# The Essence of Multi-Stage Evaluation in LMS

Tiark Rompf

Purdue University, USA: {firstname}@purdue.edu

**Abstract.** Embedded domain-specific languages (DSLs) are the subject of wide-spread interest, and a variety of implementation techniques exist. Some of them have been invented, and some of them discovered. Many are based on a form of *generative* or *multi-stage* programming, where the host language program builds up DSL terms during its evaluation. In this paper, we examine the execution model of LMS (Lightweight Modular Staging), a framework for embedded DSLs in Scala, and link it to evaluation in a two-stage lambda calculus. This clarifies the semantics of certain ad-hoc implementation choices, and provides guidance for implementing similar multi-stage evaluation facilities in other languages.

**Keywords:** Multi-stage programming, Scala, domain-specific languages, LMS (Lightweight Modular Staging), partial evaluation

## 1 Introduction

Embedded domain-specific languages (DSLs) are the subject of wide-spread interest across communities and languages. If a host-language program computes and assembles DSL terms, we are dealing with a form of *generative* or *multi-stage* programming [65]. Exactly how this is done depends on the language and framework that is used. But in any case, the interactions between meta-language and object-language (DSL) semantics can be non-trivial.

In this paper, we examine the evaluation mechanism of LMS (Lightweight Modular Staging) [54]. LMS is a framework for building embedded DSLs and program generators in Scala that uses types to identify staged expressions: any normal Scala expression of type `Int`, `String`, or in general `T`, is evaluated at program generation time, and expressions of type `Rep[Int]`, `Rep[String]`, or in general `Rep[T]`, produce staged expressions that will evaluate to values of the corresponding type `T` when the generated code is run. In general, operations on `Rep[T]` mirror those on `T` but lifted to the `Rep` level. For example, adding two `Rep[Int]` values will produce a `Rep[Int]` result.

While this description succeeds in giving an overall idea of the approach, it leaves many details unspecified. For example, how does one deal with functions? Primitive constants of, e.g., type `Int` can be lifted to `Rep[Int]` values without much effort as constant expressions, but it is less clear how to create a staged function of type `Rep[A=>B]`. The approach chosen by LMS is to use an explicit constructor `fun` that converts a present-stage function of type `Rep[A]=>Rep[B]` into a staged function of type `Rep[A=>B]`.

Armed with this knowledge, we can build up some intuition on how `Rep` types enable interesting program transformations. Here is a staged version of Ackermann's function (note the use of the `fun` constructor):

```
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)
```

What will be the result of this computation? The type of `ack` is a mixed-stage function type `Int => Rep[Int => Int]`, so `ack(m)` for a given integer `m` will have type `Rep[Int => Int]`: a staged function from `Int` to `Int` that corresponds to the body of `ack` but with all computations that depend on `m` but not `n` (nor any other `Rep` value) readily evaluated.

```
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
```

The question we would like to address in this paper is: *how does all this work?* To do so, we will link the evaluation strategy of Scala and LMS to a two-stage lambda calculus. Such multi-level calculi have been popularized by Nielson and Nielson [46] and play a key role in the partial evaluation literature to describe *binding-time* information similar to the `Rep[T]` vs `T` distinction in LMS.

Our two-level language is derived from Chapters 8 and 10 in the book by Jones, Gomard, and Sestoft [32]. We will build an evaluator for this calculus from the ground up and show how it computes the specialized `ack` function above, pointing to similarities and differences with systems from the literature.

While partial evaluation experts will most likely not find much novelty in this paper, we hope that this exposition will be useful in clarifying the semantics of multi-stage evaluation as it is implemented in LMS and that it may serve as guidance for implementing similar facilities in other languages or frameworks.

## 2 A Two-Stage Lambda Calculus

The term language of our calculus is as follows:

```
Exp ::= Lit(Int)  | Var(Int) | Let(Exp,Exp)
      | Lit2(Exp)
      | Lam(Exp)  | App(Exp,Exp)  | Tic
      | Lam2(Exp) | App2(Exp,Exp) | Tic2
      | Ifz(Exp,Exp,Exp)  | Plus(Exp,Exp)  | Minus(Exp,Exp)
      | Ifz2(Exp,Exp,Exp) | Plus2(Exp,Exp) | Minus2(Exp,Exp)
```

For ease of implementation, we use DeBrujin levels to represent binders. The term syntax contains integer literals, variables, let expressions, lambda abstractions, function application, simple side effects (`Tic`), conditionals, and arithmetic. Most (but not all) constructs come in two versions. For example, `Lam` is the present-stage lambda abstraction and `Lam2` is the future-stage lambda abstraction. We will see later that variables and let bindings need not be duplicated.

In what sense does this calculus model Scala and LMS? Since `Rep` types are just regular Scala types and operations on `Rep[T]` are implemented within the Scala language using operator overloading, Scala's local type inference mechanism essentially performs a local *binding-time analysis* [32] for us.

Here is the desugared version of `ack`:

```
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)
```

We can see that Scala's typechecker has replaced all operations that involve `Rep` values with method calls like `__plus` and `__minus`, using standard language facilities like method overloading and implicits. These methods serve as smart constructors that create corresponding expression terms when the program is executed.

If we want to model the same behavior in our calculus, we can directly translate the program above to our two-level language, replacing the primitive Scala constructs with first-level terms and method calls with second-level terms:

```
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)))
```

A few expressions have been factored out for clarity:

```
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)))
```

Note that we map staged function definitions such as `fun { m => ... }` to nested expressions `Lam2(Lam(...))`. For a faithful correspondance, we need to model both the constructor invocation and the Scala function passed as argument. Likewise, we map staged constant literals like `__lit(1)` to nested terms `Lit2(Lit(1))`.

Now that we have introduced the two-level term language, how should multi-stage evaluation proceed? Intuitively, each future-stage operator should create a program term at runtime, but a priori it is not clear how these terms are to be composed. For example:

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

This term could either evaluate to `Lit(1)`, assuming that `Tic2` evaluates to `Tic`, which is then discarded, or it could evaluate to `Let(Tic,Lit(1))`. Arguably, the second option is the desired one, but it takes some additional work to preserve the evaluation order and sharing behavior across evaluation stages. A well estab-

lished approach to achieve this is to insert let-bindings eagerly for all "serious" expressions and pass around only atomic identifiers or primitve constants. We will build our multi-stage evaluator in steps, starting from a standard single-stage evaluator, and add the necessary let-insertion logic in a second step, by composing the evaluator with a transformation into administrative normal form (ANF) [23].

## 3 Single-Stage Evaluation

The starting point for our multi-stage evaluator is a completely standard lambda calculus evaluator with environments and closures:

```
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)
}
```

In Scala, the *snoc* operator `:+` appends an element to the right of a list, so `env:+eval(env,e1)` will first evaluate `e1` in environment `env`, and then append the result to the right of `env`. Variable lookup uses DeBruijn levels, where `env(n)` returns the `n`th element from the left of a list.

To make matters slightly more interesting, we include a side-effecting operation `Tic` in our language, which we implement using side effects of the meta language: the implicit Scala monad. Of course we could also build a purely functional version based on monads, but for our purposes it is beneficial to trade off this bit of impurity for ease in composition later.

## 4 ANF Conversion / Let-Insertion

The second building block is ANF conversion [23], which brings programs into a normal form that makes the evaluation order explicit in a program and extracts all intermediate results into let bindings. The ANF syntax is given by the nonterminal `N` as follows:

```
V ::= Lit(Int) | Var(Int)
M ::= V | Lam(N) | App(V,V) | Tic
N ::= V | Let(M,N)
```

We will implement ANF conversion again using side-effects in the meta language of our evaluator. We need two pieces of state: an accumulator for generated let bindings `stBlock` and a counter for the next fresh variable name `stFresh`. We also introduce a function `run` that implements a dynamic scoping discipline for these two variables:

```
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 }
}
```

With this handling of mutable state in place, we can implement a first cut of our ANF conversion function:

```
def anf(env: List[Exp], e: Exp): Exp = e match {
  case Lit(n) => Lit(n)
  case Var(n) => env(n)
  case Tic    =>
    val s = Tic
    val f = stFresh
    stBlock:+= s
    stFresh += 1
    Var(f)
  case App(e1,e2) =>
    val ex1 = anf(env,e1)
    val ex2 = anf(env,e2)
    val s = App(ex1,ex2)
    val f = stFresh
    stBlock:+= s
    stFresh += 1
    Var(f)
  case Lam(e) =>
    val e1 = run {
      val f = stFresh
      stBlock = Nil
      stFresh += 1
      val res = anf(env:+Var(f),e)
      stBlock.foldRight(res)(Let)
    }
    val f = stFresh
    stBlock:+= Lam(e1)
    stFresh += 1
    Var(f)
  case Let(e1,e2) =>
    val ex1 = anf(env,e1)
    anf(env:+ex1,e2)
}
```

Function `anf` recurses structurally over the given term `e` and emits a let binding for each encountered non-trivial expression. For lambda abstractions, we use the dynamically scoped `run` construct to clear the stateful list of bindings when entering the nested scope and to reset the bindings and the fresh name reservoir when exiting the nested scope.

While this implementation gets the job done, it is not pretty. But it is easy to recognize some patterns: fresh variables are created in several places, and the

emitting of let bindings can also be abstracted over. We factor out the repetitive operations as follows:

```
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) }
```

Function `fresh` creates a fresh variable, `reflect` emits a let binding and returns the new identifier, and `reify` accumulates all let bindings created in a given block (note that `f` is a by-name parameter `=>Exp`).

Now we can go ahead and greatly simplify our implementation:

```
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)
}
```

In fact, we no longer mention mutable state directly in the conversion function anymore, which brings it much closer to common definitions of ANF conversion based on evaluation contexts [23].

Based on this formulation it is easy to convince oneself that ANF conversion preserves semantics:

```
eval(Nil, anf(Nil, e)) = eval(Nil, e)
```

## 5 Multi-Stage Evaluation

We now turn our attention to multi-stage evaluation. We start off with a slightly more uniform term language than the one given above:

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

For every operation in the term language there is a corresponding future-stage variant that serves as *term constructor*. In addition, the syntax of values now includes a `Code(e)` case for future-stage expressions.

The key idea now is to combine present-stage evaluation with future-stage normalization into ANF. The ANF conversion part is adapted slightly to map present-stage term constructors (`App2`, `Lam2`, ...) to code values containing their normal term equivalents (`App`, `Lam`, ...). This wrapping in `Code` values necessitates a few tweaks to the helper functions:

```
def freshc()          = Code(fresh())
def reflectc(s: Exp)  = Code(reflect(s))
def reifyc(f: => Val) = reify { val Code(r) = f; r }
```

Now we are ready to put together `eval` and `anf` into a single multi-stage evaluator (Figure 1). As we can see, term constructors like `App2` first evaluate their arguments in the present-stage to code values, from which the argument terms are extracted and the new `App` term is reflected.

```
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))
}
```

**Fig. 1.** Multi-stage evaluation: composing eval and anf

It is important to note that there are only three ways in which code values are constructed: (1) via `reflectc`, which creates a variable reference; (2) via `freshc` in the environment, and then accessed through variable lookup; (3) directly from a constant literal. The results of the only call to `reifyc`, which may create non-atomic expressions, are only used to fill in the bodies of `Lam` terms which are itself reflected immediately.

Thus, all code values are atomic expressions, terminating, and side-effect free. They can be freely stored, duplicated and passed around without changing the meaning of the program. Staged expressions will appear in the generated code in the same order they were encountered when running the multi-stage program.

Partial evaluators that support side effects work in a similar way, inserting let-bindings to prevent reordering or duplication of code. Many specializers are implemented in continuation-passing style, which also leads to an elegant on-the-fly ANF conversion. A version that corresponds very closely to the one above is shown by Lawall and Thiemann [37,67]. Our method of ANF conversion with direct-style `reflect` and `reify` operators can be seen as an instance of normalization by evaluation [40,17,22] or monadic reflection [21,44].

To obtain the full generated code for a top-level expression `e` one can use `reifyc(evalms(Nil,e))`. Another thing that becomes clear when looking at `evalms` is that `Var2` and `Let2` behave in exactly the same way as `Var` and `Let`. Therefore, we can drop them from our term language, as we had done in Section 2. Adding the other constructs, `Ifz`, `Plus`, and `Minus`, is straightforward.

## 6 Recursion

Where does this multi-stage evaluator leave us? We can almost, but not quite, run our `ack` example. The last missing bit is support for recursion. We could add an explicit `fix` combinator but instead, we chose to pass closures to themselves as first argument when called. Thus, all functions can potentially be recursive.

However, if we leave it at that, we will find that our example (and in fact many programs) will not terminate in the first stage. Here is a simpler example:

```
def f: Rep[Int => Int] = fun { n: Rep[Int] =>
  if (n == 0) 1 else f(n-1)
}
```

If we multi-stage evaluate this code naively, we will end up in a sequence of calls trying to compute the body of `f` — but in order to do that, we must compute the body of `f` again!

We need an additional insight: the body of `fun` in LMS or of `Lam2` in the calculus is itself an expression that will be evaluated. And if two functions have the same body expression, and the same free variables, then they must compute the same results for all inputs: intensional, i.e., structural equality implies extensional equality.

Thus, we can memoize the function expressions we are in the process of evaluating and tie the recursive knot to an enclosing function if we hit a recursive call. We achieve this by adding a small lookup table to the existing dynamically scoped mutable state:

```
var stFun: List[(Int,Env,Exp)] = Nil
def run[A](f: => A): A = {
  ...
  val sN = stFun
  try f finally { ...; stFun = sN }
}
```

In addition, we modify the handling of lambda abstractions to check whether we are trying to expand a given function body inside itself. If that is the case, we just return the already existing parent symbol. Otherwise, the function is new. We create an entry in our memo table and proceed to evaluate the body.

```
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))))
    }
  ...
}
```

The code snippet also shows the modified `App` case that passes the closure to itself as first argument. The `Lam` case is updated correspondingly to create two fresh vars as placeholders for the arguments.

With this support for specializing recursive functions in place, we can successfully evaluate our `ack` example. We obtain the following result for `ack(2)`. Term syntax on the left, translated to Scala on the right:

```
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
```

We can see that this code does the right thing and corresponds to the right recursion pattern, but it contains duplicated function definitions for `ack0` and `ack1`. These are easily reclaimed by a code motion and common subexpression elimination or global value numbering algorithm. Hoisting functions and removing duplicates yields exactly the desired result:

```
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
```

In the partial evaluation literature such patterns are known as polyvariant specialization [8]: one function in the source yields multiple specialized variants in the residual generated code. Sophisticated partial evaluators such as Similix or PGG implement similar techniques [6,28,37,67].

## 7 Relation to LMS

We have seen how we can model and describe a simple multi-stage evaluation algorithm for a small language. But how can we relate it to the actual implementation of LMS? Essentially we take the `evalms` function and turn it inside out, from an interpreter over an initial term language to an evaluator in tagless-final [10] style. But there is a problem: since we are working inside a *running* Scala program we cannot get our hands on *unevaluated* Scala expressions, as would be required by our implementation of staged literals and staged lambdas:

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case Lit2(n)        => Code(Lit(n))
  case Lam2(e)        => reflectc(Lam(reifyc(evalms(env:+freshc(),e))))
}
```

The argument `n` for `Lit2` would correspond to a literal constant in the Scala source code, and the argument `e` for `Lam2` corresponds to an unevaluated Scala expression, which we would evaluate at our own leisure, in an extended context, on the right hand side. The fact that tools like LMS operate as libraries inside a general-purpose host language is a key difference to offline partial evaluation systems that operate on the program text.

We generalize our implementation in such a way that `Lit2` takes an expression that *evaluates* to a constant, and `Lam2` takes an expression that *evaluates* to a function. The intention is that `Lit2(Lit(n))` and `Lam2(Lam(e))` correspond to the previous behavior. The construct `Lit2` now implements a *lifting* facility, which enables embedding of arbitrary values computed in the present stage as constants in the generated code. Note, however, that lifting (or *cross-stage-persistance*) is not automatically valid for all possible types. It is easy to see, for example, that a present-stage Scala closure does not necessarily have a valid representation as C source code. For this reason, we restrict `Lit` to integers here, and handle functions separately using `Lam2`. The implementation of `Lam2` can be seen as a pretty standard use of higher-order abstract syntax.

We modify our evaluator as follows to immediately evaluate all subexpressions and introduce the required indirection for functions:

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case Lit2(e)        => val Cst(n) = evalms(env,e); Code(Lit(n))
  case Tic2           => reflectc(Tic)
  case Lam2(e)        =>
    val f = evalms(env,e) // memoization elided
    reflectc(Lam(reifyc(evalms(env,App(f,freshc())))))
  case App2(e1,e2)    =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    reflectc(App(s1,s2))
}
```

As the last step towards our tagless-final interpreter, we refactor the evaluator to introduce constructor methods for each kind of term:

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case Lit2(e)        => val Cst(n) = evalms(env,e); Code(lit(n))
  case Tic2           => Code(tic())
  case Lam2(e)        =>
    val f = evalms(env,e)
    val f1 = { x => val Code(r) = evalms(env,App(f,Code(x))); r }
    Code(lam(f1))
  case App2(e1,e2)    =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    Code(app(s1,s2))
}
```

These term constructors are defined as follows to abstract over explicit environments, recursive calls to `evalms`, and `Code` values:

```
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
```

At this point, we no longer need the explicit evaluator, and we can just use the standard Scala evaluation to construct staged terms.

This API corresponds almost directly to the actual implementation in LMS. Behind the `reflect` and `reify` API, the internal representation of LMS is different though. Rather than as simple expression trees, LMS has a graph-based intermediate representation (IR) that directly supports code motion, common subexpression elimination, dead code elimination and a range of other optimizations [54,55,14]. LMS also makes pervasive use of smart constructors to perform rewriting while the IR is constructed.

But can we still handle recursion correctly in this setting? By switching away from our explicit evaluator we lose control over the representation of closures, which was key for detecting recursive calls through memoization. Fortunately, Scala implements closures as objects on the JVM, with free references stored as fields that can be queried via runtime reflection. In practice, LMS uses Java serialization to convert closures to byte arrays, and then takes this binary fingerprint as a key for memoization. This method neatly takes care of cyclic references, too. To implement a framework similar to LMS in other languages, a similar facility would likely be needed, or an alternative strategy for supporting recursive functions would need to be used (e.g. explicit staged fixpoint combinators).

## 8 Case Study: Regular Expression Matchers

While the `ack` example in the previous sections served as a good example for exposition, it is arguable not one that has a lot of practical uses. But the pattern directly translates to more interesting applications.

In this section, we give a short overview of a fast regular expression matcher that was developed using essentially the same technique by Nada Amin and the author. Part of the code and description below has appeared in the author's PhD thesis [53] and in a paper at POPL 2013 [55]. Specializing string matchers and parsers is a popular benchmark in the partial evaluation and supercompilation literature [15,58,1,61,69,59].

We consider regular expression matchers that are "multi-threaded" and spawn a new conceptual thread to process alternatives in parallel. Of course these matchers do not actually spawn OS-level threads, but rather need to be advanced manually by client code. Thus, they are similar to coroutines.

From these non-deterministic matchers, we generate a set of mutually tail-recursive functions that implement a deterministic finite automaton (DFA) that recognizes the same regexp pattern without backtracking.

### 8.1 Regexp Matchers as Nondeterministic Finite Automata (NFA)

Here is a simple example for the fixed regular expression '.*AAB':

```
def findAAB(): NIO = {
  guard(Set('A')) {
    guard(Set('A')) {
      guard(Set('B'), Found)) {
        stop()
  }}} ++
  guard(None) { findAAB() } // in parallel...
}
```

We can easily add combinators on top of the core abstractions that take care of producing matchers from textual regular expressions. However the point here is to demonstrate how the implementation works. The given matcher uses an API that models nondeterministic finite automata (NFA):

```
type NIO = List[Trans]   // state: many possible transitions
case class Trans(c: Set[Char], x: Flag, s: () => NIO)
def guard(cond: Set[Char], flag: Flag)(e: => NIO): NIO =
  List(Trans(cond, flag, () => e))
def stop(): NIO = Nil
```

An NFA state consists of a list of possible transitions (type `NIO`). Each transition may be guarded by a set of characters and it may have a flag to be signaled if the transition is taken. It also knows how to compute the following state. We use `Char`s for simplicity, but of course we could use generic types as well. From a given NFA transition, we can obtain the following state, and thus recursively unfold the state space. Thus, it is sometimes useful to identify the notions of NFA and NFA state. NFA states, and thus NFAs, can be composed in parallel through list concatenation. Method `guard` implements sequential composition. It creates a new NFA state with a single start transition and "the rest of the automaton" given as a by-name expression (type `=>NIO`) that computes the following state. Note that the API does not mention where input is obtained from (files, streams, etc).

### 8.2 From NFA to DFA using Staging

We will translate NFAs to DFAs using LMS. This is the unstaged DFA API that will be used by the generated code:

```
abstract class DfaState {
  def hasFlag(x: Flag): Boolean
  def next(c: Char): DfaState
}
def dfaFlagged(flag: Flag, link: DfaState) = new DfaState {
  def hasFlag(x: Flag) = x == flag || link.hasFlag(x)
  def next(c: Char) = link.next(c)
}
def dfaState(f: Char => DfaState) = new DfaState {
  def hasFlag(x: Flag) = false
  def next(c: Char) = f(c)
}
```

The staged API is just a thin wrapper:

```
type DIO = Rep[DfaState]
def dfa_flag(x: Flag)(link: DIO): DIO
def dfa_trans(f: Rep[Char] => DIO): DIO
```

Translating an NFA to a DFA is accomplished by creating a DFA state for each encountered NFA configuration (removing duplicate states via `canonicalize`):

```
def convertNFAtoDFA(state: NIO): DIO = {
  val cstate = canonicalize(state)
  dfa_trans { c: Rep[Char] =>
    exploreNFA(cstate, c)(dfa_flag) { next =>
      convertNFAtoDFA(next)
    }
  }
}
convertNFAtoDFA(findAAB())
```

Since LMS memoizes functions in the same was as we have shown in Section 6 (here, the argument to `dfa_trans`), the framework ensures termination if the NFA is indeed finite. We use a separate function to explore the NFA space (see below), advancing the automaton by a symbolic character `cin` to invoke its continuations `k` with a new automaton, i.e. the possible set of states after consuming `cin`. The given implementation assumes that character sets contain either zero or one characters, the empty set `Set()` denoting a wildcard match. More elaborate cases such as character ranges are easy to add. The algorithm tries to remove as many redundant checks and impossible branches as possible. This only works because the character guards are staging-time values.

```
def exploreNFA[A](xs: NIO, cin: Rep[Char])(flag: Flag => Rep[A] => Rep[A])
                                          (k: NIO => Rep[A]):Rep[A] = xs match {
  case Nil => k(Nil)
  case Trans(Set(c), e, s)::rest =>
    if (cin == c) {
      // found match: drop transitions that look for other chars and
      // remove redundant checks
      val xs1 = rest collect { case Trans(Set('c')|None,e,s) => Trans(Set(),e,s) }
      val maybeFlag = e map flag getOrElse (x=>x)
      maybeFlag(exploreNFA(xs1, cin)(acc => k(acc ++ s())))
    } else {
      // no match, drop transitions that look for same char
      val xs1 = rest filter { case Trans(Set('c'),_,_) => false case _ => true }
      exploreNFA(xs1, cin)(k)
    }
  case Trans(Set(), e, s)::rest =>
    val maybeFlag = e map flag getOrElse (x=>x)
    maybeFlag(exploreNFA(rest, cin)(acc => k(acc ++ s())))
}
```

### 8.3 Generated State Machine Code

The generated matcher code for regular expression `.*AAB` is shown below, with some slight syntactic changes to make it more readable. Each function corresponds to one DFA state. Note how negative information has been used to prune the transition space: given input such as `...AAB` the automaton jumps back to

the initial state, i.e. it recognizes that the last character B cannot also be A, and it starts looking for two As after the B.

```
def stagedFindAAB(): DfaState = {
  val found = dfaFlagged(Found, matched_nothing)
  val matched_AA = dfaState { c: (Char) =>
    if (c == B) found
    else if (c == A) matched_AA
    else matched_nothing
  }
  val matched_A = dfaState { c: (Char) =>
    if (c == A) matched_AA
    else matched_nothing
  }
  val matched_nothing = dfaState { c: (Char) =>
    if (c == A) matched_A
    else matched_nothing
  }
}
```

## 9 Related Work

Multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [65] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [9] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [54] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code [53]. LMS draws inspiration from earlier work such as TaskGraph [4], a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [55,7,56,38,63].

**Partial Evaluation** Partial evaluation [32] is an automatic program specialization technique. Despite their automatic nature, most partial evaluators also provide annotations to guide specialization decisions. Some notable systems include DyC [26], an annotation-directed specializer for C, JSpec/Tempo [57], the JSC Java Supercompiler [34], and Civet [58].

Partial evaluation has addressed higher-order languages with state using similar let-insertion techniques as discussed here [6,28,37,67]. Further work has studied partially static structures [43] and partially static operations [66], and compilation based on combinations of partial evaluation, staging and abstract interpretation [60,16,33]. Two-level languages are frequently used as a basis for describing binding-time annotated programs [32,47].

**Embedded DSLs** Embedded languages have a long history [36]. Hudak introduced the concept of embedding DSLs as pure libraries [30,31]. Steele proposed the idea of "growing" a language [62]. The concept of linguistic reuse goes back to Krishnamurthi [35]; Language virtualization to Chafi et al. [12]. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [10] and Hofer et al. [29], going back to much earlier work by Reynolds [52]. Compiling embedded DSLs through dynamically generated ASTs

was pioneered by Leijen and Meijer [39] and Elliot et al. [20]. All these works greatly inspired the development of LMS. Haskell is a popular host language for embedded DSLs [64,25], examples being Accelerate [42], Feldspar [3], Nikola [41]. Recent work presents new approaches around quotation and normalization for DSLs [45,13]. Other performance oriented DSLs include Firepile [48] (Scala), Terra [18,19] (Lua). Copperhead [11] (Python). Rackets macros [68] provide full control over the syntax and semantics.

**Program Generators** A number of high-performance program generators have been built, for example ATLAS [70] (linear algebra), FFTW [24] (discrete fourier transform), and Spiral [49] (general linear transformations). Other systems include PetaBricks [2], CVXgen [27] and Halide [51,50].

## References


1. M. S. Ager, O. Danvy, and H. K. Rohde. Fast partial evaluation of pattern matching in strings. *ACM Trans. Program. Lang. Syst.*, 28(4):696–714, 2006.
2. S. P. Amarasinghe. Petabricks: a language and compiler based on autotuning. In M. Katevenis, M. Martonosi, C. Kozyrakis, and O. Temam, editors, *High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings*, page 3. ACM, 2011.
3. E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar: An embedded language for digital signal processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.
4. O. Beckmann, A. Houghton, M. R. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.
5. H. Boehm and C. Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013.
6. A. Bondorf. *Self-applicable partial evaluation.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1990.
7. K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.
8. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Inf.*, 21:473–484, 1984.
9. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. GPCE, pages 57–76, 2003.
10. J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
11. B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

12. H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Onward!, 2010.
13. J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 403–416. ACM, 2013.
14. C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.
15. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Inf. Process. Lett.*, 30(2):79–86, 1989.
16. C. Consel and S.-C. Khoo. Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.*, 15(3):463–493, 1993.
17. O. Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*, pages 367–411. Springer, 1998.
18. Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In Boehm and Flanagan [5], pages 105–116.
19. Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 11. ACM, 2014.
20. C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.
21. A. Filinski. Representing monads. In H. Boehm, B. Lang, and D. M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 446–457. ACM Press, 1994.
22. A. Filinski. Normalization by evaluation for the computational lambda-calculus. In *TLCA*, pages 151–165, 2001.
23. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In R. Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993.
24. M. Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.
25. A. Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30:30–30:43, Apr. 2014.
26. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000.
27. M. Hanger, T. A. Johansen, G. K. Mykland, and A. Skullestad. Dynamic model predictive control allocation using CVXGEN. In *9th IEEE International Conference on Control and Automation, ICCA 2011, Santiago, Chile, December 19-21, 2011*, pages 417–422. IEEE, 2011.
28. J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

29. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.
30. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.
31. P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.
32. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
33. O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In G. C. Buttazzo, editor, *EMSOFT*, pages 249–258. ACM, 2004.
34. A. V. Klimov. A java supercompiler and its application to verification of cache-coherence protocols. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2009.
35. S. Krishnamurthi. *Linguistic reuse*. PhD thesis, Computer Science, Rice University, Houston, 2001.
36. P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.
37. J. L. Lawall and P. Thiemann. Sound specialization in the presence of computational effects. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190. Springer, 1997.
38. H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.
39. D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.
40. S. Lindley. Normalisation by evaluation in the compilation of typed functional programming languages. 2005.
41. G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.
42. T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.
43. T. A. Mogensen. Partially static structures in a self-applicable partial evaluator. 1988.
44. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
45. S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: Quoted domain specific languages. Technical report, University of Edinburgh, 2015.
46. F. Nielson and H. R. Nielson. Code generation from two-level denotational meta-languages. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17-19, 1985*, volume 217 of *Lecture Notes in Computer Science*, pages 192–205. Springer, 1985.
47. F. Nielson and H. R. Nielson. Multi-level lambda-calculi: An algebraic description. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International*

*Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, volume 1110 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 1996.
48. N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for GPUs in Scala. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE, pages 107–116, New York, NY, USA, 2011. ACM.
49. M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.
50. J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.
51. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Boehm and Flanagan [5], pages 519–530.
52. J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. 1975.
53. T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming.* PhD thesis, EPFL, 2012.
54. T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.
55. T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. In *POPL*, 2013.
56. T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.
57. U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.
58. A. Shali and W. R. Cook. Hybrid partial evaluation. OOPSLA, pages 375–390, 2011.
59. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *J. Funct. Program.*, 6(6):811–838, 1996.
60. M. Sperber and P. Thiemann. Realistic compilation by partial evaluation. In *PLDI*, pages 206–214, 1996.
61. M. Sperber and P. Thiemann. Generation of lr parsers by partial evaluation. *ACM Trans. Program. Lang. Syst.*, 22(2):224–264, 2000.
62. G. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.
63. A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*, 2013.
64. B. J. Svensson, M. Sheeran, and R. Newton. Design exploration through code-generating DSLs. *Queue*, 12(4):40:40–40:52, Apr. 2014.
65. W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
66. P. Thiemann. Partially static operations. In E. Albert and S.-C. Mu, editors, *PEPM*, pages 75–76. ACM, 2013.

67. P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state. Technical report, 1999.
68. S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.
69. V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.
70. R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.