Scala

Lightweight Modular Staging

Steven Both, Toby Rufinus, Jaspreet Singh, Daniël Stekelenburg

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?

Generative Programming

Commonalities of LMS and generative programming

Generative Programming

Differences of LMS and generative programming

Language Virtualization

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)
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

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 }
```

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?