Scala

Lightweight Modular Staging

Steven Both, Toby Rufinus, Jaspreet Singh, Daniël Stekelenburg

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Now that we are familiar with Scala…

Now that we are familiar with Scala…

lets look at an awesome library implemented in Scala.

Now that we are familiar with Scala…

lets look at an awesome library implemented in Scala.

Lightweight Modular Staging (LMS)

LMS is..

‘A library-based multi-stage programming approach that uses types to distinguish between binding time.’

Outline

- A gentle introduction to LMS
- Generative Programming
- How to write a multi-staged program
- Language virtualization
- Intermediate representation
- How do we program in LMS?

A gentle introduction to LMS

Power function in Scala

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

Power function in Scala LMS

```
trait PowerA { this: Arith =>
  def power(b: Rep[Double], x: Int): Rep[Double] =
    if (x == 0) 1.0 else b * power(b, x - 1)
}
```

A gentle introduction to LMS

What did we just see?

- `T` versus `Rep[T]`
- `def` versus `trait`

Productivity vs Performance

Software performance depends more on programmer productivity

- Processor clock speed doesn't double every 18 months
- High-level programming is hard to translate to efficient code
- Shift towards big data workloads

Result: Hand-optimized programs (BAD IDEA)

- abandoning all best practices and benefits of high-level programming
- programs become hard to read, maintain, verify…
- this attracts bugs and security vulnerabilities

Solution: Generative Programming

Write a program generator

- Produces the code of a program as output
- Reorganization of a programs' execution into stages, also called multi-stage programming

How do we write MSP-programs?

- A single-stage program is developed, implemented and tested
- Ensure the the program can be used in a staged manner. Otherwise “refactor”
- Introduce staging annotations

Another stage!

From

- Compilation-based program execution:
- Compile-time, run-time.

To

- Generated program execution:
- Generation-time, Compile-time, run-time.

Generative Programming

What is generative programming?

Commonalities of LMS and generative programming

Differences of LMS and generative programming

Drawbacks of LMS

- The Scala LMS library only implements staged operations for a subset of Scala.
- Debugging is painful - LMS can give obscure errors, run into an infinite loop, or generate wrong code.
- Documentation is lacking.

Intermediate representation

How to LMS:

- Staging
- Generating code
- Data types

How to LMS: Simple staging

Power function in Scala:

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

How to LMS: Simple staging

Staged power function in LMS:

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

How to LMS: Simple staging

Staged power function in LMS:

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

power(b, 3)
```

How to LMS: Simple staging

```
power(b, 3)
```

Generated code:

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

How to LMS: Staging with recursion

Simple factorial function

```
def fac(n: Rep[Int]): Rep[Int] = {
  if (n == 0) 1
  else n * fac(n - 1)
}
```

How to LMS: Staging with recursion

Simple factorial function

```
def fac(n: Rep[Int]): Rep[Int] = {
  if (n == 0) 1
  else n * fac(n - 1)
}

fac(n)
```

How to LMS: Staging with recursion

Simple factorial function

```
def fac(n: Rep[Int]): Rep[Int] = {
  if (n == 0) 1
  else n * fac(n - 1)
}

fac(n)
```

```
...
[error] (run-main) java.lang.StackOverflowError
[error] (compile:run) Nonzero exit code: 1
...
```

How to LMS: Staging with recursion

```
power(b, 3)    vs.    fac(n)
```

How to LMS: Staging with recursion

```
power(b, 3)    vs.   fac(n)
```

Make use of a lambda function:

```
def fac: Rep[Int => Int] = doLambda { n =>
  if (n == 0) 1
  else n * fac(n-1)
}
```

Now we can try it again:

```
fac(n)
```

How to LMS: Staging with recursion

Generated code

```
def apply(x12:Int): Int = {
  var x1 = {x2: (Int) =>
    val x3 = x2 == 0
    val x8 = if (x3) { 1 }
    else {
      val x4 = x2 - 1
      val x5 = x1(x4)          // recursion
      val x6 = x2 * x5
      x6 }
    x8: Int }
  val x13 = x1(x12)            // recursion
  x13 }
```

How to LMS: Generating code

```
power(b, 3)
```

Generated code:

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

How to LMS: Generating code

Optimized power function

```
def powerOpt(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)
}
```

How to LMS: Generating code

```
powerOpt(b, 3)
```

Generated code:

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

How to LMS: Generating code

```
powerOpt(b, 3)
```

Generated code:

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

LMS can generate the same code from different staged codes.

How to LMS: Generating code

But not per se.

How to LMS: Generating code

But not per se.

For example:

```
powerOpt(b, 6)
```

How to LMS: Generating code

But not per se.

For example:

```
powerOpt(b, 6)
```

Generated code of optimized version

```
def apply(x4:Double): Double = {
  val x5 = x4 * x4
  val x6 = x5 * x5
  val x7 = x5 * x6
  x7
}
```

How to LMS: Generated code

Trivial regular expression

```
checkRegex(".", "Hello world")
```

The char '.' is a wildcard in Scala.

How to LMS: Generated code

Partial regex code

```
def matchStar(...): Rep[Boolean] = { ... }
def matchBegin(...): Rep[Boolean] = { ... }
def matchEnd(...): Rep[Boolean] = { ... }

def matchChar(c: Char, t: Rep[Char]): Rep[Boolean] =
{ c == '.' || c == t }
```

How to LMS: Generated code

Partial regex code

```
def matchStar(...): Rep[Boolean] = { ... }
def matchBegin(...): Rep[Boolean] = { ... }
def matchEnd(...): Rep[Boolean] = { ... }

def matchChar(c: Char, t: Rep[Char]): Rep[Boolean] =
{ c == '.' || c == t }
```

Now do:

```
checkRegex(".", s)
```

How to LMS: Generated code

```
...
val x47 = while ({ val x28 = x27
  val x34 = if (x28) { false }
  else { val x30 = x26
    val x32 = x30 < x31
    x32 } x34}) {
  val x36 = x26 += 1
  val x37 = x26
  val x38 = x37 < x31
  val x42 = if (x38) {
    val x39 = x25(x37)
    val x40 = '.' == x39    // matchChar(...)
    val x41 = true || x40   // c == '.' || c == t
    x41 }
  else { false }
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

How to LMS: Generated code

Generated code is not meant to be human-readable

How to LMS: Data types

The previous examples only considered:

- Rep[Int]
- Rep[Double]
- Rep[Char]
- Rep[Boolean]

How to LMS: Data types

The previous examples only considered:

- Rep[Int]
- Rep[Double]
- Rep[Char]
- Rep[Boolean]

But what to do for datatypes that are of your own making?

How to LMS: Data types

- Scala LMS only has pre-defined operations for standard library types.
- You add `Rep[T]` types to your own functions to stage them, allowing them to work on staged values.

How to LMS: Data types

Unstaged:

```
case class Vec3(x: Double, y: Double, z: Double) {
  def +(that: Vec3): Vec3 =
    Vec3(this.x + that.x,
      this.y + that.y, this.z + that.z)
}
```

How to LMS: Data types

Unstaged:

```
case class Vec3(x: Double, y: Double, z: Double) {
  def +(that: Vec3): Vec3 =
    Vec3(this.x + that.x,
      this.y + that.y, this.z + that.z)
}
```

Staged:

```
case class Vec3(x: Rep[Double], y: Rep[Double],
    z: Rep[Double]) {
  def +(that: Vec3): Vec3 =
    Vec3(this.x + that.x,
      this.y + that.y, this.z + that.z)
}
```

How to LMS: Data types

What about this function?

Unstaged:

```
case class Vec3(x: Double, y: Double, z: Double) {
  def length: Double =
    sqrt(x * x + y * y + z * z)
}
```

How to LMS: Data types

What about this function?

Unstaged:

```
case class Vec3(x: Double, y: Double, z: Double) {
  def length: Double =
    sqrt(x * x + y * y + z * z)
}
```

Staged:

```
case class Vec3(x: Rep[Double], y: Rep[Double],
   z: Rep[Double]) {
  def length: Double =
    sqrt(x * x + y * y + z * z)
}
```

A problem: undefined operations on staged types

```
[error] main.scala:9: not found: value sqrt
[error]     sqrt(x * x + y * y + z * z)
[error]     ^
[error] one error found
[error] (compile:compileIncremental)
Compilation failed
```

- Scala LMS defines some fundamental operations on staged types, such as integer/floating point arithmetic.
- If your staged function only uses those operations (like the + operator in the Vec3 example) you're fine.

How to Stage Your Algorithm

1. Add `Rep[T]` type annotations
2. Define an interface for new operations on staged types
3. Implement the interface in terms of IR nodes
4. (optional) Define optimizations, rewriting certain patterns of IR nodes
5. Extend code generator so new IR nodes can be turned into code

An example: Fast Fourier Transform

You can create a wave-like function by summing up a number of sine waves. The Fourier transform decomposes the waveform back into its sine components. The Fast Fourier Transform (FFT) is a fast numerical algorithm that can do this.

An example: Fast Fourier Transform

```
def fft(xs: Array[Complex]): Array[Complex]
= xs match {
  case (x :: Nil) => xs
  case _ =>
    val N = xs.length // assume it's a power of two
    val (even0, odd0) = splitEvenOdd(xs)
    val (even1, odd1) = (fft(even0), fft(odd0))
    val (even2, odd2) = (even1 zip odd1 zipWithIndex)
      case ((x, y), k) =>
        val z = omega(k, N) * y
        (x + z, x - z)
    }.unzip;
    even2 ::: odd2
}
```

An example: Fast Fourier Transform

```
case class Complex(re: Double, im: Double) {
  def +(that: Complex) = Complex(this.re + that.re,
    this.im + that.im)
  def *(that: Complex) = ...
}

def omega(k: Int, N: Int): Complex = {
    val kth = -2.0 * k * Math.Pi / N
    Complex(cos(kth), sin(kth))
}
```

Note the operations we perform on doubles: arithmetic (addition, multiplication, ...) and trigonometry (sin, cos)

An example: Fast Fourier Transform

Step 1: Add Rep[T] type annotations

Unstaged:

```
case class Complex(re: Double, im: Double) {
  def +(that: Complex) = Complex(this.re +
    that.re, this.im + that.im)
  def *(that: Complex) = ...
}
def omega ...
def fftt ...
```

An example: Fast Fourier Transform

Step 1: Add Rep[T] type annotations

```
trait FFT { this: Arith with Trig =>
  case class Complex(re: Rep[Double], im:
    Rep[Double]) {
    def + Complex(this.re + that.re,
      this.im + that.im)
    def * ...
  }
  def omega ...
  def fft ...
}
```

An example: Fast Fourier Transform

Step 1: Add Rep[T] type annotations

The staged version of complex numbers consists of a pair of `Rep[Double]` We need to be able to do arithmetic and trigonometric operations on staged doubles. Fortunately, the Scala LMS library happens to define operations for `Rep[Double]`

An example: Fast Fourier Transform

Step 1: Add Rep[T] type annotations

```
trait FFT { this: Arith with Trig =>
  case class Complex(re: Rep[Double], im:
    Rep[Double]) {
    def + Complex(this.re + that.re,
      this.im + that.im)
    def * ...
  }
  def omega ...
  def fft ...
}
```

An example: Fast Fourier Transform

Step 1: Add Rep[T] type annotations

```
trait FFT { this: Arith with Trig =>
  case class Complex(re: Rep[Double],
    im: Rep[Double]) {
    def + Complex(this.re + that.re,
      this.im + that.im)
    def * ...
  }
  def omega ...
  def fft ...
}
```

- How to define our own staged operations that work on `Rep[Double]`?
- We define these operations in traits and mix them in.
- The `this: Arith with Trig` part means: whenever the trait `FFT` is instantiated, we need to mix in traits that provide arithmetic and trigonometric operations too.

An example: Fast Fourier Transform

Step 2: Define an interface for new operations on staged types

Scala LMS provides a `Base` trait; the `Rep[T]` type is defined there.

```
trait Arith extends Base {
  def infix_+(x: Rep[Double], y:
      Rep[Double]): Rep[Double]
  def infix_*(x: Rep[Double], y:
      Rep[Double]): Rep[Double]
  ...
}

trait Trig extends Base {
  def cos(x: Rep[Double]): Rep[Double]
  def sin(x: Rep[Double]): Rep[Double]
}
```

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

These traits contain only abstract members; they are interfaces. We need to create subclasses with concrete implementations.

An example: Fast Fourier Transform

Step 3: Implement the interface in terms of IR nodes

- Scala LMS uses a node-based intermediate representation for staged expressions.
- The `BaseExp` class from the LMS framework defines some related types.
- `Exp[T]` is a simple IR expression (a constant or symbol).
- `Def[T]` is a composite operation; these operations will be converted to simple expressions.
- `BaseExp` also defines `Rep[T] = Exp[T]` so staged expressions will be converted to IR expressions.

An example: Fast Fourier Transform

Step 3: Implement the interface in terms of IR nodes

```
trait ArithExp extends Arith with BaseExp {
  // These case classes are IR nodes
  case class Plus(x: Exp[Double], y: Exp[Double])
    extends Def[Double]
  case class Times(x: Exp[Double], y: Exp[Double])
    extends Def[Double]

  // The abstract functions defined in trait Arith ar
  def infix_+(x: Exp[Double], y: Exp[Double]) =
    Plus(x, y)
  def infix_*(x: Exp[Double], y: Exp[Double]) =
    Times(x, y)
}
```

An example: Fast Fourier Transform

Step 3: Implement the interface in terms of IR nodes

```
sin(x + 2 * y) + sin(0)
```

```
Plus(Sin(Plus(Sym(x),
Times(Const(2),
Sym(y)))),
Sin(Const(0)))
```

An example: Fast Fourier Transform

Step 4: Define optimizations, rewriting certain patterns of IR nodes

- Scala LMS already contains a number of generic optimizations, such as dead code elimination and reusing identical expressions.
- We can define our own optimizations, both generic optimizations and domain-specific ones.
- These are again defined in traits, so you can combine them in a modular way.

```
trait ArithExpOpt extends ArithExp {
  override def infix_*(x: Exp[Double],
    y: Exp[Double]) =
    (x, y) match {
      // Multiplying two constants? We can calculate
      case (Const(x), Const(y)) => Const(x * y)
      // 1 * x = x, and vice versa
      case (x, Const(1)) => x
      case (Const(1), x) => x
      // Base case: apply the regular base function
      case _ => super.infix_*(x, y)
    }
}
```

An example: Fast Fourier Transform

Step 4: Define optimizations, rewriting certain patterns of IR nodes

```
trait TrigExpOptFFT extends TrigExpOpt {
  override def cos(x: Exp[Double]) = x match {
    case Const(x)
      if { val z = x / math.Pi / 0.5;
           z != 0 && z == z.toInt } =>
             Const(0.0)
    case _ => super.cos(x)
  }
}
```

An example: Fast Fourier Transform

Step 5: Extend code generator so new IR nodes can be turned into code

Finally, IR nodes have to be converted to actual code. The LMS framework provides a `ScalaGenBase` class that we can use. We only have to define what to do when the generator encounters one of the new nodes we added.

An example: Fast Fourier Transform

Step 5: Extend code generator so new IR nodes can be turned into code

- It's also possible to generate code for other languages if you define your own generator.
- Scala LMS even comes with a CGenBase trait that can generate C code from your staged Scala functions.

An example: Fast Fourier Transform

Step 5: Extend code generator so new IR nodes can be turned into code

The `CompileScala` trait defines a `compile` function that lets you load the generated code immediately into the running program. Essentially, `compile` "unstages" your staged function (`Rep[A] => Rep[B]`) into a regular function (`A => B`). This function can then be called:

```
val fftCompiled = compile(fft)
// Now we can call fftCompiled with regular values
// Just like any other function in the program
fftCompiled(Array(1.0,0.0, 1.0,0.0, 2.0,0.0, 2.0,0.0))

> Array(6.0,0.0, -1.0,1.0, 0.0,0.0, -1.0,-1.0)
```

Conclusion

TODO