Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering, © JOT 2010
Online at http://www.jot.fm.

Extending Scala with Database Query Capability

Miguel Garcia[a] Anastasia Izmaylova[a] Sibylle Schupp[a]

a. Institute for Software Systems (STS),
Hamburg University of Technology (TUHH), Germany

Abstract The integration of database and programming languages is difficult due to the different data models and type systems prevalent in each field. We present a solution where the developer may express queries encompassing program and database data. The notation used for queries is based on comprehensions, a declarative style that does not impose any specific execution strategy. In our approach, the type safety of language-integrated queries is analyzed at compile-time, followed by a translation that optimizes for database evaluation. We show the translation total and semantics preserving, and introduce a language-independent classification. According to this classification, our approach compares favorably with Microsoft's LINQ, today's best-known representative. We provide an implementation in terms of Scala compiler plugins, accepting two notations for queries: LINQ and the native Scala syntax for comprehensions. The prototype relies on Ferry, a query language that already supports comprehensions yet targets SQL:1999. The reported techniques pave the way for further progress in bridging the programming and the database worlds.

Keywords Language-integrated query, Scala programming language, Type-directed program transformation, Compiler plugins.

1 Introduction

Experience has shown that having the best programming language and the best database manager (DBMS) is not enough: persistence-related code accounts still today for a large portion of development cost. Unlike the situation for "normal" programming language constructs, a compiler is not aware of the semantics of embedded database queries, and thus offers no help regarding their well-formedness checking (*e.g.*, to detect queries broken due to a reorganization of the database schema) or their processing (*e.g.*, in optimizing the query before shipping it to the DBMS for evaluation). Approaches to overcome these shortcomings fall under the general umbrella of *language-integrated query*, of which *embedded SQL* is an early example and Microsoft LINQ [23] today's best-known representative. In a nutshell, the main advantages of

Miguel Garcia, Anastasia Izmaylova, Sibylle Schupp. *Extending Scala with Database Query Capability*. In Journal of Object Technology, vol. 9, no. 4, 2010, pages 45–68. Available at http://www.jot.fm/contents/issue_2010_07/article3.html

language-integrated query are its compile-time type safety and the resulting conciseness of expression due to type inference and compact notation for common operations, such as grouping and sorting.

The integration of a (host) programming language and an (embedded) database query language is made difficult by the different forces influencing design decisions in each community: programming languages strive to support ever higher-level expressiveness while efficiency is the dominant factor for query languages. For example, each programming language generation has increased the repertoire of type constructors, while standard query languages (at least of the relational kind) rely on a few data structuring mechanisms: flat tuples, multisets, and foreign keys. Beyond type system aspects, modern programming languages (*i.e.*, those harmonizing the functional and object paradigms) support features that simply have no counterpart in the data-manipulation languages of mainstream DBMSs (higher-order functions, libraries encapsulating collection operations, and subtyping polymorphism).

In this paper, we present a solution that overcomes these difficulties. In our approach, the developer may express queries that simultaneously access data from both the database and the program spaces. These queries are written not in a database-specific query language but in terms of Scala-level constructs: comprehensions over iterables, collection operations, and closures (e.g., to indicate filters). In addition, the developer may use LINQ for the same purpose. At compile time, queries are translated in a typesafe manner, and shipped for database evaluation. The obtained resultsets can be mapped back to program-level values, for further processing on the client-side. The approach, realized as a proof of concept for Scala [26], generalizes to other comprehensions-aware programming languages. We also discuss how program values without a database-level encoding participate in query evaluation. The listed capabilities fit within a novel classification scheme that we propose for language-integrated queries (Sec. 3), a scheme that abstracts beyond particular technical realizations and focuses instead on the semantics of the program-database divide.

The translation from program queries relies on Ferry [19] as intermediate language, a state of the art, optimizing query language for relational backends that already supports comprehensions, and thus greatly simplifies targeting Core SQL:1999. We show that this translation is (a) total, in that each well-formed sentence in the input language is translated into a bundle of SQL queries; and (b) query semantics preserving, in the sense that the translated query is guaranteed to evaluate without run-time errors for all inputs on which the original query would have terminated without exceptions. Moreover, during server-side evaluation no client-server communication is incurred (unlike Object/Relational Mapping solutions, which are prone to cache misses for all but the most trivial queries). Regarding expressiveness, the current main limitation of our prototype (but not of our approach) is that only a subset of the Scala type system is supported (datatypes are supported but not objects, and collections must be monomorphic). On the plus side, more extensive compile-time checks are performed than by Microsoft LINQ query providers (Sec. 3.5).

In more detail, language-integrated query involves problems dealing with both language theory and implementation. Language theory provides a framework to solve the novel situation where (a) a query may refer to data in different locations: program space and DB space; (b) collection operators may be used that are defined in program libraries. The brute-force approach of bringing all data into program space does indeed guarantee correcteness, at the cost of unacceptable performance (otherwise, virtual memory would have displaced databases long ago). The more sensible approach of

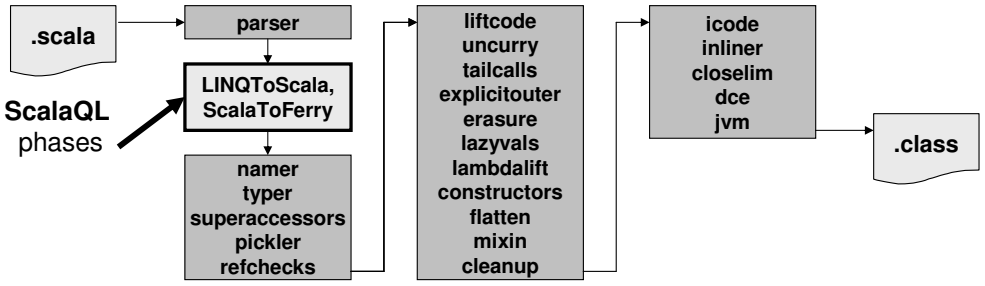

Figure 1 – Scala compiler architecture and ScalaQL extension

having the DBMS evaluate operators originally defined with programming language semantics involves showing that an isomorphism exists between types, operators, and values across program and database space (Sec. 5).

On the practical front, a programming language expression may well be statically recognized as amenable for database-based evaluation. However, compiler architectures impose a very high cost to implementing such extension, in particular for Java, whose compilers were not designed with extensibility in mind [9]. In this regard, our choice of Scala is not accidental: an interaction protocol has been defined for *compiler plugins* [25] to extend or change the behavior of the compiler, without modifying the compiler itself. Plugins can inspect and update the ASTs on the way from one phase to the next, as well as raise errors and warnings.

The structure of this paper is as follows. Sec. 2 provides background on Scala, on the underpinnings of modern query languages (comprehensions), and on the Ferry language. A summary description of the technologies under the LINQ umbrella closes that section, emphasizing aspects frequently glossed over in the literature (for example, the degree of static checking and the extent of optimization). In Sec. 3, we put forward classification criteria to rank competing approaches to language-integrated query, by defining four capability levels for language processors to handle queries involving different mixes of program vs. database semantics. The next two sections cover in detail our compilation pipeline, starting from LINQ to Scala (Sec. 4), followed by the chosen subset of Scala (types, operations, and syntactic constructs) that our Scala to Ferry translation accepts as input (Sec. 5). Looking into the future, we sketch our plans for client-side optimizations before query shipping (Sec. 6). Finally, the two last sections offer an overview of related work (Sec. 7) and discuss conclusions (Sec. 8). To make the paper self-contained, Appendix A summarizes the syntax and semantics of LINQ.

Knowledge about data query languages is assumed from the reader as well as familiarity with compiler terminology. A prototype (ScalaQL) realizing our approach can be found at `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaQL`.

2 Background

Scala [26] subsumes object-oriented and functional programming in a statically typed setting with type inference. The adopted syntax is highly uniform, where programs are trees of definitions without constraints on their nesting. The syntax evokes concepts from object orientation (classes, methods), which can be expressed in terms of more fundamental abstractions (objects, traits for mixin inheritance, type members

that generalize type parameters). Since Scala is a functional language, functions are first-class values: they can be written as literals, passed as arguments or returned from methods. Methods, which can be encoded as functions, may define one or more lists of value parameters, in addition to a list of type parameters. This way, a method abstracts over values and types. The type system is advanced and comprises intersection types, path-dependent types as companion to type members, variance indication for type parameters, and higher-kinded types [24].

Support for object decomposition by pattern matching contributes to the succinctness of queries. Queries themselves have a semantic foundation that is language-independent, as covered in the next subsection.

2.1 Semantic foundation: query comprehensions

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 [14]).

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 Object Query Language (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 [18] and for main memory [32].

2.2 Ferry: optimizing database comprehensions

Ferry [19], designed by Tom Schreiber at Uni Tübingen[1], 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 types of the form $t = a \mid [t] \mid (t, \ldots, t)$ where a represents atomic types like *string*, *int*, or *bool*. Structured types can be used to model programming language types such as lists, dictionaries (a.k.a. "maps"), 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 related

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

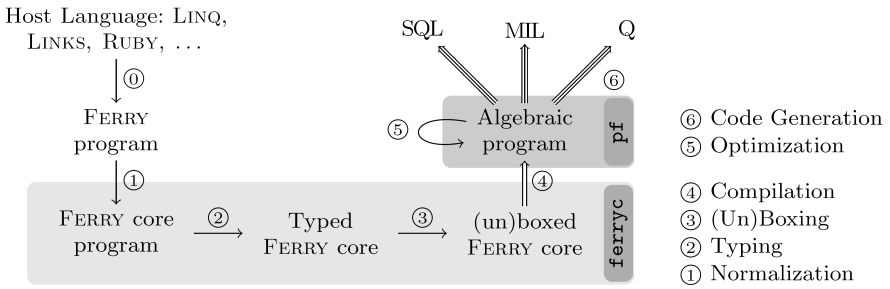

Figure 2 – Relational translation of a non-relational language, reproduced from [19]

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 (unlike 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 2 in Sec. 5 summarizes the built-in functions. Several programming language embeddings are developed by the team behind Ferry, including Ruby and C# itself, but not Scala; so far, for relational backends only.

A Ferry program is compiled by the pipeline shown in Figure 2, a translation that relies on the *loop lifting* strategy [28] originally developed for the purely relational Pathfinder[2] XQuery compiler. The resulting algebraic query plans are amenable to dataflow-based analysis and optimization [18]. 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 [2] fills a comparable niche. Entity SQL directly supports objects and association navigation.

2.3 Extended example

Using our Scala extension, the developer needs only provide the query shown in Listing 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 LINQTOSCALA compiler plugin, Figure 1) takes charge of parsing and transforming the syntax tree in question into another tree, 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 (from LINQ into Scala) is shown in Listing 2, not in the internal AST representation but as if the query had been written from scratch in Scala (the second use case fully supported by our tooling).

At this point, denotational semantics is our guide to accomplish the 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

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

```
@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 – LinqToScala: Input

```
@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 }
```

Listing 2 – LinqToScala: Output

encodings. The final query to be shipped is shown in Listing 3. The correctness of the translations is addressed in Secs. 4 and 5 resp.

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

Listing 3 – Ferry query ready to ship for database-based evaluation

2.4 Microsoft LINQ

Throughout this paper, 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 textual syntax embedded in languages such as C# and VB.NET into *Standard Query Operators* (SQO), which are comparable to the operators in collection libraries of programming languages supporting closures. For example [15], the textual syntax `from x in foo let y = f(x) select h(x, y)` actually stands for the following code: `foo.Select(x => new { x, y = f(x) }).Select(t0 => h(t0.x, t0.y))`. All we need to know about these expressions 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 [3], the project focusing on parallel

```
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 }
```

Listing 4 – Run-time exception (in LINQ to SQL) at evaluation time

evaluation of LINQ queries, puts it in these terms: "ordering operators re-establish order, shuffle points shuffle the order." As another example, `sum()` will return zero when evaluated on an array all whose elements are `null`, while the same query on an SQL column containing only `NULL`s will result in `null`.

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 [11]. This means that a query provider may be handed a query it cannot evaluate, as shown in Listing 4: upon trying to iterate the resultset, a run-time `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.

3 Levels of integration of host and query languages

A persistent programming language (Sec. 7.2) wallpapers over the different locations and longevity of data. The more modest goal of language-integrated query also poses some challenges, which we classify into the integration levels where they manifest.

3.1 Level 1: Native query syntax

At 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 [1], there are extents for entities and associations. Because usages of variables declared in the host program are disallowed, static typing is attained by relying alone on the typing rules of the query language.

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.

3.2 Level 2: Static guarantee of database evaluation

At 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, in our case 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 run-time, 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 (LHS) or right-hand side (RHS) positions, where a LHS is to be interpreted as binding as opposed to destructive update. In comprehensions-aware query languages (Sec. 2.1), binding is implied by generators and let-declarations only. 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 a certain scope.

When parameterizing queries with run-time values, LHS positions are no problem: they should be fresh names for the scope in question, as 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. 2.2), a capability we retain in all of Levels 1, 2, and 3.

At 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 the type system of the database language (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 5. For instance, Embedded SQL lies halfway between Levels 1 and 2: while some program variables are allowed, not all programming language operators may appear inside queries.

3.3 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

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

Listing 5 – A query statically known to be Level 2, using program variables

run-time type) cannot be statically determined.

At Levels 3 and 4, in order to build the database query to ship, the actual types of program variables are inspected using run-time reflection. This allows computing the database 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 DBMS) 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 Scala query `for (e <- Employees; if e.skills == fashionableSkills) yield e.name` where both `e.skills` and `fashionableSkills` have 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. In order 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.

3.4 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.

3.5 ScalaQL and LINQ under the light of integration levels

The current (beta) version of ScalaQL supports Level 2. Accepting queries at Levels 3 and 4 would require implementing an interpreter to complete query processing on the client-side. Regarding LINQ, different query providers support different levels, in a manner not clearly documented. Anecdotal evidence suggests Level 2 queries to exist that are processed client-side by the LINQ to XML provider. Irrespective of the

status of implementations as of this writing, our approach compares favorably with LINQ when targeting relational backends.

More in general, all existing approaches to language integrated query exhibit slightly different strengths and weaknesses (see Sec. 7) 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.

As discussed in Sec. 2.4 (Listing 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 run-time. Regarding this, all of Levels 1 to 4 do without exceptions of this kind. In ScalaQL, database evaluation may end abruptly due to errors like division by zero. However, exceptions like that in Listing 4 happen for *all* executions of a query, and would have been flagged at compile-time in our approach.

4 LINQ to Scala

We turn to ScalaQL, covering its first stage in this section. The correctness of this translation follows from the denotational semantics of LINQ and Scala comprehensions. After a summary of their analysis [16], the section closes with a discussion of the treatment of grouping and sorting, where Scala library operators are used instead of LINQ's dedicated syntax.

As presented in Sec. 2.1, the notation for comprehensions does not commit to a particular type system or set of operations for 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 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.

In order to translate LINQ comprehensions into Scala comprehensions it suffices to define translations for LINQ constructs whose syntax differs from their Scala counterpart (in particular, for grouping and sorting). Given that each translation rule operates on one fragment of a LINQ expression at a time (for example, the `select` clause), we must guarantee that all variables in scope of the fragment to translate will also be in scope of the resulting fragment. For `select` and other clauses, the variables in scope are those appearing as indices in the binding environments defined in Appendix A. Therefore, the translation rules in Table 1 mirror the organization of the expressions that construct those environments.

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.

4.1 Comprehensions in LINQ and Scala

The input AST for a LINQ query already has *query continuations* inlined [15], *i.e.*, subqueries were recast from the idiosyncratic "`from` x_1 `in` e_1 *`qclauses`* `into` x_2 *`qcont`*"

```
// 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
```

Listing 6 – LINQ and Scala binding environments, handling of `orderby`

into the usual "`from` x_2 `in (from` x_1 `in` e_1 *`qclauses`*`)` *`qcont`*." Therefore, 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*.

The two constructs 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. As already mentioned, for that evaluation to be semantics preserving the same variables that were made to go into scope by preceding body clauses (`FromClause`, `LetClause`, `JoinClause`, and `JoinIntoClause`) must also be made go into scope by the comprehension qualifiers *quals* inside Scala's `for (` *quals* `) yield` e. In contrast, the two remaining kinds of body clauses (`WhereClause` and `OrderByClause`) do not introduce variables. Clearly, variables in Scala should bind to types and values equivalent to their LINQ counterparts. The translation is thus compositional, with the contract just mentioned between qualifiers and head of comprehension.

4.2 Handling of `orderby` and `group by`

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 App. A. 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 6 showcases the translation pattern.

Similarly, LINQ's `group by` also introduces an irregularity as it returns nested collections: `group` *`result`* `by` *`key`* returns a sequence of *groupings*, where each grouping `g` behaves both as (a) a map entry *key* ↦ *cluster*, a cluster being a sequence of results; and (b) as a collection in its own right (the nested collection in question). When `g` is iterated with `from v in g`, the clustered values are returned. The grouping's key is obtained with any of `v.Key` or `g.Key`. Because of (b), a LINQ grouping understands collection operations, as in `g.Count()`. In order to accomodate these differences

Table 1 – Translation rules LINQ → Scala, for clauses producing Scala qualifiers

`from` $T^{0..1}_{type}$ V_{var} `in` E_{exp}	var : $[\![type]\!]$ <- $[\![exp]\!]$
`let` V_{lhs} = E_{rhs}	`val` lhs = $[\![rhs]\!]$
`where` $E_{booltest}$	`if` $[\![booltest]\!]$
`join` $T^{0..1}_{type}$ $V_{innerVar}$ `in` $E_{innerExp}$ `on` E_{lhs} `equals` E_{rhs}	`val` $outerKey_{fresh}$ = $[\![lhs]\!]$; $innerVar$: $[\![type]\!]$ <- $[\![innerExp]\!]$; `if` $outerKey$ == $[\![rhs]\!]$;
`join` $T^{0..1}_{type}$ $V_{innerVar}$ `in` $E_{innerExp}$ `on` E_{lhs} `equals` E_{rhs} `into` $V_{varResult}$	`val` $outerKey_{fresh}$ = $[\![lhs]\!]$; `val` $varResult$ = `for (` $innerVar$: $[\![type]\!]$ <- $[\![innerExp]\!]$; `if` $outerKey$ == $[\![rhs]\!]$ `) yield` $innerVar$

Table 2 – Sample of Ferry's built-in function library [19]

`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
`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
`all; any` ::	$[bool] \to bool$	quantification
`sum; min; max` ::	$[a] \to a$	list aggregation
`groupWith` ::	$(t \to (a_1, \ldots, a_m); [t]) \to [[t]]$	grouping

(Scala's `Map` is not a perfect fit, iterating it yields all key/value pairs, and a pair cannot be iterated), a given key/cluster is wrapped not as a pair but in a custom class implementing the same interfaces as LINQ's `IGrouping<TKey, TSource>`. Afterwards, the translation into Ferry (Sec. 5) retrieves the intended data (key or cluster), based on the name of the function applied to a value of the custom class.

5 Scala to Ferry

The second stage of ScalaQL is covered in this section. Given that not all Scala-level type constructors are available in the relational data model, an isomorphism between Scala and Ferry types is needed. That isomorphism rests upon injective encodings of Scala values, as described in the first two subsections. With that foundation in place, the translation of collection operators and of comprehensions can be addressed. Due to space constraints only a selection of the translation rules from the accompanying technical report [16] are discussed. The approach can handle all Scala types, and the current (beta) implementation of ScalaQL already handles $> 75\%$ of the Scala collection operators.

5.1 Relational encoding of Scala values

In order for the translation from Scala into Ferry to be semantics preserving, the encoding used to ship values for database evaluation must be lossless, *i.e.* each original Scala value must be recoverable given two pieces of information: its relational representation, and the original Scala type (which is tracked statically). As a practical example, consider *tuple flattening*, where Scala nested tuples are represented in Ferry as non-nested ones, with marshalling and query rewriting taking care of preserving the semantics of the source expression. For example, the Scala expressions `val t = (a,(b,c),d); t._2._1` must be recast in Ferry's core sublanguage as `let t = (a,b,c,d) in (t.2, t.3).1`. Although the encoded value alone does not allow recovering its source counterpart (both `((a,b),c,d)` and `(a,b,(c,d))` are flattened to `(a,b,c,d)`, for example) the type information makes the encoding injective.

5.2 Isomorphism of operators, types, and values between Scala and Ferry

Recalling the classification introduced in Sec. 3, Level 3 allows program variables to appear in queries whose values will be shipped to the database by encoding them as Ferry literals. Given that our translation from Scala to Ferry does not delve into the code blocks that define the behavior of operators, dedicated translation rules are needed for each type signature of an operator. In order to keep the catalog of rules within reasonable bounds, we have chosen the following subset of Scala types for our input language: (a) primitive types; (b) enumerations; (c) structural types and case classes, which denote recursive algebraic datatypes; and (d) `List`, `Set`, and `Map`. Although we plan to expand it in future versions of ScalaQL, the provided subset allows expressing many queries commonly occurring in practice.

Techniques from Object/Relational Mapping are reused to encode values of the supported subset in terms of Ferry types. In a nutshell, enumeration values are encoded as integers. Structural types and case classes constitute "datatypes with subtyping", *i.e.*, Scala automatically endows the semantics of structural equivalence to equality tests between them (a behavior that cannot be overridden). Related to subtyping, we impose on source language expressions a restriction from the target language: Ferry does not support non-homogeneous lists. Together with our ban of subtype tests and casts from the source language, the net effect is for subtyping-agnostic queries to be accepted as input by ScalaQL. Finally, set semantics (for example, insertion does not create duplicates) is realized by keeping track of the Scala-type for a collection (list, set, or map); and by expanding Scala operations on them into list-based counterparts. In the example of set insertion, a membership check is performed as part of an `if-then-else` subexpression, to return either the original collection or another with the new element.

As a whole, the single largest omission in ScalaQL is support for objects, *i.e.*, instances of "normal" classes. While possible, handling objects is not prioritary in the current version of ScalaQL, with its focus on total translations, and on setting the groundwork for later extensions. An extension for objects would rely on persistence by reachability, and on records with extra synthetic fields ("object-id", "class-id", created at marshalling time) for use in equality tests.

```
val salaries : List[Int] = ...
val namesWithSalaries = for (empl <− Employees zip salaries; if empl._1._3 == "US"
                            ) yield (empl._1._2, empl._2)
// Ferry : let Employees = table EmployeesTab(id int, name string, dept string )
//             in for empl in zip(Employees, [123.45, ...]) where empl.3 == "US"
//                return (empl.2, empl.4)
```

Listing 7 – Encoding-aware translation for `zip`

```
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))
```

Listing 8 – Counting employees in a given department

5.3 Translation rules: Collection operators

Concerning the translation of collection operations themselves, 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).

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 preclude a one-to-one translation. In these cases, additional encodings and query rewriting are necessary. Listing 7 illustrates a Ferry query that preserves the semantics of Scala's `zip` operator using a combination of 'flattening' and query rewriting, whereas a one-to-one translation would be incorrect. In the example, the return expression `(empl.2, empl.4)` selects the same information as the original Scala expression `(empl._1._2, empl._2)`, retrieving the employee's name and salary from their new positions in the flattened tuple.

Other operators are translated into a bunch of Ferry function invocations, as illustrated in Listing 8 by counting the employees fulfilling a certain condition. Finally, some operators can be translated only for special cases, as with Scala's `reduceLeft` that recursively applies a given binary operation between successive elements of a sequence. When used to aggregate a sum (as shown in Listing 9), or to find the

```
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) )
```

Listing 9 – Summing up employee salaries

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

Listing 10 – Translation bonus: support for patterns

minimum or maximum, a translation into Ferry is possible.

Sec. 4.2 discussed the counterpart in Scala to LINQ's sorting operator (`orderBy`), whose type signature accepted an expression for each sorting criteria and its direction which together define a lexicographic sorting. In addition to the above, Scala brings its own sorting operator (`sortWith`) with a different signature: a comparator function is taken as single argument. Both operators need to be translated into Ferry, as the user may have written a query initially in LINQ or Scala. The translated expression performs (as required) *stable sorting*, *i.e.*, those items that the comparator function reports as equal are listed in the same order in which they appeared in the input.

5.4 Translation of Scala comprehensions

The translation of a Scala comprehension takes into account the variables in scope of each of its *qualifiers* (*generators*, *guards*, and *let-declarations*). Two syntactic differences with Ferry comprehensions that do not compromise expressiveness are cosmetic: (a) Ferry does not allow let-declarations as qualifiers, a restriction that is circumvented by expanding usages with definitions (always possible because let-declarations cannot be recursive); (b) Ferry has room for a single `where` clause after all generators, thus prompting the translator to pack all Scala filters as a single logical conjunction. Otherwise, the translation is straightforward: the order of generators is kept, and the head of the comprehension is translated compositionally.

Other Scala constructs are also handled [16], *e.g.*, *patterns*, used when performing multiple-choice as part of generators, or in let-declarations, as shown in Listing 10.

6 Query optimization

There are several levels at which a language-integrated query can be optimized (broadly speaking, on the client and the server sides). A comparison with LINQ is difficult because optimizations performed by query providers are not discussed in the academic literature, an exception being [7]. For details the reader is invited to consult US Patent Application 20090144229 "Static query optimization for LINQ."

Whenever a comprehension encloses only side-effect free expressions, a number of rewritings become applicable, inspired by relational counterparts: filtering as early as possible, postponing sorting until needed, etc. [29], [8, §3.4], [18]. Regarding queries, two safe reorganizations of comprehension qualifiers are: (a) moving a filter to the earliest position where all its enclosed variables are in scope; (b) moving an `orderby` after the `select` that immediately follows.

In conjunction with the above, client-side optimizations exploit data locality by evaluating (before shipping, for some input known at run-time) those subexpressions that do not depend on database state, using *partial evaluation*. As an example, consider the query in Listing 11 that matches employees with the departments

```
for (d <− departments;
     e <− Employees;
     if (d.approvedPasstimes map (_.name) contains e.hobbys)
yield (d.name, e.name)
```

Listing 11 – A Scala query encompassing main memory (the `departments` program variable) and secondary storage (the `Employees` extent)

```
for d in [ ("d1", ["stamps"] ), ("d2", ["bowling"] ) ],
    e in table Employees ,
    where length(filter (v −> length(filter (v1 −> v1 == v, d.2)) == 0, e.hobbys)
                ) == 0 // i.e., d.2 contains e.hobbys
return (d.name, e.name)
```

Listing 12 – The Ferry query as shipped after applying client-side optimizations

that approve of their hobbys. This query spans both program and database state, as the `departments` program variable holds a `List[Department]`, where the property `approvedPasstimes` of `Department` is declared to have type `Iterable[Passtime]` (sets and lists of passtimes conform thus to that type, given that this immutable `Iterable` is covariant in its argument, with `Passtime` having subtypes other than `Hobby`). On the other hand, the `Employees` extent has a Ferry type, a list of records `( ename: string, hobbys: [string])`, with the second item holding a list of hobby names.

In a very real sense, the evaluation of this query can get away without full-insight of Scala types, by noticing that the subexpression `d.approvedPasstimes map (_.name)` returns just a list of strings. In fact, all the DBMS needs to know about departments is reproduced in Listing 12. In other words, after applying partial evaluation, not the whole transitive closure of values reachable from departments is shipped, but just those data items that may be accessed.

Client-side optimizations are a special case of *data-location-aware optimizations*, as in *distributed execution of functional programs*, with shared-nothing memories and queries that may include RHS variable usages referring to remote data. For unconstrained programs, the flow of bindings is mind numbing. For example, once remote collections are iterated, the iterator variables thus bound may appear in closures that are in turn passed as actual arguments. The resulting optimization problem consists in choosing where to insert RPC calls, and what data to move across machines.

As far as we know, none of the existing approaches to language-integrated query attacks this problem, resorting instead to client-side processing. These capabilities have been studied for *federated databases*, *e.g.* in Ch. 8 of [17] (Optimisation Strategies for Functional Queries in a Distributed Environment). These optimizations are also the goal of data-centric processing in the DryadLINQ [22] project at Microsoft, aiming at automatically translating the data-parallel portions of sequential programs into a distributed execution plan. DryadLINQ focuses on cluster-level parallelism, and internally delegates to PLINQ decisions to achieve multicore-level parallelism.

7 Related Work

Each data model comes equipped with data definition, query, and manipulation sub-languages, the two last ones usually not Turing-complete and thus not qualifying as programming languages. Still, these languages have undergone an increase in expressiveness, as reviewed below for the relational and object data models. A review of related work in the field of deductive databases has been omitted.

In parallel with these developments originating in the database community, programming language research has aimed at *orthogonal persistence*, a technique to uniformly handle values irrespective of their location, types or longevity. Alongside this technique, dedicated support for queries is provided, either in the form of (a) language-integrated query (as advocated in this paper); or by (b.1) recovering from imperative loops the declarative formulation of a query (for read-accesses), and by (b.2) inferring bulk-updates from the side-effects of loops [31].

7.1 Database query languages

Cooper [10, §5] offers a timeline of research results on unnesting of first-order and higher-order relational expressions, leading to the use of comprehension syntax. Kleisli [33] pioneered the area of comprehension queries on relational databases, supporting nested relations as intermediate values and as results. However, a query is not guaranteed to be fully translated into SQL and execution correctness is achieved by bringing into main memory sub-results to complete processing.

Stonebraker and Hellerstein cover the evolution of object databases [21, Ch. 2]. As with their relational counterparts, the lack of a "programming language Esparanto" forced any language-integration effort to be performed once per compiler, with each extension being likely unique (think about the differences between C++ and Lisp). Most of the work thus targeted C++, which meant adopting its data model for the database, and focusing on a market niche: CAD software and engineering applications. This market focus shaped OO databases in three ways: (a) no declarative query language; (b) no need for transaction management; and (c) the performance of algorithms on persistent objects should approach that of main memory. These requirements were not fully aligned with those of business computing.

Unlike its competitors, the O2 database system supported an OO data model, declarative embedded queries, and targeted business data processing rather than the CAD market. While technologically advanced, Stonebraker and Hellerstein conclude that O2 did not gain traction due to business rather than technology decisions.

7.2 Orthogonal persistence

Atkinson and Morrison [6] define orthogonal persistence in terms of three principles: (a) the movement of data between long term and short term storage is exclusive responsibility of the run-time system and not the developer; (b) irrespective of its type, each value can be made persistent; and (c) in languages with pointers, persistence by reachability is supported. In a recent review of the field, Dearle *et al.* [12] state the main goal of persistent programming: supporting the design, construction, maintenance and operation of long-lived, concurrently accessed and potentially large bodies of data and programs.

The last capability (placing code in the database) is a consequence of the principle of type-orthogonality, and is required to avoid anomalies like the following: a program

P_1 populates a database with objects of some type T. Another program P_2 inserts into this collection an object of type T', a subtype of T. If the original program accesses the newly added object and calls methods that have been overridden in T', the invoked code should be determined by late binding. This code has to be kept in the database, as the invoker may be unaware about the existence of T'.

Persistent programming languages have not become mainstream. The technology coming closest, extensions to SQL with imperative constructs (as found in *stored procedures*), have not garnered a large developer mindshare either, apparently for two reasons. First, the offerings from different DBMS vendors exhibit slight incompatibilities that Part 4 of the SQL:1999 standard (Persistent Stored Modules, SQL/PSM) has not managed to re-unite. Second, there are scalability issues when business logic is run in the database and not in the middle tier [23].

Still, the problems that orthogonal persistence set to address are real and thus pain points in the state of the practice. Tools for Object/Relational Mapping have filled this market need, not the least because they require no modifications to compilers nor to DBMSs, although their support for query optimization is basic, relying instead on caching to improve performance.

A lightweight approach to language-integrated queries, *internal DSLs* [13], has sparked several prototypes [15, §2], [27], which compromise on the readability and well-formedness checking of queries in exchange for avoiding extending the compiler.

8 Conclusions

Modern programming languages offer both opportunities and challenges when integrating a sublanguage for database queries. As a technology enabler, the extensibility architecture of modern compilers (specially in connection with *extensible syntax* [5] as in Fortress) opens the door to non-vendor-provided language processing. More fundamentally, we see the expressiveness of the functional-object paradigm (followed by Scala and LINQ) as an asset rather than a liability. In our case, it provided a conceptual framework for the analysis of query semantics preservation.

Our approach improves productivity and quality in software development, by allowing the type-safe embedding of expressive database queries using a familiar notation. Moreover, a careful analysis of the target database language enabled us to expand on the amount of static semantics checking of queries beyond that provided by Microsoft LINQ. Further work is needed to increase the extent of client-side optimization, to further weaken the case for manually tuned queries.

Looking into the future, there is the potential for persistent programming languages to support abstractions only partially supported by database managers: *invariants*, *materialized views*, and *production rules*. Without an expressive language for invariants, neither can realistic integrity constraints be captured, nor efficient analyses be devised to detect their violation. Instead, the state of the practice trusts the (procedurally, manually coded) *business logic* with their enforcement. Regarding materialized views, most work to date has focused on the relational case, an exception being the `MOVIE` system [4] for an OQL subset. Also missing is the *production rules* paradigm, where a certain data constellation (the *activation part* of a rule, expressed as a query) triggers the updates specified in an *action part* [20].

We believe that addressing this more encompassing research and engineering agenda is necessary to realize the vision of persistent programming languages.

Table 3 – LINQ-related production rules

$Q \in$ QueryExp	::=	F_{from} QB_{qbody}
$F \in$ FromClause	::=	**from** $T^{0..1}_{type}$ V_{var} **in** E_{in}
$QB \in$ QueryBody	::=	$B^{0..*}_{qbclauses}$ SG_{sel_gby} $QC^{0..1}_{qcont}$
$B \in$ 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** $T^{0..1}_{type}$ $V_{innervar}$ **in** $E_{innerexp}$ **on** E_{lhs} **equals** E_{rhs}
$K \in$ JoinIntoClause	::=	J_{jc} **into** V_{result}
$O \in$ OrderByClause	::=	**orderby** $U^{1..*\ <separator:,>}_{orderings}$
$U \in$ Ordering	::=	E_{ord} $Direction_{dir}$
Direction	∈	{ **ascending**, **descending** }
$S \in$ SelectClause	::=	**select** E_{selexp}
$G \in$ GroupByClause	::=	**group** E_{e1} **by** E_{e2}

A Appendix: Syntax and Semantics of LINQ

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 3 (listing LINQ-proper productions, with *QueryExp* being the entry rule) and in Table 4 (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

Table 4 – Other syntactic domains

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

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.

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 valuation functions, a query is regarded as a sequence S of body clauses Q, with query continuations desugared into subqueries [15].

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

$$[\![\ \texttt{select}\ E_{selexp}\ ]\!]_{envs} \stackrel{\text{def}}{=} [\ [\![selexp]\!](env) \mid env \leftarrow envs\ ] \tag{1}$$

Informally speaking, `group` *`result`* `by` *`key`* returns a *Grouping*, *i.e.*, a finite *ordered* map with entries $key \mapsto cluster$, a cluster being a sequence of results. 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 syntax common in functional languages like Haskell,

$$[\![\ \texttt{group}\ E_{result}\ \texttt{by}\ E_{key}\ ]\!]_{envs} \stackrel{\text{def}}{=} \texttt{foldl cf []}\ envs \tag{2}$$

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*.

$$[\![\,\texttt{from}\ V_{var}\ \texttt{in}\ E_{srcSeq}\,]\!]_{envs} \stackrel{\text{def}}{=} [\,env' \mid env \leftarrow envs,\ item \leftarrow [\![srcSeq]\!](env),\ let\ env' = env \cup \{var \mapsto item\}\ ] \quad (3)$$

$$[\![\,\texttt{let}\ V_{var} = E_{exp}\,]\!]_{envs} \stackrel{\text{def}}{=} [\,env' \mid env \leftarrow envs,\ let\ env' = env \cup \{var \mapsto [\![exp]\!](env)\ \}\ ] \quad (4)$$

$$[\![\,\texttt{where}\ E_{test}\,]\!]_{envs} \stackrel{\text{def}}{=} [\,env \mid env \leftarrow envs, [\![test]\!](env)\ ] \quad (5)$$

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`,

$$[\![\,\texttt{orderby}\ key_1\ dir_1\ \ldots\ key_n\ dir_n\,]\!]_{envs} \stackrel{\text{def}}{=} \texttt{sortBy comp envs} \quad (6)$$

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).

$$\begin{aligned}[\![\,\texttt{join}\ V_{innerVar}\ \texttt{in}\ E_{isrc}\ \texttt{on}\ E_{outerKey}\ \texttt{equals}\ E_{innerKey}\,]\!]_{envs} \stackrel{\text{def}}{=}\ & [\,ienv \mid env \leftarrow envs, innerItem \leftarrow [\![isrc]\!](env)\,, \\ & let\ ienv\ =\ env \cup \{\,innerVar \mapsto innerItem\,\}\,, \\ & [\![outerKey]\!](env)\ =\ [\![innerKey]\!](ienv)\,] \end{aligned} \quad (7)$$

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

References

[1] EDM. http://msdn.microsoft.com/en-us/magazine/cc700331.aspx.

[2] Entity SQL. http://msdn.microsoft.com/en-us/library/bb387118.aspx.

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

[4] 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.

[5] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Intnl. Workshop FOOL*, 2009. http://www.cs.cmu.edu/~aldrich/FOOL09/allen.pdf.

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

[7] Nicolas Bruno and Pablo Castro. Towards declarative queries on adaptive data structures. In *Intnl. Conf. on Data Engineering*, pages 1249–1258, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

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

[9] 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.

[10] 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)*, pages 36–51. http://ezrakilty.net/pubs/dbpl-sqlizability.pdf.

[11] Bart De Smet. Query validation by query providers. http://bartdesmet.net/blogs/bart/archive/2007/07/05/linq-to-sharepoint-improving-the-parser-debugger-visualizer-fun.aspx.

[12] Alan Dearle, Graham N.C. Kirby, and Ron Morrison. Orthogonal persistence revisited. In Moira C. Norris and Michael Grossniklaus, editors, *Proc. of the 2nd Intnl. Conf. ICOODB 2009*, pages 1–22, July 2009. http://www.cs.st-andrews.ac.uk/files/publications/download/DKM09a.pdf.

[13] Gilles Dubochet. On Embedding Domain-specific Languages with User-friendly Syntax. In *1st Workshop on Domain-Specific Program Development*, pages 19–22, 2006. http://infoscience.epfl.ch/record/85862.

[14] 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.

[15] Miguel Garcia. Compiler plugins can handle nested languages: AST-level expansion of LINQ queries for Java. In Moira C. Norris and Michael Grossniklaus, editors, *Proc. of ICOODB 2009*, pages 41–58, July 2009. http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2009/icoodb/compplugin.pdf.

[16] Miguel Garcia and Anastasia Izmaylova. Compiling LINQ and a Scala subset into SQL:1999. Technical report, Software Systems Institute (STS), Technische Universität Hamburg-Harburg, Germany, Sep 2009. http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2009/ScalaQLTechRep01.pdf.

[17] 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.

[18] 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*, pages 1167–1170. Extended version at http://arxiv.org/abs/0810.4809.

[19] 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.

[20] 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.

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

[22] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD'09: Proc. of the 35th SIGMOD Intnl. Conf. on Mgmt. of Data*, pages 987–994, New York, NY, USA, 2009. ACM.

[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] Adriaan Moors. *Type Constructor Polymorphism for Scala: Theory and Practice*. PhD thesis, Katholieke Universiteit Leuven, May 2009. https://lirias.kuleuven.be/handle/1979/2642.

[25] Anders Bach Nielsen. Compiler phase and plug-in initialization for Scala 2.8, 2008. Scala Improvement Document 2, http://www.scala-lang.org/sid/2.

[26] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. artima, Mountain View, Calif., 1st edition, 2008.

[27] Daniel Spiewak and Tian Zhao. ScalaQL: Language-integrated database queries for Scala. In *Software Language Engineering: 2nd Intnl. Conf., SLE 2009, Denver, Colorado, USA*, 2009. To appear. http://www.cs.uwm.edu/~dspiewak/papers/scalaql.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] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. *SIGPLAN Not.*, 43(10):19–36, 2008.

[32] 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.

[33] 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.

About the authors

Miguel Garcia completed his PhD on formal methods for model-driven software engineering at the Institute for Software Systems, Hamburg University of Technology (Germany). His current research is reported at `http://www.sts.tu-harburg.de/people/mi.garcia`.

Anastasia Izmaylova graduated from the Bauman Moscow State Technical University with an MSc degree in Applied Mathematics. Currently, she is finishing an MSc in Information and Media Technologies at the Hamburg University of Technology (Germany). She can be reached at `FSAnastasia@gmail.com`.

Sibylle Schupp is Professor and Head of the Institute for Software Systems at Hamburg University of Technology. See `http://www.sts.tu-harburg.de/people/schupp/` for contact details.