Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs

Tiark Rompf, Martin Odersky
École Polytechnique Fédérale de Lausanne (EPFL)
About Authors:

Tiark Rompf

• Received an MS in computer science from University of Lübeck (2008) and a PhD from EPFL (2012).

• A member of Martin Odersky's Scala team since 2008.

• Is joining Purdue University as assistant professor in fall 2014.

• Contributions:

  • Scala (delimited continuations, efficient immutable data structures, speedup in compiler), Scala-Virtualized, LMS, Delite…
About Authors: Martin Odersky

- Professor of programming methods at EPFL.
- Received Ph.D. from ETH Zurich under the supervision of Niklaus Wirth (the creator of Pascal).
- Founder, chairman, and chief architect of Typesafe Inc.
- Creates Scala.
Short Description

• An implementation of *multi-stage programming* in Scala, which is *lightweight* (can be implemented as a library), *modular* (can be extended and composed in a flexible way).

• For multi-stage programming, check:
  
  • my previous slides, or
  
  • A Gentle Introduction to Multi-stage Programming

We’ll use “LMS” to represent “lightweight modular staging” in the remaining of this slides.
Introduction

• Let’s start from the classic example:

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

• Using multi-stage programming enables you to write generic code that can be specialized at runtime.

• Multi-stage programming implementation usually involves quotes, escape, evaluations.

• But in LMS, you only need to…
Introduction

• Let’s start from the classic example:

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

• But in LMS, you only need:

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

• binding-times are distinguished only by types. (Rep[T] vs T)

• given a sufficiently expressive language, the whole framework can be implemented as a library. (hence lightweight)

• using component technology, operations on staged expressions, data types to represent them, and optimizations (both generic and domain-specific) can be extended and composed in a flexible way. (hence modular)

• if the generator is well-typed so is the generated code.

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

trait Base {
    type Rep[+T]
}

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

```scala
trait BaseStr extends Base {
  type Rep[+T] = String
}

trait ArithStr extends Arith with BaseStr {
  implicit def unit(x: Double) = x.toString
  def infix_+(x: String, y: String) = "(%s+%s)".format(x, y)
  def infix_*(x: String, y: String) = "(%s*%s)".format(x, y)
}
```

Bad!
A lot of common sub-expressions! and redundant $*1.0$.

**Solution:** use graphs instead of strings.
LMS

```scala
trait Expressions {
  // expressions (atomic)
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]

  def fresh[T]: Sym[T]

  // definitions (composite, subclasses provided
  // by other traits)
  abstract class Def[T]

  def findDefinition[T](s: Sym[T]): Option[Def[T]]
  def findDefinition[T](d: Def[T]): Option[Sym[T]]
  def findOrCreateDefinition[T](d: Def[T]): Sym[T]

  // bind definitions to symbols automatically
  implicit def toAtom[T](d: Def[T]): Exp[T] =
    findOrCreateDefinition(d)

  // pattern match on definition of a given symbol
  object Def {
    def unapply[T](s: Sym[T]): Option[Def[T]] =
      findDefinition(s)
  }
}
```
for common sub-expression eliminations.
Infix operators: you can’t really do this in Scala. This is only available in Scala-Virtualized.
trait ArithExpOpt extends ArithExp {
    override def infix_*(x: Exp[Int], y: Exp[Int]) = (x, y) match {
        case (Const(x), Const(y)) => Const(x * y)
        case (x, Const(1)) => x
        case (Const(1), y) => x
        case _ => super.infix_*(x, y)
    }
}

3 * 2 => 6
x * 1 => x
1 * x => x
LMS

```scala
new PowerA with ExportGraph with ArithExpOpt {
  exportGraph {
    power(fresh[Double] + fresh[Double], 4)
  }
}

trait PowerA2 extends PowerA {
  this: Compile =>
  val p4 = compile { x: Rep[Double] =>
    power(x + x, 4)
  }
  // use compiled function p4 ...
}

new PowerA2 with CompileScala
  with ArithExpOpt with ScalaGenArith

// generated code:
class Anon$1 extends ((Double) => Double) {
  def apply(x0: Double): Double = {
    val x1 = x0 + x0
    val x2 = x1 * x1
    val x3 = x1 * x2
    val x4 = x1 * x3
    x4
  }
}
```

no common sub-expression!
no redundant constant!
LMS

Wait a minute…
How to compile and run the code?

```scala
trait ScalaGenBase extends BaseExp {
  def buildSchedule(Exp[_]): List[((Sym[_], Def[_]))] = ...
  def emitNode(sym: Sym[_], node: Def[_]) =
    throw new Exception("node_" + node + "\_not\_supported")
}

trait ScalaGenArith extends ScalaGenBase with ArithExp {
  override def emitNode(sym: Sym[_], node: Def[_]) = node match {
    case Plus(a,b) => println("val \_s\_\_=_\_a\_+_\_b".format(sym,a,b))
    case Times(a,b) => println("val \_s\_\_=_\_a\_\_\_*\_\_b".format(sym,a,b))
    case _ => super.emitNode(sym, rhs)
  }
}
```
More Features

• What about side effects and control flow?
  • Scala-Virtualized and monads.

• What about function and recursion?
  • Higher order abstract syntax.

…
In Addition...

• This is not based on the standard Scala compiler but on Scala-Virtualized, because users are not able to overload Scala primitives such as “if”, “for”, “val”, etc.

• It is not really that simple if desiring to implement something complicated. See this slides for some examples and tutorials.

• Beyond multi-stage programming: Delite
Q&A

Thanks!