A functional pattern-based language in MLIR

A functional pattern-based language in MLIR

Martin Lücke[1], Michel Steuwer[2][0000−0001−5048−0741], and Aaron Smith[13]

[1] University of Edinburgh
[2] University of Glasgow
[3] Microsoft

1 Introduction

Machine learning systems are stuck in a rut. Paul Barham and Michael Isard, two of the original authors of TensorFlow, come to this conclusion in their recent HotOS paper [2]. They argue that while TensorFlow and similar frameworks have enabled great advances in machine learning, their current design and implementations focus on a fixed set of monolithic and inflexible kernels. They continue to say that *"this reliance on high performance but inflexible kernels reinforces the dominant style of programming model"* and argue that *"these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress"*.

To overcome these problems, we need new intermediate representations that break up the monolithic and inflexible kernels and represent computations using more flexible and finer grained primitives. MLIR [4] is a new framework for implementing custom compiler intermediate representations called *dialects* that integrate with one another. Recently proposed by Google, MLIR provides a dialect for the XLA HLO graph used as the intermediate representation of the TensorFlow XLA compiler. Additional dialects exist for example to represent loop-based polyhedral optimizations and for providing abstractions for linear algebra routines. MLIR provides a common infrastructure to implement dialects and transform between them, including lowering computations to a LLVM dialect that is then compiled with the normal LLVM compiler infrastructure.

In this paper, we introduce our first steps to implement RISE, a functional pattern-based language, as a MLIR dialect. RISE is a spiritual successor to LIFT [5,6] and represents computations as compositions of small generic patterns. Prior work has shown this pattern-based style efficiently represents complex multi-dimensional computations from the domains of tensor-algebra and scientific computing [6] along with convolutions and stencils [3]. Crucially, LIFT demonstrated that optimisation choices can be encoded as rewrite rules and applied automatically to achieve competitive performance.

MLIR provides a common infrastructure that enables deep technical integration among dialects. Encoding a functional pattern-based language as a MLIR dialect will enable us to explore the benefits – and drawbacks – of this representation compared to others, such as polyhedral or graph-based.

2 Rise: a Lift-like functional pattern-based language

Rise is a functional pattern-based language in the style of Lift [5,6]. Rise provides a set of data-parallel high-level patterns that are used to describe computations over higher dimensional arrays (aka tensors) in an abstract way. For example, the `map` pattern applies a given function to every element of the input array. The `zip` pattern combines two input arrays pairwise to produce an output array of pairs. The `reduce` pattern is customized with a binary reduction operator (such as addition), a matching neutral element (such as zero), and an input array that is reduced to a single value (such as the sum of all elements).

Using these patterns we can express larger computations by composition, such as matrix-matrix multiplication shown below:

```
fun(A : N.K.float => fun(B : K.M.float =>
  A |> map(fun(arow =>
    B |> map(fun(bcol =>
        zip(arow, bcol) |> map(*) |> reduce(+, 0) )) ))))
```

Using the `map` pattern twice, once on `A` and once on `B`, we perform the dot product computation to each combination of a row of matrix `A` and a column of matrix `B`. The dot product itself is expressed as a composition of the `zip`, `map` and `reduce` patterns.

Lift has shown how this high-level representation is rewritten in an exploratory process into a lower level representation using a set of low-level patterns that match features from the underlying hardware architecture (such as sequential and parallel variations of the `map` pattern). By automatically exploring different implementation and optimization choices Lift has demonstrated competitive performance for tensor-algebra [6] and stencils/convolutions [3].

In this paper we present our first steps to represent the high-level language and patterns of Rise in mlir. This requires implementing Rise's type system, fundamental language constructions such as lambda expressions, and Rise's high-level patterns in mlir. A full implementation of Rise as a mlir dialect would also include it's low-level patterns and rewrite system – which remain as future work.

3 Rise as a dialect in mlir

mlir is a framework for implementing custom intermediate representations. It uses SSA as the base representation which means that values are always defined before they are used. Operations may produce multiple results which are all themselves distinct SSA values. Each value in mlir has a type and mlir provides a set of standard types. Refer to the mlir specification for a good technical overview [1]. mlir allows dialects to add custom operations together with custom type systems. We will now discuss both for the Rise dialect, starting with the types.

3.1 Rise types

The types for the Rise dialect are shown as a class hierarchy diagram in Figure 3.1.

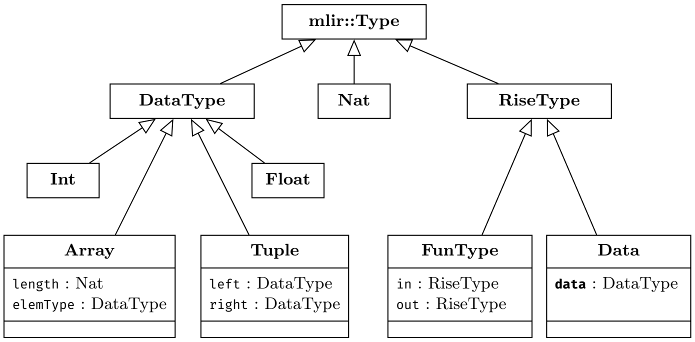


Fig. 1. Types of the Rise dialect

Rise types are divided into three categories that all inherit from `mlir::Type`:

- *Data types*: include `Int`, `Float`, `Array` and `Tuple` types.
- *Natural numbers*: this is used for tracking the length of the array in the type.
- *Rise types*: every Rise value has this type which is either a `FunType`, representing a Rise function, or a `Data`, wrapping a `DataType`.

These types follow the type system of the Rise language, but exclude – for now – type variables modelled as dependent function types. This type system prevents function types (which are a sub-type of `RiseType`) to be treated like data types and, for example, be stored in an array.

In the dialect implementation, we follow the notation of MLIR and prefix types by **`!rise.`**. For example, to express the type of a function from `Int` to `Int` we write: **`!rise.fun<data<int> -> data<int>>`**. To represent the function type of the Rise `map` pattern, written `map` : $(T \to U) \to [T]_N \to [U]_N$ in the functional notation, we write in MLIR:
`!rise.fun<fun<data<T> -> data<U>> -> fun<data<array<N,T>> -> data<array<N,U>>>>`.

3.2 Rise functional language constructs

Rise is a functional language based on lambda calculus. To express these language fundamentals, we added two operations to the Rise dialect: `lambda` and `apply` – for function abstraction and application.

Figure 2 shows the implementation sketches of the classes implementing the `lambda` and `apply` Rise operations. These are subclasses of the `mlir::Op` superclass.

The `lambda` operation models a lambda expression in Rise. It wraps a MLIR region of a single MLIR block. The arguments of the region are the arguments of

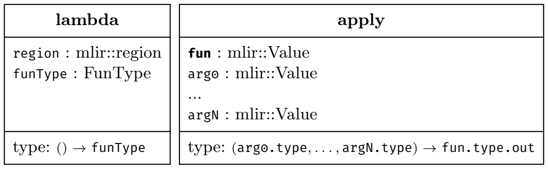


Fig. 2. Implementation sketches of `lambda` and `apply` classes.

the lambda. In addition to the region we store the RISE `FunType`. Essentially, a RISE `lambda` gives a MLIR region the meaning of a functional lambda expression.

The `apply` operation models the call of a RISE function. The function and its arguments are `mlir::Value`s and the operation type is the appropriate MLIR function type. The function in the `apply` operation can either be a RISE lambda or any other value with a RISE function type, such as a RISE pattern or a partially applied function.

The Listing below shows the representation of the identity lambda and its application. It is the representation of the RISE expression: `fun(x => x)(y)`

```
%id  = rise.lambda (%x) : !rise.fun<data<int> -> data<int>> {
     rise.return %x : !rise.data<int>
  }
%res = rise.apply %id %y
```

3.3 RISE high-level patterns

RISE is a pattern-based language with a number of data-parallel patterns used to describe computations. Each pattern is represented as an operation in the RISE dialect. Figure 3 shows the implementation sketches of the `map`, `zip`, and `reduce` patterns. The types are written in a shortened notation where $\mapsto$ represents a RISE `FunType` and arrays are written as $[T]_N$.

The `map`, `zip`, and `reduce` patterns are all customized with information that appear in their type: `N`, `dt`, `dt1`, or `dt2`. When constructing a pattern operation in MLIR these are passed as MLIR attributes.

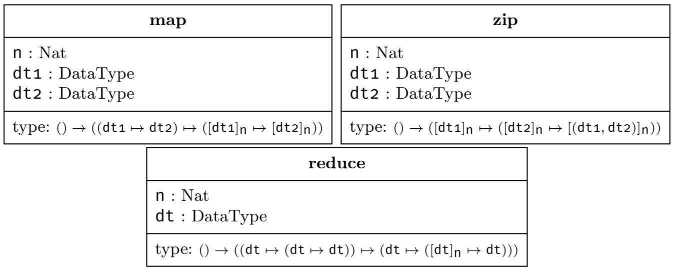
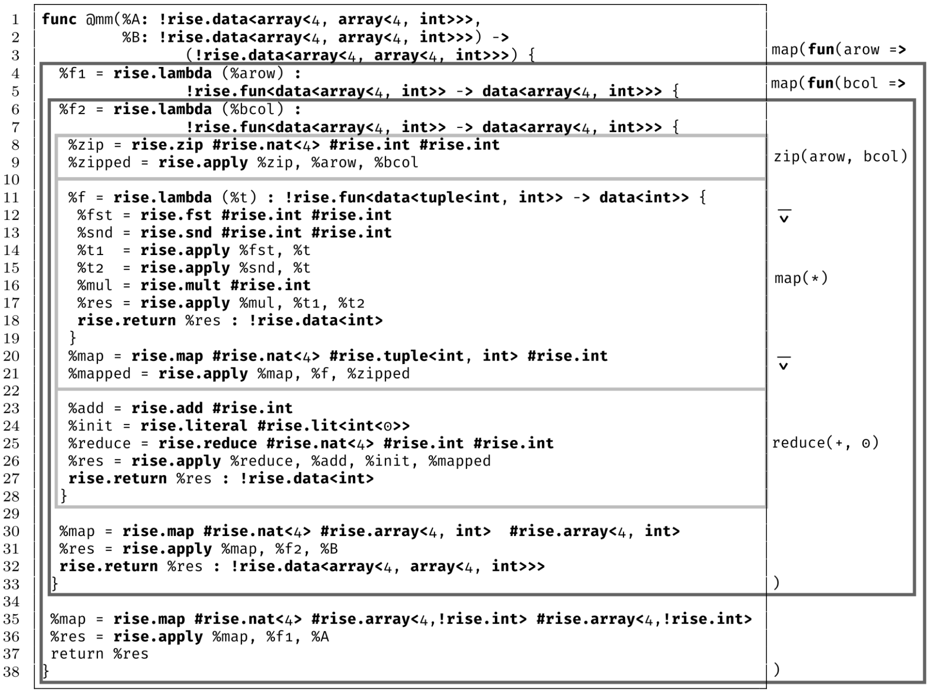

map
n : Nat
dt1 : DataType
dt2 : DataType
type: () → ((dt1 ↦ dt2) ↦ ([dt1]n ↦ [dt2]n))
zip
n : Nat
dt1 : DataType
dt2 : DataType
type: () → ([dt1]n ↦ ([dt2]n ↦ [(dt1, dt2)]n))
reduce
n : Nat
dt : DataType
type: () → ((dt ↦ (dt ↦ dt)) ↦ (dt ↦ ([dt]n ↦ dt)))

Fig. 3. Implementation sketches of classes representing RISE high-level patterns

3.4 Matrix-matrix multiplication example

Figure 4 below shows how to represent matrix-matrix multiplication in the Rise MLIR dialect. This directly corresponds to the functional Rise program seen earlier.



```
1  func @mm(%A: !rise.data<array<4, array<4, int>>>,
2          %B: !rise.data<array<4, array<4, int>>>) ->
3                 (!rise.data<array<4, array<4, int>>>) {
4    %f1 = rise.lambda (%arow) :
5                 !rise.fun<data<array<4, int>> -> data<array<4, int>>> {
6    %f2 = rise.lambda (%bcol) :
7                 !rise.fun<data<array<4, int>> -> data<array<4, int>>> {
8     %zip = rise.zip #rise.nat<4> #rise.int #rise.int
9     %zipped = rise.apply %zip, %arow, %bcol
10
11    %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> -> data<int>> {
12     %fst = rise.fst #rise.int #rise.int
13     %snd = rise.snd #rise.int #rise.int
14     %t1  = rise.apply %fst, %t
15     %t2  = rise.apply %snd, %t
16     %mul = rise.mult #rise.int
17     %res = rise.apply %mul, %t1, %t2
18     rise.return %res : !rise.data<int>
19    }
20    %map = rise.map #rise.nat<4> #rise.tuple<int, int> #rise.int
21    %mapped = rise.apply %map, %f, %zipped
22
23    %add = rise.add #rise.int
24    %init = rise.literal #rise.lit<int<0>>
25    %reduce = rise.reduce #rise.nat<4> #rise.int #rise.int
26    %res = rise.apply %reduce, %add, %init, %mapped
27    rise.return %res : !rise.data<int>
28   }
29
30   %map = rise.map #rise.nat<4> #rise.array<4, int>  #rise.array<4, int>
31   %res = rise.apply %map, %f2, %B
32   rise.return %res : !rise.data<array<4, array<4, int>>>
33  }
34
35  %map = rise.map #rise.nat<4> #rise.array<4,!rise.int> #rise.array<4,!rise.int>
36  %res = rise.apply %map, %f1, %A
37  return %res
38 }
```

Fig. 4. 4x4 matrix-matrix multiplication represented in the Rise MLIR dialect

The dot product computation is defined in lines 7–26 by the `zip` (line 7), `map` (line 19) and `reduce` (line 24) patterns. The lambda function passed to the `map` pattern (defined in lines 10 – 18) is interesting as it first unpacks the tuple (in lines 11 – 14) that was created by `zip` before it multiplies the components in line 16 using the **`rise.mult`** operation.

The dot product is then applied to each combination of row and column by the nesting of the two `map` patterns (in line 29 and 34) and their corresponding lambda expressions (in line 3 and 5).

It is worth noting that for each Rise pattern we first create a `mlir::Value` by providing the type information as attributes (prefixed with **`#`**), such as the `zip` in line 7. A value can only be applied after it is created, such as for `zip` in line 8.

4 Conclusions and future work

In this paper, we presented an initial implementation of the Rise functional pattern-based language in mlir. We described the implementation of the type system and how lambda expressions are encoded by wrapping mlir regions. Patterns are represented as mlir operations that are instantiated with information customizing the types before being used in function applications. Finally, we showed an example of how matrix-matrix multiplication is represented in this dialect.

In the future, we will complete the implementation of the dialect along with low-level patterns and an implementation of symbolic computations of natural numbers to track array sizes in types. We will also explore the implementation of dependent function types to write computations abstracting over data types and array sizes.

The implementation of a functional pattern-based representation opens many interesting possibilities for interactions with other mlir dialects including higher-level dialects such as XLA as well as lower-level dialects such as the affine dialect. We are interested in exploring transforming between and mixing dialects for finding the best trade off to combine the different strengths of the dialects. We are also interested in compiling the functional pattern-based representation directly to specialized hardware exploiting the higher level semantics of the patterns and bypassing lower-level representations.

References

1. Mlir specification. https://github.com/tensorflow/mlir/blob/master/g3doc/LangRef.md (2019 (accessed December 2, 2019))
2. Barham, P., Isard, M.: Machine Learning Systems are Stuck in a Rut. In: HotOS (2019)
3. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with Lift. In: CGO (2018)
4. Lattner, C., Pienaar, J.: MLIR Primer: A Compiler Infrastructure for the End of Moore's Law. In: Compilers for Machine Learning workshop at CGO (2019)
5. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In: ICFP (2015)
6. Steuwer, M., Remmelg, T., Dubach, C.: Lift: a functional data-parallel IR for high-performance GPU code generation. In: CGO (2017)