EPFL-REPORT-149131

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

Tiark Rompf Martin Odersky

École Polytechnique Fédérale de Lausanne (EPFL)
{first.last}@epfl.ch

Abstract

Software engineering demands generality and abstraction, performance demands specialization and concretization. Generative programming can provide both, but developing high-quality program generators takes a large effort, even if a multi-stage programming language is used.

We present *lightweight modular staging*, a library-based multi-stage programming approach that breaks with the tradition of syntactic quasi-quotation and instead uses only types to distinguish between binding times. Through extensive use of component technology, lightweight modular staging makes an optimizing compiler framework available at the library level, allowing programmers to tightly integrate domain-specific abstractions and optimizations into the generation process.

We argue that lightweight modular staging enables a form of *language virtualization*, i.e. allows to go from a pure-library embedded language to one that is practically equivalent to a stand-alone implementation with only modest effort.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Code Generation, Multi-stage programming, Domain-specific languages, Language Virtualization

1. Introduction

Building and managing complex software systems is only possible by generalizing functionality and abstracting from particular scenarios. Achieving performance, on the other hand, demands 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.

In some cases program generation is not only a means to obtain peak performance but a necessity. Server-side web applications need to output HTML and JavaScript. Programs that use GPU hardware as a coprocessor need to load CUDA or OpenCL code dynamically. In very simple cases this entire embedded code can be provided statically. But as soon as modularity is needed on the embedded level, at least parts of the code have to be generated at runtime. In this case, code generation is heterogenous: the generator is written in a language different from the generation target.

Code generation can be either static or dynamic. A static code generation system that is widely used are C++ templates [39]. Dynamic code generation is inherently more flexible because code can be specialized with respect to parameters only available at runtime. It can be re-generated based on profiling information. Multiple generated versions can be tested and the best one selected. For achieving utmost performance, this adaptivity can be important (see e.g. Spiral [33] or ATLAS [45]).

Support for building dynamic program generators in a systematic way is less widely available. Multi-stage programming languages [38] mitigate the burden to some extent but building "active" libraries [42] or domain-specific languages (DSLs) that incorporate dynamic code generation takes a huge effort nonetheless. Many successful generative toolkits such as ATLAS [45] use ad-hoc techniques.

In this paper, we explore a dynamic code generation approach that we dub *lightweight modular staging* (LMS). The concept of staging goes back at least to Jørring and Scherlis [23], who observed that 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.

The classical introductory example is to specialize the power function for a given exponent. Doing so might be worthwhile if a program will take many different numbers to the same power. Considering the usual implementation,

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

we want to turn the base `b` into a *staged* expression. The central idea of LMS (following [4]) is to reflect this change of binding time by changing `b`'s declared type from `Double` to `Rep[Double]`, meaning that `b` *represents* a computation that will yield a `Double` in the next stage. We also change the return type accordingly. In addition, we need to be able 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 in a trait:

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

A trait is similar to a class but can be used in mix-in composition [31]. The notation **`this:`** `Arith` signifies that whenever an instance of `PowerA` is created, an instance of a concrete (but unspecified) subclass of `Arith` (see Figure 3) must be mixed in, too. Inside the trait `PowerA`, the **`this`** reference will have type `PowerA` **`with`** `Arith` instead of just `PowerA`, making all members of `Arith` accessible within `PowerA` as well. This is all we need to change.

[Copyright notice will appear here once 'preprint' option is removed.]

The main characteristics of lightweight modular staging can be summarized as follows:

- binding-times are distinguished only by types; no special syntactic annotations are required
- given a sufficiently expressive language, the whole framework can be implemented as a library (hence *lightweight*)
- staged code is "very shallowly" embedded into the program generator; staged expressions inherit the static scope of the generator and if the generator is well-typed so is the generated code
- 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
- 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*)
- likewise, different code generation targets can be supported (heterogeneous staging); their implementations can share common code
- in the homogeneous case, objects that are live within the generator's heap can be accessed from generated code (cross-stage persistence)
- data types representing staged expressions can be hidden from client code (making rewrites safe that preserve only semantic equality) but exposed to modules that implement the rewriting
- common subexpression elimination/value numbering is handled globally within the framework; there is no danger of code duplication
- "unstaging", i.e. compilation and loading of staged functions is an explicit operation, independent of running the compiled code; program generators have full control over when compilation happens and how compiled code is re-used

Many of the listed points are found in other code generation approaches as well, but to the best of our knowledge, no existing system combines them all. We believe that this combination occupies a "sweet spot" in the design space (see Section 4 for a detailed comparison with related work) — most prominently by significantly reducing the effort required to go from a naively implemented algorithm to an optimizing program generator.

Lightweight modular staging provides many of the benefits of using a dedicated multi-stage programming language [38] such as MetaOCaml, in particular concerning well-formedness and type safety, but goes beyond that in systematically preventing code duplication and providing a clean interface for incorporating generic and customized optimizations.

LMS is a key technique in our work to develop high-performance parallelizable DSLs. In previous work [5], we defined criteria for what we call *language virtualization*, saying that a general-purpose language is virtualizable iff it can provide an environment to embedded languages that makes them essentially identical to corresponding stand-alone language implementations in terms of *expressiveness* (being able to express a DSL in a way which is natural to domain specialists) *performance* (leveraging domain knowledge to produce optimal code), and *safety* (domain programs are guaranteed to have certain properties implied by the DSL), while at the same time requiring only modestly more development *effort* than implementing a simple, pure-library embedding. [1]

One ingredient of LMS is a *finally tagless* [4] or *polymorphic* [21] language embedding, which ensures *expressiveness* and *safety*. Hofer et al. [21] show that a polymorphic embedding can be constructed from a pure embedding [22] with acceptable *effort*. LMS offers a systematic way to also obtain *performance* (by means of its optimization interface) while keeping the *effort* under control (by enabling modular composition, re-use and extension of DSL building blocks, including optimizations). The novel aspect is that despite the component architecture, LMS uses a uniform (but extensible) language representation for all DSL components instead of offering a choice of representations between which translations or layerings would need to be defined. This is achieved by solving the resulting "expression problem" [43] of independently adding data type variants and operations via an encoding of multi-methods (open generic functions) into a combination of mixin-composition and pattern matching.

1.1 Organization

We present lightweight modular staging using Scala as the host language. While we use a number of Scala's advanced features extensively (operator overloading, implicits, abstract types and type constructors, pattern matching, mixin-composition), LMS is not inherently tied to Scala and could be implemented in other expressive languages as well. Features that Scala lacks but other languages provide (e.g. built-in multi-methods or transparent creation of forwarder objects) could even simplify the implementation.

The rest of this paper is structured as follows: Section 2 describes the basic LMS setup in detail for a subset of language features. Section 3 outlines how more features can be added. Section 4 discusses related work. Section 5 concludes.

2. Lightweight Modular Staging

In the same way as the power function shown in the introduction, we can stage far more interesting and practically relevant programs, such as the fast fourier transform (FFT). A staged FFT, implemented in MetaOCaml, has been presented by Kiselyov et al. [27]. 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 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 trans-

[1] a virtualizable language is also a universal language according to the definition of Veldhuizen [42] but virtualization adds the *effort* criterium

```
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.

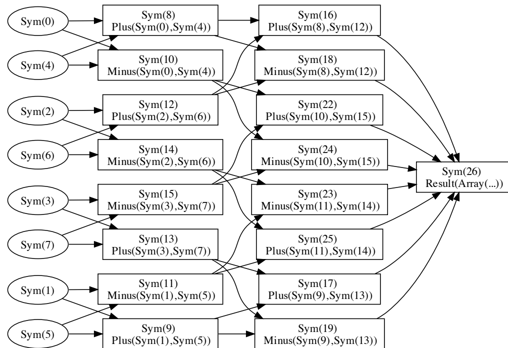


Figure 2. Computation graph for size-4 FFT. Auto-generated from staged code in Figure 1.

```
trait Base {
  type Rep[+T]
}
trait Arith extends Base {
  implicit def unit(x: Double): Rep[Double]
  def infix_+(x: Rep[Double], y: Rep[Double]): Rep[Double]
  def infix_*(x: Rep[Double], y: Rep[Double]): Rep[Double]
  ...
}
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. `infix_` methods not presently legal Scala, but can be encoded with implicits. For simplicity, operations are defined for `Double` only.

```
trait BaseStr extends Base {
  type Rep[+T] = String
}
trait ArithStr extends Arith with BaseStr {
  implicit def unit(x: Double) = x.toString
  def infix_+(x: String, y: String) = "(%s+%s)".format(x,y)
  def infix_*(x: String, y: String) = "(%s*%s)".format(x,y)
}
trait TrigStr extends Trig with BaseStr {
  def sin(x: String) = "sin(%s)".format(x)
  def cos(x: String) = "cos(%s)".format(x)
}
```

Figure 4. Implementing the interface traits of Figure 3, representing staged code as strings.

formations without any further changes to the code in Figure 1, just by mixing in the trait `FFT` with a few others.

Before considering specific optimizations, however, 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 [30] (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 `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 [5]. 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. The embedding is *tagless* (i.e. method resolution happens at compile time without runtime dispatch overhead) [4] and *polymorphic* [21], 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 [37].

2.1 Representing Staged Code: as Strings (bad)

With the aim of generating code, we might be tempted to represent staged expressions uniformly as strings. We could achieve this with traits `BaseStr` and `ArithStr` shown in Figure 4 (we neglect `Trig`). Figure 5 shows how these traits can be put to use to generate code (a simple variant, that is) from two different staged power functions. The basic task is to assemble the desired traits into an actual object:

```
object PowerStrA extends PowerA with ArithStr
```

Mixing in `ArithStr` will satisfy the previously defined self-type requirement of `PowerA` since `ArithStr` extends `Arith` (`BaseStr` and `Base` are analogous) and will install `ArithStr`s implementation of the interface defined in `Arith`.

Figure 5 also reveals the main drawback of using strings or any other solely expansion-based representation: unrestricted code duplication. The generated code will re-evaluate the operation `(x0+x1)` four times. Surprisingly, choosing a better algorithm can

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


new PowerA with ArithStr {
  println {
    power("(x0+x1)",4)
  }
}

// result:
((x0+x1)*((x0+x1)*((x0+x1)*((x0+x1)*1.0))))
```

```
trait PowerB { this: Arith =>
  def power(b: Rep[Double], x: Int): Rep[Double] =
    if (x == 0) 1.0
    else if ((x&1) == 0) { val y = power(b, x/2); y * y }
    else b * power(b, x - 1)
}

new PowerB with ArithStr {
  println {
    power("(x0+x1)",4)
  }
}

// result:
((((x0+x1)*1.0)*((x0+x1)*1.0))*(((x0+x1)*1.0)*((x0+x1)*1.0)))
```

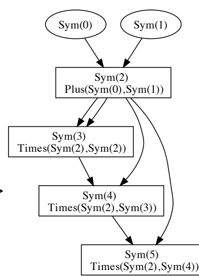
Figure 5. Two algorithms to implement the power function. Using strings as code representation results in code duplication and undoes the improvement obtained by re-using intermediate results.

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


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

new PowerA2 with CompileScala
  with ArithExpOpt with ScalaGenArith

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




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


trait PowerB2 extends PowerB { this: Compile =>
  val p4 = compile { x: Rep[Double] =>
    power(x + x, 4)
  }
  // use compiled function p4 ...
}

new PowerB2 with CompileScala
  with ArithExpOpt with ScalaGenArith

// generated code:
class Anon$2 extends ((Double)=>(Double)) {
  def apply(x0:Double): Double = {
    val x1 = x0+x0
    val x2 = x1*x1
    val x3 = x2*x2
    x3
  }
}
```

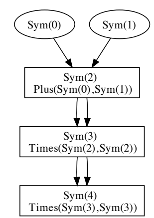


Figure 6. Using expression trees instead of strings and adding symbolic rewritings removes the `* 1.0` operations, prevents code duplication and mirrors algorithmic improvement in generated code. Code to output graph (top), code to generate and load Scala code (bottom).

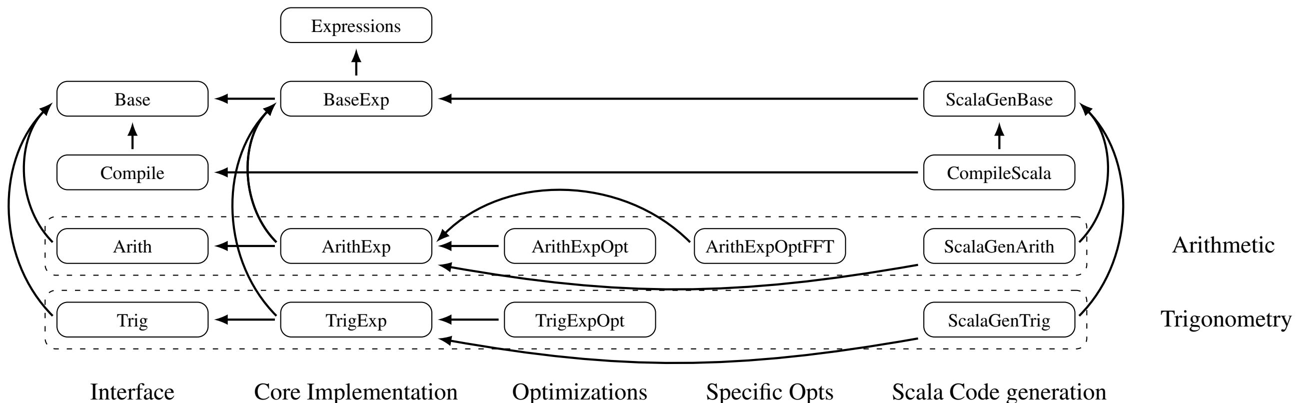


Figure 7. Component architecture. Arrows denote **extends** relationships, dashed boxes represent units of functionality.

make matters worse. The repeated-squaring power algorithm in PowerB, which normally reduces the overall number of multiplications to $O(\log x)$, generates less efficient code than the linear algorithm in PowerA. Even if the target compiler would remove the trivial `* 1.0` operations, the seemingly clever algorithm would not have had a positive effect. The problem is that instead of re-using the results of intermediate computations, the computations themselves are duplicated. This effect of "undoing" value binding and memoization is characteristic for all inherently syntactic staging approaches and has been studied in the context of MSP languages at length [13, 36].

Moreover, there is no evident way of implementing more elaborate optimizations that need to analyze staged expressions in a semantic manner.

2.2 Representing Staged Code: as Graphs (good)

Instead of strings we choose a representation based on expression trees, or, more exactly, a "sea of nodes" [7] 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 8.

There are two categories of objects involved: expressions, which are atomic (subclasses of Exp: constants and symbols) and definitions, which represent composite operations (subclasses of Def, to be provided by other components). There is also a "gensym" operator fresh that creates fresh symbols.

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) [18]. Trait Expressions provides methods to find a definition given a symbol or vice versa. The extractor object [15] Def allows to pattern-match on the definition of a given symbol, a facility that is used for implementing rewritings (see below).

Through the implicit conversion method toAtom, a definition can be used anywhere an atomic expression is expected. Doing so will search the already encountered definitions, which are kept in an internal table (omitted in Figure 8), for a structurally equivalent one. If a matching previous definition is found, its symbol will be returned. Otherwise the definition is seen for the first time. It will be associated with a fresh symbol and saved for future reference. In effect, this simple scheme provides a powerful global value numbering (common subexpression elimination) optimization that effectively prevents generating duplicate code. Since all operations in interface traits such as Arith are defined to return Rep types, defining Rep[T] = Exp[T] in trait BaseExp (see Figure 9) means that conversion to symbols will take place already within those methods, making sure that the created definitions are actually registered.

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 9, 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.

2.3 Implementing Optimizations

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 imple-

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

  def fresh[T]: Sym[T]

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

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

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

  // pattern match on definition of a given symbol
  object Def {
    def unapply[T](s: Sym[T]): Option[Def[T]] =
      findDefinition(s)
  }
}
```

Figure 8. Expression representation (method implementations omitted).

```
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)
}
```

Figure 9. Implementing the interface traits from Figure 3 using the expression types from Figure 8.

```
trait ArithExpOpt extends ArithExp {
  override def infix_*(x:Exp[Int],y:Exp[Int]) = (x,y) match {
    case (Const(x), Const(y)) => Const(x * y)
    case (x, Const(1)) => x
    case (Const(1), y) => x
    case _ => super.infix_*(x, y)
  }
}
trait ArithExpOptFFT extends ArithExp {
  override def infix_*(x:Exp[Int],y:Exp[Int]) = (x,y) match {
    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 10. Extending the implementation from Figure 9 with generic (top) and specific (bottom) optimizations.

mentations 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, a number of rewritings are described by Kiselyov et al. [27] 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, such 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` → `x`, we have to 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 10).

In essence, we are confronted with the classical expression problem of independently extending a data model with new data variants and new operations [43]. 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 10 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 [31]. We package each set of arithmetic optimizations into its own trait that inherits from `ArithExp` and overrides the desired methods (e.g. `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.

Comparing expression-graph realizations of the power function (see Figure 6) to the previous realizations based on strings (see Figure 5), we observe much better results. The generated code no longer contains any trivial operations and no duplicate code. Moreover, the staged code produced by the repeated squaring algorithm is exactly what one would expect. The optimized FFT computation graph was already shown in Figure 2.

2.4 Generating Code

Code generation is an explicit operation. Figure 6 shows how it is invoked for the power function. 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 11). 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 to linearize the internal dependency graph into a flat code representation and generate Scala code for individual definition nodes. Subclasses of `ScalaGenBase` are structured in a similar way as those of `Base`, i.e. one for each unit of functionality (see Figure 12). Generating a valid schedule is straightforward in the code model we have been considering so far (no staged branches or conditionals, no staged function definitions): all that needs to be done is sort the graph nodes in reverse topological order. More powerful code models will need some form of dominator computation such as described by Click [6]. Graph nodes unreachable from the final result node are discarded from the schedule, i.e. any computation whose result is never used is removed; another example of a powerful generic optimization.

```
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 = f(x)
    // emit header
    for ((sym, node) <- buildSchedule(y))
      emitNode(sym, node)
    // emit footer
    // invoke compiler
    // load generated class file
    // instantiate object of that class
  }
}
```

Figure 11. Code generation interface and skeleton of Scala compilation component.

```
trait ScalaGenBase extends BaseExp {
  def buildSchedule(Exp[_])): List[(Sym[_], Def[_])] = ...
  def emitNode(sym: Sym[_], node: Def[_]) =
    throw new Exception("node " + node + " not supported")
}
trait ScalaGenArith extends ScalaGenBase with ArithExp {
  override def emitNode(sym: Sym[_], node: Def[_]) = node match {
    case Plus(a,b) =>  println("val %s = %a + %b".format(sym,a,b))
    case Times(a,b) => println("val %s = %a * %b".format(sym,a,b))
    case _ => super.emitNode(sym, rhs)
  }
}
```

Figure 12. Scala code generation for selected expressions.

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. Examples of code generated for the power function are shown in Figure 6.

An alternative "code generation" strategy is outputting the intermediate representations as GraphViz code, suitable for producing graphical output. This is how the graphs in this paper were generated. The implementation is straightforward and a small example of its use is also shown in Figure 6.

2.5 Putting it all Together

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. Figure 6 already showed a simple end-to-end implementation with the power function as example.

For the FFT, putting LMS to use is only slightly more complex. One obstacle is that the FFT algorithm in Figure 1 expects an array of `Complex` objects as input, each of which contains fields of type `Rep[Double]`. When applying `compile`, however, we will receive input of type `Rep[Array[Double]]`, assuming we want to generate

```
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 13. Extending the FFT component from Figure 1 with explicit compilation.

functions that operate on arrays of Double (with the complex numbers flattened into adjacent slots). Thus, we will extend trait FFT to FFTC (see Figure 13), importing support for staged arrays and Compile. The implementation of staged arrays is straightforward and omitted for brevity.

We can define code that uses compiled FFT "codelets" by embedding it in a subtrait of 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

Up to now we have been working with a very simple language at the staged level. Prominent missing features are side effects, control flow (conditionals, loops) and function definitions. There is not sufficient space to explain their implementations in full detail. Large parts are standard compiler technology and orthogonal to the choice between LMS and a stand-alone compiler. We will visit only the main points in this section to give an overall idea of how implementations can be approached.

3.1 Side Effects and Control Flow

In Section 2, all staged code was pure. Many practical programs, however, need to incur side-effects, especially if the goal of staging is improved performance. We can extend the previous model to include effectful computations in a relatively simple way. The basic idea is to make all effects explicit and include effect-dependencies in the graph-based representation besides the data dependencies.

We will maintain a *current state* in a mutable fashion, taking the view that state is an abstraction of an effect history. How this abstraction is actually defined can be controlled by mixing in a suitable trait. In the simplest case, the current state is a list of previous effects.

A suitable programming model is suggested by the notion of monadic reflection and reification [16, 17]. An effectful operation needs to be *reflected* at the point where its effect should occur. Reflection amounts to updating the current state in a mutable fashion

```
trait Parsers { this: Matching =>
  type Input = List[Char]

  abstract class Parser extends (Input => Result) {
    def ~(p: =>Parser) =  new Parser { // sequence
      def apply(in: Rep[Input]) = this(in) switch {
        case SuccessR(rest) => p(rest)
      } orElse {
        case _ => FailureR()
      } end
    }
    def |(p: =>Parser): Parser = ... // alternative
  }

  implicit def acceptChar(c: Char): Parser = ...
  implicit def acceptString(s: String) =
    s.map(acceptChar).reduceLeft(_ ~ _)
}
trait TestParsers extends Parsers {
  val phraseA = "scala" ~ '␣' ~ "rules"
  val phraseB = "scala" ~ '␣' ~ "rocks"
  val main = phraseA | phraseB
}
```

Figure 14. Staged parser combinators. Matching alternatives with a common prefix.

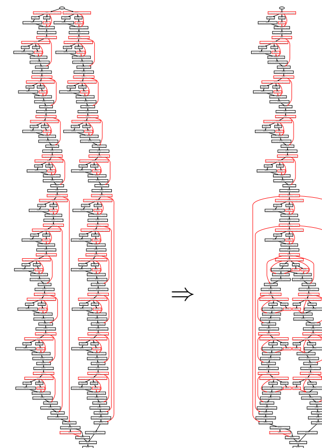


Figure 15. Resulting computations. Generic value numbering optimization (disabled on the left) prevents unnecessary backtracking.

to include the new effect. Not surprisingly, an effect can thus be seen as defining a state transition. How exactly this transition works is again customizable. Optimizing rewritings on the effect level can be implemented in the same manner as can be done for the value level.

The counterpart of reflection is *reification*. Reifying the effects of a block of code amounts to executing the code with an empty current state, and returning a representation of the result value together with the resulting state, e.g.:

```
def print(x: Exp[String]): Exp[Unit] = reflect(Print(x))
reify {
  print("A")
  print("B")
  3+4
}
↪ Reified(Const(7), List(Print(Const("A"), Const("B"))))
```

Control flow can also be described in terms of effects (after all a jump instruction modifies the program counter). To implement conditionals, we can use the notion of an *abort* effect, possibly incurred by the operation Test. A conditional expression `if (c) a else b` will be represented as:

```
reflect(OrElse(reify { reflect(Test(c)); a }, reify(b)))
```

```
trait Functions extends Base {
  def lambda[A,B](f: Rep[A] => Rep[B]): Rep[A=>B]
  def app[A,B](f: Rep[A=>B], x: Rep[A]): Rep[B]
}
trait FunctionsExp extends Functions with BaseExp {
  case class Lambda[A,B](f: Exp[A] => Exp[B]) extends Def[A=>B]
  case class Apply[A,B](f: Exp[A=>B], x: Exp[A]) extends Def[B]

  def lambda[A,B](f: Exp[A] => Exp[B]) = Lambda(f)
  def app[A,B](f: Exp[A=>B], x: Exp[A]) = Apply(f,x)
}
```

Figure 16. representing λ-abstractions as Scala function values (higher-order abstract syntax)

The `OrElse` operation is similar to Φ-nodes in customary SSA representations but captures the priority of the then-part.

This notion of representing conditionals extends naturally to more complicated structures such as pattern matching. An interesting aspect is that effect nodes are subject to the same value numbering optimization as data nodes. An example, which we present without going into the details, is a staged implementation of parser combinators (see Figure 14). Using similar combinators in their unstaged form can be very expensive because of unnecessary backtracking. In the example, the grammar consists of two alternatives that share a common prefix. Looking at the computation graph of the staged program (see Figure 15), we observe that backtracking is automatically removed.

Code generation for code including conditionals and side effects is more involved than what is shown in Section 2. The graph representation no longer corresponds to ANF since conditionals can appear conceptually "within" other expressions. This is not a problem if the code generation target language is expressive enough. For targeting simpler languages however, more work needs to be done. It should be fairly straightforward, though by no means trivial, to extract a customary control flow graph (CFG) from the representation described above (this has not been implemented yet). With a CFG and a separation into flat basic blocks at hand, almost any target should be feasible.

3.2 Functions and Recursion

Basic support for staged function definitions and function applications can be defined in terms of a simple higher-order abstract syntax (HOAS) [32] representation, similar to those of Carette et al. [4] and Hofer et al. [21] (see Figure 16). Alternatively, if we are interested mainly in first-order functions (which is often the case, since one goal of staging is to translate away the abstraction offered by higher-order functions at the meta-program level), we can hide function definitions inside the representations of conditionals or pattern matching. In the pattern matching interface described above, pattern alternatives are reified as instances of `PartialFunction`, which is a subclass of function values. One avenue is to stage these pattern alternatives. The heuristic here is that user-defined functions will do some form of matching on their arguments anyways. If staged functions are implemented that way, `lambda` and `app` do not leak into client code. An example is the staged factorial function in Figure 17.

Whether we use `lambda` and `app` directly or not, 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 analyze functions in the same way as other program constructs, we need a way to transform the HOAS encoding into our flat graph representation. For a HOAS term `Lambda(f)`, we can call `f(fresh[A])` to "unfold" the function definition. The result is a symbol that represents the entire computation defined by the function. But too 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 is detect recursion and generate a finite representation that makes the recursive call explicit. However this is difficult because recursion might be very indirect:

```
trait Fac { this: Matching =>
  def fac(n: Rep[Int]): Rep[Int] = n switch {
    case n if n guard 0 => 1
  } orElse {
    case n => n * fac(n - 1)
  }
}
```

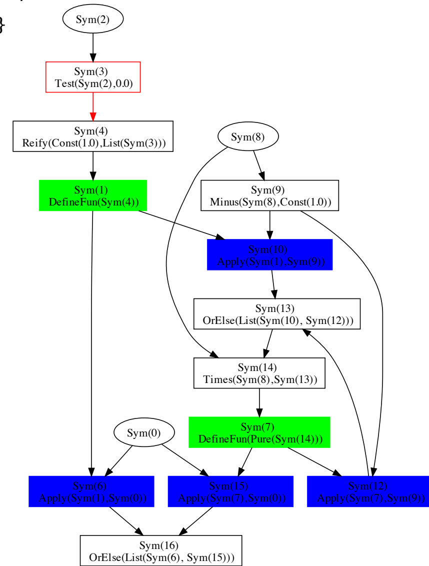


Figure 17. Staged factorial function (top). Computation unfolded once (bottom).

```
def foo(x: Rep[Int]) = {
  val f = (x: Rep[Int]) => foo(x + 1)
  app(lambda(f), 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. The standard reference equality, on the other hand, 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 in most cases turns out to be sufficient. 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" [12] 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. The result of unfolding the factorial function once (at the definition site) is again shown in Figure 17 (bottom).

3.3 Cross-Stage Persistence

Cross-stage persistence (CSP) means making objects that are live at the generator stage available to the generated program [38]. In general, this is only applicable if the generated code is to be loaded and executed while the previous stage is still available, and if the code generation target allows to call back into the generator. This would not be the case for, say, OpenCL GPU code produced from a generator written in Scala, but homogenous setups are fine. Restricted forms of heterogeneous CSP are also feasible, e.g. for immutable data that has a corresponding target-language counterpart.

In terms of implementation, general CSP can be achieved by generalizing the implicit `unit` method from trait `Arith` (see Figure 3) to lift arbitrary values into the staged representation instead of just `Doubles`:

```
implicit def unit[T](x: T): Rep[T]
```

For all lifted objects that are not primitives, we can then create a corresponding definition node of class `External`:

```
case class External[T](x: T) extends Def[T]
```

Primitives retain their representation as objects of class `Const`.

During code generation, we map each `External` node to a field in the generated Scala class. When instantiating the code object, these fields are initialized with the corresponding external references. This approach is similar to a classic closure conversion.

The Scala implementation does not currently provide an automatic lifting of *all* operations for a given type of object. Operations must be "white-listed" by providing staged versions explicitly, which can be tedious if there is no pre-fabricated component that can be readily mixed in. On the other hand this implies that programmers have full control over what operations are available to staged programs.

4. Related Work

Static meta-programming approaches include C++ templates [39], and Template Haskell [34]. Building on C++ templates, customizable generation approaches are possible through Expression Templates [40], e.g. used by Blitz++ [41]. An example of runtime code generation in C++ is the TaskGraph framework [1]. Active libraries were introduced by Veldhuizen [42], telescoping languages by [26]. Specific toolkits using domain-specific code generation and optimization include FFTW [19], SPIRAL [33] and ATLAS [45].

This paper draws a lot of inspiration from the work of Kiselyov et al. [27] 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. [4] and Hofer et al. [21]. Chafi et al. [5] 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 [38], MetaOCaml [2] and Mint [44] 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. On one hand, MSP languages transparently support staging of all language constructs, where LMS components have to be provided explicitly. On the other hand, the syntactic MSP approach incurs the risk of duplicating code [3, 8, 13, 36]. 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. Often this is impractical since monadic style or CPS significantly complicates the generator code. The other suggested solution is to make extensive use of side-effects in the meta-program, either in the form of mutable state or by using delimited control operators [10, 11]. However, side-effects pose serious safety problems and invalidate much of the static guarantees of MSP languages. This dilemma is described as an "agonizing trade-off", due to which one "cannot achieve clarity, safety, and efficiency at the same time" [25]. Only very recently have type-systems been devised to handle both staging and effects [24, 25, 44]. They are not excessively restrictive but not without restrictions either. Mint [44], a multi-stage extension of Java, restricts non-local operations within escapes to **`final`** classes which excludes much of the standard Java library.

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 cause scope extrusion or invalidate safety assurances in other ways. Even though less likely, scope extrusion can happen in the LMS setting as well, e.g. if the argument of the function passed to `compile` escapes its dynamic scope. Combining LMS with the type system of Westbrook et al. [44] would be an interesting avenue for future research, if utmost security is strived for.

Another central characteristic of MSP languages is that staged code cannot be inspected due to safety considerations [37]. This implies that domain-specific optimizations must happen before code generation. One approach is thus to first build an intermediate code representation, upon which symbolic computation is performed, and only then use the MSP primitives to generate code [27]. The burden of choosing and implementing a suitable intermediate representation is on the programmer. It is not clear how different representations can be combined or re-used. In the limit, programmers are tempted to use a representation that resembles classic abstract syntax trees (AST) since that is the most flexible. At that point, one could argue that the benefit of keeping the actual code representation hidden has been largely defeated.

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 and Meijer [28] and Elliott et al. [14], can also be implemented using MSP languages by writing an explicit interpreter and adding staging annotations in a second step [9, 20, 35]. This is simpler than writing a full compiler but compared to constructing explicit interpreters, purely embedded languages have many advantages [22]. LMS allows as simpler approach, by starting with a pure embedding instead of an explicit interpreter. In simple cases, adding some type annotations in strategic places is all that is needed to get to a staged embedding [21]. If domain-specific optimizations are needed, new AST classes and rewriting rules are easily added.

5. Conclusions

In this paper we have presented lightweight modular staging, a library-based dynamic code generation approach. In particular we have shown how LMS complements the notion of polymorphic DSL embedding [21] with a systematic interface for domain-specific optimizations.

Acknowledgments

The authors would like to thank Hassan Chafi, Zach Devito, Ingo Maier and Adriaan Moors for insightful discussions and/or helpful comments on draft versions of this paper.

References

[1] 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 Lengauer et al. [29], pages 291–306.

[2] 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.

[3] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2005.

[4] 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.

[5] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Technical Report 148814, EPFL, 2010.

[6] C. Click. Global code motion / global value mumbering. In *PLDI*, pages 246–257, 1995.

[7] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *Intermediate Representations Workshop*, pages 35–49, 1995.

[8] 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.

[9] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell, and c++. In Lengauer et al. [29], pages 51–72.

[10] O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[11] O. Danvy and A. Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4): 361–391, 1992.

[12] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP*, pages 162–174. ACM, 2001.

[13] J. L. Eckhardt, R. Kaiabachev, K. N. Swadi, W. Taha, and O. Kiselyov. Practical aspects of multi-stage programming. Technical Report TR05-451, Rice University, 2004.

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

[15] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 273–298. Springer, 2007.

[16] A. Filinski. Representing monads. In *POPL*, pages 446–457, 1994.

[17] A. Filinski. Monads in action. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 483–494. ACM, 2010.

[18] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

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

[20] 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.

[21] 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.

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

[23] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *POPL*, pages 86–96, 1986.

[24] Y. Kameyama, O. Kiselyov, and C. chieh Shan. Closing the stage: from staged code to typed closures. In R. Glück and O. de Moor, editors, *PEPM*, pages 147–157. ACM, 2008.

[25] 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.

[26] 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.

[27] 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.

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

[29] C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, 2004. Springer.

[30] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In G. E. Harris, editor, *OOPSLA*, pages 423–438. ACM, 2008.

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

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

[33] 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.

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

[35] T. Sheard, Z.-E.-A. Benaissa, and E. Pasalic. Dsl implementation using staging and monads. In *DSL*, pages 81–94, 1999.

[36] 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.

[37] W. Taha. A sound reduction semantics for untyped cbn multi-stage computation. or, the theory of metaml is non-trivial (extended abstract). In *PEPM*, pages 34–43, 2000.

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

[39] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

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

[41] T. L. Veldhuizen. Arrays in blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *ISCOPE*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998.

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

[43] P. Wadler. The expression problem. Posted on the Java Genericity mailing list, 1998.

[44] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *PLDI*, 2010.

[45] 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.