subqueries for SQL

SQL supports a particular feature called *scalar subqueries*. SQL allows the use of tables (results of queries in particular) in expressions that expect a basic type. Syntactically, this is a correct SQL query:

```
SELECT salary +
  (SELECT bonus FROM team AS t WHERE t.teamid = e.teamid)
FROM employee AS e
```

If the subquery returns more than one row (or zero rows) or more than one column then a runtime error is returned, otherwise SQL automatically extracts the unique value returned by the subquery and uses it as the second argument of the addition. This is not just limited to operations in an operator. For instance, these two queries give the same result in SQL:

```
SELECT 1 AS id
```

```
SELECT (SELECT 1) AS id
```

since SQL automatically extracts the only value in the table created by `SELECT 1`. In order to represent this feature, we could add a type conversion rule stating that if an expression could be typed as a flat list which records contain only one association from a label to a basic type, then it can also be typed as that basic type:

$$\frac{\Gamma \vdash_{\mathsf{SQL}} q : \{l : B\}\ \mathbf{list}}{\Gamma \vdash_{\mathsf{SQL}} q : B}$$

Adding this rule requires to modify Property 4.3 of coherence of a specific type system to include type conversion rules as possible rules to add in a database type system. The major problem with this rule is the integration to the typing algorithm for SQL since it is not algorithmic and not obvious to integrate to our constraint system. However, in SQL, the context always makes it clear whether or not a query should be considered a scalar subquery or not, in particular because SQL only allows flat records and tables composed of flat records, thus leaving the latter as the only type of scalar subqueries possible. Thus, handling this type conversion rule entails considering type constraints such as $\texttt{int} = \{l : \texttt{int}\}\ \mathbf{list}$ to be trivially true. To do this, we need to modify the equality relation $=$ between types, so that a type $\{l : B\}\ \mathbf{list}$ representing scalar subqueries are equal to the basic type B in the case of SQL.

The property of safety of translation of Corollary 6.1 still holds with the type conversion rule with a small modification in Lemma 6.1: an expression which is given a basic type can be an expression with a list type for which the type conversion rule has been applied.

6.4 Type-oriented normalization

As explained at the end of Section 3.5, our normalization procedure is based on a syntactic criterion on data operator applications which can lead to non-optimal reductions, and relies on calling database translations to figure out if reductions are useful or not which can be expensive with no guarantee of improving the query.

We solve both of these issues using our typing relation of Definition 4.7: $\Gamma \vdash q : T, \mathcal{D}$. This typing relation gives us the database that should evaluate the root of a QIR expression without having to call translations, as well as a guarantee in the case of SQL that the normalization terminates and yields a translatable query. Therefore, we can define a new type-oriented normalization that only uses the normalization of Section 3.5 as a default case.

Definition 6.5 (Type-oriented normalization)**.** The type-oriented normalization first applies the relation $\overset{\vdash}{\hookrightarrow}$ to an input QIR expression q to obtain a result q'. The relation $\overset{\vdash}{\hookrightarrow}$ is derived by the following rules:

$$\frac{\Gamma \vdash q : T, \mathsf{SQL} \quad T \equiv B \text{ or } R \text{ or } R \textbf{ list} \quad q \hookrightarrow^* v}{q \overset{\vdash}{\hookrightarrow} v}\ (\text{tnorm-SQL})$$

$$\frac{\Gamma \vdash q : T, \mathsf{MEM} \quad q_i \overset{\vdash}{\hookrightarrow} q_i'}{q \overset{\vdash}{\hookrightarrow} \{q_i \mapsto q_i'\}q}\ \{q_i\}=\mathfrak{C}(q)\ (\text{tnorm-MEM}) \qquad \frac{}{q \overset{\vdash}{\hookrightarrow} q}\ (\text{tnorm-default})$$

where (tnorm-SQL) and (tnorm-MEM) always have priority over (tnorm-default). Then, the type-oriented normalization returns q'' where $q' \overset{\mathsf{H}}{\hookrightarrow} q''$.

The relation $\overset{\vdash}{\hookrightarrow}$ of Definition 6.5 states with rule (tnorm-SQL) that if the QIR expression is typed for SQL by the generic type system with a type compatible with SQL, then it applies the relation $\hookrightarrow$ of Definition 3.13 until reaching a normal form. This is guaranteed by Corollary 6.1 to terminate and return a normal form translatable into SQL. Rule (tnorm-MEM) states that if the QIR expression is typed with MEM, then $\overset{\vdash}{\hookrightarrow}$ calls itself recursively on all the children

of the QIR expression. This rule aims to find potential subexpressions that are typed for SQL. Finally, rule (tnorm-default) returns the input expression itself meaning that the typing relation did not give any relevant information for the normalization of this expression. The type-oriented normalization first applies $\overset{\vdash}{\hookrightarrow}$ to the input expression in order to make use of our Corollary 6.1 as much as possible, then it applies the heuristic normalization $\overset{\mathsf{H}}{\hookrightarrow}$ for the rest of the expression. This type-oriented normalization can easily be extended to other database languages. For instance, if a database $\mathcal{D}$ were to prove a similar property to our Corollary 6.1 for SQL, then we could add a rule to Definition 6.5 such as:

$$\frac{\Gamma \vdash q : T, \mathcal{D} \quad q \hookrightarrow {}^{*}v}{q \overset{\vdash}{\hookrightarrow} v}$$

In this chapter, we showed how to safely manipulate QIR expressions using our type system for QIR. In the next chapter, we discuss our existing implementation of BOLDR, as well as the results of our benchmarks and what these results imply.

Chapter 7

Implementation and experiments

In this chapter, we talk about the implementation of BOLDR, and the results of our experiments. First, we illustrate how to translate a host language into QIR using the programming language R as an example.

7.1 Translation from a host language to QIR

In this section, we describe how to interface a general-purpose programming language with BOLDR by creating a driver for the programming language R as an example of a host language. Proving any type of formal properties on the host language is (unsurprisingly) out of scope. As explained in Chapter 1, our goal is to allow programmers to write queries using the constructs of the language they already master. Therefore, instead of extending the syntax of R, we extend existing functionalities, in particular by overloading existing functions.

We abstract R as the following language:

Definition 7.1 (R expressions)**.** The set $\mathsf{E}_{\mathcal{R}}$ of expressions denoted by e and values denoted by v of R are generated by the following grammars:

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

where $\mathtt{x} \in \mathsf{I}_{\mathcal{R}}$, and $\gamma \in 2^{\mathsf{I}_{\mathcal{R}} \times \mathsf{V}_{\mathcal{R}}}$ is the environment of the closure.

Definition 7.1 only defines expressions that can be translated into QIR. Expressions of R not listed in the definition are translated to host language expressions. 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; and structured data types such as vectors and tables

with named columns called *data frames*. We recall that, in R, `c` is the basic operator to create vectors, and `data.frame` is the basic operator to create data frames. R variables are statically scoped in the way it is usually implemented in dynamic languages (e.g., as in Python or JavaScript), in which 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 now highlight how data frames are manipulated in standard R. As mentioned in Chapter 1, the `subset` function filters a data frame:

```
13 subset(emp, 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. 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 `emp`. For instance:

```
employee = data.frame(
  name=c("Lily Pond", "Daniel Rogers", "Olivia Sinclair"),
  sal=c(5200, 4700, 6000)
)
subset(employee, sal > 5000, c(name))
```

applies a projection and a filter to the data frame `employee` and returns a data frame containing `name` as its only column and two rows which values are `"Lily Pond"` and `"Olivia Sinclair"`.

The join between two data frames is implemented with the function `merge`. For instance, the following example:

```
employee = data.frame(
  name=c("Lily Pond", "Daniel Rogers", "Olivia Sinclair"),
  sal=c(5200, 4700, 6000),
  teamid=c(2, 1, 1)
)
team = data.frame(
  teamid=c(1, 2),
  teamname=c("R&D", "Sales"),
```

```
  bonus=c(500, 600)
)
merge(employee, team)
```

performs the Join operation described in Figure 1.3c.

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 into QIR of their value from the current R environment, sends it to the QIR runtime for evaluation, and translates the results to R values

We also extend the set of values $\mathsf{V}_{\mathcal{R}}$:

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

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

The functions `subset` and `merge` are overloaded to call the translation ${}^{R}\overrightarrow{\texttt{EXP}}$ 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 QIR closure q_γ. Free variables in q_γ that are not in $\texttt{dom}(\gamma)$ are global identifiers whose bindings are to be resolved when q_γ is executed using `executeQuery`.

Even though we do not modify the parsing of R programs, we still want to translate R closures to QIR functions. For instance, we want to translate the following R program:

```
less2500 = function (x) { x <= 2500 }
t = tableRef("employee", "PostgreSQL")
subset(t, less2500(sal))
```

into this QIR term:

$$\begin{array}{l}(\textbf{fun}(\mathit{less2500}, t) \rightarrow \\ \quad \texttt{Filter}\langle \textbf{fun}(r) \rightarrow (\mathit{less2500})\ (r \cdot \mathit{sal}) \mid t \rangle) \\ (\textbf{fun}(x) \rightarrow x \leq 2500, \texttt{From}\langle \mathcal{D}, \mathit{employee} \rangle)\end{array}$$

which becomes, after normalization:

$$\texttt{Filter}\langle \textbf{fun}(r) \rightarrow r \cdot \mathit{sal} \leq 2500 \mid \texttt{From}\langle \mathcal{D}, \mathit{employee} \rangle \rangle$$

While it seems obvious from this example that the function `less2500` should be translated to $\textbf{fun}(x) \rightarrow x \leq 2500$, it is not always sound to do so. Indeed,

a variable `x` can be soundly translated to a QIR variable x if it is not the subject of side effects, otherwise accesses to `x` must be nested inside host language expressions $\blacksquare_R(\gamma, \texttt{x})$ 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 7.2 (Approximation of modified variables)**.** Let $e \in \mathsf{E_R}$ be an expression and γ an evaluation environment for R. The set $Mod(\gamma, e)$ of modified variables in e ranged over by $\mathcal{M}$ is inductively defined as:

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

The first five cases of the *Mod* function are the most interesting ones (the others being only bureaucratic children 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 7.3 (Translation from R to QIR)**.** We define the judgment $\mathcal{M}, \gamma \vdash e \overset{\mathsf{R}}{\rightsquigarrow} q$, which means that given a set of modified variables $\mathcal{M}$ and an R environment γ, the R expression e can be translated to a QIR expression q. The derivation of this judgment is given by the rules in Figure 7.1. We define the translation ${}^{R}\overrightarrow{\texttt{EXP}}(\gamma, e) = q$ as $Mod(\gamma, e), \gamma \vdash e \overset{\mathsf{R}}{\rightsquigarrow} q$.

Constants and identifiers are translated to QIR equivalents. Anonymous functions are translated to anonymous QIR functions. 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

$$\frac{}{\mathcal{M}, \gamma \vdash c \overset{\mathsf{R}}{\rightsquigarrow}^{R} \overrightarrow{\texttt{VAL}}(c)} \qquad \frac{x \notin \mathcal{M}}{\mathcal{M}, \gamma \vdash \texttt{x} \overset{\mathsf{R}}{\rightsquigarrow} x} \qquad \frac{\mathcal{M}, \gamma \vdash e \overset{\mathsf{R}}{\rightsquigarrow} q}{\mathcal{M}, \gamma \vdash \mathsf{function}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})\ \{e\} \overset{\mathsf{R}}{\rightsquigarrow} \mathbf{fun}(x_1, \ldots, x_n) \to q}$$

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

$$\frac{\mathcal{M}, \gamma \vdash e_1 \overset{\mathsf{R}}{\rightsquigarrow} q_1 \quad \mathcal{M}, \gamma \vdash e_2 \overset{\mathsf{R}}{\rightsquigarrow} q_2}{\mathcal{M}, \gamma \vdash \texttt{merge}(e_1, e_2) \overset{\mathsf{R}}{\rightsquigarrow} \begin{array}{l} \texttt{Join}\langle \mathbf{fun}(x, y) \to x \bowtie y, \\ \mathbf{fun}(a, b) \to \texttt{true} \mid q_1, q_2 \rangle \end{array}} \qquad \frac{\mathcal{M}, \gamma \vdash e_1 \overset{\mathsf{R}}{\rightsquigarrow} q_1 \quad \ldots \quad \mathcal{M}, \gamma \vdash e_n \overset{\mathsf{R}}{\rightsquigarrow} q_n}{\mathcal{M}, \gamma \vdash \texttt{c}(e_1, \ldots, e_n) \overset{\mathsf{R}}{\rightsquigarrow} [\, q_1, \ldots, q_n \,]}$$

$$\frac{\mathcal{M}, \gamma \vdash e_1 \overset{\mathsf{R}}{\rightsquigarrow} q_1 \quad \ldots \quad \mathcal{M}, \gamma \vdash e_n \overset{\mathsf{R}}{\rightsquigarrow} q_n \quad \mathcal{M}, \gamma \cup \{\texttt{x}_1 \mapsto e_1, \ldots, \texttt{x}_\texttt{n} \mapsto e_n\} \vdash e \overset{\mathsf{R}}{\rightsquigarrow} q}{\mathcal{M}, \gamma \vdash (\mathsf{function}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})\ \{e\})(e_1, \ldots, e_n) \overset{\mathsf{R}}{\rightsquigarrow} (\mathbf{fun}(x_1, \ldots, x_n) \to q)\ (q_1, \ldots, q_n)}$$

$$\frac{\mathcal{M}, \gamma \vdash e_1 \overset{\mathsf{R}}{\rightsquigarrow} q_1 \quad \ldots \quad \mathcal{M}, \gamma \vdash e_n \overset{\mathsf{R}}{\rightsquigarrow} q_n}{\mathcal{M}, \gamma \vdash \texttt{op}\ e_1\ \ldots\ e_n \overset{\mathsf{R}}{\rightsquigarrow} op\ (q_1, \ldots, q_n)}$$

$$\frac{\mathcal{M}, \gamma \vdash e_1 \overset{\mathsf{R}}{\rightsquigarrow} q_1 \quad \mathcal{M} \setminus \{\texttt{x}\}, \gamma \cup \{\texttt{x} \mapsto e_1\} \vdash e_2 \overset{\mathsf{R}}{\rightsquigarrow} q_2}{\mathcal{M}, \gamma \vdash (\texttt{x} = e_1); e_2 \overset{\mathsf{R}}{\rightsquigarrow} (\mathbf{fun}(x) \to q_2)\ q_1}\ \texttt{x} \notin \mathit{Mod}(\gamma, e_2)$$

$$\frac{\mathcal{M}, \gamma \vdash e_{1,1} \overset{\mathsf{R}}{\rightsquigarrow} q_{1,1} \quad \ldots \quad \mathcal{M}, \gamma \vdash e_{1,m} \overset{\mathsf{R}}{\rightsquigarrow} q_{1,m} \quad \mathcal{M}, \gamma \vdash e_{n,1} \overset{\mathsf{R}}{\rightsquigarrow} q_{n,1} \quad \ldots \quad \mathcal{M}, \gamma \vdash e_{n,m} \overset{\mathsf{R}}{\rightsquigarrow} q_{n,m}}{\mathcal{M}, \gamma \vdash \texttt{data.frame} \left(\begin{array}{l} \texttt{x}_1 = \texttt{c}(e_{1,1}, \ldots, e_{1,m}), \\ \ldots, \\ \texttt{x}_\texttt{n} = \texttt{c}(e_{n,1}, \ldots, e_{n,m}) \end{array} \right) \overset{\mathsf{R}}{\rightsquigarrow} \begin{array}{l} [\, \{\, x_1 : q_{1,1}, \ldots, x_n : q_{n,1} \,\}, \\ \ldots, \\ \{\, x_1 : q_{1,m}, \ldots, x_n : q_{n,m} \,\}\,] \end{array}}$$

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

Figure 7.1 – Translation from R to QIR terms

case of `subset`, the second argument e_2 contains free variables bound to column names. We simulate this behavior by introducing a function 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 function to project over these names. The `merge` function is similarly translated to a `Join` operator. The last interesting case is when a local variable is defined in a sequence of expressions. If the variable is not modified in the subsequent expression, then we translate this definition into a function application. Expressions that are not handled are kept in host expression nodes to 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 into QIR. The translation of constants, sequences and data frames is straightforward. The translation of a closure $\texttt{function(x}_1, \ldots, \texttt{x}_n)_\gamma\{e\}$ is simply the translation of the body wrapped in a function: $\mathbf{fun}(x_1, \ldots, x_n) \rightarrow^R \overrightarrow{\texttt{EXP}}(\gamma, e)$.

We now have everything we need to interface R and QIR. When `executeQuery` is called on a QIR closure value q, we translate the values associated to its free variables in the runtime environment to QIR values, and bind each of them to corresponding QIR variables with applications of functions, yielding a new closed QIR term that can be sent to QIR.

Let us illustrate the whole process on the introductory example of Chapter 1.

Evaluation of the query expression

When an expression recognized as a query is evaluated, it is translated into QIR (using Definition 7.3). In the introductory example, the function call

```
16  richUSPeople = atLeast(2500, "USD")
```

triggers the evaluation of the function `atLeast`:

```
10 atLeast = function(minSalary, cur) {
11   # table employee has two columns: name, salary
12   emp = tableRef("employee", "PostgreSQL")
13   subset(emp, salary >= minSalary * getRate("USD", cur),
14          c(name))
15 }
```

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:

$$\begin{array}{l}\texttt{Project}\langle\textbf{fun}(t)\rightarrow\{\ name : t \cdot name\ \}\ |\\ \texttt{Filter}\langle\textbf{fun}(e)\rightarrow e \cdot sal \geq\ \ minSalary * (getRate\ (\texttt{"USD"}, cur)))\ |\\ \texttt{From}\langle\mathsf{PostgreSQL}, employee\rangle\rangle\rangle\\ \{minSalary\ \mapsto 2500,\ \ getRate\ \mapsto \mathsf{function}_{\gamma}(\texttt{rfrom}, \texttt{rto})\{\ldots\}, cur\ \mapsto\ \ \texttt{"USD"}\}\end{array}$$

Query execution

A QIR closure is executed using the function `executeQuery`. In our example, this happens at Lines 18 and 19:

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

`executeQuery` then resolves each free variable by applying them to the translation into QIR of their value in the R environment:

$$\begin{array}{l}(\textbf{fun}(getRate)\rightarrow\\ \quad(\textbf{fun}(minSalary, cur)\rightarrow\\ \quad\quad\texttt{Project}\langle\textbf{fun}(t)\rightarrow\{\ name : t \cdot name\ \}\ |\\ \quad\quad\texttt{Filter}\langle\textbf{fun}(e)\rightarrow \geq\ (e \cdot sal, * (minSalary, getRate\ (\texttt{"USD"}, cur)))\ |\\ \quad\quad\texttt{From}\langle\mathsf{PostgreSQL}, employee\rangle\rangle\rangle\\ \quad)(2500, \texttt{"USD"})\\ )(\textbf{fun}(rfrom, rto)\rightarrow \ldots)\end{array}$$

which will be normalized to:

$$\begin{array}{l}\texttt{Project}\langle\textbf{fun}(t)\rightarrow\{\ name : t \cdot name\ \}\ |\\ \texttt{Filter}\langle\textbf{fun}(e)\rightarrow \geq\ (e \cdot sal, 2500)\ |\\ \texttt{From}\langle\mathsf{PostgreSQL}, employee\rangle\rangle\rangle\end{array}$$

then translated to SQL using $\overrightarrow{\mathsf{EXP}}^{\mathsf{SQL}}$ as:

```
SELECT T.name AS name FROM (
  SELECT * FROM (SELECT * FROM employee) AS E
  WHERE E.sal >= 2500
) AS T
```

This query is sent to PostgreSQL, and the results are translated back into QIR using ${}^{\text{PostgreSQL}}\overrightarrow{\texttt{VAL}}$, then to R using $\overrightarrow{\texttt{VAL}}^{\mathcal{R}}$.

7.2 Truffle

Truffle is an open-source framework allowing language developers to implement abstract syntax tree (AST) interpreters with speculative runtime-specialization. Language implementors typically write a parser for the target language that produces an AST composed of *Truffle nodes*. These nodes implement the basic operations of the AST interpreter (control-flow, typed operation on primitive types, object model operations such as method dispatch, etc.). A simple example of a leaf node class for a literal number looks as shown in Example 7.1.

Example 7.1.

```
public class Number extends Node {
  public final long number;

  public Number(long number) {
    this.number = number;
  }

  @Override
  public long executeLong(VirtualFrame frame) {
    return this.number;
  }

  @Override
  public Object execute(VirtualFrame frame) {
    return this.number;
  }

  @Override
  public String toString() {
    return "" + this.number;
  }
}
```

The node of Example 7.1 contains a Java long integer that stores the value of the number, as well as methods to run the node. The framework ensures that the right `execute` method will be called depending on the context of evaluation.

Nodes use the Truffle API to implement runtime specialization and inform the *Graal* JIT compiler of various key optimization aspects, such as runtime profiles on value, type, branches, or to implement runtime rewriting of the AST on de-optimization path when a speculative optimization failed [WWW+13].

As explained in the introduction, the reasons why we use Truffle for the implementation of BOLDR are:

1. Truffle languages compile expressions to abstract syntax trees which makes the manipulation of expressions easier

2. Truffle expressions can be evaluated on any JVM, giving us a simple way to evaluate host language expressions in databases

3. Several open-source Truffle languages are already implemented and ready to be experimented on

7.3 Implementation

BOLDR consists of QIR, host languages, and databases. To evaluate our approach, we implemented the full stack, with R and SimpleLanguage as host languages and PostgreSQL, HBase and Hive as databases.

The code of our open-source prototype for BOLDR can be found at:

- `https://gitlri.lri.fr/jlopez/qir` for the QIR
- `https://gitlri.lri.fr/jlopez/QueryR` for the interfaced FastR
- `https://gitlri.lri.fr/jlopez/qsl` for the interfaced SimpleLanguage and Truffle
- `https://www.lri.fr/~lopez/phd/index_en.html` for the main page of the project

Table 7.1 gives the numbers of lines of Java code for each component to gauge the relative development effort needed to interface a host language or a database 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 built-ins/operators
R to QIR / SL to QIR	750 / 1000	the translation of Section 7.1
QIR	7000	nodes, types, normalization, translation, ...
QIR to SQL / HBase language	500 / 400	the translation $\overset{\text{SQL}}{\rightsquigarrow}$ / $\overset{\text{HBase}}{\rightsquigarrow}$
PostgreSQL / HBase / Hive binding	150 / 100 / 100	low-level interface

Table 7.1 – BOLDR components and their sizes in lines of code.

As expected, the bulk of our development lies in the QIR (its definition and normalization) which is completely shared between all languages and database back-ends. Compared to its 7500 l.o.c., the development cost of languages or database drivers, including translations to and from QIR is modest (between 700 and 1000 l.o.c.).

From bottom to top, one can see that adding a non-trivial database back-end to the framework is little work (500 l.o.c. for translating QIR into SQL and another 150 to add some glue code in PostgreSQL using PL/Java [AB16]). Note that to support any other Relational DBMS, only the glue code part has to be changed. Indeed, our translation from QIR to SQL can be reused heavily, even for languages of NoSQL databases such as Cassandra's CQL, since CQL is very similar to SQL. But, of course, supporting vendor specific extensions of SQL would require to write an extended specific database translation. For HBase, we translate QIR expressions to Java objects representing HBase queries using the Java interface provided by HBase. The QIR part contains the definition of QIR operators; the normalization procedure; the generic part of the translation to databases; the type systems; and the in-memory evaluator. This amounts to a fair 7000 l.o.c. but is shared by all host languages and back-ends. Furthermore, any improvement made on that component (e.g., in the normalization module or in the runtime) benefits to all back-ends and host languages. As for the integration with the host language, given a non-trivial language (FastR amounts to 173000 l.o.c. without counting the l.o.c. of the Truffle framework), extending its parser and implementing the translation of Definition 7.3 takes about 1350 l.o.c. This last number is roughly the same for SimpleLanguage in which we defined a syntax for queries, and we expect it would be similar for other languages.

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 R, the `environment` function returns the environment affixed to a closure as a modifiable R value, the `body` function returns the body of a closure as a manipulable abstract syntax tree, and the `formals` function returns the modifiable names of the arguments of a function. Our implementation includes an experimental interface to Python as a host language using the introspection capabilities of this language.

Our implementation additionally includes an experimental driver for Spark databases, with a translation from QIR to Scala.

7.3.1 QIR

QIR is implemented in Java using Truffle under GPL v2 licence. The QIR implementation is described by Figure 7.2. It is composed of the nodes representing QIR expressions in package `ast`; drivers for databases as well as an interface for host languages in package `driver`; a parser allowing to build QIR expressions di-

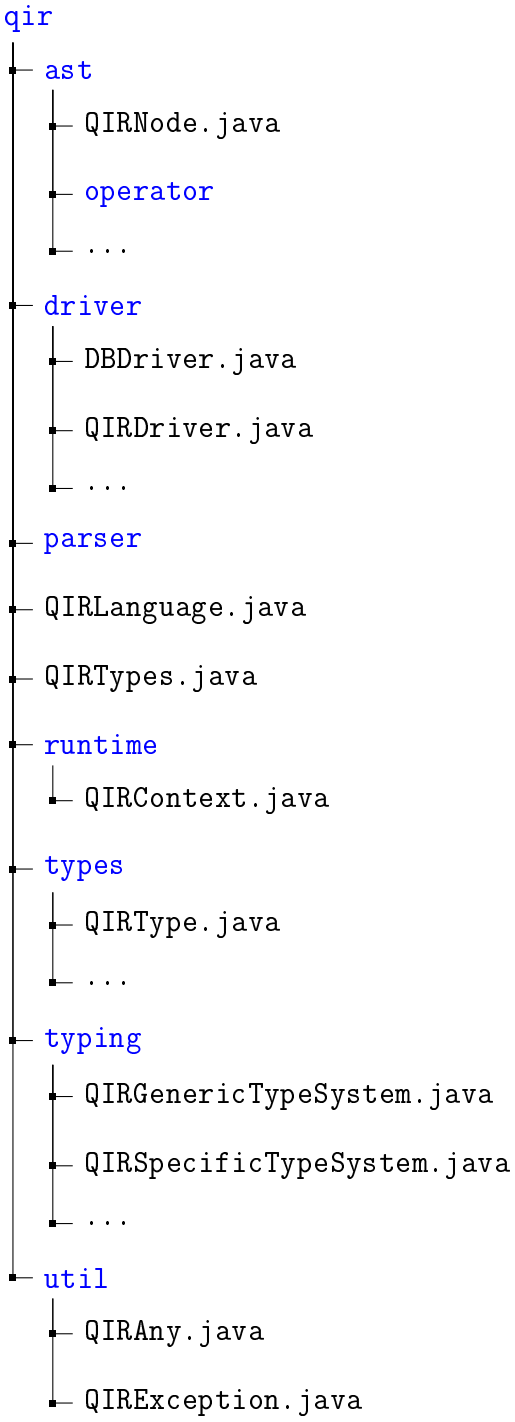


Figure 7.2 – Overview of QIR implementation

rectly from a representation as a string in package `parser`; types and type systems in package `types` and `typing`. The files `QIRLanguage.java`, `QIRTypes.java`, and `runtime/QIRContext.java` are mandatory files for the definition of QIR as a Truffle language, `util/QIRAny.java` is used as a placeholder for any possible QIR expression, and `util/QIRException.java` is an exception thrown by the components of QIR in case of error.

QIR nodes

QIR nodes are defined using the Truffle framework. Every QIR node inherits from `QIRNode.java`. This includes QIR data operators, for instance Figure 7.3 shows the implementation of `Filter` in QIR. A `VirtualFrame` is the Truffle version of a host language evaluation environment. The `executeGeneric` method is a method defined in all QIR nodes to evaluate them. Every definition of this method in a QIR node is the implementation of a rule used to infer the relation $\rightarrow$ from Definition 3.6, except for QIR nodes representing data operators in which `executeGeneric` is the implementation of the data operator by MEM. Technically, these implementations of data operators for MEM should not exist since these operators should be translated into QIR expressions as shown in Definition 3.11, but it is obviously more efficient in practice to directly implement the operator in Java and in `executeGeneric`. Finally, the `accept` method defined in every QIR node is part of an implementation of the *Visitor* design pattern which we will use for various algorithms on QIR nodes including translations. Such algorithms have to implement the `IQIRVisitor` interface, with `T` being the return type of the algorithm.

Types and type systems

The package `types` contains objects defining QIR types. All QIR types inherit from `QIRType.java`. The package `typing` contains the generic type system as a class, as well as an abstract class as interface for specific type systems which implements `IQIRVisitor`.

Our type systems use a subtyping relation instead of constraints for ease of implementation. For instance, Figure 7.4 shows how the type system for SQL returns a type for a conditional expression. The methods `checkSubtype` and `expectCommonType` defined by the abstract class `QIRSpecificTypeSystem.java` respectively throw an expression if the first argument is not a subtype of the second argument, and if the two arguments do not share a common subtype. So this method called on a QIR conditional expression **if** q_1 **then** q_2 **else** q_3 checks that the condition q_1 has a boolean type (Line 2), then checks that the "then" expression q_2 has a basic type represented by the abstract class `QIRConstantType` using the special value `QIRConstantType.ANY` which is a special type that is considered a supertype to any basic type (Line 4), and finally returns the common

```
package qir.ast.operator;
...
/**
 * {@link QIRFilter} represents the selection operation of relational
 * algebra.
 */
public final class QIRFilter extends QIROperator {
  /**
   * The filter function that takes a tuple and returns a value
   * (typically a boolean) that describes whether or not the tuple
   * should be kept in the result set.
   */
  private final QIRNode filter;
  /**
   * The next {@link QIROperator} in the tree.
   */
  private final QIRNode child;
  ...
  @Override
  public final QIRNode executeGeneric(final VirtualFrame frame) {
    final QIRNode f = filter.executeGeneric(frame);
    final QIRList input;
    try {
      input = child.executeList(frame);
    } catch (UnexpectedResultException e) {
      throw new QIRException("Expected list as input in QIRFilter");
    }
    return input.filter(e -> {
      try {
        return new QIRApply(null, f, e).executeBoolean(frame).isTrue();
      } catch (UnexpectedResultException e2) {
        throw new QIRException("Expected boolean as result of
            configuration application in QIRFilter");
      }
    });
  }

  @Override
  public final <T> T accept(final IQIRVisitor<T> visitor) {
    return visitor.visit(this);
  }
}
```

Figure 7.3 – Implementation of `Filter` in QIR

subtype of the two expressions q_2 and q_3 if it exists (Line 5).

```
1public final QIRType visit(final QIRIf qirIf) {
2  checkSubtype(qirIf.getCondition().accept(toAccept),
       QIRBooleanType.getInstance());
3  final QIRType thenType = qirIf.getThenNode().accept(toAccept);
4  checkSubtype(thenType, QIRConstantType.ANY);
5  return expectCommonType(qirIf.getThenNode().accept(toAccept),
       qirIf.getElseNode().accept(toAccept));
6}
```

Figure 7.4 – The type system for SQL on a conditional expression

Additionally, our type systems implement the possibility to give more powerful constraints to our types. For instance, Figure 7.5 shows the method that infers a type for the `Project` data operator of relational algebra. This method first uses `expectIfSubtype` to check that the data argument of `Project` has a list type given by the attribute `anyListType` (Line 2), then checks that the configuration of `Project` has a function type that takes the type of an element of the data argument and returns the type `expectedFormatterReturnType` (Line 3), and finally it returns a list type which elements have the return type of the configuration (Line 4).

`expectedFormatterReturnType` restricts accepted QIR types for data operators. In the type system for MEM, `expectedFormatterReturnType` is set to `QIRAnyType`, which is, similarly to `QIRConstantType.ANY`, a special type considered as a supertype of any other type:

```
public QIRType visit(final QIRProject qirProject) {
  return visit(qirProject, QIRAnyType.getInstance());
```

```
1protected QIRType visit(final QIRProject qirProject, final QIRType
    expectedFormatterReturnType) {
2  final QIRListType childType = expectIfSubtype(qirProject.getChild().
       accept(toAccept), anyListType);
3  final QIRFunctionType formatterType = expectIfSubtype(qirProject.
       getFormatter().accept(toAccept), new QIRFunctionType(childType.
       getElementType(), expectedFormatterReturnType));
4  return new QIRListType(formatterType.getReturnType());
5}
```

Figure 7.5 – The type system for SQL on the `Project` data operator

```
}
```

But in the type system for SQL, the expected return type is a flat record, a record that can only contain basic types. To achieve this, our implementation of a QIR record type gives the possibility to add a global restriction to the types of elements of a QIR record type. So for the type inference for SQL, `expectedFormatterReturnType` is the empty record type with a global restriction set to the special value `QIRConstantType.ANY`:

```
@Override
public final QIRType visit(final QIRProject qirProject) {
  return visit(qirProject,
    QIRRecordType.anyRestrictedTo(QIRConstantType.ANY));
}
```

We also see here how the Visitor pattern and the dynamic dispatch of Java allow us to factorize common code between algorithms.

As for `anyListType`, it would be set in the case of MEM to a list which argument type is `QIRAnyType`, meaning that MEM accepts any type of list. SQL would set the attribute to a list for which the type of elements is a record type that can only contain basic types.

Drivers

Our implementation of BOLDR does not have a fixed definition of a host language driver. It only has an interface `QIRInterface.java` that contains a `run` function taking a QIR term as a query and returning a QIR term as results of the evaluation of the query or throwing a `QIRException` in case of error. Less naive ways to transmit information from database tables such as cursors [EN89] are not yet supported.

However, database drivers have a fixed definition in our implementation in the form of the following abstract functions present in a Java abstract class `DBDriver.java`:

```
public abstract void openConnection(final String newConfigFile);
public abstract void closeConnection();
public abstract boolean isConnOpen();
public abstract QIRType type(final QIRNode query);
public abstract DBRepr translate(final QIRNode query);
public abstract QIRNode run(final DBRepr query);
```

`openConnection`, `closeConnection`, and `isConnOpen` are handling the connection to the database. For instance, our implementation of a PostgreSQL driver

uses a JDBC driver to establish a connection to the database using the method `getConnection()` from the class `java.sql.DriverManager`.

The function `type` gives a `QIRType` to a QIR expression. This function corresponds to the specific type system defined in Definition 4.6 for the database. The function `translate` takes a `QIRNode` and returns its translation into a representation that can be evaluated by the database. This function corresponds to the specific translation defined in Definition 6.1 for the database. The return type of `translate` is the parametric type `DBRepr` specified by the driver as the query representation for the corresponding database. For instance, in the case of a SQL database, `DBRepr` can simply be the type `String` since we can translate the query to a SQL string as in Definition 6.3. Both of the functions `type` and `translate` are algorithms that can implement `IQIRVisitor` as mentioned before.

Finally, the function `run` takes a query in the representation of `DBRepr`, sends it to the database for evaluation, then retrieves the results and translates them into QIR.

7.3.2 Interface to FastR

Our implementation of BOLDR interfaces with FastR, the implementation in Truffle of the language R.

As already mentioned in Section 7.1, our implementation overrides the already existing builtin functions `$` and `subset` to work on queries. The integration is not yet completely implemented, so a syntax for queries has also been added for temporary use for other operators such as `Group` or for complex configurations. So the query:

```
subset(emp, sal >= min_salary, c(emp_id, emp_name))
```

can also be written:

```
query.select(function (x) {
  res = new.env()
  res$empno = x$emp_id
  res$ename = x$emp_name
  res
},
query.where(function (x) x$sal >= min_salary,
query.from(emp)))
```

The interface between FastR and QIR also retrieves values of free variables using Truffle frames, then binds them using QIR applications of QIR functions to the translation of values into QIR as intended.

Finally, the interface defines functions using Truffle in order to evaluate Truffle closures. These functions are used inside databases to evaluate host language expressions.

7.3.3 Host language expressions in databases

In order to evaluate a host language expression in Truffle inside databases, we call the Java Virtual Machine present in the database, or if none exist natively we find a way to interface one to the database.

PostgreSQL

PostgreSQL is a database written in C, that does not natively have access to a JVM. To evaluate host language expressions in PostgreSQL, our implementation of BOLDR uses PL/Java[AB16] which interfaces PostgreSQL, allowing us to import a jar file of our Truffle languages in PostgreSQL.

Hive

Hive is a software built on top of Hadoop which provides a SQL-like query language compiled to MapReduce operations. Hive is written in Java, thus we can directly use the intended way to call foreign Java functions from jar files [Hiv].

Syntax of host language expression application in practice

We get to a bit of a technical problem when a host language expression depends on data stored in the database. Indeed, a function stored in a host language expression can be difficult to represent as an object that a query language can manipulate.

Let us take the example of PostgreSQL in which we want to execute R code. We use the extension called PL/Java to execute FastR code in a PostgreSQL database. This allows us to create a function that calls the runtime of R to evaluate an R expression. For instance:

```
SELECT r.executeR('2')
```

However, if we use this function to evaluate an R function:

```
SELECT r.executeR('function(x) x + 2')
```

we get a result that cannot be easily recovered as a SQL value. Because of this, `r.execute` returns an opaque object that represents the return value of the evaluation of the program in the R runtime. Unfortunately, we cannot directly apply this opaque object to data present in tables. Thus, we create another function in

R that evaluates an R function with arguments, and interface it with PostgreSQL using PL/Java by creating another SQL function named `r.executeApply`:

```
SELECT r.executeApply(r.executeR('function (dol){
              a = dol * 89.0 / 100.0
              while (a > 1000.0) a = a * 89.0 / 100.0
              a
            }'), array[r.translateToR(x.sal)]) AS salary,
       x.name AS name,
       x.id AS id
FROM public.employee AS x
WHERE NOT ((x.sal) < (2500.0))) AS x;
```

The function `r.translateToR` is another function created using PL/Java allowing us to translate the arguments for `r.executeApply`. This translation makes use of the automatic translation of PL/Java between SQL values and Java objects as a temporary solution, but in the future, it should rely on the translations defined in BOLDR: from SQL to QIR, then from QIR to R.

7.4 Experiments

The test machine for our experiments is 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, HBase 1.2.6, and Java 1.8, all with default parameters.

The results of our evaluation are reported in Figures 7.6, 7.7, 7.8, 7.9, and 7.10. Queries labeled TPCH-n are SQL queries taken from the TPC-H performance benchmark [TPC17]. These queries feature joins, nested queries, grouping, ordering, and various arithmetic subexpressions. Figures 7.6, 7.7, and 7.8 illustrate how our approach fare against hand-written SQL queries. Figure 7.6 reports the expected cost in disk page fetches as reported by the `EXPLAIN ANALYZE` commands scaled on the cost of the queries in pure SQL, and Figure 7.7 reports the execution time on a 1GB data set. In the legends of the figures, Pure SQL represents the hand-written SQL queries, Pure SQL+UDFs represents the same SQL queries where some subexpressions are expressed as function calls of stored functions written in PL/SQL. BOLDR R represents the SQL queries generated by BOLDR from equivalent R expressions, and BOLDR R+UDFs represents the same SQL queries as in SQL+UDFs generated by BOLDR from equivalent R expressions with R UDFs. Lastly, for BOLDR 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 Pure SQL with BOLDR R, and that BOLDR outperforms PostgreSQL in BOLDR R+UDFs against Pure SQL+UDFs. This last result comes from the

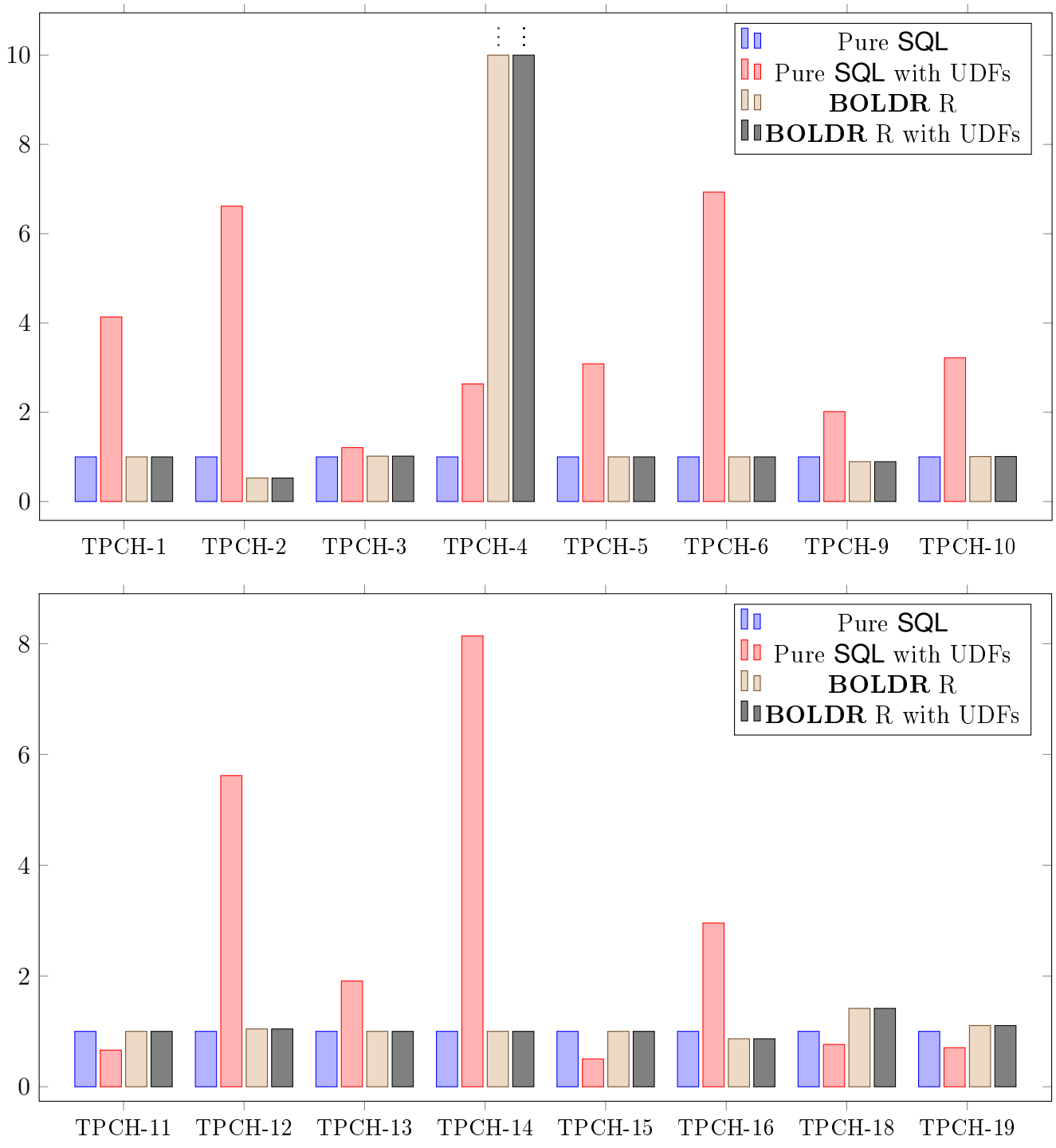
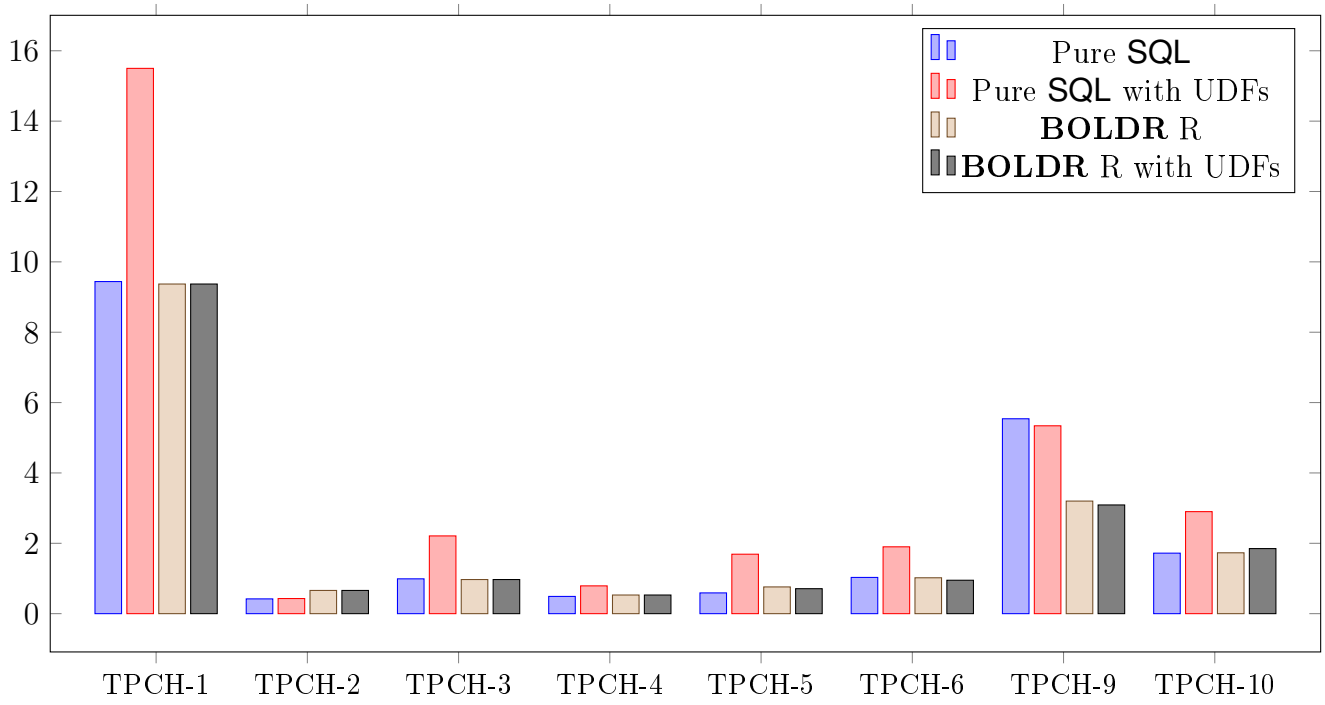
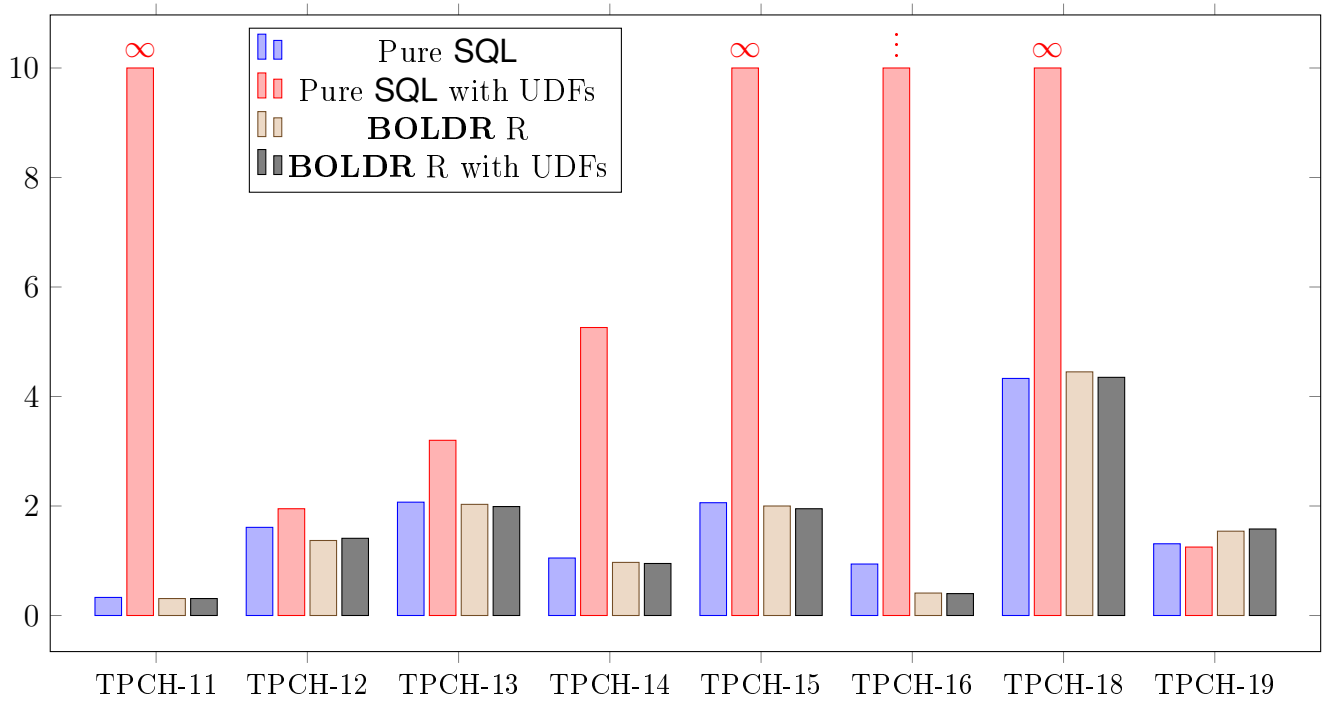


Figure 7.6 – Page fetches on TPC-H queries (scaled on the "Pure SQL" column)





∞: evaluation took more than 5 minutes.

Figure 7.7 – Time elapsed on TPC-H queries (in seconds)

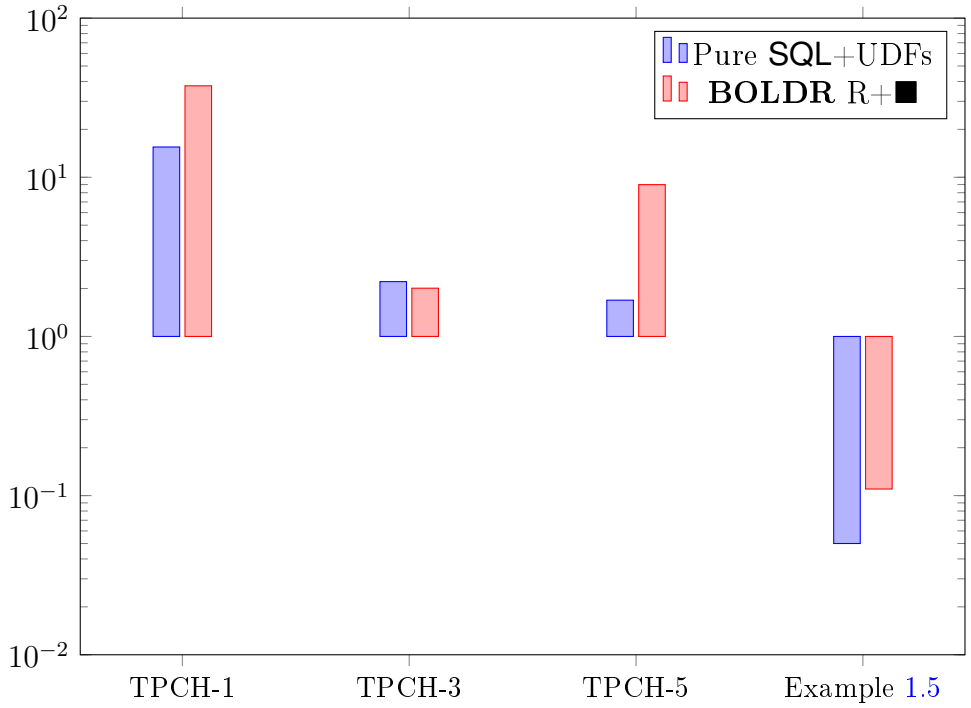


Figure 7.8 – Time elapsed on TPCH queries with host language expressions

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 BOLDR R+UDFs as:

```
supplier = tableRef("supplier", "PostgreSQL", "pg.conf", "tpch")
revenue = tableRef("revenue", "PostgreSQL", "pg.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 Pure SQL+UDFs could not be inlined by the optimizer of PostgreSQL.

Figure 7.8 illustrates the overhead of calling the host language evaluator from

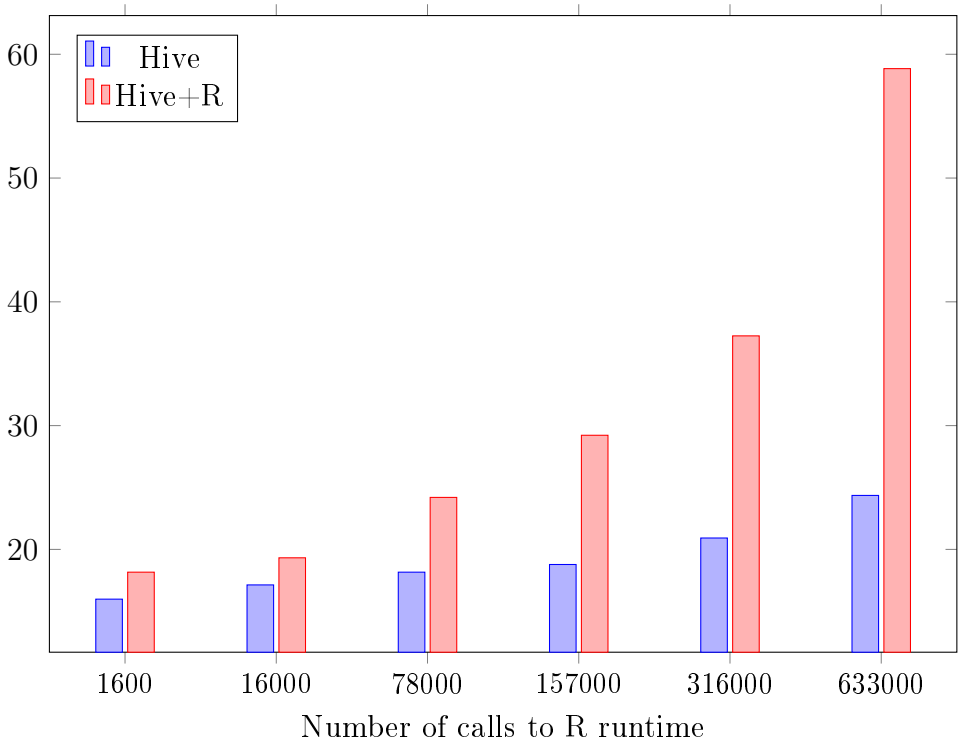


Figure 7.9 – Comparison on execution time (in seconds) for Hive queries with and without host language expressions

`getRate` \ `atLeast`	PostgreSQL	HBase	Hive
PostgreSQL	0.34	1.47	1.17
HBase	1.44	1.33	2.07
Hive	0.74	1.78	0.66

Figure 7.10 – Comparison on execution time (in seconds) for Example 1.5 targeting two databases

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 into the database language.

Figure 7.9 illustrates the overhead of calling the host language evaluator from Hive against a pure inlined Hive query. For instance:

```
SELECT rexecuteapply('function(x, y) x+y', array(p.pid, d.mid))
FROM PEOPLE p, DIRECTOR d WHERE p.pid < N
```

against

```
SELECT rexecuteapply('function(x, y) x+y', array(p.pid, d.mid))
FROM PEOPLE p, DIRECTOR d WHERE p.pid < N
```

The results are that the overhead of calling the R runtime is small compared to the execution of the query in Map/Reduce for an input data inferior to 80000 rows. The cost of 315000 calls to the runtime of the R runtime is shown as roughlt twice slower than the same query with no such calls. The cost of 633000 calls to the runtime of the R runtime is shown as two and a half times slower than the same query with no such calls. These results show that, even using a naive interface between the R runtime and Hive and a regular JVM, BOLDR generates queries containing application code that are executable with decent performances, especially compared to transferring data to the application side.

Figure 7.10 gives the performances of queries mixing two data sources between a PostgreSQL, a HBase, and a Hive database. We executed Example 1.5 and varied the data sources for the functions `getRate` and `atLeast`. In the current implementation, a join between tables from different databases is performed on the client side (see our future work in Section 8.3), therefore the queries in which the two functions target the same database perform better, since they are evaluated in a unique database implying less network delays and less work on the client side.

Chapter 8

Conclusion

In this chapter, we first talk about work in the literature relevant to our work on BOLDR. Next, we give a global conclusion to this document by summarizing its contributions. Finally, we explore possible future developments for BOLDR.

8.1 Related work

The work in the literature closest to BOLDR is T-LINQ [CLW13] 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 the translation of a query cannot cause a query avalanche, (*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. 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 of host languages. T-LINQ works on the pure aspects of the host language, with quotation and anti-quotation support and a type-system, although the implementations of LINQ, including P-LINQ presented in [CLW13], make a best effort to handle a larger set of host language expressions in queries. Also, T-LINQ only supports one (type of) database per query and a 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 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 implementors (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 [Ein11]. Lastly, we obtained formal results corresponding to those of T/P-LINQ by interfacing a specific SQL type system to our framework.

In order to detect queries in regular code, T-LINQ uses a system of *quotations* that syntactically delimits queries, and *anti-quotations* used as an escape environment in queries in order to refer to F# constructs. This syntactic technique has the advantage of making the detection of queries trivial, and therefore it is very commonly used [Kis14, SPJ03], but it does not offer a seamless integration into the programming language.

In the footsteps of T-LINQ, Suzuki et al. [KSK16] define a language-integrated query framework for which both the query language and the translation rules to SQL are safely user-extensible. In [KK17], a denotational approach is taken to create a language-integrated query framework that supports sound `ORDER BY` and `LIMIT` operations.

Links [CLWY07] is a programming language that generates type-safe SQL queries in the context of creating Web applications in one single language. The type system of Links ensures that well-typed queries can be translated to SQL queries [LC12].

Ur/Web [Chl10] is a domain-specific language for the creation of Web applications. It relies on the programming language Ur and its type inference engine to type-check metaprograms that generate programs to build HTML documents and SQL queries.

SML# [OU11] is a version of Standard ML that seamlessly integrate SQL. In this language, a legal SQL expression is a polymorphically typed first-class citizen that can be freely combined with any features of Standard ML, including high-order functions, data type definition, and its module system. SQL expressions are then sent to a database server to be evaluated.

Efforts have been made to extend the Scala programming language [GIS10] to the expression of queries using the syntax of LINQ and the native Scala syntax for comprehensions [PJW07] and taking advantage of the strong static type system to analyse the type safety of queries at compile-time. Libraries such as Quill [Qui] and Slick [Sli] provide access to databases and language-integrated queries.

QIR is not the first intermediate language of its kind. While LINQ proposes the most used intermediate query representation, recent work by [OPV14] introduced SQL++, an intermediary query representation whose goal is to subsume SQL and NoSQL. 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 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.

While the sets of operators of QIR and of SQL++ are roughly the same, their design and use is quite different. In particular, QIR is designed as a calculus, allowing us to supplement the semantics of our operators with arbitrary functions and to perform high-level optimizations through guided partial evaluation while SQL++ lacks such flexibility.

[GRS10] 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.

dotConnect [dot] uses ADO.NET, an object-relational mapping framework for the .NET framework, to give access to data sources for .NET languages. It also provides an interface with the LINQ framework.

UnityJDBC [Uni] is a solution that can evaluate queries written in SQL targeting several databases at the same time. UnityJDBC supports any data source accessible with a JDBC interface, as well as other databases such as Cassandra and MongoDB.

Our current implementation of BOLDR is at an early stage and, as such, it suffers several shortcomings. Some are already addressed in existing literature. First, 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 [CW11] shows how client-side effects can be re-ordered and split apart from queries. Third, at the moment, when two subqueries target different databases, their aggregation is done in the QIR runtime. [CLF15] 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. Fourth, we do not use the table schemas provided by databases. These would allow us to detect more errors in queries before their translation.

An alternative to modifying the semantics of operators in the language and adding explicit query evaluation functions such as `executeQuery` of Chapter 7 to express and evaluate queries in already existing programming languages is to identify parts of code that should be considered as queries using imperative code extraction [ERBS16], transparent persistence [WC07] or synthesis [CSLM13]. Additionally, it is possible to reduce the latency of applications by sending queries to databases before the results are needed, this is called *query result prefetching*. Recent techniques [RS12] allows applications to apply query result prefetching

efficiently and in a way that is transparent to programmers.

8.2 Conclusion

In this thesis, we studied how to create, translate and evaluate queries safely and efficiently, through a language-integrated framework.

QIR. We defined an intermediate representation of queries as a language called QIR, which allowed us to apply a database-agnostic optimization called *normalization* in order to merge subqueries together by reducing application code that glues them together.

Type systems. We defined a type system for QIR allowing data operators to be typed according to the semantics of the particular targeted database. This was achieved by creating a modular *generic* type system that can be extended with *specific* type systems created by databases typing QIR expressions that are compatible with the query language of the database. We used those type systems to prove safety properties on the evaluation of QIR expressions and on the normalization. We also defined a specific type system for SQL.

Typing algorithms. We defined typing algorithms suitable for implementation that we prove to be equivalent to our type systems. These typing algorithms generate a type that contains type variables along with constraints on types. These constraints are resolved by our unification algorithm which generates a substitution of type variables to be applied to the type generated by the typing algorithm. We also proved that this typing algorithm terminates and returns substitutions that indeed solve the constraints given as input.

Typed evaluation. We defined how to translate a QIR expression into a database language expression, and showcased this by defining a translation from QIR to SQL. We also showed that a term typed by our specific type system for SQL is guaranteed to be translatable to SQL by our translation. Additionally, we define a normalization based that makes use of our type systems to detect if reducing a QIR expression is guaranteed to terminate.

Implementation. We implemented BOLDR, with support for several host languages and databases. We experimented on our framework and showed that, for most queries, BOLDR generates queries as efficient as hand-written queries in SQL, and it can handle queries between different databases and containing application logic with decent performances.

We have shown with BOLDR that it is possible to create a language-integrated framework allowing application programmers to write queries in the language they are experts in, without having to be expert in the query language or data model of the targeted databases. We additionally showed it is possible to guarantee some safety of execution of these queries, and all of this without sacrificing performances.

8.3 Future work

First, we want to create more host language and database drivers to link more components to BOLDR. Our experiments on Python call for interfacing more languages that are not implemented using Truffle. On the other side of the framework, there are many field-specific database languages the framework could interface with. For instance, the query language Cypher [FGG+18] created for the database Neo4j is designed for graph databases. Integrating these languages is crucial in order to allow BOLDR to query specialized databases efficiently. Additionally, it would be interesting to explore how to interface a statically typed programming language to BOLDR, in particular to study how to make our generic type system work with the type system of the language. Intuitively, our type system being standard, it should be possible to transfer information from the type system of the language to the type system of QIR.

Currently, queries targeting more than one data source 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. For instance, as explained in Chapter 4, in a join between two distinct data sources, it could be more efficient to transfer data from one data source to the other data source which could then complete the join. This requires a study on how to guarantee such an optimization would really be efficient, likely based on the quantity of information to be transferred.

Similarly, a whole new range of optimization possibilities could be exploited if databases could evaluate QIR code. Indeed, if that were possible, then more queries that are not translatable in the query language of the database could be evaluated in the database. For instance, consider the following query:

$$\texttt{Join}\langle \textbf{fun}(x,y) \rightarrow x \bowtie y, \textbf{fun}(x,y) \rightarrow x.\texttt{teamid} = y.\texttt{teamid} \mid \\ \texttt{From}\langle \text{"}\texttt{Employee}\text{"}, \mathsf{Cassandra}\rangle, \texttt{From}\langle \text{"}\texttt{Team}\text{"}, \mathsf{Cassandra}\rangle\rangle$$

Since `Join` is not supported by the database Cassandra, BOLDR performs the `Join` in-memory after sending queries to Cassandra to fetch the data from the two tables. However, if Cassandra could evaluate QIR code, then we would be able to send the entire query to the database, and evaluate the `Join` there, in the embedded QIR runtime. The key difference here is that instead of fetching the data from the two tables, it is the result of the operation that is sent to the application side. Thus, just like for data transfer between databases, the usefulness of this optimization depends on the query: performing it would be useful if the result of the operation in QIR yields significantly less data to be transferred, since network delays would then be greatly reduced. Our implementation of QIR being written in Java (using Truffle), evaluating QIR code in a database is absolutely feasible, and we will explore this possibility in the near future.

Our type system tells us where every subexpression of a query should be evaluated. By adding security constraints, it would be possible to ask the generic

type system to ensure that there is no transfer of sensitive data from the database to a client by making sure operators manipulating this sensitive data are never typed for MEM.

Our typing algorithms currently deduce the type of the data present in a table using type constraints. Some databases such as relational databases give types to their tables. BOLDR could use this information to have an additional verification of the validity of queries.

ORMs and LINQ can always type queries since they target statically typed programming languages. BOLDR cannot do the same since its queries might contain code from dynamically typed languages, and we do not plan to statically type Python. In order to extend the type system of QIR to some host language expressions, we could use a recent technique called *gradual typing* [ST06] which mixes static and dynamic typing in the same language. For instance, *mypy* [MyP] allows to freely mix between static and dynamic typing in Python. This would allow us to extend our type system to host language expressions containing statically typed code, and therefore give guarantees even on QIR expressions that contain host language code.

Our developments on BOLDR use data operators from relational algebra. Unfortunately, these operators are not directly adaptable for operations in graph or map-reduce databases. Of course, those databases can interface their specific operators to the framework. However, having to use the specific operators for each database makes queries written in host languages dependant to the targeted databases, which goes against our goal of allowing programmers to write queries without being experts in the data models and languages of their targeted databases. Therefore, a future line of improvement for BOLDR is the definition of more generic data operators that would allow programmers to write queries in the same way no matter which databases they target.

The translations from QIR to databases are directly written in Java in our implementation. A possible improvement would be the creation of a domain-specific language to define translations from QIR into database languages, leaving the implementation details to the language itself, with the associated gains of speed, clarity, and concision. A starting point for this extension named *DCDL* for Database Capabilities Description Language can be found in [Lop16]. DCDL bases itself on *macro tree transducers* [CDG+07, BD13] to define sequences of translation rules.

Our implementation of the default database MEM that evaluates QIR expressions is rather naive. We could achieve better performances, for instance, by compiling our QIR expressions into LLVM code. The runtime Weld [PTS+17, PTN+18] has shown that it is possible to increase the performances of data-oriented programs considerably by compiling into multi-threaded LLVM code.

Finally, our integration of host languages in databases in our experiments is not optimized. First, our databases use standard JVMs to evaluate code. As explained in Chapter 7, we could improve our performances using Graal, a JVM

designed to execute Truffle code efficiently. Second, our implementation currently stores the code of the program in a host language expression as a string. Work is underway to allow the serialization of arbitrary Truffle ASTs, which would make BOLDR able to store those ASTs in host language expressions instead, thus avoiding to parse the program in the database.

Appendices

Annexe A

Résumé étendu

A.1 Contexte

Le stockage, l'accès, et la manipulation de données sont des opérations vitales et critiques dans la plupart des applications. Les applications Web, statistiques, l'intelligence artificielle, l'Internet des objets, tous doivent accéder à une grande quantité d'information stockée dans des sources de données hétérogènes.

Les applications sont écrites dans des *langages de programmation généralistes* souvent choisis en fonction de leur compatibilité avec un domaine spécifique (par exemple, R ou Python pour l'analyse statistique ou la fouille de données, JavaScript pour la programmation Web). Ces langages de programmation sont souvent *impératifs*, ce qui signifie que les utilisateurs de ces langages doivent décrire comment accéder à la mémoire de l'ordinateur et la manipuler. Pour cela, les utilisateurs écrivent des séquences d'expressions, et chacune de ces expressions modifient l'état du programme.

Les données, elles, sont stockées dans des *bases de données* gérées par des *systèmes de gestion de bases de données* (SGBD). Ces systèmes gèrent le stockage, l'accès optimisé aux données en utilisant un *langage de requêtes*, la tolérance aux pannes, la modularité et la confidentialité des données, et plus encore. Une expression d'un langage de requêtes, appelée *requête*, décrit les données demandées au lieu de détailler comment y accéder, laissant au SGBD le soin de choisir la meilleure façon de procéder.

Pour accéder aux données stockées dans une base de données, une application envoie une requête à la base dans son langage de requêtes. Souvent, une application orientée données contient des composants qui jouent le rôle d'interface entre l'application et les différentes bases de données cibles. Par exemple, une technique récente appelée *polyglot persistence* consiste à séparer les données nécessaires aux différents composants d'une application dans différents types de bases de données. Cela permet de tirer avantage des capacités des différentes bases de données où elles sont les plus efficaces. La Figure A.1 montre un exemple d'une

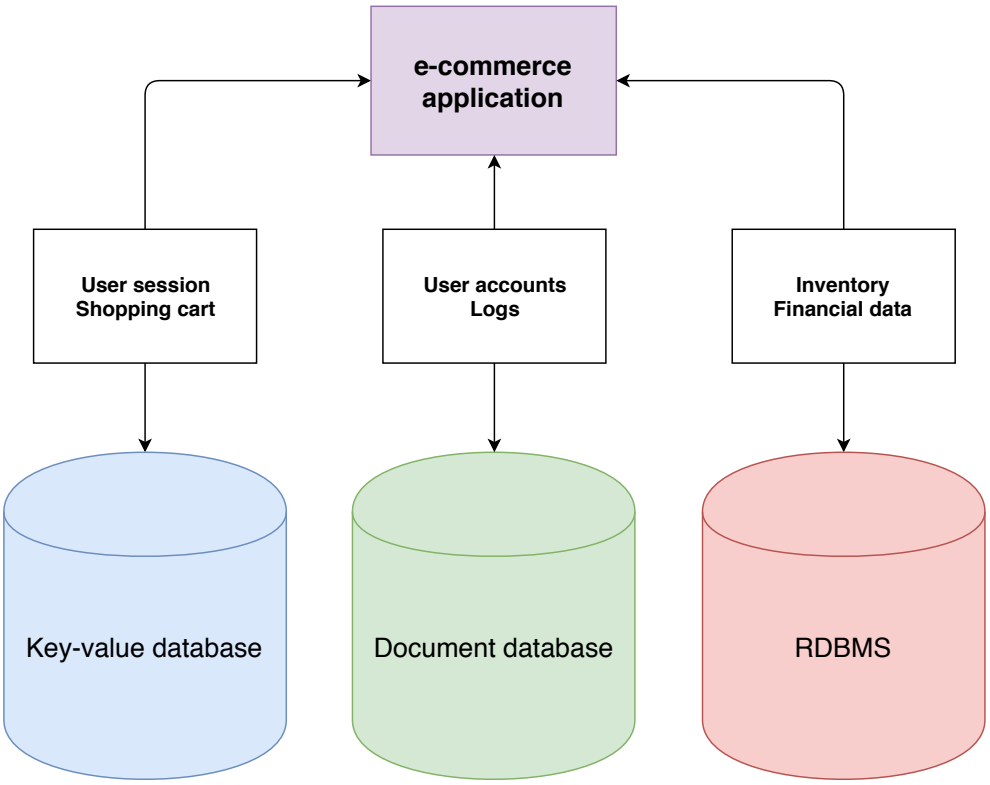


FIGURE A.1 – Exemple d'application utilisant différents types de bases de données

telle application.

Cette thèse est une étude dont le but est de créer une solution permettant aux développeurs d'applications d'écrire des requêtes sûres et efficaces sans avoir besoin d'être des experts dans les modèles de données et les langages de requêtes des bases de données ciblées.

Nous donnons ici un aperçu des langages de requêtes des bases de données, des langages de programmation utilisés dans le développement d'applications, et les solutions existantes pour interfacer ces deux mondes ainsi que les problèmes rencontrés. Ensuite, nous décrivons une nouvelle solution sous la forme d'un nouveau framework de requêtes intégrées au langage appelée BOLDR.

A.2 SQL

SQL (Structured Query Language) est le langage de requêtes le plus populaire. C'est un langage dédié (ou domain-specific language en anglais) basé sur l'*algèbre relationnelle*.

A.2.1 Algèbre relationnelle

L'algèbre relationnelle créée par Edgar F. Codd [Cod70], définit des opérations sur des données représentées comme des ensembles de *n-uplets* dans lesquels chaque élément correspond à un attribut dénoté par un nom. Les bases de données relationnelles appellent ces constructions des *tables*, composées de *lignes* et de *colonnes*. La Figure A.2 montre des exemples de tables.

id	name	salary	teamid
1	Lily Pond	5200	2
2	Daniel Rogers	4700	1
3	Olivia Sinclair	6000	1

(a) Table Employee

teamid	teamname	bonus
1	R&D	500
2	Sales	600

(b) Table Team

FIGURE A.2 – Un exemple de données organisées en tables

La plupart des langages de requêtes des bases de données sont basés sur l'algèbre relationnelle [AHV95]. Les opérations les plus basiques de l'algèbre relationnelle sont la *projection*, qui restreint les n-uplets à un ensemble d'attributs ; la *sélection* (ou restriction), qui ne conserve que les n-uplets satisfaisant une condition ; et la *jointure*, qui renvoie l'ensemble des combinaisons de n-uplets provenant de deux tables dont les valeurs sont égales sur leurs attributs communs. La Figure A.3 montre des exemples d'applications de ces opérations sur des tables. La Figure A.3a montre la projection de la table *People* sur les attributs *firstname* et *lastname*, la Figure A.3b montre la sélection des n-uplets de la table *People* pour lesquels la valeur de l'attribut *zipcode* est 13000, et la Figure A.3c montre le résultat de la jointure entre les tables *People* et *Team*.

name	salary
Lily Pond	5200
Daniel Rogers	4700
Olivia Sinclair	6000

(a) Projection sur *Employee*

id	name	salary	teamid
1	Lily Pond	5200	2
3	Olivia Sinclair	6000	1

(b) Sélection sur *Employee*

id	name	salary	teamname	bonus
1	Lily Pond	5200	Sales	600
2	Daniel Rogers	4700	Sales	600
3	Olivia Sinclair	6000	R&D	500

(c) Jointure entre *Employee* et *Team*

FIGURE A.3 – Exemple d'applications de l'algèbre relationnelle

A.2.2 Exprimer des requêtes en SQL

SQL permet d'utiliser les opérations de l'algèbre relationnelle dans un langage de programmation *déclaratif*. Au lieu de décrire étape par étape comment le calcul doit être fait pour obtenir le résultat désiré, la programmation en SQL consiste à décrire le résultat voulu. Pour cela, la syntaxe de SQL est faite pour se rapprocher d'un langage naturel. Par exemple, la projection de la Figure A.3a peut être écrite en SQL ainsi :

```
SELECT name, salary FROM Employee
```

où `SELECT` représente la projection, et `FROM` représente l'opération sur les données `From` qui renvoie le contenu d'une table à partir de son nom.

La sélection de la Figure A.3b peut être écrite :

```
SELECT * FROM Employee WHERE salary > 5000
```

où * signifie toutes les colonnes. Enfin, la jointure de la Figure A.3c peut être écrite :

```
SELECT * FROM Employee NATURAL JOIN Team
```

ou encore :

```
SELECT * FROM Employee, Team WHERE Employee.deptno = Team.deptno
```

Comme le montre cette dernière requête, les noms des tables peuvent être utilisés comme les noms de la ligne courante de la table correspondante. Cela permet de lever l'ambigüité sur quelle n-uplet doit être accédé pour la valeur de la colonne. De plus, SQL permet la création d'*alias* pour donner un nom temporaire aux tables avec le mot-clé `AS`. Par exemple, notre dernière requête peut s'écrire :

```
SELECT * FROM Employee AS e, Team AS t WHERE e.deptno = t.deptno
```

Dans ce cas, les alias ne sont pas nécessaires puisque les noms de tables permettent déjà de distinguer les lignes, mais ils sont requis dans certaines requêtes comme les jointures entre une table et elle-même, ou pour donner un nom au résultat d'une *sous-requête* :

```
SELECT * FROM Employee e,
  (SELECT 1 AS teamid, 'R&D' AS teamname, 500 AS bonus UNION ALL
   SELECT 2 AS teamid, 'Sales' AS teamname, 600 AS bonus) AS t
WHERE e.deptno = t.deptno
```

Dans cette dernière requête, la sous-requête `UNION ALL` crée la table anonyme suivante :

teamid	teamname	bonus
1	R&D	500
2	Sales	600

qui est ensuite lié au nom `t` en utilisant un alias, et ce nom est alors utilisé dans la clause `WHERE` pour faire référence à la table.

La syntaxe simple de SQL est l'une des raisons de sa popularité, et de sa place comme le plus utilisé des langages de requêtes. La plupart des bases de données sont compatibles avec SQL, même celles qui n'ont pas un modèle de données directement adapté pour l'algèbre relationnelle. Par conséquent, SQL est un langage de base de données indispensable pour l'étude de solutions dont le but est de permettre aux programmeurs d'envoyer des requêtes aux bases de données.

A.3 Langages de programmation applicatifs

La majorité des applications orientées données sont écrites dans des langages de programmation impératifs. *Python* est utilisé en particulier pour les applications Web et l'apprentissage automatique. C'est un langage de programmation très populaire de part sa syntaxe simple et ses très nombreuses bibliothèques spécifiques à différents domaines, notamment pour l'apprentissage automatique, les algorithmes généraux, et les statistiques. *JavaScript* est très largement utilisé dans les applications Web. *R* est un langage créé pour les applications statistiques et d'analyse de données. *Java* est un langage de programmation généraliste très utilisé proposant de nombreuses bibliothèques pour le développement Web, l'apprentissage automatique, le traitement de textes, et plus encore.

Contrairement à la programmation déclarative des langages comme SQL, la programmation impérative demande aux programmeurs de décrire étape par étape la manière dont la machine doit générer le résultat voulu. Par exemple, l'opération de sélection dans un langage impératif serait écrite de cette manière en Python :

```
filteredTable = []
for employee in employees:
  if (employee['salary'] > 5000):
    filteredTable.append(employee)
```

Cependant, les langages de programmation modernes ont fait un effort pour implémenter certains aspects de la programmation *fonctionnelle*, permettant de rendre les applications orientées données moins techniquement détaillées, et écrites

de façon plus déclarative. Par exemple, il est possible d'écrire notre exemple en Python en utilisant des listes en compréhension [Kuh11] :

```
[employee for employee in employees if employee['salary'] > 5000]
```

Les programmes applicatifs peuvent utiliser des fonctionnalités impératives et fonctionnelles, et souvent un peu des deux. Cependant, la plupart des langages d'application sont d'abord impératifs, et en particulier, ils sont souvent plus efficaces à évaluer du code impératif.

De plus, la plupart des langages d'application (Python, R, Ruby, JavaScript, ...) sont *typés dynamiquement*, ce qui signifie que la sûreté du typage est vérifiée à l'exécution. Par exemple, un programme comme

```
(function (x) { return x; })(2, 3)
```

qui applique la fonction identité à deux arguments serait reconnu comme une erreur à son exécution (ou il le devrait, mais JavaScript ignore simplement le second argument dans ce cas ...).

A.4 Requêtes depuis des langages d'application

Comme expliqué plus tôt, la plupart des applications sont écrites dans des langages de programmation généralistes. Ces langages n'ont pas de moyen natif d'envoyer des requêtes à des bases de données, et la vaste majorité d'entre eux sont des langages impératifs, dont la syntaxe est très différente des langages de requêtes. De nombreuses solutions existent pour permettre aux programmeurs d'envoyer des requêtes aux bases de données depuis leurs langages de programmation. Dans cette section, nous décrivons des solutions existantes, et discutons leurs avantages et inconvénients.

A.4.1 JDBC

Java Database Connectivity (JDBC) [Cor16] est une *interface de programmation applicative* (API) qui fournit au langage de programmation Java un accès aux sources de données, incluant les bases de données.

L'Exemple A.1 est un exemple d'utilisation de JDBC dans un programme Java pour récupérer des données stockées dans une base.

Example A.1.

```
final Connection conn = ...
final Statement stmt = conn.createStatement();
final String query =
  "SELECT id, name, salary FROM employee WHERE salary > 2500";
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
  System.out.println(rs.getInt("ID") + " "
  + rs.getString("NAME") + " " + rs.getFloat("SALARY"));
}
```

Dans cet exemple, le programme récupère l'identifiant, le nom, et le salaire des employés dont le salaire est plus grand que 2500 provenant d'une table `employee` stockée dans une base de données.

Avec JDBC, l'utilisateur doit d'abord créer un objet `Connection` en utilisant des identifiants valides pour obtenir l'accès à une base cible, puis créer un objet `Statement` depuis l'objet de connexion pour envoyer une requête. Cette requête est écrite sous forme de chaîne de caractères dans le langage de requêtes de la base (SQL dans l'exemple). Les résultats sont représentés par un objet `ResultSet` contenant un curseur qui commence au début de l'ensemble résultat. `ResultSet` fournit une méthode `next()` qui déplace le curseur sur la ligne suivante, et des méthodes d'accès telles que `getInt()` qui renvoie la valeur d'un attribut dont le nom est donné en argument dans la ligne courante pointée par le curseur. Par exemple, `rs.getInt("ID")` accède à l'entier stocké dans la colonne nommée `"ID"` dans la ligne courante de l'ensemble des résultats `rs`.

Bien que JDBC soit populaire et simple d'utilisation pour des programmeurs experts dans le langage de requêtes de leur base cible, ce type de solution très courant a de nombreux défauts :

- Les programmeurs doivent maîtriser les langages de requêtes de toutes les bases de données ciblées.
- L'intégration de logique d'application est très limitée car la traduction d'expressions du langage d'application vers le langage d'une base de données est restreint aux valeurs de bases (chaînes de caractères, entiers, ...), ce qui force les programmeurs à décomposer des requêtes complexes en requêtes plus simples à envoyer aux bases de données, et à combiner les résultats dans l'application, ce qui entraîne plus de travail pour les programmeurs, de la duplication de code, et des performances potentiellement désastreuses.
- Les erreurs dans les requêtes, mêmes syntaxiques, sont seulement détectées à l'exécution car les outils des langages de programmation comme les systèmes

de types ne peuvent pas détecter d'erreurs dans des requêtes écrites dans des chaînes de caractères.

- Une attention toute particulière doit être apportée aux entrées utilisateur intégrées aux requêtes pour éviter les attaques par injection de code [HVO06].
- Des conversions de type doivent être utilisées pour traduire des valeurs de la base de données vers le langage d'application (avec JDBC, cela se fait en utilisant des méthodes comme `getInt`).
- Changer une base de données cible pour une autre base avec un langage de requête différent implique la réécriture de toutes les requêtes de l'application.
- D'un point de vue d'ingénierie, cette solution demande le développement d'une nouvelle interface pour chaque connexion entre un langage d'application et une base de données. Cela explique la multiplication et la diversité des solutions, même dans un seul langage d'application.

Cet ensemble de problèmes est connu dans la litérature sous le nom d'*impedance mismatch* entre la base de données et le langage d'applications [CM84]. D'autres solutions ont été proposées pour résoudre ces difficultés.

A.4.2 ORMs

Les *mappings objet-relationel* (en anglais *Object-Relational Mappers* ou ORMs) et les équivalents comme les *mappings objet-document* (en anglais *Object-Document Mappers* ou ODMs) sont des patrons de conception permettant de traduire et manipuler des données entre des systèmes de types incompatible dans des langages de programmation orientés objet. En représentant la source de données par un objet, ces patrons de conception permettent de s'abstraire de la source de données et de manipuler l'information directement dans le langage de programmation. Parmi ces ORMs, on trouve par exemple Hibernate [KBA+09] pour Java, ActiveRecord [ct17] pour Ruby, Doctrine [WV10] pour PHP (qui est également un ODM), ou encore Django [KMH07] pour Python.

Bien que la plupart de ces bibliothèques se basent sur des requêtes écritent dans des chaînes de caractères en SQL ou dans des langages cousins tels que les OQLs (HQL, DQL, JPQL, ...) [ASL89], des efforts ont été fait pour améliorer l'intégration des requêtes dans le langage d'application. Par exemple, au lieu d'écrire des requêtes dans des chaînes de caractères, Criteria pour Hibernate permet à l'utilisateur de créer un objet `CriteriaQuery` sur lequel des opérations telles que des sélections peuvent être appliquées en utilisant des méthodes de l'objet. En utilisant Criteria et Hibernate, l'Exemple A.1 peut être écrit dans le langage Java comme montré par l'Exemple A.2.

Example A.2.

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cr = cb.createQuery(Employee.class);
Root<Item> r = cr.from(Employee.class);
cr.multiselect(r.get("id"), r.get("name"), r.get("salary"))
  .where(cb.gt(r.get("salary"), 2500));

Query<Employee> query = session.createQuery(cr);
List<Employee> results = query.getResultList();
```

La requête est créée en utilisant des expressions de haut niveau, et elle n'est donc pas liée à un langage de requêtes particulier comme SQL, et sa syntaxe est vérifiée en utilisant le système de types du langage. Cependant, cette solution est toujours verbeuse ; elle implique la création d'une nouvelle bibliothèque pour chaque lien entre un langage et une base de données ; et son expressivité est fortement restreinte. De plus, cette solution demande au programmeur de dupliquer les schémas des tables de la base de données dans l'application sous forme de classes.

A.4.3 LINQ

Une considérable avancée dans le domaine est le framework *LINQ* [Mic] de Microsoft, un composant du framework *.NET* qui donne à ses langages la possibilité d'envoyer des requêtes. LINQ définit des requêtes comme un concept de première classe dans les langages .NET, ce qui permet aux programmeurs de définir des requêtes destinées à des bases de données écrites dans la syntaxe de leur langage. L'Exemple A.3 est l'équivalent de l'Exemple A.1 écrit dans le langage C# en utilisant LINQ.

Example A.3.

```
var results =
  from e in db.Employee
  where e.salary > 2500
  select new { id = e.id, name = e.name, salary = e.salary };
foreach (var e in results) {
  Console.WriteLine(e.id + " " + e.name + " " + e.salary);
}
```

Bien que LINQ ajoute de nouvelles constructions syntaxiques pour la création de requêtes comme les mots-clés `from`, `where`, et `select`, les expressions se

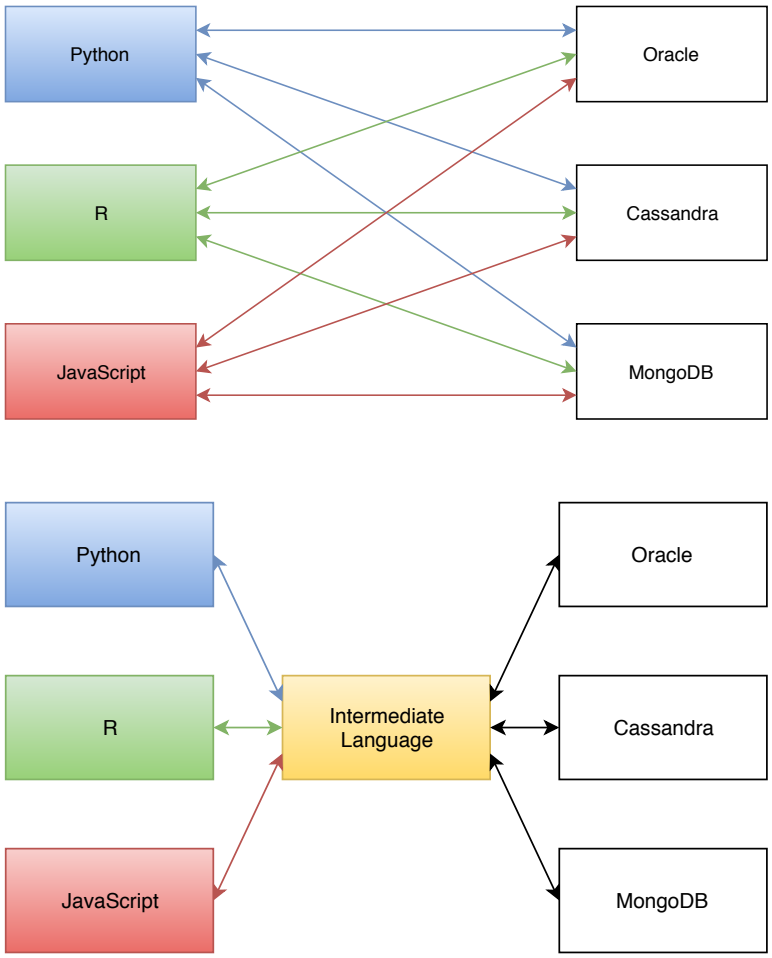


FIGURE A.4 – Bénéfices à haut niveau d'une représentation intermédiaire

trouvant dans les requêtes sont des expressions natives de C#. Par exemple, l'expression `new {...}` utilisée dans l'Exemple A.3 est le moyen natif en C# de créer un objet anonyme. En plus de cette syntaxe de requêtes, LINQ expose également une manière orientée objet d'exprimer une requête. Par exemple, la requête de l'Exemple A.3 peut être écrite :

```
db.Employee
.Where(e => e.salary > 2500)
.Select(e => new { id = e.id, name = e.name, salary = e.salary })
```

où `x => e` représente une fonction anonyme de paramètre x et de corps e.

Les requêtes exprimées en utilisant LINQ sont donc intégrées au langage de programmation et sûres du point de vue du typage. De plus, l'utilisation du framework .NET et d'un langage intermédiaire permet à un langage de programmation ou à une base de données de s'interfacer avec LINQ indépendamment.

En effet, comme le montre la Figure A.4, au lieu de créer une interface entre chaque langage d'application, ou *langage hôte*, et chaque base de données, cette approche ne demande aux langages de programmation et aux bases de données que de s'interfacer avec le langage intermédiaire. Par conséquent, les implémenteurs de langages de programmation et de bases de données ont seulement besoin de maîtriser leur langage et le langage intermédiaire pour s'interfacer avec le framework.

Cependant, toutes les requêtes ne peuvent pas être exécutées in LINQ, car seulement le code qui peut être traduit dans le langage intermédiaire de LINQ est accepté. Par conséquent, l'expressivité des requêtes est limitée dans ce framework. Par exemple, les requêtes en LINQ ne peuvent pas inclure des *fonctions définies par l'utilisateur* arbitraires (fonctions définies en utilisant la syntaxe du langage de programmation, en anglais user-defined functions ou UDFs). Par exemple, l'Exemple A.4 renvoie une erreur à l'exécution car LINQ échoue à traduire la fonction `dolToEuro` dans un équivalent dans le langage de la base de données. Cela n'est pas limité aux fonctions définies par l'utilisateur : toute expression qui ne peut pas être traduite est rejetée. C'est la responsabilité de l'implémenteur du *LINQ provider*, la partie de l'architecture de LINQ qui traduit les expressions C# vers des expressions d'un langage de requêtes, de gérer autant d'expressions du langage hôte que possible. LINQ offre peu d'aide à ce sujet et cette traduction est un point problématique majeur pour s'interfacer avec LINQ [Ein11].

Example A.4.

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee
.Where(e => e.salary > dolToEuro(2500));
```

Il existe deux moyens de contourner ce problème, et aucun n'est satisfaisant. Une solution est de répliquer manuellement la définition de `dolToEuro` dans la base de données, en tant que procédure stockée. Cette solution est particulièrement attrayante maintenant que les bases de données s'efforcent d'intégrer les langages d'application : Oracle R Enterprise [Orad], et PL/R [PL/a] pour R ; PL/Python [Pos], Amazon Redshift [Ama], Hive [Apab], et SPARK [Apac] pour Python ; ou MongoDB [Mon] et CQL de Cassandra [Apad] pour JavaScript. Cependant, cela implique la duplication de code exécutant de la logique d'application du côté de la base de donnée, ce qui cause d'important problèmes de maintenance, surtout pour les requêtes ciblant plus d'une base de données. Pire, une telle fonction pourrait même ne pas être traduisible du côté de la base de données, car elle pourrait utiliser des fonctionnalités non-supportées par la base, ou avoir besoin d'accéder à des valeurs présentes dans l'environnement d'exécution du langage d'application que le programmeur aurait alors à envoyer explicitement à la fonction à l'exécution, alourdissant sa définition avec des paramètres

supplémentaires.

Une autre solution est de récupérer les données dans l'application, puis d'appliquer les opérations. Cette solution semble être préférée des développeurs, car elle est syntaxiquement très simple avec LINQ :

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee
.AsEnumerable()
.Where(e => e.salary > dolToEuro(2500));
```

Ce programme est exécutable avec LINQ. Mais cette addition de l'appel à la méthode `AsEnumerable()` qui parait innocente cache de gros problèmes de performance : les données sont transférées dans l'environnement d'exécution du langage d'application par la méthode `Enumerable.AsEnumerable()`, ce qui pourrait résulter dans de très mauvaises performances à cause des retards causés par le transfert des données sur le réseau, et dans des erreurs de mémoire insuffisante. De plus, les opérations sur les données sont alors exécutées dans l'application, et donc n'utilisent pas les capacités d'optimisation fournies par les bases de données (par exemple en utilisant les index). Le même problème apparaît avec les requêtes entre différentes bases de données, puisque la solution en LINQ serait également d'exécuter ces requêtes côté application à l'aide d'appels explicites à `AsEnumerable()`.

Une solution pratique à ce problème est apporté par *T-LINQ* [CLW13], qui donnent des bases théoriques aux requêtes intégrées au langage en se basant sur des quotations de code et une normalisation des requêtes. Cette solution permet l'utilisation de fonctions définies par l'utilisateur dans des requêtes tant qu'il est possible de les traduire et de les intégrer directement dans les requêtes. Cependant, T-LINQ est restreint au modèle de données de SQL, ainsi qu'à une poignée d'opérations sur les données. Des implémentations existent dans de vrais langages comme C# qui font de leur mieux pour normaliser des requêtes contenant des fonctionnalités qui ne sont pas gérées par T-LINQ.

A.4.4 Apache Calcite

Apache Calcite [BCRH+18] est un framework de compilation de requêtes qui permet la manipulation de requêtes de manière indépendante des sources de données et des optimisations personnalisées sur des requêtes qui peuvent cibler différents types de bases de données. Calcite fournit aux implémenteurs de bases de données un framework unifié, incluant le support de langages de requêtes comme SQL, et des optimisations de requêtes. De plus, Calcite accepte des requêtes ciblant des sources de données hétérogènes en utilisant une abstraction relationnelle unifiée, et en choisissant les plans les plus efficaces pour l'évaluation des requêtes, en particulier en utilisant la migration de données pour exécuter des requêtes en-

tièrement dans les bases de données si possible. Calcite prend en entrée du SQL et du JDBC, et est donc limité dans l'expressivité des requêtes. Une syntaxe de requêtes intégrées au langage similaire à LINQ existe pour le langage de programmation Java, mais ce travail est préliminaire et ne gère pour l'instant que les aspects syntaxiques.

A.4.5 Autres interfaces

Dans le langage R, *RODBC* permet aux programmeurs d'envoyer des requêtes SQL aux bases de données d'une manière similaire à JDBC. *Dplyr* est une bibliothèque de manipulation de données pour R. *SparkR* fournit une interface pour *Apache Spark*. En Python, de nombreuses bibliothèques comme *pyodbc* ou *PySpark* donne accès à des bases de données, et *NumPy* permet la manipulation de grandes quantités de données.

Toutes ces interfaces sont similaires aux autres solutions présentées dans cette section, et partagent les mêmes problèmes. Plus de solutions encore sont décrites en Section 8.1.

A.5 Une nouvelle solution : BOLDR

Comme montré dans la Section A.4, les solutions existantes pour les applications orientées données ont toutes leur lot de problèmes. De plus, une application peut avoir besoin d'utiliser plusieurs de ces solutions pour accéder à différentes bases de données. Il faut une solution qui permet aux programmeurs d'écrire des requêtes dans leur langage de programmation, pouvant utiliser un maximum de fonctionnalités du langage, avec une interface unifiée permettant d'accéder à toutes les bases de données.

Dans cette thèse, nous créons une nouvelle solution appelée BOLDR (**B**reaking boundaries **O**f **L**anguage and **D**ata **R**epresentations), un framework de requêtes intégrées aux langages permettant aux développeurs d'application d'écrire des requêtes sûres, complexes, efficaces, et indépendantes des sources de données, dans le langage de programmation de leur choix.

A.5.1 Fonctionnalités

La Figure 1.5 donne une comparaison des fonctionnalités des différentes solutions. Dans un framework moderne de requêtes intégrées au langage, il nous faut toutes les fonctionnalités listées dans cette figure. Les requêtes doivent pouvoir : être exprimées dans le langage de l'application ; contenir de la logique d'application complexe ; cibler plusieurs bases de données en même temps ; et être vérifiées comme sûres avant exécution.

Fonctionnalités	BOLDR	T-LINQ	LINQ	Calcite	ORMs	JDBC
Création, envoi, et résultats de requêtes	✓	✓	✓	✓	✓	✓
Requêtes intégrées au langage	✓	✓	✓	✓	✓	✗
UDFs dans les requêtes	✓	✓	✗	✗	✗	✗
Export de l'environnement d'exécution	✓	✓/✗(1)	✓/✗(1)	✗	✗	✗
Différentes sources du même modèle	✓	✓/✗(2)	✗	✓	✗	✗
Différents modèles de données	✓	✗	✓	✓	✗	✓
Plusieurs interfaces disponibles	✓	✗	✓	✓	✓	✓
Exécution d'UDF dans la base	✓	✓/✗(3)	✓/✗(3)	✗	✗	✗
Fusion et normalisation de requêtes	✓	✓	✗	✗	✗	✗
Requêtes ciblant plus d'une base	✓	✗	✗	✓	✗	✗
Détection d'erreurs avant l'exécution	✓	✓	✓	✓	✓	✗
Bases théoriques	✓	✓	✗	✗	✗	✗

(1). Pour les types de base (2). Pas dans la même requête (3). Pour les UDFs inlinées

FIGURE A.5 – Fonctionnalités des différentes solutions

Tout comme LINQ, BOLDR utilise une représentation intermédiaire de requêtes appelée QIR pour **Q**uery **I**ntermediate **R**epresentation. En plus des atouts apportés par le fait d'avoir des interfaces indépendantes pour chaque langage hôte et chaque base de données, QIR réécrit les requêtes pour les rendre plus simple à traduire vers les langages de bases de données.

BOLDR *n'applique pas d'optimisations de plans de requêtes*. De manière générale, BOLDR optimise les requêtes QIR pour bénéficier autant que possible des optimisations fournies par les bases de données en utilisant l'information inutilisable par ces bases, et donc ne se substitue pas à leurs moteurs d'optimisation. Le but recherché est de générer des requêtes pouvant être les plus optimisées possible par les bases de données.

QIR permet à BOLDR de réaliser de la vérification de types sur les requêtes permettant de détecter des erreurs avant l'exécution. Par exemple, examinons cette requête écrite en R et utilisant BOLDR :

```
t = tableRef("people", "PostgreSQL")
q = query.filter(function (x) x$name > 5000, t)
```

Notez que le nom est comparé à un entier. Cette requête qui cible une base *PostgreSQL* est correcte syntaxiquement, mais renvoie une erreur à l'exécution à cause de cette comparaison incorrecte. En appliquant une vérification de types sur les requêtes QIR, BOLDR peut détecter l'erreur avant même que la requête soit traduite dans des langages de requêtes, et donc évite d'envoyer une requête invalide à la base de données par le réseau avec son lot de problèmes de performances.

De plus, BOLDR donne des garanties sur ses processus de manipulation de

requêtes, comme la terminaison de ses phases d'optimisation, et qu'une requête bien typée peut être traduite dans des langages de requêtes.

BOLDR définit les interfaces entre un langage hôte et le framework, ainsi qu'entre un langage de base de données et le framework. BOLDR n'est pas lié à une combinaison particulière de bases de données et de langages de programmation, et permet de cibler plusieurs bases à la fois. Par exemple, cette requête :

```
t1 = tableRef("people", "PostgreSQL")
t2 = tableRef("team", "HBase")
q = query.join(function (t1, t2) t1.teamid = t2.teamid, t1, t2)
```

est parfaitement valide avec BOLDR et réalise l'opération de *jointure* de l'algèbre relationnelle sur deux tables provenant de différentes bases de données, sans avoir à importer explicitement les données dans l'environnement d'exécution de l'application. Comme décrit plus tôt, BOLDR traduit automatiquement les sous-requêtes dans les différents langages de requêtes des différentes bases cibles, les envoie dans les bases en question, récupère les résultats et les traduits en retour vers le langage d'application.

Le framework permet également l'utilisation d'expressions arbitraires du langage hôte dans les requêtes. Par conséquent, notre Exemple A.4 en LINQ qui posait problème :

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee.Where(e => e.salary > dolToEuro(2500));
```

est également valide avec BOLDR. Le framework traduit la fonction à l'intérieur même de la requête si cela est possible et efficace, ou la conserve comme une fonction de langage d'application qui devra être évaluée plus tard dans la base de données ou dans l'application elle-même. Dans tous les cas, cette requête est évaluée avec succès. Ce procédé est entièrement automatisé, les programmeurs n'ont pas à migrer leur code d'application vers les bases de données.

A.5.2 Description détaillée

Le déroulement général de l'évaluation de requêtes avec BOLDR est décrit dans la Figure A.6. Durant l'exécution ① d'un programme hôte, les requêtes sont traduites en QIR puis évaluées par QIR ②. Ces deux étapes n'ont pas à être contiguës. Souvent, les requêtes sont traduites à leur création, mais évaluées seulement quand le programme a besoin des résultats. Le système d'évaluation de QIR prend alors le relais et essaie de typer les requêtes QIR. S'il réussit, il normalise ③ les termes QIR pour les défragmenter en utilisant une stratégie qui est garantie de réussir. Cette étape est essentielle pour permettre aux traductions vers les langages de bases de données de fonctionner de manière optimale. Si le typage

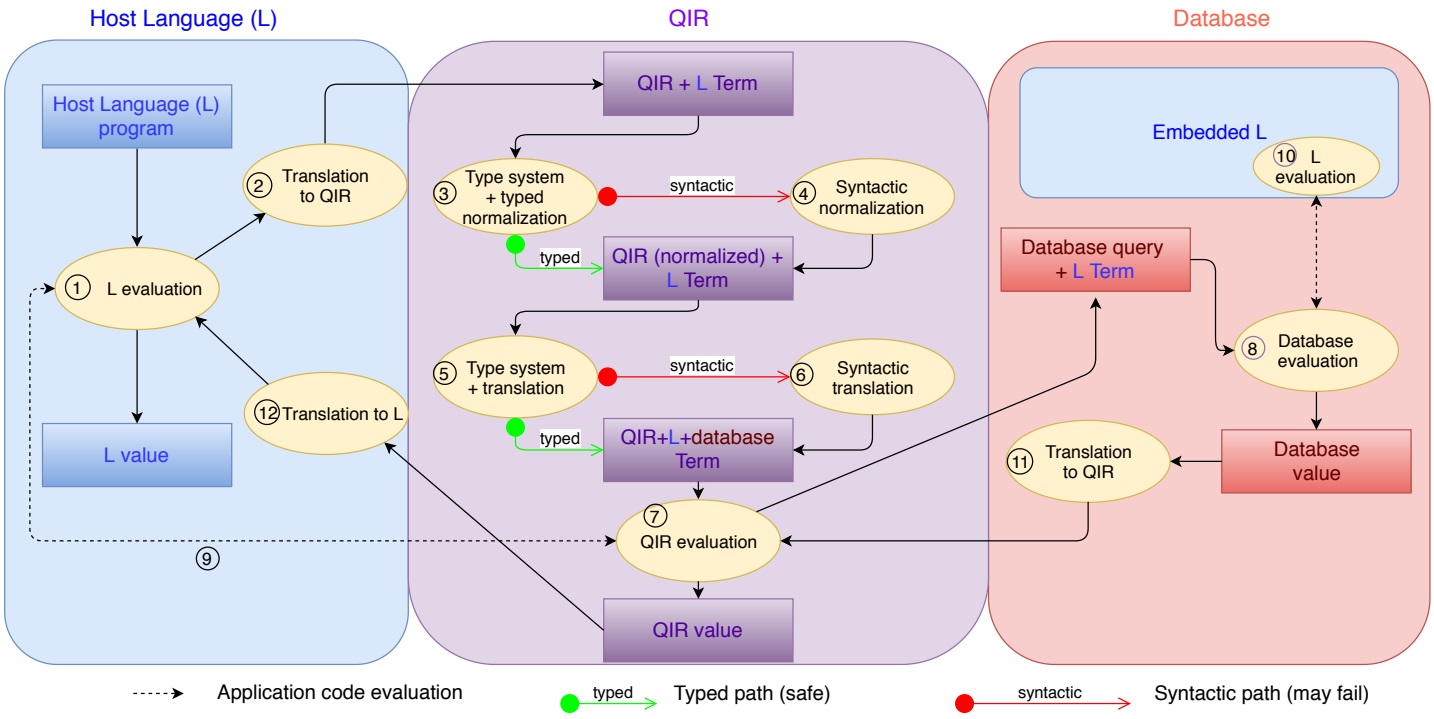


FIGURE A.6 – Évaluation d'un programme avec BOLDR

échoue, une stratégie basée sur la structure syntaxique des expressions QIR est utilisée ④ qui pourrait échouer. Les termes QIR sont ensuite typés de nouveau ⑤ pour obtenir des informations utiles à la traduction sur l'endroit où les sous-requêtes doivent être exécutées, mais aussi pour détecter des erreurs avant exécution et obtenir d'autres propriétés formelles intéressantes. Si cela réussit, BOLDR traduit les termes QIR dans de nouveaux termes QIR qui contiennent des requêtes écrites dans des langages de bases de données (par exemple SQL) en utilisant une stratégie de traduction également garantie de réussir. Sinon, une fois encore, c'est une stratégie syntaxique qui est utilisée ⑥ et qui pourrait échouer. Ensuite, les différentes parties de ces termes sont évaluées où elles doivent l'être, soit dans l'application ⑦ ou dans une base de données ⑧. Les expressions du langage hôte se trouvant dans ces termes sont évaluées soit par le système d'évaluation du langage hôte qui a appelé le QIR ⑨, ou dans le système d'évaluation intégré dans une base de données cible ⑩. Les résultats sont alors traduits de la base de données vers QIR ⑪, puis de QIR vers le langage hôte ⑫.

L'Exemple A.5 illustre les aspects clés de BOLDR.

Example A.5.

```
1  # Taux de change entre rfrom et rto
2  getRate = function(rfrom, rto) {
3    # La table change a trois colonnes: cfrom, cto, rate
4    change = tableRef("change", "PostgreSQL")
5    if (rfrom == rto) 1
6    else subset(change, cfrom == rfrom && cto == rto, c(rate))
7  }
8
9  # Employees gagnant au moins min dans la monnaie cur
10 atLeast = function(min, cur) {
11   # La table employee a deux colonnes: name, salary
12   emp = tableRef("employee", "PostgreSQL")
13   subset(emp, salary >= min * getRate("USD", cur), c(name))
14 }
15
16 richUSPeople = atLeast(2500, "USD")
17 richEURPeople = atLeast(2500, "EUR")
18 print(executeQuery(richUSPeople))
19 print(executeQuery(richEURPeople))
```

Notre Exemple A.5 est un programme R standard avec deux exceptions : la fonction `tableRef` qui renvoie une référence vers une table d'une source externe ; et la fonction `executeQuery` qui évalue une requête. Pour rappel, en R, la fonction `c` crée un vecteur, la fonction `subset` filtre une table en utilisant un prédicat, et optionnellement ne conserve que les colonnes spécifiées. La première fonction `getRate` prend le code de deux monnaies et envoie une requête en utilisant `subset` pour obtenir leur taux de change. La seconde fonction `atLeast` prend un salaire minimum et un code de monnaie et récupère les noms des employés gagnant au moins le salaire minimum. Puisque le salaire est stocké en dollars dans la base de données, la fonction `getRate` est utilisée pour faire la conversion.

Avec BOLDR, `subset` est surchargée pour fabriquer une requête QIR si elle est appliquée sur une référence de source externe. L'évaluation du premier appel à la fonction `atLeast` (`atLeast(2500, "USD")` à la Ligne 16) a pour résultat la création d'une requête obtenue par traduction de l'expression R vers QIR. Quand `executeQuery` est appelée sur la requête, alors (*i*) les valeurs de l'environnement d'exécution liées aux variables libres de la requête sont traduites en QIR, puis liées à ces variables dans la requête, créant ainsi une requête QIR *close* ; (*ii*) la requête est *normalisée*, procédé qui, en particulier, remplace les variables liées par leurs valeurs ; (*iii*) la requête normalisée est traduite vers le langage de base de données cible (ici SQL) ; et (*iv*) la requête résultante est évaluée dans la base

de données et les résultats sont récupérés. Après normalisation et traduction, la requête générée pour l'exécution de `richUSPeople` est :

```
SELECT name FROM employee WHERE sal >= 2500 * 1
```

ce qui est optimal, dans le sens où une seule requête SQL est générée. Le code généré pour `richEURPeople` est également optimal grâce à la combinaison entre la construction paresseuse de la requête et la normalisation :

```
SELECT name FROM employee WHERE sal >= 2500 *
  (SELECT rate FROM change WHERE rfrom = "USD" AND rto = "EUR")
```

Dans ce cas, BOLDR combine les sous-requêtes ensemble pour créer des requêtes moins nombreuses et plus larges, bénéficiant ainsi des optimisations des bases de données autant que possible et évitant le phénomène d'"avalanche de requêtes" [GRS10].

Les fonctions définies par l'utilisateur qui ne peuvent pas être traduites sont également supportées par BOLDR. Par exemple, regardons l'Exemple A.6.

Example A.6.

```
getRate = function(rfrom, rto) {
  print(rto)
  change = tableRef("change", "PostgreSQL")
  if (rfrom == rto) 1
  else subset(change, cfrom == rfrom && cto == rto, c(rate))
}
```

Cette version de la fonction `getRate` appelle la fonction `print` qui ne peut pas être traduite en QIR, donc BOLDR génère la requête suivante :

```
SELECT name FROM employee
WHERE sal >= 2500 * R.eval("@...", array("USD", "EUR"))
```

où la chaîne de caractères `"@..."` est une référence à une clôture pour `getRate`.

Mélanger différentes sources de données est supporté, mais moins efficacement. Par exemple, nous pourrions faire référence à une table d'une base HBase [Apaa] dans la fonction `getRate`. BOLDR serait toujours capable d'évaluer la requête en envoyant une sous-requête aux deux bases HBase et PostgreSQL, et en exécutant dans l'application ce qui ne peut pas être traduit.

A.5.3 Implémentation

Notre implémentation de BOLDR utilise *Truffle* [WWW+13, Wim14], un framework développé par Oracle Labs pour implémenter des langages de programmation. Truffle permet aux développeurs de langages d'implémenter des interpréteurs d'arbres de syntaxe abstraite (AST) avec de l'évaluation spéculative. Les implémenteurs de langages écrivent généralement un parseur pour le langage ciblé qui produit un AST composé de *nœuds Truffle*. Ces nœuds implémentent les opérations basiques de l'interpréteur (flot de contrôle, opérations typées sur les types primitifs, opérations du modèle objet comme le dispatch multiple, ...), et utilisent l'API Truffle pour implémenter de la spécialisation à l'exécution et informer le compilateur JIT des différents aspects d'optimisation, comme les profils d'exécution sur les valeurs, types, branches, ou pour implémenter de la réécriture à l'exécution de l'AST sur les chemins de dé-optimisation quand une optimisation spéculative a échoué.

Plusieurs fonctionnalités rendent Truffle attirant pour BOLDR. D'abord, les implémentations de langages en Truffle doivent compiler dans un arbre de syntaxe abstraite exécutable que BOLDR peut manipuler directement, ce qui, en particulier, fournit un moyen simple de traduire des requêtes en QIR. Ensuite, les langages implémentés avec Truffle peuvent être exécutés dans n'importe quelle machine virtuelle Java (en anglais, Java Virtual Machine ou JVM), bien que de meilleures performances peuvent être obtenues quand la machine virtuelle utilise le compilateur JIT Graal [DWM14], ce qui rend leur intégration comme langage externe très simple dans les bases de données écrites en Java (comme Cassandra, HBase, ...), et relativement simple dans les autres comme PostgreSQL. Par conséquent, cela nous donne la possibilité d'évaluer dans les bases de données n'importe quelle expression provenant de tout langage hôte implémenté par Truffle. Enfin, plusieurs langages de programmation sont déjà implémentés, avec des degrés variables de maturité, en utilisant le framework, comme Zippy pour Python [Orae] ; JRuby pour Ruby [Orac] ; FastR pour R [Oraa] ; ou Graal.js pour JavaScript [Orab], et le travail réalisé pour un langage Truffle peut facilement être réutilisé dans ces implémentations.

Notre implémentation supporte les bases de données *PostgreSQL*, *HBase* et *Hive*, ainsi que *FastR*, l'implémentation du langage R utilisant Truffle, et *SimpleLanguage* d'Oracle, un langage dynamique expérimental dont la syntaxe et les fonctionnalités sont inspirées par JavaScript (typé dynamiquement, orienté prototype avec des fonctions de haut niveau et un système de types avec seulement trois types primitifs : nombre, chaîne de caractères et booléen). SimpleLanguage est développé par Oracle Labs pour montrer les fonctionnalités de Truffle. Une description détaillée de notre implémentation se trouve au Chapitre 7.

A.6 Contributions

Cette thèse étudie la conception d'un framework de requêtes intégrées au langage avec une définition formelle ainsi qu'une implémentation de BOLDR et de ses différents composants. Le Chapitre 2 définit des notations et définitions utilisées dans tout le document, et le Chapitre 8 conclut en discutant les possibles extensions et améliorations. Les chapitres de cette thèse correspondent aux différentes parties du framework illustrées en Figure A.6 et sont décrits dans les sous-sections suivantes.

A.6.1 Représentation intermédiaire de requêtes (QIR)

Chapitre 3, pages 27-53

Le point central de BOLDR est sa représentation intermédiaire de requêtes appelée QIR. Comme expliqué plus tôt, une requête est d'abord traduite dans cette représentation avant d'être traduite en requête pour bases de données. Dans ce chapitre, nous définissons le langage et sa sémantique ⑦⑨, dont la sémantique des opérateurs de données implémentés dans les bases de données ; une base de données par défaut qui implémente d'important opérateurs de données pour supporter des requêtes qui ne peuvent pas être entièrement traduite dans les langages des bases ; et l'optimisation appliquée sur les requêtes avant traduction appelée la *normalisation de QIR* ④ qui transforme une requête pour la rendre plus simple à traduire vers un langage de base de données. En effet, notre exemple d'application d'une fonction définie par l'utilisateur :

```
Func<float, float> dolToEuro = x => x * 0.88f;
db.Employee.Where(e => e.salary > dolToEuro(2500));
```

n'est pas une expression qui peut être traduite telle quelle vers la plupart des langages de bases de données. En effet, ces langages ne sont généralement pas faits pour la définition et l'application de fonctions définies par l'utilisateur. En particulier, le langage SQL standard ne supporte pas cette fonctionnalité (bien qu'il soit possible de définir des procédures dont les corps sont strictement limités à des requêtes). Certaines bases de données supportent des extensions d'SQL (PL/SQL d'Oracle [PL/b], T-SQL de Microsoft [T-S], ...) qui permettent la définition et l'application de fonctions définies par l'utilisateur, mais cette fonctionnalité n'est pas très optimisée. Par conséquent, traduire cette requête directement résulterait soit en une erreur, forçant QIR à gérer la plupart de l'évaluation, ou en une requête inefficace. Pour ces raisons, nous voulons appliquer `dolToEuro` dans le QIR avant la traduction pour générer une requête efficace. De plus, nous définissons des *drivers* dont le rôle est d'interfacer QIR à un langage hôte ou à une base de données en fournissant des fonctions de traduction depuis et vers le QIR. Pour résumer, les contributions de ce chapitre sont :

- Une syntaxe et une sémantique pour QIR
- Une relation de réduction pour les expressions de base de QIR
- Une relation de réduction pour le QIR complet utilisant les interfaces vers les langages hôtes et les bases de données
- Une base de données par défaut incluant des implémentations par défaut de certains opérateurs de données
- Une procédure de normalisation garantie de terminer mais sans autres propriétés formelles

A.6.2 Système de types pour QIR

Chapitre 4, pages 55-79

L'évaluation de requêtes implique un échange d'informations avec les bases de données. Ce procédé peut être très coûteux, en fonction de la quantité de données, à cause du temps de calcul et de transfert par le réseau. Par conséquent, éviter d'envoyer des requêtes aux bases de données quand ce n'est pas nécessaire, en particulier quand les requêtes présentent des erreurs, est un gain de performance majeur. Les systèmes de types sont un moyen efficace et classique de détecter à l'avance les erreurs dans les programmes. Cependant, puisque BOLDR cible principalement des langages hôtes dynamiques, les expressions traduites vers QIR ne sont pas typées. Il est donc logique de définir un système de types fort pour le QIR pour détecter autant d'erreurs que possible avant l'exécution plutôt que de compter sur la détection d'erreurs des bases de données. De plus, BOLDR supporte les requêtes ciblant plusieurs bases de données, et des sémantiques différentes pour les opérateurs de données selon la base de données qui les évaluent. Supporter ces fonctionnalités demande d'être capable d'établir dans quelle base chaque sous-expression d'une requête doit être évaluée.

Dans ce chapitre, nous définissons un système de types pour QIR ③⑤ appelé le système de types *générique*. Notre système de types générique est extensible avec des systèmes de types fournis par les bases de données. Ceux-ci, appelés systèmes de types *spécifique*, permettent aux implémenteurs de bases de données de décrire quelles expressions elles supportent. À cause du nombre inconnu de bases de données interfacées avec BOLDR, et parce que les requêtes pourraient cibler plusieurs bases en même temps, ce modèle de processus générique faisant travailler ensemble des composants spécifiques fournis par les bases est fréquent dans cette thèse. Pour montrer comment la base de données peut fournir un système de types spécifique, ce chapitre définit également un système de types pour SQL, ainsi qu'un système de types pour notre base de données par défaut, et nous prouvons des propriétés de sûreté d'exécution obtenues à l'aide de nos systèmes de types.

A.6.3 Inférence de types pour QIR

Chapitre 5, pages 81-107

Les systèmes de types du Chapitre 4 sont faits pour faciliter les développements formels et pour la présentation. Cependant, ces systèmes ne sont pas algorithmiques, et ne sont donc pas directement utilisables pour une implémentation.

Dans ce chapitre, nous créons des algorithmes de typages ③⑤ en utilisant la résolution de contraintes de types, et prouvons que nos algorithmes de typage sont équivalents aux systèmes de types du Chapitre 4. Nous définissons également un algorithme de résolution de contraintes, et nous prouvons qu'il résout les contraintes générées par nos algorithmes de typage.

A.6.4 Évaluation orientée par les types

Chapitre 6, pages 109-128

Dans ce chapitre, nous utilisons nos systèmes de types pour définir une traduction d'expressions QIR vers des langages de bases de données. Tout comme pour notre système de types, notre traduction de QIR vers les langages de bases de données est composée de traductions spécifiques fournies par les bases de données, et d'une traduction générique qui utilise ces traductions spécifiques. Notre traduction utilise également notre système de types pour traduire autant de requêtes que possible dans les langages de bases de données, et laisser le reste pour évaluation dans notre base par défaut. Nous définissons une traduction syntaxique ⑥ qui se déclenche si le système de types échoue. De plus, nous définissons une traduction pour SQL et montrons que si notre système de types parvient à typer une expression QIR en utilisant notre système de types pour SQL alors cette expression est traduisible en SQL avec notre traduction. Enfin, nous définissons une normalisation orientée par les types ③.

A.6.5 Implémentation et expériences

Chapitre 7, pages 129-151

Dans ce chapitre, nous créons une interface entre le langage de programmation R et BOLDR en définissant une traduction de R vers QIR ②. Nous décrivons également notre prototype d'implémentation de BOLDR et présentons nos résultats qui montrent que BOLDR est capable de gérer efficacement la plupart des requêtes contenant des fonctions définies par l'utilisateur, obtenant donc des résultats au moins aussi bons que des requêtes écrites manuellement, et évalue des requêtes entre différentes bases de données ainsi que des requêtes contenant des expressions intraduisibles avec des performances convenables.

Publications

La syntaxe et la sémantique de QIR décrites dans le Chapitre 3, ainsi que certaines parties des Chapitres 4, 6, et 7 sont présentés dans [BCD+18].

Appendix B

Full proofs

Lemma 3.1. Let $q \in \mathsf{E}_{\mathtt{QIR}}$. Either q is in normal form, or $\exists q'.q \hookrightarrow q'$.

Proof. By case analysis on q:

- If $q = x$, then q is in normal form.
- If $q = \mathbf{fun}^x(x) \rightarrow q_1$, then either
  - q_1 is in normal form, in which case q is in normal form;
  - or $q_1 \hookrightarrow q_1'$, in which case rule (norm-fun-red) applies.
- If $q = q_1\ q_2$, then either
  - $q_1 \equiv \mathbf{fun}^x(x) \rightarrow q_3$ and q_2 is in normal form and pure, in which case rule (norm-app-β) applies;
  - $q_1 \equiv \mathbf{fun}^x(x) \rightarrow v$ and q_2 are in normal form and q_2 is not pure, in which case q is a normal form;
  - $q_1 \equiv op$ and q_2 are in normal form, in which case rule (app-op) applies;
  - or $q_1 \not\equiv \mathbf{fun}^x(x) \rightarrow v$ or op and q_2 are in normal form, in which case q is in normal form;
  - or either q_1 or q_2 is not in normal form, in which case rules (app-red1) or (app-red2) apply.
- If $q = c$, then q is in normal form.
- If $q = op$, then q is in normal form.

- If $q = \textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3$, then either
  - $q_1 = \textbf{true}$, in which case rule (app-true) applies;
  - or $q_1 = \textbf{false}$, in which case rule (app-false) applies;
  - or $q_1 \neq \textbf{true}$ or **false** is in normal form, in which case q is in normal form;
  - or q_1 is not in normal form, in which case (if-red) applies.
- If $q = \{\, l_i : q_i \,\}_{i=1..n}$, then either
  - all q_i are in normal form, in which case q is in normal form;
  - or at least one q_i is not in normal form, in which case rule (rec-red) applies.
- If $q = q_1 \bowtie q_2$, then either
  - $q_1 \equiv \{\, l : v, \ldots, l : v \,\}$ and $q_2 \equiv \{\, l : v, \ldots, l : v \,\}$ are in normal form, in which case rule (rconcat-rec) applies;
  - or q_1 and q_2 are in normal form and either $q_1 \not\equiv \{\, l : v, \ldots, l : v \,\}$ or $q_2 \not\equiv \{\, l : v, \ldots, l : v \,\}$, in which case q is in normal form;
  - or either q_1 or q_2 are not in normal form in which case either rule (rconcat-red1) or (rconcat-red2) apply.
- If $q = [\,]$, then q is in normal form.
- If $q = q_1 :: q_2$, then either
  - q_1 and q_2 are in normal form, in which case q is in normal form;
  - or either q_1 or q_2 are not in normal form in which case either rule (lcons-red1) or (lcons-red2) apply.
- If $q = q_1 \mathbin{@} q_2$, then either
  - $q_1 \equiv [\,]$ and q_2 are in normal form, in which case rule (lconcat-lempty) applies;
  - or q_1 and $q_2 \equiv [\,]$ are in normal form, in which case rule (lconcat-rempty) applies;
  - $q_1 \equiv v :: v$ and q_2 are in normal form, in which case rule (lconcat-lcons) applies;
  - or q_1 and q_2 are in normal form and $q_1 \not\equiv [\,]$ or $v :: v$ and $q_2 \not\equiv [\,]$, in which case q is in normal form;

  - or either q_1 or q_2 are not in normal form in which case either rule (lconcat-red1) or (lconcat-red2) apply.

- If $q = q \cdot l$, then either
  - $q \equiv \{\!\{ \ldots, l : v, \ldots \}\!\}$ is in normal form, in which case rule (rdestr-rec) applies;
  - or $q \not\equiv \{\!\{ \ldots, l : v, \ldots \}\!\}$ is in normal form, in which case q is in normal form;
  - or q is not in normal form, in which case rule (rdestr-red) applies.
- If $q = q_1$ **as** $h :: t$ **?** q_2 **:** q_3, then either
  - $q_1 = [\,]$, in which case rule (ldestr-empty) applies;
  - or $q_1 \equiv v_1 :: v_1'$, in which case rule (ldestr-nonempty) applies;
  - or $q_1 \not\equiv [\,]$ or $v_1 :: v_1'$ is in normal form, in which case q is in normal form;
  - or q_1 is not in normal form, in which case rule (ldestr-red) applies.
- If $q = o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n \rangle$, then either
  - all q_i are in normal form, in which case q is in normal form;
  - or at least one q_i is not in normal form, in which case either rule (dataop-conf) or (dataop-data) apply.

□

Theorem 4.1 (Subject reduction for MEM)**.** Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus \mathsf{MEM}$, we have subject reduction on $\hookrightarrow$. Then, we have subject reduction for MEM on $\hookrightarrow$.

Proof. If $\mathcal{D} \neq \mathsf{MEM}$, then the property is true by hypothesis. Suppose now that $\mathcal{D} = \mathsf{MEM}$. We prove the property by induction on the derivation of $\Gamma \vdash_{\mathsf{MEM}} q : T$, since it is the only possible step after $\Gamma \vdash q : T, \mathsf{MEM}$. We use L4.1 to denote Lemma 4.1, and P4.3 to denote Property 4.3.

- For all the (*-red*) and (dataop-*) rules, the property is immediately true by applying the induction hypothesis and the hypothesis that we have subject reduction for $\mathcal{D} \neq \mathsf{MEM}$.

- $(\mathbf{fun}^f(x){\rightarrow}q_1)\ v_2 \hookrightarrow \{f \mapsto \mathbf{fun}^f(x){\rightarrow}q_1, x \mapsto v_2\}q_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \{f \mapsto \mathbf{fun}^f(x){\rightarrow}q_1, x \mapsto v_2\}q_1 : T_2, \_}{\Gamma, f : T_1 \to T_2, x : T_1 \vdash_{\mathsf{MEM}} q_1 : T_2}\ \text{L4.1}}{\Gamma \vdash_{\mathcal{D}} \mathbf{fun}^f(x){\rightarrow}q_1 : T_1 \to T_2}\ \text{P4.3}}{\Gamma \vdash \mathbf{fun}^f(x){\rightarrow}q_1 : T_1 \to T_2, \mathcal{D}} \qquad \Gamma \vdash v_2 : T_1, \_}{\Gamma \vdash_{\mathsf{MEM}} (\mathbf{fun}^f(x){\rightarrow}q_1)\ v_2 : T_2}$$

  We used the substitution lemma twice here: once for f and once for x.

- $op\ v \to q$: true ensuring $\to^\delta$ preserves types.

- $\mathbf{if\,true\,then}\ q_1\ \mathbf{else}\ q_2 \hookrightarrow q_1$:

$$\dfrac{\Gamma \vdash \mathbf{true} : \mathtt{bool}, \_ \quad \Gamma \vdash q_1 : T, \_ \quad \Gamma \vdash q_2 : T, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathbf{if\,true\,then}\ q_1\ \mathbf{else}\ q_2 : T}$$

- $\mathbf{if\,false\,then}\ q_1\ \mathbf{else}\ q_2 \hookrightarrow q_2$:

$$\dfrac{\Gamma \vdash \mathbf{false} : \mathtt{bool}, \_ \quad \Gamma \vdash q_1 : T, \_ \quad \Gamma \vdash q_2 : T, \_}{\Gamma \vdash_{\mathsf{MEM}} \mathbf{if\,false\,then}\ q_1\ \mathbf{else}\ q_2 : T}$$

- $q_1 \bowtie q_2 \hookrightarrow \{l_i : v_i\}_{i \in 1..n}$ where $q_1 = \{l_i : v_i\}_{i \in 1..m}$ and $q_2 = \{l_i : v_i\}_{i \in m+1..n}$:

$$\dfrac{\Gamma \vdash q_1 : \{l_i : T_i\}_{i \in 1..m}, \_ \quad \Gamma \vdash q_2 : \{l_i : T_i\}_{i \in m+1..n}, \_}{\Gamma \vdash_{\mathsf{MEM}} q_1 \bowtie q_2 : \{l_i : T_i\}_{i \in 1..n}}$$

- $[\,] \mathbin{@} v \hookrightarrow v$:

$$\dfrac{\Gamma \vdash [\,] : T, \_ \quad \Gamma \vdash v : T, \_}{\Gamma \vdash_{\mathsf{MEM}} [\,] \mathbin{@} v : T}$$

- $v \mathbin{@} [\,] \hookrightarrow v$:

$$\dfrac{\Gamma \vdash v : T, \_ \quad \Gamma \vdash [\,] : T, \_}{\Gamma \vdash_{\mathsf{MEM}} v \mathbin{@} [\,] : T}$$

- $(v_1 :: v_2) \mathbin{@} v_3 \hookrightarrow v_1 :: (v_2 \mathbin{@} v_3)$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} v_1 : T \quad \Gamma \vdash_{\mathsf{MEM}} v_2 : T\ \mathbf{list}}{\Gamma \vdash_{\mathcal{D}} v_1 :: v_2 : T\ \mathbf{list}}\ \text{P4.3}}{\Gamma \vdash v_1 :: v_2 : T\ \mathbf{list}, \mathcal{D}} \qquad \Gamma \vdash v_3 : T\ \mathbf{list}, \_}{\Gamma \vdash_{\mathsf{MEM}} (v_1 :: v_2) \mathbin{@} v_3 : T\ \mathbf{list}}$$

so:

$$\dfrac{\Gamma \vdash v_1 : T, \_ \qquad \dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} v_2 : T\ \mathbf{list} \quad \Gamma \vdash_{\mathsf{MEM}} v_3 : T\ \mathbf{list}}{\Gamma \vdash_{\mathcal{D}} v_2 \mathbin{@} v_3 : T\ \mathbf{list}}\ \text{P4.3}}{\Gamma \vdash v_2 \mathbin{@} v_3 : T\ \mathbf{list}, \mathcal{D}}}{\Gamma \vdash_{\mathsf{MEM}} v_1 :: (v_2 \mathbin{@} v_3) : T\ \mathbf{list}}$$

- $\{\ldots, l : v, \ldots\} \cdot l \hookrightarrow v$:

$$\dfrac{\Gamma \vdash \{\ldots, l : v, \ldots\} : \{\ldots, l : T, \ldots\}, \_}{\Gamma \vdash_{\mathsf{MEM}} \{\ldots, l : v, \ldots\} \cdot l : T}$$

- $[\,]\ \mathbf{as}\ h :: t\ ?\ q_2\ :\ q_3 \hookrightarrow q_3$:

$$\dfrac{\Gamma \vdash [\,] : T_2\ \mathbf{list}, \_ \quad \Gamma \vdash q_2 : T_2 \to T_2\ \mathbf{list} \to T_1, \_ \quad \Gamma \vdash q_3 : T_1, \_}{\Gamma \vdash_{\mathsf{MEM}} [\,]\ \mathbf{as}\ h :: t\ ?\ q_2\ :\ q_3 : T_1}$$

- $v_1 :: v_1'\ \mathbf{as}\ h :: t\ ?\ q_2\ :\ q_3 \hookrightarrow q_2\ (v_1, v_1')$:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} v_1 : T_2 \quad \Gamma \vdash_{\mathsf{MEM}} v_1' : T_2\ \mathbf{list}}{\Gamma \vdash_{\_} v_1 :: v_1' : T_2\ \mathbf{list}}\ \text{P4.3}}{\Gamma \vdash v_1 :: v_1' : T_2\ \mathbf{list}, \_}$$

gives us:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash v_1 :: v_1' : T_2\ \mathbf{list}, \_} \quad \Gamma \vdash q_2 : T_2 \to T_2\ \mathbf{list} \to T_1, \_ \quad \Gamma \vdash q_3 : T_1, \_}{\Gamma \vdash_{\mathsf{MEM}} v_1 :: v_1'\ \mathbf{as}\ h :: t\ ?\ q_2\ :\ q_3 : T_1}$$

so:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash q_2 : T_2 \to T_2\ \textbf{list} \to T_1, \_ \quad \Gamma \vdash v_1 : T_2, \_}{\Gamma \vdash_{\_}\ q_2\ v_1 : T_2\ \textbf{list} \to T_1}}{\Gamma \vdash q_2\ v_1 : T_2\ \textbf{list} \to T_1, \_} \qquad \Gamma \vdash v_1' : T_2\ \textbf{list}, \_}{\Gamma \vdash_{\mathsf{MEM}} q_2\ (v_1, v_1') : T_1}$$

□

Theorem 4.2 (Progress)**.** Let $q \in \mathsf{E}_{\mathtt{QIR}}$, and $\mathcal{D}$ a database language. If $\emptyset \vdash q : T, \mathcal{D}$ and all data operators in q are translatable into a database language, then either q is a QIR value, or $\exists q'.q \twoheadrightarrow q'$.

Proof. By induction on typing derivations:

- If $q = x$, then impossible since the rule

  $$\dfrac{}{\Gamma, x : T \vdash_{\mathsf{MEM}} x : T}$$

  is not applicable since our environment is the empty set. And by Property 4.3, any typing rule for variables in other specific type systems cannot be applied either.

- If $q = \textbf{fun}^x(x) \to q'$, q is a value.

- If $q = q_1\ q_2$, then either

  - $q_1 \equiv \textbf{fun}^x(x) \to q_1'$ and q_2 is a value, in which case rule (app-β) applies;
  - $q_1 \equiv op$ and q_2 is a value, in which case rule (app-op) applies;
  - or $q_1 \not\equiv \textbf{fun}^x(x) \to q_1'$ or op, in which case q_1 is not a value by typing and can be reduced by induction hypothesis.

- If $q = c$, then q is a value.

- If $q = op$, then q is a value.

- If $q = \textbf{if}\, q_1\, \textbf{then}\, q_2\, \textbf{else}\, q_3$, then either

  - $q_1 = \textbf{true}$, in which case rule (app-true) applies;
  - or $q_1 = \textbf{false}$, in which case rule (app-false) applies;

  - or $q_1 \neq \mathbf{true}$ or $\mathbf{false}$, in which case q_1 is not a value by typing and can be reduced by induction hypothesis.

- If $q = \{\, l_i : q_i \,\}_{i=1..n}$, then either
  - all q_i are values, in which case q is a value;
  - or at least one q_i is not a value, in which case rule (rec-red) applies.

- If $q = q_1 \bowtie q_2$, then either
  - $q_1 \equiv \{\, l : v, \ldots, l : v \,\}$ and $q_2 \equiv \{\, l : v, \ldots, l : v \,\}$ are values, in which case rule (rconcat-rec) applies;
  - or either $q_1 \not\equiv \{\, l : v, \ldots, l : v \,\}$ or $q_2 \not\equiv \{\, l : v, \ldots, l : v \,\}$, in which case either q_1 or q_2 are not values by typing and can be reduced by induction hypothesis.

- If $q = [\,]$, then q is a value.

- If $q = q_1 :: q_2$, then either
  - q_1 and q_2 are values, in which case q is a value;
  - or either q_1 or q_2 are not a value by typing and can be reduced by induction hypothesis.

- If $q = q_1 \mathbin{@} q_2$, then either
  - $q_1 \equiv [\,]$ and q_2 is a value, in which case rule (lconcat-lempty) applies;
  - or q_1 is a value and $q_2 \equiv [\,]$, in which case rule (lconcat-rempty) applies;
  - $q_1 \equiv v :: v$ and q_2 is a value, in which case rule (lconcat-lcons) applies;
  - or either q_1 or q_2 are not a value by typing and can be reduced by induction hypothesis.

- If $q = q \cdot l$, then either
  - $q \equiv \{\, \ldots, l : v, \ldots \,\}$, in which case rule (rdestr-rec) applies;
  - or $q \not\equiv \{\, \ldots, l : v, \ldots \,\}$ is not a value by typing and can be reduced by induction hypothesis.

- If $q = q_1$ **as** $h :: t$ **?** q_2 **:** q_3, then either

- $q_1 = [\,]$, in which case rule (ldestr-empty) applies;
- or $q_1 \equiv v_1 :: v_1'$, in which case rule (ldestr-nonempty) applies;
- or $q_1 \not\equiv [\,]$ or $v_1 :: v_1'$, in which case q_1 is not a value by typing and can be reduced by induction hypothesis.

- If $q = o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n \rangle$, then either
  - all q_i are values, in which case (ext-database) applies by hypothesis;
  - or at least one q_i is not a value, in which case either rule (dataop-conf) or (dataop-data) apply.

□

Lemma 4.5. Let $q \in \mathsf{E}_{\mathtt{QIR}}$ and $v_1, \ldots, v_m \in \mathsf{V}_{\mathtt{QIR}}$ be closed QIR values. If there exists QIR typing environments $\Gamma_1, \ldots, \Gamma_m$ and database languages $\mathcal{D}$, $\mathcal{D}_1, \ldots, \mathcal{D}_m$ such that $x_1 : T_1, \ldots, x_m : T_m \vdash q : T, \mathcal{D}$ and $\forall j \in 1..m . \Gamma_j \vdash v_j : T_j, \mathcal{D}_j$ and $v_j \in R_{T_j}$, then $q\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$.

Proof. We note $\Gamma = \{x_j : T_j\}_{j=1..m}$. By induction on the derivation of $\Gamma \vdash q : T, \mathcal{D}$:

- If $q = x$, then $q = x_j$ and $T = T_j$, in which case the property is obviously true as if $v_j \in R_{T_j}$ then $x_j\{x_j/v_j\} = v_j \in R_{T_j}$.

- If $q = \mathbf{fun}(x) \rightarrow q_1$, then:

$$\dfrac{\dfrac{\Gamma, x : T' \vdash_{\mathsf{MEM}} q_1 : T''}{\Gamma \vdash_{\mathcal{D}} \mathbf{fun}(x) \rightarrow q_1 : T' \to T''}\ \text{P4.3}}{\Gamma \vdash \mathbf{fun}(x) \rightarrow q_1 : T' \to T'', \mathcal{D}}$$

  Let $q' \in R_{T'}$. By Lemma 4.2, we have $q' \hookrightarrow^* v$ for some v. By Lemma 4.3, $v \in R_{T'}$. By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m, x/v\} \in R_{T''}$. But, $(\mathbf{fun}(x) \rightarrow q_1\{x_1/v_1, \ldots, x_m/v_m\})\ (q') \hookrightarrow^* q_1\{x_1/v_1, \ldots, x_m/v_m, x/v\}$, which by Lemma 4.3 gives us $(\mathbf{fun}(x) \rightarrow q_1\{x_1/v_1, \ldots, x_m/v_m\})\ (q') \in R_{T''}$. Then, by definition of $R_{T' \to T''}$, since q' was chosen arbitrarily, we have: $(\mathbf{fun}(x) \rightarrow q_1)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T' \to T''}$.

- If $q = q_1\ q_2$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : T' \to T'' \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T'}{\Gamma \vdash_{\mathcal{D}} q_1\ q_2 : T' \to T''}\ \text{P4.3}}{\Gamma \vdash q_1\ q_2 : T' \to T'', \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T' \to T''}$ and $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'}$. And by definition of $R_{T' \to T''}$, we have $(q_1\ q_2)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T''}$.

- If $q = c$, then $T = \texttt{typeofC}(c) \in \mathcal{B}$, in which case the property is obviously true as c is obviously strongly normalizing therefore $c \in R_{\texttt{typeofC}(c)}$.

- If $q = op$, true by Lemma 4.4.

- If $q = \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : \texttt{bool} \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T \quad \Gamma \vdash_{\mathsf{MEM}} q_3 : T}{\Gamma \vdash_{\mathcal{D}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : T}\ \text{P4.3}}{\Gamma \vdash \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : T, \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\texttt{bool}}$, $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$, and $q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$. Therefore, by Lemma 4.2, we have $q_i\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* v_i'$ for $i \in 1..3$, and so $(\mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3)\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* \mathbf{if}\, v_1'\, \mathbf{then}\, v_2'\, \mathbf{else}\, v_3'$. If $v_1' = \texttt{true}$ or $\texttt{false}$, we have $\mathbf{if}\, v_1'\, \mathbf{then}\, v_2'\, \mathbf{else}\, v_3' \hookrightarrow^* v_2'$ or v_3'. But, by Lemma 4.3, since $q_i\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$, we have $v_i' \in R_T$ for $i \in 2..3$, and so by Lemma 4.3 again $\mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_T$. Otherwise, either $T \not\equiv T_1 \to T_2$, in which case the property is trivially true by Definition 4.15, or $T = T_1 \to T_2$, in which case the property is true by Lemma 4.4.

- If $q = \{\, l_i : q_i \,\}_{i=1..n}$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_i : T_i \quad i \in 1..n}{\Gamma \vdash_{\mathcal{D}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}}\ \text{P4.3}}{\Gamma \vdash \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}, \mathcal{D}}$$

  By induction hypothesis, we have $q_i\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_i}$ for $i \in 1..n$. So, by definition of $R_{\{l_i:T_i\}_{i=1..n}}$, we have $\{\, l_i : q_i \,\}_{i=1..n}\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\{l_i:T_i\}_{i=1..n}}$.

- If $q = q_1 \bowtie q_2$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : \{l_i : T_i\}_{i \in 1..m} \Gamma \vdash_{\mathsf{MEM}} q_2 : \{l_i : T_i\}_{i \in m+1..n}}{\Gamma \vdash_{\mathcal{D}} q_1 \bowtie q_2 : \{l_i : T_i\}_{i \in 1..n}}\ \text{P4.3}}{\Gamma \vdash q_1 \bowtie q_2 : \{l_i : T_i\}_{i \in 1..n}, \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\{l_i:T_i\}_{i \in 1..m}}$ and $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\{l_i:T_i\}_{i \in m+1..n}}$. So, by definition of $R_{\{l_i:T_i\}_{i=1..n}}$, we have $(q_1 \bowtie q_2)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\{l_i:T_i\}_{i \in 1..n}}$.

- If $q = [\,]$, then $T = T'\ \mathbf{list}$, in which case the property is obviously true as $[\,]$ is obviously strongly normalizing therefore $[\,] \in R_{T'\mathbf{list}}$.

- If $q = q_1 :: q_2$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : T' \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T'\ \mathbf{list}}{\Gamma \vdash_{\mathcal{D}} q_1 :: q_2 : T'\ \mathbf{list}}\ \text{P4.3}}{\Gamma \vdash q_1 :: q_2 : T'\ \mathbf{list}, \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'}$ and $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'\mathbf{list}}$. So, by definition of $R_{T'\mathbf{list}}$, we have $(q_1 :: q_2)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'\mathbf{list}}$.

- If $q = q_1 \mathbin{@} q_2$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : T'\ \mathbf{list} \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T'\ \mathbf{list}}{\Gamma \vdash_{\mathcal{D}} q_1 \mathbin{@} q_2 : T'\ \mathbf{list}}\ \text{P4.3}}{\Gamma \vdash q_1 \mathbin{@} q_2 : T'\ \mathbf{list}, \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'\ \mathbf{list}}$ and $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'\mathbf{list}}$. So, by definition of $R_{T'\mathbf{list}}$, we have $(q_1 \mathbin{@} q_2)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T'\mathbf{list}}$.

- If $q = q_1 \cdot l$, then:

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{MEM}} q_1 : \{\ldots, l : T, \ldots\}}{\Gamma \vdash_{\mathcal{D}} q_1 \cdot l : T}\ \text{P4.3}}{\Gamma \vdash q_1 \cdot l : T, \mathcal{D}}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{\{\ldots,l:T,\ldots\}}$, so $q_1\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow v_1'$.

If $v'_1 = \{\!\{ \ldots, l : v, \ldots \}\!\}$, by induction hypothesis on the typing derivation of the record, we have $v \in R_T$. But $v'_1 \cdot l \hookrightarrow v$. Therefore, by Lemma 4.3, $(q_1 \cdot l)\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_1}$.

Otherwise, either $T \not\equiv T_1 \to T_2$, in which case the property is trivially true by Definition 4.15, or $T = T_1 \to T_2$, in which case the property is true by Lemma 4.4.

- If $q = q_1$ **as** $h :: t$ **?** q_2 **:** q_3, then:

$$\frac{\Gamma \vdash_{\mathsf{MEM}} q_1 : T_2 \ \mathbf{list} \quad \Gamma \vdash_{\mathsf{MEM}} q_2 : T_2 \to T_2 \ \mathbf{list} \to T_1 \quad \Gamma \vdash_{\mathsf{MEM}} q_3 : T_1}{\dfrac{\Gamma \vdash_{\mathcal{D}} q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 : T_1}{\Gamma \vdash q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 : T_1, \mathcal{D}}}\ \text{P4.3}$$

  By induction hypothesis, we have $q_1\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_2\ \mathbf{list}}$, $q_2\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_2 \to T_2\ \mathbf{list} \to T_1}$, and $q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_1}$. Therefore, by Lemma 4.2, we have $q_i\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* v'_i$ for $i \in 1..3$, and so $(q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3)\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* v'_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ v'_2\ \mathbf{:}\ v'_3$.

  If $v'_1 = [\,]$, we have $v'_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ v'_2\ \mathbf{:}\ v'_3 \hookrightarrow^* v'_3$. But, by Lemma 4.3, since $q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_1}$, we have $v'_3 \in R_{T_1}$. Therefore, by Lemma 4.3 again, $q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_1}$.

  If $v'_1 = v :: v'$, we have $v'_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ v'_2\ \mathbf{:}\ v'_3 \hookrightarrow^* v'_2\ (v, v')$, and by induction hypothesis on the typing derivation of the list, we have $v \in R_{T_2}$ and $v' \in R_{T_2\mathbf{list}}$. Then, by definition of $R_{T_2 \to T_2\ \mathbf{list} \to T_1}$, $v'_2\ (v, v') \in R_{T_1}$. Therefore, by Lemma 4.3, $q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3\{x_1/v_1, \ldots, x_m/v_m\} \in R_{T_1}$.

  Otherwise, either $T_1 \not\equiv T_3 \to T_4$, in which case the property is trivially true by Definition 4.15, or $T_1 = T_3 \to T_4$, in which case the property is true by Lemma 4.4.

- If $q = o\langle q_1, \ldots, q_m \mid q_{m+1}, \ldots, q_n \rangle$, then $q\{x_1/v_1, \ldots, x_m/v_m\} \hookrightarrow^* o\langle v'_1, \ldots, v'_m \mid v'_{m+1}, \ldots, v'_n \rangle$, so by Lemma 4.3, we have to prove that $o\langle v_1, \ldots, v_m \mid v_{m+1}, \ldots, v_n \rangle \in R_T$. If $T \not\equiv T_1 \to T_2$ it is obviously true, otherwise if $T = T_1 \to T_2$ then it is true by Lemma 4.4 as $o\langle v'_1, \ldots, v'_m \mid v'_{m+1}, \ldots, v'_n \rangle$ is a value.

□

Theorem 5.1 (Soundness of the typing algorithm for MEM)**.** Let $q \in \mathsf{E}_{\mathtt{QIR}}$ and Γ a QIR typing environment. Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus \mathsf{MEM}$, $\vdash^{\mathcal{A}}_{\mathcal{D}}$ is

sound. Then, $\vdash^{\mathcal{A}}_{\mathsf{MEM}}$ is sound.

Proof. By induction on the typing derivation of $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K})$:

- $\Gamma, x : \mathsf{T} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \mathsf{T}, (\emptyset, \emptyset)$

  The property is immediately true since we have $\sigma\Gamma, x : \sigma T \vdash_{\mathsf{MEM}} x : \sigma T$ which is true for any σ.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow q : \alpha_1 \rightarrow \mathsf{T}, (\mathbb{C} \cup \{\alpha_2 = \mathsf{T}\}, \mathbb{K} \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C} \cup \{\alpha_2 = \mathsf{T}\}$ and for $\mathbb{K} \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}$ and $\mathbb{K}$ and $\sigma\alpha_2 = \sigma\mathsf{T}$. Since we have $\Gamma, f : \alpha_1 \rightarrow \alpha_2, x : \alpha_1 \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \_$, by induction hypothesis we get $\sigma\Gamma, f : \sigma\alpha_1 \rightarrow \sigma\alpha_2, x : \sigma\alpha_1 \vdash q : \sigma T, \_$, so $\sigma\Gamma, f : \sigma\alpha_1 \rightarrow \sigma T, x : \sigma\alpha_1 \vdash_{\mathsf{MEM}} q : \sigma T$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \rightarrow q : \sigma\alpha_1 \rightarrow \sigma T$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1\ q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \rightarrow \alpha\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \rightarrow \alpha\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$ and $\sigma\mathsf{T}_1 = \sigma\mathsf{T}_2 \rightarrow \sigma\alpha$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \sigma T_2 \rightarrow \sigma\alpha, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1\ q_2 : \sigma\alpha$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} c : \mathtt{typeofC}(c), (\emptyset, \emptyset)$

  The property is immediately true since we have $\sigma\Gamma \vdash_{\mathsf{MEM}} c : \mathtt{typeofC}(c) = \sigma\mathtt{typeofC}(c)$ for any σ.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} op : \mathtt{polytypeofOP}(op), (\emptyset, \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \ldots, \alpha_n \overset{\mathrm{k}}{=} \mathtt{U}\})$

  The property is immediately true since we have $\sigma\Gamma \vdash_{\mathsf{MEM}} op : T = \sigma(\mathtt{polytypeofOP}(op))$ using Property 5.1 since $TV(\sigma(\mathtt{polytypeofOP}(op))) = \emptyset$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \alpha_2, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\mathbb{K}_3$ and $\sigma\alpha_1 = \sigma\mathsf{T}_1 = \mathtt{bool}$, $\sigma\alpha_2 = \sigma\mathsf{T}_2$, $\sigma\alpha_2 = \sigma\mathsf{T}_3$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$, $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, and $\Gamma \vdash^{\mathcal{A}} q_3 : \mathsf{T}_3, (\mathbb{C}_3, \mathbb{K}_3), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$, $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, and $\sigma\Gamma \vdash q_3 :$

$\sigma T_3, \_$, so $\sigma\Gamma \vdash q_1 : \mathtt{bool}, \_$, $\sigma\Gamma \vdash q_2 : \sigma\alpha_2, \_$, and $\sigma\Gamma \vdash q_3 : \sigma\alpha_2, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \sigma\alpha_2$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : \mathsf{T}_i\}_{i=1..n}, (\bigcup_{i=1}^{n} \mathbb{C}_i, \bigcup_{i=1}^{n} \mathbb{K}_i)$

  Let σ be a solution for $\bigcup_{i=1}^{n} \mathbb{C}_i$ and for $\bigcup_{i=1}^{n} \mathbb{K}_i$, then σ is a solution for $\mathbb{C}_i$ and $\mathbb{K}_i$ for $i \in 1..n$. For $i \in 1..n$, since we have $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_i : \sigma T_i, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : \sigma T_i\}_{i=1..n}$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \bowtie q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} (\mathsf{T}_1, \mathsf{T}_2)\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} (\mathsf{T}_1, \mathsf{T}_2)\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, and $\sigma\mathsf{T}_1 = \{l_i : \mathsf{T}'_i\}_{i=1..m}$, $\sigma\mathsf{T}_2 = \{l_i : \mathsf{T}'_i\}_{i=m+1..n}$, $\sigma\alpha = \{l_i : \mathsf{T}'_i\}_{i=1..n}$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \{l_i : T'_i\}_{i=1..m}, \_$, $\sigma\Gamma \vdash q_2 : \{l_i : T'_i\}_{i=m+1..n}, \_$. Therefore, we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1 \bowtie q_2 : \{l_i : T'_i\}_{i=1..n} = \sigma\alpha$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} [\,] : \alpha\ \mathbf{list}, (\emptyset, \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

  The property is immediately true since we have $\sigma\Gamma \vdash_{\mathsf{MEM}} [\,] : T\ \mathbf{list} = \sigma T\ \mathbf{list}$ which is true for any σ.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 :: q_2 : \alpha\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\sigma\alpha = \sigma\mathsf{T}_1$, $\sigma\alpha\ \mathbf{list} = \sigma\mathsf{T}_2$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \sigma\alpha, \_$, and $\sigma\Gamma \vdash q_2 : \sigma\alpha\ \mathbf{list}, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1 :: q_2 : \sigma\alpha\ \mathbf{list}$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 @ q_2 : \alpha\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha\ \mathbf{list} = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha\ \mathbf{list} = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, and $\sigma\alpha\ \mathbf{list} = \sigma\mathsf{T}_1$ $\sigma\alpha\ \mathbf{list} = \sigma\mathsf{T}_2$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \sigma\alpha\ \mathbf{list}, \_$, and $\sigma\Gamma \vdash q_2 : \sigma\alpha\ \mathbf{list}, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1 @ q_2 : \sigma\alpha\ \mathbf{list}$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q \cdot l : \alpha_2, (\mathbb{C} \cup \{\alpha_1 = \mathsf{T}\}, \mathbb{K} \cup \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C} \cup \{\alpha_1 = \mathsf{T}\}$ and $\mathbb{K} \cup \{\alpha_1 \overset{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}$ and $\mathbb{K}$ and $\sigma\alpha_1 = \sigma\mathsf{T}$, $\sigma\alpha_1 \preceq \{l : \sigma\alpha_2\}$, so $\sigma\alpha_1 = \{\ldots, l : \sigma\alpha_2, \ldots\}$. Since we have $\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}, (\mathbb{C}, \mathbb{K}), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q : \sigma T, \_$, so $\sigma\Gamma \vdash q : \{\ldots, l : \sigma\alpha_2, \ldots\}, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q \cdot l : \sigma\alpha_2$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 : \alpha_2, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1\ \mathbf{list} = \mathsf{T}_1, \alpha_1 \to \alpha_1\ \mathbf{list} \to \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1\ \mathbf{list} = \mathsf{T}_1, \alpha_1 \to \alpha_1\ \mathbf{list} \to \alpha_2 = \mathsf{T}_2, \alpha_2 = \mathsf{T}_3\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\mathbb{K}_3$ and $\sigma\alpha_1\ \mathbf{list} = \sigma\mathsf{T}_1$, $\sigma\alpha_1 \to \sigma\alpha_1\ \mathbf{list} \to \sigma\alpha_2 = \sigma\mathsf{T}_2$, $\sigma\alpha_2 = \sigma\mathsf{T}_3$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$, $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, and $\Gamma \vdash^{\mathcal{A}} q_3 : \mathsf{T}_3, (\mathbb{C}_3, \mathbb{K}_3), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$, $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, and $\sigma\Gamma \vdash q_3 : \sigma T_3, \_$, so $\sigma\Gamma \vdash q_1 : \sigma\alpha_1\ \mathbf{list}, \_$, $\sigma\Gamma \vdash q_2 : \sigma\alpha_1 \to \sigma\alpha_1\ \mathbf{list} \to \sigma\alpha_2, \_$, and $\sigma\Gamma \vdash q_3 : \sigma\alpha_2, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} q_1\ \mathbf{as}\ h :: t\ \mathbf{?}\ q_2\ \mathbf{:}\ q_3 : \sigma\alpha_2$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Project}\langle q_1 \mid q_2\rangle : \alpha_2\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha_1 \to \alpha_2 = \mathsf{T}_1, \alpha_1\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha_1 \to \alpha_2 = \mathsf{T}_1, \alpha_1\ \mathbf{list} = \mathsf{T}_2\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, and $\sigma\alpha_1 \to \sigma\alpha_2 = \sigma\mathsf{T}_1\ \sigma\alpha_1\ \mathbf{list} = \sigma\mathsf{T}_2$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \sigma\alpha_1 \to \sigma\alpha_2, \_$, and $\sigma\Gamma \vdash q_2 : \sigma\alpha_1\ \mathbf{list}, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \mathtt{Project}\langle q_1 \mid q_2\rangle : \sigma\alpha_2\ \mathbf{list}$.

- $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Filter}\langle q_1 \mid q_2\rangle : \alpha\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha \to \mathtt{bool} = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha \to \mathtt{bool} = \mathsf{T}_1, \alpha\ \mathbf{list} = \mathsf{T}_2\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, and $\sigma\alpha \to \mathtt{bool} = \sigma\mathsf{T}_1$, $\sigma\alpha\ \mathbf{list} = \sigma\mathsf{T}_2$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$ and $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$ and $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, so $\sigma\Gamma \vdash q_1 : \sigma\alpha \to \mathtt{bool}, \_$, and $\sigma\Gamma \vdash q_2 : \sigma\alpha\ \mathbf{list}, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \mathtt{Filter}\langle q_1 \mid q_2\rangle : \sigma\alpha\ \mathbf{list}$.

$\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \texttt{Join}\langle q_1, q_2 \mid q_3, q_4\rangle : \alpha_3\ \mathbf{list}, (\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \mathbb{C}_4 \cup$

- $\{\alpha_1 \to \alpha_2 \to \alpha_3 = \mathsf{T}_1, \alpha_1 \to \alpha_2 \to \texttt{bool} = \mathsf{T}_2, \alpha_1\ \mathbf{list} = \mathsf{T}_3,$
  $\alpha_2\ \mathbf{list} = \mathsf{T}_4\}, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \mathbb{K}_4 \cup \{\alpha_1 \overset{\mathtt{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathtt{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathtt{k}}{=} \mathtt{U}\})$

  Let σ be a solution for $\mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \mathbb{C}_4 \cup \{\alpha_1 \to \alpha_2 \to \alpha_3 = \mathsf{T}_1, \alpha_1 \to \alpha_2 \to \texttt{bool} = \mathsf{T}_2, \alpha_1\ \mathbf{list} = \mathsf{T}_3, \alpha_2\ \mathbf{list} = \mathsf{T}_4\}$ and $\mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \mathbb{K}_4 \cup \{\alpha_1 \overset{\mathtt{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathtt{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathtt{k}}{=} \mathtt{U}\}$, then σ is a solution for $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{C}_4$, $\mathbb{K}_1$, $\mathbb{K}_2$, $\mathbb{K}_3$, and $\mathbb{K}_4$, and $\sigma\alpha_1 \to \sigma\alpha_2 \to \sigma\alpha_3 = \sigma\mathsf{T}_1$, $\sigma\alpha_1 \to \sigma\alpha_2 \to \texttt{bool} = \sigma\mathsf{T}_2$, $\sigma\alpha_1\ \mathbf{list} = \sigma\mathsf{T}_3$, $\sigma\alpha_2\ \mathbf{list} = \sigma\mathsf{T}_4$. Since we have $\Gamma \vdash^{\mathcal{A}} q_1 : \mathsf{T}_1, (\mathbb{C}_1, \mathbb{K}_1), \_$, $\Gamma \vdash^{\mathcal{A}} q_2 : \mathsf{T}_2, (\mathbb{C}_2, \mathbb{K}_2), \_$, $\Gamma \vdash^{\mathcal{A}} q_3 : \mathsf{T}_3, (\mathbb{C}_3, \mathbb{K}_3), \_$, $\Gamma \vdash^{\mathcal{A}} q_4 : \mathsf{T}_4, (\mathbb{C}_4, \mathbb{K}_4), \_$, by induction hypothesis we get $\sigma\Gamma \vdash q_1 : \sigma T_1, \_$, $\sigma\Gamma \vdash q_2 : \sigma T_2, \_$, $\sigma\Gamma \vdash q_3 : \sigma T_3, \_$, $\sigma\Gamma \vdash q_4 : \sigma T_4, \_$, so $\sigma\Gamma \vdash q_1 : \sigma\alpha_1 \to \sigma\alpha_2 \to \sigma\alpha_3, \_$, $\sigma\Gamma \vdash q_2 : \sigma\alpha_1 \to \sigma\alpha_2 \to \texttt{bool}, \_$, $\sigma\Gamma \vdash q_3 : \sigma\alpha_1\ \mathbf{list}, \_$, and $\sigma\Gamma \vdash q_4 : \sigma\alpha_2\ \mathbf{list}, \_$. Therefore we have $\sigma\Gamma \vdash_{\mathsf{MEM}} \texttt{Join}\langle q_1, q_2 \mid q_3, q_4\rangle : \sigma\alpha_3\ \mathbf{list}$.

□

Theorem 5.2 (Completeness of the typing algorithm of MEM)**.** Let $q \in \mathsf{E}_{\mathtt{QIR}}$ and Γ a QIR typing environment. Suppose that for all $\mathcal{D} \in \mathbb{D} \setminus \mathsf{MEM}$, $\vdash^{\mathcal{A}}_{\mathcal{D}}$ is complete. Then, $\vdash^{\mathcal{A}}_{\mathsf{MEM}}$ is complete.

Proof. By induction on the typing derivation of $\Gamma \vdash_{\mathsf{MEM}} q : T$:

- $\Gamma, x : T \vdash_{\mathsf{MEM}} x : T$

  Immediate since $\Gamma, x : \mathsf{T} \vdash^{\mathcal{A}}_{\mathsf{MEM}} x : \mathsf{T}, (\emptyset, \emptyset)$ by taking $\sigma = \emptyset$.

- $\Gamma \vdash_{\mathsf{MEM}} \mathbf{fun}^f(x) \to q : T_1 \to T_2$

  We have $\Gamma, f : T_1 \to T_2, x : T_1 \vdash q : T_2, \_$. By induction hypothesis, we get $\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2, x : \mathsf{T}_1 \vdash^{\mathcal{A}} q : \mathsf{T}', (\mathbb{C}, \mathbb{K}), \_$, and there exists σ' that satisfies $\mathbb{C}$ and $\mathbb{K}$ such that $\sigma'\mathsf{T}' = \mathsf{T}_2$. So by Property 5.2 $\Gamma, f : \alpha_1 \to \alpha_2, x : \alpha_1 \vdash^{\mathcal{A}} q : \mathsf{T}'', (\mathbb{C}', \mathbb{K}'), \_$, and $\sigma = \sigma' \circ \{\alpha_1 \mapsto \mathsf{T}_1, \alpha_2 \mapsto \mathsf{T}_2\}$ satisfies $\mathbb{C}'$ and $\mathbb{K}'$ and $\sigma\mathsf{T}'' = \sigma'\mathsf{T}' = \mathsf{T}_2$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{fun}^f(x) \to q : \alpha_1 \to \mathsf{T}'', (\mathbb{C}' \cup \{\alpha_2 = \mathsf{T}''\}, \mathbb{K}' \cup \{\alpha_1 \overset{\mathtt{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathtt{k}}{=} \mathtt{U}\})$, and σ satisfies $\mathbb{C}' \cup \{\alpha_2 = \mathsf{T}''\}$ and $\mathbb{K}' \cup \{\alpha_1 \overset{\mathtt{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathtt{k}}{=} \mathtt{U}\}$. And we have $\sigma(\alpha_1 \to \mathsf{T}'') = \mathsf{T}_1 \to \mathsf{T}_2$.

- $\Gamma \vdash_{\mathsf{MEM}} q_1\ q_2 : T_2$

  We have $\Gamma \vdash q_1 : T_1 \to T_2, \_$ and $\Gamma \vdash q_2 : T_1, \_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}_1 \to \mathsf{T}_2$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}_1$.

But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1\ q_2 : \alpha, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathsf{T}'_1 = \mathsf{T}'_2 \to \alpha\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \mathsf{T}_2\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, $\mathbb{K}_2$, so σ satisfies $\mathbb{K}$ and since α is fresh $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}_1$ and $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T}_1 \to \mathsf{T}_2 = \sigma\mathsf{T}'_2 \to \sigma\alpha$, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha = \mathsf{T}_2$.

- $\Gamma \vdash_{\mathsf{MEM}} c : \mathtt{typeofC}(c)$

  Immediate since $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{c} : \mathtt{typeofC}(c), (\emptyset, \emptyset)$ by taking $\sigma = \emptyset$.

- $\Gamma \vdash_{\mathsf{MEM}} op : T$

  Immediate since
  $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} op : \mathtt{polytypeofOP}(op), (\emptyset, \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \ldots, \alpha_n \overset{\mathrm{k}}{=} \mathtt{U}\})$ by Property 5.1.

- $\Gamma \vdash_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : T$

  We have $\Gamma \vdash q_1 : \mathtt{bool}, \_$, $\Gamma \vdash q_2 : T, \_$, and $\Gamma \vdash q_3 : T, \_$. By induction hypothesis, we get, for $i \in 1..3$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathtt{bool}$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}$, $\sigma_3\mathsf{T}'_3 = \mathsf{T}$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathbf{if}\, q_1\, \mathbf{then}\, q_2\, \mathbf{else}\, q_3 : \alpha_2, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1 = \mathsf{T}'_1, \alpha_1 = \mathtt{bool}, \alpha_2 = \mathsf{T}'_2, \alpha_2 = \mathsf{T}'_3\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 \circ \{\alpha_1 \mapsto \mathtt{bool}, \alpha_2 \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\mathbb{K}_3$, so σ satisfies $\mathbb{K}$ and since α_1 and α_2 are fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathtt{bool} = \sigma\alpha_1$, $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T} = \sigma\alpha_2$, $\sigma\mathsf{T}'_3 = \sigma_3\mathsf{T}'_3 = \mathsf{T} = \sigma\alpha_2$, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha_2 = \mathsf{T}$.

- $\Gamma \vdash_{\mathsf{MEM}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}$

  For $i \in 1..n$, we have $\Gamma \vdash q_i : T_i, \_$. By induction hypothesis, we get, for $i \in 1..n$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ such that $\sigma_i\mathsf{T}'_i = \mathsf{T}_i$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \{\, l_i : q_i \,\}_{i=1..n} : \{l_i : \mathsf{T}'_i\}_{i=1..n}, (\bigcup_{i=1}^{n} \mathbb{C}_i, \bigcup_{i=1}^{n} \mathbb{K}_i)$, and $\sigma_1 \circ \ldots \circ \sigma_n\{l_i : \mathsf{T}'_i\}_{i=1..n} = \{l_i : \mathsf{T}_i\}_{i=1..n}$.

- $\Gamma \vdash_{\mathsf{MEM}} q_1 \bowtie q_2 : \{l_i : T''_i\}_{i=1..n}$

  We have $\Gamma \vdash q_1 : \{l_i : T''_i\}_{i=1..m}, \_$ and $\Gamma \vdash q_2 : \{l_i : T''_i\}_{i=m+1..n}, \_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i)$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \{l_i : \mathsf{T}''_i\}_{i=1..m}$, $\sigma_2\mathsf{T}'_2 = \{l_i : \mathsf{T}''_i\}_{i=m+1..n}$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 \bowtie q_2 : \alpha, (\mathbb{C}_1 \cup \mathbb{C}_2, \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} (\mathsf{T}'_1, \mathsf{T}'_2)\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \{l_i : \mathsf{T}''_i\}_{i=1..n}\}$ satisfies $\mathbb{C}$ and $\mathbb{K}$ and since α is fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \{l_i : \mathsf{T}''_i\}_{i=1..m}$ and $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \{l_i : \mathsf{T}''_i\}_{i=m+1..n}$. And we have $\sigma\alpha = \{l_i : \mathsf{T}''_i\}_{i=1..n}$.

- $\Gamma \vdash_{\mathsf{MEM}} [\,] : T$ **list**

  Immediate since $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} [\,] : \alpha$ **list**$, (\emptyset, \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$ by taking $\sigma = \{\alpha \mapsto \mathsf{T}\}$.

- $\Gamma \vdash_{\mathsf{MEM}} q_1 :: q_2 : T$ **list**

  We have $\Gamma \vdash q_1 : T, \_$ and $\Gamma \vdash q_2 : T$ **list**$, \_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}$ **list**. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 :: q_2 : \alpha$ **list**$, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha = \mathsf{T}'_1, \alpha$ **list** $= \mathsf{T}'_2\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, so σ satisfies $\mathbb{K}$ and since α is fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T} = \sigma\alpha$ and $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}$ **list** $= \sigma\alpha$ **list**, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha$ **list** $= \mathsf{T}$ **list**.

- $\Gamma \vdash_{\mathsf{MEM}} q_1 @ q_2 : T$ **list**

  We have $\Gamma \vdash q_1 : T$ **list**$, \_$ and $\Gamma \vdash q_2 : T$ **list**$, \_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}$ **list**$, \sigma_2\mathsf{T}'_2 = \mathsf{T}$ **list**. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1 @ q_2 : \alpha$ **list**$, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha$ **list** $= \mathsf{T}'_1, \alpha$ **list** $= \mathsf{T}'_2\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \stackrel{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, so σ satisfies $\mathbb{K}$ and since α is fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T}$ **list** $= \sigma\alpha$ **list** and $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}$ **list** $= \sigma\alpha$ **list**, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha$ **list** $= \mathsf{T}$ **list**.

- $\Gamma \vdash_{\mathsf{MEM}} q \cdot l : T$

  We have $\Gamma \vdash q : \{\ldots, l : T, \ldots\}, \_$. By induction hypothesis, we get $\Gamma \vdash^{\mathcal{A}} q : \mathsf{T}', (\mathbb{C}, \mathbb{K}), \_$, and there exists σ' that satisfies $\mathbb{C}$ and $\mathbb{K}$ such that $\sigma'\mathsf{T}' = \{\ldots, l : \mathsf{T}, \ldots\}$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q \cdot l : \alpha_2, (\mathbb{C} = \mathbb{C}' \cup \{\alpha_1 = \mathsf{T}'\}, \mathbb{K} = \mathbb{K}' \cup \{\alpha_1 \stackrel{\mathrm{k}}{=} \{\{l : \alpha_2\}\}, \alpha_2 \stackrel{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma' \circ \{\alpha_1 \mapsto \{\ldots, l : \mathsf{T}, \ldots\}, \alpha_2 \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}'$ and $\mathbb{K}'$, and since α is fresh $\sigma\mathsf{T}' = \sigma'\mathsf{T}' = \{\ldots, l : \mathsf{T}, \ldots\} = \{\ldots, l : \sigma\alpha_2, \ldots\} = \sigma\alpha_1$, so since $\{\ldots, l : \sigma\alpha_2, \ldots\} \preceq \{l : \sigma\alpha_2\}$, σ satisfies $\mathbb{C}$ and $\mathbb{K}$. And we have $\sigma\alpha_2 = \mathsf{T}$.

- $\Gamma \vdash_{\mathsf{MEM}} q_1$ **as** $h :: t$ **?** q_2 **:** $q_3 : T_1$

  We have $\Gamma \vdash q_1 : T_2$ **list**$, \_$, $\Gamma \vdash q_2 : T_2 \to T_2$ **list** $\to T_1, \_$, and $\Gamma \vdash q_3 : T_1, \_$. By induction hypothesis, we get, for $i \in 1..3$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}_2$ **list**$, \sigma_2\mathsf{T}'_2 = \mathsf{T}_2 \to \mathsf{T}_2$ **list** $\to \mathsf{T}_1$, and $\sigma_3\mathsf{T}'_3 = \mathsf{T}_1$. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} q_1$ **as** $h :: t$ **?** q_2 **:** $q_3 : \alpha_2, (\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\alpha_1$ **list** $=$

$\mathsf{T}'_1, \alpha_1 \to \alpha_1$ **list** $\to \alpha_2 = \mathsf{T}'_2, \alpha_2 = \mathsf{T}'_3\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 \circ \{\alpha_1 \mapsto \mathsf{T}_2, \alpha_2 \mapsto \mathsf{T}_1\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{K}_1$, $\mathbb{K}_2$, and $\mathbb{K}_3$, so σ satisfies $\mathbb{K}$ and since α_1 and α_2 are fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T}_2$ **list** $= \sigma\alpha_1$ **list**, $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}_2 \to \mathsf{T}_2$ **list** $\to \mathsf{T}_1 = \sigma\alpha_1 \to \sigma\alpha_1$ **list** $\to \sigma\alpha_2$, $\sigma\mathsf{T}'_3 = \sigma_3\mathsf{T}'_3 = \mathsf{T}_1 = \sigma\alpha_2$, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha_2 = \mathsf{T}_1$.

- $\Gamma \vdash_{\mathsf{MEM}} \mathtt{Project}\langle q_1 \mid q_2\rangle : T_1$ **list**

  We have $\Gamma \vdash q_1 : T_2 \to T_1, \_$ and $\Gamma \vdash q_2 : T_2$ **list**, $\_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}_2 \to \mathsf{T}_1$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}_2$ **list**. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Project}\langle q_1 \mid q_2\rangle : \alpha_2$ **list**, $(\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha_1 \to \alpha_2 = \mathsf{T}'_1, \alpha_1$ **list** $= \mathsf{T}'_2\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha_1 \mapsto \mathsf{T}_2, \alpha_2 \mapsto \mathsf{T}_1\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, so σ satisfies $\mathbb{K}$ and since α_1 and α_2 are fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T}_2 \to \mathsf{T}_1 = \sigma\alpha_1 \to \sigma\alpha_2$ and $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}_2$ **list** $= \sigma\alpha_1$ **list**, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha_2$ **list** $= \mathsf{T}_1$ **list**.

- $\Gamma \vdash_{\mathsf{MEM}} \mathtt{Filter}\langle q_1 \mid q_2\rangle : T$ **list**

  We have $\Gamma \vdash q_1 : T \to \mathtt{bool}, \_$ and $\Gamma \vdash q_2 : T$ **list**, $\_$. By induction hypothesis, we get, for $i \in 1..2$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T} \to \mathtt{bool}$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}$ **list**. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Filter}\langle q_1 \mid q_2\rangle : \alpha$ **list**, $(\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\alpha \to \mathtt{bool} = \mathsf{T}'_1, \alpha$ **list** $= \mathsf{T}'_2\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \{\alpha \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \{\alpha \mapsto \mathsf{T}\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{K}_1$, and $\mathbb{K}_2$, so σ satisfies $\mathbb{K}$ and since α is fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T} \to \mathtt{bool} = \sigma\alpha \to \mathtt{bool}$ and $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}$ **list** $= \sigma\alpha$ **list**, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha$ **list** $= \mathsf{T}$ **list**.

- $\Gamma \vdash_{\mathsf{MEM}} \mathtt{Join}\langle q_1, q_2 \mid q_3, q_4\rangle : T_1$ **list**

  We have $\Gamma \vdash q_1 : T_3 \to T_4 \to T_1, \_$, $\Gamma \vdash q_2 : T_3 \to T_4 \to \mathtt{bool}, \_$, $\Gamma \vdash q_3 : T_3$ **list**, $\_$, and $\Gamma \vdash q_4 : T_4$ **list**, $\_$. By induction hypothesis, we get, for $i \in 1..4$, $\Gamma \vdash^{\mathcal{A}} q_i : \mathsf{T}'_i, (\mathbb{C}_i, \mathbb{K}_i), \_$, and there exists σ_i that satisfies $\mathbb{C}_i$ and $\mathbb{K}_i$ such that $\sigma_1\mathsf{T}'_1 = \mathsf{T}_3 \to \mathsf{T}_4 \to \mathsf{T}_1$, $\sigma_2\mathsf{T}'_2 = \mathsf{T}_3 \to \mathsf{T}_4 \to \mathtt{bool}$, $\sigma_3\mathsf{T}'_3 = \mathsf{T}_3$ **list**, $\sigma_4\mathsf{T}'_4 = \mathsf{T}_4$ **list**. But then we have $\Gamma \vdash^{\mathcal{A}}_{\mathsf{MEM}} \mathtt{Join}\langle q_1, q_2 \mid q_3, q_4\rangle : \alpha_3$ **list**, $(\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \mathbb{C}_4 \cup \{\alpha_1 \to \alpha_2 \to \alpha_3 = \mathsf{T}'_1, \alpha_1 \to \alpha_2 \to \mathtt{bool} = \mathsf{T}'_2, \alpha_1$ **list** $= \mathsf{T}'_3, \alpha_2$ **list** $= \mathsf{T}'_4\}, \mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2 \cup \mathbb{K}_3 \cup \mathbb{K}_4 \cup \{\alpha_1 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_2 \overset{\mathrm{k}}{=} \mathtt{U}, \alpha_3 \overset{\mathrm{k}}{=} \mathtt{U}\})$. $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 \circ \sigma_4 \circ \{\alpha_1 \mapsto \mathsf{T}_3, \alpha_2 \mapsto \mathsf{T}_4, \alpha_3 \mapsto \mathsf{T}_1\}$ satisfies $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_3$, $\mathbb{C}_4$ $\mathbb{K}_1$, $\mathbb{K}_2$, $\mathbb{K}_3$ and $\mathbb{K}_4$, so σ satisfies $\mathbb{K}$ and since α_1, α_2 and α_3 are fresh $\sigma\mathsf{T}'_1 = \sigma_1\mathsf{T}'_1 = \mathsf{T}_3 \to \mathsf{T}_4 \to \mathsf{T}_1 = \sigma\alpha_1 \to \sigma\alpha_2 \to \sigma\alpha_3$, $\sigma\mathsf{T}'_2 = \sigma_2\mathsf{T}'_2 = \mathsf{T}_3 \to \mathsf{T}_4 \to \mathtt{bool} = \sigma\alpha_1 \to \sigma\alpha_2 \to \mathtt{bool}$, $\sigma\mathsf{T}'_3 = \sigma_3\mathsf{T}'_3 = \mathsf{T}_3$ **list** $= \sigma\alpha_1$ **list**, and

$\sigma\mathsf{T}'_4 = \sigma_4\mathsf{T}'_4 = \mathsf{T}_4$ **list** $= \sigma\alpha_2$ **list**, so σ satisfies $\mathbb{C}$. And we have $\sigma\alpha_3$ **list** $= \mathsf{T}_1$ **list**.

□

Lemma 6.1 (Relation between SQL types and syntactic forms)**.** Let v be a normal form of QIR and Γ a QIR typing environment such that $\forall x \in \texttt{dom}(\Gamma).\Gamma(x) \equiv R$, and $\Gamma \vdash v : T, \mathsf{SQL}$, then:

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

Proof. The only valid rule for $\Gamma \vdash v : T, \mathsf{SQL}$ being the first one where $\mathcal{D} = \mathsf{SQL}$, we have to prove the property for $\Gamma \vdash_{\mathsf{SQL}} v : T$.

We prove the property by structural induction on the typing derivation of $\Gamma \vdash_{\mathsf{SQL}} v : T$. If the last rule used is the subsumption rule, then it is immediately true by induction hypothesis, otherwise we proceed by case analysis on T:

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

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

- If $T \equiv B$ then
  - If $v = x$ then impossible since $\Gamma, x : T \vdash_{\mathsf{SQL}} x : T \equiv R$ by Hypothesis H2
  - If $v = \mathbf{fun}^f(x) \to v'$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{fun}^f(x) \to v' : T_1 \to T_2$

- If $v = v_1\ v_2$ then by the typing rule of the application: $\Gamma \vdash_{\mathsf{SQL}} v_1 : T_1 \to T_2$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to v_3$ which is impossible by Hypothesis H1, or $v_1 \equiv op$, then by the typing rule of operators: $\Gamma \vdash_{\mathsf{SQL}} v_2 : B$, so by induction hypothesis $v_2 \equiv b$ so $v = op\ v_2 \equiv op\ b \equiv b$
- If $v = c$ then $v \equiv b$
- If $v = op$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \to \ldots \to B_n \to B$
- If $v = \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3$ then by the typing rule of the conditional expression:
  $\forall i \in 1..3, \Gamma \vdash_{\mathsf{SQL}} v_i : B_i$, 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 $v = \{\, l_i : v_i \,\}_{i=1..n}$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \{\, l_i : v_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}$
- If $v = [\,]$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} [\,] : T$ **list**
- If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 :: v_2 : T$ **list**
- If $v = v_1 \mathbin{@} v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 \mathbin{@} v_2 : T$ **list**
- If $v = v' \cdot l$ then by the typing rule of the record destructor: $\Gamma \vdash_{\mathsf{SQL}} v' : \{l_i : T_i\}_{i=1..n}$, 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
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{Project}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{From}\langle \mathcal{D}, v' \rangle$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{From}\langle \mathcal{D}, v' \rangle : R$ **list**
- If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{Filter}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle : R$ **list**
- If $v = \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle : R$ **list**
- If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{Sort}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Limit}\langle v_1 \mid v_2 \rangle$ then impossible since
  $\Gamma \vdash_{\mathsf{SQL}} \texttt{Limit}\langle v_1 \mid v_2 \rangle : R$ **list**

- If $v = \texttt{Exists}\langle v' \rangle$ then by the typing rule of $\texttt{Exists}$: $\Gamma \vdash_{\mathsf{SQL}} v' : R$ **list**, so by induction hypothesis $v' \equiv s$, so $v = \texttt{Exists}\langle v' \rangle \equiv \texttt{Exists}\langle s \rangle \equiv b$

- If $T \equiv R$ then
  - If $v = x$ then $v \equiv r$
  - If $v = \mathbf{fun}^f(x) \rightarrow v'$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{fun}^f(x) \rightarrow v' : T_1 \rightarrow T_2$
  - If $v = v_1\ v_2$ then impossible since by the typing rule of the application: $\Gamma \vdash_{\mathsf{SQL}} v_1 : T_1 \rightarrow T_2$, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \rightarrow v_3$ which is impossible by Hypothesis H1, or $v_1 \equiv op$, which is impossible as $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \rightarrow \ldots \rightarrow B_n \rightarrow B$
  - If $v = c$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} c : \texttt{typeof}(c) \equiv B$
  - If $v = op$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \rightarrow \ldots \rightarrow B_n \rightarrow B$
  - If $v = \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3 : B$
  - If $v = \{\, l_i : v_i \,\}_{i=1..n}$ then by the typing rule of the record constructor $\forall i \in 1..n, \Gamma \vdash_{\mathsf{SQL}} v_i : B_i$, so by induction hypothesis $\forall i \in 1..n, v_i \equiv b$, so $v = \{\, l_i : v_i \,\}_{i=1..n} \equiv \{\, l : b, \ldots, l : b \,\} \equiv r$
  - If $v = [\,]$ then impossible since $[\,]$ cannot be typed
  - If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 :: v_2 : R$ **list**
  - If $v = v_1 \mathbin{@} v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 \mathbin{@} v_2 : R$ **list**
  - If $v = v' \cdot l$ then by the typing rule of the record destructor: $\Gamma \vdash_{\mathsf{SQL}} v' : \{l_1 : T_1, \ldots, l_n : T_n\}$, 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 $\Gamma, x : T \vdash_{\mathsf{SQL}} v' \equiv x : T \equiv R'$, so impossible since by the typing rule of the record destructor $\Gamma \vdash_{\mathsf{SQL}} v = v' \cdot l : B$
  - If $v = \texttt{Project}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Project}\langle v_1 \mid v_2 \rangle : R$ **list**
  - If $v = \texttt{From}\langle \mathcal{D}, v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{From}\langle \mathcal{D}, v' \rangle : R$ **list**
  - If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Filter}\langle v_1 \mid v_2 \rangle : R$ **list**
  - If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle : R$ **list**

- If $v = \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle : R$ **list**
- If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Sort}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Limit}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Limit}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Exists}\langle v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Exists}\langle v' \rangle : B$

• If $T \equiv R$ **list** then

- If $v = x$ then impossible since $\Gamma, x : T \vdash_{\mathsf{SQL}} x : T \equiv R$ by Hypothesis H2
- If $v = \mathbf{fun}^f(x) \rightarrow v'$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{fun}^f(x) \rightarrow v' : T_1 \rightarrow T_2$
- If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv R$
- If $v = c$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} c : \texttt{typeof}(c) \equiv B$
- If $v = op$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \rightarrow \ldots \rightarrow B_n \rightarrow B$
- If $v = \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3 : B$
- If $v = \{\, l_i : v_i \,\}_{i=1..n}$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \{\, l_i : v_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}$
- If $v = [\,]$ then impossible since $[\,]$ cannot be typed
- If $v = v_1 :: v_2$ then by the typing rule of the list constructor: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **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 $v = v_1 \mathbin{@} v_2$ then by the typing rule of the list concatenation: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R$ **list** and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **list**, so by induction hypothesis $v_1 \equiv s$ and $v_2 \equiv s$, so $v = v_1 \mathbin{@} v_2 \equiv s \mathbin{@} s \equiv s$
- 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`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R' \rightarrow R$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R'$ **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 $v = \texttt{From}\langle \mathcal{D}, v' \rangle$ then by the typing rule of `From`: $\Gamma \vdash_{\mathsf{SQL}} v' : \texttt{string} \equiv B$, so by induction hypothesis $v' \equiv b$, so $v = \texttt{From}\langle \mathcal{D}, v' \rangle \equiv \texttt{From}\langle \mathcal{D}, b \rangle \equiv s$

- If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then by the typing rule of `Filter`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R' \to \texttt{bool}$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R'$ **list**, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to b$ and $v_2 \equiv s$, so $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle \equiv \texttt{Filter}\langle \mathbf{fun}^x(x) \to b \mid s \rangle \equiv s$
- If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then by the typing rules of `Join`: $v_1 = \mathbf{fun}(x, y) \to x \bowtie y$ or $\Gamma \vdash_{\mathsf{SQL}} v_1 : R' \to R'' \to R$, $\Gamma \vdash_{\mathsf{SQL}} v_2 : R' \to R'' \to \texttt{bool}$, $\Gamma \vdash_{\mathsf{SQL}} v_3 : R'$ **list** and $\Gamma \vdash_{\mathsf{SQL}} v_4 : R''$ **list**, so $v_1 \equiv \mathbf{fun}^x(x, x) \to x \bowtie x$ or by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x, x) \to r$, $v_2 \equiv \mathbf{fun}^x(x, x) \to b$, $v_3 \equiv s$ and $v_4 \equiv s$, so $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle \equiv \texttt{Join}\langle \mathbf{fun}^x(x, x) \to r, \mathbf{fun}^x(x, x) \to b \mid s, s \rangle \equiv s$
- If $v = \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle$ then by the typing rule of `Group`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R'' \to R$, $\Gamma \vdash_{\mathsf{SQL}} v_2 : R'' \to R'$ and $\Gamma \vdash_{\mathsf{SQL}} v_3 : R''$ **list**, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to r$, $v_2 \equiv \mathbf{fun}^x(x) \to r$ and $v_3 \equiv s$, so
  $v = \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle$
  $\equiv \texttt{Group}\langle \mathbf{fun}^x(x) \to r, \mathbf{fun}^x(x) \to r \mid s \rangle \equiv s$
- If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then by the typing rule of `Sort`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : R \to R'$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **list**, so by induction hypothesis $v_1 \equiv \mathbf{fun}^x(x) \to r$ and $v_2 \equiv s$, so $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle \equiv \texttt{Sort}\langle \mathbf{fun}^x(x) \to r \mid s \rangle \equiv s$
- If $v = \texttt{Limit}\langle v_1 \mid v_2 \rangle$ then by the typing rule of `Limit`: $\Gamma \vdash_{\mathsf{SQL}} v_1 : \texttt{int}$ and $\Gamma \vdash_{\mathsf{SQL}} v_2 : R$ **list**, so by induction hypothesis $v_1 \equiv b$ and $v_2 \equiv s$, so $v = \texttt{Limit}\langle v_1 \mid v_2 \rangle \equiv \texttt{Limit}\langle b \mid s \rangle \equiv s$
- If $v = \texttt{Exists}\langle v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Exists}\langle v' \rangle : B$

- If $T \equiv R \to B$ then
  - If $v = x$ then impossible since $\Gamma, x : T \vdash_{\mathsf{SQL}} x : T \equiv R$ by Hypothesis H2
  - If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R \vdash_{\mathsf{SQL}} v' : B$, so by induction hypothesis $v' \equiv b$, so $v = \mathbf{fun}^f(x) \to v' \equiv \mathbf{fun}^x(x) \to b$
  - If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv R$
  - If $v = c$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} c : \texttt{typeof}(c) \equiv B$
  - If $v = op$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \to \ldots \to B_n \to B$
  - If $v = \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3 : B$

- If $v = \{\, l_i : v_i \,\}_{i=1..n}$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \{\, l_i : v_i \,\}_{i=1..n} : \{l_i : T_i\}_{i=1..n}$
- If $v = [\,]$ then impossible since $[\,]$ cannot be typed
- If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 :: v_2 : R$ **list**
- If $v = v_1 \mathbin{@} v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 \mathbin{@} v_2 : R$ **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 $\Gamma \vdash_{\mathsf{SQL}} \texttt{Project}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{From}\langle \mathcal{D}, v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{From}\langle \mathcal{D}, v' \rangle : R$ **list**
- If $v = \texttt{Filter}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Filter}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle : R$ **list**
- If $v = \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Group}\langle v_1, v_2 \mid v_3 \rangle : R$ **list**
- If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Sort}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Limit}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Limit}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Exists}\langle v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Exists}\langle v' \rangle : B$

• If $T \equiv R \to R$ then

- If $v = \mathbf{fun}^f(x) \to v'$ then by typing rule of the function: $\Gamma, x : R \vdash_{\mathsf{SQL}} v' : R$, 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: $\Gamma, x : R \vdash_{\mathsf{SQL}} v' : R \to B$, 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 $v = \mathbf{fun}^f(x) \rightarrow v'$ then by typing rule of the function: $\Gamma, x : R \vdash_{\mathsf{SQL}} v' : R \rightarrow R$, so by induction hypothesis $v' \equiv \mathbf{fun}^x(x) \rightarrow r$, so $v = \mathbf{fun}^x(x) \rightarrow v' \equiv \mathbf{fun}^x(x, x) \rightarrow r$
- all other cases are impossible for the same arguments as for $T \equiv R \rightarrow B$

- If $T \equiv T_1 \rightarrow T_2$ then
  - If $v = \mathbf{fun}^x(x) \rightarrow v'$ or $v = op$ then the property is true
  - all other cases are impossible for the same arguments as for $T \equiv R \rightarrow B$

- If $T \equiv \{l_1 : T_1, \ldots, l_n : T_n\}$
  - If $v = x$ then the property is true
  - If $v = \mathbf{fun}^f(x) \rightarrow v'$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{fun}^f(x) \rightarrow v' : T_1 \rightarrow T_2$
  - If $v = v_1\ v_2$ then impossible for the same argument as for $T \equiv R$
  - If $v = c$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} c : \mathtt{typeof}(c) \equiv B$
  - If $v = op$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} op : B_1 \rightarrow \ldots \rightarrow B_n \rightarrow B$
  - If $v = \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathbf{if}\, v_1\, \mathbf{then}\, v_2\, \mathbf{else}\, v_3 : B$
  - If $v = \{\, l_i : v_i \,\}_{i=1..n}$ then the property is true
  - If $v = [\,]$ then impossible since $[\,]$ cannot be typed
  - If $v = v_1 :: v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 :: v_2 : R\ \mathbf{list}$
  - If $v = v_1 \mathbin{@} v_2$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} v_1 \mathbin{@} v_2 : R\ \mathbf{list}$
  - If $v = v' \cdot l$ then impossible for the same argument as for $T \equiv R$
  - If $v = \mathtt{Project}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathtt{Project}\langle v_1 \mid v_2 \rangle : R\ \mathbf{list}$
  - If $v = \mathtt{From}\langle \mathcal{D}, v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathtt{From}\langle \mathcal{D}, v' \rangle : R\ \mathbf{list}$
  - If $v = \mathtt{Filter}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathtt{Filter}\langle v_1 \mid v_2 \rangle : R\ \mathbf{list}$
  - If $v = \mathtt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathtt{Join}\langle v_1, v_2 \mid v_3, v_4 \rangle : R\ \mathbf{list}$
  - If $v = \mathtt{Group}\langle v_1, v_2 \mid v_3 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \mathtt{Group}\langle v_1, v_2 \mid v_3 \rangle : R\ \mathbf{list}$

- If $v = \texttt{Sort}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Sort}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Limit}\langle v_1 \mid v_2 \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Limit}\langle v_1 \mid v_2 \rangle : R$ **list**
- If $v = \texttt{Exists}\langle v' \rangle$ then impossible since $\Gamma \vdash_{\mathsf{SQL}} \texttt{Exists}\langle v' \rangle : B$

□

Bibliography

[AB16] Tada AB. PL/Java add-on module. https://github.com/tada/pljava, 2016.

[Acz77] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. Elsevier, 1977.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Ama] Python Language Support for UDFs. http://docs.aws.amazon.com/redshift/latest/dg/udf-python-language-support.html.

[Apaa] Apache HBase. https://hbase.apache.org/.

[Apab] Hive Manual - MapReduce scripts. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Transform.

[Apac] PySpark documentation - pyspark.sql.functions. http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html.

[Apad] User Defined Functions in Cassandra 3.0. http://www.datastax.com/dev/blog/user-defined-functions-in-cassandra-3-0.

[ASL89] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: a query language for manipulating object-oriented databases. In *Proceedings of the 15th international conference on Very Large Data Bases (VLDB)*, pages 433–442, New York, NY, USA, 1989. ACM.

[Bar92] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[BCD+18] Véronique Benzaken, Giuseppe Castagna, Laurent Daynès, Julien Lopez, Kim Nguyen, and Romain Vernoux. Language-integrated queries: A boldr approach. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 711–719, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[BCRH+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 221–230, New York, NY, USA, 2018. ACM.

[BD13] Patrick Bahr and Laurence E. Day. Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 61–72, New York, NY, USA, 2013. ACM.

[BV07] Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Trans. Comput. Logic*, 9(1), December 2007.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[Car88] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 70–79, New York, NY, USA, 1988. ACM.

[CDG+07] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. http://tata.gforge.inria.fr/, 2007.

[Chl10] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. volume 45, pages 122–133, 06 2010.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.

[CLF15] João Costa Seco, Hugo Lourenço, and Paulo Ferreira. A Common Data Manipulation Language for Nested Data in Heterogeneous Environments. In *Proceedings of the 15th Symposium on Database*

Programming Languages, DBPL 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[CLW13] J. Cheney, S. Lindley, and P. Wadler. A Practical Theory of Language-Integrated Query. In *International Conference on Functional Programming (ICFP) 2013*, pages 403–416, New York, NY, USA, 2013. ACM.

[CLWY07] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *In FMCO 2006, volume 4709 of LNCS*, pages 266–296, 2007.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. *SIGMOD Rec.*, 14(2):316–325, June 1984.

[CNXA15] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 289–302, New York, NY, USA, 2015. ACM.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[Cor16] Oracle Corporation. Java JDBC API. http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, 2016.

[CSLM13] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *Programming Language Design and Implementation (PLDI) '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, New York, NY, USA, 2013. ACM.

[ct17] Rails core team. Active Record - Ruby on Rails. http://guides.rubyonrails.org/active_record_basics.html, 2017.

[CW11] William R. Cook and Ben Wiedermann. Remote Batch Invocation for SQL Databases. In *Database Programming Languages - DBPL 2011, 13th International Symposium, Seattle, Washington, USA, August 29, 2011. Proceedings*, 2011.

[dot] dotconnect. https://www.devart.com/dotconnect/.

[DWM14] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation without regret: reducing deoptimization meta-data in

the Graal compiler. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 187–193, Trier, Germany, 2014. dblp.

[Ein11] O. Eini. The Pain of Implementing LINQ Providers. *Commun. ACM*, 54(8):55–61, August 2011.

[EN89] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[ERBS16] V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *SIGMOD '16 Proceedings of the 2016 International Conference on Management of Data*, pages 1781–1796, New York, NY, USA, 2016. ACM.

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[GIS10] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with Database Query Capability. *Journal of Object Technology*, 9(4):45–68, 2010.

[GRS10] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-Safe LINQ Compilation. *Proceedings of Very Large Data Bases (PVLDB)*, 3(1):162–172, 2010.

[Hiv] Hive Language Manual. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateFunction.

[HVO06] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, Washington D.C., USA, March 2006.

[ISO16] ISO. ISO/IEC 9075-2:2016, Information technology-Database languages–SQL–Part 2: Foundation (SQL/Foundation), 2016.

[JSF12] P J Sadalage and M Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 01 2012.

[KBA+09] Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, and Steve Ebersole. HIBERNATE - Relational Persistence for Idiomatic Java. https://docs.jboss.org/hibernate/orm/3.3/reference/en-US/pdf/hibernate_reference.pdf, 2009.

[Kis14] Oleg Kiselyov. The Design and Implementation of BER MetaOCaml. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 06 2014. Springer International Publishing.

[KK17] Oleg Kiselyov and Tatsuya Katsushima. Sound and efficient language-integrated query. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems*, pages 364–383, Cham, 11 2017. Springer International Publishing.

[KMH07] Jacob Kaplan-Moss and Adrian Holovaty. *The Definitive Guide to Django: Web Development Done Right*. Apress, December 2007.

[KSK16] Oleg Kiselyov, Kenichi Suzuki, and Yukiyoshi Kameyama. Finally, safely-extensible and efficient language-integrated query. 01 2016.

[Kuh11] D. Kuhlman. A python book: Beginning python, advanced python, and python exercises. section 3.4.2.4, pages 180–182. PLATYPUS GLOBAL MEDIA, 2011.

[LC12] Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 91–102, New York, NY, USA, 2012. ACM.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284 – 304, 2009. Special issue on Structural Operational Semantics (SOS).

[Lop16] J. Lopez. Master's thesis - Breaking the wall between general-purpose languages and databases. https://www.lri.fr/~lopez/docs/mpri_report.pdf, January 2016.

[Mic] LINQ (Language-Integrated Query). https://msdn.microsoft.com/en-us/library/bb397926.aspx.

[Mil78] Robin Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*. Elsevier, 1978.

[Mon] MongoDB User Manual - Server-side JavaScript. https://docs.mongodb.com/manual/core/server-side-javascript/.

[MyP] Mypy - an experimental optional static type checker for Python. http://mypy-lang.org/about.html.

[NN99] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design*, 1999.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, November 1995.

[OPV14] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. 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.

[Oraa] FastR. https://github.com/graalvm/fastr.

[Orab] Grall.js. https://www.youtube.com/watch?v=OUo3BFMwQFo.

[Orac] JRuby. https://github.com/jruby/jruby.

[Orad] Oracle R Enterprise. http://www.oracle.com/technetwork/database/database-technologies/r.

[Orae] ZipPy. https://github.com/securesystemslab/zippy.

[OU11] Atsushi Ohori and Katsuhiro Ueno. Making standard ml a practical database programming language. *SIGPLAN Not.*, 46(9):307–319, September 2011.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[PJW07] S. Peyton Jones and P. Wadler. Comprehensive comprehensions. In *Haskell '07 Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 61–72. ACM, 2007.

[PL/a] PL/R Project. http://www.joeconway.com/plr.html.

[PL/b] Oracle Database 18c PL/SQL. https://www.oracle.com/technetwork/database/features/plsql/index.html.

[Pos] PL/Python - Python Procedural Language. https://www.postgresql.org/docs/current/plpython.html.

[Pot98] François Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998.

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. A draft extended version is also available.

[PTN+18] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of Very Large Data Bases (PVLDB) Endowment*, 11(9):1002–1015, May 2018.

[PTS+17] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. A common runtime for high performance data analysis. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[Qui] Compile-time queries with quill. https://scalac.io/quill-compile-time-queries/.

[Rém89] D. Rémy. Type checking records and variants in a natural extension of ml. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 77–88, New York, NY, USA, 1989. ACM.

[Rém92] Didier Rémy. Typing record concatenation for free. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 166–176, New York, NY, USA, 1992. ACM.

[Rey99] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, New York, NY, USA, 1999.

[RS12] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 133–144, New York, NY, USA, 2012. ACM.

[Sli] Functional relational mapping for scala. http://slick.lightbend.com/.

[SPJ03] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, 37, 01 2003.

[ST06] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Proceedings, Scheme and Functional Programming Workshop 2006*, pages 81–92, Chicago, USA, 2006. University of Chicago TR-2006-06.

[T-S] Transact-SQL Reference (Database Engine). https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-2017.

[TPC17] TPC. The TPC-H benchmark. http://www.tpc.org/tpch/, 2017.

[Uni] Unityjdbc. http://www.unityjdbc.com/.

[Van05] Stijn Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Informatica*, 41(6):367–381, May 2005.

[VdBW02] Jan Van den Bussche and Emmanuel Waller. Polymorphic type inference for the relational algebra. *J. Comput. Syst. Sci.*, 64(3):694–718, May 2002.

[Ver16] Romain Vernoux. Design of an intermediate representation for query languages. *CoRR*, abs/1607.04197, 2016.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, Berlin, Heidelberg, 1995. Springer-Verlag.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, 1987. IEEE.

[WC07] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In Martin Hofmann 0001 and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 199–210, New York, NY, USA, 2007. ACM.

[Wim14] C. Wimmer. One VM to Rule Them All. http://lafo.ssw.uni-linz.ac.at/papers/2014_SPLASH_OneVMToRuleThemAll.pdf, October 2014.

[WV10] Jonathan H. Wage and Konsta Vesterinen. *Doctrine ORM for PHP*. Sensio SA, March 2010.

[WWW+13] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 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*, pages 187–204, New York, NY, USA, 2013. ACM.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Au-delà des frontières entre langages de programmation et bases de données

Mots clés : Requêtes intégrées au langage, bases de données, langages centrés données

Résumé : Plusieurs classes de solutions permettent d'exprimer des requêtes dans des langages de programmation: les interfaces spécifiques telles que JDBC, les mappings objet-relationnel ou object-relational mapping en anglais (ORMs) comme Hibernate, et les frameworks de requêtes intégrées au langage comme le framework LINQ de Microsoft. Cependant, la plupart de ces solutions ne permettent pas d'écrire des requêtes visant plusieurs bases de données en même temps, et aucune ne permet l'utilisation de logique d'application complexe dans des requêtes aux bases de données.

Cette thèse présente un nouveau framework de requêtes intégrées au langage nommé BOLDR qui permet d'écrire des requêtes dans des langages de programmation généralistes et qui contiennent de la logique d'application, et de les évaluer dans des bases de données hétérogènes. Dans ce framework, les requêtes d'une application sont traduites vers une représentation intermédiaire de requêtes. Puis, elles sont typées en utilisant un système de type extensible par les bases de données pour détecter dans quel langage de données chaque sous-expression doit être traduite. Cette phase de typage permet également de détecter certaines erreurs avant l'exécution. Ensuite, les requêtes sont réécrites pour éviter le phénomène "d'avalanche de requêtes" et pour profiter au maximum des capacités d'optimisation des bases de données. Enfin, les requêtes sont envoyées aux bases de données ciblées pour évaluation et les résultats obtenus sont convertis dans le langage de programmation de l'application. Nos expériences montrent que les techniques implémentées dans ce framework sont applicables pour de véritables applications centrées données, et permettent de gérer efficacement un vaste champ de requêtes intégrées à des langages de programmation généralistes.

Title: Breaking boundaries between programming languages and databases

Keywords: Language-integrated queries, databases, data-centric languages

Abstract: Several classes of solutions allow programming languages to express queries: specific APIs such as JDBC, Object-Relational Mappings (ORMs) such as Hibernate, and language-integrated query frameworks such as Microsoft's LINQ. However, most of these solutions do not allow for efficient cross-databases queries, and none allow the use of complex application logic from the programming language in queries.

This thesis studies the design of a new language-integrated query framework called BOLDR that allows the evaluation in databases of queries written in general-purpose programming languages containing application logic, and targeting several databases following different data models. In this framework, application queries are translated to an intermediate representation. Then, they are typed with a type system extensible by databases in order to detect which database language each subexpression should be translated to. This type system also allows us to detect a class of errors before execution. Next, they are rewritten in order to avoid query avalanches and make the most out of database optimizations. Finally, queries are sent for evaluation to the corresponding databases and the results are converted 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.