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

Tiark Rompf
EPFL, Lausanne, Switzerland
first.last@epfl.ch

Martin Odersky
EPFL, Lausanne, Switzerland
first.last@epfl.ch

ABSTRACT

Good software engineering practice demands generalization and abstraction, whereas high performance demands specialization and concretization. These goals are at odds, and compilers can only rarely translate expressive high-level programs to modern hardware platforms in a way that makes best use of the available resources.

Generative programming is a promising alternative to fully automatic translation. Instead of writing down the target program directly, developers write a program generator, which produces the target program as its output. The generator can be written in a high-level, generic style and still produce efficient, specialized target programs. In practice, however, developing high-quality program generators requires a very large effort that is often hard to amortize.

We present *lightweight modular staging* (LMS), a generative programming approach that lowers this effort significantly. LMS seamlessly combines program generator logic with the generated code in a single program, using only types to distinguish the two stages of execution. Through extensive use of component technology, LMS makes a reusable and extensible compiler framework available at the library level, allowing programmers to tightly integrate domain-specific abstractions and optimizations into the generation process, with common generic optimizations provided by the framework.

LMS is well suited to develop embedded domain specific languages (DSLs) and has been used to develop powerful performance-oriented DSLs for demanding domains such as machine learning, with code generation for heterogeneous platforms including GPUs. LMS has also been used to generate SQL for embedded database queries and JavaScript for web applications.

1. INTRODUCTION

Building and managing complex software systems is only possible by generalizing functionality and abstracting from particular use cases. Achieving performance, on the other hand, requires concretizing configurations and specializing code to its particular environment. Generative programming can bridge this gap by translating away abstraction overhead and effectively specializing generic programs.

Generative programming can be broadly classified as static or dynamic. Static code generation happens at compile time, a widely used example is the C++ template language or macro systems in other languages. Dynamic code generation that takes place at program runtime brings additional flexibility because code can be specialized with respect to parameters not yet available at compile time.

Many computations can naturally be separated into stages distinguished by frequency of execution or availability of information. Staging transformations aim at executing certain pieces of code less often or at a time where performance is less critical. In the context of program generation, multi-stage programming (MSP, *staging* for short) as established by Taha and Sheard [22] allows programmers to explicitly delay evaluation of a program expression to a later stage (thus, *staging* an expression). The present stage effectively acts as a code generator that composes (and possibly executes) the program of the next stage. A nice property of this approach is that generator and generated code are expressed in a single program, with the aim that programmers can construct a multi-stage program from a naive implementation of the same algorithm by adding staging annotations in a selective way.

Basic mechanisms for composing program fragments at runtime have existed for a long time in the form of Lisp's "code is data" model and its use of *quasi-quotation*: syntactic annotations to denote expressions that should remain unevaluated and to mark holes within them, to be filled in with expressions computed elsewhere. Dedicated MSP languages such as MetaML [22] and MetaOCaml [1] add well-scoping and well-typing guarantees about the generated code. Despite these advances, incorporating dynamic code generation into larger systems in the form of self-optimizing "active" libraries or adding compilation to embedded domain-specific languages (DSLs) that are implemented as libraries remains a significant challenge.

1.1 Contributions

In this paper, we present *lightweight modular staging* (LMS), a dynamic code generation approach that further reduces the effort to develop sophisticated program generators. The presentation will use the Scala programming language but the core concepts are not tied to any language in particular.

The classical introductory staging example is to specialize the power function for a given exponent, assuming that a program will take many different numbers to the same power. Considering the usual implementation,

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

we want to turn the base b into a *staged* expression. Following Carette et al. [2] and Hofer et al. [8], and in contrast to the quasi-quotation approach of syntactic annotations, the central idea of LMS is to use types to distinguish the computational stages. In particular, we change b's declared type from Double to Rep[Double]. The meaning of having type Rep[Double] is that b *represents* a computation that will yield a Double in the next stage. We also change the function's return type accordingly.

Now we need to regain the ability to do arithmetic on b, which is no longer a plain Double. The second idea of LMS is to package operations on staged types as components. To make its required functionality explicit, we wrap the power function itself up as a component (a trait):

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

In Scala, traits are similar to classes but they can be used in mix-in composition, a limited form of multiple inheritance [15]. The self-type annotation **this:** Arith denotes a "requires" relationship: whenever an instance of Power is created, an instance of a concrete (but unspecified) subclass of Arith (see Figure 3) that provides staged double arithmetic must be mixed in, too.

LMS shares many benefits with earlier staging approaches:

- Code generators and generated code are expressed in the same program.
- Objects that are live within the generator's heap can be accessed from generated code if the code is invoked directly (cross-stage persistence).
- Staged expressions inherit the static scope of the generator and if the generator is well-typed so is the generated code.
- Data types representing staged expressions are inaccessible to the program itself (making optimizations safe that preserve only semantic but not structural equality).

At the same time, LMS differs from previous approaches in important ways:

- Staging is driven entirely by types, no special syntax is required.
- Given a sufficiently expressive programming language, the whole framework can be implemented as a library (hence *lightweight*).
- Staged code fragments are composed through explicit operations, in particular lifted variants of the usual operators and control flow statements extended with optimizing symbolic rewritings (semantic composition instead of syntactic expansion).
- 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*).
- Different code generation targets can easily be supported, their implementations can share common code.
- The relative evaluation order of expressions is preserved across stage boundaries. There is no danger of accidentally omitting, reordering, or duplicating computation.

The last item deserves more explanation, as it is arguably the most prevalent difficulty programmers face with other staging approaches. Since freely composing code fragments voids any evaluation order guarantees (such as call-by-value semantics), adding quasi-quotation annotations can easily change the result of a program in unforeseen ways or slow down the program by duplicating computation. In practice, quasi-quotation is therefore often used as a low-level tool, like an "assembly language" for code generation, and combined with application specific front-end layers that apply high-level code optimizations and ensure correct evaluation order. LMS provides predictable execution order, makes the composition of staged expressions extensible and thus removes the need for extra front ends.

LMS is a key constituent of Delite [13, 18], an open-source framework for high-performance parallel domain specific languages (DSLs) that has been used to develop DSLs for demanding application areas such as machine learning, graph processing or mesh-based partial differential equation solvers. In this context, LMS enables "abstraction without regret": DSL programmers can use arbitrary Scala features to structure their programs in the generator stage, with the comforting knowledge that LMS guarantees to remove these abstraction during staging and no runtime price will need to be paid. Coupled with advanced compiler optimizations such as data parallel loop fusion and architecture-specific data structure transformations, Delite generates efficient code for a variety of parallel platforms like multi-core CPUs and GPUs.

LMS has also been used to generate SQL statements for queries embedded in Scala programs and to generate JavaScript from staged Scala fragments, allowing web-applications to execute parts of their logic in the client's browser.

1.2 Organization

The rest of this paper is structured as follows. Section 2 presents an end-to-end example, turning a naive algorithm into an efficient code generator, while introducing the major LMS components on the way. Section 2.1 discusses representations of staged code, Section 2.2 is concerned with optimizations, Section 2.3 with target code generation, and Section 2.4 concludes the example by showing how generated code can be integrated in larger programs. Section 3 describes how additional features can be added, in particular functions and recursion. Section 4 discussed related work. An earlier version of this paper appeared at GPCE 2010. The original version contains a few additional examples and a more thorough discussion of how effectful statements are represented, while the present version has been updated with some new material in Section 3.

2. AN END-TO-END EXAMPLE

In the same way as the simple power function shown above, we can stage far more interesting and practically relevant programs, for example the fast fourier transform (FFT). A staged FFT, implemented in MetaOCaml, has been presented by Kiselyov et al. [12] Their work is a very good showcase for how staging allows to transform a simple, unoptimized algorithm into an efficient program generator. Achieving this in the context of MetaOCaml, however, required restructuring the program into monadic style and adding a front-end layer for performing symbolic rewritings. Using our approach of just adding Rep types, we can go from the

```
trait FFT { this: Arith with Trig =>
  case class Complex(re: Rep[Double], im: Rep[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))
  }
  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) map {
        case ((x, y), k) =>
          val z = omega(k, N) * y
          (x + z, x - z)
      }.unzip;
      even2 ::: odd2
  }
}
```

Figure 1: FFT code. Only the real and imaginary components of complex numbers need to be staged.

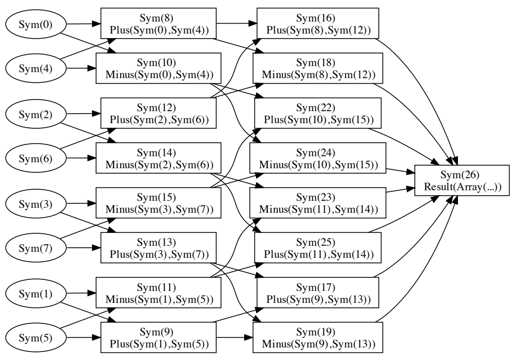
Sym(0)
Sym(4)
Sym(2)
Sym(6)
Sym(3)
Sym(7)
Sym(1)
Sym(5)
Sym(8) Plus(Sym(0),Sym(4))
Sym(10) Minus(Sym(0),Sym(4))
Sym(12) Plus(Sym(2),Sym(6))
Sym(14) Minus(Sym(2),Sym(6))
Sym(15) Minus(Sym(3),Sym(7))
Sym(13) Plus(Sym(3),Sym(7))
Sym(11) Minus(Sym(1),Sym(5))
Sym(9) Plus(Sym(1),Sym(5))
Sym(16) Plus(Sym(8),Sym(12))
Sym(18) Minus(Sym(8),Sym(12))
Sym(22) Plus(Sym(10),Sym(15))
Sym(24) Minus(Sym(10),Sym(15))
Sym(23) Minus(Sym(11),Sym(14))
Sym(25) Plus(Sym(11),Sym(14))
Sym(17) Plus(Sym(9),Sym(13))
Sym(19) Minus(Sym(9),Sym(13))
Sym(26) Result(Array(...))

Figure 2: Computation graph for size-4 FFT. Auto-generated from staged code in Figure 1.

naive textbook-algorithm to the staged version (shown in Figure 1) by changing literally two lines of code:

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

All that is needed is adding the self-type annotation to import arithmetic and trigonometric operations and changing the type of the real and imaginary components of complex numbers from `Double` to `Rep[Double]`.

Merely changing the types will not provide us with the desired optimizations yet. We will see below how we can add the transformations described by Kiselyov et al. to generate the same fixed-size FFT code, corresponding to the famous FFT butterfly networks (see Figure 2). Despite the seemingly naive algorithm, this staged code is free of branches, intermediate data structures and redundant computations. The important point here is that we can add these transformations without any further changes to the code in Figure 1, just by mixing in the trait `FFT` with a few others.

In the remainder of this section we present the LMS framework that is used to generate the code in Figure 2 from the algorithm in Figure 1. Before considering specific optimizations, a closer look at the definition of `Rep` and the traits `Arith` and `Trig` is in order. The definitions are given in Figure 3. In trait `Base`, the declaration **type** `Rep[+T]` defines an abstract type constructor (also called a higher-kinded type) `Rep` which we take to range over possible representations of staged expressions. Since `Rep` is abstract, no concrete representation is defined yet; the declaration merely postulates the existence of *some* representation.

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

Figure 3: Interface traits defining staged operations. For simplicity, operations are defined for `Double` only.

Trait `Arith` extends trait `Base` and contains only abstract members, too. These postulate the existence of an implicit lifting of `Double`s to staged values and the usual arithmetic operations on staged expressions of type `Rep[Double]`. The restriction to `Double`s is just to keep the presentation concise. Any suitable means to abstract over numeric types, such as the "type class" `Numeric` from the Scala standard library could be used to define `Arith` in a generic way for a range of numeric types. Analogously to `Double`, we could define arithmetic on matrices and vectors and implement optimizations on those operations in exactly the same way. Trait `Trig` is similar to `Arith` but defines trigonometric operations.

One way to look at `Base`, `Arith` and `Trig` is as the definition of a typed embedded language (or its syntax). The embedding is *tagless* (i.e. method resolution happens at compile time without additional runtime dispatch overhead) [2] and *polymorphic* [8], in the sense that we are free to pick any suitable concrete implementation that fulfills the given interface.

From a safety point of view, keeping the actual representation inaccessible from the program generator is very important. Otherwise, the program generator could execute different code depending on the exact structure of a staged expression. Optimizations that replace staged code with simpler but semantically equivalent expressions would risk changing the meaning of the generated program.

2.1 Representing Staged Code

With the aim of generating code, we could represent staged expressions directly as strings. But for optimization purposes we would rather have a structured intermediate representation that we can analyze in various ways.

We choose a representation based on expression trees, or, more exactly, a "sea of nodes" representation that is in fact a directed (and for the moment, acyclic) graph but can be accessed through a tree-like interface. The necessary infrastructure is defined in trait `Expressions`, shown in Figure 4.

There are three categories of objects involved: expressions, which are atomic (subclasses of Exp: constants and symbols; with a "gensym" operator fresh to create fresh symbols), definitions, which represent composite operations (subclasses of Def, to be provided by other components), and blocks, which model nested scopes.

The guiding principle is that each definition has an associated symbol and refers to other definitions only via their symbols. In effect, this means that every composite value will be named, similar to administrative normal form (ANF). Trait Expressions provides methods to find a definition given a symbol or vice versa. The extractor object Def allows to pattern-match on the definition of a given symbol, a facility that is used for implementing rewritings (see below).

The framework ensures that code that contains staging operations will always be executed within the dynamic scope of at least one invocation of reifyBlock, which returns a block object and takes as call-by-name argument the present-stage expression that will compute the staged block result. Block objects can be part of definitions, e.g. for loops or conditionals.

The implicit conversion method toAtom converts a definition to an atomic expression and links it to the scope being built up by the innermost enclosing reifyBlock call. When the definition is known to be side-effect free, toAtom will search the already encountered definitions for a structurally equivalent one. If a matching previous definition is found, its symbol will be returned, possibly moving the definition to a parent scope to make it accessible. If the definition may have side effects or it is seen for the first time, it will be associated with a fresh symbol and saved for future reference. This simple scheme provides a powerful global value numbering (common subexpression elimination) optimization that effectively prevents generating duplicate code. More complicated optimization schemes can be added, too.

Since all operations in interface traits such as Arith return Rep types, defining Rep[T] = Exp[T] in trait BaseExp (see Figure 5) means that conversion to symbols will take place already within the constructor methods (e.g. cos or infix_*). This fact is important because it establishes our correspondence between the evaluation order of the program generator and the evaluation order of the generated program: at the point where the generator calls toAtom, the composite definition is turned into an atomic value, i.e. its evaluation will be recorded now and played back later in the same relative order with respect to others within the closest reifyBlock invocation.

We observe that there are no concrete definition classes provided by trait Expressions. Providing meaningful data types is the responsibility of other traits that implement the interfaces defined previously (Base and its descendents). For each interface trait, there is one corresponding core implementation trait. Shown in Figure 5, we have traits BaseExp, ArithExp and TrigExp for the functionality required by the FFT example. Trait BaseExp installs atomic expressions as the representation of staged values by defining Rep[T] = Exp[T]. Traits ArithExp and TrigExp define one definition class for each operation defined by Arith and Trig, respectively, and implement the corresponding interface methods to create instances of those classes.

2.2 Implementing Optimizations

```
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]

  // block scopes
  abstract class Block[T]

  def reifyBlock[T](x: => Exp[T]): Block[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)
  }
}
```

Figure 4: Expression representation (method implementations omitted).

Some profitable optimizations, such as the global value numbering described above, are very generic. Other optimizations apply only to specific aspects of functionality, for example particular implementations of constant folding (or more generally symbolic rewritings) such as replacing computations like x * 1.0 with x. Yet other optimizations are specific to the actual program being staged. In the FFT case, Kiselyov et al. [12] describe a number of rewritings that are particularly effective for the patterns of code generated by the FFT algorithm but not as much for other programs.

What we want to achieve again is modularity, so that optimizations can be combined in a way that is most useful for a given task. To implement a particular rewriting rule (whether specific or generic), say, x * 1.0 $\to$ x, we can provide a specialized implementation of infix_* (overriding the one in trait ArithExp) that will test its arguments for a particular pattern. How this can be done in a modular way is shown by the traits ArithExpOpt and ArithExpOptFFT, which implement some generic and program specific optimizations (see Figure 6). Note that the use of x*y within the body of infix_* will apply the optimization recursively.

In essence, we are confronted with the classic "expression problem" of independently extending a data model with new data variants and new operations. There are many solutions to this problem but most of them are rather heavyweight. More lightweight implementations are possible in languages that support multi-methods, i.e. dispatch method calls dynamically based on the actual types of all the arguments. Figure 6 shows how we can achieve essentially the same (plus *deep* inspection of the arguments) using pattern matching and mixin composition, making use of the fact that composing traits is subject to linearization [15]. We package each set of arithmetic optimizations into its own trait that inherits from ArithExp and overrides the desired methods (e.g.

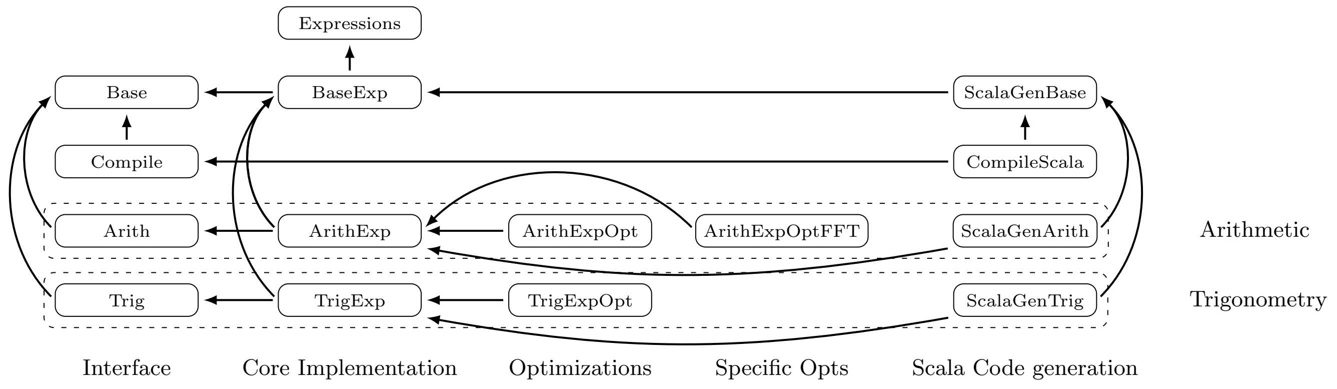

Figure 7: Component architecture. Arrows denote extends relationships, dashed boxes represent units of functionality.

```
trait BaseExp extends Base with Expressions {
  type Rep[+T] = Exp[T]
}
trait ArithExp extends Arith with BaseExp {
  implicit def unit(x: Double) = Const(x)
  case class Plus(x: Exp[Double], y: Exp[Double])
                                           extends Def[Double]
  case class Times(x: Exp[Double], y: Exp[Double])
                                           extends Def[Double]
  def infix_+(x: Exp[Double], y: Exp[Double]) = Plus(x, y)
  def infix_*(x: Exp[Double], y: Exp[Double]) = Times(x, y)
}
trait TrigExp extends Trig with BaseExp {
  case class Sin(x: Exp[Double]) extends Def[Double]
  case class Cos(x: Exp[Double]) extends Def[Double]
  def sin(x: Exp[Double]) = Sin(x)
  def cos(x: Exp[Double]) = Cos(x)
}
```

Figure 5: Implementing the interface traits from Figure 3 using the expression types from Figure 4.

```
trait ArithExpOpt extends ArithExp {
  override def infix_*(x:Exp[Double],y:Exp[Double]) = (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)
  }
}
trait ArithExpOptFFT extends ArithExp {
  override def infix_*(x:Exp[Double],y:Exp[Double]) = (x,y) match {
    case (Const(k), Def(Times(Const(l), y))) => Const(k * l) * y
    case (x, Def(Times(Const(k), y))) => Const(k) * (x * y))
    case (Def(Times(Const(k), x)), y) => Const(k) * (x * y))
    ...
    case (x, Const(y)) => Times(Const(y), x)
    case _ => super.infix_*(x, y)
  }
}
```

Figure 6: Extending the implementation from Figure 5 with generic (top) and specific (bottom) optimizations (analog of TrigExp omitted).

infix_*). When the arguments do not match the rewriting pattern, the overridden method will invoke the "parent" implementation using **super**. When several such traits are combined, the **super** calls will traverse the overridden method implementations according to the linearization order of their containing traits.

Implementing multi-methods in a statically typed setting usually poses three problems: separate type checking/compilation, ensuring non-ambiguity and ensuring exhaustiveness. The described encoding supports separate type-checking and compilation in as far as traits do. Ambiguity is ruled out by always following the linearization order and the first-match semantics of pattern matching. Exhaustiveness is ensured at the type level by requiring a default implementation, although no guarantees can be made that the default will not choose to throw an exception at runtime. In the particular case of applying optimizations, the default is always safe as it will just create an expression object.

2.3 Generating Code

Code generation is an explicit operation. For the common case where generated code is to be loaded immediately into the running program, trait Compile provides a suitable interface in form of the abstract method compile (see Figure 8). The contract of compile is to "unstage" a function from staged to staged values into a function operating on present-stage values that can be used just like any other function object in the running program.

For generating Scala code, an implementation of the compilation interface is provided by trait CompileScala. This trait extends another trait, ScalaGenBase, whose subclasses are responsible for traversing nested blocks and generating Scala code for individual definition nodes. Of course, the traversal code can also be factored out and shared by multiple code generation targets. Subclasses of ScalaGenBase are structured in a similar way as those of Base, i.e. one for each unit of functionality (see Figure 9). Code generation can omit side-effect free graph nodes that are unreachable from the final result, effectively performing a dead code elimination optimization.

The overall compilation logic of CompileScala is relatively simple: emit a class and apply-method declaration header, emit instructions for each definition node according to the schedule, close the source file, invoke the Scala compiler, load the generated class file and return a newly instantiated object of that class.

2.4 Putting it all Together

```
trait Compile extends Base {
  def compile[A,B](f: Rep[A] => Rep[B]): A=>B
}
trait CompileScala extends Compile with ScalaGenBase =>
  def compile[A,B](f: Exp[A] => Exp[B]) = {
    val x = fresh[A]
    val y = reifyBlock { f(x) }
    // emit header
    emitBlock(y)
    // emit footer
    // invoke compiler
    // load generated class file
    // instantiate object of that class
  }
}
```

Figure 8: Code generation interface and skeleton of Scala compilation component.

```
trait ScalaGenBase extends BaseExp {
  def emitBlock[T](Block[T]) = ...
  def emitNode[T](sym: Sym[T], node: Def[T]) =
    throw new Exception("node " + node + " not supported")
}
trait ScalaGenArith extends ScalaGenBase with ArithExp {
  override def emitNode(sym: Sym[T], node: Def[T]) = 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)
  }
}
```

Figure 9: Scala code generation for selected expressions.

In the previous sections, we have discussed the major ingredients of lightweight modular staging, focusing mostly on individual components. Figure 7 shows an overview of the traits encountered so far and their relationships.

Using the staged FFT implementation as part of some larger Scala program is straightforward but requires us to interface the generic algorithm with a concrete data representation. The algorithm in Figure 1 expects an array of `Complex` objects as input, each of which contains fields of type `Rep[Double]`. The algorithm itself has no notion of staged arrays but uses arrays only in the generator stage, which means that it is agnostic to how data is stored. The enclosing program, however, will store arrays of complex numbers in some native format which we will need to feed into the algorithm. A simple choice of representation is to use `Array[Double]` with the complex numbers flattened into adjacent slots. When applying `compile`, we will thus receive input of type `Rep[Array[Double]]`. Figure 10 shows how we

```
trait FFTC extends FFT { this: Arrays with Compile =>
  def fftc(size: Int) = compile { input: Rep[Array[Double]] =>
    assert(<size is power of 2>) // happens at staging time
    val arg = Array.tabulate(size) { i =>
      Complex(input(2*i), input(2*i+1))
    }
    val res = fft(arg)
    updateArray(input, res.flatMap {
      case Complex(re,im) => Array(re,im)
    })
  }
}
```

Figure 10: Extending the FFT component from Figure 1 with explicit compilation.

```
trait Functions extends Base {
  def lambda[A,B](f: Rep[A] => Rep[B]): Rep[A=>B]
  def infix_apply[A,B](f: Rep[A=>B], x: Rep[A]): Rep[B]
}
```

Figure 11: representing λ-abstractions as Scala function values (higher-order abstract syntax)

can extend trait `FFT` to `FFTC` to obtain compiled FFT implementations that realize the necessary data interface for a fixed input size.

We can then define code that creates and uses compiled FFT "codelets" by extending `FFTC`:

```
trait TestFFTC extends FFTC {
  val fft4: Array[Double] => Array[Double] = fftc(4)
  val fft8: Array[Double] => Array[Double] = fftc(8)

  // embedded code using fft4, fft8, ...
}
```

Constructing an instance of this subtrait (mixed in with the appropriate LMS traits) will execute the embedded code:

```
val OP: TestFFC = new TestFFTC with CompileScala
  with ArithExpOpt  with ArithExpOptFFT with ScalaGenArith
  with TrigExpOpt   with ScalaGenTrig
  with ArraysExpOpt with ScalaGenArrays
```

We can also use the compiled methods from outside the object:

```
OP.fft4(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)
```

Providing an explicit type in the definition `val OP: TestFFC = ...` ensures that the internal representation is not accessible from the outside, only the members defined by `TestFFC`.

3. ADDING MORE FEATURES

Many features can be added in a way that is analogous to what we have seen above but some require a bit more thought. In this section we will take a closer look at staged functions. Basic support for staged function definitions and function applications can be defined in terms of a simple higher-order abstract syntax (HOAS) [16] representation, similar to those of Carette et al. [2] and Hofer et al. [8] (see Figure 11). The idea is to provide a `lambda` operation that transforms present-stage functions over staged values (type `Rep[A] => Rep[B]`) to staged function values (type `Rep[A=>B]`). Note how this is similar to the signature of `compile`. To give an example, the staged recursive factorial function will look like this:

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

As opposed to the earlier power example, an invocation `fac(m)` will not inline the definition of `fac` but result in an actual function call in the generated code.

However the HOAS representation has the disadvantage of being opaque: there is no immediate way to "look into" a Scala function object. If we want to treat functions in the same way as other program constructs, we need a way to transform the HOAS encoding into our graph representation. We can implement `lambda(f)` to call

```
reifyBlock { f(fresh[A]) }
```

which will unfold the function definition into a Block that represents the entire computation defined by the function. But eagerly expanding function definitions is problematic.

For recursive functions, the result would be infinite, i.e. the computation will not terminate. What we would like to do instead is to detect recursion and generate a finite representation that makes the recursive call explicit. However this is difficult because recursion might be very indirect:

```
def foo(x: Rep[Int]) = {
  val f = (x: Rep[Int]) => foo(x + 1)
  val g = lambda(f)
  g(x)
}
```

Each incarnation of foo creates a new function f; unfolding will thus create unboundedly many different function objects.

To detect cycles, we have to *compare* those functions. This, of course, is undecidable in the general case of taking equality to be defined extensionally, i.e. saying that two functions are equal if they map equal inputs to equal outputs. By contrast, the standard reference equality, which compares memory addresses of function objects, is too weak for our purpose:

```
def adder(x:Int) = (y: Int) => x + y
adder(3) == adder(3)
↪ false
```

However, we can approximate extensional equality by intensional (i.e. structural) equality, which is sufficient in most cases because recursion will cycle through a well defined code path in the program text. Testing intensional equality amounts to checking if two functions are defined at the same syntactic location in the source program and whether all data referenced by their free variables is equal. Fortunately, the implementation of first-class functions as closure objects offers (at least in principle) access to a "defunctionalized" data type representation on which equality can easily be checked. A bit of care must be taken though, because the structure can be cyclic. On the JVM there is a particularly neat trick. We can serialize the function objects into a byte array and compare the serialized representations:

```
serialize(adder(3)) == serialize(adder(3))
↪ true
```

With this method of testing equality, we can implement *controlled* unfolding. Unfolding functions only once at the definition site and associating a fresh symbol with the function being unfolded allows us to construct a block that contains a recursive call to the symbol we created. Thus, we can create the expected representation for the factorial function above.

3.1 Guarantees by Construction

Making staged functions explicit through the use of lambda enables tight control over how functions are structured and composed. For example, functions with multiple parameters can be specialized for a subset of the parameters. Consider the following implementation of Ackermann's function:

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

Calling ack(m)(n) will produce a set of mutually recursive functions, each specialized to a particular value of m. For ack(2)(n), for example, we will get:

```
def ack_2(n: Int) = if (n == 0) ack_1(1) else ack_1(ack_2(n-1))
def ack_1(n: Int) = if (n == 0) ack_0(1) else ack_0(ack_1(n-1))
def ack_0(n: Int) = n+1
acc_2(n)
```

In essence, this pattern implements what is known as "polyvariant specialization" in the partial evaluation community. But unlike automatic partial evaluation, which might or might not be able to discover the *right* specialization, the use of staging provides strong guarantees about the structure of the generated code. In this case, we are guaranteed that specialization will happen for each value of m (but never for n), statically evaluating tests on values of m and inserting constants for all occurrences of m in the generated code.

Other strong guarantees can be achieved by restricting the interface of function definitions. Being of type Rep[A=>B], the result of lambda is a first-class value in the generated code that can be stored or passed around in arbitrary ways. However we might want to avoid higher-order control flow in generated code for efficiency reasons, or to simplify subsequent analysis passes. In this case, we can define a new function constructor fundef as follows:

```
def fundef[A,B](f: Rep[A] => Rep[B]): Rep[A] => Rep[B] =
  (x: Rep[A]) => lambda(f).apply(x)
```

Using fundef instead of lambda produces a restricted function that can only be applied but not passed around in the generated code (type Rep[A]=>Rep[B]). At the same time, a result of fundef is still a first class value in the code *generator*. If we do not expose lambda and apply at all to client code, we obtain a guarantee that each function call site unambiguously identifies the function definition being called and no closure objects will need to be created at runtime.

4. RELATED WORK

Static program generation approaches include C++ templates, and Template Haskell [20]. Building on C++ templates, customizable generation approaches are possible through Expression Templates [23]. An example of runtime code generation in C++ is the TaskGraph framework.Active libraries were introduced by Veldhuizen [24], telescoping languages by Kennedy at al. [11]. Specific toolkits using domain-specific code generation and optimization include FFTW [6], SPIRAL [17] and ATLAS [25].

This paper draws a lot of inspiration from the work of Kiselyov et al. [12] on a staged FFT implementation. Performing symbolic rewritings by defining operators on lifted expressions and performing common subexpression elimination on the fly is also central to their approach. LMS takes these ideas one step further by making them a central part of the staging framework itself.

Immediately related work on embedding typed languages includes that of Carette et al. [2] and Hofer et al. [8]. Lee et al. [13, 18] describe how LMS is used in the development of DSLs for high-performance parallel computing on heterogenous platforms.

Multi-Stage Programming Languages such as MetaML [22] and MetaOCaml [1] have been proposed as a disciplined approach to building code generators. These languages provide three syntactic annotations, *brackets*, *escape* and *run* which together provide a syntactic quasi-quotation facility that is similar to that found in Lisp but statically scoped and statically typed.

MSP languages make writing program generators easier and safer, but they inherit the essentially syntactic notion of combining program fragments, which incurs the risk of duplicating or reordering computation [3, 21]. Code duplication can be avoided systematically by writing the generator in continuation-passing or monadic style, using appropriate combinators to insert let-bindings in strategic places. However this is often impractical since this style significantly complicates the generator code. Another possibility is to make extensive use of side-effects in the generator part, but side-effects invalidate some of the static guarantees of MSP languages. This dilemma has been described as an "agonizing trade-off", due to which one "cannot achieve clarity, safety, and efficiency at the same time" [10].

By contrast, lightweight modular staging prevents code duplication by handling the necessary side effects inside the staging primitives, which are semantic combinators instead of syntactic expanders. Therefore, code generators can usually be written in purely functional direct style and are much less likely to invalidate safety assurances.

Another characteristic of some MSP languages is that staged code cannot be inspected due to safety considerations. Thus, domain-specific optimizations must happen on an external intermediate representation, before using the MSP primitives to generate code [12]. The burden of choosing and implementing a suitable representation is on the programmer and it is not clear how different representations can be combined or re-used.

Lightweight modular staging provides a systematic interface for adding symbolic rewritings. Safety is maintained by exposing the internal code structure only to rewriting modules but keeping it hidden from the client generator code.

Compiled embedded DSLs, as studied by Leijen et al. [14] and Elliott et al. [5], can also be implemented using MSP languages by writing an explicit interpreter and adding staging annotations in a second step [19, 4, 7]. This is simpler than writing a full compiler but compared to constructing explicit interpreters, "purely" embedded languages that are implemented as plain libraries have many advantages [9]. LMS allows as simpler approach, by starting with a pure embedding instead of an explicit interpreter. In many cases, adding some type annotations in strategic places is all that is needed to get to a staged embedding. If domain-specific optimizations are needed, new AST classes and rewriting rules are easily added.

5. ACKNOWLEDGMENTS

The development of lightweight modular staging has benefited greatly from the work on Delite in collaboration with the Stanford Pervasive Parallelism Lab, in particular Arvind Sujeeth, Hassan Chafi, Kevin Brown, HyoukJoong Lee and Kunle Olukotun. A number of members of the authors' group at EPFL have applied LMS to interesting use cases and contributed new insights or valuable extensions to the framework.

6. REFERENCES

[1] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In F. Pfenning and Y. Smaragdakis, editors, *GPCE*, volume 2830 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2003.

[2] 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.

[3] A. Cohen, S. Donadio, M. J. Garzarán, C. A. Herrmann, O. Kiselyov, and D. A. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, 62(1):25–46, 2006.

[4] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72, 2003.

[5] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[6] M. Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[7] M. Guerrero, E. Pizzi, R. Rosenbaum, K. N. Swadi, and W. Taha. Implementing dsls in metaocaml. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA Companion*, pages 41–42. ACM, 2004.

[8] 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.

[9] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[10] Y. Kameyama, O. Kiselyov, and C. chieh Shan. Shifting the stage: staging with delimited control. In G. Puebla and G. Vidal, editors, *PEPM*, pages 111–120. ACM, 2009.

[11] K. Kennedy, B. Broom, K. D. Cooper, J. Dongarra, R. J. Fowler, D. Gannon, S. L. Johnsson, J. M. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *J. Parallel Distrib. Comput.*, 61(12):1803–1826, 2001.

[12] 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.

[13] 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.

[14] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[15] M. Odersky and M. Zenger. Scalable component abstractions. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 41–57. ACM, 2005.

[16] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.

[17] 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.

[18] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. In *DSL*, pages 93–117, 2011.

[19] T. Sheard, Z.-E.-A. Benaissa, and E. Pasalic. Dsl implementation using staging and monads. In *DSL*,

pages 81–94, 1999.

[20] T. Sheard and S. L. P. Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[21] K. N. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In J. Hatcliff and F. Tip, editors, *PEPM*, pages 160–169. ACM, 2006.

[22] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[23] T. Veldhuizen. Expression templates, C++ gems, 1996.

[24] T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[25] 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.