Introduction to MLIR

Dong-hee Na
donghee.na92@gmail.com

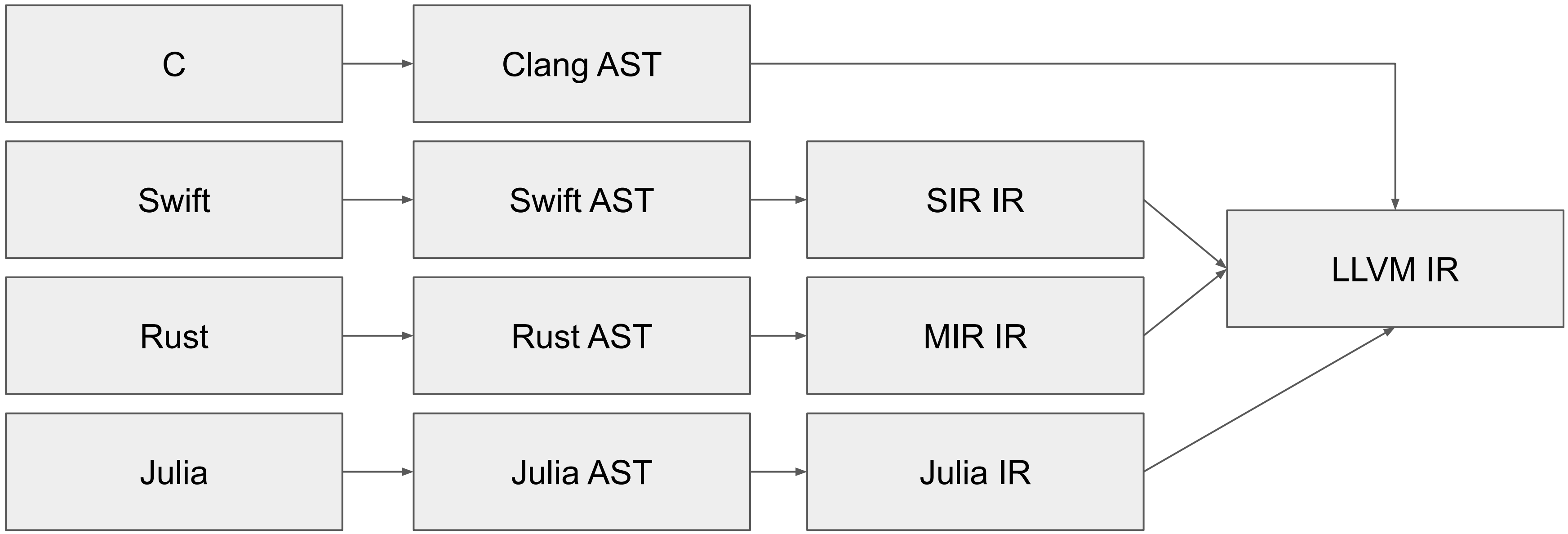
MLIR

- **Multi-Level Intermediate Representation**
- Multi-dimensional Loop IR
- Machine Learning IR
- …..
-

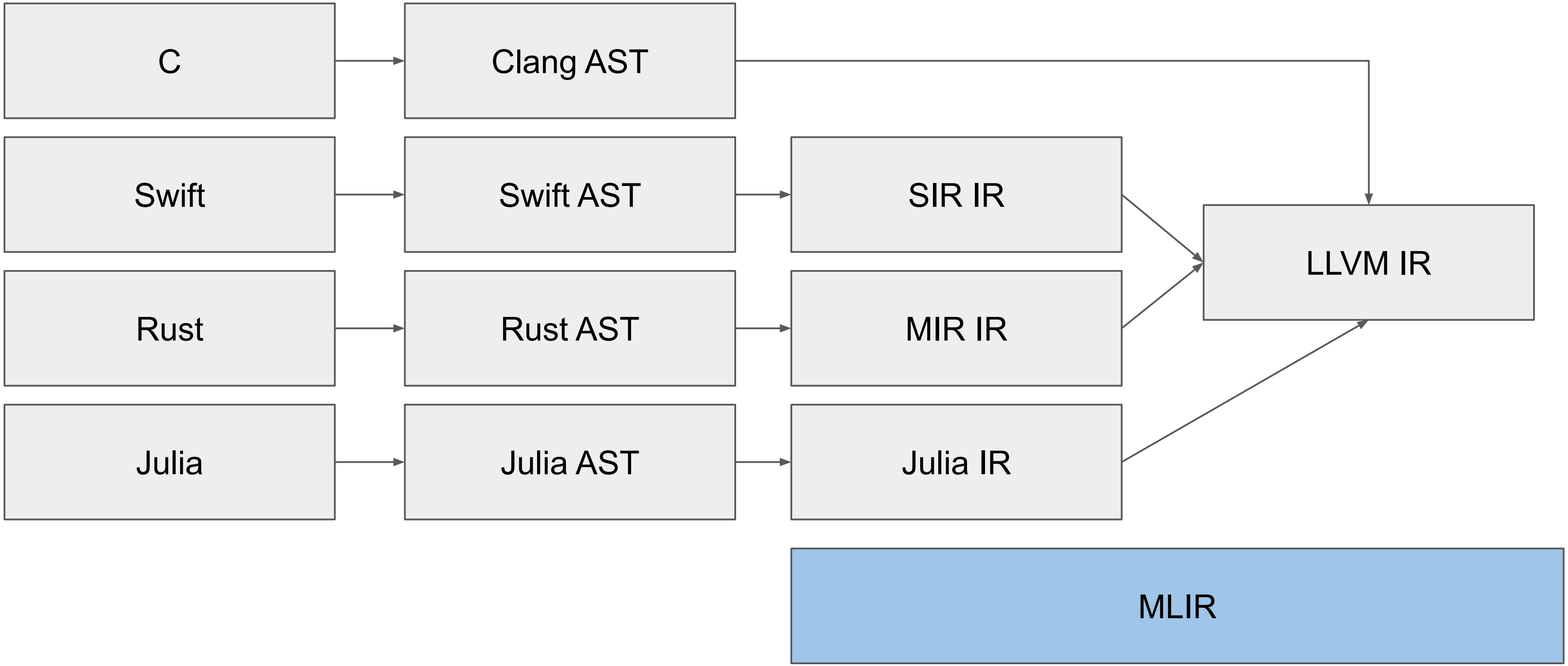
Chris Lattner

- SiFive (2020 - current): CIRCT / MLIR
- Google (2017 - 2020): Tensorflow / MLIR
- Tesla (2017)
- Apple (2013 - 2017): Swift
- University of Illinois, Urbana-Champaign (2000 - 2005): LLVM

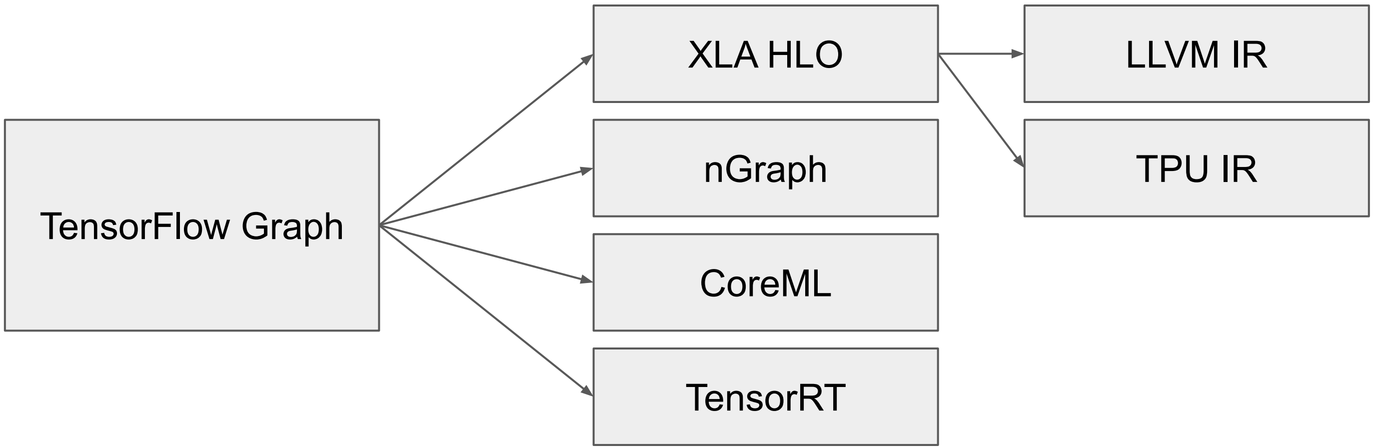
Why MLIR: General Purpose Compiler



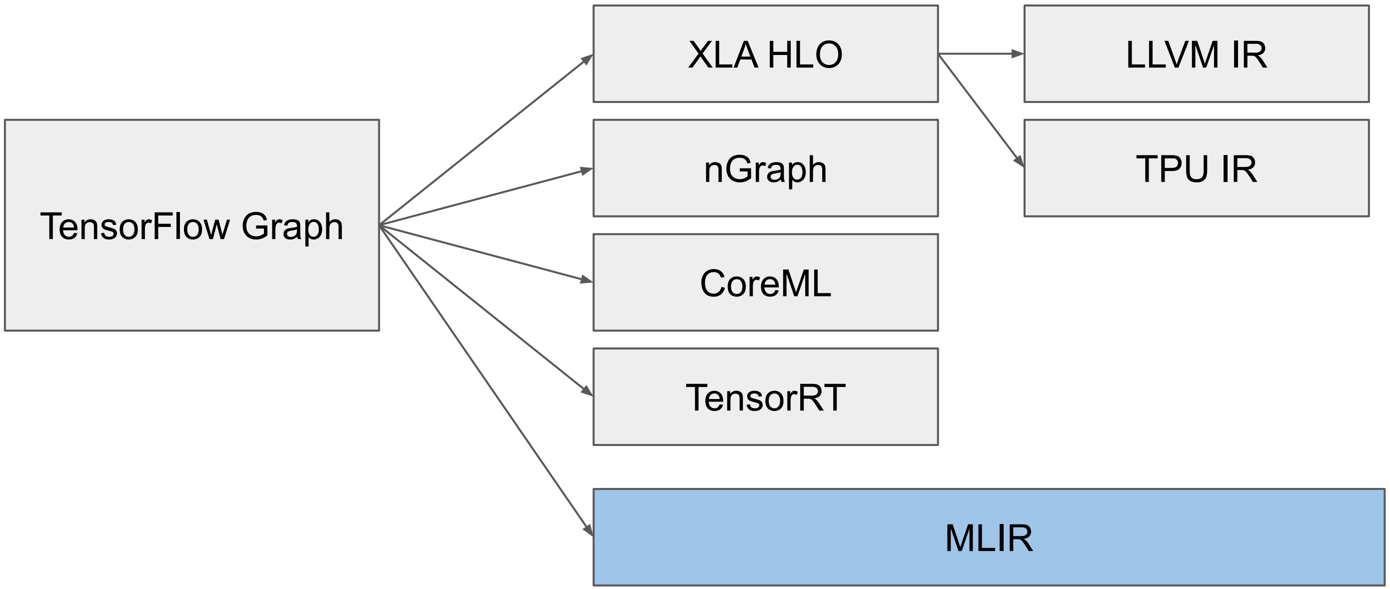
Why MLIR: General Purpose Compiler



Why MLIR: TensorFlow Graph



Why MLIR: TensorFlow Graph



Goal of MLIR

- **Framework for building (High-level) domain specific IRs**
- **Overcome the LLVM optimization limitation**
- Support optimizations for parallel / heterogeneous programming (OpenMP, OpenACC etc..)
- Progressive lowering to target specific forms
- Easy representation and optimizations of deep loop nests
- Easy to define new Operations, types etc: **Dialect**

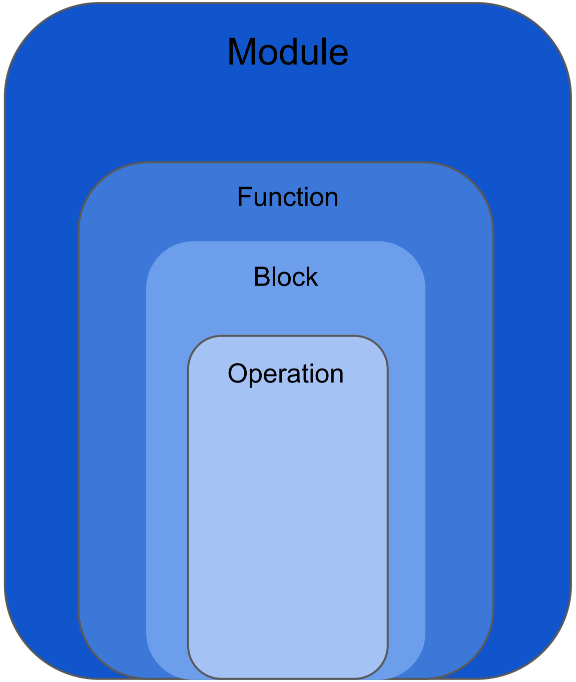
Design Features

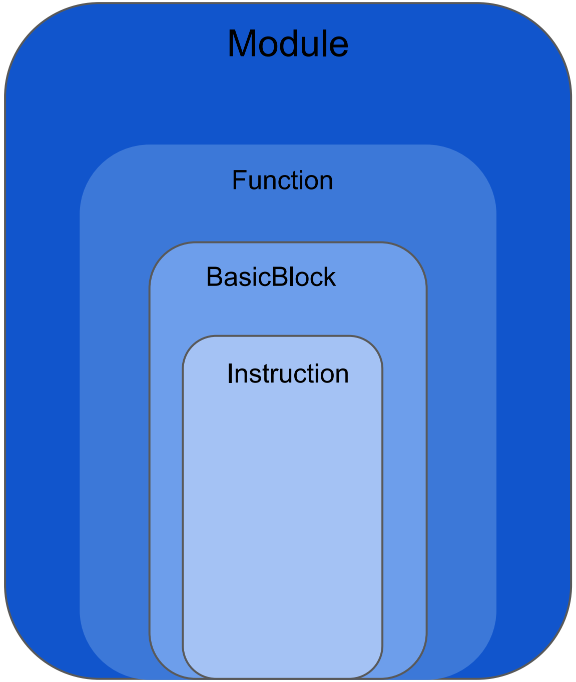
- Reusable Compiler Pass
- Dialect Specific Pass
- Mixing Dialects Together
- Parallel Compilation
- Interoperability

Users

- High level IR for general compiler project: Flang(FIR), Verona
- ML Graph: TensorFlow / ONNX-MLIR / PlaidML
- HW design: CIRCT
- more: https://mlir.llvm.org/users/

MLIR VS LLVM


Module
Function
Block
Operation


Module
Function
BasicBlock
Instruction

MLIR Core Concept

- Region: A list of basic blocks to form a CFG
- Block: A sequential list of Operations.
- Operation: A generic single unit of “code”

MLIR

```
func @toy_func(%tensor: tensor<2x3xf64>) -> tensor<3x2xf64> {
  %t_tensor = "toy.transpose"(%tensor) { inplace = true } : (tensor<2x3xf64>) -> tensor<3x2xf64>
  return %t_tensor : tensor<3x2xf64>
}
```

Dialect

- 20+ Builtin Dialects are supported: affine, tensor, llvm, gpu, omp, avx512
- Users can easily define their own custom Dialect

Dialect (ODS Framework)

```
// An Op is a TableGen definition that inherits the "Op" class parameterized
// with the Op name
def LeakyReluOp: Op<"leaky_relu",
    // and a list of traits used for verification and optimization.
    [NoSideEffect, SameOperandsAndResultType]> {
  // The body of the definition contains named fields for a one-line
  // documentation summary for the Op.
  let summary = "Leaky Relu operator";

  // The Op can also a full-text description that can be used to generate
  // documentation for the dialect.
  let description = [{
    Element-wise Leaky ReLU operator
      x -> x >= 0 ? x : (alpha * x)
  }];

  // Op can have a list of named arguments, which include typed operands
  // and attributes.
  let arguments = (ins AnyTensor:$input, F32Attr:$alpha);

  // And a list of named and typed outputs.
  let results = (outs AnyTensor:$output);
}
```

Toy compiler

This tutorial runs through the implementation of a basic toy language on top of MLIR. The goal of this tutorial is to introduce the concepts of MLIR; in particular, how dialects can help easily support language specific constructs and transformations while still offering an easy path to lower to LLVM or other codegen infrastructure.

Toy Compiler

Parser → AST → MLIR → Transform → Lowering

```
def multiply_transpose(a, b) {
  return transpose(a) * transpose(b);
}

def main() {
  var a = [[1, 2, 3], [4, 5, 6]];
  var b<2, 3> = [1, 2, 3, 4, 5, 6];
  var c = multiply_transpose(b, a);
  print(c);
}
```

Emit MLIR

- Implement Parser
- Build AST
- Traverse AST and emit MLIR

Define Dialect and operations

ODS

```
def TransposeOp : Toy_Op<"transpose",
    [NoSideEffect, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  let summary = "transpose operation";

  let arguments = (ins F64Tensor:$input);
  let results = (outs F64Tensor);

  let assemblyFormat = [{
    `(` $input `:` type($input) `)` attr-dict `to` type(results)
  }];

  // Enable registering canonicalization patterns with this operation.
  let hasCanonicalizer = 1;

  // Allow building a TransposeOp with from the input operand.
  let builders = [
    OpBuilderDAG<(ins "Value":$input)>
  ];

  // Invoke a static verify method to verify this transpose operation.
  let verifier = [{ return ::verify(*this); }];
}
```

C++

```
void TransposeOp::build(mlir::OpBuilder &builder, mlir::OperationState &state,
                        mlir::Value value) {
  state.addTypes(UnrankedTensorType::get(builder.getF64Type()));
  state.addOperands(value);
}

void TransposeOp::inferShapes() {
  auto arrayTy = getOperand().getType().cast<RankedTensorType>();
  SmallVector<int64_t, 2> dims(llvm::reverse(arrayTy.getShape()));
  getResult().setType(RankedTensorType::get(dims, arrayTy.getElementType()));
}

static mlir::LogicalResult verify(TransposeOp op) {
  auto inputType = op.getOperand().getType().dyn_cast<RankedTensorType>();
  auto resultType = op.getType().dyn_cast<RankedTensorType>();
  if (!inputType || !resultType)
    return mlir::success();

  auto inputShape = inputType.getShape();
  if (!std::equal(inputShape.begin(), inputShape.end(),
```

Generated MLIR

```
module  {
  func private @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
    %0 = toy.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = toy.transpose(%arg1 : tensor<*xf64>) to tensor<*xf64>
    %2 = toy.mul %0, %1 : tensor<*xf64>
    toy.return %2 : tensor<*xf64>
  }

  func @main() {
    %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>
    %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
    %4 = toy.generic_call @multiply_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %5 = toy.generic_call @multiply_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    toy.print %5 : tensor<*xf64>
    toy.return
  }
}
```

Canonicalizing optimization

- MLIR provides various optimization passes
- Canonicalizing optimization can be apply to some patterns of IR
- Good example would be **transpose(transpose(x)) == x**,
  this pattern of dialect can be rewritten by passing canonical optimization

Canonicalizing optimization


```
mlir::LogicalResult
 matchAndRewrite(TransposeOp op,
                 mlir::PatternRewriter &rewriter) const override {
   // Look through the input of the current transpose.
   mlir::Value transposeInput = op.getOperand();
   TransposeOp transposeInputOp = transposeInput.getDefiningOp<TransposeOp>();

   // Input defined by another transpose? If not, no match.
   if (!transposeInputOp)
     return failure();

   // Otherwise, we have a redundant transpose. Use the rewriter.
   rewriter.replaceOp(op, {transposeInputOp.getOperand()});
   return success();
}
```

Canonicalizing optimization

Origin IR

```
func @transpose_transpose(%arg0: tensor<*xf64>) ->
tensor<*xf64> {
  %0 = toy.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
  %1 = toy.transpose(%0 : tensor<*xf64>) to tensor<*xf64>
  toy.return %1 : tensor<*xf64>
}
```

Optimized IR

```
func @transpose_transpose(%arg0: tensor<*xf64>) ->
tensor<*xf64> {
  toy.return %arg0 : tensor<*xf64>
}
```

Inlining - enable inlining & define type casting

Parser → AST → MLIR → Transform → Lowering

- DialectInlinerInterface should be implemented
- Add inliner pass to pass manager
- Define whether type cast is enabled.

```
 void handleTerminator(Operation *op,
                        ArrayRef<Value> valuesToRepl) const final
{
   // Only "toy.return" needs to be handled here.
   auto returnOp = cast<ReturnOp>(op);

   // Replace the values directly with the return operands.
   assert(returnOp.getNumOperands() ==
valuesToRepl.size());
   for (const auto &it :
llvm::enumerate(returnOp.getOperands()))
     valuesToRepl[it.index()].replaceAllUsesWith(it.value());
  }
```

Inlining with type casting op

Origin IR

```
func @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
  %0 = toy.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
  %1 = toy.transpose(%arg1 : tensor<*xf64>) to tensor<*xf64>
  %2 = toy.mul %0, %1 : tensor<*xf64>
  toy.return %2 : tensor<*xf64>
}


func @main() {
  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00],
[4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
  %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
  %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00]> : tensor<6xf64>
  %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
  %4 = toy.generic_call @multiply_transpose(%1, %3) : (tensor<2x3xf64>,
tensor<2x3xf64>) -> tensor<*xf64>
  %5 = toy.generic_call @multiply_transpose(%3, %1) : (tensor<2x3xf64>,
tensor<2x3xf64>) -> tensor<*xf64>
  toy.print %5 : tensor<*xf64>
  toy.return
}
```

Inlined IR

```
func @main() {
  %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00,
3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>}
: () -> tensor<2x3xf64>
  %1 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00,
3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>}
: () -> tensor<2x3xf64>
  %2 = "toy.cast"(%1) : (tensor<2x3xf64>) -> tensor<*xf64>
  %3 = "toy.cast"(%0) : (tensor<2x3xf64>) -> tensor<*xf64>
  %4 = "toy.transpose"(%2) : (tensor<*xf64>) -> tensor<*xf64>
  %5 = "toy.transpose"(%3) : (tensor<*xf64>) -> tensor<*xf64>
  %6 = "toy.mul"(%4, %5) : (tensor<*xf64>, tensor<*xf64>) -> tensor<*xf64>
  toy.print %6 : tensor<*xf64>
  toy.return
}
```

Inlining - shape inference

- Define ShapeInferenceOpInterface by using ODS framework
- Implement ShapeInferencePass.
- Register ShapeInferenceOpInterface into each dialect operation.
- Add ShapeInferencePass to the MLIR pass manager.

Inlining with shape inference

Before

```
func @main() {
  %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00],
[4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>} : () ->
tensor<2x3xf64>
  %1 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00],
[4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>} : () ->
tensor<2x3xf64>
  %2 = "toy.cast"(%1) : (tensor<2x3xf64>) -> tensor<*xf64>
  %3 = "toy.cast"(%0) : (tensor<2x3xf64>) -> tensor<*xf64>
  %4 = "toy.transpose"(%2) : (tensor<*xf64>) -> tensor<*xf64>
  %5 = "toy.transpose"(%3) : (tensor<*xf64>) -> tensor<*xf64>
  %6 = "toy.mul"(%4, %5) : (tensor<*xf64>, tensor<*xf64>) -> tensor<*xf64>
  toy.print %6 : tensor<*xf64>
  toy.return
}
```

After

```
func @main() {
  %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00,
3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>}
: () -> tensor<2x3xf64>
  %1 = "toy.transpose"(%0) : (tensor<2x3xf64>) -> tensor<3x2xf64>
  %2 = "toy.mul"(%1, %1) : (tensor<3x2xf64>, tensor<3x2xf64>) ->
tensor<3x2xf64>
  toy.print %2 : tensor<3x2xf64>
  toy.return
}
```

Dialect <-> Dialect transformation

- One of key concepts of MLIR is exchanging between various dialects
- By transforming from toy dialect to affine dialect we can use optimization pass of affine dialect
- Partial transform

Dialect <-> Dialect transformation


```
mlir::LogicalResult
matchAndRewrite(mlir::Operation *op, ArrayRef<mlir::Value> operands,
                mlir::ConversionPatternRewriter &rewriter) const final {
  auto loc = op->getLoc();
  lowerOpToLoops(
      op, operands, rewriter,
      [loc](mlir::PatternRewriter &rewriter,
            ArrayRef<mlir::Value> memRefOperands,
            ArrayRef<mlir::Value> loopIvs) {
        TransposeOpAdaptor transposeAdaptor(memRefOperands);
        mlir::Value input = transposeAdaptor.input();
        SmallVector<mlir::Value, 2> reverseIvs(llvm::reverse(loopIvs));
        return rewriter.create<mlir::AffineLoadOp>(loc, input, reverseIvs);
      });
  return success();
}
```

Dialect <-> Dialect transformation

Toy Dialect

```
%2 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
%3 = toy.mul %2, %2 : tensor<3x2xf64>
```

Affine Dialect

```
affine.store %cst, %2[0, 0] : memref<2x3xf64>
 affine.store %cst_0, %2[0, 1] : memref<2x3xf64>
 affine.store %cst_1, %2[0, 2] : memref<2x3xf64>
 affine.store %cst_2, %2[1, 0] : memref<2x3xf64>
 affine.store %cst_3, %2[1, 1] : memref<2x3xf64>
 affine.store %cst_4, %2[1, 2] : memref<2x3xf64>

 // Load the transpose value from the input buffer and store it
into the
 // next input buffer.
 affine.for %arg0 = 0 to 3 {
   affine.for %arg1 = 0 to 2 {
     %3 = affine.load %2[%arg1, %arg0] : memref<2x3xf64>
     affine.store %3, %1[%arg0, %arg1] : memref<3x2xf64>
   }
 }

 // Multiply and store into the output buffer.
 affine.for %arg0 = 0 to 3 {
   affine.for %arg1 = 0 to 2 {
     %3 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>
     %4 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>
     %5 = mulf %3, %4 : f64
     affine.store %5, %0[%arg0, %arg1] : memref<3x2xf64>
   }
}
```

Optimized Affine Dialect

```
affine.for %arg0 = 0 to 3 {
   affine.for %arg1 = 0 to 2 {
     // Load the transpose value from the input buffer.
     %2 = affine.load %1[%arg1, %arg0] : memref<2x3xf64>

     // Multiply and store into the output buffer.
     %3 = mulf %2, %2 : f64
     affine.store %3, %0[%arg0, %arg1] : memref<3x2xf64>
   }
}
```

MLIR to LLVM IR lowering

Parser → AST → MLIR → Transform → **Lowering**

- mlir::translateModuleToLLVMIR API lowers IRs into LLVM IR.
- By lowering, compiler can use various LLVM tech stack
  e.g ORCJIT, LLVM optimization pass, LLVM backend support, etc..
- If IR supports LLVM IR conversion, nothing to do but if not the conversion pattern should be implemented.

Generated LLVM IR

```
; ModuleID = 'LLVMDialectModule'
source_filename = "LLVMDialectModule"
target datalayout = "e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-darwin20.3.0"

@frmt_spec = internal constant [4 x i8] c"%f \00"

; Function Attrs: nofree nounwind
declare noundef i32 @printf(i8* nocapture noundef readonly, ...) local_unnamed_addr #0

; Function Attrs: nounwind
define void @main() local_unnamed_addr #1 !dbg !3 {
.preheader3:
  %0 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @frmt_spec, i64 0, i64 0), double 1.000000e+00), !dbg !7
  %1 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @frmt_spec, i64 0, i64 0), double 1.600000e+01), !dbg !7
  %putchar = tail call i32 @putchar(i32 10), !dbg !7
  %2 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @frmt_spec, i64 0, i64 0), double 4.000000e+00), !dbg !7
  %3 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @frmt_spec, i64 0, i64 0), double 2.500000e+01), !dbg !7
  %putchar.1 = tail call i32 @putchar(i32 10), !dbg !7
  %4 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @frmt_spec, i64 0, i64 0), double 9.000000e+00), !dbg !7
  %5 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @frmt_spec, i64 0, i64 0), double 3.600000e+01), !dbg !7
  %putchar.2 = tail call i32 @putchar(i32 10), !dbg !7
  ret void, !dbg !9
}

; Function Attrs: nofree nounwind
declare noundef i32 @putchar(i32 noundef) local_unnamed_addr #0

attributes #0 = { nofree nounwind }
attributes #1 = { nounwind }

!llvm.dbg.cu = !{!0}
!llvm.module.flags = !{!2}

!0 = distinct !DICompileUnit(language: DW_LANG_C, file: !1, producer: "mlir", isOptimized: true, runtimeVersion: 0, emissionKind: FullDebug)
!1 = !DIFile(filename: "LLVMDialectModule", directory: "/")
!2 = !{i32 2, !"Debug Info Version", i32 3}
!3 = distinct !DISubprogram(name: "main", linkageName: "main", scope: null, file: !4, line: 5, type: !5, scopeLine: 5, spFlags: DISPFlagDefinition | DISPFlagOptimized, unit: !0, retainedNodes: !6)
!4 = !DIFile(filename: "sample.toy", directory: "/Users/user/oss/mlir-standalone-toy/standaloneToy")
!5 = !DISubroutineType(types: !6)
!6 = !{}
!7 = !DILocation(line: 10, column: 3, scope: !8)
!8 = !DILexicalBlockFile(scope: !3, file: !4, discriminator: 0)
!9 = !DILocation(line: 5, column: 1, scope: !8)
```

Summary

- MLIR is focusing on phases between graph/AST and low level IR(LLVM).
- **MLIR is framework for high-level IR developers.**
- MLIR provides common optimization pipeline.
- **MLIR is framework for both C4ML and general purpose compilers.**

E.O.D