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

Abstract

We propose a novel approach for compiling MATLAB and similar languages that are characterized by tensors with dynamic shapes and types. We stage an evaluator for a subset of MATLAB using the Lightweight Modular Staging (LMS) framework to produce a compiler that generates C code. But the first Futamura projection alone does not lead to efficient code: we need to refine the rigid stage distinction based on type and shape inference to remove costly runtime checks.

To this end, we introduce a stage-polymorphic data structure, that we refer to as *metacontainer*, to represent MATLAB tensors and their type and shape information. We use metacontainers to efficiently "inject" constructs into a high-level intermediate representation (IR) to infer shape and type information. Once inferred, metacontainers are also used as the primary abstraction for lowering the computation, performing type, shape, and ISA specialization. Our prototype MATLAB compiler MGen produces static C code that supports all primitive types, heavily overloaded operators, many other dynamic aspects of the language, and explicit vectorization for SIMD architectures.

CCS Concepts • **Software and its engineering → Polymorphism**; **Source code generation**.

Keywords staging, polymorphism, generic programming, MATLAB, tensor computations, SIMD

ACM Reference Format:
Alen Stojanov, Tiark Rompf, and Markus Püschel. 2019. A Stage-Polymorphic IR for Compiling MATLAB-Style Dynamic Tensor Expressions. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357765.3359514

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

ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359514

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 haven 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

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

will simply execute the arithmetic operation, while

```
val (a, b): (Rep[Int], Rep[Int]) = (2, 3)
val c: Rep[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:

```
type Idt[T] = T
def add(a: Idt[Int], b: Idt[Int]) = a + b  // regular
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`:

```
def add[R[_]] (a: R[Int], b: R[Int])
  (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:

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

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

```
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`:

```
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:

```
val value: Rep[Any] // Staged expression, type unknown
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:

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

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

```
class Number (        // class wrapper - metacontainer
  val value: Rep[Any] // metacontainer reference
  val typ  : Rep[Typ] // reference metadata
) {}
```

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:

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

or can become routines in the metacontainer itself:

```
class Number(val typ: Rep[Typ], val value: Rep[Any]) {
  def plus (rhs: Number): Number =
    (typ, rhs.typ) match {
      case (DoubleTyp, DoubleTyp) =>
        new Number(DoubleTyp,
          numeric_plus[Double](value, rhs.value))
      case (DoubleTyp, FloatTyp) =>
        plus(rhs.cast(DoubleTyp))
      case (DoubleTyp, IntTyp) => {/* all others */}
}}
```

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:

```
class FloatNumber(value: Rep[Float])
    extends Number (FloatType, value) {
  override def plus (rhs: Number) = rhs.typ match {
    case DoubleTyp => cast(DoubleTyp).plus(rhs)
    case FloatTyp  => new FloatNumber(
        numeric_plus[Float](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:

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

Table 1. Subset of MATLAB supported in MGen

Function	Description
`+ - ./ .* .^`	Pointwise arithmetics
`< <= > >= == ~=`	Relational operators
`&& || xor not`	Logical operators
`mtimes`	Matrix-matrix multiplication
`A'` / `transpose`	Matrix transpose
`a/sin a/cos a/tan`	Trigonometry functions
`abs sqrt`	Absolute value or square root of a tensor
`exp log log10`	Power and logarithmic functions
`ceil floor round fix`	Rounding
`mod rem`	Modulo and reminder
`conj`	Complex conjugate
`length size numel`	Tensor dimension inspection.
`min max sum mean`	Min, max, sum, mean of a tensor
`zeros ones rand`	Tensor initializations with 0, 1 or random
`: colon`	Unit-spaced vector indexing
`horzcat vertcat`	Horizontal and vertical tensor concatenation

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.

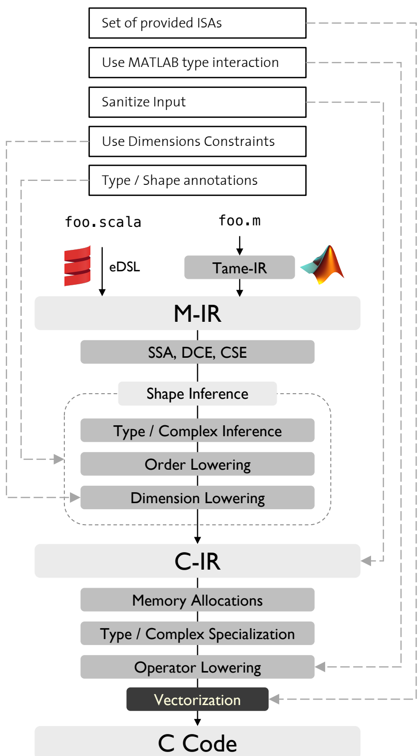

Figure 1. MGen Design Overview

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:

```
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:

```
abstract class F[T]{ def apply(t: Seq[T]): Option[T] }
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:

$$
\begin{array}{rcrcl}
\textit{double} & \times & \textit{double} & \longrightarrow & \textit{double} \\
\textit{float} & \times & \textit{double} & \longrightarrow & \textit{float} \\
\textit{double} & \times & \textit{int16} & \longrightarrow & \textit{int16} \\
\ldots & \times & \ldots & \longrightarrow & \ldots
\end{array}
$$

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:

```
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:

```
case class ShapePlus  (...) extends ShapeDef {...}
case class ShapePlus1 (...) extends ShapeDef {...}
case class ShapePlusN (...) 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:

```
class Shape (
  val exp: Exp[Shape] // metacontainer reference
) {
  var pType = Set[Type]()    // possible types
  var pComplex = Set[Bool]() // possible complex types
  var pLen = Option.empty[Exp[Int]] // tensor order
  var dims = Option.empty[DArray] // 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 raise an exception. With the 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.

Infering 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:

```
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:

```
// override LMS methods for anti-dependencies
def softSyms(e: Any): List[Sym[Any]] = e match {
  case d: ShapeDef => findDefinition(d) match {
    case Some(TP(s, _)) =>
      Shape(s).deps() ::: super.softSyms(e)
  }
  case _ => super.softSyms(e)
}
```

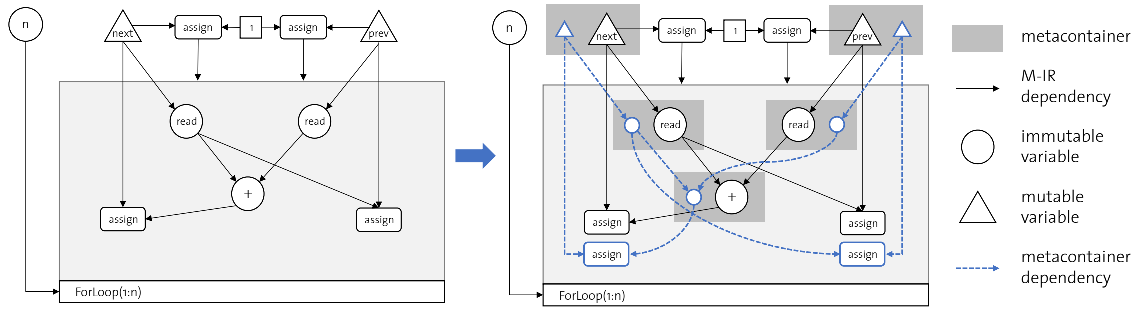

Figure 2. Transformation of the initial M-IR Fibonacci program using metacontainers. In this transformation we add definitions for the number of dimensions in the metacontainer.

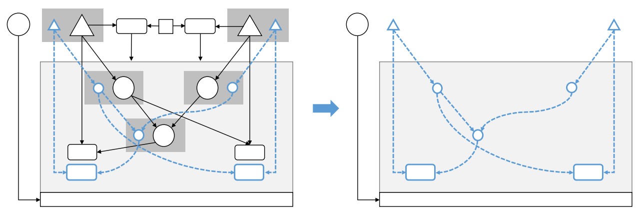

Figure 3. Inferring tensor orders is done by disposing the metacontainers, which effectively means analyzing integer operations which are overlaid on top of the M-IR

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:

```
abstract class DArray {
  def length    (): Exp[Int]
  def apply     (i: Exp[Int]) : Exp[Int]
  def update    (i: Exp[Int], v: Exp[Int]): Unit
  def deps      (): List[Sym[Any]]
  def transform (f: Transformer): DArray
  def isScalar  (): Exp[Boolean]
  def isVector  (): Exp[Boolean]
  def isMatrix  (): Exp[Boolean]
}
```

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:

```
type RInt   = Exp[Int]
type Scalar = Array[Exp[Int]]
type Staged = Exp[Array[Int]]

case class DArrayScalar(dims: Scalar) extends DArray {
  def length (): RInt = Const(dims.length)
  def apply (idx: RInt) : RInt = idx match {
    case Const(c) => dims.applyOrElse(c, Const(1))
  }
  def update (idx: RInt, v: RInt) = idx match {
    case Const(c) => dims(c) = v
  }
}
class DArrayStaged(n:RInt,arr:Staged) extends DArray {
  def length (): RInt              = n
  def apply  (i: RInt): RInt       = arr(i)
  def update (i: RInt, v: RInt) = { arr(i) = v }
}
```

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 upton 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 metacontainer 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 ropagate 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:

```
override def transformStm(stm: Stm) = stm match {
 case TP(lhs_sym, rhs_def@ShapePlus(a, b)) => {
   val (sa, sb) = (Shape(a), Shape(b))
   val (da, db) = (sa.dims.get, sb.dims.get)
   // 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(a,b), da)
     case _ =>
       assert(check_for_equal_dimensions(da, db))
       (shape_plus_n(fa, fb), da)
   }
   // reconstruct the resulting metacontainer
   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 metacontainers involved, which performs substitutions in the Shape metacontainer and in the DArray metacontainer 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 metacontainer 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 metacontainer into a CShape metacontainer. 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:

```
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 ld parameter represents a DArray instance, which is a one-to-one translation of an M-IR Shape.dims metacontainer. The metadata in this metacontainer 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 dimension 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 informations 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 metacontainer to a CArray metacontainer with fixed type and fixed complex type:

```
abstract class CArray {
  val tpe       : Type      // fixed type
  val complex   : Boolean   // fixed complex type
  def 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:

```
trait MIR2CIR extends NestedBlockTraversal {
  val IR: MIR, C: MCIR
  def traverseStm (stm: IR.Stm): Unit = stm match {
    case (s: Sym[_], op@ShapePlusN(a, b)) =>
      val (ca, cb) = CShape(s), CShape(a))
      val cs = CShape(s)
      // use 'concretize' to generate all possible
      // type combinations for the addition operator
      concretize(op)(ca, cb)(cs)(
        (p: CArray, q: CArray, r: CArray) => {
          // lower a specific type combination
          C.loop(r.size(), i => r(i) = p(i) + q(i))
        }
      )
    // all other cases are implemented here
}
```

The routine shown above iterates through all elements of 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:

```
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:

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

The primitive numerical operations are then abstracted in the `Element` metacontainer with polymorphic structures that can specialize on real or complex elements, having any of the 12 primitives in MATLAB. This metacontainer 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 metacontainers. We use the `Packed` metacontainer 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:

```
def packed_plus (
  ap: Packed, bp: Packed, setISAs: Set[ISA]
): Packed = {
  assert(ap.tp() == bp.tp())
  val exp = ap.getISA() match {
    case AVX if setISAs.contains(AVX2) =>
      packed_plus_avx2(ap, bp, setISAs)
    case AVX if setISAs.contains(AVX)  =>
      packed_plus_avx (ap, bp, setISAs)
    case SSE if setISAs.contains(SSE2) =>
      packed_plus_sse2(ap, bp, setISAs)
    case SSE if setISAs.contains(SSE)  =>
      packed_plus_sse (ap, bp, setISAs)
    case _ => packed_plus_error(ap, bp, setISAs)
  }
  Packed(ap.tp(), exp)
}
protected def packed_plus_sse2 (
  a: Packed, b: Packed, setISAs: Set[ISA]
) = a.typ match {
  case DoubleType =>
    _mm_add_pd (a.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 _ if setISAs.contains(SSE) =>
    packed_plus_sse(a, b, setISAs)
  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 metacontainers 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 metacontainer, representing complex or real vector elements. They can reason about the layout of the vector primitive they represent, providing specialized

Table 2. MGen features compared to (1) MGen, (2) MATLAB Coder, (3) Tamer, (4) MiX10 and Mc2For, (5) MATISSE.

	1	2	3	4	5
Analysis					
Type inference	✓	×	✓	N / A	×
Shape inference	✓	×	✓	N / A	×
Overloaded built-ins	✓	×	✓	N / A	×
Range analysis	×	✓	✓	N / A	✓
Code Generation					
Arithmetic operations	✓	×	N / A	✓	×
Saturation arithmetics	✓	✓	N / A	×	×
Memory optimizations	×	✓	N / A	✓	✓
Explicit vectorization	✓	×	N / A	×	×

implementations for split complex representation or an interleaved complex representation, using the already defined vector operations.

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

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

`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.

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-generated code against MATLAB virtual machine execution for a given MATLAB function. Then we use the infrastructure to derive thorough tests to validate type and shape inference.

We compare MGen's capabilities to MATLAB Coder [26], a MathWorks commercial solution for MATLAB to `C/C++` compilation. We also compare it to the recent open source research alternatives MATISSE [5], Mc2For [25] and MiX10 [23]. The high level overview of this comparison is shown in Table 2. Finally, we evaluate MGen by generating correct `C` code for common MATLAB computations and discuss limitations.

5.1 Evaluation

Testing infrastructure Our evaluation is based on MATLAB version 2016A. MGen requires Java 8 and Scala SBT to compile, generates `C11` compliant code, and is compatible with Mac OS X, Linux and Windows. Our testing environment uses Mac OS X 10.11.6, Java JDK version 1.8.0 144-b01, Intel ICC compiler version 17.0.4, and LLVM using clang version 800.0.42.1. All tests are implemented using ScalaTest [3]. For validation, MGen assumes a pre-installed MATLAB version. It is able to spawn a MATLAB VM process from within the JVM and execute arbitrary MATLAB code. The testing infrastructure allows us to execute any MATLAB function, in MATLAB VM, or as a standalone MGen-generated `C` program, and compare the results.

Validation of operators With the testing infrastructure we can easily invoke any function of any arity in both MGen and MATLAB. We generate a small MATLAB snippet replacing `foo` with the corresponding operator:

```
function [x] = validate (A, B)
  x = foo(A, B);
end
```

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 functons, 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.

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 [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:

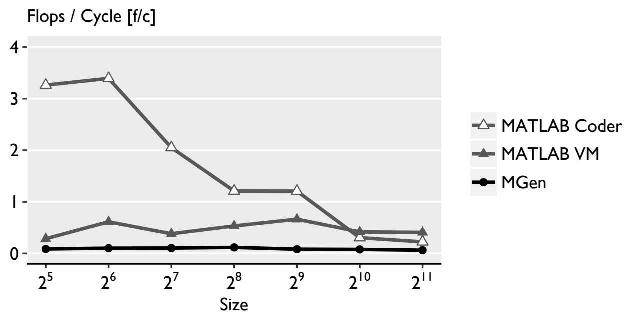

Figure 4. Performance evaluation of MGen: 10 iterations of gradient descent applied on various matrix sizes

```
function [result] = addition(a)
  while a < 10
    a = a + int32(1);
  end;
  result = a;
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

Table 3. Test-cases used in MGen

fib	calculates the n-th Fibonacci number. The benchmark includes a simple `for`-loop.
gd	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.
adpt	finds the adaptive quadrature using Simpson's rule. This benchmark features an array whose size cannot be predicted before compilation.
bbai	uses Babai's algorithm computed on fixed-sized arrays.
bubl	is the standard bubble sort algorithm. This benchmark contains nested loops and consists of many array read and write operations.
capr	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.
clos	calculates the transitive closure of a directed graph. It contains matrix multiplication operations between two 450-by-450 arrays.
crni	computes the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a 2300-by-2300 array.
dich	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.
diff	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*.
fiff	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.
mbrt	computes a Mandelbrot set with a specified number of elements and number of iterations. This benchmark contains elementary scalar operations on complex data.
nb1d	simulates the gravitational movement of a set of objects. It computes on vectors inside nested loops.

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.

To illustrate the diminishing effects due to lack of memory optimizations on performance, we consider the the gradient descent benchmark. We evaluate 3 versions: MATLAB VM, MGen, and MATLAB Coder generated code. We benchmark on an Intel Core i7-4980HQ machine with 16GB of RAM and 25.6 GB/s bandwidth to main memory.

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.

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

Mc2For [25] and *MiX10* [23] are backends to MATLAB Tamer, generating Fortran and X10 code respectively. They support integer types but no saturation arithmetic and rely on autovectorization by the compiler.

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 Coder [26] is a MathWorks proprietary solution for MATLAB to C/C++ compilation. It supports a large subset of MATLAB and allows code customizations through directives and options but has also limitations. Until version R2016b, it did not support changing types through assignment or generation of standalone vector code.

MATISSE [5] compiles MATLAB to C or OpenCL [6] with focus on embedded systems and heterogeneous architectures. It uses aspect-oriented programming (AOP) concepts, using the LARA [7] domain specific language. It does not support complex numbers or variables that can hold multiple types at the same point in the program.

MATLAB has several free open-source alternatives, including Octave [19], Scilab [45] and FreeMat [11].

Orthogonally to our work, there have been approaches to translate MATLAB / Octave to languages suitable for multicore or GPU architectures [2, 18, 31, 38, 39]. 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

[1] George Almasi and David A. Padua. 2000. MaJIC: A Matlab Just-In-time Compiler. In *Languages and Compilers for Parallel Computing, 13th International Workshop, LCPC 2000, Yorktown Heights, NY, USA, August 10-12, 2000, Revised Papers (Lecture Notes in Computer Science)*, Samuel P. Midkiff, José E. Moreira, Manish Gupta, Siddhartha Chatterjee, Jeanne Ferrante, Jan Prins, William Pugh, and Chau-Wen Tseng (Eds.), Vol. 2017. Springer, 68–81. https://doi.org/10.1007/3-540-45574-4_5

[2] Prithviraj Banerjee, U. Nagaraj Shenoy, Alok N. Choudhary, Scott Hauck, C. Bachmann, Malay Haldar, Pramod G. Joisha, Alex K. Jones, Abhay Kanhere, Anshuman Nayak, S. Periyacheri, M. Walkden, and David Zaretsky. 2000. A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems. In *FCCM*. IEEE Computer Society, 39–48.

[3] Bill Venners, George Berger, Chua Chee Seng. [n.d.]. ScalaTest. http://www.scalatest.org. (Online; accessed 2019-04-19).

[4] Neil Birkbeck, Jonathan Levesque, and José Nelson Amaral. 2007. A Dimension Abstraction Approach to Vectorization in MATLAB. In *CGO*. IEEE Computer Society, 115–130.

[5] João Bispo and João M. P. Cardoso. 2017. A MATLAB subset to C compiler targeting embedded systems. *Softw., Pract. Exper.* 47, 2 (2017), 249–272. https://doi.org/10.1002/spe.2408

[6] João Bispo, Luís Reis, and João M. P. Cardoso. 2015. C and OpenCL generation from MATLAB. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong (Eds.). ACM, 1315–1320. https://doi.org/10.1145/2695664.2695911

[7] João M. P. Cardoso, Tiago Carvalho, José Gabriel F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro C. Diniz, and Zlatko Petrov. 2012. LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel (Eds.). ACM, 179–190. https://doi.org/10.1145/2162049.2162071

[8] Andrew Casey, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, Rahul Garg, Soroush Radpour, Olivier Savary Bélanger, Laurie J. Hendren, and Clark Verbrugge. 2010. McLab: an extensible compiler toolkit for MATLAB and related languages. In *Canadian Conference on Computer Science & Software Engineering, C3S2E 2010, Montreal, Quebec, Canada, May 19-20, 2010, Proceedings (ACM International Conference Proceeding Series)*, Bipin C. Desai, Carson Kai-Sang Leung, and Sudhir P. Mudur (Eds.). ACM, 114–117. https://doi.org/10.1145/1822327.1822343

[9] Hanfeng Chen, Alexander Krolik, Erick Lavoie, and Laurie J. Hendren. 2016. Automatic Vectorization for MATLAB. In *Languages and Compilers for Parallel Computing - 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers (Lecture Notes in Computer Science)*, Chen Ding, John Criswell, and Peng Wu (Eds.), Vol. 10136. Springer, 171–187. https://doi.org/10.1007/978-3-319-52709-3_14

[10] Maxime Chevalier-Boisvert, Laurie J. Hendren, and Clark Verbrugge. 2010. Optimizing Matlab through Just-In-Time Specialization. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Rajiv Gupta (Ed.), Vol. 6011. Springer, 46–65. https://doi.org/10.1007/978-3-642-11970-5_4

[11] Clement David. [n.d.]. FreeMat. http://freemat.sourceforge.net/. (Online; accessed 2019-04-19).

[12] Cliff Click and Keith D. Cooper. 1995. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.* 17 (March 1995), 181–196. Issue 2.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[14] Luiz De Rose and David Padua. 1996. A MATLAB to Fortran 90 Translator and Its Effectiveness. In *Proceedings of the 10th International Conference on Supercomputing (ICS '96)*. ACM, New York, NY, USA, 309–316. https://doi.org/10.1145/237578.237627

[15] Anton Willy Dubrau and Laurie J. Hendren. 2012. Taming MATLAB. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 503–522. https://doi.org/10.1145/2384616.2384653

[16] Daniel Elphick, Michael Leuschel, and Simon Cox. 2003. Partial Evaluation of MATLAB. In *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering (GPCE '03)*. Springer-Verlag New York, Inc., New York, NY, USA, 344–363. http://dl.acm.org/citation.cfm?id=954186.954207

[17] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

[18] Malay Haldar, Anshuman Nayak, Abhay Kanhere, Pramod G. Joisha, U. Nagaraj Shenoy, Alok N. Choudhary, and Prithviraj Banerjee. 2000. Match Virtual Machine: An Adaptive Runtime System to Execute MATLAB in Parallel. In *ICPP*. 145–152.

[19] John W. Eaton. [n.d.]. GNU Octave. https://www.gnu.org/software/octave/. (Online; accessed 2019-04-19).

[20] Pramod G. Joisha and Prithviraj Banerjee. 2003. The MAGICA Type Inference Engine for MATLAB. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Görel Hedin (Ed.), Vol. 2622. Springer, 121–125. https://doi.org/10.1007/3-540-36579-6_9

[21] Pramod G. Joisha and Prithviraj Banerjee. 2003. Static array storage optimization in MATLAB. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 258–268. https://doi.org/10.1145/781131.781160

[22] Pramod G. Joisha and Prithviraj Banerjee. 2007. A translator system for the MATLAB language. *Softw., Pract. Exper.* 37, 5 (2007), 535–578. https://doi.org/10.1002/spe.781

[23] Vineet Kumar and Laurie J. Hendren. 2014. MIX10: compiling MATLAB to X10 for high performance. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 617–636. https://doi.org/10.1145/2660193.2660218

[24] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 270–282. https://doi.org/10.1145/503272.503298

[25] Xu Li and Laurie J. Hendren. 2014. Mc2FOR: A tool for automatically translating MATLAB to FORTRAN 95. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and*

Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014, Serge Demeyer, Dave W. Binkley, and Filippo Ricca (Eds.). IEEE Computer Society, 234–243. https://doi.org/10.1109/CSMR-WCRE.2014.6747175

[26] MathWorks. [n.d.]. MATLAB Coder. https://www.mathworks.com/products/matlab-coder.html. (Online; accessed 2019-04-19).

[27] MathWorks. [n.d.]. MATLAB Documentation. https://mathworks.com/help/matlab/referencelist.html?type=function&listtype=alpha. (Online; accessed 2019-04-19).

[28] MathWorks. 2016. MATLAB Central File Exchange. https://mathworks.com/matlabcentral/fileexchange/.

[29] MathWorks. 2016. MATLAB Release Notes. https://www.mathworks.com/help/matlab/release-notes.html.

[30] Vijay Menon and Keshav Pingali. 1999. High-level semantic optimization of numerical codes. In *International Conference on Supercomputing*. 434–443.

[31] Vijay Menon and Anne E. Trefethen. 1997. MultiMATLAB Integrating MATLAB with High Performance Parallel Computing. In *SC*. IEEE, 30.

[32] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

[33] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for generic programming in space and time. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 15–28. https://doi.org/10.1145/3136040.3136060

[34] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 125–134. https://doi.org/10.1145/2517208.2517228

[35] Karina Olmos and Eelco Visser. 2003. Turning Dynamic Typing into Static Typing by Program Specialization in a Compiler Front-end for Octave. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), 26-27 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 141–150. https://doi.org/10.1109/SCAM.2003.1238040

[36] Terence John Parr and Russell W. Quong. 1995. ANTLR: A Predicated-*LL(k)* Parser Generator. *Softw., Pract. Exper.* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705

[37] Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. 2011. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 152–163. https://doi.org/10.1145/1993498.1993517

[38] Michael J. Quinn, Alexey G. Malishevsky, Nagajagadeswar Seelam, and Yan Zhao. 1998. Preliminary Results from a Parallel MATLAB Compiler. In *IPPS/SPDP*. 81–87. https://doi.org/10.1109/IPPS.1998.669894

[39] Shankar Ramaswamy, Eugene W. Hodges IV, and Prithviraj Banerjee. 1996. Compiling MATLAB Programs to ScaLAPACK: Exploiting Task and Data Parallelism. In *IPPS*. IEEE Computer Society, 613–619.

[40] Luís Reis, João Bispo, and João M. P. Cardoso. 2016. SSA-based MATLAB-to-C compilation and optimization. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*, Martin Elsman, Clemens Grelck, Andreas Klöckner, and David A. Padua (Eds.). ACM, 55–62. https://doi.org/10.1145/2935323.2935330

[41] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 2–9. https://doi.org/10.1145/2784731.2784760

[42] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 238–261. https://doi.org/10.4230/LIPIcs.SNAPL.2015.238

[43] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[44] Luiz De Rose and David A. Padua. 1999. Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Trans. Program. Lang. Syst.* 21, 2 (1999), 286–323. https://doi.org/10.1145/316686.316693

[45] Scilab Enterprises. [n.d.]. Scilab. https://www.scilab.org/. (Online; accessed 2019-04-19).

[46] Gaurav Sharma and Jos Martin. 2009. MATLAB®: A Language for Parallel Computing. *International Journal of Parallel Programming* 37, 1 (2009), 3–36. https://doi.org/10.1007/s10766-008-0082-5

[47] Chun-Yu Shei, Adarsh Yoga, Madhav Ramesh, and Arun Chauhan. 2011. MATLAB Parallelization through Scalarization. In *15th Workshop on Interaction between Compilers and Computer Architectures, INTERACT 2011, San Antonio, Texas, USA, February 12, 2011*. IEEE Computer Society, 44–53. https://doi.org/10.1109/INTERACT.2011.18

[48] Alen Stojanov, Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2014. Abstracting Vector Architectures in Library Generators: Case Study Convolution Filters. In *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom, June 12-13, 2014*, Laurie J. Hendren, Alex Rubinsteyn, Mary Sheeran, and Jan Vitek (Eds.). ACM, 14–19. https://doi.org/10.1145/2627373.2627376

[49] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD intrinsics on managed language runtimes. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle (Eds.). ACM, 2–15. https://doi.org/10.1145/3168810

[50] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 134:1–134:25. https://doi.org/10.1145/2584665

[51] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.

[52] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

[53] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. https://doi.org/10.1145/103135.103136