ScalaQL: Language-Integrated Database Queries for Scala

Daniel Spiewak and Tian Zhao

University of Wisconsin – Milwaukee
{dspiewak,tzhao}@uwm.edu

Abstract. One of the most ubiquitous elements of modern computing is the relational database. Very few modern applications are created *without* some sort of database backend. Unfortunately, relational database concepts are fundamentally very different from those used in general-purpose programming languages. This creates an impedance mismatch between the the application and the database layers. One solution to this problem which has been gaining traction in the .NET family of languages is Language-Integrated Queries (LINQ). That is, the embedding of database queries within application code in a way that is statically checked and type safe. Unfortunately, certain language changes or core design elements were necessary to make this embedding possible. We present a framework which implements this concept of type safe embedded queries in Scala without any modifications to the language itself. The entire framework is implemented by leveraging existing language features (particularly `for`-comprehensions).

1 Introduction

One of the most persistent problems in modern application development is that of logical, maintainable access to a relational database. One of the primary aspects of this problem is impedance mismatch [7] between the relational model and the paradigm employed by most general-purpose programming languages. Concepts are expressed very differently in a relational database than in a standard memory model. As a result, any attempt to adapt one to the other usually results in an interface which works well for most of the time, but occasionally produces strange and unintuitive results.

One solution to this problem of conceptual orthogonality is to "give up" attempting to adapt one world to the other. Instead of forcing objects into the database or tables into the memory model, it is possible to simply allow the conceptual paradigms to remain separate. This school of thought says that the application layer should retrieve data as necessary from the relational store by using concepts *native* to a relational database: declarative query languages such as SQL. This allows complete flexibility on the database side in terms of how the data can be expressed in the abstract schema. It also gives the application layer a lot of freedom in how it deals with the extracted data. As there is no relational store to constrain language features, the application is able to deal with data on

its own terms. All of the conflict between the dissonant concepts is relegated to a discrete segment of the application.

This is by far the simplest approach to application-level database access, but it is also the most error-prone. Generally speaking, this technique is implemented by embedding relational queries within application code in the form of raw character strings. These queries are unparsed and completely unchecked until runtime, at which point they are passed to the database and their results converted using more repetitive and unchecked routines. It is incredibly easy even for experienced developers to make mistakes in the creation of these queries. Even excluding simple typos, it is always possible to confuse identifier names, function arities or even data types. Worse yet, the process of constructing a query in string form can also lead to serious security vulnerabilities — most commonly SQL injection. None of these problems can be found ahead of time without special analysis.

The Holy Grail of embedded queries is to find some way to make the host language compiler aware of the query and capable of statically eliminating these runtime issues. As it turns out, this is possible within many of the .NET language family through a framework known as LINQ [8]. Queries are expressed using language-level constructs which can be verified at compile-time. Furthermore, queries specified using LINQ also gain a high degree of composability, meaning that elements common to several queries can often be factored into a single location, improving maintainability and reducing the risk of mistakes. It is very easy to use LINQ to create a trivial database query requesting the names of all people over the age of 18:

```
var Names = from p in Person
            where p.Age > 18
            select p.Name;
```

This will evaluate (at runtime) an SQL query of the following form:

```
SELECT name FROM people WHERE age > 18
```

Unfortunately, this sort of embedding requires certain language features which are absent from most non-homoiconic [10] languages. Specifically, the LINQ framework needs the ability to directly analyze the structure of the query at runtime. In the query above, we are filtering the query results according to the expression `p.Age > 18`. C# evaluation uses call-by-value semantics, meaning that this expression *should* evaluate to a `bool`. However, we don't actually want this expression to evaluate. LINQ needs to somehow inspect this expression to determine the equivalent SQL in the query generation step. This is where the added language features come into play.

While it is possible for Microsoft to simply extend their language with this particular feature, lowly application developers are not so fortunate. For example, there is no way for anyone (outside of Sun Microsystems) to implement any form of LINQ within Java because of the language modifications which would be required. We faced a similar problem attempting to implement LINQ in Scala.

Fortunately, Scala is actually powerful enough in and of itself to implement a form of LINQ even without adding support for expression trees. Through a combination of operator overloading, implicit conversions, and controlled call-by-name semantics, we have been able to achieve the same effect without making any changes to the language itself. In this paper, we present not only the resulting Scala framework, but also a general technique for implementing other such internal DSLs requiring advanced analysis and inspection prior to evaluation.

Note that throughout this paper, we use the term "internal DSL" [4] to refer to a domain-specific language encoded as an API within a host language (such as Haskell or Scala). We prefer this term over the often-used "embedded DSL" as it forms an obvious counterpoint to "external DSL", a widely-accepted term for a domain-specific language (possibly not even Turing Complete) which is parsed and evaluated just like a general-purpose language, independent of any host language.

In the rest of the paper, Section 2 introduces ScalaQL and shows some examples of its use. Section 3 gives a general overview of the implementation and the way in which arbitrary expression trees may be generated in pure Scala. Finally, Section 4 draws some basic comparisons with LINQ, HaskellDB and similar efforts in Scala and other languages.

2 ScalaQL

The entire ScalaQL DSL is oriented around a single Scala construct: the `for`-comprehension. This language feature is something of an amalgamation of Haskell's `do`-notation and its list-comprehensions, rendered within a syntax which looks decidedly like Java's enhanced `for`-loops. One trivial application of this construct might be to construct a sequence of 2-tuples of all integers between 0 and 5 such that their sum is even:

```
val tuples = for {
  x <- 0 to 5
  y <- 0 to 5
  if (x + y) % 2 == 0
} yield (x, y)
```

There are really three separate components to this syntax. The first is the generator (e.g. `x <- ...`), which sets up the local variable `x` containing the current element in the comprehension. The second component is the filter (`if ...`), which defines the conditions under which this comprehension holds. Finally, we have the `yield` clause, which defines the result in terms of the variables set up by the generator(s). There may be any number of generators and filters, but only one `yield`.

Every `for`-comprehension is parsed into a corresponding series of calls to methods `flatMap`, `map` and `filter`.[1] The `map` and `filter` methods are standard

[1] Unless the `for`-comprehension lacks a `yield`, in which case `foreach` replaces `flatMap` and `map`.

higher-order utility functions. The `flatMap` method is effectively Scala's version of Haskell's `>>=` operator (monadic *bind*). It is defined for collections as a composition of the `map` and `flatten` functions. By rewriting `for`-comprehensions in terms of other language elements at parse time, Scala empowers third-party frameworks (such as ScalaQL) to exploit the syntax simply by implementing the relevant methods.

Altogether, this syntax provides a way of working with Scala collections in an almost declarative fashion reminiscent of a query language. In fact, it is possible to make use of `for`-comprehensions to perform SQL-like queries against Scala collections. For example:

```
// regular Scala collections, not ScalaQL

val people: List[Person] = ...
val companies: List[Company] = ...

val underAge = for {
  p <- people
  c <- companies

  if p.company == c
  if p.age < 14
} yield p
```

This expression yields a `List` of all people under the age of 14 who are employed by some company. If we were to formulate this same query in SQL, the result would be something like this:

```
SELECT p.* FROM people p JOIN companies c ON p.company_id = c.id
  WHERE p.age < 14
```

Intuitively, `for`-comprehensions are a natural syntactic device for representing declarative queries against generic collections. ScalaQL makes it possible to use that same syntax to represent database queries. Using ScalaQL, we can take our query example from earlier and slightly adapt it into something that will actually run against a database:

```
val underAge = for {
  p <- Person
  c <- Company

  if p.company is c
  if p.age < 14
} yield p
```

Recall that `for`-comprehensions are translated into a corresponding series of calls to `flatMap`, `map` and `filter`. In this case, the first (outermost) call to `flatMap` will be targetted on the `Person` object. This is what allows ScalaQL to "hijack"

the `for`-comprehension syntax. `Person` must implement — or inherit from a type which implements — the `flatMap` and `map` methods such that an an abstract representation of the query is produced (see Section 3).

The primary syntactic difference between this and the same query run against Scala `List`(s) is the use of the `is` operator (rather than `==`) to test equality. This is necessary because of the way that Scala handles the `==` method.[2] Amazingly enough, it is the only syntactic concession made by the framework. All other `String`, `Int` and `Boolean` operators work exactly as expected. For example, the `<` operator is used above to compare `p.age` to the integer literal, `14`.

The above expression will produce an instance of `Query[Person]`, one which will produce a sequence of `Person` entities when evaluated. ScalaQL does not evaluate queries at declaration point. Instead, evaluation is deferred until the query is actually *used* as a sequence. For example:

```
underAge foreach { p => println(p.firstName + ' ' + p.lastName) }
```

The `foreach` method is not declared for type `Query`. When the Scala compiler sees this invocation, it determines that an implicit conversion from `Query[Person]` to `Seq[Person]` is required in order to make everything work. This implicit conversion is transparently injected into the bytecode by the Scala compiler and invoked at runtime just prior to the invocation of `foreach`. It is this implicit conversion, defined by ScalaQL, which handles the query evaluation.

The primary advantage to this deferred evaluation is it allows queries to be treated compositionally. For example, we might want to construct a query which finds all of the under-age employees working at MegaCorp. Rather than redundantly defining the query constraints for under-age workers, we can simply build our new query by composing with the old:

```
val megaCorpEmps = for {
  p <- underAge
  if p.company.name is "MegaCorp"
} yield p
```

If we were to evaluate the `megaCorpEmps` query, it would execute SQL against the database very similar to the following:

```
SELECT p.* FROM people p JOIN companies c ON p.company_id = c.id
  WHERE p.age < 14 AND c.name = 'MegaCorp'
```

2.1 Projection

So far, all of the queries we have expressed using ScalaQL have had a very simple `yield` statement, producing an instance of `Query` parameterized against

[2] Unlike other symbolic methods, Scala defines `==` as an alias for `equals`. Our experiments revealed some bugs in Scala's type checker when either `equals` or `==` are defined to return anything other than `Boolean` (unrelated and well-formed sections of code would arbitrarily fail to type-check).

an entity type. ScalaQL is also capable of projecting on single fields as well as arbitrary record types defined as anonymous classes. This makes it possible to define type safe projections with arbitrary fields.

Single-field and single-expression projection works exactly as expected. We define our `yield` clause in terms of the row locals defined in the generators (e.g. `p` or `c`), using fields, operators and values in the same fashion as in the filters. For example:

```
val names = for {
  p <- Person
  if p.age > 18
} yield p.lastName
```

This defines an instance of type `Query[Varchar]` which produces the last names of all of the people in the database over the age of 18. With a few slight modifications, we can actually produce the concatenation of the first and last names in the standard "Last, First" format:

```
val names = for {
  p <- Person
  if p.age > 18
} yield p.lastName + ", " + p.firstName
```

When evaluated, this query will execute SQL similar to the following:

```
SELECT CONCAT(CONCAT(p.last_name, ', '), p.first_name)
  FROM people p
  WHERE p.age > 18
```

One particularly thorny aspect of projection which has been a difficult area for similar query DSLs in the past is that of multi-field projection. In SQL, it is possible to construct a query which produces a subset of the resulting fields; not just one field, but several. This is difficult because it requires the ad-hoc definition of new record types corresponding to the fields in question. While classes are technically a form of record type, very few languages sufficiently facilitate the definition of classes on a case-by-case basis. When each query requires a different record type (class) for its projection, query definition becomes a very tedious affair.

Fortunately, Scala provides a lightweight syntax for defining Java-style anonymous inner-classes which extend `AnyRef`. This syntax (which actually comes from C#) makes it easy to define new classes at query-site without becoming syntactically burdensome:

```
val people = for {
  p <- Person
  if p.age > 18
} yield new {
  val firstName = p.firstName
  val lastName = p.lastName
}
```

This query selects only the `first_name` and `last_name` fields from the `people` table. The `new { ... }` syntax defines a new anonymous inner-class containing two fields: `firstName` and `lastName`. This type will be used to populate the query results. Thus, the type of the `people` value is `Query[$t]`, where `$t` is the type of the anonymous inner-class (this type is hidden by Scala's type inference, hence the use of the "`$t`" notation). We can demonstrate this fact by iterating over the query results and accessing fields:

```
people foreach { p => println(p.firstName + ' ' + p.lastName) }
```

3 Implementation

The most important guiding concept of ScalaQL's implementation is that of the abstract query tree, which is similar in principle to an abstract syntax tree used in the implementation of most programming languages. Unlike most internal DSLs, ScalaQL does not immediately evaluate the invocation syntax into a final result. Instead, it creates an abstract representation of the desired query in an AST-like structure. This structure is what is actually contained by a value of type `Query`. When the `Query` is converted to a `Seq`, the abstract query tree is converted into the corresponding SQL, which is evaluated against the database to produce the final result.

The query tree is composed of three elements: views, projections and expressions. Views directly correspond to relations in relational algebra and may be either tables or queries (another abstract query tree). Projections have three different forms, each corresponding to one of the three different projection types supported by ScalaQL: single field, single table and field subset. Projections may also contain expressions in cases where the `yield` clause is not a simple field or entity:

```
for {
  p <- Person
} yield p.firstName + " " + p.lastName
```

Expressions are where most of the interest lies. The addition of abstract expression trees as first-class values was one of the primary changes in C# 3.0 as required by LINQ. Since Scala does not have this feature, we must find a way to construct expression trees using a different approach.

The solution is a combination of implicit conversions and operator overloading. In the above example, we have given the sub-expression `p.firstName + " "`. While `p.firstName` may appear to be a field of type `String`, it actually has type `Varchar`, which extends the `StringExpression` class. This class defines a number of methods, including `+`, an operator which takes another `StringExpression` as a parameter. We have defined an implicit conversion from type `String` to `StringExpression`, allowing literal strings to be concatenated onto abstract `StringExpression`(s). The result of this `+` method is an abstract expression node, `AddStr`, which also extends `StringExpression`.

Of course, strings are not the only data type manipulated by SQL expressions. For this reason, we have also created implementations for `NumericExpression`, `BooleanExpression` and `TimeExpression`. Each of these classes defines operator methods according to how their respective type is expected to behave. Thus, `NumericExpression` defines `+`, `*`, `%` and more, while `BooleanExpression` defines `&&`, `||` and so on. Every expression class extends `Expression`, which defines operator methods common to all expressions: `is` and `!=`.

All of these operations return abstract expression nodes representing the specific operation in question. These nodes each resolve to a different SQL operation or function, making it possible to effectively compile Scala expressions into SQL at runtime. In a sense, the expression DSL parses code which *appears* to be conventional `String`, `Int` and `Boolean` expressions into a structure very reminiscent of a compiler's abstract syntax tree. This tree can then undergo a code generation phase, which produces the corresponding SQL.

Type safety is ensured by the fact that the operator methods in each expression class will only accept certain parameter types. Thus, it is impossible to concatenate a `StringExpression` and a `NumericExpression`; the `+` operator method in `StringExpression` only accepts another `StringExpression`. Inherited operator methods like `is` and `!=` are guaranteed type safety through the use of an abstract type declared in the `Expression` superclass. This type effectively allows the parameters for any operator methods in `Expression` to vary *covariantly* with subtyping, ensuring that it is impossible to test a `NumericExpression` and a `BooleanExpression` for equality.

The other advantage to this approach in general (besides type safety) is that it allows optimizations and other in-depth analysis to be performed against the abstract expression tree prior to resolution (code generation). Normally, a DSL evaluates directly to its final result, making it very difficult to perform any sort of non-trivial processing on the instructions. This is because direct evaluation effectively restricts any processing to a single pass over the instructions. By evaluating to an intermediate form (the expression tree), we make it possible to perform multi-pass analysis (including optimization) against a complete representation of the DSL instructions.

4 Related Work

SQLJ [9] embeds SQL into Java and is statically typed. However, dynamic queries are not supported as every SQLJ query is converted in a pre-compilation step. While not technically a language extension, SQLJ is certainly not "plain-old Java". SchemeQL [12] is similar to SQLJ in that it processes embedded query statements using an external preprocessor, but without providing any static typing. Safe Query Object [1] achieves many of the same goals as SQLJ, all while working within regular Java syntax. Users specify queries using special Java classes which are compiled into JDO queries. Safe Query Object also supports a wide variety of query operations including existential quantification, parameters and dynamic queries. However, like SQLJ, a special compilation step is

required to perform the conversion. As mentioned previously, systems such as LINQ [8] support SQL-like queries through language extensions. Java Language Extender [11] is another framework which operates in this fashion.

Of all of the projects in this field, HaskellDB [6] is likely the most similar to our approach in that it functions as an internal domain-specific language. Operations such as filter, join and conditionals are all supported in a statically checked, type safe environment provided by Haskell's type system. However, Haskell imposes heavier restrictions on function overloading than does Scala. Thus, HaskellDB is forced to use operators like `.+.` instead of the more familiar `+` when summing query values. Also, Scala's implicit conversions are in some ways more powerful than Haskell's type classes. ScalaQL allows the use of integer literals directly in query expressions, while HaskellDB requires the explicit use of the `constant` function.

Related to HaskellDB is the Pan language [3]. While Pan has very little to do with database queries, it does demonstrate the power of internal DSL construction with an intermediate form. Like ScalaQL, Pan relies on carefully-constructed ADTs to statically ensure well-formedness of DSL expressions. The authors of Pan also discuss ways in which the intermediate form of the DSL may be leveraged in the implementation of advanced optimizations and analyses.

The AraRat [5] framework provides similar query functionality in C++ through the use of preprocessor directives, operator overloading and templates. Its focus is primarily on directly representing relational algebra within the syntax of C++, rather than a more "familiar" dialect like SQL. Thus, a join is represented using the `*` operator, rather than through a more mainstream nomenclature. AraRat does share what is perhaps ScalaQL's most important feature in that it represents views in their abstract form, allowing queries to be highly compositional and easily optimized. AraRat provides a large amount of type safety in the construction and composition of queries, but it does not extend that safety to the *evaluation* of those queries and subsequent parsing of the results. Queries are simply converted to `char*` using the `asSQL()` function. This differs from ScalaQL, which converts abstract views into properly type safe sequences during evaluation. This limitation is not entirely surprising given the fact that C++ lacks a generic database access framework like JDBC.

Various non-academic efforts have also been made to solve this problem of language embedded queries based on real-world requirements. Ambition [2] is a widely-used internal DSL for Ruby which provides a very natural syntax for constructing queries. Notably, its core framework is not restricted to merely database access; it has also been applied to other query domains such as LDAP and XPath. However, as can be expected from a framework designed for a dynamically-typed language like Ruby, Ambition provides no static guarantees regarding query correctness.

A project very similar to ScalaQL has been developed independently by Stefan Zeiger [13]. Like ScalaQL, this project aims to provide a framework for type safe queries within Scala using `for`-comprehensions. However, despite this similarity, there are some important differences. ScalaQL makes use of the pseudo-

monadic `filter` operation for declaring query conditionals, allowing the use of the `if` syntax in `for`-comprehensions. Zeiger's framework defines a separate series of methods for this (though it can use `filter` for some conditionals). Projection differs greatly between the frameworks, with ScalaQL relying on anonymous inner-classes while Zeiger's framework uses field combinators to generate arbitrary views (e.g. `firstName - lastName - age`).

5 Summary

In this paper, we have given a brief overview of the ScalaQL framework, focusing specifically on static type safety and syntactic intuitiveness. By exploiting the existing `for`-comprehension construct, ScalaQL blends seamlessly with conventional query-like operations performed on Scala collections. We predict that ScalaQL — or something like it — will become an important part of general-purpose Scala ORM frameworks in the future.

References

1. W. R. Cook and S. Rai. Safe Query Objects: Statically typed objects as remotely executable queries. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 97–106, 2005.
2. Chris Defunkt. Ruby's Ambition. `http://ambition.rubyforge.org/`, 2008.
3. Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling Embedded Languages. *Journal of Functional Programming*, 13(03):455–481, 2003.
4. Martin Fowler. Domain Specific Language. `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`, 2007.
5. Joseph (Yossi) Gil and Keren Lenz. Simple and safe SQL queries with C++ templates. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 13–24, 2007.
6. D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, 1999.
7. David Maier. Representing database programs as objects. In *Advances in database programming languages*, pages 377–386. ACM, 1990.
8. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM Symposium on Principles Database Systems*, 2006.
9. Jim Melton and Andrew Eisenberg. *Understanding SQL and Java together: a guide to SQLJ, JDBC, and related technologies*. Morgan Kaufmann, 2000.
10. Guy L. Steele, Jr. *Common LISP: the language*. Digital Press, 1984.
11. Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and Eric Johnson. Adding domain-specific and general purpose language features to java with the java language extender. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 728–729, 2006.
12. Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL: Two little languages. In *Third Workshop on Scheme and Functional Programming*, 2002.
13. Stefan Zeiger. A Type-Safe Database Query DSL for Scala. `http://szeiger.de/blog/2008/12/21/a-type-safe-database-query-dsl-for-scala/`, 2008.