The Essence of Multi-Stage Evaluation in LMS

Tiark Rompf
Purdue University

WadlerFest, April 10, 2016

Embedded DSLs

"Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things." (P. J. Landin, 1965)

Embedded DSLs

- Optimize expressions before running
  (math, queries over collections, ...)
- Generate code for non-standard targets
  (SQL, GPU, ...)

Deep Embedding

```scala
// Exp ::= n | Exp + Exp
abstract class Exp
case class Lit(n:Int) extends Exp
case class Plus(a:Exp,b:Exp) extends Exp
```

Deep & Shallow Embedding

```
type Exp
def lit(n:Int): Exp
def plus(a:Exp,b:Exp): Exp
```

Deep & Shallow Embedding

```
type Exp
implicit def lit(n:Int): Exp
implicit class ExpOps(a:Exp) {
  def +(b:Exp): Exp
}
```

Deep & Shallow Embedding

```
def times8(n: Exp) = {
  val times2 = n + n
  val times4 = times2 + times2
  times4 + times4
}
```

Deep & Shallow Embedding

```
def times8(n: Exp) = {
  val times2 = n + n
  val times4 = times2 + times2
  times4 + times4
}

times8(2) -->
```

Deep & Shallow Embedding

```
def times8(n: Exp) = {
  val times2 = n + n
  val times4 = times2 + times2
  times4 + times4
}

times8(2) -->

  Plus(Plus(Plus(Lit(2),Lit(2)),Plus(Lit(2),Lit(2))),
       Plus(Plus(Lit(2),Lit(2)),Plus(Lit(2),Lit(2))))
```

Deep & Shallow Embedding

```
def times8(n: Exp) = {
  val times2 = n + n
  val times4 = times2 + times2
  times4 + times4
}

times8(2) -->

  Plus(Plus(Plus(Lit(2),Lit(2)),Plus(Lit(2),Lit(2))),
       Plus(Plus(Lit(2),Lit(2)),Plus(Lit(2),Lit(2))))

times8(read()) -->
```

Deep & Shallow Embedding

```
def times8(n: Exp) = {
  val times2 = n + n
  val times4 = times2 + times2
  times4 + times4
}

times8(2) -->

  Plus(Plus(Plus(Lit(2),Lit(2)),Plus(Lit(2),Lit(2))),
       Plus(Plus(Lit(2),Lit(2)),Plus(Lit(2),Lit(2))))

times8(read()) -->

  Plus(Plus(Plus(Read(),Read()),Plus(Read(),Read())),
       Plus(Plus(Read(),Read()),Plus(Read(),Read())))
```

Semantics?

- EDSL semantics cannot be defined in isolation
- Need to look at meta language and object language together

Semantics?

- EDSL semantics cannot be defined in isolation
- Need to look at meta language and object language together
- Multi-stage programming (Taha & Sheard)

Lightweight Modular Staging (LMS)

- Type `T`: normal Scala expression (evaluate now)
- Type `Rep[T]`: DSL expression (generate code, eval later)

- Extensible IR, reusable optimizations, code generators, ...

Lightweight Modular Staging (LMS)

```
type Rep[T]
implicit def lit(x:Int): Rep[Int]
implicit class IntOps(a:Rep[Int]) {
  def +(b:Rep[Int]): Rep[Int]
}
```

What about functions?

How to create an expression of type `Rep[A => B]`?

```
def lit(x:Int): Rep[Int]

def fun[A,B](f:A=>B): Rep[A=>B] ?
```

What about functions?

How to create an expression of type `Rep[A => B]`?

```
def lit(x:Int): Rep[Int]

def fun[A,B](f:Rep[A]=>Rep[B]): Rep[A=>B]
```

Example

```
val ack: Int => Rep[Int => Int] = { m: Int =>
  fun { n: Rep[Int] =>
    if (m==0) n+1
    else if (n==0) ack(m-1)(1)
    else ack(m-1)(ack(m)(n-1))
  }
}
ack(2)
```

Example – Desugares

```
val ack: Int => Rep[Int => Int] = { m: Int =>
  fun { n: Rep[Int] =>
    if (m==0) __plus(n, __lit(1))
    else      __ifThenElse(__equals(n, __lit(0)),
                __apply(ack(m-1), __lit(1)),
                __apply(ack(m-1), __apply(ack(m),
                                      __minus(n, __lit(1)))))
  }
}
ack(2)
```

Example

```
val ack0: Int => Int = { n: Int =>  n+1 }
val ack1: Int => Int = { n: Int =>  if (n==0) ack0(1)
                                    else ack0(ack1(n-1)) }
val ack2: Int => Int = { n: Int =>  if (n==0) ack1(1)
                                    else ack1(ack2(n-1)) }
ack2
```

How to think about this?

Program specialization?
Partial evaluation?
Multi-stage programming?
Embedded DSL?

How does all this work?

Two-level λ calculus (Nielsen & Nielson '93)

λ Evaluator

```
Exp ::= Lit(Int) | Var(Int) | Tic
      | Lam(Exp) | Let(Exp,Exp) | App(Exp,Exp)
Val ::= Cst(Int) | Clo(Env,Exp)
Env   = List[Val]

var stC = 0
def tick() = { stC += 1; stC - 1 }

def eval(env: List[Val], e: Exp): Val = e match {
  case Lit(n)          => Cst(n)
  case Var(n)          => env(n)
  case Tic             => Cst(tick())
  case Lam(e)          => Clo(env,e)
  case Let(e1,e2)      => eval(env:+eval(env,e1),e2)
  case App(e1,e2)      =>
    val Clo(env3,e3) = eval(env,e1)
    eval(env3:+eval(env,e2),e3)
}
```

Two-Level λ Syntax

```
Exp ::= Lit(Int)  | Var(Int)  | Tic
      | Lit2(Int) | Var2(Int) | Tic2
      | Lam(Exp)  | Let(Exp,Exp)  | App(Exp,Exp)
      | Lam2(Exp) | Let2(Exp,Exp) | App2(Exp,Exp)
Val ::= Cst(Int) | Clo(Env,Exp) | Code(Exp)
```

Two-Level λ Evaluator

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)          => Cst(n)
  case Var(n)          => env(n)
  case Tic             => Cst(tick())
  case Lam(e)          => Clo(env,e)
  case Let(e1,e2)      => evalms(env:+evalms(env,e1),e2)
  case App(e1,e2)      =>
    val Clo(env3,e3)  = evalms(env,e1)
    evalms(env3:+evalms(env,e2),e3)

  case Lit2(n)         => Code(Lit(n))
  case Var2(n)         => ...
  case Tic2            => Code(Tic)
  case Lam2(e)         => ...
  case Let2(e1,e2)     => ...
  case App2(e1,e2)     =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    Code(App(s1,s2))
}
```

```
Let(Tic2,
  Lit2(Lit(1)))

--->
```

```
Let(Tic2,
  Lit2(Lit(1)))

--->

Lit(1)
```

But want:

```
Let(Tic,Lit(1))
```

Towards Let-Insertion

```
var stFresh = 0
var stBlock: List[Exp] = Nil
def run[A](f: => A): A = {
  val sF = stFresh
  val sB = stBlock
  try f finally { stFresh = sF; stBlock = sB }
}
def fresh()           = { stFresh += 1; Var(stFresh-1) }
def reflect(s:Exp)    = { stBlock :+= s; fresh() }
def reify(f: => Exp) = run {
  stBlock = Nil; val last = f; stBlock.foldRight(last)(Let) }
```

```scala
def anf(env: List[Exp], e: Exp): Exp = e match {
  case Lit(n)          => Lit(n)
  case Var(n)          => env(n)
  case Tic             => reflect(Tic)
  case Lam(e)          => reflect(Lam(reify(anf(env:+fresh(),e))))
  case App(e1,e2)      => reflect(App(anf(env,e1),anf(env,e2)))
  case Let(e1,e2)      => anf(env:+(anf(env,e1)),e2)
}
```

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)          => Cst(n)
  case Var(n)          => env(n)
  case Tic             => Cst(tick())
  case Lam(e)          => Clo(env,e)
  case Let(e1,e2)      => evalms(env:+evalms(env,e1),e2)
  case App(e1,e2)      =>
    val Clo(env3,e3)  = evalms(env,e1)
    evalms(env3:+evalms(env,e2),e3)

  case Lit2(n)         => Code(Lit(n))
  case Var2(n)         => env(n)
  case Tic2            => reflectc(Tic)
  case Lam2(e)         => reflectc(Lam(reifyc(evalms(env:+freshc(),e))
  case Let2(e1,e2)     => evalms(env:+evalms(env,e1),e2)
  case App2(e1,e2)     =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    reflectc(App(s1,s2))
}
```

Recursion

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case App(e1,e2) =>
    val Clo(env3,e3) = evalms(env,e1)
    val v2 = evalms(env,e2)
    evalms(env3:+Clo(env3,e3):+v2,e3)
  ...
  case Lam2(e) =>
    stFun collectFirst { case (n,`env`,`e`) => n } match {
      case Some(n) =>
        Code(Var(n))
      case None =>
        stFun :+= (stFresh,env,e)
        reflectc(Lam(reifyc(evalms(env:+freshc():+freshc(),e))))
    }
  ...
}
```

Example

```
val ack: Int => Rep[Int => Int] = { m: Int =>
  fun { n: Rep[Int] =>
    if (m==0) n+1
    else if (n==0) ack(m-1)(1)
    else ack(m-1)(ack(m)(n-1))
  }
}
ack(2)
```

Example – Desugared

```
val ack: Int => Rep[Int => Int] = { m: Int =>
  fun { n: Rep[Int] =>
    if (m==0) __plus(n, __lit(1))
    else      __ifThenElse(__equals(n, __lit(0)),
                __apply(ack(m-1), __lit(1)),
                __apply(ack(m-1), __apply(ack(m),
                                       __minus(n, __lit(1)))))
  }
}
ack(2)
```

Example – Desugared

```
val ack: Int => Rep[Int => Int] = { m: Int =>
  fun { n: Rep[Int] =>
    if (m==0) __plus(n, __lit(1))
    else      __ifThenElse(__equals(n, __lit(0)),
                __apply(ack(m-1), __lit(1)),
                __apply(ack(m-1), __apply(ack(m),
                                      __minus(n, __lit(1)))))
  }
}
ack(2)
```

```
Let(Lam(
  Lam2(Lam(
    Ifz(m,n+1,
      Ifz2(n,App2(App(ack,m-1),Lit2(Lit(1))),
             App2(App(ack,m-1),App2(App(ack,m),n-1)))))),
  App(ack,Lit(2)))
```

With:

```
ack = Var(0)    m   = Var(1)              n   = Var(3)
                m-1 = Minus(m,Lit(1))     n-1 = Minus2(n,Lit2(Lit(1)))
                                          n+1 = Plus2(n,Lit2(Lit(1)))
```

```
Let(Lam(
  Let(Ifz(Var(1),
    Let(Lam(Let(Ifz(Var(3),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(App(Var(4),Lit(1)),Var(5))),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(Minus(Var(3),Lit(1)),Let(App(Var(2),Var(5)),
          Let(App(Var(4),Var(6)),Var(7)))))
      ),Var(4))),
      Let(App(Var(2),Lit(1)),Var(3))),
    Let(Lam(Let(Ifz(Var(3),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(App(Var(4),Lit(1)),Var(5))),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(Minus(Var(3),Lit(1)),Let(App(Var(2),Var(5)),
          Let(App(Var(4),Var(6)),Var(7)))))
      ),Var(4))),
    Let(Minus(Var(1),Lit(1)),Let(App(Var(0),Var(3)),
      Let(App(Var(2),Var(4)),Var(5)))))
    ),Var(2))),
  Let(App(Var(0),Lit(2)),Var(1)))
```

```
val ack2: Int => Int = { n: Int =>
  if (n==0) {
    val ack1: Int => Int = { n: Int =>
      if (n==0) {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(1)
      } else {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(ack1(n-1))
      }
    }
    ack1(1)
  } else {
    val ack1: Int => Int = { n: Int =>
      if (n==0) {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(1)
      } else {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(ack1(n-1))
      }
    }
    ack1(ack2(n-1))
  }
}
ack2
```

```
val ack0: Int => Int = { n: Int => n+1 }
val ack1: Int => Int = { n: Int => if (n==0) ack0(1)
                                   else ack0(ack1(n-1)) }
val ack2: Int => Int = { n: Int => if (n==0) ack1(1)
                                   else ack1(ack2(n-1)) }
ack2
```

From 2-Level Evaluator back to EDSL

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)           => Cst(n)
  case Var(n)           => env(n)
  case Tic              => Cst(tick())
  case Lam(e)           => Clo(env,e)
  case Let(e1,e2)       => evalms(env:+evalms(env,e1),e2)
  case App(e1,e2)       =>
    val Clo(env3,e3)  = evalms(env,e1)
    evalms(env3:+evalms(env,e2),e3)

  case Lit2(n)          => Code(Lit(n))
  case Var2(n)          => env(n)
  case Tic2             => reflectc(Tic)
  case Lam2(e)          => reflectc(Lam(reifyc(evalms(env:+freshc(),e))
  case Let2(e1,e2)      => evalms(env:+evalms(env,e1),e2)
  case App2(e1,e2)      =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    reflectc(App(s1,s2))
}
```

Removing the Evaluator

```
type Rep[T] = Exp
def lit(n: Int): Rep[Int]                          = Lit(n)
def tic()                                          = reflect(Tic)
def app[A,B](f:Rep[A=>B],x:Rep[A]): Rep[B] = reflect(App(f,x))
def lam[A,B](f:Rep[A]=>Rep[B]): Rep[A=>B]
                                        = reflect(Lam(reify(f(fresh()))))
                                         // memoization elided
```

Conclusion

- EDSLs as multi-stage programming
- Discover instead of invent