Yin-Yang: Concealing the Deep Embedding of DSLs

No Author Given

No Institute Given

Abstract. Deep EDSLs intrinsically compromise programmer experience for improved program performance. Direct EDSLs complement them by trading program performance for good programmer experience.
We present Yin-Yang, a framework for DSL embedding that uses compile-time meta-programming to reliably translate direct EDSL programs to the semantically equivalent deep EDSL programs. The reliable translation allows program prototyping and development with the user friendly direct embedding, while the equivalent deep embedding is used where performance is important.
Yin-Yang greatly simplifies deep EDSL development and ensures the semantic equivalence of dual EDSL implementations. The core translation is used to generate the deep EDSLs implementation from their directly embedded counterparts. The deep EDSL generation greatly simplifies the design of the EDSL interface, Yin-Yang abstractions shield the EDSL authors from complicated compiler internals and allow simple domain-specific error-reporting at compile time.

Keywords: Embedded Domain-Specific Languages, Compile-Time Meta-Programming

1 Introduction

Domain-specific languages (DSLs) provide a restricted high-level and user-friendly interface crafted for a specific domain. The restriction of the language allows writing software with the high-level of abstraction, with extreme performance, and code portability in heterogeneous computing environments. However, restrictions in the language, often without justification, limit the user's expressiveness. Furthermore, implementation of a DSL requires building the parser, the type-checker, and the complete tool chain consisting of IDE plugins, debugging and documentation tools. This is a monumentally difficult undertaking that is often not justified by the benefits of a DSL. A promising alternative to DSLs are the embedded DSLs (EDSLs)[17] that are hosted in a general-purpose language and reuse its facilities. For the purpose of the following discussion, we distinguish between two main types of embeddings: *shallow* and *deep* embeddings.

– In a shallowly embedded DSL, values of the embedded language are *directly* represented by values in the host language. Consequently, terms in the host language that represent terms in the embedded language are evaluated directly into host-language values that represent DSL values. In other words, evaluation in the embedded language corresponds directly to evaluation in the host language.

– In a deeply embedded DSL, values of the embedded language are represented *symbolically*, that is, by host-language data structures, which we will refer to as the *intermediate representation (IR)*. Terms in the host language that represent terms in the embedded language are evaluated into the intermediate representation. An additional evaluation step is necessary to reduce the intermediate representation to a direct representation. This additional evaluation is typically achieved through *interpretation* of the IR in the host language, or through *code generation* and subsequent *execution*.

An important advantage of deep embeddings over shallow ones is that DSL terms can easily be manipulated by the host language. This enables domain-specific optimizations [25,26] that lead to orders-of-magnitude improvement in program performance, and multi-target code generation [7], that allows high-level programs to run on heterogeneous hardware platforms (e.g., clusters with GPUs).

On the other hand, shallow embeddings typically suffer from less *linguistic mismatch*: This is particularly obvious for a class of shallow embeddings that we will refer to as *direct* embeddings. Direct embeddings preserve the intrinsic constructs of the host language *on the nose*. That is, DSL constructs such as **if** statements, loops or function literals, as well as primitive data types such as integers, floating-point numbers or strings are represented directly by the corresponding constructs the host language.

Deep EDSLs intrinsically compromise the programmer experience by leaking their implementation details [11] cf. (§3). Firstly, the IR construction is achieved through complex type system constructs (e.g., type classes or implicit conversions). These make the EDSL interface hard to understand and lead to incomprehensible error messages during program development. Furthermore, debugging of programs is side-tracked by the IR construction and its optimization. Programmers cannot easily relate their programs to the code that is finally executed. In essence, the abstraction leaks greatly hinder programmer experience.

By creating an IR of the program Deep EDSLs provide means for domain-specific error reporting . For example, an EDSL might require constant data-structure sizes [34], or restrict functionality based on the back-end platform [2]. However, domain-specific error reporting in Deep EDSLs is achieved at run-time by analyzing the IR of the program . The analysis is easy for the EDSL author, but the users can only capture errors at run time. This can lead to bugs and prolongs the program development cycle. Compile-time domain-specific error reporting can also be achieved through the type system and macros . However, these approaches further complicate the interface and EDSL development so we do not elaborate on them.

Ideally, we could combine the high-performance, code portability, restricted language, and domain-specific error reporting of the deep EDSLs with the programmer friendly interface of the Direct EDSLs. This is difficult for two reasons: *i)* interfaces are intrinsically different so Direct EDSL programs are not valid Deep EDSL programs, and *ii)* semantically equivalent implementations of the dual EDSLs are hard to maintain.

Complementing the dual EDSL embeddings is a goal of numerous projects. They can be roughly classified in those that: *i)* require manual translation between dual embeddings [13,9], ii) use a modified shallow embedding, such that its programs are always valid in the deep embedding [1,6,32], or *iii)* simply hide the abstraction leaks of the Deep EDSLs [15,12,23] . However, to the best of our knowledge, none of the solutions completely conceal the abstraction leaks of the deep embedding and provide all of its benefits. We defer the full discussion to the related work section (§9).

The main idea of this paper is that users should use the unmodified direct embedding as a user-friendly interface. In addition language restrictions and domain-specific error reporting of the deep embedding should be visible at host language compile-time—during program development. Since the performance of Deep EDSL is relevant only in production, *after* the program has been developed, the presence of a deep embedding should be deferred until then. In other words, a direct embedding should be *translated* into its equivalent deep embedding.

We present Yin-Yang, a library for DSL embedding, that uses compile-time meta-programming to reliably transform Direct EDSL programs into the corresponding deep EDSL programs. The virtues of the direct embedding are used during program development when performance is not of importance. The translation is applied when performance is essential or alternative interpretations of a program are required (e.g. hardware generation).

The reliability of the translation completely assures that Direct EDSL programs will always be valid in the Deep EDSL. Furthermore, the translation prevents usage of language constructs unsupported by the Deep EDSL in the Deep EDSL. Finally, minimal maintenance overhead and semantic equivalence of dual EDSL implementations is achieved by automatic generation of the Deep EDSL implementation based on translating the Direct EDSL implementation. Deep EDSL generation tremendously simplifies the EDSL design.

Yin-Yang provides abstractions for EDSL authors that allow EDSL programs to be compiled at either host language run time, or at host language compile time. These abstractions allow compile-time error reporting for domain-specific analysis, and also completely hide the host-language compiler-internals from the DSL author, and support EDSLs that are interpreted and that generate code.

Yin-Yang is implemented in Scala. It uses macros [8] as a meta-programming construct to perform the direct-to-deep translation. Throughout the paper, we target LMS as the deep embedding back-end due to the plethora of existing EDSLs and its high-performance. Yin-Yang is applicable to other deep embedding back-ends (e.g. polymorphic embedding [16]) but, for simplicity, we omit them from discussion.

Yin-Yang contributes to the state of the art as:

– It completely conceals leaky abstractions of deep EDSLs from the users. The virtues of direct embedding are used for prototyping and the deep embedding enables high-performance in production. The reliable translation ensures that programs written in the direct embedding will always be correct in the deep embedding. The core translation is described in (§4).
– It simplifies the Deep EDSL development and guarantees semantic equivalence between the direct embedding and the deep embedding (§7). The core translation is re-used to translate the Direct EDSL definition into the Deep EDSL definition.
– It allows domain-specific error reporting at host language compile-time without requiring type-level computations and complicated macros. The EDSL author implements the domain-specific analysis in the host language by analyzing the EDSL IR. Error reporting is further described in (§5).
– It restricts the host language features in the Direct EDSL based on the supported features of the Deep EDSL. Specialized type checking of the translated Direct EDSL displays comprehensible error-messages to the user. Language restriction is further described in (§6.1).

We evaluate Yin-Yang by generating 3 EDSLs out of their direct embedding, and providing interface for 2 existing EDSLs. The effects of concealing the deep embedding and reliability a of the translation are evaluated on programs from EDSLs OptiGraph [30] and OptiML [31]. In all programs the direct implementation obviates deep embedding related type annotations. Complete evaluation is presented in (§8).

2 Background on Scala

In this section we provide information necessary for understanding Yin-Yang's implementation in Scala. We briefly explain Scala Macros [8] and Lightweight Modular Staging [25,26]. Throughout the paper we assume familiarity with the basics of the Scala Programming Language [20].

2.1 Scala Macros

Scala Macros [8] are a compile-time meta-programming feature of Scala. Macros are written as programs that operate on Scala abstract syntax trees (ASTs). They can construct new ASTs, or transform and analyze the existing Scala ASTs. Macro programs can use common functionalities of the Scala compiler like error-reporting, type checking, transformations, traversals, and implicit search. Currently, macro definitions are abstracted by the regular method bodies. The difference is that arguments to the macro function definitions are type-checked trees of the arguments.

Additionally, macros have access to the `eval[T](e: Expr[T]): T` function that will reflectively compile the expression `e`, load and run the compiled code, and return the result at compile time. We use this facility for instantiating

EDSLs to achieve compile-time domain-specific error reporting cf. (§5) and code generation cf. (§6.3).

For users, macro invocations are presented as regular method calls. The difference is that the body of the method operates on ASTs and will be executed at compile-time while its result is inlined at the invocation site. All the arguments to the method call are fully type-checked before macro expansion. An example of a macro that evaluates a *pow* function with constant arguments at compile-time is presented in Figure 1: the `evalPow` function defers its implementation to the `_evalPowImpl` macro. At compile-time, we check whether, under the compilation context `c`, the expression matches a call to the `pow` function, with literals as arguments. Note that we use quasiquotes (expressions inside `q""` strings) to abstract over compiler ASTs [28].

```
def evalPow(expr: Double): Double = macro _evalPowImpl
def _evalPowImpl(c: Context)(expr: c.Expr[Double]): c.Expr[Double] = {
  import c.universe._
  expr.tree match {
    case q"math.pow(${b: Literal}, ${e: Literal})" =>
      c.Expr(q"${c.eval[Double](expr)}")
    case _ => expr
  }
}
```

Fig. 1: Macro that evaluates a `pow` function at compile-time if arguments are literals.

In the event of a match, we evaluate the expression (using the `eval` function). Otherwise, we return the same expression. For clarity purposes:

- `evalPow(math.pow(x,2))` will remain the same.
- `evalPow(math.pow(3,2))` will be rewritten as `9`.

2.2 Deep Embedding of DSL with LMS

Lightweight Modular Staging (LMS) is a staging [33] framework and an embedded compiler for developing deep EDSLs. The LMS EDSL programs are written inside a scope of a *trait* that forms a *mix-in* composition of all EDSL components [21]. Each trait in the composition introduces EDSL declarations in the scope of the program. In Figure 2 we see the trait `VectorDSL` that defines a simplified EDSL for creating numerical vectors. In the mix-in composition, `Base` introduces core LMS constructs (e.g., `Rep[_]`), and `NumericOps` introduces the type class for numeric operations. The example usage of the EDSL is displayed in the bottom of the figure.

All types in the `VectorDSL` interface are a parametric type `Rep[_]`. ~~The `Rep[_]` is an abstract type member introduced to the scope by the trait `Base`.~~ The type `Rep[_]` abstracts over IR nodes of the Deep EDSL. It denotes that a term with type `Rep[T]` will evaluate to a term of type `T` after EDSL execution.

```
// The EDSL declaration
trait VectorDSL extends NumericOps with Base {
  object Vector {
    def fill[T:Numeric](value: Rep[T], size: Rep[Int]): Rep[Vector[T]] =
      vector_fill(value, size)
  }

  implicit class VectorOps[T:Numeric](v: Rep[Vector[T]]) {
    def +(that: Rep[Vector[T]]): Rep[Vector[T]] = vector_+(v, that)
  }
  // Operations vector_fill and vector_+ are elided
}

new VectorDSL { // EDSL program
  Vector.fill(lift(1),lift(3)) + Vector.fill(lift(2), lift(3))
} // returns a regular Scala Vector(3,3,3)
```

Fig. 2: Minimal EDSL for vector manipulation.

Operations on `Rep[T]` terms are added by implicit conversions [22]. For example, the implicit class `VectorOps` introduces the `+` operation on every term of type `Rep[Vector[T]]`. The type class `Numeric` ensures that vectors contain only numerical velues. This type class is introduced into the scope by the `NumericOps` trait.

The LMS framework is modular and each module is defined as a separate trait. The framework includes an embedded compiler that explicitly tracks effects and applies loop fusion, decomposition of structures, and code motion. Common compiler optimizations, like CSE and DCE, are performed on both high-level DSL abstractions and low level code. Depending on the code generation trait used, the output of the compilation can be Scala, C/C++, CUDA, or JavaScript code.

LMS has been successfully used by Kossakowski et al. for a JavaScript DSL [18], by Ackermann et al. [5] for distributed processing, and by Ureche et al. [34] for multi-dimensional array processing. Brown et al. present Delite [7,30], a heterogeneous parallel computing framework based on LMS. EDSLs developed with Delite cover domains of machine learning, graph processing, data mining, mathematics etc. Due to its wide use and high-performance we choose it as a back-end for Yin-Yang.

3 Motivation

The main idea of this paper is that EDSL users should program in a Direct EDSL, while the corresponding Deep EDSL should be used only in production. To motivate this idea, we consider a simple DSL for manipulating vectors, in both its direct and deep embedding forms.

Figure 3 shows a simple Direct EDSL for manipulating numerical vectors. Vectors are instances of a `Vector` class, and have only two operations, addition (the `+` operation, which can be used in infix position) and the higher-order `map` function, which applies a function `f` to each element of the vector. The `Vector` object provides factory methods for instantiation of vectors. Note that though the type of elements of a vector is generic, we require it to be an instance of the `Numeric` type class.

```
object Vector {
  def fromSeq[T: Numeric](seq: Seq[T]): Vector[T] = new Vector(seq)
  def fill[T: Numeric](value: T, size: Int): Vector[T] = fromSeq(Seq.fill(size)(value))
  def range(start: Int, end: Int): Vector[Int] = fromSeq(Seq.range(start, end))
}
class Vector[T: Numeric](val data: Seq[T]) {
  def map[S: Numeric](f: T => S): Vector[S] = Vector.fromSeq(data.map(x => f(x)))
  def +(that: Vector[T]): Vector[T] =
    Vector.fromSeq(data.zip(that.data).map(x => x._1 + x._2))
}
```

Fig. 3: The interface for the Direct EDSL for numerical vectors.

For a programmer ~~who has some basic Scala knowledge~~, this is an easy to use library. Not only can we write expressions such as `v1 + v2` for summing vectors (resembling mathematical notation), we can also get meaningful type error messages because the DSL is simply a library and follows the common patterns for the Scala libraries. Moreover, using the Scala debugger can help with runtime related issues.

The problem, however, is that code written in such a direct embedding suffers from major performance issues. These issues have been studied in detail elsewhere . For some intuition, however, consider the following code for adding 3 vectors: `v1 + v2 + v3`:

- each `+` operation creates an intermediate `Vector` instance.
- each `+` operation uses the `zip` function, which itself creates an intermediate `Seq` instance.
- each `+` operation calls a higher-order `map` function, which amounts to ~~a dispatch to a closure.~~

The abstractions of the language that allow us to write DSL-like code are also our downfall in terms of performance. We would have performed much faster with a simple while loop.

3.1 Deep Embedding

For the above case, there may be general compiler techniques which eliminate some of the overhead [10], but to properly exploit domain knowledge in general, and to potentially target other platforms, we have no choice but to have some form of node representation of our DSL program, which we analyze on its own merits. To this effect, we can lift our embedding to LMS (cf. (§??)). Figure 4 depicts a deep version of the numerical vector EDSL.

In the interface every method has an implicit parameter of type `SourceContext` and every generic type requires an additional `TypeTag` type class. The `SourceContext` is used for mapping generated code to the original program source while the

TypeTag is used to propagate run-time type information through the EDSL compilation. The run-time type information is used for generating code for statically typed target languages.

```
trait VectorDSL extends Base {
  object Vector {
    def fromSeq[T:Numeric:TypeTag](seq: Rep[Seq[T]])
      (implicit sc: SourceContext): Rep[Vector[T]] = vector_fromSeq(seq)
    def fill[T:Numeric:TypeTag](value: Rep[T], size: Rep[Int])
      (implicit sc: SourceContext): Rep[Vector[T]] = vector_fill(value, size)
    def range(start: Rep[Int], end: Rep[Int])
      (implicit sc: SourceContext): Rep[Vector[Int]] = vector_range(start, end)
  }

  implicit class VectorRep[T:Numeric:TypeTag](v: Rep[Vector[T]]) {
    def data(implicit sc: SourceContext): Rep[Seq[T]] = vector_data(v)
    def +(that: Rep[Vector[T]])(implicit sc: SourceContext): Rep[Vector[T]] =
      vector_plus(v, that)
    def map[S:Numeric:TypeTag](f: Rep[T] => Rep[S])
      (implicit sc: SourceContext): Rep[Vector[S]] = vector_map(v, f)
  }

  // Other IR construction methods and IR nodes are elided
  case class VectorFill[T:TypeTag](v: Rep[T], s: Rep[Int])(implicit sc: SourceContext)
  def vector_fill[T:Numeric:TypeTag](value: Rep[T], size: Rep[Int])
    (implicit sc: SourceContext): Rep[Vector[T]] = VectorFill(value, size) // IR node
}
```

Fig. 4: A LMS based Deep EDSL for manipulating numerical vectors.

3.2 Abstraction Leaks of Deep EDSLs

Not only is the deep embedding much more verbose (ie. harder for a DSL developer to implement), but it is also harder for a DSL user to use, because the implementation internals invariably leak into user space. Some of the main problems are:

Convoluted interfaces In comparison to the direct embedding our idealized EDSL from Figure 2 was exposing only the `Rep[_]` types to the user . However, once we introduce code generation the method signatures inevitably become very complex. This makes the interface very complicated to understand. The user of the EDSL, who might not be an expert programmer, needs to understand concepts like `TypeTag` and `SourceContext` to grasp the interface.

Difficult debugging The method bodies of the Direct EDSL contained concrete implementations. This allows users to trivially use debugging tools to inspect the program behavior while prototyping. With the Deep EDSL, method definitions contain the IR node instantiation. In the classical debugging mode this does not convey any useful information to the user. Furthermore, debugging generated code or an interpreter is extremely difficult. Users cannot relate the debugger position and the original line of code.

Complicated Type Errors The `Rep[_]` types leak to the user through type errors. Even for simple type errors the user will be exposed to the non-standard error messages. In certain cases (e.g., wrong overloading), the error messages can become completely incomprehensible. In Figure 5 we present a type error that appears when an invalid method is invoked. In specific cases, type-errors caused by the deep embedding span more than 10 lines and are completely incomprehensible.

```
found   : Int(1)
required: this.Rep[this.Vector[Int]]
     (which expands to)  this.Rep[vector.shallow.Vector[Int]]
     x + 1
       ^
```

Fig. 5: Visibility of the `Rep[_]` types in a typical type error.

Run-time domain-specific error reporting Let us change the requirements for our EDSL and require that all the vectors are created with constant size. Now, in the modified EDSL methods `fill` and `range` must accept only constants. In the Direct EDSL and the Deep EDSL this is not achievable at compile-time. In the Deep EDSL one can check if the values passed to the `fill` and `range` methods are the IR node that represents a constant and throw an exception. However, error reporting at the host language compile-time is not possible.

The described issues with Deep EDSLs greatly hinder the programmer experience. This is not specific to EDSLs implemented in Scala. Similar issues arise in other statically typed programming languages [14,32,29]. Furthermore, these problems among of the main disadvantages of deeply embedded EDSLs and prevent their usage by non-expert programmers.

3.3 Yin-Yang

The holy grail of embedded DSLs is to strike the balance between direct embeddings and deep embeddings:

- DSL users should be exposed only to the direct embedding.
- Domain-specific error reporting, possible due to the deep embedding, should be done at compile-time.
- Direct programs should be constrained by the capabilities of the deep embedding.

To achieve these goals, Yin-Yang requires existence of semantically equivalent EDSLs: one directly embedded, and the other deeply embedded (e.g., EDSLs from Figure 3 and Figure 4). Yin-Yang then provides a reliable translation from the direct to the deep embedding that can be triggered by a compilation flag.

The next section describes the core translation of Yin-Yang. Then we explain how the core translation can be used for restricting the host language features in the EDSL. Then we discuss how Yin-Yang achieves domain-specific error reporting and compile-time code generation. Finally, we show how the core translation of Yin-Yang can be used for maintenance of dual EDSL embeddings.

4 Translation of the Direct Embedding

The purpose of the core Yin-Yang translation is to make the transition from a directly embedded DSL program to its deeply embedded counterpart in a reliable and automated way. As seen in the previous sections, this transition is non-trivial for several reasons: *i)* host language constructs such as **`if`** statements are strongly typed and accept only primitive types for some of their arguments (e.g. a condition has to be of `Boolean` type), *ii)* (primitive) types in the direct embedding need to be translated into their IR counterparts (e.g. `Int` to `Rep[Int]`), *iii)* the directly embedded DLS operations need to be mapped onto their deeply embedded counterparts, and *iv)* methods defined in the deep embedding require additional parameters, such as run-time type information and source positions. Solutions for some of these issues have been described in previous work [9,16,24,26,1]. However, they typically fail to address some of the above issues or leak the details of the deep embedding into the direct one. We therefore propose a more straight-forward solution: a type-driven program translation from direct to deep embeddings implemented using Scala macros.

The translation is based on the idea of representing language constructs as method calls [9,24] and systematically mapping DSL operations and types of the direct embedding to their deep counterparts [9].

To illustrate the Yin-Yang translation, we will use a simple example program for calculating $\sum_{i=0}^{n} i^{exp}$ using the vector EDSL defined previously.Figure 6 contains three versions of the program: Listing 6a depicts the direct embedding version, Listing 6b represents the program after type checking (as the translation sees it), and Listing 6c shows the result of the translation.

The translation consists of two main steps:

Language virtualization converts host language intrinsics into function calls, which can then be evaluated to the appropriate IR values in the deep embedding.

EDSL Intrinsification converts DSL intrinsics (operations and types) from the direct embedding into their deep counterparts.

Language virtualization, as the name suggests, allows the basic constructs of a language, such as **`if`** statements or function abstraction, to be redefined. This can achieved by translating them into suitable method invocations [24]. Figure 6 illustrates this process: the **`if`** statement in Figure 6b is converted to a call to the `__if` method in figure 6c, and similarly, **`val`** is converted to `valDef`. In addition, note that we explicitly ascribe types to expressions and function calls, so as to avoid type inference in future stages of the translation.

In Yin-Yang we virtualize most of the Scala intrinsics that might be used to write direct DSL programs. This includes control flow constructs (e.g., **`if`**, **`while`**), function abstraction and application, and variable binding. Notable exceptions are class and trait definitions, including *case class* definitions (which correspond to algebraic data types in other languages), and pattern matching, although we are planning to add the later in future versions of Yin-Yang. Scala

```
import vect._; import math.pow;
val n = 100; val exp = 6;
optiVect {
  if (n > 0) {
    val v = Vector.range(0, n)
    v.map(x => pow(x, exp)).sum
  } else -1
}
```

(a) A program in direct embedding for calculating $\sum_{i=0}^{n} i^{exp}$.

```
import vect._; import math.pow;
val n = 100; val exp = 6;
optiVect {
  if (n > 0) {
    val v: Vector[Int] =
      vect.Vector.range(0, n)
    v.map[Int](x: Int =>
      math.`package`.pow(x, exp)
    ).sum[Int](math.Numeric.IntIsIntegral)
  } else -1
}
```

(b) The program from Listing 6a after desugaring and type inference.

```
import vect._; import math.pow;
val n = 100; val exp = 6;
new VectorDSL with IfOps with MathOps { def main() = {
  __if[Int](hole[Int](typeTag[Int], 0) > lift[Int](-1),{
     val v: Rep[Vector[Int]] = valDef[Vector[Int]](
       this.Vector.range(lift[Int](0), hole[Int](typeTag[Int], 0)) })
    v.map[Int](lam[Int, Int](x: Rep[Int] =>
       this.`package`.pow(x, hole[Int](typeTag[Int], 1))
     ).sum[Int](this.Numeric.IntIsIntegral)
  },{
    lift[Int](-1)
  }
)}
```

(c) The Yin-Yang translation of the program from Listing 6b.

Fig. 6: Transformation of an EDSL program for calculating $\sum_{i=0}^{n} i^{exp}$.

is designed such that the types `Any` and `AnyRef`, which reside at the top of the Scala class hierarchy, contain **`final`** methods: through inheritance, these methods are defined on all types. Hence, they need to be virtualized as well.

Figure 7 lists some of the translation rules used to virtualize the Scala intrinsics, with $[\![t]\!]$ denoting the translation of a term t. Note that method definitions[1] need to be translated into function abstractions in order to be virtualized.

$$\frac{\Gamma \vdash t : T_2}{[\![x : T_1 \Rightarrow t]\!] = \texttt{lam}[T_1, T_2](x : T_1 \Rightarrow [\![t]\!])} \qquad \frac{\Gamma \vdash t_1.f : [T_1](T_2 \Rightarrow T_3)}{[\![t_1.f[T_1](t_2)]\!] = \texttt{app}[T_2, T_3]([\![t_1]\!].f[T_1], [\![t_2]\!])}$$

$$[\![\textbf{def } f[T_1](x : T_2) : T_3 = t]\!] = \textbf{def } f[T_1] : (T_2 \Rightarrow T_3) = [\![x : T_2 \Rightarrow t]\!]$$

$$\frac{\Gamma \vdash t : T}{[\![\textbf{if}(t_1)\ t_2 \textbf{ else } t_3]\!] = \texttt{\_\_if}[T]([\![t_1]\!], [\![t_2]\!], [\![t_3]\!])}$$

Fig. 7: Translation rules for language virtualization.

[1] In Scala, the **`def`** keyword is used to define (possibly recursive) methods. This is similar to of the `let` and `let rec` constructs in other functional languages.

DSL Intrinsification maps directly embedded versions of the DSL intrinsics to their deep counterparts. The constructs that need to be converted are: *i)* DSL types, *ii)* DSL operations, *iii)* constant literals, and *iv)* captured variables in the direct program:

– The *type translation* maps every DSL type in the already virtualized term body to an equivalent type in the deep embedding. In other words, the type translation is a function `tpMap` on types. Note that this function is inherently DSL-specific, and hence needs to be configurable by the DSL author. We will investigate aspects of different type translation in more detail in (§4.1).
  Looking at our example in Figure 6, we see that not all instances of a given type are translated in a uniform fashion. In particular, types in type-argument position (e.g., `lam[Int, Int]`) are not modified, while the type of the function argument is transformed from `Int` to `Rep[Int]`. Hence, our type translation function `tpMap` needs to take the *context* of a type term into account when translating it. As an example, consider the following type mapping function, which corresponds to the translation in figure 6:

```
def tpMap(t: Type, c: TypeContext): Type = (t, c) match {
  case (_, TypeApply)    => t
  case (q"t1 => t2", _) => q"${tpMap(t1)}, _) => ${tpMap(t2)}"
  case (_, _)            => q"this.Rep[$t]"
}
```

  The `TypeContext` type enumerates the possible contexts where a type can appear. In our case it is only necessary to distinguish between the `TypeApply` context (representing a type in type application) and other contexts.

– The *operation translation* maps directly embedded versions of the DSL operations into equivalent deep embeddings. To this end, we define a function `opMap` on terms that returns deep versions for each shallowly embedded operation. In our example the function `vect.Vector.range` is translated to the **`this`**`.Vector.range`.
  For deep embeddings using LMS, or polymorphic embeddings [16] in general, this function simply injects operations into the scope of the Deep EDSL (i.e., by adding a suitable module prefix). Of course, other approaches, such as name mangling or context injection via additional function parameters, are also possible. In the current implementation of Yin-Yang, the `opMap` function is fixed to simply inject the **`this`** prefix, although this might change in the future.

– *Constants* can be intrinsified in the deep embedding in multiple ways. They can be converted to a method call for each constant (e.g. $[\![1]\!]$ =`_1`), type (e.g. $[\![1]\!] =$ `liftInt(1)`), or with a unified polymorphic function (e.g. $[\![1]\!] =$ `lift[Int](1)`). In the example, we use the polymorphic function approach for constants `-1` and `0`.

– *Free variables* are external variables captured by an Direct EDSL term. All that deep embedding knows about these terms is their type and that they will become available only during evaluation (i.e., interpretation or after code generation). Hence free variables need to be treated specially in by the translation, and the deep embedding needs to provide support for their evaluation. In our example, the free variables `n` and `exp` are replaced with appropriate calls to the polymorphic method `hole[T]`, which handles the evaluation of free variables in our Deep EDSL.

4.1 Alternative Type Translations

Having type translation as a function opens a number of possible deep embedding strategies. Alternative type translations can also dictate the interface of `lam` and `app` and other core EDSL constructs. Here we discuss the ones that we find useful in EDSL design:

Automatic Inlining. In high-performance EDSLs it is often desired to automatically inline all functions and to completely prevent dynamic dispatch in user code (e.g., storing functions into lists). This is achieved by translating function types of the form `A => B` in the direct embedding into `Rep[A] => Rep[B]` in the deep embedding (where `Rep` designates IR types). This approach is used by LMS [25] and Delite [7], and is also the one illustrated by our example above.

To ensure type-safety the translation must also be modified to ensure only primitive types in type parameter position. Without these restrictions, generic functions would not type-check in the deep embedding. In this case this is a desired behavior as it fosters high-performance code by avoiding dynamic dispatch.

Complete Abstraction. If we want to completely abstract over the host language expressions we can transform all types uniformly. This is achieved by a simple type function:

```
def tpMap(t: Type, c: TypeContext): Type = (t, c) match {
  case (_, TypeApply) => t
  case (_, _)         => q"this.Rep[$t]"
}
```

Simple Types. If Deep EDSL authors want to avoid complicated types (e.g., `Rep[T]`), the `tpMap` function can simply transform all types to the base type of all IR nodes (e.g., `Exp`). If one would like the complete freedom of types the `tpMap` can return the `Dynamic` type [4] that makes the behavior equivalent to dynamic languages.

Translation of Base Types. All previous translations ignored types in the type parameter position. The reason is that the `tpMap` function behaved like a higher-kinded type. If we would like to map some of the base types to something completely different those types need to be changed in the position of type application as well. This translation is used for EDSLs based on polymorphic embedding [16] that use **`this`**`.T` to represent type `T`.

With the previous translations the type system of the direct embedding was ensuring that the term will type-check in the deep embedding. With this translation one should be very careful about the correctness of the translation.

4.2 Restricting Host Language Constructs

Deep EDSLs alone have no means of restricting the usage of certain langauge constructs and features of the host language. This can lead to confusion, type-errors, and bugs. For example, in the deep embedding one could always use a **try/catch** construct like this:

```
new VectorDSL {
  try Vector.fill(-1,  100) catch { case _ => println("Error") }
}
```

However, the functionality of **try/catch** will be executed during IR construction time, and this would capture errors of domain-specific error reporting.

With the core translation we restrict the language features of the host language by simply omitting functions from the deep embedding. For example, if our deep embedding does not contain a function `lam` the user will not be able to define lambdas in the direct embedding.

5 Yin-Yang Work-Flow and Interface

Yin-Yang is implemented using Scala macros [8] and consists of a single configurable transformer that can be used in different EDSLs, the `YYTransformer`. The `YYTransformer` accepts *i)* a `dslTrait` holding the name of the EDSL (e.g, trait `vect.VectorDSL`), *ii)* a `tpMap` holding a function from Scala types to compiler trees, and *iii)* the `prototype` flag. A `YYTransformer` is defined once per EDSL. Its interface and an example usage is shown in Figure 8.

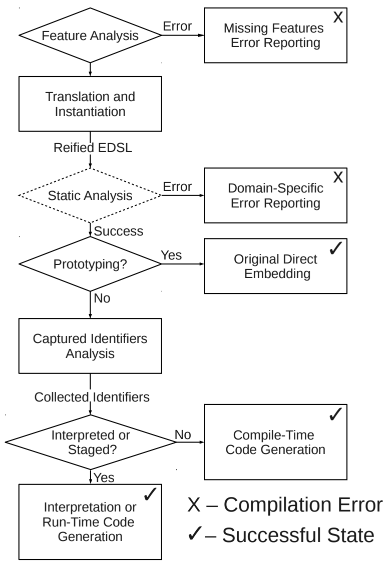
```
object YYTransformer {
  def apply[C <: Context, T](c: C)
    (dslTrait: String, tpMap: (c.Type, TypeContext) => c.Type, prototype: Boolean) = ...
}
def lmsTpMap(c: Context)(tpe: c.Type, c: TypeContext): c.Type = ...

def optiVect[T](dslBlock: => T): T = macro _optiVect
def _optiVect[T](c: Context)(dslBlock: c.Expr[T]): c.Expr[T] =
  YYTransformer[c.type, T](c, "opti.VectorDSL", lmsTpMap)(dslBlock)
```

Fig. 8: The `YYTransformer` factory method and its usage for the Vector EDSL.

The `YYTransformer` makes a series of decisions guided by the programmer, the EDSL author, and the program structure. The work-flow for the `YYTransformer` is shown in Figure 9a and the Yin-Yang interface used in the work-flow in Figure 9b. The workflow has the following phases:

Feature Analysis is a step for checking if language features unsupported by the deep embedding are used in the direct embedding. In (§4) we did the translation and left it to the type-checker to verify that all language constructs used in the direct embedding are valid in the deep embedding. Unfortunately, this does not yield comprehensible error messages.


Feature Analysis
Error
Missing Features Error Reporting
X
Translation and Instantiation
Reified EDSL
Static Analysis
Error
Domain-Specific Error Reporting
X
Success
Prototyping?
Yes
Original Direct Embedding
✓
No
Captured Identifiers Analysis
Collected Identifiers
Interpreted or Staged?
No
Compile-Time Code Generation
✓
Yes
Interpretation or Run-Time Code Generation
✓
X – Compilation Error
✓– Successful State

(a) Flowchart of the Yin-Yang operation. Dashed boxes are the optional phases.

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

trait BaseYinYang {
  abstract class LiftEvidence[T: TypeTag, R] {
    def hole(tpe:TypeTag[T], symbolId: Int): R
    def lift(v: T): R
  } // extended by the EDSL author

  final def hole[T,R](t: TypeTag[T], sym: Int)
    (implicit lftEv: LiftEvidence[T, R]): R =
    lftEv hole (t, sym)

  final def lift[T,R](v: T)
    (implicit lftEv: LiftEvidence[T, R]): R =
    lftEv lift (v)

  def requiredHoles(): List[Int] = Nil
}

trait Interpreted {
  def reset(): Unit
  def interpret[T: TypeTag](prms: Any*): T
}
trait CodeGenerator {
  def generateCode(clsNme: String): String
  def compile[T: TypeTag, Ret]: Ret
}
```

(b) The Yin-Yang interface for the EDSL author.

Here we analyze language features used in the direct embedding in a fine-grained manner and report comprehensible error messages to the user. For example, if a deep embedding does not support the **`try/catch`** construct the user would get a following message: *"VectorDSL does not support the try/catch construct."*. After this phase, if the deep embedding corresponds to the direct embedding, type checking will always succeed.

Translation to the Deep Embedding translates the direct program into the deep one (as described in (§4)). After translation, once the deep EDSL program is defined, Yin-Yang uses `eval[T](t: Expr[T])` to instantiate the EDSL component and reify the IR of the program. The EDSL instance with the reified IR is used to guide the rest of the work-flow.

Static Analysis optionally performs domain-specific analysis of the program. It is applied if the EDSL inherits the `StaticallyChecked` trait. The `staticallyCheck` method is then invoked on the EDSL instance. The method accepts a host language error reporter (`Reporter`) so it can report errors during host-language compilation. In case of errors, the domain-specific error messages are reported to the user at compile-time. For correctly reporting positions in the error message, the EDSL authors can use the source information passed to every method (e.g., `SourceContext`).

Prototyping checks if the EDSL is used in the prototyping mode—with the direct embedding. If so, Yin-Yang will simply return the unmodified block of the direct program. If run on large data-sets and in testing environments, we start the translation to the deep embedding. This decision is configured by the `prototype` configuration variable or by a compilation flag.

Captured Identifiers Analysis checks which captured variables (replaced by `holes`) are required for optimizations at run time. The EDSL analyzes its internal IR and decides which captured values are required for optimizations and returns them to Yin-Yang. If some variables are required for optimization the EDSL is executed at run time and the required values are converted to lifted values. If no variables are required, and the DSL can generate code, the EDSL code generation is performed at compile time. This analysis is described in (§6.2).

Interpreted or Staged is a decision step based on the EDSLs type and *captured identifiers*. If a EDSL inherits the `Interpreted` trait it can be interpreted at run time, and if it inherits the `CodeGenerator` trait it can generate code. Code generating EDSLs are executed at compile time when no captured variables are required for optimization. Interpreted EDSLs are always compiled at run time. The compile time code generation is explained in (§6.3), and run time code generation and interpretation in (§6.4).

The simplistic interface of Yin-Yang completely hides compiler internals from the EDSL author and at the same time allows compile-time error reporting and code generation. Achieving this with only macros would require great knowledge of the compiler internals while achieving it with just the deep embedding is impossible.

6 Yin-Yang Implementation

In this section we discuss the implementation decisions that we find interesting. We discuss specifics about feature analysis, compile-time code generation, and run-time code generation. We omit discussion about the implementation of the translation as its implementation resembles the specification from (§4).

6.1 EDSL Feature Analysis

A direct EDSL program can contain functions, and language constructs, that are not supported by the deep embedding, thus yielding programs that can not be translated. Firstly the virtualization of language constructs is performed to normalize all constructs to method calls. Then the analysis checks if every method in the direct embedding can be translated to the deep embedding. If it cannot, we report an error message containing the name of the method with its position in the program.

Our translation depends on host language type-checking making it hard to quickly check if a method invocation is valid. To check if an invocation exists in the deep embedding we must create a synthetic call to it, inside the EDSL component, and invoke the type checker. If the type checker succeeds the method is a valid part of the deep embedding. In case of failure the user is notified with an appropriate error message.

The consequence of *Feature Analysis* is that arbitrary Scala code cannot be combined with the EDSL code within the DSL program. This restriction is often desired in EDSLs but was not possible with previous approaches . In the future versions of Yin-Yang we will make this restriction optional.

Feature analysis is costly and slows down error reporting. Since it must be used with the shallow embedding, this can affect user experience. To improve performance, we first collect all method invocations in the program and build a distinct set of calls. Then we perform type checking in the deep embedding context on this set, significantly improving performance

6.2 Captured Identifiers

After the reflective instantiation with `eval`, and domain-specific static analysis, the method `requiredHoles` from the `BaseYinYang` component is invoked. This method is defined by the EDSL author and should perform analysis on the EDSL IR that define which captured identifiers can be used for optimization. If the EDSL does not support this feature it can return an empty collection for run-time EDSLs and return all holes in case of compile-time EDSLs.

This analysis allows the EDSLs which generate code to be optimized and compiled at either compile time, if none of the holes are required, or at run time, if run-time values are of interest. This analysis also allows a compilation stage choice per individual program. Code generation at compile time can significantly reduce run-time overheads induced by EDSL compilation, code generation, and host language compilation.

Once the required variables are acquired holes of these methods would be reverted to `lift` invocations. In the example from Figure 6c, the `hole` of the variable `exp` (with the symbol identifier 1) carries information about the exponent. This information can be used to optimize the `pow` function. In this case the EDSL would return a list containing the number `1`. In the following transformation, this hole would be replaced with a `lift` method for literals. This imposes that code generation and optimization must be done at run time.

6.3 Compile-Time Code Generation

In absence of run-time optimizations, code generating EDSLs can be completely compile at the host language compile time. To acquire generated code Yin-Yang invokes the `generateCode` method from the `CodeGenerator` interface. The EDSLs are required to return a string containing a class named by the passed argument (`clsName` which extends a Scala function. The number and the type of arguments of the function must match the number and types of holes captured by the direct EDSL. The `clsName` parameter is used by Yin-Yang to provide unique class names.

Then Yin-Yang parses, type-checks, and returns the generated class. The class, which extends a function, is invoked with the variables that were marked as holes. After code generation, the body of the shallow EDSL is completely discarded. Only the generated code is returned by the macro.

6.4 Run-Time EDSL Compilation

Run time compilation is a costly operation and therefore should be performed only when required. In order to minimize run-time overheads, Yin-Yang installs a guard statement before the EDSL recompilation. The guard checks if variables required for optimization have the same values as in the previous run. When the guard is satisfied, the pre-optimized EDSL program is executed, and when the guard fails the EDSL is re-optimized by calling the `reset` method from `Interpreted` trait. The new values for captured variables are populated prior to optimization/generation.

For run-time EDSL compilation, Yin-Yang returns the EDSL component containing the program, together with the invocation of the main method. If the EDSL is interpreted, the `interpret` method is invoked. In case of EDSLs that generate code the `compile` method is invoked to generate a new version of the program. Arguments passed to `interpret` and `compile` are the captured variables that are not required for optimization.

7 Automatic Deep EDSL Generation

So far, we have seen how Yin-Yang translates the *programs* written in Direct EDSLs to their deep counterparts. This makes the development task of an EDSL user easier, as he can still use the interface of direct embedding. However, the EDSL author still needs to maintain semantically equivalent implementations of both embeddings. This is a tedious and error prone task that greatly increases the development effort. For example, a change in the direct embedding must also be applied to the deep EDSL. This a repetitive task which can lead to bit rot and bugs.

We can automatically generate a deep embedding from the implementation of the direct embedding. This happens in two steps. First, we generate high-level IR nodes through a systematic conversion of methods written in a direct embedding to their corresponding methods in deep embedding (§7.1). This way a direct DSL program which uses the direct API, still uses the same direct API after core translation and code generation. In particular, calls to higher-order functions and other library methods have not been *lowered* yet. We make the observation that the implementation of a method is itself a direct DSL program. Thus, it can be transformed to its deep equivalent using the same core translation of Yin-Yang as the second step (§7.2). Then, we put the generated code in a trait which we call a lowering transformer.

The automatic code generation tool not only reduces boilerplate, but also generates traits in which the EDSL author can implement the domain-specific optimizations. In the event of a change in the direct embedding, these optimization traits are not re-generated; only the traits which correspond to the interface and IR nodes are modified.

7.1 Constructing High-Level IR Nodes

Every method has a corresponding high-level IR node in the deep EDSL. By using the Scala reflection API, we extract the method signatures. From these, we generate the interface, implementation, and code generation traits as prescribed by LMS. Figure 9 illustrates the way of defining IR nodes for `Vector` EDSL. The case classes in the `VectorOps` trait define the IR nodes for each method. The fields of these case classes are the callee object of the corresponding method (e.g., `v` in `VectorMap`), and the arguments of that method (e.g., `f` in `VectorMap`). Now we discuss how to generate such a trait, having the interface of direct embedding.

```
trait VectorOps extends SeqOps with NumericOps with Base {
  // elided implicit enrichment methods (e.g., Vector.fill(v, n) = vector_fill(v, n))

  // High level IR node definitions
  case class VectorPlus[T:Numeric](a: Rep[Vector[T]], b: Rep[Vector[T]])
    extends Rep[Vector[T]]
  case class VectorMap[T:Numeric,S:Numeric](v: Rep[Vector[T]], f: Rep[T] => Rep[S])
    extends Rep[Vector[S]]
  case class VectorFill[T:Numeric](value: Rep[T], size: Rep[Int])
    extends Rep[Vector[T]]
  case class VectorSeq[T:Numeric](data: Rep[Seq[T]]) extends Rep[Vector[T]]

  def vector_plus[T:Numeric](a: Rep[Vector[T]], b: Rep[Vector[T]]) = VectorPlus(a, b)
  def vector_map[T:Numeric, S:Numeric](v: Rep[Vector[T]], f: Rep[T] => Rep[S]) =
    VectorMap(v, f)
  def vector_fill[T:Numeric](value: Rep[T], size: Rep[Int]) = VectorFill(value, size)
  def vector_fromSeq[T:Numeric](seq: Rep[Seq[T]]) = VectorSeq(seq)
}
```

Fig. 9: High-level IR nodes for Vector.

So far, we assumed that there is no side-effect in EDSL. However, some methods are causing side-effects, and we have to keep track of them. The EDSL author must specify a method causing side-effect with an appropriate annotation. For side-effect tracking annotations, we use Scala FX [27]. Scala FX is a compiler plugin that adds an effect system to the type system of Scala. With Scala FX the regular Scala type inference also infers the effects of expressions. As a result, if the direct EDSL is using libraries which are already annotated, like the Scala collection library, then the EDSL author does not have to annotate the direct EDSL. Otherwise, there is a need for manual annotation of the direct embedding by the EDSL author. Finally, the Scala FX annotations are mapped to the corresponding effect construct in LMS.

Figure 10 shows how we automatically transform the I/O effect of `print` method to the appropriate construct in LMS. As the Scala FX plugin knows the effect of `System.out.println`, the effect for `print` method is inferred together with its result type (`Unit`). Based on the fact that the `print` method has an I/O effect, we wrap the high-level IR node creation method into `reflectSimple`, which is an effect construct in LMS to specify an I/O effect.

```
class Vector[T: Numeric](val data: Seq[T]) {
  def print() = {
    System.out.println(data)
  }
}
```

(a) Direct embedding for Vector with side-effects.

```
trait VectorOps extends SeqOps with NumericOps with Base {
  case class VectorPrint[T:Numeric](v: Rep[Vector[T]]) extends Rep[Vector[T]]

  def vector_print[T:Numeric](v: Rep[Vector[T]]) = reflectSimple(VectorPrint(v))
}
```

(b) Deep embedding for Vector with side-effects.

Fig. 10: Direct and deep embedding for Vector with side-effects.

7.2 Lowering High-Level IR Nodes to their Low-Level Implementation

Having domain-specific optimizations on the high-level representation is not enough for generating high performance code. In order to improve the performance, we must transform these high-level nodes into their corresponding low-level implementations. Hence, we must represent the low-level implementation of each method in the deep EDSL. After creating the high-level IR nodes and applying domain-specific optimizations, we transform these IR nodes into their corresponding low-level implementation. This is achieved by using a lowering phase as described below.

Figure 11 illustrates how the invocation of each method results in creating an IR node together with a lowering specification for transforming it into its low-level implementation. For example, whenever the method `fill` is invoked, `VectorFill` IR node is created like before. However, this high-level IR node needs to be transformed to its low-level IR nodes in the lowering phase. This delayed transformation is specified using `atPhase(lowering)` block. Furthermore, the low-level implementation uses constructs requiring deep embedding of other interfaces. In particular, the `fill` method requires the `Seq.fill` method that is available by the trait `SeqLowLevel`.

```
trait VectorLowLevel extends VectorOps with SeqLowLevel {
  // Low level implementations
  override def vector_fill[T:Numeric](v: Rep[T], s: Rep[Int]) =
    VectorFill(v, s) atPhase(lowering) {
      Vector.fromSeq(Seq.fill[T](s)(v))
    }
}
```

Fig. 11: Lowering to the low-level implementation for Vector.

Generating the low-level implementation is achieved by transforming the implementation of each direct embedding method. This is done in two steps. First, the expression given as the implementation of a method is converted to a Scala AST of the deep embedding by core translation of Yin-Yang. Second, the code represented by the Scala AST must be injected back to the corresponding trait. Thus, we implement Sprinter [3], a library that generates correct and human readable code out of Scala ASTs. The generated source code is used to represent the lowering specification of every IR node.

We also implement an alternative approach of using Forge [29] to generate the deep EDSL. Forge is a framework for specifying EDSLs. It accepts as input the specification of an EDSL and produces Delite [7,30] code for it. For each method in the direct embedding, we generate a Forge specification. The high-level IR node generation is handled by Forge. For providing the implementation of the deep method, we again use the Yin-Yang's core translation. We use this approach in our evaluation in order to have access to the DSLs that the Delite framework provides (e.g. OptiQL and OptiML).

8 Evaluation

To evaluate Yin-Yang we first check the correctness of our automatic deep EDSL generation for OptiML and OptiQL from Delite in (§8.1). For checking the correctness of Yin-Yang translations, we rewrote all applications from the test suite in the direct EDSL for OptiGraph and OptiML EDSLs from Delite. Then, we investigate the process and results of adopting the embedding on these EDSLs in (§8.2). Then, we evaluate complexity of Yin-Yang adoption for and existing EDSL Slick (§8.3). Slick is not based on LMS and does not have Yin-Yang compatible interfaces.

8.1 Automatic Deep EDSL Generation

We use Forge for generating deep EDSL from given direct EDSL. For checking the correctness of this component, we have tested OptiQL and OptiML DSLs from Delite. First, we have provided the direct EDSL for these two DSLs. Then we automatically generate Forge specification out of these direct interfaces. Finally, Forge generates corresponding deep EDSL code. The generated EDSL for both of these EDSLs was correct. On these EDSLs, we evaluate the number of lines of code for the direct EDSL, the generated Forge specification, and the generated deep EDSL. For generated Forge specification and generated deep EDSL, we only counted the parts which are generated out of the given direct EDSL.

OptiML is a DSL for machine learning based on Delite and LMS. To provide the direct EDSL for this EDSL, we modified the Forge to generate the direct EDSL from an existing specification. This generated direct EDSL consists of 5 main classes and contains appropriate annotations for side-effects. Then, we passed this direct EDSL into our deep EDSL generator. It produces the corresponding Forge specification for OptiML. After running Forge with this generated specification, the deep EDSL is generated. Table 1 shows the comparison of LOCs for direct EDSL, the subset of Forge specification, and deep EDSL which is generated. The generated deep EDSL consists of 5 times more LOC in comparison with the direct EDSL.

OptiQL is a query DSL similar to LINQ [19] based on Delite and LMS. Similarly to OptiML, we generated its direct EDSL, out of its Forge specification. The main type for OptiQL is `Table`, which is written in 73 LOC in the direct interface. We generate the Forge specification for this direct EDSL, which contains 74 LOC. This Forge specification leads to a deep EDSL, consisting of 526 LOC.

Vector is the EDSL shown as an example throughout this paper. The `Vector` class and its companion object consists of 70 LOC in the direct interface. The generated Forge specification for this EDSL contains 71 LOC. Forge generates the deep EDSL consisting of 369 LOC.

DSL	Direct	Forge	Deep
OptiML	1128	1090	5876
OptiQL	73	74	526
Vector	70	71	369

Table 1: LOC for direct EDSL, generated Forge specification, and generated deep EDSL

8.2 Correctness of Yin-Yang

To evaluate the correctness of Yin-Yang in Delite and LMS based EDSLs, we have implemented direct embedding for OptiGraph DSL, and used the generated direct EDSL for OptiML. In all of the DSLs, the transformations have been correct and the direct embedding required no type annotations. On based DSLs these DSLs, we evaluate the number of user provided annotations that were obviated, the number of lines of code needed to adopt Yin-Yang and the effort required.

OptiGraph is the DSL for efficient graph processing based on Delite and LMS. It contains 15 applications in its application suite, which have 991 lines of code. The original applications contained 57 `Rep[_]` annotations and 5 function calls for explicit lifting of literals (`unit(_)`). After conversion to Yin-Yang, the application suite had no annotations and the application code is exactly the same as the original. We also removed all the `println` and `assert` which do not require annotations. The application suite without those contains 492 lines of core code, with 48 of annotations and 5 `unit(_)` calls. Overall, 5% of lines contained annotations in the original version, and 12% of lines in the the applications with printing boilerplate removed.

To adopt Yin-Yang we have created a wrapper component for OptiGraph which contains additions for static-analysis at compile time, several abstract types, and implicit conversions. Excluding the implementation of the direct embedding, the porting to Yin-Yang contains 160 lines of code and required minimal effort.

For **OptiML**, we have implemented 6 machine learning applications. And for deep embedding, we have used our automatically generated deep EDSL. The core part of the applications in deep embedding had 257 LOC and contained 39 `Rep[_]` annotations. For direct embedding, we used the Forge generated direct EDSL. The direct applications contained no `Rep[_]` annotations.

8.3 Yin-Yang for Slick

Slick is a LINQ-like [19] DSL deeply embedded in Scala. It translates queries written in Scala into SQL queries portable across different databases. Slick does not use the mix-in composition to introduce DSL operations. However, Slick interfaces are far more complicated (e.g., complicated type parameters and implicits) than LMS.

To improve the complicated interface of Slick we designed the clean direct interface for it. The interface has dummy method implementations since semantics of different database back-ends can not be mapped to Scala. Thus, this interface is used only for type-checking and documentation. The complete interface contains only 70 lines of code.

Since, Slick has very irregular operations that do not correspond to the new shallow interface. And their redesign was not possible for compatibility reasons. We decided to add simple wrapper for it. This wrapper corresponds to it in a sense of the Yin-Yang translation. The whole wrapper contains only 240 lines of code.

After these two changes Slick has a user friendly interface based on Yin-Yang. The development of the deep Slick interface lasted for more than a year while the Yin-Yang wrapper only one month. This shows how Yin-Yang simplifies the front-end development of EDSLs. Furthermore, the captured variables analysis and guarded recompilation improved the performance of the query engine for 1-4x depending on the query. Yin-Yang interface compiles and reifies the query only once while the deep embedding does it on every execution. We have also implemented over a 100 test cases for the new front-end.

9 Related Work

Sujeth et al. [29] propose Forge, a Scala based meta-EDSL for generating both shallow and deep EDSLs from a single specification. To achieve this Forge relies on higher-kind abstract type members in Scala. The generated code for both shallow and the deep embedding uses only `Rep[T]` types and has almost identical code. However, in the shallow EDSL `Rep[T] = T` that results in direct program execution, while in the deep embedding `Rep[T] = Exp[T]` which constructs the IR of the program. This way Forge provides semantic equivalence of two embeddings and allows debugging of programs in the shallow embedding.

The downsides of Forge are that the shallow EDSL interface is convoluted by numerous `Rep` type annotations. Additionally, the type checking of programs in shallow and deep embedding is not equivalent to the deep one. This makes well typed shallow programs fail type checking obligating users to manually fix type errors in the deep program. Finally, Forge users must learn a new language for EDSL design and must follow constraints of the framework.

Svenningsson and Axelsson [32], identify that shallow embedding can complement the deep embedding. They propose to use a minimal core deep embedding with a complementing shallow interface that implements all the functionality in terms of the core embedding. They present how vectors, pairs, and other constructs can be represented in terms of the small core calculus. In their approach, generic types in the shallow interface must have a type class `Syntactic` which defines conversions from and to the deep embedding. The described technique is used in the Feldspar EDSL [6].

This approach requires the `Syntactic` type class and minimal number of types from the deep embedding in the interface. Presence of type class `Syntactic` leads to more complex type errors. With this scheme, debugging, EDSL compilation, and static analysis at compile time are impossible. Yin-Yang provides interfaces identical to the shallow embedding, and allows compilation both at run time and compile time, as well as prototyping with the shallow embedding.

Awesome Prelude [1] proposes replacing all primitive types in Haskell with type classes that either construct the IR or execute programs directly. Kansas Lava [13] exposes both shallow and deep embeddings and requires manual translation between them. Hereen et al. [15] and Plociniczak and Odersky [23] propose an interface for designing comprehensible type errors specific to a EDSL. Scaladoc [12] allows hiding of complicated function signatures in the documentation.

10 Conclusion

We presented Yin-Yang, a DSL embedding framework that completely conceals the rough interface of the deep embedded DSLs while still leveraging all their benefits. With our approach the EDSL users have a user-friendly interface, compile-time error reporting, and arbitrarily restricted language for their programs. In production the high-performance is achieved by running the deep version of the program.

With Yin-Yang the burden on the EDSL is greatly removed. The interface of their EDSL is a simple direct embedding. The

We are confident to claim that interface issues with the deep EDSL embedding are mostly resolved. Yin-Yang opens a question: Did the deep embedding come to life just because we did not have a macro system?

References

1. Awesome prelude, https://github.com/tomlokhorst/awesomeprelude.
2. Slick, http://slick.typesafe.com/.

3. Sprinter, http://vladimirnik.github.io/sprinter/.
4. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, 1991.
5. Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-toend Management of Big Data*, 2012.
6. Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckeg\aard, Anders Persson, Mary Sheeran, Josef Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, page 169–178, 2010.
7. K. J Brown, A. K Sujeeth, H. J Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, page 89–100, 2011.
8. Eugene Burmako. Scala macros: Let our powers combine! In *Proceedings of the 4th Annual Scala Workshop*, 2013.
9. Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. *Journal of Functional Programming*, 2009.
10. D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, volume 42, page 315–326, 2007.
11. Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, template haskell, and c++. *Domain-Specific Program Generation*, page 51–72, 2004.
12. Gilles Dubochet and Donna Malayeri. Improving API documentation for java-like languages. In *Evaluation and Usability of Programming Languages and Tools*, page 3, 2010.
13. Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing kansas lava. In *Implementation and Application of Functional Languages*, page 18–35. Springer, 2011.
14. 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*, OOPSLA '04, page 41–42, New York, NY, USA, 2004. ACM.
15. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *ACM SIGPLAN Notices*, volume 38, page 3–13, 2003.
16. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, page 137–148, 2008.
17. Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.
18. Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. JavaScript as an embedded DSL. *ECOOP 2012–Object-Oriented Programming*, page 409–434, 2012.
19. Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.
20. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification.* Citeseer, 2004.

21. Martin Odersky and Matthias Zenger. Scalable component abstractions. In *ACM Sigplan Notices*, volume 40, page 41–57, 2005.
22. Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, October 2010.
23. Hubert Plociniczak. Scalad: an interactive type-level debugger. In *Proceedings of the 4th Workshop on Scala*, page 8, 2013.
24. Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, page 1–43, 2009.
25. Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, June 2012.
26. 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.
27. Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP 2012–Object-Oriented Programming*, pages 258–282. Springer, 2012.
28. D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala. Technical report, Technical Report EPFL-REPORT-185242, EPFL, Lausanne, Switzerland, 2013.
29. A.K. Sujeeth, H.J. Lee, K.J. Brown, M. Odersky, and K. Olukotun. Forge. *Arxiv preprint arXiv:1109.0778*, 2013.
30. Arvind Sujeeth, Tiark Rompf, Kevin Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of ECOOP*, 2013.
31. Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML*, 2011.
32. Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. *Presentation at Trends in Functional Programming, June*, 2012.
33. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, page 203–217, 1997.
34. Vlad Ureche, Tiark Rompf, Arvind Sujeeth, Hassan Chafi, and Martin Odersky. StagedSAC: a case study in performance-oriented DSL development. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, page 73–82, 2012.