École Polytechnique Fédérale de Lausanne
School of Computer and Communication Sciences IC

Master Thesis

Making supercomputing enjoyable: Improving virtualization and compilation efficiency of high-performance domain-specific languages embedded in Scala

Thesis conducted from October 1, 2015 to March 30, 2016,
at Pervasive Parallelism Laboratory at Stanford University

Author:
Boris Perović
boris.perovic@epfl.ch

Supervisors:
Professor Martin Odersky
martin.odersky@epfl.ch
Vojin Jovanović
vojin.jovanovic@epfl.ch
Programming Methods Laboratory (LAMP)
École Polytechnique Fédérale de Lausanne

Professor Kunle Olukotun
kunle@stanford.edu
Kevin J. Brown
kjbrown@stanford.edu
Pervasive Parallelism Laboratory (PPL)
Stanford University

Abstract

Computing technologies are experiencing a rift - while software engineering is getting more accessible every day, computer hardware and systems are getting ever more complex. The persistent trend in programming languages is the move towards higher levels of abstraction, which allow programmers to express their ideas more concisely and solve complex problems easily. On the other hand, hardware technologies are becoming more heterogeneous, often incorporating parallelism as a default construct. The gap of abstraction between the high-level code and hardware is becoming evident, making it very difficult for compilers to successfully bridge. This prevents programmers from taking the full advantage of the computing resources available to them, often leading to many man-hours invested into exploring and understanding low-level programming models designed for exploiting the capabilities of modern heterogeneous architectures. Lightweight Modular Staging (LMS) and Delite compiler frameworks together present a powerful infrastructure for the development of high-performance parallel domain-specific languages (DSLs) capable of overcoming this gap. Using these frameworks, DSL authors are able to provide both productivity, through a high-level, restricted language interface, and performance, through generation of efficient parallel code for heterogeneous platforms. In this thesis we make several contributions to the current LMS / Delite ecosystem. We prove that it is possible to successfully remove the dependency on a custom version of the Scala compiler, Scala-Virtualized, by employing a more principled approach of compile-time metaprogramming using Scala Macros. We perform a detailed analysis of the compilation performance of various DSL applications and show that the mixin phase and implicit resolution can contribute up to 60% of the total compilation time, individually contributing more than 40% in various benchmarks. To alleviate this, we implement a new shallow, direct embedding of DSLs, providing support for its automatic generation using the meta-DSL Forge, and demonstrate significantly improved compilation performance, obtaining 3-5x speedups when compared to the old shallow embedding. By doing so, we additionally hide the deep embedding abstractions from the user, offering a cleaner interface to Delite DSLs. Lastly, we show that we can improve the compilation times of deep embeddings by a factor of 2x, using a custom types approach.

Acknowledgements

I am truly grateful to Professor Kunle Olukotun for believing in me and providing me with this remarkable opportunity, to Professor Martin Odersky for teaching and inspiring me immensely through his writing, work and recorded talks, to Kevin J. Brown for incredible support, discussions, jokes, and accepting me as an equal, all while mentoring me throughout the project, to Vojin Jovanović for introducing me with the project and the people involved, pushing me to challenge myself, and discussing many project-related undertakings while mentoring me from EPFL's side, to Professor Tiark Rompf for discussions and mentoring support, to Madame Sylviane Dal Mas for being there to help during my master studies at every step of the way, and to all the members of the Pervasive Parallelism Laboratory at Stanford University, for all the smiles, intellectual challenges, lunches, and fun times, which made me feel very welcome and happy to be a part of this research group.

Additionally, I would like to express gratitude to my parents Vladimir and Branka Perović for their unconditional love and support, to my brother Darko Perović for his humor and adventurous spirit, to Nina Desnica for always trying to make me better, to Marko Baroš for becoming my best friend in Lausanne, to brothers Plavšić for providing me with invaluable advice and mentorship, to Nenad Božidarević, Jelena Marković, and Ognjen Marković for making the Bay Area feel like home, and to all the rest of my friends from Belgrade, Lausanne, and Stanford, for their great energy and love.

Table of Contents

Table of Figures

1. Introduction

Computers run the world. From everyday business operations to space exploration and advanced science, it is almost impossible to find a field which could operate efficiently and successfully without computers - our civilization is fundamentally dependent on computing technologies and services. As computer science becomes ever-more ubiquitous, the interest in the field rises globally and more people are starting to program computers to cater to their personal or business needs.

Naturally, there is a tendency towards making programming languages easier to reason about and use, making programming more accessible. Modern programming languages achieve this by raising the level of abstraction which they present to the programmer. High abstractions allow programmers to look at the "big picture" and express their ideas more clearly, leading to maintainable and easy-to-understand code. But these abstractions come at an inherent cost - they introduce a new layer of indirection between hardware and software.

Compilers aimed at general-purpose languages (GPL) often have difficulties compiling high-level programs to efficient code for modern hardware platforms. This is due to the fact that the high-level constructs usually present a challenge to efficiently map to low-level programing models and resources provided by hardware. Additionally, general-purpose compilers have a huge number of choices available for optimization for various platforms, making the optimization search space huge and often intractable to traverse in order to find the optimal solution. Furthermore, general-purpose compilers lack any information about the domain that the programmer is dealing with, which prevents them from making certain optimizations, leading to suboptimal performance.

In an alternative to this, programmers that seek performance are faced with the task of navigating a growing landscape of hardware architectures. Modern hardware platforms offer great performance, driving innovation in emerging scientific areas and applications such as machine learning and deep neural networks. Nevertheless, they expose a high level of complexity to the end user. Namely, these platforms are becoming increasingly parallel and often incorporate multiple heterogeneous processing elements (CPUs, GPUs, hardware accelerators…). In order to take advantage of the available performance and everything that these architectures have to offer, programmers are forced to learn low-level, hardware-specific programming models. Not only do they have to understand how to use these various programming models, but also which one is best suitable for a particular type of problem and even how to best combine them for a specific application. As a result, the code becomes tied to architectural details and specifics of the used programming models, making it much harder to maintain or port to other platforms. All of this greatly reduces programmer productivity when developing applications aimed at achieving high-performance.

Domain-specific languages are an active area of research offering a promising approach to tackling this problem. In contrast to general-purpose languages, aimed at arbitrary computations, domain-specific languages have a goal of modelling and dealing with a specific domain. They offer a restricted view of computation, providing users only with abstractions directly related to the domain. Although at a first sight this might seem as a shortcoming, it actually provides some important benefits. On one hand this restricted view leads to increased productivity, as programmers are able to express their solutions more concisely and clearly, in a manner more closely related to the domain they are dealing with. On the other hand, the compromise of reducing the expressive power of a language allows for attaining high-performance - as a compiler has more intimate knowledge of the domain, it is able to map domain abstractions to efficient implementations on modern hardware and perform very aggressive optimizations which would be dangerous or even unfeasible in a general-purpose language due to a lack of domain semantic.

Domain-specific languages can traditionally be separated into two categories:

- External domain-specific languages
- Internal domain-specific languages

External domain-specific languages are standalone languages that have a dedicated compiler. While they are able to provide performance, developing them requires a tremendous programmer effort. The first obvious challenge that the DSL author faces is developing a compiler from scratch. This includes implementing somewhat generic features of a compiler - parser, type-checker, generic optimizer etc. Furthermore, the DSL author might need to add debugging capabilities, integrated development environment support, various other tools and documentation. Having all of that in mind, more often than not the difficulty of developing a standalone DSL outweighs the benefits gained from using it.

Internal domain-specific languages present an attractive alternative to external DSLs [1]. Being embedded in a general-purpose host language, they are automatically provided with all of the common blocks necessary for building a language, including a compiler pipeline, IDE support, various tools and libraries. While there are various ways in which a language may be embedded in a host language, two general categories can be identified:

- shallowly embedded domain-specific languages
- deeply embedded domain-specific languages

Shallowly embedded domain-specific languages can be thought of as libraries. This means that they represent embedded language values and other constructs directly by values and constructs in the host language [2]. While this kind of embedding is easy to develop and use, it is fundamentally limited by the host language - a general purpose host language compiler is unable to perform domain-specific optimizations, constraining performance, and the code written in the embedded domain-specific language can run only on host-supported targets.

Deeply embedded domain-specific languages take on a different approach. Instead of directly embedding their constructs into the host language, they represent them symbolically in the form of an intermediate representation (IR). This more advanced method allows for domain-specific optimizations to be performed on the IR by a domain-specific framework. Having an optimized IR, the framework is then able to generate code and target arbitrary architectures, not necessarily tied to the ones supported by the host language compiler. All of this results in much more performant code, comparable to the one obtained when using an external DSL, but comes at the cost of a more difficult language development. Furthermore, the difficulty of using the language is increased, as the abstractions needed to reify DSL code to an intermediate representation often leak to the programmer, cluttering the interface and making it more difficult to both use and debug.

Ultimately, in the domain of high-performance computing, it would be very significant if we could achieve the flexibility and performance available to external DSLs, while maintaining the ease of development of a language similar to the one required for creating a shallowly embedded language.

Lightweight Modular Staging [3] is a compiler framework developed in Scala [4], aimed at tackling this problem. It provides facilities for building DSLs embedded in Scala and creating optimizing domain-specific compilers at a library level [5]. In order to enable this, LMS uses a fork of Scala compiler named Scala-Virtualized [6], that enables overloading of built-in Scala constructs, making it feasible to reify those constructs to a suitable representation. LMS framework employs the idea of multi-staged programming, a principled runtime code generation approach to programming. Historically, multi-staged languages and frameworks such as MetaML [7] have used syntactic annotations in order to mark staged expressions. LMS takes a different approach, successfully employing finally tagless [8] and polymorphic embedding [9] ideas. This means that LMS deals with staging at the level of types – it relies on a powerful type system provided by the Scala programming language in order to encode staged pieces of code. Moreover, LMS uses overloaded operators to combine staged code fragments in a semantic way, unlike quasi-quotation approaches that act as merely syntactic expanders. Additionally, LMS comes with strong well-formedness and typing guarantees, mostly inherited from the finally tagless embedding [5]. Finally, LMS allows developers to tightly integrate domain-specific abstractions and optimizations with the existing compiler infrastructure and generic optimizations provided by the framework, making it very suitable for the development of highly-efficient domain-specific languages.

Delite compiler framework [10] builds on top of LMS by adding parallel patterns, additional code generators and support for execution on heterogeneous targets. The result is a compiler framework and runtime for high-performance parallel embedded domain-specific languages. In order to enable rapid construction of high-performance, highly productive DSLs, Delite provides several facilities:

- Built-in parallel execution patterns
- Optimizers for parallel code
- Code generators for Scala, C++ and CUDA
- A DSL runtime for executing on heterogeneous architectures

Additionally, the framework is able to perform useful parallel analyses, allowing for expressing parallelism in DSL operations, but also among them. A Delite DSL program is translated into a machine-agnostic intermediate representation, which is scheduled and executed on a chosen target through the Delite Runtime.

Unfortunately, the dependency on a specialized version of a compiler makes LMS brittle and prevents using the latest developments in the language (specialized compiler needs to be regularly updated and tested, which is a laborious process). Additionally, applications written using domain-specific languages developed with LMS and Delite exhibit high compilation times for both shallowly embedded and deeply embedded variants, making it difficult for programmers to iterate quickly and prototype programs. Furthermore, due to the current design, abstractions from the deep embedding leak into both shallow and deep application code, making it more difficult to understand and maintain.

This thesis tackles these problems by replacing the specialized version of Scala compiler by a more principled metaprogramming approach and applying this solution to real-world Delite DSLs. Additionally, it analyzes current embeddings and explores new ones which seem promising in providing both a nicer interface to Delite-based languages and improved application compilation times.

The main contributions of this thesis are the following:

- We successfully applied macro-virtualization to real-world DSLs and showed its limits. Through the use of a combination of Scala Macros [11] flavors, we were able to reproduce the behavior of the Scala-Virtualized compiler, making LMS and Delite independent of it. The new implementation contains a large number of implicit conversions - we note that it is affected by a known problem of some of them not resolving reliably. This behavior motivated the need for a new kind of embedding, less reliant on implicit conversions.
- We analyzed Delite DSL embeddings in depth and identified compilation performance bottlenecks. It was observed that mix-in composition of DSLs and a large number of implicit conversions contribute up to 60% of the total compilation time, each individually contributing more than 40% in various benchmarks.
- We developed a new shallow (direct) embedding, and enabled its automatic generation through the use of the Forge meta-DSL. Avoiding creating large DSL “cakes” [12], instead only importing the required classes, and not utilizing implicit conversions, lead to 3-5x improvement in compilation times of various DSL applications.
- We explored new deep embedding ideas, using a custom types approach. The newly developed deep embedding defines operations directly on types, avoiding a layer of implicit conversions, leading to 2x improvement in compilation performance.

The next chapter will describe LMS and Delite in more detail, including the current implementation and organization, in order to provide the reader with enough understanding to be able to successfully follow the rest of the paper.

2. Lightweight Modular Staging

This section gives an overview of the Lightweight Modular Staging (LMS) compiler framework and introduces the Delite compiler framework for building parallel domain-specific languages. Parts of this section have been based on the paper "Building-blocks for performance oriented DSLs" by Rompf et al. [5].

Due to the advanced design of LMS and Delite, domain-specific languages (DSLs) implemented on top of them can hardly be distinguished from shallowly embedded (library-style) DSLs. A key feature of how the current design is implemented is the separation between the interface and the implementation of a DSLs. The interface is kept as clean as possible and strictly abstract, allowing for combining it with various implementations in the back. Both, the interface and the implementation of a DSL can be assembled from components in the form of Scala traits. Scala traits are similar to Java interfaces, with an exception that they are allowed to have concrete members. They can be combined together and joint with classes through a process called mixin-composition [13]. As multiple traits can be mixed-in into one class (a class can inherit multiple traits), naturally, the "diamond" multiple inheritance problem can occur. Scala deals with this by resolving super-calls according to an inheritance-preserving linearization of all the receiver's parent traits. In simple terms, if there are multiple implementations of a single member, the implementation that is mixed-in the furthest to the right wins.

2.1 Fundamentals

At the base of each LMS / Delite DSL interface lies an abstract type constructor `Rep[T]` that is used to designate an abstract representation of types in a DSL program. Instead of using regular types, DSL programs use this wrapped representation of types - all of the DSL operations defined in the DSL interface are expressed in terms of Rep types.

The implementation is then able to provide a concrete instantiation of this abstract type in several ways. For example:

- `Rep[T] = T` - identity transformation. This would be equivalent to using a DSL as a pure library and would be sufficient if we wanted to stay in the shallow world. The implemented operations would be invoked on types that were wrapped (type parameters of `Rep`), and the applications could be run directly after compilation.

- `Rep[T] = Exp[T]` - expression tree (or in the general case it could be some other representation). This would be equivalent to using a DSL in a multi-staged, compiled manner and would be suitable for deep embedding requirements. Naturally, the implemented operations would work on an intermediate representation (IR) and further code generation and compilation would be required in order to execute the program.

2.2 A simple DSL example

In order to better understand how LMS DSLs work and how they are currently organized, let us take a look at a simple LMS DSL and a corresponding application. As an example, we will use a subset of the SimpleVector DSL, which deals with operations and calculations on numeric vectors. Other than a custom Vector type, it includes some basic Scala types and operations, and additional miscellaneous functions like `println` etc. We name this DSL "MySimpleVector". The application written in this DSL, that calculates an average of 100 random numbers and outputs the result would look like this:

```
trait RandomAverage extends MySimpleVectorApplication {
  def main() = {
    val v = Vector.rand(100)
    println("Random of 100 numbers is: ")
    println(v.avg)
  }
}
```

`MySimpleVectorApplication` trait provides the DSL user with an interface for writing DSL applications in the MySimpleVector domain-specific language. Note that this is only an interface and that this program is still not instantiable and consequently not executable without mixing-in an appropriate implementation. This application can ultimately be run through the creation of a singleton object that mixes-in the appropriate implementation traits.

First, let's look at the interface:

```
// the trait that all MySimpleVector applications must extend
trait MySimpleVectorApplication extends MySimpleVector {
  def main(): Unit
}

trait MySimpleVector extends ScalaOps with VectorOps
```

`MySimpleVectorApplication` declares only one method `main`, the entry point to the application, and mixes in the `MySimpleVector` trait.

`MySimpleVector` trait declares the DSL interface, which the DSL users are able to use to write their applications, without knowing anything about the actual implementation. This is a common principle from object-oriented design and programming called encapsulation. The strict separation

between the interface and the implementation allows for increased robustness and safety [14] [15]. In this concrete example, if we would expose parts of the DSL implementation to a DSL program, the program would be able to observe its own structure, leading to some DSL optimizations that are performed by default becoming unsafe e.g. those that maintain semantic, but not structural equality.

Two traits that `MySimpleVector` mixes in are the following:

- `ScalaOps` contains a set of common Scala operations that are provided by the core LMS library. These include operations on primitive types, control structures, variables, miscellaneous operations like `println`, `exit` etc.
- `VectorOps` expands on the base `ScalaOps` and provides a custom `Vector` type and operations on it

A simplified interface of VectorOps may look like this:

```
trait VectorOps extends Base {
  self: MySimpleVector =>

  type Vector[T]

  object Vector {
    def rand[T:Numeric](n: Rep[Int]):Rep[Vector[T]] = vector_object_rand[T](n)
  }

  def vector_object_rand[T:Numeric](n: Rep[Int]): Rep[Vector[T]]

  def infix_length[T](v: Rep[Vector[T]]): Rep[Int]
  def infix_sum[T:Numeric](v: Rep[Vector[T]]): Rep[T]
  def infix_avg[T:Numeric](v: Rep[Vector[T]]): Rep[T]

}
```

Code Example 1: VectorOps interface

The self type annotation specifies that whenever we would like to instantiate an object that mixes-in `VectorOps`, we need to provide (mix-in) a concrete implementation of MySimpleVector trait too. This is a form of dependency injection mechanism that Scala provides [16] and is a part of the Cake design pattern [12] using which the current LMS and Delite DSL embeddings are built. The abstract type `Vector[T]` represents vectors of elements of type `T`. Static operations on `Vector` are defined in `object Vector`. In this case it is only a function which creates a `Vector` of random numeric values of a given size `n`. Operations on an instance of `Vector` are defined as infix operations, a functionality provided by Scala-Virtualized [6] and will be explained in more detail later. For now, for the purpose of understanding this example, it is only important to know that if there is a call like `x.foo` where `x` is a value of type `A`, the infix mechanism will take over and resolve the call in case there is an appropriately named and typed method in scope - in this case it would be `infix_foo(a: A)`. Otherwise, standard Scala typing rules apply.

Some of the operations can be performed on any type of `Vector` e.g. any `Vector` has a defined length. On the other hand, some more specific operations might be performed only on Vectors containing numeric data types. In Scala code, this is denoted by a type class [17] `Numeric`, which includes types like `Char`, `Int` and `Double`. In this concrete example, a context bound on generic type parameter `T` ensures that `sum` and `avg` operations can be invoked only on Vectors that contain elements of type that is a member of the `Numeric` type class.

It is important to note that all of the operations are declared on and return types of `Rep[T]`. As mentioned earlier, `Rep[T]` refers to an abstract representation of a generic type `T`. In the shallow embedding, this representation can simply be an identity transformation `Rep[T] = T`, which with an appropriate DSL implementation can be immediately executed. In the deep embedding, `Rep[T]` denotes an expression that represents the *computation* of a value of type `T` which will produce the value of `T` in the next computation stage. Namely, from the DSL code, the framework constructs a representation with all of the available non-staged values represented as constants, as they are already evaluated (first stage). Then, the framework is able to generate code based on this representation, which will later get compiled by a target compiler and run (second stage), computing all of the staged values and finishing the execution.

This wrapping of types is a core abstraction of LMS and a way in which it approaches multi-staged programming. At the base of LMS interface hierarchy lies the trait `Base` which defines the abstract type constructor `Rep`:

```
trait Base {
  type Rep[T]
  protected def unit[T](x: T): Rep[T]
}
```

An instance of `Rep` can be obtained from the non-staged value by using the factory method `unit`, or lifted (transformed to an intermediate representation) automatically using an implicit conversion (the details of the second approach are not essential at this point).

2.3 DSL details

In the shallow (library) embedding, the corresponding implementation trait could look like this:

```
trait BaseLib {
  type Rep[T] = T
  protected def unit[T](x: T): Rep[T] = x
}
```

All of the Scala operations can then be simply implemented as operations directly on `Rep` values - the identity transformation defined in the `BaseLib` trait will treat them as regular values of a type wrapped inside `Rep`. Custom types like `Vector` can be implemented in a simple library style.

On the other hand, in the deep embedding, non-staged values can be lifted to constants e.g. through the use of `unit`, as they have been already evaluated in the previous stage of the computation. Abstract type `Rep[T]` can be represented as an expression `Exp[T]` which could either be a constant (`Const`) or a symbol (`Sym`) representing a computation. The corresponding implementation trait of `Base` in the deep embedding could look like this:

```
trait BaseExp extends Expressions {
  type Rep[T] = Exp[T]
  def unit[T](x: T): Rep[T] = Const(x)
}
```

`Expressions` trait defines the base of the concrete intermediate representation used by LMS library and consequently Delite. Its reduced body looks like this:

```
trait Expressions {

  abstract class Exp[T] // constants/symbols (atomic)

  case class Const[T](x: T) extends Exp[T] // constant expression
  case class Sym[T](val id: Int) extends Exp[T] // symbol – to compute

  abstract class Def[T] // base class for all IR nodes
  implicit def toAtom[T](d: Def[T]): Exp[T] = { // binds Def IR nodes to Exps
    findOrCreateDefinitionExp(d)
  }
}
```

Code Example 2: Expressions trait

Sub-traits that extend `BaseExp` in the deep embedding are able to add new concrete IR nodes as subclasses of `Def`, in addition to the ones provided by LMS core library. This approach allows LMS's generic optimizers to view the IR in terms of its base nodes and (`Exp`, `Def`) typed pairs, while the DSL subclasses can make richer nodes carrying domain-specific information and use them at a higher level for optimizations. For the sake of simplicity, we assume that an appropriate implementation of `ScalaOps` can be found in a trait named `ScalaOpsExp`. Looking at a concrete example, the deeply embedded implementation of `VectorOps` could look like this:

```
trait VectorOpsExp extends VectorOps with DeliteOpsExp {
  case class VectorRand[T:Numeric](n: Exp[Int]) extends Def[Vector[T]]
  case class VectorLength[T](v: Exp[Vector[T]]) extends Def[Int]
  case class VectorSum[T:Numeric](v: Exp[Vector[T]]) extends
    DeliteOpLoop[Exp[T]] {
    val range = v.length
    val body = DeliteReduceElem[T](v)(_+_)
  }
  def vector_object_rand[T:Numeric](n: Rep[Int]): Rep[Vector[T]] =
    VectorRand[T](n)
  def infix_length[T](v: Rep[Vector[T]]): Rep[Int] = VectorLength(v)
  def infix_sum[T:Numeric](v: Rep[Vector[T]]): Rep[T] = VectorSum(v)
  def infix_avg[T:Numeric](v: Rep[Vector[T]]): Rep[T] = v.sum / v.length
}
```

Code Example 3: VectorOpsExp implementation

Static method `rand` and the method `length` are implemented as regular new IR nodes, simply extending `Def`. On the other hand, the operation `sum` is defined as a `DeliteOpLoop`, a special class of IR nodes provided by the Delite framework in `DeliteOpsExp` trait. Lastly, the operation `avg` is implemented without creating a new node, using the existing `sum` and `length` operations. Note that all of the implemented abstract methods return a value of a type `Rep`, which in this concrete case is an `Exp`, but the bodies of these methods return values that extend `Def`. This is possible due to the automatic conversion (mapping) between `Def` and `Exp`, performed implicitly by the method `toAtom` in the `BaseExp` trait.

2.4 Creating runnable applications

Having gained an understanding on how the shallow and the deep embedding share the same API and how they are implemented, we can take a look at how can we create a functioning, runnable DSL application.

Let us first remind ourselves that the shared API is composed like this:

```
trait MySimpleVector extends ScalaOps with VectorOps
```

Shallow and deep implementations corresponding to the API can be composed as follows:

```
trait MySimpleVectorLib extends MySimpleVector with ScalaOpsLib with
  VectorOpsLib
trait MySimpleVectorExp extends MySimpleVector with ScalaOpsExp with
  VectorOpsExp
```

The corresponding generic application traits may be composed as:

```
trait MySimpleVectorApplicationInterpreter extends MySimpleVectorLib with
  MySimpleVectorApplication
trait MySimpleVectorApplicationCompiler extends MySimpleVectorExp with
  MySimpleVectorApplication with DeliteApplication
```

Finally, the actual runnable singleton objects corresponding to the RandomAverage application defined above may be defined as:

```
object RandomAverageInterpreter extends RandomAverage with
  MySimpleVectorApplicationInterpreter
object RandomAverageCompiler extends RandomAverage with
  MySimpleVectorApplicationCompiler
```

This sort of embedding that LMS uses is based on the principles of polymorphic embedding of DSLs [9] and from a high level it looks like this:

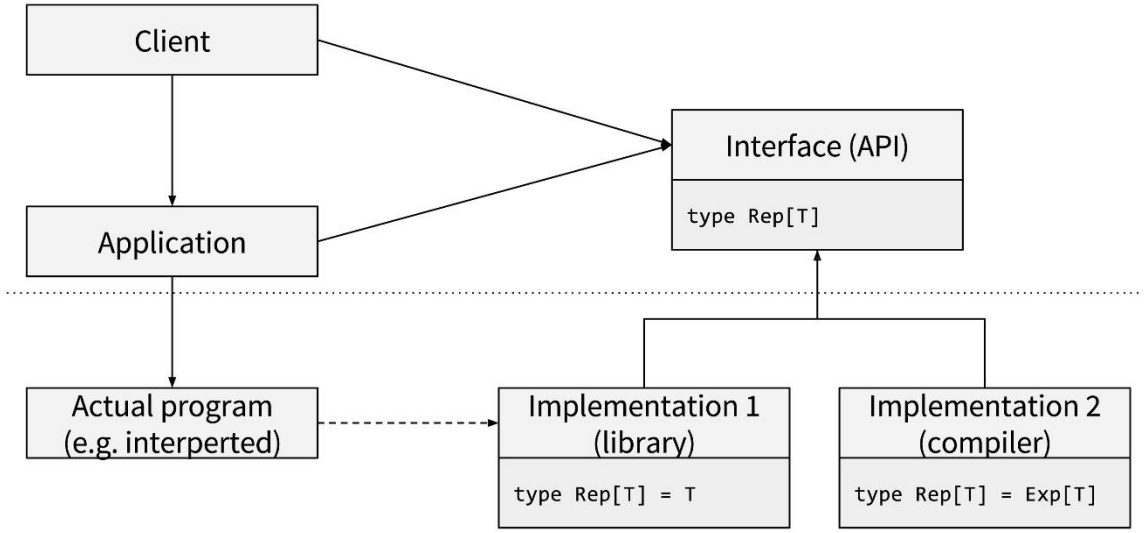


Figure 1: Polymorphic embedding of DSLs

Having an understanding of the front-end organization of LMS and Delite, we can start analyzing it in more detail.

The first notable property of the current system, disregarding the design, is the dependency on a specialized version of Scala compiler Scala-Virtualized, which among providing other features eases overloading / overriding of operations and enables lifting of control structures, otherwise inaccessible (un-overloadable) in the vanilla compiler. This presents a major disadvantage in the current system and one of the main goals of this thesis was the removal of this dependency. We would like to be able to reconstruct these abilities of the special version of the compiler through some other means. Using Scala macros [11] as a principled way of compile-time reflection [18] in Scala appeared to be a promising approach towards solving this problem. We are going to show that it is possible to recreate the majority of the features of Scala-Virtualized using Scala macros and to successfully apply the macro-based virtualization approach to real-world Delite domain-specific languages.

Secondly, the shared API provides benefits of being a clean interface to both shallow and deep embeddings, restricting the language and ensuring that the used DSL operations exist in both the shallow and the deep variant of the DSL. But this approach inevitably leads to leak of abstractions to the user code, namely `Rep` types can be visible as parameters and return types of methods in the user-facing DSL API. This calls for an investigation of new sort of embeddings which could potentially conceal these abstractions. In this thesis we are going to look at Yin-Yang translation layer, as a way moving between a shallow, direct version of code (without any Reps) and its deeply embedded counterpart automatically. Additionally, we are going to explore a different kind of deep embedding, based on custom types approach, which successfully hides away deep embedding abstractions from the application code.

Thirdly, the way that the domain-specific language trait and applications are composed currently leads to long compilation times. In this thesis, we are going to investigate more precisely what are the bottlenecks in the compilation pipeline and prove that the large portion of compilation time is spent in the mixin phase, creating large trait compositions (so called "cakes"). Additionally, necessary implicit conversions, which allow for automatic combination of non-staged and staged values through the use of operators on corresponding types, also affect the compilation times as the typer phase is unable to directly type check all expressions, rather often an implicit search needs to be performed in order to find an appropriate conversion. In order to mitigate this, we will develop a new shallow embedding, which does not require the composition of traits or the use of the Cake design pattern, leading to improved compilation times of shallowly embedded code. By utilizing one of the two approaches mentioned in the previous paragraph, we should also able to quickly move between shallow and deep variants of the applications, maintaining the speed of compilation and enabling quick iteration in program development.

The next section will present Scala-Virtualized in more detail, which will be followed by the exposition of our solution to replace it using macro-based virtualization.

3. Scala-Virtualized

Parts of this chapter have been based on Scala-Virtualized paper, by Moors et al. [6].

Scala-Virtualized is a fork of the mainline Scala compiler devised to enable better support for hosting of embedded domain-specific languages in Scala. As mentioned previously, LMS framework is able to provide the benefits of deeply embedded languages with very little overhead. Scala-Virtualized compiler allows for lifting of additional built-in language constructs and static information, allowing for a convenient support for deep embeddings.

The main feature which makes Scala-Virtualized attractive for use in LMS is its ability to blend shallow and deep embeddings of DSLs. This means that the users are able to write DSL code with very little syntactic overhead, tightly integrated with the surrounding Scala code, but still obtain a representation of their program in the form of an abstract syntax tree-like (AST) structure that is suitable for further analysis and optimizations. By doing this, Scala-Virtualized provides a good platform for code generation and program transformation techniques for embedded programs.

Scala-Virtualized builds on top of the Scala language in several areas:

- infix methods provide additional syntactic freedom without run-time overhead
- control-flow statements are expressed as method calls, which makes them easily overloadable / overridable
- implicit `SourceContext` parameters supply methods static source information at runtime

Scala syntax is already very flexible out-of-the-box:

- it imposes very little limitations on the way methods are named (symbolic method names are valid in Scala e.g. `!` or `<=>`)
- methods can be invoked using infix notation instead of the classic dot notation (`stringList contains myString` instead of `stringList.contains(myString)`)
- it allows for overriding of the behavior of for-comprehensions e.g.

  ```
  for (elem <- intList) yield elem+4
  ```

  gets de-sugared to

  ```
  intList.map(elem => elem + 4)
  ```

which can be easily redefined by redefining the `map` method on the type of `intList`. Other, more complex for expressions are also de-sugared to a combination of collection operations like `map`, `withFilter`, `flatMap`…

- implicit conversions can be used in order to externally add methods to existing types

3.1 Infix methods

Scala's syntax flexibility is still partially limited as implicit conversions do not enable overriding/overloading of existing methods. This is exactly where infix methods come into play, allowing for flexible overriding/overloading of methods outside of their class. In the regular Scala compiler, there are two usual ways in which the meaning of an expression such as `x.a` can be customized. If we control the type of `x` (we are writing the code for it, it is not in a precompiled library), we are able to simply add the needed method to the class of `x`. Otherwise, we are able to use implicit conversion if we are not trying to externally override an existing method, as shown in the next example:

```
class Complex(val real : Double, val imag : Double) {
  def +(that: Complex) : Complex =
    new Complex(this.real + that.real, this.imag + that.imag)
  override def toString = real + " + " + imag + "i"
}
object ComplexImplicits {
  implicit def Double2Complex(value : Double) = new Complex(value,0.0)
}

import ComplexImplicits._
object Program {
  def main(args : Array[String]) : Unit = {
    val a : Complex = new Complex(4.0, 5.0)
    val b = 3 + a
    println(b)  // 7.0 + 5.0i
  }
}
```

Code Example 4: Implicit conversion example

While functioning, this technique requires some boilerplate code, and imposes a certain run-time overhead used to perform the implicit search (when using the conversion i.e. object's type does not provide an appropriately typed method / member). Even more importantly, this technique cannot be used to override existing methods on types, such as `toString` or operations on primitive types.

Infix methods, provide a much more flexible approach. On one hand, they allow for a selective, external introduction of new members in types, like adding methods to existing types. On the other hand, they allow for overriding of the existing methods. All of this is performed without any runtime overhead. The idea behind this technique is simple: Scala-Virtualized compiler rewrites the expression `x.a` to `infix_a(x)`, if there is a method `infix_a` in scope, such that the expression `infix_a(x)` type-checks. As shown in the MySimpleVector example, this is one of features that makes it very easy to define clean and unified interfaces to DSLs.

3.2 Control flow statements overriding (everything is a method call)

The essence of having an embedded language is that user-defined constructs should be treated in the same manner as the built-in ones i.e. should be treated as first-class citizens. User-defined abstractions should have the same rights and privileges as built-in language abstractions. To achieve this, Scala-Virtualized represents many of the built-in abstractions as method calls. In this way, method definitions corresponding to the built-in abstractions can easily be overridden/overloaded just like any other method, for the purpose of the domain-specific language. Scala-Virtualized takes a similar approach to representing object programs as the "finally-tagless" [8] or polymorphic embedding approach [9], using method calls rather than explicit data constructors [6]. Thus, DSL authors can override / overload the default implementations appropriately in the goal of generating an explicit program representation which can be further analyzed and optimized. This characteristic was typically reserved for deeply embedded languages, using explicit data constructors.

The `EmbeddedControls` trait provides method definitions that represent control structures available in Scala and are treated by the compiler in a special way.

For example, when using a while loop:

```
while (needWork) {
  doWork()
}
```

the compiler will detect it and the parser will rewrite it to a method call:

```
__whileDo(needWork, doWork)
```

This method call will be bound to an implementation based on regular rules of scoping. If there is no overridden / overloaded definition and the method binds to the default definition in the `EmbeddedControls` trait, the type checker will replace the method call with a "While" tree node, which provides the default while loop behavior.

The following language constructs have been overridden in the `EmbeddedControls` trait:

```
// while (cond) {body}
def __whileDo(cond: Boolean, body: Unit): Unit

// do { body } while (cond)
def __doWhile(body: Unit, cond: Boolean): Unit

// if (cond) { thenp } else { elsep }
def __ifThenElse[T](cond: => Boolean, thenp: => T, elsep: => T): T

// var x = init
def __newVar[T](init: T): T

// lhs = rhs
def __assign[T](lhs: T, rhs: T): Unit

// return expr
def __return(expr: Any): Nothing

// expr1 == expr2
def __equal(expr1: Any, expr2: Any): Boolean
```

Code Example 5: Virtualized control structures and language constructs

As shown before, for-comprehensions already correspond to method calls in Scala, so this is a logical extension of that approach. Additionally, this approach provides more flexibility to the DSL author in comparison to lifting host language constructs using a fixed set of data types, as the author can easily pick which constructs to lift.

3.3 Providing source information in DSLs

When a DSL user writes a program and the program is treated as a deeply embedded one i.e. it gets lifted into a representation for further optimization, information about the source files and positions is lost. This means that debugging becomes inherently hard, as the IR does not contain the source information and there is no way for the DSL to inform the user about what exactly went wrong and at which exact location in the source file.

In order to mitigate this this problem, Scala-Virtualized provides an implicit `SourceContext` parameter that gets synthesized automatically by the compiler. This parameter can then be propagated through the IR and used by the DSL to point to the invocation site where the issue occurred and provide more specific error messages.

Here is an example from one of the Delite DSLs - OptiQL, a DSL dealing with data querying and transformation:

```
def infix_Distinct[A:Manifest](self: Rep[Table[A]])(implicit __pos:
SourceContext)
```

Whenever the `Distinct` method is called on an instance of `Rep[Table[_]]`, the `__pos` object will contain the information about the invocation site, meaning that the corresponding IR node is able to take that information and use it if needed.

While not a complete solution (no type debugger, no effect system, etc.), Scala-Virtualized introduces important new features that enable easier developing of a deeply embedded domain-specific language in Scala. Still, as it is a complete fork of the compiler, this separation introduces maintainability and distribution issues. Alternatively, as stated before, compile-time reflection presents a promising approach to resolving the same problems that Scala-Virtualized is addressing.

In the next chapter we will introduce metaprogramming as a general term. In the chapter following that one we will present Scala Macros, the current compile-time metaprogramming system in Scala. Later on we will demonstrate how we used it in order to replicate the behavior of Scala-Virtualized and remove LMS's dependency on it.

4. Metaprogramming and Scala

Metaprogramming as a general term is the process of writing computer programs that are able to treat other programs or code snippets as their data [19]. This means that the program could read, analyze, generate or even transform other programs. Another, more radical use-case would be if a program modified itself. Some of the use-cases of metaprogramming include reducing the number of lines of code that programmers need to write (removing boilerplate by automatically generating repetitive or generic code), or optimizing pieces of code (manipulating and transforming corresponding abstract syntax trees). Metaprogram is usually written in a language that can be referred to as a metalanguage, while the program that it is manipulating is written in an object language. When a programming language combines both of these characteristics i.e. it able to be a metalanguage for itself, this is called reflection. It is said that the programming language capable of this possesses reflective capabilities.

Two types of reflection can be distinguished:

- run-time reflection - the ability of a program to inspect itself at run-time
- compile-time reflection - the ability of a program to generate code and/or manipulate ASTs at compile time. This capability presents a powerful way to develop program generators and transformers

As Scala is running on the Java Virtual Machine (JVM), a part of the Java reflection API is available for use in Scala, namely the part providing the ability to dynamically inspect classes / objects and access their members. But, as Scala is a much more complex and essentially a different language than Java, many of the features available in the Scala language could not be accessed through the existing Java reflection API. Additionally, Java's take on generics is such that it uses a technique called type-erasure [20]. This prevents Java reflection to recover runtime type information of Java types that are generic at compile-time, a characteristic that is carried through to generic types in Scala. Manifests (introduced in Scala 2.7) are a way through which Scala provides generic type information at run-time. This is an especially important feature for domain-specific languages, as knowing those generic types is essential for allowing various optimizations of the domain-specific code.

Starting from Scala 2.10, a full reflection library has been introduced to Scala [18], mitigating the lack of features of Java's runtime reflection on Scala-specific types and features, but also adding a more powerful toolkit of general reflective capabilities.

As a simple example, during the development of this project, one of the issues encountered was that the output of a certain test which executed correctly was not formatted orderly. Namely fields of a table were not in the declaration order. Analyzing the legacy code, we have noted that it uses Java reflection API in order to extract the names of the fields from a class, although Java reflection does not preserve field declaration ordering. Using Scala reflection API, on the other hand, we were able to retrieve the ordering of the fields by simply invoking `sorted` method on the list of fields, which by definition sorted them in declaration order.

In addition to runtime reflection for Scala types and generics, Scala 2.10 added compile-time reflection capabilities in the form of Scala Macros, as well as the ability to reify (i.e. make explicit, bring into existence) ordinary Scala expressions into Scala abstract syntax trees.

The next chapter will present general ideas behind Scala Macros and describe various macro flavors that have been used in this project, in order to replace Scala-Virtualized compiler with macro-based language virtualization.

5. Scala-Macros

Parts of this chapter have been based on the paper “Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming”, by Eugene Burmako [11].

Scala Macros are Scala’s first take on compile-time reflection (a research project scala.meta [21] is in the making, building on the lessons learned from developing Scala Macros, attempting to make metaprogramming in Scala easier). Macros make it possible for programs to modify themselves at compile time. Scala Macros introduce various macro kinds i.e. macro flavors which provide different capabilities for interacting with Scala code and the Scala compiler. While there are many macro flavors, we will present only the ones essential for understanding this project. These are:

- def macros
- annotation macros
- implicit macros

5.1 Def macros

Def macros provide the most basic form of compile-time metaprogramming. On one hand, they look like ordinary function definitions, except for their body which starts with the keyword macro and an identifier which refers to the actual implementation method of the macro. On the other hand, their invocations behave differently, as they are expanded during compilation resulting in the macro-generated code. This is the essential difference between macros and regular methods - macros are resolved at compile-time.

5.2 Implicit macros

Implicit macros are a more advanced flavor of macros. They enable materialization of type classes and implicit parameters, which provide a powerful mechanism for reducing boilerplate code. They behave in a way such that when no appropriate implicit parameter / type-class instance can be found in the implicit search scope, the compiler invokes the implicit macro in scope which is able to create the required parameter. Instead of having to write multiple instance definitions of a type class, a programmer can define only one implicit macro, which based on its type parameter can synthesize the required implicit value. A use-case for this would be Manifests, which provide information about generic types at run-time. This mechanism is hard-coded inside the Scala

compiler, but by using an implicit macro, the feature could be removed from the actual compiler, simplifying the language and allowing it to be included on per-need basis. Another use case, more relevant to our efforts, would be synthesizing implicit `SourceContext` parameters.

5.3 Annotation macros

Note: Macro annotations are not available in the vanilla Scala distribution. Macro Paradise [22] compiler plugin needs to be used in order for macro annotations to work.

Annotation macros bring the power of compile-time metaprogramming to the level of definitions. They are able to transform arbitrary definitions (classes, functions etc.), potentially even creating multiple ones. It is important to note that annotation macros are whitebox macros. Whitebox macros in Scala are macros that cannot have a precise type signature, rather their signatures are only approximations (blackbox macros, on the other hand, fully conform to their type signature), which can later be refined. In annotation macros this approximation property is reflected through the parameter list and the return type, which are both `scala.Any`, the all-encompassing super-type in Scala.

Example of an annotation macro signature:

```
import scala.reflect.macros.Context
import scala.language.experimental.macros
import scala.annotation.StaticAnnotation
import scala.annotation.compileTimeOnly

@compileTimeOnly("Enable macro paradise to expand macro annotations")
class MyAnnotation extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro ???
}
```

Code Example 6: Annotation macro signature

The name of the annotation can be changed arbitrarily, while the rest of the definition is mandatory. A very useful optional feature is the `@compileTimeOnly` annotation. Macro annotations look like regular annotations to the Scala compiler, so if the user forgets to enable the macro paradise plugin, the compiler will simply ignore the annotations, without producing a warning about them. The `@compileTimeOnly` annotation gives a hint to the compiler that a corresponding macro annotation should be expanded away i.e. should not be referred to after type-checking (which includes macro expansion). If the macro is not expanded, the compiler will interrupt the compilation and provide user with the error message specified as a parameter of `@compileTimeOnly` annotation.

The next chapter will explore could we use these various macro flavors towards reproducing the behavior of Scala-Virtualized when using the regular Scala compiler.

6. Macro virtualization

In order to be able to remove Scala-Virtualized as a dependency of LMS, its features needed to be reproduced through other means. Macros presented a powerful alternative, as they allowed for compile-time metaprogramming through which one could modify or even generate program code by the means of manipulation of abstract syntax trees. In some ways, they are more principled than Scala-Virtualized, as they are included in the standard distribution Scala reflection library and they do not require additional changes to the Scala compiler or a complete compiler fork. Macros can even be thought of as small compiler plugins that developers are able to write to process their programs in various manners.

There has been a large discussion about the use of Scala-Virtualized versus Scala Macros, their overlapping features, complementary behavior etc. [23]. It is out of scope of this thesis to go into all the details of these differences, but if feasible, implementing virtualization through a macro-based approach can only be beneficial in the current state of Scala compiler development and is a generally promising solution for several reasons mentioned above. While some of the features from Scala-Virtualized might end up in the main compiler branch, there are no guarantees for that. Additionally, to the knowledge of the author, Scala-Virtualized has not seen widespread use other than in several research projects in academia. Scala Macros, on the other hand, are already integrated with the compiler, and they have seen larger use in the community and the industry [24].

These high-level features identified for now that needed to be reproduced from Scala-Virtualized:

- control flow statements virtualization
- providing implicit source information at runtime
- infix method behavior

6.1 Control flow statements virtualization

The first feature that needed to be reproduced was the control flow statements virtualization i.e. representing control flow statements as method calls. As macros do not have the same global view that the compiler has, each part of the code that needed to be processed or transformed required being marked in some way. The natural solution to this was using annotation macros. A macro annotation can take its annottee (any kind of definition, including methods, classes, objects etc.) and process it in a defined manner. Thus the `@virtualized` macro was born.

6.2 @virtualized macro annotation

`@virtualized` macro is a macro annotation that performs the virtualization of the annotated code. When a definition is annotated, this macro will initialize the virtualization transformer which will go through the abstract syntax tree of the definition and perform necessary transformations. The transformer takes in the tree of a definition and pattern matches it against various trees which correspond to control structures. If no match is detected, the transformer utilizes the default transformation strategy which is just breadth-first component-wise cloning.

Then it recursively proceeds to process all of the internal parts of the definition, potentially polymorphically applying the user-defined transformation. If a match is detected with any of the control structures, the transformer will replace the AST node of that control structure by the one of a method call corresponding to the virtualized control structure.

Example:

```
@virtualized
if (condition) doThen() else doElse()
```

becomes

```
__ifThenElse(condition, doThen, doElse)
```

This method call is implemented by default as a def macro. The only thing that the def macro will do is simply expand the method call (as it is actually a macro) to the original abstract syntax tree of the control structure. This gets us back to the original control structure and provides the user with the expected behavior, in case there was no redefinition of the control structure.

If, on the other hand the user overrides or overloads this method and the compiler is able to successfully resolve it, the user-defined method will be called, allowing for arbitrary behavior and accomplishing virtualization.

6.3 Providing implicit source information at runtime

Among other unique features of Scala-Virtualized is that it is able to automatically provide source information at runtime to methods that require it. This feature was hardcoded in the compiler and it functioned in a similar manner to how Manifests function. Analogously to how the compiler provides the erased type to a method at runtime through an implicit `Manifest` parameter, the modified compiler additionally provides static source information through `SourceLocation` / `SourceContext` parameters. A small difference lies in the fact that these parameters only deal with source information, meaning that they are independent of type thus they do not need to be generic, as Manifests are.

In order to un-hardcode this behavior or completely remove the dependency on Scala-Virtualized, one would need to envision a mechanism which would generate i.e. materialize `SourceContext` parameters when they are requested by a method. Fortunately, there exists a macro flavor that caters exactly to this need, namely the implicit macro flavor. As mentioned before, this macro is able to materialize type class instances encoded with implicits or implicit parameters as `SourceContext` is. This macro is defined in the `object SourceContext` as:

```
implicit def _sc: SourceContext = macro SourceContextMacro.impl
```

The `SourceContext` macro takes no parameters (i.e. no type parameters as Manifest), but simply extracts the position from the information available in the surrounding compiler `Context` available to the macro. The information extracted and encoded in the `SourceContext` is:

- path to the file in which the method is invoked
- filename
- line and column offset

By accessing the created `SourceContext` parameter, the DSL author is able to provide users with much better error messages and debugging information.

6.4 Infix method behavior

Infix methods provided a powerful mechanism to externally and selectively introduce new methods or override/overload existing ones in arbitrary types. As they are not available outside of the Scala-Virtualized compiler, a different solution needed to be found.

For the purpose of LMS and domain-specific languages based on it, it was enough to look at the way to introduce new methods rather than override existing ones. This is due to the fact that staged DSL types are represented as types wrapped inside a `Rep[T]` generic type. As explained before, in the actual implementation, this type could be interpreted in two ways:

- For the library (shallowly embedded) version, `Rep[T]` is just an identity alias for type `T`
- For the compiler (deeply embedded) version, `Rep[T]` is defined as an `Exp[T]` - expression

As stated previously, a classic way to introduce new members to a type that we have no control of is to use an implicit conversion or an implicit class containing new members. This principled approach was also used here, and it worked in most of the cases, but as we will see later, there are various problems with implicit conversions, some of which are unresolved, that create a need for looking into alternative solutions (a translation layer, new embeddings etc.).

Let us take a look on how this technique works in practice. For example, say that we want to have a DSL type which represents a date - `Date`, and we would like to be able to compare values of this type with less-than `<` and greater-than `>` operators.

The old version of the `DateOps` trait, using Scala-Virtualized, could look like this:

```
// shared Date API
trait DateOps extends Base {

  def infix_<(self: Rep[Date],__arg1: Rep[Date])(implicit
    __pos: SourceContext) = date_lt(self,__arg1)(__pos)
  def infix_>(self: Rep[Date],__arg1: Rep[Date])(implicit
    __pos: SourceContext) = date_gt(self,__arg1)(__pos)

  def date_lt(self: Rep[Date],__arg1: Rep[Date])(implicit
    __pos: SourceContext): Rep[Boolean]
  def date_gt(self: Rep[Date],__arg1: Rep[Date])(implicit
    __pos: SourceContext): Rep[Boolean]

}
```

Code Example 7: Example of an old interface using infix methods

While `Date` or `Rep[T]` do not possess `<` or `>` methods, the compiler is able to type-check the expressions that compare `Rep[Date]` instances due to the infix_ feature that Scala-Virtualized provides.

Now, removing infix_ methods, adding the implicit conversion, the `DateOps` trait looks like this:

```
// shared Date API
trait DateOps extends Base {

  implicit def repToDateOpsCls(x: Rep[Date])(implicit __pos:SourceContext) =
    new DateOpsCls(x)(__pos)

  class DateOpsCls(val self: Rep[Date])(implicit __pos: SourceContext) {
    def <(__arg1: Rep[Date])(implicit __pos: SourceContext) =
      date_lt(self,__arg1)(__pos)
    def >(__arg1: Rep[Date])(implicit __pos: SourceContext) =
      date_gt(self,__arg1)(__pos)
  }

  def date_lt(self: Rep[Date],__arg1: Rep[Date])(implicit
    __pos: SourceContext): Rep[Boolean]
  def date_gt(self: Rep[Date],__arg1: Rep[Date])(implicit
    __pos: SourceContext): Rep[Boolean]

}
```

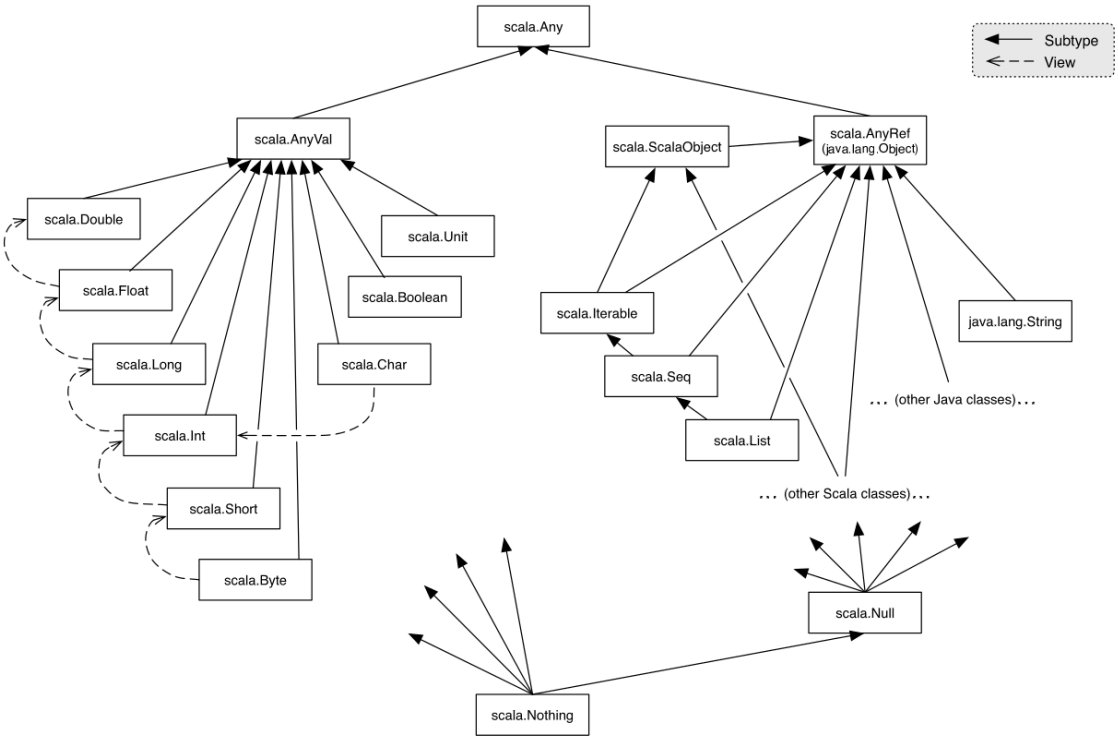
Code Example 8: Example of a new interface using implicit conversions

Due to the implicit conversion, whenever these methods are called on a value of the DSL type `Rep[Date]`, the expression will type-check and the appropriate methods will get executed. As explained before, methods in the API are left abstract, as this API is shared between both shallow

and deep embedding of the DSL, so each embedding is required to implement the needed behavior and mix it in. In the standard case, the shallow version will simply directly execute the operation, while the deep one will create the corresponding IR node for further processing by the framework and code generation.

6.5 Problems with implicit conversions

Looking from a high level, the implementation using implicit conversion method seems sufficient. But taking a deeper dive, after quick experimentation, it becomes obvious that not everything is working as expected. As mentioned before, one of the drawbacks of the implicit conversion technique is that it cannot override existing members of a type. As `Rep` is simply an identity transformation in the shallow embedding and is represented by `Exp` class in the deep embedding, one could ask oneself which members need to be overridden? In order to answer that question fully, we need to take a brief look at the Scala type system.



Code Example 9: Overview of the type system of the Scala programming language

As we know from the Java world, any user-defined type inherits at least from the all-encompassing super-type `Object` (`java.lang.Object` on the picture). In the Scala world, this type is represented by `scala.AnyRef` type. This means that even if `Rep` or `Exp` contain no methods on their own, they automatically inherit all of the methods defined in `AnyRef` and its super-type `scala.Any`. The concerning methods can be found below:

```
// Methods inherited from scala.Any
 final def ==(arg0: Any): Boolean
 final def !=(arg0: Any): Boolean
 final def ##(): Int
 def equals(arg0: Any): Boolean
 def hashCode(): Int
 final def asInstanceOf[T0]: T0
 final def isInstanceOf[T0]: Booleanh
 def toString(): String
 abstract def getClass(): Class[_]

 // Methods inherited from scala.AnyRef
 final def eq(arg0: AnyRef): Boolean
 final def ne(arg0: AnyRef): Boolean
 final def notify(): Unit
 final def notifyAll(): Unit
 final def synchronized[T0](arg0: => T0): T0
 final def wait(): Unit
 final def wait(arg0: Long, arg1: Int): Unit
 final def wait(arg0: Long): Unit
 def clone(): AnyRef
 def finalize(): Unit
```

Code Example 10: Methods defined in scala.Any and scala.AnyRef

Note that most of them are defined as final, meaning that they cannot be overridden / overloaded. Now if a user would invoke any of these methods on a value of type `Rep[T]`, the methods which would get called would be the one corresponding to the generic type `Rep` (`Exp` in the deep / compiler case), rather than the actual wrapped type inside the `Rep`. No implicit conversion could help in this case as existing inherited methods would always be preferred over implicit search.

In order to mitigate this, the `@virtualized` annotation was expanded to cover the cases when these methods / operators were encountered. In a fashion similar to the one employed in the case of control structure statements, calls to methods of `Any` and `AnyRef` would get rewritten to calls to methods with a special name. In this case, prefix infix_ was chosen, in order to resemble the style of Scala-Virtualized, but it is worth noting that these infix_ methods are not treated specially in the regular Scala compiler - they are just ordinary def macros which if overridden provide the required behavior. Again, if no override / overload is defined, the default method implementations are expanded as macros, which simply produce the trees of the original expressions, exhibiting the default behavior i.e. invoking the default method on `Rep` type.

This approach was feasible as there is only a limited set of methods that needed to be supported. Also, all of the DSL-defined types contained them, which means that there was no need to inspect the type of the tree in the annotation macro, just to detect the operation and produce a method call which can be overridden / overloaded. It is worth noting that type inspection is unavailable during the expansion of annotation macros, as they are run (get expanded) before type-checking (before the typer runs).

6.6 Problems with string concatenation

Users can write expressions like:

```
"Prefix of the value " + myValue
```

This expression will simply perform a concatenation between the `String` literal and the `String` representation of `myValue` obtained through a call to `toString` method. But that might not be the behavior we want e.g. if we would like to lift the expression. As the concatenation operator `+` is defined on the `String` type and it can have a value of `scala.Any` type as a right-hand side operand, there is no way to override this behavior through the use of implicit conversions. Rather, in this simple case, we can again turn to `@virtualized` macro annotation. Although we are unable to detect the type of the prefix `String`, as it is a `String` literal, it is recognized by the parser even before type-checking and we are able to exploit this knowledge in order to overload the behavior of the concatenation operator.

Unfortunately, this approach is limited, as it does not process chained concatenations uniformly. Namely when a user writes an expression like this:

```
"Prefix of the value " + myUnstagedValue + myStagedValue
```

The macro will rewrite it to:

```
infix_+("Prefix of the value ", myUnstagedValue) + myStagedValue
```

As the first method returns a `String` by default, the problematic behavior will be exhibited again when concatenating the staged value. While there is currently no simple solution to resolve this issue systematically, several approaches might help alleviating this problem:

1. require users to explicitly lift string literals
2. require using of another String concatenation operator, which is not already defined on the String type e.g. this operator could be the caret character `'^'`
3. use String interpolation [25] instead of concatenation

Additional features provided by the Scala-Virtualized compiler have been identified that required analysis and reproduction through macros, namely Scopes and Structs.

6.7 Scopes

Scopes were a feature of Scala-Virtualized that made writing DSL apps more concise, removing some of the required boilerplate code. Given a method like:

```
def OptiML[R](b: => R) = new Scope[OptiML, OptiMLExp, R](b)
```

The expression

```
OptiML { body }
```

got expanded to:

```
trait DSLprog$ extends OptiML {
  def apply = body
}
(new DSLprog$ with OptiMLExp): OptiML with OptiMLExp
```

This feature was replicated in the macro virtualized world through the use of `@virtualized` macro. Still, as feature this is only syntactic sugar and does not add any fundamental functionality, all of the DSLs are operational even without the Scopes.

6.8 Structs / Records

An additional feature that Scala-Virtualized provided are the `Struct` types, simple product aggregate types [26].

Structs are one of the core building blocks of domain-specific languages. The first reason they were implemented in Scala-Virtualized is that there was a need to have the ability to reify the creation of user-defined product-like datatypes. Scala-Virtualized originally virtualized the creation of Structs by rewriting the `new` operator in a type-directed manner, allowing it to be easily redefined. Namely virtualization was performed if there existed a type constructor `Rep`, so that the created value was a subtype of `Struct[Rep]`. Additionally, member accesses (selections) like `s.a`, where `s` is a `Struct` and `a` is a member of type `T` were being rewritten to a call of `s.selectDynamic[T](“a”),` which could be also redefined to support reification.

Delite used an abstract class `Record` extending the marker trait `Struct` to denote product types. In the shallow embedding Delite implemented `Struct` as a simple `Map`, whereas in the deep embedding the existing LMS nodes from the `StructOpsExp` trait captured the semantic of `Struct` (`Record`) creation and field accesses. Structs allow for various optimizations, bringing important performance benefits to DSLs. For example, by restricting the set of available types for values in a `Struct` e.g. to numeric types (`Int`, `Float`, etc.), `Boolean`, `String`, `Array`, and `Map`, Delite is able to perform optimizations such as Array-of-Struct (AoS) to Struct-of-Array (SoA) and dead field elimination automatically, and also reliably generate code for various targets in the back-end.

Outside of the Scala-Virtualized compiler, it was required to find a way to reproduce this behavior. The core of the problematic lied in the question how to virtualize the creation of Records? Namely how can we intercept the `new` operator when a user writes code like:

```
val z = new Record { val re = 1.0; val im = -1.0 }; print(z.re)
```

The solution developed for macro-virtualization-based LMS used the mechanism behind the type `Dynamic` [27] available in Scala. In more details, type `Dynamic` was originally designed to support dynamic member selection. The mechanism behind it though, could be exploited in a way to make the creation of `Record` objects re-definable, enabling its reification. As explained in [27], and simplified here, when a user invokes `qual.sel`, if the type-checking fails and `qual` conforms to `scala.Dynamic` type, the invocation gets rewritten to a call one of the special methods `applyDynamic`, `applyDynamicNamed`, `selectDynamic`, or `updateDynamic`, depending on the rest of the expression (arguments, name-argument pairs, member selection, member assignment). The method used here is `applyDynamicNamed`, as each of the parameters of a `Record` is a named field. Making the `object Record` extend the `Dynamic` trait enabled `Record` to have the aforementioned behavior. Now if a user would write a statement like

```
val z = Record(re = 1.0, im = -1.0) // invocation of the "apply" method
```

the compiler would rewrite it to

```
val z = Record.applyDynamicNamed("apply")(("re",1.0),("im",-1.0))
```

Method `applyDynamicNamed` could be defined arbitrarily in the `object Record`. This method was implemented as a def macro which ultimately forwarded the call to an overridable `record_new` method, and at the same time creating an instance of the corresponding `RefinedManifest`, a subtype of `Manifest` created for the purpose of Structs, able to preserve schema type information. Member selections were enabled through an implicit macro which expanded to a `RecordAccessor` implementation (details elided for simplicity sake), where fields were defined as methods forwarding their calls to an overridable `record_select` method. This solution allowed for arbitrary Record behavior redefinition in the shallow and the deep embedding through appropriate implementations of `record_new` and `record_select` methods.

Additionally, named Records were implemented as following in Scala-Virtualized:

```
type LineItem = Record {
  val l_partkey: Int
  val l_quantity: Double
  val l_comment: String
}
def LineItem(partKey: Rep[Int], quantity: Rep[Double], comment: Rep[String])
  : Rep[LineItem] = new Record {
  val l_partkey = partKey
  val l_quantity = quantity
  val l_comment = comment
}
```

A user could create a named `Record` by invoking a method with the appropriate name (in this case `def LineItem`), which would simply create a `Record` with the specified parameters. All of the previously presented properties still apply to this `Record` instance. Nevertheless, we can observe that the return type of this method was the wrapped `LineItem` type `Rep[LineItem]`. Although the method returns an instance of a `Record`, this was possible because of the structural refinement employed [28], in more details, the `LineItem` type was defined as a structural refinement of the `Record` type with the same structure as the `Record` returned by the method `def LineItem`, so the compiler could relate the two types.

Dealing with named Records in the macro-virtualized world, we wanted to simplify the implementation and reduce the amount of boilerplate needed to use them, when compared to Scala-Virtualized. In order to do this, we developed a new macro annotation `@Record` which specified that a definition of an annotated case class represents a `Record`. Now we could simply write:

```
@Record
case class LineItem(partKey: Int, quantity: Double, comment: String)
```

which would get rewritten to a representation similar to the one above, the only difference being using the `Record` def macro instead of the `new Record` syntax.

Later on in the project, Records were also used in the new `Rep`-less shallow embedding, which will be presented in the later sections. This presented a challenge, as `Record` macros were modelled after Scala-Virtualized Records which expected lifted `Rep` values as parameters. Experimentations with the scala-records project [29] gave successful results, but the project came with specialization optimizations which would make adapting the project to our exact needs more difficult, and were not fundamental for our use-case. It was ultimately decided to expand on the macros approach and develop a `Record` macro which when used would simply expand to an instantiation of an anonymous class extending `Record`, enabling the compiler to type-check `Record` accesses. Named `Record` presented a bigger problem, as we had no means of invoking `record_new` simply with a wanted type and parameters anymore. Rather, the RefinedManifest generated by the `Record` macro was expanded to include a `def create` method which enabled the instantiation of a named `Record` in a method based only on a type parameter (the user is able to call the `create` method on an implicit `RefinedManifest` parameter).

The various implementations shown present a great example of the strength of Scala macros. Through an intricate interaction with type `Dynamic`, not only was the `Struct` behavior from Scala-Virtualized successfully reproduced, but also improved through a reduction of code needed for using named Records.

7. Macro virtualization applied

7.1 Applied to LMS and LMS tutorials

When all of the changes resulting from the new macro-virtualized implementation were integrated successfully to LMS, we started applying the new, Scala-Virtualized-free version of LMS to the accompanying LMS tutorials available at [30] in order to confirm the validity of the approach and the implementation.

LMS distribution contains a set of tutorials which introduce new users of the framework, to the multi-staged programming paradigm and the way LMS functions. Starting off at a beginner-level examples and they build up all the way to the advanced features of the framework. They serve a double purpose - at the same time as being tutorials, they are used as tests too, checking the generated code against the expected results and reporting errors in case of a mismatch.

Through a systematic work of applying the new virtualization technique, all of the tutorials, but a few, were successfully upgraded the macro-virtualized version of LMS. A few tutorials kept exhibiting unexpected behavior, even with rigorous checking of the implementation and the produced output.

BooleanOps conversion problem

In the regular expression tutorial, LMS demonstrates the idea that a staged interpreter can actually be seen as a compiler. As a case study, a simple regular expression matcher is examined. In the non-staged version, the non-staged regex matcher is invoked on a regex string and an input string, both of type `String`. Conversely, the staged regex matcher is invoked on a static regex string of type `String` and a dynamic input string of type `Rep[String]`. This staged interpreter generates code that is specialized to match input strings against static regex patterns. This tutorial implements a small, famous, recursive regular expression matcher originally implemented in C by Rob Pike and Brian Kernighan [31].

A function of a particular interest towards presenting the encountered issue is a function at the bottom of the recursive call chain - `matchchar`, a simple function that matches the input character either to the exact same character in the regular expression or the universally matching dot character `'.'`.

In the non-staged version of the tutorial, the function looks like this:

```
// c - current regex char, t - input string char
def matchchar(c: Char, t: Char): Boolean = {
  c == '.' || c == t
}
```

In the staged version, the function looks like this:

```
// c - static current regex char, t - dynamic input string char
def matchchar(c: Char, t: Rep[Char]): Rep[Boolean] = {
  c == '.' || c == t
}
```

After virtualization is applied to the staged version, the function becomes:

```
// c - static current regex char, t - dynamic input string char
def matchchar(c: Char, t: Rep[Char]): Rep[Boolean] = {
  infix_==(c, '.') || infix_==(c, t)
}
```

Based on the type signature, the first `infix_==` is supposed to resolve to the default `infix_==`:

```
def infix_==(x1: Any, x2: Any): Boolean = macro any_==
```

which will simply expand to the default implementation of the `==` method from class `scala.Any`.

The second `infix_==` is supposed to resolve to the `infix_==` from the mixed in LMS `Equal` trait (signature reduced for clarity sake):

```
def infix_==[A,B](a: A, b: Rep[B]): Rep[Boolean] = equals(unit(a), b)
```

as that is the most specific type signature of the overloaded method, given the parameters.

Unfortunately, the second `infix_==` method call was still resolving to the default implementation with `Any` parameters. After a thorough inspection and several experiments, we have come to a conclusion why this is happening. After the `@virtualized` annotation macro is expanded, the typer runs in order to type-check the expression and employ any implicit conversions if needed.

The left hand side of the `||` expression undoubtedly resolves to `Boolean` type.

The right hand side of the `||` expression should resolve to `Rep[Boolean]` type.

Then, the implicit conversion can be employed by the compiler and apply the `||` operator on operands of `Boolean` and `Rep[Boolean]` types.

But what compiler actually does is resolve both sides of the `||` expression to the Boolean type.

How?

By a simple up-cast from `c: Char` and `t: Rep[Char]` to `Any` in the case of the right-hand side operand of `||` operator `(c == t)`, the compiler is able to employ the default, albeit less specific `infix_==` method, taking in two parameters of `Any` type. By doing this, the `||` operator now has both sides of the expression resolving to `Boolean` type. Thus, there is no need to perform the implicit search, as `||` method which takes value of `Boolean` type as a right-hand side operator is defined by default on the `Boolean` type.

A takeaway from this problem is that the compiler will always prefer up-casting and using the less specific version of a method over performing an implicit search in a complex expression. This unexpected behavior was investigated for a considerable amount of time and although it might be limited in scope, it is worthwhile being aware of it in the move towards the fully macro-powered version of LMS.

In order to circumvent this behavior, the user could simply split the expression into several smaller ones and type annotate them in order to make sure that the typer has enough information to make an unambiguous and clear decision which overload to use.

So now, the rewritten `matchchar` function could look like this:

```
// c - static current regex char, t - dynamic input string char
def matchchar(c: Char, t: Rep[Char]): Rep[Boolean] = {
  val b1: Boolean = c == '.' // Any, Any version invoked
  val b2: Rep[Boolean] = c == t // A, Rep[B] version invoked
  val res: Rep[Boolean] = b1 || b2 // implicit conversion should kick in
  res
}
```

Code Example 11: Modified match function, with correct == method behavior

If we try to compile the tutorial now, the compiler gives this error output:

```
[error]/home/boris/w/ppl/lms/ours/tutorials/src/test/scala/lms/tutorial/r
egex.scala: 176: type mismatch;
[error]  found   : StagedRegexpMatcher.this.Rep[Boolean]
[error]  required: Boolean
[error]     val res: Rep[Boolean] = b1 || b2
[error]                                  ^
[error] one error found
[error] (test:compileIncremental) Compilation failed
```

This meant that the compiler was unable to resolve the implicit conversion which was supposed to be found and employed for the expression to type-check. After much experimentation, I was able to make a minimal example which was supposed to investigate this problem.

```
import scala.language.implicitConversions

trait Base {
  class Rep[+T](val v: T) // wrapper / internal representation
}

trait BooleanOps extends Base {
  // implicit conversion
  implicit def lift2BooleanOpsCls(x: Boolean): BooleanOpsCls =
    new BooleanOpsCls(new Rep[Boolean](x))
  class BooleanOpsCls(rx: Rep[Boolean]) {
    def ||(ry: =>Rep[Boolean]): Rep[Boolean] = new Rep[Boolean](rx.v || ry.v)
  }
}

trait MyExample extends BooleanOps {
  // test method
  def foo(): Rep[Boolean] = {
    val ret: Rep[Boolean] = false || new Rep[Boolean](true)
    ret
  }
}
```

Code Example 12: Boolean implicit conversion issue

This is a self-contained example that can be tested by simply copying the code and pasting it into the Scala REPL. By performing that, we get the following output from the interpreter:

```
<console>:13: error: type mismatch;
 found   : MyExample.this.Rep[Boolean]
 required: Boolean
           val ret: Rep[Boolean] = false || new Rep[Boolean](true)
                                   ^
```

This meant that the compiler could still not employ the implicit conversion. This minimal example provided a good starting point for further investigation into the problem. Various experiments have shown that the issue remained, but if either of these two modifications was employed:

1) putting the `Rep` class in the global scope, outside of the `Base` trait
2) putting the body of `BooleanOps` inside of the `MyExample` trait

the code snippet compiled and worked as expected. Another solution that was found was to make the parameter `ry` of the `||` method (and `&&` in a more general case) have the call-by-value evaluation semantic. While in this way the example compiled, the regex tutorial test did not give satisfactory results. This was due to the fact that the right-hand side operator of `Boolean` operators was eagerly evaluated, instead of being evaluated only if used. This meant that all of the values computed in it would be generated before and put in the scope above of the computation of the actual expression (a parameter is always computed if it is call-by-value), so the generated code output of this example would not conform to the expected one.

Additionally, for the sake of argument, it is worth noting that the right-hand side of an expression containing `||` or `&&` operators does not have to be evaluated in the particular cases like:

- `||` - left-hand side of the operator is a constant value of true, meaning that the whole expression will always be true
- `&&` - left-hand side of the operator is a constant value of false, meaning that the whole expression will always be false

This is reflected in the optimization which is implemented in the `BooleanOpsExpOpt` trait of LMS (Opt signifying "Optimization"; example simplified for brevity sake):

```
override def boolean_and(lhs:Exp[Boolean], rhs: =>Exp[Boolean]):Exp[Boolean]={
  (lhs, rhs) match {
    case (Const(false), _) => Const(false)
    case _ => super.boolean_and(lhs, rhs)
  }
}

override def boolean_or(lhs:Exp[Boolean], rhs: =>Exp[Boolean]): Exp[Boolean]={
  (lhs, rhs) match {
    case (Const(true), _) => Const(true)
    case _ => super.boolean_or(lhs, rhs)
  }
}
```

Code Example 13: Boolean && and || optimizations

While we were able to modify the expected output, to conform to the calling convention that worked, this was not the behavior that we wanted to achieve, so this solution was still unsatisfactory. After consulting with several colleagues and experts from the domain, no explanation to the behavior could be found. A question presenting this example was asked on a popular programming question and answer website Stack Overflow [32], which drew attention of additional users and experts, but a solution was not found.

Thus, as this behavior seemed to be a possible bug in the compiler, an issue with a detailed explanation and instructions on how to reproduce the behavior was posted to the Scala issue tracker. The name of the issue is "[SI-9660] Implicit conversion from parent trait with a call-by-name parameter (Boolean example)" [33] and was tested in multiple representative Scala distributions. A resolution to this issue is yet to be found.

7.2 Applied to real-world domain-specific languages

Having successfully implemented macro-virtualization to almost all of the LMS tutorials, we were starting to look into applying it to Delite domain-specific languages, in order to evaluate how it behaves in a larger, real-world setting. Albeit issues encountered, this seemed as a promising path to take, as Delite DSL applications dealt mainly with staged types in their code. Unfortunately, not all of the Delite DSL applications gave expected results. After an investigation it was noted that some of the implicit conversions that are now being used had difficulties resolving. This was a known problem from before, especially in larger DSLs which contain many components. In this case, the number of function overloads and implicit conversions (implicit search space) becomes very large, making it difficult for the compiler to choose the right function.

A systematic way needed to be found to resolve this problem, as we wanted to avoid needing to manually wrap non-staged expressions with `unit` method, which would lift them. Having a layer like Yin-Yang [34] that is able to automatically translate shallowly-embedded code to its deeply-embedded counterpart and at the same time rewrite it, avoiding the need of employing implicit conversions seemed as a promising approach to solving this problem. Unfortunately, due to the time constraints and some differences in design we were unable to successfully employ Yin-Yang to Delite DSLs. We will later present Yin-Yang in more detail and mention the problems that we encountered when applying the translation to Delite DSLs.

Another approach that seemed promising in order to reduce the number of implicit conversions was using custom types in the deep embedding which directly provide the required operations. We will later present this approach and some of its benefits and limitations.

Both of these approaches required developing a different kind of shallow embedding that LMS had at that point. Namely the new shallow embedding needed to be pure library-style, so called direct, using existing types available, without any `Rep` wrappers. In the next chapter, we are going to present Forge, a meta-DSL that is able to produce DSL implementations from a specification-like program. This DSL is an integral part of the Delite ecosystem, having many of the most important Delite DSL being generated by it. Further, we will present the new direct embedding and how we went about creating a new template for automatic generation of this embedding based on DSL specifications currently used by Forge.

8. Forge meta-DSL

8.1 Overview

Forge [35] domain-specific language is a meta-DSL (meaning a DSL which deals with other DSLs) that is able to generate DSL implementations from specification-like programs. Although it has been designed primarily with the goal of generating Delite DSLs, it is generic enough to be able to produce a wide range of other DSL implementations. Through the use of common high-level abstractions (data structures, parallel patterns, effects) it is well-suited for generating different kinds of high-performance embedded DSLs.

The motivation behind Forge comes from the fact that developing efficient embedded DSL implementations is still considered very hard. Developing an expressive, safe, and efficient DSL by hand requires a considerable amount of software expertise and forces DSL authors into complicated tool-chains that usually slow down prototyping and debugging. Delite framework eases a part of this burden as it provides all of the compiler and runtime support needed for the development of a high-performance domain-specific language, including a flexible architecture for heterogeneous code generation, but this expressiveness often requires writing large amounts of boilerplate code and adds to complexity. While Delite makes developing high-performance DSLs easier than developing external DSLs from scratch, it still requires an amount of knowledge which is greater than that of a usual domain expert.

Forge addresses this problem by allowing DSL authors to write their DSLs as a high-level specification, which Forge is able to process and automatically generate both a library-like Scala implementation of the DSL and a high-performance version using the Delite compiler framework. Compared to manually written DSLs, the use of Forge reduces the required number of lines of code by a factor of 3-6x and does not sacrifice any performance. Forge is able to easily achieve this as it actually builds an intermediate representation from the specification, based on which it is able to generate various embeddings (various versions of DSL APIs/implementations). This is possible through the previously explained methodology - Forge is an embedded, staged DSL, based on the same LMS infrastructure as Delite.

Without going into the details of the Forge Language Specification, which is described in [35], from a high-level, it is worth remembering that Forge abstracts over key parts required for the development of high-performance DSLs: front-end syntax, data structures, operation semantics, and parallel implementation. Still, while Forge tries to make declaring DSL semantics simple and

concise, it is primarily meant for the high-performance embedded DSL compilers. Reproducing a sequential library interface exactly is not one of its goals.

This meant that developing a new shallow embedding might need some modification in the way Forge generates DSL implementations. In order to better understand how the new embedding fits into Forge, let us take a look at the most important pieces of the Forge infrastructure:

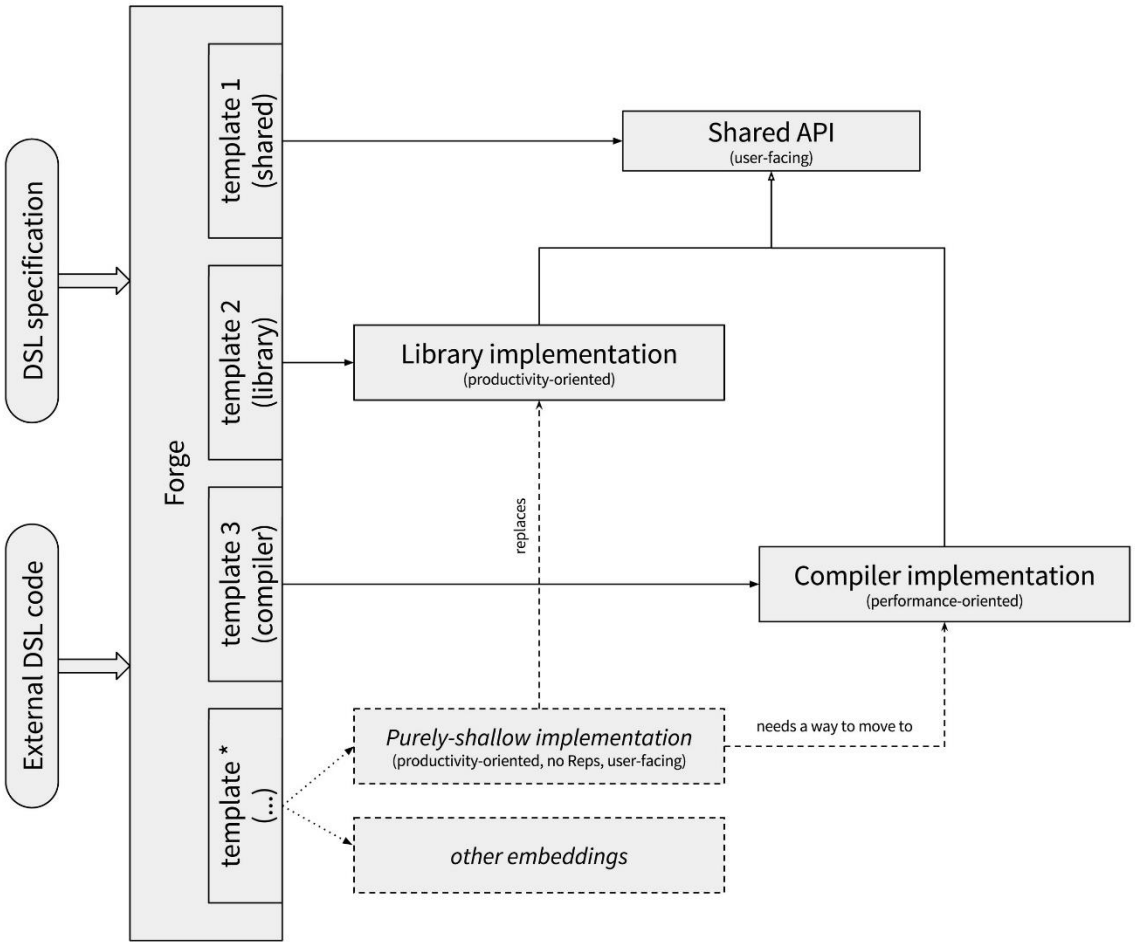


Figure 2: Overview of the Forge meta-DSL

When run, Forge constructs an IR of the DSL based on the input specification. Then it is able to traverse the IR and use different template code generators to generate different implementations. For the cases where Forge is not expressive enough, external DSL code placed in a predefined directory can provide both general (found in the "static" directory) and DSL-specific (found in the "extern" directory) code to a particular implementation.

Before starting this project Forge was able to generate 3 main types of embeddings, as show on the figure above. Namely these were:

- shared - user facing abstract DSL API, which requires mixing-in of an implementation

- library - library-style implementation, which while being directly compiled and executed, still contains Rep wrappers, as it conforms to the shared API
- compiler - deeply embedded implementation, aimed at high-performance and executing on heterogeneous architectures

A big benefit of using Forge comes from the fact that the user is simply able to create a new template based on which a new embedding can be created from the same DSL specification, enabling faster iteration and experimentation with new embeddings.

Having the high-level organization of the embeddings in mind, let us take a look at a concrete example of one of the Forge generated DSLs, OptiML. This DSL provides a solid point for comprehensive testing, namely because of its large size. Nevertheless, its size makes it unsuitable for faster testing, as needed when experimenting with new embeddings, so later we will present and use other Forge-generated DSLs, more appropriate for the task.

8.2 OptiML

OptiML [36] is an embedded domain-specific language for machine learning. It is targeted at machine learning researchers and algorithm developers, aiming to provide a productive, high-performance environment for linear algebra, supplemented with machine learning specific abstractions. In particular, OptiML is designed to allow statistical inference algorithms expressible by the Statistical Query Model to be both easy to express and very fast to execute. These algorithms can be expressed in a summation form, and can be parallelized using fine-grained map-reduce operations. OptiML employs aggressive optimizations to reduce unnecessary memory allocations and fuse operations together to make these as fast as possible. OptiML also attempts to specialize implementations to particular hardware devices as much as possible to achieve the best performance.

OptiML contains a suite of more than 20 applications implementing popular algorithms, including k-means clustering, Naive Bayes classification, Gaussian Discriminant Analysis (GDA), logistic regression etc. Doing a clean compilation of the whole OptiML DSL, including the applications took 1100s on a modern Intel Core i7 laptop. The embeddings compiled were using the way in which the OptiML trait organization was similar to the one shown in the MySimpleVector example. This was a of course a substantial amount of time that needed to be reduced.

In order gain insights on how could we go about reducing the compilation times, we logged the timings of different phases of compilation using the flag forwarded to sbt [37] (the most popular Scala build tool) `-DshowTimings`. While exposing the full log would consume too much space in this thesis, it was clearly noted that most of the compilation time was being spent in the “mixer” phase. The “mixer” phase is responsible for dealing with resolving the composed trait compositions. As stated in the overview of Scala compiler phases, it eliminates traits, replacing them with classes and interfaces [38].

A simple improvement to the current design was noted, which brought a boost in compilation performance. Generic OptiML application traits:

```
trait OptiMLApplicationCompiler extends OptiMLApplication with
  DeliteApplication with OptiMLExp

trait OptiMLApplicationInterpreter extends OptiMLApplication with OptiMLLib
```

were changed to be abstract classes:

```
abstract class OptiMLApplicationCompiler extends OptiMLApplication with
  DeliteApplication with OptiMLExp

abstract class OptiMLApplicationInterpreter extends OptiMLApplication with
  OptiMLLib
```

In a test on a local laptop machine, this sped up the compilation from 1100s to 638s (by a factor of ~1.7x). The reasoning behind this improvement was that in the old design the compiler needed to reassemble all the traits each time an application object was instantiated, while making generic applications abstract classes in the new design forced the compiler to assemble the traits only once (for each generic application abstract class).

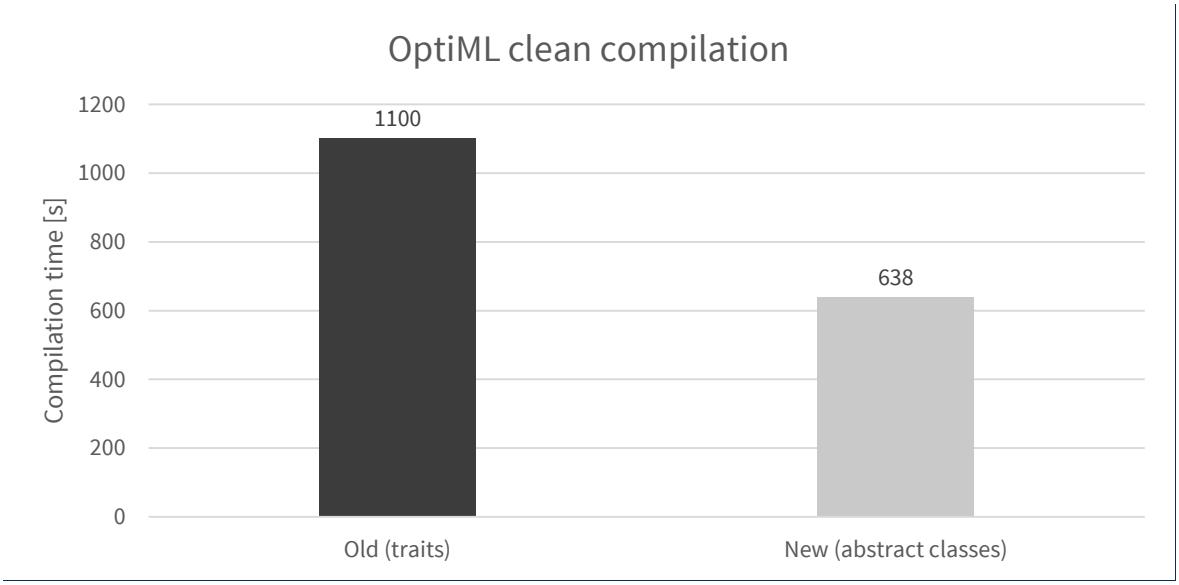


Figure 3: Trait and abstract class application compilation times

Cake	Old (traits)	New (abstract classes)	Speedup
Generic applications	1100s	638s	1.72x

Table 1: Comparison between trait and abstract class application compilation times

Nevertheless, after these improvements, the mixer phase was still taking a substantial amount of time of the whole compilation process. We went deep into analyzing application compilation times. As a testing example, we used GDA (Gaussian Discriminant Analysis) OptiML application.

The cost of compiling an empty trait:

```
trait GDA { def main() = { } }
```

amounted to 3 seconds.

The cost of compiling an empty application (only API mixed in) with no runners:

```
trait GDA extends OptiMLApplication { def main() = { } }
```

amounted to 5 seconds.

The cost of compiling an empty application with one runner:

```
object GDACompiler extends OptiMLApplicationCompiler with GDA { ... }
```

amounted to 12 seconds.

The cost of compiling an empty application with both compiler and interpreter runners was 17s.

As a table:

Code	Empty trait	Empty app (API)	1 runner	2 runners
Compilation time	3s	5s	12s	17s

Table 2: Current minimal OptiML application compilation times

Although it was clear that the creation of the cake was contributing substantially to the compilation time, the overhead seemed to be amortized when compiling two applications with both runners, as the total compilation time amounted to only 20 seconds instead of 2 * 17 seconds = 34 seconds.

Compiling an application with both runners from a smaller DSL OptiQL amounted to 8 seconds, implying that the cost is proportional to the size of the DSL cake.

We also took a look at how implicit resolution affects compile times. OptiML apps took between 18s and 30s to compile. Adding implicit conversions and parameters explicitly into several of them brought the compile time down to the 17 seconds floor. This experiment proved that while the cake pattern has the most influence, implicit search presents another significant contributor to long compilation times.

Taking our experimentation further, we were looking into obtaining more fine-grained info about application compilation times. We used the sbt flag `-DshowTimings` which outputted timing for every phase of the compilation. The following table presents our results:

Experiment	typer [ms]	refchecks [ms]	specialize [ms]	erasure [ms]	mixin [ms]
OptiML app 1	776	710	1066	484	5051
OptiML app 2	12	471	1	382	3
OptiML app 3	12	449	1	347	2
OptiQL app	771	276	531	248	1560
OptiML cont	622	464	656	423	4723

Table 3: Detailed overview of compilation phases times for various DSL applications

For simplicity sake, only the front-end phases which were the largest contributors to compile times are shown. This is enough for the discussion, as our optimizations are aiming at the front-end.

In the first experiment, we batch-compiled 3 empty OptiML applications. Only the phases refchecks and erasure were performing larger work for all 3 applications, while typer, specialize and mixin phases were contributing significantly only during the compilation of the first application, meaning that they were either expensive to boot up and/or they were dealing mostly with composing of the DSL cake, which is done only once and reused for all applications. In total, the first application took about 17 seconds to compile, while every subsequent application added between 2-3 seconds.

The second experiment compiled an empty OptiQL application. All the phases ran faster, but the longest one was still mixin. This suggested that composing the cake was indeed the main culprit for long compilation times.

The third experiment tried compiling an empty OptiML application continuously (on detected file change) using ~compile command in sbt. This lowered the floor of compilation from 17 to 12 seconds, but the mixin phase still took the longest time to execute. Unfortunately, every line change exhibited the same high mixin cost. Our intuition about this behavior was that sbt keeps the same Java Virtual Machine running, but does not actually keep the same instance of Scala compiler running as for example fsc [39] does, thus each line change pays for all phases initializing, including mixin.

The last experiment followed from the third and showed that when using continuous compilation without runners, the time floor drops to only 2 seconds. So a possibly beneficial approach for initial application development, when people usually care about type-checking the most, would be not to use runners or only include the interpreter runner (7 seconds floor), until performance is required.

Performing these detailed experiments gave us important clues that implementing the new shallow embedding in a direct style might lead to further significant speedups in compilation times.

Looking into the details why:

1. the current shallow DSL embedding relied on composing large trait “cakes” into a DSL application trait that could be used with applications. While this generic approach suited the previous design that shared a common API between the shallow and the deep embedding, it had an important downside. Namely, every application mixed in a whole DSL “cake”, even if it was only using just a few functionalities. This was especially notable with larger DSLs as OptiML which mixes-in more than 80 traits.

2. the new shallow DSL embedding was envisioned in a direct style meaning that all of the required classes / objects need to be imported. Even without specifying the exact imports and relying on the compiler to resolve wildcard imports, performing a simple search through the classpath and loading only the required classes / objects is a much more efficient and faster process for the compiler rather than loading all of them, as the previous design required. This is of course considering that we want to have the same level of convenience in both designs of not requiring the user to specify exactly which traits or classes/objects are being used, potentially making a custom “cake”.

Further, using the new shallow embeddings promised to bring additional improvements in compilation times, as the implicit conversions and consequently the implicit search did not need to be performed by the compiler - none of the types were wrapped, so the compiler could directly resolve the operations on values of DSL types, instead of having another layer of indirection to overcome.

The next chapter will present the details of the new shallow embedding.

9. New shallow embedding

The new shallow embedding was modelled after the current shared embedding, in order to remain compatible with the shared API as the current deep embedding still relied on it. There were of course several important differences:

1. no method signatures or other members contained `Rep` types
2. instead of using traits, instance methods were put inside classes while the static ones were put inside the corresponding objects
3. all of the abstract methods corresponding to the ones from the shared API were now implemented
4. classes imported required dependencies rather than mixing them in

As an example, we'll take a look at one other Forge-generated DSL, OptiQL. While being smaller than OptiML, OptiQL is still suitable to test fundamental ideas and the correctness of the new shallow embedding, as it possesses a large part of the essential common functionality which larger DSLs like OptiML are based on too.

OptiQL is a Delite DSL for data querying of in-memory collections, based on the same ideas behind LINQ to Objects [40]. OptiQL provides two custom data structures - `Table` and `Date`. The core OptiQL data structure `Table`, contains a user-defined schema and is able to be processed by a set of implicitly parallel query operators, such as `Select`, `Average`, `GroupBy`. `Date` data structure contains a date encoded in an integer - lowest 5 bits represent the day (2^5=32 > 31 days), the next 4 bits the month (2^4=16 > 12), and the rest of the higher bits represent the year. The reduced interface of `Date` was shown previously, when presenting how operations needed to look after the switch to macro-based virtualization. Now, we will take a look at a similar interface in the new shallow embedding:

```
class Date(__value: Int) { self =>
  protected var value = __value

  import Date._
  def <(__arg1: Date) = date_lt(self,__arg1)
  def >(__arg1: Date) = date_gt(self,__arg1)
}

object Date {
  // static ops
  def apply(__arg0: Int) = date_object_apply(__arg0)

  // compiler (protected) ops
  protected def date_value(self: Date): Int = {
    self.value
  }


  // abstract implemented ops
  def date_object_apply(__arg0: Int): Date = {
    new Date(__arg0)
  }
  def date_lt(self: Date,__arg1: Date): Boolean = {
    date_value(self) < date_value(__arg1)
  }
  def date_gt(self: Date,__arg1: Date): Boolean = {
    date_value(self) > date_value(__arg1)
  }
}
```

Code Example 14: Example of the new shallow (direct) embedding

As it is clearly noticeable, the new shallow embedding presents users with a much cleaner and easier-to-understand API. In order to evaluate the compilation times of the new embedding, we used the QuerySuite benchmark, available in OptiQL. The results were as follows:

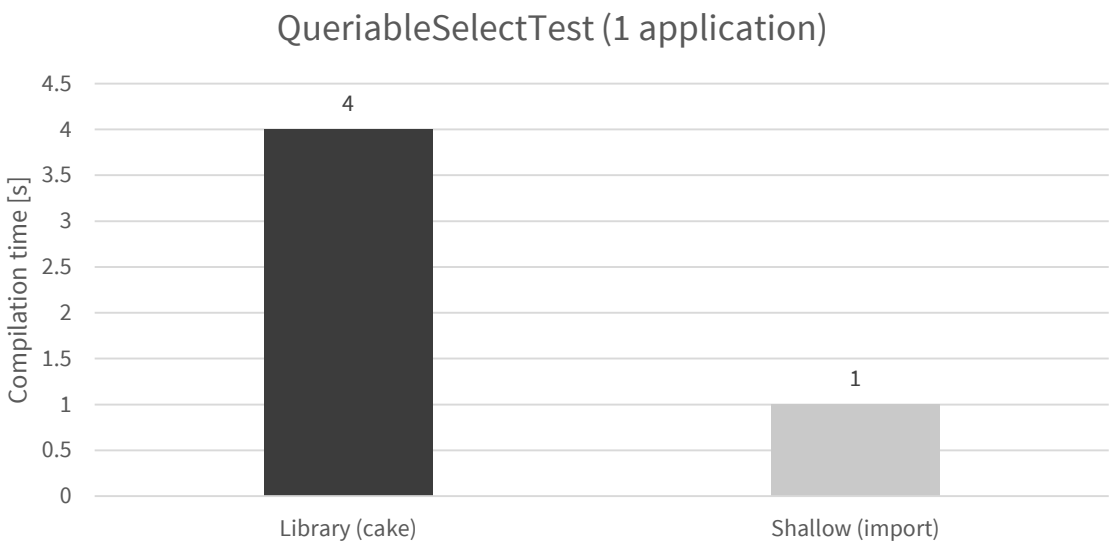


Figure 4: Compilation time of the new shallow (direct) embedding evaluated on one test

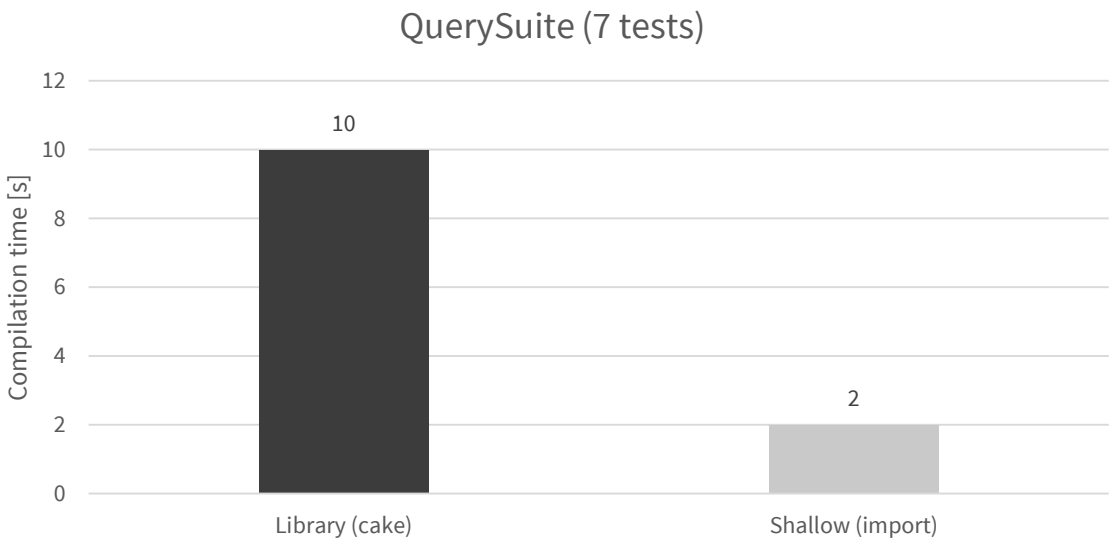


Figure 5: Compilation time of the new shallow (direct) embedding evaluated on seven tests

As a table, the results looked like this:

Code	Library (cake)	New shallow (import)	Speedup
QuerySuite - 1 test	4s	1s	4x
QuerySuite - 7 tests	10s	2s	5x

Table 4: Comparing library (cake pattern) and direct embedding (import) compilation times

The results looked very promising, scaling well from small applications to larger bodies of code. It was clear that this is a promising way to go towards the search of improved compilation times. Nevertheless, there was an important drawback in the new approach. Unlike the previous shallow embedding, which shared the API with the compiler embedding, ensuring that all of the operations from the API are available and implemented in both shallow and deep variants, the new shallow embedding did not provide us with those guarantees. Namely, as the users are now able to write their programs not using the API, but rather the whole Scala language, it becomes very easy to make a mistake. For example, a DSL user might use an operation or a function not available in the deep embedding, which type-checks / resolves in the new shallow embedding, but fails to compile when using the deep embedding. This again might lead to convoluted type errors and expose users to the underlying deep implementation which contains `Rep` wrappers. The feature that could resolve this is the so-called “language restriction”, meaning verifying that all of the features used in the shallowly embedded code are actually available in the domain-specific language. Yin-Yang, a framework for embedding DSLs promised to do exactly that, along with automatic type-translation from shallow to deep code. This made it a great candidate for bridging the gap between the new shallow embedding and the deep embedding.

In the next chapter, we will present Yin-Yang and briefly look at how we went about integrating it into the Delite ecosystem

10. Yin-Yang

Parts of this chapter have been based on the "Yin-Yang: Concealing the Deep Embedding of DSLs" paper, by Jovanović et al [34].

Yin-Yang is a framework for DSL embedding that completely conceals the unfriendly interface of deeply embedded languages. Using Scala macros, it reliably translates programs using pure library-style (direct) embeddings into their deeply embedded counterparts. Yin-Yang offers important benefits which ease the development of embedded domain-specific languages:

1) it completely conceals deep embedding abstractions from the users. The reliable translation ensures that programs written in the direct embedding will always be correct in the deep embedding.
2) it allows for using only features supported by both direct and deep embedding by restricting the use of non-supported host language features in DSL programs
3) it additionally simplifies the development of deep embeddings by reusing the core translation to generate a semantically equivalent deep embedding out of a direct embedding

Yin-Yang addresses the first problem, making the translation possible, through a two-step approach:

1) Perform language virtualization - as this functionality has been explained before, we will not go into details of it here.
2) Perform embedded DSL intrinsification - convert DSL operations and types from their direct to deeply embedded variants. Yin-Yang performs type translation by mapping every DSL type in the direct, virtualized application body to the corresponding type in the deep embedding. As many mappings between embeddings may be possible Yin-Yang stays generic in that sense that it allows configuring of the type translation. Other steps, of less significance for the given exposition include operation translation, conversion of constant literals, and the translation of free variables used in the direct program.

Secondly, Yin-Yang resolves the problem of unrestricted host language constructs by performing an additional verification step that checks if a method from the direct program exists in the deep embedding. Verification is performed in such a way that Yin-Yang traverses the tree generated by the translation and for each method call verifies if it type-checks in the deep embedding. If the type checking fails, Yin-Yang provides comprehensive error messages about both unsupported methods

and unsupported host language constructs. While this adds to the compilation time of a directly embedded program, it guarantees an important safety property that the program is legal in both embeddings.

Lastly, Yin-Yang alleviates the effort of DSL authors of performing the error-prone task of writing direct and deep embeddings in parallel. It the core translation and allows for automatic generation of the deep embedding that is conformant to the specified direct embedding.

When integrating Yin-Yang to the Delite compilation pipeline, this was one of the problems that we ran into. As Forge takes a fundamentally different approach than Yin-Yang, of generating a "shared" API used by both a library (shallow) implementation and a compiler (deep) implementation, instead of generating the deep embedding out of the shallow one, we had no guarantees that the newly developed shallow (direct) embedding will work with the translation. Namely, the deep embedding was still being generated in a way that exposed the "shared" API to the user, while the new shallow embedding (direct) was being generated by Forge in a way that tried to closely resemble the "shared" API in order to conform to it. But other than the specification and method names, these embeddings had little else to share. As observed before, resolving debugging errors originating from non-conforming embeddings proved to be a tedious task.

One of the other problems that were encountered when performing the translation was the use of context bounds like Manifest. When translating the direct embedding method calls to their deeply embedded counterparts, Yin-Yang lifted Manifest into a `Rep` version, which did not conform to the deeply embedded method, which still expected only a non-wrapped Manifest implicit parameter. A way in which we could circumvent this was to remove the Manifest completely from the direct embedding, which required modifying how Records work and how Arrays which required an implicit Manifest parameter [41] are created. This was performed, but the issue pertained with other context bounds, demonstrating differences in which the current Delite DSL design and Yin-Yang dealt with the direct embedding. Another more general way to work around this is to develop a pre-processing step in Yin-Yang, which would ignore all the context bounds / implicit parameters in the type translation. While this approach was proven to work before, it was considered more a temporary, rather than a principled solution.

This issue also made one of the fundamental differences between LMS's and Yin-Yang's approach to DSLs more obvious. Although Yin-Yang translates the direct embedding to the deep domain, postponing the application compilation to runtime (different compilation stage), it does not allow staging.

Resolving these and other encountered issues proved to be a laborious task considering the time left for conducting this project, thus we have conceded our efforts in this domain, in a search and experimentation towards potential new deep embeddings, which might provide us with improved compilation times in the deep domain. Still, bridging the differences between LMS and Yin-Yang remains an interesting task for future work and discussion, both from theoretical and practical side. The next chapter will present the experimentation towards a new deep embedding based on a custom types approach.

11. New deep embedding

The new deep embedding based on custom types was inspired by Feldspar DSL [42] and Scalan, an alternative framework for domain-specific compilation in Scala [43].

The design of the new deep embedding had two primary goals:

1. reduce the need for implicit conversions / infix methods etc. (specific goal: reduce compile times in Delite)
2. keep a type distinction between present stage and future stage values, and consequently support staging

Where we originally had a type distinction between, say:

`Int` and `Rep[Int]` (present stage vs later stage)

We now use:

`scala.Int` and `IR.Int`

One of the questions that presented itself now is what to do for constructs like if/then/else, which took a staged expression `Rep[T]` for then/else branch parameters, and now as we do not have `Rep` wrappers anymore, there was only a generic type `T`? The first solution proposed was simple: if/then/else control construct just takes a generic `T` type parameter, but we use a type class `T:Typ` to denote that `T` is a DSL type (previously wrapped in `Rep`). This type class acts as an isomorphism that converts from the user-visible type to the internal IR and back.

The main benefit of the new deep embedding is that the deep DSL interface can get away without any implicit conversions. Previously, the deep embedding needed to handle large number of overloads and implicit conversions, especially in cases such as operations on primitive types, out of which a majority can be implicitly combined (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`). Based on tests performed, the extreme overloading and implicit conversions were one of the biggest contributors to compilation times. Thus, having the operations defined directly on types in the new embedding was expected to drastically cut down compilation times.

Additionally, switching an application from the new shallow (direct) to the new deep embedding does not require a large effort - by simply instantiating the deep DSL "cake" and importing the necessary API in scope, an application becomes deeply embedded.

In order to better understand the new deep embedding, let's take a look at a reduced, simplified code example which presents the main ideas behind it:

```
trait Base {

  type Typ[T] // Denotes a DSL type
  type Lift[A,B] // Evidence that type A can be lifted to type B
  implicit def identLift[T:Typ]: Lift[T,T]
  implicit def lift[T,U](x:T)(implicit e: Lift[T,U]): U
}

trait BaseExp extends Base {
  // Wraps a String value for simplicity sake
  case class Exp(s: String) { override def toString = s }

  var numVars = 0

  implicit def reflect(s: String) = {
    numVars += 1
    println("val x"+numVars+" = "+s)
    Exp("x"+numVars)
  }

  trait Typ[T] {
    def from(e:Exp): T
    def to(x:T):Exp
  }

  trait Lift[A,B] {
    def to(x:A):B
  }

  def identLift[T:Typ]: Lift[T,T] = new Lift[T,T] {
    def to(x:T) = x
  }
  def lift[T,U](x:T)(implicit e: Lift[T,U]): U = e.to(x)
  def typ[T:Typ] = implicitly[Typ[T]]
}

trait DSL extends Base {

  trait IntOps {
    def +(y: Int): Int
    def *(y: Int): Int
  }
  type Int <: IntOps
  implicit def intTyp: Typ[Int]
  implicit def intLift: Lift[scala.Int,Int]
```

```
  trait ArrayOps[T] {
    def length: Int
    def apply(x: Int): T
    def update(x: Int, y: T): Unit
  }
  type Array[T] <: ArrayOps[T]
  def NewArray[T:Typ](x: Int): Array[T]
  implicit def arrayTyp[T:Typ]: Typ[Array[T]]

  // make sure we are able to lift both branches to the same type C
  def __ifThenElse[C,A,B](c:Boolean, a:A, b:B)(implicit mA: Lift[A,C],
    mB: Lift[B,C], mC: Typ[C]): C
}

trait Impl extends BaseExp with DSL {
  case class Int(e: Exp) extends IntOps {
    def +(y: Int) = Int(e+"+"+y.e)
    def *(y: Int) = Int(e+"*"+y.e)
  }
  val intTyp:Typ[Int] = new Typ[Int] {
    def from(e:Exp) = Int(e)
    def to(x:Int) = x.e
    override def toString="Int"
  }
  val intLift:Lift[scala.Int,Int] = new Lift[scala.Int,Int] {
    def to(x:scala.Int) = Int(Exp(x.toString))
  }

  case class Array[T:Typ](e: Exp) extends ArrayOps[T] {
    def length = Int(e+".length")
    def apply(x: Int) = typ[T].from(e+"("+x.e+")")
    def update(x: Int, y: T): Unit = reflect(e+"("+x.e+") = "+typ[T].to(y))
  }
  def NewArray[T:Typ](x: Int):Array[T]=Array("newArray["+typ[T]+"]("+x.e+")")
  implicit def arrayTyp[T:Typ]: Typ[Array[T]]=???//elided, similar to intTyp

  def __ifThenElse[C,A,B](c:Boolean, a:A, b:B)(implicit mA: Lift[A,C],
    mB: Lift[B,C], mC: Typ[C]): C =
      mC.from("if ("+c.e+") "+mC.to(mA.to(a))+" else "+mC.to(mB.to(b)))
}

def main() {
  val IR: DSL = new Impl {}
  import IR._ // remove this line -> the program uses the direct embedding
  // primitives: Int resolves to IR.Int as opposed to scala.Int (import)
  val x: Int = 5
  // conditional: 3 and 7 (type scala.Int) are automatically lifted
  val y = if (true) 3 + x else 7
  // arrays: constructor requires T:Typ
  val xs = NewArray[Int](7)
  xs(y) = x
  // nested arrays work as well
  val ys = NewArray[Array[Int]](1)
  ys(0) = xs
}
```

This simplified implementation presents the most important ideas behind the new embedding. It was further expanded and refined and it is still under active discussion of researchers involved in this project.

One of the problems noticed is that simply specifying the type class `Typ` on a generic type parameter was not enough for the compiler to disambiguate overloaded methods. In more details, the Scala compiler cannot disambiguate between overloaded methods that only differ in context bounds. For example, methods:

```
def __equal[A:Typ,B](__arg0: A,__arg1: B): Boolean =
  forge_equals(__arg0, unit(__arg1))
def __equal[A,B:Typ](__arg0: A,__arg1: B): Boolean =
  forge_equals(unit(__arg0), __arg1)
```

were detected as ambiguous when compiling an expression of type: `Int == scala.Int`, although we would expect the first overload to resolve. This approach does not work due to the fact that context bounds are simply syntactic sugar for curried implicit parameters. Scala finds the functions with the same first parameter list differing only in curried parameters ambiguous, because the function could be invoked using only the first parameter list, without resolving the second one, creating a new function instead of the result.

A current solution found for this is defining `__equals` on all types and pattern-matching the four possible combinations of lifted / non-lifted parameters in the implementation. In order to make that possible, we made all lifted types extend a common super-type that that provides an extractor for `Exp`:

```
trait Wrapped[T]{
  def e: Exp[T]
}
```

Now the signature of deep implementations looks slightly differently e.g.:

```
class Int(val e: Exp[scala.Int]) extends IntOps with Wrapped[scala.Int]
```

Now, in `__equals` method, we are able to match on the `Wrapped` super-type, disambiguating between lifted and non-lifted arguments of the method. Other solutions (e.g. using an Equality type class) have been proposed, but after a discussion we currently settled for the aforementioned one.

Taking the ideas behind this new embedding, we have implemented a Forge template that generates the new deep embedding based on the same Forge specifications behind other generated DSLs. As the idea was relatively new, and the time for testing it was limited, we have decided to evaluate it on a simple DSL, namely SimpleVector, a DSL providing facilities for calculations using vectors. Still, this DSL was a significant place to start as it contains all of the common Scala operations, which are some of the most used operations in general DSLs. Naturally, all of the primitive operations are a part of the common set, so behavior of rewritten overloaded methods for all of the primitive type operations could be tested.

Compilation times were tested on the application HelloWorld which tests most of the capabilities of the SimpleVector DSL:

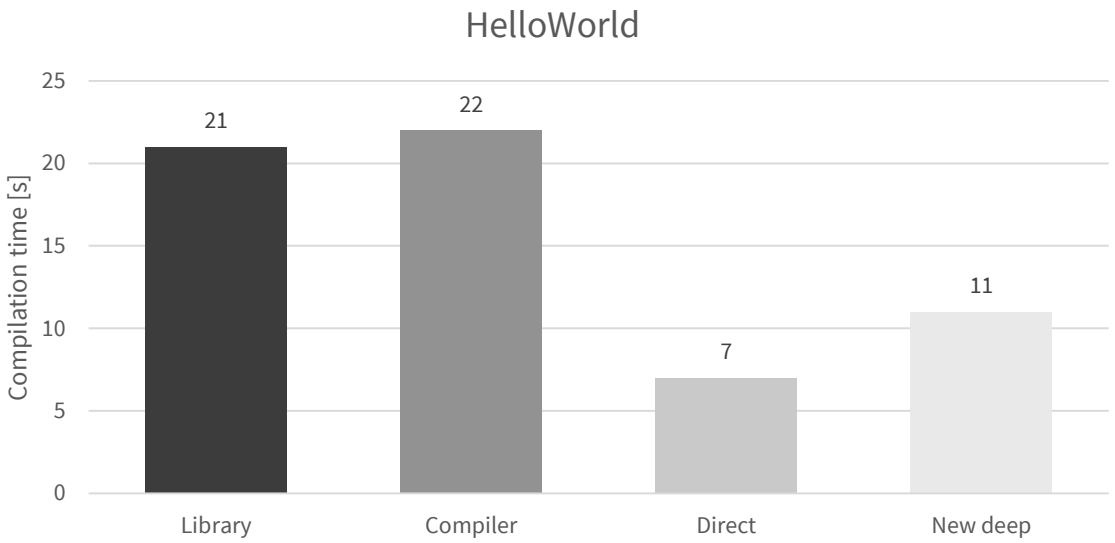


Figure 6: Compilation times of a DSL application in various embeddings

As a table, the results looked like this:

Code	Old	New	Speedup
HelloWorld shallow	21s	7s	3x
HelloWorld deep	22s	11s	2x

Table 5: Evaluating compilation times with the new custom types embedding

The results turned out to be very beneficial. As predicted, we have gained substantial speed-ups in compilation times. Although the new deep embedding is still composed as a cake of traits, defining operations directly on lifted types and having a reduced number of implicit conversions led to significant improvements in compilation times. As this approach seemed promising it is being further pursued in the scope of the LMS and Delite research projects. It is worth noting though that the new conversion from shallow to deep embedding using only an import statement makes no guarantees about the features of the language being used in the shallow (direct) embedding. If these new embeddings are to become a standard for the LMS and Delite ecosystems, this presents a research question worth looking into.

12. Conclusion

This master thesis project dealt primarily with two goals, namely removing the dependency that the Lightweight Modular Staging framework had on the specialized version of the Scala compiler, Scala-Virtualized, and reducing the compilation times of applications written in domain-specific languages based on the Delite compiler framework.

Through a systematic approach we have analyzed the first goal, specified the requirements, investigated the current state of projects that needed to be modified in the Delite ecosystem and laid out the steps in which to implement the solution. Moving away from Scala-Virtualized, to a completely macro-based virtualization, proved to be a non-trivial task. There were many unexpected issues on the way that needed to be overcome. Among many others, this endeavor presented me with a first-time experience of finding a potential bug in a compiler. Analyzing the corresponding behavior, which is described in the SI-9660 issue [33] on the Scala issue tracker took a substantial amount of time in the first half of the project, but provided me with an invaluable experience of studying various versions of LMS in a great depth, by meticulously checking their code, and finally identifying and understanding the intricate interactions which resulted in the unexpected behavior. I personally hope that this issue will see an explanation or resolution in the near future. Coming back to macro-based virtualization, we have shown that it can be successfully applied on a larger-scale project and domain-specific languages such as those designed in the Delite ecosystem. Still, the well-known implicit conversion problematic additionally motivated the need for a translation layer or a new sort of embedding, less reliant on implicit conversions.

We took on the second goal of reducing the compilation times by carefully analyzing the compilation behavior of Delite DSL applications and identifying performance bottlenecks. From those insights, we developed a new shallow (direct) embedding for Delite DSLs, which was a first step towards a more performant compilation process. As we wanted to make this generation reusable, we developed a template for Forge meta-DSL, using which Forge is able to create direct embeddings for arbitrary Forge DSL specifications. This part of the project required learning an additional domain-specific language, namely Forge meta-DSL, but this added effort was followed by great gratification once the template was functioning and we have measured the duration of new, improved compilation times.

Once we had the direct embedding functioning we needed a way to easily relate directly embedded applications to deeply embedded applications. Experimentations with using an automatic translation layer between direct and deep embeddings in the form of Yin-Yang, showed a lot of promise. The translation was unfortunately not fully implemented because of the timing constraints and priorities set for the project, but also because of some fundamental differences between the current Delite / LMS ecosystem and Yin-Yang's approach to DSLs (automatically generating a deeply embedded interface out of a directly embedded one vs using Forge for generation of both, not supporting staging vs requiring staging etc.).

On the other hand, experimentations with the new custom-types deep embedding provided us with much-improved compilation times. While having some drawbacks, the results signaled that this idea might be interesting for further use and exploring.

This master thesis project touched upon all aspects of software engineering, challenged me professionally and personally, and required me to engage in a domain which I was relatively new to, making me learn more than ever before. During the past 6 months, I greatly improved my Scala, functional programming, and advanced compilers knowledge both through studying vast amounts of written and online material, but also through direct experimentation and implementation. I studied and touched code of 8 different projects (LMS, LMS tutorials, macro-virtualization, hyperdsl, Delite, Forge, Yin-Yang, Scala Records), all of which were important for understanding and functioning of the Delite ecosystem as a whole. I faced changing requirements, and adapted accordingly. I took a methodical approach towards analyzing issues, and measured predicted results. I was blocked on problems for days, and through appropriate discussions and brainstorming with colleagues I overcame them. I have expanded my views on what is possible in programming and computing. Most of all I had a lot of fun and I feel inspired towards learning more in the future!

References

[1] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys 28,* 1996.

[2] V. Jovanović, *Language Support for Embedded Domain-Specific Languages, PhD thesis,* Lausanne: EPFL, 2016.

[3] T. Rompf and M. Odersky, "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs," *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10),* pp. 127-136, 2010.

[4] M. Odersky, *The Scala Language Specification,* http://www.scala-lang.org/, 2011.

[5] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanović, H. Lett, M. Jonnalagedda, K. Olukotun and M. Odersky, "Building-blocks for Performance Oriented DSLs," *Proceedings of the IFIP Working Conference on Domain-Specific Languages (DSL'11),* 2011.

[6] A. Moors, T. Rompf, P. Haller and M. Odersky, "Scala-Virtualized," *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM'12),* pp. 117-120, 2012.

[7] W. Taha and T. Sheard, "MetaML and multi-stage programming with explicit annotations," *Theoretical Computer Science,* no. 248, pp. 211-242, 2000.

[8] J. Carette, O. Kiselyov and C.-C. Shan, "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages," *Journal of Functional Programming,* 2009.

[9] C. Hofer, K. Ostermann, T. Rendel and A. Moors, "Polymorphic embedding of DSLs," *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08),* pp. 137-184, 2008.

[10] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," *International Conference on Parallel Architectures and Compilation Techniques (PACT'11),* pp. 89-100, 2011.

[11] E. Burmako, "Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming," *Proceedings of the 4th Workshop on Scala*, 2013.

[12] M. Christie, *Understanding the self type annotation and how it relates to the Cake Pattern,* http://marcus-christie.blogspot.com/2014/03/scala-understanding-self-type.html, 2014.

[13] M. Odersky and M. Zenger, "Scalable component abstractions," *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'05),* 2005.

[14] M. L. Scott, Programming Language Pragmatics, Morgan Kaufmann, 2000.

[15] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[16] M. Odersky, L. Spoon and B. Venners, Programming in Scala, 2nd Edition, Artima, 2011.

[17] D. Westheide, *The Neophyte's Guide to Scala Part 12: Type Classes,* http://danielwestheide.com/blog/2013/02/06/the-neophytes-guide-to-scala-part-12-type-classes.html, 2013.

[18] *Scala Reflection Documentation,* http://docs.scala-lang.org/overviews/reflection/overview.html, 2015.

[19] Wikipedia, *Metaprogramming,* https://en.wikipedia.org/wiki/Metaprogramming, 2015.

[20] Oracle, *The Java™ Tutorials: Type Erasure,* https://docs.oracle.com/javase/tutorial/java/generics/erasure.html, 2015.

[21] E. Burmako, *scala.meta,* http://scalameta.org/, 2015.

[22] E. Burmako, *Macro Paradise,* http://docs.scala-lang.org/overviews/macros/paradise.html.

[23] *scala-macros vs scala-virtualized,* http://www.scala-lang.org/old/node/12494, 2012.

[24] J. P. Moreno, *Scala Macros - Annotate your case classes,* http://www.47deg.com/blog/scala-macros-annotate-your-case-classes, 2015.

[25] J. Suereth, *Scala String Interpolation Documentation,* http://docs.scala-lang.org/overviews/core/string-interpolation.html, 2013.

[26] Wikipedia, *Product type,* https://en.wikipedia.org/wiki/Product_type, 2015.

[27] M. Odersky, *SIP-17 - Type Dynamic,* http://docs.scala-lang.org/sips/completed/type-dynamic.html, 2012.

[28] K. Malawski, *Scala's Types of Types,* http://ktoso.github.io/scala-types-of-types/, 2015.

[29] V. Jovanović, T. Schlatter and H. Plocziniczak, *Scala Records,* https://github.com/scala-records/scala-records, 2015.

[30] T. Rompf and N. Amin, *LMS Tutorials,* https://github.com/scala-lms/tutorials, 2011.

[31] R. Pike and B. Kernighan, *A Regular Expression Matcher,* http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html, 2007.

[32] *Stack Overflow,* http://stackoverflow.com/.

[33] B. Perović, *[SI-9660] Implicit conversion from parent trait with a call-by-name parameter (Boolean example),* https://issues.scala-lang.org/browse/SI-9660, 2016.

[34] V. Jovanović, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch and M. Odersky, "Yin-Yang: Concealing the Deep Embedding of DSLs," *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences (GPCE'14),* pp. 73-82, 2014.

[35] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky and K. Olukotun, "Forge: Generating a High Performance DSL Implementation from a Declarative Specification," *Proceedings of the 12th Internationa Conference on Generative Programming: Concepts and Experiences (GPCE'13),* 2013.

[36] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning,"

Proceedings of the 28th International Conference on Machine Learning (ICML'11), pp. 609-616, 2011.

[37] sbt, *sbt Reference Manual*, http://www.scala-sbt.org/0.13/docs/index.html.

[38] *Scala Internals Wiki: Overview of Compiler Phases*, https://wiki.scala-lang.org/display/SIW/Overview+of+Compiler+Phases, 2011.

[39] *Fast Scala Compiler*, http://www.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/tools/fsc.html.

[40] E. Meijer, B. Beckman and G. Bierman, "LINQ: Reconciling Object Relations and XML in the .NET Framework," *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, 2006.

[41] *Scala Collections Documentation: Arrays*, http://docs.scala-lang.org/overviews/collections/arrays.html.

[42] *Feldspar, Functional Embedded Language for Digital Signal Processing and Parallelism*, http://feldspar.inf.elte.hu/, 2011.

[43] *Scalan, Generic framework for development of domain-specific compilers in Scala*, https://github.com/scalan/scalan, 2015.

[44] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky and K. Olukotun, "Delite: A Compiler Archutecture for Performance-Oriented Embedded Domain-Specific Languages," *ACM Transactions on Embedded Computing Systems (TECS'13)*, vol. 13, 2014.

[45] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky and K. Olukotun, "Language Virtualization for Heterogeneous Parallel Computing," *Onward!*, 2010.