Master Thesis

An Embedded Query Language in Scala

Amir Shaikhha

Professor	Martin Odersky
Supervisors	Vojin Jovanovic, Eugene Burmako
Expert	Stefan Zeiger, Typesafe
Semester	Spring 2013

Abstract

In this thesis we address the problem of integrating general purpose programming languages with relational databases. An approach to solving this problem is using raw strings to represent SQL statements. This approach leads to run-time errors and security vulnerabilities like SQL injection. The second approach is integrating the query in a host language. The most well-known example of the second approach is LINQ. This approach provides static checking of types and syntax during compilation.

This thesis presents an embedded query language in Scala, namely *Shadow Embedding in Slick*. *Shadow Embedding* provides even stronger compile-time guarantees than LINQ and similar systems in Scala. The experimental results show that the performance of our approach is very similar to the case of using raw Strings, thanks to static code analysis and clever code caching.

Acknowledgement

First, I would like to thank Prof. Martin Odersky for giving me the opportunity to do my master thesis in Typesafe and LAMP.

Vojin Jovanovic helped me tremendously during the whole period of my thesis. Thank you to Stefan Zeiger for believing in me and providing me with the opportunity to work on the Slick project. I would like to thank Eugene Burmako for all his support, Christopher Vogt for the great discussions, and everybody at Typesafe and the LAMP team.

I can never thank my parents enough for believing in me and supporting me through all periods of my life. If it wasn't for their love and support, I would for sure be nothing.

Last but not least, I would like to thank my wife Fatemeh for all her love, help, and support. This work is dedicated to her.

Contents

List of Tables

List of Figures

1 Introduction

One of the main components of modern applications are the systems using databases to store their persistent information. The developers of these systems are mainly using object-oriented languages and relational databases. Therefore, it is important to integrate these two concepts.

Impedance mismatch described by David Maier in [27] explains why object-oriented programming languages cannot be integrated with relational databases. Relational databases represent data in a tabular format, whereas object-oriented languages use objects. Additionally, relational databases are using declarative queries, like SQL, whereas object-oriented languages are mostly imperative. William Cook et al. in [43] have discussed several *Object-Relational Mapping* (*ORM*) frameworks which are designed to facilitate bridging the mismatch between objects and relations.

In order to integrate databases with programming languages, an interface needs be defined in the program, which is responsible for executing the queries in the database. There are two interfaces to query a database:

- *Call Level Interface (CLI) [41]:* The program is separated into two conceptual paradigms in this way. One part is using object-oriented paradigm for computation, and the second one is for accessing the relational database. ODBC [41] and JDBC [20] are standard call level interfaces for C and Java programming languages respectively. This approach is highly error-prone and provides no type-safety. Even expert users can make typing errors or confuse identifier names, function parameters, and types. Moreover, there is a possibility of security vulnerabilities such as SQL injection. The main problem is that catching these errors will be postponed until run-time. Although there are tools in order to perform these checks in the compile-time, they need a very complicated infrastructure and require lots of effort. In addition, there is no way to reuse a part which is very common in different places in a type-safe manner.
- *Embedding Queries in a Host Language:* The second approach is to embed the queries in a host language. In this way, the previous errors are caught at compile time by the host language compiler. One of the most popular efforts towards this goal, is Microsoft LINQ [29] in .NET family programming languages. The queries are expressed by using language-level constructs, which makes the queries verifiable at compile-time. All the query expressions are represented as expression trees during compilation. Afterwards, this tree representation is converted into an SQL statement for the desired database engine backend in the run-time. Most of the non-homoiconic programming languages could not provide this feature, and there is a need for language extension to support it. LINQ also allows type-safe composition of the queries. Meaning that the user can define the part of query commonly used in different queries, and then use it in different places. This way, the code is more maintainable and less error-prone.

This thesis presents an approach for embedding queries in Scala programming language. Embedding queries in Scala can be achieved in two ways:

- *Purely embedding* [22] the query in the Scala language. This approach is used in Polymorphic Embedding [21] and LMS [33] to embed *Domain Specific Languages* (DSLs) in Scala. *Pure embedding* is also used in *Scala-Integrated Query* [42] and *Lifted Embedding* front-end (Section 3.6) of *Slick* [9] to embed queries in Scala.
- Directly embedding queries by retrieving tree representations of queries at compile-time

with Scala macros [13] and storing them for run-time processing. This approach is used in *Direct Embedding* front-end (Section 3.7) in Slick.

Our approach is *Shadow Embedding* in Slick to overcome the limitations of Lifted Embedding and Direct Embedding. Lifted Embedding lifts query expressions to their *deep* representation, and makes the deep representation visible to the user, whereas Direct Embedding converts query expressions to the Scala AST nodes behind the scenes (by using Scala macros [13]). As a result, the types are not standard Scala types in Lifted Embedding and can confuse the user, but they encode the operators supported by query engine (Unintuitive and complete). In Direct Embedding, the type of query expressions is standard Scala type, but converting from Scala AST nodes to deep representation is done at run-time. Therefore, unsupported operations are not caught during compilation (Intuitive and incomplete). Shadow Embedding, converts every query expression to corresponding deep representation *transparently* by Yin-Yang [24] using Scala macros. As a result, every query expression is of standard Scala type from user perspective, but behind the scenes every query expression is mapped to its deep representation during compilation. Hence, type-errors are intuitive (like Direct Embedding) and comprehensive (like Lifted Embedding) at the same time. Furthermore, by having AST of query expression, it is possible to analyse the code and perform clever code caching, which will lead to an increase in performance speed (from **1x** to **250x**, see Section 10) without any effort by the user. Also, Shadow Embedding uses the same deep representation as Lifted Embedding, and therefore, it is interoperable with Lifted Embedding.

In order to integrate query expressions in Scala, the user should define the types for each table, which requires some boilerplate. *Type provider* is an approach to providing table definitions for the user automatically by reading information about tables from database schema or from the code annotations provided by the user. As Shadow Embedding has the same deep representation as Lifted Embedding, *type provider* of Lifted Embedding is available for Shadow Embedding as well.

The thesis is structured as follows. First, we will present the related work in Section 2. In Section 3 we give an overview of Slick. Afterwards, the thesis is divided into two main parts.

In Part I, the effort towards having *type providers* in Slick is explained. First, we explain Scala Macros in Section 4. Afterwards, in Section 5 we will explain how to create an appropriate Schema Model to abstract over database entities. Then, in Section 6 we will explain the main design and implementation aspects for *type providers* in Slick. Finally, we will review the limitations of *type providers* in Section 7.

Part II discusses Shadow Embedding. In Section 8, we give an overview of the Yin-Yang system. Thereafter, in Section 9 the main design and implementation concerns about Shadow Embedding will be explained. Section 10 is dedicated to experimental results for Shadow Embedding in comparison with other front-ends of Slick and other systems. We explain the limitations of Shadow Embedding in Section 11 and future work in Section 12. Finally, Section 13 is dedicated to concluding remarks.

2 Related Work

There are several tools for performing the type checking of SQL statements at compile-time. For example, the IntelliJ plugin [2] for Hibernate [1] will report errors and do auto-completion for HQL. JDBC Checker [16] provides the same thing for JDBC, as well as the work done by Card Gould et al. [17] and SQL DOM [28]. These tools need a very complicated infrastructure and require lots of effort for development. In addition, they can generate false positives when separate compilation is used [14].

For Java, SQLJ [30] in a pre-compilation step converts the embedded queries into corresponding Java code by using a component called *SQLJ Translator*. As a result, it provides static typing. Safe Query Object [14] uses standard Java classes to represent queries and uses reflection and OpenJava [39] to convert these classes into JDO [3] queries during compilation.

HaskellDB [26] is an embedded domain-specific language in Haskell. The query expressions in this DSL are statically typed. These expressions use Haskel as meta-language. These query expressions are converted to SQL statements.

Scala-Integrated Query [42] uses LMS [33] to embed a query language in Scala. ScalaQL [36] uses the same approach as Lifted Embedding (Section 3.6) front-end of Slick [9], to embed query language in the host language.

James Cheney et al. presented T-LINQ [23], an extension to LINQ [29] that supports nested data, but does not support queries returning nested data. Ferry [18], in addition to these features, supports the queries returning nested data.

Type providers in F# 3.0 [38] are a type-bridging mechanism which allow strongly typed programming that uses external information systems. *Type providers* produce the types necessary for the program, which are fetched from an information source. The user can use these types and do the programming in a type-safe way.

Type provider in F# is a compile-time component, for which two inputs need to be provided:

- The static parameters, available at compile time, which identify an external information space.
- A way of accessing this information space.

Then this component will provide the compiler with two things:

- A programming interface to the given information space.
- An implementation of this interface.

In other words, *type provider* is an adapter component which reads external information sources with schematized data and then converts them into types. As a result, there will be no need for the user to perform any explicit transcription or code generation for types. The provided types are used not only in the type-checker and run-time, but also in the tools which rely on the type-checker, like IDE code auto-completion.

An important feature of *type providers* in F# is that type providing is done lazily. Therefore, whenever a type is requested by the compiler, the information about it will be fetched and corresponding members will be provided. As a result, the information space could be very large and only the required subset of information space will be fetched.

Type providers in Slick support all features of *type providers* in F#, except lazy *type providers*. Lazy *type providers* can not be easily implemented in the Scala programming language. It needs enough infrastructure to generate the byte code lazily, which is not currently supported by *type*

macros (See Section 4.2). However, lazy type providers can be implemented using *def macros* and by using *Dynamic* classes which is not investigated in this thesis. All other features can be implemented with *type macros*.

3 Slick

Slick provides a type-safe way for accessing databases. User can write the query expression in the Scala language, and Slick translates this query expression into an appropriate SQL statement. Slick will be creating its own AST out of the given query expression. Then, query compiler will convert it into a corresponding SQL statement. This statement is passed to the given query engine and the result is returned to the user.

3.1 Architecture

Slick is composed of three main components. The first component, namely front-end, is the component with which the user is interacting. Query expressions are input for this component, and as output this component will produce Slick AST. The next component is query compiler which is responsible for translating the Slick AST from the previous component to an appropriate SQL statement. The last component is the Invoker component which is a wrapper for SQL statements. This component executes given SQL statement and translates the result to a form which is recognizable in the client side.

Figure 1: Slick architecture

3.2 Front-end

The user is interacting with the whole system via the front-end component. There are three types of tasks which are needed to be handled by front-end.

- *Querying:* Represents selection.
- *Data Manipulation:* Represents insertion, update and deletion.
- *Data Definition:* For defining a table or dropping it.

A data-structure for representing a table and its columns is an essential component for all of these tasks. For querying, a query interface is needed to be defined in order to provide the operations which can be handled by SQL select statements. For manipulating data, an executor component is necessary to provide an interface for the database engine. Furthermore, an interface for defining and dropping a table is necessary. Finally, to restrict the domain of the elements of a table, it is necessary to have table constraints.

3.2.1 Table Data-Structure

This kind of data-structure represents each relation in relational databases. The main motivating reason for having this element is having an appropriate AST node for each specific relation. By having this data-structure in the user code, we will avoid any misspelling in relation entities or any other errors of this kind.

3.2.2 Column Data-Structure

With the same reasoning as having a data-structure for each relation, we could conclude that a data-structure is needed for each column of that relation. As a result, we could create AST nodes representing columns.

3.2.3 Constraints

By using constraints, we could restrict the domain of elements that a table could contain. We have the following constraints:

Not Null ensures that a column cannot be *null*.

Unique indicates that each record should have a unique value for a particular column.

Primary Key specifies that a table record can be identified uniquely by a single column or a combination of columns. A table can have only one primary key constraint. The column(s) with this constraint, could not be *null*. In other words, Primary Key is a combination of Not Null and Unique constraints.

Foreign Key enforces the relationship between two tables. A foreign key in one table refers to a column or a combination of columns in the other table.

Index An index over some columns of a table, provides quicker access to that table using indexed columns.

3.2.4 Query Interface

By using this interface, we can query over a table or a projection of some columns of that table. This interface contains the elements which are the result of that query. In other words, it is a *view* of the result elements. This interface should define these operations in order to provide select statement capabilities.

Map Applies the function which is passed to it, to every element, and returns a view with new elements. The `SELECT` part of the SQL statement can be expressed by this operation.

Filter Returns a view with the elements satisfying the given predicate. It corresponds to the `WHERE` part of the SQL statement.

Flatten Converts a two layer nested query view into a one layer view.

FlatMap This operation is a composition of the map and flatten operations. It applies the given function to each element, and then it flattens the result view.

SortBy It sorts the query view by the given order. This operation is corresponding to `ORDER BY` in the SQL statement.

GroupBy Divides the elements into different groups specified by given grouping criterion. `GROUP BY` is its corresponding clause in the SQL statement.

Join It should merge two query views into one query monad. This operation corresponds to `INNER JOIN`, `JOIN`, `OUTER LEFT JOIN`, `OUTER RIGHT JOIN` and `CROSS JOIN`.

Zip Creates a query view out of two views, in which each element is a tuple of corresponding elements from each view. Both views must have the same length, and the result view has the

same length as the two input views.

Union Unions the elements of two views and returns the result as a new view. This operation is mapped to the `UNION` operation in the SQL statement.

Count Returns the number of elements of the given query view. It is mapped to `COUNT` in the SQL statement.

Take Whenever this operation is applied to a query view, another view will be returned containing the first n elements of the original view, in which n is the input to this operation.

Drop Returns a query view without the last n elements of the given query view, n being the input for this operation.

As the query interface provides the implementation of `map`, `flatMap`, and `filter`, Scala for-comprehensions can be used. These expressions are translated to equivalent monadic expressions using these three operations.

3.2.5 Executor

In addition to query views, each table and associated query view requires another interface for manipulating data. This interface provides these operations:

Update Updates the elements of the view with the new value. It is translated to the `UPDATE` statements in SQL.

Insert Inserts the given element to the table. It is translated to the `INSERT` SQL statements.

Delete Deletes the given element from the table. It is translated to the `DELETE` statement.

3.3 Slick AST

Slick AST is an intermediate representation for all SQL statements. It provides a high-level abstraction over SQL statements in order to make them independent of the query engine. Slick AST nodes are used among different phases of query compilation. In Appendix A different AST nodes are explained in detail.

3.4 Query Compiler

This component is responsible for converting Slick AST into SQL statement depending on the given query engine. Like every other compiler, it consists of different phases which are divided into different categories:

- *Standard Phases:* These phases are common for all query engines, whether they are relational, or non-relational. Cleaning up the AST is the main responsibility of standard phases.
- *Relational Phases:* These phases are only specific to relational databases. Assigning type to the nodes, converting to comprehension form (a form similar to SQL comprehension), and fusing are the main phases of relational phases.
- *Statement-Specific Code Generation:* The last phase is to generate SQL statement, according to the type of statement. It has different code generators for selection, update, deletion and insertion. After this phase, the result SQL statement will be generated and wrapped by an Invoker object.

More details about each phase are given in Appendix B.

3.5 Invoker

An Invoker is a wrapper over JDBC SQL statements. Depending on the type of statement we have different invokers:

Select Invoker A wrapper over select statement. It will provide the user with two main methods `first` and `list` which return first element and all elements consecutively. This invoker should also include type mapping between a tuple and *case class*, in the case that a *case class* is representing each row of a table. Whenever the result is fetched from the database, it will be automatically converted to the *case class*.

Query Template Invoker Very similar to Select Invoker. The main difference between these two is that, the former has a fixed SQL statement, whereas the latter has a prepared statement, which contains some parameters that need to be filled later. This invoker will set these parameters to the values which are passed to it, and then query against this statement.

Insert Invoker Contains a prepared statement instead of a fixed statement. Because the value which should be inserted is not specified. Whenever `insert` method is invoked, the prepared statement is filled in with the parameters which are passed to it. It should also embed conversion from the *case class* to tuple representation in instances where a *case class* represents each row of a table. If an instance of that *case class* has been passed to the `insert` method, it should know how to deconstruct it to separate attributes.

Update Invoker It is very similar to Insert Invoker. It has the limitation that the only values which will be set are the parameterised values. These parameters will be set as soon as `update` method is invoked with the desired values.

3.6 Lifted Embedding

Lifted Embedding, the main front-end in Slick, uses an approach similar to Lightweight Modular Staging (LMS [33]). Every expression is lifted explicitly or by using implicit conversions to its *deep* representation. As a result, every expression has the deep type (usually a higher-kinded type [31]), and not the Scala standard type. The good thing about this design is that it causes the type checker to check for the existence of every operation in the deep type. As a result, if an operation is not supported by the query engine, a type error will be raised. The negative aspect of this design is that the types of expressions do not make sense to the user and error messages are not very intuitive.

Table Data-Structure There exists an abstract class `Table[T]` wherein each class that extends it, will be representing a table. The type of each record of that table should be specified by `T` type parameter. Method `*` should be implemented by user in order to show which columns represent the full projection over that table and if of type `Rep[T]`. The constructor for `Table` class is a string which represents the name for this table in the database. `Table[T]` extends `Rep[T]`

Column Data-Structure There is a *trait* called `Column[T]` which is a super-trait for each column. In order to create a column, whenever the user is defining a table by extending `Table` class, he will have a method called `column`, with one parameter, which represents the name of

this column in the database. `Column[T]` extends `Rep[T]`.

Projection Every column with type `Column[T1]`, will have a method `~` which accepts another column of type `Column[T2]`. The result of this method is projection of these two columns and is of type `Projection2[T1, T2]`. This type extends `Rep[Tuple2[T1, T2]]`. Consecutively, `Projection2` has method `~` which returns `Projection3` and so on.

Type Mapping As it was noted, `Table[T]` has a method `*` which specifies which columns are representing the full projection of type `Rep[T]`. If `T` is a `Tuple` of some types, `*` can be implemented by chaining `~` over desired columns. But, what will happen if `T` is a *case class*? Somehow, the tuple which is created from the projection of these columns needs to be translated to its corresponding *case class* and vice versa. Every `Projection` class has method `<>` which accepts as arguments `apply` and `unapply` method of companion module for that *case class*. Hence, it returns an object of type `MappedProjection[T]` which extends `Rep[T]` and encodes how to convert from tuple to *case class*, and from *case class* to tuple.

Constraints Class `Table` has an object that includes flags denoting whether a column is primary key, or whether it should be an auto-incrementing key. Also, it provides two methods `foreignkey`, and `index`, which are responsible for defining foreign key and index constraints.

Compiling queries and creating invokers are done by an implicit conversion. Whenever the user uses `first`, `list`, `insert`, `update`, and `delete` over a query, it will be implicitly converted to another type which triggers query compilation, uses the appropriate invoker for it, and calls associated method over that invoker.

3.7 Direct Embedding

Direct Embedding in Slick uses the same approach as LINQ. Every query expression is converted to its AST in the compile-time. During run-time, by using reflection API this tree representation is converted to the SQL statement which is associated with the given query engine. As a result, the user will only see the standard Scala types, and the type errors are understandable for the user. However, as the conversion of tree representation to SQL statement is done at the run-time, the errors for unsupported operations are not caught during compilation.

Table Data-Structure A *case class* which is annotated by `@table` represents a table in the database. This annotation accepts one string parameter which represents the name of this table in the database.

Column Data-Structure Every field of the *case class* which is annotated by `@column` represents a column of that table. Like `@table` it has one string parameter for its name in the database.

This front-end is experimental and lacks many features of Lifted Embedding.

3.8 Plain SQL

There are several operations which are not supported by Lifted Embedding. Plain SQL is a wrapper over JDBC and has a nicer Scala-based API. This API bypasses producing Slick AST nodes and Query Compiling. It will be translated directly to parameterised SQL statements. As a result, there is no need to define any data-structure representing tables and columns. It also provides an API which uses String interpolation to simplify creating parameterised statements.

Part I
Type Providers

4 Scala Macros

Macros are a compile-time meta-programming facility in Scala. Meta-programming means writing a program which can write another program. For example, in C++ one could define macro strings, which will be expanded to string code snippets. The main difference between macros in C++ and Scala is that in C++ the macros are plain strings and will be substituted by plain strings. Also, there is a notion of Template Meta-programming in C++[40] in which the code written in *templates* is used by compiler for generating the code. Haskel has meta-programming facility provided by Template Haskell [35].

Macros in Scala are using abstract syntax trees. Macros manipulate the AST of the program at compile-time by using the reflection API that abstracts over the compiler's internals. They have been available as experimental features in the Scala programming language since version 2.10.

Two kinds of macros are being used in the present thesis. The first type includes *def macros*, which are methods whose calls are expanded at compile-time. This kind of macros is available in the 2.10 release. The other category consists of *type macros*, which are types that are computed and expanded during compilation. This macro is available in another branch of the Scala compiler, called *Macro Paradise* [8].

4.1 Def Macros

Def macros are methods that their calls are expanded during compilation. A *def macro* is defined the same way as regular methods, except that its body will be followed by a `macro` keyword and a static reference to a method that operates on ASTs. Every call to a *def macro* will be substituted by the result of the method which was working on ASTs. In other words, the arguments to a *def macro* will be lifted to their AST representation and will be passed to the method. Then, this method does computation over an AST and as a result returns another AST. Every call to a *def macro* is substituted by the result AST.

As an example we present a *def macro* named `assert` in Figure 2. The first line of code shows that the actual implementation for this *def macro* is done in `assertImpl` method.

```
def assert(cond: Boolean, msg: String) = macro assertImpl
def assertImpl(c: Context)(cond: c.Expr[Boolean], msg: c.Expr[String]):
    c.Expr[Unit] = {
  import c.universe._
  q"if (!$cond) raise($msg)"
}
```

Figure 2: The implementation for assert

In the second line of code, `assertImpl` has a parameter of type `Context` which encodes the context in which `assert` method has been invoked. In addition, instead of two arguments of type `Boolean` and `String`, `assertImpl` has arguments of type `c.Expr[Boolean]` and `c.Expr[String]`. These two arguments represent AST of the two arguments passed to `assert` method. `q` is a quasi-quote [34] *String Interpolator*[1], which returns trees of type `c.Tree` (and its subtypes) out of the given string. As `q` is a String Interpolator, it is possible to use other variables with type

[1] A way to embed variable references directly in processed string literals [13]

`c.Expr` in it. In addition, the quasi-quote itself is a macro [34]. In this example, `assertImpl` method creates an expression which represents an `if` statement, in which the first argument is the condition for it, and it will raise an exception that contains the second argument as its message.

```
assert({
  val x = 2 * 2
  val y = 2 + 2
  x == y
}, "incorrect computation")
```

Figure 3: The call to assert before macro expansion

As a result, the code in Figure 3 will be expanded to the code shown in Figure 4

```
if(!{
  val x = 2 * 2
  val y = 2 + 2
  x == y
})
  raise("incorrect computation")
```

Figure 4: The call to assert after macro expansion

4.2 Type Macros

Type macros are types that are expanded into their underlying macro implementation during compilation. They are using the same syntax as type aliasing (`type ListInt = List[Int]`) with the difference that *type macros* can accept value arguments.

Figure 5 shows the implementation for a *type macro* which creates a connection to H2 database and provides the types for tables which exist in that database. The first line of this code shows that `h2DbImpl` is the macro implementation for *type macro* `H2Db`.

```
type H2Db(url: String) = macro h2DbImpl
def h2DbImpl(c: Context)(url: c.Expr[String]): c.Tree = {
  val Template(_, _, existingCode) = c.enclosingTemplate
  Template(..., existingCode ++ generateCode())
}
```

Figure 5: The implementation for H2Db

In the second line of Figure 5, `h2DbImpl` uses the AST representation of the module or the class which is extending it, and appends the code which is generated by method `generatedCode()` to its body. `generatedCode()` returns an object of type `List[c.Tree]`. Each element of this list is an AST representation for the code, which will be generated.

```
object Db extends H2Db("jdbc:h2:coffees.h2.db") {
  val dbId = "h2-coffee"
}
```

Figure 6: Module Db before macro expansion

For example the code shown in Figure 6 will be expanded to the code which is shown in Figure 7.

```
object Db {
  val dbId = "h2-coffee"
  // generated code
  case class Coffee(...)
  val Coffees: Table[Coffee] = ...
}
```

Figure 7: Module Db after macro expansion

In Figure 7 the generated code will be injected to the definition of the module `Db`. As a result, the user could have access to `Coffees` field and `Coffee` *case class* of this module in a type-safe manner.

The main use case for *type macros* is code generation out of an abstract model defining the data-structure in an abstract level. The schema over which we are defining the abstract model, must be available at compile time. This abstract model can be inferred either by another part of code or from information encoded in an external resource.

5 Schema Modelling

In order to transform information space into types, creating an abstract model out of information space is essential. In this case, different information spaces showing the same concept, can be mapped to the same representation. In Slick, we have to create an abstract model out of database entities. Therefore, we need an abstract model over table entities and their columns. Also, such an abstract model should encode the constrains defined over the tables.

5.1 Naming

Each table in the database, must be identified uniquely. For this purpose, a qualified name is assigned to each table, which encodes the schema and catalog of that table. Therefore, this qualified name will be the identity for each table.

Additionally, it is necessary to have an identity for each column. The qualified name for each column encodes the qualified name of its table, as well as the name of that column.

5.2 Table Data-Structure

As it was discussed in Section 3.2, it is necessary to have a data-structure for each table. The model for a table includes the qualified name for that table, the list of columns of that table (which will be presented in Section 5.3), and the list of constraints of that table (which will be presented in Section 5.4). Table model is represented using *case class* in Figure 8.

```
case class Table(name: QualifiedName, columns: List[Column],
    constraints: List[Constraint])
```

Figure 8: Table model

5.3 Column Data-Structure

This model includes the qualified name for the column, the type of this column in Scala code, and its type in the database. As the qualified name of the table of a column is implicitly encoded in the qualified name of that column, there is no need to have any pointer to its table. Column *case class* is represented in Figure 9.

```
case class Column(name: QualifiedName, tpe: Type, dbType: String)
```

Figure 9: Column model

5.4 Constraints

Constraints should not include any reference to the table over which they are making restriction. For the purposes of this thesis, we implemented four different kinds of constraints:

Not Null Represents which column cannot store *null*.

Primary Key Contains the list of columns which are the primary key. If the list contains a single element, the primary key is that single column, and if it contains more than one element, this constraint represents a compound primary key.

Foreign Key Consists of the qualified name of the two tables, and a paired list of the columns from the two tables which are mapped together. It also features an update rule which specifies what needs to be done in the case of an update in the referenced table, as well as a delete rule, which specifies the action that must be taken in the case of deletion in the referenced table.

Index Includes the list of columns used for indexing the table.

Figure 10 represents the implementation of these constraints.

```
sealed trait Constraint
case class NotNull(field: Column) extends Constraint
case class PrimaryKey(fields: List[Column]) extends Constraint
case class ForeignKey(pkTableName: QualifiedName, fkTableName:
   QualifiedName, fields: List[(Column, Column)], updateRule:
   ForeignKeyAction, deleteRule: ForeignKeyAction) extends Constraint
case class Index(fields: List[Column]) extends Constraint
```

Figure 10: Constrains model

6 Type Provider

The architecture of *type providers* is shown in Figure 11. The *type provider* component uses existing schema in the database and creates a schema model out of it. Then, this component generates corresponding Scala AST according to the desired front-end (Lifted Embedding or Direct Embedding). Afterwards, two choices can be made. Either using *type macros* in order to provide the types in the same compilation stage or generating the code in order to provide the types by printing the corresponding code into another source file which will be used later for accessing the types.

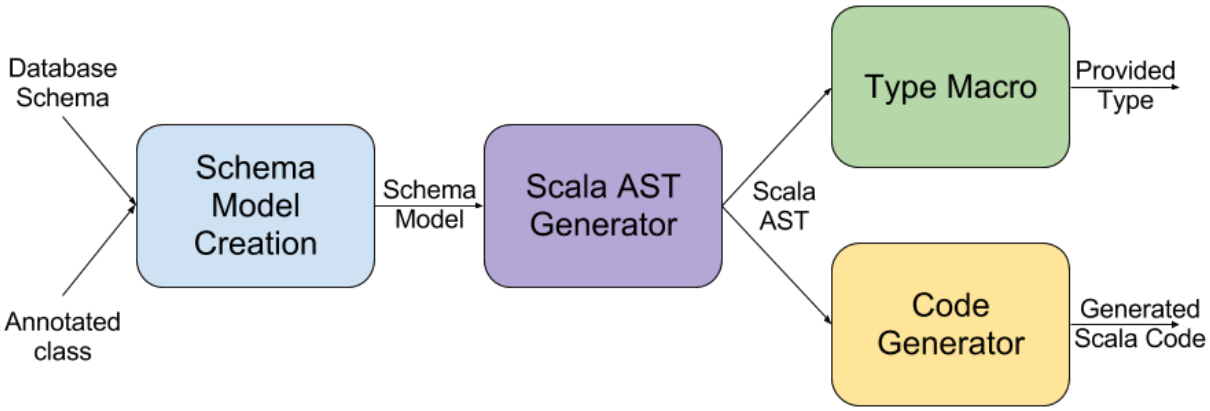

Figure 11: Type provider architecture

6.1 Schema Model Creation

Schema model is created in three phases in order to have support for providing the types lazily:

Table signatures In this phase only the table data-structures are created with their qualified names. The columns and constraints are not retrieved in this phase. After this phase, the type signature for each table will be provided without any member of that table. If a member of a table is needed to be accessed, the next two phases should be triggered for that particular table. However, there is no infrastructure from Scala *type macros* for providing the types lazily.

Table with columns In this phase, the information about columns of given tables is retrieved. The Scala type for each column is set by a default mapping from database types to Scala types. Additionally, this default mapping could be customised, which will be discussed later in Section 6.4. The only constraint retrieved in this phase is Not Null.

Table with columns and constraints Finally, the information about the constraints for the given tables is retrieved.

There is a configuration file that provides the URL, the username, the password, and the driver of the desired database engine. By using this information, a connection is made to the database and the schema information will be fetched. Then, the schema model will be created out of the given schema information.

6.2 Scala AST Generation

Tree generation converts each table model in the schema model to the Scala AST, representing table data-structure for the appropriate front-end. For example, in the case of Lifted Embedding two types should be provided for each table. One represents a *case class* that is the type of each record of the table, which we call *record class*. The other one represents the table intermediate representation which we call *IR class*.

6.2.1 Table Record Class

An instance of this *case class* represents a record from a table. The name for this *case class* is computed from the name of the table in the database. The default name for the *case class* is computed by appending `"Row"` to the table name and making it *camel case*. For example, the table name "COFFEE" is converted to "CoffeeRow". As a table denotes a class, the first character of its name should be upper case by convention. In Section 6.3 customising the naming mechanism is explained.

The fields of this *case class* represent the columns of the table. The type of each field is encoded in Column Data-Structure. The name of each field is computed by making the corresponding column name, camel case. As a column denotes a field, the first character of its name should be lower case. For example "COFFEE_NAME" is converted to "coffeeName".

6.2.2 Table IR Class

The name of this class is computed by making it camel case and making the first character upper case. For example "COFFEE" is converted to "Coffee". An *IR class* extends Lifted Embedding `Table` class. The type parameter for this class is the *record class*.

Each column has a method that invokes the `column` method of Lifted Embedding `Table` class, and the name of the column is passed to this method. Also, the type of each column needs to be passed as a type parameter to `column` method. The name of each of these methods is computed in the same way as the field name in *record class*.

For each constraint, we generate a method that encodes the information about that constraint.

Finally, for full projection `*` needs to be implemented, as it is discussed in Section 3.6. All of the columns are chained together by using `~`. Then, by using `<>` method, `apply` and `unapply` methods of the *record case class* are passed to it.

6.3 Custom Naming

There are two ways to customise the names of database entities in generated code.

6.3.1 Configuration File

The first way is by encoding the naming mechanism in the configuration file, which includes the information necessary for connecting to the database. For that purpose, we use a language very similar to JSON, but more readable than it, called *HOCON* [7]. The naming configurations are put into the `naming` object in the configuration file.

At the top-level, we have four different members.

- Naming rule for *IR class* of each table.
- Naming rule for *record class* of each table.
- Naming rule for the fields of *IR class*, which represent the columns of each table.

- Naming rule for the fields of *record class*, which represent the columns of each table.

For these top-level members, we can only provide a chain of functions that will be applied one by one consecutively. The default configuration for naming is given in Figure 12:

```
naming {
  ir-class = [lowercase, capitalize, camelize]
  record-class = [lowercase, capitalize, camelize, addrow]
  ir-field = [lowercase, camelize]
  record-field = ${ir-field}
}
```

Figure 12: Default naming configuration

With default configuration, for table "COFFEES", the *IR class* name will be "Coffees" and *record class* name will be "CoffeesRow". And, for a column of this table named "COFFEEE_NAME", *IR class* field and *record class* will be "coffeeName".

In order to customise the default rule for naming, one can override the desired members with the new rule appropriate for the use case. If no member is overridden, the default naming mechanism is used.

There are use cases in which we are interested in having different naming rules for each table. For these use cases, `naming` object has another member called `custom`. Each member of this object corresponds to a table. The name for each *table object* is the name of the corresponding table in the database. Each table object, contains four members, like top-level `naming` object, with the difference that `ir-class` and `record-class` are not representing any rule, but are instead representing a String. `ir-field` and `record-field` are still representing a chain of rules. The definition of these members in this level has higher priority than their definition in the top-level. For example, assume the configuration given in Figure 13:

```
naming {
  ir-class = [lowercase, capitalize, camelize]
  ir-field = [lowercase, camelize]
  custom {
    COFFEE {
      ir-class = Coffees
      ir-field = [lowercase]
    }
  }
}
```

Figure 13: Naming configuration in table-level

The name for *IR class* of table "COFFEE" will be "Coffees" and not "Coffee". Also, assume a column named "COFFEE_NAME" in this table. The field name for this column in *IR class* will be "coffee_name" and not "coffeeName".

Each table object has another member named `custom` which is for customising the name of each individual column of this table. The members of this object, are corresponding to the

columns of that table. Each column object, has the name as the columns. In addition, it has two members `ir-field` and `record-field`. These objects are strings that represent the name for the corresponding field for that column in *IR class* and *record class*. Column-level definitions have higher priority than table-level definitions.

Let's assume the configuration given in Figure 14:

```
naming {
  ir-field = [lowercase, camelize]
  custom {
    COFFEE {
      ir-field = [lowercase]
      custom {
        COFFEE_NAME {
          ir-field = name
        }
      }
    }
  }
}
```

Figure 14: Naming configuration in column-level

The field for column "COFFEE_NAME" in *IR class* is neither "coffeeName" nor "coffee_name". The name for this field is "name".

6.3.2 Naming API

Using configuration files for customising the names is a working solution, but it is not extremely convenient. Configuration files provide the user with enough flexibility to have customised naming for database entities. However, using configuration files requires learning the HOCON language and knowing the protocol for writing the naming configuration.

An API that provides the same flexibility as configuration files is a better solution. In order to specify that we are interested in using naming API, in the `naming` object of the configuration file, there is a member named `scala-source`, that must be set to the name of the *naming class*. This naming class must implement the naming API. The naming class must extend a class called `NamingConfigured`, and must accept a parameter of type `MappingConfiguration`. `MappingConfiguration` encodes the naming configuration done in the configuration file, as well as the default naming configuration. Naming class has four methods related to the name of *IR class*, *record class*, *IR class* fields, and *record class* fields. Each method has an input which shows the qualified name for that element. By overriding each of these methods, the naming for that particular element will be configured. The implementation for the previous example is given in Figure 15.

```
class CustomNaming(mapping: MappingConfiguration) extends
    NamingConfigured(mapping) {
  override def tableSQLToIR(name: QualifiedName): String =
    name.lastPart match {
      case "COFFEE" => "Coffees"
      case _ => super.tableSQLToIR(name)
    }

  override def columnSQLToIR(name: QualifiedName): String =
    name.getPartName(TableName) match {
      case "COFFEE" => name.lastPart match {
        case "COFFEE_NAME" => "name"
        case s => s.toLowerCase
      }
      case _ => super.columnSQLToIR(name)
    }
}
```

Figure 15: An example for custom naming

6.4 Custom Typing

There are two ways for customising the types. The first is by using configuration files, and the second one is by using *typing API*. As using configuration files is not type-safe, types are customized by using *typing API*.

There are two kinds of customizable types. The type of the entities representing a table, and the types of the entities representing a column.

Custom Column Type Assume that we want to have a type for a column different than the type defined by default. For example, in the database we have a column which shows the day of week. In the database it can be defined as integer and the corresponding default type in Scala will be `Int`. If we are interested in expressing it using another type called `Day`, we should define a bidirectional mapping between `Day` and `Int`.

Custom Table Type Custom type for a table, means custom type for the *record class*. This type will be passed as type parameter to Lifted Embedding `Table` class instead of the default *record class*. Also, the type mapping of full projection must encode how to construct a record instance from a tuple and how to deconstruct a record instance into a tuple. There is a *trait* named `TypeExtractor` which accepts two type parameters, one for the Tuple type, the other one for the record type. It has the interface for the two methods `apply` and `unapply` which converts a tuple to a record and vice versa, consecutively. Whenever a custom type is defined for a *record class*, corresponding type extractor class needs to be implemented as well.

As an example, assume that one has a field which uses integer representation in the database, knowing that it encodes a boolean value. For example, it shows the state of a coffee machine, whether it is off or on. The code in Figure 16 represents customising the type for that column and its table.

```
object CustomTyping extends TypeMapper {
  implicit val boolTypeMapper = MappedJdbcType.base[Boolean, Int](
    { b =>
      if (b) 1 else 0
    }, { i =>
      if (i == 1) true else false
    }
  )
  type MyCoffee = Tuple3[Int, String, Boolean]
  class MyCoffeeExtractor extends TypeExtractor[MyCoffee, MyCoffee] {
    override def apply(s: MyCoffee): MyCoffee = s
    override def unapply(s: MyCoffee): Option[MyCoffee] =
       Tuple3.unapply(s)
  }

  override def tableType(name: QualifiedName)(implicit universe:
     Universe): Option[universe.Type] = name.lastPart match {
    case "COFFEE" => Some(getType[MyCoffee])
    case _ => super.tableType(name)(universe)
    }

  override def tableExtractor(name: QualifiedName)(implicit universe:
     Universe): Option[universe.Type] = name.lastPart match {
    case "COFFEE" => Some(getType[MyCoffeeExtractor])
    case _ => super.tableExtractor(name)(universe)
    }
  override def columnType(name: QualifiedName)(implicit universe:
     Universe): Option[universe.Type] = name.lastPart match {
    case "STATE" if name.getPartName(TableName).equals("COFFEE") =>
       Some(getType[Boolean])
    case _ => super.columnType(name)(universe)
    }

}
```

Figure 16: An example for custom typing

`MappedJdbcType.base` accepts two `Function` objects for having a bidirectional mapping between the two types. As this implicit value should be visible when the Lifted Embedding `Table` is being defined, all the implicit members of the custom typing class must be imported into the scope where lifted embedding `Table`s are defined.

Method `tableType` returns the custom type for the table associated with the given qualified name and `tableExtractor` returns the type extractor type for that table. As one can see, `TypeMapper` provides a method named `getType` which accepts a type parameter and returns the associated `Type` object. `columnType` returns the type for a particular column. Every result is wrapped into `Option`. `None` result shows that the default type should be used for that particular table or column.

6.5 Type Macro

There is a *type macros* in module `TypeProvider` named `Db` which accepts a parameter of type `String` as input that represents the path to the configuration file. Whenever this type is mixed in or extended, the module or class which is inheriting it will be provided with the types and values provided by *type provider* component.

For example, let us assume that we have defined a table named `COFFEES` with the schema given in Table 1:

Attribute	Type	Options
COF_NAME	VARCHAR	Primary Key
SUP_ID	INTEGER	Foreign Key
PRICE	DOUBLE	
SALES	INTEGER	
TOTAL	INTEGER	

Table 1: The schema of COFFEES table

On the other hand, there is another table with name `SUPPLIERS`, which has the schema shown in Table 2:

Attribute	Type	Options
SUP_ID	INTEGER	Primary Key
SUP_NAME	VARCHAR	
STREET	VARCHAR	
CITY	VARCHAR	
STATE	VARCHAR	
ZIP	VARCHAR	

Table 2: The schema of SUPPLIERS table

The meta model for these two tables is given in Figure 17.

```
Table("COFFEES", List(
    Column("COFFEES.COF_NAME", typeOf[String], "VARCHAR", "name",
       "name"),
    Column("COFFEES.SUP_ID", typeOf[Int], "INTEGER", "supId", "supId"),
    Column("COFFEES.PRICE", typeOf[Double], "DOUBLE", "price", "price"),
    Column("COFFEES.SALES", typeOf[Int], "INTEGER", "sales", "sales"),
    Column("COFFEES.TOTAL", typeOf[Int], "INTEGER", "total", "total")
  ), List(
    PrimaryKey(List(Column("COFFEES.COF_NAME", typeOf[String],
       "VARCHAR", "name", "name"))),
    ForeignKey("SUPPLIERS", "COFFEES", List(
        (
          Column("SUPPLIERS.SUP_ID", typeOf[Int], "INTEGER", "id",
             "id"),
          Column("COFFEES.SUP_ID", typeOf[Int], "INTEGER", "supId",
             "supId")
        )
      ), NoAction, NoAction
    )
  ), "Coffees", "Coffee"
)

Table("SUPPLIERS", List(
    Column("SUPPLIERS.SUP_ID", typeOf[Int], "INTEGER", "id", "id")
    Column("SUPPLIERS.SUP_NAME", typeOf[String], "VARCHAR", "name",
       "name")
    Column("SUPPLIERS.STREET", typeOf[String], "VARCHAR", "street",
       "street")
    Column("SUPPLIERS.CITY", typeOf[String], "VARCHAR", "city", "city")
    Column("SUPPLIERS.STATE", typeOf[String], "VARCHAR", "state",
       "state")
    Column("SUPPLIERS.ZIP", typeOf[String], "VARCHAR", "zip", "zip")
  ), List(
    PrimaryKey(List(Column("SUPPLIERS.SUP_ID", typeOf[Int], "INTEGER",
       "id", "id")))
  ), "Suppliers", "SuppliersRow"
)
```

Figure 17: The model for COFFEES and SUPPLIERS table

Each table model encodes its qualified name, columns, constraints, and the name of the *IR class* and *record class*. As mentioned previously in Section 5, the first argument of `Table` represents the qualified name for the table. Since in this example there is no information specified for catalog and schema, there will be no part for them in the qualified name. The second argument is the list of the columns of that table. The third argument represents the list of constraints for that table. The two last arguments represent the names for *IR class* and *record class* of that table.

A column is represented by its qualified name, Scala type, database type, and the name of its corresponding field in *IR class* and *record class*.

As it is shown in Figure 17, the *record class* name for `COFFEES` is `Coffee` and not `CoffeesRow`. As it is using the Custom Naming which was explained in Section 6.3. The same thing has happened to `COFFEES.COF_NAME`, `SUPPLIERS.SUP_ID` and `SUPPLIERS.SUP_NAME`.

Now assume that we have a module named `MyDb`, shown in Figure 18, that extends *type macro* `TypeProvider.Db` and sets the configuration path parameter to the corresponding configuration file. The configuration file states that `H2Driver` (the driver for accessing H2 query engine) must be used.

```
object MyDb extends TypeProvider.Db("path/to/configuration/file")
```

Figure 18: Module MyDb before macro expansion

This module will be expanded to the code which is shown in Figure 19.

```
object MyDb {
  import scala.slick.driver.H2Driver.simple._
  val driver = scala.slick.driver.H2Driver
  case class Coffee(val name: String, val supId: Int, val price:
      Double, val sales: Int, val total: Int)
  case class SuppliersRow(val id: Int, val name: String, val street:
      String, val city: String, val state: String, val zip: String)
  object Coffees extends Table[Coffee] ("COFFEES"){
    def name = Coffees.this.column[String]("COF_NAME")
    // other columns implementation
    def * = Coffees.this.name ~ Coffees.this.supId ~ Coffees.this.price
        ~ Coffees.this.sales ~ Coffees.this.total <> (Coffee.apply,
        Coffee.unapply _)
    def pkCoffees = Coffees.this.primaryKey("pkCoffees",
        Coffees.this.name)
    def fkSuppliers = Coffees.this.foreignKey("fkSuppliers",
        Coffees.this.supId, Suppliers)(supplier => (supplier.id),
        scala.slick.lifted.ForeignKeyAction.NoAction,
        scala.slick.lifted.ForeignKeyAction.NoAction)
  }

  object Suppliers extends Table[SuppliersRow] ("SUPPLIERS"){
    def id = Suppliers.this.column[Int]("SUP_ID")
    // other columns implementation
    def * = Suppliers.this.id ~ Suppliers.this.name ~
        Suppliers.this.street ~ Suppliers.this.city ~
        Suppliers.this.state ~ Suppliers.this.zip <> (SuppliersRow.apply
        _, SuppliersRow.unapply _)
    def pkSuppliers =
        Suppliers.this.primaryKey("CONSTRAINT_PK_SUPPLIERS",
        Suppliers.this.id)
  }
}
```

Figure 19: Module MyDb after macro expansion

As a result, the module `MyDb` will include members `driver`, `Coffee`, `SuppliersRow`, `Coffees`, and `Suppliers`. These member can be accessed in a type-safe manner.

6.6 Code Generation

The other approach, which does not use *type macros*, is using code generation. In a separate compilation step, the code which holds the types of the tables will be generated into a module. By using this generated module, the user can have access to the *IR class* and *record class* of the tables defined in the database.

The main advantage of having code generation is that it is not dependent on *type macros* that are less likely be adopted by the core Scala language. In addition, the generated source code is visible for the user, whereas by using *type macros* the code generation is performed on the fly and is not visible to the user.

The main disadvantage of this approach is that the user should trigger the compiler explicitly for using the provided types. Also, if there is a change in the schema of the database, the generated code should be regenerated.

The code which is generated by code generation for `COFFEES` and `SUPPLIERS` tables (which were defined in Table 1 and 2) is identical to the expanded code which is shown in Figure 19.

7 Limitations

One of the limitations of *type providers* in Slick is that they do not support lazily providing of types. The schema model creates the model lazily, but there is no support from *type macro* side. There are techniques for implementing lazy *type providers*, but these techniques are still a research prototype. Hence, it is not possible to provide the types coming from an infinite information space, since they must be fetched lazily.

The other limitation is that the schema model is only created from relational databases. The main reason for this limitation is that Slick does not support non-relational databases. Therefore, having support for non-relational databases in *type providers* needs the appropriate infrastructure from Slick.

Part II
Shadow Embedding

8 Yin-Yang

There are two approaches to embedding a DSL into a host language:

Shallow Embedding Means executing programs directly on the host language values, without the need for creating any IR. As there is no need for creating IR, the type errors are comprehensible, the compilation is fast, and the debugging is easy. But, as the program is executed directly, it cannot perform any code optimization or code analyses. Therefore, *shallow DSL*s are suffering from low performance and cannot perform domain-specific analyses.

Deep Embedding Means creating an IR of the program, and then either interpreting this IR, or generating a new code. In the meantime, the created IR can be passed to a code optimization and a code analysis phase, before passing to the final phase. As a result, the performance is improved and domain-specific analyses are allowed. deep embedding can be achieved in two ways. The first way is at run-time by using type system (for example implicit conversions in Scala) or language extensions. This approach is used in LMS [33] and Polymorphic Embedding [21] in Scala, and the work by Miguel Guerrero et al. [19] in MetaOCaml. Runtime deep Embedding uses complex types which are hard to understand, and as a result, the error messages are incomprehensible [15]. Also, debugging is harder and domain-specific analysis can be done only during run-time. The second way is at compile-time by using meta-programming facilities like macros and template meta-programming. Compile-time deep Embedding does not require complex interfaces, but the debugging is still hard and their development needs knowledge of compiler internals.

Shadow Embedding Yin-Yang [24] is a DSL embedding framework, which translates shallow DSLs to deep DSLs using macros. This framework has the advantages of shallow embedding and deep embedding at the same time, without having most of their drawbacks. We coin the term *Shadow Embedding* for the approach Yin-Yang is using. Sha*d*ow is a mixture of two words shallow and *D*eep, and shows that Shadow Embedding is a combination of shallow embedding and deep embedding.

8.1 Architecture

The flowchart of the operations in Yin-Yang is shown in Figure 20. In this figure, dashed boxes represent optional phases, boxes with X represent error states, and boxes with check-mark represent successful states.

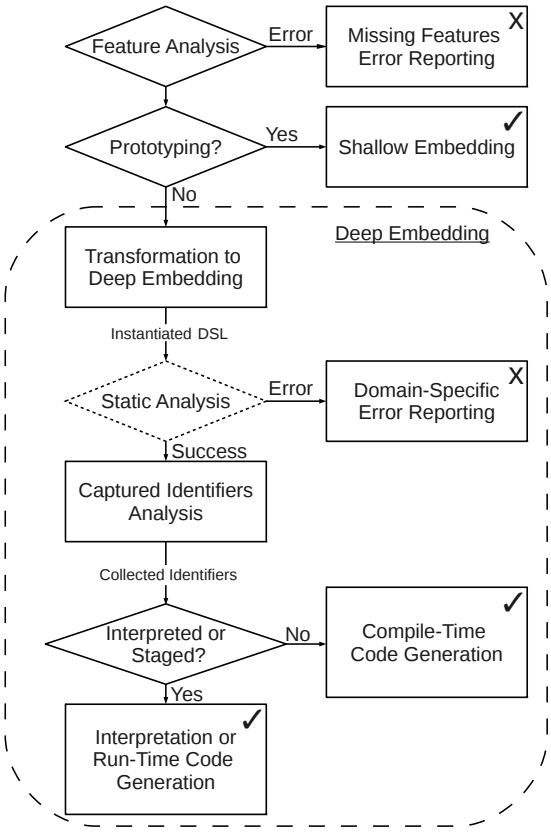

Figure 20: Yin-Yang operations flowchart

First, by using the *Feature Analysis* component, Yin-Yang checks whether all methods are supported by DSL or not. If one of the methods is not supported by DSL, it will report it using appropriate error message.

Prototyping checks the user configurations to decide whether it should use shallow DSL, or it should transform it to deep DSL.

Transformation transforms the shallow DSL to its corresponding deep DSL.

Static Analysis is an optional operation, which performs domain-specific static analysis, and reports if there is any error.

Captured Identifiers Analysis checks which captured identifiers are needed for optimization according to the DSL author configuration. If an identifier is necessary for the optimization, the associated value will be captured and the DSL will be recompiled if its value is changed.

Finally, *Interpreted or Staged* uses the given configuration and captured identifiers to decide whether it should generate the code or it should interpret it during run-time.

Yin-Yang consists of the following components:

8.1.1 Feature Analysis

This component analyses methods and language constructs to check whether they are supported by deep DSL or not. Firstly, language constructs are virtualized, by being normalized to method calls. Then, *Feature Analysis* checks whether all methods in shallow DSL can be translated to deep DSL. If it is not possible for all methods, an appropriate error message will be reported.

8.1.2 Captured Identifier Analysis

This component analyses captured identifiers to check which identifier's run-time value is necessary to be captured for optimization and which one should not be captured. If the value of an identifier is required for the optimization, the associated value will be lifted at run-time. As deep DSL encodes this kind of captured identifier by its associated value, if this value changes, the deep DSL must be recompiled. If there is no captured variable required for the optimization, Yin-Yang can generate code for the result deep DSL.

8.1.3 Transformation

The transformation component, transforms shallow DSL into deep DSL according to the given configuration. This component is described in more detail in Section 8.2.

8.1.4 Interpretation or Code Generation

This component uses the deep DSL, and according to the given configuration and captured identifiers, either interprets the deep DSL during run-time, or generates the corresponding code which will be executed later. If the configuration states that the DSL must be interpreted, Yin-Yang interprets the deep DSL during run-time. If the configuration states that the code for DSL must be generated, the situation is a bit different. If the optimizations do not require run-time values, code generation will be executed at compile time. However, if some run-time values are necessary for optimizations, those values are captured in run-time and will be passed to the code generator, in order to generate the code according to captured values.

8.1.5 Stage Guards

As run-time compilation is a very costly operation, it should be performed only when required. Yin-Yang installs a guard around DSL recompilation, which checks whether the captured identifiers for optimizations have the same values as in the previous run or not. If one of the values is different, DSL will be recompiled with the new values, and if all of the values are constant, the previously compiled DSL will be used.

8.2 Transformation

Transformation from shallow DSL to deep DSL is done in different phases shown in Figure 21. First, *Language Virtualization* translates language constructs into method calls. Then, *Ascription* places type ascriptions for each expression to assure successful type checking. Afterwards, *Lifting* converts captured identifiers and literals to the appropriate method calls. Then, in the *Type Transformation* phase every type will be transformed according to the rule defined by the DSL author. Finally, *Scope Injection* injects the DSL block into the *deep DSL component*.

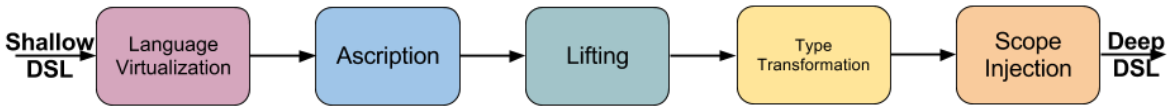

Figure 21: Yin-Yang transformation architecture

8.2.1 Language Virtualization

Yin-Yang uses the same approach as Scala Virtualized [32] for *Language Virtualization*, except that Yin-Yang uses macros for the conversion, whereas in Scala Virtualized, the compiler converts language constructs into method calls. The language constructs which are translated in this phase are: conditionals, loops, `new`, `return`, mutable variable operations, and the `try` construct. To summarize, this phase normalizes the shallow DSL to contain only method applications, method definition, immutable variable definition, variables, functions, and objects.

8.2.2 Ascription

This phase is for assuring the success of type checking, specially in the case of dealing with `Rep[T]`. There are cases in which type inference for `T` succeeds in shallow DSL, whereas the type inference for `fails` in deep DSL. As a result, the inferred type for each expression will be extracted and will be placed in the appropriate place in shallow DSL, and later in the *Type Transformation* phase, this type will be transformed to appropriate deep type. The ascription transformation is applied on method arguments, method invocations, the return type of method definition, function objects (of type `T1 => T2`), and variable definitions.

8.2.3 Lifting

Replaces all literals and captured identifiers with corresponding IR in deep DSL. There are two kinds of captured identifiers. First, the identifiers which were classified by *Captured Identifier Analysis* as not needed for optimization, which are denoted by *Lifted Captured Identifiers*. Second, the identifiers possessing a value that is needed for optimization, which are denoted by *Hole Captured Identifiers*.

Literal is replaced with the application of a method named `lift` to the literal value. `lift` method is in charge of conversions from shallow DSL value to deep DSL IR.

Lifted Captured Identifier `i` is also replaced with the method invocation `lift(i)`.

Hole Captured Identifier `h` is replaced with the method invocation `hole[h.type](classTag[h.type], symbolId(h))`, in which `classTag[h.type]` returns a `ClassTag` object which encodes the type information of `h` and `symbolId(h)` returns the identifier of the corresponding symbol for `h`.

8.2.4 Type Transformation

According to the configuration given by the DSL author, *type transformation* transforms every shallow type, which was ascribed previously in the *Ascription* phase, to the corresponding deep type.

If we are using Polymorphic Embedding [21] to embed deep DSL, every type appearing in

shallow DSL must have a corresponding abstract type member in the *deep DSL component* (Will be explained in Section 8.3.1). In Polymorphic Embedding, the non-function type `T` is translated to `this.T`, the function type `T1 => T2` is translated to `this.T1 => this.T2`, and the higher-kinded type `H[M]` is transformed to `this.H[this.M]`. The translation is identical if the given type is a method type parameter. These rules are summarized in Table 3.

Shallow Type	Context	Deep Type
`T`	Normal	`this.T`
`T1 => T2`	Normal	`this.T1 => this.T2`
`H[M]`	Normal	`this.H[this.M]`
`T`	Type Apply	`this.T`
`T1 => T2`	Type Apply	`this.T1 => this.T2`
`H[M]`	Type Apply	`this.H[this.M]`

Table 3: Type transformation rules for Polymorphic Embedding

In LMS [33], every non-function type `T` is translated to `Rep[this.T]`, every function type `T1 => T2` is converted to `Rep[this.T1] => Rep[this.T2]`, and every higher-kinded type `H[M]` is transformed to `Rep[this.H[this.M]]`. In the context of *Type Apply*, the translation will be identical to the Polymorphic Embedding. These rules are summarized in Table 4.

Shallow Type	Context	Deep Type
`T`	Normal	`Rep[T]`
`T1 => T2`	Normal	`Rep[this.T1] => Rep[this.T2]`
`H[M]`	Normal	`Rep[this.H[this.M]]`
`T`	Type Apply	`this.T`
`T1 => T2`	Type Apply	`this.T1 => this.T2`
`H[M]`	Type Apply	`this.H[this.M]`

Table 4: Type transformation rules for LMS

8.2.5 Scope Injection

Injects the shallow DSL into the deep DSL Component. Shallow DSL is using the objects of shallow DSL, which must be transformed to the corresponding object in the deep DSL Component. The objects containing shallow interface, and the `scala.Predef` which provides common definitions, are exceptions and are transformed to `this` instead of the corresponding element in the deep DSL Component.

8.3 API

Yin-Yang framework provides the DSL authors with an API which connects the shallow interface and deep interface to the framework. A deep DSL component must be defined by the DSL author. The name of this component and an appropriate Type Transformer must be passed to Transformation API. Also, there are other configurations which the DSL author can pass to Transformation API in order to disable or modify the phases.

8.3.1 Deep DSL Component

deep DSL component is the main component which must be implemented by the DSL author. There is a method named `main` in this component that contains the deep DSL, which is the result of transforming shallow DSL. This component must support the following functionalities:

Type and Value Rewiring The component must contain deep version for the types and values appeared in shallow DSL.

Captured Identifier Analysis The component must provide a method `requiredHoles`, which returns the Hole Captured Identifiers (see Section 8.2.3). The signature of this method is in *trait* `BaseYinYang`.

Lifting The definitions of `lift` and `hole` is given in `BaseYinYang` *trait*. The implementation of these two methods for each specific type, uses the implementation of these two methods in an implicit parameter of type `LiftEvidence`. DSL author must implement an appropriate implementation of `LiftEvidence` for each type. The *trait* `BaseYinYang` is shown in Figure 22.

```
trait BaseYinYang {
  def requiredHoles(): List[Symbol]
  abstract class LiftEvidence[T: TypeTag, Ret] {
    def hole(tpe: TypeTag[T], symbolId: Int): Ret
    def lift(v: T): Ret
  }
  def hole[T, Ret](tpe: TypeTag[T], symbolId: Int)(implicit liftEv:
      LiftEvidence[T, Ret]): Ret = liftEv hole (tpe, symbolId)
  def lift[T, Ret](v: T)(implicit liftEv: LiftEvidence[T, Ret]): Ret =
      liftEv lift (v)
}
```

Figure 22: Interface of BaseYinYang

Static Analysis `StaticallyChecked` *trait*, contains `staticallyCheck` method. If deep component extends this *trait*, it must statically check given DSL during compilation. The interface for this *trait* is given in Figure 23.

```
trait StaticallyChecked {
  def staticallyCheck(c: Reporter)
}
```

Figure 23: Interface of StaticallyChecked

Interpretation If the result deep DSL is interpreted during run-time, the component must extend a *trait* called `Interpreted`. DSL author must provide an implementation for method `interpret`, which will be invoked when the program is executed. This method accepts the values for holes, and returns the result of executing DSL program. Whenever, the value of one of the lifted captured identifiers changes, `reset` method will be invoked for invalidating optimizations.

Code Generation If the DSL author is interested in using code generation, the component must extend `CodeGenerator`. If no captured identifier is needed for optimization, the deep DSL code can be generated in the compile-time. In this case, the `generateCode` will be invoked, and the shallow DSL code will be substituted by the generated deep DSL code. If some captured identifiers are needed for optimization, compile-time code generation is not possible, and compilation should be done in the run-time. In this case, the DSL author should implement the method `compile`. `Ret` type is instantiated to a Scala function type `(Any, ..., Any) => T` which represents a function object that accepts the values for holes, and returns the result of executing the DSL program.

The interfaces for `Interpreted` and `CodeGenerator` are shown in Figure 24.

```
trait Interpreted {
  def reset(): Unit
  def interpret[T: TypeTag](params: Any*): T
}

trait CodeGenerator {
  def generateCode(className: String): String
  def compile[T: TypeTag, Ret]: Ret
}
```

Figure 24: Interface of Interpreted and CodeGenerator

8.3.2 Transformation

The interface for Transformation accepts the name of deep DSL component, an object which encodes type transformation, and other configurations for disabling or modifying the transformation phases (For example a flag that specifies whether the program should be run in Prototype mode or not, which was explained in Section 8.1). Also, Transformation interface could be provided with a post processing phase, which can perform manual transformations by the DSL author. The factory for Yin-Yang Transformer is shown in Figure 25.

```
object Transformer {
  def apply[C <: Context, T](c: C)(
    dslComponentName: String,
    tpeTransformer: TypeTransformer[c.type],
    postProcessing: Option[PostProcessing[c.type]],
    config: Map[String, Any])
}
```

Figure 25: Yin-Yang Transformer factory

Type Transformation An object of the *abstract class* `TypeTransformer`, transforms every given `Type` object, to the tree representation that encodes transformed type. Method `transform`, takes care of transforming the type to the tree representation of transformed type. In addition to the `Type` object, the context in which that type occurs is also given. In other words, it is given whether the given type is in the context of a *Type Apply*, or it occurs in a *Normal* context.

Figure 26 shows the interface of `TypeTransformer`.

```
abstract class TypeTransformer[C <: Context](val c: C) {
  trait TypeContext
  case object TypeApplyCtx extends TypeContext
  case object NormalCtx extends TypeContext
  def transform(ctx: TypeContext, t: c.universe.Type): c.universe.Tree
}
```

Figure 26: Interface of TypeTransformer

9 Shadow Embedding

Shadow Embedding is the proposed front-end for Slick with the goal of having the virtues of Direct Embedding and Lifted Embedding, without having their drawbacks. Yin-Yang converts transparently every query expression, written in shallow interface, to its corresponding deep IR during compilation by using macros. Hence, from the user perspective every query expression has shallow type, which is standard Scala type, but behind the scenes every query expression is mapped to the corresponding deep IR. As a result, type-errors are comprehensible (like Direct Embedding). Furthermore, as the conversion from shallow interface to deep IR is performed during compilation, if there are any unsupported operations in deep IR, an appropriate type-error will be reported by compiler. Additionally, by using *feature analysis* in Yin-Yang (See Section 8.1.1), we can do more domain-specific analysis and report errors which are not possible to be reported by Lifted Embedding or Direct Embedding during compilation. Hence, type-errors are comprehensive (even more than Lifted Embedding).

The deep IR of Shadow Embedding uses deep IR of Lifted Embedding. More precisely, Shadow Embedding deep IR is a wrapper over Lifted Embedding deep IR. As a result, Shadow Embedding is interoperable with Lifted Embedding, which causes lots of code reusability, and makes the code more maintainable. Furthermore, Shadow Embedding can reuse the *type providers* of Lifted Embedding.

Shadow Embedding provides *composability* for the user, which means that a part of code used in lots of places, can be defined once and be accessed in many different places. To provide composability, it is necessary to convert deep IR to shadow interface. This feature is provided by using *Transferable Pattern* which will be discussed in Section 9.4.2.

Captured Identifiers Analysis of Yin-Yang (See Section 8.1.2), specifies the run-time value of which captured identifiers are necessary for optimization. The run-time value of these identifiers is lifted and is encoded in deep IR. The captured identifiers without this criterion are considered as the input parameter, and are encoded as a special deep IR node, called `Hole`. The *hole* identifiers are mapped to *Query Parameter* node in Slick AST, which corresponds to *parameters* in `PreparedStatement` of JDBC [20].

Finally, *Stage Guards* of Yin-Yang (See Section 8.1.5), keeps track of the changes in the value of lifted captured identifiers. When the query expression is interpreted for the first time, the query compiler is invoked and the result SQL statement is cached. For the next executions, there is no need for recompiling the query, and the SQL statement cached previously can be used. However, if there is a change in the value of an identifier in the query expression, which causes a change in the result SQL statement, the cache is invalidated by the guard, and the query expression must be recompiled. Therefore, for a fixed query expression, Shadow Embedding saves the time that Lifted Embedding was consuming to recompile the query.

9.1 Architecture

The architecture of Shadow Embedding is given in Figure 27. The query expressions are written using shallow interface. *Yin-Yang Transformer* converts the shallow query expressions into corresponding *deep IR*. Then, *shadow interpreter* either compiles the deep IR or uses the previous cached compiled query in order to create an appropriate SQL statement. Afterwards, *shadow Interpreter* runs the given SQL statement against the given query engine, and returns the result of executing that query.

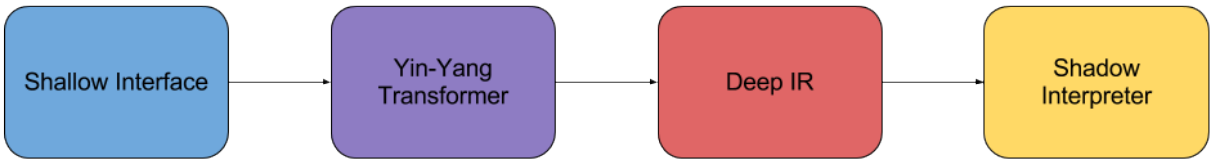

Figure 27: Shadow Embedding architecture

9.2 Shallow Interface

Shallow interface provides the interfaces which are necessary for the user to interact with. These interfaces should not necessarily implement the functionality, it is enough to have only the signature for each method. Having the implementation is necessary for Prototyping (See Section 8.1) which is not needed in Slick. Shallow interface consists of the following components:

Query Interface is the interface for query expressions. It contains all the methods which were discussed in Section 3.2.4.

Join Query Interface is a subtype of *Query interface* that has another method called `on` for specifying the join predicate.

Single Column Query Interface is an interface for the queries which result in a single column, and provides aggregation methods (`min`, `max`, `avg`, and `sum`) for these kinds of queries.

Query Factory for a Table is a module that whenever is applied to a type `T` creates a Query object for a table in which each record is of type `T`.

Numeric Extension contains the additional operations for numeric types, which are not part of the numeric types signature in the Scala standard library, but are supported in SQL (such as `toRadians`, `toDegrees`, `sign`, etc.)

String Extension contains String additional operations which do not exist in the Scala standard library, but are supported in SQL (such as `like`, `ltrim`, `rtrim`, etc.).

Null Ordering provides two methods for controlling the ordering of the tuples containing *null* elements.

Additional Implicit Conversions provides implicit conversions for converting from the Scala standard types to their extension types in shallow interface. For example, for an expression of type `Int`, in order to use method `toRadians` on it, it needs to be converted to the appropriate numeric extension type that includes additional operations.

9.3 Deep IR

The deep IR for Shadow Embedding is a wrapper over Lifted Embedding IR. As a result, each method invocation on shadow deep IR will delegate the corresponding method on Lifted Embedding. The result of the invocation will be wrapped into another shadow deep IR node, which contains the underlying Lifted Embedding IR result node.

The main reason for wrapping over Lifted Embedding IR, and not using Lifted Embedding IR directly, is that the method signatures in shallow interface do not match with the ones in Lifted Embedding. For example, `map` method in shallow *Query interface* accepts only one type

parameter (`map[T]`), whereas the one in Lifted Embedding accepts two type parameters (`map[T, S]`). As Scala compiler does not have the capability of partially applying the types, it is not possible to leave the inference of not given types to the compiler. But, the method signatures in shadow deep IR match with the ones in shallow interface. In the previous example, `map` method in shadow deep IR accepts one type parameter (`map[T]`). shadow deep IR is responsible for rewiring the methods in Lifted Embedding IR.

The main deep IR nodes are:

Query includes the methods of *Query interface* and whenever each method is invoked an IR node representing appropriate query expression will be created.

Table is the IR node for a table in the database.

Column represents a column in the database.

Constant Column represents a constant value.

Hole represents a parameter the value of which will be specified later. It corresponds to the `QueryParameter` node in the Slick AST.

Projection represents a tuple of columns. Hence, every element of this tuple must represent a column.

Tuple represents a tuple of arbitrary elements. The difference with *Projection* node is that the *Tuple* node can contain elements other than columns. These kinds of tuples can also represent the nested tuples. The main usage of a *Tuple* node is in *Group By* and *Join*s.

Struct represents a companion module for a *case class*. A companion module of a *case class* creates a *case class* instance whenever it is applied to the fields of that *case class*.

Ordering represents the priority of each element in the ordering of a tuple. Additionally, it encodes the direction in which each element of a tuple is ordered, as well as the priority of `null` values.

9.4 Yin-Yang Integration

DSL author is responsible for configuring Yin-Yang to make it work for the desired DSL. There is no need for Slick to enable *Prototyping* mode in Yin-Yang. All the domain specific analysis is performed by the compiler when shallow interface is transformed to the deep IR. Therefore, no *Static Analysis* has been implemented. *Captured Identifier Analysis* is done by analysing the type of each identifier (See Section 9.4.1). *Lifting* expressions is explained in Section 9.4.2. *Type transformation* for Shadow Embedding in Slick, is different than the type transformation in LMS and Polymorphic Embedding (See Section 9.4.2). The *record IR class*, for the Record types defined by the user, are created using *type providers* (See Section 9.4.4).

9.4.1 Captured Identifiers Analysis

It is necessary to analyse captured identifiers to distinguish the identifiers whose value must be encoded in the deep IR (*Lifted Captured Identifiers*), and the ones that must be substituted by a *Hole* (*Hole Captured Identifiers*, see Section 8.2.3).

Lifted captured identifiers mainly consist of the query expressions. The main use-case for lifting

captured query expressions is for the composition. A query is calculated in a *shadow block* and another block is using that query. Therefore, the identifier encoding this query expression is a captured identifier and its value does not change frequently. As a result, its value can be used for optimization and if its value is changed the optimization will be invalidated by *Stage Guards*.

Hole captured identifiers include primitive identifiers which represent the parameters in a query expression. Let us assume a method with an argument of type `Int` with a body that is the *shadow block*. This *shadow block* is using the argument of that method as a part of the query expression. As the value of this identifier is being changed frequently, it is not performant to encode its value in deep IR, because that would result in lots of unnecessary query recompilations. Therefore, we will put a `Hole` node in the deep IR that corresponds to the `QueryParameter` node in Slick AST. The Slick query compiler converts the `QueryParameter` node to a *parameter* in the `PreparedStatement` of JDBC.

We can distinguish the captured identifiers based on their type. If the type of an identifier is `Query` it must be lifted, and if an identifier has a type other than `Query`, it must be substituted by a *Hole*. There is a marker *trait*[2] called `TypeAnalyser`, that whenever a *deep DSL component* (See Section 8.3.1) is extending it, there is no need for implementing `requiredHoles` method. It is enough for the DSL author to provide the *Yin-Yang Transformer* with the types that must be lifted. In this case, we are passing `Query` type for the *Yin-Yang Transformer*.

9.4.2 Lifting

Literals are converted to an appropriate `Constant Column`.

Lifted Captured Identifiers which are representing the identifiers of type `Query`, must be converted to a deep `Query`. As a result, the identifiers of type `Query` must encode their deep IR outside of *shadow block*. For this purpose, we are using an approach called *Transferable Pattern*.

Transferable Pattern is used for composing the result of a DSL into another DSL block. If the result of a *shadow block* is a `Query`, its deep IR must be encoded in an object of type `TransferableQuery`, which is a subtype of shallow `Query`. Therefore, when this query object is being used in another *shadow block*, this object will be lifted to a deep `Query` simply by accessing its underlying deep `Query`.

Hole Captured Identifiers which are identifiers of primitive types, are converted to a `Hole` node in deep IR.

9.4.3 Type Transformation

The type transformation of Slick Shadow Embedding is different than Polymorphic Embedding and LMS. In the *Normal* context, the non-function type `T` is translated to `this.T`, the function type `T1 => T2` is translated to `this.T1 => this.T2`, and the higher-kinded type `H[M]` is transformed to `this.H[M]`. The types are not changed in the context of *Type Apply*. Table 5 shows the rules of the type transformation in Shadow Embedding.

[2] a *trait* that has no member and is used as a flag.

Shallow Type	Context	Deep Type
`T`	Normal	`this.T`
`T1 => T2`	Normal	`this.T1 => this.T2`
`H[M]`	Normal	`this.H[M]`
`T`	Type Apply	`T`
`T1 => T2`	Type Apply	`T1 => T2`
`H[M]`	Type Apply	`H[M]`

Table 5: Type transformation rules for Shadow Embedding

9.4.4 Class Virtualization

If we are using a table record type in the query expression, which is not defined in the Scala standard library (for example a user-defined *case class*), the corresponding `Table` node in deep IR must be created. In other words, the table record type must be *virtualized* to *table IR class*. The *type provider* component provides the definitions for Lifted Embedding *table IR class*, that can be used in Shadow Embedding. The architecture of *class virtualization* is shown in Figure 28.

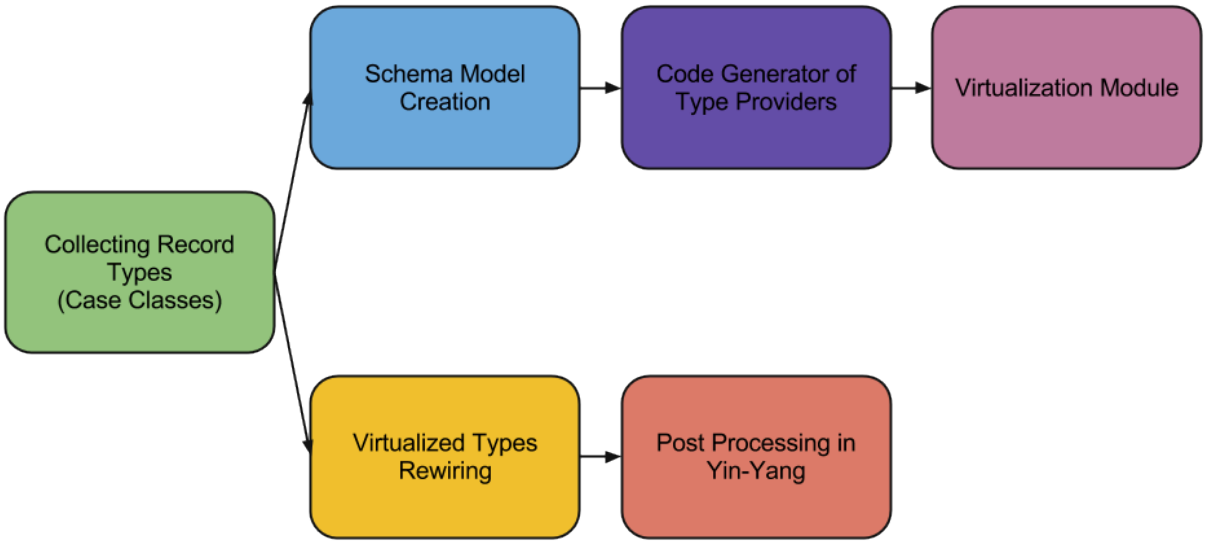

Figure 28: Class virtualization architecture

Collecting Virtualized Types is the step in which the table record types, defined as a *case class* outside of the scope, are collected. Additionally, there is an option to provide the definition of the *case class*es in the *shadow block*, which is implemented and can be used. The *case class* definitions are using JPA annotations [25] for transforming into corresponding entity in the database.

Schema model creation creates the schema model out of the collected annotated *case class*es. Figure 29 shows an example of an annotated *case class* representing `COFFEES` table.

The schema of this table was shown in Table 1.

```
@Table(name = "COFFEES")
case class Coffee(
  @Id
  @Column(name = "COF_NAME")
     name: String,
  @Column(name = "SUP_ID")
     supId: Int,
  @Column(name = "PRICE")
     price: Double,
  @Column(name = "SALES")
     sales: Int,
  @Column(name = "TOTAL")
     total: Int
)
```

Figure 29: Annotated case class for COFFEES table

There is no support for the foreign keys and other constraints. However, as there is enough infrastructure in *type providers* and the *Schema Model*, supporting the other constraints is feasible.

Code Generator of Type Providers generates the shadow deep IR and Lifted Embedding IR definitions for the given record types, and saves the code into *virtualization module*.

Virtualization Module for a table named `Coffee`, must have the members shown in Figure 30.

```
object VirtualizationModule {
  type CoffeeRow = /*original Coffee case class*/
  class CoffeeTable extends lifted.Table[CoffeeRow] {
    /* Lifted Embedding Table provided by type providers */
  }
  class ShadowCoffeeTable(lifted: CoffeeTable) extends
      deep.Table[CoffeeRow] {
    /* A wrapper for CoffeeTable */
    // delegates the method calls to the ones in CoffeeTable
  }
  val CoffeeRow = /* companion module of the original Coffee case class
      */
  object CoffeeTable extends CoffeeTable
  implicit object ShadowCoffeeTable extends
      ShadowCoffeeTable(CoffeeTable)
  implicit def convertCoffeeToIR(coffeeRep: ShadowRep[CoffeeRow]):
      ShadowCoffeeTable
}
```

Figure 30: Virtualization module of class virtualization

Virtualized Types Rewiring imports the types which were provided in the *virtualization module* into transformed deep DSL. In addition, for each record type this component defines corresponding abstract type member in the *deep DSL component*. Furthermore, by using *Struct* (See Section 9.3) a companion module will be created for each record type. Afterwards, in the *post processing* phase of the transformation the codes shown in Figure 31 are injected into the *deep DSL component*.

```
import VirtualizedModule._
type Coffee = ShadowRep[CoffeeRow]
val Coffee = new Struct[CoffeeRow](
  CoffeeRow.apply _
)
```

Figure 31: Virtualized types rewiring

This design made the type transformer simpler. At the beginning, record types had special rules which made the design of type transformer very complicated. Then, by using this design the type transformer is very simple (See Table 5).

9.5 Shadow Interpreter

Shadow interpreter is responsible for evaluating a query expression written using Shadow Embedding. If the query expression is evaluated for the first time, firstly, this query expression must be transformed to the corresponding deep IR. Then, the query compiler must compile it into the SQL statement. Afterwards, an *Invoker* object (See Section 3.5) wrapping the compiled SQL statement will be created. This *Invoker* object is cached for the following executions, if the cached values are still valid, there is no need to recompile the query again. Therefore, the cached invokers are used to execute the query. The workflow is shown in Figure 32.

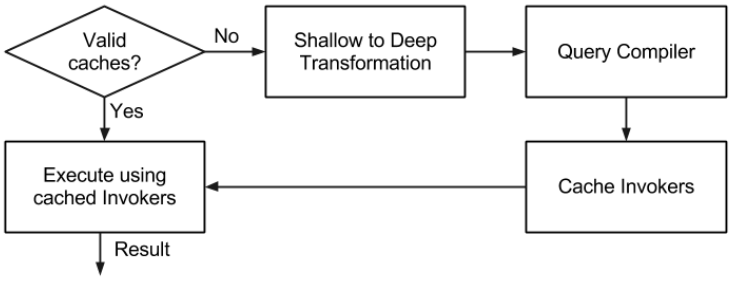

Figure 32: Interpretation workflow in Shadow Embedding

If the value of a lifted captured identifier changes, the cache is invalidated. This situation occurs when the value of a captured query is changed. Since a part of the query expression is changed, the previous compiled SQL statement is no longer a valid translation for the query expression. Therefore, the query must be recompiled.

If there is a hole captured identifier, the run-time value for this identifier is passed to *shadow interpreter*. In this case, *shadow interpreter* creates a *parameterized query* instead of a simple

query, and passes the run-time values for the parameters to this *parameterized query*.

Another component is necessary for executing the query. *Shadow interpreter* only maintains the cached values for the invokers, but it does not execute the queries. Additionally, a *shadow block* only expresses the query expression and does not execute any query. *Shadow Executor* is responsible for the execution of each query expression.

9.5.1 Parameterized Query

This query contains parameters which need to be filled with appropriate values. The deep IR encodes these parameters with `Hole` nodes which will be translated to *parameters* in the final `PreparedStatement` of JDBC. The *parameters* in JDBC `PreparedStatement` will be set by the run-time values, which are provided by *shadow interpreter*.

As in the deep IR of a *Parameterized Query* there is no node changing during different executions, it can be compiled only once. These queries have the same functionality as *Query Templates* in Lifted Embedding.

We can use *Parameterized Queries* for updates in Shadow Embedding. There is no *Query Template* for update implemented in Lifted Embedding. In order to implement it, Lifted Embedding needs a change in lots of components. However, thanks to the hole captured identifiers in the *Yin-Yang*, it is implemented in the Shadow Embedding with less effort.

9.5.2 Shadow Executor

Executing a query against the appropriate query engine is done by *shadow executor*. Whenever an execution operation is performed on a `Query` object, the `Query` object will be implicitly converted to a `ShadowExecutor` object. The `ShodowExecutor` will use the corresponding *shadow interpreter* for the given `Query` object in order to manage the caching of *invokers*. Afterwards, by using the cached *invokers*, the query will be executed.

There are four types of executions done by *shadow executor*:

- Selection
- Update
- Insertion
- Deletion

Shadow executor performs the execution of the *parameterized queries* in a *thread-safe* manner. *Shadow interpreter* passes the run-time values of the parameters by creating immutable *parameterized query* objects. As a result, there will be no race condition.

9.6 Example

Assume that we have a table `Coffee` and are interested in retrieving the name of all elements of this table.

Figure 33 shows the Lifted Embedding code for this query. The `CoffeeTable` object is an object of table IR type which can be provided either by the user or by *type provider*. The `query` object is Lifted Embedding `Query` which will be executed in the next line. This object is implicitly converted to an *invoker* by executing the query compiler. This implicit conversion triggers the query compiler to compile the query. The `list` method executes the query and returns the result.

```
val query: lifted.Query[Coffee, CoffeeTable] = for (c <-
    lifted.Query(CoffeeTable)) yield (c.name)
query.list()
```

Figure 33: An example of a selection query in Lifted Embedding

Figure 34 shows the Shadow Embedding code for this selection query. This code will be transformed by Yin-Yang to the code shown in Figure 35. The keyword `stage` is used to specify the *shadow block*. The `shallow.Queryable` object is the factory method for creating queries out of the tables (See Section 9.2). Finally, the `list` method forces the compiler to implicitly convert `query` from `shallow.Query` to `ShadowExecutor`.

```
val query: shallow.Query[Coffee] = stage {
  for (c <- shallow.Queryable[Coffee]) yield c.name
}
query.list()
```

Figure 34: An example of a selection query in Shadow Embedding

In Figure 34, `deep.Queryable` accepts an implicit parameter which is an object of table IR type. This object is provided by *type provider* by using the record type. `convertCoffeesToIR` is an implicit method, in order to specify that `c` must be of table IR type and not of table record type (See Figure 30). Otherwise, it is not possible to create a correct deep IR. `TransferableQuery` wraps the computed deep IR, to make it composable in other *shadow block*s. In addition, `TransferableQuery` encodes the `dslComponent` which takes care of caching the invokers and a list of parameter values needed for the parameterized queries. Finally, `ShadowExecutor` executes the `query` and returns the result. For the sake of simplicity, no information about the *Stage Guards* is shown in Figure 35. `ShadowExecutor` uses the caching information stored in the `dslComponent` object in order to not recompile the query every time.

```
val query: deep.Query[Coffee] = new TransferableQuery(
  for(c <- deep.Queryable[Coffee](ShadowCoffeeTable)) yield
      (convertCoffeeToIR(c).name), dslComponent, Nil
)
(new ShadowExecutor(query)).list()
```

Figure 35: Expanded version of the selection query example in Shadow Embedding

10 Evaluation

We have implemented the *type provider* in approximately one month and a half. Shadow Embedding was designed and developed in three months. As Shadow Embedding shares its deep representation with Lifted Embedding, there was no need for reimplementing all the IR nodes for deep DSL. However on the other hand, this design required more tricks to integrate the deep IR of Shadow Embedding and Lifted Embedding IR.

The *schema modelling* required around 230 LoC, in which 120 LoC was dedicated to create the schema model out of a given database, 70 LoC was dedicated to *naming*, and the rest (40 LoC) used for the schema models themselves. The *type provider* overall consists of around 900 LoC, from which 300 LoC is dedicated to creating Scala AST out of the given schema model. This amount of code will be significantly decreased by using *quasi-quotes* [34]. 250 LoC is dedicated to *custom naming*, 50 LoC is dedicated to *custom typing*, 80 LoC is used for reading the information from configuration files. The rest (220 LoC) is used for generating the code from the given Scala AST. This amount can also be decreased by using *quasi-quotes*.

Shadow Embedding is written in around 1200 LoC. 90 LoC is used for defining the *shallow interface*. 260 LoC is dedicated for the *shadow deep IR*, in which the rewiring with Lifted Embedding IR is performed. 120 LoC is used for handling tuples and projection of columns. As we are generating the code for tuples of arbitrary size, the generated code will be expanded to 1300 LoC. 30 LoC is used for creating *holes*. 30 LoC is used for creating *structs*. 30 LoC is dedicated to *lifting*. *Type transformer* requires 50 LoC. *Class virtualization* overall needs 280 LoC. *Shadow interpreter* is written in 150 LoC. 110 LoC is used for defining the *deep DSL component*. Finally, instantiating *Yin-Yang transformer* requires 50 LoC.

All of our experiments were performed on Intel Core i7-2600K CPU running at 3.40GHz, with 16 GB of DDR3 memory running at 1333 MHz. We used Scala 2.10.2, on the JDK 1.6 with the JIT compilation stabilized. Performance evaluation is investigated in Section 10.2 and 10.3.

10.1 Correctness

We have designed two sets of test units. The first set of unit tests is identical to the test units of Direct Embedding, consisting of the tests for `map`, `filter`, `flatMap`, and `sort` methods of `Query`. There are also several tests for evaluating correctness of standard String operations and numeric operations. In addition, a number of tests are dedicated to construct the tuples in the query expression. Each tuple in the query expression must represent a projection of different columns, and there is no test for nested tuples. However, nested tuples are tested in the second test suites. Furthermore, the ordering based on different priority, different direction, and dealing with `null` elements of a tuple is investigated in the first set of unit tests.

The second set consists of the important tests of Lifted Embedding. Below you can find the list of unit tests of this category:

- creating projections of different columns and accessing different elements of the constructed tuple.
- creating and using nested tuples.
- language virtualization. For example, testing whether `==` and `if then else` statements are translated correctly.
- using simple operations on a query over a virtual record type.
- sorting based on different priority, different direction, and in the presence of `null` elements

in a tuple.
- using *for-comprehension* syntactic sugar.
- standard string and numeric operations.
- additional string and numeric operations, such as `like`, `ltrim`, `toRadians`, etc.
- using `Option` values.
- union operation.
- join operations consisting of `leftJoin`, `rightJoin`, `innerJoin`, `outerJoin` and `zip`.
- using *Group By* for a query.
- composition of queries.
- using parameterized queries for selection.
- inserting in a table.
- updating specific elements of a table.
- creating parameterized queries for update.

10.2 Micro-benchmarking

For measuring the performance of the operations taking a small amount of time, it is necessary to perform micro-benchmarking. For this purpose we are using *ScalaMeter* [10], a micro-benchmarking tool for the JVM platform.

For all of the benchmarks, we are assuming a simple table named `COFFEE` with two columns `ID` and `NAME`, of type `Int` and `String` respectively. All the benchmarks are performed after 50 rounds of warm up.

The X-axis in every figure specifies the number of iterations in which the query is executed, and the Y-axis denotes the time in milliseconds.

The performance for Direct Embedding is not included in the figures, because its performance was approximately **30x** worse than Lifted Embedding. The reason was explained previously in Section 3.7.

10.2.1 Selection

Case I Let us assume a very simple query which returns the element with the given identity. Figure 36 shows that the speed up of Shadow Embedding is around **250x** in comparison with Lifted Embedding. The reason for this high amount of speed up is that a very high amount of the running time for this query is consumed for the query compilation. Furthermore, the amount of time needed for fetching the result is negligible in comparison with the time needed for query compilation. However, Shadow Embedding is `20%` slower than Plain SQL, because Shadow Embedding requires boxing and unboxing the result in a *case class* object.

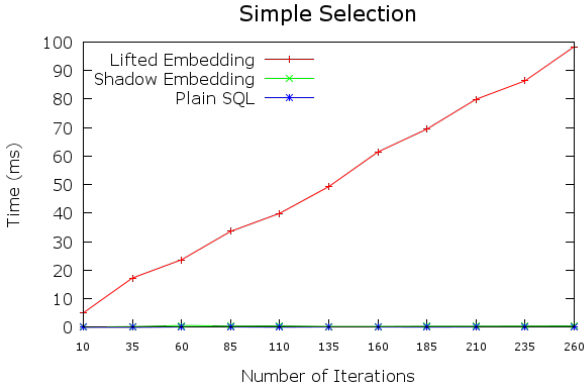

Figure 36: Performance results for simple selection

Case II The other performance test for selection is performed for a query which contains an input parameter. The query will return the elements with the identity values smaller than the given input parameter. In each iteration, a different value will be passed as the input parameter. Figure 37 shows that in this case the performance of Shadow Embedding is almost identical to the performance of Lifted Embedding using *query templates*. This is due to Shadow Embedding is using the *query template* of Lifted Embedding behind the scenes. But, if Lifted Embedding is not using *query templates*, Shadow Embedding has **4x** speed up. Again, Shadow Embedding is around **20%** slower than Plain SQL, because of the overhead of boxing and unboxing.

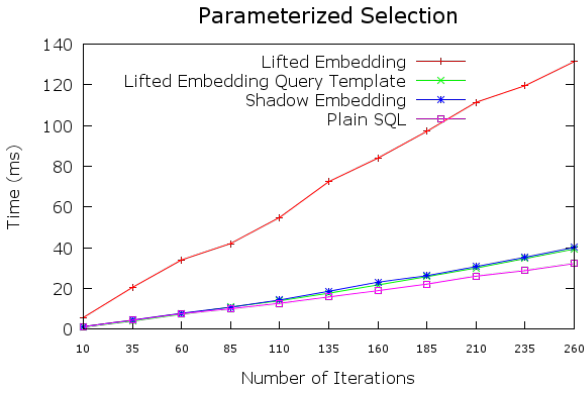

Figure 37: Performance results for parameterized selection

10.2.2 Insertion

For insertion we perform two benchmarks. The first benchmark is for inserting a constant element into the table in different iterations. The second one is inserting an element which will be varied during each iteration.

Case I Figure 38 shows the performance results for the case of inserting a constant element in all iterations. Shadow Embedding is around **10x** faster than Lifted Embedding. The perfor-

mance of Shadow Embedding and Plain SQL are almost identical in this case, because there is no overhead for boxing and unboxing.

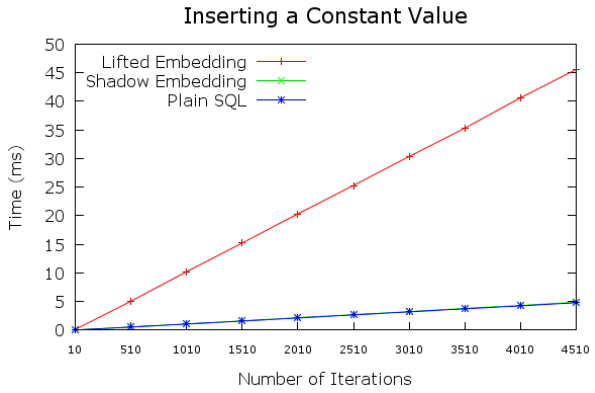

Figure 38: Performance results for insertion of a constant value in all iterations

Case II Performance results for inserting different values during iterations is shown in Figure 39. The element which will be inserted in the table is constructed using a captured identifier. Shadow Embedding is **7x** faster than Lifted Embedding in this case. As the inserted element must be constructed, again we will have boxing and unboxing overhead. Therefore, Shadow Embedding is **20%** slower than Plain SQL.

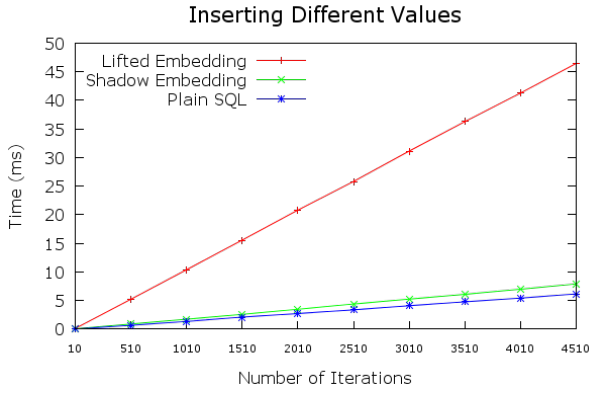

Figure 39: Performance results for insertion of different values in each iteration

10.2.3 Update

For update we are considering three different cases. Shadow Embedding codes for these three cases are given in Figure 40. In this code `coffee` is an object which will be changing during each iteration. In *Case II* and *III*, `id` and `name` are captured identifiers which are being changed in each iteration.

```
// Case I
stage {
  Queryable[Coffee].filter(_.id == 10)
}.update(coffee)
// Case II
stage {
  Queryable[Coffee].filter(_.id == id)
}.update(coffee)
// Case III
stage {
  Queryable[Coffee].filter(_.id == id && _.name == name)
}.update(coffee)
```

Figure 40: Shadow Embedding code for update benchmarks

Case I: In this case we are updating a fixed element. Figure 41 shows that Shadow Embedding is **5x** faster than Lifted Embedding, and its performance is almost the same as Plain SQL.

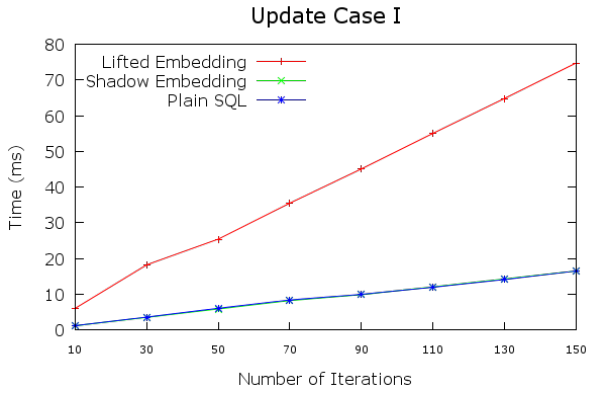

Figure 41: Performance results for update case I

Case II: The element which is updated in this case is being changed in each iteration. As shown in Figure 42, Shadow Embedding and Plain SQL are again **5x** faster than Lifted Embedding.

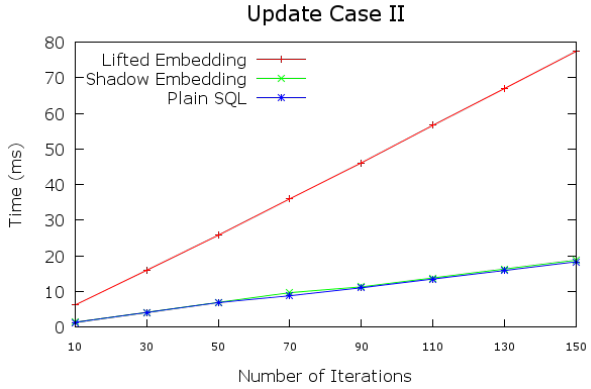

Figure 42: Performance results for update case II

Case III: The condition for selecting the element is more complicated than the previous two cases. Furthermore, there are two captured identifiers which are being changed during different iterations. Figure 43 shows that Shadow Embedding is around **4x** faster than Lifted Embedding. Again, its performance is similar to Plain SQL.

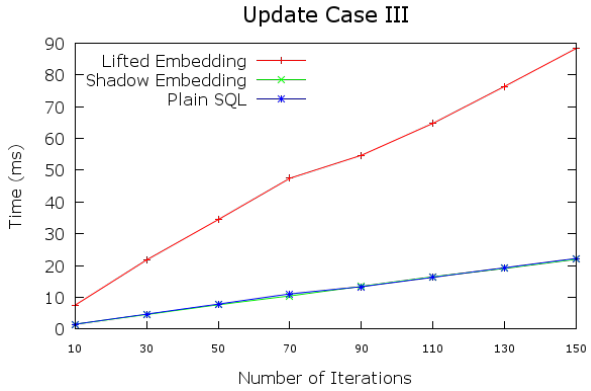

Figure 43: Performance results for update case III

10.3 Databench

Databench [5] is a persistence benchmark for JVM, which compares the performance of different persistence systems. This benchmark uses the table `ACCOUNT` with the schema given in Table 6, and uses Postgres [37] as the query engine.

Attribute	Type	Options
ID	INTEGER	Primary Key
BALANCE	INTEGER	
TRANSFERS	VARCHAR	

Table 6: The schema of ACCOUNT table

In this benchmark there are *50,000* accounts, and *100,000* transactions are executed between these accounts. In each transaction, a value will be decreased from the `BALANCE` of an account and will be added to the `BALANCE` of another account. Also, the transferred value will be concatenated to the `TRANSFERS` field of both accounts. Furthermore, during these transactions the information from database is read *400,000* times, to check the correctness of the transactions. Figure 44 shows the performance of different systems.

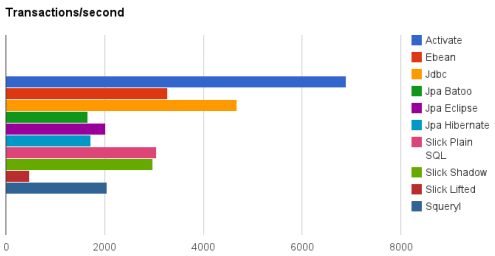

Figure 44: Performance results for Databench

Shadow Embedding is around **6.3x** faster than Lifted Embedding and has almost similar performance as Plain SQL. Activate Framework [4] is **2.3x** faster than Shadow Embedding, because of the more clever backend that it provides. Ebean [6] is around **10%** faster than Shadow Embedding. JDBC [20] is processing the transactions **50%** faster than Shadow Embedding. The reason for this amount of difference is that in JDBC the low-level caching for the `PreparedStatements` can be performed, whereas Slick does not possess the proper API to support it. Shadow Embedding is around **50%** faster than different JPA [25] systems as well as Squeryl [12].

11 Limitations

As Shadow Embedding is dependent on *type provider*, another compilation stage is necessary to provide the data-structures for virtual record types. This extra compilation stage makes the workflow more complicated for the user. However, an appropriate workflow template is designed using *simple build tool* [11].

Lots of standard Scala types are not supported in the *Shadow block*. Additionally, it is not possible to use user-defined types and methods in the *Shadow block*. Because all types and methods must be mapped to appropriate element in the *deep DSL component*, but the ones which are not defined for the *deep DSL component* will cause a type error.

12 Future Work

12.1 Type Providers

Using *macro annotations* is not as fragile as *type macros* for *type providers*. If a class or the members of a class are annotated by *macro annotations*, the class will be expanded by defining new members or modifying the defined members. Hence, by annotating a class by macro annotations, it is possible to create a type member representing a Lifted Embedding `Table`.

12.2 Shadow Embedding

By using macro annotations there is no need for an extra compilation stage for generating the code for the data-structures of virtualized record types.

In the current design of Slick, the `update` method in Lifted Embedding and Shadow Embedding accepts a value. As a result, the updating value must be computed at the client side, and then passed to the server. If `update` accepts a function instead of a value, it will be possible to compute the new value based on the previous values in the server side. Therefore, the `update` function will be more expressible in this case.

shadow block captures the identifiers of primitive types and query expressions. By adding more deep IR nodes, it will be possible to capture tuple identifiers as well.

Additionally, it is very useful to define a function in a *shadow block* and use it in that block or even in other blocks. This feature requires lifting functions. For functions having a single argument, this feature is already implemented in the Shadow Embedding prototype. However, the support for lifting functions with more arguments is considered as a future work.

Supporting nested data is related to Slick backend. If Slick uses an approach similar to Ferry [18] or T-LINQ [23], Shadow Embedding will be able to support nested data.

12.3 Shadow Programming

The Shadow Embedding approach used in this thesis for Slick, can be used for embedding other DSLs in a host language. The transformation from *shallow DSL* to *deep DSL* can be performed by a transformation component such as Yin-Yang. The result deep DSL contains several types which are needed to be virtualized to an appropriate type in deep DSL. Virtualizing the shallow types to corresponding deep types can be done by using the *type provider* component. Hence, by using Yin-Yang and *type provider* together it is possible to convert a *shallow DSL* to its corresponding *deep DSL* in a robust and type-safe manner.

13 Conclusion

We have presented Shadow Embedding in Slick, which uses Yin-Yang to transparently transform shallow DSL to deep DSL.

Shadow Embedding is as beautiful as Lifted Embedding and Direct Embedding. The type errors are comprehensible like Direct Embedding and comprehensive like Lifted Embedding. Finally, Shadow Embedding has almost the same performance as Plain SQL.

Using Shadow Embedding can be generalized for embedding other DSLs into a host language like Scala. The main components are a transformation component like Yin-Yang and a component for generating the types like *type providers* in Slick.

The implementation for Shadow Embedding in Slick is publicly available in the Slick repository[3]. Shadow Embedding in Slick is dependent on the Yin-Yang project[4].

[3] https://github.com/slick/slick
[4] https://github.com/vjovanov/mpde

References

[1] Hibernate. `http://www.hibernate.org/`.

[2] Hibernate Plugin for IntelliJ IDEA. `http://www.jetbrains.com/idea/webhelp/hibernate.html`.

[3] Java Data Objects (JDO) Specification JSR-12, 2003.

[4] Activate Framework. `http://activate-framework.org/`, 2013.

[5] Databench. `http://databen.ch/`, 2013.

[6] Ebean ORM Persistence Layer. `http://www.avaje.org/ebean/introduction.html`, 2013.

[7] HOCON (Human-Optimized Config Object Notation). `https://github.com/typesafehub/config/blob/master/HOCON.md`, 2013.

[8] Macro Paradise. `https://github.com/scalamacros/kepler`, 2013.

[9] Scala Language-Integrated Connection Kit (Slick). `http://slick.typesafe.com`, 2013.

[10] ScalaMeter. `http://axel22.github.io/scalameter/`, 2013.

[11] Simple Build Tool. `http://www.scala-sbt.org/`, 2013.

[12] Squeryl - A Scala ORM for SQL Databases. `http://squeryl.org/`, 2013.

[13] Eugene Burmako. Scala Macros: Let Our Powers Combine! In *4th Annual Scala Workshop*, 2013.

[14] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *International Conference on Software Engineering*, pages 97–106, 2005.

[15] Krzysztof Czarnecki, John T O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72. Springer, 2004.

[16] Carl Gould, Zhendong Su, and Premkumar Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 697–698. IEEE Computer Society, 2004.

[17] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 645–654. IEEE, 2004.

[18] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 1063–1066, New York, NY, USA, 2009. ACM.

[19] Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar Swadi, and Walid Taha. Implementing dsls in metaocaml. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 41–42. ACM, 2004.

[20] G. Hamilton and R. Cattell. $JDBC^{TM}$: A Java SQL API, 1997.

[21] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 137–148. ACM, 2008.

[22] Paul Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.

[23] Sam Lindley James Cheney and Philip Wadler. A practical theory of language-integrated query. In *The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2013.

[24] Vojin Jovanovic, Vladimir Nikolaev, Ngoc Duy Pham, Vlad Ureche, Sandro Stucki, Christoph Koch, and Martin Odersky. Yin-yang: Transparent deep embedding of dsls. Technical report, Technical Report, EPFL, Lausanne, Switzerland, 2013.

[25] Mike Keith and Merrick Schincariol. *Pro EJB 3: Java Persistence API*. Apress, 2006.

[26] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[27] David Maier. Advances in database programming languages. chapter Representing database programs as objects, pages 377–386. ACM, New York, NY, USA, 1990.

[28] Russell A McClure and Ingolf H Kruger. SQL DOM: compile time checking of dynamic SQL statements. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 88–96. IEEE, 2005.

[29] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD Conference*, page 706, 2006.

[30] Jim Melton. Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies, 2000.

[31] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Acm Sigplan Notices*, volume 43, pages 423–438. ACM, 2008.

[32] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *PEPM'12*, pages 117–120, 2012.

[33] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[34] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for scala. Technical report, Technical Report. EPFL, Lausanne, Switzerland, 2013.

[35] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[36] Daniel Spiewak and Tian Zhao. Scalaql: language-integrated database queries for scala. In *Software Language Engineering*, pages 154–163. Springer, 2010.

[37] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[38] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, et al. Strongly-typed language support for internet-scale information sources. Technical report, Technical Report. Microsoft Research, 2012.

[39] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*, pages 117–133. Springer, 2000.

[40] Todd Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, 1995.

[41] Murali Venkatrao and Michael Pizzo. SQL/-CLI—a new binding style for SQL. *ACM SIGMOD Record*, 24(4):72–77, 1995.

[42] Jan Christopher Vogt. Type safe integration of query languages into scala. Master's thesis, Diplomarbeit, RWTH Aachen, Germany, 2011.

[43] A. H. Ibrahim W. R. Cook. Integrating Programming Languages and Databases: What is the Problem?, Sept 2006.

Appendix

A Slick AST

Slick AST, is an intermediate representation for all SQL statements. It provides a high-level abstraction over SQL statements in order to make them independent of the query engine and its driver. These nodes are used among different phases of query compilation. Here we only introduce the nodes which are visible to first phase of compilation. Hierarchy of AST is shown in Figure 45.

Node Represents a node in AST.

Symbol Represents symbol which can be used in AST.

UnaryNode A node with only one child.

BinaryNode A node with two children.

RefNode Super type for nodes which are referencing to a symbol.

DefNode Super type for all nodes which are introducing symbols.

TypedNode The nodes which are associated with a type.

TableNode Represents a table.

ColumnNode Represents a column.

ProductNode Represents a node which is conjunction of some other nodes, which are its children nodes.

ElementSymbol Symbol of an element of a `ProductNode`.

StructNode Represents a structure, which is a conjunction of components, in which each individual component has associated `Symbol` with them.

LiteralNode Represents a literal value.

Pure Represents a plain value which is lifted into a `Query`.

FilteredQuery Super type for nodes which are applying a filtering over a Query.

Filter Represents `.filter` call.

SortBy Represents `.sortBy` call. It also uses another kind of node named `Ordering` which encodes the nodes, priority and direction of ordering.

GroupBy Represents `.groupBy` call.

Take Represents `.take` call.

Drop Represents `.drop` call.

Join Represents a join expression. It uses another node called `JoinType` which carries the information about type of join(`Inner`, `Left`, `Right`, `Outer` and `Zip`).

Union Represents a union operation.

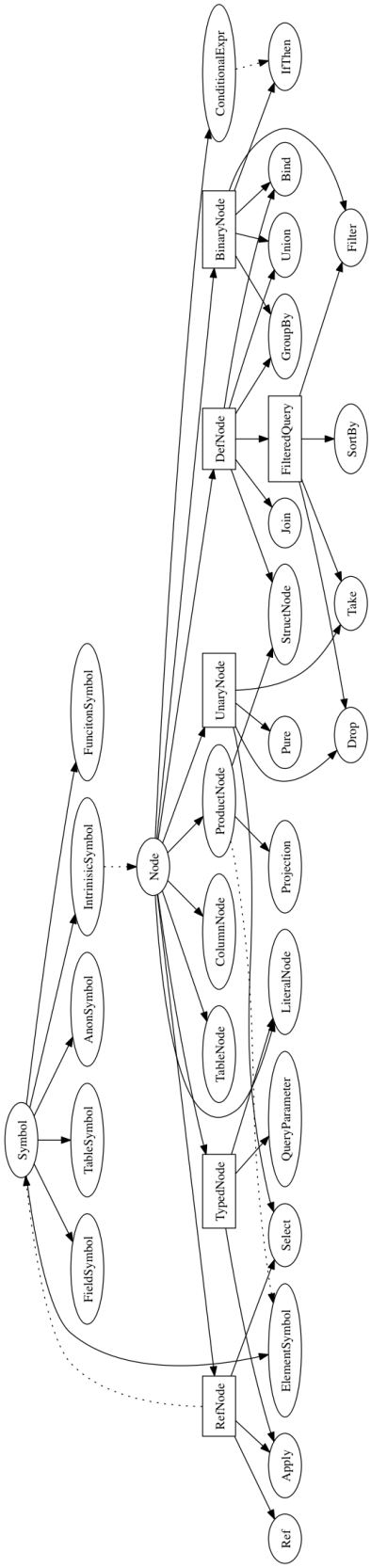

Figure 45: Slick AST hierarchy

Bind Represents `.flatMap` call.

Select Represents selection of a filed in another expression.

Apply Represents a function call expression.

Ref Represents a reference to a symbol.

ConditionalExpr Represents a conditional expression.

IfThen Represents if-then parts of `ConditionalExpr`

QueryParameter Represents a parameter in `QueryTemplate` which should be turned into a bind variable.

FieldSymbol A named symbol which refers to a field.

TableSymbol A named symbol which refers to a table.

AnonSymbol An anonymous symbol which is defined in AST.

IntrinisicSymbol A symbol which is associated with the identity of a `Node` object. It can be used to reference a specific `Node` object when constructing the AST from the front-end.

FunctionSymbol A symbol which represents a library function or an operator.

B Query Compiler

B.1 Standard Phases

Localize References Converts all `IntrinsicSymbol` nodes into `AnonSymbol` and will put all of them in a `LetDynamic`. So the references to them same node, refer to the same symbol.

Reconstruct Products In order to represent a tuple, we have used `ProductNode`. The children of this node are the elements of that tuple. In this phase, the representation of each element is converted from `Select(ref, ElementSymbol(idx))`, in which `ref` is a `Ref` node, to `ref`.

Inline The references to a global symbol which occurs only once, will be inlined and removed from `LetDynamic`.

Assign Unique Symbols In this phase, it is ensured that symbol definitions are unique.

Expand Tables In this phase, `TableNode`s are replaced by `TableExpansion`s. `TableExpansion` includes `TableNode` and list of nodes corresponding to its columns.

Create Result Set Mapping All `TypeMapping` nodes are removed and all type mapping information is hoisted into `ResultSetMapping`. This phase also insures that the top-level node should be of `CollectionType`. So if its type is not `CollectionType`, it will put it into `Pure` node, and then this node is put into `First` node to denote only this single element.

Force Outer Binds All collection operations will be wrapped in a `Bind` so that we have a place for expanding references in the next phases.

Expand References As it was noted in Slick AST section, `Path`s are representing nested `Select`s starting at `Ref` node. This phase will expand `Path`s to `ProductNode`s into `ProductNode`s of `Path`s. Also it will convert `Path`s to `TableExpansion`s into `TableRefExpansion`s of `Path`s.

Replace Field Symbols In this phase, the references to `FieldSymbol`s in `TableExpansion` nodes are replaced by appropriate `ElementSymbol` which includes the index of element as it was noted.

Rewrite Paths In this phase all `TableExpansion` and `TableRefExpansion` nodes are removed. Also all `ProductNode`s are flattened into `StructNode`s. And unnecessary columns are being removed from them.

Prune Fields Unreferenced fields are removed from `StructNode`s in this phase. There exists a problem for right-side nodes of `Union` for this phase. There is no reference for left-side nodes of `Union` which is resolved by assigning the symbols of the references of left-side nodes to right-side nodes.

B.2 Relational Phases

Resolve Zip Joins It converts zip joins to a SQL standard, by using inner join and `RowNumber` nodes. `RowNumber` needs the information about the ordering of rows, which will be filled in phase Fix Row Number Ordering.

Assign Types In this phase, type information is computed for every node.

Convert To Comprehensions Basic ASTs are converted to a suitable shape for relational databases. In this phase, all nodes of type `Bind`, `Filter`, `SortBy`, `Take`, and `Drop` are substituted by a `Comprehension` node, which represents a SQL comprehension. Also, nested `Comprehension` nodes are merged in this phase.

Fuse Comprehensions This phase fuses sub-comprehensions into their parents.

Fix Row Number Ordering In this phase, appropriate ordering is injected into `RowNumber` nodes which was produced previously in Resolve Zip Joins.

Hoist Client Operations The operations which are preferred to be done at client-side, are hoisted out of sub-queries.