New lms frontend and staged computation graphs

Ruben Fiszel

December 7, 2016

~ *Supervised by Nada Amin and Prof. Martin Odersky* ~

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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 constraint 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 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.

Nevertheless, growing complexity in compiler is not the panacea. In a wide range of programs, compilers are limited by the lack of domain specific knowledge of the program. Indeed, constraining a program to one specific domain open the door for a wide range of specific optimization. Furthermore, code generation by “macro” is a rather poor way to enable the further abstraction brought by code that generate code.

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 DSL in a staged environment. It is the latter that is explored with LMS, a scala library for runtime code generation. In this report, we will explore a new lms implementation that brings a 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.

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. The generation can go through multiple stage. Each stage is another compilation time that can leverage new informations to build the final program.

One clear example of this is sparse vector multiplication.

```
def rd(i: Int) = util.Random.nextInt(i)
def newVec(size: Int) = {
    val a = Array.fill(size)(0)
    for (i <- (1 to 10))
        a(rd(size)) = rd(100)
    a
}
val v1 = newVec(1000)
val v2 = newVec(1000)
```

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

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

Staging

Why staging

Exp tree

Deep reuse of embedding language order

Frontend/Backend ?

lms

library author

delite

user

The new frontend

Lift

Typeclass

Typeclass overloading

Primitives types and collections

Computation Graph

Graph

Cycle check

Arithmetic

Benchmark

DerivableGraph

Backpropagation

MatrixGraph

Conclusion

blablabla

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.