Directembedding: Concealing the Deep Embedding of DSLs

Ólafur Páll Geirsson

School of Computer and Communication Sciences

Semester Project

June 2015

Responsible
Prof. Martin Odersky
EPFL / LAMP

Supervisor
Vojin Jovanović
EPFL / LAMP

Abstract

Authors of embedded domain-specific languages (EDSLs) commonly struggle to find the right balance between the capability and usability of their DSL. On one hand, deeply embedded DSLs give great power to the DSL author but have a steep learning curve for end users. On the other hand, shallowly embedded DSLs are more limiting for the DSL author but offer a more familiar interface to the end users that enables them to quickly become productive with the DSL.

This report presents work on *Directembedding*, a Scala library to implement a thin user-friendly layer on top of an existing deeply embedded DSL[1]. The library accomplishes this using annotations and macros, and requires little to no knowledge of the Scala reflection API. We used Directembedding to implement *slick-direct*, a front-end for the functional relational mapping library *Slick*. Leveraging Directembedding features, slick-direct is able to support a large feature set of Slick in under 300 lines of code.

Contents

[1]Note. This work builds on a previous semester project on the Directembedding library

1 Introduction

Domain-specific languages (DSLs) provide a simple and high-level way for programmers to accomplish a domain-specific task. DSLs differ from general purpose programming languages in the sense that they enable the programmers to think at a higher level of abstraction at the price of having restricted capabilities. One common use case for DSLs is to enable novice programmers and experts in fields outside of software development to become productive programmers.

One method to implement DSLs is to embed them inside a host language. This has the benefit that the DSL can leverage the facilities of the host language. The downside is that an embedded DSL has less flexibility to give arbitrary semantics to a given program. An embedded DSL must obey the host language's syntax and predefined behavior. EDSLs largely fall into two categories:

- *Shallowly embedded DSLs* offer an interface on top of values that are directly provided by the host language. In Scala, these are values such as `Int` and `String`. The benefit of shallow EDSLs is that they have a small learning curve for end users. The interface is familiar to programmers who already have some experience with the host language. The downside to shallow EDSLs is that they are inconvenient for the DSL author. The values in the DSL may have predefined behavior by the host language or third-party libraries. The DSL author must work around these limitations in order to give domain-specific meaning to the programs in her DSL.

- *Deeply embedded DSLs* offer an interface on top of host-language data-structures, which we refer to as an intermediate representation (IR). In Scala, this could be a type such as `Column[Int]` or `Column[String]` for a database DSL. The benefit of deep EDSLs is that they are convenient for the DSL author. The DSL author has full control over the IR, and can therefore give any meaning to programs which invoke operations on the IR. Moreover, deeply embedded DSLs have shown promising results for domain-specific optimizations [11, 10] and multi-target code generation [1]. The downside to deep EDSLs is that they can have a steep learning curve for end users. The types in the IR and their behavior may be unfamiliar to the programmers even though they may have some experience with the host language. In a way, deep EDSLs are not too different from ordinary libraries in a general purpose programming language.

There is a clear struggle between DSL users and authors: the users prefer shallow EDSLs while the authors prefer deep EDSLs. Directembedding aims to please both parties. The DSL author can conveniently create her deeply

embedded DSL and then use Directembedding to provide a shallow EDSL-like interface for end users. For an in-depth discussion on combining shallow and deep EDSLs, see Svenningsson and Axelsson (2013) [14].

The main contributions presented in this report are the following:

- Extend previous work on the Directembedding library by adding the possibility to 1) override behavior of predefined and third party types 2) give arbitrary semantics to many standard Scala features 3) configure the reification of DSL programs. Moreover, much work has been put into improving the error messages generated by the library. This work is explained in Section 2.

- Do the first case study on the practical use of the Directembedding library. In under two weeks, we implemented *slick-direct*: a front-end for the `Query` API in the functional relational mapping library Slick. Slick-direct is under 300 lines of code and delegates all implementation logic to the underlying Slick API. Slick-direct supports query operations such as `map`, `flatMap`, `filter`, and `join` with greatly simplified type signatures compared to the lifted embedding in Slick. This work is covered in Section 3.

Throughout the paper we assume familiarity with the basics of the Scala Programming Language [8].

2 Directembedding

The architecture of the Directembedding library went through a major overhaul in this project. The reification has been extended with new annotations and new capabilities such as language virtualization. The reification is now highly customizable by the DSL author. The library also aims to provide useful error messages where possible.

The following sections explain the improvements that have been made to the Directembedding library in this project. For more details on how Directembedding works please consult the project's Github site, linked at the end of this report.

2.1 Architecture

Figure 1 shows the new architecture of Directembedding. PreProcessing is an optional pass in the shallow embedding where the DSL author can transform the program in any way necessary before reification. PreProcessing requires knowledge of the Scala reflection API. The DSLVirtualization pass performs the language virtualization explained in Section 2.2. This pass happens in the shallow embedding. `ReificationTransformation` is the major component of Directembedding and lifts the shallow embedding into the deep embedding.

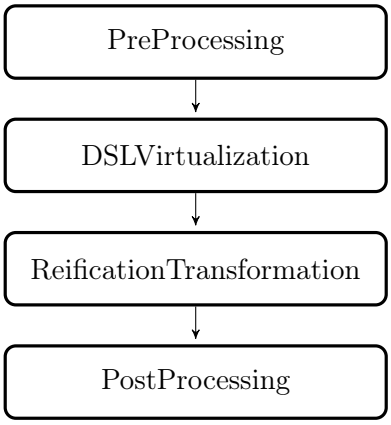

Figure 1: The Directembedding transformation pipeline.

In this pass, the metadata attached to the shallow embedding is used to reify the program into the DSL author's IR. `PostProcessing` is an optional pass through the deep embedding where the DSL author can transform the program in any way necessary before the program is passed back to the user.

The entry point to using Directembedding is now `DETransformer`. The design of the DETransformer is inspired by the `YYTransformer` in Yin-Yang [6], and uses *mixin composition* [7] to compose a series of transformation stages. An example Directembedding DSL is provided the `example` package object.

2.2 Language virtualization

Language virtualization is the process of converting standard language features into method calls, in order to give them arbitrary semantics. Such language features include if-then-else statements, loops, and variable assignments. It is not possible to override the semantics of such statements in Scala without macros.

Directembedding uses the language virtualization provided by the Yin-Yang [6] framework. This transformation happens in the `DSLVirtualization` pass. The DSL author is able to configure which language features to override through the `DslConfig` trait. The `LanguageVirtualizationSpec` shows 43 examples of how to use the language virtualization feature in Directembedding.

2.3 Overriding predefined and third-party types

Directembedding supports the ability to override behavior of predefined and third-party types. Predefined types are types provided by standard Scala libraries, such as `Int` and `String`. Third-party types can be any types in a third-party library supported by the DSL.

Reification for overriden types works the same way as reification with any other types. The `typeMap` argument to `DETransformer` tells Directembedding where to to look for reification annotations. If Directembedding does not find metadata to an invoked symbol, Directembedding will look for annotations on types in the `typeMap`. This search on types and finding matching symbols is currently implemented in a naïve way, and could be improved in future implementations. `TypeOverridingSpec` provides 6 examples of how to use the type overriding feature in Directembedding.

2.4 Improved error messages

Much effort has been put into making error messages produced by Directembedding helpful. These error messages broadly fall into two categories: i) DSL author and ii) end user error messages.

The error messages aimed at the DSL author are mostly meant to assist the author detect a DSL misconfiguration. For instance, the configuration is now provided through a trait type parameter. The trait determines the path from which all language virtualization and lift methods are implemented. If the `compile` method—the receiver of DSL program after PostProcessing—is missing, Directembedding will fail with a compilation error indicating that the method is missing. Another example is that if a reification annotation is used incorrectly, Directembedding will return a compilation error pointing to the misuse of the annotation. Finally, detailed logging of all the steps of Directembedding transform can be enabled for debugging purposes.

The error messages aimed at the DSL user are mostly meant to surface incorrect use of the DSL in a user-friendly way. Most importantly, if the user invokes a method that is missing a reification annotation, Directembedding will return a compilation error saying that the method is not supported in the DSL, pointing to the culprit invocation in the DSL program. Prior to this project, Directembedding threw a cryptic `EmptyIteratorException` in the same situation.

2.5 Configurable reification

Many of the Directembedding features are now configurable by the DSL author. The DSL author has increased control over how the reification is performed through new reification annotations beyond the original `@reifyAs` annotation. Moreover, the DSL author has now fine-grained control over how literals are lifted into deep IR. For more details on the following configuration options, please refer to the Directembedding example DSLs and documentation.

Listing 1: `@reifyAs` example

```
// Inside Query trait.
@reifyAs(Take)
def take(i: Long): Query[T, C] = ???
// Shallow query.
query {
  Query.take(1)
}
// Deep query.
Take(Query, lift(1))
```

Listing 2: `@reifyAsInvoked` example

```
// Inside Query trait.
@reifyAsInvoked
def take(i: Long): Query[T, C] = ???
// Shallow query.
query {
  Query.take(1)
}
// Deep query.
lift(Query).take(lift(1))
```

2.5.1 `reifyAs`

The `@reifyAs` annotation is useful to move the invocation of a method in the shallow EDSL into any method in the deep EDSL. This approach is inspired by related work on *finally-tagless, polymorphic embedding* [3, 5]. Listing 1 shows an example usage of the `@reifyAs` annotation. As seen in the example, the DSL author attaches the deep method as an argument to the `@reifyAs` annotation. If the argument is missing, Directembedding will fail with a compilation error.

2.5.2 `reifyAsInvoked`

The `@reifyAsInvoked` annotation is useful to preserve the invocation a front-end on top an existing deep EDSL. Instead of invoking a static method as `@reifyAs`, the `@reifyAsInvoked` preserves the shallow embedding invocation order. Listing 2 shows an example usage of the `@reifyAsInvoked` annotation. The deed query compiles only since the underlying `lifted.Query` has a matching `take` method. The use of `@reifyAsInvoked` is therefore only encouraged in cases like slick-direct, where there is a close correspondance between the shallow EDSL and deep EDSL.

Listing 3: `@passThrough` example

```
// Inside Query trait.
@passThrough
def take(i: Long): Query[T, C]
def missingAnnotation(): Int = 1
// Shallow query.
query {
  Query.take(missingAnnotation())
}
// Deep query.
lift(Query).take(missingAnnotation())
```

2.5.3 passThrough

The `@passThrough` annotation is useful to preserve values in the shallow DSL program. By default, any invoked method in a DSL program should be reified during ReificationTransformation. If a method is missing a reification annotation, the `ReificationTransformation` will return with a compilation error. Methods annotated with `@passThrough` skip reification in the ReificationTransformation. Listing 3 shows an example usage of the `@passThrough` annotation. The example exhibits a subtle yet important difference between `@passThrough` and `@reifyAsInvoked`, the former will stop the reification at the invocation point and while the latter will recursively reify the arguments to the invoked method. A careful observer may notice that the example may fail to compile, since the argument to the lifted query mustbe a lifted value. For this reason, the use of `@passThrough` should only be used with great care.

2.5.4 customLifts

By default, all literal are lifted through a method with the following signature

```
  def lift[T](e: T): Rep[T]
```

where `Rep` is the supertype of all elements in the IR. The issue with default configuration is that it ignores the hierarchy of the IR, all literals will have the type `Rep` although they may be lifted into a subtype of `Rep`. The `customLifts` parameter to `DETransformer` alleviates this issue by giving the DSL author fine grained control over which types are lifted into which IR types. For instance, the DSL author can provide a custom lift for values of type `Int` and another lift method for values of type `String`. Directembedding does not enforce that the return type of a custom lift methods is a subtype of `Rep`

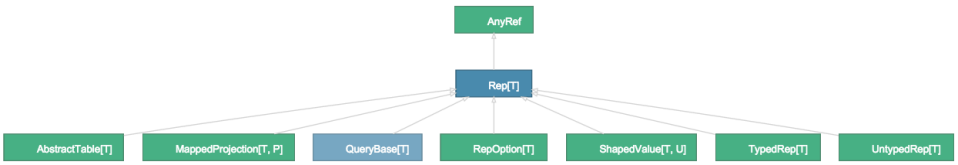

Figure 2: The type hierarchy of `slick.lifted.Rep[T]` in Slick.

2.5.5 `liftIgnore`

The `liftIgnore` configuration parameter allows the DSL author to list which literals should not be lifted during ReificationTransformation. This can be useful if certain literals are introduced in the PreProcessing step which should not be lifted.

3 Case study: slick-direct

Slick [13] is a popular Scala library used to query databases. Slick is recommended by Typesafe as the functional relational mapper for their well known Play framework. Professional Scala consultancies such as underscore.io offer public and private training on Slick and underscore.io even recently released a book about the library[2]. The hefty prices on the private training indicates that there is commercial interest in using Slick. The fact that the book is close to 300 pages may also indicate that the library has a steep learning curve.

We chose to evaluate Directembedding by implementing a front-end for Slick for a few reasons. Firstly, Slick is a widely used library in the industry. Secondly, the lifted embedding API is quite elaborate and makes extensive of many advanced features of the Scala type system, such as *lifted embedding* [9]. Thirdly, there exists a lot of related work on direct embedding for Slick.

In this case study, we compare in detail the lifted embedding with our implementation of the direct embedding. Sections 3.1 and 3.2 cover how queries are created in the lifted embedding and direct embedding, respectively. In Section 3.3, we look at the related work on direct embedding with Slick.

3.1 Lifted embedding

The lifted embedding is the recommended way to query data with Slick. The supertype of all members in the lifted embedding is the `slick.lifted.Rep[T]` trait. Figure 2 shows the type hierarchy of `Rep[T]`. To create a query with the lifted embedding, a Slick user must in one way or another interact with all subtypes in the hierarchy. The purpose of the lifted embedding API is to create an abstract syntax tree (AST) of type `slick.ast.Node`, which is

[2] http://underscore.io/training/courses/essential-slick/

Listing 4: Original case classes

```
case class User(id: Int, name: String)
case class Car(id: Int, name: String, ownerId: Int)
```

Listing 5: `Table[T]` definition

```
class Users(tag: Tag) extends Table[User](tag, "User") {

  def id: Rep[Int] = column[Int]("id")
  def name: Rep[String] = column[String]("name")

  def * = ProvenShape.proveShapeOf((id, name) <> ((User.apply
      _).tupled, User.unapply))
}

class Cars(tag: Tag)
  extends Table[Car](tag, "Car") {

  def id: Rep[Int] = column[Int]("id", O.PrimaryKey)
  def name: Rep[String] = column[String]("name")
  def ownerId: Rep[Int] = column[Int]("ownerId")
  // Foreign key constraint
  def ownerIdFk = foreignKey("ownerIdFk", ownerId,
      TableQuery[Users])(_.id)

  def * = ProvenShape.proveShapeOf((id, name, ownerId) <>
      (Car.tupled, Car.unapply))
}
```

then passed onto the query optimization engine of Slick. In the next few paragraphs, we will follow an example to see how a query inside the lifted embedding is created from the point of view of a Slick users. The following example is made up of simple database of users and cars owned our users and is split into four steps. Observe that some details such as database drivers are left out for clarity.

Listing 4 shows the definitions of our basic Scala classes: `User` and `Car`. These classes will be the values which we want to insert into and fetch from our database.

Listing 5 shows our `Table[T]` definitions, where we provide necessary metadata to query on our user and car objects. Observe that the names of the members in our standard user and car objects are repeated at least four times: i) in the case class definition, ii) in the method names in the table definition, iii) in the string literals to identify the column names in the

Listing 6: `TableQuery[T]` definition

```
val users = TableQuery[Users]
val cars = TableQuery[Cars]
```

Listing 7: Slick queries

```
users
// select * from User

user.map(_.name)
// select u.name from User u

user.filter(_.id === 1)
// select * from User where id = 1

for {
  user <- users
  car <- cars if car.ownerId === user.id
} yield user.name ++ " drives a " ++ car.name
// select concat(u.name, " drives a ", c.name)
// from User u, Car c
// where u.id = c.id

for {
  (u, c) <- user leftJoin cars on (_.ownerId === _.id)
} yield (u.name, c.map(_.name))
// select u.name, c.name
// from User u left out inner join Car c
// on u.id = c.id
```

database, and iv) in the `ProvenShape` mapping in the $*$ method. In the case of the `ownerId` — which has the foreign key constraint — we must repeat the name two more times, a combined of six times. The benefit to this table definition is that it gives great flexibility to the user. The downside is that the table definition forces a lot of boilerplate onto users — violates DRY — and may be likely to introduce bugs in the code. To alleviate this issue, Slick provides a library to generate these table definitions from a database schema.

Listing 6 shows how we create `TableQuery` objects, which we use to write queries. Queries are written with a similar syntax as with Scala collections. Listing 7 shows several examples of Slick queries and their corresponding translations to (simplified) SQL. Take a moment to appreciate some of the benefits to writing queries like this: Queries can be type-checked; IDEs can

provide auto-completion to the end user; and if a user knows how to operate on Scala collections the user can write SQL. Once we invoke a `map` or `filter` operation on a `TableQuery[T]`, we get a value of type `Query[E, T, C]` where in our example i) `E` will be `Users` and `Cars`, ii) `T` will be `User` and `Car`, and iii) `C` will typically be the collections container `Seq[T]`.

Finally, we run our queries on a database object. In Slick 3.0, the argument supplied to the database object is of type `DBIOAction[T]` — which can be created from a `slick.ast.Node` — and the database returns a values of type `T`, which will be `User` and `Car` in our case. Observe that the database object does not work with values of the lifted embedding, that is `slick.lifted.Rep[T]`. The lifted embedding is only required to provide the metadata to create a `slick.ast.Node`.

3.2 Direct embedding

The value proposition of slick-direct is twofold. Firstly, in slick-direct queries are written on the original Scala case classes — `User` and `Car` in our example — and, thus, obviates the need for the `Table[T]` in the second step. Secondly, because queries are written on the original scala case classes, type signatures in the Query API are simplified. Besides these two differences, writing queries in the direct embedding is similar to writing queries in the lifted embedding.

Slick-direct eliminates the need for the `Table[T]` by extracting necessary metadata from the original case classes. In slick-direct, a class of type `Table[T]` is generated during the `PreProcessing` step described in Section 2.1. In our example, the names of the classes and class members mapped directly into database tables and columns. However, it is possible to use annotations to customize the naming translation, as has been done in related work (see Section 3.3).

It is difficult to understate the benefit for the end user of not having to provide the `Table[T]`. First of all, users do not have to learn the `Table[T]` API to use Slick. Secondly, the user avoids repeating the same member names multiple times, which can prevent various kinds of bugs and simplifies refactoring as the database model evolves. Finally, users avoid the confusion between the basic case classes such as `User` and classes that inherits `Table[T]` such as `User` yet mostly have the same members and look similarly to the basic classes.

The second benefit to slick-direct is that the type signatures in the Query API are simplified. Herein, we highlight a few major differences between the two querying APIs. In all examples, assume the type of `this` to be `lifted.Query[E, T, _]` for `direct.Query[T, _]`, respectively. Listing 8 shows the type signatures for the `map` operation. In order to guarantee that `f` produces a value that can be persisted into a database, `lifted.Query` adds an implicit shape parameter on the type of `F`. Slick-direct eliminates the need for this shape parameter by restricting the values that can be introduces in

Listing 8: Map API

```
// slick.lifted
def map[F, G, T](f: E => F)
  (implicit shape: Shape[_ <: FlatShapeLevel, F, T, G]): Query[G,
      T, C]
// slick.direct
def map[F](f: T => F): Query[F, C]
```

Listing 9: Filter API

```
// slick.lifted
def filter[T <: Rep[_]](f: E => T)(implicit wt:
    CanBeQueryCondition[T]): Query[E, U, C]
// slick.direct
def map[U](f: T => U): Query[U, C]
```

it's shallow DSL. If the user introduces an illegal value with `f`, slick-direct will return a compilation error that the value is not supported in the DSL. However, if the value produced is valid in the shallow DSL but does not have an implicit shape, the user will receive an unexpected implicit missing error message.

Listing 9 shows the type signatures of the `filter` operation. In order to guarantee that `f` produces a value that can be a boolean condition, `lifted.Query` adds an implicit `CanBeQueryCondition` parameter on the type of `T`. Slick-direct eliminates the need for this implicit parameter by forcing the method to be of type `T => Boolean`. The issue with this elimination is that query conditions on wrapped column types such as `Option[Boolean]`cannot be supported.

Listing 10 shows the type signatures of the `join` operation. This example may be a bit unfair against `lifted.Query`, but shows that type signatures in the lifted embedding can become unwieldy complicated. The type signature of the equivalent method in slick-direct is undeniably more user-friendly.

Listing 11 shows the type signatures operations on column types. A big benefit to the slick-direct API is that operations on column types such as equality of types and concatenation of strings is done with `==` and `+`, respectively. As this example exhibits, the lifted embedding requires queries to use the less widely adopted syntax `===` and `++` due to constraints of the Scala language. In fact, using `==` and `+` in the lifted embedding will not result in a compilation error but unexpected behavior instead. The equality will most likely evaluate to a false boolean literal column and the concatenation operation will evaluate to a literal string column with the string representation

Listing 10: Join API

```
// slick.lifted
def joinFull[E1 >: E, E2, U2, D[_], O1, U1, O2](q2: Query[E2, _, D])
  (implicit ol1: OptionLift[E1, O1],
            sh1: Shape[FlatShapeLevel, O1, U1, _],
            ol2: OptionLift[E2, O2],
            sh2: Shape[FlatShapeLevel, O2, U2, _]):
                BaseJoinQuery[O1, O2, U1, U2, C, E1, E2]
// slick.direct
def joinFull[T2, D[_]](q: Query[T2, D]): BaseJoinQuery[Option[T],
    Option[T2], T, T2, C]
```

Listing 11: Column extension methods API

```
// slick.lifted
for {
  user <- users
  car <- cars if car.ownerId === user.id
} yield user.name ++ " drives a " ++ car.name
// slick.direct
query {
  for {
    user <- users
    car <- cars if car.ownerId == user.id
  } yield user.name + " drives a " + car.name
}
```

of the runtime memory address of the column values.

One caveat of slick-direct queries, as seen in the previous listing, is that they need to be wrapped inside a *query* block. In the previous listing, the queries for slick-direct have identical shape as the lifted embedding except that they must passed to the slick-direct `query` macro. If queries are created outside a query block they will, as of now, fail with `NotImplementedError`. However, it should be possible provide a user friendly message explaining this peculiar requirement of slick-direct.

Due to time constraint, slick-direct currently only supports 6 categories of queries. Nevertheless, we consider that we have picked the methods that offer the strongest proof of concept that the Directembedding approach can used for Slick. We believe that the remaining methods that are not supported in our case study could be added to our API with a small additional effort. All supported queries in slick-direct have a corresponding test suite in the project's repository on Gihub, linked at the end of this report. We encourage the reader to look at the test suites for a more in-depth comparison between the two APIs.

3.3 Related work

A lot of work has been made to create a simplified Query API to Slick. Most of this work relies on Scala macros [2], like Directembedding. We cover a few of these attempts to see how they differ from slick-direct.

The first attempt was made by the Slick team and resulted in a direct embedding API which has now been deprecated and will be removed in the upcoming 3.1 release of Slick. The approach taken with the direct embedding API differs greatly from our approach in slick-direct. The Queryable API from the direct embedding implemented a separate macro for each method and produced values of type `ast.Node`, obviating the need for the `lifted.Query`. Slick-direct, on the other hand, implements one macro logic for all invocations on our Query API and produces values of type `lifted.Query`. Slick-direct does not implement any query logic, it delegates it to `lifted.Query`.

Another attempt to simplify the Slick Query API was a made by Amir Shaikhha [12] using the Yin-Yang Framework [6]. The approach taken in this second attempt is similar to the approach taken in our case in many ways. The library implements one macro to reify values in a shallow Query API. However, this second attempt produced values in a shadow embedding that operates on values of type `lifted.Query` at runtime. The main difference between our case study and this attempt is that Directembedding obviates the need for the shadow embedding by transforming in one step the values in the shallow embedding into the deep embedding.

slick-macros [4] is a Scala library to generate boilerplate definitions in the lifted embedding The library accomplishes this using macro annotations.

slick-macros offers an impressive amount of customization options to generate `Table[T]` definitions. The library even has an experimental DSL to configure the database model. However, the library still falls short with two regards in comparison with slick-direct. Firstly, macro annotations need to be used in a separate compilation unit from where they are invoked. Secondly, slick-macros still forces users to write queries on the liffted embedding Query API, which have complicated type signatures as we have seen in listings above.

4 Future research

There are two areas which require more work. Firstly, it would be interesting to see more Directembedding case studies on other domains besides slick. It may be Slick is not the best library to showcase the usefulness of embedding Deep DSLs. In particular, we are interested to how a deeply embedded DSL would be designed if Directembedding was kept in mind from the start. A DSL designed from scratch with Directembedding could give experience on the usefulness of the `@reifyAs` — which has limited use in slick-direct — and also showcase more usage of the new language virtualization feature. Secondly, although our initial experience with slick-direct is positive, we believe that the library requires more thorough evaluation to be considered as a viable alternative to the lifted embedding DSL. In particular, the following areas need a closer examination:

- The `compile` method is overloaded with 5 implementations.
- The type-provider for the `driver.Table` generates a class inside the query macro, which is redundant, slows down the compilation and introduces unnecessary complexity. It would be welcome to see a solution that uses the Slick's standard type providers.
- The `ProjectionProcessing` class implements a unnecessarily amount of logic, which might be unnecessary with Slick's standard type providers.

5 Conclusion

This report presented work on *Directembedding*, a Scala library to implement a thin user-friendly layer on top of an existing deeply embedded DSL. We believe that much progress has been made with the library, in particular with regards to the library's architecture and features. Moreover, we are proud to show our results with slick-direct, which with a very small amount of code was able to support a large feature set of Slick. However, we consider that much work is left to be done to the see the real usefulness of Directembedding. These areas, which we believe need more focus, are listed in Section 4.

The source code for the Directembedding and slick-direct libraries are freely available on Github[3][4].

References

[1] Kevin J. Brown et al. "A heterogeneous parallel framework for domain-specific languages". In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 89–100. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6113791 (visited on 05/31/2015).

[2] Eugene Burmako. "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming". In: *Proceedings of the 4th Workshop on Scala*. ACM, 2013, p. 3. URL: http://dl.acm.org/citation.cfm?id=2489840 (visited on 05/31/2015).

[3] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages". In: *Journal of Functional Programming* 19.05 (2009), pp. 509–543. URL: http://journals.cambridge.org/abstract_S0956796809007205 (visited on 02/16/2015).

[4] ebiznext. *slick-macros*. June 2014. URL: https://github.com/ebiznext/slick-macros (visited on 05/31/2015).

[5] Christian Hofer et al. "Polymorphic embedding of DSLs". In: *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 2008, pp. 137–148. URL: http://dl.acm.org/citation.cfm?id=1449935 (visited on 05/31/2015).

[6] Vojin Jovanovic et al. "Yin-yang: concealing the deep embedding of DSLs". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences-GPCE 2014*. ACM Press, 2014, pp. 73–82. URL: http://infoscience.epfl.ch/record/203432 (visited on 05/31/2015).

[7] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *ACM Sigplan Notices* 40.10 (2005), pp. 41–57. URL: http://dl.acm.org/citation.cfm?id=1094815 (visited on 05/31/2015).

[8] Martin Odersky et al. *The Scala language specification*. 2004. URL: http://www-dev.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf (visited on 05/31/2015).

[3] https://github.com/directembedding/directembedding
[4] https://github.com/olafurpg/slick-direct

[9] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits". In: *ACM Sigplan Notices*. Vol. 45. ACM, 2010, pp. 341–360. URL: http://dl.acm.org/citation.cfm?id=1869489 (visited on 05/31/2015).

[10] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". In: *Communications of the ACM* 55.6 (2012), pp. 121–130. URL: http://dl.acm.org/citation.cfm?id=2184345 (visited on 05/31/2015).

[11] Tiark Rompf et al. "Optimizing data structures in high-level programs: new directions for extensible compilers based on staging". In: *Acm Sigplan Notices*. Vol. 48. ACM, 2013, pp. 497–510. URL: http://dl.acm.org/citation.cfm?id=2429128 (visited on 05/31/2015).

[12] Amir Shaikhha. "An Embedded Query Language in Scala". Master Thesis. EPFL, Apr. 2014. URL: https://github.com/amirsh/master-thesis (visited on 05/31/2015).

[13] *Slick*. Apr. 2015. URL: http://slick.typesafe.com/ (visited on 05/31/2015).

[14] Josef Svenningsson and Emil Axelsson. "Combining deep and shallow embedding for EDSL". In: *Trends in Functional Programming*. Springer, 2013, pp. 21–36. URL: http://link.springer.com/chapter/10.1007/978-3-642-40447-4_2 (visited on 05/31/2015).