Internship report - Breaking the wall between general-purpose languages and databases

Julien Lopez

January 6, 2017

Abstract

This internship is part of a collaboration between Laurent Daynes at Oracle Labs (Grenoble), Kim Nguyễn and Romain Vernoux at LRI (Orsay), and Giuseppe Castagna and myself at PPS (Proofs, Programs and Systems) laboratory (Paris 7). Even though I was as an intern at Oracle Labs under the supervision of Laurent Daynes, I worked most of the time at PPS with Giuseppe Castagna. The project was split into three independant parts, one for each group, and we reported our respective status during regular meetings.

Contents

1 Introduction

1.1 Overview

General context

In today's context of "big data", the volume of information never stops increasing. Because of this and of the large variety of usages of this information, applications that manipulate data are becoming more and more complex. However, these applications are always composed of two main components: the front-end which includes the user interface and is written in general-purpose programming languages (Java, Ruby, JavaScript, R, . . .), and the database side that takes care of data manipulation. Queries are sent from the application to the database to be evaluated, and the results are then translated into the application datatypes. However, these queries cannot contain user-defined functions (UDFs) from the application language, therefore either UDFs are not allowed resulting in a loss of expressivity, or round trips from the application to the database are necessary to execute UDFs in a query resulting to terrible performances regardless of the technological choices.

Besides, because there is no clean separation between the front-end and the back-end, the creation of a new interface to support another database needs work on the entire process, which implies that developers need to be expert in both application side and database. Additionally, applications may access several databases with different models (relational, NoSQL, document, . . .), so experts in both the database model and the application language are required.

In this context, *Oracle Corporation* is developing *Graal* and *Truffle*: Graal is a just-in-time compiler for Java that focuses on providing high performances by applying Oracle's research in compiler technology, using in particular new techniques in speculative evaluation; Truffle is a framework for executing dynamic languages that achieves high performance when combined with Graal (more detailed description in Section 1.2). A full execution engine of a JavaScript implementation in Truffle as well as open source projects building Truffle-based runtimes for Ruby, R and Python are available. The use of Truffle and Graal allows the efficient evaluation of UDFs from dynamic languages in a Truffle-capable database (a database embedding a Truffle runtime of the dynamic language).

Objectives

The goal of this internship is to tackle the issues described above using the solutions developed by Oracle, by designing a framework that will simplify the implementation of high-performance data-centric programming languages on top of Truffle and the integration of Truffle-capable databases regardless of their models.

To reach these goals, the following questions need to be explored:

- How to design a framework that can link any dynamic programming language to any data source?
- How to evaluate application code in the database?

- How to define in a program what to evaluate in the application runtime and what to send to the database?

Some databases already give the possibility to evaluate code from some programming languages[1], and some research as been done on frameworks that allow programmers to express queries in general-purpose programming languages[2][3][4][5]. However, none of these solutions are satisfactory since they lack genericity and poorly handle UDFs.

Proposed solution

The solution I developed in this internship consists of three steps :

- implement an abstract query language that can represent queries from any database;
- use this language as a unique interface between programming languages and databases;
- interface a programming language (*host language*) and a database to the intermediate language.

The intermediate language I used is *QIR* (Query Intermediate Representation) which was defined at LRI. In the context of the project my internship was developed in, I implemented this abstract representation of queries in Java and added constructions to interact with it (see Section 2).

I extended a dynamic language implemented in Truffle called SimpleLanguage (SL) with constructions to express queries by experimenting with different possibilities based on existing solutions (see Section 3).

After experiencing the limits of the classic techniques to translate an host language query to a database query, I defined a declarative language called *Database Capabilities Description Language* (DCDL) to publish database capabilities and transform a query in QIR to a native query expressed in a language the targeted database can understand (see Section 4).

Validity of the solution

My implementation of QIR respects the specifications written by the LRI, therefore my approach benefits from the properties proven on this abstract representation.

Since the goal of my internship is to study the specifics of interfacing languages implemented in Truffle and Truffle-capable databases, it is validated:

- on the host language side by the extension I implemented for SL;
- on the database side by the mapping from QIR operators to a special extension of SQL + PL/SQL provided by Oracle.

As for DCDL, not only does it factorize code and simplify translations of queries, it also allows us to prove some interesting properties such as termination of these translations.

Furthermore, the approach as a whole is being validated by another internship taking place at Oracle Labs where a different host language (JavaScript) and querying paradigm is being mapped on the QIR.

1.2 Truffle and Graal

Truffle is a framework for implementing dynamic languages in Java. The language implementer writes a parser for the language (or a translator for an already existing parser) that generates *Truffle ASTs.* Truffle provides an API to create and instantiate *Truffle nodes* that

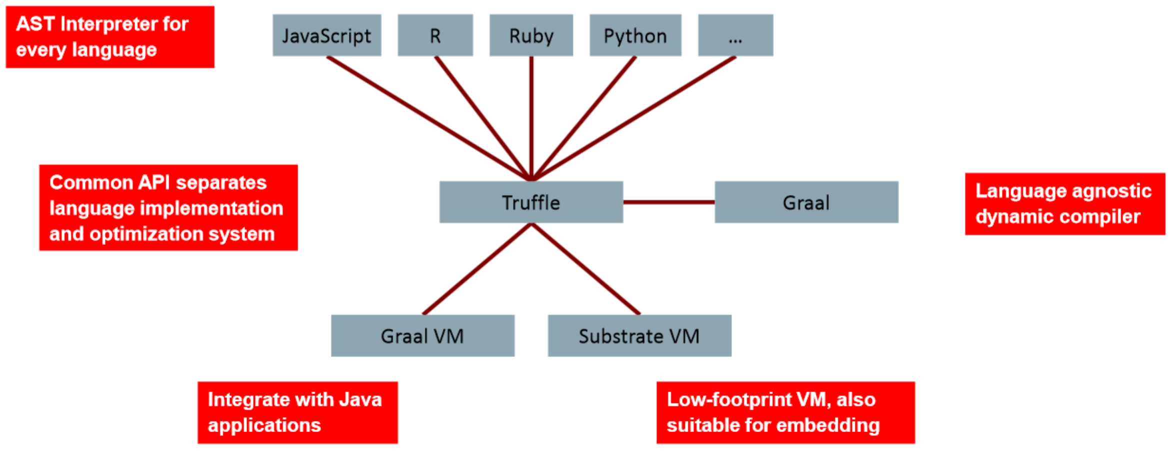

AST Interpreter for every language
JavaScript
R
Ruby
Python
...
Common API separates language implementation and optimization system
Truffle
Graal
Language agnostic dynamic compiler
Graal VM
Substrate VM
Integrate with Java applications
Low-footprint VM, also suitable for embedding

Figure 1: Truffle and Graal

compose Truffle ASTs. A Truffle node describes a part of a program and contains methods to execute it. An example of Truffle node can be found in Appendix A. It also provides guidelines[6] to teach language implementers how to generate "good" ASTs that will be efficiently optimized by Graal.

As Figure 1 shows, Truffle allows Graal to be language agnostic since the compiler works on Truffle ASTs rather than on an AST specific to a particular language. The language can then be seamlessly integrated to an existing virtual machine that supports Truffle. This framework is an important gain of time for the language implementer, and the automatic integration to a Just-In-Time compiler immediately gives competitive performances at the cost of learning how to use Truffle the right way (which is negligible compared to building the whole architecture from scratch)[7][8].

In the next section, I will describe the architecture of the joint LRI/Oracle/PPS project my internship took place in.

1.3 Architecture

The QIR workflow works as follows:

1. Queries specified in the syntax of the host language (possibly augmented with a special syntax for queries) are mapped to QIR constructs if possible, and kept as Truffle node black boxes otherwise. This translation phase also maps the data model of the host language to the QIR data model.

2. QIR constructs are rewritten and partially evaluated within the QIR framework, using a description (written in DCDL) of the capabilities of the target database.

3. The QIR operators and expressions that are supported by the database (i.e., that can be implemented by the database operators and expressions) are translated into the database native language, whereas expressions that are not supported and black box Truffle nodes are delegated to the Truffle runtime in the database. This translation phase also maps the QIR data model to the database data model.

4. The generated queries and the Truffle nodes are evaluated inside the database and the result is returned to the host language. This backward translation maps database native data to the data model of the host language.

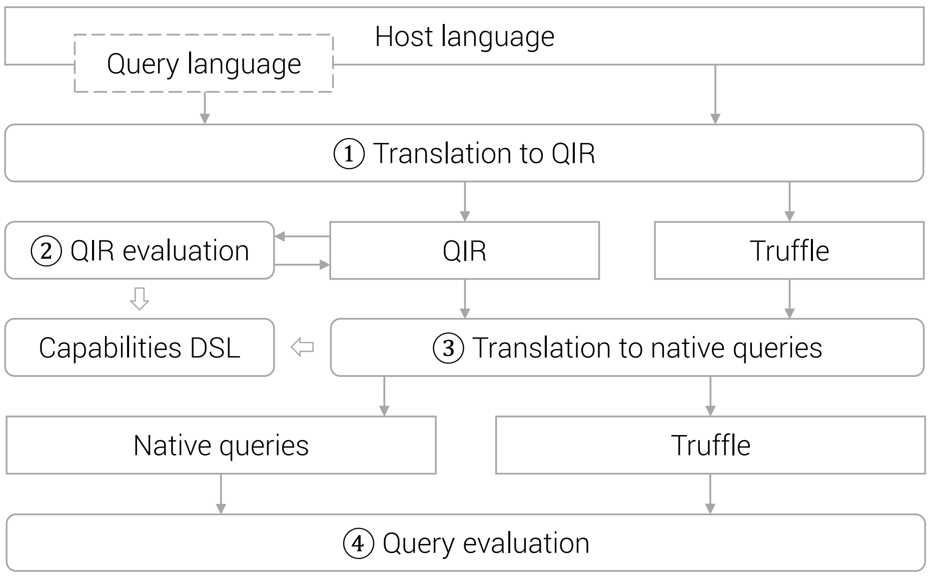


Figure 2: Architecture

2 QIR

2.1 Definition

QIR (Query Intermediate Representation) is a language based on *lambda-calculus* extended with high-level operators that represent computations in a query (in particular from *relational algebra*), and low-level constructions to represent data.

The expressions of QIR are:

$$
\begin{array}{llcll}
\textbf{Expressions} & e & ::= & x & \text{(variables)} \\
 & & | & \lambda x.e & \text{(lambda expressions)} \\
 & & | & ee & \text{(applications)} \\
 & & | & \texttt{true} \mid \texttt{false} \mid 1 \mid 2 \mid \ldots & \text{(values)} \\
 & & | & \texttt{tnil} \mid \texttt{tcons}\, e\, e\, e \mid \texttt{tdestr}\, e\, e & \text{(tuple manipulators)} \\
 & & | & \texttt{lnil} \mid \texttt{lcons}\, e\, e \mid \texttt{ldestr}\, e\, e\, e & \text{(list manipulators)} \\
 & & | & o & \text{(operators)} \\
 & & | & b & \text{(builtins)} \\
 & & | & d & \text{(data references)} \\
 & & | & t & \text{(Truffle nodes)} \\
\textbf{Operators} & o & ::= & \texttt{Scan}_e() & \\
 & & | & \texttt{Select}_e(e) & \\
 & & | & \texttt{Project}_e(e) & \\
 & & | & \texttt{Sort}_e(e) & \\
 & & | & \texttt{Group}_{e,e}(e) &
\end{array}
$$

where `tnil` represents the empty tuple; `tcons` the tuple constructor that takes a string for the mapping name, an expression for the mapping value and an expression for the tail of the mapping list; `tdestr` the tuple destructor that takes a tuple and an identifier and returns the value mapped to the identifier in the tuple; `lnil` represents the empty list; `lcons` the list constructor that returns a list with its first argument as head and its second argument as tail; `ldestr` the list destructor that has three arguments: the list to destruct, the term to return when the list is nil and a function with two arguments to treat the case when the list has a head and a tail; a data reference is a reference to a database, a table, . . . ; a Truffle node (in QIR) is a QIR construction that encapsulates an expression described by a Truffle AST that can not or should not be translated in QIR; finally, an operator represents a computation on a data source (see Section 2.4 for details on operators).

2.2 Benefits of QIR

High-level

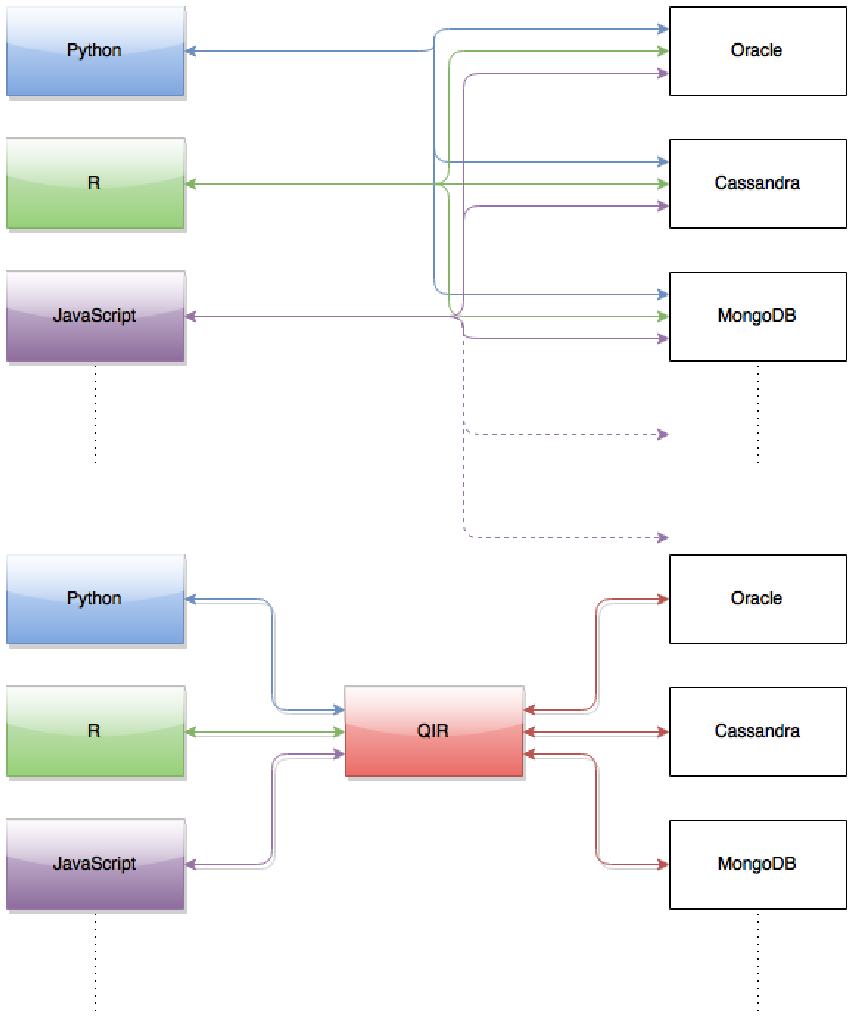


Figure 3: High-level benefits of an intermediate representation

As Figure 3 shows, an intermediate representation in the middle of the architecture is a huge gain for the language implementer since only a link to the QIR is necessary instead of an interface for every targeted database. Besides, every time an interface is created between QIR and a new database, all the languages get access to this database for free. However, this means that the QIR will have to be able to represent a non-negligible part of the constructions from programming languages and databases (Oracle number, date, . . .), but this cost is negligeable compared to the gain in the architecture.

QIR evaluation

The module of QIR evaluation (point 2 in Figure 2) has been developed at Orsay. Its goal is to transform a QIR that is the direct translation of a query expressed in the host language to a QIR that is easier to translate into a query in the targeted database.[1] For instance:

$$(\lambda x.(Select_{\lambda t.t.id})(x))(Scan_{table}()) \rightarrow Select_{\lambda t.t.id}(Scan_{table}())$$

In the above example, the QIR evaluation module uses partial evaluation to not only simplify the QIR tree, but also remove the "glue" between two parts of a query ($Select_{\lambda t.t.id}(x)$ and $Scan_{table}()$) to merge them in one single query, thus allowing the database to perform more optimizations.

Properties of QIR

As explained in Section 2.1, QIR is based on the lambda-calculus, a well-known formal system on which a lot of properties have been shown. Thanks to this, the team at Orsay was able to prove some properties on the QIR and the QIR evaluation module, such as exhaustivity and optimality of the reduction strategy used in this module.

2.3 Implementation

In my implementation of QIR, I created a Java class for every construction of the language. The classes of my implementation of QIR are:

- **Lambda-calculus:** Var, Lambda, Apply
- **Operators:** Project, Scan, Select, Group, Order
- **Values:** Number, BigNumber, String, Boolean
- **Data:** Tnil, Tcons, Tdestr, ...
- **Other:** Plus, Star, And, Or, Null, ...

where Null represents the *null* value.

For the time being, QIR being used as a proof of concept, it only includes few values, but it will be extended in the future to be able to encode more primitive types efficiently.

Strictly speaking, QIR supports operators *join* and *limit*, but since these were introduced in the specifications near the end of the internship and since the support of these operators with UDFs was not guaranteed yet in the Oracle side, I restrained my implementation to the five operators listed above which are enough to describe reasonably complex queries.

Examples of QIR trees can be found in Appendix B.

Definition 1. *Let $\mathcal{Q}$ be the set of QIR node constructors:*

$$\mathcal{Q} = \{Project, Scan, Select, Group, Order, Tnil, String, \ldots\}$$

A QIR tree *is a tree whose nodes are assigned one symbol in $\mathcal{Q}$.*[2] *Nodes in a QIR tree are called* QIR nodes.

[1]Note that this module does not try to take the job of the query optimizer of the database, the QIR evaluation works on the expressions from the host language, not on the order of the operators.

[2]Thus, a QIR tree is a labeled tree.

2.4 Representation of a query

A query in QIR is represented as a QIR tree whose root is an operator. As defined in the specifications, every operator has a fixed number of *configurations* (indexes) and *children* (arguments), where children are QIR nodes that provide the input tables to the operator and configurations are computations on the rows of the input tables.

For example, *Scan* is an operator that has one configuration and no child, since it does not take any table has input and returns the rows of the table denoted by its configuration, therefore $Scan_{TableRef(''employee'')}()$ returns a sequence of all the rows in the table named “employee”.

Project is an operator that has one configuration and one child, $Project_{formatter}(input)$ applies the configuration $formatter$ to all the elements of the child *input* and returns the results as a sequence.

The other operators follow the classic semantics of relational algebra.

2.5 Representation of results

The result of a query being a list of tuples, their representation in QIR is naturally a List of Tuples. Therefore:

userId	userName
158	"John"
4985	"Jack"

is represented as:

```
Lcons(tuple1, Lcons(tuple2, Lnil))
```

where:

```
tuple1 = Tcons("userId", 158, Tcons("userName", "John", Tnil))
tuple2 = Tcons("userId", 4985, Tcons("userName", "Jack", Tnil))
```

2.6 Interface

In this section, I describe the interfaces I added to my implementation of QIR to communicate with host languages and databases.

2.6.1 QIRFactory

The first interface called *QIRFactory* is a simple implementation of the design pattern *factory* which gives an abstraction to the constructor of a class.

```
public QIRApply create_call(SourceSection src, QIRNode f, List<QIRNode> args)
{
  if (args.isEmpty())
    return new QIRApply(src, f, null);

  QIRApply res = new QIRApply(src, f, args.remove(0));
  for (QIRNode arg : args)
    res = new QIRApply(src, res, arg);
  return res;
}
```

In the example above, we create the representation of a function call in QIR. Indeed, using the method `create_call` allows the user to create QIR trees without having to manipulate low-level constructors.

QIRFactory is used in the host language side to create a QIR of a query.

2.6.2 Interface to the database

I created a Java interface that defines what is a translater between QIR and a database. However, this interface is only used as a temporary solution. See section 4 for more details.

I also created an interface that allows a host language to send a QIR to a database. For now, this interface is a simple structure that calls the corresponding driver for the targeted database and returns results expressed in the QIR data model. But this interface could:

- return results in streaming, in other words return result rows by fixed packets instead of sending everything at once;
- fetch schemas of tables and therefore help type-checking queries as it would then be possible to check if a column exists; has the right type; ...

This is left as future work.

3 Host language integration: QSL

3.1 Introduction

As explained in Section 1.1, one of the goals of this internship was to interface a general-purpose programming language to QIR. This involves extending the language with query expressions and interpreting these expressions to generate a QIR tree that represents the query.

3.2 SimpleLanguage

For this internship, the language used as host language is *SimpleLanguage* (SL) which is a dynamic functional language, close to JavaScript but much simpler, used by Oracle as a showcase of how to implement a language in Truffle as well as a simple test language for Graal. This language has been chosen for this internship for its simplicity, allowing us to work on the integrated queries without having to handle too many constructions, but also because it is a language already implemented in Truffle, saving us the trouble of implementing one from scratch which would have been outside the scope of this internship.

SimpleLanguage has first-class functions and objects, but does not have (yet) many constructions to manipulate its values which are functions, objects, long integers, strings and booleans. The absence of constructions such as lists and records is problematic for data manipulation, but fortunately these constructions can easily be simulated with objects.

Examples of SL programs can be found in Appendix C.1.

3.3 Integrating queries to a language

There are two different ways to add the possibility to express queries in a language:

- extend the syntax of the language:
  - keywords
  - user/reader-friendly
  - close to usual languages for data manipulation
  - modification of the parser
  - not very smooth integration
  - LINQ[2]
- use the constructions of the language (loops, comprehensions, ...):

  - objects, loops, annotations, ...
  - no modification of the parser
  - good integration
  - painful to read/write especially for complex queries
  - LINQ[2]; Hibernate[3]; Ruby on Rails[9]; ...

Originally, the syntax of SL being simple, I planned to extend it first and add an object-oriented way to express queries later (with the intention of getting closer to LINQ), but due to the lack of time and this matter not being the priority in this internship, I implemented only the first method.

3.4 QuerySimpleLanguage

QuerySimpleLanguage (QSL) is SimpleLanguage augmented with a syntax to express queries, as well as an interface to run these queries in Truffle-capable databases and retrieve the results inside SL values.

Thus, the syntax of QSL is the syntax of SL with queries added as expressions that return an object. Result manipulation is done via the methods of these objects (see Section 3.4.5 for details).

In the rest of this section, I will describe the different problems that I encountered during the implementation of QSL, namely:

- **Query boundaries**: how to separate queries from the rest of the program;
- **Handling of UDFs**: what to do with user-defined functions and in general expressions of the host language when they appear in a query;
- **Function dependencies**: how to send host language expressions to the runtime of the database when necessary;
- **Data model translation**: how to translate the data model of QSL to the one of QIR;
- **Retrieving results of a query**: how to get the results back in a QSL program.

Let us develop each point in one of the following subsections.

3.4.1 Query boundaries

Regardless of the method chosen to integrate queries into a language, one needs to define the *boundaries* of a query. In other words, one needs to define what to send to the database (in our case what to translate into QIR nodes) and what to keep for execution in the runtime of the language.

In the case of QSL, query boundaries are defined by *quotations*.

Quotations are syntactic constructs used to isolate a query from the host language. Since we also need to refer to host language symbols in a query (variables, UDFs, ...), we also introduce *anti-quotations*.

Consider the following example in QSL:

```
function f(x)
{
  if (x < 100) { return 0; }
  else { return x; }
}

select $f(id) from Customers
```

Here, `f` is a SimpleLanguage symbol (a function name), and `id` is a database expression (a column name). In this example quotations are *implicit*: it is the use of the `select` expression, which does not belong to the syntax of SL, which determines that the expression at issue is quoted, that is, it belongs to the QIR/database realm. A different solution that is used in practice[10] is to use *explicit* quotations by enclosing expressions between specific delimiters such as:

```
<@ select $f(id) from Customers @>
```

the advantage of explicit quotations being that it is much easier for the developer of the language to compute the boundaries of a query, and generally clearer to spot in a program what is sent to the database. The downsides are that explicit quotations are heavy to read and write, and they are unnecessary in our case, since the boundaries of a query in QSL are the limits of the `select` expression.

In both these examples, the anti-quotations "`$`" are explicit prefixes that indicate the presence of a symbol from the host language in the quoted expression. In QSL, an anti-quote is used to prefix exactly one identifier. A different approach is to allow a whole expression to be anti-quoted, which is handy to inline a computation expressed in the host language directly inside the query. However, quotations become a lot more complex to handle in this case, since they may appear in anti-quotations, so one needs to handle nesting of quotations. Moreover, one can always put an host language expression in a function that is then called in the query, and inlining may occur thanks to the QIR evaluation module (see Section 1.3).

3.4.2 Handling of UDFs

If the anti-quoted symbol represents an access to a function, it must be translated whenever possible into a QIR tree. The reason is that Truffle nodes[3] in QIR must be kept to a minimum, since we want as much information as we can to generate the best execution plan possible. Take for instance this program:

```
function f(id)
{
  return id > 100;
}

function main()
{
  o = select * from customer where f(id);
  ...
}
```

If `f` is kept as a Truffle node, we will need to send `f` to the database that will call it on every row of the table, the corresponding query would be `select * from customer where f(id)`. If `f` is translated into a Lambda of QIR, the QIR evaluation module will be able to inline the call `f(id)` using partial evaluation, allowing the database to possibly run optimizations, the corresponding query would be `select * from customer where id > 100`.

However, some problems remain to be solved:

- when should the function be translated?
- what to do if the function is not translated?
- how to know whether the function should be translated?

For the first problem, consider for example the following function in SimpleLanguage:

[3]a Truffle node in QIR is considered a "black box" that cannot be modified

```
function f(h, x)
{
  if (x > 2) { return h(x); }
  else       { return g(x); }
}
```

Here, `f` should be translated if `g` and `h` can be translated. However, `h` is only known during the execution of `f`, therefore in some cases the translation of a function in a query can only be done just before its execution.

The second problem is solved if there exists a way to send a Truffle AST to a database. Unfortunately, the serialization of a Truffle tree is complex… For this internship, the chosen solution is to send the source code to the function and its dependencies, then parse the code in the language runtime of the database to get back the Truffle tree. Obviously, this solution can only be temporary, since it involves parsing the functions again which is a small additional cost compared to the execution of the query itself, but a cost nonetheless that will be removed once a method to serialize a Truffle AST has been defined. It will then be the job of the language implementer to create a module that can serialize an AST from his language.

The last problem is trickier as there is no standard solution: it is the job of the language expert to decide whether a function should either be (i) executed in the host language of the "host machine" (ii) or executed in the host language of the database (iii) or translated into a QIR tree for execution in the query. For example:

- most *side-effects* are incompatible with any execution in the database, since we need to modify the state of the "host machine" and not the state of the database;
- a function too complex should not be translated into a QIR tree, the reason being the same as inlining in classic compiler optimization, that is we don't want to process and inline a block that represents a too large number of computations. The hard part is: where to put the limit?

For this internship, functions were all kept as Truffle nodes and sent to the database. In the future, we will need a module that is able to decide for a function between the three possible fates (executed in the application runtime, executed in the language runtime in the database, or translated into a QIR tree).

More examples of QSL programs can be found in Appendix C.2, and the grammar of QSL is detailed in Appendix C.3.

3.4.3 Function dependencies

A subproblem of translating a function occurring in a query is to retrieve its *dependencies*. In order to execute a function in the database, it is obviously necessary to send not only the function itself but also the values of the references used by the function.

Recall our example:

```
function f(h, x)
{
  if (x > 2) { return h(x); }
  else       { return g(x); }
}
```

To execute this function inside a query, we need the value bound to the variable `x`; and the source code of the functions `f`, `g` and `h`.

In the case of SimpleLanguage, the only global symbols are functions, so the dependencies are retrieved by looking recursively for definitions of the functions called in the body of the function used in a query (recursively, since of course we also need the dependencies of the functions called).

3.4.4 Data model translation

As explained in Section 2.3, QIR does not support a lot of primitive constructions for now, but since it is also the case for SimpleLanguage, I used the following simple translation rules:

QSL boolean	$\leftrightarrow$	QIR boolean	(both stored as a Java boolean)
QSL string	$\leftrightarrow$	QIR string	(both stored as a Java string)
QSL number	$\leftrightarrow$	QIR number	(both stored as a Java long)
QSL function	$\rightarrow$	QIR lambda	
QSL object	$\leftarrow$	QIR tuple	
QSL object	$\leftarrow$	QIR list	

where $x \rightarrow y$ is the translation of x to y from the data model of QSL to the one of QIR (used when creating and sending a query to the database), $x \leftarrow y$ is the translation of y to x from the data model of QIR to the one of QSL (used when getting back the results of a query), and $x \leftrightarrow y$ is equivalent to $x \rightarrow y \wedge x \leftarrow y$.

As explained in Section 2.5, the results of a query in QIR are represented as QIR lists of QIR tuples. Therefore, we need to translate these constructions to those of QSL to get back these results in QSL programs. Since QSL has objects, I chose to create objects that represent lists and tuples.

3.4.5 Retrieving results of a query

In this section, we will use this running example of QSL query:

```
o = select productId, productName from product;
```

where `productId` and `productName` are column identifiers, and `product` is a table name in the targeted database. As stated before, the evaluation of this expression in QSL returns an object `o` that represents the results of the described query. Results can then be accessed one by one by calling the method `next` on `o`.

```
tuple = o.next();
```

`next` takes no argument and returns a tuple from the result set[4]. Tuples being also represented as objects in QSL, `tuple` is an object. Columns from tuples in QSL can be accessed via the method `get` or the method `get<columnname>`:

```
productName = tuple.get("productName");
productName = tuple.getProductName();
```

Both methods return the value bound to the name of the column in the tuple, the only difference is that `get` takes the name of the column as argument while `get<columnname>` carries it in its name.

4 Database integration

4.1 Introduction

In this section, we will describe the translation of a query from QIR to a targeted database. From a query expressed in QIR, we need to generate the best translation possible into a representation that is understood by the database.

To achieve this, we need a module that is able to:

1. describe which QIR trees a particular database back-end is compatible with;

2. define the translation of a compatible QIR tree into a construction usable by a database.

[4]since the scan operator from QIR returns results in a non-deterministic order, it is the same for `next`

The first point provides the information required for the QIR evaluation (point 2 in Figure 2) component to work, the second answers our main concern.

4.2 Rationale for DSL

As a first solution, I simply wrote Java code to explore a QIR tree using the *visitor* pattern:

```
public class QIROracleVisitor implements IQIRVisitor<String>
{
  public String visit(Project qirProject)
  {
    return "select␣" + qirProject.getFormatter().accept(this) + "␣" +
        qirProject.getChild().accept(this);
  }

  public String visit(Scan qirScan)
  ...
```

where `IQIRVisitor` is an interface that defines what is a translater between QIR and a database.

This method has some problems though:

- it is not easy to use, and even harder to understand from an external point of view: the execution flow is hard to follow particularly because it is split between several classes, and because it unnecessarily contains all the technical details;
- it is difficult to reuse, therefore difficult to factorize;
- static analysis such as termination or exhaustivity checks would be cumbersome in this formulation: being written in pure Java, such a translation would be difficult to analyse to ensure, for instance, its soundness with respect to the semantics or the absence of bugs in the generated QSL code.

For these reasons, declarative programming seems to be the best choice for the translation of QIR to native queries, as it will allow the user to describe what happens instead of how it happens, with the associated gains of speed, clarity, and concision.

4.3 Oracle database

In the case of the Oracle database, I created a driver that respects the interface of QIR for database drivers and that handles:

- the registering of a UDF that has been kept in a Truffle node in QIR;
- the translation of a QIR tree to an SQL query by defining a visitor in the way described above;
- the translation from the data model of the QIR to the one of Oracle and vice versa.

Note that since the solution proposed by the team at Oracle Labs is still under development, the data types from Oracle that could be used was restricted, so I only manipulated strings (`varchar` in Oracle) and numbers (`number` in Oracle).

4.4 Contribution

The first problem described in Section 4.1 is clearly a subset of the second one since describing compatibility can be done by returning simple constructions that describe the values true (compatible) and false (not compatible).

Therefore, the goal being the translation of finite labeled trees to finite labeled trees, the logical choice was to look into *tree transducers*[11]. Tree transducers are automata that explore a finite labeled tree given as input, changing state according to the current label and a set of rules that are mutually recursive, and creates another finite labeled tree.

More precisely, I consider *macro tree transducers* (MTT)[12]. MTTs satisfy nice properties such as decidable exact typechecking, and can express more complex tree transformations than tree transducers by allowing rules to be defined in terms of an arbitrary number of accumulation parameters (subtrees that are kept for comparaisons but not to be explored), which is essential for expressivity in our case since we will often need to compare a value in the tree to some value we encountered before.

However, MTTs are "low-level" and hard to implement. Since our purpose is to produce a usable DSL, we made some choices that make our version of MTT more practical (from the point of view of language design) and make sure that the important property of termination is ensured by a simple syntactic property:

- restriction to deterministic tree transducers;
- addition of arbitrary guards to the rules as it is common in our case to write a rule that has to be applied only if some expression on data evaluates to *true*;
- use of richer patterns than those usually found in litterature and inspired by the ℂDuce [13] language.

4.5 DCDL

In this section, I define the Database Capabilities Description Language (DCDL), a domain-specific language that aims to solve the two problems described in Section 4.1. It is inspired by macro tree transducers and my experiences in programming languages. A DCDL program consists of a finite ordered sequence of named rules. Here is an example (rule names are in blue):

```
lambda ::= Lambda(_, x) -> unsigned(x)
lambda ::= _ -> false

unsigned ::= Number(x) when x >= 0 -> true
unsigned ::= _ -> false
```

In the above example, the `lambda` rule recognizes (i.e., returns true for) a lambda expression of QIR only if its body is a positive number. Most of the features used in this program come from the MTT, however the `when` guard is a necessary addition that allows us to test if a number is unsigned. The same example in Java would be much more verbose and complicated since it would include all the low-level accesses.

In the rest of this section, I describe its syntax, semantics and properties, and give more examples.

4.5.1 Syntax

First, we define the syntax of patterns. Patterns are constructs used to recognize an input tree of a particular form and to capture parts of it in order to use them in the construction of the output of the transformation represented by the MTT.

Definition 2. *Let Σ and $\mathcal{V}$ be two distinct sets of symbols respectively denoting constructors and variables.* Patterns *are the terms inductively generated by the following grammar:*

$$\begin{array}{lcll} p & ::= & x & \textit{(capture variable, } x \in \mathcal{V}\textit{)} \\ & | & \_ & \textit{(any)} \\ & | & p \,\&\, p & \textit{(intersection of patterns)} \\ & | & p \mid p & \textit{(union of patterns)} \\ & | & c(p, \ldots, p) & (c \in \Sigma) \end{array}$$

Definition 3. *For a pattern p, the set of* variables *occuring in p, noted $vars(p)$, is defined as:*

$$\begin{array}{rcl} vars(x) & = & x \\ vars(\_) & = & \emptyset \\ vars(p_1 \,\&\, p_2) & = & vars(p_1) \cup vars(p_2) \\ vars(p_1 \mid p_2) & = & vars(p_1) \cap vars(p_2) \\ vars(c(p_1, \ldots, p_m)) & = & vars(p_1) \cup \ldots \cup vars(p_m) \end{array}$$

Definition 4. *For a pattern p, the set of its* subvariables, *noted $subvars(p)$, is defined as:*

$$\begin{array}{rcl} subvars(x) & = & \emptyset \\ subvars(\_) & = & \emptyset \\ subvars(p_1 \,\&\, p_2) & = & subvars(p_1) \cup subvars(p_2) \\ subvars(p_1 \mid p_2) & = & subvars(p_1) \cap subvars(p_2) \\ subvars(c(p_1, \ldots, p_m)) & = & vars(p_1) \cup \ldots \cup vars(p_m) \end{array}$$

Remark. $subvars(p)$ then contains the variables that capture strict subtrees of the input tree.

Definition 5. *A pattern is* well-formed *if:*

- *It is a variable x ($x \in \mathcal{V}$);*
- *or it is $\_$;*
- *or it is an intersection $p_1 \,\&\, p_2$, p_1 and p_2 are well-formed, and $vars(p_1) \cap vars(p_2) = \emptyset$;*
- *or it is a union $p_1 \mid p_2$, p_1 and p_2 are well-formed, and $vars(p_1) = vars(p_2)$;*
- *or it is a constructor $c(p_1, \ldots, p_m)$ such that p_i is well-formed for all i in $1..m$ and $vars(p_i) \cap vars(p_j) = \emptyset$ for all i, j in $1..m$, $i \neq j$*

Note

In the rest of this document, we will assume that all patterns are well-formed.

Definition 6. *Let $\mathcal{V}$, Σ and $\mathcal{N}$ be three pairwise distinct sets of symbols that are respectively a set of variables, an alphabet, and a set of rule names. A* rule *is a 5-tuple (n, l, p, e, o) that we write:*

$$n, l, p \text{ when } e \rightarrow o$$

where n is the name of the rule, l is a sequence of variables called accumulators, p is a pattern as defined in definition 2, e is a logical formula, *that is, a term inductively generated by the following productions:*

$$\begin{array}{lcll} e & ::= & x & \textit{(capture variable, } x \in \mathcal{V}\textit{)} \\ & | & op(e, \ldots, e) & \textit{(application of the logical operator } op\textit{)} \end{array}$$

and o is an output, *that is a finite term of a language produced by the following grammar:*

$$\begin{array}{lcll} o & ::= & x & \textit{(capture variable, } x \in \mathcal{V}\textit{)} \\ & \mid & n[o,\ldots,o](o) & \textit{(call to a rule, } n \in \mathcal{N}\textit{)} \\ & \mid & c(o,\ldots,o) & (c \in \Sigma) \end{array}$$

where outputs between brackets in a call to a rule are accumulators and the constructors are those of the language.

Note

We write $n, l, p \to o$ as syntactic sugar for n, l, p *when* $true \to o$.

Definition 7. *Let $\mathcal{U}$ be a set and $m \in \mathbb{N}$. A sequence $\mathcal{S}$, also noted $\langle u_i \rangle_{i\in[1,m]}$, is a total function from $\mathbb{N}$ to $\mathcal{U}$ such that:*

$$\begin{array}{ll} \mathcal{S}(k) = u_k & (\textit{if } k \in [1,m]) \\ \mathcal{S}(k) = \Omega & (\textit{otherwise}) \end{array}$$

Notations

- $\mathcal{S}(k)$ will be refered to as the *k-th element of the sequence $\mathcal{S}$*, the notation $\langle u_i \rangle_{i\in[1,m]}(k)$ will also be used.
- The empty sequence $\langle u_i \rangle_{i\in\emptyset}$ is noted $\langle\rangle$.

Definition 8. *A DCDL program π is a 5-tuple $(\mathcal{V}, \Sigma, \mathcal{N}, n_0, \mathcal{R})$ where $\mathcal{V}$, Σ and $\mathcal{N}$ are three pairwise distinct sets of symbols that are respectively a set of variables, an alphabet, and a set of rule names, $n_0 \in \mathcal{N}$ is the initial rule, and $\mathcal{R}$ is a finite sequence of rules.*

4.5.2 Semantics

Now that we defined our language, we need to define its semantics.

Definition 9. *Let $\mathcal{V}$ be a set of variables. An* environment *Γ is a mapping from variables in $\mathcal{V}$ to labeled trees. The access to the labeled tree associated to the variable $x \in \mathcal{V}$ is noted $\Gamma(x)$.*

Definition 10. *Let Σ be a set of symbols. $T(\Sigma)$ is defined as the set of labeled trees whose labels are in Σ.*

Definition 11. *The* matching *of a pattern p against a labeled tree t is either an environment from the variables in $vars(p)$ to subtrees of t or an error Ω, and is defined as follows:*

$$\begin{array}{rcll} t/x & = & \{x \mapsto t\} & (x \in \mathcal{V}) \\ t/\_ & = & \{\} & \\ t/p_1 \,\&\, p_2 & = & t/p_1 \,\cup\, t/p_2 & \\ t/p_1 \mid p_2 & = & \begin{cases} t/p_1 & (\textit{if } t/p_1 \neq \Omega) \\ t/p_2 & (\textit{otherwise}) \end{cases} & \\ c(t_1,\ldots,t_n)/c'(p_1,\ldots,p_m) & = & \begin{cases} \bigcup_{i=1}^{n} t_i/p_i & (\textit{if } n = m \wedge c = c') \\ \Omega & (\textit{otherwise}) \end{cases} & \end{array}$$

with $\Omega \cup e = \Omega$, $e \cup \Omega = \Omega$.

Definition 12. *Let $n \in \mathcal{N}$, $\pi = (\mathcal{V}, \Sigma, \mathcal{N}, n_0, \mathcal{R})$ be a DCDL program, and $t \in T(\Sigma)$. The evaluation of π on t noted $\pi(t)$ is defined as:*

$$\pi(t) = eval(\pi, \{\}, n_0, \langle\rangle, t)$$

where eval is defined as:

$$\begin{aligned}
& eval(\pi, \Gamma, n, \langle t_i \rangle_{i\in[1,m]}, t) \\
& \quad = eval\_out(\pi, \Gamma \cup t/p \cup \textstyle\bigcup_{i=1}^{m} \{x_i \mapsto t_i\}, o) \quad && (\textit{if } select\_rule(\mathcal{R}, \Gamma, n, t) = n, \langle x_i \rangle_{i\in[1,m]}, p \text{ when } e \to o) \\
& \quad = \Omega && (\textit{if } select\_rule(\mathcal{R}, \Gamma, n, t) = \Omega)
\end{aligned}$$

and where the auxiliary functions eval_out and select_rule are defined as:

$$\begin{aligned}
eval\_out(\pi, \Gamma, x) &= \Gamma(x) \\
eval\_out(\pi, \Gamma, n[o_1, \ldots, o_m](o)) &= eval(\pi, \Gamma, n, \langle eval\_out(\pi, \Gamma, o_i) \rangle_{i\in[1,m]}, eval\_out(\pi, \Gamma, o)) \\
eval\_out(\pi, \Gamma, c(o_1, \ldots, o_m)) &= c(eval\_out(\pi, \Gamma, o_1), \ldots, eval\_out(\pi, \Gamma, o_m))
\end{aligned}$$

$$select\_rule(\mathcal{R}, \Gamma, n, t) = \langle n, l, p \text{ when } e \to o \in \mathcal{R} \mid t/p \neq \Omega \ \wedge \ eval\_expr(\Gamma \cup t/p, e) = \mathit{true} \rangle \ (1)$$

where eval_expr is a total function from expressions to booleans.

The *select_rule* auxiliary function selects the first rule (with respect to the order in $\mathcal{R}$) that has the name given as argument and such that the input tree matches its pattern. So *select_rule* corresponds to a pattern matching with a first-match policy.

4.5.3 Properties

Definition 13. *We define the directed graph $\mathcal{G}_{\mathcal{R}} = (V, A)$ of a set of rules $\mathcal{R}$ as the graph such as:*

- *there is one node in V for each rule in R;*
- *two nodes v_1 and v_2 are linked by an arc $a_{v_1 v_2}$ if and only if a call to the rule v_2 appears in (the output expression of) v_1.*

Intuitively, $\mathcal{G}_{\mathcal{R}}$ represents the static call graph of a DCDL program.

Definition 14. *The set of* strongly connected components *of a node n in a graph $\mathcal{G}_{\mathcal{R}} = (V, A)$, noted $\mathcal{C}^{\mathcal{R}}(n)$, is the biggest set of nodes in V such that $n \in \mathcal{C}^{\mathcal{R}}(n)$ and every node in $\mathcal{C}^{\mathcal{R}}(n)$ is reachable from every other node in $\mathcal{C}^{\mathcal{R}}(n)$.*

Definition 15. *A DCDL program $\pi = (\mathcal{V}, \Sigma, \mathcal{N}, n_0, \mathcal{R})$ is well-formed if:*

$$\forall ((n, l, p) \text{ when } e \to o) \in \mathcal{R}, \forall n'[o'_1, \ldots, o'_m](o') \subset o, n' \in \mathcal{C}^{\mathcal{R}}(n) \implies o' \in subvars(p)$$

Note

In the rest of this document, we will assume that DCDL programs are well-formed.

The point of a program being well-formed is that one cannot call a rule in the same set of strongly connected rules unless the tree given as input to the called rule is "smaller" than the input of the calling rule. This way, we avoid non-termination since either we reduce the number of rules that can be called or we reduce the input. This will come into play later in the proof of termination of a DCDL program.

```
rule1 ::= Lcons(x, Lcons(_) as y) -> rule2(y)

rule2 ::= Lcons(x, y) when x = y -> rule3[Lnil](x)

rule3[acc] ::= Tcons(id, x, tail) -> rule3[Lcons(x, acc)](tail)
rule3[acc] ::= Tnil -> rule1(acc)
```

This example would be rejected: in the last line, `acc` is not a subvariable of the pattern of `rule3`, therefore we could have an infinite loop here. It is not the case, but we would need a more complicated condition to accept this example and still ensure termination.

Definition 16. *Let $\mathcal{G}_\mathcal{R} = (V, A)$ be the graph of the set of rules $\mathcal{R}$. Let $(\mathcal{C}_i^\mathcal{R})$ be the sequence of sets of strongly connected components of $\mathcal{G}_\mathcal{R}$ ordered by topological sort.*[5] *The binary relation $<_\mathcal{C}$ on sets of strongly connected components is defined as:*

$$i < j \Leftrightarrow \mathcal{C}_i^\mathcal{R} <_\mathcal{C} \mathcal{C}_j^\mathcal{R}$$

Note

$<_\mathcal{C}$ is a well-founded order since there is an equivalence between $<_\mathcal{C}$ and the standard ordering of the natural numbers.

Definition 17. *Let t_1 and t_2 be labeled trees. The binary relation $\sqsubseteq$ is a relation such that $t_1 \sqsubseteq t_2$ if and only if t_1 is a subtree of t_2 or exactly t_2.*

Proposition 4.1. *The binary relation $\sqsubseteq$ is a well-founded order.*

Proof. We consider labeled trees that are finite and do not have cycles as input for a DCDL program, therefore the proof is immediate. □

Theorem 4.2. *Let $\pi = (\mathcal{V}, \Sigma, \mathcal{N}, n_0, \mathcal{R})$ be a DCDL program, and $t \in T(\Sigma)$. The evaluation $\pi(t)$ terminates.*

Proof. Our language gives us the right to write finite patterns over finite constructions which evaluation obviously terminates since there is no recursion in these patterns; external constructions which evaluation terminates by hypothesis; and calls to rules which are the only point where non-termination can occur.

Suppose that there can be an infinite reduction of the program π on an input tree t, since the sequence $(\mathcal{C}_i^\mathcal{R})$ is finite, then there is a set of rules $\mathcal{C}^\mathcal{R}$ such that these rules are called an infinite number of times. However, since a program is well-formed, these rules are called on trees (that are finite) of strictly decreasing size, therefore the reduction ends which is a contradiction.

There cannot be an infinite reduction of a DCDL program, so a DCDL program always terminates. □

4.5.4 Concrete syntax

In this section, I show how DCDL is used in practice.

```
main ::= Project(Lambda(Var(x), ids), Join(Scan(Table(t1)), Scan(Table(t2)))
    -> "select␣" + tuple(ids) + "␣from␣" + t1 + "␣natural␣join␣" + t2

tuple ::=
| Tnil -> ""
| Tcons(id, value, Tnil) when id = value -> tdestr(value)
| Tcons(id, value, Tnil) -> tdestr(value) + "␣as␣" + id
```

[5] There are several possible sequences that respect this definition, we pick one among them.

```
| Tcons(id, value, tail) when id = value -> tdestr(value) + ",␣" + tuple(tail
   )
| Tcons(id, value, tail) -> tdestr(value) + "␣as␣" + id + ",␣" + tuple(tail)

tdestr ::=
| Tdestr(Tcons(_) & z, id) -> findInTuple[id](z)
| Tdestr(_, id) -> id

findInTuple[id] ::=
| Tcons(tid, value, _) when tid = id -> value
| Tcons(_, _, tail) -> findInTuple[id](tail)
```

This example translates a “select ...from ...join” expression in QIR that only contains column names in the “select” expression:

```
findInTuple[id] ::=
| Tcons(tid, value, _) when tid = id -> value
| Tcons(_, _, tail) -> findInTuple[id](tail)
```

`findInTuple` is a simple recursive function that returns the value mapped in the input tuple to the identifier `id` given as accumulator;

```
tdestr ::=
| Tdestr(Tcons(_) & z, id) -> findInTuple[id](z)
| Tdestr(_, id) -> id
```

`tdestr` is a simplified implementation of the tuple destructor;

```
tuple ::=
| Tnil -> ""
| Tcons(id, value, Tnil) when id = value -> tdestr(value)
| Tcons(id, value, Tnil) -> tdestr(value) + "␣as␣" + id
| Tcons(id, value, tail) when id = value -> tdestr(value) + ",␣" + tuple(tail
   )
| Tcons(id, value, tail) -> tdestr(value) + "␣as␣" + id + ",␣" + tuple(tail)
```

`tuple` returns a serialized version of a tuple in SQL style, the only specificity is that we need an extra operation if the considered value is under an alias, i.e. if `id` $\neq$ `value`;

```
main ::= Project(Lambda(Var(x), ids), Join(Scan(Table(t1)), Scan(Table(t2)))
    -> "select␣" + tuple(ids) + "␣from␣" + t1 + "␣natural␣join␣" + t2
```

finally, the `main` rule creates the “select” expression using the `tuple` rule.

One can notice that the rules are organized differently from the previous examples. Indeed, in order to make the language usable, one of the syntactic sugar to add is the possibility to factorize rules with the same name and the same accumulators. So I can write:

```
findInTuple[id] ::=
| Tcons(tid, value, _) when tid = id -> value
| Tcons(_, _, tail) -> findInTuple[id](tail)
```

instead of:

```
findInTuple[id] ::= Tcons(tid, value, _) when tid = id -> value
findInTuple[id] ::= Tcons(_, _, tail) -> findInTuple[id](tail)
```

with a syntax inspired by the pattern matching in OCaml and ℂDuce.

Additionally, I use an infix syntax for `+` (here concatenation of strings) and other usual constructors.

Other things to note in this example are:

- the utilisation of the `when` expression to check the equality of subtrees or values;
- the reconstruction of an SQL expression in a string format in the output expressions, as well as the use of calls to rules in these expressions;

- rules that are in the same set of strongly connected components in the call graph of the program as the current rule are called on subtrees of the input tree, indeed `tuple` calls itself recursively on the tail of the input tuple and it is the same for `findInTuple`;
- in particular, the accumulator in `findInTuple` is not used as argument.

This example shows that DCDL can express transformations in a concise way, where the equivalent in Java would need much more lines of code and low-level manipulations.

5 Related work

In this section, I will describe already existing extensions that allow users to write queries in a programming language and compare them with our framework.

Traditional solutions

A classic solution, used in MySQL for PHP[14], JDBC for Java[15] and others consists in representing a query by a string that is interpreted by the external library. Here is an example of a program that uses this solution:

```
// conn being a connection to the database
Statement stmt = conn.statement("SELECT␣*␣FROM␣Customer␣WHERE␣city=?");
stmt.setString(1, "Seattle");
ResultSet rslt = stmt.executeQuery();
String companyName = rslt.getString(1);
String contactName = rslt.getString(2);
```

Even though this method makes code simple to understand, it is verbose, poorly integrated to the language, and makes composition of queries difficult. Also, any error in a query is only caught at runtime when it is fully known.

ORMs

Object-relational mappings (ORM) aims to convert data between incompatible type systems in object-oriented programming languages. Most of the inconvenients of traditional solutions are found in ORMs, for example Hibernate[3] relies on queries stored as strings and expressed in a language called *HQL* which is very close to SQL, but they also propose syntactic abstractions such as *Criteria* queries that use constructions of the language. The example above using Criteria in Hibernate would be:

```
// session being a connection to the database
Criteria criteria = session.createCriteria(Customer.class);
criteria.add(Restrictions.eq("city", "Seattle"));
Iterator customer = criteria.list().iterator();
String companyName = customer.getCompanyName();
String contactName = customer.getContactName();
```

The query is described using high-level expressions, therefore it is abstracted from a particular database language such as SQL. However, ORMs are still heavily restricted to the API: most of the application code in a query has to be executed in the application resulting in the round trips described in Section 1.1 and performance issues.

LINQ

Language Integrated Query (LINQ)[2] is a component of the .NET Framework that adds native data querying capabilities to .NET languages. Instead of trying to make a correspondance between the object-oriented model and the relational one, LINQ defines queries as a first-class concept within the language semantics. Our running example in LINQ would be written this way:[6]

```
// db being a connection to the database
var results = from c in db.Customer
              where c.city == "Seattle"
              select c;
var result = results.FirstOrDefault();
String companyName = result.companyName;
String contactName = result.contactName;
```

which is more concise than the previous examples. Moreover, it allows LINQ to offer a degree of static type checking lacking in other approaches.

However, LINQ cannot send application code that appear in queries to the database, and unlike our framework, it can only rely on the capabilities of the database itself as it does not make the assumption that a runtime of the language is present in the back-end. Thus, LINQ is limited in the expressivity of its queries as UDFs have to be executed in the application side.

In **A Practical Theory of Language-Integrated Query**[10], efforts are made to separate application code and code to be sent to the database using a system of quotations/anti-quotations. The authors also defined a type system ensuring that a well-typed query can be translated into SQL.

But since they cannot use expressions from the host language in their queries, it is impossible in T-LINQ to have nesting of quotations. Therefore the following example in QSL:

```
function f(x, y) {
  return select table_name from $x where id = $y;
}

function g(x, y) {
  return select id from $f($x,$y) where age < 20;
}
```

has no equivalent in T-LINQ.

Other contributions

SML#[16] 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. This work should be useful to us in the future as we intend to create a type system for QIR.

Efforts have been made to extend the Scala programming language[17] to the expression of queries using the syntax of LINQ and the native Scala syntax for comprehensions[18], and taking advantage of the strong static type system to analyse the type safety of queries at compile-time.

6 Conclusion and future work

In this report, I described my work during my internship at Oracle Labs: I implemented in Java a representation of the QIR language and used it as an intermediate representation of queries; I extended SimpleLanguage to the expression of queries; I implemented in Java a translation from QIR to the Oracle database; and I defined a DSL which purpose is to allow a database to describe what QIR ASTs it can translate into its own representation of queries and how to do it. Combined with the works of the teams at Orsay and Oracle, we obtained a framework that allowed a general-purpose programming language to send a query containing user-defined functions to a database using the Truffle framework.

[6]The query can also be written `db.Customer.Where(c => c.city == "Seattle")`.

Our work not only shows that it is possible to efficiently evaluate application code in a database, thus breaking the barrier between general-purpose programming languages and databases to achieve a level of expressivity in database-oriented applications that was never reached before. It also shows that a high-level separation between the front-end and the database side is reachable even with this gain in expressivity, which gives an abstraction on both sides making the integration to the framework simple.

The next main step after this internship is to test our framework with more complex programming languages; different ways to integrate queries; and databases of different models. Besides, this document does not only describe problems that I solved, but also problems that remain open and too strong hypotheses that were made to restrict the scope of the internship. I will continue to work on this project next year during my PhD which will allow me to propose solutions to these problems and to test our framework on real-world applications.

References

[1] MongoDB Inc. http://docs.mongodb.org/manual/core/server-side-javascript/#Server-sideCodeExecution-Storingfunctionsserverside.

[2] Microsoft. Linq (language-integrated query), 2015. https://msdn.microsoft.com/en-us/library/bb397926.aspx.

[3] Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, and Steve Ebersole. Hibernate - relational persistence for idiomatic java, 2009. https://docs.jboss.org/hibernate/orm/3.3/reference/en-US/pdf/hibernate_reference.pdf.

[4] Jonathan H. Wage and Konsta Vesterinen. *Doctrine ORM for PHP*. Sensio SA, March 2010.

[5] Jacob Kaplan-Moss and Adrian Holovaty. *The Definitive Guide to Django: Web Development Done Right*. Apress, December 2007.

[6] Oracle Corporation. Truffle faq and guidelines, 2014. https://wiki.openjdk.java.net/display/Graal/Truffle+FAQ+and+Guidelines.

[7] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario 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. ACM, 2013. http://lafo.ssw.uni-linz.ac.at/papers/2013_Onward_OneVMToRuleThemAll.pdf.

[8] Christian Wimmer. One vm to rule them all. SPLASH 2014: http://lafo.ssw.uni-linz.ac.at/papers/2014_SPLASH_OneVMToRuleThemAll.pdf, October 2014.

[9] Rails core team. http://rubyonrails.org/.

[10] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 403–416. ACM, 2013. http://homepages.inf.ed.ac.uk/slindley/papers/practical-theory-of-linq.pdf.

[11] 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.

[12] Patrick Bahr and Laurence E. Day. Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM, 2013. http://www.diku.dk/~paba/pubs/files/bahr13wgp-paper.pdf.

[13] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63. ACM, 2003. http://www.cduce.org/.

[14] The PHP Group. MySQL. http://php.net/manual/en/book.mysql.php.

[15] Oracle Corporation. Java JDBC API. http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/.

[16] Atsushi Ohori and Katsuhiro Ueno. Making standard ml a practical database programming language. In *ICFP*, pages 307–319, 2011.

[17] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending scala with database query capability. In *Journal of Object Technology (JOT)*, volume 9, no. 4, pages 45–68, July 2010.

[18] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions. In *Haskell '07 Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 61–72. ACM, 2007. http://research.microsoft.com/en-us/um/people/simonpj/papers/list-comp/list-comp.pdf.

Appendices

A Truffle

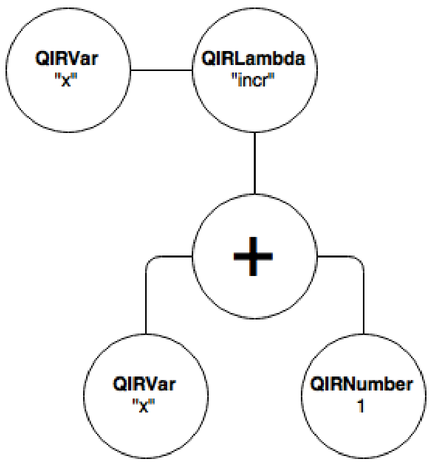
Here is an example of a Truffle node class for a literal number:

```java
public class NumberNode extends MumblerNode {
  public final long number;

  public NumberNode(long number) {
    this.number = number;
  }

  @Override
  public long executeLong(VirtualFrame virtualFrame) {
    return this.number;
  }

  @Override
  public Object execute(VirtualFrame virtualFrame) {
    return this.number;
  }

  @Override
  public String toString() {
    return "" + this.number;
  }
}
```

The node contains a Java long 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.

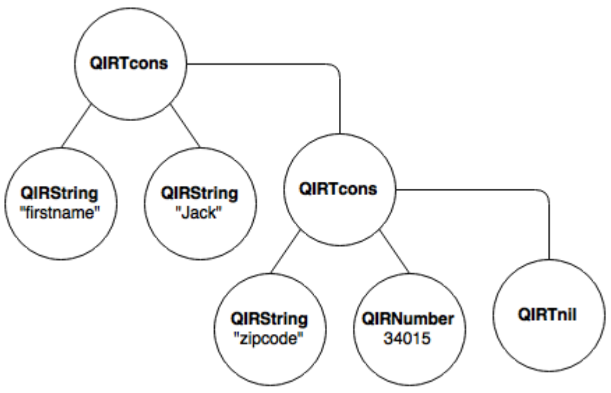
B QIR

In this section, I give examples of QIR trees with their equivalent representation in a programming language.



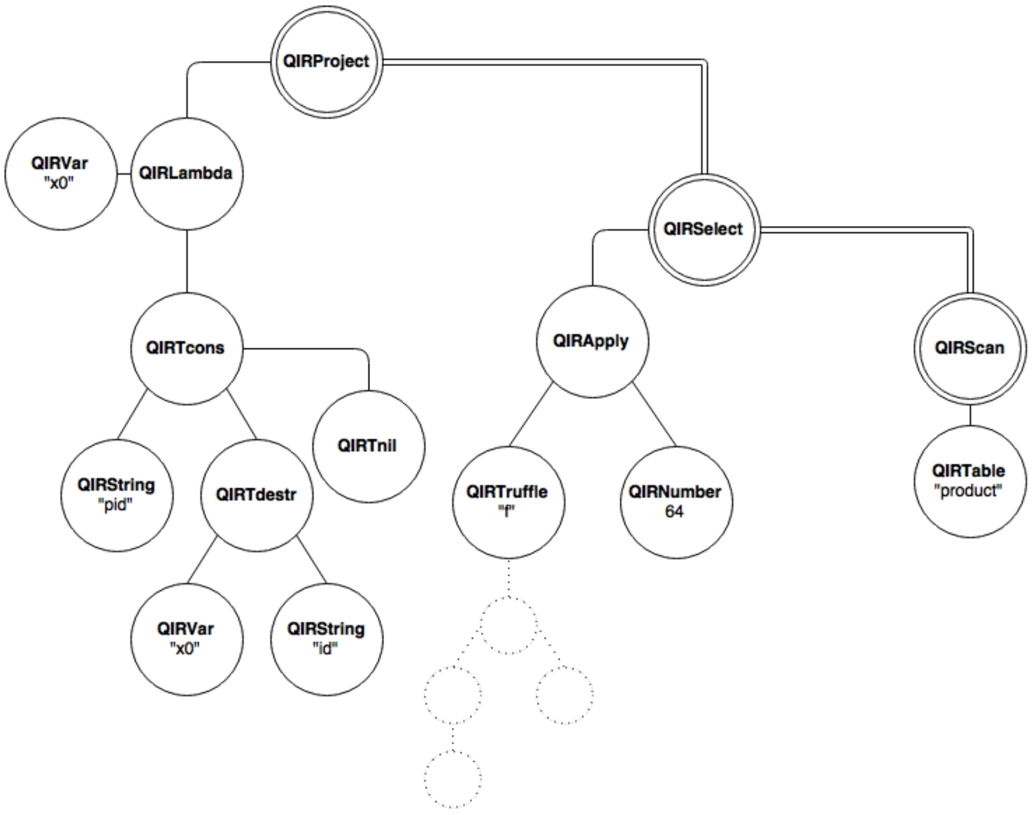
```
function incr(x)
{
  return x + 1;
}
```

Figure 4: Representation of a function in QIR



```
{
  firstname: "Jack"
  zipcode: 34015
}
```

Figure 5: Representation of a tuple in QIR



select id as pid from product where $f(64)

$Project_{\lambda p.\{pid:p.id\}}(Select_{<f>(64)}(Scan_{db.product}()))$

Figure 6: Representation of a query in QIR

C QSL

C.1 SL samples

This code is the SL version of the "Hello World!" program:

```
function main()
{
  println("Hello World!");
}
```

Another example of an SL program below:

```
  function main() {
    obj1 = new();
    println(obj1.x);
    obj1.x = 42;
    println(obj1.x);

    obj2 = new();
    obj2.o = obj1;
    println(obj2.o.x);
    obj2.o.y = "why";
    println(obj1.y);

    println(mkobj().z);

    obj3 = new();
    obj3.fn = mkobj;
    println(obj3.fn().z);

    obj4 = new();
    write(obj4, 1);
    read(obj4);
    write(obj4, 2);
    read(obj4);
    write(obj4, "three");
    read(obj4);
  }

  function mkobj() {
    newobj = new();
    newobj.z = "zzz";
    return newobj;
  }

  function read(obj) {
    return obj.prop;
  }

  function write(obj, value) {
    return obj.prop = value;
  }
```

C.2 QSL samples

Here is a simple example of a QSL program that processes the results of a query:

```
  function main()
  {
    o = select ename from emp;
    res = "";
    tuple = o.next();
    if (tuple != null)
    {
      res = res + tuple.getEname();
      tuple = o.next();
      while (tuple != null)
      {
        res = res + ", " + tuple.getEname();
        tuple = o.next();
      }
    }
    println(res);
  }
```

A more complex example using a UDF:

```
function dolToEuro(dol)
{
  return dol * 89 / 100;
}

function main()
{
  minsalary = 2500;
  o = select empno, ename, $dolToEuro(sal) as salary from emp where sal >=
      $minsalary;
  res = "";
  tuple = o.next();
  if (tuple != null)
  {
    res = res + "(" + tuple.getEmpno() + ", " + tuple.getEname() + ", " +
        tuple.getSalary() + ")";
    tuple = o.next();
    while (tuple != null)
    {
      res = res + ", (" + tuple.getEmpno() + ", " + tuple.getEname() + ", " +
          tuple.getSalary() + ")";
      tuple = o.next();
    }
  }
  println(res);
}
```

C.3 Grammar

The complete grammar of QSL is described below.

```
QSL ::= functionDef { functionDef }

functionDef ::= "function" id "(" [ id { "," id } ] ")" "{" block "}"

block ::= statement { statement }

statement ::= "break" ";"
| "continue" ";"
| "return" [ exp ] ";"
| exp ";"
| whileStatement
| ifStatement

whileStatement ::= "while" "(" exp ")" block

ifStatement ::= "if" "(" exp ")" block [ "else" block ]

exp ::= conjExp { "||" conjExp }
| query
```

```
query ::=
"select" formatter
"from" exp
[ "where" exp ]
[ "group␣by" queryExp ]
[ "order␣by" queryExp ]

formatter ::= exp [ "as" id ] { "," formatter }
| "*" { "," formatter }

queryExp ::= exp [ "as" id ] { "," queryExp }

conjExp ::= logicExp { "&&" logicExp }

logicExp ::= arithExp [ logicSymbol arithExp ]

logicSymbol ::= "<" | "<=" | ">" | ">=" | "==" | "!="

arithExp ::= starExp { arithSymbol starExp }

arithSymbol ::= "+" | "-"

starExp ::= term { starSymbol term }

starSymbol ::= "*" | "/"

term ::= id "(" [ exp { "," exp } ] ")"
| id "=" exp
| id { "." id }
#if inQuery
| "$" id "(" [ exp { "," exp } ] ")"
| "$" id "=" exp
| "$" id { "." id }
#endif inQuery
| string
| number
| "(" exp ")"

id ::= letter { alphanum }

number ::= digit { digit }

letter ::= 'A'..'Z' | 'a'..'z'

digit ::= '0'..'9'

alphanum ::= letter | digit
```

To simplify the grammar, I use the notation “#if inQuery ... #endif inQuery” to indicate that the constructions inside these guards are valid only if inside a query expression. In other words, the anti-quotation symbol “$” is only valid in a query (inside quotations).