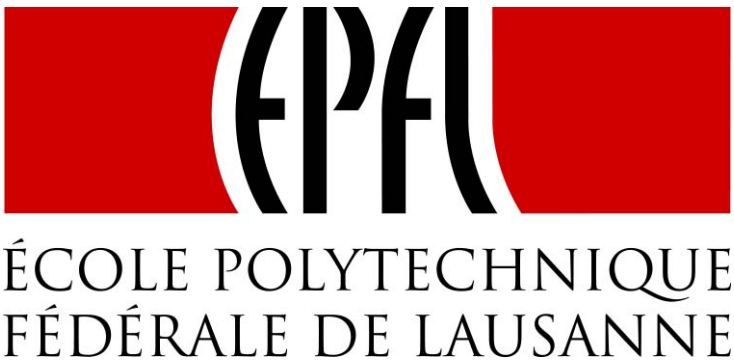
Staged meta-programming, new lms frontend and computation graphs

Ruben Fiszel

December 12, 2016

~ Supervised by Nada Amin and Prof. Martin Odersky ~



Abstract

In this report, we explore staging, in particular the LMS framework and the development of its new frontend whose aim is ease the writing of staged dsl through, among others, shadowing of types. We also explore the usage of this new frontend for a particular case study: Staged computation graphs.

Contents

Introduction

If programming can be considered as an art, then programming languages are the brush of the artists. The tools are chosen according to individual preferences, and of course the constraints of the desired final work. One of those major constraints is efficiency. Efficiency is less crucial nowadays than it was back when computing power was expensive and scarce, but it is still a very desirable goal. Efficiency can be achieved by writing explicitely and very precisely each step of the program. To enable this, some programming languages are relatively close from the model of the hardware. Those programming languages are called "low-level", because close from the machine and thus distant from the heights of abstraction. But as programs grow more complex, the need for abstraction does too.

To raise abstraction power, expressive languages with complex compilers were made. Ironically, the first programmng language, lambda-calculus, was the epitomy of abstraction and thought of before the first very computer to execute it. Those languages, mostly from the functionnal programming world, aim the user to express the intent than the steps, the what rather than the how. The compilers needed for them are heavy machinery that applies all sort of optimization before translating source code to machine language. The more abstract the language, the more gap there is to fill, and the more compilers have opportunity to optimize.

Hence, in an ideal world, we can achieve **abstraction without regret**, writing very abstract programs compiled with optimal efficiency for any platform and any hardware. Unfortunately, we are not there yet and expert implementors are still in need. So should we just throw all our efforts into writing the ultimate compiler for the ultimate language ?

Growing complexity in compiler is not the panacea. In a wide range of programs, compilers are limited by the lack of domain specific knowledge. Indeed,

constraining a program to one specific domain open the door for a wide range of specific optimizations. Furthermore, code generation by "macro" is a rather poor way to enable the further abstraction brought by code generation.

One solution to both issue could be to extend the compiler for each domain and having a very powerful macro system. The other one is to write specific domain-specific language in a staged environment. To avoid reinventing the wheel, embedded DSL is a nice compromise between specialized and general-purpose languages. It is this embedded DSL staged meta-programming path that is explored with LMS, a scala library for runtime code generation. In this report, we will explore a new lms implementation that offer a new user frontend, more convenient for the end-user and with multiple benefits enabled by extended typeclass usage. We also cover a case study of lms applied to the domain of computation graphs.

1. LMS

1.1 Why staging

To see the benefits of a staged-metaprogramming, we will start by an example where it benefits shine: sparse vector multiplication.

```
val v1: IndexedSeq
val block = {
    val prng = new util.Random(123)
    def rd(i: Int) = prng.nextInt(i)

    def newVec(size: Int) = {
        val a = Array.fill(size)(0)
        for (i <- (1 to 10))
            a(rd(size)) = rd(100)
        a
    }

    val v2 = newVec(1000)

    v1.zip(v2) map { case (x1, x2) => x1 * x2 } sum
}
```

This work but it is terribly inefficient. What about all those 0 multiplications. Surely, we can do something better. There is 3 cases.

- v1 and v2 are both known at compile time
- v1 is known at compile time

- v1 and v2 are both only known at runtime

In all three cases, staged-metaprogramming is able to generate efficient code that avoid the (1000 [zip]+ 1000 [*]) operations at runtime and reduce it to 10 operations (no more zip) and even 0 operations in the first case!

How ? Well it is all about available information. We will start with the first case which is the simplest to understand: We know at staging time all the data and all the operation that is applied to the data. Instead of waiting for runtime to apply those operations, we can generate a new block that is strictly equivalent. *Note* that to ensure equivalence, we use a fixed seed so that the number generation is deterministic and the current state of the world at execution is not relevant. So the generated code can be reduced to:

```
val stagedBlock = C
```

where C is a constant which value depend on v1

In the second and third case the code can be generated to this reduced form:

//TODO

```
val stagedBlock = {
    val x0 = new util.Random(123)
    val x1 = v1(x0.nextInt(100)) * rd(100)
    val x2 = v1(rd(100)) * rd(100)
    val x3 = x2 + x1
    val x4 = v1(rd(100)) * rd(100)
}
```

1.2 LMS

Lightweight Modular Staging (LMS) is a library written in Scala that enable staged meta-programming. Meta-Programming is the art of writing computer programs that can handle other programs as data. Thus, they are able to reason about programs, which they are themselves, hence the name. In staged meta-programming, the goal is to generate new programs from annotated programs. Those annotations

are called staged annotions. The program source is called the **meta-program**. The program being generated is the the **object program**. The generation can go through multiple stage. Each stage is another transformation that can leverage new informations to build the final program. LMS is heterogenous: the meta-program and the object program can be written in different programming languages (eg: generating C).

In LMS, the meta-program is written in a subset of Scala. This subset is:

- every operations on any type that has a "lifted" implementation. Int/Float/Double/Array and more are already part of LMS (*batteries included*)
- if-then-else
- for-loop/while
- lambda-functions and named functions
- equalities

We will see later what a lifted implementation is but for now let say that any type can be added to LMS as long as we write some appropriate implementation for it. Other projects like scala-native are focused only on LLVM generation but supports the full scala-language.

1.3 Staging

In meta-programming, the staged annotations can usually take multiple forms:

- String (Yes, you read that right, the full program is written as a string)
- Abstract-Data-Type (Add(Int(2), Int(3)))
- quasiquotes (Meta-OCaml)

The LMS way is ... neither. It is based on a virtualized extension of Scala, DSL whose operations on lifted types creates an expression tree and deep reuse of the embedding language order. Those combined achieve a transparent and convenient meta-programming environment for the user because it is virtually no different from a non staged program. We will analyze those three components of LMS below.

1.3.1 Lifted types

A lifted type represents that same type at a future stage. For instance, the lifted type of an integer is a declaration that, once staged, this same value will represent an integer. So instead of manipulating an integer directly, you manipulate an "integer once staged".

Although similar in form those programs are different in nature:

This one is a standard program stating that a, b are integers and that c is their sum.

```
val a: Integer
val b: Integer
cal c: Integer = a + b
```

This one is a meta-program stating that a, b will be integers in the object program and that c in your object program should represent their sum.

```
val a: LiftedInteger
val b: LiftedInteger
val c: LiftedInger a + b
```

Notice that even though we use the same operator +, we necessary have to redefine it for LiftedInteger. But instead of actually summing integers, this new + operator will build the right Internal Represenation for this object-program operation.

1.3.2 Internal representation and the Exp tree

In order to manipulate meta-programs, it has to first convert the source code of a meta-program into a more convenient representation, easier to manipulate. This representation is called Internal Representation (IR). The IR of LMS is based on typed expression trees and single static assignments (SSA). This representation is a «sea of nodes» representation.

```
trait Exp[+T]
// Constant expression of type T
case class Const[T](x: T) extends Exp[T]
// Symbol referencing a definition of type T
case class Sym[T](id: Int) extends Exp[T]
// Composite node that is defined by a library or DSL author
trait Def[+T]
// Statement in the IR
case class TP[+T](sym: Sym[T], rhs: Def[T])
```

An assignment links a symbol to a definition. A definition is a composite node that represents an operation on other expressions (eg: Minus(e1:Exp[Int], e2:Exp[Int]) extends Def[Int])) and defines the result type. The expression tree in itself is a typed tree made of only two leaves: Constants and Symbols: «Wait, where are the nodes ?». The nodes are in fact the symbols, or more precisely, the composite node Def that the symbol represents (As stated by a TP). Constants are meta-programs values (eg: a meta-program "val a:LiftedInteger = 2" is represented in the IR as this constant expression: Const(2).

By using Scala pattern extractors and implicit conversions, we can reconstruct and manipulate this tree as if it was made only of Def and Const which is more natural.

For instance:

```
val a: Integer = ...
val b: Integer = ...
cal c: Integer = (a + b) - (a * b)
```

is represented as:

```
TP(Sym(0), ...)
TP(Sym(1), ...)
TP(Sym(2), Add(Sym(0), Sym(1)))
TP(Sym(3), Times(Sym(0), Sym(1)))
TP(Sym(4), Minus(Sym(2), Sym(3)))
```

But can be manipulated as:

```
Minus(Add(a, b), Times(a, b))
```

We will not describe further the IR since this project focus on the frontend. It will be clearer later what separates the backend from the frontend. But a simple informal definition is that the subset of scala that is available to the user to write meta-programs are the frontend. This include the lifted types interface and the building of the IR as a tree mode solely of the Def hybrid nodes. The IR representation as a sea of nodes, the IR manipulation such as transformers and traversals and code generation are the backend.

1.3.3 Lifted types and DSL operations

The previous frontend of LMS was representing lifted types as wrapped in a Rep monad. `LiftedInteger` would be written as`Rep[Int]`. The idea is that if A is the expected type in the object-program then we manpulate its Rep-resentation. This monad was the staging annotation.

The way to define operations on DSL was to define operators in scope that would manipulate the given Rep.

For instance, the DSL provided operations on Integers such as

```
def infix_+(e1: Rep[Int], e2:Rep[Int])
```

So as long as the corresponding methods or operators were in scope, the user was able to manipulate conveniently `Rep[A]` as if they were `A`.

1.3.4 Lift

The LMS solution to use Literals and other explicitly declared object is to use global conversion methods that know how to convert some present value to staged value. Their role is to convert `A` into `Rep[A]`. Those conversions are implicits and enable to write such declaration `val a: Rep[Int] = 2`. What is really happening is the scala solve the implicit lift conversion and it becomes `val a: Rep[Int] =`

`lift(2)(intLift)` with `intLift` being an instance of `Lift[Int,Rep[Int]]`. The sole purpose of `Lift` instance is to lift values into lifted types.

1.4 Scala virtualization

In order to enable the if-then-else/loops and other control, we use a modified version of the Scala language: scala-virtualized. Scala-virtualized enable to overload the controls as common functions: eg: 'if (t1) t1 else t2 becomes' '___ifThenElse(t1, t2, t3)'. We can use this overloading to extend the controls to lifted type.

1.5 Smart constructors

Smart constructors are optimised constructors of composite def that can apply optimisations based solely on the argumen of the constructor. For instance, for the constructor of IntTimes which represents multiplication of integer we can potentially apply some early reductions.

```
override def int_times(e1: Exp[scala.Int], e2: Exp[scala.Int])
: Exp[scala.Int] = (e1, e2) match {
  case (Const(0), r) => Const(0)
  case (l, Const(0)) => Const(0)
  case (Const(1), r) => r
  case (l, Const(1)) => l
  case (Const(x), Const(y)) => Const(x*y)
  case _ => IntTimes(e1, e2)
}
```

1.5.1 Deep reuse of the evaluation order

Deep reuse of the embedding language order is simpler than it sounds. The embedding language is Scala. The transformations on Expressions are such that they depend on the order of evaluation of the operations applied to the lifted types.

For instance:

```
val a: Rep[Int] = 2 //Const(2)
val b: Rep[Int] = 3 //Const(3)

//IntAdd(Const(2), Const(3))
val c: Rep[Int] = 2 + 3
//IntAdd(IntAdd(Const(2), Const(3)), Const(4))
val d: Rep[Int] = c + 4
```

Contrary to plain Scala, `a`, `b`, `c`, `d` here are not «simple» values. They are each a representation of a tree of `Exp`. `a` and `b` are trivial trees of one node: A constant leaf. However, `c` and `d` start to becomes more complex. The actual content of each tree depend of the evaluation order of the frontend operations by Scala. At code generation, the tree order is conserved through let bindings. Nevertheless, between the tree construction and code generation, some transformation might have changed the tree. Common Subexpression Elimination, Code motion, Loop Unrolling, Dead Code Elimination, Loop Fusion and more are among such potential transformations. Transformations are written in LMS as subtype of `Transformer`.

1.6 DSL as libraries

LMS enables meta-programming and doesn't necessarily require the use of DSL. Nevertheless, a powerful way to use LMS is to provide DSL written on top of LMS that includes all the necessary lifted types and `Transformer` to build application from. The `common/` part of LMS are a regroupment of lifted types and `Transformer` that should be needed in most cases. It includes, if-then-else, primitive types, and useful transformers such as CSE and code motion. On top of that, libraries author can add their own lifted types and transformer to provide a new « flavor » of LMS that form a DSL.

1.7 Delite

Delite is a research project from Stanford University's Pervasive Parallelism Laboratory (PPL). Delite is built on top of LMS and could be seen as an alternative of `common/` targeted for high-performance parrallel DSL.

1.8 User

Users are writers of meta-programs using a DSL or even just bare LMS.

2. The new frontend

LMS is reimplemented with a new frontend. One goal of this reimplementation is to achieve sufficient feature parity with the previous version of LMS to enable to implement meta-programs in the computation graph domain as explained further in part 3.

The new frontend change the type of lifted type from Rep monads to a Rep context bound. A context bound in Scala is used in the typeclass pattern.

2.1 Typeclass pattern

The typeclass patterns is an alternative to inheritance to write programs that can handle any type that share a common interface. With subtyping, we declare that if A is a child of B (A <: B), then any instance of A can be treated as B (A is a B). Then we can write functions that are polymorphic to the common interface of B.

With the typeclass pattern, we declare that as long as there exists a typeclass instance for the right parametrized type, we can write our polymorphic function.

```
//inheritance
trait Num[A] {
    def *(y: A): A
    ...
}

class Int extends Num[Int]{
```

```
    def *(y: Int): Int = ...
}

def square[A](x: Num[A]) = x*x
```

```
//typeclass pattern
trait Num[A] {
    def times(x: A, y: A): A
    ...
}

class Int {
    def *(y:Int): Int = ...
    ...
}

implicit object numInt extends Num[Int] {
    def times(x: Int, y: Int) = x*y
}

def square[A: Num](x: A) = {
    val num = implicitly[Num[A]]
    num.times(x, x)
}
```

The idea behind the new frontend is to use that typeclass pattern as staging annotation instead of the Rep monad.

In the previous LMS, we manipulated lifted types Rep[A] and define functions specific function for that Rep[A]. In the new LMS, we manipulate dsl.A that have Rep typeclass instances defined in scope.

```
trait Rep[A]

case class Int(e: Exp[scala.Int]) {
    def *(y: Int) = IntTimes(e, y.e)
}

implicit object intRep extends Rep[Int]
```

```
def ifThenElse[A: Rep](a: dsl.Boolean, b: A, c: A): A
```

2.2 Type shadowing

The convenient benefit of using dsl.A is that once the dsl is imported, we can shadow the type A by dsl.A. For instance, dsl.Int shadows scala.Int. This is specially useful to write meta-programs for which the lifted type of A, dsl.A, is the most common meaning of the type A.

2.3 Typeclass overloading

One of the issue encountered with the new frontend of LMS is «typeclass overloading». In the previous LMS, equals could be defined in such way:

```
    def __equal[A, B](a: Rep[A], b:Rep[B]): Rep[Boolean]
    def __equal[A, B](a: Rep[A], b:B): Rep[Boolean]
    def __equal[A, B](a: A, b:Rep[B]): Rep[Boolean]
    def __equal[A, B](a: A, b:B): Boolean
```

The scala choosed the right dispatch based on the most specialized definition of __equal. Note that `def __equal[A, B](a: A, b:B)` is a more general function than all above but only get triggered if none of the above case do not trigger.

Now with the typeclass pattern we could attempt such rewrite:

```
    def __equal[A:Rep, B:Rep](a: A, b:B): Boolean
    def __equal[A:Rep, B](a: A, b:B): Boolean
    def __equal[A, B:Rep](a: A, b:B): Boolean
    def __equal[A, B](a: A, b:B): scala.Boolean
```

The issue is that context bound are only syntaxic sugar and those functions

are eventually desugared into:

```
def __equal[A, B](a: A, b:B)
(implicit repA: Rep[A], implicit repB: Rep[B]): Boolean
def __equal[A, B](a: A, b:B)(implicit repA: Rep[A]): Boolean
def __equal[A, B](a: A, b:B)(implicit repB: Rep[B]): Boolean
def __equal[A, B](a: A, b:B): scala.Boolean
```

The issue is that in this curried form, none of the first three definitions of ___equal are more specialized than the last one. Hence, it is impossible to overload equal in this manner.

2.4 Primitives types and collections

The new LMS frontend required to adapt the whole `internal/` to the new typeclass pattern but it also required to write primitive types and collections in a modified manner. We implementend the primitive types: Int, Float, Double, Long as well as String, Boolean and the collections Array, Matrix and List. We also implemented staged lambda functions, staged named functions, staged if-then-else blocks.

Lifted types are organized into three or more traits:

- As extends Base
- AsExp extends BaseExp
- AsImpl extends AsExp
- AsOptImpl extends AsImpl

3. Computation Graph

Computation graphs are directed graph made of nodes that represent a computation. It is an abstract model that can generalize many functions. It is very common in distributed computing and it is how most deep learning (TensorFlow, Deeplearning4j) represents their neural networks models.

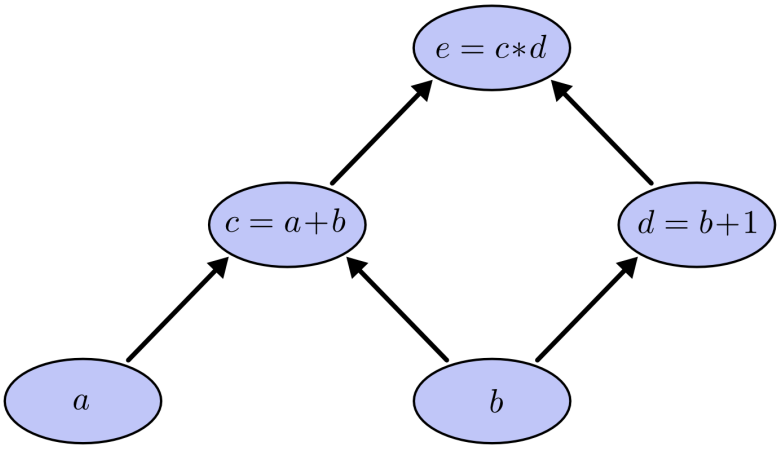


Figure 3.1: Example of a simple arithmethic computation graph

Each node has 0 or more inputs and 0 or n outputs and 1 operation. Having 0 inputs is the special case of Input nodes and having 0 output is the special case of output nodes. The input nodes form the input layer. The output nodes form the output layer. We will only consider feedforward computation graphs without any cycles. An example of a node can be the Add node. It takes two entry and output their sum.

The benefits of staging for Computation Graph is that there it would be advantageous to separate the construction of the graph and the execution of

computation on this graph. Indeed, it is possible to check some properties on the computation graph that ensure that the graph has proper form. Furthermore, it is possible to apply transformation to the graph such as computing the gradient of the function represented by the graph. Last but not least, computation graph are abstractions that are convenient to build, conceptualize and handle but not very efficient because of the level of abstraction. Fortunately, after staging unecessary indirections are removed and the whole function is linearized into the bare required operations.

```
trait Graphs {

  type Data
  type Input = List[Data]
  type Output = Data
  type G <: Graph
  type N <: Node

  trait Node {
      def output(input: Input): Output
  }


  ...

  trait Graph {

    //check if there is cycle in the graph
    checkCycle()
    def inputSize: Int
    type R = (N, List[String])
    def nodes: Map[String, R]
    def apply(input: List[Data]) = { ... }
    def checkCycle() = { ...  }
    def forward(input: List[Data]): (Data, Map[String, Data]) = { ... }

}


trait DerivableGraphs extends Graphs {

  def zero: Data
```

```
  def one: Data

  type Data <: AddTimeAble[Data]
  type Derivatives = List[Data]
  type N = DerivableNode
  type G = DerivableGraph
  ...
}
```

One of the clear benefits of using types as staged annotations is the ability to share codes between non staged programs and meta-programs. Indeed, the graph can set the type of data it manipulates as an abstract type member. Then depending on the type of graph and the type of nodes the graph may contain, this type can be upper bounded by the appropriate interface.

Below the common forward evaluation of the graph implemented using memoization and recusion:

```
    def forward(input: IndexedSeq[Data], dbg:Boolean = false) = {

      //init datas with provided input as input Node
      var datas: Map[String, Data] =
        input.zipWithIndex map { case (data, ind) => ("IN"+(ind+1), data) } toMap

      def output(s: String): Data = {
        val (n, inp) = nodes(s)
        if (datas.contains(s))
          datas(s)
        else {
          val l = inp.map(output)
          val r = n.output(l)
          datas += ((s, r))
          r
        }
      }

      output("OUT")

    }
```

`AddTimeAble[Data]` guarantees that Data has the operations `+` and `*` necessary for derivations.

Here Data can either be either `dsl.Int` for a staged computation graph or else a simple `scala.Int` (or at least a wrapper that implements AddTimeAble)

3.1 Cycle check

A staged computation graph builds the graph during staging. This means that all the verification that should happen during the building of the graph can happen during staging. This adds additional guarantees at runtime, here that our staged computation graph does not contain any cycle.

3.2 Arithmetic

We implement a graph with each node being a basic arithmetic operation (+,*,-,%,min,max) and a data type that is upper bounded by a basic arithmetic operation interface such that the one of Int, Double and Float implement. Those kind of computation graph are a good sanity check as well as a clear example of a compuation graph. We will use them to do benchmarking of their evaluation time perfomance as a staged meta-program and as a normal program. We do not take into account the time the efficiency of building those graphs and do the safety checks (like cycle check) because we want to measure efficiency in a "build once, evaluate often" context. We also add Constant Node that represent some fixed weights inside the graph.

3.2.1 Benchmark

For benchmarking purposes, we will randomly generate 2000 nodes big graph. The graph are build in a way that they stay balanced: each node is at most input of only 1 more node than any other node. We use Int as the Data type.

We compare the non-staged computation graph to a meta-program that doesn't benefit from the optimised implemenation of Int. The optimisd implementation of Int differs from the non optimised optimisation of Int by the usage smart-constructors that can optimize some operation such as multiplication with 0 or 1, or addition with 0 or binary operation on constants.

We build 100 different graph and average the evaluation time to achieve meaninful results. Benchmarks are run on a thinkpad t440s:

Average evalation time:

- non-staged program: 2156 microseconds
- staged program: 171 microseconds.

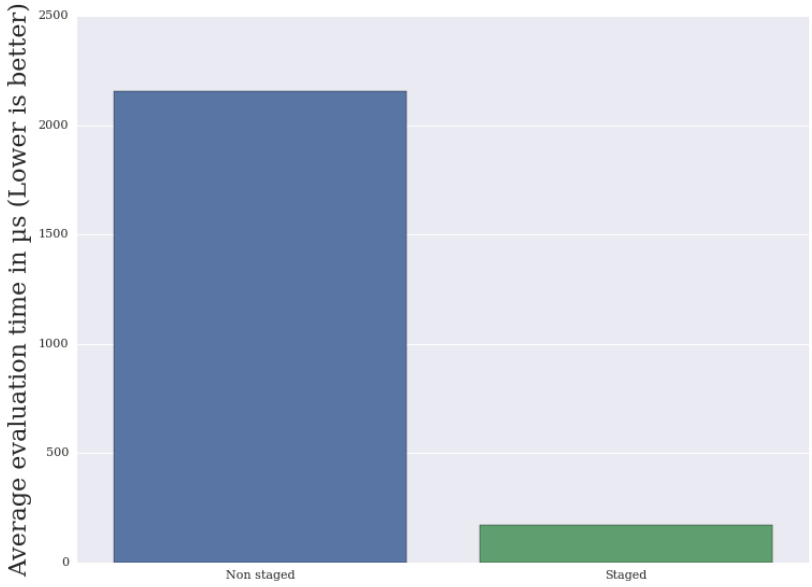


Figure 3.2: Performance chart

3.3 DerivableGraph

We also implement backpropagation in both meta-programs and common programs. Backpropagation is an algorithm that enable fast computation of all partial deriva-

tives in a computation graph. our backpropagation algorithm is common for both environment. This shows the flexibility of LMS.

3.4 MatrixGraph

Last but not least, we implement computation graphs able to handle Matrix as Data. Matrix as data is common in computation graph of neural networks and machine learning models. One interesting feature that we can achieve is to check the correct dimensions of the matrix between nodes. This can be problematic if only checked at runtime but fortunately, in a staged environment we can check the appropriate dimensions during staging.

Conclusion

References

- Tiark's thesis: Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming
- Calculus on Computational Graphs: Backpropagation

...

Acknowledgement

Thanks to my beloved parents, my awesome supervisor Nada Amin, Prof. Martin Odersky, the lms master and author Tiark Rompf, and the delite folks Kevin James Brown and David Koeplinger.