Embedding Staged Domain-Specific Languages

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

To my son David.

Acknowledgements

Lausanne, Switzerland, October 30th, 2015 V. J.

Abstract

Zusammenfassung

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

In this section we provide background information necessary for understanding Yin-Yang's implementation in Scala. We briefly explain the core concepts of Lightweight Modular Staging [Rompf and Odersky, 2012b, Rompf et al., 2013b] and Scala Macros [Burmako, 2013]. Throughout the paper we assume familiarity with the basics of the Scala Programming Language [Odersky et al., 2011].

1.1.1 Deep Embedding of DSLs with LMS

Lightweight Modular Staging (LMS) is a staging [Taha and Sheard, 1997] framework and an embedded compiler for developing deeply embedded DSLs. LMS provides a library of reusable language components organized as *traits* (Scala's first-class modules). An EDSL developer selects traits containing the desired language features, combines them through *mix-in* composition [Odersky and Zenger, 2005] and adds DSL-specific functionality to the resulting EDSL trait. EDSL programs then extend this trait, inheriting the selected LMS and EDSL language constructs. Figure 1.1 illustrates this principle. The trait `VectorDSL` defines a simplified EDSL for creating and manipulating vectors over some numeric type `T`. Two LMS traits are mixed into the `VectorDSL` trait: the `Base` trait introduces core LMS constructs and the `NumericOps` trait introduces the `Numeric` type class and the corresponding support for numeric operations. The bottom of the figure shows an example usage of the EDSL. The constant literals in the program are lifted to the IR through *implicit conversions* introduced by `NumericOps` [Oliveira et al., 2010].

All types in the `VectorDSL` interface are instances of the parametric type `Rep[_]`. The `Rep[_]` type is an abstract type member of the `Base` LMS trait and abstracts over the concrete types of the IR nodes that represent DSL operations in the deep embedding. Its type parameter captures the type underlying the IR: EDSL terms of type `Rep[T]` evaluate to host language terms of type `T` during EDSL execution.

```
// The EDSL declaration
trait VectorDSL extends NumericOps with Base {
  object Vector {
    def fill[T:Numeric]
      (v: Rep[T], size: Rep[Int]): Rep[Vector[T]] =
      vector_fill(v, 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(1,3) + Vector.fill(2,3)
} // returns a regular Scala Vector(3,6)
```

Figure 1.1 – Minimal EDSL for vector manipulation.

Operations on `Rep[T]` terms are added by implicit conversions that are introduced in the EDSL scope. For example, the implicit class `VectorOps` introduces the + operation on every term of type `Rep[Vector[T]]`. In the example, the type class `Numeric` ensures that vectors contain only numerical values.

LMS has been successfully used by in project Delite [Brown et al., 2011, Sujeeth et al., 2013a] for building DSLs that support heterogeneous parallel computing. EDSLs developed with Delite cover domains such as machine learning, graph processing, data mining, etc. Due to its wide use and high performance we choose Delite as a back-end for Yin-Yang.

1.1.2 Scala Macros

Scala Macros [Burmako, 2013] are a compile-time meta-programming feature of Scala. Macros 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 functionality of the Scala compiler like error-reporting, type checking, transformations, traversals, and implicit search.

Yin-Yang uses a particular flavor of Scala macros called *def macros*, though we will often drop the prefix “def” for the sake of brevity. From a programmer’s point of view, def macros are invoked just like regular Scala methods. However, macro invocations are *expanded* during compile time to produce new ASTs. Macro invocations are type checked both before and after expansion to ensure that expansion preserves well-typedness. Macros have separated declarations and definitions: declarations are represented to the user as regular methods while macro definitions operate on Scala ASTs. The arguments of macro method definitions are the type-checked ASTs of the macro arguments.

For DSLs based on Yin-Yang we use a macro that accepts a single block of code as its input. At compile time, this block is first type checked against the interface of the direct embedding. Then, Yin-Yang applies the generic transformation to translate the directly embedded AST to the corresponding deeply embedded AST. For example, given the following DSL snippet, Yin-Yang produces the `VectorDSL` program in Figure 1.1:

```
vectorDSL {
  Vector.fill(1,3) + Vector.fill(2,3)
}
```

2 Support for Embedding Domain-Specific Language

Domain-specific languages (DSLs) provide a restricted, high-level, and user-friendly interface crafted for a specific domain. Restricting the language to a particular domain allows programming at a high level of abstraction while retaining good run-time performance by leveraging domain knowledge for optimized code generation or interpretation. In certain cases, code can even be targeted at heterogeneous computing environments [Rompf et al., 2013b].

The implementation of a usable *external* (or *stand-alone*) DSL requires building a parser, type-checker, and possibly a complete tool chain consisting of IDE integration, debugging, and documentation tools. This is a great undertaking that is often not justified by the benefits of having an external DSL. A promising alternative to external DSLs are *embedded DSLs* (EDSLs) [Hudak, 1996] which 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 refer to as the *intermediate representation (IR)*. Terms in the host language that represent terms in the embedded language are evaluated into this 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 be easily manipulated by the host language. This enables domain-specific optimizations [Rompf and Odersky, 2012b, Rompf et al., 2013b] that lead to orders-of-magnitude improvements in program performance, and multi-target code generation [Brown et al., 2011].

On the other hand, shallow embeddings typically suffer less from *linguistic mismatch*: this is particularly obvious for a class of shallow embeddings that we 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 of the host language.

Deep EDSLs intrinsically *compromise programmer experience* by leaking their implementation details (§2.1.2). Often, IR construction is achieved through complex type system constructs that are, inevitably, visible in the EDSL interface. This can lead to cryptic type errors that are often incomprehensible to DSL users. In addition, the IR complicates program debugging as programmers cannot easily relate their programs to the code that is finally executed. Finally, the host language often provides more constructs than the embedded language and the usage of these constructs can be undesired in the DSL. If these constructs are generic in type (e.g., list comprehensions or `try\catch`) they can not be restricted in the embedded language by using complex types (§2.1.2).

Ideally, we would like to complement the high performance of deeply embedded DSLs, along with their capabilities for multi-target code generation, with the usability of their directly embedded counterparts. Reaching this goal turns out to be more challenging than one might expect: let us compare the interfaces of a direct embedding and a deep embedding of a simple EDSL for manipulating vectors[1]. The direct version of the interface is declared as:

```
trait Vector[T] {
  def map[U](fn: T => U): Vector[U]
}
```

The interface of the deep embedding, however, fundamentally differs in the types: while the (polymorphic) `map` operation in the direct embedding operates directly on values of some generic type `T`, the deep embedding must operate on whatever intermediate representations we chose for `T`. For our example, we chose the abstract, higher-kinded type `Rep[T]` to represent values of type `T` in the deep embedding:

```
trait Vector[T] {
  def map[U](fn: Rep[T => U]): Rep[Vector[U]]
}
```

The difference in types is necessarily visible in the signature and thus inevitably leaks into user programs. This might seem like a low price to pay for all the advantages offered by a deep embedding. However, as we will see in §2.1.2, this difference in types is at the heart of many of the inconveniences associated with deep embeddings. How then, can we conceal

[1] All code examples are written in *Scala*. Similar techniques can be applied in other statically typed languages. Cf. [Carette et al., 2009, Lokhorst, 2012, Svenningsson and Axelsson, 2013].

this fundamental difference?

In Forge [Sujeeth et al., 2013b], Sujeeth et al. propose maintaining two parallel embeddings, shallow and deep, with a single interface equivalent to the deep embedding. In the shallow embedding, `Rep` is defined to be the identity on types, that is `Rep[T] = T`, effectively identifying IR types with their direct counterparts. As a result, shallowly embedded programs may be executed directly to allow for easy prototyping and debugging. In production, a simple “flip of a switch” enables the deep embedding. Unfortunately, artifacts of the deep embedding still leak to the user through the fundamentally “deeply typed” interface. We would like to preserve the idiomatic interface of the host language and completely conceal the deep embedding.

The central idea of this paper is the use of *reflection* to convert programs written in an unmodified direct embedding into their deeply embedded counterparts. Since the fundamental difference between the interfaces of the two embeddings resides in their types, we employ a configurable *type translation* to map directly embedded types T to their deeply embedded counterparts $[\![T]\!]$. For our motivating example the type translation is simply:

$$\begin{array}{lcll} [\![T]\!] & = & T & \text{if } T \text{ is in type argument position,} \\ [\![T]\!] & = & \texttt{Rep}[T] & \text{otherwise.} \end{array}$$

In §2.2 we describe this translation among several others and discuss their trade-offs.

Together with a corresponding translation on terms, the type translation forms the core of Yin-Yang, a generic framework for DSL embedding, that uses Scala’s macros [Burmako, 2013] to reliably translate directly embedded DSL programs into corresponding deeply embedded DSL 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., for hardware generation). 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.

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

- It completely conceals leaky abstractions of deep EDSLs from the users. The virtues of the direct embedding are used for prototyping, while 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 §2.2.
- It restricts 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 §2.3.

- It simplifies deep EDSL development and guarantees semantic equivalence between the direct embedding and the deep embedding by reusing the core translation to generate the deep EDSL definition out of the direct EDSL definition (§2.4).

We evaluate Yin-Yang by generating 3 deep EDSLs from their direct embedding, and providing interfaces for 2 existing EDSLs. The effects of concealing the deep embedding and reliability of the translation were evaluated on 21 programs (1284 LOC), from EDSLs OptiGraph [Sujeeth et al., 2013a] and OptiML [Sujeeth et al., 2011]. In all programs combined the direct implementation obviates 101 type annotations related to the deep embedding. The complete evaluation is presented in §2.5.

Throughout the paper, we target the LMS [Rompf and Odersky, 2012b] framework as a deep embedding back-end due to the plethora of existing LMS EDSLs. Consequently, we will assume that deep embeddings use LMS' extensible IR. However, Yin-Yang is applicable to other types of IR (e.g., polymorphic embeddings [Hofer et al., 2008]) and possibly other statically typed languages (§2.6).

2.1 Motivation

The main idea of this paper is that EDSL users should program in a direct embedding, while the corresponding deep embedding should be used only in production. To motivate this idea we consider the direct embedding and the deep embedding of a simple EDSL for manipulating vectors. Here, we use Scala to show the problems with the deep embedding that apply to other statically typed programming languages (e.g., Haskell and OCaml). These languages achieve the embedding in different ways [Carette et al., 2009, Guerrero et al., 2004, Lokhorst, 2012, Svenningsson and Axelsson, 2013], but this is always reflected in the type signatures. In the context of Scala, there are additional problems with type inference and implicit conversions, however, we omit those from the discussion as language specific.

Figure 2.1 shows a simple direct EDSL for manipulating numerical vectors. Vectors are instances of a `Vector` class, and have only two operations: *i)* vector addition (the `+`), and *ii)* the higher-order `map` function which applies a function `f` to each element of the vector. The `Vector` object provides factory methods `fromSeq`, `range`, and `fill` for vector construction. Note that though the type of the elements in a vector is generic, we require it to be an instance of the `Numeric` type class.

For a programmer, this is an easy to use library. Not only can we write expressions such as `v1 + v2` for summing vectors (resembling mathematical notation), but we can also get meaningful type error messages. The EDSL is an idiomatic Scala and displayed type errors are comprehensible. Finally, in the direct embedding, all terms directly represent values from the

```
object Vector {
  def fromSeq[T: Numeric](seq: Seq[T]): Vector[T] =
    new Vector(seq)
  def fill[T: Numeric](v: T, size: Int): Vector[T] =
    fromSeq(Seq.fill(size)(v))
  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))
}
```

Figure 2.1 – The interface of a direct EDSL for manipulating numerical vectors.

embedded language and inspecting intermediate values with the debugger is straightforward.

The problem, however, is that the code written in such a direct embedding suffers from major performance issues [Rompf et al., 2013b]. For some intuition, consider the following code for adding 3 vectors: `v1 + v2 + v3`. Here, each + operation creates an intermediate `Vector` instance, uses the `zip` function, which itself creates an intermediate `Seq` instance, and calls a higher-order `map` function. The abstractions of the language that allow us to write code with high-level of abstraction have a downfall in terms of performance. Consecutive vector summations would perform much better if they were implemented with a simple while loop.

2.1.1 The Deep Embedding

For the DSL from Figure 2.1, the overhead could be eliminated with optimizations like stream fusion [Coutts et al., 2007] and inlining, but to properly exploit domain knowledge, and to potentially target other platforms, one must introduce an intermediate representation of the EDSL program. The intermediate representation can be transformed according to the domain-specific rules (e.g., eliminating addition with a null vector) to improve performance beyond common compiler optimizations [Rompf et al., 2013b]. To this effect, we use the LMS framework and present the deep version of the EDSL for manipulating numerical vectors in Figure 2.2.

In the `VectorDSL` interface every method has an additional implicit parameter of type `SourceContext` and every generic type requires an additional `TypeTag` type class. The `SourceContext` contains information about the current file name, line number, and character offset. `SourceContexts` are used for mapping generated code to the original program source. `TypeTags` carry all information about the type of terms. They are used to propagate run-time type information through the EDSL compilation for optimizations and generating code for statically typed target languages. In the EDSL definitions the `SourceContext` is rarely used explicitly (i.e., as an argument). It is provided “behind the scenes” by implicit

```
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)
  }

  // IR constructors for ‘map‘ and ‘plus‘ are elided
  case class VectorFill[T:TypeTag]
    (v: Rep[T], s: Rep[Int])
    (implicit sc: SourceContext)
  def vector_fill[T:Numeric:TypeTag]
    (v: Rep[T], size: Rep[Int])
    (implicit sc: SourceContext): Rep[Vector[T]] =
    VectorFill(v, size) // IR node construction
}
```

Figure 2.2 – A LMS based deep EDSL for manipulating numerical vectors.

definitions that are provided in the DSL.

2.1.2 Abstraction Leaks in the Deep Embedding

The programs in the deep embedding construct the IR instead of the values in the embedded language. This inevitably leaks to the users in the following ways:

Convoluted interfaces. The interface of the EDSL has `Rep[_]` types in all its method signatures. Furthermore, once we introduce code generation, the method signatures must be enriched with source and type information (`SourceContext` and `TypeTag`) and inevitably become 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. In the methods of the direct EDSL all terms directly represent values in the embedded language (there is no intermediate representation). This allows users to trivially use debugging tools to step through the terms and inspect the values of the embedded language. With the deep EDSL, method definitions only instantiate the IR nodes. 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 is exposed to non-standard error messages. In certain cases (e.g., incorrect call to an overloaded function), the error messages can become hard to understand. To illustrate, we present a typical type error for invalid method invocation:

```
found   : Int(1)
required: Vector[Int]
     x + 1
         ^
```

In the deep embedding the corresponding type error contains `Rep` types and the `this` qualifier:

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

This example represents one of the most common type errors. For more complicated type errors cf. [Jovanovic et al., 2014b].

Unrestricted host language constructs. In the deep embedding all generic constructs of a host language can be used arbitrarily. For example, `scala.List.fill[T](count: Int, el: T)` can, for the argument `el`, accept both direct and deep terms. This is often undesirable as it can lead to code explosion and unexpected program behavior.

In the following example, assume that generic methods `fill` and `reduce` are not masked by the `VectorDSL` and belong only to the host language library. In this case, the invocation of `fill` and `reduce` performs meta-programming over the IR of the deep embedding:

```
new VectorDSL {
  List.fill(1000, Vector.fill(1000,1)).reduce(_+_)
}
```

Here, at DSL compilation time, the program creates a Scala list that contains a thousand IR nodes for the `Vector.fill` operation and performs a vector addition over them. Instead of producing a small IR the compilation result is a thousand IR nodes for vector addition. This is a typical case of code explosion that could not happen in the direct embedding which does not introduce an IR.

On the other hand, some operations can be completely ignored. In the next example, the `try/catch` block will be executed during EDSL compilation instead during DSL program execution:

```
new VectorDSL {
  try Vector.fill(1000, 1) / 0
  catch { case _ => Vector.fill(1000, 0) }
}
```

Here, the resulting program always throws a `DivisionByZero` exception.

2.2 Translation of the Direct Embedding

The purpose of the core Yin-Yang translation is to reliably and automatically make a transition from a directly embedded DSL program to its deeply embedded counterpart. The transition requires a translation for the following 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 type `Boolean`), *ii)* all types in the direct embedding need to be translated into their IR counterparts (e.g., `Int` to `Rep[Int]`), *iii)* the directly embedded DSL 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. To address these inconsistencies we propose a straightforward solution: a type-directed program translation from direct to deep embeddings.

Since the translation is type-directed it requires reflection that supports *type introspection* and *type transformation*. The translation is based on the idea of representing language constructs as method calls [Carette et al., 2009, Rompf et al., 2013a] and systematically intrinsifying direct DSL operations and types of the direct embedding to their deep counterparts [Carette et al., 2009]. The translation operates in 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.

To illustrate the core translation, we use an example program for calculating $\sum_{i=0}^{n} i^{exp}$ using the vector EDSL defined in Figure 2.1. Figure 2.3 contains three versions of the program: Figure 2.3a depicts the direct embedding version, Figure 2.3b represents the program after type checking (as the translation sees it), and Figure 2.3c shows the result of the translation.

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

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

```
val n = 100; val exp = 6;
vectorDSL {
  if (n > 0) {
    val v: Vector[Int] =
      vector.Vector.range(0, n)
    v.map[Int](x: Int =>
      math.‘package‘.pow(x, exp)
    ).sum[Int](math.Numeric.IntIsIntegral)
  } else 0
}
```

(b) The original program after desugaring and type inference.

```
val n = 100; val exp = 6;
new VectorDSL with IfOps
  with MathOps { def main() = {
  ifThenElse[Int](
    hole[Int](typeTag[Int], 0) > lift[Int](0),{
    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](0)
  }
)}
```

(c) The Yin-Yang translation of the program from Figure 2.3b.

Figure 2.3 – Transformation of an EDSL program for calculating $\sum_{i=0}^{n} i^{exp}$.

Language virtualization allows to redefine intrinsic constructs of the host language, such as `if` and `while` statements. This can be achieved by translating them into suitable method invocations as shown by Rompf et al. in the modified Scala compiler named Scala-Virtualized [Rompf et al., 2013a].

Yin-Yang follows the same approach as Scala-Virtualized but uses macros of unmodified Scala to virtualize Scala intrinsics required to write direct DSL programs. In addition to Scala-Virtualized we virtualize function definition and application, and variable binding. Furthermore, Scala 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 making it impossible to override their functionality. Since Yin-Yang uses unmodified Scala we must virtualize all methods on `Any` and `AnyRef`.

Figure 2.3 illustrates this process and Figure 2.4 lists a subset of the translation rules used to virtualize the Scala intrinsics, with $[\![t]\!]$ denoting the translation of a term t. Note that method definitions[2] need to be translated into function definitions in order to be virtualized. In all expressions the original types are introspected and used as a type argument of the virtualized method. These generic types are translated later during the DSL intrinsification phase in order to avoid type inference in future stages of the translation.

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

$$\frac{\Gamma \vdash t : T_2}{[\![x : T_1 \Rightarrow t]\!] = \texttt{lam}[T_1, T_2](x : T_1 \Rightarrow [\![t]\!])}$$

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

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

$$[\![\textbf{val}\, x : T = t]\!] = \textbf{val}\, x : T = \texttt{valDef}[T](t)$$

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

Figure 2.4 – Subset of rules for language virtualization.

Constructs that are not virtualized are class and trait definitions, including *case class* definitions, and pattern matching. We are planning to add the latter in future versions of Yin-Yang.

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 on types. Note that this function is inherently DSL-specific, and hence needs to be configurable by the DSL author. We discuss aspects of different type translation in more detail in §2.2.1.

  The type mapping depends on the input type and the context. In practice, we need only distinguish between types in type-argument position, e.g. the type argument `Int` in the polymorphic function call `lam[Int, Int]`, and the others. To this end, we define a pair of mutually recursive functions $\tau_{\texttt{arg}}, \tau : \mathbb{T} \rightarrow \mathbb{T}$ where $\mathbb{T}$ is the set of all types and $\tau_{\texttt{arg}}$ and τ translate types in argument and non-argument positions, respectively.

- The *operation translation* maps directly embedded versions of the DSL operations into corresponding deep embeddings. To this end, we define a function `opMap` on terms that returns deep operation for each directly embedded operation. For deep embeddings based on LMS, or polymorphic embeddings [Hofer et al., 2008] in general, `opMap` simply injects operations into the scope of the deep EDSL (i.e., by adding the prefix `this`). Of course, other approaches, such as name mangling or importing definitions from a different module, 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.

  In Figure 2.3, calls to `range` on the object `vector.Vector` and `pow` on the package object `math.‘package‘` are respectively translated to calls `range` and `pow` on `this.Vector` and `this.‘package‘`. For simplicity, passing source information (`SourceContext`) and type information `TypeTag` is handled implicitly by the Scala compiler. In absence of implicit parameters they should be handled by the translation.

- *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 the constants.

- *Free variables* are external variables captured by a 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 execution after code generation). Hence, free variables need to be treated specially by the translation and the deep embedding needs to provide support for their evaluation. In Figure 2.3, the free variables `n` and `exp` are replaced with calls to the polymorphic method `hole[T]`, which handles the evaluation of free variables in the deep embedding. Each captured identifier is assigned with a unique number that is, together with type information, passed as an argument to the `hole` method (0 and 1 in Figure 2.3). The identifiers are later sorted and passed as arguments to the Scala function that is a result of EDSL compilation. The DSL author is required to ensure that the position and the type of the resulting function matches the order and types of the sorted identifiers passed by Yin-Yang.

2.2.1 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:

The identity translation. If we choose τ to be the identity function and virtualization methods such as `lam`, `app` and `ifThenElse` to be implemented in the obvious way using the corresponding Scala intrinsics, the resulting translation will simply yield the original, directly embedded DSL program.

Generic polymorphic embedding. If instead we choose τ to map any type term T (in non-argument position) to `Rep[`T`]`, for some abstract, higher-kinded IR type `Rep` in the deep EDSL scope, we obtain a translation to a *finally-tagless, polymorphic* embedding [Carette et al., 2009, Hofer et al., 2008]. For this embedding, the translation functions are defined as:

$$\tau_{\texttt{arg}}(T) = T$$
$$\tau(T) \quad = \texttt{Rep}[T]$$

By choosing the virtualized methods to operate on the IR-types in the appropriate way, one obtains an embedding that *preserves well-typedness*, irrespective of the particular DSL it implements. We will not present the details of this translation here, but refer the interested reader to [Carette et al., 2009].

Eager inlining. In high-performance EDSLs it is often desired to eagerly 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` again designates IR types). Instead of constructing an IR node for function application, such functions reify the whole body of the function starting with IR nodes passed as arguments. The effect of such reification is equivalent to inlining. This function representation is used in LMS [Rompf and Odersky, 2012b] by default and we use it in Figure 2.3. The translation functions are defined as:

$$\begin{aligned}
\tau_{\texttt{arg}}(T[I_1,\ldots,I_n]) &= T[\tau_{\texttt{arg}}(I_1),\ldots,\tau_{\texttt{arg}}(I_n)]\\
\tau_{\texttt{arg}}(T_1 \Rightarrow T_2) &= \texttt{error}\\
\tau_{\texttt{arg}}(T) &= T\text{, otherwise}\\
\tau(T_1 \Rightarrow T_2) &= \texttt{Rep}[\tau_{\texttt{arg}}(T_1)] \Rightarrow \texttt{Rep}[\tau_{\texttt{arg}}(T_2)]\\
\tau(T) &= \texttt{Rep}[T]\text{, otherwise}
\end{aligned}$$

This translation preserves well-typedness but rejects programs that contain function types in the type-argument position. In this case this is a desired behavior as it fosters high-performance code by avoiding dynamic dispatch. As an alternative to rejecting function types in the type-argument position the deep embedding can provide coercions from `Rep[`A`] => Rep[`B`]` to `Rep[`A=>B`]` and from `Rep[`A=>B`]` to `Rep[`A`] => Rep[`B`]`.

Untyped backend. If DSL authors want to avoid complicated types in the back-end (e.g., `Rep[T]`), the τ functions can simply transform all types to the `Dynamic` [Abadi et al., 1991] type. Giving away type safety can make transformations in the back-end easier for the DSL author.

Custom types. All previous translations preserved types in the type parameter position. The reason is that the τ functions behaved like a higher-kinded type. If we would like to map some of the base types in a custom way, those types need to be changed in the position of type-arguments as well. This translation is used for EDSLs based on polymorphic embedding [Hofer et al., 2008] 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. We applied this translation to Slick [Typesafe] with great success (§2.5.3).

Interestingly, just by changing the type translation, the EDSL author can modify the behavior of an EDSL. For example, with the generic polymorphic embedding the EDSL will reify function IR nodes and thus allow for dynamic dispatch. In the same EDSL that uses the eager inlining translation, dynamic dispatch is restricted and all function calls are inlined.

2.2.2 Correctness

To completely conceal the deep embedding all type errors must be captured in the direct embedding or by the translation, i.e., the translation must never produce an ill-typed program. Proving this property is verbose and partially covered by previous work. Therefore, for each version of the type translation we provide references to the previous work and give a high-level intuition:

- *The identity translation* ensures that well-typed programs remain well typed after the translation to the deep embedding [Carette et al., 2009]. Here the deep embedding is the direct embedding with virtualized host language intrinsics.
- *Generic polymorphic embedding* preserves well-typedness [Carette et al., 2009]. Type `T` is uniformly translated to `Rep[T]` and thus every term will conform to its expected type.
- *Eager inlining* preserves well-typedness for programs that are not explicitly rejected by the translation. We discuss correctness of eager inlining in [Jovanovic et al., 2014b] on a Hindley-Milner based calculus similar to the one of Carette et al. [Carette et al., 2009].

  For the intuition why type arguments can not contain function types consider passing an increment function to the generic identity function:
  ```
  id[T => T](lam[T, T](x => x + 1))
  ```
  Here, the `id` function expects `Rep[_]` type but the argument is `Rep[T] => Rep[T]`.
- The *Dynamic* type supports all operations and, thus, static type errors will not occur. Here, the DSL author is responsible for providing a back-end where dynamic type errors will not occur.
- *Custom types* can cause custom type errors since EDSL authors can arbitrarily redefine types (e.g., `type Int = String`. Yin-Yang provides *no guarantees* for this type of the translation.

2.3 Restricting Host Language Constructs

The direct DSL programs can contain well-typed expressions that are not supported by the deep embedding. Often, these expressions lead to unexpected program behavior (§2.1) and we must rule them out by reporting meaningful and precise error messages to the user.

We could rule out unsupported programs by relying on properties of the core translation. If a direct program contains unsupported expressions, after translation it will become ill-typed in the deep embedding. We could reject unsupported programs by simply reporting type checking errors. Since, the direct program is well-typed and the translation preserves well-typedness all type errors must be due to unsupported expressions.

Unfortunately, naively restricting the language by detecting type-checking failures is leaking information about the deep embedding. The reported error messages will contain virtualized language constructs and types. This is not desirable as users should not be exposed to the internals of the deep embedding.

Yin-Yang avoids leakage of the deep embedding internals in error messages by performing an additional verification step that, in a fine grained way, checks if a method from the direct program exists in the deep embedding. This step traverses the tree generated by the core translation and verifies for each method call if it correctly type-checks in the deep embedding. If the type checking fails Yin-Yang reports two kinds of error messages:

- Generic messages for unsupported methods:

  ```
  List.fill(1000, Vector.fill(1000,1)).reduce(_+_)
  ^
  Method List.fill[T] is unsupported in VectorDSL.
  ```

- Custom messages for unsupported host language constructs:

  ```
  try Vector.fill(1000, 1) / 0
  ^
  Construct try/catch is unsupported in VectorDSL.
  ```

With Yin-Yang the DSL author can arbitrarily restrict virtualized constructs in an embedded language by simply omitting corresponding method definitions from the deep embedding. Due to the additional verification step all error messages are clearly shown to the user. This allows easy construction of embedded DSLs that support only a subset of the host language.

2.4 Automatic Generation of the Deep Embedding

So far, we have seen how Yin-Yang translates programs written in the direct embedding to the deep embedding. This arguably simplifies life for EDSL users by allowing them to work with the interface of the direct embedding. However, the EDSL author still needs to maintain synchronized implementations of the two embeddings, which can be a tedious and error prone task.

To alleviate this issue, Yin-Yang automatically generates the deep embedding from the implementation of the direct embedding. This happens in two steps: First, we generate high-level

IR nodes and methods that construct them through a systematic conversion of methods declared in a direct embedding to their corresponding methods in the deep embedding (§2.4.1). Second, we exploit the fact that method implementations in the direct embedding are also direct DSL programs. Reusing our core translation from §2.2, we translate them to their deep counterparts (§2.4.2). In the translated method bodies, in addition to the translated DSL itself, we also allow usage of the Scala library constructs that supported by the target back-end (cf. [Jovanovic et al., 2014b]).

The automatic generation of deep embeddings reduces the amount of boilerplate code that has to be written and maintained by EDSL authors, allowing them to instead focus on tasks that can not be easily automated, such as the implementation of domain-specific optimizations in the deep embedding. However, automatic code generation is not a silver bullet. Hand-written optimizations acting on the IR typically depend on the structure of the later, introducing hidden dependencies between such optimizations and the direct embedding. Care must be taken in order to avoid breaking optimizations when changing the direct embedding of the EDSL.

For further information on how to use Yin-Yang's code generation together with the core translation, and how to specify rewrite rules, cf. [Jovanovic et al., 2014b].

2.4.1 Constructing High-Level IR Nodes

To make the generation regular Yin-Yang provides a corresponding IR node and construction method for every operation in the direct embedding. By using reflection, we extract the method signatures from the direct embedding. From these, we generate the interface, implementation, and code generation traits as prescribed by LMS. This part of the translation is LMS specific and applying it to other frameworks would require changing the code templates. Based on the signature of each method, we generate the *case class* that represents the IR node. Then, for each method we generate a corresponding method that instantiates the high-level IR nodes. Whenever a method is invoked in the deep EDSL, instead of being evaluated, a high-level IR node is created.

Figure 2.5 illustrates the way of defining IR nodes for `Vector` EDSL. The case classes in the `VectorOps` trait define the IR nodes for each method in the direct embedding. 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`).

Deep embedding should, in certain cases, be aware of side-effects. The EDSL author must annotate methods that cause side-effects with an appropriate annotation. To minimize the number of needed annotations we use Scala FX [Rytz et al., 2012]. Scala FX is a compiler plugin that adds an effect system on top of the Scala type system. 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

```
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 VectorMap[T:Numeric,S:Numeric]
    (v: Rep[Vector[T]], f: Rep[T] => Rep[S])
    extends Rep[Vector[S]]
  case class VectorFill[T:Numeric]
    (v: Rep[T], size: Rep[Int])
    extends Rep[Vector[T]]

  def vector_map[T:Numeric,S:Numeric]
    (v: Rep[Vector[T]], f: Rep[T] => Rep[S]) =
      VectorMap(v, f)
  def vector_fill[T:Numeric]
    (v: Rep[T], size: Rep[Int]) =
    VectorFill(v, size)
}
```

Figure 2.5 – High-level IR nodes for Vector.

```
class Vector[T: Numeric](val data: Seq[T]) {
  // effect annotations not necessary
  def print() = System.out.print(data)
}
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]]) =
    reflect(VectorPrint(v))
}
```

Figure 2.6 – Direct and deep embedding for Vector with side-effects.

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 2.6 shows how we automatically transform the I/O effect of a `print` method to the appropriate construct in LMS. As the Scala FX plugin knows the effect of `System.out.println`, the effect for the `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 `reflect`, which is an effect construct in LMS to specify an I/O effect [Rompf et al., 2011]. In effect, all optimizations in the EDSL will have to preserve the order of `println` and other I/O effects. We omit details about the LMS effect system; for more details cf. [Rompf et al., 2011].

```
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))
    }
}
```

Figure 2.7 – Lowering to the low-level implementation for Vector.

2.4.2 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 can be achieved by using a *lowering* phase [Rompf et al., 2013b].

Figure 2.7 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, a `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 an `atPhase(lowering)` block [Rompf et al., 2013b]. Furthermore, the low-level implementation uses constructs requiring deep embedding of other interfaces. In particular, an implementation of the `fill` method requires the `Seq.fill` method that is provided by the `SeqLowLevel` trait.

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. To this effect, we implemented Sprinter [Nikolaev], 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.

2.5 Evaluation

We compared the deep embedding generation of Yin-Yang with Forge on three Delite-based deep EDSLs: OptiML, OptiQL, and Vector (§2.5.1). Then, we measured the effect of concealing the deep embedding by counting the number of obviated annotations related to deep embedding in the test suites of OptiML and OptiGraph EDSLs (§2.5.2). Finally, we evaluated the ease of adopting Yin-Yang for the existing deep EDSL Slick [Typesafe] (§2.5.3) and compare the effort of designing the interface with the current version of the interface. We do not report on execution speed since performance benefits of the deep embedding have been studied previously [Rompf et al., 2013b, Sujeeth et al., 2013b].

2.5.1 Automatic Deep EDSL Generation

To evaluate the automatic deep EDSL generation for OptiML, OptiQL, and Vector, we used Forge [Sujeeth et al., 2013b], a Scala based meta-EDSL for generating both direct and deep EDSLs from a single specification. Forge already contained specifications for OptiML and OptiQL.

To avoid re-typing OptiML and OptiQL we modified Forge to generate the direct embedding from its specification and generated the direct embeddings from the existing Forge based EDSL specifications. Then, we used our automatic deep generation tool to convert these direct embeddings into their deep counterparts. Since, deep EDSLs mostly consist of boilerplate the generated embeddings have a similar number of LOC as the handwritten counterparts. For all three EDSLs, we verified that tests running in the direct embeddings behave the same as the tests for the deep embeddings.

In Table 2.1, we give a line count comparison for the code in the direct embedding, Forge specification, and deep embedding for three EDSLs: *i) OptiML* is a Delite-based EDSL for machine learning, *ii) OptiQL* is a Delite-based EDSL for running in-memory queries, and *iii) Vector* is the EDSL shown as an example throughout this paper. We are careful with measuring lines-of-code (LOC) with Forge and the deep EDSLs: we only count the parts which are generated out of the given direct EDSL.

Overall, Yin-Yang requires roughly the same number of LOC as Forge to specify the DSL. This can be viewed as positive result since Forge relies on a specific meta-language for defining the two embeddings. Yin-Yang, however, uses Scala itself for this purpose and is thus much easier to use. In case of OptiML, Forge slightly outperforms Yin-Yang. This is because Forge supports meta-programming at the level of classes while Scala does not.

Table 2.1 – LOC for direct EDSL, Forge specification, and deep EDSL.

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

We did not compare the efforts required to specify the DSL with Yin-Yang and Forge. The reason is twofold:

- It is hard to estimate the effort required to design a DSL. If the same person designs a single DSL twice, the second implementation will always be easier and take less time. On the other hand, when multiple people implement a DSL their skill levels can greatly differ. Finally, DSL design is technically demanding and it is hard to find a large enough group to conduct a statistically significant user study.

- Writing the direct embedding in Scala is arguably simpler than writing a Forge specification. Forge is Delite-specific language and uses a custom preprocessor to define method bodies in Scala. Thus, learning a new language and combining it with Scala snippets must be harder than just writing idiomatic Scala.

2.5.2 No Annotations in the Direct Embedding

To evaluate the number of obviated annotations related to the deep embedding we implemented a direct embedding for the OptiGraph EDSL (an EDSL for graph processing), and used the generated direct EDSL for OptiML. We implemented the whole application suites of these EDSLs with the direct embedding. All 21 applications combined have 1284 lines of code.

To see the effects of the direct embedding as the front-end we counted the number of deep embedding related annotations that were used in the application suite. The counted annotations are `Rep[T]` for types and `lift(t)` for lifting literals when implicit conversions fail. In 21 applications the direct embedding obviated 96 `Rep[T]` annotations and 5 `lift(t)` annotations.

2.5.3 Yin-Yang for Slick

Slick is a deeply embedded Scala EDSL for database querying and access. Slick is not based on LMS, but still uses `Rep` types to achieve reification. To improve the complicated interface of Slick we used Yin-Yang. However, since the deep embedding of Slick already exists, we first designed the new interface (direct embedding). The new 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 user friendly error reporting and documentation. The interface is completely new, covers all the functionality of Slick, and consists of only 70 lines of code (cf. [Jovanovic et al., 2014b]).

Slick has complicated method signatures that do not correspond to the simple new interface. In order to preserve backward compatibility, the redesign of Slick to fit Yin-Yang's core translation was not possible. We addressed this by adding a wrapper for the deep embedding of Slick that fits the required signature. The wrapper contains only 240 lines of straightforward code.

We compare the effort required for the interface design with Yin-Yang and with traditional type system based approaches. The development of the previous Slick interface required more than a year of development while the Yin-Yang version was developed in less than one month. The new front-end passes all 54 tests that cover the most important functionalities of Slick.

When using Slick all error messages are idiomatic to Scala and resemble typical error messages from the standard library.

This study was performed by only two users and, thus, is not statistically significant. Still, we find the difference in required effort large enough to indicate that Yin-Yang simplifies front-end development of EDSLs.

2.6 Discussion

Yin-Yang consistently translates terms to the embedded domain and, thus, postpones DSL compilation to run-time. Although, compilation happens in a different compilation stage, Yin-Yang does not allow staging [Taha and Sheard, 1997]. EDSLs can, however, achieve partial evaluation [Jones et al., 1993] if their implementation supports it.

We implemented Yin-Yang in Scala, however the underlying principles are applicable in the wider context. Yin-Yang operates in the domain of statically typed languages based on the Hindley-Milner calculus with a type system that is advanced enough to support deep EDSL embedding. The type inference mechanism, purity, laziness, and sub-typing, do not affect the operation of Yin-Yang. Different aspects of Yin-Yang require different language features, which we discuss separately below.

The core translation and language restriction are based on term and type transformations. Thus, the host language must support reflection, introspection and transformation on types and terms. This can be achieved both at run-time and compile-time.

Semantic equivalence between the direct embedding and deep embedding is required for debugging and prototyping. If there is a *semantic mismatch* [Czarnecki et al., 2004] between the two embeddings, e.g., the host language is lazy and the embedded language is strict, Yin-Yang can not be used for debugging. In this scenario the direct embedding can be implemented as stub which is used only for its user friendly interface and error reporting.

2.7 Related Work

Yin-Yang is a framework for developing embedded DSLs in the spirit of Hudak [Hudak, 1996, 1998]: embedded DSLs are *Scala libraries* and DSL programs are just *Scala programs* that do not, in general, require pre- or post-processing using external tools. Yin-Yang translates directly embedded DSL programs into finally-tagless [Carette et al., 2009] deep embeddings. Our approach supports (but is not limited to) polymorphic [Hofer et al., 2008] deep embeddings, and – as should be apparent from the examples used in this paper – is particularly well-adapted for deep EDSLs using an LMS-type IR [Rompf et al., 2013a,b].

As discussed in §**??**, Sujeeth et al. propose Forge [Sujeeth et al., 2013b], a Scala based meta-EDSL for generating equivalent shallow and deep embeddings from a single specification. DSLs generated by Forge provide a common abstract interface for both shallow and deep embeddings through the use of abstract `Rep` types. A shallow embedding is obtained by defining `Rep` as the identity function on types, i.e. `Rep[T] = T`.

A DSL user can switch between the shallow and deep embeddings by changing a single flag in the project build. Unfortunately, the interface of the shallow embedding generated by Forge remains cluttered with `Rep` type annotations. Additionally, some plain types that are admissible in a directly embedded program may lack counterparts among the IR types of the deep embedding. This means that some seemingly well-typed DSL programs become ill-typed once the transition from the shallow to the deep embedding is made, forcing users to manually fix type errors in the deeply embedded program. Finally, DSL authors must learn a new language for EDSL design whereas with Yin-Yang this language is Scala itself.

Project Lancet [Rompf et al., 2013c] by Rompf et al. and work of Scherr and Chiba [Scherr and Chiba, 2014] interpret Java bytecode to extract domain-specific knowledge from directly embedded DSL programs compiled to bytecode. These solutions are similar to Yin-Yang in that the direct embedding is translated to the deep embedding, however, they do not provide functionality to generate a deep embedding out of a direct one.

Awesome Prelude [Lokhorst, 2012] proposes replacing all primitive types in Haskell with type classes that can then be implemented to either construct IR or execute programs directly. This allows to easily switch between dual embeddings while the type classes ensure equivalent type checking. Unfortunately, this approach does not extend easily to native language constructs, and requires changing the type signatures of common functions.

3 Polyvariant Staging

Multi-stage programming (or *staging*) is a meta-programming technique where compilation is separated in multiple *stages*. Execution of each stage outputs code that is executed in the next stage of compilation. The first stage of compilation happens at the *host language* compile time, the second stage happens at the host language runtime, the third stage happens at runtime of runtime generated code, etc. Different stages of compilation can be executed in the same language [Nielson and Nielson, 2005, Taha and Sheard, 1997] or in different languages [Brown et al., 2011, DeVito et al., 2013]. In this work we will focus on staging systems where all stages are in the same language and that, through static typing, assure that terms in the next stage are well typed.

Notable staging systems in statically typed languages are MetaOCaml [Calcagno et al., 2003, Taha and Sheard, 1997] and LMS [Rompf and Odersky, 2012a]. These systems were successfully applied as a *partial evaluatior* [Jones et al., 1993]: for removing abstraction overheads in high-level programs [Carette and Kiselyov, 2005, Rompf and Odersky, 2012a], for domain-specific languages [Czarnecki et al., 2004, Jonnalagedda et al., 2014, Taha, 2004], and for converting language interpreters into compilers [Futamura, 1999, Rompf et al., 2013c]. Staging originates from research on two-level [Davies, 1996, Nielson and Nielson, 2005] and multi-level [Davies and Pfenning, 1996] calculi.

We show an example of how staging is used for partial evaluation of a function for computing the inner product of two vectors[1]:

```
def dot[T:Numeric](v1: Vector[T], v2: Vector[T]): T =
  (v1 zip v2).foldLeft(zero[T]) {
    case (prod, (cl, cr)) => prod + cl * cr
  }
```

In function `dot`, if vector sizes are constant, the inner product can be partially evaluated into a sum of products of vector components. To achieve partial evaluation, we must communicate to the staging system that operations on values of vector components should be executed in the next stage. The compilation stage in which a term is executed is determined by *code* *quotation* (in MetaOCaml) or by parametric types Rep (in LMS). In LMS marking that the vector size is statically known is achieved by annotating only vector elements with a Rep type[2]:

```
def dot[T:Numeric]
  (v1: Vector[Rep[T]], v2: Vector[Rep[T]]): Rep[T]
```

Here the Rep annotations on Rep[T] denote that elements of vectors will be known only in the next stage (in LMS, this is a stage after run-time compilation). After run-time compilation zip, foldLeft, and pattern matching inside the closure will not exist as they were evaluated in the previous stage of compilation (host language runtime). Note that in LMS unannotated code is always executed during host-language runtime and type-annotated code is executed after run-time compilation.

Staging at host language compile time. How can we use staging for programs whose values are statically known at the host language compile-time (the first stage)? Existing staging frameworks treat unannotated terms as runtime values of the host language and annotated terms as values in later stages of compilation. Even if we would take that the first stage is executed at the host language compile time, we would have to annotate all run-time values. Annotating all values is cumbersome since host language run-time values comprise the majority of user programs (§3.3).

MacroML [Ganz et al., 2001] expresses macros as two-stage computations that start executing from host language compile time. In MacroML, parameters of macros can be annotated as an early stage computation. These parameters can then used in escaped terms for compile-time computation. Terms scheduled for runtime execution, withing the escaped terms, again need to be quoted with brackets. This, in effect, imposes quotation for both escaping and brackets which requiring additional effort.

Code Duplication. Staging systems based on type annotations (e.g., LMS and type-directed partial evaluation [Danvy, 1999]) inherently require code duplication as, a priory, no operations are defined on Rep annotated types. For example, in the LMS version of the dot function, all numeric types (i.e., Rep[Int], Rep[Double], etc.) must be re-implemented in order to typecheck the programs and achieve code generation for the next stage.

Sujeeth et al. [Sujeeth et al., 2013b] and Jovanovic et al. [Jovanovic et al., 2014a] propose generating code for the next stage computations based on a language specification. These approaches solve the problem, but they require writing additional specification for the libraries, require a large machinery for code generation, and support only restricted parts of Scala.

Annotating the Previous Stage. The main idea of this paper is that *annotated types* should denote computations that happen during the *previous stage* of compilation. The reason is that static terms appear less frequently than run-time terms in a large set of analyzed programs (§3.3). Therefore, annotating static terms introduces less overhead for the programmer.

[1]All code examples are written in *Scala*. It is necessary to know the basics of Scala to comprehend this paper.
[2]In this work we use LMS as a representative of type-based staging systems.

We treat annotated types as *compile-time views* of existing data types. Compile-time view of a type denotes that all operations on that type are executed at host language compile time. We promote types to their compile-time views with the `@ct` annotation (e.g., `Int@ct`). Similarly, statically known terms can be promoted their compile time duals with the `ct` function on the term level. By having two views of the same type we obviate the need for introducing reification and code generation logic for existing types.

With compile-time views, to require that vectors `v1` and `v2` are static and to partially evaluate the function, a programmer needs to make a simple modification of the `dot` signature:

```
def dot[V: Numeric@ct]
  (v1: Vector[V]@ct, v2: Vector[V]@ct): V
```

Since, vector elements are polymorphic the result of the function can be a dynamic value, or a compile-time view that can be further used for compile-time computations. The binding time of the return type of `dot` will match the binding time of vector elements:

```
  // [el1, el2, el3, el4] are dynamic decimals
  dot(Vector(el1, el2), Vector(el3, el4))
    ↪ (el1 * el3 + el2 * el4): Double

  // static terms are internally tracked
  dot(Vector(2.0, 4.0), Vector(1.0, 10.0))
    ↪ (2.0 * 1.0 + 4.0 * 10.0): Double@static

  // ct promotes static terms to compile-time views
  dot(Vector(ct(2), ct(4)),
      Vector(ct(1), ct(10)))
    ↪ 42: Double@ct
```

In this paper we contribute to the state of the art:

- By introducing compile-time views (§3.1) as means to succinctly achieve type safe two-stage programming starting from host language compile time.
- By obviating the need for reification and code generation logic in type based staging systems.
- By demonstrating the usefulness of compile-time views in four case studies (§3.2): inlining, partially evaluating recursion, removing overheads of variable argument functions, and removing overheads of type-classes [Hall et al., 1996, Oliveira et al., 2010, Wadler and Blott, 1989].

We have implemented a staging extension for Scala Yin-Yang[3]. Yin-Yang has a minimal interface (§3.1) based on type annotations. We have evaluated performance gains and the validity of Yin-Yang on all case studies (§3.2) and compared them to LMS. In all benchmarks (§**??**) our evaluator performs the same as LMS and gives significant performance gains compared to

```
package object scalact {
  final class ct extends StaticAnnotation

  @compileTimeOnly def ct[T](body: => T): T = ???
}
```

Figure 3.1 – Interface of Yin-Yang.

Table 3.1 – Compile-time views of types and their corresponding method signatures.

Annotated Type	Type's Method Signatures
`Int@ct`	`+(rhs: Int@ct): Int@ct`
`Vector[Int]@ct`	`map[U](f: (Int => U)@ct): Vector[U]@ct` `length: Int@ct`
`Vector[Int@ct]@ct`	`map[U](f: (Int@ct => U)@ct): Vector[U]@ct` `length: Int@ct`
`Map[Int@ct, Int]@ct`	`get(key: Int@ct): Option[Int]@ct`

original programs.

3.1 Compile-Time Views in Scala

In this section we informally present Yin-Yang, a staging extension for Scala based on compile-time views. Yin-Yang is a compiler plugin that executes in a phase after the Scala type checker. The plugin takes as input typechecked Scala programs and uses type annotations [Odersky and Läufer, 1996] to track and verify information about the biding-time of terms. It supports only two stages of compilation: host language compile-time (types annotated with `@ct`) and host language run-time (unannotated code).

To the user, Yin-Yang exposes a minimal interface (Figure **??**) with a single annotation `ct` and a single function `ct`.

Annotation `ct` is used on types (e.g., `Int@ct`) to promote them to their compile-time views. The annotation is integrated in the Scala's type system and, therefore, can be arbitrarily nested in different variants of types.

Since all operations on compile-time views are executed at compile time, non-generic method parameters and result types of compile-time views also become compile-time views. Table 3.1 shows how the `@ct` annotation can be placed on types and how it affects method signatures on annotated types.

In Table 3.1, on `Int@ct` both parameters and result types of all methods are also compile-time views. On the other hand, `Vector[Int]@ct` has parameters of all methods transformed

[3]Source code: https://github.com/scala-inline/scala-inline.

Table 3.2 – Promotion of terms to their compile-time views.

Promoted Term	Term's Type
`ct(Vector)(1, 2, 3)`	`: Vector[Int]@ct`
`ct(Vector)(ct(1), ct(2), ct(3))`	`: Vector[Int@ct]@ct`
`ct((x: Int@ct) => x)`	`: (Int@ct => Int@ct)@ct`
`ct((x: Int) => x)`	`: (Int => Int)@ct`
`new (::@ct)(1, Nil)`	`: (::[Int])@ct`
`new (::@ct)(ct(1), ct(Nil))`	`: (::[Int@ct])@ct`

except the generic ones. In effect, this, makes higher order combinators of `Vector` operate on dynamic values, thus, function `f` passed to `map` accepts the dynamic value as input. Type `Vector[Int@ct]@ct` has all methods executed at compile-time. The return type of the function `map` on `Vector[Int@ct]@ct` can still be either dynamic or a compile-time view due to the type parameter `U`.

Annotation `ct` can be used to achieve simple inlining of statically known methods and functions. This is achieved by putting the annotation of the method/function definition:

```
def dot[V: Numeric]
 (v1: Vector[V], v2: Vector[V]): V
```

Annotated methods will have an annotated method type

```
((v1: Vector[V], v2: Vector[V]) => V)@ct
```

which can not be written by the users. This is not the first time that inlining is achieved through partial evaluation [Monnier and Shao, 2003].

Function `ct` is used at the term level for promoting literals, modules, and methods/functions into their compile-time views. Without `ct` we would not be able to instantiate compile-time views of types. Table 3.2 shows how different types of terms are promoted to their compile-time views. An exception for promoting terms to compile-time views is the `new` construct. Here we use the type annotation on the constructed type.

3.1.1 Tracking Binding-Time of Terms

Internally Yin-Yang has additional type annotations for tracking the binding time of terms. Type of each term is annotated with either `dynamic`, `static`, or `ct`. `dynamic` denotes that the term can only be known at runtime, `static` that the term is known at compile-time but it will not be computed at compile time, and `ct` that the term will be computed at compile-time.

Tracking static terms was studied in the context of binding-time analysis in partial evaluation for typed [Nielson and Nielson, 1988] and untyped [Gomard and Jones, 1991] languages. We use similar techniques, however, unlike in partial evaluation we do not evaluate static terms at compile time. They are tracked for verifying correctness and providing convenient implicit conversions. Static terms are evaluated only when they are explicitly marked by the programmer with `ct`.

In Yin-Yang language literals, functions, direct class constructor calls with static arguments, and static method calls with static arguments are marked as static. Examples of static terms are

```
1.0, "1", (x: Int => x), new Cons(1, Nil), List(1,2,3)
```

3.1.2 Least Upper Bounds

We use subtyping of Scala to simplify tracking of binding times by introducing a subtyping relation between `dynamic`, `static`, and `ct`. We argue that a `static` type is a more specific `dynamic` as it is statically known and that `ct` is more specific than `static` as its operations are executed at compile time. Therefore we establish that

```
ct <: static <: dynamic
```

The use of subtyping simplifies tracking binding times of terms as in all cases where least upper bounds are calculated we can use the same mechanism for binding-times. An interesting example are the binding times of type parameters:

```
ct(List)(1, ct(2)): List[Int@static]@ct
ct(List)(ct(1), ct(2)): List[Int@ct]@ct
ct(List)((x: Int@dynamic), ct(2)): List[Int@dynamic]@ct
```

Notable exception are control flow constructs for which the original Scala rules for least upper bounds do not hold. The binding-time of control flow constructs does not depend only on the return type of the branches but also on the conditionals. For example, if both branches of an `if` construct are `static` the result can still be `dynamic` if the condition is `dynamic`. Here subtyping also helps as the binding type of the return value is simply calculated as `lub(c, thn, elz)` where `lub(tps: Type*)` is a function for computing the least upper bounds of types, and `c`, `thn`, `elz` are respectively binding times of the condition, the then branch, and the else branch. The same principles can be applied for pattern matching.

3.1.3 Well-Formedness of Compile-Time Views

Earlier stages of computation can not depend on values from later stages. This property, defined as *cross-stage persistence* [Taha and Sheard, 1997, Westbrook et al., 2010], imposes that all operations on compile-time views must known at compile time.

To satisfy cross-stage persistence Yin-Yang verifies that binding time of composite types (e.g., polymorphic types, function types, record types, etc.) is always a subtype of the binding time of their components. In the following example, we show malformed types and examples of terms that are inconsistent:

```
xs: List[Int@ct]      => ct(Predef).println(xs.head)
fn: (Int@ct=>Int@ct) => ct(Predef).println(fn(ct(1)))
```

In the first example the program would, according to the semantics of `@ct`, print a head of the list at compile time. However, the head of the list is known only in the runtime stage. In the second example the program should print the result of `fn` at compile time but the body of the function will be known only at runtime. By causality such examples are not possible.

On functions/methods the `ct` annotation requires that function/method bodies are known at compile-time. Otherwise, inlining of such functions/methods would not be possible at compile-time. In Scala, method bodies are statically known in objects and classes with final methods, thus, the `ct` annotation is only applicable on such methods.

3.1.4 Implicit Conversions

If method parameters require compile-time views of a type the corresponding arguments in method application would always have to be promoted to `ct`. In some libraries this could require an inconveniently large number of annotations.

To minimize the number of required annotations we introduce implicit conversions from certain `static` terms to `ct` terms. The conversions support translation of language literals, direct class constructor calls with static arguments, and static method calls with static arguments into their compile-time views. Since our compile-time evaluator does not use Asai's [Asai, 2002, Sumii and Kobayashi, 2001] method to keep track of the value of each static term, we disallow implicit conversions of terms with static variables.

For example, for a factorial function

```
def fact(n: Int @ct): Int@ct =
  if (n == 0) 1 else fact(n - 1)
```

we will not require promotions of literals `0`, and `1`. Furthermore, the function can be invoked without promoting the argument into it's compile-time view:

```
fact(5)
  ↪ 120
```

Without implicit conversions the factorial functions would be more verbose

```
def fact(n: Int @ct): Int@ct =
  if (n == ct(0)) ct(1) else fact(n - ct(1))
```

as well as each function application (`fact(ct(5))`).

3.2 Case Studies

In this section we present selected use-cases for compile-time views that, at the same time, demonstrate step-by-step the mechanics behind Yin-Yang. We start by inlining a simple function with staging (§3.2.1), then do the canonical staging example of the integer power function (§3.2.2), then we demonstrate how variable argument functions can be desugared into the core functionality (§3.2.3). Finally, we demonstrate how the abstraction overhead of the `dot` function and all associated type-class related abstraction an be removed (§3.2.5).

3.2.1 Inlining Expressed Through Staging

Function inlining can be expressed as staged computation [Monnier and Shao, 2003]. Inlining is achieved when a statically known function body is applied with symbolic arguments. In Yin-Yang we use the `ct` annotation on functions and methods to achieve inlining:

```
@ct def zero[T](implicit num: Numeric[T]) = num.zero

zero[Double]
  ↪ num.zero
```

3.2.2 Recursion

The canonical example in staging literature is partial evaluation of the power function where exponent is an integer:

```
def pow(base: Double, exp: Int): Double =
  if (exp == 0) 1 else base * pow(base, exp - 1)
```

When the exponent (`exp`) is statically known this function can be partially evaluated into `exp` multiplications of the `base` argument, significantly improving performance [Calcagno et al., 2003].

With compile-time views making `pow` partially evaluated requires adding only one annotation:

```
def pow(base: Double, exp: Int@ct): Double =
  if (exp == 0) 1 else base * pow(base, exp - 1)
```

To satisfy cross-stage persistence (§3.1.3) the `pow` must be `@ct`. However, to reduce the number of required annotations we implicitly add the `ct` annotation when at least one parameter type or the result type is marked as `ct`. In the example the `ct` annotation on `exp` requires that the function must be called with a compile-time view of `Int`. Yin-Yang ensures that the definiton of the `pow` function does not cause infinite recursion at compile-time by invoking the power function only when the value of the `ct` arguments is known.

The application of the function `pow` with a constant exponent will produce:

```
pow(base, 4)
```

```
def min(vs: Int*): Int = macro
  if (isVarargs(vs)) q"min_CT(vs)"
  else q"min_D(vs)"

def min_CT(vs: Seq[Int]@ct): Int =
  vs.tail.foldLeft(vs.head) { (min, el) =>
    if (el < min) el else min
  }

def min_D(vs: Seq[Int]): Int =
  vs.tail.foldLeft(vs.head) {
    (min, el) => if (el < min) el else min
  }
```

Figure 3.2 – Function `min` is desugared into a `min` macro that based on the binding time of the arguments dispatches to the partially evaluated version (`min_CT`) for statically known varargs or to the original min function for dynamic arguments `min_D`.

```
  ↪ base * base * base * base * 1
```

Constant 4 is promoted to `ct` by the implicit conversions (§3.1.4).

3.2.3 Variable Argument Functions

Variable argument functions appear in widely used languages like Java, C#, and Scala. Such arguments are typically passed in the function body inside of the data structure (e.g. `Seq[T]` in Scala). When applied with variable arguments the size of the data-structure is statically known and all operations on them can be partially evaluated. However, sometimes, the function is called with arguments of dynamic size. For example, function `min` that accepts multiple integers

```
def min(vs: Int*): Int = vs.tail.foldLeft(vs.head) {
  (min, el) => if (el < min) el else min
}
```

can be called either with statically known arguments (e.g., `min(1,2)`) or with dynamic arguments:

```
val values: Seq[Int] = ... // dynamic value
min(values: _*)
```

Ideally, we would be able to achieve partial evaluation if the arguments are of statically known size and avoid partial evaluation in case of dynamic arguments. To this end we translate the method `min` into a partially evaluated version and a dynamic version. The call to these methods is dispatched, at compile-time, by the `min` method which checks if arguments are statically known. Desugaring of `min` is shown in Figure 3.2.

```
object Numeric {
  implicit def dnum: Numeric[Double]@ct =
    ct(DoubleNumeric)
  def zero[T](implicit num: Numeric[T]@ct): T =
    num.zero
}

trait Numeric[T] {
  def plus(x: T, y: T): T
  def times(x: T, y: T): T
  def zero: T
}

object DoubleNumeric extends Numeric[Double] {
  def plus(x: Double, y: Double): Double = x + y
  def times(x: Double, y: Double): Double = x * y
  def zero: Double = 0.0
}
```

Figure 3.3 – Removing abstraction overheads of type classes.

3.2.4 Removing Abstraction Overhead of Type-Classes

Type-classes are omnipresent in everyday programming as they allow abstraction over generic parameters (e.g., `Numeric` abstracts over numeric values). Unfortunately, type-classes introduce *dynamic dispatch* on every call [Rompf et al., 2013b] and, thus, impose a performance penalty. Type-classes are in most of the cases statically known. Here we show how with Yin-Yang we can remove all abstraction overheads of type classes.

In Scala, type classes are implemented with objects and implicit parameters [Oliveira et al., 2010]. In Figure 3.3, we define a `trait Numeric` serves as an interface for all numeric types. Then we define a concrete implementation of `Numeric` for type `Double` (`DoubleNumeric`). The `DoubleNumeric` is than passed as an implicit argument `dnum` to all methods that use it (e.g., `zero`).

When `zero` is applied first the implicit argument (`dnum`) gets inlined due to the `ct` annotation of the return type, then the function `zero` gets inlined. Since `dnum` returns a compile-time view of `DoubleNumerc` the method `zero` on `dnum` is evaluated at compile time. The constant `0.0` is promoted to `ct` since `DoubleNumeric` is a compile time view. Finally the `ct(0.0)` result is coerced to a dynamic value by the signature of `Numeric.zero`. The compile-time execution is shown in the following snippet

```
Numeric.zero[Double]
  ↪ Numeric.zero[Double](DoubleNumeric)
  ↪ ct(DoubleNumeric).zero
  ↪ (ct(0.0): Double)
  ↪ 0.0
```

3.2.5 Inner Product of Vectors

Here we demonstrate how the introductory example (§??) is partially evaluated through staging. We start with the desugared `dot` function (i.e., all implicit operations are shown):

```
def dot[V](v1: Vector[V]@ct, v2: Vector[V]@ct)
  (implicit num: Numeric[V]@ct): V =
  (v1 zip v2).foldLeft(zero[V](num)) {
    case (prod, (cl, cr)) => prod + cl * cr
  }
```

Function `dot` is generic in the type of vector elements. This will reflect upon the staging annotations as well (`ct` and `static`). When we apply the `dot` function with static arguments we will get the vector with static elements back:

```
dot[Double@static](
  ct(Vector)(2.0, 4.0), ct(Vector)(1.0, 10.0))(
  Numeric.dnum)
↪
  (ct(Vector)(2.0, 4.0) zip ct(Vector)(1.0, 10.0))
    .foldLeft(ct(0.0)) {
      case (prod, (cl, cr)) => prod + cl * cr
    }
↪ (2.0 * 1.0 + 4.0 * 10.0): Double@static
```

When `dot` is evaluated with the `ct` elements the last step will further execute to a single compile-time value that can further be used in compile-time computations:

```
dot[Double@ct](
  ct(Vector)(ct(2.0), ct(4.0)),
  ct(Vector)(ct(1.0), ct(10.0)))(Numeric.dnum)
↪ ct(2.0) * ct(1.0) + ct(4.0) * ct(10.0)
↪ 42.0: Double@ct
```

3.3 Discussion

To distinguish terms executed at compile-time from terms executed at runtime with type annotations we have the following possibilities:

1. Annotate types of all terms that should be executed at runtime. Here all types analyzed LMS and realized that this is not an option.

2. Annotate types of terms that should be executed at runtime but introduce scopes (e.g., method bodies) for which this rule applies. In this way we would avoid annotating types of all runtime terms. This approach is taken by MacroML where macro functions are executed at compile time and quoted terms are executed at runtime. First approach is, also, a special case of this approach where there is a single scope for the whole language.

3. Annotate types of terms that are executed at compile time. This approach is used with Yin-Yang and annotated types are called compile-time views.

In Yin-Yang we decided to annotate types of terms that are executed at compile time. Compared to the first solution our approach takes requires less annotations. We analyzed

Compared to the second approach our solution is simpler to comprehend and communicate. In the second approach there are two things that users need to understand when reasoning about staged programs: *i)* where does the compile time scope start, and *ii)* which terms are annotated. With Yin-Yang the comprehension is simple: terms whose types are annotated with `ct` are executed at compile time.

It is also interesting to the second and third approaches. Here the number of annotations depends on the program. If the programs are mostly partially evaluated the second approach is better. These category of programs could also be regarded as code generators as most of the code is executed at compile time and produces large outputs. When programs are comprised of mostly runtime values the approach of Yin-Yang requires less annotations.

3.4 Limitations

- Interaction with type variables.
- Type variables.
- Type annotations and overloading and implicit search.
- Can not inherit from a compile time view.

3.5 Related Work

MetaOCaml [Calcagno et al., 2003, Taha and Sheard, 1997] is a staging extension for OCaml. It uses quotation to determine the stage in which the term is executed. Types of quoted terms are annotated to assure cross-stage persistence. Staging in MetaOCaml starts at host language runtime and can not express compile-time computations. Further, operations on annotated types do not get automatically promoted to the adequate stage of computation as with compile-time views. Finally, there are no implicit conversions so all stage promotions of terms must be explicit.

MacroML [Ganz et al., 2001] is a language that translates macros into MetaML staging executed at compile time to provide a “clean” solution for macros. In MacroML, within the `let mac` construct function parameters can be annotated as an early stage computation. These parameters can then be used in escaped terms, i.e., terms scheduled to execute at compile time. Unlike Yin-Yang, MacroML uses escapes and early parameters to mark terms scheduled for to execute at compile time. Within escapes terms scheduled for runtime again need to be marked with brackets. This kind of dual annotations are not required as compile-time views are automatically promoted to runtime terms.

In LMS [Rompf and Odersky, 2012a] terms that are annotated with `Rep` types will be executed at the stage after runtime compilation. Therefore, LMS can not directly be used for compile time computation. Furthermore, LMS requires additional reification logic and code generation for all `Rep` types.

Programming language Idris [Brady and Hammond, 2010] introduces the `static` annotation on function parameters to achieve partial evaluation. Annotation `static` denotes that the term is statically known and that all operations on that term should be executed at compile-time. However, since `static` is placed on terms rather then types, it can mark only *whole terms* as static. This restricts the number of programs that can be expressed, e.g., we could not express that vectors in the signature of *dot* are static only in size. Finally, information about `static` terms can not be propagated through return types of functions so `static` in Idris is a partial evaluation construct, i.e., it hints that partial evaluation should be applied if function arguments are static.

Hybrid Partial Evaluation (HPE) [Shali and Cook, 2011] is a technique for partial evaluation that does not perform binding time analysis (similarly to online partial evaluators) but relies on the user provided annotation `CT`[1]. HPE implementations exist for both Java and Scala [Sherwany et al., 2015]. Although, `CT` is used for partial evaluation, it does not affect typing of user programs. Furthermore, behavior of `CT` in context of generics is not described. Yin-Yang can be seen as statically typed version of hybrid partial evaluation with support for parametric polymorphism. Due to the support for parametric polymorphism Yin-Yang can express compile-time data structures with dynamic data.

Forge [Sujeeth et al., 2013b], by Sujeeth et al., uses a DSL to declare a specification of the libraries. Forge then generates both unannotated and annotated code based on the specification. Their language also supports generating staged code (comprised of terms different from multiple stages). Forge specification and code generation supports only a subset of Scala guided towards the Delite [Brown et al., 2011, Sujeeth et al., 2013a] framework.

The Yin-Yang framework, by Jovanovic et al. [Jovanovic et al., 2014a], solves the problem of code duplication by generating reification and code generation logic based on Scala code of existing types. With their approach there is no code duplication for the supported language features. However, not all of the Scala language is supported and all generated terms are generated for the next stage, thus, making a stage distinction is impossible.

[1] Name `ct` in Yin-Yang is inspired by hybrid partial evaluation.

Bibliography

Martin 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.

Kenichi Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–51, 2002.

Edwin C Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *International Conference on Functional Programming (ICFP)*, 2010.

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.

Eugene Burmako. Scala macros: Let our powers combine! In *Workshop on Scala (SCALA)*, 2013.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering*, 2003.

Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *Generative Programming and Component Engineering (GPCE)*, 2005.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *International Conference on Functional Programming (ICFP)*, 2007.

Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. *Domain-Specific Program Generation*, 2004.

Olivier Danvy. *Type-directed partial evaluation*. Springer, 1999.

Rowan Davies. A temporal-logic approach to binding-time analysis. In *Symposium on Logic in Computer Science (LICS)*, 1996.

Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Symposium on Principles of Programming Languages (POPL)*, 1996.

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013.

Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

Steven E Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *International Conference on Functional Programming (ICFP)*, 2001.

Carsten K Gomard and Neil D Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(01):21–69, 1991.

Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar Swadi, and Walid Taha. Implementing DSLs in metaOCaml. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transansactions on Programming Langanguages and Systems (TOPLAS)*, 18 (2):109–138, 1996.

C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2008.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28 (4es):196, 1996.

Paul Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse (ICSR)*, 1998.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014. doi: 10.1145/2660193.2660241. URL http://doi.acm.org/10.1145/2660193.2660241.

V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, Koch C., and M Odersky. Yin-Yang: Concealing the deep embedding of DSLs. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2014a.

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir NIkolaev, Christoph Koch, and Martin Odersky. Yin-Yang: Concealing the deep embedding of DSLs. Technical Report EPFL-REPORT-200500, EPFL Lausanne, Switzerland, 2014b.

Tom Lokhorst. Awesome prelude, 2012. Dutch Haskell User Group, http://vimeo.com/9351844.

Stefan Monnier and Zhong Shao. Inlining as staged computation. *Journal of Functional Programming*, 13(03):647–676, 2003.

F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed &lgr;-calculus. In *Symposium on Principles of Programming Languages (POPL)*, 1988.

Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*, volume 34. Cambridge University Press, 2005.

Vladimir Nikolaev. Sprinter. http://vladimirnik.github.io/sprinter/.

M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification*. 2011.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Symposium on Principles of Programming Languages (POPL)*, 1996.

Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2005.

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

T. Rompf, A.K. Sujeeth, H.J. Lee, K.J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. In *IFIP Working Conference on Domain-Specific Languages (DSL)*, 2011.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012a.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, June 2012b.

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, 2013a.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanović, 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 *Symposium on Principles of Programming Languages (POPL)*, 2013b.

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Kunle Olukotun, and Martin Odersky. Project Lancet: Surgical precision JIT compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013c.

Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

Maximilian Scherr and Shigeru Chiba. Implicit staging of EDSL expressions: A bridge between shallow and deep embedding. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

Amin Shali and William R. Cook. Hybrid partial evaluation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

Amanj Sherwany, Nosheen Zaza, and Nathaniel Nystrom. A refactoring library for scala compiler extensions. In *Compiler Construction (CC)*, pages 31–48, 2015.

Arvind Sujeeth, Tiark Rompf, Kevin Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanović, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013a.

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 *International Conference on Machine Learning (ICML)*, 2011.

Arvind K Sujeeth, Austin Gibbons, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2013b.

Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2-3):101–142, 2001.

Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming (TFP)*, pages 21–36, 2013.

Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation (DSPG)*. 2004.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.

Typesafe. Slick. http://slick.typesafe.com/.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1989.

Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2010.

Curriculum Vitae

Personal Information

Full Name	Vojin Jovanovic
Address	Rue St-Roch 21, 1004, Lausanne, Switzerland
Telephone	+41 (0)21 69 37691 Mobile: +41 (0)78 871 91 74
Email	vojin.jovanovic@epfl.ch
Date of birth	14th January 1985

Goals and aspirations	I believe that programs can be written abstractly and yet execute as fast as their hand tuned counterparts. To this end, I am making a framework that allows effortless addition of domain-specific optimizations to existing libraries. I am also working on a high-level programming model for dynamic compilation where dynamic information is used to perform domain-specific optimizations at runtime; yet managing assumptions, deoptimization, and code caches is done behind the scenes.

Selected Work Experience

Position and Dates	**PhD Candidate** — October 2010 – present
Employer	Scala Laboratory (LAMP), EPFL, Switzerland
Main activities and responsibilities	Author and maintainer of the Yin-yang framework which is used for seamless embedding of DSLs. Yin-yang is used for generating and reifying queries in the new version of LegoBase. Co-author and initiator of Scala Records. Scala Records are used for type-safe manipulation of SparkSQL query results. Co-author of SIP 14 – Futures and Promises LMS contributor: implemented a loop fusion prototype, added record support, enabled and helped removal of the dependency to Virtualized Scala. Author of sbt-coursera which is used for automatic grading of Java based projects Coursera. Co-author of the Actors Migration Kit. Currently working on the Scala interpreter.
Position and Dates	**Research Intern** — June 2013 – September 2013
Employer	Oracle Labs, Switzerland
Main activities and responsibilities	Implemented the Graal backend for Lightweight Modular Staging with support for vectorization. Performance on all (at the time) supported vectorization features was within 10% of hand-written C.
Position and Dates	**Research Intern** — April 2010 – September 2010
Employer	Network Systems Laboratory (NSL), EPFL, Switzerland
Main activities and responsibilities	Implemented DiCE, a system that makes a snapshot of a network of BGP routers and uses concolic execution to explore the live system state. Exploration ensures that the faulty system states can not be reached. DiCE detects common errors in BGP networks like the cybernuke vulnerability.
Position and Dates	**Software Developer – Team Leader** — March 2008 – September 2009
Employer	Margintech Corporation, Toronto Working for Taleo inc. on the TBE product
Main activities and responsibilities	Developed software for a large SaaS system that is used daily by over 50.000 customers. Lead a team of 3 people on several enterprise projects. At the same time developed algorithms for cache re-balancing (10x improvement in memory utilization), fixed critical concurrency bugs and integrated a semantic search engine.

Education

School and Dates	School of Electrical Engineering, University of Belgrade, Serbia — October 2008 – April 2010
Title	Engineer of Electrical Engineering and Computer Science – Master — GPA 10.00/10.00 Thesis: Human Computer Interaction Device for Visually Impaired People
School and Dates	School of Electrical Engineering, University of Belgrade, Serbia — October 2003 – October 2008
Title	Engineer of Electrical Engineering — GPA 9.02/10.00 Thesis: Tactile Web Browser Simulator

Selected Publications

V. Jovanovic, D. Shabalin, E. Burmako, and M. Odersky, Annotating the Previous Stage: Succinct Type-Driven Staging at Compile Time, Scala'15 (under submission)

V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky, Yin-Yang: Concealing the deep embedding of DSLs, GPCE'14

A. Sujeeth, T. Rompf, K. Brown, H. Lee, H. Chafi, V. Popic, M.Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, Composition and reuse with compiled domain-specific languages, ECOOP'13

T. Rompf, A. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olkotun, and M. Odersky, Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers based on Staging, POPL '13

S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky, Jet: An Embedded DSL for High-Performance Big Data Processing, BigData'12

M. Canini, V. Jovanovic, D. Venzano, D. Novakovic, and D. Kostic, Online Testing of Federated and Heterogeneous Distributed Systems, Computer Communication Review, vol. 41, p. 434-435, 2011.

M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, and O. Crameri, Toward Online Testing of Federated and Heterogeneous Distributed Systems, USENIX'11

Activities

Selected talks	Programming DSLs Made Simple, ScalaDays 2014 Yin-Yang: Transparent Deep Embedding of DSLs, ScalaCamp 2013 High-Performance DSLs Embedded in Scala, GeeCon 2013
Reviewing	Artefact reviewer for OOPSLA'15 Subreviewer for HLPP'14, GPCE'14, and ICFP'14
Demos	Yin-Yang: Concealing the Deep Embedding of DSLs, ECOOP'15
Organizing	Summer School on Domain Specific Programming Languages, Lausanne, July 2015 Scala Workshop, 2013 PL Seminar at EPFL
Teaching	Reactive Programming and Parallelism (2015) Functional Programming Principles in Scala *(2013, 2014, 2015)* Principles of Reactive Programming *(2013, 2015)* Foundations of Software (2012) Operating Systems (2011)

References

Martin Odersky, Professor of Computer Science at EPFL, martin.odersky@epfl.ch

Christoph Koch, Professor of Computer Science at EPFL, christoph.koch@epfl.ch

Tiark Rompf, Professor of Computer Science at Purdue University, tiark@purdue.edu

Leonid Igolink, VP of Engineering, App. Perf. Management at CA Technologies, lim@igolnik.com

Anjan Goswami, Head of Search Science Engineering at Walmart Labs, goswami.anjan@gmail.com