Concepts of Programming Design

Scala and Lightweight Modular Staging (LMS)

Alexey Rodriguez Blanter, Ferenc Balla, Matthijs Steen,
Ruben Meerkerk, Ratih Ngestrini

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Scala and Lightweight Modular Staging (LMS)

Recap of Scala - The Scalable Language

First, a short recap which language Scala was again:

- Statically typed
  - Rich type system similar to Haskell and ML
- Hybrid object-oriented / functional programming language
  - It is object-oriented
    - It has the usual stuff found in Java
    - Can be extended using traits
  - It is functional
    - Has immutable variables and objects: val keyword and case classes
    - It makes everything an expression, e.g. obviating the need for a ternary operator: `cond-expr ? then-expr : else-expr`

Recap of Scala - The Scalable Language (continued)

- Built upon the JVM
  - Great language interoperability with Java
- Syntax sugar to improve the programming experience
- It is strict by default, but has built-in support for non-strict evaluation
- Macros

Different from Java

- Unified interface for value types (primitives in Java) and reference types (objects)
- Type inference
- Functions are first class citizens
- Nested Functions
- Better DSL support
- Traits

Traits

- Encapsulates method and field definitions
- Can be reused by mixing them into classes
- A class can use any number of traits (unlike class inheritance)

Generative Programming

- High-level language
- Writing programs that write programs (meta)
- Conversion to efficient (optimized) tasks
- Useful for DSLs
- Reduces programming time
- Becoming more popular
  - IT departments for productivity increase
  - End users for programming time

Generative Programming - Compiler

- Takes away lots of tasks from programmer
- Instruction reduction at current level
- Intermediate code representation to be further optimized
- Instruction selection from lower level
- Register allocation
- Garbage collection

Generative Programming versus Template metaprogramming

- Temporary source code generated, temporarily combined with the original source code
- Runs at compile-time like general generative programming
- Contains only immutable variables
- Used for optimization as well

Generative Programming versus Macros

- Overlap between the 2 techniques
- Often same level code generation
- Can be used as building blocks for generative programming
- Same resulting ease of use

What is LMS and why should we care?

Multi-Stage Programming (MSP)

- Multi-stage programming (MSP) is a paradigm for developing generic software.

Multi-Stage Programming (MSP)

- Is an instance of Generative Programming
- Can be used for implementing DSL's
- Divides the compilation of a program in phases
- Allows programmers to write generic/abstract programs, without a runtime penalty
- Ensures type safe programs

Lightweight Modular Staging (LMS)

- Lightweight: purely library based
- Modular: features can be mixed and matched
- Staging: uses the MSP staging technique

Just add another level of indirection!

A famous quote, attributed to David Wheeler, says that:

> *Any problem in computer science can be solved by adding a level of indirection.*

Less known is the second part:

> *Except problems that are caused by too many levels of indirection.*

The problems spoken of in the second part will be mostly performance problems, causing programming language to choose between a balance between **performance** or **productivity**.

Abstraction without regret

This balance is necessary, because

- If you **add an indirection** to **increase the productivity** you are likely to **decrease the performance**.
- If you **remove an indirection** to **increase the performance** you are likely to **decrease the productivity**.

The LMS library solves this issue by allowing the programmer to write programs with all the indirections present (max. productivity), but subsequently removes them from the final program via staging (max. performance).

LMS power function

Scala:

```
def power(b: Double, p: Int): Double =
  if (p == 0) 1.0 else b * power(b, p - 1)
```

Scala with LMS:

```
def power(b: Rep[Double], p: Int): Rep[Double] =
  if (p == 0) 1.0 else b * power(b, p - 1)
```

LMS power function

```
power(x,5)
```

```
def apply(x1: Double): Double = {
  val x2 = x1 * x1
  val x3 = x1 * x2
  val x4 = x1 * x3
  val x5 = x1 * x4
  x5
}
```

LMS (optimized) power function

```
def power(b: Rep[Double], p: Int): Rep[Double] = {
  def loop(x: Rep[Double], ac: Rep[Double], y: Int): Rep[Double] = {
    if (y == 0)
      ac
    else if (y % 2 == 0)
      loop(x * x, ac, y / 2)
    else
      loop(x, ac * x, y - 1)
  }
  loop(b, 1.0, p)
}
```

LMS (optimized) power function

```
power(x,11)
```

```
def apply(x1: Double): Double = {
  val x2 = x1 * x1  //x * x
  val x3 = x1 * x2  //ac * x
  val x4 = x2 * x2  //x * x
  val x8 = x4 * x4  //x * x
  val x11 = x3 * x8 //ac * x
  x11
}
```

Scala-Virtualized

Scala-Virtualized is a branch of the Scala compiler and standard library that provides better support for embedded DSL's.

Key features:

- overloadable control structures, variables, object creation, etc
- extension methods: define new infix-methods on existing types (pimp-my-library with less boilerplate)

LMS-Core

LMS-core is a framework providing a library of core components for building high performance code generators and embedded compilers in Scala.

- smart-constructor representations for constants, loops, conditions, etc.
- Scala-virtualized definitions for their syntax
- an internal AST for a code generation framework

How does LMS work?

How LMS works

- `Rep[T]` represents a delayed computation of type `T`
- During staging, an expression of type `Rep[T]` becomes part of the generated code, while an expression of bare type `T` becomes a constant in the generated code
- Abstractions are not part of the generated code when their type is a `T` as opposed of a `Rep[T]`.

Example

```
def snippet(x: Rep[Int]) = {
  def compute(b: Boolean): Rep[Int] = {
    // the if is executed in the first stage
    if (b) 1 else x
  }
  compute(true) + compute(1 == 1)
}
```

Turns into…

```
/*****************************************
Emitting Generated Code
*****************************************/
class Snippet extends ((Int)=>(Int)) {
  def apply(x1:Int): Int = {
    2
  }
}
/*****************************************
End of Generated Code
*****************************************/
```

Constant. As opposed to:

With Rep type:

```
def snippet(x: Rep[Int]) = {
  def compute(b: Rep[Boolean]): Rep[Int] = {
    // the if is deferred to the second stage
    if (b) 1 else x
  }
  compute(x == 1)
}
```

```
/*****************************************
Emitting Generated Code
*****************************************/
class Snippet extends ((Int)=>(Int)) {
  def apply(x3:Int): Int = {
    val x4 = x3 == 1
    val x5 = if (x4) {
      1
    } else {
      x3
    }
    x5
  }
}
/*****************************************
End of Generated Code
*****************************************/
```

Regular expressions example

- LMS generates code for the interpreter specialized to a particular program (e.g. checking strings against regexp)
- The program is fixed at staging time, while the input(s) to the program may vary in the generated code.
- To make it work: wrap the types of expressions that vary in `Rep[_]` while leaving the types of expressions we want specialized as is.
- Thus, the staged reg.exp. matcher:
  - Is invoked on a static regular expression of type String and a dynamic input string of type `Rep[String]`.
  - Generates code specialized to match any input string against the constant regular expression pattern.

Unstaged

```
def matchsearch(regexp: String, text: String): Boolean = {
  if (regexp(0) == '^')
    matchhere(regexp, 1, text, 0)
  else {
    var start = -1
    var found = false
    while (!found && start < text.length) {
      start += 1
      found = matchhere(regexp, 0, text, start)
    }
    found
  }
}
```

Staged

```
def matchsearch(regexp: String, text: Rep[String]): Rep[Boolean] = {
  if (regexp(0) == '^')
    matchhere(regexp, 1, text, 0)
  else {
    var start = -1
    var found = false
    while (!found && start < text.length) {
      start += 1
      found = matchhere(regexp, 0, text, start)
    }
    found
  }
}
```

Generated code for a fixed regular expression (^ab):

```
class Snippet extends ((java.lang.String)=>(Boolean)) {
  // reg.exp. is not an input argument anymore!
  // input string changed its type from Rep[String] to String
  def apply(x0:java.lang.String): Boolean = {
    val x1 = x0.length
    val x2 = 0 < x1
    val x5 = if (x2) {
      val x3 = x0(0)
      val x4 = 'a' == x3
      x4
    } else {
      false
    }
    ...
  }
}
```

Why the approach taken by LMS?

Futamura projections

Futamura projections are a particulary interesting application of partial evaluation that given a `specializer` and an `interpreter` allow you to generate a `executable`, `compiler`, and a `compiler generator`.

Partial evaluation is the optimization of a program by specializing it with already known inputs. Most often used to speed up the execution time of a program.

```
specializer :: (Input a -> Input b -> Output Normal) -> Input a
  -> (Input b -> Output (SpecializedWith (Input a)))
```

First Futamura projection

Specializing an `interpreter` with `source code`, resulting in an `executable`.

```
executable :: Input -> Output
executable = specializer interpreter sourceCode where
  interpreter :: SourceCode -> Input -> Output
```

It is like a compiler!

Applying the specializer with these arguments makes it act like a compiler:

```
type Executable = Input -> Output

compiler :: SourceCode -> Executable
compiler sourceCode = specializer interpreter sourceCode where
  interpreter :: SourceCode -> Input -> Output
```

So is there is no need for a compiler, having a specializer and an interpreter can be enough to achieve the same.

It can even be more practical than a compiler!

This is not just some **theoretical** application, there are **practical** applications for using a specializer like this.

For example you might want to write a specializer instead of a compiler when targeting a architecture that takes a lot of effort to write a compiler for.

By writing a specializer that incorporates this knowledge, you will allow others to only have to supply their interpreters instead of each having to write their own compiler.

In general writing an interpreter is simpler and this way not everybody has to have to have a deep understanding of the target architecture.

Second Futamura projection

Specializing the `specializer` with an `interpreter`, resulting in a `compiler`.

```
type Executable = Input -> Output

compiler :: SourceCode -> Executable
compiler = specializer specializer interpreter where
  interpreter :: SourceCode -> Input -> Output
```

This will result in an actual compiler, it will not still call the interpreter at some point. The performance of the compiler will mostly depend on how good the specializer is.

Third Futamura projection

Specializing the `specializer` with the `specializer` itself, resulting in a `compiler generator`.

```
type Interpreter = SourceCode -> Input -> Output
type Executable = Input -> Output

compilerGenerator :: Interpreter -> Compiler
compilerGenerator = specializer specializer specializer where
  compiler :: SourceCode -> Executable
```

Giving the resulting compiler generator the specializer as the compiler to specialize will result in the compiler generator itself, making it a **quine**, a non-empty program with no arguments that results in a copy of its own source code.

```
compilerGenerator specializer == compilerGenerator
```

How do the Futamura projections apply to LMS?

The LMS library is a specializer, hence we should at least be able to achieve the **first** Futamura projection: specializing an `interpreter` with `source code`, resulting in an `executable`.

We actually have seen an example of this with the regexp example. We had an `interpreter` (regexp engine) and `source code` (regexp string), which resulted in program that still expected some `input` (regexp input) to produce an `output` (regexp match).

```
compiledRegexp = LMS.specializer regexpEngine regexpString
regexpMatch = compiledRegexp regexpInput
```

How about the other Futamura projections?

The LMS library is a manual specializer, i.e. the programmer has to manually guide the specializer. However the **second** Futamura projection needs an automatic specializer to be able specialize itself: specializing the `specializer` with an `interpreter`, resulting in a `compiler`.

So the second Futamura projection **cannot** be achieved with LMS.

What are the problems with the Futamura projections?

The **second** and **third** Futamura projections are not that useful in practice.

The performance of the generated `compiler` will depend on the optimizations built within the `specializer`. And how the `interpreter` has been written can greatly affect how the code will be specialized.

This makes the specialization process a **leaky abstraction**, i.e. implementation details of an abstraction become exposed.

Examples

- It is undeterministic whether partially evaluating something is beneficial or not.
  - Inlining function definitions
- There are few programs that can be automatically specialized to be of comparible quality as a hand-optimized program, e.g. no optimized runtime data structures in the case of a compiler.
  - Manual strictness annotations in Haskell
- In **theory** the programmer does not have to get familiar with the result of specializing something, in **practice** the programmer does.
  - Compile-to-JavaScript programming languages
- Even worse, to get the intended result, the programmer might also need to get in-depth knowledge about the specializer itself.
  - Parser generators

Good enough automatic specializer

Having a good enough automatic specializer for which the above examples would not be a problem would be akin to writing a **sufficiently-smart compiler**.

Sufficiently-smart compiler

A **sufficiently-smart compiler** is a compiler that is somehow able to transform an unoptimized program to the level of a program optimized by a programmer.

These compilers are considered hypothetical for the simple reason that the information necessary to make some of the optimizations done by the programmer cannot be inferred by the compiler, e.g. knowledge about problem domain.

However there exist some compilers that sometimes can be considered sufficiently smart, at least for those programs that do not require special knowledge restricted to the programmer. Examples of these are GHC (Haskell), JVM (e.g. Java), and .NET (e.g. C#).

An alternative to a sufficiently smart compiler

LMS is a pragmatic alternative to a sufficiently-smart compiler.

- It is a multi-pass compiler, because each time you go to a lower level, you lose information in the process. Multiple lowerings allow for multiple oppertunities for optimizations.
- It encourages the programmers to embed knowledge about the program in the form of domain specific languages.
- Because the knowledge of the programmer has been applied to the program in the different stages, the resulting program will be one of those programs that do not require special knowledge, the kind the JVM is very well equiped to optimize.

More than just staging

The LMS library is more than just a multi-stage compiler, it also includes many advanced analyses and optimizations.

- Dead code analysis
- Constant propagation
- Loop fusion
- Loop hoisting
- Many others…

Example

To get the best results for their analyses it employs a optimistic strategy:

```
var x = 7
var c = 0
while (c < 10) {
  if (x < 10) print("!")
  else x = c
  print(x)
  print(c)
  c += 1
}
```

```
var x = 7
var c = 0
while (c < 10) {
  print("!")

  print(7)
  print(0)
  c = 1
}
```

```
var c = 0
while (c < 10) {
  print("!")

  print(7)
  print(0)
  c += 1
}
```

Real-world applications and related work of LMS

Applications of LMS

- Delite (Stanford):

A compiler framework and runtime for parallel embedded DSLs. The goal is to enable the rapid construction of high performance, highly productive DSLs (OptiML: a DSL for machine learning, OptiQL: a DSL for data querying, OptiGraph: a DSL for graph analytic, etc).

- Spiral (ETH Zurich):

A program generator for linear transforms (Discrete Fourier Transform, Discrete Cosine Transforms) and other mathematical (numeric) functions.

- LegoBase (EPFL DATA):

A query engine (Databases and query processing). LegoBase builds query engine in high-level language with performance of hand-written low-level C.

- Lancet (Oracle Labs):

Integrate LMS with JVM / Just-in-time (JIT) compilation. LMS produces a simple bytecode compiler, adds abstract interpretation to turn the simple compiler into an optimizing compiler.

- Hardware (EPFL PAL):

Developing domain-specific High-Level Synthesis (HLS) tools (to design hardware) by embedding its input DSL in Scala and using LMS to perform optimizations.

- Js-Scala (EPFL, INRIA Rennes):

A Scala library providing composable JavaScript code generators as embedded DSLs.

Related Works

Other Meta-programming approaches

- Static: C++ templates, Template Haskell
- Customizable: Expression Templates
- Runtime: TaskGraph framework (C++)

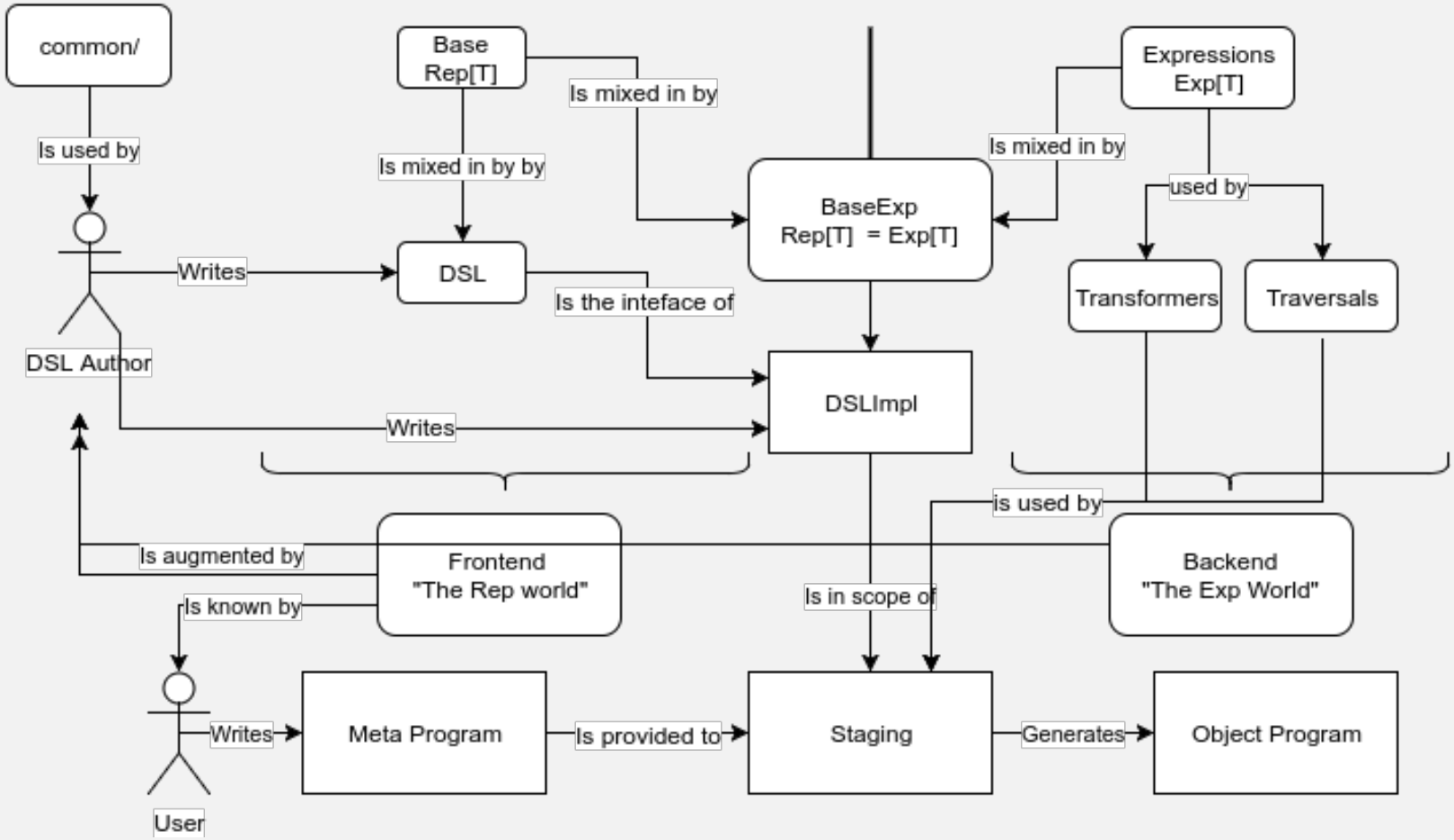
Multi-Stage Programming Languages

- MetaML
- MetaOCaml
- Mint

Compiled embedded DSLs

- DSLs implementation in metaocaml, template haskell, and C++
- DSLs implementation using program staging and monadic abstraction

Recap



(Source: LMS org)

Summary

- LMS is a library-based dynamic code generation approach
- LMS programmatically removes abstraction overhead through staging
- LMS builds an intermediate representation (IR)
- Uses a special Scala compiler (Scala-virtualized)

Thank you for your attention!

Any Questions?

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]