Reflections on LMS: Exploring Front-End Alternatives

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

Abstract

Metaprogramming techniques to generate code at runtime in a general-purpose meta-language have seen a surge of interest in recent years, driven by the widening performance gap between high-level languages and emerging hardware platforms. In the context of Scala, the LMS (Lightweight Modular Staging) framework has contributed to “abstraction without regret”—high-level programming without performance penalty—in a number of challenging domains, through runtime code generation and embedded compiler pipelines based on stacks of DSLs. Based on this experience, this paper crystallizes some of the design decisions of LMS and discusses potential alternatives, which maintain the underlying spirit but differ in implementation choices: specifically, strategies for realizing more flexible front-end embeddings using type classes instead of higher-kinded types, and strategies for type-safe metaprogramming with untyped intermediate representations.

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

Keywords Multi-stage programming, domain-specific languages, intermediate representation

1. Introduction

Recent years have seen a surge of interest in staging and related metaprogramming techniques, driven by the widening gap between high-level languages and emerging hardware platforms. In the context of Scala, The LMS (Lightweight Modular Staging) \[61, 62\] framework has seen applications in DSLs for heterogeneous parallelism (Delite \[8, 65, 44, 81\]), machine learning (OptiML \[80\]), library generation for numeric kernels (Spiral \[52, 77\]), hardware generation \[24, 25\], SQL database engines \[40, 59\], protocol and data format parsers \[36\], and recently for safe systems development by generating verifiable C code with contracts in a specification language \[2\].

With such a broad variety of applications, spanning code generation targets from JavaScript, Scala, C/C++, CUDA, all the way to Verilog, the desire to change different aspects of LMS has come up from time to time. However, most such change attempts that arose in a given use case have turned out to break key properties necessary for at least one other key use case of LMS. Hence, the design of LMS as it has evolved seems to be a local optimum—but it seems hard to believe that this particular point in the design space should be globally optimal, in particular since there are recurring pain points, for example repetitive boilerplate for DSL definitions and excessive Scala compile times.

What are then the constraints on the design space? We identify five key aspects that make LMS what it is and which we deem essential for embedded DSL compiler frameworks to achieve applicability on a similar broad scale as LMS:

• Type-based embedding enables mixing present-stage and future-stage computations seamlessly in a single program, while keeping the stages stratified. Type-based embedding further enables us to abstract over the stage.
• True multi-stage semantics guarantees, in constrast to syntactic template expansion, that evaluation order within each stage follows the normal call-by-value rules.
• Graph-based IR, where computations can move freely, enables sophisticated and generic optimizations, implemented once and for all, for many DSLs.
• Horizontal extensibility enables users to add custom IR nodes and optimizations, based on eager exhaustive rewriting, to an existing language definition.
• Vertical extensibility enables users to add analysis and transformation passes, potentially introducing new intermediate languages. Such transformations are implemented as staged interpreters.
Together, these facilities enable high-bandwidth DSL compiler pipelines as illustrated in Figure 1. The left-hand side depicts a general-purpose compiler, whose generic abstractions like objects, modules, or higher-order functions, can be used to implement domain-specific abstractions like matrices or graphs, but do not allow the compiler to reason about such domain-specific objects, which leads to a semantic bottleneck.

The right-hand of Figure 1 depicts a DSL compiler pipeline consisting of multiple IR levels. At the top, we have a considerable variety of DSLs or domain specific abstractions, e.g. an IR that models matrices, graphs, or queries in relational algebra, with a corresponding set of domain-specific optimizations, e.g. rewrites that exploit algebraic identities. Once these optimizations are applied exhaustively, the program is lowered to the next abstraction level, for example, arrays and while loops. On this level, there is a different set of optimizations (e.g. tiling, loop fusion), which is again applied exhaustively before lowering to a variety of hardware-specific targets. In general, the different abstraction levels proceed from mostly declarative at the programmer-facing side towards more imperative at the hardware-facing side. To avoid the bottleneck of a general purpose compiler, it is important that the stack can be extended vertically, for example by adding new front-ends, back-ends, or IR levels in the middle, and also that the IR on each level can be extended horizontally with new data types and optimizations. Note that there is an explicit distinction between lowering transforms and optimizations in this model [63], a distinction which is absent in most traditional compiler frameworks.

Finally, generic optimizations such as dead code elimination, common subexpression elimination, or code motion are as important as domain-specific optimizations, and they need to be available on all levels of the stack. With such an architecture, DSL compilers are easy to construct from existing stacks, by horizontal or vertical extension.

Note that even though LMS primarily targets embedded DSLs, it is easy to add textual front-ends, as has been done, for example, with regular expressions [63] and SQL [59].

Based on these observations, this paper makes the following contributions:

- We sketch the design space of LMS as outlined above and review current design decisions (Section 2).
- We discuss certain issues that frequently arise in practice (Section 3).
- We describe an alternative type-based embedding, using type classes instead of higher-kind types (Section 4).
- We discuss using Scala macros for certain aspects of the embedding that have previously been implemented in a compiler fork (Section 5).
- We present an alternative definition of a graph-based IR which is untyped, and we discuss how to still achieve type-safe rewritings and transformations (Section 6).

It is important to note that these patterns have been known for some time, and that they reflect the work of a large number of people from different institutions (EPFL, ETH, Huawei, Stanford, and Purdue). The main contribution of this paper is to document and crystallize them. Section 7 surveys generic related work, and Section 7 provides further attribution for the work described in this paper.

2. The LMS Design Space

We discuss the five key elements of LMS in more depth, and describe the design decisions taken by the current implementation.

2.1 Type-Based Embedding

The essence of multi-stage programming is that we can operate with present-stage and future-stage values in the same program. But how shall we distinguish the two?

There is a whole spectrum of techniques which make the distinction more or less prominently visible in the program, which we can classify according to their noise level. This characterization is inspired by a remark of Bjarne Stroustrup [78]: “People confuse the familiar for the simple. For new features, people insist on LOUD explicit syntax. For established features, people want terse notation.”

Syntactic quotations on the one end make the stage distinction obvious, but introduce clutter and are thus somewhat cumbersome to use. Traditional partial evaluation [45] does not distinguish the stages in any observable way, but relies on automatic binding-time analysis to separate the program into static and dynamic components, or on automatic online specialization [89]. Either method of fully automatic separation is brittle in practice. Programmers have little control over the process, and in particular few effective means for debugging.

A similar noise level comparison can be drawn for typed vs untyped languages and explicit type annotations vs inferred types, with well-known pro and contra arguments about reading code vs writing code, refactoring, types as documentation, and so on.

A key design decision of LMS is to treat the stage distinction in exactly the same way as ordinary types, and use the meta-language type system to distinguish present-stage from future-stage expressions. This strikes a good balance in practice, reducing syntactic noise to a comfortable volume, consistent with the normal use of type annotations in the host language, but at the same time keeping the programmer in full control of the staging decisions.

In Scala, type annotations are required on method arguments, but generally not within method bodies. Thus, we can say that Scala’s local type inference performs a
semi-automatic binding-time analysis for us, local within a method body.

Following the idea of finally tagless and polymorphic embedding, LMS introduces an abstract type constructor `Rep[T]` to denote staged expressions that will generate code of type `T`.

Here is an example—the ubiquitous power function:
```scala
val driver = new LMS_Driver[Int,Int] {
  def power(b: Rep[Int], x: Int): Rep[Int] = 
    if (x == 0) 1 else b * power(b, x - 1)

  def main(x: Rep[Int]): Rep[Int] = {
    power(x,4)
  }
}
driver(3) 
```

The LMS_Driver class is provided by the framework. We create a new instance driver, parameterized to create a compiled function from `Int` to `Int`. Inside its scope, we can use `Rep` types and the corresponding operations. Method `main` is the entrypoint for the generated code. At staging time, the driver will execute `main` with a symbolic input. This will completely evaluate the recursive power invocations (since it is a present-stage function) and record the individual expressions in the IR as they are encountered. On exit of main, the driver will compile the generated source code and load it as executable into the running program. Here, the generated code corresponds to:
```scala
class Anon12 extends ((Int)=>(Int)) {
  def apply(x0:Int): Int = {
    val x1 = x0*x0
    val x2 = x0*x1
    val x3 = x0*x2
    x3
  }
}
```

The performed specializations are immediately clear from the types: in the definition of power, only the base `b` is dynamic (type `Rep[Int]`), everything else will be evaluated statically, at code generation time. The expression `driver(3)` will then execute the generate code, and return the result 81.

From the view of client code, `Rep` does not have any operations of its own. Hence, operations on, e.g., `Rep[Int]` have to be defined externally:
```scala
trait Base {
  type Rep[T]
}
trait IntOps extends Base {
  implicit def unit(x: Int): Rep[Int]
  def infix_*(x: Rep[Int], y: Rep[Int]): Rep[Int]
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
```

Note that from the client's view, all the operations are abstract. Hence, we can think of these definitions as axiomatizing the future-stage language.

For a multi-stage expression such as `b * x`, we have tricked Scala’s type inference into performing a simple local binding-time analysis (as mentioned earlier), which determines which parts of the expression are evaluated now and which parts later. Note that this analysis is guided by type annotations present in the code, and hence developers can more easily reason about the process as with traditional partial evaluators that do not make such a distinction.

**Virtualization of the host language** If we add booleans, how can we use the normal `if (c) a else b` syntax if `c` has type `Rep[Boolean]` instead of `Boolean`? The solution is to **virtualize** the host language, redefining such primitives as method calls:
```scala
def __ifThenElse[T](c: Rep[Boolean],
  a: =>Rep[T],
  b: =>Rep[T]): Rep[T]
```

While Scala has always had customizable for comprehensions in this way, features like `if/else` are not part of the standard. Hence, this syntax requires a fork of the Scala compiler, called Scala-Virtualized.

In the signature of `__ifThenElse` above, it is important to note the by-name arguments of type `=>Rep[T]`: these serve to inherit the evaluation order of the meta-language, as described next.

### 2.2 Multi-stage Semantics

LMS takes care to not duplicate or re-order staged computations. This is similar to semantics in partial evaluation, but unlike quasiquotations in languages like Lisp or MetaML, which are based on syntactic expansion. To see the difference, consider a more sophisticated algorithm to compute exponents in logarithmic time by repeated squaring:
```scala
def power(b: Rep[Int], x: Int): Rep[Int] = 
  if (x == 0) 1 
  else if ((x&1) == 0) ( val y = power(b, x/2); y * y )
  else b * power(b, x - 1)
```

The generated code in LMS will be:
```scala
class Anon13 extends ((Int)=>(Int)) {
  def apply(x0:Int): Int = {
    val x1 = x0*x0
    val x2 = x1+x1
    val x3 = x2*x2
    x3
  }
}
```

By contrast, purely syntactic expansion would yield:
```scala
(((x0+x1)*(x0+x1))*(x0+x1))
```

Here, staging has undone the effect of the `val y = ...` binding and turned our fast algorithm back into a linear one. This behavior is widely known, and has been studied in the context of MSP languages at length. While the result is “only” a slowdown here, syntactic expansion and the resulting re-ordering or duplication of code can be disastrous in the presence of side effects.
LMS achieves “proper” multi-stage semantics by eager let-insertion \[6\] [29] [43] [87], i.e. binding every new staged operation to a fresh identifier as soon as the staged operation is encountered in the meta-language.

2.3 Graph-Based IR

Initially, the LMS API is wired to create an intermediate representation (IR) which can be further transformed and finally unparsed to target code:

```scala
trait BaseExp {
  // IR base classes: Exp[T], Def[T]
  type Rep[T] = Exp[T]
  def reflect[T](x:Def[T]):Exp[T] =..< //add x to IR graph
}
```

```scala
trait IntOpsExp extends BaseExp {
  case class Plus(x:Exp[Int],y:Exp[Int]) extends Def[Int]
  case class Times(x:Exp[Int],y:Exp[Int]) extends Def[Int]
  implicit def unit(x: Int): Rep[Int] = Const(x)
  def infix_+(x:Rep[Int],y:Rep[Int]) = reflect(Plus(x,y))
  def infix_*(x:Rep[Int],y:Rep[Int]) = reflect(Times(x,y))
}
```

Another way to look at this is as combining a shallow and a deep embedding for an IR object language \[82\].

The fact that we are using (directed, but not necessarily acyclic) graphs instead of trees enables comparatively straightforward implementation of rather sophisticated optimizations \[53, 16\]. Common subexpression elimination is just hash-consing. Methods like infix_+ can serve as smart constructors that perform optimizations on the fly while building the IR \[63\]. Dead code elimination is just reachability. These optimization crucially rely on the ability of (effect-free) nodes to move arbitrarily across the graph. Before unparsing, a code motion algorithm needs to schedule the graph, and decide in which scope each expression should be computed.

The LMS_Driver class mixes in a number of base traits to define the core language:

```scala
abstract class LMS_Driver[A,B] extends BaseExp {
  with IntOpsExp
  val IR: VectorExp;
  override def transformDef[d](d: Def[T]) = x =>
  
  case class vec_plus_ll(a:Rep[Vec[T]],b:Rep[Vec[T]]) =
  vec_plus_ll(transformExp(a), transformExp(b))
}
```

In the same way, we can add new optimizations via smart constructors by overriding the default term constructor:

```scala
trait IntOpsExpOpt extends IntOpsExp {
  override def infix_+(x: Exp[Int], y: Exp[Int]) =
    x match {
      case Const(0) => y
      case (x, Const(0)) => y
      case _ => super.infix_+(x,y)
    }
}
```

Multiple such extensions can be layered, thanks to the super call in the fall-through case of the pattern match. The linearization order of the traits will determine the order in which the rewrite rules are tried.

All such rewrites are applied eagerly in an online-fashion while constructing the IR, together with all other smart constructors. As a result, we do not require auxiliary mechanism such as applying rewrite rules until a fixpoint is reached. With smart constructors, we directly obtain a strong guarantee that, in this case, no IR node of the form \(0 + y\) or \(x + 0\) will ever be created.

2.5 Vertical Extensibility

Horizontal extensibility and optimizations with smart constructors go a long way, but in some cases we need additional power. In particular, it is often necessary to first perform rewrites on a high-level of abstraction, say, matrices and vectors in linear algebra, which would be implemented in the same way as the plus operator on integers shown above. But we also want staging to transform matrix products into tight loops after high-level algebraic optimizations are applied exhaustively \[63\].

For such use cases, LMS provides a traversal and transformer API:

```scala
trait LowerVectors extends ForwardTransformer {
  val IR: VectorExp; import IR._
  def transformDef[d](d: Def[T]) =
    case VectorZeros(n) =>
      Vec.zeros(n) =>
    case VectorPlus(a,b) =>
      Vec.plus(a,b) =>
    case _ =>
      super.transformDef(d)
}
```

Here, Vec.zeros and Vec.plus are taken to be methods defining how zero vectors and vector addition are represented on a lower-level, using just arrays and loops:

```scala
def Vec.zeros[T](n: Rep[Vec[T]]) = Vector.fromArray(Array.fill(n) { i => zero[T] })
def Vec.plus[T](a, b) =
  Vector.fromArray(a.data.zipWith(b.data)(_ + _))
```

The definitions of these methods can use staging again on the lower level, and hence the overall pattern corresponds to implementing translation passes as staged interpreters over the higher-level IR.
If we want to use the transformers API with custom IR nodes, we need to implement a default transformation case called `mirroring` for each of them:

```scala
// mirror: transformation default case
def mirror[T](d: Def[T])(t: Transformer) = d match {
  case VectorZeros(n) =>
    Vector.zeros(t.transformExp(n))
  case VectorPlus(a,b) =>
    t.transformExp(a) + t.transformExp(b)
  case _ => super.mirror(d)
}
```

### 3. Practical Issues

Here are some issues encountered in practice with the current design.

**Non-problem: Rep types** Sometimes Rep types themselves are pointed out as an issue, or the fact that they show up in type errors even though they might have been inferred in the source (see e.g. [37]). We do not consider this an actual problem, but rather a symptom of unfamiliarity. In our experience with new LMS users, in particular students, these issues disappear once an initial understanding of the different stages of execution sets in. From that point on, Rep types seem to strike a good balance in terms of noise, and type errors are no more complicated (and often less) than with other Scala libraries that rely on advanced types.

**Conversions to/from Rep** The distinction between normal Scala types and staged Rep types sometimes leads to inconvenient conversions. Assume we declare a user-defined class of complex numbers:

```scala
case class Complex(re:Rep[Double], im:Rep[Double])
```

Since `Complex` is not a `Rep` type, we cannot easily return complex numbers from an `if` expression. To do so, we would either need to support complex numbers as first-class type `Rep[Complex]` in our IR, or we would need to convert them into a tuple `Rep[(Double,Double)]` and back. The issue gets worse if we have a whole class hierarchy, e.g. `Complex` as an abstract base class with subclasses `Polar` and `Cartesian`, which necessitates a uniform encoding of all possible alternatives, e.g. as tagged union [65].

**GADT Pattern matching** Smart constructors or transformers rely on pattern matching over typed IR nodes, which are implemented as (Scala’s approximation of) GADTs. Unfortunately, Scala’s GADT support in pattern matching is incomplete. As a result, it is often required to insert type casts. For example, the definition of `mirror` above will actually look like:

```scala
def mirror[T](d: Def[T])(t: Transformer) = (d match {
  case VectorZeros(n) => ...
}).asInstanceOf[Exp[T]]
```

This necessity for explicit casts somewhat negates the benefits of a typed IR and makes pattern matching fundamentally unsafe.

### Boilerplate for DSL definition

Defining custom IR nodes incurs a fair bit of boilerplate. We first need to provide the interface trait, with abstract methods for all operations. Then we need to provide the implementation trait, with `Def` classes and implemented methods. We may need additional traits for optimizations, again with overridden methods. And finally we need to implement mirroring for each `Def` class.

**Two-thirds-solution: Forge** While this boilerplate is repetitive, it follows a clear structure, and often large parts can be auto-generated. Forge [79] is a meta-DSL that takes a declarative specification as input, and generates all the necessary LMS definitions.

But there is still a problem: even though the boilerplate no longer needs to written, the Scala compiler still needs to deal with all the complexity.

**Problem: Scala compile time (1)** The embedding using `Rep[T]` means that we cannot use normal Scala methods for staged functionality. For example, an operation `Rep[Int] + Rep[Int]` cannot be implemented as a method on type `Rep` or `Int`. We could use implicits, but they are imprecise, in particular for operations involving numeric types. A key problem is to model the right widening behavior for `Rep[Float] + Rep[Double], Int + Rep[Int]` and so on.

Scala-Virtualized implements `infix_` methods, which enable more precision. But leveraging this precision requires spelling out many alternative signatures: more than 30 for `+` in Delite, with `{Int,Long,Float,Double} x {T,Rep} x {Scalar,Vector}`. This large set of alternatives leads to a drastic slowdown, since overloading resolution has \(O(n^2)\) complexity in `scalac`. In Delite, type checking a single `+` could take up to 0.5s. A notable worst case was an unrolled determinant calculation with 33 plus operations that took a good minute to type check.

**Problem: Scala compile time (2)** Another issue with compile times is due to mixin composition of a large number of traits, each contain a large number of members. To compile traits into JVM classes, the Scala compiler needs to generate bridge methods and other auxiliary definition. And when mixing traits together, the compiler needs to check for overriding pairs, which is again an \(O(n^2)\) operation in the worst case.

### 4. Type-Based Embedding

Going back to our starting point, we observe that we do not have to use `Rep[T]` to realize a type-based embedding. It is enough to have some distinction between, say, the normal Scala `Int` and the staged `Int`. In particular, we can just define a new `Int` type in a given scope of the program:

```scala
package dsl

trait Int {
  def +(y: Int): Int
  def *(y: Int): Int
}
```
and rely on scoping and path-dependent types to distinguish between scala.Int and dsl.Int.

But how can we tell on a more abstract level whether a given type is staged or not? The key idea here is to use type classes: we require an instance of a type class Typ[T] whenever T is a staged type.

Putting these ideas together, we come up with the following alternative definition of the core LMS traits seen before:

```scala
trait Base {
  // preliminaries
  @implicitNotFound("$T is not a DSL type")
  type Typ[T]
}

trait IntOps extends Base {
  trait IntBase {
    def +(y: Int): Int
    def *(y: Int): Int
  }
  type Int <: IntBase
  implicit def intTyp: Typ[Int]
}
```

Again the interface is purely abstract, achieving separation of concerns, and enabling us to pick a suitable IR representation of our choice.

Generic code now requires a Typ[T] instance instead of using Rep types. For example:

``` scala
def __ifThenElse(c:Boolean, a: =>T, b: =>T): T
```

Note that we take Boolean to be a staged boolean here, different from the normal scala.Boolean, and implemented in the same way as the staged Int type above.

As a pleasant side effect, the use of implicitNotFound leads to better error messages than higher-kindred types.

For user-facing code, we now have a couple of choices, depending on the desired noise level. Instead of going with overloaded Int or Boolean, we could also pick unique name for types (e.g., DInt), or use qualified prefixes like dsl.Int. Either way, using imports and scoping we can define whether Int resolves to scala.Int or dsl.Int by default.

Depending on this choice, the power example can be written in different ways:

```scala
// default is staged, i.e. dsl.Int
def power(b: Int, x: scala.Int): Int =
  if (x == 0) 1 else b * power(b, x - 1)

// default is unstaged, i.e. scala.Int
def power(b: dsl.Int, x: Int): dsl.Int =
  if (x == 0) 1 else b * power(b, x - 1)
```

5. Virtualization using Macros

Instead of having to add all Int methods to Rep[Int] externally, we now use normal objects and classes. Everything is still implemented purely as a library, and we do not require any other mechanisms such as macros per se.

But a number of things become simpler: since we use normal objects and classes for staged types, we do not need infix methods anymore, and thus the first compiler performance problem goes away.

Moreover, the Scala-Virtualized compiler had special support for staged structural types via Struct[T]. This is also no longer needed. We just need a macro that creates a Typ[T] instance for structural types T <: Struct where all fields have an implicit Typ instance available.

The remaining functionality, like virtualizing built-ins such as if (c) a else b as __ifThenElse(c,a,b), can easily be implemented using annotation macros, obviating the need for a dedicated compiler fork.

6. Graph-Based IR

As a final piece in the puzzle, we can use the new embedding with an untyped IR, based essentially on S-expressions.

```scala
trait BaseExp extends Base {
  trait Typ[T] {
    def decode(e: Exp): T
    def encode(x: T): Exp
  }
}

trait IntOpsExp extends BaseExp with DSL {
  case class Int(e: Exp) extends Base {
    def decode(e: Exp): T = e
    def encode(x: T): Exp = new Int(x)
    override def toString = "Int"
  }
}
```

The encode and decode methods in Typ[T] are required to convert between the user-facing and the IR-internal representation, suitable for unparsing.

This removes the need for 100s of classes for different IR nodes, leads to fewer generated bytecode, and faster compile times, eliminating the second performance problem.

How can we add new types in this model? It’s simple, we just create a new class and provide a Typ[T] instance. For generic types, such as functions or collections, we require that the type parameters come with Typ[T] instances as well. We also have the option of introducing a notion of second-class types DemiTyp[T] to denote staged but non-first-class values, which is useful, for example, to model functions when generating C [2]. How can we add new operations to an existing type? We use normal Scala implicit classes.

**Optimizations and lowerings** How can we add new optimizations? We provide a new facility:

```scala
rewrite { v1: Vector =>
  (v1 + 0) ===> v1
}
```

Method rewrite can treat the passed closure as higher-order abstract syntax, and invoke it with a fresh symbol as v1. Both sides of the ===> arrow are now S-expressions, and the arrow
maps to an S-expression, too. Thus, internally, rewrite will obtain a reified representation of the rewrite rule:

\[(\text{vector-+} \ ?X1 \ 0) \ ?X1\]

Such rules are added to a rulebase that is automatically queried whenever a new IR node is created. Thus, we achieve the same behavior as with smart constructors.

How can we add new transformations? Using the same new mechanism:

\[
\text{lower } \{ \text{v1: Vector } \Rightarrow \\
\text{v1 } \Rightarrow \text{v1.data} \\
\} \\
\]

The only difference is that the rule will be stored in a different rulebase.

With further uses of annotation macros we can even automate this process, and derive IR definitions as well as lowering transformations directly from a class definition. For this purpose, we introduce a macro @ir, which, used as follows:

\[
\text{@ir class Vector(data: Array[Int]) } \\
\text{def } +\text{(o: Vector) } = \\
\text{ new Vector((data zip o.data) map (_ + _))} \\
\]

Expands the class definition to the following:

\[
\text{class Vector(data: Array[Int]) } \\
\text{def } +\text{(o: Vector) } = \\
\text{ reflect[Vector]("vector-+", this, o) } \\
\text{def } +\_\text{body(o: Vector) } = \\
\text{ new Vector((data zip o.data) map (_ + _))} \\
\]

\[
\text{lower } \{ \text{v1: Vector, v2: Vector } \Rightarrow \\
\text{(v1 + v2) } \Rightarrow \text{(v1.+\_body(v2))} \\
\} \\
\]

One challenge in this model is dealing with generic types, e.g., a class Vector[T]. Our current API is based on a set of global placeholders Generic.T1, .T2, etc, so above lowering for generic vectors becomes:

\[
\text{lower } \{ \text{v1: Vector[Generic.T1], } \\
v2: Vector[Generic.T1] } \Rightarrow \\
\text{(v1 + v2) } \Rightarrow \text{(v1.+\_body(v2))} \\
\]

In summary, even though the IR is now untyped, we can still specify rewrites and transformations in a typed way, and get the benefits of horizontal and vertical extensibility.

**Stage polymorphism** As a final note, with this new typed embedding, we can keep the split between interface and implementation traits (e.g. IntOps and IntOpsExp), and therefore also retains a key benefit of the finally tagless approach, namely the ability to abstract over staging decisions. Providing a directly executable implementation of a DSL interface alongside the staged implementation is not only useful for prototyping or debugging, but stage polymorphism, is also a key programming pattern for abstracting over precomputation decisions and controlling data layout and code shape in generators for high-performance numeric libraries [52, 77].

7. Related Work

Multi-stage programming (MSP, staging for short), as established by Taha and Sheard [85] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [10] implements a classic staging system based on quasi-quotiation. Lightweight Modular Staging (LMS) [62] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code [58]. LMS draws inspiration from earlier work such as TaskGraph [4], a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [63, 65, 64, 81].

Partial evaluation [35] is an automatic program specialization technique. Some notable systems include DyC [27] for C, JSpec/Tempo [68], the JSC Java Supercompiler [39], and Civet [69] for Java. Lancet [64] is a partial evaluator for Java bytecode built on top of LMS. Preserving proper semantics in the presence of state has been an important goal [6, 29, 43, 57]. Further work has studied partially static structures [45] and partially static operations [86], and compilation based on combinations of partial evaluation, staging and abstract interpretation [75, 17, 58]. Two-level languages are frequently used as a basis for describing binding-time annotated programs [35, 50].

Embedded languages have a long history [42]. Hudak introduced the concept of embedding DSLs as pure libraries [32, 33]. Steele proposed the idea of “growing” a language [76]. The concept of linguistic reuse goes back to Krishnamurthi [41]; Language virtualization to Chaf i et al. [13]. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [11] and Hofer et al. [31], going back to much earlier work by Reynolds [57]. Recent work in this line builds modular interpreters using object algebras [34], similar to LMS. There are many other language extension mechanisms: preprocessors or generators, and built-in meta-language facilities, such as Template Haskell [70], Metaborg [7]. SugarJ [22], macros in Racket [88] or Scala [9]. We refer to [21] for an overview. Compiling embedded DSLs through dynamically generated ASTs was pioneered by Leijen and Meijer [45] and Elliot et al. [20]. All these works greatly inspired the development of LMS. Haskell is a popular host language for embedded DSLs [83, 26]. examples being Accelerate [47], Feldspar [3], Nikola [46]. Recent work presents new approaches around quotation and normalization for DSLs [49, 14]. Other performance oriented DSLs include Firepile [51] (Scala), Terra [18, 19] (Lua). Copperhead [12] (Python). A line of work by Scherr and Chiba [65, 57, 15] is also related.

Program generators for high-performance code include, for example, ATLAS [90] (linear algebra), FFTW [23] (discrete fourier transform), and Spiral [54] (general linear transformations). Other systems include PetaBricks [1], CVX-gen [28] and Halide [56, 55].
Acknowledgements

This paper represents work by a large group of people from different institutions. At EPFL, Nada Amin and Vojin Jovanovic worked on virtualization via macros to support LMS front-ends. From Kunle Olukotun’s group at Stanford, Kevin Brown, HyoukJoong Lee, and Arvind Sujeeth made key contributions to LMS and Delite front-ends, supported by EPFL exchange students Cédric Bastin and Boris Perović. The use of type-classes as alternative to higher-kindred types has its roots in Haskell-based work on Feldspar and also in Lightweight Polytypic Staging and Isomorphic Specialization. Georg Ofenbeck, Alen Stojanov, and Markus Püschel at ETH have explored this line further in the context of core LMS, and made additional contributions. Nada Amin explored such an embedding with type classes in the context of LMS-Verify. The work on alternative rewriting interfaces is inspired by Kiama, and was further explored by Alexander Slesarenko, Alexey Romanov and others at Huawei in the context of Scalac. The illustration in Figure was originally conceived by Markus Püschel. This research was supported by NSF through awards 1553471 and 1564207.

References


tures as complementary approaches to data abstraction. 1975.


