A Stage-Polymorphic IR for Compiling MATLAB-Style Dynamic Tensor Expressions

Alen Stojanov
Department of Computer Science
ETH Zurich, Switzerland
astojanov@inf.ethz.ch

Tiark Rompf
Department of Computer Science
Purdue University, USA
tiark@purdue.edu

Markus Püschel
Department of Computer Science
ETH Zurich, Switzerland
pueschel@inf.ethz.ch

1 Introduction

MATLAB is a flexible, dynamic, high-level language that is widely used in science and engineering for prototyping. Once a prototype is built, it is typically reimplemented using a low-level language for efficiency and deployment. To automate this process, there has been extensive research on automatically optimizing, interpreting, translating and compiling MATLAB [1, 2, 5, 6, 10, 14, 15, 18, 20, 22, 23, 25, 31, 35, 37–39, 44, 46, 47]. The rich set of 3520 functions [27] (as of version 2019a), of which 473 are considered as language-fundamental and 536 as mathematical, is the reason why all prior work focused on suitable subsets of MATLAB.

A convenient strategy for building a compiler is to stage an interpreter, a pragmatic realization of the first Futamura projection [17]. Frameworks such as LMS [43] have been successfully used in this way for compiling not only embedded DSLs [50] but also external DSLs like SQL [41, 52]. Applying the same idea to a dynamic language like MATLAB is possible, but the compiled code will not be efficient. In contrast to statically typed languages like SQL, which specialize well into low-level code, a naive staged interpreter for a dynamic language like MATLAB has to residualize essentially all of the dynamic type and shape dispatch, since type and shape information is not statically apparent from the program text. The solution, then, is to infer the necessary information and attach it to the source program where available. Thus, the staged evaluator needs to deal with varying degrees of static information present or absent, a setting naturally captured by the concept of stage polymorphism [33, 34].

In this paper, we propose an implementation technique that is effective and elegantly blends in with the core Futamura projection idea. We tweak the MATLAB evaluator so that it performs type inference first and then stages the remaining computations, passing reified types as values. As variables can have multiple types in a dynamic language, type values can be either static or dynamic. This suggests an encapsulation into a single internally stage-polymorphic object of a staged expression with its type value and other possible metadata, such as shape and order, each of which can be either static or dynamic. We refer to such object instances as metacontainers.

To compile MATLAB code, we start with an initial mapping of MATLAB source to a representation as metacontainers without any metadata. Then, we gradually insert type
and dimension information. Using metacontainers, this information is overlaid on the IR representing the computation.

Once type and dimension information is inferred, the information is then applied back to the metacontainer, creating a specialized metacontainer. Instead of performing complicated rewrites, the specialization is done through substitution of the metacontainer components, which results in a process of combining analyses and transformations [24]. Having a specialized metacontainer for a specific type or specific dimensions, we obtain a specialized MATLAB interpreter for a particular program. As a consequence of the first Futamura projection, our interpreter becomes a compiler that produces a standalone executable.

In summary, this paper makes the following contributions:

1. We propose the metacontainer abstraction and demonstrate its use for building compilers. We show that metacontainers can reduce the complexity and specialize IRs, simplifying analysis and transformations.

2. We present a prototypical MATLAB to C compiler based on this idea. Unlike prior work, our prototype provides systematic type inference involving all numeric primitives in the supported subset, handles many dynamic aspects of MATLAB, and generates correct code with explicit vectorization.

This paper does not focus on generating high-performance code for MATLAB, as this would require a variety of orthogonal optimizations, which are outside the scope of this project.

2 Background

In this section we provide brief background on MATLAB, LMS, and stage polymorphism.

2.1 MATLAB

**Types and variables** Variables in MATLAB are mutable and untyped, and can be either global or local. We classify them into three groups: multidimensional matrices, cells and functions.

Multidimensional matrices, which we refer to as tensors, consist of single or double precision floats, 8-, 16-, 32- or 64-bit signed or unsigned integers, representing real or complex values. Tensors can have unbounded number of dimensions, which we refer to as tensor order, and each dimension can have unbounded value. We use the term shape to refer to the dimension of a given tensor. As long as the imaginary component is not present, tensors can represent character and boolean values. In total there are 12 primitive types, with double precision floating point as default type. Cells represent structures and objects. Each cell can consist of tensors, cells, or functions, which correspond to the fields and methods in an object oriented model. Functions can be nested or anonymous. Each function takes multiple tensors, cells or functions as input and produces one or multiple outputs. Functions can be passed as function handles, and both input and output of functions can have variable length.

Apart from the three base groups above, recent versions of MATLAB also provide additional datatypes, including: sparse matrices, strings, tables, categorical arrays, persistent variables, datetime arrays and duration arrays.

**Indexing** MATLAB employs a versatile mechanism for indexing tensors. The indexing is done using one or several subscripts. These can be scalar literals, scalar variables, vector expressions, vector variables and colon notation (:). The behaviour depends on the shape of the variable at a given point in the program. The indexing can be used to obtain values or set values in a tensor. We refer to the former as indexed read and to the latter as indexed assignment.

**Built-in operators** in MATLAB provide arithmetic, ordering, and boolean logic that can be applied to tensors of any order. There are exceptions to the rule, such as tensor contractions and transposition operators, that are designed to work with two-dimensional matrices (i.e. second-order tensors) only. Unlike arithmetic operators in C/C++, MATLAB uses saturated arithmetic, avoiding overflows and underflows. We refer to the different type combination in operators as type interaction.

**Loops** The language supports for and while loops. The for-loops are specified with a start and an end value, as well as an iterator, each represented by a scalar of the 12 primitive types. If a non-scalar value is used, MATLAB will extract the non-imaginary part of the first element. Alternatively, the for loops can iterate over all values of a tensor, similarly as a foreach construct. MATLAB loops support early exit and continuation of loops using break and continue.

**Conditionals** Conditionals are implemented through if-then-else and switch constructs. The if and while conditionals are formed with a boolean expression that results in a tensor, which is consider true iff all elements in the non-imaginary component of the tensor are non-null values. switch is comprised of switch-expressions and constant case-expressions which can be of any type.

**Exceptions** Exception handling is supported in MATLAB through throw-try-catch constructs. The exception terminates the running function and returns control either to the keyboard or to an enclosing catch-block.

2.2 LMS

Multi-stage programming [51] was introduced to simplify program generator development by expressing the program generator and parts of the generated code in a single program, using the same syntax. LMS uses only types to distinguish the computational stages. Expressions of type Rep[T] in the first stage yield a computation of type T in the second stage. For example, the operation
will simply execute the arithmetic operation, while

```
1 val (a, b): (Int, Int) = (2, 3)
2 val c: Int = a + b
```

uses the higher-kinded type `Rep[_]` as a marker type to redirect the compiler to use an alternative plus implementation. LMS does not directly generate code in source form, as earlier approaches, but provides an extensible intermediate representation (IR) instead. The overall structure is that of a “sea of nodes” dependency graph [12].

Building compilers with LMS  LMS and its extensible IR can be used as a framework to implement compilers, by specializing a high-level interpreter with respect to a given program (the first Futamura projection [17]). Prior work [41, 42] demonstrates compilation of SQL queries to an efficient C-code, by a sequence of small transformations used to specialize a query evaluator with respect to a query plan.

Transformations in LMS  Transformations in LMS work as IR interpreters, using iterated staging, that are built as chain of LMS-based compilers one after another. Each transformer schedules the sea-of-node IR representation, following node dependencies, and then traverses the IR to apply the transformations. In this process, instead of modifying existing nodes, for each transformation LMS creates new nodes, with new dependencies to the existing nodes.

Parametric stage polymorphism  maintains in a single code base a staged version and one for regular execution. To illustrate, consider the two versions of a plus operator:

```
1 type Idt[T] = T
2 def add(a: Idt[Int], b: Idt[Int]) = a + b // regular
3 def add(a: Rep[Int], b: Rep[Int]) = a + b // staged
```

The two functions can be merged into a single polymorphic function that takes a compile-time higher-kinded type parameter `R[_]` that can be either `Idt` or `Rep`:

```
1 def add[R[_]] (a: R[Int], b: R[Int]) = n.plus(a, b)
2 (implicit n: NumericOps[R[Int]]) = n.plus(a, b)
```

and an implicit type class `NumericOps` that provides the staged or the regular implementation of the plus operator:

```
1 class NumericOps[R[T]] {
2   def plus (a: R[T], b: R[T]) : R[T] = { ... }
3   def minus (a: R[T], b: R[T]) : R[T] = { ... }
4   def times (a: R[T], b: R[T]) : R[T] = { ... }
5 }
```

When applied to Scala for comprehensions, we can encode loop unrolling optimization in a single type parameter:

```
1 for[R[_]] (range: R[Range]) { body }
```

Similarly parametric stage polymorphism can be applied to arrays, effectively encoding `scalar replacement` as a type parameter, controlled by two parameters R1 and R2:

```
1 val a: R1[Array[R2[T]]
```

This allows us to have either a staged array, or an array of staged elements. The abstraction can also be used to encode SIMD vectorization also as a type parameter [48]. Finally, we can combine all these axes of polymorphism into a single array abstraction with multiple type parameters. Each type parameter will control whether the array is scalar or not, whether we generate loops or unrolled code, having scalar or vector instructions.

3 Metacontainers  In this section we define the metacontainer abstraction and describe its use in staging functions. We consider staging a dynamically typed language with numerical computations in LMS, where an expression is represented with a staged symbol, as well as its type value:

```
1 val value: Rep[Any] // Staged expression, type unknown
2 val typ : Rep[Typ] // Corresponding type value
```

In this situation, a staged function must reason about values, as well as their corresponding types. Consequently, a simple implementation of a `plus` would result in:

```
1 def plus (a: Rep[Any], a_typ: Rep[Typ],
2   b: Rep[Any], b_typ: Rep[Typ]) = { ... }
```

To simplify usage, staged expressions can be “boxed” in wrapper classes, along with their type values:

```
1 class Number (  // class wrapper - metacontainer
2   val value: Rep[Any] // metacontainer reference
3   val typ : Rep[Typ] // reference metadata
4 )
```

We call this wrapper class a metacontainer. As the metacontainer is defined for the staged variable value, we refer to it as a `metacontainer reference`. The staged variable `typ` represents a property of the metacontainer reference or, in other words, it carries its `metadata`.

With this approach, operator signatures can be simplified:

```
1 def plus (a: Number, b: Number) = { ... }
```

or can become routines in the metacontainer itself:

```
1 class Number(val typ: Rep[Typ], val value: Rep[Any]) {
2   def plus (rhs: Number) : Number =
3     (typ, rhs_typ) match {
4       case (DoubleTyp, DoubleTyp) =>
5         new Number(DoubleTyp,
6           numeric_plus[Double](value, rhs.value))
7       case (DoubleTyp, FloatTyp) =>
8         plus(rhs.cast[DoubleTyp])
9       case (DoubleTyp, IntTyp) => /* all others */
10   }
```

36
The metacontainers are abstractions that will be removed during code generation, emitting code in which the metacontainer reference and its metadata are “unboxed.” In the example above, the generated code will perform type inspection for the two primitive operands to invoke the type-specific computation, such that it corresponds to the pattern matching expression. However, during the staging phase, metacontainers are regular Scala object instances. Therefore, we can define them more formally:

**Definition 3.1.** A metacontainer is a runtime object wrapper for a given metacontainer reference that includes its metadata and a set of routines associated with it. In every metacontainer, the metacontainer reference is a set of at least one staged variable and the metadata is a set of runtime objects, staged variables or other metacontainers.

Metacontainers can leverage all properties from the object-oriented paradigm. They can be nested, can be polymorphic or their routines can be overridden. To illustrate this, consider a variable that does not change its type in a given program. If we can infer this information, we can specialize its representation in the `Number` metacontainer:

```scala
class FloatNumber(value: Rep[Float]) {
  override def plus(rhs: Number) = rhs.typed match {
    case DoubleTyp => cast[DoubleTyp].plus(rhs)
    case FloatTyp => new FloatNumber(
      numeric_plusFloat[a.value, b.value])
    case IntTyp => (...)
    // Implement other type combinations
  }
}
```

In this particular case, the type inspection will be specialized to pattern match only one operand instead of two, which provides a more efficient program.

The polymorphic structures can be built on top of each other, using nested metacontainers. To illustrate, consider a stage polymorphic implementation of a complex number that is built on top of the `Number` metacontainer:

```scala
abstract class Element {
  def plus(rhs:Element) = rhs.match {
    case Real (r) => Real (r+r)
    case Complex(r,i) => Complex(r+r, i+i)
  }
}
```

Apart from stage polymorphic specialization, an important property of metacontainers is that they can be used to define relationships between staged expression stored in the metacontainer. The high-level dependencies are only visible in the metacontainer, but are not explicitly present in the underlying representation of the staged program in LMS. In

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - ./ .\ .^</td>
<td>Pointwise arithmetics</td>
</tr>
<tr>
<td>&lt;= &gt;= == &lt;&gt; ~= &amp;</td>
<td>Relational operators</td>
</tr>
<tr>
<td>mtimes</td>
<td>Logical operators</td>
</tr>
<tr>
<td>A'</td>
<td>Matrix-matrix multiplication</td>
</tr>
<tr>
<td>a/sin a/cos a/tan</td>
<td>Trigonometry functions</td>
</tr>
<tr>
<td>abs sqrt</td>
<td>Absolute value or square root of a tensor</td>
</tr>
<tr>
<td>exp log log10</td>
<td>Power and logarithmic functions</td>
</tr>
<tr>
<td>ceil floor round fix</td>
<td>Rounding</td>
</tr>
<tr>
<td>mod rem</td>
<td>Modulo and remainder</td>
</tr>
<tr>
<td>conj</td>
<td>Complex conjugate</td>
</tr>
<tr>
<td>length size numel</td>
<td>Tensor dimension inspection.</td>
</tr>
<tr>
<td>min max sum mean</td>
<td>Min, max, sum, mean of a tensor</td>
</tr>
<tr>
<td>zeros ones rand : colon</td>
<td>Tensor initializations with 0, 1 or random</td>
</tr>
<tr>
<td>horzcat vertcat</td>
<td>Unit-spaced vector indexing</td>
</tr>
<tr>
<td>transpose</td>
<td>Horizontal and vertical tensor concatenation</td>
</tr>
</tbody>
</table>

the examples above, this is the relationship between a variable and its type. The implications and use of this property are discussed in the subsequent sections.

## 4 MGen

In this section we describe MGen. First we define the supported subset of MATLAB and provide a high-level overview of the design and generator phases. Then, we describe in detail the methodology used to infer types and shapes, and how we lower the computations to low-level C representation.

### 4.1 Supported Subset

MGen supports user-defined global functions, provided as `.m` files, but no nested functions, anonymous functions or function handles. Excluded are also variable length arguments, global variables, structures, cell arrays, MATLAB classes, and recently included MATLAB types such as strings, sparse matrices, tables, timetables, categorical arrays, date and duration arrays, and persistent variables. MGen supports all 12 primitive types including numeric classes, boolean types and characters, and multi-dimensional arrays with variable-size complex or real data.

MGen supports double-precision, single-precision, and integer saturated arithmetic. The supported built-in functions are listed in Table 1. `if`, `switch`, `for`, `while` are fully supported, while `continue` and `break` are not. Indexing is supported for tensors of any order and shape, including linear and logical indexing and indexing with multiple subscripts.

### 4.2 High-level Overview

Fig. 1 shows a high-level overview of MGen. The input is a MATLAB or a Scala program, and the output is a standalone C program. The interface allows several MATLAB functions as input, one of them being the entry point. For this paper, we omit the description of the Scala front-end.
The result of the generator is a self-contained C-code bundle that can include one or several functions. Next we provide details on each step.

4.3 Building the M-IR

**M-IR overview** The M-IR provides LMS definitions for all supported MATLAB operations in Table 1. Each definition extends the `ShapeDef` interface:

```scala
abstract class ShapeDef extends Def[Shape] {
  val typedFunction : TypedFunction // tensor type
  val complexFunction: ComplexFunction // complex type
  val dimLenFunction : LengthFunction // tensor order
}
```

Since MATLAB is typeless, `Exp[Shape]` is used as the default type. For each function, we provide three functions that propagate the resulting type, complex type, and tensor order for each operator:

```scala
abstract class F[Int]( def apply(t: Seq[Int]): Option[Int] )
abstract class TypedFunction extends F[Type]
abstract class ComplexFunction extends F[Boolean]
abstract class LengthFunction extends F[Exp[Int]]
```

TypedFunction holds type rules for each built-in function. For example, for addition, they are:

- `double + double → double`
- `float + double → float`
- `double + int16 → int16`
- `... × ... → ...

When a given type combination is not valid, the function returns None. ComplexFunction is similar, describing rules that determine real or complex data type. LengthFunction describes the change of the tensor order in each supported functions. To illustrate, consider a point-wise operations, e.g., addition or multiplication, applied either to one scalar and a tensor, or to two tensors. The resulting tensor then has the maximum order of the two tensors:

```scala
object PointwiseLength extends LengthFunction {
  def apply(operands: Seq[Exp[Int]]): Option[Exp[Int]] = Some(math.max(operands(0), operands(1)))
}
```

M-IR also includes definitions for overloaded built-in operators. For example, for addition:

```scala
case class ShapePlus (... extends ShapeDef (...)
```

Once these phases are complete, we obtain an M-IR that provides constant or symbolic values describing the type and the shape of each variable. With these informations, we can lower the M-IR representation into a C-like intermediate representation in several steps:

1. For each program variable, we create memory references that will be allocated in subsequent phases.
2. For each computation, we take the inferred types from the M-IR, and generate code that performs type specialization on all the type combinations of the variables involved in the computation. We use the inferred type and dimensions to complete memory allocations.
3. Once types are specialized for each tensor, we perform operator lowering to a low-level C-IR.
4. If a vector ISA is specified, each operator is vectorized and ISA-specialized code is generated.

Besides the input functions, MGen also takes type and shape information for the function input and switches to toggle runtime checks in the generated code.

First, MGen builds the initial Math IR (M-IR) representation of the program in LMS. For parsing we use McLab Core [8] that outputs the Tame-IR [15] representation, which is translated to LMS. Next, LMS converts the program into SSA form, and performs dead-code elimination and common subexpression elimination. Once complete, we initiate a set of analysis and transformation phases to infer program properties:

1. The first phase is type and complex type inference.
2. For each operator we propagate and infer the resulting tensor order.
3. Once the tensor order is known to be a constant or variable, we perform analysis and transformations to infer each dimension value.

Once these phases are complete, we obtain an M-IR that provides constant or symbolic values describing the type and the shape of each variable. With these informations, we can lower the M-IR representation into a C-like intermediate representation in several steps:

1. For each program variable, we create memory references that will be allocated in subsequent phases.
2. For each computation, we take the inferred types from the M-IR, and generate code that performs type specialization on all the type combinations of the variables involved in the computation. We use the inferred type and dimensions to complete memory allocations.
3. Once types are specialized for each tensor, we perform operator lowering to a low-level C-IR.
4. If a vector ISA is specified, each operator is vectorized and ISA-specialized code is generated.

The result of the generator is a self-contained C-code bundle that can include one or several functions. Next we provide details on each step.

4.3 Building the M-IR

**M-IR overview** The M-IR provides LMS definitions for all supported MATLAB operations in Table 1. Each definition extends the ShapeDef interface:

```scala
abstract class ShapeDef extends Def[Shape] {
  val typedFunction : TypedFunction // tensor type
  val complexFunction: ComplexFunction // complex type
  val dimLenFunction : LengthFunction // tensor order
}
```

Since MATLAB is typeless, `Exp[Shape]` is used as the default type. For each function, we provide three functions that propagate the resulting type, complex type, and tensor order for each operator:

```scala
abstract class F[Int]( def apply(t: Seq[Int]): Option[Int] )
abstract class TypedFunction extends F[Type]
abstract class ComplexFunction extends F[Boolean]
abstract class LengthFunction extends F[Exp[Int]]
```

TypedFunction holds type rules for each built-in function. For example, for addition, they are:

- `double + double → double`
- `float + double → float`
- `double + int16 → int16`
- `... × ... → ...

When a given type combination is not valid, the function returns None. ComplexFunction is similar, describing rules that determine real or complex data type. LengthFunction describes the change of the tensor order in each supported functions. To illustrate, consider a point-wise operations, e.g., addition or multiplication, applied either to one scalar and a tensor, or to two tensors. The resulting tensor then has the maximum order of the two tensors:

```scala
object PointwiseLength extends LengthFunction {
  def apply(operands: Seq[Exp[Int]]): Option[Exp[Int]] = Some(math.max(operands(0), operands(1)))
}
```

M-IR also includes definitions for overloaded built-in operators. For example, for addition:

```scala
case class ShapePlus (... extends ShapeDef (...)
```
ShapePlus represents a standard addition, while ShapePlus1 represents addition of tensor and scalar and ShapePlusN represents addition of two tensors with identical dimensions.

Each tensor is represented with the metacontainer Shape:

```scala
class Shape {
  val exp: Exp[Shape] // metacontainer reference
  var pType = Set[Type]() // possible types
  var pComplex = Set[Boolean]() // possible complex types
  var pLen = Option.empty[Exp[Int]] // tensor order
  var dims = Option.empty[DArr] // tensor dimensions
  // define metacontainer dependencies
  def deps() =
    pLen.toList ::: (dims.toList.flatMap(_.deps()))
  // apply substitutions
  def transform(f: Transformer) = {
    if (pLen.nonEmpty) pLen = Some(f(pLen.get))
    if (dims.nonEmpty) dims = dims.transform(f)
  }
}
```

The MATLAB frontend builds the M-IR for each MATLAB function and generates a metacontainer for each variable. With the Shape metacontainer we avoid explicit representation of tensors in LMS, effectively simplifying the M-IR. This allows us to partially define its components, starting with an empty metacontainer that will get gradually populated in the subsequent phases.

### 4.4 Type Inference

#### Profiling type rules

The MATLAB language is continuously extended with new features and new functions [29]. To avoid manual inspection of type interaction for operators, we developed a profiler in MATLAB, that probes our supported built-in functions with different types and dimensions.

The profiler "brute-forces" for a each function all possible combination of types, including real and complex types. Not supported combinations raises an exception. With others, we generate a Scala object that extends TypedFunction and includes all valid type combinations. For example, for addition currently 56 out of 144 type combinations are valid. For example, double + int32 is valid and yields int32. With the profiler the basic type rules can easily be regenerated for new versions of MATLAB.

#### Type inference

MATLAB variables obtain their types when defined and can only change upon assignment. All elements of a tensor have identical types and an indexed assignment does not change the type of the variable being assigned. MATLAB does not support variable declarations and each variable is the result of a MATLAB expression that is either a built-in or a user-defined function. Consequently, we can use data-flow analysis to propagate and infer types.

The type profiles described above allow us to specify types rules used as type definition for each variable. This allows us to infer types using analysis based on reaching definitions [32]. The inferred types are then stored in the pType parameter of the Shape metacontainer.

#### Inferring real or complex data

To infer real or complex type of a tensor, we use the same approach for type inference, using the complex type definitions in complexFunction. The results of the analysis are then stored in the pComplex component of the metacontainer.

### 4.5 Shape Inference

MATLAB variables obtain their shape as the result of a MATLAB function, and can only change upon assignment or indexed assignment. In each case, tensor dimensions or tensor order can change or both. Therefore, we perform shape inference in two steps. First, we infer the tensor order of each variable. Then, we infer the tensor dimensions.

#### Inferring tensor order

To infer the order of tensors, we use the pLen component of the metacontainer. Initially, only the input arguments of the program have values set for pLen, while others are set to None. The first step is lowering the tensor order, by inserting code into the M-IR that reasons about a possible change of order. LMS uses a sea-of-nodes representations; thus, inserting code into the M-IR means creating new nodes. We perform the transformations by traversing forward through the M-IR, using the LengthFunction to propagate the order of tensors.

To illustrate, consider a program computing the n-th Fibonacci number:

```scala
function [ result ] = fib (n)
prev = 1; next = 1;
for i = 1:n
  tmp2 = next;
  next = next + prev;
  prev = tmp2;
end
result = next;
end
```

The lowering is illustrated in Figure 2. For every node in the initial M-IR (left), we create a metacontainer that contains the parameters for the tensor orders (right). Effectively, this process creates new definitions that are overlaid on top of the initial M-IR.

Unfortunately, adding new nodes into an LMS sea-of-node IR does not automatically make them part of the IR itself, as long as there are no dependencies between the old nodes and the new nodes. LMS allows us to define an anti-dependency relationship, that establishes a “must not after” scheduling of nodes. To achieve that, we use the deps method of the Shape metacontainer, and ensure that the metacontainer interfaces with LMS:

```scala
// override LMS methods for anti-dependencies
def softSyms(e: Any): List[Sym[Any]] = e match {
  case d: ShapeDef => findDefinition(d) match {
    case Some(TP(_)) =>
      Shape().deps() :::: super.softSyms(e)
    case _ => super.softSyms(e)
  }
}
```
A Stage-Polymorphic IR for Compiling MATLAB-Style Dynamic Tensor Expressions

GPCE ’19, October 21–22, 2019, Athens, Greece

In MGen, we use metacontainers: and can change their respective values, we use an array-like to reason about dimensions. Since they can grow in length First we need an abstraction Infering dimension values

Figure 2. Transformation of the initial M-IR Fibonacci program using metacontainers. In this transformation we add definitions for the number of dimensions in the metacontainer.

Figure 3. Inferring tensor orders is done by disposing the metacontainers, which effectively means analyzing integer operations which are overlaid on top of the M-IR

Consequently, the relationship between a given tensor, and its order is no longer stored in the IR, but in the metacontainer. This property plays a crucial role in the next step of inferring the orders. Namely, by ignoring the metacontainers, we implicitly remove the dependencies with the tensor computations, and thus obtain only the overlaid IR as shown in Figure 3. This IR contains integer operations only, which are natively supported by LMS and due to its low-level representation, allows us to perform analysis and transformations using the optimizations already available in LMS. In this respect, we extend LMS with support for data-flow analysis based on constant propagation [53].

The result of the analysis is a mapping of mutable variables to immutable variables or constants, which can be substituted in the overlaid IR for specialization. Using the transform method we do the same for each metacontainer. After transform is executed on all metacontainers, pLen is substituted with another symbol or a constant. Consequently, dependencies in deps are automatically restored. Finally, we obtain an M-IR that holds all possible types and complex types of a variable, as well as the tensor orders at each point in the program. We can now proceed do the next step: inferring dimension values.

Infering dimension values First we need an abstraction to reason about dimensions. Since they can grow in length and can change their respective values, we use an array-like structure. In MGen, we use metacontainers:

Similarly to the Shape metacontainer used to represent tensor variables, we provide a method to return all dependencies of a DArray metacontainer. We also provide a transformation method, but unlike the Shape metacontainers, transformations in DArray return new instances of DArray. As metacontainers can be nested, deps and transform of the Shape metacontainer also take into consideration the DArray dependencies, and specialize the data-structure.

DArray metacontainer specialization is done through polymorphic structures that are built with arrays of staged elements for tensors with fixed order or staged arrays for tensors with variable order:

The two polymorphic structures are created in a forward pass of the M-IR and are stored in the dim component of each Shape metacontainer. This abstraction allows us to treat dimensions in a single codebase upon lowering an operation, but more importantly, once we use DArrayScalar,
dimension values are treated as scalar variables, which are simpler to analyze through data-flow analysis.

In addition, the DArray metaclass allows us to perform dimension inspection and retrieve the order of the tensor. Depending on the type of the instance of the DArray, these checks will either be resolved during staging, or to propagate in the runtime of the generated code. This allows us to separate the overloaded operators in MATLAB. For example, consider the pointwise addition operator that is overloaded for two tensors of order \( n \) as well as one tensor of order \( n \) and a scalar:

```java
override def transformStn(stm: Stm) = stm match {
  case TP(lhs_sym, rhs_def@ShapePlus(a, b)) => {
    val (da, db) = (Shape(a), Shape(b))
    // separate the overloaded implementations
    var (r, d) = (da.isScalar(), db.isScalar()) match {
      case (Const(true), _) => (shape_plus_1(b,a), db)
      case (_, Const(true)) => (shape_plus_1(b,a), da)
      case _ =>
    }
    assert(check_for_equal_dimensions(da, db))
    (shape_plus_n/fa, fb), da)
  }
  // reconstruct the resulting metaclass
  val shape = Shape(r)
  shape.dims = Some(d)
  shape.pType = Shape(lhs_sym).pType
  shape.pComplex = Shape(lhs_sym).pComplex
  shape.exp
}
```

When the lowering phase completes, we obtain yet another overlaid IR that reasons about dimension values. Then we perform another round of analysis based on constant propagation. Similarly as before, the analysis is then applied to all metaclasses involved, which performs substitutions in the Shape metaclass and in the DArray metaclass as well. When substitutions in DArray instances occur, a staged DArray can also be specialized to a scalar DArray. In that particular case, the old DArray metaclass is disposed, and a new one is generated.

At the end of the shape inference phase, we have inferred the shape and the type of each tensor. With that information, the next phase is lowering the computation into a C-like representation.

### 4.6 Conversion to C-IR

To start the conversion, we traverse the M-IR forward and map each Shape metaclass into a CShape metaclass. CShape resembles Shape, but differs in important ways. Instead of maintaining a set of possible types for a tensor, it includes a type descriptor \( td \) and a complex type descriptor \( cd \) that keep track of the current type and complex type of the tensor at a particular location in the C-IR. It also includes a staged variable that represents the array responsible for the tensor data. As MATLAB tensors can take arbitrary shape, we represent the tensor data as a HeapArray of any type, which (as the name suggests) represents an array allocated on the heap. The interface of CShape is given as follows:

```java
abstract class CShape(shapeMIR: IR.Shape) {
  val id: C.Exp[C.HeapArray[Any]] // tensor data
  val td: C.Var[Type] // tensor type
  val cd: C.Var[Boolean] // complex type
  val ld: DArray // tensor dimensions
  def allocate (): C.Rep[Unit] // allocate memory
  def terminate (): C.Rep[Unit] // free memory
  def getCArray (tp: Type, isComplex: Boolean): CArray
}
```

The 1d parameter represents a DArray instance, which is a one-to-one translation of an M-IR Shape .dims metacontainer. The metadata in this metaclass provides us with all necessary information to allocate tensor data. For this purpose we include a method allocate to govern heap allocations, and terminate to relinquish them.

CShape specializes on input and output tensors, as well as tensors that represent intermediate operations. The input and output tensors are represented with C-struct objects containing memory regions for the data, as well as memory and type information, while the intermediate tensors are kept "unboxed" in the generated code.

#### Memory allocations

When lowering a MATLAB computation to C-IR, we need to allocate memory regions to store the results of the computation. When assigning and reading tensor variables in the M-IR, we also need to allocate and copy tensor data in the C-IR. As the CShape holds all required information including dimension, type and complex type, each allocation generates code that calculates the required memory in bytes, and invokes malloc. Each assignment and each read results in a memcpy call. Each CShape is terminated at the end of the block where it is allocated.

#### Type specialization

To lower M-IR computations into C-IR, we need to know the exact type of every variable at a given point in the program. To achieve that, we specialize a CShape metaclass to a CArray metaclass with fixed type and fixed complex type:

```java
abstract class CArray {
  val type : Type // fixed type
  val complex : Boolean // fixed complex type
  val size (): Rep[Int] // linear tensor size
  def apply (i: Rep[Int]): Element
  def update (i: Rep[Int], s: Element): Unit
}
```

Since CShape can hold multiple types, we devise a method concretize, that takes all possible types of all operands of a MATLAB operator, and generates nested switch statements for all possible type combinations. The possible types are provided by the M-IR Shape metacontainer for each operand, and the valid type combinations per operator are then pruned through the corresponding TypeFunction also provided by the M-IR. This routine generates all possible combinations of CArray, and for each we include the lowering as a continuation in the concretize routine.

The abstraction vastly simplifies the lowering to C-IR. Consider the lowering of the addition operator:
The primitive numerical operations are then abstracted in the tensor and performs addition. The abstraction allows us to handle any type combination and any complex type combination in a single codebase.

**Numerical operations** LMS supports the operations for all MATLAB primitives, but does not provide support for saturated arithmetic. Therefore we extended LMS, providing the standard operations including addition, subtraction, multiplication, division and power with saturation. Some of these functions are straightforward to implement:

```java
void saturated_subtraction_u32_t (u32_t a, u32_t b) {
    if (a < b) return 0; else return a - b;
}
```

For others, however, we used the MATLAB Coder [26] to generate different type combinations, and then we staged the generated code back in LMS. For example, consider a power function using 32-bit unsigned integers:

```java
void saturated_power_u32_t (uint32_t a, uint32_t b) {
    uint32_t x, ak, bu; int32_t exitg1; uint64_t u0;
    ak = a; x = 1U; bu = b;
    do {
        exitg1 = 0;
        if ((bku & 1U) == 0U) {
            u0 = uint64_t (ak * x);
            if (u0 > 4294967295UL) u0 = 4294967295UL;
            x = (uint32_t)u0;
        }
        bku *= bu;
        if ((int) bku == 0) exitg1 = 1; else {
            u0 = uint64_t (ak * x);
            if (u0 > 4294967295UL) u0 = 4294967295UL;
            ak = (uint32_t)u0;
        }
    } while (exitg1 == 0);
    return x;
}
```

The primitive numerical operations are then abstracted in the Element metaclass with polymorphic structures that can specialize on real or complex elements, having any of the 12 primitives in MATLAB. This metaclass is implemented in a similar fashion as described in Section 3.

### 4.7 Vectorization

In MGen we vectorize each operator individually. The vectorization is done through layers of abstractions that build on top of the lms-intrinsics package [49]. We provide a short description for each layer, starting from low-level abstractions and working our way upwards.

**Packed operations** On top of the lms-intrinsics, we build abstractions using metaclasses. We use the Packed metaclass that contains a staged variable representing a vector primitive and its base type. Then we provide operators that also include a parameter that describes the set of available ISAs. This parameter allows us to opportunistically select the best ISA, and dispatch the corresponding implementation:

```java
def packed_plus (ap: Packed, bp: Packed, setISAs: Set[ISA] = {}): Packed = {
    a: Packed, b: Packed, setISAs: Set[ISA]
    a typ match {
        case DoubleType =>
            mm_add_pd (ap.get[__m128d], b.get[__m128d])
        case ULongType | LongType =>
            mm_add_epi64 (a.get[__m128i], b.get[__m128i])
        case UIntType | IntType =>
            mm_add_epi32 (a.get[__m128i], b.get[__m128i])
        case UShortType | ShortType =>
            mm_add_epi16 (a.get[__m128i], b.get[__m128i])
        case UByteType | ByteType =>
            mm_add_epi8 (a.get[__m128i], b.get[__m128i])
        case CharType =>
            mm_add_epi8 (a.get[__m128i], b.get[__m128i])
        case _ => packed_plus_error (a, b, setISAs)
    }
}
```

In MGen, we do not provide all possible type combinations for packed operators. Instead, we only focus on vector operations that involve equal types.

**Vector operations and vector elements** From this point forward, we follow [48] to build abstractions for data-level parallelism, however, we use metaclasses instead of type classes. We abstract vector operations on top of packed operations. They are responsible for dispatching the correct implementation to satisfy MATLAB type interaction. For example, if saturated arithmetic is expected, they dispatch the implementation that uses saturation, and if not, they proceed with packed operators that do not reason about overflows and underflows.

VectorElement is a metaclass, representing complex or real vector elements. They can reason about the layout of the vector primitive they represent, providing specialized
implementations for split complex representation or an interleaved complex representation, using the already defined vector operations.

\textbf{VArray} Finally, to enable vectorization, we specialize \texttt{CArray} into \texttt{VArray}, providing vectorized loads and stores that apply or update an instance of \texttt{VectorElement} in the array.

\begin{verbatim}
  abstract class VArray { import C._
    val tpe: Type
    def apply (i: Rep[Int]): VectorElement
    def update (i: Rep[Int], s: VectorElement): Unit
  }
\end{verbatim}

\texttt{VArray} can also be further specialized, representing arrays of complex or real elements.

Every time an operator is lowered to the C-IR, we use the available set of ISAs provided as input in MGen. If vectorization is available, the operator will generate vectorized code. If not, MGen will switch to the scalar version that is available for all type combinations.

\section{5 Results}

In this section we evaluate MGen’s capability to generate correct C code for a given set of MATLAB functions. First, we give an overview on the infrastructure built to validate MGen features compared to (1) MGen, (2) MATLAB Coder, (3) Tamer, (4) MiX10 and Mc2For, (5) MATISSE.

\begin{table}[h]
\centering
\caption{MGen features compared to (1) MGen, (2) MATLAB Coder, (3) Tamer, (4) MiX10 and Mc2For, (5) MATISSE.}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
& 1 & 2 & 3 & 4 & 5 \\
\hline
\textbf{Analysis} & & & & & \\
Type inference & ✓ & X & ✓ & N / A & X \\
Shape inference & ✓ & ✓ & ✓ & N / A & X \\
Overloaded built-ins & ✓ & ✓ & ✓ & N / A & X \\
Range analysis & X & ✓ & ✓ & N / A & ✓ \\
\hline
\textbf{Code Generation} & & & & & \\
Arithmetic operations & ✓ & X & N / A & ✓ & X \\
Saturation arithmetics & ✓ & ✓ & N / A & X & X \\
Memory optimizations & X & ✓ & N / A & ✓ & ✓ \\
Explicit vectorization & ✓ & ✓ & N / A & X & X \\
\hline
\end{tabular}
\end{table}

For validation, MGen assumes a pre-installed MATLAB version 800.0.42.1. All tests are implemented using ScalaTest \cite{3}. We validate correct type inference by performing assignments in combination with nested if-conditional branches and loops. MGen passes all our tests, inferring all types at each point in the code. Mc2For \& MiX10 use Tamer \cite{15} as a front-end to perform type inference and analysis; thus, we only consider Tamer-generated IR. All our tests suggest that types are correctly inferred at each location in the code. MATLAB Coder and MATISSE variables expect each variable to have a single type at each point of the program and would fail on programs such as:

\begin{verbatim}
  function [x] = validate (A, B)
  x = foo(A, B);
  end
\end{verbatim}

Then we generate code for each operator, and instantiate it with different inputs. For each tensor input:

1. The type can be one of the 12 primitive types.
2. The tensor can represent complex or real data.
3. The tensor order can either be fixed to 1, 2, 3, or be passed as a variable.
4. If the tensor order is variable, then the values for each dimensions are passed as an array. If the order is constant, then we pass a tensor of order 1, 2, 3 with constant or variable dimensions.
5. We repeat the test several times, producing scalar, SSE, and AVX code in each case.

The above creates more than 144 different instantiations per single tensor. MGen produces results which are binary compatible with MATLAB in most cases. All operations using integral types are bitwise identical, as well as IEEE754 computations with standalone implementation. However, for trigonometric, logarithms, power functions and other libc-dependant functions, binary compatibility is no longer guaranteed. MATLAB Coder handles all numerical types, with few exception. For example, the plus operator fails once characters are added to 8-bit integers. MATISSE does not support complex numbers, and, along with Mc2For / MiX10, none of them support saturated arithmetic.

\textbf{Type inference validation} We validate correct type inference by performing assignments in combination with nested if-conditional branches and loops. MGen passes all our tests, inferring all types at each point in the code. Mc2For \& MiX10 use Tamer \cite{15} as a front-end to perform type inference and analysis; thus, we only consider Tamer-generated IR. All our tests suggest that types are correctly inferred at each location in the code. MATLAB Coder and MATISSE variables expect each variable to have a single type at each point of the program and would fail on programs such as:
Flops / Cycle [f/c]

Figure 4. Performance evaluation of MGen: 10 iterations of gradient descent applied on various matrix sizes

1 function [result] = addition(a)
2 while a < 10
3 a = a + int32(1);
4 end;
5 result = a;
6 end

Shape inference validation We validate shape inference in the same manner as validating type inference, but we also consider indexed assignments. MGen and MATLAB Tamer pass all our tests. MATLAB Coder, as well as MATISSE fail when extending a matrix to a tensor of order 3. Both generators have troubles handling overloaded operators. Consider the function below, where a is a matrix and b is a scalar:

1 function [result] = foo(a, b)
2 if (rand > 0.5) c = a .* a, else c = b .* b, end;
3 result = a + c;
4 end

In this function, the addition at line 3 will either result in a matrix-matrix addition, or a matrix-scalar addition, which is not handled in either generator.

Validation of programs We collected a set of benchmarks, acquiring MATLAB code from a variety of sources and related projects, including Mc2For [25], FALCON [44], OTTER [38], and The MathWorks Central File Exchange [28]. The benchmarks sample the subset of MATLAB supported by MGen, including standard constructs such as if-else, for, and while loop statements, dynamic growth of tensor order, array growth by out-of-bound array indexing, multiple type values per variable, and built-in function overloading. A brief description of each benchmark is given in Table 3. Our experiments confirm that MGen-generated code produces the same functionality as the MATLAB runtime.

Performance evaluation In the current state, MGen does not offer competitive performance. The biggest impediment is the lack of memory optimizations. Namely, for each intermediate computation, memory is allocated to facilitate the temporary tensor. When a mutable tensor is read or assigned, MGen performs copy-on-read and copy-on-write, or in other words copies the memory region from the temporary tensor to the mutable tensor or vice-versa, instead of computing it in-place. Consequently, this imposes significant overheads, in particular if the computation is in the hot-path of the code.

Table 3. Test-cases used in MGen

<table>
<thead>
<tr>
<th>benchmark</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>calculates the $n$-th Fibonacci number. The benchmark includes a simple for-loop.</td>
</tr>
<tr>
<td>gd</td>
<td>calculates 1000 iterations of the gradient descent algorithm. The benchmark includes a for-loop with matrix transposition, matrix-vector multiplication, and pointwise addition and multiplication of vectors.</td>
</tr>
<tr>
<td>adpt</td>
<td>finds the adaptive quadrature using Simpson's rule. This benchmark features an array whose size cannot be predicted before compilation.</td>
</tr>
<tr>
<td>bbas</td>
<td>uses Babai's algorithm computed on fixed-sized arrays.</td>
</tr>
<tr>
<td>babt</td>
<td>is the standard bubble sort algorithm. This benchmark contains nested loops and consists of many array read and write operations.</td>
</tr>
<tr>
<td>capr</td>
<td>computes the capacitance of a transmission line using the finite difference and Gauss-Seidel method. It is loop-based, having scalar operations on small-sized arrays.</td>
</tr>
<tr>
<td>clos</td>
<td>calculates the transitive closure of a directed graph. It contains matrix multiplication operations between two 450-by-450 arrays.</td>
</tr>
<tr>
<td>crni</td>
<td>computes the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a 2300-by-2300 array.</td>
</tr>
<tr>
<td>dich</td>
<td>computes the Dirichlet solution to Laplace's Equation. It's also a loop-based program which involves basic scalar operation on a small-sized array.</td>
</tr>
<tr>
<td>diff</td>
<td>calculates the diffraction pattern of monochromatic light through a transmission grating for two slits. This benchmark also features an array whose size is increased dynamically like as in the benchmark adpt.</td>
</tr>
<tr>
<td>ffi</td>
<td>computes the finite-difference solution to the wave equation. It is a loop-based program which involves basic scalar operation on a 2-dimensional array.</td>
</tr>
<tr>
<td>mbrit</td>
<td>computes a Mandelbrot set with a specified number of elements and number of iterations. This benchmark contains elementary scalar operations on complex data.</td>
</tr>
<tr>
<td>nb1d</td>
<td>simulates the gravitational movement of a set of objects. It computes on vectors inside nested loops.</td>
</tr>
</tbody>
</table>

We set up the benchmark to run for 10 iterations, operating on an $n \times n$ double precision matrix and a double precision vector of size $n$. For all 3 versions, the flop count is given as $4n^2 + n$ flops, and we report the results in flops / cycle.

Figure 4 shows the performance profile. MGen generated code is up to 37 times slower than MATLAB Coder and up to 8 times slower than MATLAB VM execution time.
5.2 Limitations
The work presented in this paper focused on the abstractions required to stage MATLAB code and to produce correct C code. However, many performance optimizations are still missing. These limitations are not due to the proposed abstraction model of metacontainers, but opportunities that this work has not yet explored.

High level optimizations MGen performs certain high level optimization such as dead code elimination and common subexpression elimination. However, others such as multi-level tiling, loop merging, and loop exchange are not implemented in this work. Even before applying loop optimization, semantic properties of matrix operations can be used to perform high-level optimizations [4]. Prior work [9, 16, 30] shows that MATLAB programs can be partially evaluated and optimized by automatically transforming loops to equivalent computations already available in the built-in operations. This suggests that optimization opportunities are available even before staging the initial MATLAB code.

Memory optimizations As indicated in the evaluation subsection, MGen does not employ any memory optimizations. These problems can be alleviated by performing in-place computations when possible, reusing memory regions in subsequent computations and coalescing arrays as suggested in prior work [21].

Range analysis Range analysis is not implemented in MGen. As a result, tensor indexing and indexed assignment operations must undergo bound checks to ensure valid memory accesses. Many of these runtime checks can be removed, as shown in prior work [13, 40].

6 Related Work
The Sable Lab at McGill University has done extensive work around the MATLAB language [8], developing an open-source MATLAB virtual machine, toolkits for static compilation and source-to-source MATLAB translation. We start with a brief overview.

M2VM [10] is a virtual machine that performs function specialization based on the runtime knowledge of the types and shapes in function calls. M2VM supports hundreds of built-in functions, but no integer or single-precision float matrices. M2VM uses an LLVM-based JIT compiler and relies on autovectorization.

MATLAB Tamer [15] is a compiler toolkit that translates MATLAB programs into an IR suitable for static compilation. It uses static analysis techniques to infer shape, class, and complex information, supporting more than 300 built-in MATLAB functions. Their type and shape behaviour is specified using a set of DSLs, which are then processed by ANTLR [36], resulting in AST trees that will be interpreted by the MATLAB Tamer tool to perform analysis.

Prior work on compiling MATLAB started with the FALCON compiler [1, 14, 44] to generate FORTRAN90 code. It uses aggressive type inference for base types (doubles and complex) and matrix shapes. Further inference techniques were introduced in [20, 22] and Olmos et al. [35] for partial support of the MATLAB type system.

MATLAB Tamer, generating Fortran and X10 code respectively. They support integer types but no saturation arithmetic and rely on autovectorization by the compiler.


Orthogonally to our work, there have been approaches to translate MATLAB/Octave to languages suitable for multicore or GPU architectures. MEGHA [37] compiler MATLAB/Octave scripts to CUDA and C/C++ code using heuristics to determine the partition.

7 Conclusion
We used stage polymorphism as an abstraction tool for building a MATLAB-to-C code generator. In particular, we demonstrated that by using the proposed metacontainer abstraction, we can simplify the analysis and transformation of tensor expressions, and handle many aspects in generating code for a dynamically typed numeric DSL such as MATLAB.

Specifically, we show how metacontainers maintain relationships between IR nodes that no longer require to be explicitly encoded in the IR and thus reduce its complexity. This property allowed us to systematically add nodes in the IR to build an overlay IR that can reason about type and shape propagation of tensor computations. Using the overlay IR we performed type and shape inference with data-flow analysis in a way agnostic towards the structure of the metacontainer and the actual tensor computation. By specializing metacontainers, we demonstrated that we can provide a powerful abstraction to build a code generator for a subset of MATLAB, supporting all primitive types, including explicit vectorization to produce SIMD code.

Acknowledgments
This work was supported in part by NSF awards 1553471, 1564207, 1918483, and DOE award DE-SC0018050.
References


