# 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

Our society's infrastructure is controlled by billions of lines of code executed in data-centers, mobile devices, routers, personal computers, and device controllers. There are several million programmers worldwide writing that code mostly in *general-purpose programming languages* such as JavaScript, Java, Python, and C#. Advancements in our society's infrastructure depend on evolution of the code that controls it and evolution of code is directly influenced by *programmer productivity*.

Modern general-purpose programming languages allow programmers to be productive by providing constructs that allow high levels of *abstraction*. Good abstractions lead to concise programs that are easy to comprehend. Unfortunately, abstraction comes with a cost: each layer of abstraction must be executed on a *target platform* and makes programs *inefficient*.

Inefficient, long running, programs slow down decision making and use excess energy for computation. The amount of energy used for computation is becoming a significant portion of the overall energy consumption in the world. It is estimated that 2% of electricity budget in the United States is used for only data center computations [Mukherjee et al., 2009]. If we would write more efficient programs the IT infrastructure would advance faster and consume less energy.

Writing efficient programs in general-purpose programming languages, however, lead us back to low productivity. To make programs efficient programmers usually remove abstractions and hand-craft their programs for the particular platform where the program is executed [Lee et al., 2011]. The problem becomes even worse on *heterogeneous platforms* where programmers are faced with multiple *computing targets* such as parallel CPUs, GPUs, and FPGAs. With heterogeneous platforms programmers must specialize their programs for each target separately.

**Why general-purpose languages can not optimize abstract programs?** For compilers of general-purpose programming languages it is hard to remove the abstraction overheads and at the same time target heterogeneous platforms. The main reasons for this are:

- General purpose compilers reason about general computations. They are agnostic to *specific domains* such as linear algebra and relational algebra. This reduces the number of possible optimizations they can perform.
- General purpose compilers are faced with a overwhelming number of choices for optimization. Each choice exponentially increases a *search space* that the compiler needs to explore. Finding a specific solution that is optimal for a given platform in this wast space is in most cases unfeasible. The only way to efficiently explore the search space is to use the domain knowledge to guide the optimizer towards a close-to-optimal solution.
- Specific target platforms, such as GPUs, do not support code patterns that can be written in general-purpose programming languages. Once such code patterns are written it is hard or impossible for a compiler to transform them to executable code. If the code was written in a *restricted language* that allows only supported code patterns it would be much easier to target different platforms.

## 1.1 Domain-Specific Languages

Domain-specific languages (DSLs) provide a *restricted* interface that allows users to write programs at the high-level of abstraction and at the same time highly optimize them for execution on different target platforms. The restricted interface allows the optimizer to extract the *domain knowledge* from user programs. The domain knowledge is used to define a larger space of possible program executions and to guide the optimizer in exploring that space. The restricted interface and the domain knowledge can further be used to target heterogeneous platforms [Chafi et al., 2010].

A successful example of a DSL is the Standard Query Language (SQL) that has millions of active users worldwide. SQL concisely expresses the domain of data querying and uses the knowledge about relational algebra to optimize data queries as good as performance experts. SQL, as such, provides the base for almost all enterprise applications in the world.

### 1.1.1 Kinds of DSLs

DSLs can be categorized in two major categories: *i)* as *external DSLs* that have a specialized compiler for the language, and *ii)* as *internal* or *embedded DSLs* that are

*embedded* inside a general-purpose host language.

**External DSLs.** The implementation of a usable external (or *stand-alone*) DSL requires building a *language ecosystem* which consists of a compiler and a complete tool chain (IDE integration, debugging, documentation tools, etc.). This is a great undertaking that can in many cases outweigh the benefits of building an external DSL.

External DSLs usually start as concise restricted languages but through their development grow towards general-purpose languages. As DSLs become popular their language designers can not resist the user's demand for features of general-purpose languages. These features, as they are added after the initial language design, do not always fit well into the original language. For example, SQL in most databases supports constructs like loops, variables, and hash-maps which diverge from the domain of relational algebra.

**Embedded DSLs.** A promising alternative to external DSLs are *embedded DSLs* (EDSLs) [Hudak, 1996]. Embedded DSLs are hosted in a general-purpose language and reuse large parts of its infrastructure: *i)* IDE support, *ii)* tools (e.g., debuggers and code analysis), *iii)* compilation pipeline (e.g., parser, type-checker, optimizations, and code generation), and *iv)* standard library. Since general-purpose languages are designed for general purpose constructs, growth towards general-purpose constructs is well supported.

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*.

### 1.1.2 Comparison of DSL Kinds

In this section we will compare DSL kinds with respect to programmability and performance. As it is hard to quantify and exactly judge programmability, in the following discussion we will make a binary decision whether a DSL kind is easy to program or not. Scientifically proving how programmable is a DSL kind would require user studies which we did not perform—we rather build on informal collective experience.

To compare programmability of different DSL kinds we introduce two types of programmers:

- **DSL users** are people that use a DSL to model and solve their tasks. This is a larger group of programmers as, usually, there are more language users than language authors. Therefore, it is good to optimize the design of DSLs for this group of programmers.
- **DSL authors** are the programmers that develop domain-specific languages. This group is smaller than DSL users, but is still very important. If developing a DSL is hard then it will be harder to introduce new DSLs and features of existing DSLs will be developed at a slower pace.

**External DSLs.** For the DSL users it is, in the ideal case, *easy to program* in external DSLs. Given that the DSL authors implement a good language, the language syntax is crafted for the domain, and easy to comprehend and write. Error reporting and tooling should be built such that DSL users easily prevent, identify, and finally fix the errors in their programs.

For the DSL authors developing external DSLs is a big undertaking. Although, the development process is not hard as DSL authors design their own compiler, the amount of work required to build a language ecosystem is tremendous. Therefore, we categorize external DSLs as *hard to develop*.

Finally, external DSLs exhibit *high performance*. A language and its compiler can be designed such that they extract required domain-knowledge from user programs. This domain knowledge can then be used to optimize programs. Good example of high-performance DSLs is Spiral [Püschel et al., 2005] as it thoroughly defines and uses the domain knowledge to explore the entire search space in order to find optimal programs.

**Shallowly embedded DSLs.** For the DSL users it is *easy to program* in shallow DSLs but less so than the in external DSLs. Syntax and error reporting of the host language, typically, can not be modified to perfectly fit the domain. However, languages with flexible syntax [Moors et al., 2012, Odersky, 2010] and powerful type systems can closely

model many domains. Some host languages have language extensions for introducing extensible syntax [Erdweg et al., 2011] and customizable error reporting [Hage and Heeren, 2007, Heeren et al., 2003] further improving the interface of DSLs. Finally, shallow DSLs have perfect interoperability with the host language libraries as the values in the embedded language directly correspond to the values in the host language.

For the DSL authors basic shallowly embedded DSLs are *easy to program* as their development is similar to development of host language libraries. This makes it easy to evolve the language and experiment with different features. For DSLs with complex error reporting or extensible syntax the development becomes more difficult for the DSL authors.

Shallowly embedded DSLs exhibit *low performance*. The lack of the intermediate representation prevents exploiting the domain knowledge to implement optimizations. Further, having a language with the high-level of abstraction leads to layers of indirection and, thus, performance overheads.

**Deeply embedded DSLs.** For the DSL users deeply embedded DSLs are not ideal, and we argue that it is *hard to program* in them. The reification in the host language inevitably leads to abstraction leaks (§4.2) such as convoluted interfaces, difficult debugging, incomprehensible type errors, run-time error reporting, and others (see 4.2).

For the DSL author developing a DSL in the deep embedding is *not easy*. Unlike with external DSLs where the difficulty comes from the amount of work required to develop the language ecosystem, in deep embeddings it is difficult to retrofit reification in the host language. The DSL author is required to exploit complicated type system features to minimize the abstraction leaks caused by the deep embedding.

Deeply embedded DSLs exhibit *high performance*. 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, 2012, Rompf et al., 2013b] that lead to orders-of-magnitude improvements in program performance, and multi-target code generation [Brown et al., 2011]. Another advantage of deeply embedded DSLs is their compilation at host language run-time. Compilation at run-time allows for dynamic compilation [Auslander et al., 1996, Grant et al., 2000]: values in the host language are treated as constants during DSL compilation. Dynamic compilation can improve performance in certain types of DSLs (e.g., linear algebra and query languages).

**Comparison summary.** We summarize our discussion in Table 1.1. We can see that none of the DSL kinds is ideal for both DSL users and DSL authors, and exhibits high-performance. For this reason, depending on the domain that is being targeted by the language, some kinds of DSLs might be more suitable than the others.

Table 1.1 – Compares different DSL kinds with respect to ease of programming and performance. The sign ✓stands for "good" and the sign X stands for "bad".

| | External | Shallow | Deep |
|---|---|---|---|
| For DSL Users | ✓ | ✓ | X |
| For DSL Authors | X | ✓ | X |
| Performance | ✓ | X | ✓ |

**Choosing the right DSL kind.** Some DSLs greatly benefit from extracting the domain knowledge. Typically, languages for domains with well defined transformation rules (e.g., relational algebra, linear algebra, logical formulas, etc.) benefit the most as those rules can be used to define the space of possible transformations. The DSL optimizer can explore the space of possible executions and find the optimal one.

For languages whose domain allows transformations based on the domain knowledge external and deeply embedded DSLs are a good fit. With those approaches the DSL author can extract the domain knowledge from programs and use it for optimizations. Some DSL authors choose to use the deep embedding (e.g., OptiML [Sujeeth et al., 2011]) and some external DSLs (e.g., WebDSL [Groenewegen et al., 2008]). Both of these approaches are important–discussing which DSL kind is a better choice is out of the scope of this thesis.

Shallow embeddings, on the other hand, are a good fit for languages where exploiting domain knowledge is not beneficial and where DSL users need features of general-purpose programming languages. Good examples of such DSLs are languages for generating content in formats like JSON and XML, testing frameworks, and Actors [Haller and Odersky, 2009].

## 1.2 Importance of Support for DSLs

To allow both high-level of abstraction and high program performance it is necessary enable wide adoption of domain-specific languages. However, from Table 1.1 we see that support for domain-specific languages is not ideal. External DSLs require tremendous amounts of work to be implemented, deep embeddings are not ideal in terms of DSL user experience and DSL author productivity, and shallow embeddings lack in ways to remove the abstraction overheads.

For wide adoption of domain specific languages it is imperative to improve support for DSLs. It is necessary to: *i)* have good experience for the DSL users in all kinds of DSLs, *ii)* have good infrastructure for building external and deeply embedded domain specific languages, and *iii)* a way to remove the abstraction penalty of shallowly embedded DSLs.

In the recent years *language workbenches* [Fowler, 2005] such as Spoofax [Kats and Visser, 2010] and Rascal [Klint et al., 2009, van der Storm, 2011] have been designed to generate large parts of the language ecosystem based on a declarative specification. With language workbenches it is possible to generate the parser, type-checker, name binding logic [Konat et al., 2013], IDE support [Lorenzen and Erdweg, 2013], and debuggers (for a detailed overview of language workbenches see work by Erdweg et al. [Erdweg et al., 2013]).

Support for deeply embedded domain-specific languages has improved the DSL user interface and ease of development. Frameworks like Forge [Sujeeth et al., 2013a], similarly to language workbenches, allow generating the deep embedding from a declarative specification. Scala-Virtualized [Rompf et al., 2013a] proposes overriding host language constructs to better support deeply embedded DSLs. Svenningsson and Axelsson [Svenningsson and Axelsson, 2013] propose combining shallow and deep embeddings for better user experience. For detailed comparison of these approaches see §12.

Removing abstraction overheads in shallow embeddings can be roughly divided into two categories: *i)* automatic schemes where compiler tries to optimize the code without user or author guidance, and *ii)* DSL author guided schemes for optimization where the DSL author declares how to remove abstraction overheads.

The automatic schemes for optimization can not match the performance of hand-optimized code [Rompf et al., 2013b]. Automatic optimizations must rely on heuristics that are designed to perform the best on all programs. In many cases what is best for the average case is not the best solution for the given domain-specific language.

Guided schemes for removing abstraction overheads allow the DSL author and the user to precisely control how abstraction overheads are minimized. From this category, we emphasize multi-stage programming [Taha and Sheard, 1997, Taha, 2004], and schemes based on partial evaluation [Jones et al., 1993, Brady and Hammond, 2010]. For the detailed overview see of approaches and their comparison see §17.8.

## 1.3 Language Support for Embedded Domain-Specific Languages

In this dissertation we improve the language support for embedded domain-specific languages. The thesis is divided in three parts: *Part I* improves on DSL user experience in deeply embedded DSLs, *Part II* describes automation of the deep embedding development in order to minimize the effort required by the DSL author, and *Part III)* describes a method for improving performance of shallow DSLs by removing abstraction overheads in a user-controlled and predictable way.

### 1.3.1 Part 1: Improving User Experience in the Deep Embedding

We improve user experience in the deeply embedded languages by exploiting the complementary nature of shallow and deep embeddings. We use the shallow embedding for program development when good DSL user interface is important and not the performance. In production, when performance is important.

We define an automatic translation based on *reflection* from shallow programs into the corresponding deep programs (§5). The translation is configurable to support different types of deep embedding that we show in Yin-Yang (§6). Then we introduce DSL reification at host language compile-time (§7) to improve error reporting (§8) and reduce run-time overheads in the deep embedding. Finally, we show the overview of the framework (§10).

### 1.3.2 Part 2: Automating Development of the Deep Embedding

In the second part of the thesis we ease and automate development of the deep embedding:

- By re-using the shallow-to-deep translation to allow user friendly transformations in the deep embedding (§13).
- By re-using the shallow-to-deep translation to automatically generate the deep embedding based on the shallow embedding (§14).
- By providing an abstraction in the deep embedding for automatically managing dynamic compilation of deeply embedded DSLs (§15). Compilation guards, and code caches are introduced with minimal DSL author intervention.

### 1.3.3 Part 3: Removing Abstraction Overheads in DSLs

In the introductory chapter (§16) we evaluate existing approaches for predictably removing abstraction overheads. Then we show that none of the approaches is ideal for domain-specific languages. Approaches like partial evaluation are not predictable and type-safe, while approaches like staging (both quotation and type based) require code duplication on the part of the DSL author which hampers development of shallow DSLs.

We propose polyvariant staging (§17) for removing abstraction overheads in a predictable, type-safe, and concise way. We enhance the programs with information about polymorphic binding-time analysis [Rytz and Gengler, 1992]. Then we provide a small set of annotations on program types to achieve predictable partial evaluation. The programmer has complete control over partial evaluation but since we only use types to propagate binding-time information, user programs do not require quotation and polymorphic binding-time analysis obviates the need to duplicate DSL code.

## 1.4 Terminology

In §3 we introduce a term *direct embedding* for a particular kind of shallow embeddings. For embedded domain-specific languages we use the abbreviation EDSL, however, in cases where it is clear from the context we simply use DSL. For kinds of DSLs we interchangeably use terms: *i)* deep embedding and deeply embedded DSL, *ii)* shallow embedding and shallowly embedded, and *iii)* direct embedding and directly embedded DSL. We also interchangeably use terms: *i)* DSL user, user, and programmer, and *ii)* DSL author and author.

For the programs whose execution results have influence on human or machine decisions and ultimately change the physical world we use a colloquial term and say that programs run *in production*.

# 2 Background

In this chapter we provide a brief introduction to the Scala Programming Language [Odersky et al., 2014] (§2.1) to deeply embedded DSLs in Scala (§2.2), and to partial evaluation and staging (§16). This chapter provides necessary background for comprehending this thesis. Throughout the thesis we assume that the reader has basic knowledge about programming languages.

## 2.1 The Scala Programming Language

Scala is a multi-paradigm programming language. It supports object-oriented programming as well as functional programming. The primary target of Scala is the Java Virtual Machine (JVM), although, recently Scala's dialects also target JavaScript.

Object-Oriented Features of Scala

Scala's types are organized in a *type hierarchy*. At the *top* of the hierarchy is the `Any` type with its two subtypes `AnyVal` and `AnyRef`. Type `AnyVal` is the supertype of all primitive types (e.g., `Int`) while `AnyRef` is the supertype of all *reference* types. At the bottom of the hierarchy is the type `Nothing` which has no corresponding instances. Type `Null` is a subtype of all reference types and has a single instance `null`.

Types at the top of the hierarchy define *universal methods* that are available on all their subtypes and therefore all Scala types. An example of such method are methods `equals` and `hashCode` that are used for checking equality between two objects and computing a hash code of an object.

### 2.1.1 Functional Features of Scala

Besides methods Scala provides native support for functions. In Scala functions can be defined as terms with a special syntactic construct (e.g., `(x: Int)=> x + 1`). Functions are internally represented as classes with an `apply` method that defines a function body. For each cardinality (number of parameters) of a function there is a corresponding Scala class. For example, function `(x: Int)=> x + 1` is represented with an anonymous subclass of `Function2`:

```
class anonymous$uid extends Function2[Int, Int] {
  def apply(x: Int) = x + 1
}
```

Functions and methods can be *curried*: they can have multiple *parameter lists*. A curried function is simply a function that returns another function. For methods, Scala has special syntax to support multiple parameters in the definition. The following example shows a curried method and a curried function:

```
def fill(v: Int)(size: Int) // curried method
(v: Int) => (size: Int) => fill(v)(size) // curried function
```

Scala function evaluation first executes function arguments and then the function body (*by-value* evaluation order). However, it is possible, at the method definition site, to declare method parameters as evaluated *by-name*: the function is evaluated before its arguments. To achieve by-name evaluation the type of a parameter must be perpended with `=>`. For example, a by-name parameter of type `Int` is written as `p: => Int`.

Scala supports *pattern matching* over terms. In Scala `case classes` are used to define that classes can be *deconstructed* with pattern matching. A case class is a data-type that, among other things, has a synthesized factory method `apply` and an *extractor* for deconstruction. Extractor is a method named `unapply` that is used by pattern matching to deconstruct an object. By using methods for deconstruction Scala decouples the deconstruction of a type and its data representation [Emir et al., 2007].

The signature of an extractor method for a case class must correspond to the factory method that constructs the object. If the constructor is defined as `(T1,...,Tn) => U)` the deconstructor must have the signature `Any => Option[(T1,...,Tn)]`. Type `Option[_]` is an equivalent of the `Maybe` monad for Scala.

In deep DSLs it is common to perform deconstruction of internal nodes in order to transform them. In the following example we show a definition of the IR node that represents constants, and how we can use pattern matching on it:

```
case class Const[T](v: T) extends Exp
val node = Const(42) // factory method defined by the case class
```

```
node match {
  case Const(42) => true  // invokes a synthesized deconstructor
}
```

### 2.1.2 Implicit Parameters and Conversions

Value, object, and method definitions in Scala can be declared as *implicit*. Marking a definition as implicit allows the Scala compiler to use this method as an *implicit argument* or for *implicit conversions*. The definition is declared as implicit by writing a keyword implicit in front:

```
implicit val stream: PrintWriter
```

Methods in Scala can have implicit parameters. Only the parameters in the last parameter list of a curried method can be implicit. They are declared implicit by writing the keyword `implicit` in the beginning of the parameter list. Inside the method definition implicit parameters are further treated as implicit definitions. For example, a code generation method can accept an implicit `PrintWriter` by declaring it as implicit:

```
def emitNode(sym: Sym, rhs: Def)(implicit stream: PrintWriter) = {
// in the method body stream is treated as implicit
}
```

The implicit parameters can be passed explicitly by the programmer, however, if they are omitted the Scala compiler tries to find an implicit definition that can satisfy that parameter. The `emitNode` method can be called in two ways:

```
emitNode(sym, rhs)(stream) // parameter passed explicitly
emitNode(sym, rhs)         // parameter added by the compiler
```

**Type classes.** Type classes are in Scala [Oliveira et al., 2010] introduced as a combination of traits, implicit definitions, and implicit parameters. A type class declaration is achieved by defining a trait with the interface for that type class, e.g., `Numeric[T]`. Then a type class instance is defined as an implicit definition that provides a type-class instance (i.e., instance of a trait) for a concrete type:

```
implicit val intNumeric: Numeric[Int] = new IntNumeric()
```

To *constrain* a type parameter of a method or a class one adds an implicit parameter that requires presence of a type-class instance. For example, constraining `T` to be `Numeric` is achieved by requiring an implicit instance of `Numeric` for type `T`:

```
def sum[T](xs: List[T])(implicit num: Numeric[T]): T
```

The same can be expressed with the shorthand notation for declaring type classes:

```
def sum[T: Numeric](xs: List[T]): T
```

**Implicit conversions.** Scala allows user-defined implicit conversions, besides the standard implicit conversions that are applied to primitive types. Implicit conversions are defined as implicit method definitions that have a single non-implicit parameter list and an optional implicit parameter list. An implicit conversion for an abstract type `Rep[Int]` can be defined as:

```
implicit def intOps(x: Rep[Int]): IntOps = new IntOps(x)
```

The implicit conversions are applied when a given term has an incorrect type. The compiler then tries to find an implicit conversion method that would "fix" the program to have correct types. Implicit conversions are categorized as: *i)* value conversions that happen when the *expected type* of term is not satisfied, and *ii)* method conversions that happen when the invoked method does not exist on a type. Given that `IntOps` has a method `+` the following example demonstrates both types of conversion:

```
val y: Rep[Int] = ...
val ops: IntOps = y // value conversion
y + 1               // method conversion
```

**Extension methods.** Implicit conversions subsume the mechanism of extension methods. In Scala an extension method is introduced by providing a wrapper-class (e.g., `IntOps`) that introduces the method and an implicit conversion that applies the wrapper. A class and an implicit conversion can be more concisely written as an `implicit class`.The following examples shows the implicit class that adds the `+` method to `Rep[Int]`:

```
implicit class IntOps(lhs: Rep[Int]) {
  def +(lhs: Rep[Int]): Rep[Int] = Plus(lhs, rhs)
}
```

### 2.1.3 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.

In this work we use a particular flavor of Scala macros called *black-box 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 in this thesis 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 shallow embedding. We will use this type of macros for defining DSL boundaries, e.g., the following snippet is how we will define DSL programs: Figure 2.1:

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

## 2.2 Deep Embedding of DSLs in Scala

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 2.1 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 (e.g., abstract type `Rep`) 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`.

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)
} // after execution returns a regular Scala Vector(3,6)
```

Figure 2.1 – Minimal EDSL for vector manipulation.

Operations on `Rep[T]` terms are added by implicit conversions (as extension methods) 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 project Delite [Brown et al., 2011, Sujeeth et al., 2013b] for building DSLs that support heterogeneous parallel computing. EDSLs developed with Delite cover domains such as machine learning, graph processing, data mining, etc.

## 2.3 Partial Evaluation and Multi-Stage Programming

**[TODO: ETA Monday 9th ]**

# Part I

# Improving User Experience with Deep Embeddings

# 3 Introduction: Concealing the Deep Embedding of DSLs

In §1 we introduced domain-specific languages and how they can be embedded into a host language. Then we discussed strengths and weaknesses of deep and shallow embeddings. In this section we compare deep and shallow embeddings and then show how they can be combined in order to keep all the strengths and cancel-out the weaknesses with embedded DSLs (EDSLs).

Deep EDSLs intrinsically *compromise programmer experience* by leaking their implementation details (§4.2). 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 (§4.2).

Shallow embeddings typically suffer less from *linguistic mismatch* than deep embeddings: 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.

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

[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].

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 §4.2, this difference in types is at the heart of many of the inconveniences associated with deep embeddings such as long compilation times, execution overheads, and inability to restrict the host language constructs. How then, can we conceal this fundamental difference?

In Forge [Sujeeth et al., 2013a], 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 work 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:

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

In §5 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).

Once we "broke the ice" by using reflection it becomes simpler to further improve the deep embeddings. Yin-Yang enables *domain-specific error reporting* at host-language compile time by compiling a DSL program during host-language compilation. It *restricts* the embedded language by providing an additional verification step for producing comprehensible error messages, and reduces run-time overheads by compiling the deep programs at host language compile-time.

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 §5.
- It allows choosing different deep embedding back-ends with simple configuration changes. We discuss different deep embeddings supported by Yin-Yang in §6.
- It improves error reporting (§8) in the direct embedding by: *i)* allowing domain-specific error reporting at host language compile-time (§8.2) and *ii)* restricting host language features in the direct EDSL based on the supported features of the deep DSL (§8.1).
- It reduces the run-time overheads of the deeply embedded DSL programs (§9). The deep embeddings reify their IR before execution at runtime and thus impose execution overheads. For DSLs which are fundamentally not staged Yin-Yang uses *compile-time reification* to compile DSLs at host language compile time. For staged DSLs (compiled at host language run-time) Yin-Yang avoids re-reification of programs by storing them into a cache.

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., 2013b] and OptiML [Sujeeth et al., 2011]. In all programs combined the direct implementation obviates 101 type annotations related to the deep embedding.

We use Yin-Yang as to introduce a user-friendly frontend for the Slick DSL [Typesafe]. This case study shows that developing a front-end for existing DSLs requires little effort and that developing an API with Yin-Yang takes far less time than developing the deep embedding. The complete evaluation is presented in §11.

# 4 Motivation: Abstraction Leaks in the Deep Embedding

The main idea of this work 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 [Svenningsson and Axelsson, 2013, Lokhorst, 2012, Carette et al., 2009, Guerrero et al., 2004], but this is always reflected in the type signatures. In the context of Scala, there are additional problems with type inference and implicit conversions that we discuss in §4.2.7.

Figure 4.1 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. This EDSL is an idiomatic library in Scala and displayed type errors are comprehensible. Finally, in the direct embedding, all terms directly represent values from the 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

```
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 4.1 – The interface of a direct EDSL for manipulating numerical vectors.

performance. Consecutive vector summations would perform much better if they were implemented with a simple while loop.

## 4.1 The Deep Embedding

For the DSL from Figure 4.1, 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 4.2.

In the `VectorDSL` interface every method has an additional implicit parameter of type `SourceContext` and every generic type requires an additional `TypeTag` type class[1]. The `SourceContext` contains information about the current file name, line number, and character offset. `SourceContext`s are used for mapping generated code to the original program source. `TypeTag`s 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 definitions that are provided in the DSL.

[1] `SourceContext` and `TypeTag` are only an example of source information and run-time type information. A particular DSL can use other types but they would still be used in the similar way.

```
trait VectorDSL extends Base {
  val Vector = new VectorOps(Const(Vector))

  implicit class VectorOps(o: Rep[Vector.type]) {
    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)
  }

  // Elided IR constructors of ‘map‘, ‘data‘, ‘fromSeq‘, and ‘range‘
  case class VectorFill[T:TypeTag](v: Rep[T], s: Rep[Int],
    sc: SourceContext)
  def vector_fill[T:Numeric:TypeTag](v: Rep[T], size: Rep[Int])
    (implicit sc: SourceContext): Rep[Vector[T]] =
    VectorFill(v, size, sc) // IR node construction

  case class VectorPlus[T:TypeTag](lhs: Rep[T], rhs: Rep[T],
    sc: SourceContext)
  def vector_plus[T:TypeTag](l: Rep[Vector[T]], r: Rep[Vector[T]])
    (implicit sc: SourceContext): Rep[Vector[T]] =
    VectorPlus(l, r, sc) // IR node construction
}
```

Figure 4.2 – A deep EDSL for manipulating numerical vectors based on LMS.

## 4.2 Abstraction Leaks in the Deep Embedding

The the deep embedding programs construct an IR instead of the values in the embedded language and this inevitably leaks to the users. In this section we discuss how DSL interfaces are convoluted in the deep embedding (§4.2.1), how debugging is made difficult (§4.2.2), how type errors can become incomprehensible (§4.2.3), how it is not possible to restrict certain host language constructs (§4.2.4), how domain-specific error reporting can only be achieved at run time (§4.2.5), how run-time compilation at host-language runtime creates execution overheads (§4.2.6), , and how what are the problems specific to Scala in deep embeddings (§4.2.7).

### 4.2.1 Convoluted Interfaces

The interface of the EDSLs have `Rep[_]` types in their method signatures in order to abstract over the IR construction. These `Rep[_]` types inevitably leak to the user through code documentation, auto-completion, etc. The user defined functions, given that it has to work with DSL constructs, must also have `Rep` types in the interface, further convoluting the intended interface.

In DSLs that use code generation, method signatures must be enriched with source code information for purposes of debugging (`SourceContext`) and type information for generating right types (`TypeTag`). This information also leaks in the DSL interface. In Scala `TypeTag`s and `SourceContext`s are passed with implicit parameters. This makes the interface harder to understand as the user of the EDSL, who might not be an expert programmer, needs to understand concepts like `TypeTag` and `SourceContext`. A method `map` from the introductory example (§3) with source and type information has two additional implicit parameters.

The convoluted interfaces are well presented in Figure 4.2. Here we see how all methods have additional implicit arguments `SourceContext`, how the method definition is placed in an `implicit class` instead on the type it self, and how it is necessary to have additional methods for defining the organization of methods is

```
def map[U](fn: Rep[T => U])
  (implicit sc: SourceContext, tpe: TypeTag[U]): Rep[Vector[U]]
```

### 4.2.2 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, user programs in the reification phase only instantiate the IR nodes. In the classical debugging mode this leads to difficulties as: *i)* users inspecting variable values will be faced with IR nodes, *ii)* the control flow follows all branches in the host language constructs as they get reified, and *iii)* stepping into the DSL operations will only display reificaiton logic.

Debugging generated code or an interpreter is more difficult as users cannot relate the debugger position to the original line of code. The domain-specific and general purpose optimizations applied to the program will likely reorder instructions and rename variables.

The only way to achieve debugging is to make exact maps from the generated code to the deep program and implement a specialized debugger. The debugger must track exact maps between the source code and the generated code. This requires extra effort and decreases DSL author productivity.

### 4.2.3 Convoluted and Incomprehensible 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.

The errors get more involved when artifacts of language virtualization leak to the user:

```
val x = HashMap[Int, String](1 -> "one", 2 -> "two")
x.keys()
```

yields an error message with `SourceContext` parameters:

```
error: not enough arguments for method keys: (implicit pos
: scala.reflect.SourceContext)Prog.this.Rep[Iterable[Int]].
Unspecified value parameter pos.
  x.keys()
       ^
```

### 4.2.4 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.

### 4.2.5 Domain-Specific Error Reporting at Runtime

Domain-specific languages often provide additional program analysis and verification beyond the type system. DSLs perform verification if data-sources are present [McClure

et al., 2005, Zaharia et al., 2012], operations are supported on a given platform [Typesafe], whether multi-dimensional array shapes are correct [Ureche et al., 2012], etc. Ideally, error reporting with the domain-specific analysis should be performed at host language compile-time in order to avoid run-time errors in production and to reduce the time between the error is introduced and detected.

Deep EDSLs do not support compile-time error reporting due to their execution at host language run time. As a consequence, users must execute the DSL body in order to perform domain-specific error detection. This can lead to accidental errors in production code, requires more time find the errors as the error reporting requires running tests, and errors are not integrated into the host language error reporting environment.

### 4.2.6 Runtime Overheads of DSL Compilation

The deeply embedded DSLs are compiled at run time when the code is executed. This compilation introduces overheads on the first execution of the program as well as subsequent executions. The first execution has a larger overhead as the DSL needs to be fully compiled, and the overhead of subsequent executions depends on the implementation of the deep embedding. Some deep embeddings recompile their programs every time [Rompf and Odersky, 2012, Typesafe] which can lead to significant execution overheads [Shaikhha and Odersky, 2013]. Others support guards for re-compilation which lower the overheads but introduce additional constructs in the user programs further leaking deep embedding abstractions.

### 4.2.7 Abstraction Leaks in the Deep Embedding Specific to Scala

Scala specific features like *weak least upper bounds* [Odersky et al., 2014] and *type erasure* lead to further abstraction leaks in the deep embedding. In this section we discuss those and show how they affect the DSL user.

**Weak least upper bounds.** The Scala language introduces *weak type conformance* where primitive types can conform although they are not in a subtyping relation. According to the language specification [Odersky et al., 2014], a type $T$ weakly conforms to $U$ ($T <:_\omega U$) if $T$ is a subtype of $U$ ($T <: U$) or $T$ precedes $U$ in the following ordering:

$$\texttt{Byte} <:_\omega \texttt{Short} <:_\omega \texttt{Int}$$
$$\texttt{Char} <:_\omega \texttt{Int}$$
$$\texttt{Int} <:_\omega \texttt{Long} <:_\omega \texttt{Float} <:_\omega \texttt{Double}$$

The weak least upper bounds are computed with respect to weak conformance: *i)* when inferring the return type of conditional statements (e.g., `if` and pattern matching), and *ii)*

in type inference of type arguments. For example, term `if (cond) 1 else 1.0d` has a type `Double`. Weak least upper bounds are computed before the numeric conversions so the `then` branch of the previous term is widened to `1.0d`.

In case of the deep embedding there are no weak least upper bounds applied as term types are never primitive types as they represent the IR. This leads to inconsistent behavior with the host language. For example, type checking

```
val res = if(positive) 1 else -1.0
res * 42
```

leads to the following type error:

```
error: value * is not a member of Prog.this.Rep[AnyVal]
res * 42
    ^
```

This behavior in the deep embedding can not be improved with implicit conversions as the programs are type correct and implicit conversions are triggered only for incorrect programs.

**Type erasure.** Scala introduces *erasure* [Odersky et al., 2014] of abstract types to the base reference type (`AnyRef`). As a result, overloaded methods with apparently different argument types erase to the same signature. Having two methods in the same scope with the same signature is illegal and leads to compilation errors.

In the deep embedding all types are abstract and erase to the type `Object` on the JVM. Consequently methods that could be defined in the direct embedding erase to the same signature and become illegal. Defining methods

```
def from(json: Rep[String]): Rep[Vector[Double]]
def from(bytes: Rep[Array[Byte]]): Rep[Vector[Double]]
```

yields an compilation error. This code would otherwise be valid in the host language.

The DSL authors are left with two options: *i)* to rename one of the methods and diverge from the original design (e.g., rename `from` to `fromBytes`, or *ii)* to introduce additional implicit parameters which will disambiguate the two methods after erasure

```
def from(json: Rep[String]): Rep[Vector[Double]]
def from(bytes: Rep[Array[Byte]])
  (implicit p: Overloaded): Rep[Vector[Double]]
```

where implicit argument `Overloaded` is always present in scope. Adding implicit

arguments further convolutes the interface and increases compilation times thus causing additional abstraction leaks.

# 5 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, such as types and operations, from the direct embedding into their deep counterparts.

Figure 5.1 shows the translation pipeline of Yin-Yang. Language virualization is covered in §5.1 and DSL intrinsification in §5.2. The whole translation is explained on a concrete example in §5.2.4. Finally we argue for correctness of the translation in §5.2.5.

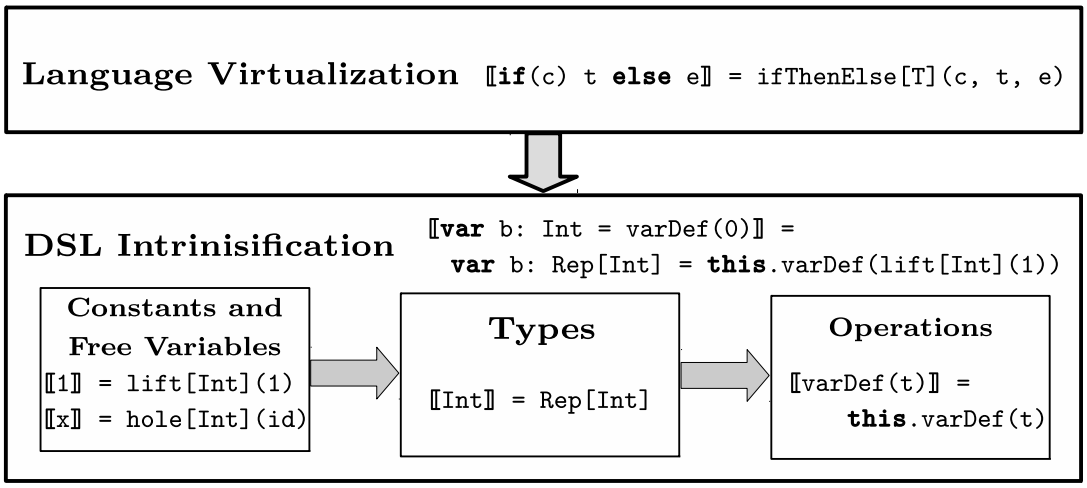


Figure 5.1 – Translation from the direct to the deep embedding.

## 5.1 Language Virtualization

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 of Carette et al. [Carette et al., 2009] and Rompf et al. in the modified Scala compiler named Scala-Virtualized [Rompf et al., 2013a].

Yin-Yang follows the ideas of Scala-Virtualized but virtualizes all Scala language constructs that appear as expressions; uses macros of unmodified Scala to virtualize its intrinsics. Practice has shown that DSL authors are reluctant to use a modified compiler and that for the wide adoption of embedded DSLs it is important to provide a solution based on unmodified Scala.

Compared to Scala-Virtualized we translate additional language constructs: function/method definition and function application, exception handling, and all kinds of value-binding constructs (i.e., values, lazy values, and variables). Translation rules for supported language constructs are represented in Figure 5.2 with $[\![t]\!]$ denoting the translation of a term $t$. In some expressions the original types are introspected and used as a type argument of the corresponding virtualized method. These generic types are later translated to the deep embedding during the DSL intrinsification phase.

Defined translation rules convert language constructs into method calls where each language construct has a corresponding method. The signature of each method is partially defined by Yin-Yang. Method names, the number of type parameters and the number of type arguments are predefined while types of arguments and return types are open for the DSL author to define (§6).

Binding of the translated language constructs to the corresponding methods in the deep

**Function Virtualization**

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

**Method Virtualization**

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

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

**Control Constructs**

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

$$\frac{\Gamma \vdash \textbf{try } t_1 \ \textbf{catch } t_2 \ \textbf{finally } t_3 : \ T}{[\![\textbf{try } t_1 \ \textbf{catch } t_2 \ \textbf{finally } t_3 \ ]\!] = \texttt{try}[T]([\![t_1]\!], \ [\![t_2]\!], \ [\![t_3]\!])}$$

$$[\![\textbf{while}(c) \ b]\!] = \texttt{whileDo}([\![c]\!], \ [\![b]\!]) \qquad [\![\textbf{do } b \ \textbf{while}(c)]\!] = \texttt{doWhile}([\![c]\!], \ [\![b]\!])$$

$$[\![\textbf{return } t]\!] = \texttt{ret}([\![t]\!])$$

**Value Bindings**

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

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

$$\frac{\Gamma \vdash x : T}{[\![x]\!] = \texttt{read}[T]([\![x]\!])} \qquad \frac{\Gamma \vdash x : T}{[\![x = t]\!] = \texttt{assign}[T](x, [\![t]\!])}$$

Figure 5.2 – Rules for virtualization of Scala language intrinsics.

embedding is achieved during DSL intrinsification; language virtualization is agnostic of this binding. Further, in the implementation, all method names are prepended with $ [1]

[1] In Scala it is a convention that user defined method's names should not contain $ characters as those are reserved for the name mangling performed by the Scala compiler.

which avoids collisions with other user functions.

**Functions.** We virtualize function definition and application to support full abstraction over the host language expressions. This allows DSL authors to define how functions are treated by reifying them and optionally providing analysis and transformations over them. For example, DSL authors can define different inlining strategies, perform call graph analysis, or instrument all function calls. With Scala-Virtualized this is not possible as functions are not translated and thus it is impossible to abstract over them.

**Methods.** Method definitions follow a similar philosophy as functions. The difference is that in Scala, the `def` keyword is used to define universal quantification and possibly recursion. This is similar to the `let` and `letrec` constructs in other functional languages. This translation is optional as in some DSLs it is more concise to reuse method application of the host language.

**Control constructs.** We translate all Scala control constructs (e.g.,`if` and `try` to method calls. Scala's type system supports parametric polymorphism, by-name parameters, and partial functions that can model the semantics of all control constructs. How these features are used to model the original constructs is presented in §6).

**Value bindings.** Scala has multiple constructs for value binding: values, variables, and lazy values. Yin-Yang translates definition of all values into methods as well as value accesses. Abstraction over values accesses is necessary for tracking effects in case of variables, access order in case of lazy values, and for instrumentation and verification[2] in case of simple values.

**Universal methods.** 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 without translation.

Yin-Yang virtualizes all methods on types `Any` and `AnyRef`. Method calls on objects are translated into the representation where the `this` pointer is passed as the first argument and, by convention, all methods start with a prefix `infix_`.

This representation is convenient for methods that are defined once for the whole hierarchy as the DSL author needs to define this method only once, as opposed to adding it to each

[2]Spores [Miller et al., 2014] by Miller et al. is an example where simple values should be virtualized for verification purposes.

data type. The caveat with this approach is that in case of methods that are overridden the virtual dispatch must be performed manually by the DSL author.

For DSLs that require extension of these methods we provide an alternative translation of universal methods into the name mangled infix form:

$$[\![ t_1 \ == \ t_2 ]\!] = [\![t_1]\!] \,.\, \$ ==( \, [\![t_2]\!])$$

**The `new` construct.** The new construct of Scala can not be virtualized to a single method as signatures of data-type constructors differ in number of type arguments, the number of arguments and in their types. If the method name was the same for all data types the method would have to have a different type signature based on one of its type arguments. In Scala this would be possible with by using a combination of overloading and implicit parameters but this requires usage of complicated constructs in the deep embedding.

Instead, we rely on the power of the translation, and we virtualize constructor calls to method calls whose name depends on the type that is being constructed:

$$\frac{\Gamma \vdash methodName = \text{"}\texttt{new\_}\text{"} + \texttt{path}(type)}{[\![\textbf{new } type[T](arg)]\!] = methodName[T]([\![arg]\!])}$$

**Not virtualizing class definitions.** Yin-Yang does not virtualize class and trait definitions, including the *case class* definitions. For the given set of DSL compiler frameworks that use Yin-Yang it was hard to identify an abstraction that would allow virtualization of Scala classes.

This limitation, however, does not preclude class virtualization for embedded DSLs. We allow extensions to Yin-Yang that virtualize classes and traits through the use of the reflection API. The drawback of this approach is that DSL authors are required to know the reflection API compared to the simple interface of language virtualization. For now, each framework that uses Yin-Yang defines its own translation scheme.

**Configuring method virtualization.** When we virtualize methods and their application we effectively override all expressions of Scala. In this case the DSL author has to: *i)* write the DSL definition in a way that corresponds to the translation and is not idiomatic to Scala, and *ii)* do additional transformation that removes all combinations of applications over domain-specific operations.

This can be cumbersome and we leave method virtualization as a configuration option that is disabled by default. This way DSL authors can write DSLs in the Scala idiomatic

**Methods on the `Any` type**

$$[\![t_1 \texttt{ == } t_2]\!] = \texttt{infix\_==}([\![t_1]\!], [\![t_2]\!]) \qquad [\![t_1 \texttt{ != } t_2]\!] = \texttt{infix\_!=}([\![t_1]\!], [\![t_2]\!])$$

$$[\![t.\texttt{\#\#}]\!] = \texttt{infix\_\#\#}([\![t]\!]) \qquad [\![t.\texttt{getClass}]\!] = \texttt{infix\_getClass}([\![t]\!])$$

$$[\![t.\texttt{isInstanceOf[T]}]\!] = \texttt{infix\_isInstanceOf[T]}([\![t]\!])$$

$$[\![t.\texttt{asInstanceOf[T]}]\!] = \texttt{infix\_asInstanceOf[T]}([\![t]\!])$$

**Methods on the `AnyRef` type**

$$[\![t_1 \texttt{ eq } t_2]\!] = \texttt{infix\_eq}([\![t_1]\!], [\![t_2]\!]) \qquad [\![t_1 \texttt{ ne } t_2]\!] = \texttt{infix\_ne}([\![t_1]\!], [\![t_2]\!])$$

$$[\![t.\texttt{notify}]\!] = \texttt{infix\_notify}([\![t]\!]) \qquad [\![t.\texttt{notifyAll}]\!] = \texttt{infix\_notifyAll}([\![t]\!])$$

$$[\![t.\texttt{wait}]\!] = \texttt{infix\_wait}([\![t]\!]) \qquad [\![t_1.\texttt{wait}(t_2)]\!] = \texttt{infix\_wait}([\![t_1]\!], [\![t_2]\!])$$

$$[\![t_1.\texttt{wait}(t_2, t_3)]\!] = \texttt{infix\_wait}([\![t_1]\!], [\![t_2]\!], [\![t_3]\!])$$

$$[\![t_1.\texttt{synchronized}[T](t_2)]\!] = \texttt{infix\_synchronized}[T]([\![t_1]\!], [\![t_2]\!])$$

Figure 5.3 – Rules for virtualization of methods on `Any` and `AnyRef`.

way and the `app`/`lam` pairs for DSL operations never appear in the DSL intermediate representation.

### 5.1.1 Virtualizing Pattern Matching

The Scala compiler has a virtual pattern matcher that allows for overloading the semantics of the pattern matching construct. Yin-Yang reuses the functionality of this pattern matcher in combination with DSL intrinsification to allow reasoning about it in DSL compilers. In this section, for purposes of explaining DSL intrinsification, we explain the functioning of the virtualized pattern matcher.

Scala's pattern matching can be interpreted with deconstructors and operations on the `Option` type of Scala. The successful match is represented with the `Some` type which is the monadic return of the `Option` monad and failures with the `None` type which is a monadic *zero* operation. The pattern nesting is represented with the monadic *bind* operation `flatMap`; alternation is represented with the `orElse` combinator on the `Option` monad which represents monadic addition. The semantics of Scala pattern matching can be represented completely with the operations of the zero-plus monad.

The virtual pattern matcher converts Scala pattern matching to the operations on a

user-defined zero-plus monad. A pattern match is virtualized if the object `__match` is defined in the scope. For the original semantics of Scala pattern matching is defined by the object `__match` defined in Figure 5.4. To virtualize pattern matching users can provide their own definitions of methods in `__match`, implementations of the zero-plus monad, and implementations of the case class deconstructors.

```
object __match {
  def zero: Option[Nothing] = None
  def one[T](x: T): Option[T] = Some(x)
  def guard[T](cond: Boolean, then: => T): Option[T] =
    if(cond) one(then) else zero
  def runOrElse[T, U](x: T)(f: T => Option[U]): U =
    f(x) getOrElse (throw new MatchError(x))
}
```

Figure 5.4 – The implementation of the virtualized pattern matcher with the original Scala semantics and with `Option` as the zero-plus monad.

If the object `__match` is in scope a simple pattern match

```
p match {
  case Pair(l, r) => f(l,r)
}
```

is translated into

```
__match.runOrElse(p) { x1: Any =>
  Pair.unapply(x1).flatMap(x2: (Int, Int) => {
    val l: Int = x2._1; val r: Int = x2._2;
    __match.one(f(l, r))
  })
}
```

In case of multiple case clauses

```
p match {
  case Pair(l, r) => f(l,r)
  case Tuple2(l, r) => f(l,r)
}
```

the monadic addition `orElse` is used for matching alternative statements in order:

```
Pair.unapply(p).flatMap(x2: (Int, Int) => {
  val l: Int = x2._1; val r: Int = x2._2;
  __match.one(f(l, r))
}).orElse(
  Tuple2.unapply(p).flatMap(x3: (Int, Int) => {
```

```
    val l: Int = x3._1; val r: Int = x3._2;
    __match.one(f(l, r))
}))
```

Nested pattern matches

```
p match {
  case Pair(Pair(ll, lr), r) => f(f(ll,lr), r)
}
```

are translated into nested calls to `flatMap`:

```
Pair.unapply(p).flatMap(x2: (Int, Int) => {
  val r: Int = x2._2;
  Pair.unapply(x2._1).flatMap(x4: (Int, Int) => {
    val ll: Int = x4._1; val lr: Int = x4._2;
    __match.one(f(f(ll, lr), r))
  })
})
```

Finally the pattern guards are translated into the call to the `guard` function that executes the by-name body of the case when the `cond` statement is satisfied.

## 5.2 DSL Intrinsification

DSL intrinsification maps directly embedded versions of the DSL intrinsics to their deep counterparts. The constructs that we translate (Figure 5.1) are: *i)* constants and free variables (§5.2.1), *ii)* DSL types (§5.2.2), and *iii)* DSL operations in the direct program (§5.2.3).

### 5.2.1 Constants and Free Variables

**Constants.** Constant values can be intrinsified in the deep embedding in multiple ways. They can be converted to a method call for each constant (e.g., $[\![\texttt{1}]\!] = \texttt{\_1}$), type (e.g., $[\![\texttt{1}]\!] = \texttt{liftInt(1)}$), or with a unified polymorphic function (e.g., $[\![\texttt{1}]\!] = \texttt{lift[Int](1)}$) that uses type classes to define behavior and the return type of `lift`.

In Yin-Yang we use the polymorphic function approach for to translate constants

$$\frac{\Gamma \vdash c : T}{[\![c]\!] = \texttt{lift}[T](c)}$$

where $c$ is a constant. We choose this approach as DSL frameworks commonly have a

single IR node for all constants and it is easiest to implement such behavior with a single lift method.

The deep embedding can, given that Scala supports type-classes, provide an implementation of lift that depends on the type of *c*. The DSL author achieves this by providing a type class instance for lifting a set of types (defined by upper and lower bounds) of constants.

In Yin-Yang we treat as constants:

1. *Scala literals* of all primitive types `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`, as well as literals of type `String` (`"..."`), `Unit` (`()`), and `Null` (`null`).
2. *Scala object accesses* e.g., `Vector` in `Vector.fill` is viewed as a constant. This allows the DSL authors to re-define the default behavior of object access. Translating objects is optional as leaving their original semantics simplifies the implementation of the deep embedding, but requires special cases in type translation.

**Free variables.** 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.

Yin-Yang tracks free variables in the DSL scope and translates them into a call to the polymorphic function `hole[T]`. As the arguments Yin-Yang passes the unique identifier for that variable:

$$\frac{\Gamma \vdash x : T \quad x \texttt{ is a free variable}}{[\![x]\!] = \texttt{hole}[T](\texttt{uid}(x))}$$

In Figure 5.5, 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 passed as an argument to the `hole` method (0 and 1 in Figure 5.5). 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.

### 5.2.2 Type Translation

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.

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 $\tau_{\texttt{arg}}, \tau \colon \mathbb{T} \to \mathbb{T}$ where $\mathbb{T}$ is the set of all types and $\tau_{\texttt{arg}}$ and $\tau$ translate types in argument and non-argument positions, respectively.

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 $\tau$ 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. Usages of this translation are shown in §6.2.

**Generic polymorphic embedding.** If instead we choose $\tau$ 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:

$$\begin{aligned} \tau_{\texttt{arg}}(T) &= T \\ \tau(T) &= \texttt{Rep[}T\texttt{]} \end{aligned}$$

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] and the description of the corresponding deep embedding for this translation (§6.3).

**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, 2012] by default and we use it in Figure 5.5. The translation functions are defined as:

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

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[`$B$`]`.

This translation disallows usage of curried functions in the direct programs. If we represent the curried functions as the polymorphic types then functions would appear in the type argument position. To re-introduce the curried functions we provide a translation with the modified rule for translation of functions:

$$\tau(T_1 \Rightarrow T_2) = \tau(T_1) \Rightarrow \tau(T_2)$$

which is correct only when the virtualization of functions and methods is disabled.

**Custom types.** All previous translations preserved types in the type parameter position. The reason is that the $\tau$ 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`.

This translation simply prepends each type with a prefix (e.g., `this`) so that the deep embedding can redefine it:

$$\begin{aligned}
\tau_{\texttt{arg}}(T) &= this.T \\
\tau(T) &= this.T
\end{aligned}$$

Unlike with the previous translations, where the type system of the direct embedding

was ensuring that the term will type-check in the deep embedding, this translation gives no guarantees. The responsibility for the correctness of the translation is on the DSL author. The deep embedding for this translation is presented in §6.5 and we applied this translation for embedding Slick [Typesafe] with great success (§11.2).

**Untyped backend.** If DSL authors want to avoid complicated types in the back-end (e.g., `Rep[T]`), the $\tau$ 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.

**Changing the translation.** By simply changing the type translation, the EDSL author can modify 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.

### 5.2.3 Operation Translation

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 a deep operation for each directly embedded operation.

The `opMap` function in Scala intrinsifies direct EDSL body in the context of the deep embedding. `opMap` can be defined as a composition of two functions: *i)* function `inject` that inserts the direct EDSL body into a context where deep EDSL definitions are visible, and *ii)* `rebind` rebinds operations of the direct EDSL to the operations of the deep EDSL. Function `opMap` is equivalent to the composition of `inject` and `rebind` (written in Scala as `rebind andThen inject`).

**Operation `rebind`.** Operations in Scala are resolved through a *path*. For example, a call to the `println` method of the Scala object `Predef`

```
scala.collection.immutable.Vector.fill(1)(42)
```

has a path `scala.collection.immutable.List.fill`[3].

This phase of the translation translates paths of all operations so they bind to operations in the deep embedding. The translation function can perform an identity operation, prefix addition, name mangling of function names, etc.

[3] Scala packages have a hidden prefix `_root_` that we always omit for simplicity.

**Operation `inject`.** For Yin-Yang and polymorphic embeddings [Hofer et al., 2008] in general we chose to inject the body of our DSLs into the refinement of the Scala component that contains all deep DSL definitions

$$[\![dsl]\!] = \textbf{new}\ \ de\ \ \{\ \ \textbf{def}\ \ \texttt{main}() :\ \ \tau(T)\ \ =\ \ dsl\ \}$$

where *dsl* is the direct DSL program after rebinding and *de* is the component name that is provided by the DSL author that holds all definitions in the deep embedding, and $\tau$ is the type translation.

Alternatively the DSLs can be injected into the scope by importing all DSL operations from the object that contains the deep embedding definitions.

$$[\![dsl]\!] = \textbf{val}\ \ c\ \ =\ \ \textbf{new}\ \ de;\quad \textbf{import}\ \ c.\_;\quad dsl$$

In both cases the corresponding `rebind` function can be left as an identity. If the deep embedding has all the required methods with corresponding signatures all operations will rebind to the operations of the deep embedding by simply re-type-checking the program in the new context.

For example in a simple function application

```
Vector.fill(1)(42)
```

the injection will rebind the operation through the extension methods of Scala. The resulting code will contain wrapper classes that extend DSL types with deep operations:

```
VectorOps(lift(Vector)).fill(lift(1))(lift(42))
```

Finally injection in the DSL scope allows the deep embedding to introduce additional implicit arguments to all DSL operations. The implicit arguments introduced in the direct embedding are treated as explicit in the deep embedding. For example, the `fill` operation can be augmented with the source information from the direct embedding and run-time type representation:

```
VectorOps(lift(Vector)).fill(lift(1))(lift(42))(
  sourceContext("example.scala",1,1), typeTag[Int]
)
```

In Yin-Yang we chose the operation translations that closely match the structure of the direct embeddings. This allows the authors of the deep embedding to use the DSL itself for development of new DSL components. In this case the DSL author can use the deep interface augmented with implicit conversions that all simplify lifting constants, calling operations etc. With such interface the previous example resembles the direct embedding

```
Vector.fill(1)(42)
```

except for the abstraction leaks of the deep embedding (§4.2). This approach requires less code than "naked" AST manipulation [Visser, 2004], e.g:

```
VectorFill(Const(1), Const(42))
```

### 5.2.4 Translation as a Whole

To present the translation as a whole we use an example program for calculating $\sum_{i=0}^{n} i^{exp}$ using the vector EDSL defined in Figure 4.1. Figure 5.5 contains three versions of the program: Figure 5.5a depicts the direct embedding version, Figure 5.5b represents the program after type checking[4] (as the translation sees it), and Figure 5.5c shows the result of the translation.

In Figure 5.5c we see two divergences we omitted from the translation rules for clarity:

- All translated methods are prepended with $. They are added to the code as this makes translation specific methods treated specially by the Scala tool-chain. Methods with $ are invisible in the deep embedding documentation, REPL, and IDE code auto-completion.
- The method `$hole` is not populated with type parameters but it has a call to `$tpe` added as a second argument. This is added as in Scala one can not provide partial type arguments to the method and the idea of Yin-Yang is to leave the definition of `$hole` open to the DSL authors. With `$tpe` added Scala's type-inference can infer the type parameters of method `$hole`. Similarly `$lift` is not provided by the type parameter but the type is provided by the first argument.

In Figure 5.5c, on line 2 we see the DSL component in which the DSL program is injected. On line 3 we see how Scala's `if` is virtualized to `$ifThenElse`. On line 4, in the condition, `n` as it is a captured variable is lifted to the call to `$hole(0,$tpe[Int])` as it is a captured variable and the identifier `0` as it shows as the first variable in the DSL program. On the same line constant `0` is lifted to `$lift(0)`. On line 5 we see how the type of the `val` is translated transformed to `Rep[Vector[Int]]` and how `val` is virtualized. On line 6 `vector.Vector` is lifted as a constant. On line 7 we see how variable `n` is replaced with a hole with the same identifier (`0`). This identifier communicates to the deep embedding that it is the same value. On line 8 the Scala function is virtualized to the `$lam` call. On line 9 the identifier `exp` is translated to

[4]The Scala type-checker modifies the program by resolving identifiers to their full paths, adding implicit parameters, etc.

`$hole(1,$tpe[Int])` (with the identifier `1`) as `n` appears as the second identifier in the program.

```
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 $\sum_{i=0}^{n} i^{exp}$.

```
1  val n = 100; val exp = 6;
2  vectorDSL {
3    if (n > 0) {
4      val v: Vector[Int] =
5        vector.Vector.range(0, n)
6      v.map[Int](x: Int =>
7        math.‘package‘.pow(x, exp)
8      ).sum[Int](
9        math.Numeric.IntIsIntegral)
10   } else 0
11 }
```

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

```
1  val n = 100; val exp = 6;
2  new VectorDSL with IfOps with MathOps { def main() = {
3    $ifThenElse[Int](
4      $hole(0, $tpe[Int]) > $lift(0),{ // then
5      val v: Rep[Vector[Int]] = $valDef[Vector[Int]](
6        $lift(vector.Vector).range(
7          $lift(0), $hole($tpe[Int], 0)))
8      v.map[Int]($lam[Int, Int](x: Rep[Int] =>
9        $lift(math.‘package‘).pow(x, $hole($tpe[Int], 1))
10     ).sum[Int]($lift(math.Numeric).IntIsIntegral)
11   },{ // else
12     $lift(0)
13   })
14 }
```

(c) The Yin-Yang translation of the program from Figure 5.5b.

Figure 5.5 – Transformation of an EDSL program for calculating $\sum_{i=0}^{n} i^{exp}$.

### 5.2.5 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 the appendix 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 a `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.

## 5.3 Translation in the Wider Context

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 DSL 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** is 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.

# 6 Deep Embedding with Yin-Yang

The translation to the deep embedding assumes existence of an adequate deep embedding. This deep embedding should have an interface that corresponds to: *i)* methods emitted by language virtualization, *ii)* translation of constants and free variables, and *iii)* DSL operations defined in the direct embedding.

For each type translation, interface of the deep embedding and IR reification is achieved differently. In this section we describe how to define the deep embedding interface (§6.1) and achieve reification for relevant type translations: *i)* the identity translation (§6.2), *ii)* the polymorphic embedding (§6.3), *iii)* the polymorphic embedding with eager inlining (§6.4), and *iv)* the embedding with custom types (§6.5). Finally, we discuss how the deep embedding is compiled and executed (§6.7). For conciseness, in all embeddings we omit the interface of pattern matching and virtualized universal methods.

## 6.1 Relaxed Interface of the Deep Embedding

The Yin-Yang translation requires a certain interface from the deep embedding. This interface, however, does not conform to the typical definition of an interface in object oriented programming languages. In most translation rules only the method name, number of type arguments, and number of explicit arguments is defined.

The number of implicit arguments and the return type are left open for the DSL author to define. The implicit parameters are added by the type-checker after the operation translation phase if they are present in the deep embedding. For example, if we define the `$ifThenElse` function as

```
def $ifThenElse[T](cond: Boolean, thn: => T, els: => T)(
  implicit tpe: TypeTag[T]): T = \\...
```

after the translation the Scala type-checker will provide the arguments that carry run-time

type information.

This feature can be used in the deep embeddings for various purposes. The ones that were used in DSLs based on Yin-Yang and LMS are: *i)* tracking type information, *ii)* tracking positions in the source code of the direct embedding, *iii)* changing the method return type based on its argument types, and *iv)* allowing multiple methods with the same erased signature (§4.2.7).

## 6.2 Embedding for the Identity Translation

The DSL embedding for the identity translation is not truly a deep embedding. The types are unmodified and, thus, can not represent the DSL intermediate representation. This translations is still interesting as a mechanism for instrumenting language constructs of Scala. The basic interface that contains all the language features, without any additional implicit parameters, is defined in Figure 6.1.

With this embedding the order of execution in all control flow constructs of Scala is preserved with by-name parameters. For example, in `$ifThenElse` both the `thn` and the `els` parameters are by-name and their execution order is defined by the implementation of the method. Similarly, `$whileDo`, `$doWhile`, `$try`, and `$lazyValDef` have their parameters by-name.

The DSL author can instrument Scala language features by re-defining virtualized methods. For example, collecting profiles of `if` conditions can be simply added by overriding the `$ifThenElse` method:

```
def $ifThenElse[T](cond: Boolean, thn: => T, els: => T)(
  implicit sc: SourceContext): T = {
  val (thnCnt: Long, elsCnt: Long) =
    globalProfiles.getOrElse(sc, (0, 0))

  globalProfiles.update(sc,
    if (cond) (thnCnt + 1, elsCnt)
    else (thnCnt, elsCnt + 1))
  if (cond) thn else els
}
```

Achieving the same effect with a facility like macros would require the DSL author to know the reflection API of Scala which involves details about the Scala intermediate representation. Further, the DSL author would be required to write transformers of that intermediate representation.

Supported features that are not of interest in the deep embedding can be removed with an interesting trick. If we implement each method in the deep embedding as a macro

that inlines the method to its original language construct all abstraction overhead of virtualized methods disappears. For example, `$ifThenElse` can be returned to a regular if with

```
def $ifThenElse[T](cond: Boolean, thn: => T, els: => T): T =
  macro ifThenElseImpl[T]

def ifThenElseImpl[T](c: Context)(cond: c.Tree,
  thn: c.Tree, els: c.Tree): c.Tree = { import c.universe._
  q"if ($cond) $thn else $els"
}
```

Yin-Yang provides a Scala component that has methods of all language features overridden with macros that rewrite them to the original language construct. By using this component the DSL author can override functionality of individual language constructs without a need to redefine all other language features.

In Figure 6.1 method `hole` has a slightly different representation than what is presented in §5. Method `hole` has an additional parameter for the run-time type information (`tpe: TypeTag[T]`). This parameter is passed explicitly by Yin-Yang to allow the compiler to infer the type arguments of the method hole. This allows additional freedom for defining the interface of hole (see §6.5).

## 6.3 Polymorphic Embedding

With the generic polymorphic embedding we uniformly abstract over each type in the direct embedding. This abstraction can be used to give different semantics (thus polymorphic) to the deep programs. In the context of the deep DSLs the most common semantics is reification of the intermediate representation.

In Figure 6.2 we show the basic interface for the generic polymorphic embedding. In trait `PolymorphicBase` the type `R[+T]`[1] is the abstraction over the types in the direct embedding. This type is covariant in type `T` and therefore the abstract types have the same subtyping relation as the types in the direct embedding.

In this embedding both function definition and function application are abstracted over. `lam` converts functions from the host language into the functions in the embedded language and `app` returns functions from the embedded language into the host language. This way the DSL author has complete control over the function definition/application in the deep embedding.

[1] Some deep embeddings call this type `Rep`, `Repr`. We use the name `R` as in languages with local type inference this type is omnipresent in method signatures and makes them longer.

```
trait FunctionsInterface {
  // only a subset of function arities is displayed
  def $app[U](f: () => U): () => U
  def $lam[U](f: () => U): () => U

  def $app[T_1, U](f: T_1 => U): T_1 => U
  def $lam[T_1, U](f: T_1 => U): T_1 => U
}

trait IdentityInterface with FunctionsInterface {
  // constants and captured variables
  def $lift[T](v: T): T
  def $hole[T](id: Long, tpe: TypeTag[T]): T
  def $tpe[T]: TypeTag[T]

  // control structures
  def $ifThenElse[T](cond: Boolean, thn: => T, els: => T): T
  def $return(expr: Any): Nothing
  def $whileDo(cond: Boolean, body: => Unit): Unit
  def $doWhile(body: => Unit, cond: Boolean): Unit
  def $try[T](body: => T, catches: Throwable => T, fin: => T): T
  def $throw(t: Throwable): Nothing

  // variables
  def $valDef[T](init: T): T
  def $lazyValDef[T](init: => T): T
  def $varDef[T](init: T): T
  def $read[T](init: T): T
  def $assign[T](lhs: T, rhs: T): Unit
}
```

Figure 6.1 – Interface of the identity DSL.

```
trait PolymorphicBase { type R[+T] }

trait GenericFunctionsBase extends PolymorphicBase {
  // only a subset of function arities is displayed
  def $app[U](f: R[() => U]): () => R[U]
  def $lam[U](f: () => R[U]): R[() => U]

  def $app[T_1, U](f: R[T_1 => U]): R[T_1] => R[U]
  def $lam[T_1, U](f: R[T_1] => R[U]): R[T_1 => U]
}

trait PolymorphicInterface extends GenericFunctionsBase {
  // constants and captured variables
  def $lift[T](v: T): R[T]
  def $hole[T](id: Long, tpe: TypeTag[T]): R[T]
  def $tpe[T]: TypeTag[T]

  // control structures
  def $ifThenElse[T](cnd: R[Boolean], thn: => R[T], els: => R[T]): R[T]
  def $return(expr: R[Any]): R[Nothing]
  def $whileDo(cnd: R[Boolean], body: => R[Unit]): R[Unit]
  def $doWhile(body: => R[Unit], cond: R[Boolean]): R[Unit]
  def $try[T](body: => R[T], catches: R[Throwable => T], fin: => R[T]): R[T]
  def $throw(e: R[Throwable]): R[Nothing]

  // variables
  def $valDef[T](init: R[T]): R[T]
  def $lazyValDef[T](init: => R[T]): R[T]
  def $varDef[T](init: R[T]): R[T]
  def $read[T](init: R[T]): R[T]
  def $assign[T](lhs: R[T], rhs: R[T]): R[Unit]
}
```

Figure 6.2 – Interface of the generic polymorphic embedding.

Compared to the deep embeddings in Scala (e.g., LMS) the DSL author should follow the translation rules for the deep embedding to be applicable to Yin-Yang. Similarly to deep DSLs the DSL operation are added to `R[T]` types in the same manner. In addition the DSL author should: *i)* provide lifting for Scala objects and *ii)* transform the bodies of deep embedding operations.

**Objects in the deep embedding.** Objects in the deep embedding must correspond to the translated objects in the direct embedding. Since we treat objects as constants they are handled by the `lift` method.

An object from the direct embedding should be represented with an appropriate `lift` method and a set of extension methods that represent its operations. For example, to support the `println` method on the `Predef` object the deep embedding must introduce a lifting construct and an extension method on `R[Predef.type]`:

```
def $lift(c: Predef.type): R[Predef.type] = Const(Predef)
implicit class PredefOps(predef: R[Predef.type]) {
  def println(p: R[Any]): R[Unit] = \\...
}
```

To preserve the original API of the deep embedding one can introduce a shortcut for the `Predef` object in the DSL component:

```
val Predef = lift[Predef.type](scala.Predef)
```

This way the deep embedding can be used in a similar way to the direct embedding for the purpose of developing internal DSL components.

**Operations in the direct embedding.** Implementation of the deep embedding operations depends on which rules of virtualiztion are applied. If we apply method virtualization in the DSL programs the bodes of the deep embedding operations should be transformed with the same transformation. For example, an identity method in the deep embedding

```
def id[T](v: R[T]): R[T] = Identity(v)
```

should be transformed into

```
def id[T]: R[T => T] = $lam[T]((x: R[T]) => Identity(v))
```

In practice, the deep DSL operations only reify the deep program and tracking function application of these methods introduces superfluous IR nodes. Further, this transforma-

```
trait InliningFunctionsBase extends PolymorhpicBase {
  def $app[U](f: () => R[U]): () => R[U]
  def $app[T_1, U](f: R[T_1] => R[U]): R[T_1] => R[U]

  def $lam[U](f: () => R[U]): () => R[U]
  def $lam[T_1, U](f: R[T_1] => R[U]): R[T_1] => R[U]
}
```

Figure 6.3 – Interface of the polymorphic embedding with eager inlining.

tion convolutes the implementation of the deep embedding. For these reasons, in most of the DSLs it is common to disable method virtualization. This way the domain-specific operations are always executed in the host language.

## 6.4 Polymorphic Embedding With Eager Inlining

Polymorphic embedding with eager inlining differs from the generic polymorphic embedding in the way host language functions are translated. With this type of embedding functions are left unmodified, and thus executed, in the host language and therefore always executed during DSL compilation. Effectively, this way of treating functions always inlines all functions in the deep embedding embedding.

With the type translation for eager inlining there two possibilities for the DSL author: *i)* use the full language virtualization but disallow curried functions, and *ii)* to completely disable virtualization of functions and allow curried functions. In both of these cases the DSL author must implement the interface `PolymorphicInterface` from Figure 6.2, however, the interface for functions is different.

**Eager inlining with function virtualization.** If the full virtualization is used the deep embedding should provide an interface shown in Figure 6.3. This way the user can track function applications and definitions in the deep embedding with a drawback that curred functions are not allowed.

**Eager inlining without function virtualization.** Without virtualization the interface for functions in the deep embedding is not necessary. The deep embedding will use the functions from the host language. This kind of embedding is the primary choice of DSLs based on LMS[2].

[2]LMS supports both generic polymorphic embedding and embedding with eager inlining. However, in LMS DSLs [Rompf and Odersky, 2012] and tutorials eager inlining is more common.

```
trait CustomFunctionsBase {
  type Function0[U]
  type Function1[T, U]
  def $app[U](f: Function0[U]): () => U
  def $app[T_1, U](f: Function1[T_1, U]): T_1 => U

  def $lam[U](f: () => U): Function0[U]
  def $lam[T_1, U](f: T_1 => U): Function1[T_1, U]
}
```

Figure 6.4 – Interface of the embedding with custom types. The DSL author can arbitrarily override each type in the embedding.

## 6.5 Embedding With Custom Types

In the embedding with custom types the DSL author is required to redefine every type of the direct embedding in the deep embedding. The overriding is achieved by the abstract types of Scala. For example, the type `scala.Boolean` can be overridden with a type `scala.Int` (as it is in the C based languages) inside of the DSL component with:

```
type Boolean = scala.Int
```

The interface of this type of embeddings is equivalent to the interface of the identity embedding (Figure 6.1) except that all types need to be abstract. Further, Scala functions can not be directly converted into abstract types so all functions in the deep embedding must be represented with their nominal equivalents (e.g., `Int => Int` must be represented as `Function1[Int, Int]`). An example of the interface for functions is presented in Figure 6.4. Here, `type Function0[U]` makes the function type abstract.

A common pitfall with naming abstract types is using only their type name (e.g., `Function0`). The problem arises when two different types (with different paths) have the same name. In these cases the DSL author must use the full type path as the abstract type. Yin-Yang can be configured to add a prefix to the types with their full path or only to their name. In all examples, for conciseness reasons, we use the translation that does not use the full path but only the type name.

In the embedding with custom types a common way to reify the DSL IR is to define abstract types with the IR nodes of the deep embedding. In this scheme, in order to preserve well-typedness after the translation each IR node must satisfy the interface of the type from the direct embedding. The methods in the overridden type all methods can perform reification and still return the correct program.

In Figure 6.5 we show how one reifies operations on the `Boolean` type. All nodes in the IR usually have a common subtype (`Exp` in the example). Then the abstract type for

```
trait DSLBase {
  trait Exp // base class for all nodes
}
trait BooleanDSL extends DSLBase {
  type Boolean = BooleanOps
  trait BooleanOps with Exp {
      def &&(y: Boolean): Boolean = BooleanAnd(this, y)
      def ||(y: Boolean): Boolean = BooleanOr(this, y)
      def unary_!: Boolean = BooleanNot(this)
    }
  }

  case class BooleanAnd(lhs: Boolean, rhs: Boolean) extends BooleanOps
  case class BooleanOr(lhs: Boolean, rhs: Boolean) extends BooleanOps
  case class BooleanNot(lhs: Boolean) extends BooleanOps
}
```

Figure 6.5 – Overriding semantics of `Boolean` with the reification semantics for the deep embedding.

`Boolean` is overridden with a trait (`BooleanOps`) that redefines its operations. These redefined operations can now perform reification of the program in the deep embedding. With this embedding all IR nodes of type `Boolean` extend `BooleanOps` and therefore all method implementations have valid types.

The types in the deep embedding can not be expressed as a type function and therefore defining the interface of `hole` and `lift` is different than the other embeddings. Here, for each type in the direct embedding there needs to be a definition that maps the type to the deep embedding.

One way to achieve such type map is to use function overloading in Scala. The deep embedding should have one `hole` and `lift` method for each type in the direct embedding. For example, lifting the type `Int` is achieved with:

```
def $hole(id: Long, tpe: TypeTag[scala.Int]): this.Int =
  new Hole(sym, v) with IntOps
def $lift(v: scala.Int): this.Int =
  new Const(v) with IntOps
```

With this type of embedding the DSL author is free to redefine types arbitrarily thus giving the deep embedding different semantics. The embedding can perform pretty printing or provide a wrapper for another DSL. For further information on such embeddings see [Hofer et al., 2008] and §11.2.

```
trait Executable { =>
  def execute[T](args: Seq[Any]): T
}
```

Figure 6.6 – The trait for that Yin-Yang uses to execute the deep embedding.

## 6.6 Embedding Pattern Matching

## 6.7 The Yin-Yang Interface

Yin-Yang does not require an exact interface for the translation to the deep embedding and therefore there are no `traits` that a DSL author must extend in order to use the translation. The interface is defined by method names defined in the translation and by the types dictated by the type translation.

After the translation a program in the deep embedding should be executed and should produce the same result[3] as the direct embedding. Execution of a program should happen transparently to the DSL, i.e., it should completely resemble the execution in the direct embedding.

Execution of the deep program depends on all the variables that were captured by the direct embedding. Yin-Yang, after the core translation, must assure that the deep embedding gets all the values that were replaced by holes and that the DSL compiler or interpreter can distinguish to which hole does a value belong.

To achieve this Yin-Yang requires that the deep embedding implements the `Executable` interface (Figure 6.6). This interface contains a single method `execute` that is used for execution of the DSL at host language run-time. Yin-Yang simply calls the method `execute` on the translated DSL with the arguments that were converted to holes.

Yin-Yang invokes `execute` with all captured variables sorted in the ascending order by the identifier that is passed to the corresponding `hole` methods. Since the identifiers passed to `hole` are strictly increasing, the DSL can uniquely map arguments to the holes they belong to. For example, the DSL program in the example translation (Figure 5.5c) has two captured variables (`n` and `exp`) with identifiers `0` and `1` respectively. Yin-Yang will, after translation, invoke this DSL with

```
dsl.execute[Int](Seq(n, exp))
```

This way of invoking a DSL is convenient for the DSL author as one needs to implement a single method, however, it is not optimal. Putting arguments in a `Seq` introduces an additional level of indirection and imposes boxing of all the arguments. In short

[3]The result can be different in case of floating-point operations executed on different back-end platforms, however this difference should not change decisions that are made based on the result.

running DSL programs that are executed in tight loops the overhead of boxing can become significant.

To avoid this overhead, before translating to the generic `execute` method, Yin-Yang looks for more specific methods that would correspond to the holes that are passed. If this method is found Yin-Yang will execute that method although it is not in the interface. For example, if a DSL from the example translation ((Figure 5.5c)) would implement a method with a signature

```
def execute(v0:Int, v1:Int): Int
```

this method would be invoked instead.

The deeply embedded DSLs can generate and compile code in the host language, other languages, or be interpreted. We have abstracted over these types of DSLs with a simple interface. With this approach all compilation decisions are left to the DSL author. This simplifies the framework, however, it complicates management of DSL compilation in presence of multiple DSLs.

In the case where multiple DSL frameworks simultaneously invoke compilation of a program it is possible to exhaust memory resources of the running process. For example, each compilation in Scala requires loading the whole compiler and requires significant amounts of memory. When multiple frameworks use compilation without coordination the system resources are easily exhausted.

Yin-Yang currently does not provide a way to coordinate compilation of different DSLs. Resolving this problem has been studied before in context of JIT compilation [Arnold et al., 2000, Kulkarni, 2011] and it falls out of the scope of this work.

## 6.8 Using Yin-Yang for the Deep Embedding

In this section we show how to define a DSL with Yin-Yang, given that we have both the direct and deep embedding that conforms to the required interface. First we describe how to apply the translation (§6.8.1) and then how the DSL user uses Yin-Yang based DSLs.

### 6.8.1 Defining a Translation for a DSL

The direct-to-deep embedding translation is defined as a macro that accepts a single by-name parameter that represents a body of a DSL program. For example, the `vectorDSL` from Figure 5.5 is defined as

```
def vectorDSL[T](body: => T): T = macro vectorDSLImpl[T]
```

```
object YYTransformer {
  def apply[C <: Context, T](c: C)(
    dslType: c.Type,
    tpeTransformer: TypeTransformer[c.type],
    config: YYConfig): YYTransformer[c.type, T]
}
```

Figure 6.7 – Interface to the Yin-Yang translation.

where `vectorDSLImpl` represents the macro implantation.

The macro implementation calls into a single transformer (`YYTransformer`) that implements the whole translation. The `YYTransformer` is configured with: *i)* the type of a DSL component that implements the deep embedding (e.g., `la.VectorDSL`) , *ii)* the type transformer (e.g., generic polymorphic transformer), and *iii)* the configuration object (i.e., whether to virtualize functions, methods, lift Scala objects, etc.). Interface of the object that creates a `YYTransformer` is shown in Figure 6.7.

With the `YYTransformer` object the method `vectorDSLImpl` is implemented as:

```
  def vectorDSLImpl[T](c: Context)(body: c.Expr[T]): c.Expr[T] =
    YYTransformer[c.type, T](c)(
      typeOf[la.VectorDSL],
      new EagerInliningTypeTransformer[c.type](c),
      YYConfig(virtualizeFunctions = false)(
      body)
```

here the type of the `VectorDSL` is provided in with `typeOf` construct, `EagerInliningTypeTransformer` represents the translation for polymorphic embedding with eager inlining, and `YYConfig` defines that functions should not be virtualized.

# 7 DSL Reification at Host-Language Compile Time

To integrate the deep embedding tightly with the host-language it is necessary to have the DSL IR reified at host language compilation-time. With the reified IR domain-specific analysis and error reporting can be integrated with the host language (§8.2). Also, DSLs that do not depend on run-time values for optimizations can be completely compiled at host-language compile time and their generated code integrated with the IR of the host language (§9.

Reifying the DSL IR after the translation requires executing the code that is generated after the direct-to-deep translation. Since the translation happens during program type-checking the executable version of the translated DSL does not exist. To execute this code there are two possibilities:

- **Use compilation.** Since all the captured variables are replaced by holes (§5.2.1) the deep embedding can be compiled down to executable code separately from the rest of the host language program. Then the deep embedding binary can be executed to obtain the domain-specific IR.
- **Use interpretation.** Use the interpreter of the host language trees to obtain the domain-specific IR. Since the deep DSL has no captured variables it is possible to perform interpretation.

Compilation has an upfront cost of compiling the code but the execution is optimal while interpretation does not have an up-front cost but the execution is slower. For Scala and different sizes of DSL programs we evaluate both approaches in §7.3.

## 7.1 Reification by Compilation

The DSL program that is being compiled to the executable during host language type-checking does not have terms that can not be compiled:

- All captured variables have been converted into calls to `hole` by the translation.

- Class and object definitions in the DSL are forbidden by the virtualization phase.

- Usage of definitions (i.e., classes, traits, objects, and methods) that are compiled in the same compilation unit is impossible. The direct embedding does not contain equivalents for these objects and they can not be accepted by operation translation.

Given that the translated DSL body is independent of the current compilation unit it is possible to compile it separately and produce the executable file. For the DSL body the compilation pipeline is executed to the last phase starting from type-checking. Then the DSL body can be executed and we acquire an instance of the DSL with its trees reified.

In Yin-Yang for the purpose of compile-time reification we use the `eval` construct in Scala. The `eval` function accepts the Scala trees and compiles them to bytecode, loads the class that contains the compiled trees and executes it.

In case of the JVM the reification by compilation has a relatively large overhead. Compiling, loading the bytecode, and executing that code in the JVM interpreter has a relatively large initial overhead. Then the bytecode that is loaded must be executed in the JVM interpreter as it is executed for the first time. Finally, the code is executed only once in order to reify the DSL program and discarded afterwards.

## 7.2 Reification by Interpretation

An alternative to compilation is interpretation of the DSL trees. With interpretation an *interpreter* interprets the trees emitted by the direct-to-deep translation to obtain an instance to a reified DSL at host-language compile time. Since the DSL trees do not depend on the definitions in the current compilation unit it is possible to interpret them.

Since the Yin-Yang translation greatly simplifies the executed trees we use a specialized interpreter. In this interpreter we treat only a subset of Scala that is emitted by the DSL translation. Further, the interpreter invokes all functions in the program with reflection. This way the execution of the DSL operations is achieved by executing pre-compiled code and the interpreter executes the DSL programs.

With the specialized interpreter special care must be taken with higher-order functions. The DSL operations that accept functions as arguments must accept function instances while the function bodies must be executed in the interpreter. To this end, to pre-compiled operations we pass function instances that call back into the interpreter code.

## 7.3 Performance of Compile-Time Reification

We compare compilation and interpretation on synthetic DSL programs based on LMS that vary in size from 0 to 100 lines of code. In the benchmark we use programs that reify 5 DSL IR nodes per line of code[1]. For each program we measure the time it takes to instantiate the DSL compiler and reify the IR for a program. Benchmarks use the methodology, as well as hardware and software platform defined in §??.

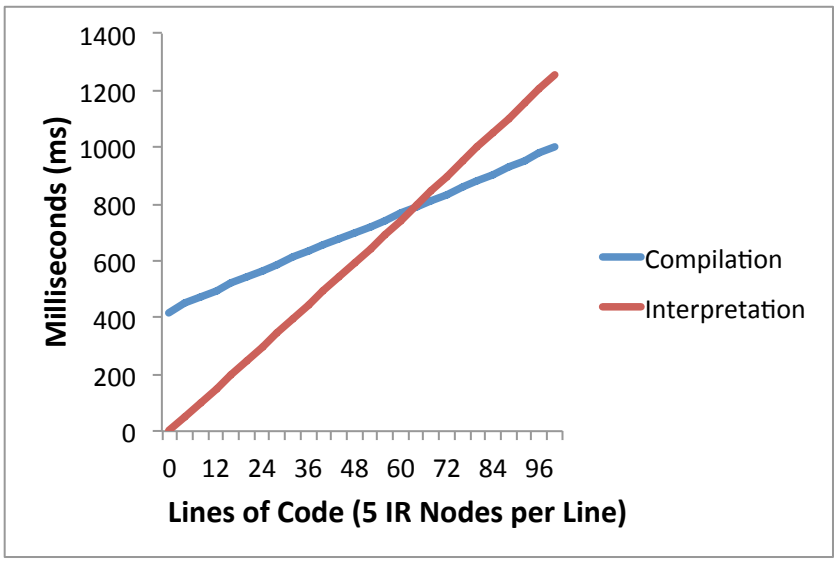


Figure 7.1 – Time to reify DSL programs by compilation and interpretation. In the compiled programs each line of code reifies 5 IR nodes.

In 7.1 we see that reification by compilation-and-execution has a constant overhead of 0.4 seconds after which it reifies 166 lines of code per second. On the other hand, interpretation has small initial overhead of 2ms but reifies 62.5 lines of code per second. Compilation becomes faster than interpretation for programs that have more than 60 lines of code. We explain the small difference in execution speed (without the initial overhead) between compilation-and-execution and interpretation by the manner in which newly compiled code is executed on the JVM. Since the newly compiled code is loaded for the first time the JVM will use its own interpreter to execute it. This interpreter is only slightly faster than the Scala interpreter and significantly slower than the JIT compiled code which reifies 16000 lines per second.

By analyzing the application suites of LMS based DSLs we see that more than 90% of the DSL programs have less than 60 lines of code. To this end we chose interpretation as the default choice for compile-time reification of DSLs in Yin-Yang.

[1]We take 5 as an average number of IR nodes per line. We calculated this number on a small set of applications from the OptiML application suite. This number, however, greatly depends on the coding style of the DSL user and the embedded language.

# 8 Improving Error Reporting of Embedded DSLs

The error reporting of the deep embedding does not blend into the host language in two ways: *i)* the DSL is not able to restrict generic language features of the host language and errors can happen at run-time (§4.2.4) and *ii)* domain-specific errors can only be reported at host-language run-time (§4.2.5). In this chapter we show how once "we broke the ice" with using the host-language reflection improving error reporting becomes a simple addition and how the DSL author can be completely agnostic of the host language reflection API. In §8.1 we show how restricting host language constructs is achieved by omitting operations in the deep embedding and in §8.2 we explain error reporting at host language compile-time.

## 8.1 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 (§4) 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 without the need to know the Scala reflection API.

## 8.2 Domain-Specific Error Reporting at Compile Time

The domain-specific error reporting is performed on the reified DSL program and to do it at host language compile-time we must first reify the DSL IR. To this end Yin-Yang uses the techniques described in §7 to reify the program.

Once the program is reified Yin-Yang calls the `staticallyCheck` method from Figure 8.1 on the DSL component. This method accepts as an argument the `Reporter` interface that the DSL author can use to report messages on the host language console. The DSL author can use the `Reporter` to report errors (the `error` method), warnings (the `warning` method), and to provide informational messages (the `info` method).

All methods in the `Reporter` interface accept the `pos` argument of type `Option[Position]` for displaying where the information should be displayed. If the position argument is not passed the error will be displayed without a position, and if it is provided the message will be displayed for the source code at the position in the same manner as the host language would.

The interface that is provided for error reporting is much simpler than the Scala reflection equivalent. The API is simplified for error reporting in domain-specific languages and

```
trait Position {
  def source: File
  def line: Int
  def column: Int
}

trait Reporter {
  def info(pos: Option[Position], msg: String): Unit
  def warning(pos: Option[Position], msg: String): Unit
  def error(pos: Option[Position], msg: String): Unit
}

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

Figure 8.1 – Interface for domain-specific error reporting in the deep embedding.

does not require the DSL author to comprehend the Scala internals. For the values of the `Position` interface the DSL author can choose an adequate implementation (e.g., one presented by Rompf et al. [Rompf et al., 2013a]).

# 9 Reducing Run-Time Overheads of the Deep Embedding

## 9.1 Introduction: Runtime Overheads

The deep embedding without translation necessarily introduces overheads. As the DSL compilation is performed at host language run-time it incurs additional execution steps. We categorize overheads related to run-time DSL compilation in the following categories:

- **DSL program reification** is the time necessary to reify the program.
- **DSL compilation** is the process after reification that applies domain-specific optimizations and generates the final version of the code.

Since the deep programs are compiled at run-time captured variables are seen by the DSL as compile-time constants. Those captured variables are, then, further used in compilation decisions. The DSL compiler, without user modifying the program, can not distinguish between captured variables and constants. Let's examine a simple Slick DSL program that queries the database for finding all cars with price higher than `minPrice`:

```
def expensiveCars(minPrice: Double): Seq[Car] =
  TableQuery[Car].filter(c => c.price > minPrice).result
```

Here the query must be compiled with every call to `expensiveCars` method as for each call to the method the generated SQL will be different. In Slick both reification and query compilation will happen at every execution of the method `expensiveCars`.

Without translation or modification of the DSL programs the deep embedding must at every program execution at least reify the program. Without users modifying the code, DSL compilers can not distinguish between constants and captured variables. Therefore, in order to see if recompilation is required the deep embedding must reify the IR every time and verify that all DSL compilation-time constants are the same.

Some DSLs solve this by making explicit parametrization of captured variables. This way the captured variables are marked by the programmer and the DSL program is written outside of the scope to avoid re-reification of the program. For example, Slick supports query templates where the user first defines a query that is compiled once and then uses it later. For example, definition of the `expensiveCars` query is:

```
val expensiveCarsCompiled = Compiled((minPrice: Rep[Double]) =>
  TableQuery[Car].filter(c => c.price > minPrice))

def expensiveCars(minPrice: Double): Seq[Car] =
  expensiveCarsCompiled(minPrice).result
```

and its later usage would not recompile the query on every execution. However, we see that the user has to augment the program significantly in order to avoid re-compilation.

In all deep embeddings we have seen, both reification and compilation are performed on every execution. Compilation on every execution, however, could be avoided by modifying the user program to include a unique identifier for each program. With the unique identifier the DSL could only reify the program and compare it to the previous reifications of the same program that are stored in global storage. If they are the same the compiled result could be reused. This way the deep DSLs could avoid full recompilation at every execution.

In §9.2 we measure run-time overheads for reification of LMS based DSLs. We do not measure the case with full compilation as this depends on a DSL at hand. Then, in §9 and §9.3.2, we describe a translation-based solution for minimizing run-time overheads of the deep embedding.

## 9.2 Measuring Run-Time Overheads

We measured the cost of reification in three different IRs: *i)* the core IR of LMS, *ii)* the IR of the Slick DSL, and the *iii)* a synthetic IR that only builds a simple object graph and has no reification overheads for maintaining symbol tables etc. The benchmark first initializes the compiler for Slick and LMS and then it runs a mix of synthetic lines of code where each line reifies an average of 5 IR nodes. Benchmarks use the methodology, as well as hardware and software platform defined in §**??**.

In the benchmark the initialization phase of the reification instantiates the compiler in case of LMS and the table in case of Slick. This phase is relatively cheap of only 1 and 2 most respectively. We notice that the simple IR is far less costly than the IR in existing DSLs. The reason for this difference is that DSLs, besides simple reification, do other operations during reification (e.g., tracking source information, tracking types and simplifications of the IR). Finally, we see that reification in real-world DSLs costs

Table 9.1 – The initialization cost and cost of reification per line of code for the simple IR, Slick, and LMS.

| EDSL | Initialization Cost ($\mu s$) | Reificaiton Cost ($LoC/\mu s$) |
|---|---|---|
| Simple IR | 0 | 0.2 |
| Slick | 2 | 5 |
| LMS | 1 | 24 |

between 5 and 24 $\mu s$ per line of code.

On large computations the reification costs are negligible, however, the overhead is relevant when DSLs are used in tight loops. To put these numbers in perspective for 24 $\mu s$ the same hardware platform can traverse an **[TODO: 1000 ]** element array or sort **[TODO: 100 ]** an element array.

## 9.3 Reducing Run-Time Overheads

Before we describe our solution we will categorize DSLs based on their interaction with captured variables:

- **One stage DSLs.** This is the category of DSLs that never uses run-time values for optimization or compilation decisions. This category of DSLs could be compiled together with the host language, thus, incurring run-time overheads. For this category of DSLs we enable compilation at host-language compile time (§9.3.1).
- **Two-stage DSLs.** This category makes compilation decisions (e.g., optimizations) based on the captured values. These DSLs benefit from compilation at host-language run time. For this category of DSLs we provide a translation that removes the reification overheads (§9.3.2)workflow for deciding whether they will be compile

  Even in multi-stage DSLs run-time compilation is not always necessary. If the DSL program does not capture interesting values, or the captured values are not of interest for compilation decisions, compilation can be executed at host language compile-time. For this case we introduce a DSL author controlled workflow for deciding a compilation-stage (§9.3.3).

### 9.3.1 Avoiding Run-Time Overheads for One Stage DSLs

For this category of DSLs we exploit compile-time reification of Yin-Yang to generate code at host-language compile time. After the compile-time reification of the program Yin-Yang demands from the DSL to generate code. The generated code is then inlined into the block where the original direct DSL program was and that direct DSL program

```
trait Generator

trait CodeGenerator extends Generator {
  def generate(holes: String*): String
}

trait TreeGenerator extends Generator {
  def generate(c: Context)(holes: c.Tree*): c.Tree
}
```

Figure 9.1 – Interface for generating code at host-language compile time.

is discarded.

To achieve compile-time compilation the DSL author must extend one of the two traits from Figure 9.1. The `CodeGenerator` requires from the DSL author to return a string with the generated code. The code should be hygienic (the framework does not provide hygiene) and all captured variables should be replaced with values from the `holes` parameter. Similarly `TreeGenerator` generates directly Scala trees. Both trees and strings are supported since some DSL authors might prefer working with quasi-quotes and trees while the others prefer strings.

### 9.3.2 Reducing Run-Time Overheads in Two-Stage DSLs

In case of two-stage DSLs, ideally, we would reify and compile the DSL only for the first execution and when the captured values that affect compilation decisions are changed. Also, the DSL would decide on its recompilation based on the values that are passed to its `execute` method.

Ideally, we would simply convert the DSL instantiation and reification into a the global value definition and instead of the original direct program invoke `execute` on that global definition. The invocations of the `execute` method would take care of recompilation in case when input values change. For example, a simple direct program

```
vectorDSL { Vector.fill(1000, 1) }
```

would be converted into a definition

```
val prog<UID> = new VectorDSL {
  def main() = lift(Vector).fill(lift(1000), lift(1))
}
```

and the original DSL would be replaced with

```
trait Staged { self: Generator =>
  def compileTimeCompiled: Boolean
}
```

Figure 9.2 – Interface for determining the compilation stage of a DSL at host-language compilation time.

```
prog<UID>.execute[Vector[Int]]()
```

In case of Yin-Yang this is not possible due to Scala limitations. In Scala macros are not allowed to introduce global definitions. Therefore the unique value `prog<ID>` can not be defined from the translation macro.

To overcome this issue Yin-Yang gives DSL programs a unique identifier and stores them in a global `ConcurrentHashMap`. With this modification the previous program becomes

```
val prog = YYStorage.programs.computeIfAbsent(<UID>, { _ =>
  new VectorDSL {
    def main() = lift(Vector).fill(lift(1000), lift(1))
})
prog.execute[Vector[Int]]()
```

where `computIfAbsent` assures thread-safe access and instantiates the DSL only once: the DSL is instantiated when the unique identifier is not found in a map and that is only on the first execution. All issues related to concurrency that might arise from recompilation inside the `execute` method are left for the DSL author to resolve.

### 9.3.3 Per-Program Decision on the DSL Compilation Stage

Compilation of two-stage DSLs still introduces a large compilation overhead in the first run and minor overheads for fetching the DSL in consecutive runs. This is not always necessary as two-stage DSLs can be compiled at host-language run-time when their compilation decisions do not depend on captured values. This happens in two cases: *i)* no values are captured by the DSL program, and *ii)* captured values can not be used for optimization.

In order to compile these two categories at host-language compile time Yin-Yang must get an answer from the DSL compiler: are captured values of interest for optimizations. To this end Yin-Yang provides an interface `Staged` from Figure 9.2. The method `compileTimeCompiled` is invoked during host-language compilation time to determine the stage of a program. The trait `Staged` depends on the trait `Generator` as in order to perform compile-time DSL compilation the DSL must be able to generate code.

If the method `compileTimeCompiled` returns `true` the DSL will be compiled at compile time with no run-time overheads (as described in §9.3.1). If the method returns `false` the DSL will be compiled at run-time (as described in §9.3.2). The decision about compilation stage is domain-specific: it is made in the DSL component by analyzing usage of holes in the reified body.

# 10 Putting It All Together

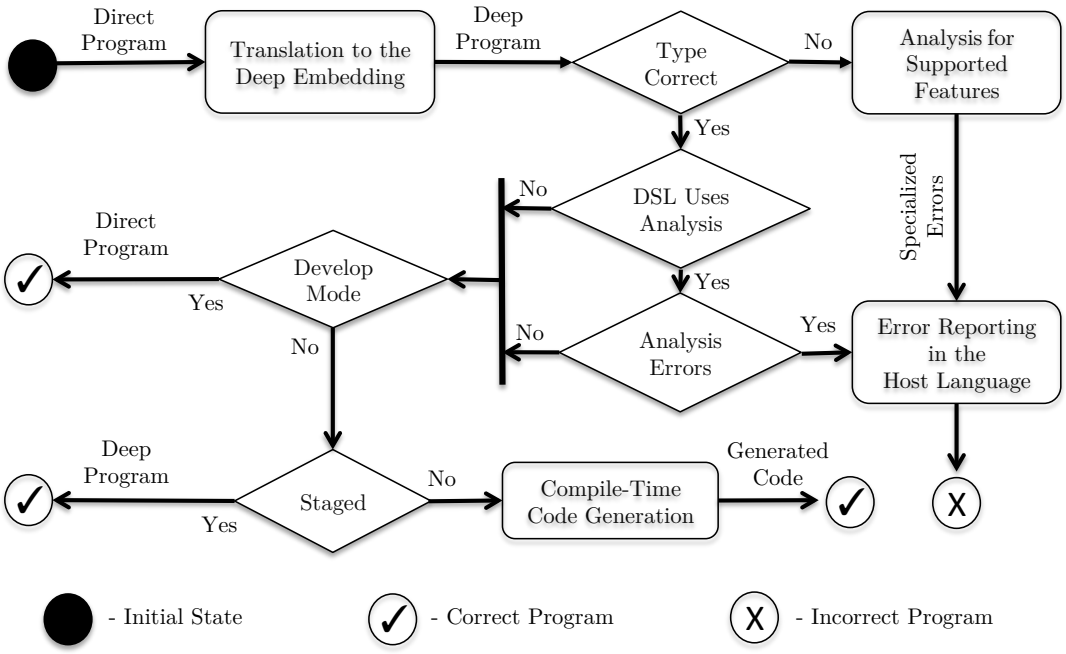


Figure 10.1 – Overview of the Yin-Yang framework: the workflow diagram depicts what happens to the DSL program throughout the compilation pipeline.

In this chapter we provide an overview over the individual steps in Yin-Yang operation. Yin-Yang makes decisions based on the development mode, the user written programs, and the DSL definition. These decisions use reflection to achieve debugging, language restriction, domain-specific error reporting, and DSL compilation at host-language compile time.

Figure 10.1 shows the workflow around the core translation that allows all benefits of the deep embedding during program development and hides all its abstraction leaks.

The direct program is first translated into the deep embedding and type checked. At this point, since the translation is guaranteed to preserve well typedness the only possible source of type errors are unsupported language constructs.

At this point if the program not correct the specialized analysis for supported features is performed. The unsupported features in the direct embedding are detected and reported with the precise position and specialized error messages. The reported errors use the standard reporting mechanism of the host language that works in all environments (e.g., IDEs, Repl, standard error). After, supported feature analysis the compilation is failed.

In the correct program the next decision checks whether the DSL uses domain-specfic analysis. This check is performed by looking into the hierarchy of the DSL component for the `StaticallyChecked` trait. If the DSL uses analysis the DSL is reified at host-language compile time and the `staticallyCheck` method is called. The errors are reported at host-language compile time and use the standard reporting mechanism. After, error reporting the host-language compilation is failed.

If there are no domain-specific errors or the DSL does not use domain-specific analysis Yin-Yang checks whether the DSL is used in development mode. If the development mode is active the original direct program is returned from and the compilation succeeds. Returning the original program allows the DSL user to perform debugging in the direct embedding in by inspecting the host language values.

In the non-development mode the DSL is checked for the stage it should be compiled in. DSLs that do not extend the `Generator` trait are directly sent for run-time compilation as they do not expose the interface to produce the host-language code.

For DSLs that expose generation there are two options:

- The program does not capture any variables in which case it is sent to compilation at host-language compile time.
- The program captures variables. At this point the decision on the compilation stage is based on the DSL and the program. The method `compileTimeCompiled` is invoked on the reified DSL to determine the compilation stage.

If the DSL is compiled on the code generation is invoked on the reified DSL (method `generate`). The code is then inlined in the DSL invocation site and the original direct embedding is discarded. The compile-time compilation completely removes the run-time overheads of the deep embedding.

a

In case of run-time compilation the deep program is augmented with the the logic to

execute it with the parameters ordered by convention and to store the code in a common storage. Storing the code in storage avoids reification of the deep program on every execution.

# 11 Evaluation and Case Studies

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 (§11.1). Finally, we evaluated the ease of adopting Yin-Yang for the existing deep EDSL Slick [Typesafe] (§11.2) 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., 2013a].

## 11.1 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.

## 11.2 Case Study: 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.

# 12 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].

We will classify related work to the approaches that use the shallow embedding as an interface for the deep embedding (§12.1) and approaches that try to improve the deep embedding interface (§12.2).

## 12.1 Shallow Embedding as an Interface

Sujeeth et al. propose Forge [Sujeeth et al., 2013a], 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, DSL authors must learn a new language for EDSL design whereas with Yin-Yang this language is Scala itself. Finally, 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.

We can distinguish following categories of errors that occur after the translation:

- Using a method that from the shallow embedding that is not present in the deep embedding. The types in the shallow embedding correspond to types in Scala and therefore can support more method than the deep embedding. After transition to the deep embedding these methods would not be detected.
- Scala's weak type conformance does not hold in the deep embedding. If weak type conformance is applied in the shallow embedding, after translation, users will get a cryptic type error.
- Accidentally using functions that are not in the shallow embedding. For example, if a user passes a higher-order function that does not use `Rep` types it will work in the shallow embedding but fail in the deep embedding:

  ```
  list.map((x: Int) => x + 1) // fails in the deep embedding
  ```

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, there are several differences:

- Bytecode interpretation happens after the host language has finished compilation to produce bytecode. Therefore, it is hard to incorporate error reporting in the host-language from this stage.
- The proposed solutions do not support compile-time code generation. They always postpone compilation to runtime.
- These approaches do not support language restriction. This feature could be executed but it the implementation is not straight forward. For example, logical operators are represented as conditionals in bytecode and distinguishing between a conditional and a logical operation is not possible.

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 the two 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. In the following example we show the modified signatures of the `Eq` type class:

```
class Eq j a where
  (==) :: (BoolC j) => j a -> j a -> j Bool
  (/=) :: (BoolC j) => j a -> j a -> j Bool
```

In the Haskell's `Eq` the constraint `BoolC` and variable `j` do not exist.

Kansas Lava [Gill et al., 2011] is a Haskell EDSL for hardware specification. In Kansas Lava Gill et al. propose a design flow where the hardware is first specified in a Haskell standard library, then it is manually translated into the shallow embedding that uses the applicative functor `Signal` to guarantee that the circuit can be synthesized. Finally, the DSL user uses the deep embedding to generate hardware. In this workflow the translation is not automated and DSL user must do it manually for each program. The interface of the shallow embedding is not equivalent to the direct embedding, however, this is desired as the difference in the interface is used to verify if the code can be synthesized.

## 12.2 Improving the Deep Embedding Interface

Scala Virtualized [Rompf et al., 2013a] allows usage of host language constructs on DSL types. This is achieved by translating all language constructs to method calls and letting the DSL author override them with alternative semantics. The same idea was proposed by Carette et al. and Yin-Yang builds upon it. In Yin-Yang we also translate the types with a configurable type translation and integrate the translation into the host language.

Heeren et.al [Heeren et al., 2003, Hage and Heeren, 2007] introduce extensible error reporting to type systems. With their approach the library author can provide custom error messages for specific usage patterns of their libraries and DSLs. This can be used in the deep embedding to improve convoluted and incomprehensible error messages.

Svenningsson and Axelson [Svenningsson and Axelsson, 2013] propose combining a small deeply embedded core with a shallow front-end. The shallow front end uses the small core to provide all language features to DSL users: it is a layer over the core deep embedding that is used to improve its interface. In their approach, however, the shallow interface is still modified to hide the IR construction.

# Part II
# Automating Deep Embedding Development

# 13 Simplifying Transformations in the Deep Embedding

In the deep embedding the DSL author often performs transformations over the intermediate representation. The transformations can be *lowering* transformations that convert domain-specific IR nodes to a lower-level representation, program optimizations, global transformations, etc. In all of these categories the DSL author must *deconstruct* ASTs and constructs new ASTs based on the information from deconstructed ASTs.

In the deep embeddings the common ways to achieve transformations are:

- **AST manipulation.** The "naked" AST manipulation is typically achieved by using the constructors and deconstructors of the AST nodes. The disadvantage of this technique is that, although it gives complete control over the ASTs, it is verbose. To demonstrate the verbosity of the approach we show the lowering transformation of `Vector.fill`:

```
case VectorFill(size, v) =>
  VectorFromSeq(SeqFill(s, v))
```

- **Using the deep embedding interface.** An alternative to AST manipulation is the reuse of the deep embedding for transformations [Rompf et al., 2013b]. This approach improves the interface for the DSL authors. The lowering of `VectorFill` could be performed as:

```
case VectorFill(size, v) =>
  Vector.fromSeq(Seq.fill(s, v))
```

This approach, however, has several drawbacks that we describe in the next section.

## 13.1 Drawbacks of the Deep Embedding as a Transformer

**Non-idiomatic interface.** Using deep embedding for transformations improves over AST manipulations but it is not ideal. The subset of the problems with the deep embedding interface (§4.2) persists. The DSL authors are, however, more accustomed to the convoluted interfaces and complicated type errors.

**Masking the standard library.** In DSLs based on polymorphic embedding there is a particular difficulty. When the DSL back-end uses the interface of the deep embedding it often masks the interface of the standard library. The DSL authors then must use full paths to access objects in the standard library. For example, constructing a simple hash-map must be achieved with

```
collection.immutable.HashMap("Int"->typeTag[Int])
```

because simply calling `HashMap("Int"->typeTag[Int])` would be captured by the deep interface and result in producing a `HashMapApply` IR node.

**Implicitly passed parameters.** Implicit parameters that were used during construction of case classes are not implicit during their deconstruction. This makes it hard to use the same implicit mechanisms in the deep embedding in the presence of deconstruction.

For the constructor of `Vector.fill` the type information can be passed as an implicit argument and stored in the IR node:

```
def vector_fill[T](v: Rep[T], size: Rep[Int])
  (implicit tp: TypeTag[T]) = VectorFill(v,size,tp)
```

In the scope where the node is deconstructed, however, the parameter is not present:

```
case VectorFill(v, size, tp) => // tp is not implicit in this scope
```

In order to pass the type information in their transformations the users are faced with two choices: *i)* pass the information manually, *ii)* or re-introduce the implicit information manually in the scope. Both approaches make it harder to transform the ASTs with the deep embedding.

## 13.2 Quotation for Specifying Transformations

To improve transformations in the deep embedding we again use the direct embedding as the front end. From the direct Instead of using the interface of the deep embedding we

simply use the the direct embedding and use the translation to achieve node reification.

By using only the direct interface there are no issues with the abstraction leaks of the deep embedding. Further, the difference between the host language terms and deep terms is clearly separated with the scope of the translation. The lowering of `VectorFill` with the translation goes as follows:

```
case VectorFill(size, v) => trans {
  Vector.fromSeq(Seq.fill(s, uq(v)))
}
```

where the `trans` is the translation defined similarly to the DSL translation.

The translation we use here has several differences compared to translation for DSL programs:

- Signature of the translation method. The return type of the translation method (`trans` in the example) is modified:

  `def trans[T](body: => T): τ(T)`

  The return type is now the type in the deep embedding ($\tau$ is the type translation for a given DSL).

- Captured variables. The captured variables (`s` and `v` in the example) have incompatible types with the direct embedding. We tackle this problem by allowing *unquoting* of captured variables with the function `uq` that has a signature:

  def uq[T](t: $\tau$(T)): T

  An implicit version of this function is also provided that allows the user to omit the function in many cases.

  In the example, the value `v` must use the `uq` function as the function `fill` is generic and passing a `τ(t)` would pass the type checking and never introduce the implicit version of the function. The captured variable `s`, on the other hand, has the expected type `Int`. The wrong expected type will trigger the implicit version of `uq` that unquotes the term.

- Scope injection. For AST transformation the scope injection phase is defined differently. In case of the polymorphic embeddings, where all DSL definitions are inside Scala components, the scope injection phase is not necessary. The translated code is already in the scope of the deep embedding definitions.

### 13.2.1 Passing Implicit Parameters

To minimize the work required to deal with implicit arguments we modify the transformation method introduce them automatically. First we allow the method to accept a list of nodes whose parameters with type information should be implicit in the scope:

```
def trans[T](nodes: Any*)(body: => T): τ(T)
```

To allow introduction of implicits we modify the scope injection phase to introduce the implicits required by the DSL author. The way how implicits are extracted from the node are left to the DSL author. In the following example we will find all the `TypeTag`s in the signature of the node and provide them in scope implicitly. With this scheme the previous example becomes:

```
case node@VectorFill(size, v, _) => trans(node) {
  Vector.fromSeq(Seq.fill(s, uq(v)))
}
```

and it expands to

```
case node @ VectorFill(size, v, _) =>
  class T // A synthetic class
  implicit val impl$0 = node.tpe.asInstanceOf[TypeTag[T]]
  val v$1 = v.asInstanceOf[τ(T)]

  lift[Vector].fromSeq(lift[Seq].fill(s, v$1))
}
```

## 13.3 AST Deconstruction

A similar mechanism as with node construction could be used for AST deconstruction. In this case the deep embedding domain must be changed to perform deconstruction instead of construction of IR nodes. The deconstruction of nodes can be achieved by using Scala quasi-quotation in pattern matching. Deconstruction of `VectorFill`, from the example, would be achieved with:

```
case ext"Vector.fill($size, $v)" =>
  // here variables ‘size’ and ‘v’ have expected types
```

In order to achieve this we must provide an alternative deep embedding for node deconstruction with the interface prescribed by the core translation. We leave this for future work.

# 14 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 (§14.1). Second, we exploit the fact that method implementations in the direct embedding are also direct DSL programs. Reusing our core translation from §5, we translate them to their deep counterparts (§14.1.1).

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.

While these optimizations are not re-generated; only the components that correspond to the interface and the IR nodes are modified. Therefore, the DSL author is only responsible for maintaining analysis and optimizations in the deep embedding. A change in the direct embedding interface should affect only optimizations related to that change. However, this is back-end dependent so Yin-Yang does not provide any guarantees.

```
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 14.1 – High-level IR nodes for Vector.

## 14.1 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 14.1 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

```
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 14.2 – Direct and deep embedding for Vector with side-effects.

the direct EDSL is using libraries which are already annotated, like the Scala collection library, then the EDSL author does not have to annotate the direct EDSL. Otherwise, there is a need for manual annotation of the direct embedding by the EDSL author. Finally, the Scala FX annotations are mapped to the corresponding effect construct in LMS.

Figure 14.2 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].

### 14.1.1 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 14.3 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.

```
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 14.3 – Lowering to the low-level implementation for Vector.

Generating the low-level implementation is achieved by transforming the implementation of each direct embedding method. This is done in two steps. First, the expression given as the implementation of a method is converted to a Scala AST of the deep embedding by core translation of Yin-Yang. Second, the code represented by the Scala AST must be injected back to the corresponding trait. 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.

## 14.2 Evaluation

To evaluate the automatic deep EDSL generation for OptiML, OptiQL, and Vector, we used Forge [Sujeeth et al., 2013a], 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 14.1, 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 14.1 – 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.

## 14.3 Related Work

# 15 Automatic Management of Dynamic Compilation of DSLs [TODO: Not Finished ]

Domain-specific language (DSL) compilers often require knowledge of values known only at program run-time to perform optimizations. For example, in matrix-chain multiplication, knowing matrix sizes allows choosing the optimal multiplication order [Cormen et al., 2001, Ch. 15.2] and in relational algebra knowing relation sizes is necessary for choosing the fastest order of join operators [Selinger et al., 1979]. Consider the example of matrix-chain multiplication:

```
val (m1, m2, m3) = ... // matrices of unknown size
m1 * m2 * m3
```

In this program, without knowing the matrix sizes, the DSL compiler can not determine the optimal order of multiplications. There are two possible orders `(m1*m2)*m3` with an estimated cost `c1` and `m1*(m2*m3)` with an estimated cost `c2` where:

```
c1 = m1.rows*m1.columns*m2.columns+m1.rows*m2.columns*m3.rows
c2 = m2.rows*m2.columns*m3.columns+m1.rows*m2.rows*m3.columns
```

Ideally we would change the multiplication order at run-time only when the condition `c1 > c2` changes and not the individual inputs. Matrix-chain multiplication requires global transformations based on conditions outside the program scope. For this task *dynamic compilation* [Auslander et al., 1996] seems as a good fit.

Yet, dynamic compilation systems—such as DyC [Grant et al., 2000] and JIT compilers—have shortcomings. They use run-time information primarily for program specialization. In these systems, values are profiled for *stability*, i.e., having the same value over multiple executions. Then, *recompilation guards* and *code caches* are based on *checking equality* of current values and previously stable values.

To perform domain-specific optimizations, however, we must check stability, introduce recompilation guards, and code caches, based on the computation specified in the DSL

optimizer—outside the user program. Ideally, the DSL optimizer should be agnostic of the fact that input values are collected at runtime. In the example stability is only required for the condition `c1 > c2`, while the values `c1` and `c2` themselves are allowed to be unstable. Finally, recompilation guards and code caches would recompile and reclaim code based on the same condition.

An exception to existing dynamic compilation systems is Truffle [Würthinger et al., 2013]. Truffle allows creation of user defined recompilation guards based on DSL author defined computations. However, with Truffle, language designers do not have the full view of the program, and thus, can not perform global optimizations (e.g., matrix-chain multiplication optimization). Further, recompilation guards must be manually introduced and the code in the DSL optimizer must be modified to specially handle decisions based on runtime values.

We propose a dynamic compilation system aimed for domain-specific languages where:

- DSL authors declaratively, at the definition site, state the values that are of interest for dynamic compilation (e.g., array and matrix sizes, vector and matrix sparsity). These values can regularly be used for making compilation decisions throughout the DSL compilation pipeline.

- The instrumented DSL compiler transparently reifies all computations on the run-time values that affect compilation decisions. In our example, the compiler reifies and stores all computations on run-time values in the unmodified dynamic programming algorithm [Cormen et al., 2001] for determining the optimal multiplication order (i.e., `c1 > c2`).

- Recompilation guards are introduced automatically based on the reified decisions made during DSL compilation. In the example the recompilation guard would be `c1 > c2`.

- Code caches are automatically addressed with outcomes of DSL compilation decisions instead of stable values from user programs. In the example the code cache would have two entries addressed with a single boolean value computed with `c1 > c2`.

We evaluate 15.3 on matrix-chain multiplication algorithm described in 15.2. We show that the DSL user needs to add only 4 annotations in the original algorithm to enable automatic dynamic compilation. We further show that automatically introduced compilation guards reduce compilation overheads by orders of magnitude compared to classical dynamic compilation.

## 15.1 Abstractions for Managing Dynamic Compilation

### 15.1.1 Reifying the Compilation Process

## 15.2 Case Study: Matrix-Chain Multiplication

## 15.3 Evaluation

## 15.4

# Part III

# Removing Abstraction Overheads in DSLs

# 16 Introduction: Approaches to Predictably Removing Abstraction Overheads in Direct DSLs

# 17 Polyvariant Staging [TODO: Not Finished ]

## 17.1 Introduction

*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 [Taha and Sheard, 1997, Nielson and Nielson, 2005] or in different languages [Brown et al., 2011, DeVito et al., 2013]. In this work we will focus on staging where all stages are in the same language and that, through static typing, assures that terms in the next stage are well typed.

Notable staging systems in statically typed languages are MetaOCaml [Taha and Sheard, 1997, Calcagno et al., 2003] and LMS [Rompf and Odersky, 2012]. 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, 2012], for domain-specific languages [Czarnecki et al., 2004, Jonnalagedda et al., 2014, Taha, 2004], and for converting language interpreters into compilers [Rompf et al., 2013c, Futamura, 1999]. Staging originates from research on two-level [Nielson and Nielson, 2005, Davies, 1996] 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 we denote 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 in the *residual* program 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.

**Stage monovariance.** The `dot` function in LMS is monovariant: it can only accept vectors that are statically known but their elements are dynamic. In order to support both versions the author of `dot` would have the body of `dot` with slightly different stage annotations:

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

Designers of *binding-time analysis* for offline partial-evaluators had the same difficulties with monovariance. First solutions duplicate code [Rytz and Gengler, 1992] to achieve polyvariance. Duplication is much less troublesome with partial evaluation as users do not write the duplicate code. Henglein and Mossin introduce polymorphic binding-time analysis [Henglein and Mossin, 1994] to avoid code duplication, which was further refined to parametric polymorphism [Heldal and Hughes, 2001], and recursion and subtyping [Dussart et al., 1995]. Although effective, these approaches are not used for staging but for offline partial evaluation. Finally, Ofenbeck et al. [Ofenbeck et al., 2013] propose a solution to this problem based on type classes and higher-kinded types. Their solution requires additional annotations and implicit parameters in the type signature of polyvariant methods.

**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., 2013a] 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.

**Staging at host language compile time.** How can we use type based 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 (§17.5).

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.

**Polymorphic binding-time information.** The main idea of this paper is to use polymorphic binding-time analysis: *i)* to enable stage polivariant code through bounded parametric polymorphism, and *ii)* to allow reusing existing data types in different stages of computation without code duplication.

With bounded parametric polymorphism we can make functions stage polyvariant in their arguments by replacing them with type parameters upperbounded by their original type. For example, polymorphic power function is defined as:

```
@ct def pow[V <: Long](base: Long, exp: V): Long
```

*Annotated types* denote terms whose instances and non-polymorphic fields are known, and whose methods can be executed at in the *previous stage* (i.e., compile time). We call annotated types *compile-time views* of existing data types. Types are promoted to their compile-time views with type qualifiers [Foster et al., 1999] expressed with the `@ct` annotation (e.g., `Int@ct`). In terms, statically known terms can be promoted their compile time duals with the function `ct`. 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`

[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.

signature:

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

Since, vector elements are stage 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

// 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 using bounded parametric polymorphism as a vehicle to succinctly achieve polyvariant staging. This allows writing polyvariant stage programs without code duplication or additional language features. Stage annotations passed through type parameters are used by underlying polymorphic binding-time analysis to determine the correct binding-times.

- By obviating the need for reification and code generation logic in type based staging systems. The same binding-time analysis is to allow types defined in monovariant way to be used in multiple stages.

- By introducing compile-time views (§17.2) as means to achieve type safe type based two-stage programming starting from host language compile time.

- By demonstrating the usefulness of compile-time views in four case studies (§17.4): inlining, partially evaluating recursion, removing overheads of variable argument functions, and removing overheads of type-classes [Wadler and Blott, 1989, Hall et al., 1996, Oliveira et al., 2010].

We have implemented a staging extension for Scala ScalaCT. ScalaCT has a minimal interface (§17.2) based on type annotations. We have evaluated performance gains and the validity of ScalaCT on all case studies (§17.4) and compared them to LMS. In all benchmarks (§17.6) our evaluator performs the same as LMS and gives significant performance gains compared to original programs.

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

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

Figure 17.1 – Interface of ScalaCT.

## 17.2 ScalaCT Interface

In this section we present ScalaCT, a staging extension for Scala based polymorphic binding-time analysis. ScalaCT 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, ScalaCT exposes a minimal interface (Figure 17.1) 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 operations on compile-time views should be executed at compile time, the `ct` type can be viewed as the original type whose instance is known at compile time with additional methods whose non-generic method parameters and result types also become compile-time views (if possible). Generic parameters remain unchanged as their binding time is defined during corresponding type application. Table 17.1 shows how the `@ct` annotation can be placed on types and how it affects method signatures of additional methods. Note that methods are not in fact added to the type but the binding-time polymorphic behavior of original methods can be observed as additional methods.

In Table 17.1, 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 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. Note that `hashCode` of `Vector[Int]@ct` returns a dynamic value–this is due to its implementation that internally operates on generic values that are dynamic. Type `Vector[Int@ct]@ct` has all additional methods promoted to 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`.

Table 17.1 – Compile-time views of types and additional methods that will be available to the user.

| 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`<br>`length: Int@ct`<br>`hashCode: Int` |
| `Vector[Int@ct]@ct` | `map[U](f: (Int@ct => U)@ct): Vector[U]@ct`<br>`length: Int@ct`<br>`hashCode: Int@ct` |
| `Map[Int@ct, Int]@ct` | `get(key: Int@ct): Option[Int]@ct` |

Table 17.2 – 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` |

Annotation `ct` can also be used to achieve inlining of statically known methods and functions. This is achieved by putting the annotation of the method/function[1] definition:

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

Annotated methods will have an annotated method type

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

which can not be written by the users.

**Function `ct`** is used at the term level for promoting literals, modules, and methods/-functions into their compile-time views. Table 17.2 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.

[1] This is not the first time that inlining is achieved through partial evaluation [Monnier and Shao, 2003].

### 17.2.1 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 be known at compile time.

To satisfy cross-stage persistence ScalaCT 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.

### 17.2.2 Minimizing the Number of Annotations

One of the design goals of ScalaCT is to minimize the number of superfluous staging annotations. We achieve this by implicitly adding annotations and term promotions that would with high probability be added by the user.

**Adding `ct` to Functions.** Due to cross-stage persistence if a single parameter of a function is annotated with `ct` the whole function must also be `ct`. Since forgetting this annotation would only result in an error we implicitly add the `ct` annotation when at least one parameter type or the result type is marked as `ct`.

**Implicit Conversions.** If method parameters require the compile-time view 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. Note that we use a custom mechanism for implicit conversions based on type annotations. This is necessary as Scala implicit conversions are oblivious to type annotations (§17.7).

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

```
@ct 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))`).

Implicit conversions are safe in all cases except when the user should be warned about potential code explosion. For example, a user could accidentally call `fact` with a very large number and cause code explosion without even knowing that staging is happening. If `ct` was required it would remind the users about the potential problems. In the design of ScalaCT we decided to prefer less annotations over code-explosion prevention.

## 17.3 Polymorphic Binding-Time Analysis

### 17.3.1 Nominal Types and Subtyping

### 17.3.2 Compile-Time Evaluation

### 17.3.3 Mutable State

## 17.4 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 ScalaCT. We start by inlining a simple function with staging (§17.4.1), then do the canonical staging example of the integer power function (§17.4.2), then we demonstrate how variable argument functions

can be desugared into the core functionality (§17.4.3). Finally, we demonstrate how the abstraction overhead of the `dot` function and all associated type-class related abstraction an be removed (§17.4.5).

### 17.4.1 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 ScalaCT 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
```

### 17.4.2 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 (§17.2.1) the `pow` must be `@ct`. This annotation is automatically added by ScalaCT as described in §17.2.2. In the example, the `ct` annotation on `exp` requires that the function must be called with a compile-time view of `Int`. ScalaCT 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 produces:

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

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

Constant 4 is promoted to `ct` by the implicit conversions (§17.2.2).

### 17.4.3 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 17.2.

### 17.4.4 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 ScalaCT 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 17.3, 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`).

```
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 17.2 – 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`.

```
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 17.3 – Removing abstraction overheads of type classes.

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
```

### 17.4.5 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
```

## 17.5 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 ScalaCT and annotated types are called compile-time views.

To compare approach of ScalaCT with the first approach we analyzed 817 functions from the OptiML [Sujeeth et al., 2011] DSL based on LMS. With the ScalaCT scheme OptiML would require more than 2x less annotations to implement.

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 ScalaCT 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 ScalaCT requires less annotations.

## 17.6 Evaluation

In this section we evaluate the amount of code that is obviated with ScalaCT compared to existing type directed staging systems (§17.6.1). Then we evaluate performance of ScalaCT compared to LMS and hand optimized code (§17.6.2)

### 17.6.1 Reduction in Code Duplication

Evaluating reduction of duplicated code (for reification and code generation) in type based staging systems is difficult as the factor varies from program to program. To avoid benchmark dependent results we instead calculate the lower bound on the duplication factor.

Given that we have a method on a type `T` whose body contains `n` lines of code (without the method definition). To introduce the same method on an annotated type `Rep[T]` we need another method for reification which has at least 1 line of code. Then we need code generation logic, which, if we use the same language should not have less lines than the original method plus at least one line for matching the reified method. For method of $n$ lines we get a lower bound on the code duplication factor of:

$$2n + 3/n + 1$$

For single line methods ($n = 0$) the factor is 3 and for large methods ($n \to \infty$) it converges to 2.

### 17.6.2 Performance of Generated Code

In this section we compare performance of ScalaCT with LMS and original code. All benchmarks are executed on an Intel Core i7 processor (4960HQ) working frequency of 2.6 GHZ with 16GB of DDR3 with a working frequency of 1600 MHz. For all benchmarks we use Scala 2.11.5 and the HotSpot(TM) 64-Bit Server (24.51-b03) virtual machine. In all benchmarks the virtual machine is warmed up, no garbage collection happens, and all reported numbers are a mean of 5 measurements.

In Table 17.3 we show execution time normalized to original code for: *i)* `pow(42.0, 10)`, *ii)* `min(a, b, c, d, e)`, *iii)* inner product of two statically known vectors of size 50, and *iv)* the butterfly network of size 4 for fast Fourier transform `fft` (equivalent to code presented by Rompf and Odersky [Rompf and Odersky, 2012]). For all benchmarks the performance results are equivalent to LMS.

Table 17.3 – Speedup of LMS and ScalaCT compared to the naive implementation of the algorithms.

| Benchmark | LMS | ScalaCT |
|---|---|---|
| `pow` | 221.75 | 221.70 |
| `min` | 1.82 | 1.79 |
| `dot` | 246.08 | 246.08 |
| `fft` | 12.14 | 12.88 |

## 17.7 Limitations

**Type Annotations.** Using type annotations for annotating the compilation stage is not ideal. Type annotations are not fully integrated into the Scala language. Major drawbacks are that overloading resolution and implicit search are oblivious about annotations. For example, if two methods have the same signatures but different staging annotations the compiler will report an error. Implicit search will fail in two ways: *i)* if two implicit functions with the same type are in scope but annotations differ the compiler will report an error about ambiguous implicits and *ii)* if a method requires an implicit parameter with the `ct` annotation the compiler might provide an implicit argument without the annotation. These are not fundamental limitations and, in practice, stage polyvariance can be used to avoid difficulties with annotations.

**Type Annotation Position.** Annotations in Scala can be used in many different positions and ScalaCT supports only some of them. Annotation `ct` can not be used in following positions: *i)* on classes, traits, and modules, *ii) in the list of inherited classes and traits*, *iii)* on the right hand side of the type variable definitions, and *iv)* on all terms outside the method definitions (constructors, constructor arguments, etc.).

**Access Modifiers.** Scala supports access modifiers of members. If methods that use ScalaCT internally access `private` members of the class ScalaCT will fail as all staged methods are inlined at the call site. Similar limitations exist with inlining functions in Scala. This problem could be circumvented inside Scala, however, the JVM will not allow this in the bytecode verification phase.

**Code Explosion.** Annotation `ct` inlines all functions that are annotated and unrolls all loops on compile-time executed data structures. This can, for a staged function, lead to unnaceptable code explosion for some inputs while the code can behave regularly for other inputs. For example, calling `pow(0.5, 10000000)` on the function from §17.4 will make unacceptably large code while `pow(0.5, 10)` will work as expected.

### 17.7.1 Nominal Typing, Lower Bounds, and Higher-Kinded Types

## 17.8 Related Work

MetaOCaml [Taha and Sheard, 1997, Calcagno et al., 2003] 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 ScalaCT, 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, 2012] 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`[2]. 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. ScalaCT can be seen as statically typed version of hybrid partial evaluation with support for parametric polymorphism. Due to the support for parametric polymorphism ScalaCT can express compile-time data structures with dynamic data.

Forge [Sujeeth et al., 2013a], 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., 2013b] framework.

**[TODO: what to do with this ]** The Yin-Yang framework, by Jovanovic et al. [Jo-

[2]Name `ct` in ScalaCT is inspired by hybrid partial evaluation.

vanovic 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.

# 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.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the Jalapeno JVM. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

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

Joel Auslander, Matthai Philipose, Craig Chambers, Susan J Eggers, and Brian N Bershad. Fast, effective dynamic compilation. In *International Conference on Programming Language Design and Implementation (PLDI)*, 1996.

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.

H. Chafi, Z. DeVito, A. Moors, T. Rompf, A.K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, page 835–847, 2010.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

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.

Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Static Analysis*, pages 118–135. Springer, 1995.

Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007–Object-Oriented Programming*, pages 273–298. Springer, 2007.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *Software language engineering*, pages 197–217. 2013.

Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *International Conference on Programming Language Design and Implementation (PLDI)*, 1999.

Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

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.

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Symposium on Implementation and Application of Functional Languages (IFL)*. 2011.

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1):147–199, 2000.

Danny M Groenewegen, Zef Hemel, Lennart CL Kats, and Eelco Visser. WebDSL: a domain-specific language for dynamic web applications. In *Companion to the International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA Companion)*, 2008.

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

Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In *Symposium on Implementation and Application of Functional Languages (IFL)*. 2007.

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.

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Scripting the type inference process. In *International Conference on Functional Programming (ICFP)*, 2003.

Rogardt Heldal and John Hughes. Binding-time analysis for polymorphic types. In *Perspectives of System Informatics (PSI)*, 2001.

Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In *Programming Languages and Systems (ESOP)*, 1994.

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.

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.

Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2009.

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *International Conference on Software Language Engineering (SLE)*, pages 311–331. 2013.

Prasad A. Kulkarni. JIT compilation policy for modern machines. In *International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

HyoukJoong Lee, Kevin J Brown, Arvind K. Sujeeth, Hassan Chafi, Kunle Olukotun, Tirark Rompf, and Martin Odersky. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, (5):42–53, 2011.

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

Florian Lorenzen and Sebastian Erdweg. Modular and automated type-soundness verification for language extensions. In *International Conference on Functional Programming (ICFP)*, 2013.

Russell McClure, Ingolf H Krüger, et al. SQL DOM: compile time checking of dynamic sql statements. In *International Conference on Software Engineering (ICSE)*, 2005.

Heather Miller, Philipp Haller, and Martin Odersky. Spores: a type-based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming (ECOOP)*. 2014.

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

A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, page 117–120, 2012.

Tridib Mukherjee, Ayan Banerjee, Georgios Varsamopoulos, Sandeep KS Gupta, and Sanjay Rungta. Spatio-temporal thermal-aware job scheduling to minimize energy consumption in virtualized heterogeneous data centers. *Computer Networks*, 53(17): 2888–2904, 2009.

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 (Version 2.9)*. 2014.

Martin Odersky. Contracts for scala. In *Runtime Verification*, pages 51–57, 2010.

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 *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2005.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2013.

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

Markus Püschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gačic, Yevgen Voronenko, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2): 232–275, 2005.

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, 2012.

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.

Bernhard Rytz and Marc Gengler. A polyvariant binding time analysis. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1992.

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.

P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *International conference on Management of data (SIGMOD)*, pages 23–34, 1979.

A. Shaikhha and M. Odersky. An embedded query language in Scala. Technical report, Technical Report EPFL-STUDENT-213124, EPFL, Lausanne, Switzerland, 2013.

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 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)*, 2013a.

Arvind K. 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)*, 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*. 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/.

Vlad Ureche, Tiark Rompf, Arvind K. Sujeeth, Hassan Chafi, and Martin Odersky. StagedSAC: a case study in performance-oriented DSL development. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2012.

Tijs van der Storm. *The Rascal language workbench*. CWI. Software Engineering [SEN], 2011.

Eelco Visser. Program transformation with Stratego/XT. In *Domain-Specific Program Generation*. 2004.

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.

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.