
# Language-Integrated Queries in Scala

Kaichuan Wen
20729335

December 2009

supervised by
Prof. Dr. Sibylle Schupp
Prof. Dr. Friedrich Mayer-Lindenberg
Dr. Miguel Garcia

Hamburg University of Technology(TUHH)
Institute of Software Technology and Systems(STS)

## Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, Germany
Dec. 20th, 2009

Kaichuan Wen

**Acknowledgements**

First of all, I would like to thank Prof. Dr. Sibylle Schupp, the head of Institute of Software Technology and Systems(STS), for offering me this interesting Master Thesis topic, and her precious support, valuable advice and encouragement for my work.

I would also like to thank my second supervisor Prof. Dr. Friedrich Mayer-Lindenberg, head of Institute of Computer Technology, for his interest in my Master Thesis work.

Many thanks are to my co-supervisor Dr. Miguel Garcia, for his generous suggestions and guidance. I really enjoyed the moments when we exchanged ideas.

Thanks are also to other staffs in STS, for providing me with comfortable working conditions and friendly atmosphere.

Last but not least, special gratitude goes to my parents and grandparents in China. I can never thank you enough for all your endless support and love.

**Abstract**

The integration of modern programming language and functional database query languages brings in new innovation in data access mechanism. On the one hand, programming languages combine object-orientation paradigm and functional programming style, benefiting from the joint advantage and convenience of both. On the other hand, functional database query languages like LINQ provide a manner of *language-integrated query*, which enables deep participation of the host-language in query processing.

This Master Thesis investigates and implements this integration. It goes one step further by achieving another integration of object model and relational model. This O/R mapping is implemented through the translation into an intermediate language. The resulting prototype enables a scenario where queries in functional query language(LINQ), embedded in programming language(Scala), are triggered on object model(EMF), which has been persisted in relational model(DBMS). In this scenario, each party acts out its own expertise respectively.

In this thesis it is also argued why (a) the persistent representation(in the RDBMS) is isomorphic to its programming language counterpart(in main-memory); and (b) similarly for queries at the programming and query language level.

**Keywords:** Language-integrated, Query, ORM, Object model, Relational model, Isomorphism, EMF, LINQ, Ferry, Scala

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Modern development of programming languages and database technologies raises higher demand on the integration of them. On the one hand, programming languages like Scala and F# combine object-orientation paradigm and functional programming style. With this combination, object model now also benefits from the advantage and convenience provided by functional programming, including the enriched support for immutable collections and list comprehensions, one foundation of query processing. On the other hand, functional database query languages like LINQ provide a manner of *language-integrated query* to query on object model, also based on the support of functional operations and list comprehensions. This integration enables deep participation of host-language in query processing, in particular, regarding type checking and type inference.

However, from an abstract point of view, this integration does not eliminate the mismatch between object model and relational model. Besides that, Object-Relational-Mapping(ORM) should also play its role. The open question of O/R mismatch can be addressed by low-level processing of both models. This processing can also benefit from the integration of functional programming and the support of list comprehensions. For example, in O/R mapping, collections in object model can be expanded and processed as comprehensions in order to adhere to `1NF`(First Normal Form) in relational model.

This Master Thesis achieves an O/R mapping prototype in a language-integrated manner, through the application of functional programming as well as processing of list comprehensions. This mapping is bi-directional, supporting both *persistence* from object model to relational model, and the reformulation of *query* results from relational model back to object model. We also preserve the isomorphism between models in both directions of mappings, so as to ensure the correctness and applicability of this prototype.

## 1.2 Background

### 1.2.1 Involved Technologies

- EMF

Eclipse Modeling Framework (EMF) is a data modeling and code generation facility for building tools and other applications based on a structured object-oriented data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the object-oriented model in question.

In our project, we adopt a subset of EMF to construct our object model as the departure point of our journey of translation and query. This subset mainly involves its static modeling facility, meaning that we use EMF to model classes, fields of them, as well as relationship and interplay among them, rather than their runtime behaviors. Also for the persistence, we will address the problem of static object population, which is instance of classes modeled by EMF, but do not consider any dynamic object instantiation as part of the chosen EMF subset.

- LINQ

Under the term "LINQ", actually several related technologies are involved, including (a) a language-embedded functional query language, which is the foundation for LINQ to be a query language; (b) Query Provider, allowing possibly third-party back-end specific query implementation; (c) a software component (ADO.Net Entity Framework), which provides O/R mapping facility by introducing a new conceptual data model[1]; and (d) a LINQ-aware Integrated Development Environment (IDE), offering usability features such as syntax completion(*IntelliSense* in the newest versions of Microsoft Visual Studio).

The concept underlying LINQ, in particular the query language itself, is however platform-independent. This enables us to employ LINQ as the query language on object model(in particular, on the one generated with EMF), and to implement the translation from it based on publicly available specifications only.

- Ferry

Developed by the team at University of Tübingen[1], Ferry is an intermediate language that provides the possibility of generating relational query language by translating from query languages of other model, such as LINQ, LINKS or Ruby. Ferry runs on data model of a relatively lower level, supporting comprehensions of records, lists and tuples. Ferry's syntax foundation is the `for-where-group by-order by-return` construct, which is also that

[1] `http://www-db.informatik.uni-tuebingen.de/research/ferry`

of most functional query languages supporting list comprehensions. This provides the advantage of translating query language on higher data level(in our project, LINQ on object model) to a lower one with high level guarantee of syntax preservation.

We translate LINQ query into Ferry, from which it can then be further translated and optimized into relational algebra and then into target language SQL:1999.

- Scala

We use Scala as our host language because it is a functional language with rich libraries supporting list comprehensions, yet it also seamlessly supports object-orientation paradigm. Further, it provides a more flexible compiler-extension architecture, which enables deeper participation of compiler in the translation process of queries, and hence yielding a higher level of language integration.

### 1.2.2 Related work

Language-integrated query on object models is a relatively large topic and several related works have been reported. A previous Student Project Work by the same author [2] implements a prototype that enables embedded LINQ query in Java. The resulting prototype LINQExpand4Java[2] further expands the embedded query through the participation of a compiler-plugin. Rather than performing a translation on object model, Master Thesis [3] implements the translation of records and comprehensions from LINQ to Scala then to Ferry, also adopting the compiler-plugin mechanism. Also, Master Thesis [4] implements a LINQ-SQL query provider to allow evaluation of LINQ query in a DBMS. Because of the nature of LINQ-SQL Query Provider, the underlying data model is thta of SQL'92, ie. object orientation is also not addressed. Technical report [5] aims at achieving a automatic translation between database and a more convenient representation in the software by providing generic versions of the elementary CRUD(Create, Read, Update, Delete) operations.

## 1.3 Overview

### 1.3.1 Supported use case

Being a proof of concept rather than commercial product, the reported prototype supports such a use case: it takes EMF model along with the object population underlying this model as input. During the persistence phase, this input is stored in DBMS, and necessary information and meta data are

[2] http://www.sts.tu-harburg.de/people/mi.garcia/LINQExpand4Java/

created for the preparation of query phase. After the persistence, queries on this model in terms of LINQ are available. These queries are first translated to Ferry expressions, with the help of the meta data provided above. Using Ferry compiler `ferryc`, the resulted Ferry expressions can then be translated to relational algebra, from where target SQL:1999 expressions are generated for database evaluation.

### 1.3.2 Outline of this report

The rest of this Master Thesis is organized as follows.

Chapter 2 examines the phase of persistence from EMF object model to relational model. Section 2.1 introduces some necessary data structures used in the persistence phase and later also in the query phase. Section 2.2 gives an overview of the persistence phase, as well as its two sub-phases. Section 2.3 and 2.4 explain in detail these sub-phases in regarding to the adopted algorithm, implementation and results.

Chapter 3 moves ahead to the query phase by discussing the translation from LINQ to Ferry. After an overview of the translation in Section 3.1, a normalization step is discussed in Section 3.2. The actual translation rules will then be investigated in Section 3.3. Section 3.4 presents some implementation issues and the final result of this phase. Section 3.5 concludes this chapter by a brief introduction to the following step.

Chapter 4 focuses on relational query plans in terms of relational algebra. First each operator of relational algebra will be given in Section 4.1, following by Section 4.2 that introduces query plans by combining these operators. Section 4.3 explains the implementation aspect, including translation from relational algebra into SQL statements and the well-formedness checking. Section 4.4 shows a method and tooling for generation of query plans for automated testing.

Chapter 5 argues the isomorphism between object model and relational model in both persistence and query phases, followed by Chapter 6, which gives the conclusions from this Master Thesis, and proposes possible future works.

The source code of the prototype can be found at `http://www.wen-k.com/msc` as well as in the attached CD.

# Chapter 2

# Persistence: EMF to Relational Model

This chapter covers the aspect of persistence of EMF model into relational data model.

First in Section 2.1, some important data structures that are used in this process are introduced. These data structures are also referred to in Chapter 3.

In Section 2.2, two main steps of the persistence are introduced, as well as the reason why this process are divided into 2 steps: the persistence of EMF class schema and that of EMF object population. The term *EMF class schema* refers to the classes, the structures of them as well as inter-class relationships such as inheritance, interface implementation and so on. On the other hand, the term *EMF object population* refers to actual runtime objects that are instances of the EMF class schema. This section also clarifies the chosen ORM schemes and makes explicit some assumptions and restrictions during the persistence process.

In Section 2.3 and Section 2.4, persistences of both EMF class schema and EMF object population are explained in detail, respectively.

## 2.1 Required data structures

### 2.1.1 Ferry Types

All data that Ferry can process must be of the types illustrated in Figure 2.1.

`Tuple type` of Ferry represents records or objects that other languages like LINQ can process. Each element of a `Tuple type` is of type `boxed type`, which is an abstract type that can be either `atomic type` or list of `Ferry type`. There is no distinction at the level of relational representation between a 1-tuple of atomic and the atomic itself. Similar to List in most

| | |
|---|---|
| $t ::= (b, b, \ldots, b)$ | Tuple Type |
| $\mid\ b$ | Boxed Type |
| $b ::= [t]$ | List Type |
| $\mid\ a$ | Atomic Type |
| $a ::= int \mid string \mid bool \mid \ldots$ | Atomic Basic Type |

Figure 2.1: Types of Ferry

programming language, Ferry `list type` represents a collection of elements, each of which is of the abstract `Ferry type`, as defined recursively. `Atomic type`, as the terminal of the recursive definition, can be one of the primitive types such as Int, Boolean, String, and so on. Figure 2.2 illustrates the UML diagram of the type hierarchy of Ferry types.

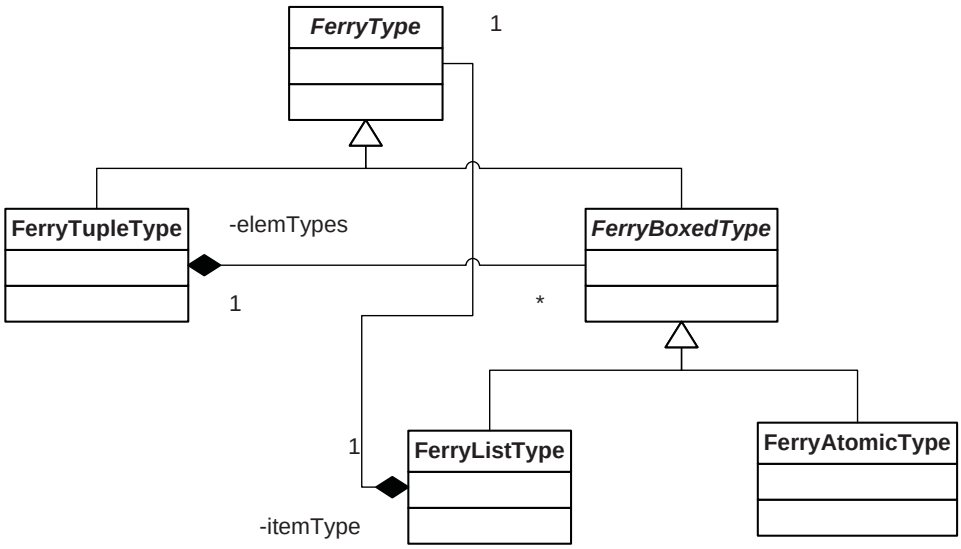


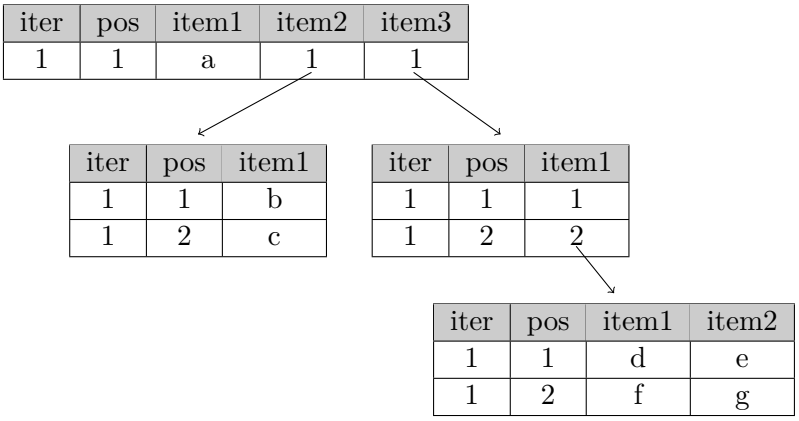

Figure 2.2: UML diagram of Ferry types

Although a Ferry tuple is used to represent a record or an object, according to the above definition, the element of a Ferry tuple cannot be of Ferry `tuple type` in turn. This restriction ensures that a Ferry tuple can only contain plain value but not nested structure, and hence be able to fit into a row of a table. To represent nested structure(for example, reference in an object), the nested tuple should be located within a list, regardless of whether it is the unique element within the list, and the pointer to the list, a surrogate value, is then included as one element of the nesting tuple. The following sections cover more detail about the mapping technique of references within objects, as well as the loop-lift algorithm used to process

nested list within a tuple.

### 2.1.2 Table Nodes and Table Information Nodes

Conceptually, an object population is stored in instances of a data structure called `Table Node`. Similar to tables in database, each node of this kind corresponds to one class, holding one instance of this class in a row. Through the loop-lift algorithm, different table nodes of this kind are connected with each other due to references and list expanding, hence leading to a tree-shaped topology. Figure 2.3 illustrates an example, showing how a tuple `("a",["b","c"], [("d","e"), ("f", "g")])` is mapped into such a tree. More details about this encoding can be found in Section 2.3 and 2.4.




Tuple `("a",["b","c"],[("d","e"),("f", "g")])`

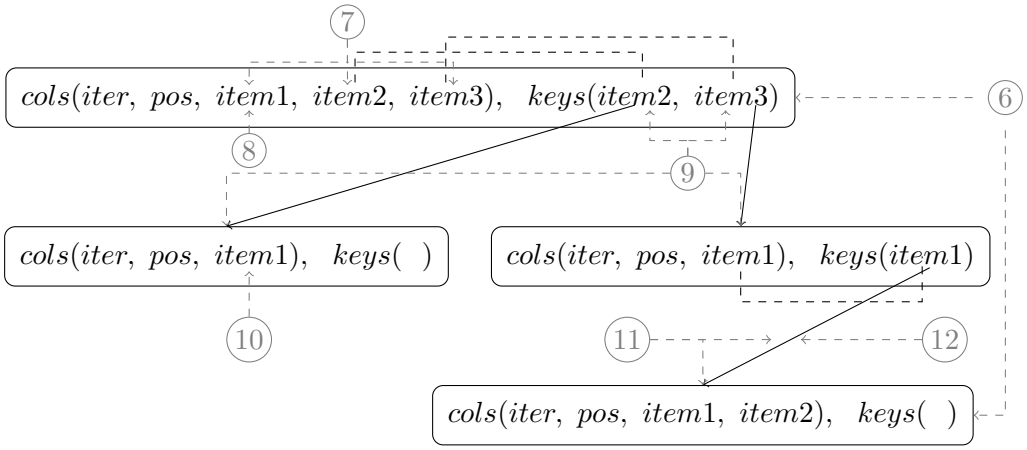
Figure 2.3: Tree of Table Nodes

Analogous with tables in database, each of which has its own table schema, each Table Node also has its descriptive meta data, specifying for example the names and types of all columns, as well as the key information that specifies the link between one node and another via reference and list expanding. A data structure called `Table Information Node` is defined, as illustrated in Listing 2.1. Also in Listing 2.1 the definition of `key` is given, which specifies how these nodes are connected with each other.

```
1 case class TableInfoNode(
2       var cols:List[(String,FerryType)],var keys:List[key]){}
3 case class key(
4       var name:String, var order:Int,
5       var pointTo:TableInfoNode,var keyType:FerryType){}
```

Listing 2.1: Definition of `Table Information Node`

Since each Table Information Node corresponds to one Table Node, group of related Table Information Nodes therefore build up the same tree-shaped topology as that built up by their corresponding Table Nodes. Figure 2.4 shows the tree of Table Information Nodes corresponding to the tree of Table Nodes in Figure 2.3.(Given that no column names are specified, the default names "item1","item2",... are assigned.)




*This tree of Table Information Nodes corresponds to tree of Table Nodes in Figure 2.3. Circled numbers illustrate the actions of transformation rules, which will be introduced in Section 2.3.1.*

Figure 2.4: Tree of Table Information Nodes

However, on the level of implementation, it is not practical nor scalable to hold the entire object population in main memory, especially when dealing with object models of large scale. Instead, only the meta data of these Table Nodes, the Table Information Nodes, are hold in main memory, and the values of object population are stored in tables in database. In this sense, instance of Table Node does not need to exist at all.

To sum up, the data structure Table Information Node serves as the meta data of a conceptual data structure Table Node, the relation between which are identical to that between database table schema and database table. Due to scalability concern, Table Nodes are not kept in main memory. Nevertheless, Table Information Node is still adopted, since Table Node and database table should be semantically equal, and the meta data of the former should also be competent in describing the latter. Therefore, Table Information Node is used as the meta data of database tables. It is such kind of meta data that, unlike normal database schema, is in terms of Ferry Type and hence ready to participate in queries that are written in LINQ and later translated into Ferry expressions. Details about translating queries from LINQ to Ferry expression will be covered in Chapter 3.

## 2.2 Overview of the persistence

### 2.2.1 The separated phases

Persistence of EMF model includes two phases, first mapping the EMF class schema into database and get the meta data of it at the same time, and second mapping the object population that are instances of the EMF class model into the database.

One reason of this separation is the scalability concern, as mentioned in Section 2.1.2. Furthermore, persistence of EMF class schema involves the construction of trees of Table Information Node and a number of table creation operations onto the database, which are to be executed only once, given no modification to the model. Persistence of the EMF object population, on the other hand, involves then the insertion operations into the tables and these are to be executed more frequently when new objects are introduced.

Figure 2.5 illustrates both phases of the sketched persistence process. Persistence of EMF class schema will be introduced in detail in Section 2.3, followed by persistence of EMF object population in Section 2.4.

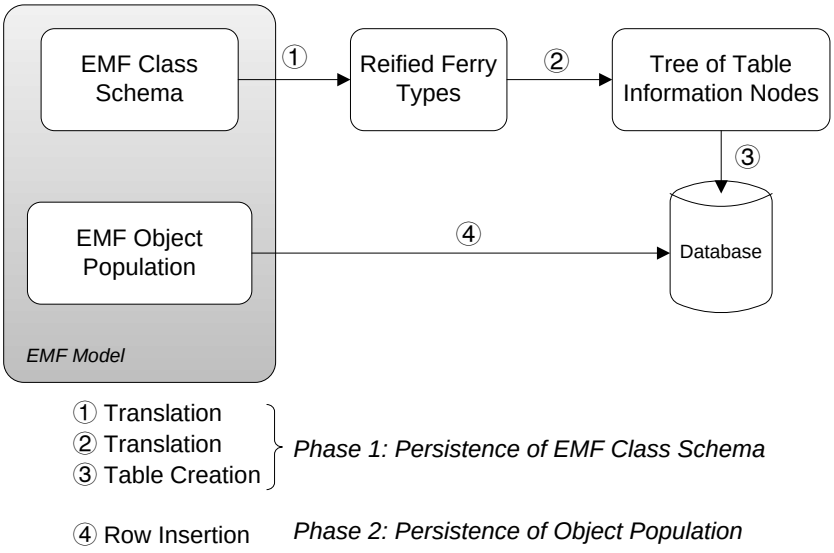



Figure 2.5: Two phases of the persistence

### 2.2.2 ORM scheme

There are many different object-relational mapping schemes, none of which is ideal for all situations. The scheme, by which one class is mapped to one table, is adopted in our work. With this scheme, inheritance can be handled straightforwardly, in that new table is merely to be added once new

sub-class is introduced in the hierarchy. What's more, data size grows in direct proportion to growth in the number of objects, hence yielding good scalability for large data model.

With one-table-per-class scheme, each table only holds values within the "class-frame" of the corresponding class. For each class, its class-frame comprises the attributes and references(called *fields* together) declared in this class. Values of fields declared in super-class are within another class frame, and hence stored in the table corresponding to that super class. As a consequence, different parts of one object may be stored separately in different tables. While this enables better support to polymorphism, it also results in potential performance drawback, in that single object access may require access to multiple tables.

General ORM techniques also cover other aspects, such as how to map associations with high multiplicity. These techniques are relatively fixed and widely applicable, and they will be mentioned in the following sections about rules and algorithms of persistence of both EMF class schema and EMF object population.

### 2.2.3 Assumptions and restrictions

Before persistence of EMF class schema and EMF object population can be explained in detail, some assumptions and restrictions should be clarified first.

First, it is assumed that all classes in an EMF model inherit from either `EObject` or other class within the same EMF model, although they can be within different packages declared in the EMF model. In other words, it is impossible to process classes that are inherited from any class in Java or Scala API or other external APIs. This restriction is in fact derived from EMF, in that one can only declare class inherited from classes within EMF framework or those within the same EMF model.

Second, similar to the first restriction, referenced objects can only be of EMF built-in types or types declared within the same EMF model. This restriction is again derived from EMF itself.

Further, we do not accept *Data Type* as part of the input. Data Type is another kind of classifier other than Class introduced in EMF. The distinction between Class and Data Type is somewhat similar to that between classes and primitive types in Java. Another main difference between them is how equality testing is performed. Two values v and w of the same datatype are compared for structural equality in the expression `v == w`, that is, pairs of fields in v and w are compared accordingly. In contrast, `v == w` in the case when `v` and `w` are instances of Class results in comparing their object identities. EMF has defined a group of Data Types that wrap the primitive types in Java, and these Data Types can serve as the type of attributes. While EMF allows user to define their own Data Type, we suggest and as-

sume our user to declare Class instead to represent relatively complicated structure. This ensures that attributes can only be of primitive types whose value can be persisted into DBMS directly. User trying to include Data Type in the input EMF model will receive an error message, prompting that the input should be adjusted to avoid Data Type. Figure 2.6 shows the UML diagram of EMF core metamodel that our prototype supports.

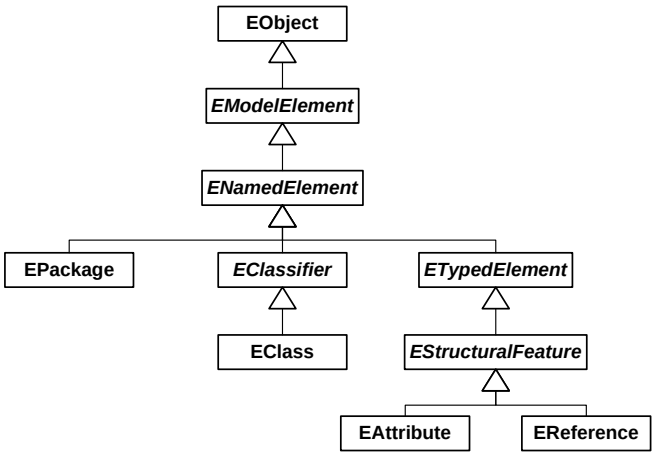



Figure 2.6: UML diagram of supported EMF core metamodel

## 2.3 Persistence of EMF class schema

### 2.3.1 Rules and algorithm

The persistence of EMF class schema involves three steps, first to generate instances of Ferry Types, then from which to generate trees of Table Information Nodes, followed by the creation of empty tables in database. These steps are illustrated in Steps *1,2* and *3* in Figure 2.5.

Given the following notations(Here we follow the Scala notation for collections, instead of the usual mathematic one like $A\{a_1, \cdots, a_n\}$.):

- $Classes(c_1, \cdots, c_n)$: the set of all classes and interfaces defined within the EMF model.
- $c \xrightarrow{\text{imp}} i$: class $c$ implements interface $i$.
- $Types(name_1 \rightarrow type_1, \cdots, name_n \rightarrow type_n)$: a map to hold the results, having names of the classes as keys, and instances of Ferry Tuple Type as values.
- $Types.keys = (name_1, \cdots, name_n)$: list of all keys within map $Types$

- $Types(key)$: retrieves the value in the map $Types$ according to key $key$.

The following rules and algorithm are applied when translating EMF class schema into Ferry Types:

1. Each class(or interface) $c$, if not yet been translated, is translated into an instance of `Ferry Tuple Type` $\widehat{ftt}$. Pair $(c.name \rightarrow \widehat{ftt})$ is added to map $Types$.

$$\frac{\forall c \ (\ c \in Classes \ \ \wedge \ \ c.name \notin Types.keys\ )}{\begin{array}{c} c \mapsto \widehat{ftt}, \\ Types \mathrel{+}= (c.name,\ \widehat{ftt}) \end{array}}$$

2. Each attribute $a$ in $c$ with upper bound=1 is translated into an instance of `Ferry Atomic Type` $\widehat{fa}$, whose name and atomic type are set accordingly. $\widehat{fa}$ is added to list of columns of $\widehat{ftt}$.

$$\frac{\forall a \ (\ a \in c.Atts \ \wedge \ a.UpperBound = 1\ )}{\begin{array}{c} a \mapsto \widehat{fa}, \\ \widehat{fa}.type = a.type,\ \widehat{fa}.name = a.name,\ \widehat{ftt}.cols \mathrel{+}= \widehat{fa} \end{array}}$$

3. Each attribute $a'$ in $c$ with upper bound>1 is translated into an instance of `Ferry List Type` $\widehat{fl}$ and an instance of `Ferry Atomic Type` $\widehat{fa'}$, the element type of the former is $\widehat{fa'}$, whose name and type are set to those of $a'$. $\widehat{fl}$ is added to the list of columns in $\widehat{ftt}$.

$$\frac{\forall a \ (\ a \in c.Atts \ \wedge \ a.UpperBound > 1\ )}{\begin{array}{c} a' \mapsto \ (\widehat{fl},\ \widehat{fa'}), \\ \widehat{fa'}.name = a'.name, \widehat{fa'}.type = a'.type, \widehat{fl}.type = \widehat{fa'},\ \widehat{ftt}.cols \mathrel{+}= \widehat{fl} \end{array}}$$

4. Each class of reference $c_R$, if there exists no element in $Types$ whose key equals to $c_R.name$, is translated into an instance of `Ferry Tuple Type` $\widehat{ftt_R}$ and a bridging instance of `Ferry List Type` $\widehat{fl_{br}}$. Element type of $\widehat{fl_{br}}$ is set to $\widehat{ftt_R}$. $\widehat{fl_{br}}$ is added to the list of columns in $\widehat{ftt}$. Pair $(c_R.name \rightarrow \widehat{ftt_R})$ is added to map $Types$.

$$\frac{\forall c_R \ (c_R \in Classes \ \wedge \ c_R \in c.Refs \ \wedge \ c_R.name \notin Types.names\ )}{\begin{array}{c} a' \mapsto \ (\widehat{ftt_R},\ \widehat{fl_{br}}), \\ \widehat{fl_{br}}.type = \widehat{ftt_R},\ \widehat{ftt}.cols \mathrel{+}= \ \widehat{fl_{br}},\ Types \mathrel{+}= (c_R.name, \widehat{ftt_R}) \end{array}}$$

5. Each class of reference $c'_R$, if there already exists an element in $Types$ whose key equals to $c'_R.name$, is translated into a bridging instance of `Ferry List Type` $\widehat{fl'_{br}}$. Given that the element `Ferry Tuple Type` with key $c'_R.name$ in $Types$ is denoted as $\widehat{ftt'_R}$, the element type of $\widehat{fl'_{br}}$ is set to $\widehat{ftt'_R}$. $\widehat{fl'_{br}}$ is added to the list of columns in $\widehat{ftt}$.

$$\frac{\forall c_R\ (c_R \in Classes\ \wedge\ c_R \in c.Refs\ \wedge\ c_R.name \in Types.names\ ),\quad \widehat{ftt'_R} = Types(c_R.name)}{a' \mapsto\ (\widehat{fl'_{br}}),\quad \widehat{fl'_{br}}.type = \widehat{ftt'_R},\ \widehat{ftt}.cols\ +=\ \widehat{fl'_{br}}}$$

6. If $c$ implements any interface, an instance of `Ferry List Type` $\widehat{fl_i}$ and an instance of `Ferry Tuple Type` $\widehat{ft_i}$ are created. `Ferry Atomic Type` $String$ and $Int$ are inserted as element types of $\widehat{ft_i}$, with names "interface" and "obj_id", respectively. Element type of $\widehat{fl_i}$ is set to $\widehat{ft_i}$. $\widehat{fl_i}$ is added to columns of $\widehat{ftt}$.

$$\frac{\exists i\ (\ c \xrightarrow{\text{imp}}\ i\ )}{i \mapsto\ (\ \widehat{fl_i}\ ,\ \widehat{ft_i}\ )\quad \widehat{ft_i}.cols = (\text{“interface”},\text{“obj\_id”}),\ \widehat{fl_i}.type = \widehat{ft_i},\ \widehat{ftt}.cols+ = \widehat{fl_i}}$$

Once the reified Ferry Types have been obtained, trees of Table Information Nodes can be constructed from them. For this, following rules and algorithm are applied:

Given the following notations:

- $InfoNodes(name_1 \rightarrow node_1, \cdots, name_n \rightarrow node_n)$: a map holding the pairs of class names and the resulting Table Information Nodes.
- $InfoNodes.names$: a list holding all the keys in map $InfoNodes$.
- $\texttt{Rule}_n(node)$: Apply $\texttt{Rule}_n$ to node $node$.
- $InfoNodes(key)$: retrieves the value in the map $InfoNodes$ according to key $key$.

7. Each instance of Ferry Tuple Type $\widehat{ftt}$ in $Types$, if has not been translated, is translated into one instance of `Table Information Node` $n$. $n$ is added in $InfoNodes$.

$$\forall \widehat{ftt}\ (\widehat{ftt} \in Types \ \ \wedge\ \widehat{ftt}.names \notin InfoNodes.names\ )$$

$$\widehat{ftt} \mapsto n,$$
$$InfoNodes\ +\!= n$$

8. Names of columns in $n$ are taken from those in $\widehat{ftt}$. If these optional names are absent, default names "$item_1$","$item_2$",$\cdots$ ,"item$_n$" will be assigned.

$$\widehat{ftt},\ n$$

IF $\widehat{ftt}.names$ EXISTS ( $n.name_1 = \widehat{ftt},\ \cdots,\ n.name_n = \widehat{ftt}.name_n$ )

ELSE ( $n.name_1$ ="$item_1$", $\cdots$ , n.name$_n$ ="$item_n$" )

9. For each column $c$ in $\widehat{ftt}$ whose type is Ferry Atomic Type $\widehat{fa}$, type of corresponding column in $n$ is set to $\widehat{fa}$.

$$\forall c(c.type = \texttt{Ferry Atomic Type}),\ \widehat{fa} = c.type,\ n,$$
$$\{\ i \mid n.cols[i].name = c.name\ \}$$

$$n.cols[i].type = \widehat{fa}$$

10. For each column $l$ in $\widehat{ftt}$ whose type is Ferry List Type $\widehat{flt}$, a new `Table Information Node` $n'$ is created. `Key`$(name = l.name, pointTo = n')$ is added to $n.keys$. Type of corresponding column(surrogate value) in $n$ is set to `Ferry Atomic Type`.

$$\forall l(l.type = \texttt{Ferry List Type}),\ n,\ \{\ i \mid n.cols[i].name = l.name\ \}$$

$$l \mapsto n', key = (l.name, n'), n.keys+\!= key,$$
$$n.cols[i].type = \texttt{Ferry Atomic Type}$$

11. For each $l$ whose elements are of `Ferry Atomic Type` $\widehat{la}$, $n'$ should contain only one column which is set to $\widehat{la}$.

$$\{\ l \mid l.type = \texttt{Ferry Atomic Type}\}, \widehat{la} = l.type,\ n'$$

$$n'.cols = (\widehat{la})$$

12. For $l$ whose elements are of `Ferry Tuple Type` $\widehat{lt}$, if $\widehat{lt}$ has not been translated, apply `Rule 7` to $\widehat{lt}$ and replace $n'$ with this result.

$$\frac{\{\ l \mid \widehat{lt} = l.type,\ (\ l.type = \texttt{Ferry Tuple Type} \ \wedge\ \widehat{lt}.name \notin InfoNodes.names\ )\},\ n'}{n' = \texttt{Rule7}(\widehat{lt})}$$

13. For $l$ whose elements are of `Ferry Tuple Type` $\widehat{lt'}$, if $\widehat{lt'}$ has already been translated into Table Information Node $n''$, replace $n'$ with $n''$.

$$\frac{\{\ l \mid \widehat{lt} = l.type,\ (\ l.type = \texttt{Ferry Tuple Type} \ \wedge\ \widehat{lt}.name \in InfoNodes.names\ )\},\ n',\ n'' = InfoNodes(\widehat{lt}.name)}{n' = n''}$$

14. For $l$ whose elements are of `Ferry List Type` $\widehat{ll}$,a new `Table Information Node` $n'''$ is created. `Key(name="`$item_1$`", pointTo=`$n'''$`)` is added to $n'.keys$. Type of corresponding column in $n'$ is set to `Ferry Atomic Type`. Apply `Rule 11`, `12`, `13` or `14` according to type of elements of $\widehat{ll}$, and replace $n'''$ with this result.

$$\frac{\forall l(l.type = \texttt{Ferry List Type}),\ \widehat{ll} = l.type,\ n',\ \{\ i \mid n'.cols[i].name = \widehat{ll}.name\ \ \}}{l \mapsto n''', key = (\text{“}item_1\text{”},\ n'''), n'.keys+ = key,\ n'.cols[i].type = \texttt{Ferry Atomic Type},\ n''' = (\texttt{Rule11} \mid \texttt{Rule12} \mid \texttt{Rule13} \mid \texttt{Rule14})(\widehat{ll})}$$

To sum up, `Rule 1` is the entry point for each class. According to different types of its fields, `Rule 2` to `Rule 5` are applied accordingly. `Rule 2` and `Rule 3` handle single and multiple attributes respectively, while `Rule 4` and `Rule 5` handle references. `Rule 4` handles the case where the type of the reference target has not yet been processed. Otherwise, `Rule 5` takes effect to ensure that one class is only mapped to one instance of `Ferry Tuple Type`. This issue will be covered again under Point 2 in Section 2.3.2.

The fact that table cells can only contain atomic value lead to the necessity of applying the *loop lifting* algorithm[6] for nested structures, namely in `Rule 3`, `Rule 4` and `Rule 5`.

After loop-lifting, bridging tables are created, whose `iter` column specifies the round of iteration, and the `pos` column specifies the current position within that round. Intuitively, a multiple attribute in an object is encoded as a list, the elements of which are stored in that bridging table, each having its own value of column `pos` but sharing the same value of column `iter`. This value of `iter` is also stored in the node of the containing class as a surrogate value, showing that all rows in the bridging tables that share this particular value belong to that containing object. In the case of reference, only object IDs are stored in the bridging table. Similarly, value `iter` groups up all these IDs that belong to the containing object, and this value is also stored as a surrogate value. That means this reference can be either single or multiple, in the latter case each ID having its own value of `pos`. Because of this, single and multiple references result in the same tree topology of Table Information Node(only one bridging table between referring and referred classes).

`Rule 6` creates a node of `Ferry Tuple Type` to hold the list of interfaces a class implements. Unlike single inheritance, one class may implement multiple interfaces. It suffices to have one element in a tuple to specify its super-class in a surrogate value, an additional node is however necessary in order to keep the surrogate values for all the implemented interfaces.

With the one-class-per-table scheme, each table only stores fields declared in this class. This also applies to the case of diamond-shaped class hierarchy, as shown in Figure 2.7.

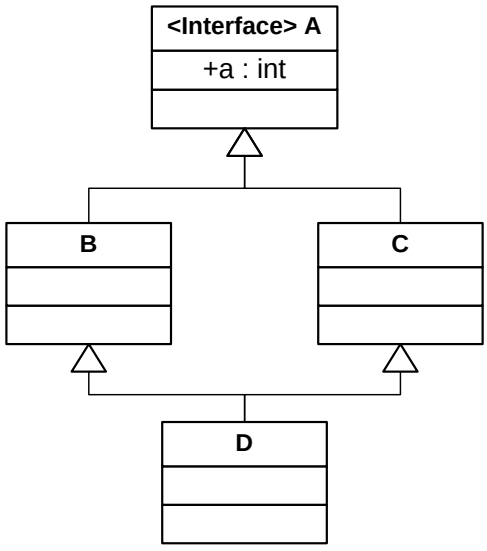



Figure 2.7: Diamond-shaped class hierarchy

In Figure 2.7, interface `A` declares field `a`. At least one of types `B` and `C` is also interface because of single inheritence of class `D`. In the persistence

result, field `a` only presents in the table for `A`, although class `D` can inherit it from both paths.

The last step of persistence of EMF class schema is to create the empty tables in database according to the resulting Table Information Nodes. This step is straightforward, in that each instance of Table Information Node is mapped to one table in database, and the columns are also mapped accordingly regarding to their names and types. Listing 2.2 shows the code snippet that create the empty tables in database.

```
1  def createTable(name:String, node:TableInfoNode){
2    conn.createStatement.execute("DROP TABLE IF EXISTS " + name + "
         ;\n")
3    var sql = "create table " + name + "( "
4    sql = sql + (
5      (
6        for {
7        col <- node.cols
8        } yield (COLUMN_NAME_PREFIX + col._1 + " " + getDBDataType(col
             ._2) + {if (col._1=="id") " UNSIGNED NOT NULL
             AUTO_INCREMENT " else ""})
9      ) mkString(",")
10   )
11   sql = sql + ", PRIMARY KEY (" + COLUMN_NAME_PREFIX + "id) );"
12   conn.createStatement.execute(sql)
13 }
```

Listing 2.2: Snippet that creates the empty tables in database

### 2.3.2 Implementation and result

According to the rules and algorithms described above, a visitor-like pattern is followed in the implementation process due to the tree-shaped topology of both EMF class schema and trees of Table Information Nodes.

Furthermore, some details regarding implementation are explained here:

1. In Step *1*, besides the columns defined in each class, additional auxiliary columns are added to each resulting nodes of Ferry Tuple Type. These columns include “object-id”, “superclass-id”, “superobj-id”, “subclass-id”, “subobj-id”. These columns are necessary to keep track of different parts of individual objects located in different table nodes. These columns are introduced in Step *1* and are propagated through Step *2* and *3* to become part of the table schema in database. Also in Listing 2.2 one can see that column `_wen_id` is assigned as the primary key of the created table.

2. Objects of one class should be mapped to only one Table Node if that had existed, ensuring that objects of this type could be located

from one uniform place, regardless whether they had been created isolated or through association with other objects. Despite only the conceptual existence of Table Node, this uniqueness should also be maintained in its meta data: the corresponding Table Information Node. In terms of the Rules described above, there should exist only one Table Information Node corresponding to one class, regardless of whether it is created from Ferry Tuple Types created by Rule *1* or *4*. The replacement operations in Rule *12*, *13* and *14* result in the single instance of Table Information Node in the map.

Practically, the implementation of this unification acts a bit different. All Table Information Nodes being referred to are collected in a map first. After all classes have been translated, each node within that map is checked if they are within the result map. If not, the `pointTo` field of that reference key is updated to point to its counterpart within the result map. The one outside the result map is therefore discarded. The snippet in Listing 2.3 illustrates this process.

```
1  refTypeMap.keys.foreach(
2    key => {
3      val from = infoNodes(key._1)
4      val to = infoNodes(refTypeMap(key))
5      var found = from.keys.find(_.name == key._2).get
6      found.pointTo.keys(0).pointTo = to
7    }
8  )
```

Listing 2.3: Snippet that performs the unification of reference class

3. In the creation of empty tables in database, a prefix "_wen_" is added to each column names in order to avoid any potential conflict with any SQL keywords or predefined SQL functions.

In Figure 2.8 on page 19, UML diagram of a simple EMF class schema is given, which will be translated in the resulting trees of Table Information Nodes shown in Figure 2.9 on page 19.

In Figure 2.8, concepts of inheritance(class `Employee` inherits from class `Person`) and interface implementation (class `Employee` implements interface `SpeaksEnglish`) are demonstrated. Furthermore, reference (class `Address` by class `Employee`), multiple attribute(`email`) and single attribute(the other fields) are also included. From the translation result in Figure 2.9, one can see how these features are represented in the Table Information Nodes. Columns in blue are also included in the keys, which points to bridging tables resulted from the loop lifting algorithm. Note that in Figure 2.9, node for class `Person` is not yet connected with node for class `Employee`. Rather, this connection will be done in the persistence of object population,

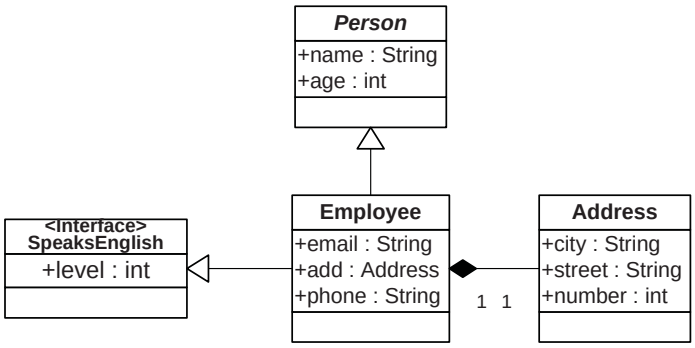



Figure 2.8: UML diagram of a simple EMF class schema

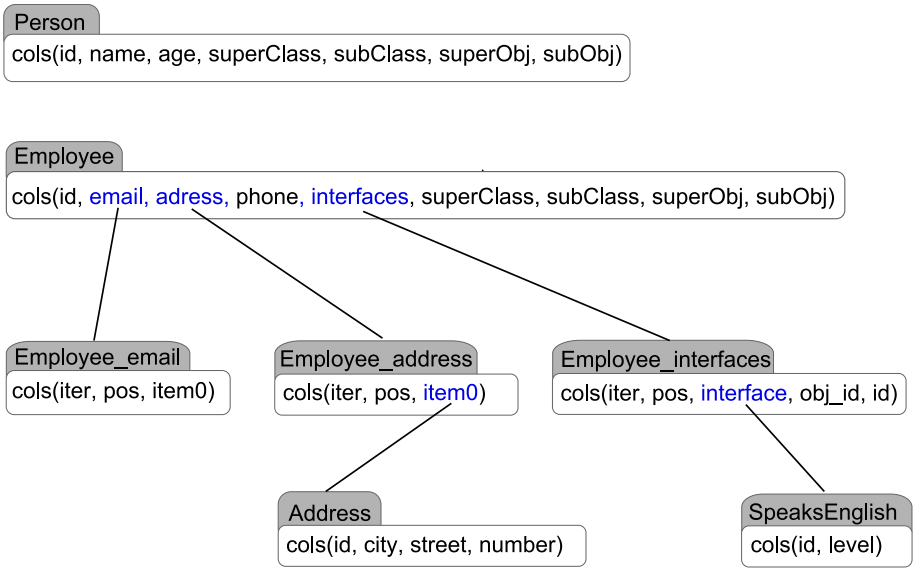



Figure 2.9: Table Information Nodes of a simple EMF class schema

in that column `superClass` and `subClass` will be filled accordingly to tie up different parts of an object.

At this point, the translation from EMF class schema to trees of Information Nodes is completed.

## 2.4 Persistence of object population

### 2.4.1 Algorithm

The persistence of EMF object population involves Step *4* in Figure 2.5. After the empty tables in database have been created by Step *3*, data of these objects is inserted into these tables accordingly, in such a way that different parts of data that belong to different class frames are stored in

the corresponding tables, and surrogate values are generated and inserted properly so that to tie up these parts across multiple tables.

In Listing 2.4, the algorithm for this step is given as Scala-styled pseudocode, with the comments in blue as explanation.

```
1  // persists an object in class-frame ofType, returning the ID of o
       in the table of ofType.
2  def persist(o:Object, ofType:String):Int = {
3  //if o in class-frame ofType has already been persisted
4    if (population contains (o, ofType)
5       return population((o, ofType))
6    var node = getTableInfoNode(ofType)
7    var sql
8    node.cols.foreach( col => {
9  // for columns of surrogate values, first insert null, and update
       later(line 31 and 39) with surrogate values
10   if (col._1==``superClass_id'') || (col._1==``superObj_Id'')||
11      (col._1==``suubClass_id'')||(col._1==``subObj_id'')||
12      (cols._1==``interfaces'')
13      sql += ``null''
14 // for columns of references, first insert null, and update later(
       line 27) with surrogate value of bridging tables
15   if (col is reference)
16      sql += ``null''
17 // for columns of attributes, call method persistAttr(line 58) and
       add the return value to sql
18   if (col is attribute)
19      sql += persistAttr(o, ofType, col)
20         }
21   )
22 //execute the insertion of current object
23   var where = executeInsert(sql)
24 //keep that o in class frame ofType has been persisted, with ID
       number where in its corresponding table.
25   population += ((o, fixedType) -> where)
26 //call method persistRefs(line 78) to persist all references of o in
       class frame ofType.
27   persistRefs(trans, o, ofType, where)
28 //if class ofType has super type, call persistSuper(line 46)
29   var superType = getSuperClassFromName(ofType)
30   if (superType != None)
31     persistSuper(o, superType, ofType, where)
32 //call persist(line 2) on each interface that class ofType
       implements to get result interID(line 35), insert interID into
       bridging table(line 37) to get bridgingID, and update the
       surrogate value in table ofType with bridgingID(line 39).
33   interfaces.foreach(
34       inter => {
35         var interID = persist(o, inter)
```

```
36          var sql = "..." //SQL statements inserting interID into
                bridging table
37          var bridgingID = executeInsert(sql)
38          sql = "..." //SQL statements updating table of ofType with
                value bridgingID
39          executeUpdate(sql)
40        }
41      )
42  //return the ID of current object in table ofType
43    where
44  }
45  //persist object o within class-frame superType, returning the ID of
        o in the table of superType.
46    def persistSuper(o:Object, superType:String, subType:String, subID:
          Int):Int = {
47  // call persist(line 2) on o with class-frame superType
48      var superID = persist(o, superType, trans)
49  //update surrogate values in table of superType
50      var sql = "UPDATE...id=" + superID;
51      executeUpdate(sql)
52  //update surrogate values in table of subclass
53      sql = "UPDATE...id=" + subID;
54      executeUpdate(sql)
55      superID
56    }
57  //persist columns of object o within class-frame superType,
        returning a String representing either the iter value in
        bridging table(for multiple attributes) or the primitive value(
        for single attributes).
58    def persistAttrs(o:Object, ofType:String, col:String):String = {
59  //if current column is also in key, it is a multiple attribute
60      if (infoNodes.contains(ofType)  infoNodes(ofType).keys.exists(key
            => key.name==col)) {
61  // get multiple attribute as a list
62        var list = ...
63  //get the next available value of iter, by adding 1 to the maximal
        existing value
64        var iter = getMax(col, ofType) + 1
65        for (i <- 1.to(list.size)) {
66  //SQL statement to insert each value in the list, having iter as the
        common value of column iter, and each i as each value of column
        pos
67          var sql = "...iter, i, ..."
68          executeInsert(sql)
69        }
70  //return value of iter, so that to update the surrogate value in the
        containing table
71        iter.toString
72      }
```

```
73 //single attribute
74     else
75 //get primitive value of column col
76       o.getClass.getMethod(getMethodName(col)).invoke(o).toString
77   }
78 //persist all references of object o within class-frame ofType
79   def persistRefs(o:Object, ofType:String, where:Int) = {
80     for each reference (
81 //get target object of reference
82           var target = ...
83 //when target is a multiple reference
84           if (target.isInstanceOf[EList[Object]]){
85             for each object in target {
86               var obj = target.asInstanceOf[EList[Object]].get(i)
87 //call persist(line 2) on each object obj, getting its ID in
       corresponding table
88               var newID = persist(obj, getEMFTypeName(obj))
89 //SQL statement to insert newID into bridging table
90               var sql = "INSERT ... newID ...";
91               executeInsert(sql)
92             }
93           }
94 //when target is single reference, or target is null
95           else {
96             if (target == null)
97               newID = -1
98             else
99 //call persist(line 2) on object target, getting its ID in
       corresponding table
100               newID = persist(target, getEMFTypeName(target))
101 //SQL statement to insert newID into bridging table
102             var sql = "INSERT ... newID ... "
103             executeInsert(sql)
104           }
105 //update referring table
106           var sql = "UPDATE ... where ... "
107           executeUpdate(sql)
108         }
109     )
110   }
111 //execute an SQL insertion, returning the ID of rows that has just
       been inserted
112   def executeInsert(sql:String) :Int = { ... }
113 //execute an SQL update
114 def executeUpdate(sql:String) = { ... }
```

Listing 2.4: Pseudocode that perform the persistence of object population

To sum up, this algorithm iterates through all objects. By visiting each

object, it always limits itself in one particular class-frame at a time, storing data of columns and references within this class-frame into the database. Then fields belonging to interfaces that this class-frame *directly* implements are also persisted. At this time, object of this class-frame has been completely persisted. If it has superclass, the same procedure will be repeated in the class-frame of its superclass, until it reaches the top class of the entire hierarchy.

### 2.4.2 Implementation and result

According to the algorithm described above, a visitor-like pattern is followed to implement the persistence of object population. Furthermore, some details regarding implementation are explained here:

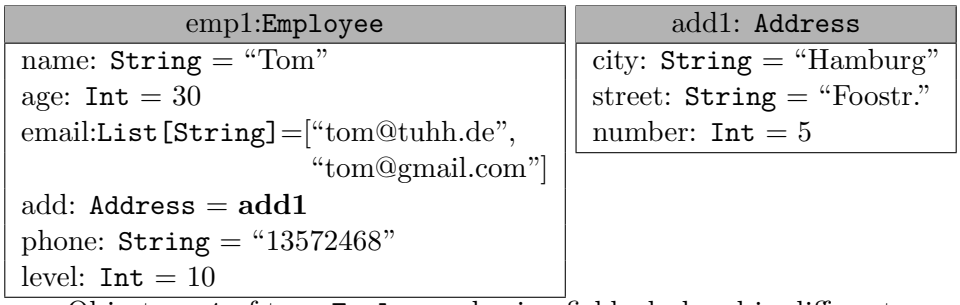
1. As described above, surrogate values are essential in tying up different parts of objects as well as keeping track of references, interfaces and so on. In our work, objects are stored in database as rows, therefore IDs of rows can be used as identifiers of objects, that is, as the surrogate values. After each insertion operation, a query operation is performed immediately to retrieve the ID of the row that has just been inserted. It is necessary to ensure that always the correct ID representing that particular object is retrieved.

2. Although one object can be persisted in different tables in database, the entire process should be atomic. That is, it is not allowed to have some parts of object persisted successfully while others are failed due to for example runtime exception or database access error, which would otherwise lead to inconsistency within different tables.

3. Summing up point 1 and point 2, the entire persistence process of object population is included in a database transaction.

4. By default objects in EMF model have "private" modifier for fields, and hence don't support the member access syntax of `obj.field`. Rather, all these classes implement setters and getters for each field, through which to access them. For example, to access single reference `address` in object `o` of class `Person`, one should call method `o.getAddress()` instead of `o.address`. In the persistence process, we used Java reflection mechanism to invoke these getter methods.

Figure 2.10 gives a sample object `emp1`, which is instance of class `Employee` illustrated in Figure 2.8 on page 19. After persisted, this object is stored in database, whose layout is depicted in Figure 2.11. Dashed arrows in Figure 2.11 show how surrogate values point to rows in other tables to tie up objects.




Object `emp1` of type `Employee`, having fields declared in different class-frames. field `add` refers to object `add1` of type `Address`

Figure 2.10: UML object diagram of a sample object

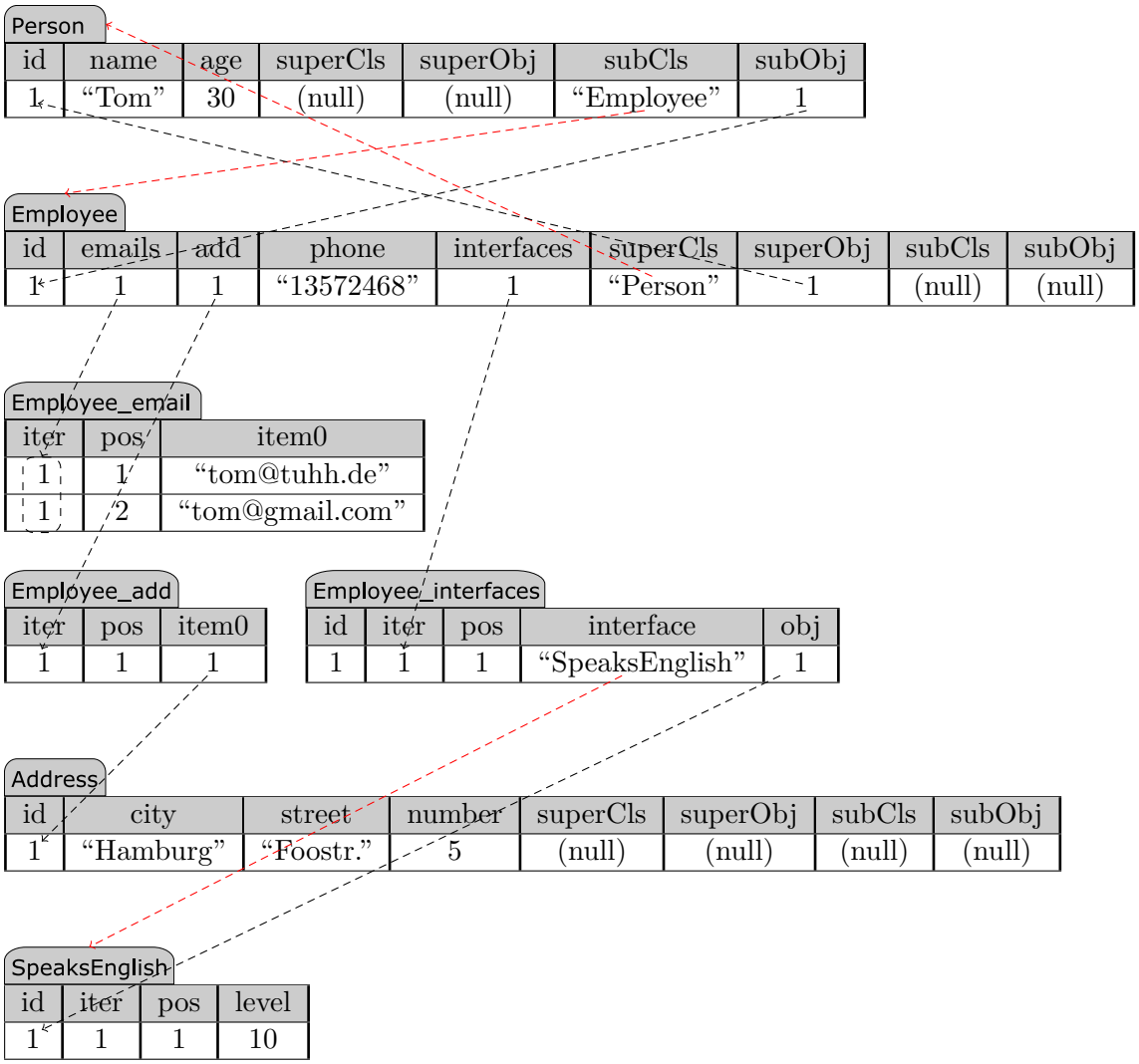



Figure 2.11: Database tables after persistence

# Chapter 3

# Query: LINQ to Ferry

In the previous chapters, we set out to persist EMF data model into database with a Ferry-aware result. Being Ferry-aware, the persistence result can be queried with Ferry and hence the original object in EMF model can be retrieved back.

Nevertheless, Ferry is a lower-level query language than LINQ on object-oriented level. To also enable its query facility on object model, we need to perform a translation from an object-oriented query language into Ferry. This chapter reports this translation process from LINQ to Ferry.

In Section 3.1, overview of the translation will be given. It covers the syntax of source and target languages: LINQ and Ferry, which subset of LINQ will be supported in this project, as well as the different translation phases that are involved. In Section 3.2, the normalization step will be explained in detail, which is a prerequisite step to the following translation. Section 3.3 explains in detail the translation rules for all supported LINQ SQOs, followed by the implementation aspect and the result. Finally, further step following the arrival at Ferry target is discussed briefly in Section 3.5.

## 3.1 Overview of the translation

### 3.1.1 Syntax of LINQ

In this chapter, the term "LINQ" refers only to the LINQ query language and ignores other technologies such as *LINQ Query Provider* and *Entity Framework*.

*Query Expressions*(or *textual LINQ query*) was first introduced since Microsoft .Net Framework 3.5, which provides a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery. Query Expressions is one of the two ways to compose a LINQ query (The other way is using *Standard Query Operators (SQO)[7]*, which will be introduced later).

Each query expression begins with a *from* clause and ends with either a *select* or *group* clause. The initial *from* clause can be followed by zero or more *from*, *let*, *where*, *join* or *orderby* clauses, which are altogether called *query body* clauses.

A *from* clause defines the data source of a query or sub-query and a range variable that defines each single element to query from that data source. In terms of list comprehension, the source sequence is the input set, and the range variable represents each member of that set. The syntax of the *form* clause is:

```
from range-variable in data-source
```

Each *let* clause introduces a range variable representing a value computed by means of previous range variables or a result list of sub-query. Once defined, the new variable is always in scope in the rest of the query. As expected, the syntax of the *let* clause is:

```
let name=expression
```

A *where* clause is a function that filters items on the input list, keeping only items that satisfies the predicate. A single query can have multiple *where* clauses or a *where* clause with multiple predicates that are combined by logical operators. Here is the syntax of the *where* clause:

```
where boolean-expression
```

Each *join* clause compares specified keys of the inner-sequence with keys of the outer-sequence, yielding matching pairs. The predicates *outerKeySelector* and *innerKeySelector* define how to extract the identifying keys from both sequence items. Here is the syntax of the *join* clause:

```
join range-var in inner
on outerKeySelector equals innerKeySelector
```

Each *orderby* clause reorders items by using one or more keys that combines different sorting directions and comparator functions. When not specified, *Ascending* will be the default direction. The syntax of the *orderby* clause is:

```
orderby sort-on direction
```

The ending *select* or *group* clause specifies the shape of the result in terms of the range variables in scope. A *select* clause specifies what the query outputs, based on a projection that determines what to select from the result of the evaluation of all the clauses and expressions that precede it. A *group* clause projects a result grouped by a key, providing an effective way to retrieve data that is organized into sequences of related items. Following are the syntax of the *select* clause and *group* clause:

```
select expression
```

| | | |
|---|---|---|
| $Q \in$ QueryExp | ::= | $F_{from}$ $QB_{qbody}$ |
| $F \in$ FromClause | ::= | `from` $T_{type}^{0..1}$ $V_{var}$ `in` $E_{in}$ |
| $QB \in$ QueryBody | ::= | $B_{qbclauses}^{0..*}$ $SG_{sel\_gby}$ $QC_{qcont}^{0..1}$ |
| $B \in$ BodyClause | = | (FromClause ∪ LetClause ∪ WhereClause ∪ JoinClause ∪ JoinIntoClause ∪ OrderByClause) |
| $QC \in$ QueryCont | ::= | `into` $V_{var}$ $QB_{qbody}$ |
| $H \in$ LetClause | ::= | `let` $V_{lhs}$ `=` $E_{rhs}$ |
| $W \in$ WhereClause | ::= | `where` $E_{booltest}$ |
| $J \in$ JoinClause | ::= | `join` $T_{type}^{0..1}$ $V_{innervar}$ `in` $E_{innerexp}$ `on` $E_{lhs}$ `equals` $E_{rhs}$ |
| $K \in$ JoinIntoClause | ::= | $J_{jc}$ `into` $V_{result}$ |
| $O \in$ OrderByClause | ::= | `orderby` $U_{orderings}^{1..* <separator:,>}$ |
| $U \in$ Ordering | ::= | $E_{ord}$ $Direction_{dir}$ |
| *Direction* | ∈ | { `ascending`, `descending` } |
| $S \in$ SelectClause | ::= | `select` $E_{selexp}$ |
| $G \in$ GroupByClause | ::= | `group` $E_{e1}$ `by` $E_{e2}$ |

Figure 3.1: LINQ-related production rules

`group` *`range-variable`* `by` *`key`*

Finally, an optional *into* clause can be used to connect queries by treating the results of one query as a generator in a subsequent query. This is called a *query continuation*. Here is the syntax of the into clause:
`into` *`name query-body`*

The detailed LINQ grammar is captured in Figure 3.1, with *QueryExp* being the entry rule. The notation conventions in the grammar follow Turbak and Gifford[8].

The other alternative to compose query besides textual query is using *Standard Query Operators (SQO)*, defined as extension methods[9]. In terms of .Net Framework, these methods can work with any object that implements either the *IEnumerable<T>* or *IQueryable<T>* interface. In fact, being the syntactic sugar of SQOs, the textual LINQ syntax has been designed for good readability but not for direct evaluation[10]. Rather, textual LINQ query is internally translated to SQOs. For example, the query
`from p in Persons where p.age>20 select p.name`

is translated into its SQO equivalence:
`Person.Where(p=>p.age>20).Select(p=>p.name).`

LINQ accepts queries in both kinds of syntax. Despite that all queries in textual syntax can be translated into SQOs successfully and deterministically[2][11], one should however note that both of them have different expressive power. In particular, many SQOs have various method overloadings, some of which can only be called directly but not translated from textual syntax. Section 3.2 will examine this issue again.

As described in [2], some translation rules from textual LINQ to SQOs produce so-called *transparent identifiers* that have not occurred in the original textual query. Such kind of identifiers represents items in sequence of so-called *anonymous types*, which are generated by the preceding clause. For example, a textual query in Line 1 in Listing 3.1 is translated into SQOs in Line 3 to 5.

```
1 from e in Emp from t in e.task where t.level>10 select e.name
2
3 Emp.SelectMany(e => e.task, (e, t) => {e = e, t = t}).
4 Where( TransID_0 => TransID_0.t.level >10 ).
5 Select( TransID_0 => TransID_0.e.name)
```

Listing 3.1: Example of translating transparent identifier

Method `SelectMany` in Line 3 produces a sequence of anonymous type `new {e = e, t = t}` for the following methods (`Where` and `Select`) to consume. To represent items of this type, transparent identifier `TransID_0` is generated, allowing further references to items of that type, as well as fields declared in it(`e` and `t` in this example).

### 3.1.2 Syntax of Ferry

The syntax of Ferry is given in Figure 3.2(reproduced from page 6 in [6]).

Notice that the grammar in Figure 3.2 is the surface-syntax of Ferry. In Tom Schreiber's implementation, programs in Ferry are internally desugared to Ferry Core, which will be then type-checked and translated to relational algebra[6]. In our implementation, we take Ferry as our translation target from LINQ, taking the advantage of similar semantics of language constructs in both LINQ and Ferry.

In the following, syntax and semantic of each Ferry expression will be explained, along with its typing rule. For that, the following notation is defined:[6]

$$\Gamma \vdash e :: t$$

which means that expression $e$ is of type $t$ in the type environment $\Gamma$, where $e$ represents a Ferry Expression defined in Figure 3.2, and t represents one of the Ferry types defined in Figure 2.1. Additionally, $v$ represents a variable.

Ferry Expressions

| | | | |
|---|---|---|---|
| $e$ | := | $l \mid v$ | Literal, variable |
| | \| | $(e, e, \ldots, e) \mid e.c \mid e.n$ | Tuple, Tuple access |
| | \| | `[ ]` $\mid$ `[`$e, \ldots, e$`]` | (Empty) lists |
| | \| | `table` $R(c\ a, \ldots, c\ a)$ | Table access |
| | | `with keys` $(k, \ldots, k)$ | Key specification |
| | | [`with order` $(c, \ldots, c)$] | Optional order specification |
| | \| | `let` $v = e, \ldots, v = e$ `in` $e$ | Variable bindings |
| | \| | `if` $e$ `then` $e$ `else` $e$ | Conditional evaluation |
| | \| | `for` $v$ `in` $e, \ldots, v$ `in` $e$ | List comprehension |
| | | [`where` $e$ ] | Optional predicate |
| | | [`group by` $e, \ldots, e$ [`where` $e$] ] | Optional grouping |
| | | [`order by` $e, \ldots, e$ ] | Optional ordering |
| | | `return` $e$ | Head of the list comprehension |
| | \| | $e * e \mid f(e, \ldots, e) \mid f(v \rightarrow e, e)$ | Application |

Meta Variables

| | | | |
|---|---|---|---|
| $e$ | Ferry Expression | $l$ | Literal (of atomic type $a$) |
| $v$ | Variable name | $n$ | Tuple position ($n \in \{1, 2, \ldots\}$) |
| $c$ | Column name | $R$ | Table name |
| $a$ | Atomic basic type | $k$ | Keys ($(c, \ldots, c)$) |
| $f$ | Function name | $*$ | Binary operator ($* \in \{+, <, \texttt{and}, \ldots\}$) |

Figure 3.2: Syntax of Ferry

#### 3.1.2.1 Literal Expression ($l$)

Literal Expression is evaluated directly, whose result is its corresponding Ferry Atomic Type. Here's the typing rule of a Literal Expression.

$$\frac{l \in \mathit{IntLiteral} \quad l \Rightarrow}{\Gamma \vdash l :: \mathit{int}}$$

$$\frac{l \in \mathit{StringLiteral} \quad l \Rightarrow}{\Gamma \vdash l :: \mathit{string}}$$

$$\frac{l \in \mathit{BoolLiteral} \quad l \Rightarrow}{\Gamma \vdash l :: \mathit{bool}}$$

#### 3.1.2.2 Binary Operator ($*$)

Binary operator is evaluated directly with its two operands. Different Binary Operator requires operands of different types, yielding results of different types. Nevertheless, each binary operator requires its operands of the same type. Here's the typing rule of a Binary Expression.

$$\frac{\begin{array}{c}e_1 * e_2 \Rightarrow \\ * \in \{\texttt{ and, or }\} \quad \Gamma \vdash e_1 :: \mathit{bool} \quad \Gamma \vdash e_2 :: \mathit{bool}\end{array}}{\Gamma \vdash e_1 * e_2 :: \mathit{bool}}$$

$$\frac{\begin{array}{c}e_1 * e_2 \Rightarrow \\ * \in \{\texttt{ +, -, *, / }\} \quad \Gamma \vdash e_1 :: \mathit{int} \quad \Gamma \vdash e_2 :: \mathit{int}\end{array}}{\Gamma \vdash e_1 * e_2 :: \mathit{int}}$$

$$\frac{\begin{array}{c}e_1 * e_2 \Rightarrow \\ * \in \{\texttt{ ==, != }\} \quad \Gamma \vdash e_1 :: a \quad \Gamma \vdash e_2 :: a\end{array}}{\Gamma \vdash e_1 * e_2 :: \mathit{bool}}$$

$$\frac{\begin{array}{c}e_1 * e_2 \Rightarrow \\ * \in \{\texttt{ <, >, <=, >= }\} \quad \Gamma \vdash e_1 :: \mathit{int} \quad \Gamma \vdash e_2 :: \mathit{int}\end{array}}{\Gamma \vdash e_1 * e_2 :: \mathit{bool}}$$

#### 3.1.2.3 Tuple Expression ($(e, \ldots, e)$)

Tuple expression represents a record-like structure, having each element in the tuple evaluated individually. The type of the result is a tuple with types of each individual result(i.e. *boxed type*, which can be *list type* or *atomic type*). Here's the typing rule of a Tuple Expression.

$$\frac{\begin{array}{c}e = (e_1, ..., e_n) \Rightarrow \\ \forall_{i=1}^{n}\ \Gamma \vdash e_i :: b_i\end{array}}{\Gamma \vdash e :: (b_1, \ldots, b_n)}$$

#### 3.1.2.4 Positional access of Tuple ($e.c$)

An element of a tuple $e$ can be accessed through a one-based position $c$. Here's the typing rule of a positional access of Tuple.

$$\frac{\begin{array}{c}e = (e_1, ..., e_m), c \in \mathbb{N}, c \leq m \quad e \Rightarrow \\ \Gamma \vdash e :: (\ldots, e_c, \ldots)\end{array}}{\Gamma \vdash e.c :: (b_c)}$$

#### 3.1.2.5 Nominal access of Tuple ($e.n$)

An element of a tuple $e$ can be accessed through the column name $c$ of that element. Here's the typing rule of a nominal access of Tuple.

$$\frac{e = (e_1, ..., e_m), c \in \mathbb{N}, c \leq m, n \mapsto c \quad e \Rightarrow \quad \Gamma \vdash e :: (\ldots, e_c, \ldots), n \mapsto c}{\Gamma \vdash e.n :: (b_c)}$$

#### 3.1.2.6 List Expression ($[e, \ldots, e]$)

A list expression produces a list, whose elements are of Ferry type $t$. Elements of a ferry List type is homogeneous, meaning all elements of a list should have the same concrete type. Here's the typing rule of a List Expression.

$$\frac{e = [e_1, ..., e_n] \;\Rightarrow \quad \forall_{i=1}^{n}\ \Gamma \vdash e_i :: t}{\Gamma \vdash e :: [t]}$$

#### 3.1.2.7 If Expression ($e =$ `if` $e_1$ `then` $e_2$ `else` $e_3$)

If Expression in Ferry is nothing else than its counterpart in other programming languages. Expression $e_1$ must be of type *bool*, and $e_2$ $e_3$ can be any Ferry type. Note that $e_2$ and $e_3$ must be of he same type, which will also be the type of $e$. Here's the typing rule of a If Expression.

$$\frac{e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \quad \Gamma \vdash e_1 :: \mathit{bool},\ \ \Gamma \vdash e_2 :: t,\ \ \Gamma \vdash e_3 :: t}{\Gamma \vdash e :: [t]}$$

#### 3.1.2.8 Let Expression ($e =$ `let` $e_1 = v_1, \ldots, e_m = v_m$ `in` $e_n$)

In a Let expression, each expression $e_i$ in the binding clauses will be first evaluated, and its result is bonded to the corresponding variable $v_i$. Afterwards, $e_m$ is evaluated, whose result is taken as the result of $e$. Here's the typing rule of a Let Expression.

$$\frac{e = \texttt{let}\ e_1 = v_1, \ldots, e_m = v_m\ \texttt{in}\ e_n \;\Rightarrow \quad \forall_{i=1}^{m}\ \Gamma \vdash e_i :: t_i,\ \forall_{i=1}^{m}(\Gamma + [v_i \mapsto t_i] \vdash :: t_n)}{\Gamma \vdash e :: [t_n]}$$

#### 3.1.2.9 Table reference expression

A table reference expression with schema $(c_1a_1, \ldots, c_na_n)$ will be evaluated as a list $[(a_1, \ldots, a_n)]$. The mandatory key specification is important for the optimization phase([6],p.108, Optimization will not be discussed in this thesis.) The optional ordering specification gives the key, on which the list taken from database table should be sorted. Prerequisite for this expression is that column specification $[c_1, \ldots, c_n]$ are distinct from each other. This is checked by Ferry internally through a built-in macro `distinct`. Further, all items in key specification and order specification should be within $[c_1, \ldots, c_n]$. Here's the typing rule of a Table Reference Expression.

$$\frac{\begin{array}{c} e = \texttt{table}\ (c_1\ a_1, \ldots, c_n\ a_n) \\ \quad \texttt{with keys}\ (k_1, \ldots, k_n) \\ \quad \texttt{with order}\ (o_1, \ldots, o_n) \Rightarrow \\ \texttt{distinct}([c_1, \ldots, c_n]), \\ [k_1, \ldots, k_n] \subseteq [c_1, \ldots, c_n], \\ [o_1, \ldots, o_n] \subseteq [c_1, \ldots, c_n] \end{array}}{\Gamma \vdash e :: [(a_1, \ldots, a_n)]}$$

#### 3.1.2.10 For expression

For expression is the central building block of Ferry with the following syntax.

$e =$ `for` $v_1$ `in` $e_1, \ldots, v_n$ `in` $e_n$
`[where` $e_w$`]`
`[group by` $e_{g1}, \ldots, e_{gm}$ `[where` $e_{gw1}, \ldots, e_{gwp}$ `] ]`
`[order by` $e_{o1}, \ldots, e_{oq}$ `]`
`[return` $e_r$`]`

Representing a list comprehension, $e_1$ to $e_n$ represent the input sequences, each of whose individual elements are represented as variable $v_1$ to $v_n$. $e_w$ acts as a predicate to filter out items that could go into the result. The result are clustered and sorted according to the optional `group by` clause and `order by` clause. Expression $e_r$ shapes the return sequence. Therefore the evaluated type of $e_r$ is also that of $e$. Besides this, we do not dive into the detailed typing rules of every other clauses. One should notice that in Ferry ordering can only be applied to atomic values[3], hence we do not support ordering on custom comparer. The same also applies to custom comparer for grouping. Further, since the optional embedded `where` clause within `group by` cannot be translated by any LINQ statement, we do not discuss that either. Here's the typing rule of a For Expression as a whole.

$$\frac{\begin{array}{c} e \Rightarrow \\ \forall_{i=1}^{n}(\Gamma \vdash e_i :: t_i), \\ \forall_{i=1}^{n}(\Gamma' = \Gamma + [v_i \mapsto t_i]), \end{array}}{}$$

$$\frac{\Gamma' \vdash e_r :: t_r, \Gamma' \vdash e_w :: t_w, \quad \forall_{i=1}^{m}(\Gamma' \vdash e_{gi} :: t_{gi}), \forall_{i=1}^{q}(\Gamma' \vdash e_{oi} :: t_{oi}),}{\Gamma \vdash e_r :: t_r}$$

### 3.1.3 Supported LINQ subset

A subset of LINQ is supported in our translation. As described in Section 3.1.1, queries can be composed in either textual syntax or SQO syntax, and the former is then translated into the latter. Therefore, when introducing the supported LINQ subset, we are talking about which SQO methods are supported.

Note that in C# Code Convention, method names are started with uppercase letters. Although this is not the convention for Java and Scala, we still adopt the C# one for the names of SQO methods in the following.

1. SQO methods that are translated from textual LINQ syntax are all supported. These methods include `Select`, `SelectMany`, `Where`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, `Join` and `GroupJoin`. Some of these methods have several overloadings, but not all of them are supported. Exact coverages of these overloadings will be explained in Section 3.2 and Section 3.3.

2. Some other extended operators for collection, including aggregation operators `Sum`, `Average`, `Min`, `Max` and `Count`, partitioning operator `Take` and casting operator `Cast`.

3. Single and multiple field access(for both attributes and references) across other class-frames as well. This only applies to the case of up-casting(to super-class or interfaces). Any downcasting might raise a wrong casting exception, and this cannot be checked until runtime. Therefore, in our case we are conservative about which queries are allowed, and thus reject queries involving down-castings. For up-castings, we support both implicit(by accessing fields declared in super class or interfaces directly) and explicit castings(by calling method `Cast`, mentioned in point *2* above).

4. By enabling point *1*, queries that may generate transparent identifiers are also supported.

On the other hand, some features that we do *not* support should be pointed out clearly here. A selected list of these includes:

1. For methods `Group`, `Join`, `GroupJoin` and `OrderBy`, `ThenBy`, `OrderByDescending`, `ThenByDescending`, we do not support custom comparers. In other words, comparison can only be performed on primitive types.

2. Any kind of down-casting, as mentioned in point *3* above.

3. We do not support method invocation on object within query. User-defined classes may declare methods that behaves differently, or returns unpredictable results under different circumstances that might corrupt the query. Actually, we do not support defining methods in the input EMF model(called "behaviour" in EMF) for the same reason.

### 3.1.4 Translation phases

The entire translation process involves several phases, as depicted in Figure 3.3. As shown, different translation phases are activated depending on

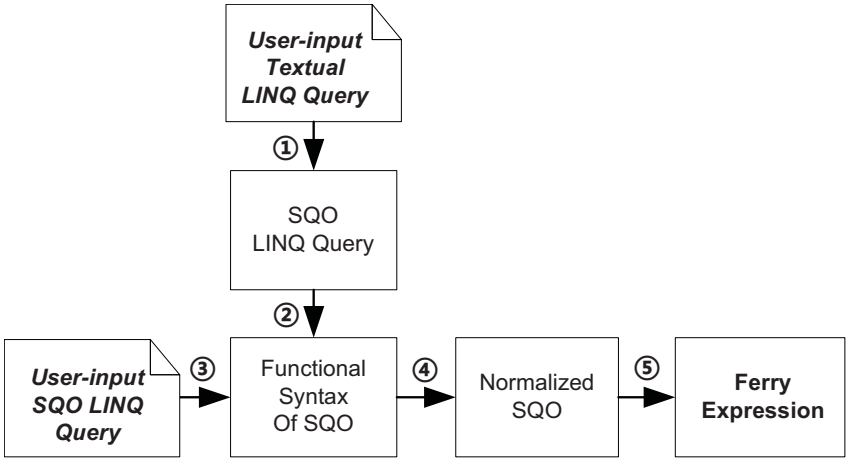



Figure 3.3: Phases of translation from LINQ to Ferry

what kind of query is provided by user. Steps *1*, *2* and *3* have been implemented in Project work *Translation of Java-Embedded Database Query, with a Prototype Implementation for LINQ* [2] and LinqExpand4Java [1]. Hence, functional syntax of SQO is the departure point of the rest of the translation. Step *4 Normalization* will be covered in Section 3.2, and Step *5 Translation* in Section 3.3.

[1] `http://www.sts.tu-harburg.de/people/mi.garcia/LINQExpand4Java/`

## 3.2 Normalization

### 3.2.1 Functional syntax of SQO

Functional syntax of SQO is merely another representation of sequence of calls to SQO methods. Briefly speaking, by converting to functional syntax, object on which a method is called(the whole "part" preceding the " . " of a method calling syntax) becomes the first argument of that method. For example, given the following query:
`from c in Customers where c.age>30 select c.name`
whose SQO counterpart will be:
`Customers.Where(c=>c.age>30).Select(c=>c.name)`
whose functional syntax expression then looks like:
`Select(Where(Customers, c=>c.age>30),c=>c.name)`
This translation simplifies the further steps in evaluating the SQO expression, because in this way all that need to be evaluated first will have been evaluated(this time including the input sequence of a method call, as it has become the first argument), by the time the current method is visited.

In terms of the CST metamodel implemented in Project[2], a chain of SQOs is a node of type `DotSeparated`, one of its child is another node of type `DotSeparated`, in case the length of the chain exceeds 2. The resulting SQO in functional syntax will be a node of type `MethodCall`, one of its arguments is also a node of `MethodCall` in the same case.

Besides, this step of translation also replaces any occurrence of method `ThenBy` and `ThenByDescending` by a new method named `OrderByMultiKey`, which provides a more convenient way to evaluate oderings with multiple keys at one time. Detail of this will be covered in the following subsection.

### 3.2.2 Normalization of SQO overloads

Many SQO methods have several overloadings, some of which have different expressive power but cannot be translated from any textual syntax of LINQ. Those with more expressive power usually take more arguments than the others. To handle this great variety of possible overloadings, a normalization phase is introduced here. It tries to reduce the total numbers of overloadings by adding some trivial arguments to those methods taking less arguments, and therefore provides the next step with particular SQO methods with unified signatures. In the following, all methods that require normalization are explained.

#### 3.2.2.1 Normalization of `SelectMany`

The supported method `SelectMany` have the following overloadings:

1. `public static IEnumerable<TResult> SelectMany<Tsource, TResult>(`

```
    this IEnumerable<TSource> source,
    Func<Tsource, IEnumerable<TResult> > selector);
```

2. 
```
public static IEnumerable<TResult> SelectMany<Tsource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<Tsource, IEnumerable<TCollection> > collectionSelector,
    Func<Tsource, TCollection, TResult> resultSelector);
```

The first overloading enumerates the `source` sequence, applies the `selector` projection on it, and merges the resulting items into a single sequence of items of type `TResult`.

If a custom result is needed instead of merely merging the items from the source sequence, the second overloading should be used. It invokes the `collectionSelector` projection over the `source` sequence, applies the `resultSelector` projection to each item of the above resulting sequence, and returns the merged result. Because `reseultSelector` applies to the sequence resulting from `collectionSelector`, the final result is also a sequence of items of type `TResult`. Textual syntax of LINQ can only be translated to the second overloading.

The difference between them is, whether the temporary sequence is merged and returned directly(in the first overloading), or further shaped by `resultSelector`(in the second overloading) before being merged and returned. Therefore, the first overloading can be normalized as the second by introducing a `resultSelector` that simply returns items of that sequence unmodified.

The normalization of method `SelectMany` is given as following:

$$\frac{\Gamma \vdash \texttt{src::TSource}, \texttt{col} \in \texttt{src.CollSel}}{\texttt{SelectMany(Source, src=>selector)} \mapsto \texttt{SelectMany(Source, src=>source.CollSec, (src, col)=>col)}}$$

#### 3.2.2.2 Normalization of `OrderBy` and `OrderByDescending`

The supported method `OrderBy` has the following overloading:

```
public static IOrderedEnumerable<TSource> OrderBy<Tsource, TKey>(
     this IEnumerabe<TSource> source,
     Func<Tsource, TKey> keySelector);
```

The supported method `OrderByDescending` has the following overloading:

```
public static IOrderedEnumerable<TSource> OrderByDescending<Tsource, TKey>(
     this IEnumerabe<TSource> source,
     Func<Tsource, TKey> keySelector);
```

In the translation from SQO to functional syntax of SQO, method `ThenBy` and `ThenByDescending` resulted from multiple key occurrences are eliminated by introducing a new method `OrderByMultiKey`, in order to have

all keys evaluated at a time. In particular, the first argument of method `OrderByMultiKey` is the source sequence being sorted. Each key and the direction on this key(encoded as a *Boolean* value, where *true* for ascending and $false$ for descending) composite the following 2 arguments. Therefore, method `OrderByMultiKey` takes $2 * n + 1$ arguments, where $n$ is the number of keys. But in functional syntax of SQO, orderings with single key still remain as the method `OrderBy` and `OrderByDescending`.

The normalization goes one step further, in that it translate method `OrderBy` and `OrderByDescending` into method `OrderByMultiKey`, although it only has one key. The resulting `OrderByMultiKey` hence has 3 arguments, the first one being the source and the last two being the key specification. This simplifies the following translation phase in that the translator does not need to distinguish method `OrderBy` (eventually `OrderByDescending`) and `OrderByMultiKey`, which would otherwise require two pieces of very similar translation code. Nevertheless, method name `OrderByMultiKey` actually loses its original meaning in the case of single key.

The normalization of method `OrderByMultiKey` is given as follows.

$$\frac{\Gamma \vdash \texttt{src::TSource}}{\texttt{OrderBy(Source, src=>key)} \;\mapsto\; \texttt{OrderByMultiKey(Source, src=>key, true)}}$$

$$\frac{\Gamma \vdash \texttt{src::TSource}}{\texttt{OrderByDescending(Source, src=>key)} \;\mapsto\; \texttt{OrderByMultiKey(Source, src=>key, false)}}$$

#### 3.2.2.3 Normalization of `GroupBy`

Excluding those involving custom comparers, method `GroupBy` has two overloadings, one including and one without an argument `elementSelector`, as shown below.

1. ```
   public static IEnumerable<IGrouping<TKey, TSource> > GroupBy<Tsource, TKey>(
     this IEnumerable<Tsource> source,
     Func<Tsource, TKey> keySelector);
   ```

2. ```
   public static IEnumerable<IGrouping<TKey, TElement> > GroupBy<Tsource, TKey,
   TElement>(
     this IEnumerable<Tsource> source,
     Func<Tsource, TKey> keySelector);
     Func<Tsource, TElement> elementSelector);
   ```

Both overloadings have a common argument `keySelector`, which is a predicate used to extract the key value from each item to group results based on the different key values.

In textual syntax of LINQ, `group by` clause is another clause besides `select` that can conclude a query. There should be a way for `group by` clause to shape the output result as `select` clause does. For this, the second overloading is needed. The argument `elementSelector` shapes the items of output sequence that has been clustered by argument `keySelector`. In fact, in textual syntax of LINQ, if the variable between keywords `group` and `by` is the same as the source variable representing items in source sequence being grouped, the first overloading is generated, otherwise the second is generated. For example, the following query:
`from c in customers group c by c.country`
is translated into the first overloading as:
`customers.GroupBy(c=>c.country)`
while the following query:
`from c in customers group c.name by c.country`
is translated into the second overloading as:
`customers.GroupBy(c=>c.country, c.name)`
The first query does not specify how the result should be shaped, while in the second it is specified that only the field `c.name` is projected in the result. Based on this difference, we can normalize the first overloading to the second by providing an argument `elementSelector` which is the same as the source sequence, meaning items of the source go into the result unmodified. In this case, types `TElement` and `TSource` would be the same.

The normalization rule for method `GroupBy` is given below.

$$\frac{\Gamma \vdash \texttt{src::TSource}}{\texttt{GroupBy(Source, src=>key)} \;\mapsto\; \texttt{GroupBy(Source, src=>key, src=>src)}}$$

The other SQO methods do not need to be normalized and can be consumed directly by the translation phase afterwards. However, for completeness, the supported(excluding those requiring custom comparers) overloadings of methods `Join` and `GroupJoin` will also be given here.

1. ```
   public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
       this IEnumerable<TOuter> outer,
       IEnumerable<TInner> inner,
       Func<TOuter, TKey> outerKeySelector,
       Func<TInner, TKey> innerKeySelector,
       Func<TOuter, TInner, TResult> resultSelector);
   ```

1. ```
   public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
       this IEnumerable<TOuter> outer,
       IEnumerable<TInner> inner,
       Func<TOuter, TKey> outerKeySelector,
       Func<TInner, TKey> innerKeySelector,
       Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
   ```

## 3.3 Translation

In this section, detail of translation from LINQ to Ferry will be given. Input to the translation phase is functional syntax of LINQ, while output is block of Ferry expressions which is semantically equivalent to the input query. In Section 3.3.1, translation rules regarding to different building blocks of functional syntax of LINQ are given.

### 3.3.1 Translation rules

A LINQ query in its functional syntax contains some or all of these building blocks: method invocation, literal expression, dot-separated expression, binary expression, unary expression, and object initialization(invocation of object constructor or `new` expression for anonymous type).

To define the translation rules, the translation function $\mathbb{T}$ is defined:

$$\frac{e_l = \texttt{[MethodCall|Literal|DotSeparated|BinaryExpr|UnaryExpr|NewExprWithInit]}}{\mathbb{T}[e_l]^{\Gamma} = [e_f]^{\Gamma'}}$$

Function $\mathbb{T}$ takes argument $e_l$ which is one of the building blocks of functional syntax of LINQ, and a variable environment $\Gamma$, returns Ferry expression $e_f$ and updates $\Gamma$ to $\Gamma'$.

Additionally, two auxiliary functions $\mathbb{G}$ and $\mathbb{V}$ are defined as follows:

$$\frac{\mathcal{N}_{tab}[\ (col_1, \ldots, col_n),\ (key_1, \ldots, key_m)\ ]}{\mathbb{G}[tab]^{\Gamma} = \quad [\ \texttt{table}\ tab(col_1\ t_1, \ldots, col_n\ t_n)\ \texttt{with keys}\ (key_1, \ldots, key_m)\ ]^{\Gamma}}$$

Function $\mathbb{G}$ takes argument $tab$ and generates a table reference expression in Ferry according to the Table Information Node $\mathcal{N}_{tab}$, whose column specifications are $(col_1, \ldots, col_n)$ with types $(t_1, \ldots, t_n)$ respectively, and key specifications are $(key_1, \ldots, key_m)$.

Function $\mathbb{V}$ returns a variable with the next available fresh name.

#### 3.3.1.1 Translation of literal expression

Literal expression in functional syntax of LINQ can represent either a literal or access to a source sequence(for example, after the `from` keyword). Translation of literal expression is straightforward. In the CST metamodel implemented in Project[2], class `Literal` has a field `literalKind` of type `LiteralKind`, which is an enumeration type that can distinguish whether the literal is of type `Int`, `String` or `Boolean` or `ID`. In the first three cases, it is translated to the corresponding literal expression in Ferry. In the last case, it is translated to a Table Reference expression in Ferry.

The translation rule for literal expression is given below:

$$\frac{\texttt{l.literalKind = LiteralKind.INT\_LIT}}{\mathbb{T}[l]^{\Gamma} \;=\; IntLiteral(l)^{\Gamma}}$$

$$\frac{\texttt{l.literalKind = LiteralKind.STRING\_LIT}}{\mathbb{T}[l]^{\Gamma} \;= StringLiteral(l)^{\Gamma}}$$

$$\frac{\texttt{l.literalKind = LiteralKind.BOOLEAN\_LIT}}{\mathbb{T}[l]^{\Gamma} \;=\; BooleanLiteral(l)^{\Gamma}}$$

$$\frac{\texttt{l.literalKind = LiteralKind.ID\_LIT}}{\mathbb{T}[l]^{\Gamma} \;=\; \mathbb{G}[l]^{\Gamma}}$$

#### 3.3.1.2 Translation of binary expression

In a binary expression, both operands are translated first. Prerequisite for this is that both operands are of the same type. A binary expression in Ferry is then created, by connecting the translation results of both operands with the same binary operator. The translation rules for binary expression are given below, separated by different kinds of operators:

$$\frac{\begin{array}{c}\texttt{BinOp} = [\ +\ |\ -\ |\ *\ |\ /\ |\ >\ |\ <\ |\ ==\ |\ >=\ |\ <=\ |\ !=\ ] \\ t = \texttt{Int} \\ \Gamma \vdash e_1 :: t \\ \Gamma \vdash e_2 :: t \\ \mathbb{T}[e_1]^{\Gamma} \;=\; e_1' \\ \mathbb{T}[e_2]^{\Gamma} \;=\; e_2'\end{array}}{\mathbb{T}[e_1\ \ \texttt{BinOp}\ e_2]^{\Gamma} \;=\; (e_1'\ \ \texttt{BinOp}\ e_2')^{\Gamma}}$$

$$\frac{\begin{array}{c}\texttt{BinOp}' = [\ ==\ |\ !=\ ] \\ t' = \texttt{String} \\ \Gamma \vdash e_1 :: t \\ \Gamma \vdash e_2 :: t \\ \mathbb{T}[e_1]^{\Gamma} \;=\; e_1' \\ \mathbb{T}[e_2]^{\Gamma} \;=\; e_2'\end{array}}{\mathbb{T}[e_1\ \ \texttt{BinOp}'\ e_2]^{\Gamma} \;=\; (e_1'\ \ \texttt{BinOp}'\ e_2')^{\Gamma}}$$

$$\frac{\begin{array}{c}\texttt{BinOp}'' = [\ \texttt{and}\ |\ \texttt{or}\ ] \\ t'' = \texttt{Boolean} \\ \Gamma \vdash e_1 :: t \\ \Gamma \vdash e_2 :: t \\ \mathbb{T}[e_1]^{\Gamma} \;=\; e_1' \\ \mathbb{T}[e_2]^{\Gamma} \;=\; e_2'\end{array}}{\mathbb{T}[e_1\ \ \texttt{BinOp}''\ e_2]^{\Gamma} \;=\; (e_1'\ \ \texttt{BinOp}''\ e_2')^{\Gamma}}$$

#### 3.3.1.3 Translation of unary expression

In a unary translation, operand is translated first. A unary expression in Ferry is then created, by prefixing the translation result of the operand with the same binary operator. The translation rules for unary expression are given below:

$$\frac{\begin{array}{c}\texttt{UnOp} = [\ \texttt{not}\ ] \\ \Gamma \vdash e_1 :: \texttt{Boolean} \\ \mathbb{T}[e_1]^{\Gamma} \ = \ e_1'\end{array}}{\mathbb{T}[\texttt{UnOp}\ e_1]^{\Gamma} \ = \ (\ \texttt{UnOp}\ e_1'\ )^{\Gamma}}$$

$$\frac{\begin{array}{c}\texttt{UnOp}' = [\ \texttt{-}\ ] \\ \Gamma \vdash e_1 :: \texttt{Int} \\ \mathbb{T}[e_1]^{\Gamma} \ = \ e_1'\end{array}}{\mathbb{T}[\texttt{UnOp}'\ e_1]^{\Gamma} \ = \ (\ \texttt{UnOp}'\ e_1'\ )^{\Gamma}}$$

#### 3.3.1.4 Translation of dot-separated expression

Since we exclude method invocation on object, dot-separated expression can only occur in field access. Field access involves the access to single or multiple attribute or reference. Also, these access can either be within the own class-frame, or within the class-frame of superclass or implemented interfaces. Here a rule for each of these situations will be given, assuming that we have already determined whether the field access in question is single or multiple attribute or reference access. How this is determined will be explained in Section 3.4 about implementation.

`o.f`, where object `o` is of class `C` and `f` is a single attribute declared in class `C`, is translated into a nominal access of Tuple in Ferry. Since `o.f` can only occur within context of Lambda expression which is argument of SQO method, it can be guaranteed that variable `o` is in scope. Following is the translation rule for `o.f` in the case of single attribute. Here a slightly different notation is used: $C$ in $\mathbb{T}[\texttt{o.f}]^{\Gamma}_{C}$ means that this translation assumes that field `f` is declared in class `C`.

$$\frac{\Gamma \vdash \texttt{o::C}}{\mathbb{T}[\texttt{o.f}]^{\Gamma}_{C} \ = \ (\ \texttt{o.f}\ )^{\Gamma}}$$

`o.f`, where object `o` is of class `C` and `f` is a multiple attribute declared in class `C`, is translated into a Let expression in Ferry. Recalling Section 2.3, Section 2.4 and the example shown in Figure 2.11, values of multiple attribute are stored in the bridging table called `C_f`. The resulted Let expression in Ferry accesses that bridging table, iterates through it and filters out those rows that belong to object `o`. The filtering takes place based on the matching between the surrogate value in column `f` in table `C` and the

values of column `iter` in table `C_f`. Since `o.f` can only occur within context of Lambda expression which is argument of SQO method, it can be guaranteed that variable `o` is in scope. Following is the translation rule for `o.f` in the case of multiple attribute.

$$\frac{v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}() \qquad \Gamma \vdash \texttt{o::C}}{\begin{array}{rcl}\mathbb{T}[\texttt{o.f}]^{\Gamma}_{C} & = & (\ \texttt{let}\ v_1 = \mathbb{G}(\texttt{C\_f}) \\ & & \quad \texttt{in for}\ v_2\ \texttt{in}\ v_1 \\ & & \quad\quad \texttt{where C.f =}\ v_2\texttt{.iter} \\ & & \quad\quad \texttt{return}\ v_2)^{\Gamma}\end{array}}$$

`o.f`, where object `o` is of class `C` and `f` is a reference(regardless of its multiplicity) of type `R` declared in class `C`, is translated into a Let expression in Ferry. Recalling Section 2.3, Section 2.4 and the example shown in Figure 2.11, id of object(s) being referred to are stored in column `item0` in the bridging table called `C_f`. In the case of single reference, each row in table `C_f` has individual value of column `iter`, while in the case of multiple reference, rows in table `C_f` that belong to the same reference share the same value of column `iter`. This value, as a surrogate value, is then stored in column `f` of the row representing object `o` in table `C`. The resulted Let expression in Ferry accesses that bridging table, iterates through it and filters out those rows that belong to object `o` based on the matching between the surrogate value in column `f` in table `C` and the values of column `iter` in table `C_f`. Within each iteration, an embedded Let expression accesses table `R` and locates the object(s) being referred to based on the matching of column `item0` in table `C_f` and the id of objects in table `R`. Since `o.f` can only occur within context of Lambda expression which is argument of SQO method, it can be guaranteed that variable `o` has already been in scope. This translation also update the variable environment by adding a binding indicating that field `f` is of type `R`. This is to handle the possible case of chained references like $\texttt{o.f}_1 \ldots \texttt{f}_n$.

$$\frac{v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}(),\ v_3 = \mathbb{V}(),\ v_4 = \mathbb{V}() \qquad \Gamma \vdash \texttt{o::C} \qquad \Gamma' = \Gamma + [f \mapsto R]}{\begin{array}{rcl}\mathbb{T}[\texttt{o.f}]^{\Gamma}_{C} & = & (\ \texttt{let}\ v_1 = \mathbb{G}(c\_f) \\ & & \quad \texttt{in for}\ v_2\ \texttt{in}\ v_1 \\ & & \quad\quad \texttt{return}\ ( \\ & & \quad\quad\quad \texttt{let}\ v_3 = \mathbb{G}(B) \\ & & \quad\quad\quad \texttt{in for}\ v_4\ \texttt{in}\ v_3 \\ & & \quad\quad\quad\quad \texttt{where o.f =}\ v_2\texttt{.iter and}\ v_2\texttt{.item0 =}\ v_4\texttt{.id} \\ & & \quad\quad\quad\quad \texttt{return}\ v_4 \\ & & \quad\quad ) \\ & & )^{\Gamma'}\end{array}}$$

In the case of `o.f`, where object `o` is of class `C` and `f` is a field(either attribute or reference) which is *not* declared in class `C`, the translator first attempts to apply the same rules to `f` on class `S`, the superclass of `C`. If `f` is not declared in class `S` either, it will then try every interfaces $\texttt{I}_n$ that `C` implements, until it finds an interface that declared `f`. This also applies to interfaces that these interfaces implement. Following is the translation rules for member access within class-frame of super-class or interface.

$$\frac{\Gamma \vdash \texttt{o::C}}{\mathbb{T}[\texttt{o.f}]^{\Gamma}_{C} = \mathbb{T}[\texttt{o.f}]^{\Gamma}_{S}}$$

$$\frac{\Gamma \vdash \texttt{o::C}}{\mathbb{T}[\texttt{o.f}]^{\Gamma}_{C} = \mathbb{T}[\texttt{o.f}]^{\Gamma}_{I_n}}$$

#### 3.3.1.5 Translation of `MethodCall`

`MethodCall` is the root node of a functional syntax of LINQ. SQO methods are not defined as sub-classes of `MethodCall`. Rather, each instance of `MethodCall` has a string field to specify the name of this method call, and a list to specify its arguments. In the following, the translation rules for all methods within functional syntax of LINQ will be given.

- `Select(src, v=>prj)`

A `Select` method in functional syntax of LINQ takes two arguments, the first one `src` being the source sequence and the second one a lambda expression `v=>prj` that represents a projection operator. This method applies the projection on the source sequence to produce another sequence. Source sequence `src` can be a table access or other expressions that are translated from other methods of functional syntax of LINQ. Variable `v` represents each element within `src`, and projection `prj` shapes out each output element in terms of `v`. Projection `prj` can be member access, initial expression or some other expression.

A `Select` method is translated into a Let expression in Ferry, whose embedded For expression iterates through the source sequence and returns the resulted sequence being shaped by the projection. Also, it updates the variable environment $\Gamma$ by adding a mapping from `v` to the type of items in `src`.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}() \\ \mathbb{T}[\texttt{src}]^{\Gamma} = \texttt{src}' \\ \mathbb{T}[\texttt{prj}]^{\Gamma} = \texttt{prj}' \\ \Gamma + [\texttt{v::t}_{src}] = \Gamma' \end{array}}{\begin{array}{l} \mathbb{T}[\texttt{ Select(src, v=>prj) }]^{\Gamma} = \quad (\texttt{ let } v_1 \texttt{ = src}' \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{in for v in } v_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{return prj}' \;)^{\Gamma'} \end{array}}$$

- `Where(src, v=>pred)`

A `Where` method in functional syntax of LINQ filters source sequence `src` based on a predicate `pred`, and returns another sequence whose items fulfill the predicate. Source sequence `src` can be a table access or other expressions that are translated from other methods of functional syntax of LINQ. Variable `v` represents each element within `src`, and predicate `pred` is a boolean expression in terms of `v`.

A `Where` method is translated into a Let expression in Ferry, whose embedded For expression iterates through the source sequence. The optional `where` clause in this For expression is generated from the predicate `pred`. Also, it updates the variable environment $\Gamma$ by adding a mapping from `v` to the type of items in `src`.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}() \\ \mathbb{T}[\texttt{src}]^{\Gamma} = \texttt{src}' \\ \mathbb{T}[\texttt{pred}]^{\Gamma} = \texttt{pred}' \\ \Gamma + [\texttt{v::t}_{src}] = \Gamma' \end{array}}{\begin{array}{rl} \mathbb{T}[\texttt{ Where(src, v=>pred) }]^{\Gamma} = & (\texttt{ let } v_1 \texttt{ = src}' \\ & \texttt{in for v in } v_1 \\ & \quad \texttt{where pred}' \\ & \quad \texttt{return v})^{\Gamma'} \end{array}}$$

- `SelectMany(src, v=>collSel, (v,col)=>resSel)`

After normalization, a `SelectMany` method in functional syntax of LINQ uses the second argument `collsel` to retrieve a collection selector from each item of source sequence `src`, and applies the `resSel` projection on the sequence taken from collection selector. Items from this sequence are then merged together to form the final result. Source sequence `src` can be a table access or other expressions that are translated from other methods of functional syntax of LINQ. Variable `v` represents each element within `src`, and variable `col` represents each element(a sequence itself) within the sequence retrieved by collection selector. Please note that although the result projection `resSel` applies on items of sequences which is in turn items of the source sequence, the output result is a *flattened* sequence, instead of a sequence of embedded sequences.

A `SelectMany` method is translated into a Let expression in Ferry which calls the `concat` method to merge the sequence resulted from an embedded For expression. That For expression iterates through the source sequence `src`. For each round of iteration, it applies another loop over the collection selector, and performs the `resSel` projection on each item taken from the collection selector. The call to method `concat` ensures that the result is one single flattened sequence. Also, it updates the variable environment $\Gamma$ by adding two mappings: from `v` to the type of items in `src`, and from `col` to the type of items in `collSel`.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}() \\ \mathbb{T}[\texttt{src}]^{\Gamma} = \texttt{src}' \\ \mathbb{T}[\texttt{collSel}]^{\Gamma} = \texttt{collSel}' \\ \mathbb{T}[\texttt{resSel}]^{\Gamma} = \texttt{resSel}' \\ \Gamma + [\texttt{v::t}_{src}] = \Gamma' \\ \Gamma + [\texttt{col::t}_{collSel}] = \Gamma' \end{array}}{\begin{array}{l} \mathbb{T}[\ \texttt{SelectMany(src, v=>collSel, (v,col)=>resSel)}\ ]^{\Gamma} = \\ \quad (\ \texttt{let}\ v_1 = \texttt{src}' \\ \quad\ \texttt{in concat (} \\ \quad\quad \texttt{for v in}\ v_1 \\ \quad\quad \texttt{return (} \\ \quad\quad\quad \texttt{let}\ v_2 = \texttt{collSel}' \\ \quad\quad\quad \texttt{in for col in}\ v_2 \\ \quad\quad\quad\quad \texttt{return resSel}' \\ \quad\quad )\ \texttt{)} \\ \quad )^{\Gamma'} \end{array}}$$

- $\texttt{OrderByMultiKey(src, v=>key}_1\texttt{, dir}_1, \ldots, \texttt{v=>key}_n\texttt{, dir}_n\texttt{)}$

After normalization, an `OrderByMultiKey` method in functional syntax of LINQ always consumes $2 * n + 1$ arguments, the first one being the source sequence and the followings pairs being the key specifications (in each pair, the first one is the column name to specify the sorting criterion, and the second one the direction of that criterion, encoded as `true` for *ascending* and `false` for *descending*). Source sequence `src` can be a table access or other expressions that are translated from other methods of functional syntax of LINQ. Variable `v` represents each element within `src`. Each key specification $\texttt{key}_n$ can be member access or other expression which is in terms of `v`.

An `OrderByMultkKey` method is translated into a Let expression in Ferry, whose embedded For expression iterates through the source sequence `src`. The optional `order by` clause within that For expression is generated from the translation result of key specifications, with `ascending` as the default direction. After sorted, each item `v` within the sequence is returned. Also, it updates the variable environment $\Gamma$ by adding a mapping from `v` to the type of items in `src`.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}() \\ \mathbb{T}[\texttt{src}]^{\Gamma} = \texttt{src}' \\ \mathbb{T}[\texttt{key}_1]^{\Gamma} = \texttt{key}'_1 \\ \vdots \\ \mathbb{T}[\texttt{key}_n]^{\Gamma} = \texttt{key}'_n \\ \Gamma + [\texttt{v::t}_{src}] = \Gamma' \end{array}}{\mathbb{T}[\ \texttt{OrderByMultiKey(src, v=>key}_1\texttt{, dir}_1, \ldots, \texttt{v=>key}_n\texttt{, dir}_n\texttt{)}\ ]^{\Gamma} =}$$

$$
\begin{array}{l}
(\ \texttt{let}\ v_1\ \texttt{=}\ \texttt{src}' \\
\quad \texttt{in for v in}\ v_1 \\
\qquad \texttt{order by key}'_1\texttt{[descending]}, \ldots, \texttt{key}'_n\texttt{[descending]} \\
\qquad \texttt{return v}\ )^{\Gamma'}
\end{array}
$$

- `GroupBy(src, v=>key, v=>eleSel)`

After normalization, a `GroupBy` method in functional syntax of LINQ takes three arguments, the first one being the source sequence `src`, the second one being the key on which to cluster the source sequence based on different value of `key`, and the third one being the element selector that shapes out the resulted sequence. Source sequence `src` can be a table access or other expressions that are translated from other methods of functional syntax of LINQ. Variable `v` represents each element within `src`. Key specification `key` can be member access or other expression which is in terms of `v`. Element selector `eleSel` can be member access, initial expression or other expression which is in terms of `v`.

A `GroupBy` method is translated into a Let expression in Ferry, whose embedded For expression iterates through the source sequence `src`. The optional `group by` clause in that For expression is generated from the translation result of the key specification. This translation also updates the variable environment $\Gamma$ by adding a mapping from `v` to the type of items in `src`.

$$
\frac{\begin{array}{c} v_1 = \mathbb{V}() \\ \mathbb{T}[\texttt{src}]^{\Gamma} = \texttt{src}' \\ \mathbb{T}[\texttt{key}]^{\Gamma} = \texttt{key}' \\ \mathbb{T}[\texttt{eleSel}]^{\Gamma} = \texttt{eleSel}' \\ \Gamma + [\texttt{v::t}_{src}] = \Gamma' \end{array}}{\begin{array}{l} \mathbb{T}[\ \texttt{GroupBy(src, v=>key, v=>eleSel)}\ ]^{\Gamma} = \\ \quad (\ \texttt{let}\ v_1\ \texttt{=}\ \texttt{src}' \\ \quad\ \texttt{in for v in}\ v_1 \\ \qquad \texttt{group by key}' \\ \qquad \texttt{return eleSel}'\ )^{\Gamma'} \end{array}}
$$

- `Join(outer, inner, o=>okey, i=>ikey, (o, i)=>resSel)`

A `Join` method in functional syntax of LINQ takes two source sequences, represented as `outer` and `inner`, finds matching pairs from them based on keys `okey` and `ikey` extracted from both sequences, and return a sequence composed with elements that fulfill the match. Both source sequences `outer` and `inner` can be table access or other expressions that are translated from other methods of functional syntax of LINQ. Variables `o` and `i` represents elements in sequences `outer` and `inner`, respectively. Result selector `resSel` performs a projection to shapes the output in terms of `o` and `i`. Please note

that items in sequence `outer` that have no matching counterpart in sequence `inner` will not go into the resulted sequence, and vice versa.

A `Join` method is translated into a Let expression in Ferry, which has an embedded method call to `concat` that merges the resulted sequence of a For expression. That For expression iterates through the `outer` sequence. In each iteration, another For expression loops over the `inner` sequence, and returns items that fulfill the equivalence of items from both sequences based on key specifications `okey` and `ikey`. The resulted sequence are shaped by result selector `resSel`. Also, this translation updates the variable environment $\Gamma$ by adding mappings from `o` to type of items in `outer`, and from `i` to type of items in `inner`.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}() \\ \mathbb{T}[\texttt{outer}]^{\Gamma} = \texttt{outer}' \\ \mathbb{T}[\texttt{inner}]^{\Gamma} = \texttt{inner}' \\ \mathbb{T}[\texttt{okey}]^{\Gamma} = \texttt{okey}' \\ \mathbb{T}[\texttt{ikey}]^{\Gamma} = \texttt{ikey}' \\ \mathbb{T}[\texttt{resSel}]^{\Gamma} = \texttt{resSel}' \\ \Gamma + [\texttt{o::t}_{\texttt{outer}}] = \Gamma' \\ \Gamma + [\texttt{i::t}_{\texttt{inner}}] = \Gamma' \end{array}}{\begin{array}{l} \mathbb{T}[\ \texttt{Join(outer, inner, o=>okey, i=>ikey, (o, i)=>resSel)}\ ]^{\Gamma} = \\ \quad (\ \texttt{let}\ v_1 = \texttt{outer}' \\ \quad\ \texttt{in concat (} \\ \quad\quad \texttt{for o in}\ v_1 \\ \quad\quad \texttt{return (} \\ \quad\quad\quad \texttt{let}\ v_2 = \texttt{inner}' \\ \quad\quad\quad \texttt{in for i in}\ v_2 \\ \quad\quad\quad\quad \texttt{return (} \\ \quad\quad\quad\quad\quad \texttt{if o.okey == i.ikey then resSel}' \\ \quad\quad\quad\quad\quad \texttt{else []} \\ \quad\quad\quad\quad \texttt{)} \\ \quad\quad\quad \texttt{)} \\ \quad\quad \texttt{)} \\ \quad )^{\Gamma'} \end{array}}$$

- `GroupJoin(outer, inner, o=>okey, i=>ikey, (o, into)=>resSel)`

A `GroupJoin` method in functional syntax of LINQ is similar to method `Join`, with the different signature of result selector `resSel`. `resSel` requires an object of type `IEnumerable[inner]`, instead of a single object of the type of elements in `inner`, because it projects a hierarchical result of type `IEnumerable[TResult]`, where `TResult` is the type of translation result of `resSel`. Each item of type `TResult` consists of an item extracted from the source sequence `outer` and a group of items of type `TInner`, joined from another source sequence `inner`. As a result of this behavior, the output

is not a flattened sequence like the one of method `Join`, but a hierarchical sequence of items. Nevertheless, both methods `GroupJoin` and `Join` will produces equivalent result if the mapping is a one-to-one relationship. In cases in which a corresponding element group in the `inner` sequence is absent, the `GroupJoin` method extracts the `outer` sequence element paired with an empty sequence, which also behaves differently from method `Join`.

Method `GroupJoin` is translated into a Let expression in Ferry, whose embedded For expression iterates through the outer source `outer`. For each iteration, another embedded Let expression defines the `into` variable and embeds another For expression to loop over inner sequence `inner`. This inner iteration returns a sequence extracted from `inner`, filtered by the equivalence of keys `okey` and `ikey`. This result is then assigned to variable `into and` shaped out by the result selector `resSel` to produce the final result. This translation also updates the variable environment $\Gamma$ by adding mappings from `o` to type of items in `outer`, from `i` to type of items in `inner`, and from `into` to type `IEnumerable[inner]`.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}() \\ \mathbb{T}[\texttt{outer}]^{\Gamma} = \texttt{outer}' \\ \mathbb{T}[\texttt{inner}]^{\Gamma} = \texttt{inner}' \\ \mathbb{T}[\texttt{okey}]^{\Gamma} = \texttt{okey}' \\ \mathbb{T}[\texttt{ikey}]^{\Gamma} = \texttt{ikey}' \\ \mathbb{T}[\texttt{resSel}]^{\Gamma} = \texttt{resSel}' \\ \Gamma + [\texttt{o}::\texttt{t}_{\texttt{outer}}] = \Gamma' \\ \Gamma + [\texttt{i}::\texttt{t}_{\texttt{inner}}] = \Gamma' \\ \Gamma + [\texttt{into}::\texttt{t}_{\texttt{IEnumerable[inner]}}] = \Gamma' \end{array}}{\begin{array}{l} \mathbb{T}[\,\texttt{GroupJoin(outer, inner, o=>okey, i=>ikey, (o, into)=>resSel)}\,]^{\Gamma} = \\ \quad (\ \texttt{let}\ v_1 = \texttt{outer}' \\ \quad\ \texttt{in for o in}\ v_1 \\ \quad\quad \texttt{return (} \\ \quad\quad\quad \texttt{let into = (} \\ \quad\quad\quad\quad \texttt{let}\ v_2 = \texttt{inner}' \\ \quad\quad\quad\quad \texttt{in for i in}\ v_2 \\ \quad\quad\quad\quad\quad \texttt{where okey}' = \texttt{ikey}' \\ \quad\quad\quad\quad\quad \texttt{return i} \\ \quad\quad\quad\quad \texttt{)} \\ \quad\quad\quad \texttt{in resSel}' \\ \quad\quad \texttt{)} \\ \quad )^{\Gamma'} \end{array}}$$

- `Cast<T>(o)`

Method `Cast` performs a casting on object `o` of type `C` to type `T`. Recalling Section 2.3, Section 2.4 and the example shown in Figure 2.11, this involves the access to different tables that store objects of different type, and the

location of the different parts of one particular object through the surrogate values in columns `SuperCls`, `SuperObj`, `subCls` and `subObj`. Note that `Cast` is a node of type `MethodCall`, meaning that the following translation rules are only applicable to explicit casting. When introducing the translation rules for member access, the case of implicit casting has already been covered.

Because the translation rules for casting to super-class and to interface are different, they will be explained here separately. For this, a new notation is defined:

$$\texttt{C} \rightarrow \texttt{T}$$

meaning that class `C` is inherited from class `T` if `T` is a normal class, or class `C` implements `T` if `T` is an interface.

A casting to super-class is translated to a Let expression in Ferry, which access the table for the super-class and iterates it with its embedded For expression. In this For expression, the optional `where` clause is generated filter out the correct row. This row should have `C` as its value in column `SubCls`, and the id of `o` as its value in column `SubObj`. This row is return to where this method `Cast` is called, which can be the declaration of source variables, member access, result projector or any other expression.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}() \\ \Gamma \vdash \texttt{o::C} \\ \texttt{C} \rightarrow \texttt{T} \end{array}}{\begin{array}{l} \mathbb{T}[\ \texttt{Cast<T>(o)}\ ]^{\Gamma} = \quad (\ \texttt{let}\ v_1\ \texttt{=}\ \mathbb{G}\texttt{(T)} \\ \qquad\qquad\quad \texttt{in for}\ v_2\ \texttt{in}\ v_1 \\ \qquad\qquad\qquad \texttt{where}\ v_2\texttt{.SubCls = C and}\ v_2\texttt{.SubObj = o.id} \\ \qquad\qquad\qquad \texttt{return}\ v_2\ )^{\Gamma} \end{array}}$$

A casting to interface is translated to a Let expression in Ferry, which access the table for interfaces of `C` and iterates it with its embedded For expression. In this For expression, the optional `where` clause is generated filter out the correct row. This row should have `T` as its value in column `interface`, and the id of `o` as its value in column `iter`. Then another embedded For expression iterates through table `T` and filters out rows matching its value in column `obj` and the id of that row in the bridging table. This row is return to where this method `Cast` is called, which can be the declaration of source variables, member access, result projector or any other expression.

$$\frac{\begin{array}{c} v_1 = \mathbb{V}(),\ v_2 = \mathbb{V}(),\ v_3 = \mathbb{V}(),\ v_4 = \mathbb{V}() \\ \Gamma \vdash \texttt{o::C} \\ \texttt{C} \rightarrow \texttt{T} \end{array}}{}$$

$\mathbb{T}[$ `Cast<T>(o)` $]^{\Gamma} =$ ( `let` $v_1$ `=` $\mathbb{G}$`(C+"interfaces")`
`in for` $v_2$ `in` $v_1$
`where` $v_2$`.iter = o.id and` $v_2$`.interface = T`
`return`
`let` $v_3$`=` $\mathbb{G}$`(T)`
`in for` $v_4$`in` $v_3$
`where` $v_4$`.id =` $v_2$`.obj`
`return` $v_4$ $)^{\Gamma}$

- `Min(s)`
- `Max(s)`
- `Sum(s)`

These aggregation methods are self-explained, and they can be translated directly into their Ferry counterparts. prerequisite is, type of items in operand sequence `s` should be `Int`.

$$\frac{t_s = \texttt{Int}}{\mathbb{T}[\texttt{Min(s)}]^{\Gamma} = [\texttt{min(s)}]^{\Gamma}}$$

$$\frac{t_s = \texttt{Int}}{\mathbb{T}[\texttt{Max(s)}]^{\Gamma} = [\texttt{max(s)}]^{\Gamma}}$$

$$\frac{t_s = \texttt{Int}}{\mathbb{T}[\texttt{Sum(s)}]^{\Gamma} = [\texttt{sum(s)}]^{\Gamma}}$$

- `Count(s)`

Method `Count` is translated into a function application `length` in Ferry, which has a different name but identical semantic: to retrieve the number of items in a source sequence `s`.

$$\frac{}{\mathbb{T}[\texttt{Count(s)}]^{\Gamma} = [\texttt{length(s)}]^{\Gamma}}$$

- `Average(s)`

Method `Average` is translated into Ferry expression in which method `sum` is called and the result is divided by the result of another method call to `length`. prerequisite is, type of items in operand sequence `s` should be `Int`.

$$\frac{t_s = \texttt{Int}}{\mathbb{T}[\texttt{Average(s)}]^{\Gamma} = [\texttt{sum(s)/length(s)}]^{\Gamma}}$$

- `Take(src, n)`

Method `Take` takes the first $n$ items from source sequence `src`. It is translated into a function application `take` in Ferry. If `n` exceeds the size of `src`, all items in `src` are returned.

$$\frac{\texttt{n} \in \mathbb{N}}{\mathbb{T}[\texttt{Take(src, n)}]^{\Gamma} = [\texttt{take(n, src)}]^{\Gamma}}$$

## 3.4 Implementation and result

The translation rules defined above are the foundation of this process. Yet for completeness, there are still several points regarding to the implementation aspects that should be explained and clarified.

### 3.4.1 From one tree to another

The translation from functional syntax of LINQ to Ferry expression is essentially a process that visits an input concrete syntax tree(CST), and generates another one. For both ends of this translation, CST metamodels are needed. Project [2] has implemented the entire CST metamodels for both textual and functional syntax of LINQ. As the input of our translation, functional syntax of LINQ is represented as a CST, with a node of type `MethodCall` as its root.

On the other end of the translation, we reused and refactored the CST metamodel for Ferry syntax which is defined by Project ScalaQL [2]. The refactor mostly lies in the definition of a "pretty-printer", which generates the string representation of a tree of Ferry expression in a more structurally indented and hence more readable manner. This will be examined in more detail in Section 3.4.5. This metamodel defined each syntax element of Ferry shown in Figure 3.2 on page 29, as case classes in Scala. During the translation, nodes and leaves of a CST, which are instances of these case classes, are instantiated accordingly, and the CST is built in this way. Listing 3.2 shows how method `Where` is translated according to the translation rule, and how the corresponding CST of Ferry is built.

```
1  case "Where" => {
2    var source = getSource(fs.getArgs.get(0))
3    var paramVar = fs.getArgs.get(1).asInstanceOf[LambdaExpr].
         getParams.get(0)
4    bindings += (paramVar -> source._2.asInstanceOf[IEnumerable].
         elemType)
5    var sourceVar = VarIDDecl(getNextVar)
6    var forVar = VarIDDecl(fs.getArgs.get(1).asInstanceOf[LambdaExpr].
         getParams.get(0))
```

[2] http://www.sts.tu-harburg.de/people/mi.garcia/ScalaQL/

```
7     var pred = translate(fs.getArgs.get(1).asInstanceOf[LambdaExpr].
          getBody)
8     var res = LetExpr(LetBindingClause(List((sourceVar, source._3))),
9             ForExpr(ForBindingClause(List((forVar, sourceVar))),
10            Some(FerrySyntax.WhereClause(pred._3)),None,None,None,
11            forVar)
12    )  //end of LetExpr
13    (sourceVar.toString, source._2, res)
14  }
```

Listing 3.2: Snippet that translates `Where` method and builds the corresponding CST of Ferry expressions

### 3.4.2 Type checking of the translation

As shown in Line 13, Listing 3.2, the return type of the translation is a triple value. Generally, the first item of that triple is a name that can identify the node of the resulted Ferry expressions in the final result, and the second item is the type of the result, and the third one is the resulted Ferry expressions. This sub-section focuses on the resulted type of each translation.

Each translation rule applies to a particular element in the functional syntax of LINQ, returning a block of Ferry expressions that has a particular type. This type is dependent on the syntax element from which it is translated, and in some cases also on the source sequence that this syntax element consumes.

In particular, literals in LINQ are translated to literal in Ferry with the identical types. Member access is translated into different kinds of Ferry expressions, whose type is taken from the type of that member being accessed. Special case is for multiple attributes and references, resulting in Ferry expression with type `IEnumerable`, which then has the same element type as that of attribute/reference. For `MethodCall`, the resulting Ferry expression is produced with type `IEnumerable`, with various possibilities of element types. For method `Select` and `Selectmany`, this type is taken from type `TResult` in the SQO signature. These two methods return sequences of elements that are shaped by the projections, therefore type `TResult` is dependent on this projector. For method `OrderBy`, the result is a sequence of type `IOrderedEnumerable`, whose element type is `TSource`, which is the type of elements of the source sequence. Similarly, method `GroupBy` results in a sequence of type `Grouping`, which consists a member of type `TKey`, given by the key specification, and another member of type `TSource`, the element type of the source sequence. For methods `Join` and `GroupJoin`, the resulting Ferry expressions are of type `IEnumerable`, whose element type is `TResult`, determined by the argument `resultSelector`. For other aggregation operators such as `Min`, `Max`, the returns type is set to `Int`, which is the one of their operands.

This type information is important during the translation process, because translation needs to type check if arguments of particular method are of the correct types, or if they are of identical types. These arguments can in turn be other `MethodCall`s that are to be translated, therefore the resulting blocks of Ferry expressions should themselves have the correct types.

For this, return types are modelled in the snippet shown in Listing 3.3. Type `IEnumerable`, along with its two sub-classes `OrderedEnumerable` and `Grouping`, represents sequence returned by some SQO methods(and therefore from the resulted blocks of Ferry expressions), while type `OtherType` represents other individual types, for example, returned by a single member access.

```
1  abstract case class ReturnType() {}
2  case class IEnumerable(elemType:ReturnType) extends ReturnType{
3    override def toString = "IEnumerable<" + elemType.toString + ">"
4  }
5  case class OrderedEnumerable(override val elemType:ReturnType) extends
       IEnumerable(elemType) {}
6  case class Grouping(key:String, override val elemType:ReturnType)
       extends IEnumerable(elemType) {
7    override def toString = "Grouping<" + key + ", " + elemType.
         toString + ">"
8  }
9  case class OtherType(ofType:String) extends ReturnType {
10   override def toString = ofType
11 }
```

Listing 3.3: Snippet that models the return types of SQO methods

### 3.4.3 Translating transparent identifier

Each transparent identifier is an instance of one anonymous type which is generated when particular textual LINQ queries are translated into their corresponding SQO methods[2]. Recalling the example shown in Listing 3.1 on page 28, this anonymous type contains fields that are declared in the initial expression in terms of any variables that are in scope at the moment. Once declared, these fields can be accessed through the transparent identifier, and these fields themselves can contain any other transparent identifier. Therefore, the critical step in translating transparent identifier is to keep the binding between transparent identifier and the anonymous type it is of.

Before the translation, Table Information Nodes for all types in the EMF model have already been generated during the persistence phase. These nodes are also taken over to the translation phases. When anonymous types come into being, they should also be treated the same way as the others. Therefore, *temporary* Table Information Nodes for these types should also

be generated and included in this set of nodes. These nodes are temporary in that they are dependent on particular queries being proceeded, and only exist during the translation phases but should not affect the persistence result after this phase. For each anonymous type, a Table Information Node is generated. If any of its fields are references or multiple attributes, nodes of bridging tables for them are also generated. For reference, key from such newly-generated bridging tables points to the Table Information Node representing the type of that reference, or to the Table Information Node representing another anonymous type, in the case when this field is another transparent identifier.

Once these nodes are generated and the hierarchies are constructed, the binding from the transparent identifier to its related anonymous type will be added to the variable environment. After that, this transparent identifier and member access with it can be treated as usual.

### 3.4.4 Resolving member access

When introducing the translation rules for member access in Section 3.3.1, we assumed that it had been determined whether this member access involves single or multiple attribute or reference.

This can be determine by accessing the corresponding Table Information Node. Within each Table Information Node, only values of single attributes are stored in it. For multiple attributes and references, only surrogate values are stored in the table, plus a corresponding entries in field `keys`. What's more, during the persistence phases, a map called `ref (C, ref)` $\mapsto$ `R` is maintained, meaning that in class `C` there is a reference named `ref` whose type is `R`. Having these pieces of information, we can determine what kind a member access is. When there is no related entry in field `key`, it can only be a single attribute. Otherwise, it can be either multiple attribute or reference, which can be distinguished through the presence of the corresponding item in map `ref`.

This also applies to member access from transparent identifier, since as described in Section 3.4.3, Table Information Node for anonymous type is also generated, and any eventual reference is added to map `ref`.

### 3.4.5 Result

The result of this translation phase is CST of Ferry expressions, whose string representation is blocks of Ferry expressions. To make this string representation more readable, another group of stringify methods `indentToString` are defined for all kinds of nodes of Ferry syntax. Each of these methods takes an integer argument, specifying how "deep" the current expression should be indented. This method calls the method `indentToString` of its sub-clauses, passing an argument for the depth of one level higher. In our

project, we use string constant “ |” as one level of indentation, hence yielding the result shown in Listing 3.4. With this, it's convenient to match up keywords or phrases that belongs to the same expression.

`from p in Person where p.age>30 select p.name`
is translated into the following block of Ferry expressions, shown in a structural manner. To save space, the column specification in the table reference expression is truncated.

```
|let
|  |_wen_variable_2 =
|  |  |let
|  |  |  |_wen_variable_1 =
|  |  |  |  |table Person (_wen_iter Int,_wen_pos Int,...)
|  |  |  |  |  |with keys (_wen_id)
|  |  |in
|  |  |  |for p in _wen_variable_1
|  |  |  |  |where p.age > 30
|  |  |  |return p
|in
|  |for p in _wen_variable_2
|  |return p.name
```

Listing 3.4: Resulted block of Ferry expressions

## 3.5 Next step

The `ferryc` compiler for Ferry has been implemented by the database team at University of Tübingen, which takes Ferry expressions as input and generates equivalent expressions in relational algebra after optimization[12]. We do not repeat this work again. Rather, we have implemented the translation from relational algebra to SQL:1999 statements, as described in Chapter 4.

# Chapter 4

# Relational Query Plans

## 4.1 Relational Algebra

As the destination language of Ferry, a dialect of classical relational algebra is adopted. It is said to be a dialect because the primitive operators in the chosen relational algebra are not exactly conform to those of classical relational algebra[6]. These operators are chosen in a way that, no significant lost of expressive power in terms of the SQL statements will be shown despite the removal of some primitive operators in classical relational algebra(for example, by excluding `rename` operator), and that with this relational algebra, some extra convenience can be obtained when processing a complicated query plan(for example, by introducing `aggregate`, `application operator`, `attach` and `row rank` operators). The operators in the relational algebra of interest are `Projection`, `Selection`, `Cartesian Product`, `Equi-Join`, `Disjoint Union`, `Difference`, `Distinct`, `Attach`, `Row Rank`, `Row Number`, `Aggregation` and `Application Operators`. In the following subsections, each of these operators will be explained in detail with their meanings, prerequisites, examples as well as what SQL statements they will be transformed into.

### 4.1.1 Projection ($\pi_{a_1:b_1,\ldots,a_n:b_n}$)

**Description:**
Projection is an operator that results in a limited set of columns $b_1, \ldots, b_1$ from a relation $\mathsf{R}$, and all other columns that are not presented in the set are excluded. If the argument $a_1, \ldots, a_1$ present, corresponding columns in the result will be renamed as specified in $a_n$.

**Prerequisite:**
All elements in set $b_1, \ldots, b_n$ should be contained in the schema of $\mathsf{R}$ (short for $\mathit{sch}(\mathsf{R})$). That is, $[b_1, \ldots, b_n] \subseteq \mathit{sch}(\mathsf{R})$.

**Example:**

$\pi_{a_1:b_1,a_2:b_2,b_3}$

| $b_1$ | $b_2$ | $b_3$ | $b_4$ |
|---|---|---|---|
| 1 | "a" | 4.5 | true |
| 2 | "b" | 5.5 | false |
| 3 | "c" | 6.5 | true |

$\Longrightarrow$

| $a_1$ | $a_2$ | $b_3$ |
|---|---|---|
| 1 | "a" | 4.5 |
| 2 | "b" | 5.5 |
| 3 | "c" | 6.5 |

Figure 4.1: Example: Projection Operator

In this example, columns $b_1$ and $b_2$ are projected and renamed to $a_1$ and $a_2$, column $b_3$ is projected without renamed, and column $b_4$ is excluded in the result.

**SQL:** The projection operator will be translated into the following SQL statement. Note that `New_Table` is the name for a temporary table representing the result of the operator. Detail about how to generate this name will be explained in Section 4.3.

```
WITH New_Table (a1,...,an) AS
(SELECT b1 AS a1, ..., bn AS an FROM R)
```

Listing 4.1: SQL statements for Projection Operator

### 4.1.2 Selection ($\sigma_p(\mathsf{R})$)

**Description:**
Selection is a unary operator with a predicate $p$. This operator selects all tuples in R for which predicate $p$ holds. Selection operator is sometimes called restriction, in order to avoid confusion with the `SELECT` clause in SQL context.

**Prerequisite:**
Predicate $p$ should be a propositional formula that only consists of elements in $sch(\mathsf{R})$ and/or constants.

**Example:**

In this example, only employees who's ages are larger than 30 are selected from the relation EMPLOYEE.
**SQL:** The selection operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in $sch(\mathsf{R})$.

$\sigma_{Age>30}(\mathsf{EMPLOYEES})$

| ID | Name | Age |
|---|---|---|
| 1 | Tom | 25 |
| 2 | Bill | 35 |
| 3 | Kate | 45 |

$\Longrightarrow$

| ID | Name | Age |
|---|---|---|
| 2 | Bill | 35 |
| 3 | Kate | 45 |

Figure 4.2: Example: Selection Operator

```
WITH New_Table (a1,...,an) AS
(SELECT (a1,...,an) FROM R WHERE p)
```

Listing 4.2: SQL statements for Select Operator

### 4.1.3 Cartesian Product ($\mathsf{R} \times \mathsf{S}$)

**Description:**
Cartesian Product is a binary operator that takes two relations R and S as its arguments. The resulting relation $\mathsf{T} = \mathsf{R} \times \mathsf{S}$ is a set of all possible combinations of ordered pairs, whose first component comes from relation R and the second component from relation S. In this context, *set* does not imply any elimination of possible duplicate result, and *component* refers to an atomic tuple(row) from a relation.

**Prerequisite:**
In order that the Cartesian product $\mathsf{T} = \mathsf{R} \times \mathsf{S}$ is well-formed, relation R and relation S must be disjoint. That is, for any column c in R, there should not exist any column d in S such that c and d have the same name.

**Example:**

$\mathsf{T} = \mathsf{R} \times \mathsf{S}$

| A | B |
|---|---|
| 1 | "a" |
| 2 | "b" |

$\times$

| C | D |
|---|---|
| 3 | "c" |
| 4 | "d" |
| 5 | "e" |

$=$

| A | B | C | D |
|---|---|---|---|
| 1 | "a" | 3 | "c" |
| 1 | "a" | 4 | "d" |
| 1 | "a" | 5 | "e" |
| 2 | "b" | 3 | "c" |
| 2 | "b" | 3 | "d" |
| 2 | "b" | 3 | "e" |

Figure 4.3: Example: Cartesian Product Operator

In this example, all possible combinations from relation R and relation S are presented in the resulting relation.

**SQL:** The Cartesian Product operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in $sch(R)$, and $b_1, \ldots, b_n$ represent all elements in $sch(S)$.

```
WITH New_Table (a1,...,an, b1,...,bn) AS
(SELECT * FROM R,S)
```

Listing 4.3: SQL statements for Cartesian Product Operator

### 4.1.4 Equi Join ($R \bowtie_{a=b} S$)

**Description:**
Equi Join is a binary operator that takes two relations R and S as its arguments. The resulting relation $T = R \bowtie S$ is a set of all possible combinations of odered pairs, in that a specified column a in relations R must equal another specified column b in S.

Although Equi Join is a special case of Theta Join, where the predication to connect both relations can be other than equality, Equi Join has still be taken as one of the primitive operators. Other kinds of join-like operations can be achieved by combining Equi Join with other operator such as Selection, so as to further specify a more sophisticated prediction.

**Prerequisite:**
In order that the Equi Join $T = R \bowtie S$ is well-formed, relation R and relation S must be disjoint. That is, for any column c in R, there should not exist any column d in S such that c and d have the same name. Further more, column c and d must be of the same data type. This second restriction can be achieved in our implementation, since in our data structure representing the schema of a relation, not only the names of the columns but also their data types are specified. Detail about the implementation will be explained in Section 4.3.

**Example:**

In this example, relation Employee and relation Department are joined together through their common columns `DID` and `DepID`. Usually, it suffices to present only one of these common columns in the result relation, which can be achieved by a projection operation afterwards.

**SQL:** The Equi Join operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in $sch(R)$, and $b_1, \ldots, b_n$ represent all elements in $sch(S)$. $a_o$ and $b_p$ are the common columns from

Employee $\bowtie_{\mathrm{DID=DepID}}$ Department

| EmpID | EmpName | DID |
|---|---|---|
| 1 | “Tom” | “1001" |
| 2 | “Bill” | “1002" |
| 3 | “Kate” | “1001" |

$\bowtie$

| DepID | DepName |
|---|---|
| 1001 | “IT” |
| 1002 | “Sale” |

=

| EmpID | EmpName | DID | DepID | DepName |
|---|---|---|---|---|
| 1 | “Tom” | 1001 | 1001 | “IT” |
| 2 | “Bill” | 1002 | 1002 | “Sale” |
| 3 | “Kate” | 1001 | 1001 | “IT” |

Figure 4.4: Example: Equi Join Operator

both relations that perform the actual join operation.

```
WITH New_Table (a1,...,an, b1,...,bn) AS
(SELECT R.a1, ... , R.an , S.b1 , ... , S.bn
 FROM R,S
 WHERE R.ao = S.bp
)
```

Listing 4.4: SQL statements for Equi Join Operator

### 4.1.5 Disjoint Union ($\mathsf{R} \cup \mathsf{S}$)

**Description:**
Disjoint Union is a binary operator that connect both relations from argument into one relation. This operation does not eliminate any possible duplicated tuples in the resulting relation.

**Prerequisite:**
In order that the Disjoint Union $\mathsf{T} = \mathsf{R} \cup \mathsf{S}$ is well-formed, relation $\mathsf{R}$ and relation $\mathsf{S}$ must be compatible. That means, $\mathit{sch}(\mathsf{R})$ and $\mathit{sch}(\mathsf{S})$ must have identical numbers of columns, and each corresponding columns should have identical names as well as types.

**Example:**

In this example, relation $\mathsf{R}$ and relation $\mathsf{S}$ are connected vertically to produce a new relation, which contains all tuples from relation $\mathsf{R}$ and all tuples from relation $\mathsf{S}$.

**SQL:** The Disjoint Union operator will be translated into the following SQL statement, where $\mathrm{a}_1, \ldots, \mathrm{a}_\mathrm{n}$ represent all elements in $\mathit{sch}(\mathsf{R})$. They are

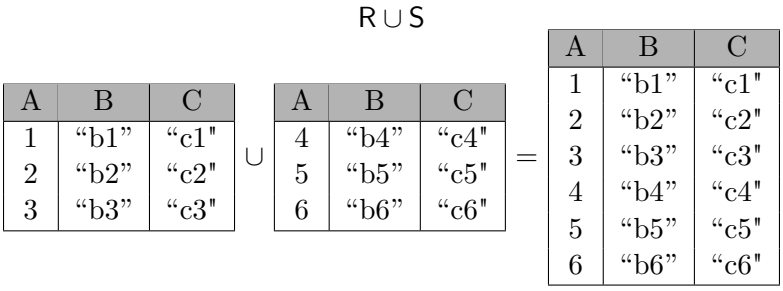



Figure 4.5: Example: Disjoint Union Operator

also all elements from $sch(S)$ since both relations have identical schema.

```
WITH New_Table (a1,...,an) AS
(
 SELECT * FROM R
   UNION ALL
 SELECT * FROM S
)
```

Listing 4.5: SQL statements for Disjoint Union Operator

### 4.1.6 Difference (R\S)

**Description:**
Difference is a binary operator that produces the difference between the first operand R and the second operand S. The resulting relation T contains all tuples from relation R, but excludes all that are contained in relation S.

**Prerequisite:**
In order that the Difference operation $T = R \backslash S$ is well-formed, relation R and relation S must be compatible. That means, $sch(R)$ and $sch(S)$ must have identical numbers of columns, and each corresponding columns should have identical names as well as types.

Compared with Disjoint Union operator, Difference operator also requires that the first (left) operand does not contain any duplicated tuples. The resulting relation hence does not contain any duplicated tuples as well.

**Example:**

In this example, the resulting relation contains elements that are only in relation R but not in relation S.

**SQL:** The Difference operator will be translated into the following SQL

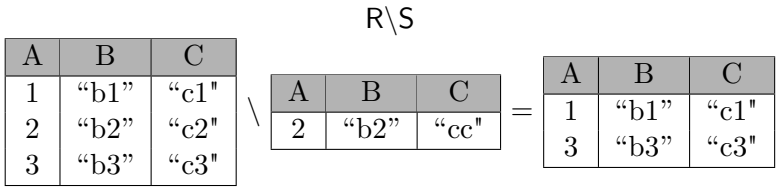



Figure 4.6: Example: Difference Operator

statement, where $a_1, \ldots, a_n$ represent all elements in $sch(R)$. They are also all elements from $sch(S)$ since both relations have identical schema.

```
WITH New_Table (a1,...,an) AS
(
 SELECT * FROM R
 WHERE NOT EXISTS
   (SELECT * FROM S WHERE R.a1 = S.a1, ..., R.an = S.an)
)
```

Listing 4.6: SQL statements for Difference Operator

### 4.1.7 Distinct ($\delta_R$)

**Description:**
Distinct is a unary operator that eliminates any possible duplicated tuples from the relation R in argument.

**Prerequisite:**
The Distinct operator does not have any prerequisite in order to be well-formed. If the input relation R does not contain any duplicated tuple, the output relation remains the same as the input.

**Example:**

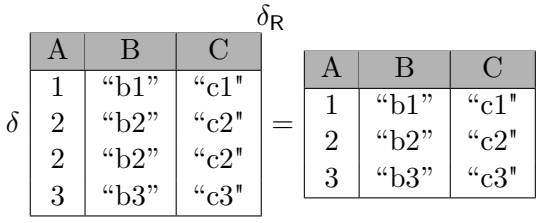



Figure 4.7: Example: Distinct Operator

In this example, the duplicated tuple is eliminated.

**SQL:** The Difference operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in $sch(R)$.

```
WITH New_Table (a1,...,an) AS
(
 SELECT DISTINCT * FROM R
)
```

Listing 4.7: SQL statements for Distinct Operator

### 4.1.8 Attach (@)

**Description:**
Attach operator is a unary operator that attaches to a relation R an additional column $a$ with constant value $v$.

This operator has the identical result as a Cartesian Product operation, having relation R as the first operand, and a temporary relation that contains only one column a with value v as the second operand. However, the Attach operation is more efficient than its corresponding Cartesian Product operation, in that instead of performing any actual production operation, each tuple in the input relation merely needs to be extended by a constant value to produce the result relation.

**Prerequisite:**
The Attach operator does not have any prerequisite in order to be well-formed. However, because of the set nature of a valid relational schema, the new column being attached should not have the same name as that of any columns in the input relation.

**Example:**

$@_{R,c:v}$

| A | B |
|---|---|
| 1 | "b1" |
| 2 | "b2" |
| 3 | "b3" |

@

| C |
|---|
| v |

=

| A | B | C |
|---|---|---|
| 1 | "b1" | v |
| 2 | "b2" | v |
| 3 | "b3" | v |

Figure 4.8: Example: Attach Operator

In this example, the new column c is attached to the input relation, and the value v is attached to each column.

**SQL:** The Attach operator will be translated into the following SQL state-

ment, where $a_1, \ldots, a_n$ represent all elements in $sch(R)$.

```
WITH New_Table (a1,...,an,c) AS
(
 SELECT R.a1,...,R.a2, v AS c FROM R
)
```

Listing 4.8: SQL statements for Attach Operator

### 4.1.9 Row Rank ($\varrho_{c,a}$)

**Description:**
Row Rank operator is a unary operator that attaches to a relation R an additional column $a$, whose value is the order of the corresponding tuple within the relation. The argument $c$ specifies the column name, by which the ordering takes place.

If more than one tuples have the same value on the specified column, the same value for Row Rank will be assigned. The following tuple will be assigned a continuous row rank. Therefore, it may be possible that the row rank value of the last tuple may be smaller than the total number of tuples.

**Example:**

$\varrho_{B,RANK}$

| | A | B |
|---|---|---|
| | 1 | 10 |
| $\varrho$ | 2 | 20 |
| | 3 | 20 |
| | 4 | 30 |

$=$

| A | B | RANK |
|---|---|---|
| 1 | 10 | 1 |
| 2 | 20 | 2 |
| 3 | 20 | 2 |
| 4 | 30 | 3 |

Figure 4.9: Example: Row Rank Operator

In this example, due to the duplicated values in column B, the row rank value has only been increased to 3 despite that the input relation contains 4 tuples.

**SQL:** The Row Rank operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in $sch(R)$.

```
WITH New_Table (a1,...,an,A) AS
(
  SELECT R.a1,...R.an, DENSE_RANK() OVER (ORDER BY c) AS A
  FROM R
```

```
)
```

Listing 4.9: SQL statements for Row Rank Operator

### 4.1.10 Row Number ($\#_{a:<b_1,...,b_n>/c}$)

**Description:**
Row Number operator is a unary operator that attaches to a relation R an additional column $a$, whose value is the order of the corresponding tuple within the relation. The argument $b_1, \ldots, b_n$ specifies the column for partitioning(grouping). For each new partition, the result starts from 1. Optional argument c specifies the key of the ordering. If this argument is absent, default oder of the tuples will be adopted.

Different value for Row Number will be assigned, even though more than one tuples within one partition may have the same value on the specified column. Therefore, contrary to how the Row Rank operator behaves, the Row Number value of the last tuple is the same as the total number of tuples within the same partition.

**Example:**

$$\#_{ROW\_NUM:<B_1,...,b_4>/A}$$

| A | B |
|---|---|
| 1 | “aa” |
| 2 | “bb” |
| 3 | “aa” |
| 4 | “bb” |

$\Longrightarrow$

| A | B | ROW_NUM |
|---|---|---|
| 1 | “aa” | 1 |
| 3 | “aa” | 2 |
| 2 | “bb” | 1 |
| 4 | “bb” | 2 |

Figure 4.10: Example: Row Number Operator

In this example, values of Row Number operator are reset between different partitions(Column `B` here).

**SQL:** The Row Number operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in *sch*(R), and `ORDER BY` clause within square brackets is optional.

```
WITH New_Table (a1,...,an,A) AS
(
  SELECT R.a1,...R.an, ROW_NUMBER() OVER (PARTITION BY b [ORDER
      BY c]) AS A
  FROM R
)
```

Listing 4.10: SQL statements for Row Number Operator

### 4.1.11 Aggregation ($agg_{a,b,c}$)

**Description:**
Aggregation operator is an operator that attaches to a relation R an additional column $c$, whose value is the result of the application of one of the aggregation operators $agg$ on column $a$, by aggregating (grouping) the values in column $b$. Column `b` specifies the partitions of tuples to which the aggregation function applies to.

Aggregation function `agg` is defined as:

$$agg \in \{average, max, min, sum, count\}$$

Details about implementation of these operators will be provided in Section 4.3.

**Example:**

$\varrho_{MAX_{AGE,DEP,MAX\_AGE}}$

| ID | NAME | DEP | AGE |
|---|---|---|---|
| 1 | "TOM" | "IT" | 30 |
| 2 | "BILL" | "SALE" | 35 |
| 3 | "KATE" | "SALE" | 27 |
| 4 | "JIM" | "IT" | 32 |

=

| DEP | MAX_AGE |
|---|---|
| "IT" | 32 |
| "SALE" | 35 |

Figure 4.11: Example: Aggregation Operator

This example is to find out the maximal age of employees in each department. The function `Max` is applied to the column `AGE` by grouping to column `DEP`, hence yielding the the desired result in column `MAX_AGE`.

**SQL:** The Aggregation operator $agg_{a,b,c}$ will be translated into the following SQL statement.

```
WITH New_Table (b, c) AS
(
    SELECT b, agg(a) AS c FROM R
    GROUP BY b
)
```

Listing 4.11: SQL statements for Aggregation Operator

### 4.1.12 Operation-Application ($\otimes$)

**Description:**
Operation-Application operator is an operator that attaches to a relation R an additional column $c$, whose value is the result of the application of an operator $o$, which takes one or two columns as its arguments. The operator o is defined as:

$$o \in \{+, -, *, /, ==, !=, <, >, <=, >=, and, or, not\}$$

According to different operators, the resulting column can be of different types. Further, these operators are divided into two groups: binary operators and unary operators. This is for the convenience of our implementation. Details about implementation of these operators will be provided in Section 4.3.

**Example:**

$+_{A,B,D}$

| A | B | C |
|---|---|---|
| 1 | 10 | "aa" |
| 2 | 20 | "bb" |
| 3 | 30 | "cc" |
| 4 | 40 | "dd" |

$\Longrightarrow$

| A | B | C | D |
|---|---|---|---|
| 1 | 10 | "aa" | 11 |
| 2 | 20 | "bb" | 22 |
| 3 | 30 | "cc" | 33 |
| 4 | 40 | "dd" | 44 |

Figure 4.12: Example: Operation-Application Operator

This example demonstrates that a new column named `D` is added to each column, by summing up column `A` and column `B` in each tuple.

**SQL:** The Operation-Application operator with binary operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all elements in *sch*(R), and $a_o$, $a_p$ are the operands for the binary operator.

```
WITH New_Table (b, c) AS
(
   SELECT a1,...,an, (ao OPER ap) AS c
   FROM R
)
```

Listing 4.12: SQL statements for Operation-Application Operator(binary operator)

The Operation-Application operator with unary operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represent all ele-

ments in $sch(R)$, and $a_o$ is the operand for the unary operator.

```
WITH New_Table (b, c) AS
(
  SELECT a1,...,an, OPER(ao) AS c
  FROM R
)
```

Listing 4.13: SQL statements for Operation-Application Operator(unary operator)

### 4.1.13 Table Reference

**Description:**
Table Reference Operator represents the access to a persisted relation R in a Database Management System(DBMS).

**SQL:** The Table Reference operator will be translated into the following SQL statement, where R is the relation being referenced, and $a_1, \ldots, a_n$ represents all columns in $sch(R)$.

```
WITH New_Table (a1,...an) AS
(
  SELECT a1,...,an FROM R
)
```

Listing 4.14: SQL statements for Table Reference Operator

### 4.1.14 Table Literal

**Description:**
Table Literal Operator represents a new relation that is generated in runtime. Also the rows of this relation are generated in runtime.

**SQL:** The Table Literal operator will be translated into the following SQL statement, where $a_1, \ldots, a_n$ represents all columns in the new relation, and $v_{1,1}, \ldots, v_{n,n}$ represents the rows generated and inserted to the relation.

Note also that since we are going to ship our query into IBM DB2, one special table called SYSIBM.SYSDUMMY1 is being used. This table is used for SQL statements in which a table reference is required, but the contents of the table are not important. Therefore it's used in our translation here as a place-holder for the *FROM* clause in order to get a well-formed SQL statement.

```
WITH New_Table (a1,...an) AS
(
  ( SELECT v1,1 AS a1, ... , v1,n AS an
      FROM sysibm.sysdummy1
  )
    UNION ALL
  ( SELECT v2,1 AS a1, ... , v2,n AS an
    FROM sysibm.sysdummy1
  )
    ...
    UNION ALL
  ( SELECT vn,1 AS a1, ... , vn,n AS an
    FROM sysibm.sysdummy1
  )
)
```

Listing 4.15: SQL statements for Table Literal Operator

## 4.2 Query Plan

A query plan is a sequence of steps used to perform a query on a DBMS. A particular query may be executed with different query plans, with different performance and efficiency. Query plan provides a way to examine the actual execution process of a query in a lower-level view, and hence provides the possibility to fine-tune a query through the adoption of different query plans.

In our work, query plan is also used to illustrate the factorization of a complicated query into primitive operators. For example, given that we have two relation schemas *sch*(CUSTOMERS) and *sch*(ORDERS).

CUSTOMERS

| CID | CNAME | CORDER_ID |
|---|---|---|
| 1001 | "TOM" | 2002 |
| 1002 | "BILL" | 2003 |
| 1003 | "JIM" | 2001 |

ORDERS

| OID | ONAME | OPRICE |
|---|---|---|
| 2001 | "IPHONE" | 179 |
| 2002 | "IPOD" | 89 |
| 2003 | "WIN7_PRO" | 119 |

Figure 4.13: Example: relations CUSTOMERS and ORDERS

In order to find out the names of the customers and the names of the products they ordered, one would compose the SQL statement in Listing 4.16.

```
SELECT customers.NAME, orders.NAME
FROM customers, orders
WHERE customers.ORDER_ID = orders.ID
```

Listing 4.16: SQL statements to find out names of customers and products

This query can be factorized into a query plan with three steps of primitive operators: first Table Reference, and then Equi Join, and finally Projection, as illustrated in Figure4.14.

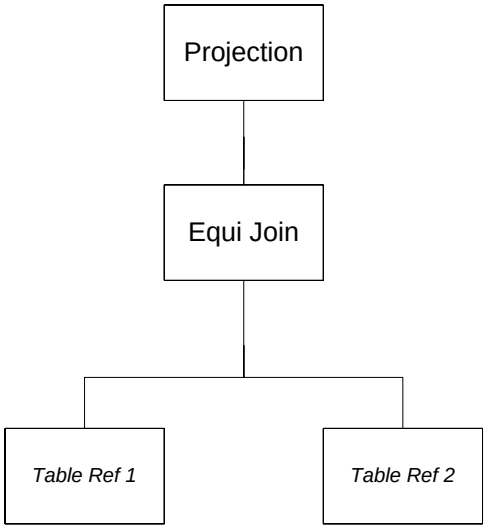



Figure 4.14: Tree Diagram of Query Plan

Each of these operators are translated to its corresponding SQL statement. In order to take the output of preceding step as the input of the following step, *WITH* statement is used. Each clause within a WITH statement generates a temporary relation as its result.[13] These temporary relations are visible and can be referenced by further clauses afterwards. To identifier these temporary relations, names are assigned at runtime. Detail about how to assign to them valid and conflict-free names will be described in Section 4.3.

Listing 4.17 shows how this query is factorized and cascaded into several individual SQL statements corresponding to one of the primitive operators, and then connected with a `WITH` statement.

```
WITH
-- Table Reference: Customers
New_Table001(CID, CNAME, ORDER_ID) AS (
   SELECT * FROM CUSTOMERS
) ,
```

```
-- Table Reference: Orders
New_Table002(OID, ONAME, OPRICE) AS(
   SELECT * FROM ORDERS
) ,
-- Equi Join on New_Table001.ORDER_ID and New_Table002.OID
New_Table003(CID, CNAME, ORDER_ID, OID, ONAME, OPRICE) AS(
   SELECT * FROM New_Table001, New_Table002
   WHERE New_Table001.ORDER_ID = New_Table002.OID
)
-- Projection to get the desired columns
SELECT CNAME, ONAME FROM New_Table003
```

Listing 4.17: factorized SQL statements with `WITH`

## 4.3 Implementation

### 4.3.1 Classes of ASTs

To implement Relation Algebra in Scala, we first design the class hierarchy to represent the operators and the relations between them. Although these classes are named after the operators, they actually represent the nodes in query plans. Therefore, each class has a field `schema`, representing the schema of the relation that this operator results in. Also, different kinds of operators may require different number of operands, and these operands are children of the operator node in the tree of query plan.

The class hierarchy is shown in Figure 4.15 and each class has a self-explaining name. Class `RAExpr` is the super class for all operators. Although it is an abstract class, it contains a field `schema` of type `List[Column]`. Type `column` represents a column in a schema, containing a field `cName`, representing the name of a column, and a field `cType`, representing the type of that column. The latter field is of Type `Type`, an enumeration type containing all the possible data types in SQL:99 standard.

Each operator takes one or more operators as its children, implying that the operands can either be a table(either Table Reference or Table Literal) or a subtree of operands. Class `Table` is also an abstract class that generalizes class `TableRef` and class `TableLit`.

The operators are declared as *case classes*. In Scala, a class declared with the *case* modifier has some syntactic conveniences. First, the Scala compiler adds automatically a factory method with the name of the class, meaning that one can construct an object of this class without the keyword `new`. For example, instead of having a sequence of `new` keywords, one can construct an object of type `TableRef` in a way shown in Listing 4.18. This is a more intuitive way to instantiate a tree-shaped construct.

```
1 val employee = TableRef("emp",
```

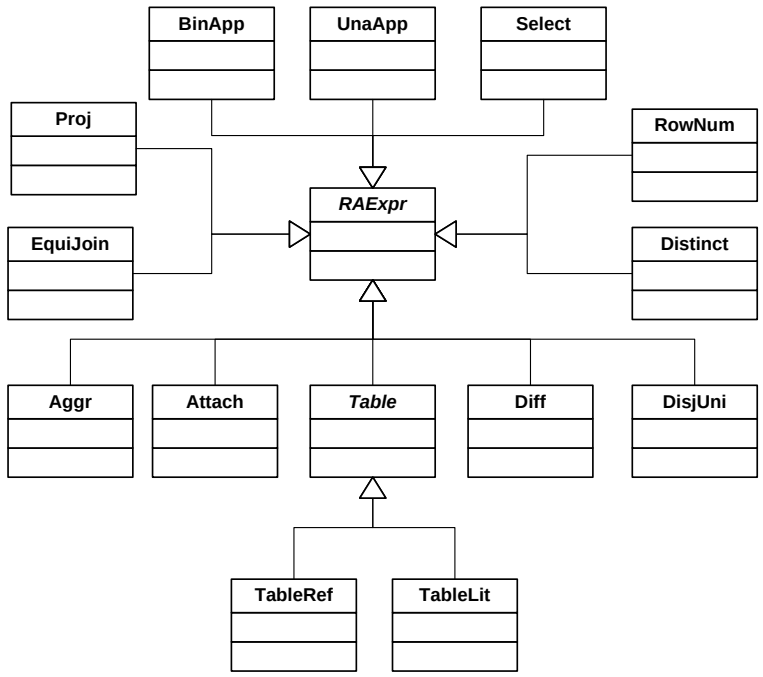



Figure 4.15: UML Diagram of Classes Representing Relational Algebra

```
2       List[Column]
3       (Column("eId",Type.VARCHAR),Column("eName",Type.VARCHAR),Column
            ("e_dId",Type.VARCHAR)
4     )
5 )
```

Listing 4.18: Constructing an object of case class

Other conveniences include that all arguments in the parameter list of a case class are implicitly treated as fields, and the compiler also adds the implementations of methods `toString`, `hashCode` and `equals` to it. These are helpful especially for tree-shaped object, while children in depth can be processed recursively as well.

In the constructors of these case classes, all fields are specified, including their names, schemas and some special field such as the joining keys for Equi Join operator, the predicate for Select operator. The schema of an operator can be determined from those of its children, and by some operators also from other arguments. For example, the schema of `Cartesian Product` Operator has the schema that is the combination of its both children, while the schema of `Different` are directly taken from one of its children(both children should have the same schema). The schema of `Projection` operator contains columns that are of the same types as those of its child, but with probably different names due to the renaming. Thanks to the functional

style of Scala, the constructor of `Projection` operator can be written as shown in Listing 4.19.

```
1 case class Proj(left:RAExpr,names:List[NamesPair]) extends RAExpr{
2   val schema = names map (col=>Column(col.newName,
3                          (left.schema find ( x => x.cName == col.
                               oldName)).get.cType
4                       )
5                     )
6 }
```

Listing 4.19: Constructor of Projection Class

The operation `map` takes as operands a list `l` of type `List[T]` and a function `f` of type `T=>U`, and returns a *list* resulting from applying the function `f` to each element of list `l`. In the example in Listing 4.19, list of `NamesPair` are mapped to a list of `Column`, with the new name as value of field `cName`, and the type found with the old name from the old schema as value of field `cType`.

There are also other classes that are defined to assist the implementation. `NamesPair` defines a pair of Strings that are used in Projection operator, reflecting the possible process of renaming of original name in the schema to a new name.

Binary operators and unary operators for operator `Operation Application` are defined as enumeration types. These two groups are defined separately because they need to be transformed into two different style of SQL syntax: binary operators are located between the operands, such as $a + b$, while unary operators are preceding the operand, such as $not(c)$.

### 4.3.2 Transforming to SQL

Having all the classes for Relational Algebra as well as all the transformation rules defined, we are ready to perform the actual transformation from Relational Algebra to SQL statements.

A query plan is a tree-shaped construct, which require a recursive traverse from leaves to root, since the results of the transformation of children are arguments for the transformation of parent node.

For this, a post-ordered visitor is used. An abstract class `Visitor` is defined as the super class for all kinds of implementations, including the transformer to perform the transformation from Relational Algebra to SQL, and the well-formedness checker that visits all nodes of a query plan to check their well-formedness. Detail about the well-formedness checker will be given in subsection 4.3.3.

Further, a walker class is implemented that can navigate the tree recursively and exhaustively. This class has only one public method `walk` that takes an instance of `RAExpr`, which is the root of a tree or a sub tree to

be walked, and an instance of an implementation of `Visitor`, which is the visitor that is being sent to visit each node, as its two arguments.

Listing 4.20 shows part of this class. One can see that in Line 4 the left child of a Projection operator has been visited before the Projection operator itself is visited, taking the result of the previous visit as an argument. In the case of Cartesian Product Operator, both left and right children have been visited first in Line 8 and 9.

```
1  class Walker[T] (v:Visitor[T]) {
2      def walk (node : RAExpr) : T = node match {
3          case proj : Proj => {
4              val resLeft = this walk proj.left
5              v.visit(proj,resLeft,proj.names)
6          }
7          case cart : CartProd => {
8              val resLeft = this walk cart.left
9              val resRight = this walk cart.right
10             v.visit(cart,resLeft,resRight)
11         }
12         ... ...
13     }
14 }
```

Listing 4.20: Snippet of Class `Walker`

Having the abstract class `Visitor` and the Walker ready, we can implement the visitor for the transformation.

A local variable `sql` of type `List[String]` is maintained to save the visiting result from each node. To visit each node, the visiting result of its children(here is the names of the relations), along with the schema information of current node(the schema is already determined by the constructor), are combined to generate the SQL statements according to the transformation rules, and these statements are appended to the variable `sql`. Listing 4.21 shows the method to visit nodes of Equi Join Operators. Method `getQualNames(RAExpr, List[Column]):String` are used to generate string of qualified names of columns in a schema. For instance, for relation R with columns $a_1, \ldots, a_n$, string `R.a1 ,...,R.an` will be generated.

```
1  override def visit(ej:EquiJoin, resLeft:String, resRight:String,
       leftCol:String, rightCol:String) : String = {
2        val name:String = getNextFreeName()
3        comments += "-- " + name + ": EquiJoin(" + resLeft + "," +
             resRight + ")\n"
4        sql += (name + " (" + ej.schema.mkString(", ") + ") AS\n" +
5                    "   (SELECT " + getQualNames(resLeft, ej.left.schema
                         ) + ", " + getQualNames(resRight, ej.right.
                         schema) +
6                  " FROM " + resLeft + ", " + resRight +
```

```
7                " WHERE " + resLeft + "." + leftCol + " = " + resRight
                     + "." + rightCol + ")")
8        return name
9      }
```

Listing 4.21: Method to Visit Equi Join

After the root node has been visited, the variable `sql` has collected all the SQL statements from the query plan. In order to for the final output to be well-formed, this sequence of string should be prefixed with a `WITH` keyword, all string should be separated with comma(,). Last but not least, the table heading of the last SQL statement should be removed. This last step is necessary because by visiting each node, the resulting statement always starts with a table declaration such as `NEW_Table0001 (a1,a2) AS`, which should however not be presented in the final statement, as shown in the last line of Listing 4.17.

Up to now, the transformation from Relational Algebra to SQL statements is finished. The output statements are ready to ship to a DBMS for evaluation.

### 4.3.3 Well-formedness Checking

The well-formedness checker not only checks a query plan to make sure (a) if it conforms to SQL standard, but also checks (b) if the prerequisite for each operator in the Relational Algebra has been met.

For (a) mentioned above, it suffices to have one single checker for all operators. For (b), individual checkers for each operator of Relational Algebra are needed due to the variety of the requirements.

Based on the abstract visitor and the walker we have already had, the well-formedness checker is just another implementation of the visitor. Some operators only have to be checked whether there is any duplication in the schemas. Such operators include Projection, Distinction, Attach, Row Number and Operation Application. Since this kind of checking is performed after any possible additional column has been added to the schema(for example, by Attach Operator), introduction of duplication through appending new column can be avoided.

Two operators require that both operands have disjoint schemas. These operators are Cartesian Product and Equi Join. This check is performed by a utility method that checks if there exists any element in one schema that is contained in another schema. In Scala there is a method `exists` that can perform a predicate over each element in a list. Using this method, this checking can be realized in a relatively concise way, as shown in Listing 4.22.

```
1 def disjoint(list1:List[Column], list2:List[Column]):Boolean =
2    !(list1 exists (col => list2.contains(col) ) )
```

Listing 4.22: Method to Determine Disjoint Schemas

Additionally, Equi Join operator also requires that `leftCol` and `rightCol`, two columns that are specified to be the joining key, should exist in both left and right children. Similar rule also apples to Aggregation operator, where the target column of the aggregating function and the grouping column should both exist in the schema.

Contrarily, two other operators that requires both children to have compatible schemas are Dsijoint Union and Different. This compatibility is also checked by a utility method to see if two schemas are equal in terms of their names and data types of corresponding columns.

## 4.4 Automated Testing

Once the transformer as well as the well-formedness checker are finished, we need to conduct some testing. One of the important aspects of performing test is the generation of test cases that simulate as many possibilities as possible. Because of the tree-shaped characteristic of query plan, manually constructing the test cases is challenging yet difficult to cover a wide spectrum of varieties. What is worse, this method would only make the testing code error-prone and difficult to read. For example, Listing 4.23 shows how to manually construct a query plan with two Table Literals and one Equi Join as its root.

```
1 val employee = TableLit("emp",List[Column(Column("eId",Type.VARCHAR)
    ,Column("eName",Type.VARCHAR),Column("e_dId",Type.VARCHAR)))
2 val department = TableLit("dep",List[Column](new Column("dId",Type.
    VARCHAR),new Column("dName",Type.VARCHAR)))
3 val ej = EquiJoin(employee, department, "e_dId","dId")
```

Listing 4.23: Manually Constructed Query Plan

One can imagine that how difficult it can be to manually generate more sophisticated and complicated test cases, which are however necessary. Therefore, a mechanism to generate test cases automatically is desired.

Scala Check[1] is exactly what we need here. With Scala Check, one can obtain test cases with randomly-generated values(or as desired, within a specific range) for different types of identifiers, constants or containers. When generating tree-shape construct, one can pick randomly one of the generators for each node. By having instances of case classes generated, we can generate fully random yet customized query plan trees as test cases as many and complicated as we need.

[1] http://code.google.com/p/scalacheck/

Scala Check is a complicated framework for unit testing, and a detailed introduction to it is beyond the scope of this thesis. Instead, two examples will be given here to explained how Scala Check is used in our work: one is the generator for a node of type `BinApp`, as shown in Listing 4.24, and the other is the generator to construct the root node, as shown in Listing 4.25.

```
1 private def genBinApp:Gen[BinApp] = {
2   val left = getChild(Gen.choose(3,10).sample.get)
3   val oper = (BinAppOper((Gen.choose(0,BinAppOper.maxId - 1)).sample.
        get))
4   for {
5     val operand1 <- Gen.pick(1, left.schema map (_.cName))
6     val operand2 <- Gen.pick(1, left.schema map (_.cName))
7     val name <- for (n<-Gen.identifier) yield n.take(10)
8   } yield BinApp(left, oper,  operand1.toList.head, operand2.toList.
        head, name)
9 }
```

Listing 4.24: Generator for Node of Type `BinApp`

In Line 2 of Listing 4.24, the left operand of a node `BinApp` is obtained by calling method `getChild`, which randomly picks one generator of an operator(which can also be a Table). Therefore, the left operand of the current node can be either an arbitrary operator as a subtree, or a table as a leaf. In our implementation, method `getChild` can return table only, once the total number of non-table operators generated so far exceeds a threshold specified by the user. This can limit the maximal depth of the query plan tree generated. In Line 3, one of the binary function is chosen. In Line 5 and Line 6, both operands are picked up from the schema of left child(by picking 1 element from all the cName fields of the schema of left child). In Line 7, a random identifier with 10 digits is generated to be the name of the new column. In Line 8, all the elements generated above are combined to achieve the final node of type `BinApp`.

```
1 private def genRoot(maxDepth:Int) :Gen[RAExpr] = {
2   this.maxDepth = maxDepth
3   val col:Int = Gen.choose(1,10).sample.get
4   return Gen.oneOf(genProj, genEquiJoin, genDisjUni, genDiff,
        genAttach, genRowRank, genRowNum, genBinApp, genUnaApp,
        genAggr)
5 }
```

Listing 4.25: Generator for Root Node

In the same way as implementing the generator for `BinApp`, we can have generators for all other types of nodes. Once they are ready, method `genRoot` can generate the root node by choosing one of these generators, as shown in Line 4 in Listing 4.25. As long as the root node is generated, its child or

children are also generated automatically, ending up with tables as leaves. In this way, a query plan tree is completely randomly generated as our test case.

# Chapter 5

# Isomorphism

In this chapter, we will argue the preservation of isomorphism in both persistence phase and query phase.

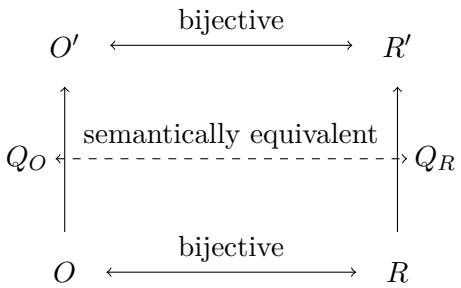



Figure 5.1: Preservation of isomorphism

Consider Figure 5.1. Given an object model $O$, it is persisted to yield its corresponding relational model $R$. The bijection between $O$ and $R$ implies that a particular $O$ can only be mapped to one $R$, and a particular $R$ can only be mapped from one $O$. We can perform different operations on $O$ and $R$, which in our project are queries $Q_O$ and $Q_R$, respectively. If queries $Q_O$ and $Q_R$ are semantically equivalent, we can conclude that their results $O'$ and $R'$ are also bijective.

In Section 5.1, we will explain the bijection between $O$ and $R$ due to persistence. In Section 5.2, we will discuss about keeping the semantic equivalence between queries $Q_O$ and $Q_R$.

## 5.1 Isomorphism between object model and relational model

In order to justify the isomorphism between object model and relational model, we adopt a compositional way: we sum up different aspects and

characteristics of object models as "features", and investigate if all of these features are preserved in their corresponding relational models. The preservation of all these features implies the preservation of the composition of them. Therefore, the size of this set of features determines whether the isomorphism between object model and relational model holds.

For object model, we define this set as including these features:

1. Data types
2. Object-oriented characteristics(including attribute, reference and the multiplicity of them)
3. Hierarchy of object population(including inheritance, interface and polymorphism)

This covers most static features of object model, which is also our supported EMF subset.

1. Data types

Here we are referring to primitive types. Defined by EMF, fields of primitive types are declared as attributes. References, instead, point only to classes. All classes can declare and be factored into attributes and references, and the latter can be further factored until eventually there are only attributes left. Therefore, in order to argue the isomorphism of data types, it suffices to argue that of primitive types between object model and relational model. The relation and combination of them, which form Point *2* and Point *3* mentioned above, is another level of isomorphism that we will discuss later.

In particular, primitive types include `Int`, `Double`, `String` and `Boolean`. Except `Double`, each of other primitive types has its direct counterpart as Ferry's atomic type.(At the time of writing, support to `Double` is absent in Ferry, but it could have been implemented in a similar way as `Int`.) A primitive value of one of these types in object model can only be translated to a value of its corresponding type in relational model without ambiguity, and also a primitive value in relational model can only be translated from a value of that type in object model. Therefore we can conclude that data types between object model and relation model are bijective. Of course we assume that there is no difference between both models regarding numeric precision, and hence no overflow would occur due to precision issue.

2. Object-oriented characteristics

This category of features includes attribute, reference and multiplicity of them. Combining them, we have single attribute, multiple attribute, single reference and multiple reference in EMF model.

These four kinds of fields are persisted in four different ways. Value of single attribute is directly stored in the table of its containing class. Values of multiple attributes are stored in a bridging table, and a surrogate value representing them is stored in the table of containing class. For (single or multiple) reference, surrogate value(s) representing the referred object(s) are stored in a similar bridging table, and another surrogate value for them is stored in the table of containing class.

On the other hand, each individual layout in the resulting tables can only be mapped from one of these kinds of fields. In particular, the direct storage of value in a table can only be mapped from single attribute. A surrogate value pointing to a bridging table with actual values stored in it can only be mapped from multiple attribute. A surrogate value pointing to a bridging table, which in turn contains surrogate values pointing to table of other classes can only be resulted from the persistence of reference. The multiplicity of values in this bridging table further distinguishes the cases of single reference or multiple reference.

Therefore, these features of object orientation can be persisted in the relational model without ambiguity, and the result can only be produced uniquely from these features. We can conclude that, regarding this category of features, the encodings between object model and relational model are bijective.

3. Hierarchy of object population

Another category of features of which we aim at keeping isomorphism include inheritance, interface and polymorphism.

Similarly, they are persisted in different ways. Recall the example shown in Figure 2.11 on page 24. Through the presence of columns `SuperCls`, `SuperObj`, `SubCls` and `SubObj` in every tables, each object can be persisted in one or more tables, based on whether its class inherits from another class. This mapping is unique because each class can only have at most one super-class, and values for columns `SuperObj` and `SubObj` are taken from the ID of rows which are also unique. On the other hand, based on the information provided by these columns, an object can be reconstructed uniquely from its encoding in relational model. In the case of interface, surrogate value for each object points to the bridging table for interfaces, which in turn contain surrogate values pointing to tables representing the implemented interfaces. This layout in relational model can only be achieved by the persistence of interfaces, it provides enough information for recovering the original object in object model regarding interfaces.

Polymorphism in object model involves the access of fields declared in other class-frames, which are super-class and interfaces. This is a runtime issue in query phase. Nevertheless, the isomorphic persistence of inheritance and interfaces does enable a success and unique orientation of these

fields across different classes and interfaces, hence achieving a isomorphic polymorphism.

Based on these, we can also conclude that the features of inheritance, interface and polymorphism are isomorphic between object model and relational model.

## 5.2 Semantic equivalence of queries on different models

To prove the semantic equivalence of queries, one possible method would be to expand them as query plans and examine each atomic operator within these plans. However, we cannot adopt this method because we have no access to the detailed implementation of both query languages, and hence the query execution plans are unknown to us. In fact, LINQ and Ferry themselves can be seen as implementation-independent languages, implying that we can instead consider both kinds of queries as *black boxes*, and only concern about what they produce based on what they consume, as long as the queries adhere to their well-defined grammar(both well-formedness of syntax and typing). Further, we again adopt the compositional way, in that semantic equivalence of individual query operators can imply the semantic equivalence of the combination of them. This can be ensured by the non-destructive nature of both LINQ and Ferry.

In Section 3.3.1 *Translation rules*, we have explained the semantics of both SQO methods in LINQ and the resulting Ferry expressions. Here we will further examine the semantic equivalence of both languages regarding the input and output as well as the types of them.

- `Select(src, v=>prj)`

Method `Select` consumes a sequence `src` of type `IEnumerable<TSource>` and returns a sequence of type `TResult`. The resulting Ferry expression takes the translation result of `src` as input, which is of type `IENumerable<TSource>`, and produces another sequence of type `TResult`, which is translated from projection `prj`.

- `Where(src, v=>pred)`

Method `Where` consumes a sequence `src` of type `IEnumerable<TSource>`. The filtering only excludes out items that does not fulfill the predicate, therefore this method returns a sequence of the same type as source. The resulting Ferry expressions takes the translation result of `src` as input, which is of type `IENumerable<TSource>`, and returns another sequence of the same type by excluding unsatisfying items.

- `SelectMany(src, v=>collSel, (v,col)=>resSel)`

Method `SelectMany` consumes a sequence `src` of type `IEnumerable<TSource>` and returns a sequence of type `TResult`. The resulting Ferry expression takes the translation result of `src` as input, which is of type `IENumerable<TSource>`, and returns another sequence of type `TResult`, which is the translation result of projector `resSel`, by concatenating the translation results of a inner loop.

- `OrderByMultiKey(src, v=>key`$_1$`, dir`$_1$`, ..., v=>key`$_n$`, dir`$_n$

Method `OrderByMultiKey` consumes a sequence `src` of type `IEnumerable<TSource>`, and returns another sequence of the same type in a specific order. The resulting Ferry expression takes the translation result of `src` of type `IEnumerable<TSource>` as input. The result is another sequence with items directly taken from the input in a specific order. The sorting does not modify the type of the result, which is hence the same as that of the input.

- `GroupBy(src, v=>key, v=>eleSel)`

Method `GroupBy` consumes a sequence `src` of type `IEnumerable<TSource>`, and returns another sequence of type `TElement`, specified by `eleSel`. The resulting Ferry expression takes the translation result of `src` of type `IEnumerable<TSource>` as input. The result is another sequence with items directly taken from the input, clustered by `key`, followed by a projection. The clustering does not modify the type of the source. The projection is translated into a sequence of type `TElement`, which is the final result of this method.

- `Join(outer, inner, o=>okey, i=>ikey, (o, i)=>resSel)`

- `GroupJoin(outer, inner, o=>okey, i=>ikey, (o, into)=>resSel)`

Both methods `Join` and `GroupJoin` take two sequences of types `IEnumerable<TOuter>` and `IEnumerable<TInner>` respectively and produce another sequence of type `TResult`, type of result selector `resSel`. The resulting Ferry expressions of both methods differ in handling the inner loop, but in both cases they consume two sequence of types `IEnumerable<TOuter>` and `IEnumerable<TInner>`, and return the final sequence of type `resSel`, which is the translation result of the result selector.

Summing up the above discussion, we can conclude that the LINQ query $Q_O$ and the resulting Ferry expression $Q_R$ are semantically equivalent. Recalling Figure 5.1 on page 81, this equivalence plus the bijection of $O$ and $R$ argued in Section 5.1 imply that the query results of them, $O'$ and $R'$, are also bijective.

# Chapter 6

# Conclusions

## 6.1 Contributions

This Master Thesis has shown two kinds of integrations that bring in new innovation to data access scenarios.

On the one hand, the integration of programming language and functional query language provides developers with enhanced expressive power and conciseness(by having functional query in programming language), as well as improved correctness and robustness(by having programming language and its compiler participate in query evaluation).

On the other hand, the integration of different data models takes advantage of each. Through the adoption of intermediate language in the translation, model of representation layer(in this work object model) and model of evaluation and storage layer(in this work relational model) can still be kept transparent to each other.

The implemented prototype showcases that despite the gap between object model and relational model, persistence from the former to the latter can still be performed in a precise manner. It also shows that fully object-oriented queries can be evaluated seamlessly in an object-oriented host language, despite the storage of data in relational model.

Last but not least, this thesis also gives the argument on the preservation of isomorphism on both persistence and query phases.

## 6.2 Future work

1. We have not implemented the translation from Ferry expressions to relation algebra due to the lack of Ferry compiler `ferryc`. Instead, we have skipped this step and implemented the translation from relational algebra to SQL statements. The reimplementation and eventually extension of Ferry(for example, by supporting nested tuples) can further raise its expressive power.

2. Our rules for translation from LINQ to Ferry are rather semantic-oriented. This strict adhering to original LINQ semantics in fact results in some optimization spaces in the resulting Ferry expressions. Therefore the Ferry optimizer can also participate in our setting to achieve better evaluation performance.

3. Our implementation carries out query in LINQ. Nevertheless, the underlying techniques reported here are generally applicable, and other functional query language can also be supported in a similar way.

4. Further, with the similar motivation, translation from other data model (for example, XML)[14] to relational model are also feasible.

# Bibliography


[1] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 877–888, New York, NY, USA, 2007. ACM.

[2] Kaichuan Wen. Translation of Java-embedded database queries with a prototype implementation for LINQ. Project Work(Studienarbeit), Technische Universität Hamburg-Harburg(TUHH), March 2009. `http://www.sts.tu-harburg.de/pw-and-m-theses/2009/wen09.pdf`.

[3] Anastasia Izmaylova. Program transformation in Scala. Master's thesis, Technische Universität Hamburg-Harburg, Sep 2009. Online version at `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaQL/MScThesis-final(A.Izmaylova).pdf`.

[4] Simone Bonetti. The construction of a type-directed LINQ to SQL provider. Master's thesis, Universität Tübingen, Oct 2009. `http://www-db.informatik.uni-tuebingen.de/files/publications/linq.thesis-sb.pdf`.

[5] Bas Lijnse and Rinus Plasmeijer. Between types and tables - using generic programming for automated mapping between data types and relational databases. In *IFL'08 : Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*, pages 115–130, 2008. `http://repository.ubn.ru.nl/handle/2066/71972`.

[6] Tom Schreiber. Übersetzung von List Comprehensions für relationale Datenbanksysteme. Master's thesis, Technische Universität München, 2008. `http://www-db.informatik.uni-tuebingen.de/files/publications/ferry.thesis-ts.pdf`.

[7] Paolo Pialorsi and Marco Russo. *Programming Microsoft LINQ*. Microsoft Press, 2008.

[8] Franklyn A. Turbak and David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.

[9] Microsoft Corporation. The .NET Standard Query Operators, February 2007. http://msdn.microsoft.com/en-us/library/bb394939.aspx.

[10] Microsoft Corporation. C# Language Specification 3.0, March 2007. http://msdn.microsoft.com/en-us/library/bb308966.aspx.

[11] Miguel Garcia. Compiler plugins can handle nested languages: AST-level expansion of LINQ queries for Java. In Moira C. Norris and Michael Grossniklaus, editors, *Proceedings of the 2nd Intnl Conf ICOODB 2009*, pages 41–58, July 2009. ISBN 978-3-909386-95-6, http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2009/icoodb/compplugin.pdf.

[12] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *Proceedings of the 28th ACM SIGMOD Int'l Conference on Management of Data (SIGMOD 2009)*, June 2009. Providence, Rhode Island (USA) http://www-db.informatik.uni-tuebingen.de/files/publications/ferry-sigmod2009.pdf.

[13] Manuel Mayr. Ein SQL:99 Codegenerator für Pathfinder. Diplomarbeit, Technische Universität München, April 2007. http://www-db.informatik.uni-tuebingen.de/files/publications/sql-code-generator.thesis-mm.pdf.

[14] James F. Terwilliger, Sergey Melnik, and Philip A. Bernstein. Language-integrated querying of XML data in SQL server. *Proc. VLDB Endow.*, 1(2):1396–1399, 2008. http://doi.acm.org/10.1145/1454159.1454182.