Type-safe SQL embedded in Scala

Christoph Wulf
University of Kiel,
Germany
cwu@informatik.uni-kiel.de

Abstract

In contrast to the industry's need of relational databases the interaction between programs and databases using the Structured Query Language is often inconvenient and error-prone.
This paper offers an approach for embedded SQL statements in Scala with validation against the corresponding database schema and result type inference based on re-writing to plain Scala code. It describes ideas of design and implementation.

1. Introduction

Most of today's industry software are programs written in a high level language, which insert, modify and read data into and out of relational databases. For whatever reason the integration between programming languages and databases using the Structured Query Language (SQL) [1] is suprisingly uncomfortable. Statements are treated as strings in most techniques, which causes the runtime of a program to be the compile time of the statements. Most errors which could be eliminated by validating statements against the database schema at compile time occur at runtime first. Due to this fact developers often avoid efficient but complex statements and solve problems less efficient in the host language.
Scala already contains a notation for embedded XML, which is re-written to Scala components for XML by the compiler. This paper discusses a notation for type-safe embedded SQL queries in Scala programs.
Type-safety applies in this case in two manners: On one hand Scala's built-in type inference is used to infer the result set type of an embedded query. On the other hand the pre-processor for embedded statements delegates the type-checking - whether compared columns are comparable or SQL functions are applied to columns with correct types for example - to the Scala compiler if possible.

- In Section 2 I describe problems of persistence frameworks inherited from Java as a motivation for embedded SQL.
- Section 3 introduces embedded statements from the user's point of view.
- I discuss the parsing of embedded SQL statements in Section 4.
- Section 5 deals with the type-safe result processing.
- I discuss in Section 6 how the re-writing of embedded statements can be implemented. I analyze which embedded statements can be supported if they are re-written directly by a pre-processor without the knowledge of the program's context or the underlying database schema. This section also covers how Scala's type inference is used to determine the type of a result set automatically.
- Section 7 covers columns declared to be optional (nullable) in the database schema and the mapping of outer joins.
- Although this approach neither claims nor intends to be a object-relational mapper it contains some ORM features. In my opinion there is no need for another ORM tool. For that reason I discuss the option for an extension to the persistence of the Lift framework [2] by these embedded statements for bulk queries in Section 8.

To avoid misunderstandings I write *comiler* for the regular Scala compiler and *pre-processor* for a new compiler phase that re-writes embedded SQL statements to plain Scala code.

2. Motivation

Since Scala is interoperable with Java, database access techniques and frameworks for Java are (usually) also techniques for Scala. As these database frameworks have been designed for Java and not for Scala they do not use of course all language features provided by Scala.
This section describes the disadvantages of several Java technologies for database access in general and especially in the context of Scala.

2.1 Java Database Connectivity

The Java Database Connectivity [3] is the basic technology for database access in Java since most database vendors offer drivers for JDBC. Due to Scala's interoperability with Java, programmers can use JDBC in Scala directly.

```
val sql = "SELEC id, name FROM user WHERE lastLogin > ?"
val stm = conn.prepareStatement(sql)
stm.setDate(1, someDate)
val rs = stm.executeQuery
while (rs.next) {
  handleUser(rs.getInt(1), rs.getString(2))
}
```

Listing 1. JDBC in Scala

Listing 1 shows a the execution and result set handling of a SQL statement using JDBC. The query itself is provided as a string. Precisely because the query is just a character string, which could be also reverse, empty or anything else but a SQL query the compiler is not able to check the query for correct syntax and semantics:

- `SELECT` is written wrong, which is even not dependent on any schema.
- The table name could be wrong. Maybe it is `users` instead of `user`.
- The type of `someDate` may be not compatible to the type of the `lastLogin` column (more precisely: convertable to a SQL type, which is not comparable with the type of `someDate`).
- It is assumed by the result handling that `id` is numeric using `getInt`, which may be of type `String` if a user's identification is his login name. This problem does not affect the query, which would for its share be correct. But the result handling would cause an exception by reason of wrong casting.

All these problems may (or will in case of the first point) be exposed at run-time. They could be avoided if the compiler is able to differentiate between a string and a statement, simply verify statements in terms of syntax and validate them against a database schema.

2.2 Object-relational mapping with JPA

The Java Persistence API [4] was introduced in the Java Enterprise Edition 5 as the specification of an object-relational mapping framework. It was most influenced by the framework Hibernate [5], which is one of the implementations for the JPA specification. Both provide a query language (JPA-QL and HQL respectively), which are derivations of SQL but not equal to SQL as they are focused on querying data from an object-oriented point of view.

```
val jpaql = "FROM User u WHERE u.lastLogin > ?"
val query = em.createQuery(jpaql)
query.setDate(1, someDate)
val rs = query.getResultList
rs.asInstanceOf[List[User]].map(handleUser _)
```

Listing 2. JPA Query Language in Scala

Listing 2 shows the execution of a JPA query. The query now selects from a class (`User`) and not from a database table or view. The JPA implementation calculates the corresponding table, executes the query transformed to SQL and maps the result set columns to fields of dynamically created instances of `User`.
All this happens at run-time, which causes several problems:

- `User` is a class, but not connected with the substring `"User"` in the query string.
- Even if the class `User` is used in the query, it is not required to add it to the import declarations.
- The query is not validated against the class `User` at compile time, whether a field `lastLogin` exists and is compatible to `someDate`.
- As the compiler cannot interpret the query string it is not able to infer the result type to be `List[User]`. An explicit casting using `asInstanceOf[List[User]]` is required.

In addition to that, JPA as a framework designed for Java is focused on objects. The JPA-QL supports several object-related notations in addition to SQL while the power of SQL is limited since complex statements (like recursion or special aggregation) are not allowed. Although JPA/Hibernate allows the mapping of database relations to classes, it is intended to map objects to database relations and calculate create statements for them.
Since Scala is an object-functional language, developers for Scala frameworks to access database are not limited to an object-oriented point of view. Scala supports tuples, which match database relations by nature.

2.3 SQLJ

SQLJ is a technique to embed SQL statements into Java code [3] like shown in **Listing 3**. These embedded statements are compiled to JDBC code, checked for syntax and optionally validated against the schema of a given JDBC connection.

```
#sql iterator UserIterator(Integer id, String name);
UserIterator iter;
#sql [ctx] iter = {
  SELECT id, name
  FROM user
  WHERE lastLogin > :someDate
};
while (iter.next()) {
  #sql {
    FETCH :iter
    INTO :id, :name
  };
  handleUser(iter.id(), iter.name())
}
iter.close();
```

Listing 3. SQLJ in Java

The general Java compiler does not support the embedded SQL notation (most likely because of the extremly limited extensibility of Sun's Java compiler for common developers). For that reason an additional compiler is required, which compiles files with the extension `.sqlj` to Java source

files.
As SQLJ was designed for Java it does not support type inference. It was designed for a Java version before 1.5. For that reason it does not support Java generics either. It is necessary to define the type of the result set iterator, which could be inferred using the selected columns.

In the end SQLJ is not usable with from Scala because the SQLJ compiler does not support Scala code but it offers several clues for SQL embedded in Scala.

2.4 Complex statements

Listing 4 shows a recursive statement to calculate the number of users directly or indirectly invited by the user with id 42. Such a recursive query is an efficient alternative, if just a part of the transitive hull is required and the calculation of the whole transitive hull of a relation is too expensive. The statement written in plain JDBC as concatenation of strings in multiple lines seems to be a foreign particle in the Java/Scala code. A recursive statement of this form cannot be expressed in HQL/JPAQL for now.

```
WITH RECURSIVE invitationTree (invitor, invited) AS (
  SELECT r.id, s.id
  FROM user r JOIN user s ON r.id = s.invitor
  UNION ALL
  SELECT t.invitor, u.id
  FROM invitationTree t, user u WHERE t.invited = u.invitor
)
SELECT invitor, COUNT(invited)
FROM invitationTree GROUP BY invitor HAVING invitor = 42
```

Listing 4. Recursive SQL statement

The lack of syntax checking and type-safety induces developers to solve problems with multiple queries in the host programming language. This revokes of course the optimization potential of the RDBMS for complex statements including sub queries or even recursion.

3. Embedded SQL with type inference

The Scala compiler already allows the definition of embedded XML documents [6], which are pre-processed to pure Scala classes. The compiler can be extended in a similar way to support embedded SQL statements. In contrast to SQLJ these statements would be translated into Scala by the Scala compiler itself and not by an additonal compiler. SQL statements would fit directly into Scala code without required delimiters like `#sql`. Futhermore the generated code can take use of generic types to provide a type-safe result handling.
Compiling embedded XML is admittedly much simpler than embedded SQL statements since start and end of a valid XML document can be recognized by opening and end tags.

```
val stm = SELECT id, name
          FROM user
          WHERE lastLogin > { someDate }
// Type could be inferred to:
// stm : BagStatement[(Int,String)]
```

Listing 5. Embedded statement

The statement in **Listing 5** started by **SELECT** can be parsed using the (extended) Backus-Naur Form of **SELECT** statements [7], which will be described in Section 4. The grammar for selecting queries is extended by the curly braces which support the usage of Scala expressions using the scope which surrounds the embedded query. This is similar to the usage of string expressions in curly braces in embedded XML. If the SQL statement is successfully syntax checked (which is indeed possible in this case) and validated against a schema containing a relation `user` with columns `id` and `name` as well as a column `lastLogin` whose type is comparable to the type of the Scala variable `someDate`, the statement can be re-written in a way that the type of `stm` is inferred by the compiler to `BagStatement[(Int,String)]`.

For the user's point of view `BagStatement` is a class that contains the SQL query encoded as Scala elements including the projection to extract a row of the result (`p`). The compiler calculates `(Int,String)` as result type since two columns are projected and the columns `id` and `name` have been encoded as mapped to `Int` and `String`.

```
class BagStatement[T](..., p : Projection[T]) ... {

  ... // encoded query

  // performs the query for a JDBC connection
  // and returns an iterator for the result set
  def execute(conn : Connection) : ResultSetIterator[T] = {
    val rs = conn.createStatement.executeQuery(query.toSql)
    new ResultSetIterator[T](rs) {
      def next = p(rs)
    }
  }
}
```

Listing 6. Abstract class `BagStatement`

The `BagStatement` can be executed for a given JDBC connection by `execute`, which serializes the encoded query to a SQL string, performs the query and returns an iterator for the inferred result type. A trait `ResultSetIterator` is used to close result set and statement to release ressources at the end of the set.

Due the calculated result type the result set can be processed without any casting:

```
stm.execute(connection).map {
  case (id,name) => handleUser(id,name)
}
```

Listing 7. Type-safe result set processing

4. Recognition of statements

Embedded statements can be recognized by the pre-processor in case of one of the SQL keywords for a selecting query (**SELECT**, **WITH**), an update statement (**INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**) or **CALL** for a stored procedure.
These keywords are usually allowed in uppercase, lowercase or a mix of uppercase and lowercase letters. It would be a strict limitation to declare all these SQL keywords as Scala keywords as well. This is even impossible in the case

of **with** (used for recursive queries amongst others), which is already a keyword in Scala for mixins. For sure many Scala programs could not be compiled if functions like `insert` or classes like `Call` would not be allowed anymore.
In opposition, it is not a strict limitation neither for embedded SQL statements nor for the Scala programs they are embedded into, if these statement leading SQL keywords will only be declared as Scala keywords in uppercase letters. All other SQL keywords can be treated as regular identifier tokens.
If there are programs left, that use identifiers **SELECT**, **WITH**, **INSERT** ... a backtracking strategy can avoid this problem: The pre-processor tries to parse the corresponding statement if it finds one of the leading SQL keywords. If the parsing fails, the pre-processor falls back to the initial keyword and leaves the tokens to the Scala compiler. But this requires a lot of fine-tuning to avoid less user friendly error messages.

The end of a statement is well-defined using the extended grammar of the statements initialized by one of the keywords above or explicit by a semicolon. The semicolon as delimiter is optional but necessary in some special cases where a Scala value is used, which has got the name of a token that is optionally expected by the pre-processor parser for embedded SQL:

```
val stm = SELECT someColumn
          FROM someTable
          WHERE otherColumn = { someValue };
ORDER.markAsDelivered
```

Listing 8. Required delimiter ';'

The semicolon is required in this case, because the parser for embedded SQL would expect an order by clause reading the **ORDER** token, which is optional at the end of a select statement.
In the majority of cases the delimiter can be inferred and is obsolete. This fact allows the mix of embedded statements and regular Scala code in the same line in the style of XML processing in Scala, which is important to define useful combinators for result processing.

5. Result processing

The result processing is the most relevant part for the user, the programmer. A type inference based approach should provide a comfortable result processing, which handles the casting and let users process the result using generic containers.
All supported embedded statements can be merged to a generic super class, which declares an unimplemented method to execute the query for a given JDBC connection and to return the result. Implementations of this method shall also release the resources of the (JDBC) statement and connection after completed result processing.

```
abstract class Statement[T] {
  def execute(conn : Connection) : T
  def >>(conn : Connection) = execute(conn)
}
```

Listing 9. Generic super class for statements

The operator >> is a shortcut for `execute`, a combinator. It is also imaginable to overload >> with a definition, which does not require a connection. That definition of >> uses a `ThreadLocal[ConnectionPool]` to get a JDBC connection and executes the query for this connection, which is releasd after the result processing. The method should of course only be used in a context of *worker threads* with an associated connection pool.

The type of `T` depends of course on the kind of the statement, whether it is a modifying statement or a selecting query. In the case of selecting queries `T` also depends on the projected columns and the binding of columns with values in search coniditions.

Modifying statements (initialized by **INSERT**, **UPDATE**, **DELETE** or **TRUNCATE**) can be compiled as an `UpdateStatement`, which extends either `Statement[Unit]` or better `Statement[Int]` to return the number of rows affected by the statement as result.

Queries initialized by **SELECT** or **WITH** return in general bags of tuples and can be compiled as bag statements (now extending the super trait) with a special operator to process the result tuples.

```
class BagStatement[T](..., p : Projection[T])
      extends Statement[ResultIterator[T]] {
  ...
  def >>>(f : PartialFunction[T,Unit]) {
    val result = >>
    for (row <- result if f.isDefinedAt(row)) f(row)
  }
  ...
}
```

Listing 10. Bag statement

The operator >>> executes the query using the connection pool from >> and processes the result by applying each row to the given (partial) action. Using that operator results of bag statements can be processed quite comfortable:

```
SELECT id, name FROM user >>> {
  case (id,name) => handleUser(id, name)
}
```

In some cases a more concise result type of a query can be inferred.
If only aggregation columns like `MAX(id)`, `AVG(someNumber)` are projected and aggregated rows are not grouped using a **GROUP BY** clause, exactly one row is returned. Such queries can be mapped as `ValueStatement[T]` which directly extends `Statement[T]`.

```
class ValueStatement[T](..., p : Projection[T])
      extends Statement[T] {

  ...
  def >>>(f : PartialFunction[T,Unit]) {
    val result = >>
    if (f.isDefinedAt(result)) {
```

```
      f(result)
    }
  }
  ...
}
```

Listing 11. Value statement

This class also defines an operator >>> to apply a partial action on the result row. The RDBMS is usually able to infer if a (sub-)query returns at most one row. This is even necessary if an expression is compared with a scalar subquery. This occurs if all columns of the primary key or any unique key of a relation are bound by values. Such queries could be compiled as a `RowStatement[T]` which extends `Statement[Option[T]]`. I describe in Section 6 why this is not possible using a simple context (schema) independent re-writing and requires a workaround to give the type inference a hint to infer a `RowStatement` instead of a `BagStatement`.

The mapping of a stored procedure call (initialized by the keyword **CALL**) depends of course on the stored procedure itself. The routine bodies of stored procedures allow different statements, input and output parameter and often very vendor specific code. Their ideal re-writing is very complicated and will fill another paper.
An appropriate handling for now would be to re-write stored procedure calls to objects of the type `java.sql.CallableStatement`, which is the corresponding JDBC type. Just one example for the beauty of Scala:

```
val stm = CALL someProcedure({ someInParam }, @someOutParam)
```

This statement can be re-written in a way that the type of `stm` is `ValueStatement[{def someOutParam : A}]` if `A` is the output parameter type mapped to Scala.

6. Re-writing of statements

The implementation of the embedded statements consists of a pre-processor, a Scala library to whose classes the pre-processor re-writes the statements and a schema compiler to convert database schemas in corresponding representations of the library. The only modification of the Scala compiler is the application of the pre-processor at first. The library contains Scala representations of the SQL statement parts and table descriptions and makes use of Scala's type inference system to calculate result set types and check for correct usage of columns and functions.

An (accurate) database schema in the form of a script in database description language (DDL, a part of SQL) is compiled by an additional compiler `scalasql`. The script includes the creation statements of all usable tables, views and stored procedures (and the views they depend on), which shall be usable in the embedded statements. They are compiled into classes which use the library.

```
CREATE TABLE user(
  id INTEGER NOT NULL,
  name VARCHAR(50) NOT NULL,
  ...
  PRIMARY KEY (id)
)
```

The create statement for table `user` can be compiled to a corresponding Scala class of the following form:

```
class user(qualifier:String) extends Table(qualifier,"user") {
  @PrimaryKey
  def id    = new IntColumn(qualifier,"id")
  def login = new StringColumn(qualifier,"login")
  ...
}
```

Listing 12. Compiled create table statement

The table is compiled to a subclass of `Table` which is part of the library and takes a qualifier string to create a qualified representation of the table in a query.
Columns of the table are compiled in corresponding classes like `IntColumn` or `StringColumn`. The column representations extend the class `Column(qualifier : String, name : String)` and mixin the trait of their SQL expression type, `IntExpr` for `IntColumn`:

```
class IntColumn(qualifier : String, name : String)
              extends Column(qualifier, name) with IntExpr

trait IntExpr extends Expr[Int] {
 def <(exp:IntExpr)  = new BinOp(this,"<",exp) with BoolExpr
 def >(exp:IntExpr)  = new BinOp(this,">",exp) with BoolExpr
 def <=(exp:IntExpr) = new BinOp(this,"<=",exp) with BoolExpr
 def >=(exp:IntExpr) = new BinOp(this,">=",exp) with BoolExpr
 def +(exp:IntExpr)  = new BinOp(this,"+",exp) with IntExpr
  ...
 def /(exp:IntExpr)  = new BinOp(this,"/",exp) with FloatExpr
 def ===(exp:IntExpr)= new BinOp(this,"=",exp) with BoolExpr
 def <>(exp:IntExpr) = new BinOp(this,"<>",exp) with BoolExpr
  ...
 def extract = i => rs => rs.getInt(i)
 def AS(alias:String) = new AliasIntColumn(this, alias)
}
```

Listing 13. Integer column and expression

Although these column representations define operators like + (numeric representations) or **AND** (`BoolExpr`) they do not perform any numerical or boolean calculations. But due to these classes the compiler may check if SQL operands are used correctly and infer the result type (if the expression is projected in the **SELECT** clause).
The pre-processor may now rewrite search conditions in where clauses like

```
WHERE (someIntCol + 42) > 50 AND someBoolCol)
```

nearly directly to

```
where = ((someIntCol + 42) > 50) AND someBoolCol
```

(using an implicit definition which converts an integer `i` to **new** `Constant(i.toString)` **with** `IntExpr`). Type-checking is now done by the regular Scala compiler which complains if the expression of a where clause representation does not

collapse to a `BoolExpr` or an operator is applied to wrong expression types like `someIntCol + someBoolCol`. The standard SQL functions are defined in a similar way as members of the `functions` object in the library:

```
object functions {
 def SUBSTRING(c1 : StringExpr, c2 : StringExpr =
   new SQLFunc("CONCAT",c1,c2) with StringExpr
 def SUBSTRING(c1 : StringExpr, c2 : IntExpr =
   new SQLFunc("SUBSTRING",c1,c2) with StringExpr
 def SUBSTRING(c1 : StringExpr, c2 : IntExpr, c3 : IntExpr) =
   new SQLFunc("SUBSTRING",c1,c2,c3) with StringExpr
...
 def COUNT(c : Expr[_]) = new SQLFunc("COUNT",c) with IntExpr
 def MAX(c : IntExpr)   = new SQLFunc("MAX",c) with IntExpr
 def MAX(c : StringExpr)= new SQLFunc("MAX",c) with StringExpr
...
 def MIN(c : IntExpr)    = new SQLFunc("MIN",c) with IntExpr
...
}
```

Listing 14. Representations of standard SQL functions

The super trait `Expr[T]` extends `Projection1[T]`. I am re-using (with some minor changes) the projections management of Stefan Zeiger's great type-safe DSL[1] for SQL in Scala. The classes `ProjectionN[T1,...,TN]` extending their base class `Projection[(T1,...,TN)]` are responsible to extract the row values with their corresponding types out of the current row of a JDBC result set. An operator `~` is defined for these classes to compose SQL expressions to projections. The pre-processor is now able to re-write a select clause like

```
SELECT 'User ' || login, id * 2, SUBSTRING(login, 1, 1)
```

nearly directly to[2]

```
select = "User " || login ~ id * 2 ~ SUBSTRING(login, 1, 1)
```

The operator is also used to define SQL's `*` operator to select all fields of a relation:

```
class user(qualifier:String) extends Table(qualifier,"user") {
  @PrimaryKey
  def id    = new IntColumn(qualifier,"id")
  def login = new StringColumn(qualifier,"login")
  ...
  def *     = id ~ login ~ ...
  ...
}
```

Listing 15. Table representation including * operator

The compiler calculates `Projection3[String,Int,String]` to be the type of `select`, which is now a function to extract a tuple `(String,Int,String)` from a JDBC result set. Apart from the usage of tuples, operators and higher-order everything until now could be done also with Java. Calculating the type of `select` does not require Scala's enhanced type inference system. This applies when a whole query is re-written.

[1] Stefan Zeiger's DSL for SQL in Scala is available at his Blog via http://szeiger.de/blog/2008/12/21/a-type-safe-database-query-dsl-for-scala/

[2] The `||` operator for string concatenation is actually not a part of the SQL standard but supported by several vendors. I am discussing vendors specific notations later in Section 8. To support this comfortable operator on one hand and vendors which do not support it on the other hand, the operator is defined as a shortcut for the function `CONCAT`, which is part of the standard.

```
val stm = SELECT u.id, 'Mr. ' || login
          FROM user u
          WHERE u.lastLogin > { yesterday }
```

Listing 16. Example query

First of all the query is transformed to a query expression (an **object** with a fresh identifier), which encodes all parts of the query and the referenced tables:

```
object stm$query extends QueryExpression {
  val u = new user("u")
  import u._
  def select = u.id ~ 'Mr. ' || login
  def from = u
  override def where = u.lastLogin < c(yesterday)
}
```

The query expression object contains a value `u` for the table `user` because it has been imported qualified using **`FROM`** `user u`. To allow the unqualified usage (like `login` in this example) of non-ambigous columns all members of `u` are imported. The trait `QueryExpression` requires the definition of `select` and **`from`** (projection and table expression) and allows re-implementing of optional parts like `where`, `having`, `groupBy`, `orderBy`. The Scala expression `yesterday` from the surrounding scope is converted to a SQL expression representation using `c`[3], which is also the implicit converter for constants.

The trait `QueryExpression` represents the skeleton of a query initialized by **`SELECT`**. It requires the implementation of a generic projection `select : Projection[_]`, but the trait as a whole is not generic - for good reason.

The Scala compiler is able to infer the actual type τ of a type parameter A for the instance of a generic class using unification if a constructor parameter of type τ is given, where A is expected:

```
final case class Some[+A](x: A) extends Option[A] { ... }

val r = Some[Int](1) // Type of r is obviously Option[Int]
val s = Some(1)      // Type of s is inferred to Option[Int]
```

The compiler unifies the constructor parameter `x : A` with given argument `1 : Int` and concludes `A` to be `Int`.

Unfortunately this does not work for the query expression. One could try to declare generic `QueryExpression`:

```
trait QueryExpression[T] extends SQL {
  def select : Projection[T]
  def where : BoolExpr = null
  def groupBy : List[Expr[_]] = List()
 ...
}
object stm$query extends QueryExpression {
  val u = new user("u")
  import u._
  def select = u.id ~ 'Mr. ' || login
  def from = u
  override def where = u.lastLogin < c(yesterday)
}
```

Listing 17. Attempt for generic query expression

[3] One may realize that the curly braces are optional as the pre-processor is not able to differ between the unqualified usage of a column or a Scala identifier of the surrounding scope. But this works only for simple identifiers and may not work for complex expressions.

The compiler *just* needs to unify `query : Projection[T]` with `u.id ~ 'Mr. ' || login : Projection3[Int,String]` ($\hat{=}$ `Projection[(Int,String)]`). However the compiler is only able to infer the type for `T` based on a constructor argument[4]. On would have to define `T` to be `(Int,String)` on his own. This fact requires a small workaround, which is admittedly not that worse, because the compiled code will not be presented to the user.
For the value of `stm` in **Listing 16** a bag statement is compiled which takes the query expression as whole along with the projection `select`:

```
val stm = new BagStatement(stm$query, stm$query.select)

// Definition of BagStatement using Scala's enhanced
// type inference system:
class BagStatement[T](q : QueryExpression, p : Projection[T])
      extends Statement[ResultIterator[T]] {

  def execute(connection : java.sql.Connection) = {
    val stm = connection.createStatement
    // the query expression as whole is required to serialize
    // the query to a SQL string:
    val rs = stm.executeQuery(q.toString)
    new ResultSetIterator[T](rs) {
      // The projection is required to infer the type of T:
      def next = p(rs)
    }
  }
  ...
}
```

Listing 18. Compiled bag statement

This works apparently only if a query expression and its projection is applied to the bag statement. But all these statement related classes are not intended to be used by programmers directly.

6.1 Data classes

Pattern matching is great, but matching `Tuple22` is quite annoying. Apart from that, joins over several tables with many columns would exceed `Tuple22` soon. In addition to that, tuples are by nature immutable. It is often required to transfer an entity to a client (a Scala remote actor for example), keep and modify it in memory and serialize it again to the database in the end.
The projection on single columns is mapped best using tuples, but entity classes are much better if the $*$ projection of SQL is used.
`scalasql` can be modified to generate a mutable class per table, too:

```
package entities {
 ...
  class user(val id : Int,
             var _login : String,
             ...
             var _lastLogin : Date) extends Entity[Int] {
    def getPrimaryKey = id
    def login = _login
    def login_=(login : String) {
      this._login = login; entityChanges += "login"
    }
    ...
    def lastLogin = _lastLogin
    def lastLogin_=(version : Date) {
      this._version = version; entityChanges += "lastLogin"
    }
}
```

Listing 19. Class generated from DDL script

Using such classes a $*$ projection will always return an object instead of a tuple.

```
val stm = SELECT u.*,e.*,et.* FROM user u // or SELECT * ...
          NATURAL JOIN email e
          NATURAL JOIN email_type et
// stm : BagStatement[(user,email,email_type)] instead of
// stm : BagStatement[(Int,String,...,Type22)]
```

Listing 20. Class generated from DDL script

A companion object for the table representation `user` can also be generated to support a `find` method for the primary key, methods `findByKey` for each unique key Key and methods to save changes, tracked by the entity on its own using the trait `Entity`. The trait contains a method `getChanges` to collect the current values of all changed fields (stored in `entityChanges`) using reflection.

```
class user(qualifier:String) extends Table(qualifier,"user") {
  @PrimaryKey
  def id    = new IntColumn(qualifier,"id")
  def login = new StringColumn(qualifier,"login")
  ...
  def * : Projection1[entities.user] = new Projection1[
       entities.user] {
    def apply(i : Int, rs : java.sql.ResultSet) = {
      val ((_id,_login,...),i1) = (id ~ login ~ ...)(i,rs)
      (new entities.user(_id,_login,...),i1)
    }
    def toSql = qualified("*")
  }
}

object users {
  def find(_id : Int) = {
    SELECT * FROM user WHERE id = { _id }
  }
  def findLogin(_login : String, _created : Date) = {
    SELECT * FROM user WHERE login = { _login }
                       AND created = { _created }
  }
  def saveChanges(user : user, conn : Connection) {
    saveChanges(user.getChanges, conn)
    user.clearChanges
  }
  def saveChanges(changes:ChangeMap[Int], conn:Connection) {
    /* saves changes using dynamically created statement */
    ...
  }
}
```

Listing 21. Adjusted table representation and its companion object

This approach leaves two questions unanswered:

- How is it possible to define additional methods or members for the generated classes?
- `user` is not a nice Scala class name, `USERS` for an imaginable table even worse. How is it possible to declare `User` to be the class name?

[4] To be correct: the Scala compiler is able to infer the actual type of a type parameter A using a function argument, but only if A is a type variable of that function, not a type variable of the surrounding class.

Because the * projection in table objects is generated from the DDL script and the generation requires name and signature of the entity class, the code to specify an individual class name and additional definitions has to be compiled into the same module. The `scalasql` compiler can be adjusted to take in addition to the DDL script a special module source file with extension `.scalasql` as second argument.
These modules may contain **`data`** classes, class stubs for the entity classes and are translated to regular Scala source files.

```
@Table("user")
data class User {

  def idAndLogin = (id, login)

  def customMethod {
    ...
  }
}
```

Listing 22. Data class declaration

Such data classes may not have constructors, which are added by the compiler. In return the custom definitions may use the values generated from the table columns. The optional annotation `@Table` if a class name different from the table name is specified. This annotation looks similar to the annotation from JPA and Hibernate. But this is a compiler annotation, not a run-time annotation.

```
package entities {
 ...
  class User(val id : Int,
             var _login : String,
             ...
             var _lastLogin : Date) extends Entity[Int] {
    def getPrimaryKey = id
    def login = _login
    def login_=(login : String) {
      this._login = login; entityChanges += "login"
    }
     ...
    def idAndName = (id, name)

    def customMethod {
     ...
    }
}
```

Listing 23. Compiled data class

6.2 Complex statements

The basic framework for embedded statements is established. Time to try to re-write more complex statements. A simple cross product is possible with an additonal class `TableReferenceList`:

```
val stm = SELECT u1.login, u2.login
          FROM user u1, user u2 WHERE u1.id = u2.admin

// is re-written to

object stm$query extends QueryExpression {
  val u1 = new user("u1")
  val u2 = new user("u2")
  import u1._
  import u2._
  def select = u1.login ~ u2.login
  def from = new TableReferenceList(u1, u2)
  override def where = u1.id === u2.admin
}

val stm = new BagStatement(stm$query, stm$query.select)
```

Listing 24. Cross product

A regular join may be supported using methods **`JOIN`**, `NATURAL_JOIN`, `LEFT_JOIN`, ... in the trait `TableReference` which is the base trait for table representations:

```
val stm = SELECT u1.login, u2.login
          FROM user u1 JOIN user u2 ON (u1.id = u2.admin)

// is re-written to

object stm$query extends QueryExpression {
  val u1 = new user("u1")
  val u2 = new user("u2")
  import u1._
  import u2._
  def select = u1.login ~ u2.login
  def from = u1 JOIN (u2,ON(u1.id === u2.id))
}

val stm = new BagStatement(stm$query, stm$query.select)
```

Listing 25. Join

A join with a `USING` clause is complicated because of the (intended) ambiguity of its argument(s) and needs to be re-written with a workaround.

```
val stm = SELECT * FROM user JOIN config USING(id)

// is re-written to

object stm$query extends QueryExpression {
 ...
 def from = user JOIN (config,
                       USING(ON(user.id === config.id), "id"))
}

val stm = new BagStatement(stm$query, stm$query.select)
```

Listing 26. USING join specification

The compiled `USING` join specification contains a generated **`ON`** specification, which allows the compiler to check the join condition for correct type (`id` columns in both relations available, `===` applicable to them and a `BoolExpr` as result). This example also shows a problem arising from the * operator. Both relations contain the * operator to select all fields to extract the corresponding class. For that reason * is ambigous in the scope of the `select` definition and cannot be used directly. The pre-processor has to replace * with `user.*,config.*`. This is not equivalent to the original query because SQL would omit all second columns of the `USING` join specification using the * projection (`id` would occur once as column in the result set). However it allows to return classes instead of tuples which may be more concise. The same *problem* occurs with natural joins which are joins with a `USING` join specification over all columns with same name.

Set operations **`UNION`**, **`INTERSECT`** and **`EXCEPT`** can be supported with additional methods for bag statements.

```
val stm = SELECT * FROM user WHERE id < 5
            UNION
          SELECT * FROM user WHERE id > 10
```

```
// is re-written to

object stm$query$1 extends QueryExpression {
  val user = new user("")
  import user._
  def select = *
  def from = user
  override def where = id > 10
}
val stm$1 = new BagStatement(stm$query$1,stm$query$1.select)

object stm$query extends QueryExpression {
  val user = new user("")
  import user._
  def select = *
  def from = user
  override def where = id < 5
}
val stm =
      new BagStatement(stm$query,stm$query.select) UNION stm$1
```

Listing 27. Set operations

Compiling the union of two statements the pre-processor generates a usual bag statement for the second operand and uses the method **UNION** on the first operand. This makes it possible to use an already defined statement as the second operand for a set operation:

```
val stm2 = SELECT * FROM user EXCEPT { stm }
```

If the select clause of an embedded query contains at least one aggregation function like COUNT, SUM, AVG, ... the pre-processor needs to check whether all expressions in the select clause are either aggregation functions or use only columns which occur in the optional **GROUP BY** clause. If the pre-processor could validate the query successfully in this regard, the aggregation functions can be treated like regular SQL functions as already shown in **Listing 14**. If there is no grouping clause, the pre-processor can use ValueStatement instead of BagStatement.

Search conditions with IN clauses are supported using a method IN defined for the expression representations:

```
trait IntExpr extends Expr[Int] {
 ...
  def IN(stm : BagStatement[T])    = new INStm(this,false,stm)
  def NOT_IN(stm : BagStatement[T])= new INtm(this,true, stm)
  def IN(c : Collection[T])        = new INColl(this,false,c)
  def NOT_IN(c : Collection[T])    = new INColl(this,true,c)
}
```

Listing 28. IN clauses

This allows the usage of IN in search conditions in three ways:

- **SELECT** * **FROM** user **WHERE** id IN (**SELECT** id ...)
- **SELECT** * **FROM** user **WHERE** id IN { someStm }
  (if someStm : BagStatement[Int])
- **SELECT** * **FROM** user **WHERE** id IN { someColl }
  (if someColl : Collection[Int])

The function **ALL** can be supported in a similar way, but **ALL** can be treated as a regular SQL function, which converts a bag statement or collection to the corresponding expression type.

```
object functions {
...
 def ALL(q:BagStatement[Int])  = new ALLStm(q) with IntExpr
 def ALL(q:BagStatement[Float])= new ALLStm(q) with FloatExpr
  ...
 def ALL(c:Collection[Int])    = new ALLColl(c) with IntExpr
 def ALL(c:Collection[Float])  = new ALLColl(c) with FloatExpr
  ...
}
```

This allows the usage of **ALL** in search conditions in the same three ways:

- **SELECT** * **FROM** user **WHERE** lastLogin <> **ALL** (**SELECT** ...)
- **SELECT** * **FROM** user **WHERE** lastLogin <> **ALL** { someStm }
  (if someStm : BagStatement[Date])
- **SELECT** * **FROM** user **WHERE** lastLogin <> **ALL** { someColl }
  (if someColl : Collection[Date])

6.3 Modifying statements and recursion

I have mostly covered select statements only so far. Modifying statements initialized by **UPDATE**, **INSERT**, **DELETE** and **TRUNCATE** are comparatively simple. **TRUNCATE** requires just a given table representation. **DELETE** an additonal where clause, which has to be a valid BoolExpr similar to where clauses in select statements.
To support **UPDATE** and **INSERT** statements, the column representations are extended by assignment methods.

```
class IntColumn(qualifier : String, name : String)
              extends Column(qualifier, name) with IntExpr {
  def :=(that : IntExpr) = SetClause(this, that)
}
```

An update statement can be re-written nearly directly by the pre-processor with a list of set clauses and an optional where clause. The same works with insert statements if a column list is given.

```
val stm = UPDATE user
            SET lastLogin = { new Date }, logins = logins + 1
          WHERE id = { someId }

// is re-written to:

val stm = new UpdateStatement {
  val user = new user("")
  import user._
  def table = user
  def setClauses = {
    lastLogin := c(new Date) :: logins := login + 1 :: Nil
  }
  def where = id === c(someId)
}

val stm2 = INSERT INTO user (id, login)
                   VALUES ({ newId }, { newLogin })

// is re-written to:

val stm2 = new InsertStatement {
  val user = new user("")
  import user._
  def table = user
  def setClauses = {
    id := c(newId) :: login := c(newLogin) :: Nil
  }
}
```

To support insert statements without a column list or a sub query the companion object of the table representation (see **Listing 21**) can be extended:

```
object users {
 ...
  def INSERT(_id : Int, _login : String, ...) = {
    INSERT INTO users (id, login, ...)
                VALUES ({ _id }, { _login }, ...)
  }
  def INSERT(stm : BagStatement[(Int,String,...)]) = {
    /* dynamically generated insert statement */
  }
}
```

Recursive statements using the `WITH` keyword are actually just special select statements. They have to be re-written in a way that the Scala type inference system is able to calculate the correct type of the recursive union.
To compile the statement in **Listing 4** the pre-processor uses the first sub query of the union for `invitationTree` as a representative expression for the type of `invitationTree` (named `stm$invitationTree$init`), which is used instead of `invitationTree` every time a column of the recursive relation is referenced (`invitor`, `invited` in the recursion and in the final query). The compiled code is actually huge, ugly and can be improved in several ways but it works for now.

```
object stm$invitationTree$init$query extends QueryExpression {
  val r = user("r")
  val s = user("s")
  import r._
  import s._
  def invitor = r.id
  def invited = s.id
  def select = r.id ~ s.id
  def from = r JOIN (s, ON(r.id === s.id))
}
val stm$invitationTree$init =
  new BagStatement[](stm$invitationTree$init$query,
                     stm$invitationTree$init$query.p)

object stm$invitationTree$rec$query extends QueryExpression {
  val t = stm$invitationTree$init$query
  val u = user("s")
  import t._
  import u._
  def select = t.invitor ~ u.id
  def from = new TableReferenceList(t, u)
  override def where = t.invited === u.invitor
}
val stm$invitationTree$rec =
  new BagStatement[](stm$invitationTree$1$query,
                     stm$invitationTree$1$query.p)
  UNION stm$invitationTree$init


object stm$query extends QueryExpression {
  val invitationTree = stm$invitationTree$init$query
  import invitationTree._
  def select = invitor ~ COUNT(invited)
  def from = stm$invitationTree$rec
  override def groupBy = List(invitor)
  override def having = invitor === 42
}
val stm = new BagStatement(stm$qery, stm$qery.p)
```

Listing 29. Compiled recursive query

6.4 Bound primary keys and scalar sub queries

Assume `id` to be the single column in the primary key of tables `user` and `config`. The result of the following query contains either exactly one row (if a user with id 1 exists) or no rows (if no such user exists):

```
SELECT * FROM user JOIN config USING(id) WHERE id = 1
```

Such a statement could be in theory compiled as a `RowStatement` (introduced in Section 5). However the pre-processor has got no knowledge about the schema (or its corresponding representation) and for that reason cannot decide if a unique key is totally bound just because a column named `id` is bound to a constant. This fact prevents users from using scalar sub queries, which project only one column of type τ, return at most one row and can be used as a regular expression of type τ. A workaround is required to hint the pre-processor that at most one row is returned. This can be achieved using a clause `LIMIT 1`. The keyword `LIMIT` is actually not a part of the SQL standard, but some vendors support it and every vendor supports an equivalent notation to limit the numbers of rows in the result set. A re-writing to the correct notation can be done at run-time (see Section 8). The pre-processor can now compile a row statement instead of a bag statement and allow an implicit conversions from `RowStatement[Int]` to `IntExpr`, `RowStatement[String]` to `StringExpr` et cetera.

7. Compiling nullable columns

Null values in relational database [8] are a topic that is dealt with quite controversial in sience and industry. The database description language included in SQL allows to define columns as `NULL` or `NOT NULL`.
The easiest approach would be to map columns to the Java primitive wrappers (`java.lang.Integer`, `java.lang.Float`, ...) instead of Scala values. Since the primitive wrappers are classes and their references can be set to the `null` reference it is possible to return Scala's null pointer for SQL's `NULL` values.
This would be a error-prone handling from the database designer's point of view and quite a breach with Scala's differentiation between `AnyVal` and `AnyRef` [6].
In a much better approach a type `Nullable` similar to `Option` is defined:

```
abstract class Nullable[T]
object Null extends Nullable[Nothing]
case class NotNull[T](value : T) extends Nullable[T]
```

Now the compiler for ddl scripts can compile nullable columns using this class. The pattern matching against `Nullable` is annoying if a projected column in a query cannot be `NULL`. For that reason the compiler needs to infer if a generally nullable column is definitely not `Null`, which occurs if the column is bound or compared to another value/column in a search/join condition, checked using `IS NOT NULL` or is a function call argument.

The opposite of specializing nullable columns to not nullable columns occurs at outer joins. These joins return each row of both relations (or the left/right relation in case of

left/right joins) at least once even if there is no corresponding row on the opposite relation. In this case all projected columns from the other relation are set to `NULL`. For that reason all projected columns have to be set as nullable.
A special case is the usage of the `*` operator in outer joins, which would return a data class in a regular inner join. In an outer join this data class can be treated as optional using `Option`.

```
val stm = SELECT u.id, e.address
          FROM user u LEFT JOIN email e ON u.id = e.user

// is re-written to:

object stm$query extends QueryExpression {
  val u = new user("u")
  val e = new email("e")
  import u._
  import e._
  def select = u.id ~ e.address.nullable
  def from = u LEFT_JOIN (e, ON(u.id === e.user))
}

val stm = new BagStatement(stm$query, stm$query.select)
//  stm : BagStatement[Int,Nullable[String]]

val stm2 = SELECT u.*, e.*
           FROM user u LEFT JOIN email e ON u.id = e.user

// is re-written to:

object stm2$query extends QueryExpression {
  val u = new user("u")
  val e = new email("e")
  import u._
  import e._
  def select = u.* ~ e.*.optional
  def from = u LEFT_JOIN (e, ON(u.id === e.user))
}

val stm2 = new BagStatement(stm$query, stm$query.select)
//  stm2 : BagStatement[User,Option[EMail]]
```

Listing 30. Outer join mappings

The projection `*.optional` uses the compiled annotation `@PrimaryKey` (as shown in **Listing 12**) and reflection to decide at run-time whether the regular projection should be used (wrapped with `Some`) or `None` should be returned instead (if and only if the primary key fields are null).

8. Related and future work

I am looking forward to continue this work in my master thesis by providing an implementation of the pre-processor and analyzing, whether more concise mappings are realizable using this approach with a simple re-writing and a libary.

8.1 Towards object-relational mapping

As already said it was not my intention to create another tool for object-relational mapping because of the quantity and quality of existing ORM tools. The Lift Framework is the leading web framework written in Scala with an own persistence framework. Although the Lift persistence supports type-safe queries, they are more focused on querying objects (records) than on bulk queries using tuples like my approach for embedded SQL statements. I think both approaches would complement one another if the `*` projection for embedded statements would return Lift's entities instead of the data classes introduced in Section 6.1.
Then one may use a projection of explicit columns without the `*` operator to display a fast list using tuples or the `*` projection if Lift entities with all the CRUD comfort like validation, form generation is preferred.
If the Lift community would like that, I would spend some time to analyze how the embedded statements could be re-written in a way to use Lift records as database schema representation.

8.2 Vendor specific notations and functions

Who ever tried to migrate a complex statement from one relational database management system to another knows, what the term *standard* means in the context of SQL. A practically usable implementation has to care about vendor specific notations, limitations and functions. But in most cases it is possible to specify a translation from the dialect of one vendor to another.

For now all SQL representation classes and traits in the library extend the trait `SQL { def toSql : String }` to serialize the representation to a SQL string. The trait can be adjusted by adding the JDBC connection as a parameter for `toSql` for a vendor dependent serialization at run-time.

```
class Column(q : String, n : String) extends SQL {
  def toSql(conn : Connection) = {
    conn.getClass.getPackage.getName match {
      case "org.postgresql" => "\"" + q + "\".\"" + n + "\""
      case "com.mysql.jdbc" => "`" + q + "`.`" + n + "`"
      case otherVendor      => q + "." +
    }
  }
}
```

A. Examples

A.1 Statements

```
val stm1 = SELECT * FROM user
// stm1 : BagStatement[User]

val stm2 = SELECT name FROM user
// stm2 : BagStatement[String]

val stm3 = SELECT name, lastLogin
           FROM user
// stm3 : BagStatement[(String,Date)]

val stm4 = UPDATE user
           SET name = { someName }
           WHERE id = { someId }
// stm4 : UpdateStatement

val stm5 = SELECT * FROM user WHERE id = { someId }
// stm5 : BagStatement[User]

val stm5 = SELECT * FROM user WHERE id = { someId } LIMIT 1
// stm5 : RowStatement[User]

val stm6 = SELECT MAX(number),AVG(number)
           FROM numbers
// stm6 : ValueStatement[(Nullable[Int],Nullable[Float])]

val stm7 = SELECT MAX(number),AVG(number)
           FROM numbers
           GROUP BY ROUND(number/100)
// stm7 : BagStatement[(Int,Float)]

val stm8 = SELECT * FROM user u, email e
           WHERE u.id = e.user
// stm8 : BagStatement[User,EMail]

val stm9 = SELECT * FROM user u LEFT JOIN email e
           ON u.id = e.user
// stm9 : BagStatement[User,Option[EMail]]

val stm10 = SELECT * FROM user u FULL OUTER JOIN user i
            ON u.id = i.invitor
            WHERE u.id = e.user
// stm10 : BagStatement[Option[User],Option[User]]
```

A.2 Result processing

```
val max = SELECT MAX(lastLogin) FROM user >> connection
// max : Nullable[Date]

val min = SELECT MIN(lastLogin) FROM user >>

val optUser = SELECT * FROM user WHERE id = { someId } >>
// optUser : Option[User]

SELECT id, name FROM user >>> {
  case (id, name) => ...
}

SELECT id, name, someNullableColumn FROM user >>> {
  case (id, name, Null)           => ...
  case (id, name, NotNull(value)) => ...
}
```

References

[1] Jim Melton, Alan R. Simon, and Jim Gray. *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2001.

[2] Derek Chen-Becker, Marius Danciu, and Tyler Weir. *The Definitive Guide to Lift*. Apress, 2009.

[3] J. Melton and A. Eisenberg. *Understanding SQL and Java Together*. Morgan Kaufmann, 2000.

[4] Bill Burke and Richard Monson-Hafel. *Enterprise JavaBean 3.0*. O'Reilly, 2006.

[5] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications, 2006.

[6] Martin Odersky et.al. An overview of the scala programming language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[7] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*, chapter Appendix G. Morgan Kaufmann, 1992.

[8] H.-J. Klein. Null values in relational databases and sure information answers. In K.-D. Schewe L. Bertossi, G. Katona and B. Thalheim, editors, *Semantics in Databases, LNCS 2582*, pages 102–121. Springer, 2002.