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 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

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

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. 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.

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.

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[2].

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. In this pass, the attached metadata 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 [1]. An example Directembedding DSL is provided the `example` package object.

[2] https://github.com/directembedding/directembedding

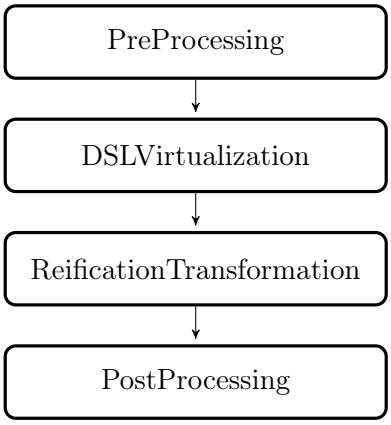

Figure 1: The Directembedding transformation pipeline.

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 [1] 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 the 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: DSL author and 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.

2.5.1 @reifyAsInvoked

The `@reifyAsInvoked` annotation is useful to create a front-end on top an existing deep EDSL. As will be covered in Section 3, the slick-direct uses this annotation to great extent.

2.5.2 @passThrough

The `@passThrough` annotation is useful when certain methods should not be reified in the ReificationTransformation. 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. A DSL author can annotate a method with `@passThrough` if she wants the functionality of the method to be preserved from the shallow DSL. Slick-direct uses this annotation in a few cases.

2.5.3 customLifts

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

```
1  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`

2.5.4 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

We evaluated Directembedding by implementing a front-end, *slick-direct*, for the Scala library *Slick*. We chose Slick for mostly two reasons. Firstly, Slick is a widely used library in the industry. Secondly, there have been made two previous attempts to provide a user-friendly front-end on top of the Slick lifted embedding.

Slick is a popular Scala library. Slick is the recommended relational mapper by Typesafe 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[3]. 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.

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

3.1 Related work

There have been made two attempts to provide a simplified Query API to Slick using macros.

- 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`.
- The second attempt was a made by Amir Shaikhha [2] using the Yin-Yang Framework [1]. 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.

Due to time constraint, we implemented only a subset of the methods supported in the previous two attempts. Nevertheless, we consider that we picked the methods that introduced the most interesting challenges for our purposes. We believe that the remaining methods that are not supported in our case study could be added to our API with little extra effort.

3.2 Comparison with lifted.Query

Slick-direct currently supports 6 categories of queries. Each of these categories has a test suite showing comparison of the slick-direct API compared to the lifted embedding in the Slick library. Herein, we highlight three major differences between the two APIs. In all examples, assume the type of `this` is `lifted.Query[E, T, _]` for `direct.Query[T, _]`, respectively.

Listing 1: Map API

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

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 not supporting selection of more than one column.

Listing 2: 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]
```

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 3: 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]
```

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

4 Future research

- Complete slick-direct, an opinionated front-end for slick. Features include: custom types for primary keys and encrypted strings, customizable type provider, and implement remaining API.
- Explore alternative uses of directembedding.

References

[1] 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 02/16/2015).

[2] Amir Shaikhha. "An Embedded Query Language in Scala". Master Thesis. EPFL, Apr. 2014. URL: https://github.com/amirsh/master-thesis (visited on 02/02/2015).