Yin-Yang: Transparent Deep Embedding of DSLs

Vojin Jovanovic Ngoc Duy Pham
Vlad Ureche Sandro Stucki
Christoph Koch Martin Odersky
École Polytechnique Fédérale de Lausanne - EPFL
{firstname.lastname}@epfl.ch

Vladimir Nikolaev
St. Petersburg National Research University of Information Technologies, Mechanics and Optics (University ITMO)
vladimir.nikolaev@cs.ifmo.ru

Abstract

Deep EDSLs intrinsically compromise programmer experience for improved program performance. Shallow EDSLs, complement them, by trading program performance for good programmer experience. We present Yin-Yang, a library for DSL embedding that uses compile-time meta-programming to transparently, and reliably, transform shallow EDSL programs to the equivalent deep EDSL programs. The transformation allows program prototyping and development with the shallow embedding, while the equivalent deep embedding is used where performance is of essence. Through language virtualization and a minimal interface Yin-Yang allows design of compiler agnostic EDSLs that perform domain-specific analysis at compile time while EDSLs can be compiled at both run and compile time. For run time compiled EDSLs Yin-Yang introduces guarded recompilation that significantly reduces EDSL invocation overheads. This allows EDSLs to be used in latency sensitive applications, and on small data sets. We show that Yin-Yang's transformation is reliable and obviates all embedding related annotations in a large set of applications from two non-trivial EDSLs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Compile-Time Meta-Programming, Embedded Domain-Specific Languages, Multi-Stage Programming

1. Introduction

Shallowly embedded DSLs (EDSLs) execute programs directly on host language values. This makes them consistent with the host language semantics, the type errors are comprehensible, the compilation is fast, and debugging is easy. However, the direct execution introduces abstraction overheads, and prevents detailed program analysis and optimizations. As a result, shallow EDSLs cannot provide domain-specific error reporting and exhibit low performance.

Deep EDSLs create an intermediate representation (IR) of the program, which is, after optimization and analysis, interpreted or generates new optimized code. The embedding can be achieved at run time [9, 12, 20] via the type system or language extensions, or at compile time [4, 23, 29] with macros or template meta-programming. The deep embedding allows different interpretations of the language, greatly improves performance, and allows domain-specific analysis of programs.

Creation of the IR in the deep embedding intrinsically causes abstraction leaks that compromise programmer experience. Run time EDSLs require complex type systems (e.g. Lightweight Modular Staging [20], polymorphic embedding [12]) to allow IR construction. The complex types make the DSL interface difficult to understand, error messages become incomprehensible [7], their compilation time is large, debugging is hard, and domain-specific analysis can be achieved only at run time. Compile time EDSLs do not require complex interfaces, but they are inconvenient for multi-stage programming, their debugging is hard, and DSL development requires knowledge of the compiler internals.

Ideally, we could combine the virtues and cancel out the drawbacks of shallow and deep EDSLs. Svenningsson and Axelsson [26], propose to complement minimal core deep EDSLs with the user friendly shallow interface. However, in their approach EDSLs still require additional typing constructs in the language interface, making them harder to understand. Kansas Lava [8] EDSL keeps parallel shallow and deep implementations, allowing program development with the shallow EDSL and final execution with the deep EDSL. The translation from the shallow to the deep version must be achieved manually, imposing additional effort on the users.

We present Yin-Yang, a library for DSL embedding, that uses compile-time meta-programming to reliably and transparently translate shallow into the corresponding deep embedding. The virtues of the shallow embedding are used during program development when performance is not of importance. The shallow is translated to the deep embedding when performance or different interpretation are of essence. The absence of complex type system constructs and reliability of the translation completely hide the deep embedding from the users, while high program performance is still achieved. With our approach virtues of both shallow and deep embedding are available to the user while drawbacks are completely canceled out.

Yin-Yang provides a minimal interface for the EDSL authors which allows EDSLs to be compiled at run time, or at compile time. The interface of Yin-Yang allows compile time error reporting for domain-specific analysis, completely hides the compiler internals from the author, and supports both interpreted and code generating EDSLs. Furthermore, Yin-Yang introduces efficient recompilation guards for runtime EDSLs which avoid IR construction on every execution.

We show reliability and transparency of the translation in 21 programs from EDSLs OptiGraph [24] and Scala Collections. In all programs the shallow implementation exactly resembles the existing deep embedding, except for numerous deep embedding related annotations that were unnecessary. Furthermore, we evaluate improvements in compilation times between Yin-Yang and the deep embedding and show that tool improves compilation times up to 4x for correct programs and 14x for incorrect programs. Finally, we evaluate performance of guarded EDSL recompilation and show that it outperforms existing techniques for several orders of the magnitude, allowing Yin-Yang EDSLs to be used in latency sensitive contexts and on frequently used small data sets.

Yin-Yang is implemented in Scala, uses macros [4] for meta-programming, and supports DSLs based on polymorphic embedding and Lightweight Modular Staging (LMS). Yin-Yang contributes to the state of the art by:

- Providing completely transparent deep embedding of DSLs based on polymorphic embedding and LMS with reliable translation of shallow EDSL programs to their deep counterparts.
- Introducing efficient guarded recompilation of run time EDSLs which avoids IR construction on every program invocation. The guarded recompilation allows usage of deep EDSLs in latency critical contexts and with frequent executions on small data sizes.
- Using language virtualization [15] and a simple interface to allow compiler agnostic implementation of DSLs. This technique allows domain-specific analysis and DSL compilation at either compile or run time based on the DSL program requirements.

In the following text we will briefly explain DSL embedding with Virtualized Scala in (§2). We will then show abstraction leaks introduced by polymorphic embedding and LMS (§3), we will analyze how complementing shallow with the deep embedding can prevent abstraction leaks (§4). Then we present Yin-Yang interface (§5), and operation (§6). We evaluate the reliability of the translation, compilation performance, and guarded recompilation (§7). Finally, we discuss our approach (§8), compare it to related work (§9) and conclude (§10).

2. Background

In this section we provide information necessary for understanding Yin-Yang. We will briefly explain Virtualized Scala [15], Scalable Components Abstraction [17], polymorphic embedding of DSLs [12], and Lightweight Modular Staging [20, 22]. We assume familiarity with basics of the Scala Programming Language [18].

2.1 Virtualized Scala

Virtualized Scala [15] is an experimental branch of Scala that allows DSL authors to override the behavior of native language constructs (conditionals, loops, variables etc.). Virtualization is achieved by translating language constructs into regular method calls. To override a language feature the author imports the appropriate method with the alternative definition into the scope. Furthermore, for methods that exist on every type, like `==` and `hashCode`, virtualization is achieved through functions prefixed with `infix_` which override methods from existing types, even when the program is correctly typed. Overridden infix functions can be used to construct intermediate representation (IR) nodes within DSLs.

2.2 Deep Embedding of DSL in Scala

Odersky and Zenger [17] describe reusable component systems for Scala, with interfaces for required and provided services, without hard references. Scala components are based on abstract type members, selftypes, and mix-in composition. Abstract type members allow to abstract over types of components and provide easy parametrization through inheritance. Selftype annotations declare the type of **`this`** which states the required interfaces and allows for the transparent reuse of dependent components. Mixin composition is used to compose different components without hard references, while providing guarantees about correct component dependencies. Without loosing generality we focus on DSLs that are based on described Scala components.

Polymorphic Embedding, introduced by Hofer et al. [12], is a DSL embedding technique based on Scala components. They introduce modular DSLs with multiple interpretations where each interpretation, as well as the interface, is a reusable component. The embedding is achieved by writing DSL programs in the components scope where all types are abstracted over by using abstract type members. Methods applied on abstract types are also virtual so they can be given different semantics. This is achieved by using a different composition of components. Deep embedding of DSLs as well as static analysis are introduced as just another

```
trait MatrixOps {
  self: VectorOps with NumberOps  =>
  type Matrix[T] = Vector[Vector[T]]

  def add(m: Matrix[Num],n: Matrix[Num])=
    map(zip(m, n), {
      case (x, y) => addV(x, y)
    })
}
```

Figure 1: Simplified matrix operations module.

```
trait PowerOp extends Base {
  def pow(b:Rep[Float],ex:Int):Rep[Float]=
    if (ex == 0) 1 else b * pow(b, ex - 1)
}
```

Figure 2: Staged power function in LMS.

reusable component. To illustrate the approach we present a DSL module in Scala for matrix addition in Figure 1.

Through a selftype, `MatrixOps` has a dependency on components `VectorOps` and `NumberOps`. This makes types `Vector` and `Num`, and operations `zip`, `map` and `addV`, available in scope. Type member `Matrix[T]` is defined as an alias for type `Vector[Vector[T]]`. Finally, `add` defines matrix addition in terms of methods defined by the depending component, on `self`, whose semantics will be finalized upon instantiation of `MatrixOps`. Since, `Matrix` and `Vector` are abstract, interpretation of the `MatrixOps` can be defined by components mixed in upon instantiation.

Lightweight Modular Staging (LMS) is a staging [27] framework and an embedded compiler for development of deep EDSLs. It is modular and consists of Scala components for the optimizations, embedded compiler, code generation, overriding language constructs, and frequently used parts of the Scala library.

In LMS, staging is achieved through the type abstraction `Rep[T]`. A term t of type `Rep[T]` will evaluate to t' `T` in the next stage of compilation. Since type `Rep[T]` does not have the same methods as `T`, methods of `T` are added by implicit conversions [19] or *infix* methods from Virtualized Scala. To illustrate the approach we present a staged exponentiation function in Figure 2. The exponent is not a `Rep` type which in the current stage unrolls the recursion. Execution of the power function will produce code, in the next stage, that contains exp number of multiplications.

In conjunction with staging, LMS includes the modular embedded compiler that explicitly tracks effects and applies loop fusion, decomposition of structures, and code motion. Common compiler optimizations, like CSE and DCE, are performed on both DSL abstractions and lower level code. Depending on the components used, the output of the compilation can be Scala, C/C++, CUDA, or JavaScript code.

LMS has been successfully used by Kossakowski et al. for a JavaScript DSL [13], by Ackermann et al. [1] for distributed processing, and by Ureche et al. [28] for multi-dimensional array processing. Brown et al. present Delite [3, 21, 24], a heterogeneous parallel computing framework based on LMS. DSLs developed with Delite cover domains of machine learning, graph processing, data mining, mathematics etc.

3. Leaky Abstractions of Deep Embedding

An advantage of external DSLs is that they can be used by the domain-experts without knowledge of type theory and complex programming language constructs. With EDSLs, however, constraints of the host language often lead to leaky abstractions, like incomprehensible type errors, complex interfaces, and inconsistency with the host language.

In this section we will identify the leaky abstractions for techniques described in (§2). We will first describe issues related to both techniques (§3.1) and then we present abstraction leaks caused by type-driven staging (§3.2).

3.1 Modular DSLs in Scala

Run time compilation of EDSLs comes at a cost. Domain-specific program analysis errors can be reported to the user only at run time. Composing an IR for the DSL comes with a performance overhead which is sometimes not negligible. Most analysis can be encoded within the type system, but this leads to incomprehensible error messages, long compilation times, and time-consuming DSL design.

Disallowing host language constructs is sometimes required by the DSL's requirements. DSLs in Scala can interleave usage of pure Scala and DSL code which can be confusing for the domain experts. Furthermore, usage of language constructs that are not supported is either allowed, for always valid constructs like **try**, or results in an incomprehensible error message.

Long compilation times are an unavoidable consequence of the complex type constructs [7]. In Scala, reusability and modularity of DSLs comes with a large compilation performance overhead. The DSL component can be composed out of more than 30 other components. Additionally, conversion of Scala types to the abstract types requires many implicit conversions. In result, compilation times can become annoyingly large. For example, K-means application in OptiML [25] has only 15 lines of code and compiles for 10 seconds.

Program modularity is only possible by mixing-in traits thus exposing much of the Scala features. Path-dependent types in the Scala components require all user defined modules to be components as well. Then they are mixed-in at the call site with the main component, or new aggregate components need to be made. This prevents the use of the standard **import** statement and requires additional boilerplate for instantiation of DSL programs.

```
val one: Rep[Int] = 1;
val void: Rep[Unit] = ()
one + void // ill typed term
```

Figure 3: Fails with: "No implicit view available from RepDSL.this.Rep[Unit] => Int".

Conversion of literals is not always achieved implicitly, so programmers must explicitly convert literals, usually with a method call. This restriction is caused by a restriction in Scala that prevents multiple implicit conversions to be applied on the same term. This problem appears in algebraic expressions that use a mixture of `Integer` and `Double` literals, in lifting of tuples with literals etc. For example, in LMS, when `v` is typed `Rep[T]`, `(1, v)` will result in the compile error and needs to be converted to `(unit(1), v)`. Some of these errors can be overcome with phantom types, or large number of implicit conversions, but these lead back to complex type errors and long compilation times. Finally, implicit conversions are very hard to debug and cause unintuitive error messages.

Inconsistency with the host language is introduced by the abstract type members. The Scala language provides certain type based transformation early in the compilation pipeline to enhance user experience. These transformations are not applied to DSL types due to their overriding. For example, closures with `Unit` type are augmented with a unit value as the return type and weak type conversions promote `Int` to `Double`.

Virtualized Scala requires a separate branch of the Scala compiler. Users are often reluctant in adopting an experimental compiler for their projects. In addition, maintenance of a compiler branch is not trivial.

3.2 Lightweight Modular Staging

We have analyzed over a hundred DSL programs based on LMS and Delite and identified main abstraction leaks with these approaches. The abstraction leaks that we identified are presented in this section.

Incomprehensible type errors are caused by the type `Rep[T]`, especially in combination with infix methods and language virtualization. The example in Figure 3 shows a simple addition on wrong types where the type checker returns an incomprehensible error. Errors can be more than 50 lines long when language virtualization is used.

Complex method signatures are hard to understand by the users. Wrapping return and argument types with `Rep[_]` increases the signature size. In addition, methods must have implicit parameters that pass information about the position in the source file (`SourceContext`). With type erasure, methods with a same name and signatures different only in the generic parameters can not be overloaded. LMS overcomes this problem by introducing the `Overloaded` implicit parameter which allows DSL authors to change the method signature. Finally, `infix` prefix to methods can further obscure the signature. In Figure 4 we present a signature of a method for subtraction of type `Float` with `Int` that exhibits all described issues.

```
def infix_-(lh: Rep[Float], rh: Rep[Int])
  (implicit o: Overloaded,
  ctx: SourceContext): Rep[Float]
```

Figure 4: Long signature of a simple method for subtraction of an Integer from a Float.

Recompilation of DSL programs is required when run-time values, captured by the DSL, change the value. Since not all captured identifiers are used in optimizations, compilation can be avoided in some cases. To achieve this, LMS must build the IR of the program on each execution, and compare the captured constants to the state from the previous execution. This is a costly operation and for a set of short running programs it can cost more than one gains by staging.

Type inference of deep DSL programs is not equivalent to their shallow versions. Type inference of higher-kinded types, like `Rep[T]` is very limited compared to regular types [14]. This requires a greater number of explicit type annotations on higher-order functions and complex expressions.

Recursive functions with return type `Rep[T]` are unfolded at staging time and can lead to infinite recursion, or code explosion. This can be a powerful feature for optimization, like in Figure 2, but can also be a caveat. For example, for `pow(0.5, 10^6)` the compiler will produce 10^6 IR nodes. Additionally, to prevent infinite unfolding users must wrap all recursive definitions with a higher order method `lam` which will prevent infinite recursion.

4. Shallow can Complement the Deep Embedding

Shortcomings of the deep embedding presented in (§3) are well paid off in the program performance. DSLs based on Delite provide more than 100x speedups in a variety of domains [22]. Pushing this further, data querying DSLs, like SQL, often offer several orders of magnitude improvement in performance over their library counterparts.

On the other hand, with large run-time overheads, shallow EDSLs exhibit much simpler interfaces, and their usage is consistent with the the host language. Usage of well known standard library types in the method signatures lowers the barrier of learning the language, and improves user experience. Type errors reported are much simpler since there are far less implicit conversions, type classes present, and there is no language virtualization. Many shallow DSLs, like Scala Collections [16] and Actors [10], have reached wide user adoption due to their simplicity and consistency.

Debugging of shallow DSLs is the equivalent to debugging standard programs, allowing users to navigate through the DSL consistently with the rest of the language. Moreover, one can inspect and change run-time values of intermediate data structures internal to the DSL. In deep embedding, IR construction, optimization, and interpretation make this process overly cumbersome pushing debugging back to `println` statements, and sometimes preventing it completely.

Interfaces of deep EDSLs should match the interfaces of shallow EDSLs. Ideally the programs in the deep embedding would be the same as their shallow counterparts, but with more complex signatures. Unfortunately, due to the restrictions of the host language, this is not always possible, as we have presented in (§3).

Could one use the shallow embedding for user friendly type checking, prototyping, and documentation, but in production, when performance matters, use the same version of the program with the deep embedding? That would provide the best of both approaches greatly increasing the usability of EDSLs. This, however raises many questions. How can we achieve this translation, when embeddings have different method signatures? Does the effort to maintain two versions of the EDSL increase development cost, pushing the required effort back towards external DSLs? What if shallow embedding is too slow for prototyping? Could we apply domain-specific analysis of the code in the shallow embedding?

Translation is possible with macros for one DSL, and regularity of the DSL embeddings described in (§2) extends this translation to a large family of DSLs. The types in LMS and with polymorphic embedding are straightforward translations of the Scala library types. Type `T` becomes `Rep[T]` in LMS, and **`this`**`.T` with polymorphic embedding. In addition, the overhead of the language virtualization is achieved as a part of the meta-program removing the dependency on the experimental compiler. Moreover, due to the simple type translation, type ascriptions translated from the already inferred shallow DSL types, are placed around every term. This removes the dependency on limited type inference of higher-kinded types and results in the typing equivalent to the shallow embedding. Finally, type driven translations from the standard library are already performed in the shallow embedding, so the deep embedding just needs to have their interpretation.

Additional effort is insignificant as the deep embedding requires far more work than the shallow one. The interface of deep DSLs is much harder to design for good user experience. Also, most DSL methods build an IR, consisting of the domain-specific nodes as well as the low-level ones. Optimizations usually perform multiple phases of rewriting rules on the IR. Finally, the IR is either interpreted, or the new source is generated, requiring another layer of interpretation. If the shallow implementation, practical for debugging, is not required, only the interface of the shallow embedding can be used. This requires far less effort than synchronizing semantics of two different implementations.

With the shallow front-end, the interface of the deep EDSL does not require high standards and therefore requires less effort to develop. Significant part of the effort in the design of deep EDSLs is spent on making user friendly interfaces, like providing implicit conversions for non-DSL types, chaining implicit conversions, and using phantom types. Part of this effort can be used for maintaining the shallow embedding.

Small data sets in development and prototyping are common, for practical reasons. Ideally, programmers would require minimal time from designing to deployment. This is achieved by running programs on small data sets. Additionally, program design requires constant debugging which is best achieved with little data. High-performance is only needed once the code is deployed and run on the large data sets. Many widely used frameworks, like Flume Java [5] and Dryad/LINQ [30], keep a parallel in-memory implementation for prototyping and debugging purposes. In comparison to these frameworks the parallel interface of Yin-Yang EDSLs would additionally provide consistency with the host language, comprehensible error messages, and short compilation times.

Domain-specific analysis at compile time is achieved once the mapping from a shallow to the corresponding deep embedding is possible. The transformed program can be reflectively instantiated and then analyzed at the host language compile time, thus reporting all errors during compilation.

5. Yin-Yang Interface

Yin-Yang is a compile-time meta-programming library that, safely and transparently, translates the shallow DSL programs, based on Scala components, into their deep counterparts. Since Yin-Yang runs at compile time, the static analysis of DSLs is achieved, and errors reported, during host language compilation. Yin-Yang also supports interpreted DSLs and code generating DSLs. If a code generating DSL does not require run-time values for optimization, Yin-Yang allows code generation at compile time.

The library consists of the single configurable compiler transformer that applies the translation. For each individual DSL the transformer is used with different configuration parameters. The factory method for instantiating transformers that is used by DSL authors is shown in Figure 5. Its usage for the OptiGraph DSL [24] is presented in Figure 6. These methods are used only once per DSL.

The transformer factory accepts *i)* `dslComponent` holding the name of the DSL Scala component that mixes in all other sub-components and *ii)* `typeTransformer` holding a function from Scala `Types` to compiler trees `Tree` that defines type mapping from the shallow to the deep embedding. The example in Figure 6 represents the usage of Yin-Yang by the DSL author. The `dslBlock` holds the AST nodes of the shallow DSL body, `c: Context` is the compiler context passed to the macro. The `lmsTransformer` is the trans-

```
object YYTransformer {
  def apply[C <: Context, T](c: C,
    dslComponent: String,
    typeTranformer: Type => Tree)
}
```

Figure 5: Yin-Yang transformer factory for defining individual DSL translations.

```
// lmsTransformer is defined once for LMS
def lmsTransformer(tpe: Type): Tree = ...
def optiGraph[T](dslBlock: => T): T =
  macro _optiGraph
def _optiGraph[T](c: Context)
  (dslBlock: c.Expr[T]): c.Expr[T] =
  YYTransformer[c.type, T](c,
    "lifted.OptiGraph", lmsTransformer
  )(dslBlock)
```

Figure 6: Defining a method that translates shallow embedding of OptiGraph to the deep embedding.

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

Figure 7: Component for enabling domain-specific analysis of DSLs.

former that we defined for LMS based DSLs. This method is defined once per family of DSLs, Yin-Yang already defines type transformers for LMS and polymorphic embedding.

For the support of domain-specific analysis, code generation, and interpretation, Yin-Yang provides a thin interface towards the DSL author. The interface consists of Scala components that the DSL author should mix-in to enable the required functionality. The interface towards the DSL author is provided in Figures 7, 8, and 9.

Figure 7 presents a `StaticallyChecked` component with a single method `staticallyCheck` which accepts an error reporter interface to the compiler. If the DSL requires domain-specific analysis of the program at compile time this method needs to be in the main component and `staticallyCheck` needs to be implemented. All generated errors are reported at compile time of the host language. Yin-Yang invokes this method at compilation time with the DSL body injected into the component so the DSL IR is present and available for analysis.

Components `Interpreted` and `CodeGenerator` are presented in Figure 8. One of these components must be included to the main DSL component and it defines if the DSL is interpreted, or it generates code. In case of interpretation, the `interpret` method of the run time compiled DSL will

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

Figure 8: Interface components for interpretation and run time code generation.

be invoked with appropriate arguments, while `reset` is used to invalidate optimizations. In case of compile time code generation, the `generateCode` method will be invoked, its return value parsed and included in the program instead of the shallow DSL. In case of run time code generation the compile method will be invoked with type `Ret` instantiated to `(Any, ..., Any) => T` returning the Scala function that will be invoked at run-time with the arguments of the DSL program.

Figure 9 presents the `BaseYinYang` component which is mandatory for all Yin-Yang DSL. The component is used by the transformer to convert literals and captured identifiers to DSL IR nodes. Yin-Yang replaces all captured identifiers with the call to `hole`, with arguments containing type info in `tpe` and identifier of the compiler symbol in the `symbolId`. All literals are replaced by `lift` method calls where the argument `v` contains the value of the literal.

Conversion of literals and holes is delegated to the DSL author who provides different values for the implicit parameter of type `LiftEvidence`. Each component can define a concrete implicit value that is added to the method calls automatically, based on the type of the argument and the return type. This implicit value defines how to map the literal, or a hole, to a suitable IR node by extending methods `hole` and `lift` of `LiftEvidence`. In case of LMS there needs to be just one implicit `LiftEvidence`, but in general specific types can require different IR nodes. This interface allows one to define IR nodes in a modular and compiler agnostic way.

The method `requiredHoles` returns a list of captured values which would be useful for staging purposes. The analysis is performed by the DSL author on the IR that contains hole IR nodes. All of the captured symbols that are returned by this method will be transformed from calls to `hole` to calls of `lift`. If the list is not empty the compilation of the DSL must be postponed to run time.

To summarize, conversion of a deep DSL to a Yin-Yang DSL requires: *i)* the shallow embedding, *ii)* definitions of the implicit `LiftEvidence`, *iii)* configuration of `YYTransformer` macro and its interface to the user, *iv)* definition of the `interpret`, or `generateCode` and `compile` methods, and *v)* optional definition of domain-

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

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

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

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

Figure 9: Interface for lifting non-DSL types.

specific analysis and analysis of required holes in methods `staticallyCheck` and `requiredHoles` respectively.

6. Yin-Yang Operation

The operation of Yin-Yang makes series of decisions guided by the programmer, DSL author, and the DSL program. The flow chart of the operation is illustrated in Figure 10.

Feature Analysis is the first phase in Yin-Yang which analyses methods and language features, used in the DSL block. If any of the features is not supported by the deep embedding a comprehensive error message is reported to the user. This allows for building DSLs that are completely free of host language code, which was not possible with existing approaches. More details about the analysis can be found in (§6.1).

Prototyping checks if the DSL is used in the prototyping mode—with the shallow embedding. If yes, Yin-Yang will simply return the unmodified block of the shallow embedding. If run on large data-sets and in testing environments, it will start the conversion to the deep embedding. This decision is configured by the compilation flag, or by changing the name of the method that encloses the DSL body.

Transformation to Deep Embedding translates the shallow to the deep embedding, if prototyping is disabled. The conversion starts by transforming the body of the shallow EDSL to the Scala component that contains the main method with the body of the deep EDSL. The transformation assures that the deep embedding is well typed, relying on the consistent type mapping between the shallow and the deep embedding. Finally, once the deep EDSL is defined, Yin-Yang reflectively instantiates it in order to invoke the methods defined in (§5). These operations will guide the rest of the conversion, based on the IR of the DSL program. This transformation is described in (§6.2).

Static Analysis performs optional domain-specific analysis of the DSL, which is applied if the DSL inherits the

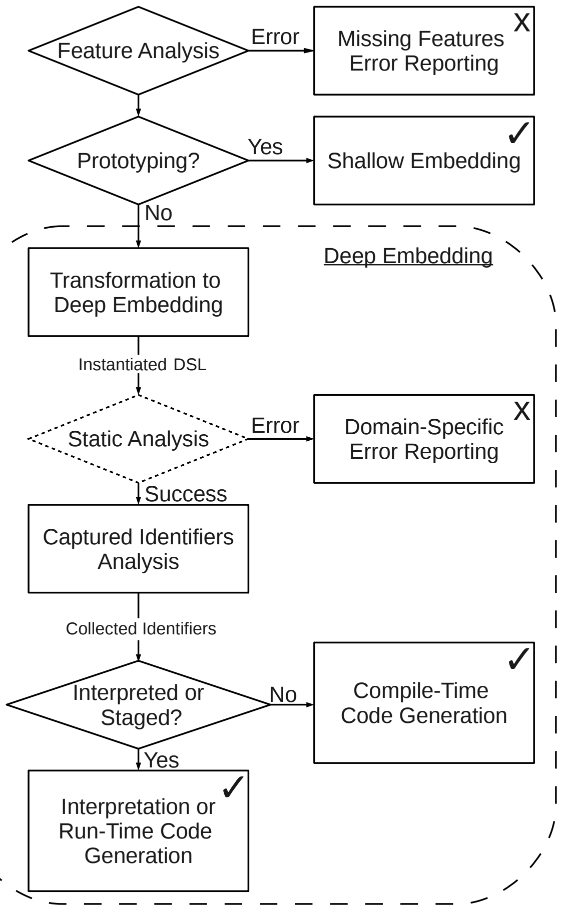
Feature Analysis
Error
Missing Features Error Reporting
X
Prototyping?
Yes
Shallow Embedding
✓
No
Deep Embedding
Transformation to Deep Embedding
Instantiated DSL
Static Analysis
Error
Domain-Specific Error Reporting
X
Success
Captured Identifiers Analysis
Collected Identifiers
Interpreted or Staged?
No
Compile-Time Code Generation
✓
Yes
Interpretation or Run-Time Code Generation
✓

Figure 10: **Flowchart of Yin-Yang operation. Dashed boxes are optional phases in the operation. Nodes containing X represent error states, and boxes containing a check-mark represent successful states.**

`StaticallyChecked` trait. In case of errors, the domain-specific error messages are reported at the standard error output of the host language. For the correct positions of the error messages, the source information is passed as an implicit parameter in the deep embedding. The information about source locations is automatically extracted from the shallow embedding ASTs. If domain-specific analysis is required during prototyping, the successful *Static Analysis* can also be configured to return the original shallow embedding.

Captured Identifiers Analysis checks which captured identifiers are required for optimizations at run time. Not all captured values are required for optimizations and capturing all run-time values causes often DSL recompilation. In this step the DSL analyses which captured values are required for optimizations and returns them to Yin-Yang. If some variables are required the DSL is executed at run time and the required values are converted to lifted values. If no variables are required, and the DSL can generate code, the DSL code generation is performed at compile time. This analysis is performed by the DSL author in the method `requiredHoles`.

Interpreted or Staged is a decision step based on the DSL's type and *captured identifiers*. If a DSL inherits the `Interpreted` trait it can be interpreted at run time, and if it inherits the `CodeGenerator` trait it provides methods for code generation. Code generators are executed at compile time if no identifiers are required for optimizations. Interpreted DSLs are always compiled at run time. The compile time code generation is explained in (§6.4), and run time code generation and interpretation in (§6.5).

6.1 DSL Feature Analysis

A shallow DSL program can contain functions, and language constructs, that are not supported by the deep embedding, thus yielding programs that can not be transformed. To prevent this, we run an analysis phase when the shallow embedding is used. Firstly the virtualization of language constructs is performed to normalize constructs to method calls. Then the analysis checks if every method in the DSL body can be translated to the deep embedding. If it can not, we report an error message containing the name of the method with its position in the program.

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

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

Feature analysis is costly and slows down error reporting. Since it must be used with the shallow embedding, this can affect user experience. To improve performance, we first collect all method invocations in the program and build a distinct set of calls. Then we perform type checking on this set, significantly improving performance. The performance of error reporting is evaluated in (§7.2).

6.2 Transformation to the Deep Embedding

The transformation from the shallow to the deep embedding is based on the consistent mapping of types from the shallow version to the deep. Each type that appears in the DSL block must have a corresponding abstract type member in the DSL component. With LMS, a non-function type `T` is transformed into `this.Rep[this.T]`. With polymorphic embedding, non-function type `T` is translated to `this.T`. For function types, each type variable is transformed according to the previous rule. The type transformation is a configuration parameter of Yin-Yang and can be arbitrarily defined, for example to include certain exceptions to the transformation rules.

The scala type system supports local type inference thus type ascriptions are rarely necessary. However, DSL users can use type ascriptions at arbitrary locations. We use the consistent type transformation to translate types, ascribed or inferred, in variable definitions, function parameters, function return types, method arguments, method return types, and method definitions. The only exception are the method type arguments where, for LMS, we transform types according to the following rule $T \rightarrow this.T$.

Yin-Yang also imposes the restriction on the structure and method signatures of the deep embedding. All objects appearing in the shallow EDSL must exist in the DSL component. Every application of the function in the DSL program must, after type transformations, be valid in the deep embedding. If this is not the case, Yin-Yang will report the error about missing DSL features as described in (§6.1).

With the given restrictions, the transformation is achieved in four phases: *i) Virtualization* translates language constructs to method calls giving the program a regular structure, *ii) Ascription* places type ascriptions in the code to assure successful type checking, *iii) Lifting* replaces captured identifiers and literals with method calls, and finally *iv) Scope Injection* inserts the DSL body into the scala component. We present all of the phases in Figure 11, on the example DSL for regular expressions.

Virtualization replaces language constructs with method calls following the rules from Virtualized Scala [15]. The translated constructs are: conditionals, loops, **new**, **return**, mutable variable operations, and the **try** construct. Since `infix` methods are not allowed, we additionally transform all operations on the `AnyRef` type which corresponds to the `Object` type in the Java class hierarchy. For example in Figure 11, the `if(c) t else e` construct is translated into the `__ifThenElse(c, t, b)` method call. Class definitions are currently not supported and their presence in the method body is regarded as an error. *Virtualization* normalizes the DSL body to contain only method applications, identifiers, functions, and objects.

Ascription tries to cope with limited type inference of existential `Rep[T]` types. Type inference of `Rep[T]` types can fail in the deep embedding although the shallow version succeeded. To assure that type inference will be successful we introduce explicit type ascriptions in the DSL body. The ascriptions are generated by extracting inferred and ascribed types from the shallow DSL, and transforming them with the type transformer.

The annotations are placed on function parameters, return types, method invocations, and variable definitions. In Figure 11, we see the higher-order function passed to `map` with ascribed parameters and the return type. Variable `str` is ascribed with `Rep[String]` and all method invocations with its transformed return type. Methods `lift` and `hole` are at this phase still not present so they are not ascribed.

```
import regex._
val text = "IBM";
val pattern = "HAL"
regexDSL {
  val ti = text.map(incChar)
  if (matches(ti, pattern))
    println("OK")
}
```

(a) A shallow DSL program.

```
import regex._
val text = "HAL";
val pattern = "IBM"
regexDSL {
  val ti = text.map(x =>
    regex.‘package‘.incChar(x))
  if (regex.‘package‘.matches(ti, pattern))
    scala.Predef.println("OK")
}
```

(b) The desugaring of program a).

```
val text = "HAL";
val pattern = "IBM"

new RegexDSL { def main() {
  val ti: Rep[String] =
    hole(typeTag[String],1).map(
      {x:Rep[Char]=>
        this.regex.incChar(x): Rep[Char]
      }:Rep[Char]=>Rep[Char]):Rep[String]

  (this.__ifThenElse(
    this.regex.matches(
      ti,
      hole(typeTag[String],2)
    ): Rep[Boolean],
    this.println(lift("OK")):Rep[Unit],
    lift(())
  ): Rep[Unit])
}}
```

(c) The deep embedding of program a).

Figure 11: Transformation of the DSL program that supports regular expressions, conditionals, and primitive types.

We do not prove that *Ascription* will ensure type checking. However, from practical knowledge about the type checker, we believe that ascription will achieve its goal. In the evaluation we did not have any problems with type inference of `Rep[T]` types.

Lifting converts literals and captured identifiers to EDSL types. The *Lifting* phase replaces all literals `l` with the invocation of `lift(l)` and captured identifiers `i` with `hole[i.type](typeTag[i.type],i.symbolId)`.

These methods propagate the type information, and in case of holes, the identifier of a symbol to the DSL. In effect, this obviates the cumbersome mechanism for lifting variables. Figure 11 shows how the `"OK"` literal is lifted and how `text` and `pattern` identifiers are replaced by holes.

The captured literals can be variables, fields, functions, and methods. Since Yin-Yang currently supports only DSLs without non-DSL features, we can only lift variables, fields, and functions and methods without parameters. Allowing, method parameters could result in arbitrary combinations of Scala and the DSL code, which can be powerful but confusing as well. Mixing Scala and DSL code will be an optional feature in the future, with the consequence that DSLs must be optimized at run time.

Scope Injection injects the DSL program into the component of the deep DSL. However, all of the DSL methods are invoked on the objects of the shallow DSL. For the scope injection to be valid, all objects must be translated to their component counterparts. Since the program is regular, this becomes a trivial transformation. An exception to this translation is the object that contains the definition of the shallow embedding. Its methods are invoked directly on **this** to avoid unnecessary indirection.

After *Scope Injection* we type-check the generated DSL. The type-checked deep DSL is then reflectively compiled and its instance acquired. The reflective compilation generates bytecode which is then loaded into the class-path of the compiler. The evaluated instance of the DSL is then passed to the later phases for DSL specific analysis, by using the interfaces described in (§5). If errors in this phase occur, this indicates that the deep embedding does not obey the restrictions imposed by the framework or that there is an error in the Scala compiler.

6.3 Captured Identifiers

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

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

Once the required variables are returned to Yin-Yang the whole transformation phase is ran again, excluding the *Feature Analysis* phase. This time the information about required variables is respected and required identifiers are replaced with the `lift` invocation.

On the example from Figure 11c, the `hole` with the symbol identifier 2 carries information about the regular expression which can be used for optimization of the matching automaton. The DSL would return a list containing the number 2. In the following transformation, this hole would be replaced with a `lift` method equivalently to literals. This imposes that code generation and optimization must be done at run time.

6.4 Compile Time Code Generation

When optimizations do not require run time values, code generating DSLs can be completely executed at compile time. To acquire generated code Yin-Yang invokes the `generateCode` method from the `CodeGenerator` interface. The DSLs are required to return a string containing a class named by the passed argument which extends a Scala function. The number and the type of arguments of the function must match the number and types of captured identifiers. The `clsNme` parameter on the method is used by Yin-Yang to provide unique class names.

Then Yin-Yang parses, type-checks, and returns the generated class, together with its invocation that contains identifiers that were marked as holes. After code generation, the body of the shallow DSL is discarded and only the generated code is returned by the macro.

6.5 Run Time DSL Compilation

Run time compilation is a costly operation and therefore should be performed only when required. In order to minimize the run-time overheads, Yin-Yang installs a guard statement before the DSL recompilation. The guard checks if the variables required for optimization have the same values as in the previous run. When the guard is satisfied, the pre-optimized DSL program is executed, and when the guard fails the DSL is re-optimized with new captured values.

For run time DSL compilation, Yin-Yang returns the DSL component containing the program, together with the invocation of the main method. If the DSL is interpreted, the `interpret` method is invoked. In case of DSLs that generate code the `compile` method is invoked along with the invocation of the `compile`'s return value. The arguments passed to `interpret` and `compile`'s return value are the captured identifiers that are not required for optimization.

The code emitted by Yin-Yang for example from Figure 11 is presented in Figure 12. Figure 12a presents the initialization of the DSL, which is the same for both interpreted and code generating DSLs. The Figure 12b presents the guard statement for code generating DSLs, while Figure 12c presents the guard and invocation of the interpreted DSLs.

```
val text = "HAL";
val pattern = "IBM"
class name_<uid> extends RegexDSL {
// code transformed by Yin-Yang
}
val inst = Storage.lookup(<uid>,
  () => new name_<uid>)
val reqVals =  (Seq(pattern), Seq())
// guard from b) or c)
```

(a) Initialization of the DSL program.

```
def recompile(): () => Any =
  inst.compile[Unit, String => Unit]
Storage.check[String => Unit]
  (<uid>, reqVals, recompile).apply(text)
```

(b) Guarded compilation of the DSL.

```
def reset(): () => Any = inst.reset
  Storage.check(<uid>, reqVals, reset)
inst.interpret[Unit](text)
```

(c) Guarded interpretation of the DSL.

Figure 12: Guarded execution of interpreted and code generating DSLs.

In Figure 12a, `inst` will contain the instantiated DSL component. To avoid costly instantiation of the DSL component, which is in case of LMS a compiler, we store the instances in the `Storage` object. The instances are fetched with the `lookup` method that accepts a unique program identifier and the closure that initializes the DSL instance, and returns the instance. Internally, instances are stored in the concurrent data structure.

The variable `reqVals` contains the identifiers that the guard will check. The value of `reqVals` contains two sequences of values, one for values that are checked for equality and the others that are checked by reference value. Besides the primitive types, types that are checked by equality are specified by the DSL author in the configuration of Yin-Yang. The reference values checked by equality are stored as `WeakReferences` and they do not introduce memory leaks. Our example will compare type `String` by semantic equality.

The DSL program instance `inst` and captured identifiers `reqVals` are used in the guard statements presented in 12c and 12b. The guard is externally implemented in `check` and it compares equality of current with previous values. In case of failure the function passed in the last argument is executed. For code generation, this function re-optimizes and compiles the new program, and for interpretation it simply resets the state of the DSL instance.

After the guard, we invoke DSLs by passing captured identifiers to the main method. For interpretation this method is `interpret` and for code generating DLSs it is a Scala function produced by compilation. In our example we pass the `text` variable as it is not required for optimization.

Without meta-programming the DSL body needs to be invoked every time, and all of its constants (not only captured identifiers) need to be compared to the previous values. This imposes constant run-time overheads to executed DSLs which, in some cases, is larger than the performance gains from the DSL implementation. We compare invocation overheads of Yin-Yang's guards to Delite in (§7.3).

Guards presented in 12 are similar to the guard statements introduced by Just-in-time (JIT) compilation in virtual machines. However, DSL guards check assumptions about the whole DSL bodies instead of single methods. Additionally, guards in Yin-Yang are installed at program level and can be configured by the DSL author, which on the VM can not be done. In future versions of Yin-Yang we will allow DSL authors to specify the exact guard statements and DSL recompilation policies.

With DSL generated guards, we open new grounds for domain-specific JIT-compilation. The domain knowledge can be used to define efficient guards which collect statistics, check data lengths, and based on those, decide about the compilation strategy. Parallel implementation of the shallow embedding allows avoiding DSL compilation completely with small data sizes and in code paths that are infrequently used. Unlike VM JIT compilers, which choose between interpretation and code generation, DSLs can choose between the shallow embedding, deep embedding with interpretation, and deep embedding with code generation.

On the example from Figure 12, the DSL author could define the JIT compilation policy. The policy would use the shallow embedding if code is infrequently used, or the value of `regex` changes often. Interpretation would be used when `text` is large, the `regex` is stable but the code is rarely invoked. The code generation would be used when the code is frequently used with a stable value of `regex`. This would provide the best trade-off between execution performance and the length of compilation times. We leave definitions of policies related to DSL JIT compilation for future work.

7. Evaluation

To evaluate Yin-Yang we have introduced the shallow embedding for DSLs OptiGraph from Delite, Scala Collections from LMS, and introduced an example linear algebra DSL using polymorphic embedding. On these DSLs we investigate the process and results of adopting the embedding in (§7.1), the efficiency of type checking and error reporting in (§7.2), and performance of guarded recompilation in (§7.3).

All our benchmarks were performed with Scala 2.10.1, on the JDK 1.7_10 with the JIT compilation stabilized. Every reported number was calculated as an average of 10 runs. The benchmarking was performed on Intel Core i7-2600K CPU running at 3.40GHz, with dynamic frequency scaling disabled, and with 16 GB of dual-channel DDR3 memory running at 1333 MHz.

7.1 Correctness of Yin-Yang

To evaluate the correctness of Yin-Yang transformations in both LMS and polymorphic embedding based DSLs we have implemented shallow embeddings for the OptiGraph DSL, the Scala Collections DSL based on LMS, and the linear algebra DSL with polymorphic embedding. In all of the DSLs, the transformations have been correct and the shallow embedding required no annotations. On LMS based DSLs we evaluate the number of user provided annotations that were obviated, the number of lines of code needed to adopt Yin-Yang, and the effort required.

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

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

The Scala Collections DSL in LMS contains frequently used collections and combinators on for their manipulation. The application suite consisted of 5 TPCH [6] queries, the application for finding mnemonics for phone numbers, and the word count application. After conversion, the applications had no annotations and the code was exactly the same as the original. The applications had 112 lines of code with 7 `Rep[T]` annotations and 17 `unit(_)` invocations. In the deep embedding 15% of lines had annotations.

The adaptation of the tool required the introduction of the shallow embedding and implementing base Yin-Yang traits. The shallow embedding was trivial to implement since the functionality already exists in Scala collections. The implementation of Yin-Yang methods required one working day and consists of 147 lines of code.

The linear algebra DSL was implemented as the proof of concept for polymorphic embedding. It consists only of a single method for every class of methods we found in the OptiLA DSL. The test suite contains 15 test cases that cover functionality of all introduced methods. Since there were no applications with the deep embedding we do not report the number of obviated annotations.

Time (ms)	Page-Rank		SCC		Conductance		Synthetic (500 LoC)		Synthetic (1000 LoC)	
	Delite	YY	Delite	YY	Delite	YY	Delite	YY	Delite	YY
Correct Program	848	565	537	346	481	472	1149	291	1883	432
Incorrect Program	531	95	194	28	159	30	761	60	1407	101
DSL Feature Error	463	137	177	151	151	183	743	249	1414	297
Feature Error Perceived	343	1	9	2	17	2	368	2	669	2
Error Perceived	390	7	8	4	17	9	398	24	679	43

Table 1: Type-checking times for Page rank, strongly connected components (SCC), graph conductance algorithm, and two synthetic benchmarks of 500 and 1000 lines of code. Time is measured for correct programs, incorrect programs, programs using a non-existing construct, and for incorrect programs until the type error is perceived.

7.2 Type Error Reporting

The type-checking of the shallow embedding certainly outperforms the deep embedding. However, in the shallow embedding Yin-Yang needs to check for absence of methods that do not exist in the deep embedding. This requires additional type checking for every method in the DSL body. In this section we quantify the times required for type checking.

We evaluate type checking times (Figure 1) on applications for calculating Page rank, strongly connected components (SCC), and conductance in a graph. Additionally we include two synthetic applications containing 500 and 1000 lines of code, to represent behavior with large DSL programs. For each of the applications we compare times between Delite and Yin-Yang in the following cases: *i)* compilation of the correct program, *ii)* compilation time of the incorrect program with a typing error in the program, *iii)* DSL compilation time if a non-existing method is used, *iv)* time until feature error is printed to standard output, and *v)* time until the typing error is written to the output. All errors placed were type errors close to the middle of a program.

With Yin-Yang the complete compilation of the correct program is 1-4x faster than the Delite versions. Incorrect programs finish compilation 5-14x faster. Non-existing feature errors are compiled 1-4.5x faster. Errors with feature errors are perceived up to 300x sooner and typing errors are perceived 2-55x sooner. The JVM of the compiler was ran 50 times before each benchmark to JIT compile the entire Scala compiler. In practice, this is rarely the case and users run the compiler without intensively warming it up. The compilation times for the first run are about 5-15x longer.

The compilation against the shallow embedding greatly improves compilation times of EDSLs. This allows EDSLs to be used with IDEs which type-check the whole source file on every key stroke. In case of large applications with Delite type-checking lasts more than 1 second. This causes IDEs to become unresponsive and practically useless. With warm JVMs the compilation times are acceptable for users if only one file is compiled. However, JVM is rarely warm and compilation often affects multiple source units. In these cases Yin-Yang greatly reduces the time from compiling to error detection.

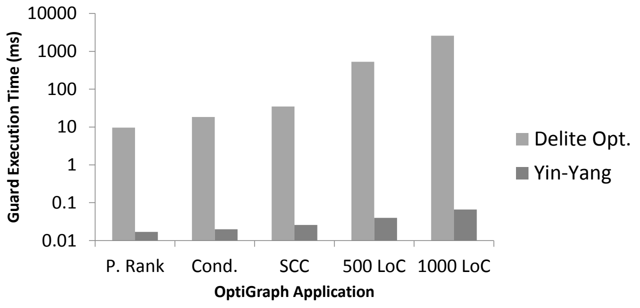

Figure 13: Guard execution times in Delite and Yin-Yang.

7.3 Guarded Recompilation of DSLs

To evaluate the performance gains from guarded recompilation, introduced in (§6.5), we compared guards used in Delite and guards from Yin-Yang in Figure 13. For the comparison we used Page rank, SCC, and Conductance applications from OptiGraph as well as 2 synthetic applications containing 500 and 1000 lines of code. In Page rank, SCC and Conductance the DSL captures 2 identifiers of small size, while in synthetic benchmarks the DSL captures 5 identifiers.

Delite currently uses guards based on comparison of the optimized IR with a previous version. This takes on average 11 seconds so it was not suitable for comparison. To overcome this, we have implemented the optimal scheme that can be achieved without replicating the behavior of Yin-Yang. Our optimization executes the main body, that builds the IR, and on the way collects all the constants that were introduced. In the benchmarks we compare only against this scheme. Our benchmarks did not have staging constructs, which can evaluate for a long time. For example, the staged power function `pow` from example 2, would unroll the recursion as a part of the guard.

Checking the guard in OptiGraph applications has the overhead of 10 - 35 ms per invocation of the program. For larger programs, the guard execution takes 528 ms for 500 lines of code and 2593 ms for 1000 lines of code. Since the whole IR needs to be constructed, the execution of the guard in Delite takes proportionally to the length of the program.

With Yin-Yang the guard execution for 2 captured parameters is minimal and lasts on average 0.02 ms. In synthetic benchmarks we capture 5 parameters and the guard executes in 0.053 ms. Yin-Yang guards take constant time, proportional to the number of captured identifiers, and perform more than 560x better for short programs and several orders of magnitude better for large programs.

The 10-35 ms guard execution is little if DSLs are used for processing large data sets. However, in latency sensitive cases, like high frequency trading and web servers, and in tight loops, this imposes a significant overhead. For example, if a dynamic web page is implemented as a deep DSL in 500 lines of code, every request would have a constant overhead of 528 ms. Furthermore, we executed the program in Figure 11a. The shallow version can process a 100 KB string in 4 ms, which is how long the guard execution takes. Here the guard overhead is greater than the performance gains for common data sizes.

With Yin-Yang guarded recompilation we enable usage of EDSLs in broader range of contexts and bring them closer to user applications. Users can use DSLs on any size of data, inside tight loops, web applications, compilers, and reactive programs, without imposing significant overheads.

8. Discussion

Yin-Yang resolves most of the issues users have with deep EDSLs. It enables comprehensible type errors, fast type checking, debugging, documentation, and allows DSLs to be used in all contexts without run-time overheads. However, hiding LMS staging constructs from users prevents the usage of user-controlled multi-stage programming and makes it available only to the DSL author. This can lead to suboptimal code in some cases. However, Yin-Yang is not imposed on the DSL users, so performance critical code can always be implemented without the shallow front-end.

From the perspective of the DSL authors, it can be viewed as a simplified interface for meta-programming specialized for EDSLs. Through virtualization, which normalizes compiler structures to method calls, and interfaces defined in (§5) it minimizes the knowledge required about the compiler internals. Furthermore, it allows DSL authors to design DSLs in the same manner independently of the compilation stage at which the DSLs are optimized.

With Yin-Yang, DSL authors have an overhead of maintaining the shallow embedding. Although shallow embedding requires much less effort, code duplication can impose bit rot and thus bugs. However, if the DSLs do not require debugging, the shallow embedding can just serve as an interface, without any implementation. In future work, we plan to introduce automated generation of DSL components based on shallow embedding enhanced with required annotations. This will reduce the duplication to minimum and DSL authors will have to implement only optimization and code generation modules for the deep embedding.

In the case that shallow embedding is used for debugging, the semantics of the deep embedding must entirely match the shallow one. For user defined types this does not pose an issue, however with standard Scala types it narrows down possible interpretations in the deep embedding. For example, if one would like to introduce a DSL for addition modulo 64, the semantics of types in Scala would not be suitable. One would need to introduce another deep embedding with overridden primitive types that would be used for debugging. We believe that keeping strict semantics prevents language fragmentation and makes comprehending and using DSLs easier. DSLs with special semantic requirements are rare, and for those one can either omit debugging or introduce another interpretation layer.

Yin-Yang can currently operates on programs within a single compilation unit. Scala macros have limited access to IRs from pre-compiled code and this limits Yin-Yang's ability to analyze and transform them. This issue can be solved by providing equivalent deeply embedded methods for each method used in the DSL. This would require a single annotation per shallow component which would invoke Yin-Yang to generate the component's deep embedding. This version could then be used from other compilation units. This is currently not implemented as tools required for this transformation are still experimental. We will introduce this feature once Scala introduces macro annotations.

9. Related Work

Svenningsson and Axelsson [26], identify that shallow embedding can complement the deep embedding. They propose to use a minimal core deep embedding with a complementing shallow interface that implements all the functionality in terms of the core interface. They present how options, vectors, pairs, and other constructs can be represented in terms of the small core calculus. In their approach, generic types in the shallow interface must have a type class `Syntactic` which defines conversions from and to the deep embedding. The described technique is used in Feldspar DSL [2].

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

Kansas Lava [8] is an extension of the EDSL Lava used for hardware description that introduces parallel shallow and deep embedding of the EDSL. In the work-flow of Kansas Lava the native Haskell implementation is used first for prototyping. Then the programmer introduces applicative functors (shallow embedding) that are used for modeling timing constraints in hardware, and finally removes the functors and introduces the deep embedding. The approach of keeping two embeddings for prototyping is the same as in Yin-Yang. However, the translation is achieved manually inducing additional development overheads. With Yin-Yang timing analysis would be performed as the domain-specific analysis and the native implementation could be automatically translated to the deep embedding.

Issues related to error reporting with complex type systems are a long standing problem. Hereen et al. [11] introduce extensible error reporting for type inference based type systems. DSL authors can introduce custom error messages based on the constraints collected by the type system. This results in arguably better error reporting than Yin-Yang but comes with an overhead of implementing these messages. Additionally DSL authors are required to know internals of the type inference process in Haskell.

10. Conclusion

We presented Yin-Yang, a DSL embedding library that reliably translates shallow EDSLs to the deep EDSLs that are based on polymorphic embedding and LMS. The translation allows leveraging virtues of the shallow embedding during program development and good performance of deep embedding when it is of essence. Through language virtualization, and a minimal interface, Yin-Yang allows DSL authors to design compiler agnostic DSLs that perform domain-specific analysis at compile time and the DSL compilation at either compile or run time. Yin-Yang enables efficient guarded recompilation of DSLs at run time allowing deep EDSLs to be used in latency critical applications and on small data sets. Yin-Yang opens new directions for researching efficient policies in JIT compilation of deep EDSLs accompanied with the shallow embedding.

Acknowledgments

This work was financed with a grant by the European Research Council (ERC). We thank Hubert Plociniczak for proofreading, Tiark Rompf, Stefan Zeiger, and Jasson Zaug for providing useful insights about EDSL design. Finally, we thank Eugene Burmako for explaining compiler internals and for fixing compiler bugs with extraordinary agility.

References

[1] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-toend Management of Big Data*, 2012.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckeg\aard, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, page 169–178, 2010.

[3] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, page 89–100, 2011.

[4] E. Burmako and M. Odersky. Scala Macros, a Technical Report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[5] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. 45(6): 363–375, 2010.

[6] T. P. Council. Tpc benchmark^{TM}, http://www.tpc.org/tpch/.

[7] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, template haskell, and c++. page 51–72, 2004.

[8] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing kansas lava. In *Implementation and Application of Functional Languages*, page 18–35. Springer, 2011.

[9] M. Guerrero, E. Pizzi, R. Rosenbaum, K. Swadi, and W. Taha. Implementing DSLs in metaOCaml. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, page 41–42, New York, NY, USA, 2004. ACM.

[10] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. 410(2):202–220, 2009.

[11] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *ACM SIGPLAN Notices*, volume 38, page 3–13, 2003.

[12] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, page 137–148, 2008.

[13] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript as an embedded DSL. page 409–434, 2012.

[14] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice*. PhD thesis, PhD thesis, Katholieke Universiteit Leuven, 2009.

[15] 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.

[16] M. Odersky and A. Moors. Fighting bit rot with types. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4, page 427–451, 2009.

[17] M. Odersky and M. Zenger. Scalable component abstractions. In *ACM Sigplan Notices*, volume 40, page 41–57, 2005.

[18] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification*. Citeseer, 2004.

[19] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. 45(10):341–360, Oct. 2010.

[20] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. 55(6):121–130, June 2012.

[21] T. Rompf, A. Sujeeth, H. Lee, K. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. *Arxiv preprint arXiv:1109.0778*, 2011.

[22] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 497–510, 2013.

[23] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, page 1–16, 2002.

[24] A. Sujeeth, T. Rompf, K. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of ECOOP*, 2013.

[25] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML*, 2011.

[26] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. 2012.

[27] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, page 203–217, 1997.

[28] V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky. StagedSAC: a case study in performance-oriented DSL development. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, page 73–82, 2012.

[29] T. Veldhuizen. Template metaprograms. 7(4):36–43, 1995.

[30] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, page 1–14, 2008.