Program Transformation in Scala

submitted by
Anastasia Izmaylova
Matr. No. 33834

supervised by
Prof. Dr. Sibylle Schupp
Prof. Dr. Dieter Gollmann
Dr. Miguel Garcia

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

Abstract

The extension of programming languages with database query capabilities is called *language-integrated query*. This is a desirable goal in connection with two recent developments (from the programming and the database communities): (a) the functional-object paradigm; and (b) enhanced expressiveness and conciseness of functional query languages. In addition, the extensible compiler architectures for modern programming languages along with the advanced optimization techniques for functional query languages bring new perspectives to persistent programming languages. The main results of the present master project work are contributions that fall under the larger ScalaQL project. The ScalaQL project proposes a translation algorithm from two source languages, LINQ and Scala, to SQL:1999 queries with the Ferry query language as an intermediate language. The underlying translations are required to be total and semantics preserving. The master work contributes to (a) providing the syntactic and semantic foundation for the second translation (from Scala into Ferry); (b) covering the relevant aspects of both the Scala and Ferry type systems; and (c) establishing the required isomorphism of types between the supported Scala subset and Ferry. Finally, the proposed approach has been implemented as a Scala compiler plugin that allows compile-time processing of LINQ queries and preparing well-formed SQL:1999 queries amenable to relational DBMS evaluation.

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,
Anastasia Izmaylova

Acknowledgements

I would like to thank the head of the Instutute for Software Technology Systems, Prof. Dr. Sibylle Schupp, for supporting the research conducted within the ScalaQL project, and her active participation in the ScalaQL project, in general, and in my master project work, in particular.

I would also like to thank my second supervisor, the head of the Institute for Security in Distributed Applications, Prof. Dr. Dieter Gollmann, for his interest in my master project work.

Many thanks are to Dr. Miguel Alfredo Garcia Gutierrez for his active participation in my work, his valuable guidance and advices.

Special thanks are to my mother, Nadezhda Tyutyunnik, for her continuous support and love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Contents

1.1 Motivation

The integration of database and programming languages has gained a significant interest raised by best practices achieved at both sides. Modern programming languages, such as Scala, F#, X10, Fortress, have started a trend of blending the traditional object paradigms with the proven benefits of the functional style programming, *e.g.* higher-order functions, enriched libraries of operations on immutable collections. At the same time, modern functional database query languages, *e.g.* LINQ, increase expressiveness and conciseness of their expressions by introducing functional operations (in addition to the traditional relational ones) and by supporting comprehensions. The functional operations, *e.g. map*, *filter*, *flatMap*, provide compact notations while preserving the semantics of *list comprehensions* [25] being a foundation for query manipulation, in particular, allowing well-known optimization techniques. Extending such programming languages with a *language-intergated query* enables programs to efficiently query

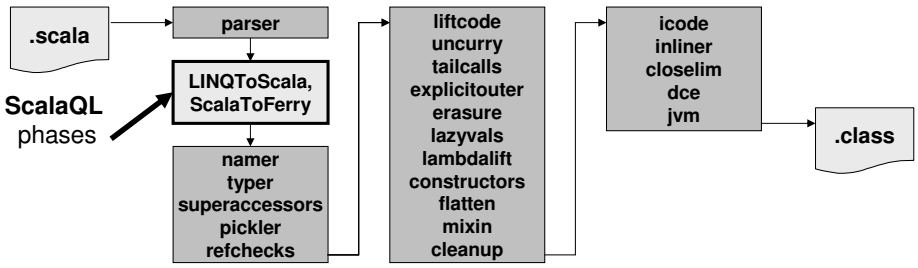

Figure 1.1: Scala compiler architecture and ScalaQL extension

data at different locations (databases) on demand, and manipulate these data in program space by using, for example, operators defined for collections. Having a *language-integrated query* benefits from enforcing well-formedness of queries at compile time while aiming at preserving conciseness of the expressions provided by the functional query languages (*e.g.*, by semantic-preserving mapping of common operations).

The topic of this master project is formulated as a part of the ScalaQL project[1]. The ScalaQL project proposes a translation algorithm from two source languages, LINQ and Scala (or rather, a subset of it), into Core SQL:1999 queries. The underlying translation is to be (a) total, i.e. each well-formed query of the input language is translated into a bunch of well-formed SQL queries; and (b) query semantics preserving. The main contribution of the master project is made to the translation from the target Scala subset to SQL:1999 queries to be evaluated at DBMS, and program transformations in Scala with correspondence to the proposed translation algorithm. The preference is given to Scala being a modern programming language combining both functional and object paradigms. The choice of Scala is underpinned by the provided compiler architecture designed to operate in phases delegating Abstract Syntax Trees (ASTs) between successive phases for analyses and transformations as shown in Figure 1.1. The default functionality of the Scala compiler phases can be easily extended by introducing a new custom compiler phase with respect to the interaction protocol [2] defined for compiler plugins. The Scala comprehensions and its extensive library on collection operations are in the core of the translation.

1.2 Problem Statement

This master project being a part of ScalaQL project addresses program transformations in Java and Scala with application to integration of mainstream functional database query languages and programming languages. The main focus is made on embedding of the modern query language LINQ into Scala by means of translating LINQ queries into Scala comprehensions. The resulting Scala queries are further translated to Ferry language [27] amenable to SQL:1999 evaluation on a relational DBMS. Both transformations are performed at compile time as a part of a custom Scala compiler (scalac) phase followed by the normal compilation phases (Figure 1.1). The transformations are applied at

[1] `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaQL`
[2] `http://www.scala-lang.org/sid/2`

the level of parse trees that are analyzed and transformed on the way from one phase to another. The well-formedness of Scala queries, and therefore LINQ queries, is guaranteed by successful completion of the compilation process. The well-formedness of resulting Ferry queries is guaranteed by the translation providing isomorphism of types between the chosen Scala subset and Ferry. The isomorphic property of the Scala-to-Ferry translation is partially ensured by final typechecking of Ferry queries.

As the initial part of the master thesis, program transformation mechanisms applied for embedding functional database query languages into Java are covered. The approach referred to as *nested languages* is applied for embedding a Semantic Web query language, OWL2 Functional Syntax, into Java in preference to *embedded DSLs* [14]. The underlying expansion of *nested* OWL2 queries is performed as a part of the compilation process of Eclipse IDE by means of Eclipse JDT plugin extension mechanisms and the provided APIs for AST rewritings.

1.3 Structure of this report

The rest of the chapter is organized as follows. Sec. 1.4 provides background on the underpinnings of modern query languages (query comprehensions), as embodied in an state of the art, optimizable query language for relational backends (Ferry). A summary description of the technologies under the LINQ umbrella closes that section, emphasizing aspects frequently glossed over in the literature (for example, degree of static checking and optimization). In Sec. 1.5 classification criteria are put forward to rank competing approaches to a *language-integrated query*, by defining four capability levels for language processors to handle queries involving different mixes of program vs. database semantics.

The following chapter, Chapter 2, discusses the typesafe embedding of a query language for the Semantic Web, OWL2 Functional Syntax, relying on the concepts of a *nested query* and *query expansion* (Sec. 2.2). Sec. 2.4 covers datails of the underlying Java program transformations supported by Eclipse JDT that are underpinned by the corresponding AST-building and AST-rewritings.

The next three chapters cover in detail our compilation pipeline, starting from LINQ to Scala (Chapter 3) followed by the chosen subset of Scala (types, operations, and syntactic constructs) that our Scala to Ferry translation accepts as input (Chapter 5). Chapter 4 provides detailed discussion of typing rules applied by our Scala-based typechecker against the full Ferry language.

Chapter 6 summarizes the underlying translations, from LINQ and Scala into Ferry, by giving the user perspective of the ScalaQL prototype and outlining its architecture. The implementation details covering Scala compiler (scalac) architecture with respect to custom compilation phases by a compiler plugin and relevant AST-building and AST-rewritings supported by scalac APIs are covered in Sec. 6.3.

The last chapter discusses conclusions (Chapter 7) and sketches possible future works with reference to related works. Appendix A summarizes the syntax and semantics of LINQ.

A prototype (ScalaQL) realizing our approach can be downloaded from `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaQL`

1.4 Background

1.4.1 Semantic foundation: query comprehensions

As summarized in [14], *query comprehensions* provide a uniform notation for denoting collections such as lists, bags and sets, based on the observation that the operations of set and bag union and list concatenation are monoid operations (that is, they are associative and have an identity element [12]).

In the list comprehension $[e \mid e_1 \ldots e_n]$ each e_i is a qualifier, which can either be a generator of the form $v \leftarrow E$, where v is a variable and E is a sequence-valued expression, or a filter p (a boolean valued predicate). Informally, each generator $v \leftarrow E$ sequentially binds variable v to the items in the sequence denoted by E, making it visible in successive qualifiers. A filter evaluating to *true* results in successive qualifiers (if any) being evaluated under the current bindings, otherwise 'backtracking' takes place. The *head* expression e is evaluated for those bindings that satisfy all filters, and taken together these values constitute the resulting sequence. For example, the meaning of the following OQL query:

```
select distinct e(x) from ( select d(y) from E as y where q(y) ) as x where p(x)
```

is captured by $\{\ e(x) \mid x \leftarrow \{\!\{\ d(y) \mid y \leftarrow E\ ,\ q(y)\ \}\!\}\ ,\ p(x)\ \}$

In tandem with closures (i.e., functions with parameters bound upon evaluation) the notation allows expressing complex queries, albeit not always compactly (the Scala collections library improves on this by encapsulating recurring patterns). In general, comprehensions contain nested queries. If evaluated as-is on large datasets, the engine would spend an excessive amount of time in nested loops, a situation that is overcome with optimizations for secondary-storage [16] and for main-memory [33].

1.4.2 Ferry: optimizing database comprehensions

Ferry [17], designed by Tom Schreiber at Uni Tübingen[3], pushes the envelope on how far a relational database engine can participate in program evaluation. Ferry's type system, constructs, and function library support computation, in particular comprehensions, over arbitrarily nested, ordered lists and tuples. A Ferry read-only query operates on data typed as per $t = a \mid [t] \mid (t, \ldots, t)$ where a represents atomic types like *string*, *int*, or *bool*. Structured types can be used to model programming language types such as lists, dictionaries ("maps" in Java), and algebraic datatypes. For performance, lists are not encoded following a (purely) recursive datatype formulation but as database tables. Unlike their program-level counterparts, Ferry lists must be homogeneous (all items sharing the same concrete type) for reasons having to do with static optimization (the detailed data layout has to be known). In other words, Ferry lists support parametric polymorphism but not subtype polymorphism.

The syntax of Ferry is fully composable (the same cannot be said of SQL'92) and revolves around the for-where-group by-order by-return construct. Additionally, let-bindings, conditionals, and primitive operators (arithmetic, relational, string) are supported. Table 5.17 summarizes the built-in functions. Several

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

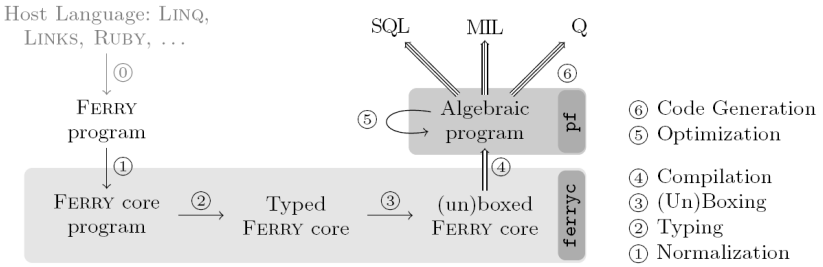

Figure 1.2: Relational translation of a non-relational language, reproduced from [17]

Table 1.1: Sample of Ferry's built-in function library

map ::	$(t \to t_1, [t]) \to [t_1]$	map over list
concat ::	$[[t]] \to [t]$	list flattening
take; drop ::	$(int, [t]) \to [t]$	keep/remove list prefix
nth ::	$(int, [t]) \to t$	positional list access
zip ::	$([t_1], \ldots, [t_n]) \to [(t_1, \ldots, t_n)]$	n-way positional
unzip ::	$[(t_1, \ldots, t_n)] \to ([t_1], \ldots, [t_n])$	merge and split
unordered ::	$[t] \to [t]$	disregard list order
length ::	$[t] \to int$	list length
all; any ::	$[bool] \to bool$	quantification
sum; min; max ::	$[a] \to a$	list aggregation
the ::	$[t] \to t$	group representative
groupWith ::	$(t \to (a_1, \ldots, a_m); [t]) \to [[t]]$	grouping

programming language embeddings are being developed by the team behind Ferry (including Ruby and itself, but not Scala), so far for relational backends only.

A Ferry program is compiled by the pipeline shown in Figure 1.2, a translation that relies on the *loop lifting* strategy [28] originally developed for the purely relational Pathfinder[4] XQuery compiler. The resulting algebraic query plans are amenable to dataflow-based analysis and optimization [16].

1.4.3 Design overview and supported use cases

Using our Scala extension, the developer needs only provide the query shown in Listing 1.1. In this case, the query has been formulated using Microsoft LINQ, thus fostering portability for queries across the .NET and JVM platforms. Internally, one component of our solution (the compiler plugin, Figure 1.1) takes charge of parsing and transforming the syntax tree in question into another, this time in terms of a Scala subset. In a nutshell, that subset comprises all features required as counterpart to LINQ-specific clauses (where, joint into, group by, and so on) as well as a subset of Scala's own operators (originating in the collections library and in supported datatypes).

Coming back to the example, the result of this source-to-source translation

[4]Pathfinder, `http://www-db.informatik.uni-tuebingen.de/research/pathfinder`

Listing 1.1: LINQ2Scala: Input

```
@LINQAnn val resultSet =
              " from how in travelTypes " +
              " join trans in transports on how equals trans.How into lst " +
              " select new { How = how, Tlist = lst } "
```

Listing 1.2: LINQ2Scala: Output

```
@Persistent val resultSet =
    for (how <- travelTypes;
         val outerKey = how ;
         val lst = for ( trans <- transports ;
                         if outerKey == trans.How )
                   yield trans
    ) yield new { val How = how; val Tlist = lst }
```

(from LINQ into Scala) is shown in Listing 1.2, not in the internal AST representation but as if the query had been written from scratch in Scala (another use case fully supported by our tooling). Sec. 3.1 addresses the correctness of this first translation. Our analysis is based on an (off-line, manually performed) detailed comparison of the semantics of source and target operators and types. In other words, the code blocks in the operators' definitions are not translated: the source-to-source translation occurs at the level of API contracts (which thus serve their purpose of abstraction barrier). The usage context for our solution (shipping of read-only queries for server-side evaluation) sidesteps many well-known difficulties from memory models (side-effects on shared mutable state, interference from updates by other threads). In anticipation of the discussion in that section, it can already be mentioned that differences between LINQ and Scala (regarding scoping, automatic coercions, and semantics of closures) do not present a big hurdle; a situation resulting from the aforementioned convergence trend of the functional and object paradigms.

At this point, denotational semantics is our guide to accomplish a second translation (from the Scala subset into Ferry) which involves: (a) formulating Scala-level operators in terms of a smaller set of built-in Ferry functions; and (b) reflecting the different container semantics (set, sequence, map, multiset) by means of appropriate encodings. Our reliance on a functional database query language (Ferry) is a departure from the architecture of established Object/Relational Mapping engines, but is in line with the design decisions embodied in Microsoft products, where Entity SQL[5] fills a comparable niche, the main difference being that Entity SQL already abstracts from records into objects. The final query to be shipped is shown in Listing 1.3.

1.4.4 Microsoft LINQ

Throughout this technical report, the term "LINQ" refers to the LINQ (embedded) query language. However, LINQ functionality results from the interplay of several technologies, the first one covering compile-time translation from LINQ

[5]Entity SQL, `http://msdn.microsoft.com/en-us/library/bb387145.aspx`

Listing 1.3: Query ready to ship for database-based evaluation

```
for how in travelTypes return
      let outerKew = how,
          lst = for trans in transports where outerKey == trans.how return trans
      in (how = how, tlist = lst) // record, not tuple
```

Listing 1.4: Runtime exception (in LINQ to SQL) at evaluation time

```
1 static void Main(string [] args)
2 { MyDatabaseDataContext ctx = new MyDatabaseDataContext();
3   var res = from s in ctx.Sites where s.UrlPath.Normalize() == "Test" select s;
4   foreach (var s in res) ;
5 }
```

textual syntax (embedded in languages such as and VB.NET) into *Standard Query Operators* (SQO), which are comparable to the operators in collection libraries of programming languages supporting closures. For example, the textual syntax from x in foo let y = f(x) select h(x, y, z) actually stands for the following code [14]: foo.Select(x => new { x, y = f(x) }).Select(t0 => h(t0.x, t0.y)). All we need to know about these ASTs is that LINQ renames the well-known map, filter, and flatMap into Select, Where, and SelectMany.

Another important component are *query providers*, i.e., implementations (possibly by third-parties) that receive SQO ASTs and return a resultset. Query providers, including that for main-memory evaluation, perform lazy evaluation of LINQ queries. This design guarantees that the minimum amount of work will be performed to obtain the first result, and that some queries on infinite input will be answered. When the query provider is connected to an RDBMS, queries operate not on sequences but on multisets: if any operators are applied after an OrderBy() there is no assurance that results will reflect the previous sorting. PLINQ, the project focusing on parallel evaluation of LINQ queries, puts it in these terms: "ordering operators re-establish order, shuffle points shuffle the order" [6].

Given that the semantics of query evaluation is at the mercy of the particular query provider in use, such evaluation may (a) produce a run-time error, (b) partition the expression into an SQL query and pre- and post-processing phases executed outside SQL, or (c) translate the expression completely to SQL.

Regarding (a), the division of labor between the C# compiler and query providers does not require the former to be fully aware about limitations of the latter[7]. This means that a query provider may be handed a query it cannot evaluate, as shown by the code in Listing 1.4: upon trying to iterate the resultset, a runtime NotSupportedException is thrown with the message "Method 'Normalize' has no supported translation to SQL". Given that in our approach both roles (query rewriting and shipping) are under control of the same compiler plugins, this mismatch is avoided.

[6] PLINQ, http://msdn.microsoft.com/en-us/magazine/cc163329.aspx

[7] http://bartdesmet.net/blogs/bart/archive/2007/07/05/linq-to-sharepoint-improving-the-parser-debugger-visualizer-fun.aspx

1.5 Levels of integration of host and query languages

A persistent programming language wallpapers over the different locations and longevity of data [4],[11],[23]. The more modest goal of language-integrated query also poses some challenges, that we classify into the integration levels where they manifest.

1.5.1 Level 1: Native query syntax

In this level, queries must be written in the language the DBMS understands (for our purposes, Ferry, but the same considerations apply to SQL, XQuery, and so on). This limitation implies that only the operators supported by the DBMS can be applied, and that expressions will only evaluate to types the DBMS can handle. After database evaluation, moving results back to program space poses no principle problem: the type system is rich enough to deliver an assignment-compatible type. Regarding variables, the only variables initially in the scope of queries are those representing persistent extents (in the relational context, each table is an extent; for the Entity Data Model[8], there are extents for entities and associations). Because usages of variables declared in the host program are disallowed, the typing rules of the query language allow static typing.

Native query syntax is the most verbose of all levels, but its embedding allows compiler plugins to detect queries broken due to refactorings of the database schema. Admittedly, Level 1 is not very useful in practice, but serves to set the stage for the next level.

1.5.2 Level 2: Static guarantee of database evaluation

In this level, a few restrictions are placed on usages of program variables, in a manner that allows finding out (conservatively) at compile time whether total translation is possible, i.e. whether the query can be fully evaluated by the DBMS without client-side processing.

First, program-level operators may appear in queries as long as they can be expressed in terms of one or more query language operators.

Second, Program-level literals and constructor invocations may appear as long as a lossless encoding exists for their types, for marshalling to and from the persistent representation (assuming that each persistable value can be denoted by a literal in the query language).

The features above could have been shoehorned into Level 1 by adding syntactic sugar to the query language. This redressing can go even further: LINQ constructs can be used as surface syntax over Ferry operators, literals, and types, adding convenience without increasing expressive power. In contrast, Level 2 enables the parameterization of queries with values known only at runtime, while retaining the property of database-only evaluation. In what follows we limit our attention to LINQ and Scala as (surface) syntaxes for language-integrated query, and Ferry as DBMS native query language.

Variable usages in queries can be either in left-hand side or right-hand side positions, where a LHS is to be interpreted as binding as opposed to destruc-

[8]Entity Data Model, `http://msdn.microsoft.com/en-us/magazine/cc700331.aspx`

Listing 1.5: A query statically known to be Level 2, using program variables

```
val paramEmp = Employee(...) // a case class instance
val parkingLots = List(North, South)
/* here comes a query where paramEmp and parkingLots appear in RHS position */
```

tive update. In comprehensions-aware query languages (Sec. 1.4.1), binding is implied by generators and let-declarations only. Additionally, LINQ and Scala add one more means to effect bindings, when constructing values of structural types (anonymous types in LINQ terminology), as with the expression new { x = 0, y = 0 }, which makes x and y visible in certain scope.

When parameterizing queries with runtime values, LHS positions are not the problem: they should be fresh names for the scope in question (neither LINQ nor Scala allow hiding of variables). Thus, no program variable can appear there anyway. On the other hand, allowing arbitrary program variables in RHS positions is a can of worms. Some usages are harmless (for example, variables of primitive types, whose declared types are final – cannot be subclassed – leading thus to statically known actual types). From a Ferry point of view, actual types are crucial, given that *the query plan fragment to generate for a given operator depends in general on the data layout of the operands*, i.e., their actual type has to be known statically. This inflexibility is the price to pay for the extensive optimizations that Ferry makes possible (Sec. 1.4.2), a capability we retain in all of Levels 1, 2, and 3.

In Level 2, a program variable is allowed in queries as long as: (a) its actual type is known statically; and (b) such type has a counterpart in Ferry's type system (possibly after marshalling and encoding). These restrictions are not as draconian as might seem. In practice, the parameters to a query are often constructed shortly before the query, in the same straight-line block of statements, as exemplified in Listing 1.5. Additionally, Embedded SQL lies halfway between Levels 1 and 2: while some program variables are allowed, not all programming language operators may appear inside queries.

1.5.3 Level 3: Optimizability known at shipping time

Levels 3 and 4 place no restrictions on RHS usages of program variables in queries, unlike Level 2 which bans usages of variables whose actual type (i.e., the precise runtime type) cannot be statically determined.

In Levels 3 and 4, in order to build the Ferry query to ship, the actual types of program variables are inspected using runtime reflection. This allows computing the Ferry type T (possibly after marshalling and encoding) for the value in question, if T exists. Otherwise, the variable's value cannot be shipped (i.e., cannot be passed as a by-value parameter to the database) and the enclosing fragment of the query is tainted for client-side processing (Level 4).

As an example of what can go wrong when translating into Ferry, consider the query for (e <- Employees; if e.skills == fashionableSkills) yield e.name where both e.skills and fashionableSkills have (Scala) type List[Skill]. When lexically enclosed in a query, == denotes structural equality, so that the query above expands into Ferry's

```
for e in Employees
where length(e. skills ) == length(⟦fashionableSkills⟧)
      and let diffs = filter ( v -> v.1 != v.2, zip(e. skills , ⟦fashionableSkills⟧))
          in length( diffs ) == 0
return e.name
```

where ⟦*fashionableSkills*⟧ is a literal in Ferry's syntax for the value in the similarly named program variable. For the Ferry expansion to be well-formed, fashionableSkills should be an homegenous collection (no instances of proper subtypes of Skill can be contained). Otherwise, the (structural) equality test v.1 != v.2 would compare apples with oranges, i.e. break a typing rule.

1.5.4 Level 4: Client-side processing

At this level, not all subexpressions in the query fulfill the conditions of previous levels. Those that do, can be given as input to the optimizer. A correct evaluation consists in shipping those fragments, and performing client-side processing after receiving their sub-results. This fallback measure makes performance contingent upon cache affinity, number of client-server roundtrips, the size of intermediate results, and the depth of nested loops. Level 4 is prone to the very situation we set out to avoid: non-optimized nested loops.

1.5.5 ScalaQL and LINQ under the light of integration levels

All existing approaches to languague integrated query exhibit slightly different strengths and weaknesses [8],[34],[20] and ours is no exception. After fixing the embedded query language to support comprehensions syntax, the dimensions for variation involve (a) whether the translation into a DBMS-supported query language is total (otherwise, a mixture of client-side and server-side processing takes place); (b) the range of target data models (relational, XML, OO databases); and (c) the level of semantic analysis performed at compile time.

Given that LINQ may resort to client-side processing (when targeting relational backends or otherwise), our approach compares favorably in all of Levels 1 to 3 (with the caveat that disambiguating whether a query is Level 3 or 4 takes place at runtime).

As discussed in Sec. 1.4.4 (Listing 1.4) some LINQ providers cannot rule out exceptions during query evaluation, given that some well-formedness checks (whether a specific target database supports certain operators) are delayed until runtime. Regarding this, all of Levels 1 to 4 do without exceptions of this kind. Database evaluation may end abruptly due to errors like division by zero, or more in general due to operands with incompatible types (for values held in program variables, Level 4). However, exceptions like that in Listing 1.4 happen for *all* executions of the query, and could have been flagged at compile-time as in our approach.

The current (beta) version of ScalaQL supports Level 2.

1.6 Contributions

As a part of the master project the following contributions have been made:

- OWL2Expander, a Java-based query expander implemented as an Eclipse plugin; it takes an OWL2 nested query as its input and performs program transformation in Java by expanding the OWL2 query to Java statements of the OWL2 API and inserting them as a Java block right after the nested query; the additional feature of folding away the resulting expansion block being a bonus *"in addition to"* feature can be added through OWL2Folding plugin;

- ScalaVisualizer, an Eclipse plugin for visualizing Scala ASTs after three scalac phases: (a) parser, (b) typer and (c) the final phase; developed as a by-product to structurally cover the details of the internal AST representation and transformation in Scala;

- ScalaQL prototype, a Scala compiler plugin aimed at supporting Level 2 according to Sec. 1.5. The ScalaQL prototype implementation carried out involved:

  1. extension of the existing standalone utility for translating LINQ queries into Scala comprehensions that comprised: (a) making sure that a larger set of collection operation invocations can be processed; and (b) integrating the resulting code into a compiler plugin;
  2. translation of the target Scala subset into Ferry language that comprised: (a) developing a utility for translating Scala queries into Ferry queries; and (b) integrating the resulting code into a compiler plugin;
  3. realization of all the required ASTs (Abstract Syntax Trees) transformations that comprised: (a) analyzing the structure of Scala ASTs including customization of traverse and transform mechanisms, and specification of the target Scala subset in the form of AST patterns; (b) applying all the necessary ASTs transformations inside the Scala compiler; and (c) preparing a test suite and accompanying technical documentation;

  ScalaQL prototype available as a *beta* version implements program transformation in Scala resulting from all the underlying translations, from LINQ into Scala and from the intended Scala subset into Ferry; designed to execute in two separate custom Scala compiler phases for each of the translations; open-source ScalaQL prototype can be downloaded from `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaQL`

- ScalaToFerryTyper, a Ferry typechecker that checks well-formedness of a Ferry query by deriving a Ferry expression type; operates at the Ferry level in contrast to Ferry Core; used by ScalaQL to partially ensure isomorphic property of the Scala-to-Ferry translation and signal incorrectness of the translation implementation;

The contributions above have been reported in the following publications:

- Paper: Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with database query capability, Journal of Object Technology, 2010, July-August, To appear. Preprint at `http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2009/jot/scalaql-preprint.pdf`

- Technical Report: Miguel Garcia and Anastasia Izmaylova, Compiling LINQ and a Scala subset into SQL:1999, Software Systems Institute (STS), Technische Universität Hamburg-Harburg, October 2009, Germany. Available online at `http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2009/ScalaQLTechRep01.pdf`

Chapter 2

AST Rewriting with Eclipse JDT

Contents

In this chapter some aspects of program transformation supported by Eclipse JDT are covered. In particular, an Eclipse plugin is contributed to support the typesafe embedding in Java of a query language for the Semantic Web, OWL2 Functional Syntax. We call the resulting prototype *OWL2Expander*. The underlying program transformation mechanism relies on two concepts: (a) the concept of *nested query*, that stands for a string encapsulating a query expressed with the embedded language syntax; and (b) the concept of *query expansion*, that refers to the host language statements generated from a detected nested query. Query expansion builds upon the existing IDE infrastructure and extends compilation functionality through a compiler plugin. This plugin enforces syntactical correctness and well-formedness checks on the embedded query at compile time, avoiding some runtime exceptions. Additionally, the compiler plugin traverses and rewrites ASTs (Abstract Syntax Trees), thus performing code transformation by inserting, replacing and deleting AST nodes representing parts of an input compilation unit.

Listing 2.1: SQL embedded in Java

```
final Sql sql = Select(ARTICLE.NAME, ARTICLE.ARTICLE_NO)
                .from(ARTICLE)
                .where(ARTICLE.OID.in(named("article_oid")))
                .toSql();
```

Listing 2.2: OWL2 expansion in Java

```
// OWL2 query: Declaration(Class(c:Cat))
OWLDataFactory dataFactory1 = new OWLDataFactoryImpl();
URI uri1 = new URI("c:Cat");
OWLClass owlClass1 = new OWLClassImpl(dataFactory1, uri1);
OWLDeclarationAxiom owlDeclarationAxiom1 = new OWLDeclarationAxiomImpl(
                                                dataFactory1, owlClass1);
```

2.1 Motivation and Background

The program transformation approach outlined above refers to the research area of extending languages with embeddings of Domain Specific Languages (DSLs). In the presence of existing technologies and with experiences already made, the current proposals to *embedded DSLs* aim at preserving the original syntax of the host language while providing well-formedness checks of the DSL expressions. These checks are performed at compile time to exclude some runtime exceptions. Examples of such approaches refer to the *Internal DSL* approach with relevant works on *Fluent Interface*, *Query Builder*, reification of the database schema, *Expression Builder*, *Native Queries*, *Criteria* API[1], etc. [14].

The approaches just mentioned rely on generation of the required APIs resembling DSL syntax and reusing the type system of the host language to enforce some well-formedness rules of the DSL along with the IDE-tooling. In Listing 2.1 an example of SQL embedded in Java is shown [2].

One of the obvious shortcomings of such approaches is that the resulting queries are not portable among different platforms. The alternative approach adopted in this work favours *nested* language where a *nested query* is taken as an input that is automatically expanded to a bunch of statements, *query expansion*, of an underlying internal DSL. The expansion is performed at compile time by a compiler plugin. The approach addresses the same goals as embedded DSLs and allows preserving an original syntax of the DSL without extending the host language grammar. The shortcoming of this approach is the lack of support for IDE refactorings that can be applied for embedded DSLs in the case of schema changes.

An example of Java query expansion for the simple OWL2 query Declaration(Class(c:Cat)) is shown in Listing 2.2.

[1] http://blogs.sun.com/ldemichiel/entry/java_persistence_2_0_public1
[2] http://www.jequel.de/

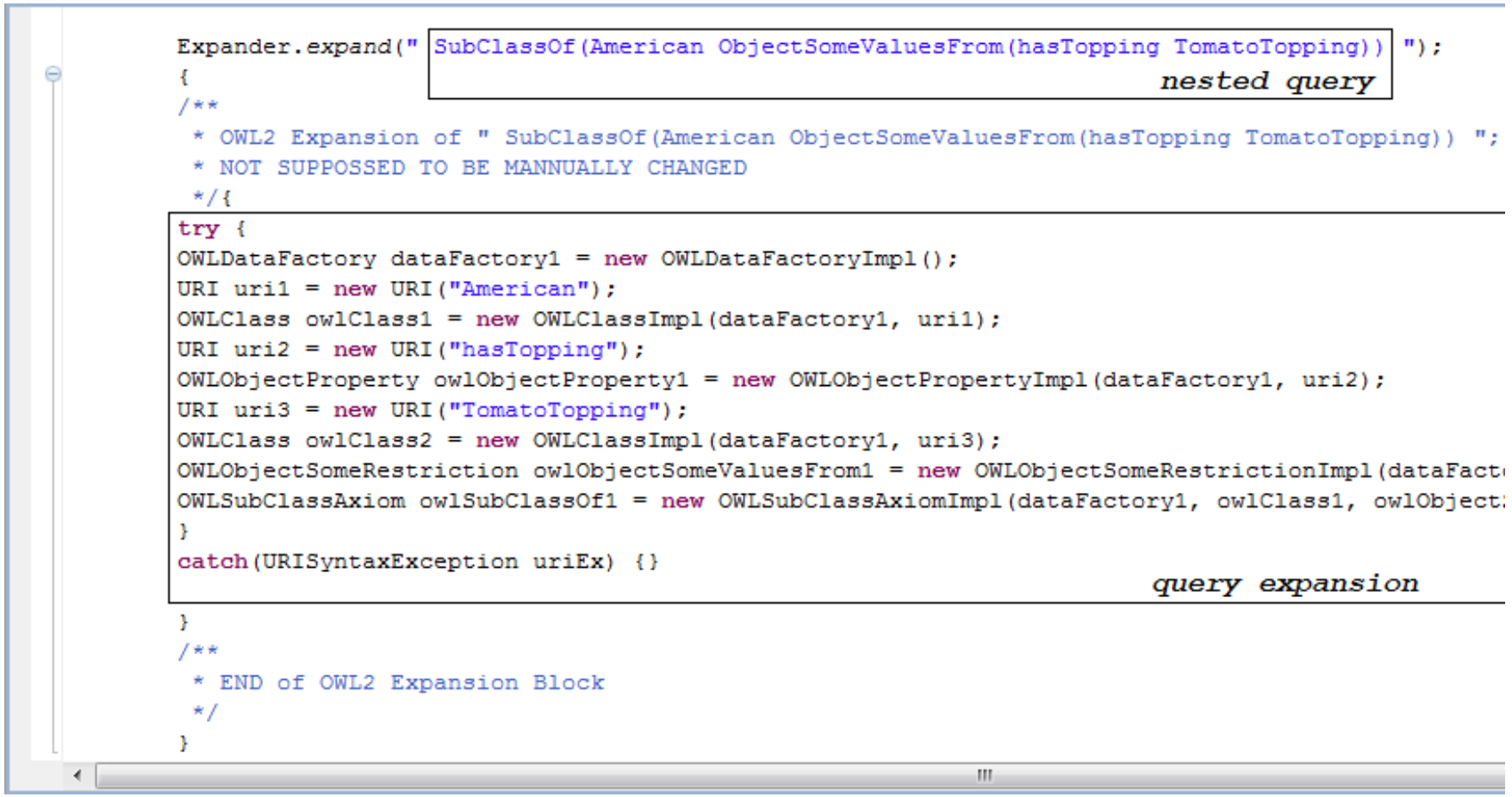

Figure 2.1: Query Expansion (OWL2Expander)

2.2 Query Expansion (user perspective)

The typesafe embedding of a functional query language implies the following steps performed at compile time

- detecting a *nested query*, i.e. a string instance encapsulating a query written following the syntax of the target embedded language
- parsing a detected nested query to a new string with statements of the host language
- parsing the obtained string to build AST nodes representing the expansion statements
- writing back the obtained query expansion to an original source code

The outlined steps are shown as part of the screenshot in Figure 2.1 on p. 22. The detected *nested query* that is a string argument of expand method is expanded to a block of Java statements associated with certain comments. The resulting block is inserted right after the *nested query* and typechecked by the compiler.

2.3 Compiler Plugin Development with Eclipse JDT/Core

Eclipse JDT/Core provides a powerful plugin extension mechanism that allows extending the IDE with, for example, static analyses. Participating in Java build process and reconciling Java editors is done by implementing a compilation participant with extension point: org.eclipse.jdt.core.compilationParticipant, as shown in Listing 2.3 on p. 24. Implementing the method reconcile allows accessing AST of the compilation unit and specifying a custom visitor (visitor pattern) for visiting its nodes and performing certain actions on some of them.

2.4 AST Manipulation with Eclipse JDT

Eclipse JDT provides a rich API for rewritable ASTs that allows visiting AST of an original compilation unit, parsing a source string with Java statements to build AST, retrieving additional information about AST nodes, performing AST rewritings, and writing back AST changes to the source file.

2.4.1 ASTVisitor

The input OWL2 queries of OWL2Expander are to be string arguments of @OWL2Expansion(toOWL2Expand = true)-annotated methods as in the example of Listing 2.1 on p. 22

```
Expander.expand(
" SubClassOf(American ObjectSomeValuesFrom(hasTopping TomatoTopping)) "
).
```

Such queries are detected by traversing an AST of a compilation unit obtained from a source file and applying a *visitor pattern* by extending the abstract class

Listing 2.3: Participating in Eclipse static analyses

```
public class OWL2CompilationParticipant extends CompilationParticipant {

        public boolean isActive (IJavaProject project) {
            return true;
        }

        public void reconcile (ReconcileContext context) {
                super. reconcile (context);

                // Getting CompilationUnit instance from the context
                CompilationUnit unitFromContext;
                try {

                        unitFromContext = context.getAST3();
                        // Building the corresponding AST of the compilation unit
                        AST astFromContext = unitFromContext.getAST();

                        // Creating an instance of the owl2 visitor
                        // that visits each AST node and perform specified actions
                        ASTVisitor owl2visitor = new OWL2Visitor(context);

                        // Accepting the owl2 visitor for the compilation unit
                        unitFromContext.accept( owl2visitor );

                } catch (JavaModelException e) {
                        // TODO Auto-generated catch block
                                e. printStackTrace ();
                }

        }

        public void buildStarting (BuildContext[] files , boolean isBatch) {
                // TODO Auto-generated method stub
                super. buildStarting ( files , isBatch );
        }

}
```

org.eclipse.jdt.core.dom.ASTVisitor. For the purposes of the OWL2Expander two methods of this class are implemented, public boolean visit(MethodInvocation invocation) and public boolean visit(Block block) (Listing 2.4 on p. 25).

2.4.2 ASTParser

After detecting an OWL2 functional query passed as a string argument to the @OWL2Expansion(toOWL2Expand = true)-annotated method, str_owl2, it is parsed to another string, str_expansion, encapsulating Java statements conforming to the OWL2 API [3]. The obtained string can be parsed to AST representation with a new instance of org.eclipse.jdt.core.dom.ASTParser as shown in Listing 2.5 on p. 25.

In the current implementation, obtained Java statements are wrapped to the compilation unit, and the resulting AST node returned by a parser is an instance

[3] http://owlapi.sourceforge.net/

Listing 2.4: Visitor pattern with ASTVisitor

```
public class OWL2Visitor extends ASTVisitor {
    ...
        // OWL2 specific strings are supposed to be passed as a string parameter of
        //                       @OWL2Expansion(toOWL2Expand = true)-annotated methods
        public boolean visit (MethodInvocation invocation) {
                // Testing if the method invocation has a OWL2 query to be expanded
                boolean toOWL2Expand = toOWL2Expansion(invocation);
                if (!toOWL2Expand) {return false;}
                // code that performs query expansion and AST rewriting on the compilation
                // unit

        // Visiting a OWL2 expansion block to check if its Java statements conform to
        // the OWL2 query above
        public boolean visit (Block node) {

                // Identifying an owl2 expansion block
                List comments = unitFromContext.getCommentList();
                // code that checks if the block is an OWL2 expansion block,
                //                  if it is preceeded by a method invocation with an OWL2 query
                //                  that conforms to its Java statements
        }

}
```

Listing 2.5: Parsing with ASTParser

```
// Creating an instance of the parser
//              and passing a char sequence derive from a string with Java statements
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(str_expansion.toCharArray());

// Parsing to the AST Node
ASTNode node = parser.createAST(null);

// Deriving the block of the expansion-related Java statements
CompilationUnit unit_expansion = (CompilationUnit) node;
Block block_expansion =((TypeDeclaration)unit_expansion.types().get(0)).getMethods()[0].getBody();
```

of org.eclipse.jdt.core.dom.CompilationUnit. The block with the Java statements, org.eclipse.jdt.core.dom.Block, as its chilren nodes can be finally obtained from the unit.

Referring to the example of Figure 2.1, the method statements() applied to the corresponding query expansion block will return the list of statements that are instances of subclasses of the abstract class org.eclipse.jdt.core.dom.Statement:

```
{ ...
URI uri1 = new URI("'American');
OWLClass owlClass1 = new OWLClassImpl(dataFactory1, uri1);
URI uri2 = new URI("hasTopping");
OWLObjectProperty owlObjectProperty1 = new OWLObjectPropertyImpl(dataFactory1, uri2);
URI uri3 = new URI("TomatoTopping");
OWLClass owlClass2 = new OWLClassImpl(dataFactory1, uri3);
...}
```

Listing 2.6: AST bindings

```
        ...
// obtaining the method binding
IMethodBinding binding = invocation.resolveMethodBinding();
// obtaining the annotation bindings
IAnnotationBinding[] annotations = binding.getAnnotations();
for (IAnnotationBinding annoBinding: annotations) {
        if (annoBinding.getName().equals("OWL2Expansion")) {
                IMemberValuePairBinding[] valuePairs = annoBinding.getAllMemberValuePairs();
                for (IMemberValuePairBinding valuePair: valuePairs) {
                        if (valuePair.getName().equals("toOWL2Expand")
                                &&valuePair.getValue().toString().equals("true")) {
                                toOWL2Expand = true;
                        }
                }
        }
}
        ...
// obtaining the parameter types binding
ITypeBinding[] parameterTypes = binding.getParameterTypes();
        ...
String parameterType = parameterTypes[0].getQualifiedName();
if (!parameterType.equals("java.lang.String")) {return false;}
        ...
```

2.4.3 AST bindings

The Eclipse JDT maintains additional information associated with each AST node, information that can be retrieved by resolving node bindings, *e.g.* by applying the method resovleMethodBinding() defined for MethodInvocation nodes as shown in Listing 2.6. The OWL2Visitor shown in Listing 2.4 visits each MethodInvocation node to discover whether it has an OWL2 query to be expanded as its string argument. In this case, a MethodInvocation binding, binding, comprises, for example, additional information about associated Java 5 annotations (annotations in the code listing, with method getAnnotations() retrieving them from a binding). A binding for an annotation comprises value pairs, and method getAllMemberValuePairs() on an annotation binding, and method parameter types, method getParameterTypes() on binding.

2.4.4 ASTRewrite

A new instance of the class org.eclipse.jdt.core.dom.rewrite.ASTRewrite can be used to protocol all rewrites on an AST of a compilation unit, as shown in Listing 2.7. All modifications are recorded in this protocol without touching the original AST. The intended modifications can be made to a rewrite list, an instance of the class org.eclipse.jdt.core.dom.rewrite.ListRewrite associated with the rewrite protocol and describing modifications to children of some AST node. For our *OWL2Expander*, an AST is modified by inserting an OWL2 expansion block to a rewrite list of the block statements containing a method invocation with OWL2 query. The modifications in the rewrite protocol are written back to the working copy by creating the text edits as an instance of org.eclipse.text.edits.TextEdit returned by the method rewriteAST() on the rewrite protocol that correspond to the document with the original source, an

Listing 2.7: Rewriting AST with ASTRewrite

```
        ...
// Creating a Document instance from the working copy referenced by the context
Document document = new Document(context.getWorkingCopy().getSource());
        ...
CompilationUnit unit = context.getAST3();
AST ast = unit.getAST();
// Creating an instance of ASTRewrite for the AST of the compilation unit
// that is used as a rewrite protocol for recording AST modifications
ASTRewrite rewrite = ASTRewrite.create(ast);
        ...
// Obtaining the expression statement encapsulating the method invocation
ExpressionStatement invocationExprStmt = (ExpressionStatement) invocation.getParent();
// Obtaining the block containing the method invocation
Block block = (Block) invocationExprStmt.getParent();
// Retrieving the list of statements to be extended by inserting an OWL2
// expansion block
ListRewrite lrw = rewrite.getListRewrite(block, Block.STATEMENTS_PROPERTY);
// Creating a string place holder from the string encapsulating an OWL2 expansion
ASTNode javastmts = rewrite.createStringPlaceholder(str_expansion, ASTNode.BLOCK);
// Inserting the resulting AST node (OWL2 expansion block) right after the
// method invocation
lrw.insertAfter(javastmts, invocationExprStmt, null);
// Applying the AST changes to the AST of the compilation unit referenced by
// 'document'
TextEdit edits = rewrite.rewriteAST(document, null);
// Writing back the recorded AST modifications to the source
context.getWorkingCopy().applyTextEdit(edits, null);
// Updating the AST of the compilation unit
context.resetAST();
        ...
```

instance of org.eclipse.jface.text.Document. Finally, the AST of the compilation unit has to be reset.

2.5 Folding Bonus with Eclipse JDT/UI

As can be seen on the screenshot shown in Figure 2.1, query expansion can result in a bulky block of statements even for a simple OWL2 query. Therefore, an additional contribution is made by defining a new folding structure of the Java editors that adds the OWL2 expansion block regions of a Java source file to the ones being folded away. Such a feature can be easily provided with Eclipse JDT/UI by implementing a Java folding structure provider with the extension point org.eclipse.jdt.ui.foldingStructureProviders. In order to add a new folding structure to the default ones, implementation of the method install of the interface org.eclipse.jdt.ui.text.folding.IJavaFoldingStructureProvider must install an instance of the default Java folding structure provider. The final step is going to *Window→Preferences→Java→Editor→Select folding to use* and selecting the folding to be used.

2.6 Evaluation

Two examples are given in this chapter illustrating expansion of the OWL2 queries, Declaration(Class(c:Cat)) (Listing 2.2 on p. 21) and SubClassOf(American ObjectSomeValuesFrom(hasTopping TomatoTopping)) (Figure 2.1 on p. 22) as well as the example of the SQL embedded in Java select ARTICLE.NAME, ARTICLE.ARTICLE_NO from ARTICLE where ARTICLE.OID in (:article_oid) (Listing 2.1 on p. 21) [4]. The examples demonstrate that queries built as a bunch of Java statements of the Internal DSL APIs require additional efforts to get familiar with a dedicated API even if the last shows resemblance to an original textual syntax. In addition, embedded queries can vastly break the conciseness and readability provided by DSLs as seen from the example of the very simple OWL2 query Declaration(Class(c:Cat)) expanded to a bunch of OWL2 API constructor invocations. Finally, as already mentioned in Sec. 2.1, such queries are not portable among different platforms. The applied approach effectively addresses the portability issue and aims at preserving the original syntax of the DSL while providing the same compile time well-formedness checks as internal DSLs. The current implementation of OWL2Expander prototype benefits from the powerful Eclipse IDE and Eclipse JDT plugin extension mechanisms allowing participation in the compilation as well as featuring existing Eclipse UI tools.

[4] http://www.jequel.de/

Chapter 3

LINQ to Scala

Contents

For the purpose of the translation into Scala, the accepted LINQ queries are those given by the grammar in Table A.1 (listing LINQ-proper productions, with *QueryExp* as entry rule) and in Table A.2 (listing other syntactic domains). In order to save space, well-known productions have been omitted (*e.g.*, those for arithmetic expressions). The notation conventions in the grammar follow Turbak and Gifford [30]. An alternative, more visual representation of the grammar is depicted in Figure 3.1.

The translation algorithm can be best understood by conceptually applying first a translation from LINQ into its *denotational semantics* formulation (Sec. A.2) followed from there by a relatively straightforward conversion to Scala comprehensions. In other words, we avoid the translation from LINQ into SQO (which is described in the C# 3.0 language spec and in more detail in [14]).

The input AST already has *query continuations* (i.e., subqueries) inlined: in LINQ, the subquery *def* in the expression "*def* into v *body*" is inlined by replacing usages of v with that definition. This is justified as the syntax for query continuations amounts to a let-declaration of the form: "let v = *def* in *body*".

As presented in Sec. 1.4.1, the notation for comprehensions does not commit to a particular type system and operations set on contained expressions, other than the requirement for one or more iterable collection types to exist. The denotational semantics for LINQ in Appendix A reinforces this point: the meaning of contained expressions was left unspecified. This perspective on LINQ is useful for integration with languages that already define a type system and operations

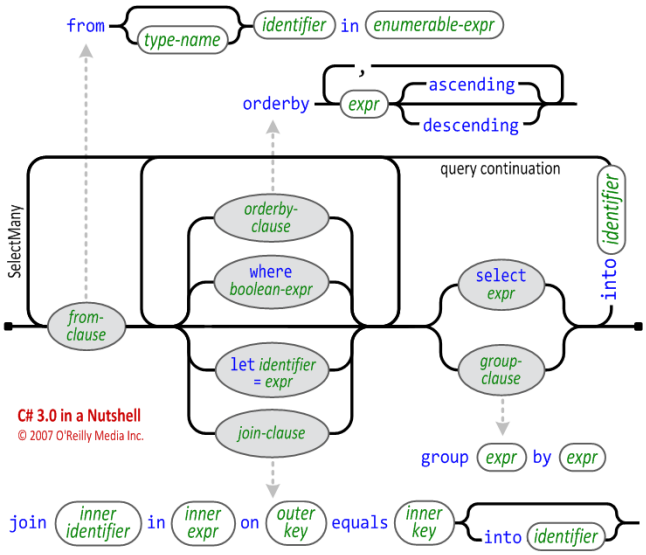

Figure 3.1: Railroad diagram for the textual syntax of LINQ, reproduced from `http://www.albahari.com/nutshell/linqsyntax.html`

set, in our case Scala. We require the enclosed expressions to be side-effect free, a property that will be maintained in the Scala and Ferry targets.

3.1 Translation algorithm

After inlining, a *QueryExp* consists of:

(a) at least one *FromClause*; followed by

(b) zero or more *BodyClause*; followed by

(c) one of a *SelectClause* or a *GroupByClause*.

A case analysis is applied by the translator to handle these AST shapes.

As with the denotational formulation, the two kinds of clauses that produce "end results" (select and group by) do so by evaluating the head of a comprehension over a collection of *input* binding environments. The Scala counterpart to the comprehension head is a yield e construct. For that evaluation to be semantically equivalent with respect to the original LINQ query:

- the same variables that were made to go into scope by preceding body clauses (FromClause, LetClause, WhereClause, JoinClause, JoinIntoClause, and OrderByClause) must also be made go into scope by the comprehension qualifiers *quals* inside the for (*quals*) yield e
- moreover, those variables should bind (in each binding environment) to the same type and values as their LINQ counterparts. Each binding environment is conceptually given by the nested iterations that comprehensions denote, or by tuples in the cartesian product of collections being iterated (details in [14]).

Listing 3.1: Correspondence between LINQ and Scala environments, handling of orderby

```
// from str in strs
// let chrArr = str.ToCharArray()
// from ch in chrArray
// orderby ch
// select ch

    val res9 = for ( ( str, chrArr, ch )
                     <- ( ( for ( str <- strs;
                                  val chrArr = str.ToCharArray();
                                  ch <- chrArray
                            ) yield ( str, chrArr, ch )
                          ) orderBy { _ match { case ( str, chrArr, ch ) => ch } }
                        )
               ) yield ch
```

The translation is thus compositional, with the contract just mentioned between qualifiers and head of comprehension.

The flow of binding environments comes to bear in the example of Listing 3.1. In the LINQ formulation, at the final "select ch", three variables are in scope (as introduced by two previous generators and a let). Correspondingly, the outermost Scala yield will produce for each binding environment the result denoted by select ch (in terms of the Scala formulation, will produce a collection item for each 3-tuple holding values for those three variables).

3.1.1 Handling of group ... by

Before discussing the Scala-level formulation of grouping, we review details of the interfaces supported in the Microsoft implementation, with the goal of making sure that operations on the resulting Scala-level values (relying on those interfaces) are supported.

A LINQ group by clause introduces an irregularity in that it returns nested collections: group *result* by *key* returns a finite *ordered* map of *groupings*, where a grouping denotes $key \mapsto cluster$, a cluster being a sequence of results.

In the Microsoft implementation, a given instance of IGrouping (holding a single key of type K and a sequence of values of type T) can also be iterated, following the convention that IGrouping<K,T> implements the interface IEnumerable<T>. If grp denotes one such group, then:

- its *values* (not keys) can be iterated as in from v in grp (and their key obtained with the v.Key getter, same as with grp.Key)
- any of the SQO operators can be applied, as in grp.Count()

Also in the Microsoft implementation, iterating the result of any overloaded SQO GroupBy operators yields one grouping at a time, in agreement with the type of the result which is IEnumerable<IGrouping<TKey, TSource> >.

In the LINQ expression group ... into v, the type of v is IGrouping<TKey, TSource>. Therefore, v can receive collection operations.

Table 3.1: JoinInto and translation scheme

(a)	**join** $T_{type}^{0..1}$ $V_{innervar}$ **in** $E_{innerexp}$ **on** E_{lhs} **equals** E_{rhs}

(b)	override def visit(jc: JoinClause, rinnerexp: String,
	rlhs: String, rrhs: String) = {
	val outerKey = fresh("outerKey")
	"val " + outerKey + " = " + rlhs + " ; " +
	tpVar(jc.tp, jc.innervar) + " <- " + rinnerexp +
	" ; if " + outerKey + " == " + rrhs
	}

3.1.2 Handling of *JoinClause*

The production *JoinClause*, reproduced in Table 3.1 (a), lists the *inner collection*, *inner variable*, as well as expressions for *outer key* and *inner key* (appearing left and right of the on ... equals ...). The outer key is evaluated in a context where the inner variable is not yet visible, thus warranting the translation into the qualifiers generated by the code snippet in (b). Those qualifiers are: a local variable declaration, an iteration, and a guard.

The Microsoft implementation of the SQO join operator (*for main-memory*) does the following[1]:

> *When the object returned by Join is enumerated, it first enumerates the inner sequence and evaluates the innerKeySelector function once for each inner element, collecting the elements by their keys in a hash table. Once all inner elements and keys have been collected, the outer sequence is enumerated. For each outer element, the outerKeySelector function is evaluated and, if non-null, the resulting key is used to look up the corresponding inner elements in the hash table. For each matching inner element (if any), the resultSelector function is evaluated for the outer and inner element pair, and the resulting object is yielded.*

3.1.3 Handling of join ... into

The SQO counterpart of a join ... into ... clause is not the method Join<T, U, K, V> but a GroupJoin<T, U, K, V>. Both have IEnumerable<V> as return type, but they differ in the resultSelector closure received as argument:

- For SQO Join, it has type Func<T, U, V>
- For SQO Join, it has type Func<T, IEnumerable<U>, V>

The book "Introducing Microsoft LINQ" goes on to add:

> *In case of the absence of a corresponding element group in the inner sequence, the GroupJoin operator extracts the outer sequence element paired with an empty sequence (Count = 0).*

[1] reproduced online technical documentation "The .NET Standard Query Operators"

Listing 3.2: join into in LINQ and Scala

```
// from how in travelTypes
// join trans in transports on how equals trans.how into lst
// select new { how = how, tlist = lst}
    val res12 = for (how <- travelTypes;
                     val outerKey = how ;
                     val lst = for ( trans <- transports ;
                                     if outerKey == trans )
                               yield trans
                ) yield new { val how = how; val tlist = lst }
```

3.1.4 Handling of orderBy

There is a big semantic difference between LINQ's orderby clause and sorting functions in programming languages. In LINQ, its effect consists in permuting the sequence of binding-environments under which the following clauses (comprehension qualifiers or head) will be evaluated, as discussed in Appendix A.2. The Scala counterpart to orderby acts instead on an input collection, not on binding-environments (which are implicit). Therefore, in order to keep the LINQ → Scala translation compositional, we feed those "following expressions" with tuples containing all variables in scope. The order in which tuples are delivered reflects the ordering criteria, an ordering resulting from making explicit (as a sequence of tuples) the binding-environments that orderby refers to.

The example in Listing 3.1 showcases the translation pattern: the environments to be permuted (determined by all body clauses before the orderby) are computed by a nested Scala comprehension, given as input to a orderBy. To maintain the contract between qualifiers and head of comprehension, the result of such orderBy is again a collection of bindings.

The current custom implementation of orderBy is based on `http://joelneely.wordpress.com/2008/03/29/sorting-by-schwartzian-transform-in-scala/`

3.2 Implementation

The parser relies on the library for packrat combinator parsers that is part of Kiama (`http://kiama.googlecode.com/`). The snippet in Listing 3.3 illustrates the structure of the parser sts.linq.Parser.

The translation from a LINQ AST into a Scala comprehension (as a String) is realized in sts.linq.Comprehend, for invocation as in Comprehend(ast) which invokes the following:

```
class Comprehend extends Stringify {
  override def apply(e: LINQAttr) : String = {
    val w = new ComprehendWalker(this, "defaultOuput")
    w walk e
  }
```

Before translation proper, the input AST is pre-processed. Firstly, query continuations are inlined (transformation T1 in [14]). After this, there is only one query to translate, which may contain nested LINQ queries as collection-valued

Listing 3.3: Snippet of the LINQ parser

```
lazy val queryexp : Parser[QueryExp] =
    fromclause ~ querybody ^^ { case from ~ qbody => QueryExp(from, qbody) }

lazy val fromclause : Parser[FromClause] =
  fromclauseWithoutType | fromclauseWithType

lazy val fromclauseWithoutType : Parser[FromClause] =
  ("from" ~> IDENTIFIER) ~ ("in" ~> exp) ^^
  { case variable ~ in => FromClause (List(), variable , in) }

lazy val fromclauseWithType : Parser[FromClause] =
  ("from" ~> strsdotsep) ~ IDENTIFIER ~ ("in" ~> exp) ^^
  { case tp ~ variable ~ in => FromClause (tp, variable, in) }

lazy val querybody : Parser[QueryBody] =
  (bodyclause*) ~ sg ~ (querycont?) ^^
  { case qbclauses ~ selgby ~ qcont => QueryBody(qbclauses, selgby, qcont) }
```

expressions. Secondly, the query-body clauses are placed in a list. Both steps are performed by the following:

```
override def visit (qe: QueryExp, rfrom: String , rqbody: String ) = {
  val transformed = Transformer inlinecontinuations qe
  val linearized = Linearize(transformed.asInstanceOf[QueryExp])
  . . .
```

3.2.1 Execution context of LinqToScala

Both LINQ queries and Scala comprehensions are desugared into the map, filter, and flatMap building blocks known from functional programming.

This desugaring is internal to the respective compilers (C# and Scala). In what follows we focus only on the scalac compiler. For example, after phase typer the comprehension:

```
val res = for ( p <- e ) yield p + 1
```

is *desugared by the namer phase* (i.e., very early) into the following intermediate program representation (as can be seen by invoking scalac -Xprint:typer):

```
val res: Array[Double] =
  e.map[Double](((p: Double) => p.+(1))): Array[Double];
```

Same AST, another visualization (scalac -Ybrowse:typer) on Figure 3.2.

In case the expressions making up a Scala comprehension are side-effect free[2], the query can be rewritten and shipped to a DBMS for evaluation (for example, a DBMS supporting the Microsoft Entity Framework [1]).

The document *Internals of Scala Annotations*[3] explains how symbol and type annotations are represented at different phases of the compiler.

[2]Functional purity in Scala, `http://jnordenberg.blogspot.com/2008/10/functional-purity-in-scala.html`

[3]Internals of Scala Annotations, `http://www.scala-lang.org/sid/5`

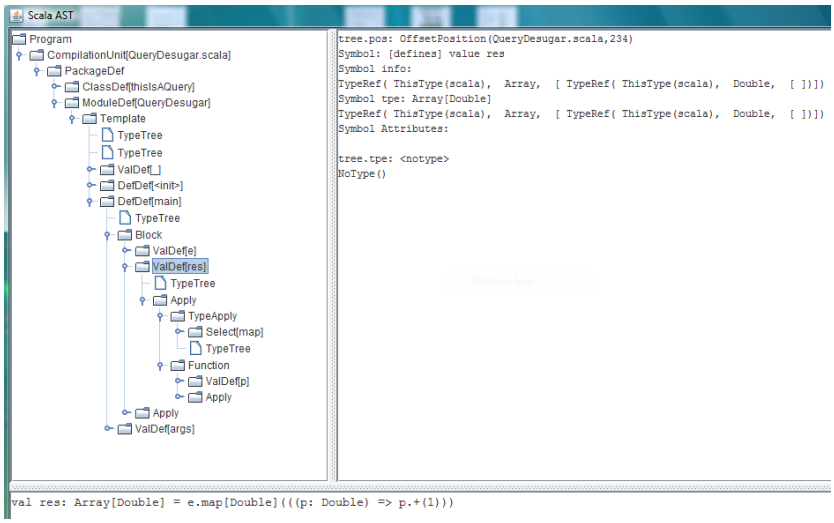

Figure 3.2: Comprehension AST after typer phase

3.2.2 LinqToScala in action

The implementation of the LINQ → Scala translation is based on the *compiler plugin* architecture[4] of Scala. ScalaQL generates an AST from a LINQ query string by:

1. detecting where the input LINQ query is found in the Scala source, parsing that string `s1`, converting the resulting syntax tree into a new string `s2` containing the Scala translation.
2. replacing (in the text of the compilation unit where `s1` was found) that occurrence with `s2`.
3. letting the `scalac` parser parse the new compilation unit, which now contains only Scala code.
4. replacing the AST of the original compilation unit with that of the expanded one.

The code shown in Listing 3.5 is responsible for these steps.

[4] `http://www.scala-lang.org/sid/2`

Listing 3.4: Snippet of the LINQ parser

```
class LINQtoScalaTransformer extends Transformer {
        ...
        override def transform(tree: Tree): Tree = {
    val newTree = super.transform(tree);
    newTree match {
       case Template(parents, self , body) =>
                copy.Template(newTree,parents, self , newStmts(body))
        case Block(stats , expr) =>
                copy.Block(tree , newStmts(stats),expr)
        case _ =>
                newTree
      }
  }
```

Listing 3.5: Parsing and replacing

```
private def newStmts(body:List[Tree]): List[Tree] = {
     var body1:List[Tree] = List()
     for (tree <- body) {
       tree match {
         case ValDef(Modifiers(_,_,List(Annotation(Apply(Select(New(Ident("LINQAnn")),_),_),_),_))),
                                          _,_, Literal(Constant(rhs))) =>
                 if (name.toString == "LINQAnn") {
                   // parsing the corresponding string taken from right hand side part
               // of the annotated value definition
                   val in = new java.io.CharArrayReader (rhs.toString.toArray)
                   val ast = sts.linq.Parser.run(in)
                   val scala_str = sts.linq.Transformer comprehend ast
                   // wrapping the resulting parse scala string to be injected
                   // into a compilation unit
               val unit_str = "object LINQObject{ " + scala_str + "; }"
                   val unit_inject = new CompilationUnit(new BatchSourceFile("", unit_str.toCharArray))
                   // intermidiate scalac parse phase aimed to create AST from the wrapped
                   // scala string
                   unit_inject.body = new global.syntaxAnalyzer.UnitParser( unit_inject ).parse
                   // unwrapping aimed to derive the AST nodes represented the scala statements
                   // generated from the initial LINQ query
                   ( unit_inject.body: @scala.unchecked) match {
                    case PackageDef(_, List(ModuleDef(_,_,Template(_,_,linqstats)))) =>
                          body1 = body1::: linqstats
                    }
           }
          ...
     }
   }
   body1
}
```

Chapter 4

Typing rules for Ferry

Contents

4.1 Detailed typing of Ferry

Ferry [27] is in fact a surface syntax that is desugared into Ferry-Core, the actual language for which a type system and translation rules are defined. However, to simplify the translation from Scala, we define the typing rules of Ferry, whose typing constraints are imposed by the typing and well-formedness conditions of Ferry-Core. These constraints are not apparent from the grammar of Ferry alone.

4.1.1 Terminology

Before delving into details of Ferry's type system, some terminology is considered.

As in other relational query languages, an important type in Ferry is the (non-nested) *tuple type*. Additionally, an *homogeneous list type* is available, as well as *atomic types* (aka DB column types). Non-atomic types are called "structured types". Our notation of choice for Ferry's type system is based on domain theory [30, Appendix A].

$$\mathrm{Ft} = \mathrm{FTuple} + \mathrm{Fb},$$

$$\text{FTuple} = \text{Fb}^2 + \ldots + \text{Fb}^n,$$
$$\text{Fb} = \text{FList} + \text{Fa},$$
$$\text{FList} = Ft^*$$
$$\text{Fa} = \{\mathsf{bool}, \mathsf{int}, \mathsf{string}, \ldots\}$$

where domain variables for atomic and structured types

$$\begin{array}{l} t \in \text{Ft}, \\ b \in \text{Fb}, \\ a \in \text{Fa}, \\ (b_1, \ldots, b_n) \in \text{FTuple}, \\ [t] \in \text{FList} \end{array}$$

(Fb^n contains types of all tuples with n elements of type $b :$ Fb, and Ft^* contains types of all finite-length lists with elements of type $t :$ Ft).

Unlike lists in OO languages, Ferry lists are homogeneous without subtyping. For example, a Ferry list cannot simultaneously contain tuples of both 2D and 3D points. Consequently, the corresponding Ferry functions taking lists as arguments, *e.g.*, append and concat require the same element type for all arguments:

$$\mathsf{append} \ : \ \forall t_1, t_2, t \in \text{Ft}, \ t_1 = t_2 = t \Rightarrow ([t_1], [t_2] \rightarrow [t])$$
$$\mathsf{concat} \ : \ \forall t_j, t \in \text{Ft}, t_1 = \ldots = t_m = t \Rightarrow [[t_1], \ldots, [t_m]] \rightarrow [t]$$

Ferry does not allow nesting of tuples. In principle, a further processing step ("flattening"/"unflattening") could be applied to lift this limitation. However, that would require extending the Ferry language (type system, etc.). We mention this possibility but for now maintain the limitation of flat tuples, both for user-provided expressions as well as for intermediate results. Therefore, tuple types have the form:

$$\text{FTuple} \subset \text{Ft}^2 + \ldots + \text{Ft}^n$$

As an aside, the Ferry expression $((1, 2), 3)$ *could* be internally treated (after flattening) as $(1, 2, 3)$. With that, the expression $((1, 2), 3).1.2$ *would* typecheck and be well-formed.

For the discussion of Ferry typing, in addition to type domains defined above and domains representing Ferry expressions Table 4.1 and Table 4.2, a type environment domain TypeEnvironment is introduced as follows

$$TE \in \text{TypeEnvitornment} = \text{Identifier} \rightarrow \text{Ft},$$

and the following domain variables are used

$$e, e_j \in \text{Exp}, \ v, v_j \in \text{Identifier}, t, t_j \in \text{Ft}, \ b, b_j \in \text{Fb}.$$

Given the formal specification of Ferry type system, the type derivation rules for checking well-typedness of Ferry expressions with reference to [27] are discussed in the next sections.

Table 4.1: Ferry-related production rules

$$
\begin{aligned}
Id,\ V,\ T,\ TC,\ F \in\ & \text{Identifier} = \text{[a-zA-Z][a-zA-Z0-9]*} \\
N \in\ & \text{NatLiteral} \\
E \in\ & \text{Exp} = (\ \text{IntLiteral} \cup \text{StringLiteral} \cup \text{BoolLiteral} \\
& \quad \cup \text{UnaryOpExp} \cup \text{BinOpExp} \\
& \quad \cup \text{TupleExp} \cup \text{ListExp} \\
& \quad \cup \text{PosAccExp} \cup \text{NomAccExp} \\
& \quad \cup \text{LetExp} \cup \text{IfExp} \cup \text{ForExp} \\
& \quad \cup \text{VarUseExp} \cup \text{TableRefExp} \cup \text{FunAppExp}
\end{aligned}
$$

$$
\begin{aligned}
TPL \in \text{TupleExp} &::= (\ E_{exprs}^{2..*\ <separator:,>}\ ) \\
LST \in \text{ListExp} &::= [\ E_{exprs}^{0..*\ <separator:,>}\ ] \\
PA \in \text{PosAccExp} &::= E_e\ .\ N_n \\
NA \in \text{NomAccExp} &::= E_e\ .\ TC_{tablecolname} \\
VE \in \text{VarUseExp} &::= V_{varid}
\end{aligned}
$$

4.1.2 Ferry's Tuples and Lists

Ferry tuples [27, p. 50]

$$\forall e \in \text{TupleExp},\ e = (e_1, \ldots, e_n) \Rightarrow$$
$$\frac{\forall_{j=1}^{m} TE \vdash e_j : b_j}{TE \vdash e : (b_1, \ldots, b_m)}$$

Taking into account 'flattening' results in $m \geq n$.

Ferry lists [27, p. 50]

$$\forall e \in \text{ListExp},\ e = [e_1, \ldots, e_n] \Rightarrow$$
$$\frac{\forall_{j=1}^{n} TE \vdash e_j : t}{TE \vdash e : [t]}$$

A Ferry tuple allows accessing its elements by using expressions

$$e.n \in \text{PosAccExp},\ n \in \text{NatLiteral} \Rightarrow$$
$$\frac{TE \vdash e : (b_1, \ldots, b_n, \ldots, b_m)}{TE \vdash e.n : b_n},$$

or

$$e.c \in \text{NomAccExp},\ c \in \text{TableColName}, c \mapsto n \Rightarrow$$

Table 4.2: Ferry-related production rules

$LBC \in$ LetBindingClause	::=	$V_{varid} = E_e$
$LTE \in$ LetExp	::=	**let** $LBC_{letbindingcls}^{1..* <separator:,>}$ in E_e
$IFE \in$ IFExp	::=	**if** E_{filter} **then** E_{e_1} **else** E_{e_2}
$FBC \in$ ForBindingClause	::=	V_{varid} **in** E_e
$WC \in$ WhereClause	::=	**where** E_e
$GBC \in$ GroupByClause	::=	**group by** $E_{exprs}^{1..* <separator:,>}$
$M \in$ OrderModifier	=	{**ascending**; **descending** }
$OBC \in$ OrderByClause	::=	**order by** $(E_e M_{ordermodifier}^{0..1})^{1..* <separator:,>}$
$FRE \in$ ForExp	::=	**for** $FBC_{forbindingcls}^{1..* <separator:,>}$
		$WC_{wherecls_1}^{0..1}$
		$(GBC_{groupbycls} WC_{wherecls_2}^{0..1})^{0..1}$
		$OBC_{orderbycls}^{0..1}$
		return E_e
$TCT \in$ TableColType	=	{**int**; **string**; **bool**}
$TCS \in$ TableColSpec	::=	$TC_{tablecolname}\ TCT_{tablecoltype}$
$TKS \in$ TableKeySpec	::=	$(TC_{tablecolname}^{1..* <separator:,>})$
$TOS \in$ TableOrderSpec	::=	$TC_{tablecolname}\ M_{ordermodifier}^{0..1}$
$TRE \in$ TableRefExp	::=	**table** $T_{tableid}$ $(TCS_{tablecolspec}^{1..* <separator:,>})$
		with keys $(TKS_{tablekeyspec}^{1..* <separator:,>})$
		$(\textbf{with order} (TOS_{tableorderspec}^{1..* <separator:,>}))^{1..* <separator:,>}$
$FE \in$ FunAppExp	::=	$F_{funid}(E_{exprs}^{0..* <separator:,>} \mid V_{varid} \rightarrow E_{e_1}, E_{e_2})$

$$\frac{TE \vdash e : (b_1, \ldots, b_n, \ldots, b_m),\ c \mapsto n}{TE \vdash e.c : b_n}.$$

Given that the following user-provided expressions

$$[\ v'.c \mid v' \leftarrow [\ (v.c, \ldots) \mid v \leftarrow e]\ ],$$

$$\frac{TE \vdash e : [(b_1, \ldots, b_n, \ldots, b_m)],\ c \mapsto n,\ TE[v : (b_1, \ldots, b_n, \ldots, b_m)] \vdash (v.c, \ldots) : (b_1, \ldots)}{TE[v' : (b_1, \ldots)] \vdash v'.c : b_1}.$$

will be evaluated by Ferry type system described in this technical report as well-formed, and a type will be assigned to it. Well-formedness of such expressions means that the column names can be propagated to an outer tuple. Such a propagation is also promoted by some of the Ferry macros, *e.g.*, groupBy [27, p. 8]

$$[v'.c \mid v' \leftarrow \mathsf{groupBy}\,(v \rightarrow (e_{g_1}, \ldots, e_{g_n}), e)]\,,$$

$$\frac{TE \vdash e : [(b_1, \ldots, b_n, \ldots, b_m)],\ c \mapsto n,\ \forall_{j=1}^{n} TE[v : (b_1, \ldots, b_n, \ldots, b_m)] \vdash e_{gj} : a_j}{TE[v' : ([b_1], \ldots, [b_n], \ldots, [b_m])] \vdash v'.c : [b_n]}.$$

and unzip [27, p. 8]

$$v' = \mathsf{unzip}(e),$$

$$\frac{TE \vdash e : [(b_1, \ldots, b_n, \ldots, b_m)],\ c \mapsto n}{TE[v' : ([b_1], \ldots, [b_n], \ldots, [b_m])] \vdash v'.c : [b_n]}.$$

Similarly, column names can be propagated to a user-provided outer tuple expression by lists and some of the Ferry functions taking list as their arguments (obvious examples would be the, take, drop [27]) provided that list element expressions are typed by mapping to the same column name.

4.1.3 Ferry's IfExp and LetExp

$$\forall e \in \text{IFExp},\ e\ =\ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow$$
$$\frac{TE \vdash e_1 : bool,\ TE \vdash e_2 : t,\ TE \vdash e_3 : t}{TE \vdash e : t}$$

Note that Ferry IFExp requires both e_2 and e_3 to be assigned to the same type t.

$$\forall e \in \text{LetExp},\ e\ =\ \text{let } v_1 = e_1, \ldots, v_n = e_n \text{ in } e_0 \Rightarrow$$
$$\frac{\forall_{j=1}^{n} TE[v_i : t_i]_{i=1}^{j-1} \vdash e_j : t_j,\ TE[v_j : t_j]_{j=1}^{n} \vdash e_0 : t_0}{TE \vdash e_0 : t_0}$$

Note that each new variable definition contributes to the type environment of all subsequent expressions.

4.1.4 Type signature of Ferry's where clause

On [27, p. 41], Ferry's where clause is desugared to filter macro that is represented in the Table 4.3.

According the signature of Ferry's filter macro

$$\frac{TE \vdash e_2 : [t],\ TE[v : t] \vdash e_1 : bool}{TE \vdash \mathsf{filter}(v \leftarrow e_1, e_2) : [t]}$$

Table 4.3: Desugaring of Ferry's where

source	for v in	e_1	where e_2 ... *more ForClauses*
target	for v in	filter	(v -> e_1 e_2) ... *more ForClauses*

4.1.5 Type signature of Ferry's order by clause

According to the rewriting rule (FOR-2) [27, p. 42], the sorting criteria (*always* ascending) are copied unchanged (well, save for normalization) into a FerryCore's for expression (which also allows order by).

According to the typechecking rule (ForExpr-2) [27, p. 52], each sorting criteria expression must evaluate to an atomic value.

It is not clear if order by clause influences typing, referring to other languages (LINQ), it should not, but the expansion of sortBy macros mentioned in [27, p. 44] puts confusion.

4.1.6 Type signature of Ferry's groupBy macro

Before discussing the type signature of Ferry's group by clause (Sec. 4.1.7), that of the groupBy macro is discussed first.

Table 4.4: groupBy macro

$$\begin{array}{lll} \textit{groupBy} & :: & (\quad (b_1 \ldots b_n) \rightarrow (a_1 \ldots a_m), \\ & & \quad [(b_1 \ldots b_n)] \\ & &) \rightarrow [([b_1] \ldots [b_n])] \end{array}$$

The desugaring [27, p. 43] of Ferry's groupBy macro (whose type signature is shown in Table 4.4) contains a usage of map (itself a macro), unzip (another macro), and the groupWith built-in function, as follows:

```
map ( y -> unzip(y) ,
      groupWith ( x -> g, e )
    )
```

Unlike the groupBy function and the group by clause, the groupWith built-in function is not desugared, but directly translated into an algebraic query plan [27, p. 130].

Given the type signature of groupWith, the constraints shown in Table 4.5 are imposed. In terms of the unexpanded expression (groupBy(x -> g, e)), the detailed types that x, g, and e must have are thus shown also on Table 4.5.

4.1.7 Type signature of Ferry's group by clause

On [27, p. 41], the desugaring of Ferry's group by clause is shown, as reproduced in Table 4.6.

Table 4.5: Types in groupBy's macro expansion

e	::	$[(b_1 \dots b_n)]$
x	::	$(b_1 \dots b_n)$
g	::	$(a_1 \dots a_m)$
y	::	same as e
unzip(y)	::	$([b_1] \dots [b_n])$

Table 4.6: Desugaring of Ferry's group by

source	for v in	e1	group by $e_{g1} \dots e_{gn}$ *... more ForClauses*
target	for v in	groupBy	(v -> ($e_{g1} \dots e_{gn}$) , e1) *... more ForClauses*

In words, a Ferry for with a group by clause is desugared into another Ferry for but with a source collection given by applying the groupBy macro to the original source collection. Because of the type signature of such macro, the detailed types in the original (non-desugared) for are those given on Table 4.7, or

$$\frac{TE \vdash e_1 : [(b_1, \dots, b_n)],\ \forall_{j=1}^{n} TE[v : (b_1, \dots, b_n)] \vdash e_{gj} : a_j}{TE \vdash \mathsf{groupBy}(v \leftarrow (e_{g1}, \dots, e_{gn}), e_2) : [([b_1], \dots, [b_n])]}$$

Table 4.7: Types in group by clause

e_1	::	$[(b_1 \dots b_n)]$
v	::	$(b_1 \dots b_n)$
e_{gi}	::	a_i

4.2 Detailed typing of Ferry's for

Taking into account the discussion of the previous sections, type derivation of Ferry's ForExp is given below

$$\begin{aligned} & \forall e \in \text{ForExp}, \\ e \quad = \quad & \text{for } v_1 \text{ in } e_{v1}, \dots, v_n \text{ in } e_{vn} \\ & (\text{where } E_{e_{w1}})^{?} \\ & \Big(\text{group by } e_{g1}, \dots, e_{gl}\ \ (\text{where } e_{w2})^{?}\Big)^{?} \\ & (\text{order by } e_{o1}, \dots, e_{ok})^{?} \\ & \text{return } e_r \ \Rightarrow \end{aligned}$$

$$\frac{\begin{array}{l}\forall_{j=1}^{n} TE[v_i : t_i]_{i=1}^{j-1} \vdash e_{vj} : [t_j], \\ \left(\, TE[v_j : t_j]_{j=1}^{n} \vdash e_{w1} : bool \, \right)^{?}, \\ \left(\begin{array}{l} TE[v_j : t_j]_{j=1}^{n-1} \vdash e_{vn} : [(b_1, \ldots, b_m)], \\ \forall_{j=1}^{l} TE[v_i : t_i, v_n : (b_1, \ldots, b_m)]_{i=1}^{n-1} \vdash e_{gj} : a_j, \\ \left(\, TE[v_j : t_j, v_n : ([b_1], \ldots, [b_m])]_{j=1}^{n-1} \vdash e_{w2} : bool \, \right)^{?} \; ^{?}, \end{array} \right. \\ \left(\, \forall_{j=1}^{k} TE[v_i : t_i, v_n : t_n \mid ([b_1], \ldots, [b_m])]_{i=1}^{n-1} \vdash e_{oj} : a_j \, \right)^{?}, \\ TE[v_j : t_j, v_n : t_n \mid ([b_1], \ldots, [b_m])]_{j=1}^{n-1} \vdash e_r : t_r \end{array}}{TE \vdash e : t_r}$$

4.3 Prototype implementation

The example given below illustrates our Scala-based typechecker for Ferry language by returning the type ascribed (with some intermediate expression types) for the longest example of [27]

```
// Initial query expression aimed to retrieve a list of two employees' names
// with info about their salaries who earn less money than other employees
// in their department.
e' =
  let e = table Employees (id int, dept string, salary int) with keys ((id)) in
      for x in e group by x.dept return ( the(x.dept), take(2,
            for y in zip(x.id,x.salary)
                  order by y.2 descending
                        return y)
                                                                    )
//      e  :: [('id' int, 'dept' string, 'salary' int)]
//      x  :: ('id' int, 'dept' string, 'salary' int)
//      x  :: ('id' [int], 'dept' [string], 'salary' [int])
//            ( as a result of variale reassignment by 'group by' clause)
//      y  :: ('id' string, 'salary' string)

// e' :: [(string, [('id' string, 'salary' string)])]
```

Chapter 5

Scala to Ferry

Contents

Listing 5.1: Before desugaring of Scala comprehensions

```
// from item in items
// join entry in statusList on item.ItemNumber equals entry.ItemNumber
// select new Temp(item.Name, entry.InStock)
    val res10 =   for (item <- items;
                       val outerKey = item.ItemNumber ;
                       entry <- statusList ;
                       if outerKey == entry.ItemNumber
                  ) yield new Temp(item.Name, entry.InStock)
```

Listing 5.2: After desugaring of Scala comprehensions

```
    val res10 = items.map(((item) => { val outerKey = item.ItemNumber;
                                       scala.Tuple2(item, outerKey)
                                     })
                         ).flatMap(((x$9) => x$9: @scala.unchecked match {
        case scala.Tuple2((item @ _), (outerKey @ _))
    => statusList.filter (((entry) => outerKey.$eq$eq(entry.ItemNumber)))
                    .map(((entry) => new Temp(item.Name, entry.InStock)))
    }));
```

5.1 Scala queries: ad-hoc or translated from LINQ

Without the ScalaQL extension, the Scala compiler desugars comprehensions into applications of map, filter, and flatMap, as can be seen in Listing 5.1 (before desugaring) and in Listing 5.2 (afterwards). A translation into Ferry taking the desugared version as starting point would need to consider many more cases. For this reason, ScalaQL takes the comprehension formulation as starting point.

5.2 Ensuring isomorphism of types between Scala subset and Ferry

The translation from Scala to Ferry abides by an *isomorphism* of types between a chosen Scala subset and Ferry. This implies that the aimed mapping between Scala and Ferry types has to be *bijective* and the *correspondence between Scala and Ferry operators* has to be defined.

Additionally, the translation from Scala to Ferry guarantees the resulting Ferry query to be type-safe, and to evaluate without runtime errors for all inputs on which the original Scala query would have terminated without exceptions.

5.2.1 Scala tuples

The direct translation of Scala tuples into Ferry's native counterpart does not satisfy the aforementioned property of bijectivity as SQL engines do not support nested tuples. Instead, a *flattening encoding* has to be applied before evaluating on the server-side nested-tuple expressions. Instead, the expression $(e_1, (e_2, e_3), e_4)$ is encoded as (e_1, e_2, e_3, e_4). That value, together the with infor-

mation on the expected type, in the example $(t_1, (t_2, t_3), t_4)$, are used to recover from the relational representation the Ferry-level counterpart.

Without the second piece of information (expected type), the sketched *flattening* encoding would not be injective, as two different Scala tuples would correspond to the same Ferry tuple, for example

$$[\![((e_1, e_2), e_3, e_4)]\!] = [\![(e_1, (e_2, e_3), e_4)]\!] = (e_1, e_2, e_3, e_4),$$

Summing up, injectivity is preserved by performing Scala tuple *encoding* at compile time in conjunction with an *encoding-aware translation*:

$$[\![((e_1, e_2), e_3, e_4).1]\!] = ((e_1, e_2, e_3, e_4).1, (e_1, e_2, e_3, e_4).2)$$

5.2.2 Scala lists, sets, and maps

Scala collections are translated to Ferry counterparts under the assumption of being *homogeneous*, i.e. no instances of proper subtypes are contained, a property that is checked if necessary at runtime, leading to classifying queries into Level 3 or Level 4 (Sec. 1.5).

Scala *sets* are represented as Ferry lists, together with a translation of operators on them that preserve set semantics (*e.g.* an equality test disregards order, and set insertion implies that no duplicates will appear in the result).

Similarly, Scala *maps* are represented as Ferry lists of two-element tuples with the first element encapsulating a key, and the second element having any of the Scala types amenable to encoding. In case the key itself is a tuple, tuple flattening is applied.

5.2.3 Scala case classes and anonymous classes

Scala case classes and final anonymous classes promote an object abstraction for records known from functional query languages

```
case class Employee (id: Int, name: String, dept: String, salary: Int)

val empl1 = Employee(1,'John', 'US', 1200)
// or
val empl2 = new { val id = 1; val name = 'John'; val dept = 'US'; val salary = 1200}
```

The concept of records is partially supported by Ferry tuples

$$\text{let } e = \text{table } \mathit{Employees}\ (\mathit{id}\ \mathit{int},\ \mathit{name}\ \mathit{string},\ \mathit{dept}\ \mathit{string},\ \mathit{salary}\ \mathit{int}) \text{ in}$$

$$\text{map}(v\ \rightarrow\ e.\mathit{name},\ e).$$

In order to add to our supported Scala subset the aforementioned user defined Scala types, records are adopted as an extension of Ferry tuples. The grammar production for a record literal is:

$$e \in \text{RecordExp}\ ::=\ (\ (TC_{\mathit{tablecolname}}\ E_e)\ ^{2..*<\mathit{separator:},>}\ ),$$

with the corresponding typing rule

$$\forall e \in \text{RecordExp},\ e = (c_1\ e_1, \ldots, c_n\ e_n) \Rightarrow$$

$$\frac{\forall_m^{j=1} TE \vdash e_j : b_j}{TE \vdash e : (b_1, \ldots, b_m)},$$

i.e. no new type is introduced and *flattening* as for tuples is applied. Column names are not stored in the relational representation, they are translated away after type checking is over.

The translation of Scala case classes and anonymous classes are considered under assumption of persistable Scala values and fields containing no references to other instances of user defined types (case classes and anonymous classes).

5.2.4 Scala enumerations

A value from a Scala enumeration is encoded as an integer.

5.3 Scala operators and their Ferry counterparts

5.3.1 Equality tests for lists and tuples

Scala provides *equality* operations for comparing values of structured types, (*e.g.* tuples and lists, which belong to the translatable Scala subset). However, out-of-the-box, Ferry allows testing for equality just atomics. Additionally, (a) the type components of a Ferry structured type are known at compile-time; and (b) structural equality for structured types sharing the same structure can be checked by and-ing the equality tests for components. The depth of component-nesting is finite, and known at compile.

We therefore *extend* the Ferry language by allowing expressions of the form $e_A == e_B$ where e_A and e_B are structured types. Our translator takes care of expanding that comparison into a longer form, as illustrated below.

- **Lists case**: xs == ys where xs and ys are lists of atomics is expanded to length(xs) == length(ys) and let nonEqualItems = filter(v -> v.1 != v.2, zip(xs,yz)) in length(nonEqualItems) == 0, and xs != ys is expanded to length(xs) != length(ys) or let nonEqualItems = filter(v -> v.1 != v.2, zip(xs,yz)) in length(nonEqualItems) != 0

- **Tuples case**: xs == ys where xs and ys are tuples of atomics is expanded to xs.1 == ys.1 and ... and xs.n==ys.m and ... and xs.N == ys.M; well-typedness of xs == ys is checked by N==M and $\forall_{n=1}^{N} \forall_{m=1}^{M}$ xs.n == xs.m, n $\rightarrow c_1$, m $\rightarrow c_2 \Rightarrow c_1 == c_2$, and xs != ys is expanded to xs.1 != ys.1 or ... or xs.n!=ys.m or ... or xs.N!=ys.M;

Please notice that preserving the semantics of equality in the translation has to take into account the case where operands are *not* of exactly the same type. For example, when enclosed in a query, a comparison of a list and a tuple can be determined statically to evaluate to false. Detecting these situations is taken care of by typechecking the resulting Ferry query. Additionally, comparing two tuples for equality requires additional checks by ScalaQL due to *flattening*, *e.g.*

$$[\![((t_1, t_2), t_3) == (t_1, (t_2, t_3))]\!] \neq (t_1, t_2, t_3) == (t_1, t_2, t_3)$$

that becomes more complicated when using operators such as zip.

5.3.2 Equality tests for sets and maps

- **Sets case** : xs == ys where xs and ys are list-based represention of sets, is expanded to length(xs) == length(ys) and let included = map ($v_1 \rightarrow$ length(filter($v_2 \rightarrow v_2 == v_1$, ys)), xs) in length(filter($v \rightarrow v! = 0$, included)) == length(xs), where an equality test $v_2 == v_1$ is adjusted with respect to the structured or atomic type of elements of the homogeneous collections xs and ys. Having xs and ys of type Set[A] implies that they will have no duplicates in a Ferry database.

- **Maps case**: xs == ys where xs and ys are list-based represention of maps, is expanded to length(xs) == length(ys) and let included = map ($v_1 \rightarrow$ length(filter($v_2 \rightarrow v_2 == v_1$, ys)), xs) in length(filter($v \rightarrow v! = 0$, included)) == length(xs), where an equality test $v_2 == v_1$ refers to the equality test provided for Ferry tuples, i.e. the(v_2.1) == the(v_1.1) and v_2.2 == v_1.2 adjusted in turn with respect to the structured or atomic type of their first and second elements.

5.3.3 Ferry counterparts to equality tests for Scala case classes and anonymous classes

Even though both Scala case classes and anonymous classes are represented as records in Ferry that can be applied an equality test defined for Ferry tuples, the semantics of equality is preserved only for Scala case classes but not for Scala anonymous classes (in this case, == compares two objects by their identities and provides reference equality if not overridden), i.e.

$$[\![cc_1 == cc_2]\!] = [\![cc_1]\!] == [\![cc_2]\!]$$

but

$$[\![ac_1 == ac_2]\!] \neq [\![ac_1]\!] == [\![ac_2]\!]$$

where $[\![cc_i]\!]$ and $[\![ac_j]\!]$ stand for a variable or the corresponding encoding.

5.3.4 Precedence test for structured types

The same technique as in the case of equility expansion for structured types can be applied for Ferry relational operations <, >, <=, >= as counterparts to the relational operations defined for Scala tuples and lists. In contrast to equality operators, Scala relational operators are defined for tuples and lists and typechecked by compiler. The case of comparison of lists with different lengths is ruled out by Ferry typer run at compile time.

5.3.5 Operators with direct counterparts

In some cases, a direct translation from Scala into Ferry is possible, as shown in Table 5.1:

Some Scala operators on collections have a direct counterpart in Ferry, *e.g.* length. However, even for similarly named operators there are side-conditions (in the form of typing rules for Ferry) that may preclude a one-to-one translation. In these cases, additional encodings or query rewritings are necessary. For example, Scala's zip operator has no restriction on the item types of its argument

Table 5.1: Operators with a direct counterpart

Scala, where	Ferry, where
e, e_1: List[A], e_2: List[B]	s: [t], s_1: [t_1], s_2: [t_2]
c: C[A] $\in$ {List[A], Set[A], Map[K, A]}	
n : int	n : int
e.length : Int	length(s) : int
c take n : C[A]	take(n,s) : [t]
c drop n : C[A]	drop(n,s) : [t]
e_1 zip e_2 : List[Tuple2[A,B]]	zip((s_1,s_2)) : [(t_1,t_2)]

collections, while Ferry's output (a pair) cannot internally be a non-flat tuple. Without flattening support (Sec. 5.2), the one-to-one translation shown below would fail to typecheck (due to zip(Employees, salaries), because Employees is a list of tuples):

```
val salaries : List[Int] = ...
val Employees1 = Employees zip salaries
// Ferry : let Employees = table EmployeesTab(id int, name string, dept string)
//         in zip(Employees, salaries)
val firstEmployeeName = Employees1(0).2
```

Moreover, the assumption of having homogeneous lists implies preserving homogeneity by operators, *e.g.* append.

Scala operators taking a function as an argument There are some higher-order operators in Scala that allow passing a function as its argument Table 5.2. Note that in the case of c being of type Map[K, A] function f_m can be not only pair-to-pair function.

5.3.6 Operators with almost direct counterparts

Other nodes in the input AST can be translated into a bunch of invocations of Ferry functions, as shown in Table 5.7, Table 5.8, Table 5.9, Table 5.10.

Translation with almost direct counterparts is illustrated with the next example of counting employees fulfilling some condition (in the case, being assigned to a particular department)

```
val numberOfEmployeesAtUK = Employees count (employee => employee.dept == "UK")
// Ferry : let Employees = table EmployeesTab(id int, name string, dept string)
//         in length( filter (employee -> employee.dept == "UK", Employees))
```

Scala operators taking a function as an argument There are some higher-order operators in Scala that allow passing a function as its argument Table 5.6.

Table 5.2: Higher-order operators with a direct counterpart

Scala, where	Ferry, where
c: C[A] $\in$ {List[A], Set[A]}	s: [t]
m: Map[K, A]	
f: A $\rightarrow$ B	s': t_1
f_m: Tuple2[K, A] $\rightarrow$ Tuple2[K_1, B] | Tuple2[K, A] $\rightarrow$ B	
p: A $\rightarrow$ Boolean	s_{test}: bool
p_m: Tuple2[K, A] $\rightarrow$ Boolean	
c map f : C[B]	map($v \rightarrow s'$, s) : [t_1]
(m map f_m : Map[K_1, B] | List[B])	
c filter p : C[A]	filter($v \rightarrow s_{test}$, s) : [t]
(m filter p_m : Map[K, A])	

5.3.7 Operators with no counterparts

Unlike a functional language, Ferry doesn't allow accumulating the results of binary operation application between successive elements of lists except for the cases of some built-in operations sum, min, max or sortBy. Therefore, some Scala operations such as shown in Table 5.3 are not translatable for the general case and translation can be considered for a few special cases, *e.g.*

```
val e2 = e1.reduceLeft((v1,v2) => if (v1 < v2) v1 else v2 ) // Ferry: e2 = min(e1)
val e3 = e1.reduceLeft((v1,v2) => if (v1 > v2) v1 else v2 ) // Ferry: e3 = max(e1)
val e4 = e1.reduceLeft((v1,v2) => v1 + v2 ) // Ferry: e4 = sum(e1)

val e5 = e1.reduceLeft((v1,v2) => if (e_g[v1] < e_g[v2]) v1 else v2 )
// Ferry: e5 = let e1' = concat(sortBy(v -> e_g[v],e1)) in nth(1, e1')

val e6 = e1.reduceLeft((v1,v2) => if (e_g[v1] > e_g[v2]) v1 else v2 )
// Ferry: e6 = let e1' = concat(sortBy(v -> e_g[v],e1)) in nth(length(e1'), e1')
```

provided that $e1$ is a list of atomics, and e_g is of atomic type.

Table 5.3: Not translatable operators

Scala, where e, e_1, e_2: List[A]	Ferry, where s, s_1, s_2: [t]
e_1 intersect e_2 : List[A]	let s'_1 = map($v \rightarrow$ (length(v), v), groupWith($v \rightarrow$ (v.1,...,v.N), s_1)), s'_2 = map($v \rightarrow$ (length(v), v), groupWith($v \rightarrow$ (v.1,...,v.N), s_2)) in concat(map($v \rightarrow$ if nth(1, v).1 > nth(2, v).1 then nth(2, v).2 else nth(1, v).2, filter($v \rightarrow$ length(v) > 1, groupWith($v \rightarrow$ let v_1 = the(v.2) in (v_1.1,...,v_1.N), append(s'_1, s'_2)))))
e_1 diff e_2 : List[A]	let s'_1 = map($v \rightarrow$ (length(v), v), groupWith($v \rightarrow$ (v.1,...,v.N), s_1)), $s_{intersec}$ = map($v \rightarrow$ (length(v), v), groupWith($v \rightarrow$ (v.1,...,v.N), ⟦ e_1 intersect e_2 ⟧)) in concat(map($v \rightarrow$ if length(v) == 1 then nth(1, v).2 else take(nth(1, v).1 - nth(2, v).1, nth(1, v).2) groupWith($v \rightarrow$ let v_1 = the(v.2) in (v_1.1,...,v_1.N)), append(s'_1, $s_{intersect}$)))
e_1 ++ e_2 | e_1 union e_2 : List[A]	let s'_1 = map($v \rightarrow$ (length(v), v), groupWith($v \rightarrow$ (v.1,...,v.N), s_1)), s'_2 = map($v \rightarrow$ (length(v), v), groupWith($v \rightarrow$ (v.1,...,v.N), s_2)) in concat(map($v \rightarrow$ if length(v) == 1 then nth(1, v).2 else if nth(1, v).1 < nth(2, v).1 then nth(2, v).2 else nth(1, v).2, groupWith($v \rightarrow$ let v_1 = the(v.2) in (v_1.1,...,v_1.N), append(s'_1, s'_2))))
e.removeDuplicates : List[A]	map($v \rightarrow$ the(v), groupWith($v \rightarrow$ (v.1,...,v.N), s))

This is often the case with Scala's reduceLeft that recursively applies a given binary operation between successive elements of a sequence, as shown below when summing up employee salaries.

```
val sumOfSalaries = Employees.map(employee =>
employee.salary ).reduceLeft(( salary1 , salary2 ) => salary1 + salary2 )
// Ferry : let Employees = table EmployeesTab(id int, name string,
//                                            dept string , salary int)
//         in sum( map(employee -> employee.salary, Employees) )
```

Scala's intersect, diff and ++/union imply multi-set intersection, difference, and union, the given translation to Ferry is possible under restriction of e_1, e_2 being lists of tuples with elements of atomic types or lists of atomics.

5.4 Scala's groupBy

Representation of group by on Scala collections defined by semantics of groupBy (see Listing 5.3) that is followed by translating it to Ferry.

Scala's groupBy considered under constraint imposed by Ferry's groupBy

$$f : v \rightarrow (e_{g1}, ..., e_{gn}) \mid e_{gi} : \ a_i,$$

i.e. a tuple of expressions evaluated with atomic values (however, having equality tests for Ferry structured types promotes redefining Ferry's groupBy to cover more general case of a grouping discriminator):

$$\mathsf{val}\ m = e.\mathsf{groupBy}(v \rightarrow f(v)),$$

where m: Map[Tuple$seq, List[A]] provided that e: List[A].

The final translation with respect to provided semantics of both Scala groupBy and Ferry groupBy is given below

$$\begin{aligned}[[e.\mathsf{groupBy}(v \ \Rightarrow \ (e_{g1}, \ldots, e_{gn}))]] = \ & \mathsf{let}\ e' = \mathsf{map}(v \rightarrow ([(e_{g1}, \ldots, e_{gn})], [v]), e)\ \mathsf{in} \\ & \mathsf{map}(v \rightarrow (\mathsf{the}(v.1), \mathsf{concat}(v.2)), \\ & \qquad \mathsf{groupBy}(v \rightarrow \mathsf{the}(v.1), e'))\end{aligned}$$

The result of Ferry expression has a type of $[([(a_1, \ldots, a_n)], [t])]$ that is similar to what is returned by the Scala groupBy. Such a structure is derived by creating a list containing $([(e_{g1}, \ldots, e_{gn})], [v])$ where boxing a tuple of key expressions guarantees under *flattening* its first position, and boxing each tuple from original list will guarantee $[([(a_1, \ldots, a_n)], [t])]$ after applying the corresponding unboxing and concatenation at the end.

The resulting groups returned by Scala groupBy may be accessed, for example, by

```
val e1 = m apply k
```

or, equivalently,

```
val e2 = for ( x <- m; if x._1 == k ) yield x._2
```

Listing 5.3: groupBy in Scala 2.8

```
/** Partition this traversable into a map of traversables
 *  according to some discriminator function.
 *  @invariant    (xs partition f)(k) = xs filter (x => f(x) == k)
 *
 *  @note This method is not re-implemented by views. This means
 *        when applied to a view it will always force the view and
 *        return a new collection.
 */
def groupBy[K](f: A => K): Map[K, This] = {
  var m = Map[K, Builder[A, This]]()
  for (elem <- this) {
    val key = f(elem)
    val bldr = m get key match {
      case None => val b = newBuilder; m = m updated (key, b); b
      case Some(b) => b
    }
    bldr += elem
  }
  m mapValues (_.result)
}
```

The corresponding translation of Scala apply defined for maps is given below

$$[\![m \text{ apply } k]\!] = \quad \text{nth}(1, \text{for } x \text{ in } m \text{ where the}(x.1) == k \text{ return } x.2)$$

where m refers to the general case of Ferry encoding defined for Scala Map[K,V] type.

5.5 Scala's sortWith

Scala 2.8 sortWith is shown in Listing 5.4, reproduced from the Scala SVN repo[1].

- In Scala one can customize both the expressions to be compared for lessThan, and the comparator function. In Ferry, no custom comparator function is accepted by sortBy.
- seeSec. 5.3.1

Scala's sortWith is translateded under constraint imposed by Ferry's sortBy

$$f : (v_1, v_2) \rightarrow (e_{g1}[v_1], ..., e_{gn}[v_1]) < (e_{g1}[v_2], ..., e_{gn}[v_2]) \mid e_{gi} : \ a_i,$$

i.e. a comparison with comparator $<$ of tuples of expressions evaluated with atomic values (as also pointed for Ferry groupBy, having precedence tests for Ferry structured types promotes redefining Ferry sortBy to cover more general case of a sorting criteria):

```
val e' = e.sortWith( (v1,v2) => f(v1,v2) ),
```

[1] http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk/src/library/scala/collection/generic/TraversableTemplate.scala

Listing 5.4: sortWith in Scala 2.8

```
/** Sort the traversable according to the comparison function
 * <code>&lt;(e1: a, e2: a) =&gt; Boolean</code>,
 * which should be true iff <code>e1</code> is smaller than
 * <code>e2</code>.
 * The sort is stable. That is elements that are equal wrt 'lt' appear in the
 * same order in the sorted traversable as in the original.
 *
 * @param lt the comparison function
 * @return   a traversable sorted according to the comparison function
 *           <code>&lt;(e1: a, e2: a) =&gt; Boolean</code>.
 * @ex <pre>
 *   List("Steve", "Tom", "John", "Bob")
 *     .sort((e1, e2) => (e1 compareTo e2) &lt; 0) =
 *   List("Bob", "John", "Steve", "Tom")</pre>
 */
def sortWith(lt : (A,A) => Boolean): This = {
  val arr = toArray
  Array.sortWith(arr, lt)
  val b = newBuilder[A]
  for (x <- arr) b += x
  b.result
}
```

where e': List[A] provided that e: List[A].

The corresponding translation into Ferry with respect to semantics of both Scala sortWith and Ferry sortBy is given below

$$[\![e.\mathsf{sortWith}((v_1, v_2) \Rightarrow (e_{g1}[v_1], ..., e_{gn}[v_1]) < (e_{g1}[v_2], ..., e_{gn}[v_2]))]\!] =$$

$$\mathsf{concat}(\mathsf{sortBy}(v \rightarrow (e_{g1}[v], ..., e_{gn}[v]),\ e))$$

The resulting Ferry expression performs *stable sorting*, i.e. those items that the comparator function reports as equal are listed in the same order as they appeared in the input.

5.6 Custom Scala's operator orderBy

Intermediate custom Scala operator orderBy defined in Listing 5.5 is used that is further replaced by Scala 2.8 operator sortWith Table 5.4.

The aforementioned translation is possible under constraint of $e_{gi} : a_j$ imposed by Ferry's sortBy.

5.7 Definition of translatable comprehensions

5.7.1 Syntax

The pointed subset of context-free syntax appears in Table 5.5.

Listing 5.5: Custom operator orderBy in Scala

```
def orderBy(lt : A => B): This = {
  val arr = toArray
  val lt1 = (e1: A, e2: A) => lt(e1) < lt(e2)
  Array.sortWith(arr, lt1)
  val b = newBuilder[A]
  for (x <- arr) b += x
  b.result
}
```

Table 5.4: Scala's custom operator orderBy

Scala, where e : List[A]	Ferry, where s : [t]
e.orderBy($v \Rightarrow (e_{g1}, \ldots, e_{gn})$) $\mid e_{gi} : a_j$ e.sortWith($(v\$1, v\$2) \Rightarrow$ $(e_{g1}[v\$1], \ldots, e_{gn}[v\$1])$ $<$ $(e_{g1}[v\$2], \ldots, e_{gn}[v\$2])$)	sortBy($v \rightarrow (e_{g1}[v], \ldots, e_{gn}[v])$, s)

5.7.2 Handling Patterns

Scala allows usage of *patterns*, in particular *variable patterns*, that promotes effective variable bindings and pattern matching. Patterns are used in (a) variable definitions that are also a part of Scala ForExpr; (b) Scala for-comprehension generators;(c) case clause definitions being appeared in pattern matching constructs with respect to the chosen syntax subset. The expected patterns are restricted to represent atomic and structured types (tuples and lists) according to available Ferry types, i.e.

```
// case v @ ( ..., v1 @_, ..., l, ..., _ ) guard => body
case CaseDef( Bind(v, Apply(Select(_, Tuple$seq),
                            List (..., Bind(v1, Ident(_ )),...,
                                  Literal (l ), ..., Ident(_) ))),
              guard,
              body ) => ...

// case v @ List( ..., v1 @_, ..., l, ..., _ ) guard => body
case CaseDef( Bind(v, Apply(Ident(Name("List")),
                            List (..., Bind(v1, Ident(_ )),...,
                                  Literal (l ), ..., Ident(_) ))),
              guard,
              body ) => ...
```

Handling *patterns in ValDef* is shown in the following table:

Scala	val V_{v_1} @ (V_{v_2} , _ ,. . .) = E_e val V_{v_1} @ (V_{v_2}@_, _ ,. . .) = E_e
Ferry	let $v_1 = E_e$, $v_2 = e.1$ in . . .

Handling *patterns in Generator* is shown in the following table:

Scala	V_{v_1} @ (V_{v_2}, _ ,. . .) $\leftarrow E_e$	(V_{v_2}, _ ,. . .) $\leftarrow E_e$
	V_{v_1} @ (V_{v_2}@_, _ ,. . .) $\leftarrow E_e$	$V_{v\$seq}$ @ (V_{v_2}@_, _ ,. . .) $\leftarrow E_e$
Ferry	v_1 in E_e . . . let $v_2 = v_1.1$ in . . .	$v\$seq$ in E_e . . . let $v_2 = v_{\$seq}.1$ in . . .

5.7.3 Handling Scala' pattern matching

Scala's pattern matching is translated to Ferry IFExp. Such a translation implies restriction of expression types in case clause blocks that have to be assigned to the same type. Handling *pattern matching* is shown in the following table.

Scala	E_{e_1} match { case V_{v_1} @ (V_{v_2}, L_l ,. . .) $\Rightarrow E_{e_2}$
	case V_{v_1} @ (V_{v_2}@_, L_l ,. . .) $\Rightarrow E_{e_2}$
Ferry	if *conds* then let *bindings* E_{e_2} else . . . *conds* ::= $e_1.2 == l$ *bindings* ::= $v_1 = e_1, v_2 = e_1.1$

5.7.4 Handling Scala's Block

Handling scala's blocks according to the intended scala subset is shown in the following table:

Scala	{
	val $V_{v_1} = E_{e_1}$
	. . .
	val $V_{v_N} = E_{e_N}$
	E_e
	}

Ferry	let *bindings* in e *bindings* ::= $v_1 = e_1, \ldots, v_2 = e_N$

5.7.5 Handling Scala's IFExpr

Handling scala's IFExp is shown in the following table, the translation is applied to $e \in$ IFExp such that $TE \vdash e_1 : t$ and $TE \vdash e_2 : t$.

Scala	if $E_{e_{bool}}$ E_{e_1} else E_{e_2}
Ferry	if e_{bool} then e_1 else e_2

5.7.6 Handling Scala's ForExpr

```
case class ValFrom(pos:Position, pat: Tree, rhs: Tree) extends Enumerator
case class ValEq(pos: Position , pat: Tree, rhs: Tree) extends Enumerator
case class  Filter ( test :Tree) extends Enumerator

case class ForComp(enums:List[Enumerator]) extends TermTree
```

Handling scala's ForExp is shown in the following tables:

Scala	for ($V_v \leftarrow E_{e_1}$) yield E_{e_2}
Ferry	for v in e_1 return e_2

Scala	for ($V_{v_1} \leftarrow E_{e_1}$; $V_{v_2} \leftarrow E_{e_2}$;...) yield E_e
Ferry	concat(for v_1 in e_1 return concat(for v_2 in e_2 return...))

Scala	for ($V_v \leftarrow E_{e_1}$; if $E_{e_{bool}}$) yield E_{e_2}
Ferry	for v in e_1 where e_{bool} and ... return e_2

Scala	for ($V_v \leftarrow E_{e_1}$; val $V_{v_1} = E_{e_1}$; ...;val $V_{v_N} = E_{e_N}$; ...) yield E_{e_2}
Ferry	for v in e_1 return let $v_1 = e_1, \ldots, v_N = e_N$ in e_2

5.7.7 Extended example of translating Scala for-comprehention

More complex translation rules are required for Scala comprehension, as rewriting their components (*qualifiers*, which can be *generators*, *guards*, and *let-declarations*) has to take into account the variables in scope.

```
val namesWithSalaries1 = for (employee <- Employees)
                         yield (employee.name,employee.salary)
// Ferry : let Employees =
//           table EmployeesTab(id int, name string, dept string, salary int)
//           in for employee in Employees return (employee.name, employee.salary)

val namesWithSalaries2 = for (employee <- Employees;
                                  val salary = employee.salary; val name = employee.name)
                         yield (name, salary)
// Ferry : let Employees = table EmployeesTab(id int, name string,
//                                 dept string, salary int)
//           in for employee in Employees return
//                 let name = employee.name, salary = employee.salary
//                 in (name, salary)

val namesWithSalariesLessThan =
   for (employee <- Employees; val salary = employee.salary;
        val name = employee.name; if salary < 200)
   yield (name, salary)
// Ferry : let Employees = table EmployeesTab(id int, name string,
//                                 dept string, salary int)
// in for employee in Employees
//         where
//             let name = employee.name, salary = employee.salary
//             in salary < 200
//         return
//             let name = employee.name, salary = employee.salary
//             in (name, salary)
```

Some additional constructs of Scala are also handled, for example, *patterns* that can be a part of *generators*, *let-declarations*:

```
val namesWithSalaries = for ( (_, name, _, salary) <- Employees)
                        yield (name, salary)
// Ferry : let Employees = table EmployeesTab(id int, name string,
//                                 dept string, salary int)
//   in for employee$1 in Employees return
//           let name = employee$1.name, salary = employee$1.salary
//           in (name, salary)
```

Table 5.5: The pointed Scala subset

Id	$\in$	Identifier, $V \in$ VarId, $SId \in$ StableId,
P	$\in$	Path, $L \in$ Literal (*constrained by Ferry atomics*)
E	$\in$	Exp = IFExp $\cup$ ForExp $\cup$ InfixExp $\cup$ MatchExp
$IFE \in$ IFExp	::=	$\mathsf{if}\ (\ E_{e_1}\ )\ nl^{\ 0..*}\ E_{e_2}\ (\ semi^{\ 0..1} \mathsf{else}\ E_{e_3}\ )^{\ 0..1}$
$FE \in$ ForExp	::=	$\mathsf{for}\ (\ (ENs_{enums}) \mid \{ENs_{enums}\}\ )\ nl^{\ 0..*} \mathsf{yield}\ E_e$
$ME \in$ MatchExp	::=	$IE_{infixexp}\ \mathsf{match}\ \{\ CC^{1..*}_{caseclause}\ \}$
$IE \in$ InfixExp	::=	$PE \mid IE\ Id\ nl^{\ 0..1} IE$
$PE \in$ PrefixExp	::=	$(\ \sim, !\ )^{\ 0..1} SE$
$SE \in$ SimpleExp	::=	$\mathsf{new}\ CT_{classtemplate} \mid \{\ BL\ \} \mid SE1$
$SE1 \in$ SimpleExp1	::=	$L \mid P_{path} \mid \_$
		$\mid (\ E^{0..*}_{exprs}{}^{<separator:,>}\ )$
		$\mid SE.Id$
		$\mid SE1\ (\ AE^{0..*\ <separator:,>}_{argexprs}\ )$
$AE \in$ ArgExpr	::=	$(\ (B^{1..*\ <\text{separator:,}>}_{\text{bindings}}) \mid Id \mid \_\ ) \Rightarrow AE_{argexp}$
$BL \in$ Block	::=	$(\ BS_{blockstats} semi\ )^{\ 0..*} E_e$
$BS \in$ BlockStat	::=	VD_{valdef}
$ENs \in$ Enumerators	::=	$G_{generator}\ (\ semi\ EN_{enums})^{\ 0..*}$
$EN \in$ Enumerator	::=	$G_{generator} \mid GD_{guard} \mid \mathsf{val}\ Pat1_{pattern} = E_e$
$G \in$ Generator	::=	$Pat1_{pattern} \leftarrow E_e\ GD_{guard}^{\ 0..1}$
$CC \in$ CaseClause	::=	$\mathsf{case}\ Pat_{pattern}\ G^{\ 0..1} \Rightarrow BL_{block}$
$GD \in$ Guard	::=	$\mathsf{if}\ IE_{infixexp}$
$Pat \in$ Pattern	::=	$Pat1^{\ 1..*\ <separator:\ \mid\ >}_{patterns}$
$Pat1 \in$ Pattern1	::=	$V_{varid}\ (\ @\ SP\ )^{\ 0..1} \mid SP$
$SP \in$ SimplePattern	::=	$\_ \mid V_{varid} \mid L \mid SId_{stableid}$
		$\mid SId_{stableid}\ (\ Pats^{0..*}_{\text{patterns}}\ )$
		$\mid SId_{stableid}\ (\ Pats^{0..*}_{\text{patterns}}\ )\ (\ V_{varid}\ @\ )^{\ 0..1} \_ *\ )$
		$\mid (\ Pats_{patterns}{}^{\ 0..1})$
$Pats \in$ Patterns	::=	$Pat_{pattern}{}^{\ 1..*\ <\text{separator: , >}} \mid \_ *$
$VD \in$ ValDef	::=	$\mathsf{val}\ PD$
$PD \in$ PatDef	::=	$Pat1_{patterns}{}^{\ 1..*\ <\text{separator: , >}} = E_e$

Table 5.6: Higher-order operators with almost direct counterparts

Scala, where	Ferry, where
c: C[A] $\in$ {List[A], Set[A]}	s, s_1, s_2: [t]
m: Map[K, A]	
f: A $\rightarrow$ C[B]	s': t_1
f_m: Tuple2[K, A] $\rightarrow$ Map[K_1, B]	
| Tuple2[K, A] $\rightarrow$ List[B]	
p: A $\rightarrow$ Boolean	s_{test}: bool
p_m: Tuple2[K, A] $\rightarrow$ Boolean	
c filterNot p : C[A]	filter($v \rightarrow$ not s_{test}, s) : [t]
(m filterNot p_m : Map[K, A])	
c flatMap f : C[B]	concat(map($v \rightarrow s_2$, s_1)) : [t]
(m flatMap f_m : Map[K_1, B] | List[B])	
c partition p : Tuple2[C[A],C[A]]	(filter($v \rightarrow s_{test}$, s), filter($v \rightarrow$ not s_{test}, s)) : ([t],[t])
(m partition p_m : Tuple2[Map[K, A], Map[K, A]])	
c forall p | m forall p_m: Boolean	length(filter($v \rightarrow$ not s_{test}, s)) == 0 : bool
c exists p | m exists p_m : Boolean	length(filter($v \rightarrow s_{test}$, s)) != 0 : bool
c count p | m count p_m: Int	length(filter($v \rightarrow s_{test}$, s)) : int
c takeWhile p : C[A]	let $prefixs$ = filter($v \rightarrow v.1$! = 0, map($v \rightarrow$ let $sPrefix$ = take(v, s) in if length(filter($v \rightarrow s_{test}$, $sPrefix$)) == length($sPrefix$) then (length($sPrefix$), v) else (0, v), $range[1, \text{length}(s)]$)) in take (nth(length($prefixs$), $prefixs$).2, s) : [t]
(m takeWhile p_m : Map[K, A])	
c dropWhile p : C[A]	let $prefixs$ = filter($v \rightarrow v.1$! = 0, map($v \rightarrow$ let $sPrefix$ = take(v, s) in if length(filter($v \rightarrow$ not s_{test}, $sPrefix$)) == length($sPrefix$) then (length($sPrefix$), v) else (0, v), $range[1, \text{length}(s)]$)) in take (nth(length($prefixs$), $prefixs$).2, s) : [t]
(m dropWhile p_m : Map[K, A])	
c span p : Tuple2[C[A], C[A]]	($[\![e$ takeWhile $p]\!]$, $[\![e$ dropWhile $p]\!]$) : ([t], [t])
(m span p_m : Tuple2[Map[K, A], Map[K, A]]	

Table 5.7: Operators with an (almost) direct counterpart

Scala, where c, c_1, c_2: C[A]$\in$ {List[A], Set[A], Map[K, A]} $from$, to: Int n: Int	Ferry, where s, s_1, s_2: [t] $range[from, to] = [from, \ldots, to]$: [int] n: int
c.isEmpty : Boolean	length(s) == 0 : bool
c.nonEmpty : Boolean	length(s) != 0 : bool
c.size : Int	length(s) : int
c.head : A	nth(1, s) : t
c.tail : C[A]	drop(1, s) : [t]
c.last : A	nth(length(s), s) : t
c.init : C[A]	take(length(s) - 1, s) : [t]
c splitAt n: Tuple2[C[A], C[A]]	(take(n, s), drop(n, s)) : ([t], [t])
c slice ($from$, to) : C[A]	map(v $\rightarrow$ nth(v, s), $range[from,\ to]$) : [t]
c takeRight n : C[A]	drop(length(s) - n, s) : [t]
c_1 sameElements c_2 : Boolean	length(s_1)==length(s_2) and let s_1' = map($v \rightarrow [v]$, s_1), s_2' = map($v \rightarrow [v]$, s_2) in let s' = filter(v $\rightarrow$ $v.1$!= $v.2$, zip((s_1', s_2'))) in length(s') == 0: bool

Table 5.8: Operators with an (almost) direct counterpart for Lists

Scala, where e, e_1, e_2: List[A], x: A $from$, to: Int n: Int	Ferry, where s,s_1,s_2: [t], x: t $range[from, to] = [from, \ldots, to]$: [int] n: int
e apply n | $e(n)$: A	nth(n,s) : t
e_1.lengthCompare e_2 : Int	if length(s_1) < length(s_2) then -1 else if length(s_1) > length(s_2) then 1 else 0 : int
e.indices : List[Int]	$range[1,$ length$(s)]$: [int]
e.isDefinedAt n : Boolean	$n <$ length(s) : bool
e.zipWithIndex : List[Tuple2[A, Int]]	zip((s,$range[1,$ length$(s)]$)) : [(t, int)]
e prefixLength p : Int	let $prefixLens$ = filter($v \rightarrow v$!= 0, map($v \rightarrow$ let $sPrefix$ = take(v, s) in if length(filter($v \rightarrow s_{test}$, $sPrefix$)) == length($sPrefix$) then length($sPrefix$) else 0, $range[1,$ length$(s)]$)) in nth(length($prefixLens$), $prefixLens$): int
e indexWhere p : Int	nth(1, filter($v \rightarrow$ let v_1 = nth(v, s) in s_{test}, $range[1,$ length$(s)]$)) : int
e indexOf x : Int	nth(1, filter($v \rightarrow$ nth(v, s) == x, $range[1,$ length$(s)]$)) : int
e.reverse : List[A]	map($v \rightarrow$ nth(v, s), $range[1,$ length$(s)]$ order by v descending)) : [t]
e_1 startsWith e_2 : Boolean	take(length(s_2), s_1))==s_2 : bool
e contains x : Boolean	length(filter($v \rightarrow v == x$, s)) != 0 : bool
e_1 patch (n_1, e_2, n_2) : List[A]	append(take(n_1 - 1, s_1), append(s_2, drop($n_1 + n_2$ - 1, s_1))) : [t]
e_1 padTo (n, x) : List[A]	append(s_1, map($v \rightarrow x$, $range[1, n]$)) : [t]

Table 5.9: Operators with an (almost) direct counterpart for Sets

Scala, where e, e_1, e_2: Set[A], x: A	Ferry, where s,s_1,s_2: [t], x: t
e contains x | e apply x | $e(x)$: Boolean	length(filter($v \rightarrow v == x$, s)) != 0 : bool
$e + x$: Set[A]	if $[\![e$ contains $x]\!]$ then s else append(s, $[x]$) : [t]
e_1 ++ e_2 : Set[A]	append(s_1, filter($v \rightarrow$ not $[\![e_1$ contains $v]\!]$, s_2)) : [t]
e - x : Set[A]	filter($v \rightarrow v$!= x, s) : [t]
e_1 - - e_2 : Set[A]	filter($v \rightarrow$ not $[\![e_2$ contains $v]\!]$, s_1) : [t]
e_1 & e_2 | e_1 intersect e_2 : Set[A]	filter($v \rightarrow [\![e_2$ contains $v]\!]$, s_1) : [t]
e_1 | e_2 | e_1 union e_2 : Set[A]	append(s_1, filter($v \rightarrow$ not $[\![e_1$ contains $v]\!]$, s_2)) : [t]
e_1 &$^\smile$ e_2 | e_1 diff e_2 : Set[A]	filter($v \rightarrow$ not $[\![e_2$ contains $v]\!]$, s_1) : [t]
e_1 subsetof e_2 : Boolean	filter($v \rightarrow$ not $[\![e_2$ contains $v]\!]$, s_1) == 0 : bool

Table 5.10: Operators with an (almost) direct counterpart for Maps

Scala, where	Ferry, where
e, e_1, e_2: Map[K, A], x: A	s,s_1: [([t_1], t)], x: t
k: K, ks : Set[K]	k: t_1, ks: [t_1]
p: K → Boolean	k_{bool}: bool
f: A → B	s_2: ([t_1], t')
$e(k)$ | e apply k : A	nth(1, filter(v → the(v.1) == k, s)).2 : t
e getOrElse (k, x) : A	let $values$ = filter(v → the(v.1) == k, s) in if length($values$) != 0 then nth(1, $values$).2 else x : t
e contains k | e isDefinedAt k : Boolean	length(filter(v → the(v.1) == k, s)) != 0 : bool
e + (k -> x) : Map[K, A]	append(s, [([k], x]) : [([t_1], t)]
e ++ e_1 : Map[K, A]	append(s, s_1): [([t_1],t)]
e - k : Map[K, A]	filter(v → the(v.1) != k, s) : [([t_1], t)]
e - - ks : Map[K, A]	filter(v → length(filter(v_1 → v_1 == the(v.1), ks)) == 0, s) : [([t_1], t)]
e updated (k, x) : Map[K, A]	append(s, [([k], x]) : [([t_1], t)]
e.keys : Set[K]	map(v → the(v), groupWith(v → (v.1, . . . , v.N), map(v → the(v.1), s))) : [t_1]
e.values : Set[A]	map(v → the(v), groupWith(v → (v.1, . . . , v.N), map(v → v.2, s))) : [t]
e filterKeys p : Map[K, A]	filter(v → k_{bool}, s) : [([t_1], t)]
e mapValues f : Map[K, B]	map(v → s_2, s) : [([t_1], t')]

Table 5.11: The pointed Scala subset operators on Traversable (1 of 2)

xs.isEmpty	: Boolean	test whether the collection is empty
xs.nonEmpty	: Boolean	test whether the collection is non-empty
xs.size	: Int	the number of elements in the collection
xs ++ ys	: Traversable[A]	a collection consisting of the elements of both xs and ys
xs map f	: Traversable[A]	the collection obtained from applying the collection-valued function f to each element in xs
xs flatMap f	: Traversable[A]	the collection obtained from applying the collection-valued function f to each element in xs and concatenating the results
xs filter p	: Traversable[A]	the collection consisting of those elements of xs that satisfy the predicate p
xs filterNot p	: Traversable[A]	the collection consisting of those elements of xs that do not satisfy the predicate p
xs partition p	: Tuple2[Traversable[A], Traversable[A]]	split xs into a pair of two collections, one with elements that satisfy the predicate p, the other with elements that do not
xs groupBy f	: Map[B, Traversable[A]]	partition xs into a map of collections according to a discriminator function f
xs forall p	: Boolean	test indicating where the predicte p holds for all elements of xs
xs exists p	: Boolean	test indicating where the predicte p holds for some element in xs
xs count p	: Int	the number of elements of xs that satisfy the predicate p
xs reduceLeft op	: A	apply binary operation op between successive elements of non-empty collection xs going left to right

Table 5.12: The pointed Scala subset operators on Traversable (2 of 2)

xs.head	: A	the first element of the collection (or some element if no order is defined)
xs.tail	: Traversable[A]	the rest of the collection except for xs.head
xs.last	: A	the last element of the collection (or some element if no order is defined)
xs.init	: Traversable[A]	the rest of the collection except for xs.last
xs take n	: Traversable[A]	a collection consisting of the first n elements of xs (or some arbitrary n elements if no order id defined)
xs drop n	: Traversable[A]	the rest of the collection except for xs take n
xs splitAt n	: Tuple2[Traversable[A], Traversable[A]]	the pair of collections (xs take n, xs drop n)
xs splice(from, to)	: Traversable[A]	a collection consisting of elements in some index range of xs
xs takeWhile p	: Traversable[A]	the longest prefix of elements in this collection which all satisfy p
xs dropWhile p	: Traversable[A]	the longest prefix of elements in this collection which all do not satisfy p
xs span p	: Tuple2[Traversable[A], Traversable[A]]	the pair of collections (xs takeWhile p, xs dropWhile p)

Table 5.13: The pointed Scala subset operators on Iterable

xs takeRight n	: Iterable[A]	a collection consisitng of the last n elements of xs (or some arbitrary n elements if no order is defined)
xs takeLeft n	: Iterable[A]	the rest of the collection except for xs takeRight n
xs sameElements ys	: Boolean	a test whether xs and ys contain the same elements in the same order

Table 5.14: The pointed Scala subset operators on Sequence

xs.length	: Int	the length of the sequence (same as size)
xs.lengthCompare ys	: Int	returns -1 if xs is shorter than ys, +1 if it is longer and 0 if they have the length
xs(i) | xs apply i	: A	the element of xs at index i
xs.indices	: Sequence[Int]	the index range of xs extending from 0 to xs.length - 1
xs isDefinedAt i	: Boolean	a test whether i is contained in xs.indices
xs zip ys	: Sequence[Tuple2[A,B]]	a sequence of pairs of corresponding elements from xs and ys
xs.zipWithIndex	: Sequence[Tuple2[A, Int]]	a sequence of pairs of elements from xs with their indices
xs prefixLength p	: Int	the length of the longest prefix of elements in xs that all satisfy the predicate p
xs indexWhere p	: Int	the index of the first element in xs that satisfies p
xs indexOf x	: Int	the index of the first element in xs equal to x
xs.reverse	: Sequence[A]	a sequence with the elements of xs in reveresed order
xs startsWith ys	: Boolean	a test whether xs has sequence ys as a prefix
xs contains x	: Boolean	a test whether xs has an element equal to x
xs intersect ys	: Sequence[A]	the multi-set intersection of sequences xs and ys which preserves the order of elements in xs
xs diff ys	: Sequence[A]	the multi-set difference of sequences xs and ys which preserves the order of elements in xs
xs ++ ys | xs union ys	: Sequence[A]	multi-set union
xs.removeDuplicates	: Sequence[A]	a subsequence of xs that contains no duplicated elements
xs patch (f, ys, r)	: Sequence[A]	the sequence resulting from xs by replacing r elements starting with f by the patch ys
xs padTo (len, x)	: Sequence[A]	the sequence resulting from xs by appending the value x until length len is reached

Table 5.15: The pointed Scala subset operators on Set

xs contains x | xs(x)	: Boolean	test whether x is an element of xs
xs + x	: Set[A]	the set containing all elements of xs as well as x
xs ++ ys	: Set[A]	the set containing all elements of xs as well as all elements of ys
xs - x	: Set[A]	the set containing all elements of xs except for x
xs - - ys	: Set[A]	the set containing all elements of xs except for the elements of ys
xs & ys | xs intersect ys	: Set[A]	the set intersection of xs and ys
xs | ys | xs union ys	: Set[A]	the set union of xs and ys
xs & ys | xs diff ys	: Set[A]	the set difference of xs and ys
xs subsetof ys	: Boolean	test whether xs is a subset of ys

Table 5.16: The pointed Scala subset operators on Map

xs(k) | xs apply k	: A	the value associated with key k in map xs, exception if not found
xs getOrElse (k, d)	: A	the value associated with key k in map xs, or the default value d if not found
xs contains k	: Boolean	test whether xs contains a mapping for key k
xs isDefinedAt k	: Boolean	the same as contains
xs + (k -> x)	: Map[K, A]	the map containing all mappings of xs as well as the mapping k -> x from key k to value x
xs ++ kvs	: Map[K, A]	the map containing all mappings of xs as well as all key/value pairs of kvs
xs - k	: Map[K, A]	the map containing all mappings of xs except for any mapping of key k
xs - - ks	: Map[K, A]	the map containing all mappings of xs except for any mapping with a key in ks
xs updated (k, v)	: Map[K, A]	the same as xs + (k -> x)
xs.keys	: Set[K]	a set containing each key in xs
xs.values	: Set[A]	a set containing each value associated with a key in xs
xs filterKeys p	: Map[K, A]	a map view containing only those mappings in xs where the key satisfies predicate p
xs mapValues f	: Map[K, A]	a map vew resulting from applying function f to each value associated with a key in xs

Table 5.17: Sample of Ferry's built-in function library

map ::	$(t \to t_1, [t]) \to [t_1]$	map over list
concat ::	$[[t]] \to [t]$	list flattening
take; drop ::	$(int, [t]) \to [t]$	keep/remove list prefix
nth ::	$(int, [t]) \to t$	positional list access
zip ::	$([t_1], \ldots, [t_n]) \to [(t_1, \ldots, t_n)]$	n-way positional
unzip ::	$[(t_1, \ldots, t_n)] \to ([t_1], \ldots, [t_n])$	merge and split
unordered ::	$[t] \to [t]$	disregard list order
length ::	$[t] \to int$	list length
all; any ::	$[bool] \to bool$	quantification
sum; min; max ::	$[a] \to a$	list aggregation
the ::	$[t] \to t$	group representative
groupWith ::	$(t \to (a_1, \ldots, a_m), [t]) \to [[t]]$	grouping (as in [25])

Chapter 6

ScalaQL prototype

Contents

6.1 ScalaQL in action (external view)

Certain aspects of ScalaQL concerning functionality and implementation details have been already covered in previous chapters. In this chapter, a more detailed account of the architecture of the implementation is given.

ScalaQL operates on Scala source files recognizing at compile time two kinds of queries: (a) embedded LINQ queries, which have been tagged with an @LINQAnn annotation; and (b) Scala queries, tagged with @Persistent. Input and output are shown in Figure 6.1 on p. 71 and Figure 6.2 on p. 72).

In the screenshot on Figure 6.1 on p. 71, two queries are shown in the editing area.

First, a LINQ query on the right-hand-side of the projectsByName value definition, annotated with @LINQAnn. The query refers to the employees data stored in the database table Employees, and groups each employee's name with the list of projects where the employee works. The list of projects is obtained from the program variable projects.

```
object MyQLAppl {
  // definitions of the case classes
  case class Employee(id: Int, name: String, dept: String, salary: Int)
  case class Project(name: String, employees: List[String])
  // partial value definition with a value assigned by
  //    querying a database for the table with the name 'Employees'
  var Employees: List[Employee] = _
  var projects: List[Project] = List(Project("project1", List("Alex", "Bert", "Cora", "Drew", "Erik")),
                                     Project("project2", List("Fred", "Gina", "Herb", "Ivan", "Jill")))


  // LINQ embedded query
  @LINQAnn val projectsByName = "from empl in Employees select " +
                                "   new { name = empl.name," +
                                "         listOfProjects = from project in projects" +
                                "                          where project.employees.contains(empl.name)" +
                                "                          select project.name }"


  val salaries = List(200, 600, 400, 300)
  // Scala query from scratch
  @Persistent val namesWithSalaries = ( ( for (empl @ Employee(_, name, dept, _) <- Employees;
                                               if dept == "GE")
                                        yield name ) take 4 ) zip salaries
```

FROM CONSOLE

```
@Persistent val projectsByName = for (empl <- Employees)                                    Undesugared
                                 yield new { val name = empl.name;
                                         val listOfProjects = for (project <- projects;
                                                                   if (project.employees.contains(empl.name)))
                                                              yield project.name }
```

```
@Persistent val namesWithSalaries = ( ( for (empl @ Employee(_, name, dept, _) <- Employees;
                                             if dept == "GE")
                                      yield name ) take 4 ) zip salaries
                                                                                            Undesugared
```

Figure 6.1: LINQToScala (external view)

```
object MyQLAppl {
  // definitions of the case classes
  case class Employee(id: Int, name: String, dept: String, salary: Int)
  case class Project(name: String, employees: List[String])
  // partial value definition with a value assigned by
  //    querying a database for the table with the name 'Employees'
  var Employees: List[Employee] = _
  var projects: List[Project] = List(Project("project1", List("Alex", "Bert", "Cora", "Drew", "Erik")),
                                     Project("project2", List("Fred", "Gina", "Herb", "Ivan", "Jill")))
  // LINQ expansion
  //    from empl in Employees select
  //        new { name = empl.name,
  //              listOfProjects = from project in projects
  //                               where project.employees.contains(empl.name)
  //                               select project.name }
  @Persistent val projectsByName = for (empl <- Employees)
                                   yield new { val name = empl.name;
                                       val listOfProjects = for (project <- projects;
                                                                 if (project.employees.contains(empl.name)))
                                                            yield project.name }
  val salaries = List(200, 600, 400, 300)
  // Scala query from scratch
  @Persistent val namesWithSalaries = ( ( for (empl @ Employee(_, name, dept, _) <- Employees;
                                               if dept == "GE")
                                        yield name ) take 4 ) zip salaries
```

FROM CONSOLE

projectsByName

```
for empl in table Employees (id int, name string, dept string, salary int) with keys ((id))
return (name empl.name,
        listOfProjects for project in [(name 'project1', employees ['Alex','Bert','Cora','Drew','Erik']),
                                       (name 'project2', employees ['Fred','Gina','Herb','Ivan','Jill'])]
                               where length(filter(v -> v == empl.name, project.employees)) != 0
                      return project.name) :: [('name' string, 'listOfProjects' [string])]
```

namesWithSalaries

```
zip(take(4, for empl in table Employees (id int, name string, dept string, salary int) with keys ((id))
    where let name = empl.name, dept = empl.dept in dept == 'GE'
          return let name = empl.name, dept = empl.dept in name), [200,600,400,300]) :: [(string, int)]
```

Figure 6.2: ScalaToFerry (external view)

Second, a Scala query on the right-hand-side of the nameWithSalaries value definition, annotated with @Persistent. The query takes the first four employees of a particular department (from table Employees) and zips the employees' names with the salaries given by program variable salaries.

The LINQToScala phase replaces the right-hand-side part of projectByName, expanding it to another query, a query annotated with @Persistent as a tag for the next phase to recognize and process.

In the second screenshot (Figure 6.2 on p. 72), two queries are shown. The first one is an expansion of the embedded LINQ query, while the second one has not been processed by the first phase. Both are translated into Ferry, where some program variables are queried from a database, i.e. those variables that have been partially defined: var v = _. One such example is Employees. Other variables encountered in a query are replaced with their Ferry encodings, *e.g.* projects, salaries. The ScalaToFerry phase terminates without errors if the derived Ferry queries are successfully typechecked.

6.2 Architecture of the prototype

The ScalaQL prototype follows the Scala compiler architecture, which is based on a compiler operating in phases and an interaction protocol defined for compiler plugins. These plugins can update ASTs on their way from one phase to the next one, as well as generate errors and warnings.

As can be seen in Figure 1.2, ScalaQL realizes two separate translation tasks in a single compiler plugin: (a) *LINQ-to-Scala translation* performed by LINQToScala; and (b) *Scala-to-Ferry translation* as second translation, ScalaToFerry. These phases are executed in that order and comprise the following steps (summarized schematically in Figure 6.3):

LINQToScala phase:

- converting an embedded LINQ query into a string with the corresponding Scala comprehension. This step is performed by LINQ2Scala Converter.
- parsing the string obtained above with Scalac Parser to AST. This API is a part of scalac.
- replacing the AST of the original compilation unit with that obtained above. This step is performed by LINQ2Scala Transformer using API that are a part of scalac.

ScalaToFerry phase:

- translating Scala comprehensions into Ferry with ScalaToFerry Translator,
- typechecking the derived Ferry queries (Scala2Ferry Typer).

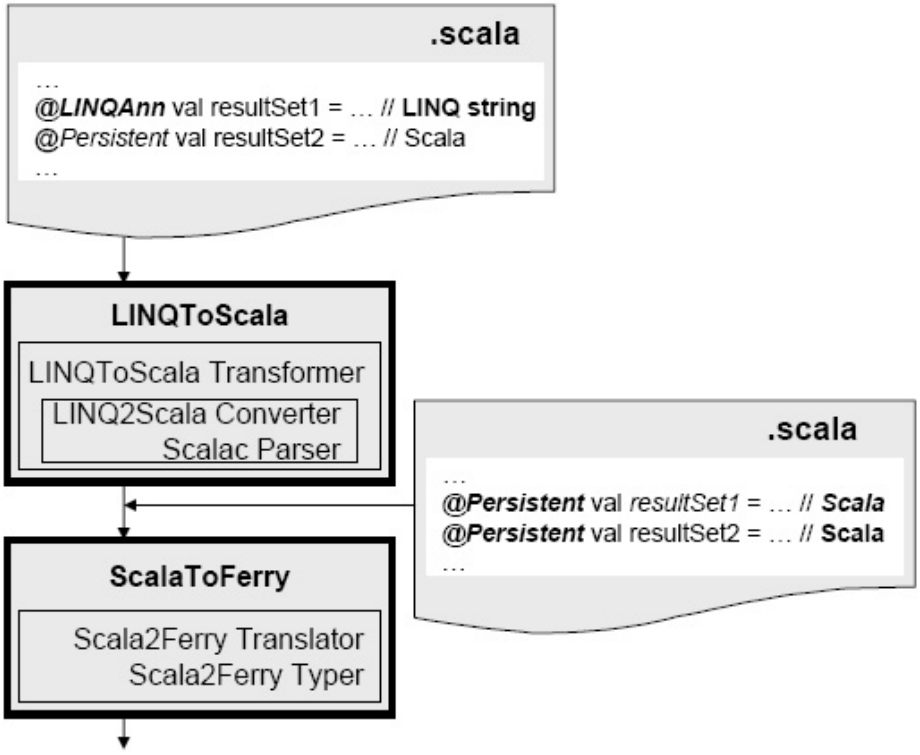

Figure 6.3: ScalaQL architecture

Listing 6.1: ScalaQL phases

```
    ...

val runsAfter = "parser"
val phaseName = "LINQToScala"

    ...

def newPhase(prev: Phase): Phase = new LINQToScalaPhase(prev)
class LINQToScalaPhase(prev: Phase) extends StdPhase(prev) {
  def apply(unit: CompilationUnit) {
    ... // code to be executed in the phase
  }
```

6.2.1 Annotation types used by ScalaQL

In more detail, the annotations that ScalaQL detects are:

class LINQAnn **extends** StaticAnnotation
class Persistent **extends** StaticAnnotation

These annotations are applied to Scala value definitions, like "**val** $Pat1_{pattern}$ = E_e", thus marking the right-hand-side of the definition for translation, either LINQ-to-Scala-to-Ferry or Scala-to-Ferry, as shown below:

LINQToScala:	ScalaToFerry:
@LINQAnn **val** $Pat1_{pattern} = SL_{StringLiteral}$	@Persistent **val** $Pat1_{pattern} = E_e$

6.3 Implementation details (internal view)

6.3.1 ScalaQL compiler plugin phases

ScalaQL extends Scala compiler functionality by defining the corresponding classes for its phases by subtyping scala.tools.nsc.SubComponent.StdPhase and running them right after the initial compiler phase parser as shown in Listing 6.1.

6.3.2 ScalaQL input queries

The target value definition nodes recognized by ScalaQL phases are shown in Listing 6.2 and Listing 6.3.

6.4 LINQToScala phase

6.4.1 LINQ2Scala Converter

An input embedded LINQ query extracted from right-hand-side part of @LINQAnn-annotated value definition *linq_str* is parsed with LINQ2Scala Converter (sts.linq.Parser and sts.linq.Transformer) to a string $linq2scala\_str$ with the corresponding Scala comprehension Listing 6.4

Listing 6.2: ScalaQL target AST nodes (internal)

```
...

val phaseName = "LINQToScala"

...

node match {

  case ValDef(Modifiers(_,_,List(Annotation(Apply(Select(New(Ident(name)),_),_),_),_))),
                          _,_, Literal (Constant(rhs))) =>
                           if (name.toString == "LINQAnn")
                                  ... // code translating LINQ in 'rhs' to Scala
 case _ =>

}
```

Listing 6.3: ScalaQL target AST nodes (internal)

```
...

val phaseName = "ScalaToFerry"

...

node match {

  case ValDef(Modifiers(_,_,List(Annotation(Apply(Select(New(Ident(name)),_),_),_),_))),
                          _,_, rhs) =>
                           if (name.toString == "Persistent")
                           ... // code translating Scala in 'rhs' to Ferry
  case _ =>

}
```

Referring to the extended example shown in Figure 6.1 and Figure 6.2, the following embedded LINQ query string

```
"from empl in Employees select " +
        "new { name = empl.name," +
              "listOfProjects = from project in projects" +
                                    "where project.employees.contains(empl.name)" +
                                    "select project.name }"
```

is converted to

```
"for (empl <- Employees)" +
"yield new { val name = empl.name;" +
              "val listOfProjects = for (project <- projects;" +
                                       "if (project.employees.contains(empl.name)))" +
                                    "yield project.name }"
```

Listing 6.4: LINQToScala Converter (internal)

```
val ast = sts.linq.Parser.run(new java.io.CharArrayReader(linq_str.toString.toArray))
val linq2scala_str = sts.linq.Transformer comprehend ast
```

Listing 6.5: LINQToScala Parser (internal)

```
object LINQToScalaParser extends global.syntaxAnalyzer.UnitParser(unit) {...}
// 'unit.body' refers to the AST of the compilation unit
unit.body = LINQToScalaParser.parse
```

6.4.2 LINQ2Scala Parser

The resulting string (*linq2scala_str*) encapsulating LINQ query expansion is parsed to the AST representation with a new instance of scalac's API, object LINQToScalaParser extends scala.tools.nsc.ast.parser.Parsers.UnitParser(unit), that is extended to return undesugared version of a Scala for comprehension. The obtained AST can be inserted to an original compilation unit (Listing 6.5)

An undesugared AST representation of a Scala for comprehension used as an input for the subsequent Scala-to-Ferry translation is a pair of a number of *qualifiers*, enums, and yield-*expression*, rhs, i.e. (enums, rhs) where each qualifier is an instance of a *generator*, *let-declaration*, or *filter* represented by the following case classes,

```
case class ValFrom(pos: Position, pat: Tree, rhs: Tree) // a generator
case class ValEq(pos: Position, pat: Tree, rhs: Tree) // a let-declaration
case class Filter(test: Tree) // a filter
```

6.4.3 LINQ2Scala Transformer

LINQ2ScalaTransformer extends scalac's API scala.tools.nsc.ast.Trees.Transformer by overriding the method transform that in addition to traversing the AST of the compilation unit allows its modification (Listing 6.6). The original LINQAnn-annotated value definition that can appear in a class template, or method body block,

```
case class Template(parents: List[Tree], self: Tree, body: List[Tree]),
case class Block(stats: List[Tree], expr: Tree),
```

is replaced with a value definition having the corresponding Scala comprehension in its right-hand-side and re-annotated for the subsequent *Scala-to-Ferry* translation with @Persistent.

For the extended example shown in Figure 6.1 and Figure 6.2, the method stmtsWalk walks through statements in the template's body of the Scala object MyQLAppl, i.e.

Listing 6.6: LINQToScala Transformer (internal)

```
class LINQ2ScalaTransformer extends Transformer {

    override def transform(tree: Tree): Tree = {
    val newTree = super.transform(tree);
    newTree match {
        case Template(parents, self, body) =>
                copy.Template(newTree, parents, self, stmtsWalk(body))
        case Block(stats, expr) =>
                copy.Block(newTree, stmtsWalk(stats), expr)
        case _ => newTree
      }
   }
```

```
List(...,
    @LINQAnn val projectsByName = "from empl in Employees select " +
                "new { name = empl.name," +
                    "listOfProjects = from project in projects" +
                                        "where project.employees.contains(empl.name)" +
                                        "select project.name }"
    val salaries = List(200, 600, 400, 300),
    @Persistent val namesWithSalaries = ( ( for (empl @ Employee(_, name, dept, _) <- Employees;
                                    if dept == "GE")
        yield name ) take 4 ) zip salaries
),
```

finds @LINQAnn-annotated value definition projectsByName, and replaces it with a new value definition of the same name by returning the following list of statements

```
List(...,
    @Persistent val projectsByName = for (empl <- Employees)
        yield new { val name = empl.name;
                    val listOfProjects = for (project <- projects;
                                            if (project.employees.contains(empl.name)))
                                        yield project.name }
    val salaries = List(200, 600, 400, 300),
    @Persistent val namesWithSalaries = ( ( for (empl @ Employee(_, name, dept, _) <- Employees;
                                    if dept == "GE")
    yield name ) take 4 ) zip salaries
).
```

6.5 ScalaToFerry phase

6.5.1 Scala2Ferry Translator

Scala2FerryTranslator extends scala.tools.nsc.ast.Trees.Traverser by overriding its traverse method. This allows traversing the AST of the compilation unit and executing specified actions when visiting certain AST nodes. In addition to

translating annotated Scala queries, Scala2FerryTranslator manages data derived from program variable definitions by encoding their values (encodings) and user provided types defined as case classes or final classes (user_provided_types) (Listing 6.7). In the case of the extended example shown in Figure 6.1 and Figure 6.2, the program variables, projects and salaries, defined as

```
val projects: List[Project] = List(Project("project1", List("Alex", "Bert", "Cora", "Drew", "Erik")),
                                   Project("project2", List("Fred", "Gina", "Herb", "Ivan", "Jill"))),
val salaries = List(200, 600, 400, 300)
```

are encoded in Ferry with

```
projects → [(name "project1", employees ["Alex", "Bert", "Cora", "Drew", "Erik"]),
            (name "project2", employees ["Fred", "Gina", "Herb", "Ivan", "Jill"])],
salaries → [200, 600, 400, 300],
```

whereas the user defined types, Employee and Project, are stored as a pair of a type name (a case class name) and a list of its fields' names and fields' values if available (in the case of final classes)

```
Employee → List( ("id", None), ("name", None), ("dept", None), ("salary", None) )
Project → List( ("name", None), ("employees", None) ).
```

New instances of user defined types are encoded with Ferry records, *e.g.*

```
Employee(1, "Alex", "GE", 600) → (id 1, name "Alex", dept "GE", salary 600)
typed with [(int, string, string, int)].
```

The translation of Scala comprehensions is realized by matching AST nodes of the compilation unit with patterns that cover the specified Scala subset, and by visiting these nodes (Listing 6.8). In the example of Figure 6.1 and Figure 6.2, the Scala queries:

```
@Persistent val projectsByName = for (empl <- Employees)
      yield new { val name = empl.name;
                  val listOfProjects = for (project <- projects;
                                            if (project.employees.contains(empl.name)))
                                       yield project.name }
@Persistent val namesWithSalaries = ( ( for (empl @ Employee(_, name, dept, _) <- Employees;
                                             if dept == "GE")
                                       yield name ) take 4 ) zip salaries
```

are translated to the following Ferry queries (with the corresponding encodings and database tables querying performed)

```
for empl in table Employees (id int, name string, dept string, salary int) with keys ((id))
return (name empl.name,
        listOfProjects for project in [(name 'project1', employees ['Alex', 'Bert', 'Cora', 'Drew', 'Erik']),
                                       (name 'project2', employees ['Fred', 'Gina', 'Herb', 'Ivan', 'Jill'])]
                       where length(filter(v → v == empl.name, project.employees)) != 0
                       return project.name)

zip(take(4, for empl in table Employees (id int, name string, dept string, salary int))) with keys ((id))
            where let name = empl.name, dept = empl.dept in dept == 'GE'
            return let name = empl.name, dept = empl.dept in name), [200, 600, 400, 300]),
```

6.5.2 Scala2Ferry Typer

As mentioned in Sec. 5.2, the typechecking of Ferry queries, which is applied before their shipping, serves two purposes: (a) ensuring the isomorphism of the Scala-to-Ferry translation for the chosen Scala subset; (b) providing an additional confidence in correctness of the given translation, thus avoiding runtime exceptions on the database side. Typechecking is performed by ScalaToFerry Typer (sts.ferry.Typer) for Ferry queries before their normalisation to Ferry Core (Listing 6.9).

For the extended example shown in Figure 6.1 and Figure 6.2, ScalaToFerry Typer results in the following types for the first and the second Ferry queries, correspondingly:

```
[('name' string, 'listOfProjects' [string])]
[(string, int)]
```

Listing 6.7: ScalaToFerry Translator (internal)

```
class ScalaToFerryTranslator[T] (v: Visitor[T], defaultVal: T) extends Traverser
{

    override def traverse(tree: Tree) = {
        tree match {
          case e @ ValDef(Modifiers(_,_,List(Annotation(Apply(Select(New(Ident(name)),
                                        _), _), _))), _, _, rhs) =>
            if (name.toString == "Persistent") {
              walk(rhs)
            }

          case e @ ValDef(_,name,_,rhs) =>
                val rhs1 = rhs match {
                        case EmptyTree =>
                          // partial variable definition
                        case Match(_,_) =>
                          ...
                          walkValDefPatterns(e, rhs);
                          ...
                        case _ =>
                          ...
                          walk(rhs)
                          ...
                }
                ...
                v.encodings = v.encodings update (v. visit (name).toString, rhs1)
                ...

          case e @ ClassDef(mods, name, _, Template(_, _, body)) =>
            if (mods.isCase || mods.isFinal) {
                val body1 = body.map(_ match {
                      case ValDef(_, name, _, rhs) => { ... walk(rhs); ...}
                      })
             ...
            v.user_provided_types = v.user_provided_types update (name.toString, body1)
             ...
            }

          case _ => super.traverse(tree);
        }

    }
```

Listing 6.8: ScalaToFerry Visitor (internal)

```
abstract class Visitor[T]() {

    var encodings: Map[String,T] = Map()
    var user_provided_types: Map[String, List[Tuple2[String, Option[T]]]] = Map()

    def visit(e: Apply, gens: List[Tuple2[T,T]], letdecls: List[Tuple2[T,T]],
                        filteres: List[T], rhs: T): T
    def visit(e: Apply, qual: T, fun: String, args: List[T]): T
    def visit(e: Apply, qual: T, fun: String, arg: Tuple2[List[T], T]): T
    def visit(e: Apply, fun: String, args: List[T]): T
    def visit(e: Apply, vparams: List[T], body: T, args: List[T]): T
    def visit(e: Apply, expr: T, posAcc: Int): T
    def visit(e: ValFrom, varid: T, bindings: List[Tuple2[T,T]], rhs: T): Tuple2[T,T]
    def visit(e: Bind, bindings: List[Tuple2[T,T]], pat: Patterns.Value): List[Tuple2[T,T]]
    def visit(e: Bind, bindings: List[Tuple2[T,T]], uptype: List[String]): List[Tuple2[T,T]]
    def visit(e: Select, qual: T, sel: String): T
    def visit(e: Block, stats: List[Tuple2[T,T]], expr: T): T
    def visit(e: If, e1: T, e2: T, e3: T): T
    def visit(e: Match, casecls: List[Tuple2[List[Tuple2[T,T]], T]]): T
    def visit(e: ValDef, name: String, exprs: List[T]): T
    def visit(e: Ident, name: String): T
    def visit(e: Literal, value: Any): T
    def visit(e: Name): T
}
```

Listing 6.9: ScalaToFerry Typer (internal)

```
try {
    val ferryExpr = ferryQ.asInstanceOf[Expr]
    ...
    sts.ferry.Typer.typing(ferryExpr)
    ...
} catch {
    case e: Error => global.error(e.getMessage + " in translation of " + tree)
}
```

Chapter 7

Conclusions

A modern functional database query language integrated with a modern programming language brings new opportunities to the developers. Such opportunities comprise the fashionable functional-object paradigms of modern programming languages, and enhanced expressiveness and conciseness by fetching persistent data into program space. The ScalaQL proposal is motivated by comprehension syntax of a programming language and pre-exisiting libraries of operations on collections and underpinned by modern extensible compiler architectures.

The ScalaQL project addresses some aspects of persistent programming languages by summarizing them as the target levels of integration (Sec. 1.5) and proposes the translation algorithm from LINQ and Scala to Ferry. The main results of the current master thesis project contributes in providing the proof of a concept covering the details of the Scala-to-Ferry translation and program transformation in Scala underlying both translations, i.e. from LINQ to Scala and from Scala to Ferry.

7.1 ScalaQL prototype

The current implementation of ScalaQL prototype targets Level 2 of the outlined integration levels (Sec. 1.5). It is designed as a single compiler plugin with two compilation phases that perform the underlying translations, one for LINQ to Scala translation, and another for Scala to Ferry translation. Both phases make use of the available scalac APIs that allows AST building and subsequent transformations on input compilation unit ASTs. In contrast to the normal scalac compilation phases (Figure 1.1), the phases of ScalaQL preserve AST nodes returned by a parser before desugaring takes place and Scala-To-Ferry translation is performed based on those AST nodes. The target Scala subset of the Scala-To-Ferry translation is extracted in the terms of Scala patterns based on the available scalac APIs (case classes) representing ASTs. The resulting Ferry queries are typechecked (as the final step of the Scala-To-Ferry translation) against the full Ferry language (i.e. not just Ferry Core). The developed prototype fulfills all the requirements set at the start of the project. Still, further improvements are motivated by available related works as outlined in the next section.

7.2 Future Work

7.2.1 Client-side processing (for levels 3 and 4)

The outlined levels of integration address different levels of client-side processing (by a query interpreter) assuming that a query evaluator and optimiser are available. Level 3 and Level 4 as formulated in Sec. 1.5 approaches the full integration of LINQ into Scala by addressing a complete client-side processing. Level 3 extends by allowing program variables whose actual type can be precisely determined only at runtime due to possible subtyping. In this case, a program variable type is computed and represented with a Ferry counterpart at runtime assuming that such a counterpart is pre-defined. The case when a Ferry type counterpart does not exist is addressed by the next level, Level 4. Level 4 applies client-side processing on a query that can be only partially processed on the DB side meaning that only fragments of it can be optimised and shipped to a database evaluating to sub-results. Support for these levels is a desirable extension of the current ScalaQL implementation.

7.2.2 Higher-level Data Models

Some aspects related to Object/Relational Mapping can be investigated, in particular, the ones motivated by Entity SQL [1] that is a part of Microsoft Entity Framework. Entity SQL is a query language developed to support the base concepts, *Entity Data Model* and *Entity-Relationship*. Entity SQL allows querying data against higher-level conceptual data models [2] with the constructs well-known from SQL.

7.2.3 Capabilities partially supported by DBPLs

The general aspects of RDBMSs and OODBMSs are considered as the potential future contributions to persistent programming languages, in particular, support for some abstractions. The related work on *incremental maintenance of materialized views* refers to the efficiency that is especially significant in information integration for distributed data sources and deals with incremental propagation of updates from the referenced database entities to the corresponding *materialized views*. Most of the current work refers to relational database systems, in particular, ones related to deriving *production rules* for incremental maintenance of materialized views [19]. Production rules are expressed with a query language syntax and specify certain data manipulation operations in an *action part*. A production rule is triggered by certain events when certain conditions are fulfilled and updates views. The work on incremental maintenance of materialized views in OQL being a query language for object databases is considered to be relevant [2, 3]. The approach relies on an algebraic *incremental maintenance plan* (IMP) constructed for each pre-defined update event that allows computing required changes to a materialized view reacting to changes made to a database referred to as a *delta*. The aspects of *data consistency* are addressed with *invariants* defining integrity constraints and the corresponding analyses detecting their violation.

[1] `http://msdn.microsoft.com/en-us/library/bb387118.aspx`
[2] `http://msdn.microsoft.com/en-us/magazine/cc700331.aspx`

Bibliography

[1] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *SIGMOD '07: Proc. of the 2007 ACM SIGMOD Intnl Conf on Mgmt of Data*, pages 877–888, New York, NY, USA, 2007. ACM.

[2] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. MOVIE: an incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[3] M. Akhtar Ali, Norman W. Paton, and Alvaro A. A. Fernandes. An Experimental Performance Evaluation of Incremental Materialized View Maintenance in Object Databases. In Yahiko Kambayashi, Werner Winiwarter, and Masatoshi Arikawa, editors, *DaWaK*, volume 2114 of *LNCS*, pages 240–253. Springer, 2001.

[4] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, 1995.

[5] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in Translation: Formalizing proposed extensions to C#. *SIGPLAN Not.*, 42(10):479–498, 2007.

[6] Daniel K. C. Chan and Philip W. Trinder. A processing framework for object comprehensions. *Information & Software Technology*, 39(9):641–651, 1997.

[7] Austin Clements. A comparison of designs for extensible and extension-oriented compilers. Master's thesis, Massachusetts Institute of Technology, Feb 2008. `http://pdos.csail.mit.edu/xoc/clements-thesis.pdf`.

[8] Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *12th Intnl. Symp. on Database Programming Languages (DBPL 2009)*. `http://ezrakilty.net/pubs/dbpl-sqlizability.pdf`.

[9] Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. Technical Report EDI-INF-RR-1327, School of Informatics, University of Edinburgh, May 2009. `http://homepages.inf.ed.ac.uk/s0567141/how-to-write-great-sql/great-sql.pdf`.

[10] Oege de Moor et al. .QL for Source Code Analysis. In *SCAM '07*, pages 3–16, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Alan Dearle, Graham N.C. Kirby, and Ron Morrison. Orthogonal persistence revisited. In Moira C. Norris and Michael Grossniklaus, editors, *Proceedings of the 2nd Intnl Conf ICOODB 2009*, pages TODO–TODO, July 2009. ISBN 978-3-909386-95-6, `http://www.cs.st-andrews.ac.uk/files/publications/download/DKM09a.pdf`.

[12] Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In *SIGMOD '95: Proc. of the 1995 ACM SIGMOD Intl Conf. on Management of Data*, pages 47–58, New York, NY, USA, 1995. ACM Press. `http://lambda.uta.edu/sigmod95.ps.gz`.

[13] Charles L. Forgy. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Expert systems: a software methodology for modern applications*, pages 324–341, 1990.

[14] 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`.

[15] Peter M. D. Gray, Larry Kerschberg, Peter J.H. King, and Alexandra Poulovassilis (Eds.). *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. SpringerVerlag, 2004.

[16] Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery join graph isolation. In *Proc. of the 25th Intnl. Conf. on Data Engineering (ICDE 2009), Shanghai, China, March/April 2009*. To appear. Extended version at `http://arxiv.org/abs/0810.4809`.

[17] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *SIGMOD'09: Proc. of the 35th SIGMOD Intnl. Conf. on Management of Data*, pages 1063–1066, New York, NY, USA, 2009. ACM.

[18] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with Datalog. In Dave Thomas, editor, *ECOOP'06: Proc. of the 20th European Conf. on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 2–27, Berlin, Germany, 2006. Springer.

[19] E. N. Hanson, S. Bodagala, and U. Chadaga. Trigger Condition Testing and View Maintenance Using Optimized Discrimination Networks. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):261–280, 2002.

[20] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems: Fourth Edition*. The MIT Press, 2005.

[21] Jernej Kovse. *Model-Driven Development of Versioning Systems*. PhD thesis, TU Kaiserslautern, Germany, August 2005.

[22] Krishna K. Mehra, Sriram K. Rajamani, A. Prasad Sistla, and Sumit K. Jha. Verification of object relational maps. In *SEFM '07: Proc. of the Fifth IEEE Intnl. Conf. on Software Engineering and Formal Methods*, pages 283–292, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD '07: Proc. of the 2007 ACM SIGMOD Intnl Conf on Mgmt of Data*, pages 461–472, New York, NY, USA, 2007. ACM.

[24] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007. ISBN 9780978739256.

[25] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proc. of the ACM SIGPLAN Workshop Haskell '07*, pages 61–72, New York, NY, USA, 2007. ACM Press. `http://research.microsoft.com/~simonpj/papers/list-comp/list-comp.pdf`.

[26] Anna Ruokonen, Imed Hammouda, and Tommi Mikkonen. Enforcing Consistency of Model-Driven Architecture Using Meta-Designs. In *European Conf. on MDA: Workshop on Consistency in Model Driven Engineering (C@MoDE 2005)*, pages 127–141, Nov. 2005.

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

[28] Jens Teubner. *Pathfinder: XQuery Compilation Techniques for Relational Database Targets*. PhD thesis, Technische Universität München, October 2006. `http://www-db.in.tum.de/~teubnerj/publications/diss.pdf`.

[29] Philip W. Trinder. *A Functional Database*. PhD thesis, Oxford University, December 1989. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.4456`.

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

[31] Kris De Volder. JQuery: A Generic Code Browser with a Declarative Configuration Language. In Pascal Van Hentenryck, editor, *PADL*, volume 3819 of *LNCS*, pages 88–102. Springer, 2006.

[32] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. *SIGPLAN Not.*, 43(10):19–36, 2008.

[33] Darren Willis, David J. Pearce, and James Noble. Efficient Object Querying for Java. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 28–49. Springer, 2006.

[34] Limsoon Wong. The functional guts of the Kleisli query system. In *ICFP '00: Proc of the Fifth ACM SIGPLAN Intnl Conf on Functional Programming*, pages 1–10, New York, NY, USA, 2000. ACM.

Appendix A

Syntax and Semantics of LINQ

A.1 Syntax

In its simplest form, a LINQ query begins with a from clause and ends with either a select or group clause. In between, zero or more *query body clauses* can be found (from, let, where, join or orderby). Queries may be nested: the collection over which a from variable ranges may itself be a query. A similar effect can be achieved by appending into *variable* S_2 to a subquery S_1: with that, S_1 is used as generator for S_2. The fragment into *variable* S_2 is called a *query continuation*.

A join clause tests for equality the key of an inner-sequence item with that of of an outer-sequence item, yielding a pair for each successful match. An orderby clause reorders the items of the incoming stream using one or more keys, each with its own sorting direction and comparator function. The ending select or group clause determines the shape of the result in terms of variables in scope.

The detailed structure of LINQ phrases is captured by the grammar in Table A.1 (listing LINQ-proper productions, with *QueryExp* being the entry rule) and in Table A.2 (listing other syntactic domains). In order to save space, well-known productions have been omitted (*e.g.*, those for arithmetic expressions). The notation conventions in the grammar follow Turbak and Gifford [30]. Terminals are enumerated (*e.g.* for the syntactic domain *Direction*). Compound syntactic domains are sets of phrases built out of other phrases. Such domains are annotated with *domain variables*, which are referred from the right-hand-side of productions. References, *e.g.* $QC^{0..1}_{qcont}$ (which ranges over the *QueryContinuation* domain) are subscripted with a *label* later used to denote particular child nodes in the transformations rules. The superscript of a reference indicates the allowed range of occurrences.

LINQ is mostly implicitly typed: only variables in from or join clauses may optionally be annotated with type casts. Several ambiguities have to be resolved with arbitrary lookahead (*e.g.* to distinguish between a *JoinClause* and a *JoinIntoClause*) requiring rule priorities or syntactic predicates [24].

Table A.1: LINQ-related production rules

$Q \in$ 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	::=	$E_{ord}\ \ Direction_{dir}$
$Direction \in$		{ **ascending**, **descending** }
$S \in$ SelectClause	::=	**select** E_{selexp}
$G \in$ GroupByClause	::=	**group** E_{e1} **by** E_{e2}

A.2 Semantics

The denotational semantics of LINQ gives meaning to a query in terms of its syntax components. An auxiliary definition and two kinds of valuation functions are needed. A *binding-set* $\mathcal{B} \equiv \{v_1 \mapsto t_1, \ldots\}$ is a finite map from non-duplicate variables v_i to values t_i. We write $v_i \mapsto t_i$ as a shorthand for the pair (v_i, t_i). LINQ forbids declaring a variable whose name would hide another, so a non-ordered map is enough. As usual, an expression E can be evaluated *in the context of* $\mathcal{B}$ by induction on its syntactic structure, with a non-defining occurrence of variable v evaluating to its image t under $\mathcal{B}$.

The kinds of valuation functions are: (1) $[\![Q]\!]_{envs}$ denotes the sequence of binding-sets generated by Q (a query body) given the *incoming* sequence of binding-sets *envs*; while (2) $[\![E]\!](env)$ denotes the evaluation of E in the context of the single binding-set *env*. To simplify the formulation of the valuation functions, a query is regarded as a sequence S of body clauses Q, resulting from having desugared query continuations into subqueries [14].

The valuation $[\![Q]\!]_{envs}$ denotes simply the (sub-)query results when Q is a *SelectClause* or a *GroupByClause*:

$$[\![\ \textbf{select}\ E_{selexp}\ ]\!]_{envs} \qquad [\ [\![selexp]\!](env) \mid env \leftarrow envs\ ] \tag{A.1}$$

Informally speaking, group *result* by *key* returns a *Grouping*, i.e. a finite *ordered* map with entries *key* ↦ *cluster*, a cluster being a sequence of results.

Table A.2: Other syntactic domains

Id, V ∈	Identifier = (([a-zA-Z][a-zA-Z0-9]*) - Keyword)	
SG ∈	(SelectClause ∪ GroupByClause)	
E ∈	Exp = (QueryExp ∪ ArithExp ∪ BoolExp ∪ UnaryExp ∪ BinaryExp ∪ PrimaryExp ∪ DotSeparated ∪ ...)	
EL ∈	ExpOrLambda = (Exp ∪ Lambda)	
P ∈	PrimaryExp = (Application ∪ QueryExp ∪ NewExp ∪ PrimitiveLit ∪ ...)	
T ∈ TypeName	::=	$Id_{fragments}^{1..*\ <separator:.>}$
D ∈ DotSeparated	::=	P_{pre} . P_{post}
A ∈ Application	::=	$Id_{head}\ Cast_{cast}^{0..1}$ ($EL_{args}^{0..*\ <separator:,>}$)
L ∈ Lambda	::=	($Id_{params}^{0..*\ <separator:,>}$) => E_{body}

The valuation of *GroupByClause* involves a left-fold, taking an empty grouping as initial value and progressively adding the valuation of *result* to the cluster given by the valuation of *key*. Using Haskell,

$$[\![\ \texttt{group}\ E_{result}\ \texttt{by}\ E_{key}\ ]\!]_{envs} \qquad \texttt{foldl cf []}\ envs \tag{A.2}$$

where cf, the combining function, captures the provided result selector and key extractor, has type *Grouping* → *BindingSet* → *Grouping*, and is defined as:

```
cf g bs = let r = ([[result]](env)) in
          let k = ([[key]](env)) in
          if hasKey g k then appendToCluster g k r
                        else append g [(k,[r])]
```

For Q other than select or groupby, $[\![Q]\!]_{envs}$ denotes a sequence of binding-sets which constitute the *envs* in effect for the next clause in S, the first Q in S being evaluated with an empty incoming *envs*.

$$[\![\texttt{from}\ V_{var}\ \texttt{in}\ E_{srcSeq}\ ]\!]_{envs} \qquad [env' \mid env \leftarrow envs,\ item \leftarrow [\![srcSeq]\!](env),\ let\ env' = env \cup \{var \mapsto item\}\ ] \tag{A.3}$$

$$[\![\texttt{let}\ V_{var} = E_{exp}\ ]\!]_{envs} \qquad [env' \mid env \leftarrow envs,\ let\ env' = env \cup \{var \mapsto [\![exp]\!](env)\}\ ] \tag{A.4}$$

$$[\![\texttt{where}\ E_{test}\ ]\!]_{envs} \qquad [env \mid env \leftarrow envs, [\![test]\!](env)\ ] \tag{A.5}$$

The valuation of an *OrderByClause* permutes the incoming binding-sets, sorting the sequence *envs* according to the multi-key given by expressions key_i and sort directions dir_i. In terms of the Haskell function Data.List.sortBy,

$$[\![\ \text{orderby}\ key_1\ dir_1\ \ldots key_n\ dir_n\ ]\!]_{envs} \qquad \texttt{sortBy comp envs} \tag{A.6}$$

where comp is a comparison function (specific to the given key_i and dir_i, $i = 1 \ldots n$) between two binding-sets bsA and bsB, returning one of GT, EQ, LT.

First, $[\![\, key_1 \,]\!](bsA)$ and $[\![\, key_1 \,]\!](bsB)$ are compared taking dir_1 into account. If they are not equal that's the outcome of comp bsA bsB. Otherwise, $[\![\, key_2 \,]\!](bsA)$ and $[\![\, key_2 \,]\!](bsB)$ are compared taking dir_2 into account, and so on. If no GT or LT is found for $i = 1 \ldots n$, EQ is returned.
The semantics is defined over a core syntax where explicit type annotations have been desugared into type casts (in from and join clauses).

$$
\begin{aligned}
&[\![\texttt{join}\ V_{innerVar} \quad \texttt{in} \quad E_{isrc}\ \texttt{on}\ E_{outerKey}\ \texttt{equals}\ E_{innerKey}\,]\!]_{envs} \\
&\quad [ienv \mid env \leftarrow envs, innerItem \leftarrow [\![isrc]\!](env)\,, \\
&\quad let\, ienv \;=\; env \cup \{\, innerVar \mapsto innerItem \,\}\,, \\
&\quad [\![outerKey]\!](env) \;=\; [\![innerKey]\!](ienv)\,] \qquad \text{(A.7)} \\
&[\![\texttt{join}\ V_{innerVar} \quad \texttt{in} \quad E_{isrc}\ \texttt{on}\ E_{outerKey}\ \texttt{equals}\ E_{innerKey}\ \texttt{into}\ V_{resVar}\,]\!]_{envs} \\
&\quad [renv \mid env \leftarrow envs, \\
&\quad let\ group \;=\; [\, innerItem \mid innerItem \leftarrow [\![isrc]\!](env) \\
&\qquad\qquad let\ ienv \;=\; env \cup \{innerVar \mapsto innerItem\}, \\
&\qquad\qquad [\![outerKey]\!](env) \;=\; [\![innerKey]\!](ienv)\,], \\
&\quad let\, renv \;=\; env \cup \{\, resVar \mapsto group \,\}\,] \qquad \text{(A.8)}
\end{aligned}
$$