Yin-Yang: Concealing the Deep Embedding of DSLs

Demonstration: a DSL for in-memory querying

Vojin Jovanovic, Amir Shaikhha, and Manohar Jonnalagedda

EPFL, Switzerland
{first}.{last}@epfl.ch

Abstract. Deeply embedded domain-specific languages (EDSLs) inherently compromise programmer experience for improved program performance. Shallow EDSLs complement them by trading program performance for good programmer experience. We present Yin-Yang, a framework for DSL embedding that uses Scala reflection to reliably translate shallow EDSL programs to their corresponding deep EDSL programs. This automatic *translation* allows program prototyping and debugging in the user friendly shallow embedding, while the corresponding deep embedding is used in production—where performance is important. The reliability of the translation completely conceals the deep embedding from the DSL user. For the DSL author, Yin-Yang generates deep DSL embeddings from their shallow counterparts by reusing the program translation. This obviates the need for code duplication and assures that the implementations of the two embeddings are always synchronized.

Keywords: Embedded Domain-Specific Languages, Macros, Deep Embedding, Shallow Embedding, Compile-Time Meta-Programming

1 Introduction

External Domain-specific languages (DSLs) are languages with a custom compiler specialized to a particular application domain. Knowledge about the domain and specialization of the language give the compiler possibilities for optimization that do not exist in general purpose languages. A restricted language makes it easy for *DSL users* to learn the language while the optimizations can yield execution times close to hand-optimized programs [3].

The development of external DSLs includes building a parser and type checker as well as a large tool-chain (i.e., debuggers, IDEs, and documentation tools). An appealing alternative are *embedded DSLs* (EDSLs) [2] that reuse the parser, type checker and the tool-chain of a general-purpose *host language* to minimize required development effort. We can classify DSL embeddings into:

– *Shallow embeddings.* Values in the embedded language are *directly* represented by the values in the host language. A special sub-category of shallow

embeddings are *direct embeddings* where language constructs and term types are exactly the same as in the host language.
– *Deep embeddings.* Values in the embedded language are symbolically (with data structures) represented in the host language.

Direct embeddings are typically friendly for the *DSL user* as they *i) linguistically match* the host language and *ii)* can be easily debugged since values of the embedded language directly correspond to the values of the host language. However, since the structure of the programs is not completely known, the number of possible domain-specific optimizations is limited.

Deep embeddings reify the DSL programs into an *intermediate representation* (IR). This reification hinders usability as it often relies on complex type system constructs that create a linguistic mismatch with the host language. Furthermore, the IR construction in the programs disallows value inspection with classical debuggers and makes the problem even harder. However, the IR being domain specific opens new opportunities for optimization.

The main idea of Yin-Yang is to use *reflection* to translate programs written in an unmodified direct embedding into their deeply embedded counterparts. Since the fundamental difference between the interfaces of the two embeddings is in their types, we propose a generic translation between the two embeddings that is configurable with a separate *type translation*.

The translation forms the core of Yin-Yang, a generic framework for DSL embedding, that uses Scala's macros [1] to reliably translate direct EDSL programs into the corresponding deep EDSL programs. The virtues of the direct embedding are used during program development when performance is not important. The translation is then applied when performance is essential or alternative interpretations of a program are required. In effect, the translation completely conceals the deep embedding from the user.

To avoid error prone maintenance of synchronized direct and deep embeddings Yin-Yang reuses the core translation to generate the deep embeddings based on the definition of direct embeddings. Since the same translation is applied both for the EDSL definition and the EDSL program the equivalence between the embeddings is assured. With deep embedding generation DSL author can focus on the domain-specific optimizations while the interface is completely handled by Yin-Yang.

2 Demonstration: A DSL for In-Memory Queries

Our demonstration shows how Yin-Yang is used by: *i)* a DSL author and *ii)* a DSL user. The demonstration shows how the DSL user can be agnostic about concept of the deep embedding. Furthermore, the DSL author can program deep embeddings and program transformations with regular Scala. For the purpose of the demonstration we use a DSL for writing in-memory queries. The demonstration proceeds as follows:

DSL author: Writes a naïve Scala implementation of common query operators. An example operator is presented in the left part of Figure 1.

```
class SelectOp[A](parent: Oper[A])
  (pred: A => Boolean) extends Oper[A]{
  def open() = parent.open
  def next() = parent findFirst pred
  def reset() = parent.reset
  def close() = ()
}
```

```
def transform(s: Sym[Any]): Exp[Any] = s match {
  case sql"join($r, join($s, $t))" =>
    ql"join(join($r, $s), $t)"
  case sql"join(join($r, $s), $t)" =>
    ql"join($r, join($s, $t))"
  case _ => super.transform(s)
}
```

Fig. 1: A selection query operator (left) and the rules for `join` reordering (right).

DSL user: Can immediately write concise queries in the DSL. Although the query performance is satisfactory for prototyping, in production, the overhead is unacceptably large.

DSL author: To improve on the situation the author decides to provide a deep embedding for the query engine. All he needs to do is place a single Scala class annotation per query operator he implemented. The annotation marks that the annotated operator we would like to have the deep embedding generated. Yin-Yang then processes the annotated classes and generates the deep embedding that includes reification logic and code generation.

DSL user: Places queries in a DSL scope. After putting queries into the DSL scopes the user is still faced with the same type errors as before. The program execution errors can still be debugged in a standard debugger. In production, the DSL scope will invoke the translation and apply all standard compiler optimizations to the DSL yielding significant performance improvements.

DSL author: Decides to further improve performance by introducing the domain specific optimizations in the query language. The DSL author declares simple rewrite rules that the DSL compiler will later apply. The rewrite rules are specified through Scala quasi-quotes [4] and thus the DSL author can stay completely agnostic of the underlying data structures. The rules for `join` associativity are presented in the right part of Figure 1.

DSL user: Then measures performance of his DSL and realizes that the generated code performs on par with the hand-optimized version of the query.

References

1. Eugene Burmako. Scala macros: Let our powers combine! In *Proceedings of the 4th Annual Scala Workshop*, 2013.
2. Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.
3. Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 497–510, 2013.
4. Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for scala. Technical report, 2013.